MICROPROCESSOR
(DSEL32)
(BSC ELECTRONICS-IV)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

UNIT – I

1. INTRODUCTION

Structure

1.0 Introduction
1-2 History of computers
1-3 Generations of computers
  1-3-1 First generation
  1-3-2 Second Generation
  1-3-3 Third generation
  1-3-4 Fourth generation
  1-3-5 Fifth generation
1-4 Classification of computers
  1-4-1 Super Computers
  1-4-2 Main Frames
  1-4-3 Mini Computers
  1-4-4 Micro Computers
1-5 Evaluation of microprocessors
  1-5-1 History of microprocessors
  1-5-2 Organization of micro computer
1-6 8085 Pin configuration and architecture
  1-6-1 Features of 8085
  1-6-2 Architecture of 8085
  1-6-3 Registers
  1-6-4 Special Purpose Registers
  1-6-5 ALU
  1-6-6 Timing and Control Unit
  1-6-7Stack
1-7 Pin configurations of 8085
  1-7-1 Address Group
  1-7-2 Data Group
  1-7-3 Control Group
  1-7-4 Interrupt & externally initiated Signals
  1-7-5 Serial I/O Ports Signals
  1-7-6 Power Supply & Clock Signals

Objectives

- Discuss history of computers
- Discuss evaluation of microprocessors
- Discuss about 8085 Pin configuration and architecture

NOTES

1.0 Introduction

Computers and computer systems are a pervasive part of the modern world. Aside from just the common desktop PC, there are a number of other types of specialized computer systems that pop up in many different places. The central component of these computers and computer systems is the microprocessor, or the CPU. The CPU (short for "Central Processing Unit") is essentially the brains behind the computer system; it is the component that "computes".

1-2 History of computers

- In 16th century, Blaise Pascal introduced first mechanical adding and subtracting machine.
- In 17th century Gottfried Leibriz developed a machine which can perform both multiplication and division.
- In 1822 Charles Babbage introduced a multi step computer which can store data.
- In 1887 Herman Hollerith invented a device for automatic census tabulation.
- In 1945 J.Presper Eckert and J.W.Mauchly developed first electronic digital computer ENIAC.
- An improved version of electronic computer was given by von Neumann.
- In 1949 Electronic Delay Storage Automatic Calculator (EDSAC) was developed at Cambridge University.
- In 1951 Universal Automatic Computer was built.
- In 1952 Electronic Discrete Variable Automatic Computer(EDVAC) was developed by J. Presper Echkert and J.W.Mauchly.
- In 1960 using solid state technology IBM developed IBM 7090 scientific computer.

1-3 Generations of computers

Computer generation is used to distinguish the computers on the basis of varying hardware and software configurations. As on today there are five generations of computers.

1-3-1 First generation:

These computers came into existence in between 1946-54. In 1943, john W.Mauchly designed ENIAC (Electronic Numerical Integrator and Calculator). In 1949, EDSAC (Electronic Delay Storage Automatic Computer) and in 1952, EDVAC (Electronic Discrete Variable Automatic Computer) was developed.

Example: EDVAC, IBM 701, IBM704, UNIVAC etc.

1) Vaccume tubes are used to develop these computers.

2) Magnetic drums are used for internal storage.

3) These computers used assembly language programming.

4) Their processing speed is very less and they occupy large space.

5) These are mainly used for scientific applications only.

1-3-2 Second Generation:

The development of solid state technology in 1950's (junction transistor in 1948) had made drastic changes in second generation of computers. In addition to main memory, auxiliary memory had been used in these computers. Input and output processors are introduced in them to perform I/O operations. It can perform floating-point operations also.

Example: IBM 1620, IBM 7090, CDC 1604, PDP5 etc.

1) Transistors are used instead of vacuum tubes.

2) Auxiliary memory came into existence.

3) High level languages such as COBOL, FORTRAN are used for programming.

4) Processing speed has been increased rapidly and size of the computer was reduced.

5) I/O processors are included.

6) Batch processing made possible

1-3-3 Third generation:

In these computers IC technology was used. They came into existence in 1960's. IC's are used in developing CPU and memory. Thus the size is reduced and reliability had been increased.

Ex: IBM370, CDC7600, PDP11, CYBER175, STAR100 etc.

1) Integrated circuits are used and hence size is very small.

2) Multiprocessing, multiprogramming and pipelining techniques were introduced.

3) Virtual memory concept has been introduced.

4) They have high processing speed and high storage capacity.

5) They can handle both scientific and commercial applications.

1-3-4 Fourth generation:

Development of microprocessors in 1970's is a milestone in computer industry. Very large scale integration and high density and high-speed logic circuits are used to develop computers. The most popularly used microprocessors in computers are Intel 8008, 8085, 8086, zilog80 etc.

Example: INTEL 8748, IBM 3081, CRAYX-MP etc.

1) VLSI technology has been used in these computers.

2) Their storage capacity and speed is high.

3) Microprocessors are used to develop CPU.

4) Computer graphics has been introduced.

5) These are used for mathematical modeling, simulation and computer aided design.

1-3-5 Fifth generation:

In 1981, Institute of New Generation Computer Technology, Japan has developed a project known as Fifth Generation Computer Systems Project. These systems include artificial intelligence, expert systems and knowledge based systems. PROLOG and LISP are popular languages used in these systems.

1-4 Classification of computers

Computers are classified mainly into four categories, on the basis of their speed (number of operations per sec.), Memory size (bytes) and number of input and output devices it can support. They are

i) Super Computers

ii) Main Frames

iii) Mini Computers

iv) Micro Computers

1-4-1 Super Computers

These computers are most powerful and are mainly used for scientific applications. They have large speed and large memory. They can perform 20 million operations per second.

Example: CRAY 2, PARAM 9000. ETA 10.

1-4-2 Main Frames:

Their capacity and aped distinguish these computers. They have large memory such as 256 MB and have a speed to process 10 million operations per second. These computers will support many users in real time. Their word length is in the order of 64 bits.

Example: VAX 8000, IBM 4300, CYBEK 180.

1-4-3 Mini Computers:

It is also known as stand alone system, which can process information The speed of the CPU is around 1 million operations per second. Their memory size ranges from 256 Kbytes to a few mega bytes. The word length of these computers is 32 bits.

Example: PDP11, HP3000 and SUPER 16.

1-4-4 Micro Computers:

These are also referred as desk top computers or personal computers. They consist a microprocessor as a central processing unit. LSI is used to develop the CPU. Their memory is in the rage of 1 MB and speed is in the rage of few hundred operations per second. Their word size is 8 to 16 bits.

Example: IBM PC, APPLE, SICLAIR.

1-5 Evaluation of microprocessors

Thus we can see microprocessor in micro computers that are developed in 1970's. A microprocessor is nothing but the CPU of a computer but only difference is, the discrete components present in CPU are integrated into a single chip known as integrated Circuit by the help of VLSI technology.

1-5-1 HISTORY OF MICROPORCESSORS

- In 1971 first 4-bit microprocessor was introduced by Intel Corporation. Later its enhanced version Intel 4040 was introduced. At the same time Rockwell International Produced PPS-4, Toshiba produced T3472 microprocessor.

- In 1972 Intel produced 8-bit microprocessor, 8008 with PMOS technology. This was very slow. The more powerful 8-bit microprocessor, 8080 was introduced, in 1973 with NMOS technology. This requires 3 power supplies. In 1975 enhanced version of 8085 which requires only one power supply was produced. At the same time Motorola produced MC6800, Zilog produced Z80 and National Semiconductor produced NSC 800.

- In 1978 – Intel produced 16 bit microprocessor 8086. At the same time Motorola 68000, Fairchild 9440, Zilog Z8000 National Semiconductors PACE, Texas Instruments TMS 9900 are produced.

- In 1985 Intel introduced first powerful 32-bit processor, Intel 80386, Some other 32-bit processors are Intel 486, Pentium Pro, Pentium II, III, IV, Advance Micro devices K5, K6, K7, Cyrix 586, 686, Motorola 68020, National Semi conductors 32032, Zilogs Z80000 etc.

1-5-2 ORGANIZATION OF MICRO COMPUTER

A micro computer is a computer in which its central processor unit is a microprocessor. A microprocessor is nothing but the central processing unit of a computer built into single chip with the help of IC technology. The important components of micro computer are Central Processing Unit, input/output units, Memory and Buses.

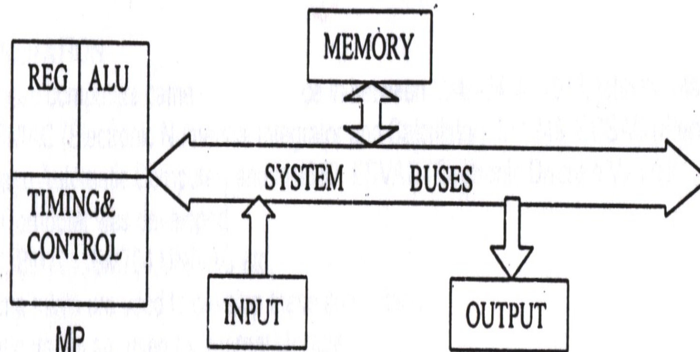


Fig 1.1

1) Central Processing Unit (Microprocessor)

It brings the instructions (commands) form memory, translates them, performs arithmetic and logical operations and stores the results temporarily. A typical microprocessor has three important parts. They are

1) Arithmetic and Logical Unit.

2) Timing & Control Unit.

3) Registers

2) Arithmetical and Logical Unit

This unit carries arithmetic operations such as addition, subtraction etc. and logical operations like less than, greater than etc. on input data in the form of binary numbers.

3) Timing and Control Unit

It provides timing and control signals to different units in the computer. It routes the instructions and data that are to be transferred from one unit to other.

4) Registers

These are memory elements to store data in the form of binary bits. A register is a group of 8-bits to store memory words.

5) Memory

It is a device to store data, instructions and results in the form of binary bits. Again memory can be classified into registers known as memory locations. Each location has an address to identify it by CPU. Memories can be classified into two categories. They are Prime memory and Secondary memory.

Prime memories are the memories, which can be accessed directly by CPU. These are mainly developed by semiconductor materials and hence known as semiconductor memories. The speed of these memories is high but cost is also high.

These are two types:

(1) RAM used to store user programs

(2) ROM used store system programs.

Secondary memories are memories, which cannot be accessed directly by CPU. These are made up of magnetic materials. Their speed is low and their cost is low. Floppies compact disks, magnetic tapes etc. come under this category.

6) Input device

An input device is used to provide information needed for CPU to perform an operation. It converts instructions; data and other information into binary form suitable for CPU.

Example: Key board, mouse etc.

7) Output Device

An output device is used to store or retrieve the results from CPU and to convert it into usable format for a particular application.

Example: Printers, CRT, D/A converters etc.

8) Buses

These are group of wires (conducting lines) used to transfer data or control information in the form of electric signals from one device to another:

1-6 8085 PIN CONFIGURATION AND ARCHITECTURE

1-6-1 Features of 8085

The important features of 8085 microprocessor are

1) Number of pins : 40

2) Technology : N-MOS

3) Package : DIP (dual line in package).

4) Speed (clock frequency) : 3.125 MHz.

5) Power Supply : + 5V DC.

6) Address Lines : 16

7) Data lines : 8

8) Control Lines: 3

9) Status lines : 3

10) Address Space : 16K (65, 536).

11) Word Size : 8 bits.

1-6-2 Architecture of 8085

The block diagram of 8085 microprocessor is shown in figure. The important units are

ARCHITECTURE OF 8085

Fig 1.2 Block Diagram of 8085

1-6-3 Registers

It has both 8-bit registers and 16-bit registers. There are 8 addressable registers of 8-bit length and 3 sixteen bit registers. These registers are again classified into general purpose registers and special purpose registers.

General Purpose Registers:

These are also known as scratch pads. These are 6 registers namely B, C, D, E, H, L of 8 bit length and are paired so that we can store data in register pairs BC, DE, HL. These registers are used for only temporary storage of data. When these registers are used in register pair

mode the higher order bits are stored in first register and lower order bits are stored in second register. Among these HL pair has a significant advantage. The 16 bit data stored in HL pair is assumed as address so that it can be used as memory pointer. It is used to hold the address of memory in indirect addressing mode.

Data can be transferred from one register to other register. If the data is transferred from a source register to destination register, the data in the destination register gets erased and the data in the source gets copied into destination. There is no possibility of performing arithmetic and logical operations between these registers.

1-6-4 Special Purpose Registers

Accumulator:

This is an 8 bit register used for temporary storage of data during execution of a program. It holds one operand during arithmetic operations and serves as one input of ALU. The size of this register, determines word length of microprocessor. After every operation result is stored in accumulator since 8085 is accumulator based processor.

Program Counter:

This is also known as instruction pointer or location counter or memory pointer. Program counter holds the address of next instruction to be executed by the processor. Thus the length of the program counter is 16 bits.

The set of instructions are stored serially in memory. The address of first instruction in memory is stored into program counter. During normal execution, after every instruction, program counter is incremented by the length of previous instruction and execution is transferred to the next instruction. This process continues until it encounters a pseudo instruction HLT (Halt). The program counter will increment one by one in normal execution when an interrupt or jump instruction or a CALL is encountered the contents of program counter are suddenly changed to specified memory location. When reset is activated during normal execution the contents of program counter gets erased and it is stored by 0000.

Flag or Status Register:

It is an 8 bit register which is used to resemble the status of ALU. In 8 bits, 5 bits carry significant information in the form of electrical signals known as flags. These flags are affected by arithmetic and logical

operations that are performed in ALU. These flags reflect the status or conditions that occur in ALU.

1) Carry Flag: It is set when ever a carry/barrow occurs during arithmetic operations such as addition or subtraction. Otherwise it is reset.

2) Auxiliary Carry Flag: It is set whenever there occurs a carry at the 4th bit during an 8 bit operation, otherwise it is reset.

3) Parity Flag: It is set when there is even number of ones in the final result after an arithmetic/logical operation. If there is odd number of ones it is reset.

4) Sign Flag: It is set if the result is negativeve after an arithmetic operation, otherwise it is reset.

5) Zero Flag: It is set if the result after an operation is zero, otherwise it is reset.

Instruction Register:

After fetching an instruction from the memory the corresponding machine code is entered into instruction register in the form of BCD. This BCD code is decoded into respective parallel signals and they are given to the timing and control unit. During the execution of an instruction processor fetches instructions byte by byte from memory. In the instructions there may be opcode and operands (data or address). Data is transferred to the registers and opcode is loaded into instruction register. Hence its length is 8 bit.

Temporary Register:

It is used to hold the one of the two operands during and arithmetic or logical operation and is given as second input to ALU. Hence its length is 8 bits.

Address Latch:

It is a 16 bit register to hold the address of operands when AD bus is converted from address but to data bus during the third clock cycle.

1-6-5 ALU (Arithmetic and Logical Unit):

An ALU is a multi operational, combinational digital circuit. It has two important units. They are arithmetic unit and logical unit. One of these two units is enabled by the help of a control signal known as mode select, (M) and a particular function is selected in respective units by the help of

function select signals S0 and S1. Both the inputs A and B are given to two units and the output is taken only from the selected unit.

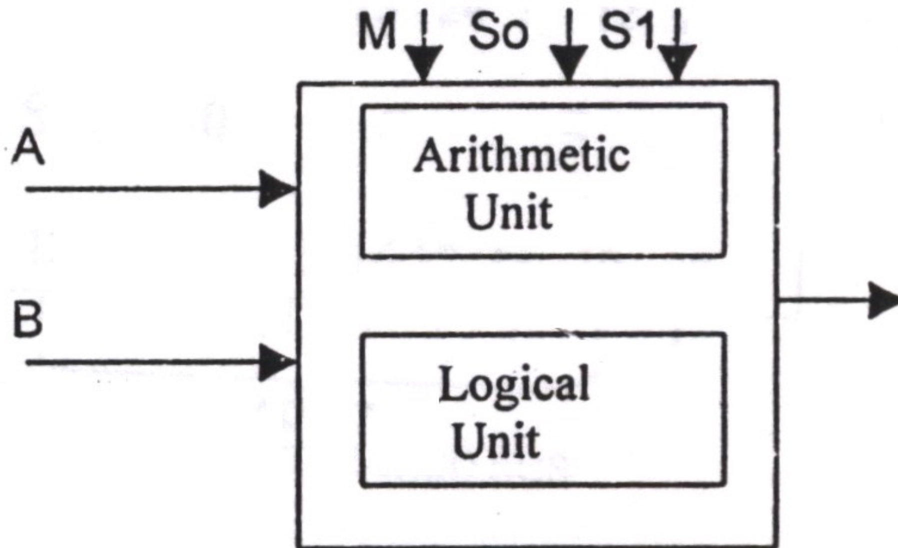


Fig 1.3

Arithmetic Unit: This unit consists of a parallel adder which can perform addition, subtraction, increment and decrement.
Logical Unit: This unit has four basic gates to perform logical operations like AND, OR, NOT and EXOR.

1-6-6 Timing and Control Unit:

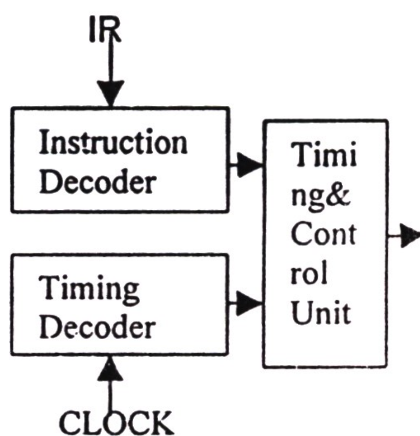


Fig 1.4

All operations that are performed by the processor must be synchronized with the clock. To sequence the operations in different steps certain

timing signals are needed. These are generated by system clock. Both timing signals from system clock and controls signals from the instruction decoder are given to a logic circuit so that the respective operation is enabled.

1-6-7Stack

Stack is a group of memory locations or an array of registers used to store information temporarily on the basis of FILO(first in last out) or LIFO (last in first out). Entering the data into stack is known as PUSH operation. Retrieving the data from stack is known as POP operation.

The first byte that is entered into the stack will be stored at the bottom of the stack. The last byte that is stored in the stack will appear at the top of the stack. Always the address of the element at the top of the stack is represented by a 16 bit register known as stack pointer.

During push operation stack pointer contents are incremented and during pop operation its contents are decremented.

Pop and push operations are similar to a manner of filling a stack of papers in a file. Always the page number of the page which is at the top is represented by stack pointer. The major applications of stack are – it is used in recursive process and used to store the information regarding sub programs.

In addition to these, address/data buffers, buses and different logic circuits are included to fulfill the requirements.

1-7 PIN CONFIGURATIONS OF 8085

The pins and signals of 8085 are classified into 6 groups. They are

1) Address Group

2) Control Group

3) Data Group

4) Interrupt and externally initiated signals

5) Serial I/O ports and signals

6) Power supply and frequency signals.

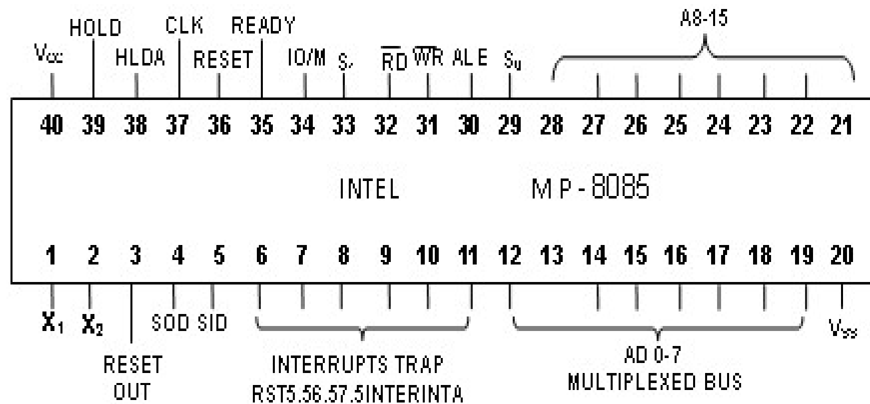


Fig 1.5 Pin Diagrams of 8085

1-7-1 Address Group (Pin No: 22-19 and 21-28: AD_{0-7} and A_{8-15}):

These 16 lines are divided into two sets, carrying address of memory / peripherals that are accessed by processor. AD_{0-7} is known as lower order address bus and A_{8-15} is known as higher order address bus. The lower order bus is a multiplexed bus (if a bus carries different types of signals during different times then it is known as multiplexed bus). During first clock cycle lower order address bus acts as address bus and in remaining cycles it acts as data bus.

1-7-2 Data Group (Pin No: 12-19: AD_{0-7}):

These 8 lines are used to carry 8 bit data. It is a bi-directional bus carrying data from processor to peripherals and vice versa.

1-7-3 Control Group

ALE (Address Latch Enable Pin No: 30):

This signal is used to latch the address on AD bus during the second clock cycle when it is changing from address bus to data bus.

WR (Pin No: 31):

A low voltage on this pin enables to store data into the memory or peripherals by the processor.

RD (Pin No: 32):

A low voltage on this pin enables to read from the memory of peripherals by the processor.

IO/M (Pin No: 34):

A low voltage on this pin selects memory otherwise it selects peripherals.

READY (Pin No: 35):

This is a request signal by the memory so that process goes into waiting state for a short interval, so that the speed of the processor is synchronized with memory.

S1 & S2 (Pin No: 29, 33):

These are status signals used to represent status of processor. By changing the signals on these two pins different operations are performed.

IO/M	S1	S2	Operation
0	1	1	Op-code fetch
0	1	0	Memory Read
0	0	1	Memory Write
1	1	0	Input Read
1	0	1	Output Write
1	1	1	INTA

1-7-4 Interrupt & externally initiated Signals

TRAP (Pin No: 6): It is non maskable interrupt and also it is vectored interrupt.

RST 5.5, 6.5, 7.5 (Restart Interrupts Pin No: 7, 8, 9): These are vectored interrupts and are maskable

INTR (Interrupt Request Pin No: 10): This is an un-vectored interrupt. When a high voltage on this pin arrives, processor suspends its normal operation and transfers the execution to the service routine.

INTA (Interrupt Acknowledge Pin No: 11):A low voltage on this pin represents that processor is acknowledging an interrupt request.

RESET IN/OUT: (Pin No: 36, 3): When the signal on the reset is low, processor suspends its current execution, buses are tri stated and it goes to first location or starting point. When reset-out is made high processor resets all other devices such as memory and peripherals.

HOLD (Pin No: 38): This signal is a hold request by the memory so that to operate in DMA mode

HLDA (Hold Acknowledge Pin No.: 39): A high voltage on this pin makes to suspend the normal operation of processor and it allows memory to deal with I/O devices in DMA mode.

1-7-5 Serial I/O Ports Signals: These signals are used to transfer serially. Data is generally transferred in parallel mode by the help of data bus. If we want to transfer serially we can use these pins.

SID (Serial Input Data Pin No: 5): When the data is given on this line it is loaded serially into the processor.

SOD (Serial Output Data Pin No.: 4): Serial data from the processor can be received on this pin.

1-7-6 Power Supply & Clock Signals

V_{cc}(Pin No: 40): Positive terminal of DC power supply + 5V is connected.

V_{ss} (Pin No: 20): Ground terminal of DC power supply.

X_1 & X_2 (Pin No: 1, 2): Quartz crystal oscillator is connected between these two pins to produce a frequency of 6.25 MHz clock signal.

CLK (Pin No: 37): This is used a system clock for memory devices.

2. 8086 MICROPROCESSOR

Structure

2-1 Introduction
2-2 8086 Microprocessor
2-3 8086 Architecture
    2-3-1General purpose registers
    2-3-2 Pointers
    2-3-3 Index registers
    2-3-4 Segment Registers
    2-3-5 Status Register
    2-3-6 Functional Units of 8086
2-4 Memory segmentation
2-5 Addressing modes of 8086

Objectives

After going through this unit you will be able to:

- Detailed description of pins of 8086
- Learn detailed architecture of 8086
- Details of internal elements of 8086
- Come across addressing modes of 8086
- Familiar with Instructions of 8086

2-1 Introduction

Intel 8086 microprocessor is a first member of x 86 families of processors. Advertised as a "source-code compatible" with Intel 8080 and Intel 8085 processors, the 8086 was not object code compatible with them. The 8086 has complete 16-bit architecture 16-bit internal registers, 16-bit data bus, and 20-bit address bus (1 MB of physical memory). Because the processor has 16-bit index registers and memory pointers, it can effectively address only 64 KB of memory. To address memory beyond 64 KB the CPU uses segment registers - these registers specify memory locations for code, stack, data and extra data 64 KB segments.

2-2 8086 Microprocessor

Features of 8086

1) Number of pints : 40
2) Technology : H-MOS
3) Package : DIP (dual line in package).
4) Number of transistors : 29000
5) Speed (clock frequency) : 5 MHz.

6)	Number of versions	: 3(8086, 8086-1, 8086-2)
7)	Power Supply	: + 5V DC.
8)	Address Lines	: 20
9)	Data lines	: 16
10)	Address Space	: 1 MB
11)	Word Size	: 16 bits.
12)	Number of Modes	: 2(Minimum/Maximum)

2-3 8086 Architecture

Register organization of 8086: Before going to the architecture of 8086, let us discuss about the register organization of 8086. The 8086 contains fourteen 16 bit registers. They are grouped into

1) General purpose registers
2) Pointers
3) Index registers
4) Segment registers
5) Instruction pointer and status register.

15 8	7 0
AH(8)	AL(8)
BH(8)	BL(8)
CH(8)	CL(8)
DH(8)	DL(8)
Stack Pointer(16)	
Base Pointer(16)	
Source Index(16)	
Destination Index(16)	
Code Segment(16)	
Data Segment(16)	
Stack Segment(16)	
Extra Segment(16)	
Instruction Pointer(16)	
Flags(16)	

2-3-1 General purpose registers:

There are 4-16 bit registers AX, BX, CX, DX. They can be used to store either 16-bit data or 8-bit data. Each 16 bit register can be viewed as two 8-bit registers, one is lower order and another is higher order as shown in the table.

16-bit register	Higher order 8-bit register	Lower order 8-bit register
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

A part from storing the operands

AX: This register serves as accumulator.

BX: This is used as base register for memory address calculations.

CX: This register is used as a counter for loop and string instructions.

DX: It is used to hold I/O address during I/O instructions. If the result is more than 16-bits, the lower order 16-bits are stored in accumulator and higher order 16-bits are stored in DX register.

2-3-2 Pointers:

There are two pointers and SP and BP. SP is used as stack pointer to represent the stack top address. It contains offset address. BP is the base register for accessing stack. Within stack several data areas may exist. BP is used to hold the offset of the base of a data area in stack segment. It is also can be used as general purpose register.

2-3-3 Index registers:

SI and DI are index registers. In the case of string instructions, SI is used to indicate source index and DI destination index. The contents of SI are added to the contents of DS to get the actual source address of the data. The contents of DI are added to the contents of ES to get the actual destination address of the data. They are also can be used in memory or stack address computation.

2-3-4 Segment Registers:

The memory used with an 8086 based system is divided into the four types of segments, each segment of size 64 K bytes. They are

1) Code segment: It contains instruction codes of a program.

2) Data segment: It contains variables and constants of the program.

3) Stack segment: Is used to store the addresses and data of a subroutine.

4) Extra segment: Contains the destination of some data of certain string instructions.

Since the size of each segment is 64KB, 16-bit address is sufficient to access it. The starting address of each segment is stored in segment registers. There are four segment registers Code segment register (CS), data segment register (DS), stack segment register (SS), extra segment register (ES).

The address of memory that is accessed by 8086 has 20 bits (Physical address); Segment register contains the upper 16-bits of starting address of a memory segment. CPU inserts four zeros for the lower 4 bits of 20 bit address.

Example:

If the contents of CS are 2000H, then the code segment begins from 20000H address.

The 64KB memory segment can reside anywhere in the 1MB memory space but its starting address is always divisible by 16(10h). How far a memory location is within memory segment from the starting address is called offset or effective address. The contents of a segment register are shifted left by 4 bits and then an offset is added to it to compute the required 20 bit physical address.

2-3-5 Status Register

It is a 16 bit register called as flag register or program status word. It has nine flags out of which 6 are conditional flags and remaining 3 are status flags.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1 0
X	**X**	**X**	**X**	**O F**	**D F**	**IF**	**T F**	**S F**	**Z F**	**X**	**A F**	**X**	**P F**	**X**	**C F**

1. **Carry Flag**: It is set when ever a carry/barrow occurs during arithmetic operations such as addition or subtraction. Otherwise it is reset.
2. **Auxiliary Carry Flag**: It is set whenever there occurs a carry at the BCD arithmetic operations, otherwise it is reset.
3. **Parity Flag**: It is set when there is even number of ones in the final result after an arithmetic/logical operation. If there is odd number of ones it is reset.
4. **Sign Flag**: It is set if the result is negative after an arithmetic operation, other wise it is reset.
5. **Zero Flag**: It is set if the result after an operation is zero, other wise it is reset.
6. **Overflow flag**: It is set when ever a carry/barrow occurs during sign-arithmetic operations such as addition or subtraction. Otherwise it is reset.
7. **Trap flag**: It is used for single step control. When it is set to 1, a program can run on single step mode. When an interrupt is recognized it is automatically cleared.
8. **Interrupt flag**: If it is set to 1 the maskable interrupt INTR of 8086 is enabled and if it is 0 the interrupt is disabled.
9. **Directional flag**: It is used in string operations. If it is set to 1 string bytes are accessed from high memory address to low memory address. When it is 0 they are accessed from low memory address to high memory address.

2-3-6 Functional Units of 8086

The architecture of 8086 is divided into two parts that are Bus Interfacing Unit (BIU) and Execution Unit (EU).

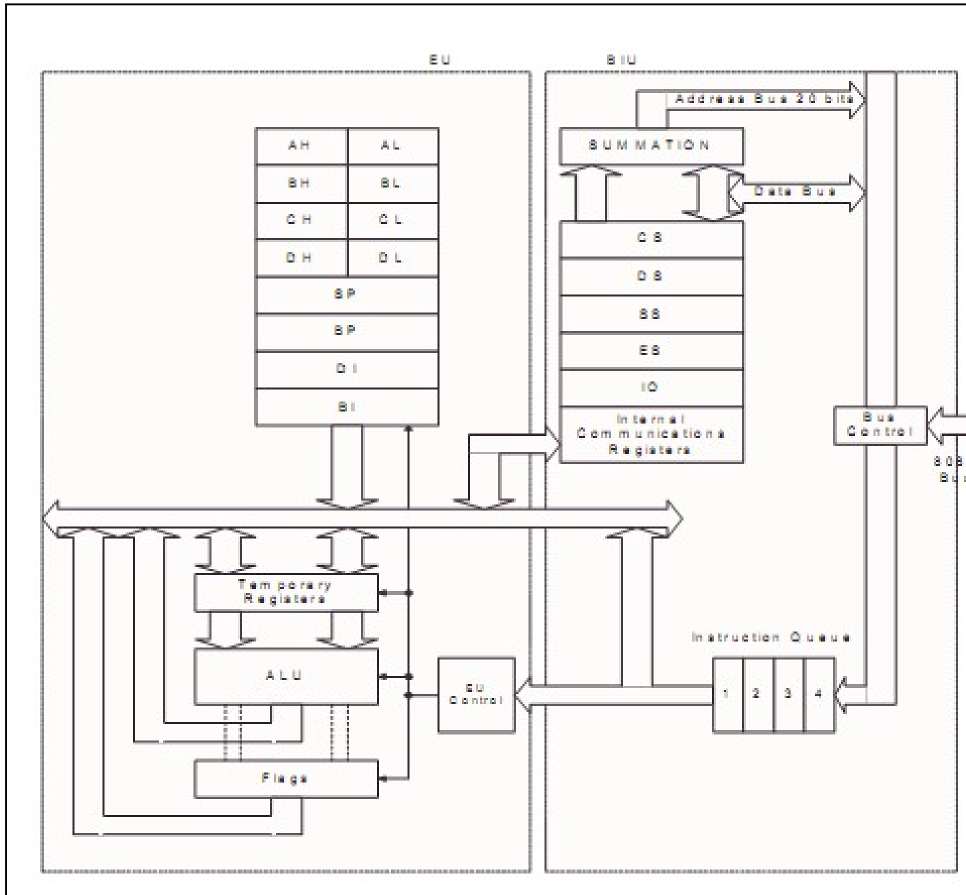


Fig 1.6 Block Diagram of 8086

BIU (Bus interfacing unit): BIU reads data from memory or ports and writes into memory of ports. It handles all interfaces with the external bus and generates external memory and I/O addresses.

It fetches instruction codes from the memory and keeps them into 6 bytes instruction queue. It fills up the queue whenever the bus is idle and there is space in queue.

The complete physical address of 20 bits length is generated using segment registers and offset register of each length 16 bit. Segment address is shifted left bit wise 4 times and the result is added to the offset address.

Example:

Segment address = 1005h

Offset address = 5555h

Physical address = 10050 + 5555 = 155A5h

Thus the segment value 1005 can have offset values from 0000 to FFFF.

While the processor internally executes an instruction fetched, the external bus remains free. This free time slot is utilized to overlap fetch and execute cycles. This concept is known as pipe lining. Now the external bus is used to fetch the machine code of next instruction and arrange it in a queue called as pre decoded instruction byte queue.

Instruction Queue:

It is of length 6 bytes and works on FIFO. The instructions from the queue are taken for decoding sequentially. Once a byte is decoded the queue is rearranged by pushing it out and the queue status is checked for the next opcode fetch cycle.

EU (Execution Unit):

The execution unit contains the register set of 8086 except segment registers and IP. It has 16 bit ALU to perform arithmetic and logical operations. A 16 bit flag register to reflect the status of operations performed by ALU. The decoding unit decodes the opcode byte issued from the instruction byte queue. The timing and control unit derives the necessary control signals to execute the instruction opcode received from the queue. By the help of these signals execution is performed. EU also informs BIU where to fetch instructions or read data from.

2-4 Memory segmentation

The memory in 8086 is divided into a number of logical segments. Each segment is of size 64KB and is addressed by one of the segment register. The complete 1MB physical memory is divided into 16 segments each of 64KB. The addresses of segments may be assigned as hexa decimal form 0000 to F000. The offset addresses range from 0000 to FFFF.

Segments may be overlapped or non-overlapped. If a segment starts at a particular address and its maximum size can be of 64KB. Another segment may start before these 64KB locations of the first segment. Then they are said to be overlapped.

The main advantage of segmented memory scheme is

1) Allows the memory capacity to be 1MB although the actual addresses to be handled are of 16 bits size.

2) Allows the placing of code, data and stack portions of the same program in different parts of memory.

3) Permits a program or data to be fit into different areas of memory each time program is executed.

Physical memory organization:

The 1MB physical memory is organized as odd bank and even banks, each of 512KB. Byte data with even address is transferred on D7-D0, while byte data with odd address is transferred on D15-D8 bus lines.

By the help of BHE and A0 we can select either even or odd or both the banks. If the processor fetches a word there are different possibilities like

1) Both the bytes may be data operands

2) Both the bytes may contain opcode bits.

3) One of the bytes may be opcode while the other may be data.

In referring word data BIU requires one or two memory cycles depending upon whether the starting byte is located at an even or odd address. It is always better to locate the word data at an even address. To read or write a complete word if it is located at an even address it requires only one cycle.

The locations from FFFF0 to FFFFF are reserved for operations including jump to initialization program and I/O processor initialization. The locations 00000 to 003FF are reserved for interrupt vector table.

2-5 Addressing modes of 8086

The way by which an operand is specified in an instruction is called addressing mode. An addressing mode specifies where the operand is available for current instruction to be executed.

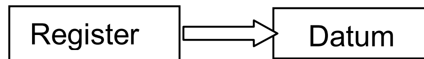
There are total eight addressing modes for 8086 instructions to specify operands.

Register Addressing Modes:

In this mode data is stored in a register and it is referred using particular register. All registers except IP can be used in this mode.

Example: MOV AX, BX

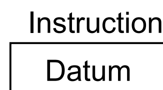
Instruction Register



Immediate Addressing Mode:

In this mode immediate data given by programmer which is a part of instruction is used as operand. This data is available in successive bytes of instruction code.

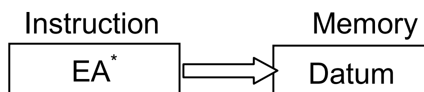
Example: MOV AX, 0005



Direct addressing Mode:

The operands will reside in memory and the address of memory is specified in the instruction as a part of it in successive bytes.

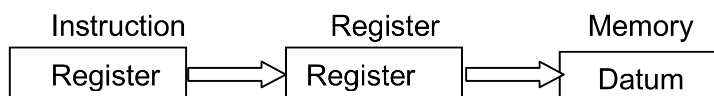
Example: MOV AX, (5000)



Register indirect addressing mode:

The operands will reside in memory and its address is specified indirectly by using offset registers. The offset address may be in either BX or SI or DI registers. The default segment is either DS or ES. The data is supposed to be present at the address pointed by above registers in the default segment

Example: MOV AX, (BX)



Indexed addressing Mode:

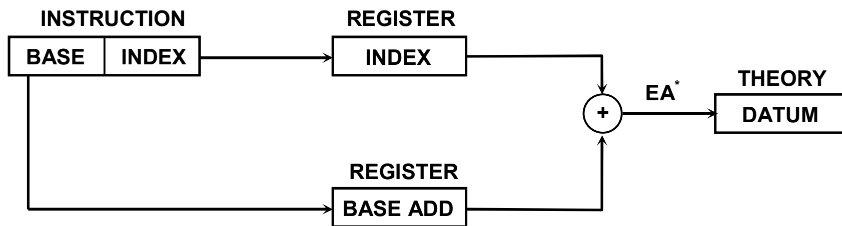
This is same as register indirect, but the offset of the operand is stored in one of the index registers. DS and ES are default segments and SI and DI are used for index registers.

Example: MOV AX, (SI)

Based indexed addressing mode:

This is same as register indirect, but the effective address of the operand is formed by adding contents of base register to the index registers. DS and ES are default segments, SI and DI are used for index registers and BX or BP is used as base registers.

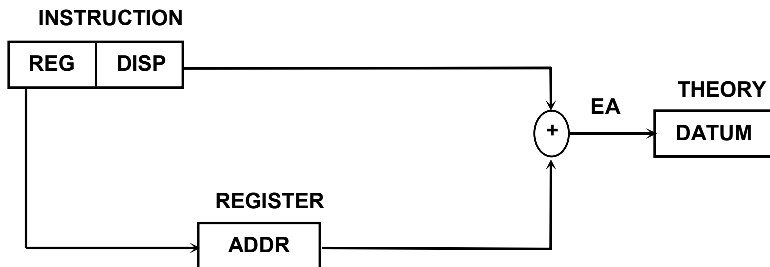
Example: MOV AX, (BX) (SI)



Register relative addressing mode:

The address of the operands is formed by adding 8-bit or 16 bit displacement with the content of any one of the registers BX, BP, SI, DI in the default segments.

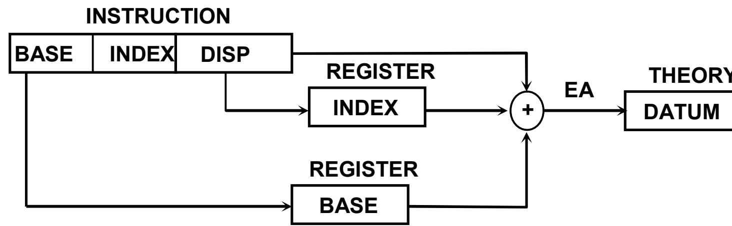
Example: MOV AX, 50(BX)



Relative Based Indexed Addressing Mode:

The address of the operands is formed by adding 8-bit or 16-bit displacement with the contents of any one of the base registers BX, BP and index register SI, DI in the default segments.

Example: MOV AX, 50[BX] [SI]

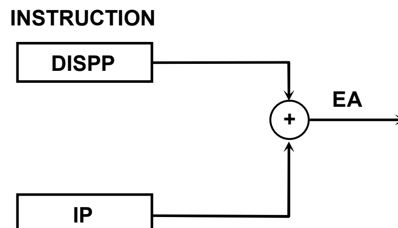


Intra segment/inter segment addressing modes:

If the location to which the control is to be transferred lies in a different segment other than the current one the mode is called inter segment mode. If the destination location lies in the same segment the mode is called intra segment mode.

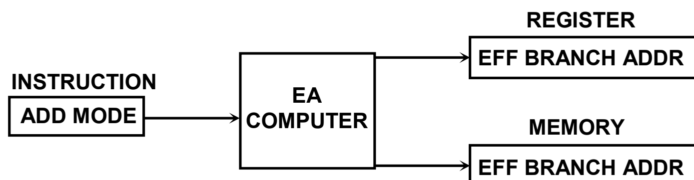
Intra segment Direct Mode:

The address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. The displacement is computed relative to the content of the instruction pointer IP. The effective address is the sum of 8 bit or 16 bit displacement and current content of IP.



Intra segment Indirect Mode:

The displacement to which the control is to be transferred is in the same segment in which the control transfer instruction lies but it is passed to the instruction directly. The addressing mode can be used in unconditional branch instructions. The branch address is found as the content of a register.



Example: The effective address in different addressing modes is calculated as shown below

Let displacement (offset) = 5000h

[AX] = 1000H [BX] = 2000H [SI] = 3000H [DI] = 4000H [BP] = 5000H [SP] = 6000H [CS] = 0000H [DS] = 1000H [SS] = 2000H [IP] = 7000H

1. Direct addressing mode:

DS: OFFSET = 1000: 5000
SHIFTING DS BY 4 BITS LEFT= 10000
OFFSET = 5000
EFFECTIVE ADDRESS = 15000h

2. Register Indirect:

DS: BX = 1000 : 2000
SHIFTING DS = 10000
[BX] = 2000
EFFECTIVE ADDRESS = 12000h

3. Register relative

DS: [5000 + BX) = 10000
SHIFTING DS = 5000
OFFSET = 2000
[BX] = 2000
EFFECTIVE ADDRESS = 17000h

4. Based Indexed

MOV AX, [BX] [SI]
DS: [BX + SI]
SHIFTING DS = 10000
[BX] = 2000
[SI] = 3000
EFFECTIVE ADDRESS = 15000h

5. Relative based indexed

DS: [BX + SI + 5000] = 10000
SHIFTING DS = 2000
[SI] = 3000
OFFSET = 5000
EFFECTIVE ADDRESS = 1A000h

3. INSTRUCTION SET OF 8086

Structure

3-1Introduction

3-2 Types of instructions

3-3 Data copy/transfer instructions

3-4 Arithmetic instructions

3-5 Logical Instructions

3-6 String Manipulation Instructions

3-7 Control Transfer or Branching Instructions

3-7-1 Unconditional Branch Instructions

3-7-2 Conditional Branch Instructions

Objectives

- Discuss varies types of instructions

3-1Introduction

Instruction: An instruction is a command given to processor to perform a specific operation, in binary form. Generally these instructions are written in mnemonics. The program that is written in the form of mnemonics is known as Assembly language program. This assembly language program is converted into machine language by the help of an assembler. If the assembler is not present, the user himself must convert the mnemonics in the ALP into respective operation codes.

Every instructions has two fields, they are op-code and operands. Op-code specifies what type operation must be performed. Where as operands, are the data elements on which the operation is carried out. Operands may be registers in CPU, memory locations or immediate data given by the programmer.

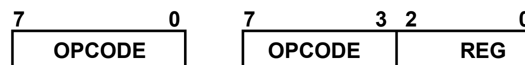
The instructions of a microprocessor, has one or more fields with in them. The first field is called opcode field which indicates the type if operation to be performed by CPU. The other fields are operand fields.

The CPU executes the instruction using the information which resides in these fields.

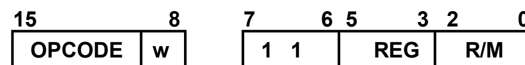
3-2 Types of instructions

There are six general formats of instructions in 8086. The length of instruction may vary from 1 byte to 6 bytes.

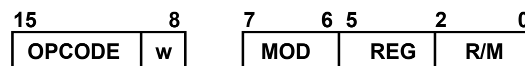
3-2-1 One byte instruction: The length of the instruction is 1 byte and may have the implied data or register operands. The least 3-bits of opcode are used to specify the register operands, otherwise 8-bits form opcode and the operands are implied.



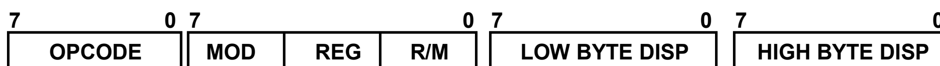
3-2-2 Register to Register: The format is 2 bytes long. First byte is opcode and width of the operand specified by w bit. The second byte of the code contains register and R/M (register or memory) fields.



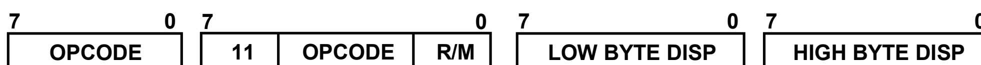
3-2-3 Register to/from Memory with no Displacement: The length of instruction is 2 bytes and similar to register to register format except for the MOD field. The MOD field shows the mode of addressing.



3-2-4 Register to/from Memory with Displacement: This instruction contains one or two additional bytes for displacement along with 2 byte format of register to/from memory without displacement.



3-2-5 immediate operand to register: The first byte and 3 bits from the second byte which are used for REG field in case of register to register format are used for opcode. It also contains 1 or 2 bytes of immediate data.



3-2-6 immediate operand to Memory with 16 bit displacement: It requires 5 to 6 bytes length. The firs 2 bytes contain the information regarding opcode, mode and R/M fields. The remaining 2 bytes of displacement and 2 bytes of data.The opcodes have the single bit indicators. Their definitions are given below.

NOTES

W-bit: Indicates, whether the instruction is to operate over an 8 bit or 16 bit data/operands. If it is 0 the operand is 8 bits long and if it is a 1 it is 16 bit long.

D-bit: It is valid in the case of double operand instructions. One of the operands must be a register specified by REG field. The register specified by REG is source operand if D = 0 else it is a destination operand.

Register Address (code)	Registers W=0	W=1	Segment code	Segment Register
000	AL	AX		
001	CL	CX		
010	DL	DX	00	ES
011	BL	BX	01	CS
100	AH	SP	10	SS
101	CH	BP	11	DS
110	DH	SI		
111	BH	DI		

Table: 3.1 Assignment of Codes with different registers

R/M	Memory operands With displacement			Register operands	
	0 bit	8-bit	16-bit		
	MOD :00	01	10	11 W=0	W=1
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16	AL	AX
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16	CL	CX
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16	DL	DX
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16	BL	BX
100	(SI)	(SI) + D8	(SI) + D16	AH	SP
101	(DI)	(DI) + D8	(DI) + D16	CH	BP
110	D16	(BP) + D8	(BP) + D16	DH	SI
111	(BX)	(BX) + D8	(BX) + D16	BH	DI

Table: 3.2 Addressing Modes and the Corresponding MOD, REG and R/M fields

S-bit: It is called as sign extension bit. The S bit is used along with W bit to show the type of the operation.

S	W	operation
0	0	8-bit operation with 8 bit immediate operand
0	1	16-bit operation with 16-bit immediate operand
1	1	16-bit operation with a sign extended immediate data

V-bit: This is used in the case of shift and rotates instructions. This bit is set to 0 if shift counts 1 and is set to 1, if CL contains the shift count.

Z-bit: This is used by REP instruction to control the loop. If Z bit is equal to 1 the instruction with REP prefix is executed until the zero flag matches the Z bit.

Segment registers are only 4 hence 2 bits are needed for them, other registers are 8 hence they need 3 bits for coding. To allow the use of 16 bit registers as two 8 bit registers they are code with W bit.

To find out MOD and R/M field of an instruction we must know addressing mode which specifies how the effective address may be computed

The 8086 instructions are categorized into the following types.

(i) Data Copy/ Transfer Instructions:

This type of instructions is used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this category.

(ii) Arithmetic and Logical Instructions:

All the instructions performing arithmetic, logical increment, decrement, compare and scan instructions belong to this category.

(iii) Branch Instructions:

These instructions transfer control of execution to the specified address. All the call, compare and scan instructions being to this class.

(iv) Loop Instructions:

If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ and LOOPZ instructions belong to this category. These are useful to implement different loop structures.

(v) Machine Control Instructions:

These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong t o this class.

(iv) Flag Manipulation Instructions:

All the instructions which directly affect the flag register, come under this group of instructions. Instructions like CLD, STD, CLI, STI, etc. belong to this category of instructions.

(vii) Shift and Rotate Instructions:

These instructions involve the bitwise shifting or rotation in either direction with or without a count in CX.

(viii) String Instructions:

These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the strings.

3-3 Data copy/Transfer instructions

MOV: Move

This data transfer instruction transfers data from one register/memory location to another register/memory location. The source may be any one of the segment registers other general or special purpose registers or a memory location and, another register or memory location may act as destination.

However, in the case of immediate addressing mode a segment register cannot be a destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general purpose register with the data and then it will have to be moved to that particular segment register.

Example: Load DS with 5000H.

1. MOV DS, 5000H; Not permitted (invalid)
   Thus to transfer an immediate data into the segment register, the correct procedure is given below.
2. MOV AX, 5000H
   MOV DS, AX
   It may be noted, here, that both the source and destination operands cannot be memory locations (Except for string instructions). Other MOV instruction examples are given below with the corresponding addressing modes.
3. MOV AX; 5000H ; Immediate
4. MOV Ax, BX ; Register
5. MOV AX, [SI] ; Indirect
6. MOV AX, [2000H] ; Direct
7. MOV AX, 50H[BX] ; Based relative, 50H displacement

PUSH: Push to Stack

This instruction pushes the contents of the specified register/memory location on the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address. The examples of these instructions are as follows:

Example:

1. PUSH AX
2. PUSH DS
3. PUSH [5000H] ; Contents of location 5000h and 5001h in DS are pushed onto the stack.

POP: Pop from Stack

This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 3.

Example

1. POP AX
2. POP DS
3. POP [500H]

ECHG: Exchange

This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a

memory location. However, exchange of data contents of two memory locations is not permitted.

Example:

1. XCHG [5000H], AX ; This instruction exchanges data between AX and a memory location [5000H] in the data segment.
2. XCHG BX ; This instruction exchanges data between AX and BX.

In: Input the port

This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly. AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is only register (implicit) which is allowed to carry the port address.

Example:

1. IN AL, 0300H ; This instruction reads data from an 8-bit pot whose address is 03000 H and stores it in AL.
2. IN AX ; This instruction reads data from a 16-bit port whose address is in DX (implicit) and stores it in AX.

Out: Output to the Port

This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D_8 D_{15} while that to an even addressed port is transferred on D_0 – D_7. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively.

Example:

1. OUT 0300H, Al ; This sends data available in AL to a whose address is 0300H.
2. OUT AX ; This sends data available in AX to a port whose address is specified implicitly in DX.

XLAT: Translate

The translate instruction is used for finding out the codes in case of code conversion problems, using look up table technique. We will explain this instruction with the aid of the following example.

A hexadecimal key pad having 16 keys from 0 to F is interfaced with 8086 using 8255. Whenever a key is pressed, the code of that key (0 to F) is returned AL. For displaying the number corresponding to the pressed key on the 7-segment display device, it is required that the 7-segment code corresponding to the key pressed is found out and sent to the display port. This translation from the code of the key pressed to the corresponding 7-segment code is performed using XLAT instruction.

For this purpose, one is required to prepare a look up table of codes, starting from an offset say 2000H, and store the 7-segment codes for 0 to F at the locations 2000H to 200FH sequentially. For executing the XLAT instruction, the code of the pressed key obtained from the keyboard (i.e. the code to be translated) is moved in AL and the base address of the look up table containing the 7-segment codes is kept in BX. After execution of the XLAT instruction, the 7-segment code corresponding to the pressed key is returned AL, replacing the key code which was in AL prior to the execution of the XLAT instruction. To find out the exact address of the 7-segment code from the base address of look up table, the content of AL is added to BX internally, and the contents of the address pointed to by this new content of BX in DS are transferred to AL.

Example

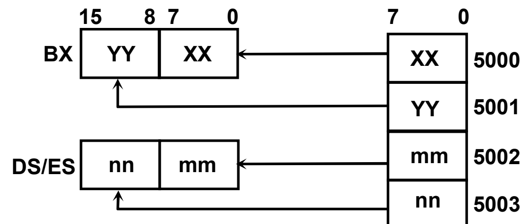
```
MOV AX, SEG TABLE          ; Address of the segment
                           containing look-up table
MOV DS, AX                   is transferred in DS.
MOV AL, CODE               ; Code of the pressed key
                             is transferred in AL.
MOV BX, OFFSET TABLE       ;Offset of the code look-up-table
                              in BX
XLAT; Find the equivalent code
                             and store in AL
```

LEA: Load Effective Address:

The load effective address instruction loads the offset of an operand in the specified register. This instruction is more useful for assembly language rather than for machine language. Suppose, in an assembly language program, a label ADR is used. The instruction LEA BX, ADR loads the offset of the label ADR in BX.

LDS/LES: Load Pointer to DS/ES

The instruction, Load DS/ES with pointer, loads the DS or ES register and the specified destination register in the instruction with the content of memory location specified as source in the instruction.



LAHF: Load AH from Lower Byte of Flag

This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except over flow) at a time.

SAHF: Store AH to Lower Byte of Flag Register:

This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

PUSHF: Push Flags to Stack

The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack

The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

3-4 Arithmetic instructions

These instructions usually perform the arithmetic operations, like addition subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. Arithmetic instructions affect all the condition code flags. The operands are either the contents of

registers or memory locations or immediate data depending upon the addressing mode.

ADD: Add

This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register or memory location. The result is in the destination operand. However, both destination operands cannot be memory operands. That means memory to memory is not possible. Also the contents of the segment registers cannot be added using this instruction. All condition code flags are affected, depending upon the result.

Example:

1. ADD AX, 0100H ; Immediate
2. ADD AX, BX ; Register
3. ADD AX, [SI] ; Register indirect
4. ADD AX, [5000 H] ; Direct
5. ADD [5000H], 0100H ; Immediate
6. ADD 0100H ; Destination AX (Implicit)

ADC: Add with Carry

This instruction performs the same operation as ADD instruction but adds the carry flag bit (which may be set as a result of the previous calculations to the result. All the condition code flags are affected by this instruction.

Example:

1. ADC 0100 H ; Immediate (AX implicit)
2. ADC AX, BX ; Register
3. ADC AX, [SI] ; Register Indirect
4. ADC AX, [5000H] ; Direct
5. ADC [5000H], 0100 H ; Immediate

INC: Increment

These instruction increments the contents of the specified register or memory location by 1. All condition code flags are affected except the carry flag CF. Instruction adds 1 to the contents of the operand. Immediate data cannot be operand of this instruction.

Example:

1) INC AX ; Register
2) INC [BX] ; Register indirect
3) INC [5000H] ; Direct

DEC: Decrement

The decrement instruction subtracts 1 from the contents of the specified register or memory location. All condition code flags except carry flag are affected depending upon the result. Immediate date cannot be operand of the instruction.

Example:

1. DEC AX ; Register
2) DEC [5000H] ; Direct

SUB: Subtract

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register, memory location or immediate data and the destination operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand cannot be an immediate data. All the condition code flags are affected by this instruction.

Example:

1) SUB 0100H ; Immediate [destination AX]
2) SUB AX, BX ; Register
3) SUB AX, [5000H] ; Direct
4) SUB [5000H], 0100 ; Immediate

SBB: Subtract with Borrow

Subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

The result is stored in the destination operand. All the flags are affected (condition code) by this instruction.

Examples:

1) SBB 0100H ; Immediate [destination AX]
2) SBB AX, BX ; Register
3) SBB AX, [5000H] ; Direct
4) SBB [5000H], 0100 ; Immediate.

CMP: Compare

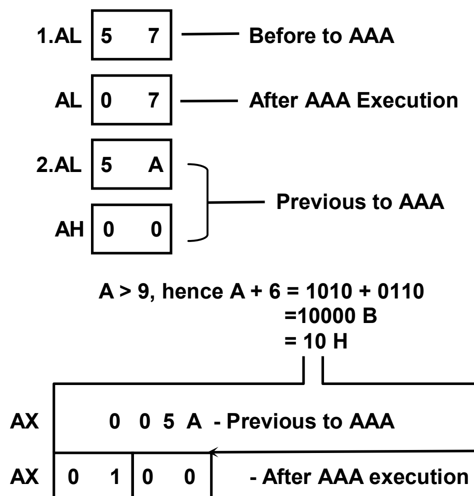
This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending upon the result of the subtraction. If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset.

Example:

1) CMP BX, 0100H ; Immediate
2) CMP 0100 ; Immediate [AX implicit]
3) CMP [5000H], 0100H ; Direct
4) CMP BX, [SI] ; Register indirect
5) CMP BX, CX ; Register

AAA: ASCII Adjust After Addition

The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4 high order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 are add to AL. The upper 4 bits of AL are cleared and AH is incremented by one. If the value in the lower nibble of AL is greater than 9 than 9 then the AL is incremented by 06, AH is incremented by 1, the AF and CF flags are set to 1, and the higher 4 bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030 H to AX.



AAS: ASCII Adjust AL After Subtraction:

AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4 bits of AL register are greater than 9 or if the AF flag is 1, the AL is decremented by 6 and AH register is decremented by 1, the CF and AF are set to 1. Other wise, the CF and AF are set to 0, the result needs no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9. The procedure is similar to the AAA instruction. AH is modified as difference of the previous contents (usually zero) of AH and borrow for adjustment.

AAM: ASCII Adjust for Multiplication:

This instruction, after execution, converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result (unpacked) remains in AL and the higher byte of result remains in AH. The example given below explains execution of the instruction. Suppose, a product is available in AL, say AL = 5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add 6(0110) to it D + 6 = 13 H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5 + 1 = 6 will be the upper unpacked byte of the result. Thus after the execution, AH = 06 and AL = 03.

AAD: ASCII Adjust for Division:

Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction 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 byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction. In the instruction sequence, this instruction appears before DIV instruction unlike AAM appears after MUL. Let AX contain 0508 unpacked BCD for 58 decimal, and DH contain 02 H.

Example:

AX	5	8	
AAD result in AL	0	3A	58d = 3Ah in AL

The result of AAD execution will give the hexadecimal number 3 A in AL and 00 in AH. Note that 3A is the hexadecimal equivalent of 58(decimal). Now, instruction DIV DH may be executed. So rather than ASCII adjust for division, it is ASCII adjust before division. All the ASCII adjust instructions are also called as unpacked BCD arithmetic

instructions. Now, we will consider the two instructions related to packed BCD arithmetic.

DAA: Decimal Adjust Accumulator

This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or if AF is set, it will add 06 to the lower nibble in AL. After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60 H to AL. The examples given below explain the instruction.

Example:

```
(i)   AL = 53, Cl = 29
      ADD Al, Cl              ; AL ← (AL) + (CL)
                              ; AL ← 53 + 29
                              ; AL ←7C
      DAA                     ; AL ← 7C + 06 (as C > 9)
                              ; AL ← 82
(ii)  AL = 73                 ; CL = 29
      ADD AL, CL              ; AL ← AL + CL
                              ; AL ← 73 + 29
                              ; AL ← 9C
      DAA                     ; AL ←02 and CF = 1
      AL = 7 3
         +
      CL = 2 9
      --------
          9 C
          + 6
      --------
          A 2
        + 6 0
      --------
   CF = 1 0 2 in AL
```

The instruction DAA affects AF, CF, PF and ZF flags. The OF is undefined.

DAS: Decimal Adjust After Subtraction

This instruction converts the result of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only if the lower nibble of AL is greater than 9, this instruction will subtract 06 from lower nibble of AL. If the result of subtraction sets the carry flag or if upper nibble is greater than 9, this will subtracts 60 H from AL. This instruction modifies the AF, CF, SF, PF and ZF flags. The OF is undefined after DAS instruction. The examples are as follows:

Example

(i)	AL = 75	BH – 46
	SUB AL, BH	; AL ← 2 F = (AL) – (BH)
		; AF = 1
	DAS	; Al ← 2 9 (as F > 9, F – 6 = 9)
ii)	AL = 38	CH = 6 1
	SUB AL, CH	; AL ← D 7 CF = 1 (borrow)
	DAS	; AL ← 7 7 (as D > 9, D – 6 = 7)
		; CF = 1 (borrow)

NEG: Negate

The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand which may be a register or memory location. If OF is set, it indicates that the operation could not be completed successfully. This instruction affects all the condition code flags.

MUL: Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. The example instructions are as shown. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is 0 CF and QF both will be set.

Example

1)	MUL BH	; (AX) ← (AL) × (BH)
2)	MUL CX	; (DX) (AX) ← (AX) × (CX)
3)	MUL WORD PTR [SI]	; (DX) (AX) ← (AX) × ([SI])

IMUL: Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF and ZF flags are undefined after IMUL. If AH and DX contain parts of 16 and 32-bit result respectively. CF and OF both will be set. The AL and AX are the implicit operands in

case of 8 bits and 16 bits multiplications respectively. The unused higher bits of the result are filled by sign bit and CF, AF are cleared.

Example

1) IMUL BH
2) IMUL CX
3) IMUL [SI]

CBW: Convert Signed Byte to Word

This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

CWD: Convert Signed Word to Double Word

This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

DIV: Unsigned Division

This instruction performs unsigned division. It divides an unsigned word or doubled word by a 16 bit or 8 bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0 (divide by zero) interrupt is generated. In case of a double word dividend (32 bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder in this case, will be in AX and DX respectively. This instruction does not affect any flag.

IDIV: Signed Division

This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word divisions. The results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by 0 interrupt is generated, if the result is too big to fit in AX(16-bit dividend operation) or AX and DX (32-bit dividend operation). All the flags are undefined after IDIV instruction.

NOTES

3-5 Logical Instructions:

These type of instructions are used for carrying out the bit by bit shift, rotate, or basic logical operations. All the condition code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set are AND, OR, NOT and XOR.

AND: Logical AND

This instruction bit by bit ANDs the source operand that may be an immediate, a register or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand. At least one of the operands should be a register or a memory operand. Both the operands cannot be memory locations or immediate operands. An immediate operand cannot be a destination operand.

Example:

1) AND AX, 0008H
2) AND AX, BX
3) AND AX, [5000H]
4) AND [5000H], DX

If the content of AX is 3F0FH, the first example instruction will carry out the operation as given below. The result 3F9FH will be stored in the AX register:

0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1	= 3FOF H[AX]
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	AND
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008H
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0008 H [AX]

The result 008H will be in AX.

OR: Logical OR

The OR instruction carries out the OR operation in the same way as described in case of the AND operation. The limitations on source and destination operands are also the same as in case of AND operation. The examples are as follows:

Example:

1) OR AX, 0098H
2) OR AX, BX
3) OR AX, [5000H]
4) OR [5000H], 0008H

The contents of AX are say 3FOFH, then the first example instruction will be carried out as given below.

0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1	= 3F0F H
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	OR
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	= 0098H
0 0 1 1	1 1 1 1	1 0 0 1	1 1 1 1	= 3F9F H

Thus the result 3F9FH will be stored in the AX register.

NOT: Logical Invert

The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:

Example

NOT AX

NOT [5000H]

In the content of AX is 200FH, the first example instruction will be executed as shown.

AX =	0 0 1 0	1 1 1 1	0 0 0 0	1 1 1 1
Invert	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓
	1 1 0 1	1 1 1 1	1 1 1 1	0 0 0 0
Result In AX =	D	F	F	0

The result DFFOH will be stored in the destination register AX.

XOR: Logical Exclusive OR

The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on the operands are also similar. The XOR operation gives a high output, when the 2 input bits are dissimilar. Otherwise, the output is zero.

Example:

1) XOR AX, 0098H
2) XOR AX, BX
3) XOR AX, [5000H]

If the content of AX is 3F0FH, then the first example instruction will be executed as explained. The result 3F97H will be stored in AX.

AX = 3F0FH =	0 0 1 1	1 1 1 1	0 0 0 0	1 1 1 1
XOR	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	↓ ↓ ↓ ↓
0098H =	0 0 0 0	0 0 0 0	1 0 0 1	1 0 0 0
AX = Result =	0 0 1 1	1 1 1 1	1 0 0 1	1 1 1 1
=	3	F	9	7

TEST: Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to 1, if the corresponding bits of both operands are 1, else the result bit is reset to 0. The result of this anding operation is not available for further use, but flags are affected. The affected flags are OF, CF, SF, ZF and PF.

SHL: Shift logical Left

This instruction performs bit-wise left shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. This instruction shifts the operand through carry flag.

SHR: Shift Logical Right

This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. This instruction shifts the operand through carry flag.

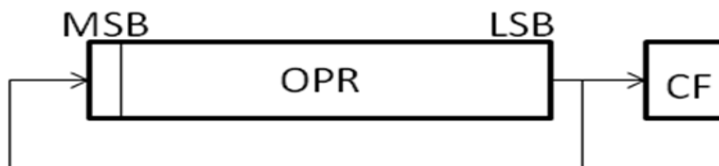
SAR: Shift Arithmetic Right

This instruction performs right shifts on the operand word or byte that may be a register or a memory location by the specified count in the instruction and inserts the most significant bit of the operand in the newly inserted positions. The result is stored in the destination operand. All the condition code flags are affected. This shift operation shifts the operand through carry flag. Immediate operand is not allowed in any of the shift instructions.

ROR: Rotate Right without Carry:

This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions.

The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it cannot be an immediate operand. The destination operand may be a register (except a segment register) or a memory location.

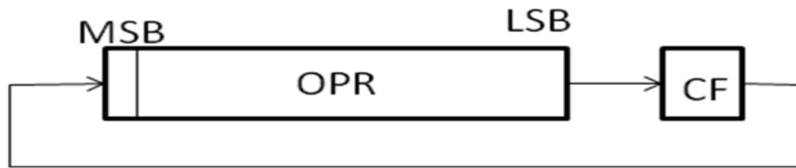


ROL: Rotate Left without Carry

This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF, and ZF flags are left unchanged by this rotate operation. The operand may be a register or a memory location.

RCR: Rotate Right through Carry

This instruction rotates the contents (bit-wise) of the destination operand right by the specified right by the specified count through carry flag (CF). For each operation, the carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location.



RCL: Rotate Left through Carry

This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB, and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location.

The count for rotation of shifting is either 1 or number of times specified using register CL, in case of the shift and rotate instructions.

3-6 String Manipulation Instructions

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte strings or word strings.

For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent.

For referring to a string, two parameters are required, (a) starting or end address of the string and (b) length of the string.

The length of a string is usually stored as count in CX register. In case of 8085, similar structures can be set up by the pointer and counter arrangements. The pointers and counters may be modified at each iteration, till the required condition for proceeding further is satisfied. On the other hand, the 8086 supports a set of more powerful instructions for string manipulations. The incrementing or decrementing of the pointer, in case of 8086 string instructions, depends upon the direction flag (DF) status. If it is a byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two. The counter in both the cases is decremented by one.

REP: Repeat Instruction Prefix:

This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one). When CX becomes zero, the execution proceeds to the next instruction in seque4nce. There are two more options of the REP instruction. The first is REPE/REPZ, i.e. repeat operation while equal/zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB/MOVSW: Move String Byte or String Word

Suppose a string of bytes, stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of the source string is located in the memory location whose address may be computed using SI (Source Index) and DS (data segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (destination index) and ES (extra segment) contents. The starting address of the source string is 10H*DS + (SI), while the starting address of the destination string is 10H*ES + [DI]. The MOVSB/MOVSW instruction thus, moves a string of bytes/ words pointed to by DS: S1 pair (source) to the memory location pointed by ES: DI pair(destination). The REP instruction prefix is used with MOVS instruction to repeat it by a value given in the counter (CX). The length of the byte string or word string must be stored in CX register. No flags are affected by this instruction.

After the MOVES instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e. SI and DI depend upon the direction flag DF. If DF is 0, the index registers are incremented, otherwise, they are decremented, in case of all the string manipulation instructions. The following string of instructions explain the execution of the MOVES instruction

Example:

MOV AX, 5000H ; Source segment address is 5000H.

MOV DS, AX ; Load it to DS

MOV AX, 6000H ; Destination segment address is 6000h.

MOV ES, AX ; Load it to ES.

MOV CX, 0FFH ; Move length of the string to counter register CX.

MOV SI, 1000H ; Source index address 1000H is moved to SI.

CLD ; Clear DF, i.e. set auto increment mode.

REP MOVSB ; Move 0FFH string bytes from source address to destination.

CMPS: Compare String Byte or String Word

The CMPS instruction can be used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS: SI and ES: DI point to the two strings. The REP instruction prefix is used to repeat the operation till CX(counter) becomes zero or the condition specified by the REP prefix is false.

The comparison of the string starts from initial byte or word of the string, after each comparison the index registers are updated depending upon the direction flag and the counter id decremented. This byte by byte or word by word comparison continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

Example:

```
MOV AX, SEG1       ; Segment address of STRING1, i.e. SEG1 is moved to AX.

MOV DS, AX         ; Load it to DS.

MOV AX, SEG2       ; Segment address of STRING2, i.e. SEG2 is moved to AX.

MOV ES, AX         ; Load it to ES

MOV SI, OFFSET STRING1; Offset of STRING1 is moved to SI.

MOV DI, OFFSET STRING2; Offset of STRING2 is moved to DI

MOV CX, 010H       ; Length of the string is moved to CX.

CLD                ; Clear DF, i.e. set auto increment mode.

REPE CMPSW  ; Compare 010H words of STRING1 and STRING2, while they are equal, If a mismatch is found, modify the flags and proceed with further execution.
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set and otherwise ZF is reset.

SCAS: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES: DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string as stated in case of MOVSB instruction. Whenever a match to the specified operand is found in the string, execution stops and the zero flag is set. If no match is found, the zero flag is reset. The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found.

Example:

MOV AX, SEG ; Segment address of the string, i.e. SEG is moved to AX.

MOV ES, AX ; Load it to ES

MOV DI, OFFSET ; String offset, i.e. OFFSET is moved to DI.

MOV CX, 010H ; Length of the string is moved to CX.

MOV AX, WORD ; The word to be scanned for, i.e. WORD is in AL.

CLD ; Clear DF.

REPNE SCASW ; Scan the 010H bytes of the string, till a match to WORD is found.

This string of instructions finds out, if it contains WORD. If the WORD is found in the word string, before CX becomes zero, the ZF is set, otherwise the ZF is reset. The scanning will continue till a match is found. Once match is found the execution of the program proceeds further.

LODS: Load String Byte or String Word

The LODS instruction loads the AL/AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF. If it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other flags are affected by this instruction.

STOS: Store String Byte or String Word

The STOS instruction stores the AL/LX register contents to a location in the string pointed by ES: DI register pair . The DI is modified accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each iteration automatically. If DF = 1, then the execution follows auto decrement mode. In this mode, SI and DI are decremented automatically after each iteration (by 1 or 2 depending upon byte or word operations). Hence, in auto decrementing mode, the strings are referred to by their ending addresses. If DF = 0, then the execution follows auto increment mode. In this mode SI and DI are incremented automatically (by 1 or 2 depending upon byte or word operation) after each iteration, hence the strings, in this case, are referred to by their starting addresses.

3-7 Control Transfer or Branching Instructions

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred. Depending upon the addressing modes specified in Chap. 1, the CS may or may not be modified.

These types of instructions are classified in two types:

Unconditional Control Transfer (Branch) Instructions

In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

Conditional Control Transfer (Branch) Instructions

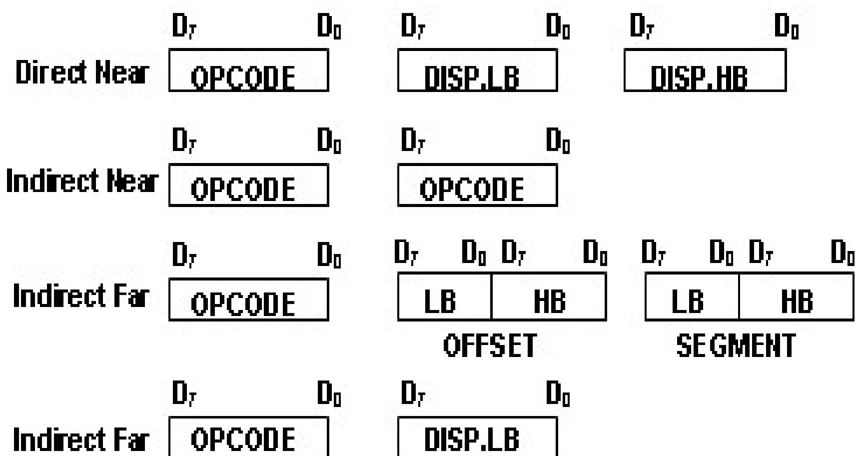
In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. The results of the previous operations are replicated by condition code flags.

In other words, using this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

3-7-1 Unconditional Branch Instructions

CALL: Unconditional Call

This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeable with subroutine. The address of the procedure may be specified directly or indirectly depending upon the addressing mode. There are again two types of procedures depending upon whether it is available in the same segment (Near CALL, i.e. ± 32K displacement) or in another segment (Far CALL, i.e. anywhere outside the segment). The modes for them are respectively called as intra-segment and intersegment addressing modes. This instruction comes under unconditional branch instructions and can be described as shown with the coding formats. On execution, this instruction stores the incremented IP (i.e. address of the next instruction) and CS onto the stack along with the flags and loads the CS and IP registers, respectively, with the segment and offset addresses of the procedure to be called.



RET: Return from the Procedure:

At each CALL instruction, the IP and CS of the next instruction are pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. The procedure may be a near or a far procedure. In case of a FAR procedure, the current contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1) return within segment

2) return within segment adding 16-bit immediate displacement to the SP contents.

3) return intersegment

4) return intersegment adding 16-bit immediate displacement to the SP contents.

INT N: Interrupt Type N

In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from 00H to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication ($N \times 4$) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4(offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine. For the execution of this instruction, the IF must be enabled.

Example:

Thus the instruction INT 20H will find out the address of the interrupt service routine as follows:

INT 20H

Type * 4 = 20 * 4 = 80H

Pointer to IP and CS of the ISR is 0000: 0080 H

Figure shows the arrangement of CS and IP addresses of the ISR in the interrupt vector table.

Memory Contents

15 8	7 0
CS high	CS Low

15 8	7 0
IP high	IP Low

Memory	Address
CS High	0000 : 0083
CS Low	0000 : 0082
IP High	0000 : 0081
IP Low	0000 : 0080

INTO: Interrupt on Overflow

This is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000 : 0000 as explained in INT type instruction. This is equivalent to a type 4 interrupt instruction.

JMP: Unconditional Jump

This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement (intrasegment relative, short or long) or CS: IP (intersegment direct far). No flags are affected by this instruction. Corresponding to the three methods of specifying jump addresses, the JUMP instruction has the following three formats.

```
JUMP    DISP 8-bit      ;Intrasegment, relative, near jump

JUMP    DISP. 16-bit    ;Intrasegment, relative, Far jump

JUMP    CS:IP           Intersegment, direct, jump
```

IRET: Return from ISR

When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the values of IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

LOOP: Loop Unconditionally

This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. The following sequence explains the execution. At each iteration, CX is decremented automatically. In other words, this instruction implements DECREMENT COUNTER and JUMP IF NOT ZERO structure.

Example:

```
        MOV CX, 0005    ; Number of times in CX

        MOV BX, 0ff7H   ; Data to BX

Label:  MOV AX, CODE1

        OR BX, AX

        AND DX, AX
```

Loop Label

The execution proceeds in sequence, after the loop is executed, CX number of times. If CX is already 00H, the execution continues sequentially. No flags are affected by this instruction.

3-7-2 Conditional Branch Instructions

When these instructions are executed, they transfer execution control to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied, otherwise, the execution continues sequentially. The conditions, here, means the status of condition code flags. These type of instructions do not affect any flag. The address has to be specified in the instruction relatively in terms of displacement which must lie with in – 80H to 7FH (or – 128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it lies wihtin the above specified range. The different 8086/8088 conditional branch instructions and their operations are listed in Table.

Conditional Branch Instructions

	Mnemonic	Displace ment	Operation
1	JZ/JE	Label	Transfer execution control to address 'Label', if ZF = 1
2	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF = 0
3	JS	Label	Transfer execution control to address 'Label', if SF = 1
4	JNS	Label	Transfer execution control to address 'Label', if SF = 0
5	JO	Label	Transfer execution control to address 'Label', if OF = 1
6	JNO	Label	Transfer execution control to address 'Label', if OF = 0
7	JP/JPE	Label	Transfer execution control to address 'Label', if PF = 1
8	JNP	Label	Transfer execution control to address 'Label', if PF = 0
9	JB/JNAE/J C	Label	Transfer execution control to address 'Label', if CF = 1
10	JNB/JAE/J	Label	Transfer execution control to address

NOTES

	NC		'Label', if CF = 0
11	JBE/JNA	Label	Transfer execution control to address 'Label', if CF = 1 or ZF = 1
12	JNBE/JA	Label	Transfer execution control to address 'Label', if CF = 1 or ZF = 0
13	JL/JNGE	Label	Transfer execution control to address 'Label', if SF = 1 nor OF = 1
14	JNL/JGE	Label	Transfer execution control to address 'Label', if SF = 0 nor OF = 0
15	JNE/JNC	Label	Transfer execution control to address 'Label', if ZF = 1 or neither SF nor OF is 1.
16	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF = 0 or atleast any one of SF and OF is 1(Both SF & OF are not 0)

The last four instructions are used in case of decisions based on signed binary number operations, while the remaining instructions can be used for unsigned binary operations. The terms above and below are generally used for unsigned numbers, while the terms less and greater are used for signed numbers. A conditional jump instruction, that does not check status flags for condition testing, is given as follows:

JCXZ Label; Transfer execution control to address Label, if CX = 0

The conditional LOOP instructions are given in Table with their meanings. These instructions many be used for implementing structures like DO_WHILE, REPEAT_UNTIL, etc.

Conditional Loop Instructions

Mnemonic	Displacement	Operation
LOOPZ/ LOOPE	Label	Loop through a sequence of instructions from 'Label' while ZF = 1 and CX $\neq$ 0.
LOOPNZ/ LOOPENE	Label	Loop through a sequence of instructions from 'Label' while ZF = 0 and CX $\neq$ 0.

The ideas about all these instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types:

(a) Flag manipulation instructions and
(b) Machine control instructions.

The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

CLC - Clear carry flag

CMC – Complement carry flag

STC – Set carry flag

CLD – Clear direction flag

STD – Set direction flag

CLI – Clear interrupt flag

STI – Set interrupt flag

There instructions modify the carry (CF), direction (DF) and interrupt (IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and auto increment or auto decrement modes. Thus the respective instructions may also be called as machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions, in this text. No direct instructions are available for modifying the status flags except carry flag.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. There machine control instructions do not require any operand.

WAIT - Wait for Test input pin to go low

HLT - Halt the processor

NOP - No operation

ESC - Escape to external device like NDP

(Numeric co-processor)

LOCK - Bus lock instruction prefix.

After executing the HLT instruction, the processor enters the half state, as explained in Chapter 1. The two ways to pull it out of the half state are to reset the processor or to interrupt in when NOP instruction is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP by one. It then continues with further execution after 4 clock cycles. ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. The LOCK prefix may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems. The WAIT instruction when executed, holds the operation of processor with the current status till the logic level on the $\overline{TEST}$ pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the $\overline{TEST}$ pin goes low. Once the $\overline{TEST}$ pin goes low, it continues further execution.

4. Summary

In this unit, we have studied one of the most popular series of microprocessors, viz., Intel 8086. It serves as a base to all its successors, 8088, 80186, 80286, 80486, and Pentium. The successors of 8086 can be directly run on any one of it successors. Therefore, though, 8086 has become obsolete from the market point of view, it is still needed to understand advanced microprocessors.

To summarize the features of 8086, we can say 8086 has:

- a 16-bit data bus
- a 20-bit address bus
- CPU is divided into Bus Interface Unit and Execution Unit
- 6-byte instruction prefetch queue
- segmented memory
- 4 general purpose registers (each of 16 bits)
- instruction pointer and a stack pointer
- set of index registers
- powerful instruction set
- powerful addressing modes
- designed for multiprocessor environment
- available in versions of 5Mhz and 8Mhz clock speed.

5. Test yourself

1. If AX contains 1234H, what is the result of ADD AL,AH?

2. What is the state of each flag after ADD AL, AH in the previous question?

NOTES

3. Memory location 2000H has the word 5000H stored in it. What does each location contain after INC BYTE PTR [2000H]?

4. Repeat Question 3 for DEC WORD PTR [2000H].

5. What is the state of the zero flag after CMP CL.30H if CL does not contain 30H?

6. What instruction is needed to check whether the upper bytes of AX and BX are equal?

7. Show the instructions needed to multiply AX by 25. Assume the results are unsigned.

8. If DX contains OOEEH and AX contains 0980, what is the result of:

```
MOV BX, 0F0H
DIV BX
```

9. If DX contains 7C9AH, what is the result of NOT DX?

10. Write assembly language program to evaluate the expression
    a = b + c –d * e considering 16 bit. Take the input in consecutive memory locations and results also.

UNIT – II

1. MACHINE LEVEL PROGRAMS

Structure

1.1 Addition of two data bytes located in same segment
1-2 Addition of two data bytes located in different segments
1-3 Block transfer
1-4 Sum of an array
1-5 Sum of 16-bit array and result is 32-bit
1- 6 Largest number
1- 7 Smallest number
1- 8 Descending order

Objectives

After going through this unit you should be able to:

- define the need and importance of an assembly program;
- define the various directives used in assembly program;
- write a very simple assembly program with simple input – output services;

1.1 Addition of two data bytes located in same segment

Machine level programs are written in the form of mnemonics, hand coded by programmer and entered byte by byte and executed on 8086 based kit. But this procedure is tedious due to complex instruction set.

Analysis:

- Let us assume the two data bytes are present in DS(data segment) starting from 2000H in memory
- Let the offset of first byte is 0500H and second byte is 600H and result is stored in 0700H.
- Therefore an effective address of first byte is 20500H and that of second byte is 20600H and that of second byte is 20700H.

Algorithm:

- Initialize data segment.
- Read first byte from memory to any one of registers.
- Add the second byte in memory to that register.
- Move the result in the register to memory.

Program:

Mnemonics	***Comments***
MOV AX, 2000	; Data segment register is loaded
	; with 200H i.e.,
MOV DS, AX	; data segment is initialized to 2000H
MOV AX, [05000]	;First byte is moved from memory to AX.
ADD AX, [0600]	; Second byte is added to AX.
MOV [0700], AX	; Result in AX is stored in 2000 :
	; 0700H location
HLT	; Stop

Note: We cannot load segment registers direction, they must be loaded indirectly.

1-2 Addition of two data bytes located in different segment

Analysis:

- Let us assume the two date bytes are present in different segments starting from 2000H and 3000H in memory.
- Let the offset of first byte is 0500H and second byte is 0600H and result is stored in 0700H.
- Therefore effective addresses of first byte is 20500H and that of second byte is 30600H
- Result is also stored in other segment starting from 5000H and having an offset of 0700H.

Algorithm:

- Initialize data segment for first byte.

- Read first byte from memory to any one of registers.
- Initialize data segment for second byte.
- Read the second byte from memory to any one of registers.
- Add the second byte in memory to first byte.
- Initialize data segment for result.
- Move the result to memory.

Program:

```
Mnemonics          Comments
MOV CX, 2000       ;  Data segment register is loaded
                      with 200H i.e.,
MOV DS, CX         ;  data segment is initialized to 2000H
MOV AX, [0500]     ;First byte is moved from memory to AX.
MOV CX, 3000       ;  Data segment register is loaded
                      with 3000H i.e.,
MOV DS, CX         ;  data segment is initialized to 3000H
MOV BX, [600]      ;  Second byte is moved to BX.
ADD AX, BX         ;  Add contents of BX to AX.
MOV CX, 500        ;  Data segment register is loaded
                      with 5000H i.e.,
MOV DS, CX         ;  data segment is initialized to 5000H.
MOV [0700], AX     ;  Result in AX is stored in 5000 :
                      0700H location
HLT                ;  Stop
```

Note: Where segment changes it must be initialized.

1-3 Block transfer

To move 16 bytes long byte string from a memory location 0200H to 0300H in same segment

Analysis:

- Let us assume the 16 date bytes are present in Data segment starting from 2000H in memory.
- Let the offset of source is 0200H and destination is 0300H. These are loaded into index registers SI and DI.

- The length of the string is loaded into CX register

Algorithm:

- Initialize data segment.
- Initialize index registers.
- Read first byte form source.
- Move it into a register.
- Move the contents of register to destination.
- Increment index registers for next byte and decrement the count.
- Repeat the process until all the bytes are moved from source to destination.

Program:

```
      Mnemonics              Comments
      MOV AX, 2000       ;   Data segment register is loaded
                             with 200H i.e.,
      MOV DS, AX         ;   data segment is initialized to 2000H
      MOV SI, 0200H      ;   Initialize index registers with offsets.
      MOV DI, 0300H      ;
      MOV CX, 0010H      ;   Initialize count.
Back: MOV AX, [SI]       ;   First byte is moved from source to AX
      MOV [DI], AX       ;   First byte in AX is moved to destination.
      INC SI             ;   Increment index registers.
      INC DI
      DEC CX             ;   Decrement counter
      JNZ Back           ;   Jump to back if count is not 0.
      HLT                ;   Stop
```

Note: 1) Index registers must be used whenever we are dealing with strings.

2) When jump instruction is used, number of bytes to be moved forward or backward must be given in the place of label.

1-4 Sum of array

To find the 16-bit sum of data array containing 16-bit numbers.

Analysis:

- Let us assume the 16-bit numbers are present in Data segment starting from 2000H in memory.
- Let the offset of source is 0200H and loaded into index registers SI.
- The length of the string is loaded into CX register.

Algorithm:

- Initialize data segment.
- Initialize index registers and count register.
- Read first byte form source.
- Add it to a register.
- Increment index registers for next byte and decrement the count.
- Repeat the process until all the bytes are added.

Program:

```
      Mnemonics            Comments
      MOV AX, 2000      ;  Data segment register is loaded
                            with 200H i.e.,
      MOV DS, AX        ;  data segment is initialized to 2000H
      MOV SI, 0200H     ;  Initialize index registers with offsets.
      MOV CX, 0010H     ;  Initialize count.
      MOV AX, 0000H     ;  Initialize AX to 0000
Back: ADD AX, [SI]      ;  First byte is added from to AX
      INC SI            ;  Increment index registers.
      INC SI            ;
      LOOP Back         ;  Jump to back if count is not 0.
      MOV [0500], AX    ;  Store the sum in memory
      HLT               ;  Stop
```

1-5 Sum of 16-bit array and result is 32-bit

To find the 32-bit sum of data array containing 16-bit numbers.

Analysis:

- Let us assume the 16, 16-bit numbers are present in Data segment starting from 2000H in memory.
- Let the offset of source is 0200H and loaded into index registers SI.
- The length of the string is loaded into CX register

Algorithm:

- Initialize data segment.
- Initialize index registers and count register.
- Read first byte form source.
- Add it to a register.
- If the sum is greater than 16 bit add carry flag to another register.
- Increment index registers for next byte and decrement the count.
- Repeat the process until all the bytes are added.

Program:

Mnemonics		*Comments*
MOV AX, 2000	;	Data segment register is loaded
	;	with 200H i.e.,
MOV DS, AX	;	data segment is initialized to 2000H
MOV SI, 0200H	;	Initialize index registers with offsets.
MOV CX, 0010H	;	Initialize count.
MOV AX, 0000H	;	Initialize AX to 0000
MOV BX, 000H	;	Initialize BX to 0000
Back : ADD AX, [SI]	;	First is moved from source to AX
JAE Go	;	Jump to label go if there is not carry
INC BX	;	Increment BX if there is carry
Go: INC SI	;	Increment index registers.
INC SI		

```
LOOP Back        ;  Jump to back if count is not 0.
MOV [0500], AX   ;  Store the lower order sum in memory
MOV [0501], BX   ;  Store the higher order sum in memory
HLT              ;  Stop
```

Note: Since the sum is greater than 16-bit data, whenever there is a carry flag it is added to BX register and answer is available in both BX : AX registers.

1- 6 Largest number

To find the largest number in a given 9-bit data array

Analysis:

- Let us assume the 16 date bytes are present in Data segment starting from 2000H in memory.
- Let the offset of source is 0300H is loaded in SI.
- The length of the string 0010H is loaded into CX register

Algorithm:

- Initialize data segment.
- Initialize index registers.
- Initialize CX with count and AL with 00.
- Compare the first byte with AL and if it is greater than the contents in AL, move it into AL.
- Increment index registers for next byte and decrement the count.
- Repeat the process until all the bytes are compared.
- Store the largest number in memory.

Program:

Mnemonics ***Comments***

```
MOV AX, 2000H    ;  Initialize DS register with 2000H
MOV DS, AX
MOV SI, 02FFH    ;  Initialize SI with offset
MOV CX, 0010H    ;  Initialize CX with count.
MOV AL, 00H      ;  Initialize AL with 00
```

```
Back:INC SI          ;  Increment SI.
     CMP AL, [SI]    ;  Compare AL with next number.
     JAE Go          ;  If the number in AL is greater, jump
                     ;   to label go.
     MOV Al, [SI]    ;  Otherwise move the number to AL.
Go:  LOOP Back       ;  Repeat if CX is not 0
     MOV [0351], AL  ;  Store largest number in memory
     HLT             ;  Stop
```

Note: 1) Loop instruction includes both decrementing count CX and jump to label if count is not zero.

2) The label at the loop is number of bytes to go back.

1- 7 Smallest number

To find the smallest number in a given 16-bit data array.

Analysis:

- Let us assume the 5, 16-bit elements are present in Data segment starting from 2000H in memory.
- Let the offset of source is 0300H is loaded in SI.
- The length of the string 0005H is loaded into CX register

Algorithm:

- Initialize data segment.
- Initialize index registers.
- Initialize CX with count and AX and if it is smaller than the contents in AX, move it into AX.
- Increment index register for next element and decrement count.
- Repeat the process until all the bytes are compared.
- Store the largest number in memory.

Program:

```
        Mnemonics              Comments
        MOV AX, 2000H      ;   Initialize DS register with 2000H
        MOV DS, AX         ;
        MOV SI, 02FFH      ;   Initialize SI with offset.
        MOV CX, 0005H      ;   Initialize CX with count.
        MOV Ax, FFFFH      ;   Initialize AX with FFFF
Back:INC SI                ;   Increment SI.
        INC SI             ;
        CMP AX, [SI]       ;   Compare AX with next number
        JBE Go             ;   If the number in AX is really, jump
                               to label to.
        MOV AX, [SI]       ;   Otherwise move the number to AX.
Go:     LOOP Back          ;   Repeat if CX is not 0
        MOV [0351], AX     ;   Store largest number in memory
        HLT                ;   Stop
```

Note: 1) SI is incremented two times since the length of each element is 16 bits.

1- 8 Descending order

To arrange the given 16-bit data array in descending order.

Analysis:

- Let us assume there are 16-bit elements are present in Data segment starting from 2000H in memory.
- Let the offset of source is 0300H is loaded in SI and offset of destination 0500H is loaded in DI.
- The length of the string 005H is loaded into CX and DX register

Algorithm:

- Initialize data segment.
- Initialize index registers.
- Initialize CX and DX with count and AX with 0000.
- Compare the first element with AX and if it is greater than the contents in AX, go to next number.
- Otherwise save the address of largest number in BX.
- Increment source index registers for next element and decrement count CX.
- Repeat the process until all the bytes are compared.
- Store the largest number in destination.

- Increment destination register DI and decrement count DX.
- Repeat the process until the DX becomes 0.

Program:

```
        Mnemonics            Comments
        MOV AX, 2000       ;  Initialize the data segment register
                              with 2000H
        MOV DS, AX         ;
        MOV DI, 03FFFH     ;  Initialize DI with destination offset
        MOV DX, 0005H      ;  Initialize counter DX
Bounds: MOV SI, 02FFFH     ;  Initialize SI with source offset.
        MOV CX, 0005H      ;  Initialize counter CX.
        MOV AX, 0000H      ;  Initialize AX with 0000H.
Back:   INC SI             ;  Increment SI.
        INC SI             ;
        CMP AX, [SI]       ;  Compare previous largest number
                              with next number
        JAE Go             ;  Jump if there is no carry flag
        MOV AX, [SI]       ;  Largest number is moved to AX
        MOV BX, SI         ;  The address of largest number is
                              moved to BX.
Go:     LOOP Back          ;  Repeat until CX becomes 0.
        MOV [DI], AX       ;  Save the largest number
        MOV SI, BX         ;  Save the memory address of largest  in SI
        MOV [SI], 0000H    ;  Insert 0 in the memory containing
                           ;  largest number
        INC DI             ;  Increment DI
        INC DI             ;
        DEC DX             ;  Decrement DX register
        JNZ Bounds         ;  Jump if count is not 0
        HLT                ;  Stop
```

Note: 1) Two index registers are used to initiate two vectors one containing input data and other to store result

2) Each time largest number is found it is moved to destination and it is made 00.

2. PROGRAMMING WITH ASSEMBLER

Structure

2-1 Disadvantages of machine level programming
2-2 Programming with assembler
2-3 Types of assemblers:
2-4 Writing Assembly language Program
2-5 Executing an Assembly Language Program
2-6 Assembler directives
2-7 Assembly language programs

Objectives

- Discuss about Programming with assembler
- Discuss Types of assemblers
- write a very simple assembly program with simple input – output services

2-1 Disadvantages of machine level programming:

In machine level programming programmer must prepare the correct listing of machine codes (hex code) for the instructions written in the program. Such a programming procedure has following disadvantages.

1) It is time consuming
2) Errors are more while hand coding and entering program byte by byte.
3) Debugging is very difficult at machine level.
4) Programs are difficult to understand and the results are not present in user friendly form.

2-2 Programming with assembler

To overcome the above disadvantages of a program assembler is used to convert the mnemonics of instructions along with data into equivalent object code modules. These object code modules are further converted into executable format using linker and loaders. Advantages of assembly language over machine language are

1) Coding is performed by assembler rather than by programmer

2) Errors are less since we enter mnemonics instead of opcode

3) Constants, address locations etc can be labeled so that program becomes more user friendly.

4) It provides advanced features like macros, dynamic allocation etc.

2-3 Types of assemblers:

There are number of assemblers available such as MASM, TASM and DOS assembler.

Operations performed while executing an assembly level program using MASAM

1) Using text editors such as Norton's editor or Turbo C or EDLIN etc., type the program listing prepared by you. Start the procedure with following command.

C > NE

The following display appears of the screen

Enter File Name:
Press any key to continue

2) Enter the file name ESR.ASM, for every assembly language program extension is ASM. Now a new file ESR.ASM is opene and enters the program.

3) Next step is assembling the program using MASM. MASM accepts source file ESR.ASM and creates object file with .OBJ extension containing coded object modules of the program assembled.

4) In addition to object file, listing files with extension .LST containing labels, offset address, opcode, memory allotment for different labels, relocation information etc.

5) Cross reference file is also created with extension. CRF containing statistical information such as size of the source file, number of labels, routines called etc.

```
A > MASM ESR

Object filename [.OBJ]:

List filename[NUL.LST]:

Cross Reference [NUL.CRF]:
```

6) Further executable file is generated by the help of linker. DOS linking program LINK.EXE links the different object modules of a source program and function library routines to generate executable code of the source program. Input for the linker is .OBJ file.

```
A > LINK
Object Module [.OBJ]:
Run file [.EXE]
List filename [NUL.LST]:
Libraries [LIB]:
```

2-4 Writing Assembly language Program

An assembly language program needs a logical space called DATA segment to store data and CODE segment containing actual instructions to be executed. Some programs may need stack segment.

Skeleton of typical Assembly Language Program

```
DATA            SEGMENT

    Operands and results

DATA            ENDS

CODE            SEGMENT

      ASSUME CS: CODE, DS : DATA

      Instruction sequence

CODE            ENDS
```

1) ASSUME is directive to declare the label CODE is used as a logical name for CODE segment and label DATA is to be used for DATA segment. Labels CODE and DATA are reserved by MASM for these purpose only.

2) DATA SEGMENT is the starting address of a logical space DATA. CODE SEGMENT is the starting address of a logical space CODE.

3) DATA ENDS and CODE ENDS are to indicate end of data segment and code segment.

2-5 Executing an Assembly Language Program

1) Enter assembly language program using in built MASM text editor.

2) Select Built ALL function in PWB. If there is no error it will ask different options RUN, DEBUG, CANCE, and VIEW RESULT etc.

3) Select Run to execute program

4) If there is an error goes to step 1 by selecting view result and see the error and correct the ALP.

2-6 Assembler directives

An assembler directive is pseudo instruction used to give direction to the assembler to perform a specific task by assembly process. The assembler supports directives for data definition, Segment organization procedure and macro definitions etc.

Description of Some Directives:

DB: Define Byte is used define a byte type variable. It reserves one byte memory.

DW: Define word is used to define a word type variable. It reserves two bytes of memory

DD: Defines double word is used to define a 4 words type variable. It reserves 8 bytes of memory.

DQ: Define quad word is used to define a 4 words type variable. It reserves 8 bytes of memory.

DT: Define Ten Bytes is used to define a ten bytes type variable. It reserves 10 bytes of memory

DUP: It is used duplicate the given data element.

Example:

1) ARRAY DB 54, 39, 50, 80, 25

2) NAMES DB 100 DUP (?)

3) PRODUCT DW 1354, 2563

EXTERN: This is used tell the assembler that the labels following the directive are in some other module.

Example:

EXTERN MULTIPLIER: WORD The variable multiplier is external

GLOBAL: The variable declared as global can be accessed from other program modules.

SEGMENT: The directive SEGMENT indicates the beginning of a logical segment

General form: Name of the segment is SEGMENT

ENDS: It informs the assembler the end of the segment.

ENDP: It informs the assembler the end of the procedure.

END: It informs the end of program module.

ASSUME: It tells the assembler the name of the logical segment used for a specified segment.

EQU: This directive is used to assign name to some value.

ORG: This is used to assign address to data items or instructions in a program.

STRUCT: It is used to create a structure which is a collection of primary data types such As DB, DW etc.

RECORD: It is used to define bit pattern within a byte or word.

OFFSET: It is an operator which tells the assembler to determine the offset of a variable from starting address of a segment.

SIZE: It gives length of a data item in bytes.

LENGTH: It gives the number of elements in a data item.

2-7 Assembly language programs

Problem: 1

Write a program for addition of two numbers.

Analysis:

The two operands are defined in data segment and instruction sequence is given in code segment.

Program:

```
DATA      SEGMENT
OPR1      DW   1234H       ;  First operand
OPR2      DW    00002      ;  Second operand
RESULT    DW    DUP (?)    ;  One word memory is reserved
                              for result
DATA      ENDS
CODE      SEGMENT
          ASSUME CS: CODE, DS: DATA
START:    MOV AX, DATA     ;  Initialize data segment
          MOV DS, AX       ;
          MOVAX, OPRI      ;  Move first operand into AX
          MOV BX, OPR2     ;Move second operand to BX
          CLC              ;  Clear carry
          ADD AX, BX       ;  Add BX to AX

          MOV RESULT, AX   ;  Store the result in memory
          MOV AH, 4CH      ;  Return to DOS prompt
          INT 21H          ;  Break point
```

```
CODE      ENDS            ;  End code segment
          END START
```

Problem: 2

Program for the addition 8-bit number series, the series contains 100 (numbers).

Analysis:

Initially, the resulting sum of the first two numbers will be stored.

To this sum, the third number will be added.

This procedure will be repeated till all the numbers in the series are added.

A conditional jump instruction will be used to implement the counter checking logic.

Program:

```
DATA            SEGMENT         ; Data segments starts
    NUMLIST     DB    52H, 23H  ; List of byte numbers
    COUNT       EQU   100D      ; Number of bytes to be added
    RESULT      DW    DUP (?)   ; Word is reserved for result.
DATA            ENDS            ; Data segment ends
ORG             200H            ; Address 0200h in code segment
CODE            SEGMENTS        ; Code segment starts at relative
                ASSUME CS: CODE, DS: DATA
START:   MOV AX, DATA           ; Initialize data segment.
         MOV DS, AX
         MOV CX, COUNT     ; Number of bytes to be added in CX.
         XOR AX, AX             ;Clear AX and CF.
         XOR BX, BX    ; Clear BH for converting the byte to word
         LEA SI, NUMLIST        ; Point to the first number in the list.
```

```
AGAIN:  MOV BL, [SI]        ;Take the first number in the list.
        ADD AX, BX          ;Add AX with BX.
        INC SI              ; Increment pointer to the byte list.
        DEC CX              ;If all numbers are added, point to result
        JNZ AGAIN           ; Decrement counter.
        MOV RESULT, AX      ;Destination and store it.
        MOV AH, 4CH         : Return to DOS.
        INT 21H
CODE        ENDS
            END             START
```

Problem: 3

To find largest number for a given un-ordered array of 8-bit numbers and store in the location starting from known address.

Analysis:

Compare i^{th} number of the series with the $(i+1)^{th}$ number using CMP instruction.

It will set the flags appropriately, depending up on whether the ith number or the $(i + 1)^{th}$ number is greater.

If the ith number is greater than $(i + 1)^{th}$, leave it in AX (any register may be used).

Otherwise, load the $(i + 1)^{th}$ number in AX, replacing the ith number in AX.

The procedure is repeated till all the members in the array have been compared.

Program:

```
DATA            SEGMENT         ; Data segments starts
    LIST        DB    52H, 23H, 56H, 45H ;List of byte numbers
    COUNT       EQU   0FH          ; Number of bytes in the list
```

```
        LARGEST DB      DUP (?)       ; One byte is reserved for result
DATA            ENDS                  ;Data segment ends
CODE            SEGMENTS              ;Code segment starts
                ASSUME CS : CODE, DS : DATA
START:  MOV  AX, DATA                 ; Initialize data segment.
        MOV  DS, AX
        LEA   SI, LIST
        MOV  CL, COUNT                ; Number of bytes in CL.
        MOV   AL, [SI]                ;Take the first number in AL
AGAIN:  CMP  AL, [SI + 1)       ; and compare it with the next number
        JNL  NEXT
        MOV  Al, [SI + 1]
NEXT:   INC  SI          ; Increment pointer to the byte list.
        DEC CL                        ; Decrement counter.
        JNZ AGAIN    ;If all numbers are compared, point to result
        MOV  LARGEST, AL              ; destination and store it.
        MOV AH, 4CH                   ;Return to DOS.
        INT 21H
CODE            ENDS
                END                   START
```

Problem: 4

Program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

Analysis:

The simplest logic to decide whether a binary number is even or odd, is to check the least significant bit of the number.

If the bit is zero, the number is even, otherwise it is odd.

Check the LSB by rotating the number through carry flag, and increment even or odd number counter.

Program:

```
DATA            SEGMENT
       LIST  DW 2357H, 0A579H, 0C322H, 0C91EH, 0C00H, 0957H
       COUNT    EQU   006H
DATA            ENDS
CODE            SEGMENTS
                ASSUME CS : CODE, DS : DATA

START:     XOR BX, BX
           XOR DX, DX
           MOV AX, DATA
           MOV DS, AX
           MOV CL, COUNT
           LEA  SI, LIST
AGAIN:     MOV AX, [SI]
           ROR AX, 01
           JC ODD
           INC BX
           JMP NEXT
ODD:       INC DX
NEXT:      ADD SI, 02
           DEC CL
```

```
                JNZ AGAIN
                MOV AH, 4CH
                INT 21H
CODE                    ENDS
                        END START
```

Problem: 5

Program to find out the number of positive numbers and negative numbers from a given series of signed numbers.

Analysis:

Take the ith number in any of the registers. Rotate it left through carry. The status of carry flag, i.e. the most significant bit of the number will give the information about the sign of the number.

If CF is 1, the number is negative; otherwise, it is positive.

Program:

```
DATA                    SEGMENT
    LIST        DW      2357H, 0A500H, 0C009H, 0159H, 0B900H
    COUNT       EQU     05H
DATA                    ENDS
CODE                    SEGMENTS
                        ASSUME CS : CODE, DS : DATA
START:      XOR BX, BX
            XOR DX, DX
            MOV AX, DATA
            MOV DS, AX
            MOV CL, COUNT
            MOV SI, OFFSET LIST or LEA SI,LIST
```

```
AGAIN:      MOV AX, [SI]
            SHL AX, 01
            JC NEG
            INC BX
            JMP NEXT
NEG:        INC DX
NEXT:       ADD SI, 02
            DEC CL
            JNZ AGAIN
            MOV AH, 4CH
            INT 21H
    CODE            ENDS
                    END START
```

Problem: 7

Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

Analysis:

The first number of the series compared with the second one.

If the first number is greater than second, exchange their positions in the series. Otherwise leave the positions unchanged.

Then compare the second number in recent form of the series with third and repeat the exchange part that you have carried out for the first and the second number, and for all the remaining numbers of the series.

Program:

```
DATA                SEGMENT
     LIST      DW   53H, 25H, 19H, 02H
     COUNT     EQU   04
```

```
DATA        ENDS
CODE        SEGMENT
            ASSUME CS : CODE, DS : DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV DX, COUNT – 1
AGAIN0:     MOV CX, DX
            MOV SI, OFFSET LIST
AGAIN1:     MOV AX, [SI]
            CMP AX, [SI]
            CMP AX, [SI + 2]
            JL PRI
            XCHG [SI], AX
PR1:        ADD SI, 02
            LOOP AGAIN1
            DEC DX
            JNZ AGAIN0
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

Problem: 8

To perform one byte BCD addition.

Analysis:

It is assumed that the operands are in BCD form, but the CPU considers it hexadecimal and accordingly performs addition. Consider the following example for addition. Carry is set to be zero.

```
     9 2
   + 5 9
   -----
     E B        actual result after addition considering hex. Operands
       1 0 1 1
   +   0 1 1 0  As 0BH(LSD of addition) > 09, add 06 to it.
     1 0 0 0 1  Least significant nibble of result (neglect the auxiliary carry)
                → AF is set to 1
```

0110 is added to most significant nibble of the result if it is greater than 9 or AF is set.

```
                    1 carry from previous digit (AF)
                    E → 1 1 1 0
                      +  0 1 1 0
CF is set to 1           0 1 0 1       next significant nibble of result
Result CF           Most significant   Least significant digit
```

Program:

```
DATA        SEGMENT
    QPR1    EQU  92H
    QPR2    EQU  52H
    RESULT  DB   02 DUP(00)
DATA        ENDS
CODE        SEGMENT
```

```
                ASSUME CS : CODE, DS : DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV BL, OPR1
            XOR AL, AL
            MOV AL, OPR2
            ADD AL, BL
            DAA
            MOV RESULT, AL
            JNC MSB0
            INC [RESULT + 1]
MSB0:       MOV AH, 4CH
            INT 21H
CODE            ENDS
                END START
```

In this program, the instruction DAA is used after ADD. Similarly, DAS can be used after Sub instruction.

Problem: 9

Program that performs addition, subtraction, multiplication and division of the given operands. Perform BCD operation for addition and subtraction.

Program:

```
DATA            SEGMENT
        OPR1    EQU   98H
        OPR2    EQU   49H
        SUM     DW     01 DUP(00)
        SUBT    DW    01 DUP(00)
```

```
        PROD    DW   01 DUP(00)
        DIVS    DW   01 DUP(00)
DATA            ENDS
CODE            SEGMENT
                ASSUME CS : CODE, DS : DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV BL, OPR2
            XOR AL, AL
            MOV AL, OPR1
            ADD AL, BL
            DAA
            MOV BYTE PTR SUM, AL
            JNC MSB0
            INC [SUM + 1]
MSB0:       XOR AL, AL
            MOV AL, OPR1
            SUB AL, BL
            DAS
            MOV BYTE PTR SUBT, AL
            JNB MSB1
            INC [SUBT + 1]
MSB1:       XOR AL, AL
            MOV AL, OPR1
            MUL BL
```

```
            MOV WORD PRT PROD, AX
            XOR AH, AH
            MOV AL, OPR1
            DIV BL
            MOV WORD PTR DIVS, AX
            MOV AH, 4CH
            INT 21H
CODE            ENDS
                END START
```

Problem: 10

Program to find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

Analysis:

The given string is scanned for the given byte.

If it is found in the string, the zero flag is set; else, it is reset.

A count should be maintained to find out the relative address of the byte found out.

Program:

```
CODE            SEGMENT
                ASSUME CS : CODE, DS:DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV ES, AX
            MOV CX, COUNT
            MOV DI, OFFSET STRING
```

```
                MOV BL, 00H
                MOV AL, BYTE1
SCAN1:          NOP
                SCASB [DI]
                JZ XXX
                INC BL
                LOOP SCAN1
XXX:            MOV AH, 4CH
                INT 21H
CODE                  ENDS
DATA                  SEGMENT
          BYTE1       EQU   25H
          COUNT       EQU   06H
          STRING      DB 12H, 13H, 20H, 20H, 25H, 21H
DATA                  ENDS
                      END START
```

Problem: 11

Program to convert the BCD numbers 0 to 9 to their equivalent sever segment codes using the look-up table technique. Assume the codes (7-seg) are stored sequentially in CODELIST at the relative addresses from 0 to 9. The BCD number (CHAR) is taken in AL.

Analysis:

Refer to the explanation of the XLAT instruction. The statement of the program itself given the explanation about the logic of the program.

Program:

```
DATA                  SEGMENT
```

```
        CODELIST  DB    34, 45, 56, 45, 23, 12, 19, 24, 21, 00
        CHAR      EQU   05
        CODEC     DB    01H DUP(?)
DATA              ENDS
CODE              SEGMENT
                  ASSUME CS : CODE, DS : DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV BX, OFFSET CODELIST
            MOV AL, CHAR
            XLAT
            MOV BYTE PTR CODEC, AL
            MOV AH, 4CH
            INT 21H
CODE              ENDS
                  END START
```

Problem: 12

To check whether the parity of a given number is even or odd. If parity is even set DL to 00; else, set DL to 01. The given number may be a multi byte number.

Analysis:

The simplest algorithm to check the parity of a multi byte number is to go on adding the parity byte by byte with 00H. The result of the addition reflects the parity of that byte of the multi byte number. Adding the parities of all the bytes of the number, one will obtain the over all parity of the number.

Program:

```
DATA            SEGMENT
        NUM     DD      335A379BH
        BYTE_COUNT      EQU         04
DATA            ENDS
CODE            SEGMENT
                ASSUME CS: CODE, DS: DATA
START:          MOV AX, DATA
                MOV DS, AX
                MOV DH, BYTE_COUNT
                XOR AL, AL
                MOV CL, 00
                MOV SI, OFFSET NUM
NEXT_BYTE:      ADD AL, [SI]
                JP EVENP
                INC CL
EVENP:          INC SI
                MOV AL, 00
                DEC DH
                JNZ NEXT_BYTE
                MOV DL, 00
                RCR CL, 1
                JNC DL
CLEAR:          MOV AH, 4CH
                INT 21H
```

```
CODE            ENDS
                END START
```

The contents of CL are incremented depending upon either the parity for that byte is even or odd. If LSB of CL is 1, after testing for all bytes, it means the parity of the multi-byte number is odd otherwise it is even and DL is modified correspondingly

Problem : 13

Program for the addition of two 3 × 3 matrices. The matrices are stored in the form of lists (row wise). Store the result of addition in the third matrix.

Analysis:

In the addition of two matrices, the corresponding elements are added to form the corresponding elements of the result matrix as shown.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

[A] + [B] = [A + B]

The matrix A is stored in the memory at an offset MAT1, as given:

a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33} etc.

A total of 3 × 3 = 9 additions are to be done. The assembly language program is written as shown.

Program:

```
DATA            SEGMENT
        DIM     EQU   09H
        MAT1    DB    01, 02, 03, 04, 05, 06, 07, 08, 09
        MAT2     DB   01, 02, 03, 04, 05, 06, 07, 08, 09
        RMT3    DW    09H DUP(?)
DATA            ENDS
CODE            SEGMENT
```

```
            ASSUME CS: CODE, DS: DATA
START:      MOV AX, DATA
            MOV DS, AX
            MOV CX, DIM
            MOV SI, OFFSET MAT1
            MOV DI, OFFSET MAT2
            MOV BX, OFFSET RMAT3
NEXT:       XOR AX, AX
            MOV AL, [SI]
            ADD AL, [DI]
            MOV WORD PRT [BX], AX
            INC SI
            INC DI
            ADD BX, 02
            LOOP NEXT
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

Problem: 14

Program to find out the product of two matrices. Store the result in the third matrix.

Analysis:

The multiplication of matrices is carried out as shown

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11}+a_{12}b_{21}+a_{12}b_{31} & a_{12}b_{12}+a_{22}b_{12}+a_{13}b_{32} & a_{13}b_{13}+a_{12}b_{12}+a_{13}b_{23} \\ a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32} & a_{21}b_{13}+a_{22}b_{23}+a_{23}b_{23} \\ a_{31}b_{11}+a_{32}b_{21}+a_{33}b_{31} & a_{31}b_{12}+a_{32}b_{22}+a_{33}b_{32} & a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33} \end{bmatrix}$$

Program:

```
DATA            SEGMENT
        ROCOL   EQU   03H
        MAT1    DB    05H, 09H, 0AH, 03H, 02H, 07H, 03H, 00H, 09H
        MAT2    DB    09H, 07H, 02H, 01H, 0H 0DH, 7H, 06H, 02H
        PMAT3   DW    09H DUP(?)
DATA            ENDS
CODE            SEGMENT
                ASSUME CS : CODE, DS : DATA
START:          MOV AX, DATA
                MOV DS, AX
                MOV CH, ROCOL
                MOV BX, OFFSET PMAT3
                MOV SI, OFFSET MAT1
NEXTROW:        MOV DI, OFFSET MAT2
                MOV CL, ROCOL
NEXTCOL:        MOV DL, ROCOL
                MOV BP, 0000H
                MOV AX, 0000H
                SAHF
```

```
NEXT_ELE:       MOV AL, [SI]
                MUL BYTE PTR[DI]
                ADD BP, AX
                INC SI
                ADD DI, 03
                DEC DL
                JNZ NEXT_ELE
                SUB DI, 08
                SUB SI, 03
                MOV [BX], BP
                ADD BX, 02
                DEC CL
                JNZ NEXTCOL
                ADD SI, 03
                DEC CH
                JNZ NEXTROW
                MOV AH, 4CH
                INT 21H
CODE            ENDS
                END START
```

Problem: 15

Write a program to add two multi byte numbers and store the result as a third number. The numbers are stored in the form of the byte lists stored with the lowest byte first.

Analysis:

This program is similar to the program written for the addition of two matrices except for the addition instruction.

Program:

```
DATA            SEGMENT
        BYTES   EQU   08H
        NUM1    DB    05H, 5AH, 6CH, 55H, 66H, 77H, 34H, 12H
        NUM2    DB    04H, 56H, 04H, 57H, 32H, 12H, 19H, 13H
        NUM3    DB    0AH DUP(00)
DATA            ENDS
CODE            SEGMENT
                ASSUME CS : CODE, DS : DATA
START:          MOV AX, DATA
                MOV DS, AX
                MOV CX, BYTES
                MOV SI, OFFSET NUM1
                MOV DI, OFFSET NUM2
                MOV BX, OFFSET NUM3
                XOR AX, AX
NEXTBYTE:       MOV AL, [SI]
                ADC AL, [DI]
                MOV BYTE PRT[BX], AL
                INC SI
                INC DI
                INC BX
```

```
                DEC CX
                JNZ NEXTBYTE
                JNC NCARRY
                MOV BYTE PTR [BX], 01
NCARRY:         MOV AH, 4CH
                INT 21H
CODE            ENDS
                END START
```

3. MODULAR PROGRAMMING

Structure

3-1 Introduction
3-2 THE STACK
3.3 Far and Near Procedures
    3-3-1 Calls, Returns, and Procedures definitions in 8086
3.4 Parameter Passing in Procedures
3.5 External Procedures

Objectives

- Discuss about internal and external programming procedure

3-1 Introduction

Modular programming refers to the practice of writing a program as a series of independently assembled source files. each source file is a modular program designed to be assembled into a separate object file. Each object file constitutes a module. The linker collects the modules of a program into a coherent whole.

There are several reasons a programmer might choose to modularize a program.

1. Modular programming permits breaking a large program into a number of smaller modules each of more manageable size.

2. Modular programming makes it possible to link source code written in two separate languages. a hybrid program written partly in assembly language and partly in higher level language necessarily involves at least one module for each language involved.

3. Modular programming allows for the creation, maintenance and reuse of a library of commonly used modules.

4. Modules are easy to comprehend.

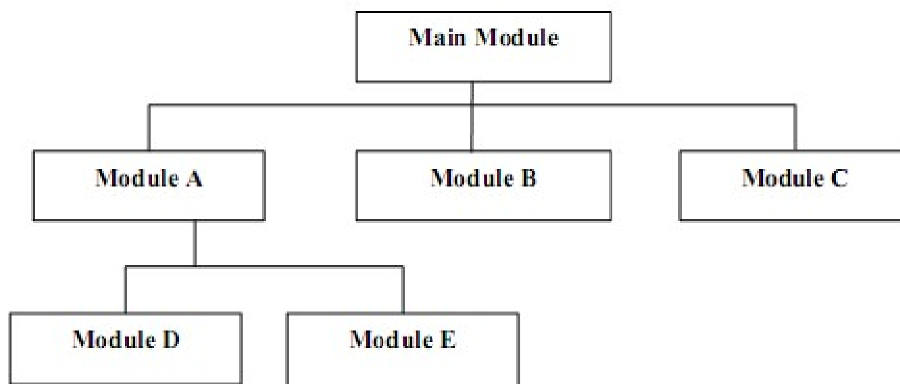
5. Different modules can be assigned to different programs.

6. Debugging and testing can be done in a more orderly fashion.

7. Document action can be easily understood.

8. Modifications may be localized to a module.

A modular program can be represented using hierarchical diagram:



The advantages of modular programming are:

1. Smaller, easier modules to manage

2. Code repetition may be avoided by reusing modules.

You can divide a program into subroutines or procedures. you need to call the procedure whenever needed. a subroutine call transfers the control to subroutine instructions and brings the control back to calling program.

3-2 THE STACK

A procedure call is supported by a stack. So let us discuss stack in assembly. Stacks are last in first out data structures, and are used for storing the return Addresses of the procedures and for parameter passing and saving the return value.

In 8086 microprocessor a stack is created in the stack segment. The ss register stores the offset of stack segment and sp register stores the top of the stack. A value is Pushed in to top of the stack or taken out (poped) from the top of the stack. The stack Segment can be initialized as follows:

```
STACK_ SEG          SEGMENT STACK
  DW 100   DUP (0)
  TOS LABEL   WORD
STACK_SEG           ENDS
 CODE               SEGMENT
```

```
ASSUME CS: CODE, SS: STACK_SEG
MOV   AX, STACK_SEG
MOV   SS,AX      ; initialise stack segment
LEA   SP,TOP     ; initialise stack pointer
CODE             ENDS
                 END
```

The directive stack_seg segment stack declares the logical segment for the Stack segment. DW 100 DUP(0) assigns actual size of the stack to 100 words. All Locations of this stack are initialized to zero. The stacks are identified by the stack top And that is why the label top of stack (TOS) has been selected. Please note that the Stack in 8086 is a WORD stack. Stack facilities involve the use of indirect addressing Through a special register, the stack pointer (SP). Sp is automatically decremented as Items are put on the stack and incremented as they are retrieved. Putting something on to stack is called a push and taking it off is called a pop. The address of the last Element pushed on to the stack is known as the top of the stack (TOS).

Name	Mnemonics	Description
Push onto the stack	PUSH SRC	SP←SP – 2 SP+1 and SP location are assign the SRC
Pop from the stack	POP DST	DST is a assigned values stored at stack top SP← SP + 2

3.3 Far and Near Procedures

Procedure provides the primary means of breaking the code in a program into modules. Procedures have one major disadvantage, that is, they require extra code to join them together in such a way that they can communicate with each other. This extra code is sometimes referred to as linkage overhead.

A procedure call involves:

1. Unlike other branch instructions, a procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.

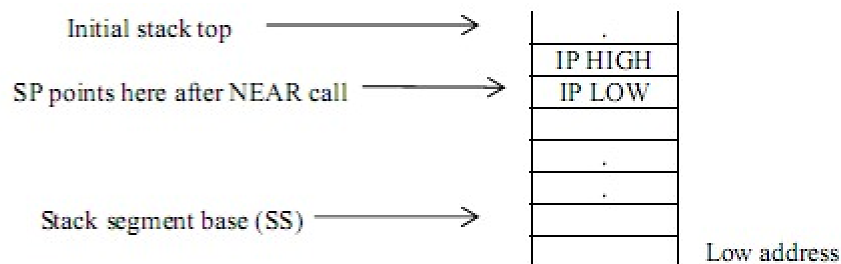
2. The registers used by the procedures need to be stored before their contents are changed and then restored just before the procedure is finished.

3. A procedure must have a means of communicating or sharing data with the procedures that call it, which is parameter passing.

3-3-1 Calls, Returns, and Procedures definitions in 8086

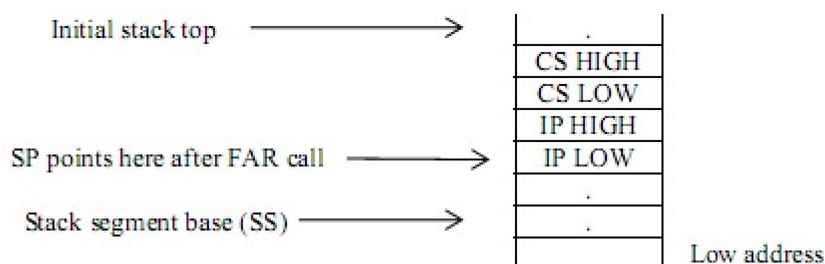
The 8086 microprocessor supports CALL and RET instructions for procedure call. The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. In addition, it also initialized IP with the address of the procedure. The RET instructions simply pops the return address from the stack. 8086 supports two kinds of procedure call. These are FAR and NEAR calls.

The NEAR procedure call is also known as Intrasegment call as the called procedure is in the same segment from which call has been made. Thus, only IP is stored as the return address. The IP can be stored on the stack as:



Please note the growth of stack is towards stack segment base. So stack becomes full on an offset 0000h. Also for push operation we decrement SP by 2 as stack is a word stack (word size in 8086 = 16 bits) while memory is byte organized memory.

FAR procedure call, also known as intersegment call, is a call made to separate code segment. Thus, the control will be transferred outside the current segment. Therefore, both CS and IP need to be stored as the return address. These values on the stack after the calls look like:



When the 8086 executes the FAR call, it first stores the contents of the code segment register followed by the contents of IP on to the stack. A

RET from the NEAR procedure. Pops the two bytes into IP. The RET from the FAR procedure pops four bytes from the stack.

Procedure is defined within the source code by placing a directive of the form:

<Procedure name> PROC <Attribute>

A procedure is terminated using:

<Procedure name> ENDP

The <procedure name> is the identifier used for calling the procedure and the <attribute> is either NEAR or FAR. A procedure can be defined in:

1. The same code segment as the statement that calls it.

2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.

3. A different source module and segment from the calling statement.

In the first case the <attribute> code NEAR should be used as the procedure and code are in the same segment. For the latter two cases the <attribute> must be FAR.

Let us describe an example of procedure call using NEAR procedure, which contains a call to a procedure in the same segment.

PROGRAM 1:

Write a program that collects in data samples from a port at 1 ms interval. The upper 4 bits collected data same as mastered and stored in an array in successive locations.

```
; REGISTERS  :Uses CS, SS, DS, AX, BX, CX, DX, SI, SP

; PROCEDURES  : Uses WAIT

 DATA_SEG      SEGMENT

  PRESSURE  DW  100  DUP(0)  ; Set up array of 100 words

 NBR_OF_SAMPLES  EQU  100

  PRESSURE_PORT EQU  0FFF8h     ; hypothetical input port
```

```
DATA_SEG      ENDS
 STACK_SEG SEGMENT STACK
    DW  40    DUP(0)    ; set stack of 40 words
STACK_TOP   LABEL WORD
STACK_SEG   ENDS
 CODE_SEG SEGMENT
   ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:  MOV   AX, DATA_SEG  ; Initialise data segment register
    MOV  DS, AX
    MOV  AX, STACK_SEG   ; Initialise stack segment register
   MOV  SS, AX
   MOV  SP, OFFSET STACK – TOP    ; initialise stack pointer  top of
                                  ; stack
   LEA  SI, PRESSURE  ; SI points to start of array   PRESSURE
   MOV  BX, NBR_OF_SAMPLES   ; Load BX with number of samples
   MOV  DX, PRESSURE_PORT   ; Point DX at input port
                         ; it can be any A/D converter or   data port.
 READ_NEXT: IN  AX, DX   ; Read data from port
                              ; please note use of IN instruction
   AND  AX, 0FFFH      ; Mask upper 4 bits of AX
   MOV  [SI], AX         ; Store data word in array
   CALL  WAIT           ; call procedures wait for delay
  INC  SI          ; Increment SI by two as dealing with
  INC  SI         ; 16 bit words and not bytes
    DEC  BX     ; Decrement sample counter
```

```
  JNZ  READ_NEXT   ; Repeat till 100  samples are collected
STOP: NOP
WAIT   PROC  NEAR
  MOV  CX, 2000H   ; Load delay value  into CX
HERE:   LOOP  HERE   ; Loop until CX = 0
  RET
WAIT   ENDP
CODE_SEG ENDS
  END
```

Discussion:

Please note that the CALL to the procedure as above does not indicate whether the call is to a NEAR procedure or a FAR procedure. This distinction is made at the time of defining the procedure.

The procedure above can also be made a FAR procedure by changing the definition of the procedure as:

```
WAIT            PROC FAR
 .
 .
WAIT            ENDS
```

The procedure can now be defined in another segment if the need so be, in the same assembly language file.

3.4 Parameter Passing in Procedures

Parameter passing is a very important concept in assembly language. It makes the assembly procedures more general. Parameter can be passed to and from to the main procedures. The parameters can be passed in the following ways to a procedure:

1. Parameters passing through registers
2. Parameters passing through dedicated memory location accessed by name
3. Parameters passing through pointers passed in registers
4. Parameters passing using stack.

Let us discuss a program that uses a procedure for converting a BCD number to binary number.

PROGRAM 2:

Conversion of BCD number to binary using a procedure.

Algorithm for conversion procedure:

Take a packed BCD digit and separate the two digits of BCD.

Multiply the upper digit by 10 (0Ah)

Add the lower digit to the result of multiplication

The implementation of the procedure will be dependent on the parameter-passing scheme. Let us demonstrate this with the help of three programs.

Program 2 (a): Use of registers for parameter passing: This program uses AH register for passing the parameter. We are assuming that data is available in memory location. BCD and the result is stored in BIN

```
;REGISTERS  : Uses CS, DS, SS, SP, AX
;PROCEDURES : BCD-BINARY
 DATA_SEG SEGMENT
   BCD  DB 25h                  ; storage for BCD value
   BIN  DB ?                    ; storage for binary value
DATA_SEG ENDS
STACK_SEG SEGMENT  STACK
      DW  200 DUP(0)            ; stack of 200 words
```

```
  TOP_STACK  LABEL  WORD
STACK_SEG ENDS
    CODE_SEG   SEGMENT
  ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:  MOV  AX, DATA_SEG     ; Initialise data segment
  MOV  DS, AX                    ; Using AX register
  MOV  AX, STACK_SEG      ; Initialise stack
  MOV  SS, AX           ; Segment register. Why  stack?
  MOV  SP, OFFSET TOP_STACK     ; Initialise stack pointer
  MOV  AH, BCD
  CALL  BCD_BINARY   ; Do the conversion
  MOV  BIN, AH        ; Store the result in the  memory
    :
    :
; Remaining program can be put here
;PROCEDURE  : BCD_BINARY - Converts BCD numbers to binary.
;INPUT           : AH with BCD value
;OUTPUT        : AH with binary value
;DESTROYS   : AX
 BCD_BINARY    PROC NEAR
 PUSHF                      ; Save flags
PUSH  BX                  ; and registers used in procedure
PUSH   CX      ; before starting the conversion Do the conversion
MOV  BH, AH   ; Save copy of BCD in BH
AND   BH, 0Fh   ; and mask the higher bits. The lower digit is in BH
```

```
AND  AH, 0F0h   ; mask the lower bits. The higher digit is in AH
                ; but in upper 4 bits.
MOV  CH, 04  ; so move upper BCD digit to lower
ROR  AH, CH  ; four bits in AH
MOV  AL, AH  ; move the digit in AL for multiplication
MOV  BH, 0Ah  ; put 10 in BH
MUL  BH ;Multiply upper BCD digit in AL  by 0Ah in BH, the result is in AL
MOV  AH, AL  ; the maximum/ minimum number would not
             ; exceed 8 bits so move AL to AH
ADD  AH, BH  ; Add lower BCD digit to MUL result
             ; End of conversion, binary result in AH
  POP  CX   ; Restore registers
  POP  BX
  POPF
  RET     ; and return to calling program
BCD_BINARY ENDP
CODE_SEG ENDS
  END  START
```

Discussion:

The above program is not an optimum program, as it does not use registers minimally. By now you should be able to understand this module. The program copies the BCD number from the memory to the AH register. The AH register is used as it is in the procedure. Thus, the contents of AH register are used in calling program as well as procedure; or in other words have been passed from main to procedure. The result of the subroutine is also passed back to AH register as returned value. Thus, the calling program can find the result in AH register.

The advantage of using the registers for passing the parameters is the ease with which they can be handled. The disadvantage, however, is the

limit of parameters that can be passed. For example, one cannot pass an array of 100 elements to a procedure using registers.

Passing Parameters in General Memory the parameters can also be passed in the memory. In such a scheme, the name of the memory location is used as a parameter. The results can also be returned in the same variables. This approach has a severe limitation. It is that you will be forced to use the same memory variable with that procedure. What are the implications of this bound? Well in the example above we will be bound that variable BCD must contain the input. This procedure cannot be used for a value stored in any other location. Thus, it is a very restrictive method of procedural call. Passing Parameters Using Pointers This method overcomes the disadvantage of using variable names directly in the procedure. It uses registers to pass the procedure pointers to the desired data. Let us explain it further with the help of a newer version of the last program.

Program 2 (c) version 2:

```
DATA_SEG SEGMENT
    BCD  DB   25h  ; Storage for BCD test value
  BIN     DB  ?  ; Storage for binary value
DATA_SEG ENDS
 STACK_SEG SEGMENT    STACK
    DW  100 DUP(0)    ; Stack of 100 words
TOP_STACK LABEL    WORD
STACK_SEG  ENDS
 CODE_SEG SEGMENT
  ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:  MOV  AX, DATA_SEG   ; Initialize data
  MOV  DS, AX   ; segment using AX register
  MOV  AX, STACK_SEG   ; initialize stack
  MOV  SS, AX  ; segment. Why stack?
  MOV  SP, OFFSET TOP_STACK  ; initialize stack pointer
```

```
; Put pointer to BCD storage in SI and DI prior to procedure call.
  MOV  SI, OFFSET BCD   ; SI now points to BCD_IN
 MOV  DI, OFFSET BIN  ; DI points BIN_VAL  (returned value)
  CALL  BCD_BINARY   ; Call the conversion  procedure
  NOP     ; Continue with program  here
 ; PROCEDURE     : BCD_BINARY Converts BCD numbers to binary.
; INPUT     : SI points to location in memory of data
; OUTPUT     : DI points to location in memory for result
; DESTROYS    : Nothing
 BCD_BINARY PROC  NEAR
  PUSHF     ; Save flag register
  PUSH  AX  ; and AX registers
  PUSH  BX  ; BX
  PUSH  CX  ; and CX
  MOV  AL, [SI]   ; Get BCD value from memory for conversion
  MOV  BL, AL   ; copy it in BL also
  AND   BL, 0Fh   ; and mask to get lower 4 digits
  AND  AL, 0F0h  ; Separate upper 4 bits in AL
 MOV  CL, 04   ; initialize counter CL so that upper digit
 ROR  AL, CL  ; in AL can be brought to lower 4 bit  positions in AL
  MOV  BH, 0Ah   ; Load 10 in BH
  MUL  BH  ; Multiply upper digit in AL by 10  The result is stored in AL
 ADD  AL, BL   ; Add lower BCD digit in BL to result of  multiplication
; End of conversion, now restore the original values prior to call. All calls
will be in
```

```
; reverse order to save above. The result is in AL register.
  MOV  [DI], AL   ; Store binary value to memory
  POP  CX  ; Restore flags and
  POP  BX  ; registers
  POP  AX
  POPF
  RET
BCD_BINARY ENDP
CODE_SEG ENDS
  END  START
```

Discussion:

In the program above, SI points to the BCD and the DI points to the BIN. The instruction MOV AL,[SI] copies the byte pointed by SI to the AL register. Likewise, MOV [DI], AL transfers the result back to memory location pointed by DI. This scheme allows you to pass the procedure pointers to data anywhere in memory. You can pass pointer to individual data element or a group of data elements like arrays and strings. This approach is used for parameters passing to BIOS procedures.

Passing Parameters through Stack The best technique for parameter passing is through stack. It is also a standard technique for passing parameters when the assembly language is interfaced with any high level language. Parameters are pushed on the stack and are referenced using BP register in the called procedure. One important issue for parameter passing through stack is to keep track of the stack overflow and underflow to keep a check on errors.

Let us revisit the same example, but using stack for parameter passing.

PROGRAM 2: Version 3

```
DATA_SEG SEGMENT
   BCD  DB  25h  ; Storage for BCD test value
   BIN  DB  ?  ; Storage for binary value
DATA_SEG     ENDS
```

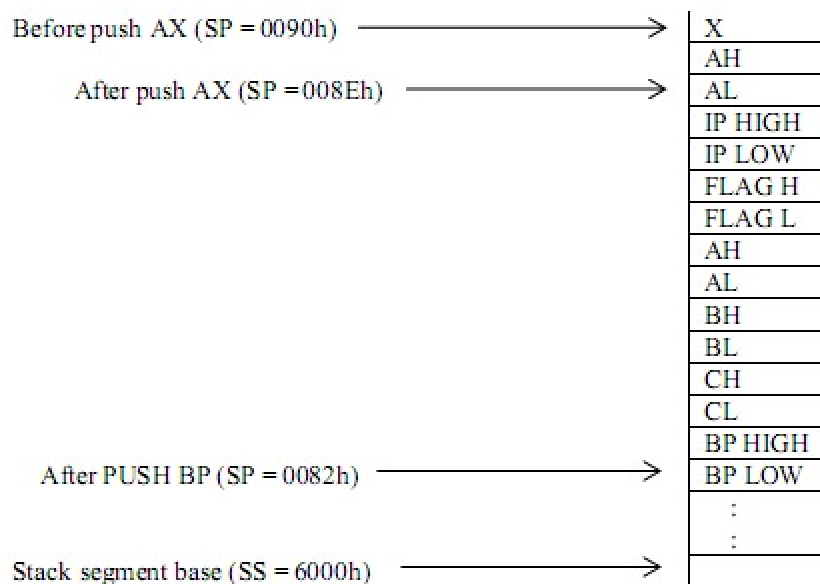
```
 STACK_SEG   SEGMENT     STACK
  DW  100 DUP(0)  ; Stack of 100 words
  TOP_STACK   LABEL    WORD
STACK_SEG   ENDS

CODE_SEG SEGMENT
   ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:   MOV  AX, DATA   ; Initialise data segment
   MOV  DS, AX  ; using AX register
   MOV  AX, STACK-SEG .  ; initialise stack segment
   MOV  SS, AX       ; using AX register
   MOV  SP, OFFSET TOP_STACK  ; initialise stack pointer
   MOV  AL, BCD  ; Move BCD value into AL
   PUSH  AX   ; and push it onto word stack
   CALL  BCD_BINARY  ; Do the conversion
   POP  AX   ; Get the binary value
   MOV  BIN, AL   ; and save it
   NOP     ; Continue with program
; PROCEDURE  : BCD_BINARY Converts BCD numbers to binary.
; INPUT   : None - BCD value assumed to be on stack before call
; OUTPUT   : None - Binary value on top of stack after return
; DESTROYS  : Nothing
 BCD_BINARY PROC  NEAR
   PUSHF     ; Save flags
   PUSH  AX  ; and registers : AX
```

```
PUSH  BX  ; BX
PUSH  CX  ; CX
PUSH  BP  ; BP. Why BP?
MOV  BP, SP  ; Make a copy of the  stack pointer in BP
MOV   AX, [BP+ 12]  ; Get BCD number from  stack. But why it is on
; BP+12 location? Please note 5 PUSH statements + 1 call which is intra-segment (so
; just IP is stored) so total 6 words are pushed after AX has been pushed and since it is
; a word stack so the BCD value is stored on 6 × 2 = 12 locations under stack. Hence
; [BP + 12] (refer to the figure given on next page).
MOV  BL, AL  ; Save copy of BCD in BL
AND  BL, 0Fh  ; mask lower 4 bits
AND  AL, F0H  ; Separate upper 4 bits
MOV  CL, 04  ; Move upper BCD digit to low
ROR   AL, CL  ; position BCD digit for multiply location
MOV   BH, 0Ah   ; Load 10 in BH
MUL  BH  ; Multiply upper BCD digit in AL by 10  the result is in AL
ADD   AL, BL  ; Add lower BCD digit to result.
MOV  [BP + 12], AX   ; Put binary result on stack
; Restore flags and registers
POP  BP
POP  CX
POP  BX
POP  AX
```

```
  POPF
  RET
BCD_BINARY    ENDP
CODE_SEG   ENDS
END   START
```

Discussion:

The parameter is pushed on the stack before the procedure call. The procedure call causes the current instruction pointer to be pushed on to the stack. In the procedure flags, AX, BX, CX and BP registers are also pushed in that order. Thus, the stack looks to be:



The instruction MOV BP, SP transfers the contents of the SP to the BP register. Now BP is used to access any location in the stack, by adding appropriate offset to it. For example, MOV AX, [BP + 12] instruction transfers the word beginning at the 12th byte from the top of the stack to AX register. It does not change the contents of the BP register or the top of the stack. It copies the pushed value of AH and AL at offset 008Eh into the AX register. This instruction is not equivalent to POP instruction.

Stacks are useful for writing procedures for multi-user system programs or recursive procedures. It is a good practice to make a stack diagram as above while using procedure call through stacks. This helps in reducing errors in programming.

3.5 External Procedures

These procedures are written and assembled in separate assembly modules, and later are linked together with the main program to form a bigger module. Since the addresses of the variables are defined in another module, we need segment combination and global identifier directives. Let us discuss them briefly.

Segment Combinations

In 8086 assembler provides a means for combining the segments declared in different modules. Some typical combine types are:

1. PUBLIC: This combine directive combines all the segments having the same names and class (in different modules) as a single combined segment.

2. COMMON: If the segments in different object modules have the same name and the COMMON combine type then they have the same beginning address. During execution these segments overlay each other.

3. STACK: If the segments in different object modules have the same name and the combine type is STACK, then they become one segment, with the length the sum of the lengths of individual segments.

These details will be more clear after you go through program 4 and further readings.

Identifiers

a) Access to External Identifiers: An external identifier is one that is referred in one module but defined in another. You can declare an identifier to be external by including it on as EXTRN in the modules in which it is to be referred. This tells the assembler to leave the address of the variable unresolved. The linker looks for the address of this variable in the module where it is defined to be PUBLIC.

b) Public Identifiers: A public identifier is one that is defined within one module of a program but potentially accessible by all of the other modules in that program. You can declare an identifier to be public by including it on a PUBLIC directive in the module in which it is defined.

Let us explain all the above with the help of the following example:

PROGRAM 3:

Write a procedure that divides a 32-bit number by a 16-bit number. The procedure should be general, that is, it is defined in one module, and can be called from another assembly module.

```
 ; REGISTERS   :Uses CS, DS, SS, AX, SP, BX, CX
; PROCEDURES  : Far Procedure SMART_DIV
DATA_SEG SEGMENT   WORD  PUBLIC
   DIVIDEND  DW  2345h, 89AB  ; Dividend =  89AB2345H
   DIVISOR  DW  5678H  ; 16-bit divisor
 MESSAGE  DB  'INVALID DIVIDE', '$'
DATA_SEG ENDS
 MORE_DATA   SEGMENT    WORD
QUOTIENT   DW   2  DUP(0)
REMAINDER   DW     0
MORE_DATA ENDS
 STACK_SEG   SEGMENT   STACK
   DW   100 DUP(0)  ; Stack of 100 words
  TOP – STACK   LABEL  WORD  ; top of stack pointer
STACK_SEG   ENDS
 PUBLIC      DIVISOR
 PROCEDURES  SEGMENT   PUBLIC  ; SMART_DIVis declared as an
  EXTRN SMART_DIV: FAR  ; external label in procedure
  ; segment of  type FAR
PROCEDURES  ENDS
```

; declare the code segment as PUBLIC so that it can be merged with other PUBLIC segments

```
CODE_SEG SEGMENT  WORD   PUBLIC
  ASSUME CS:CODE, DS:DATA_SEG, SS:STACK SEG
START:  MOV  AX, DATA_SEG  ; Initialize data segment
  MOV  DS, AX   ; using AX register
  MOV  AX, STACK_SEG   ; Initialize stack segment
  MOV  SS, AX  ; using AX register
  MOV  SP, OFFSET TOP_STACK  ; Initialize stack pointer
  MOV  AX, DIVIDEND   ; Load low word of  dividend
  MOV   DX DIVIDEND + 2  ; Load high word of dividend
  MOV  CX, DIVISOR   ; Load divisor
  CALL SMART_DIV
```

; This procedure returns Quotient in the DX:AX pair and Remainder in CX register. Carry bit is set if result is invalid.

```
  JNC  SAVE_ALL  ; IF carry = 0, result valid
  JMP  STOP    ; ELSE carry set, don't save result
  ASSUME DS:MORE_DATA   ; Change data segment
SAVE_ALL:  PUSH DS   ; Save old DS
  MOV  BX, MORE_DATA   ; Load new data segment
  MOV  DS, BX    ; register
  MOV  QUOTIENT, AX   ; Store low word of quotient
  MOV  QUOTIENT + 2, DX   ; Store high word of quotient
  MOV  REMAINDER, CX   ; Store remainder
  ASSUME DS:DATA_SEG
```

```
  POP   DS  ; Restore initial DS
 JMP  ENDING
STOP:  MOV  DL, OFFSET MESSAGE
  MOV  AX, AH 09H
 INT   21H
ENDING:   NOP
CODE_SEG ENDS
  END  START
```

Discussion:

The linker appends all the segments having the same name and PUBLIC directive with segment name into one segment. Their contents are pulled together into consecutive memory locations.

The next statement to be noted is PUBLIC DIVISOR. It tells the assembler and the linker that this variable can be legally accessed by other assembly modules. On the other hand EXTRN SMART_DIV:FAR tells the assembler that this module will access a label or a procedure of type FAR in some assembly module. Please also note that the EXTRN definition is enclosed within the PROCEDURES SEGMENT PUBLIC and PROCEDURES ENDS, to tell the assembler and linker that the procedure SMART_DIV is located within the segment PROCEDURES and all such PROCEDURES segments need to be combined in one. Let us now define the

PROCEDURE module:

```
; PROGRAM MODULE  PROCEDURES
 ; INPUT  : Dividend - low word in AX, high word in DX, Divisor in CX
; OUTPUT  : Quotient  - low word in AX, high word in DX. Remainder in CX
Carry      - carry flag is set if try to divide by zero
; DESTROYS  : AX, BX, CX, DX, BP, FLAGS
DATA_SEG  SEGMENT   PUBLIC      ; This block tells the assembler that
EXTRN DIVISOR:WORD   ; the divisor is a word variable and is
DATA_SEG   ENDS    ; external to this procedure. It would be
```

```
    ; found in segment named DATA_SEG
PUBLIC  SMART_DIV    ; SMART_DIV is available to
     ; other modules. It is now being defined
    ; in PROCEDURES SEGMENT.
PROCEDURES   SEGMENT  PUBLIC
SMART_DIV PROC  FAR
 ASSUME CS:PROCEDURES, DS:DATA_SEG
  CMP  DIVISOR, 0  ; This is just to demonstrate the use of
    ; external variable, otherwise  we can
    ; check it through CX register which  contains the divisor.
  JE  ERROR_EXIT   ; IF divisor = 0, exit procedure
  MOV  BX, AX   ; Save low order of dividend
  MOV  AX, DX  ; Position high word for Ist divide
  MOV  DX, 0000h  ; Zero DX
  DIV  CX  ; DX:AX/CX, quotient in AX, remainder in DX
  MOV  BP, AX   ; transfer high order of final result to BP
  MOV  AX, BX  ; Get back low order of dividend. Note
    ; DX contains remainder so DX : AX is  the actual number
  DIV   CX   ; DX:AX/CX, quotient in AX, 2nd remainder that is final
;remainder  in DX
  MOV  CX, DX  ; Pass final remainder in CX
  MOV  DX, BP  ; Pass high order of quotient in DX
    ; AX contains lower word of quotient
  CLC     ; Clear carry to indicate valid result
  JMP  EXIT  ; Finished
```

```
ERROR_EXIT: STC   ; Set carry to indicate divide by zero
EXIT:   RET
SMART_DIV   ENDP
PROCEDURES   ENDS
        END
```

Discussion:

The procedure accesses the data item named DIVISOR, which is defined in the main, therefore the statement EXTRN DIVISOR:WORD is necessary for informing assembler that this data name is found in some other segment. The data type is defined to be of word type. Please not that the DIVISOR is enclosed in the same segment name as that of main that is DATA_SEG and the procedure is in a PUBLIC segment.

4. Summary

In this unit, we studied some programming techniques, starting from arrays, to interrupts. Arrays can be of byte type or word type, but the addressing of the arrays is always done with respect to bytes. For a word array, the address will be incremented by two for the next access.

As the programs become larger and larger, it becomes necessary to divide them into smaller modules called procedures. The procedures can be NEAR or FAR depending upon where they are being defined and from where they are being called. The parameters to the procedures can be passed through registers, or through memory or stack. Passing parameters into registers is easier, but limits the total number of variables that can be passed. In memory locations it is straight forward, but limits the use of the procedure. Passing parameters through stack is most complex out of all, but is a standard way to do it. Even when the assembly language programs are interfaced to high level languages, the parameters are passed on stack.

Interrupt Service Routines are used to service the interrupts that could have arisen because of some exceptional condition. The interrupt service routines can be modified- by rewriting them, and overwriting their entry in the interrupt vector table.

This completes the discussion on microprocessors and assembly language programming. The above programming was done for 8086 microprocessor, but can be tried on 80286 or 80386 processors also, with some modification in the assembler directives. The assembler used here is MASM, Microsoft assembler. The assembly language instructions remain the same for all assemblers, though the directives vary from one assembler to another. For further details on the assembler, you can refer to their respective manuals. You must refer to further readings for topics such as Interrupts, device drivers, procedures etc.

5. Test yourself

1. Write an ALP of 8086 to take N numbers.

2. Write an ALP of 8086 to take N numbers as input. And Find maximum.

3. Write an ALP of 8086 to take N numbers as input. And Find average.

4. Write an ALP of 8086 to take a string of as input and find the length.

5. Write an ALP of 8086 to take a string of as input and find it is Palindrome or not.

6. We will give you an algorithm using XLAT instruction. Please code and run the program yourself.

   - Take a sentence in upper case for example 'TO BE CONVERTED TO LOWER CASE' create a table for lower case elements.
   - Check that the present ASCII character is an alphabet in upper case. It implies that ASCII character should be greater than 40h and less than 58h.
   - If it is upper case then subtract 41h from the ASCII value of the character. Put the resultant in AL register.
   - Set BX register to the offset of lower case table.
   - Use XLAT instruction to get the required lower case value.
   - Store the results in another string.

UNIT – III

1. Interrupt Processing

Structure

1.1 INTRODUCTIQN
1.2 Hardware and software Interrupts
1.3 The interrupt vector table
1.4 The interrupt processing sequence
    1-4-1Get Vector Number
    1-4-2 Save Processor Information
    1-4-3 Fetch New Instruction Pointer
1.5 Multiple interrupts
1.6 Special interrupts
    1-6-1 Divide-Error
    1-6-2 Single –step
    1-6-3 NMI
    1-6-4 Breakpoint
    1-6-5 Overflow
    1-6-6 INTR
1.7 Interrupt service routines
1-8 Interrupt procedure of 8086
1-9 8259 programmable interrupt controller
    1-9-1 Command words of 8259

OBJECTIVES

In this unit you will learn about:

• The differences between hardware and software interrupts
• The differences between maskable and non-maskable interrupts
• Interrupt processing procedures
• The vector address table
• Multiple interrupts and interrupts priorities
• Special function interrupts
• The general requirements of all interrupt handlers

1.1 INTRODUCTIQN

The 8086 provides a very flexible method for recovering from what are known as *catastrophic* system faults. Through the same mechanism,

external and internal interrupts may be handled and other events not normally associated with program execution may be taken care of. The method that does all of this for us is the 8086 interrupt handler. In this chapter we will see that there are many kinds of interrupts. Some of these deal with issues that have always plagued programmers (such as the divide-by-zero operation), while still others may be defined by the programmer. The emphasis in this chapter is on the definition of the numerous interrupts available.

1.2 Hardware and software Interrupts

An interrupt is an event that causes the processor to stop its current program execution and perform a specific task to service the interrupt. We are all interrupted many times during the day. A ringing telephone or doorbell, a knock on the door, or a question from a friend all indicate the need to communicate with you. The situation is much the same with the microprocessor. The interrupt is used to get the processor's attention. Interrupts may be used to inform the processor in an alarm system that a fire has started or a window has been opened. In a personal computer, interrupts are used to keep accurate time, read the keyboard. Operate the disk drives, and access the power of the disk operating system. Two kinds of interrupts are available: hardware and software interrupts. Hardware interrupts are generated by changing the logic levels on either of the processor's hardware interrupt inputs. These inputs are NMI (non-maskable interrupt) and INTR (Interrupt request). INTR can be enabled and disabled by the state of the interrupt-enable flag (IF). The CLI (clear interrupt-enable flag) instruction clears W, which disables INTR. The STI (set interrupt-enable flag) instruction sets IF, allowing INTR. to respond to requests. This means that INTR can be *masked* (disabled). NMI gets its name from the fact that its operation cannot be disabled. NMI *always* causes an interrupt sequence when it is activated. A rising edge is needed on NMI to trigger the interrupt mechanism, and that ENTR is level sensitive, requiring a high logic level to interrupt the processor.

Software interrupts are generated directly by an executing program. These types of interrupts are also called *exceptions* some instructions like INT and INTO initiate interrupt processing when they are executed. Other instructions are capable of generating an interrupt when a certain condition is met. DIV and DIV. for example, cause a type-O interrupt (divide error) whenever division by 0 is attempted.

What happens when a software and hardware interrupt occur at the same time? The processor has a technique for handling this situation; it requires that the interrupts be *prioritized.* Table shows the interrupt priority scheme used by the 8086.

Interrupts with the highest priority are divide-error, INT, and INTO. NMI and INTR have lower priorities, with single-step having the lowest. If both hardware interrupts are activated simultaneously. NMJ will be serviced first, with INTR *pending* until it gets its chance to be recognized by the processor. If a divide-error and NMI occur simultaneously, divide-error will be recognized first, followed by NMI.

Interrupt	*Priority*
Divide-error	**Highest**
INT. INTO	
NM I	
INTR	
Single-step	**Lowest**

1.3 The interrupt vector table

All types of interrupts, whether hardware or software generated, point to a single entry in the processor's interrupt vector table. This table is a collection of 4-byte addresses (two for CS and two for IP) that indicate where the processor should jump to execute the associated interrupt service routine, the code designed to handle the specific interrupt. Because 256 interrupts are available, the interrupt vector table is 1,024 bytes long. The 1KB block of memory reserved for the table is located in the address range 00000 to 003FFH. Some earlier processors automatically loaded their first instruction from address 0000 after a reset. The 80xS6 fetches its first address from location FFPF0H. This indicates that we are able to begin program execution without first having to place values into the interrupt vector table. If we plan on using any interrupts, it will be necessary to initialize the required vectors within the table. This can be done easily with a few MOV instructions.

Figure 1.1 shows the organization of the interrupt vector table. Each 4-byte entry consists of a 2-byte IP register value followed by a 2-byte CS register value. This indicates that interrupt service routines are considered far routines. Notice that some of the vectors are predefined. Vector 0 (the same as type-0) has been chosen to handle division-by-zero errors. Vector 1 (type-i) helps to implement single-step operation. Vector 2 is used when NMI is activated. Vector 3 (breakpoint) is normally used when troubleshooting a new program Vector 4 is associated with

the INTO instruction Vectors is used when the BOUND instruction reports n out-of-range index. Vectors 6 through 18 perform various housekeeping duties. Some of these vectors (10, 11, 14) are operational only in protected or virtual- 8086 mode. Table shows the associated vector assignments. Vectors 19 through 31 are reserved by Intel for use in their products. This does not mean that these interrupt vectors are unavailable to us, but that we should refrain from using them In an Intel machine unless we know how they have been assigned. Vectors 32 through 255 are unassigned and free for us to use. To initialize an interrupt vector, we must write the 4 bytes of the interrupt service routine address into the table locations reserved for the interrupt. A short example shows one way this can be done.

• EXAMPLE 1.1

The interrupt service routine for a type-40 interrupt is located at address 28000H how is the interrupt vector table set up to handle this interrupt? Solution: An easy way to determine the address within the interrupt vector table that is used by an interrupt is to multiply the interrupt number by 4. Multiplying 40 by 4 and converting into hexadecimal gives 000A0H as the starting address of the vector for INT *40.* The interrupt service routine address 28000H can be generated by many different combinations of CS and IP values. If CS is loaded with 28000H and U' with 0000, we get the correct address of 28000W Thus, it is necessary to write these two address values into memory starting at 000A0H. The short section of code shown here is one way to do this:

TABLE: Interrupt/exception vectors

Vector	***Description***
0	Divide error
I	Debugger call (single-step)
2	NMI
3	Breakpoint
4	INTO
5	SOUND range exceeded
6	invalid opcode
7	Device not available

8	Double fault
9	Reserved
10	Invalid task state segment
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	Reserved
16	Floating-point error
17	Alignment check
18	Machine check
19—31	Reserved
32—255	Maskable Interrupts (non-reserved)

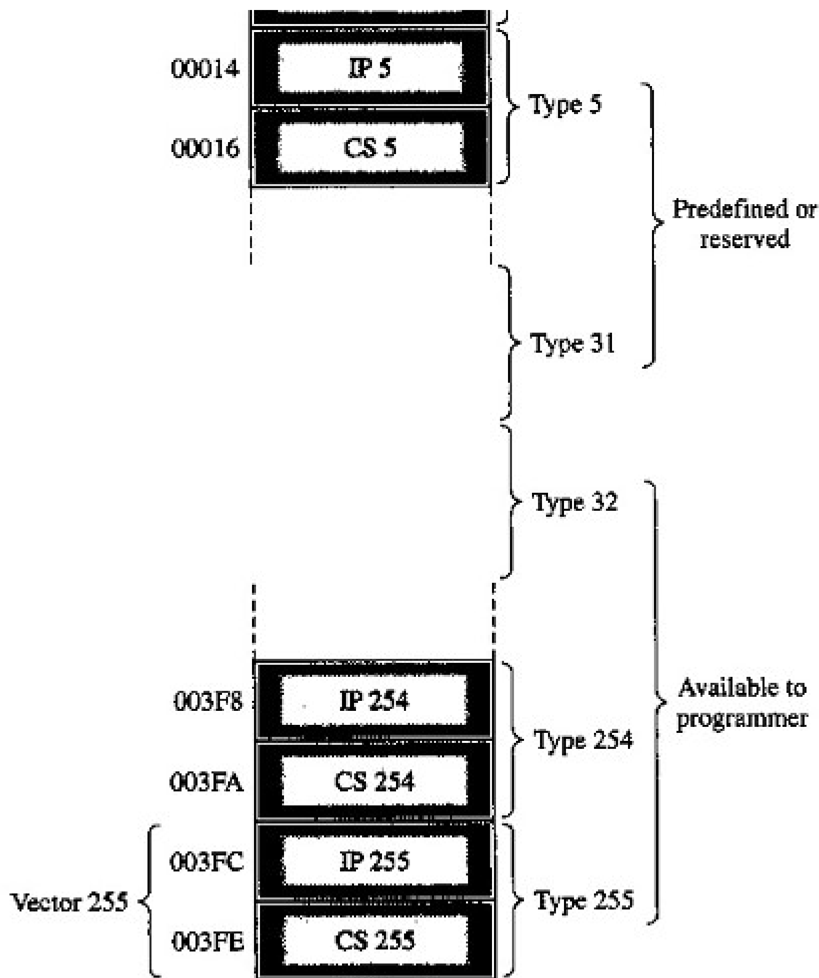


FIGURE 1.1 Interrupt vector table

```
PUSH   DS                           ;save current DS address
MOV    AX,0                         ;set new DS address at 0000
MOV    DS,AX
MOV    DI,00A0H                     ;offset for INT 40 vector
MOV    WORD PTR [DI],0              ;store IP address
MOV    WORD PTR [DI + 2],2800H      ;store CS address
POP    DS                           ;get old DS address back
```

The PUSH DS instruction is used to save the current value of the DS register on the stack, Because we need to access memory in the 00000 to 003FFFI range, it is convenient to load 1)8 with 0000 and use DI as the offset into the vector table. Notice that the first word written is 0000, which

goes into locations 000A0H and 000A1H. Then the CS value 2800H is written into locations 000A2H and 000A3H. Figure 1.2 provides a snapshot of memory after these instructions execute. Now, when INT 40 executes, it will cause a jump to the interrupt service routine located at address 28000H. The POP DS instruction restores the old value of the DS register.

Figure 1.2 also shows the contents of memory for the vector associated with INT 41. What is the address of the TSR? As usual, the two words have been byte-swapped. The word at address 000A4H is IFOOH. The word at address 000A6H is 3400H. The effective address created by the addition of these two words is 35F00H, which is the address of the ISR for INT 41.

The machine code for INT 40 is CD 28 (note that 28H equals 40 decimal). The machine code for INT 41 is CD 29. All INT instructions begin with CI) as their first byte and *have* the interrupt number as the second byte. The only exception to this rule is INT 3, breakpoint, which has only the byte CC as its opcode.

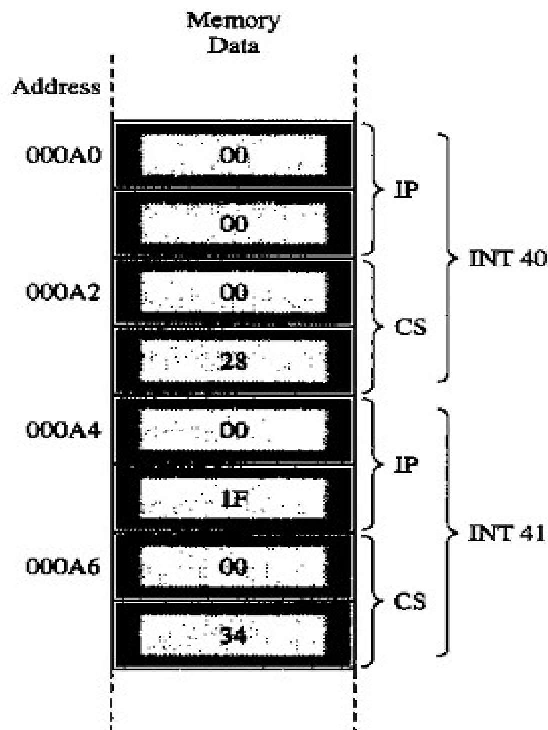


FIGURE 1.2 ISR address for INT 40H and INT 41 H

1.4 The interrupt processing sequence

Interrupt processing coverage of the INT and INTO instructions. These software interrupts initiate a sequence of steps in which the flags and return address are saved prior to loading CS and IP with the ISR address. The same process is followed for hardware interrupt NMI, which automatically generates a type-2 interrupt. The sequence initiated by INTR (when interrupts are enabled) is slightly *different,* because the processor must first read the interrupt number from the data bus. When interrupts are enabled, INTR causes the processor to perform two interrupt acknowledge cycles. In the 8086, external devices recognize these cycles by examining the state of the $\overline{INTA}$ output, which goes low during each cycle, the first low-going pulse on $\overline{INTA}$ is used to indicate to other devices on the system bus that the processor is beginning an interrupt acknowledge cycle. In minimum mode this indicates that the processor will riot acknowledge a hold request until the interrupt acknowledge cycle completes. In maximum mode the processor activates its LOCK output to prevent system bus takeovers during the interrupt acknowledge cycle.

The second low pulse on $\overline{INTA}$ indicates that the interrupt number should be placed onto the lower byte of the processor's data bus. A special peripheral designed to respond to the 8086's interrupt acknowledge cycle is the 8259A programmable interrupt controller.

Once the interrupt number is read from the data bus, the processor performs all of the steps that we are familiar with. Let us review the overall process once more.

1-4-1Get Vector Number

The processor obtains the interrupt number in one of three ways. First, the interrupt type number may be specified directly using one of the INT instructions. Second, the processor may automatically generate the interrupt type number, as it does for INTO, NMI, and divide-error. Third it may have to read the interrupt type number from the data bus (after receiving INTR).

Once the interrupt number is obtained, it is used to form the location within the interrupt vector table that contains the requested ISR address.

1-4-2 Save Processor Information

Once the interrupt vector is known, the processor pushes the flag register onto the stack. This is done to preserve some of the processor's internal state at the time of the interrupt (a very necessary step if we are to resume normal execution later). Once the flags are pushed, the

processor cigars the interrupt enable and trace flags to disable INTR while interrupt processing is taking place. Next, the Wand CS values at the time of the interrupt are pushed onto the stack. Figure 1.3 shows how the stack is used when an interrupt occurs. The contents of the flag register at the time of the interrupt were 00C2H, the address of the instruction that was about to be fetched when the interrupt occurred was 1109:3C00 (in **CS: IP** format).

1-4-3 Fetch New Instruction Pointer

Once the return address has been pushed, the processor can fetch the new values of IP and CS out of the interrupt vector table and begin execution of the interrupt service routine. The address generated by the interrupt number is used to read the two ISR address words out of the table.

One word of caution: Because the stack contains the information needed to return to the interrupt point, we must be careful not to change the contents of stack memory or alter the stack pointer in any way that would prevent the correct information from being popped off. The processor will not remember anything about the interrupt and relies only on the data popped off the stack for a proper return.

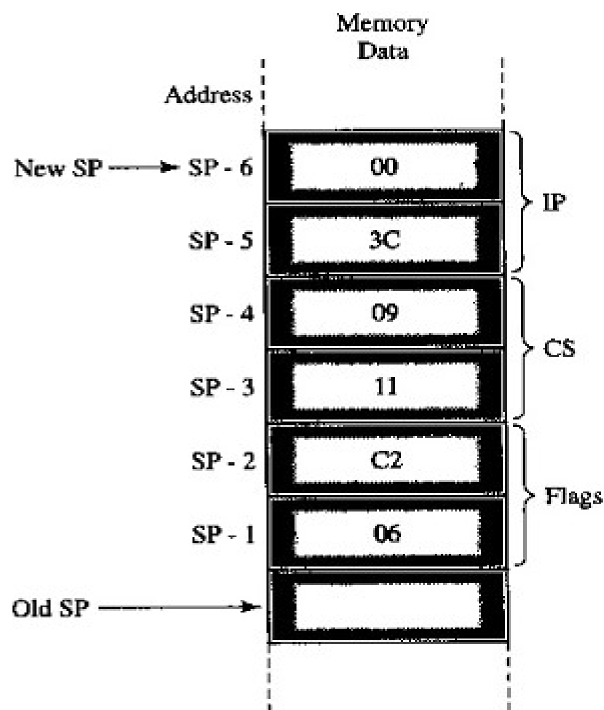


FIGURE 1.3 Stack contents during an interrupt

1.5 Multiple interrupts

In the course of program execution, chances are good that eventually two interrupts might request the processor's attention at the same time, For example, just as division-by-zero is attempted in an executing program, NMI is also activated. The processor needs to "break the tie" when this happens and recognize one of the two interrupts first. When we examined priority table, we saw that divide-error has a higher priority than NMI. So, in our current example, divide-error will be recognized first, and the following sequence of steps will occur:

1. Divide-error is recognized.
2. The flags are pushed.
3 The return address (CS and IP) is pushed.
4. The interrupt-enable and trace flags are cleared.
5. NMI is recognized.
6. The new flags are pushed.
7. The new return address is pushed.
8. The interrupt-enable and trace flags are cleared.
9. The NMI ISR is executed.
10. The second return address is popped.
11. The second set of flags is popped.
12. The divide-error ISR is executed.
13. The first return address is popped.
14. The first set of flags is popped.
15. Execution resumes at the instruction following the one that initiated the divide-error.

It is easy to see that the stack plays an important role during this process.

A more common occurrence of a multiple interrupt is seen when the processor's trace flag is set. The trace flag, when set, puts the processor into single-step mode, where a type-1 interrupt is generated after completion of every instruction. If the current instruction is INT or INTO. You can see that two interrupts will need servicing: the INT or INTO interrupt and the single-step interrupt. Single-step has the lowest priority of all interrupts and thus gets recognized last. We will see how single-stepping with the trace flag can be a useful tool when debugging a program.

1.6 Special interrupts

We will now examine the specific operation of a few selected interrupts. Some may be very useful to implement while others may never be needed in a program for proper operation. Even so, you are better off knowing how each one works and when to use it.

1-6-1 Divide-Error

Figure 1.4 shows the contents of four memory locations that contain the code for these two instructions:

```
B3  00    MOV  BL,0
F6  F3     DIV BL
```

The first instruction is located at address 05100H. The second instruction is located at address 05102H, Because DIV BL is a 2-byte instruction, the instruction must he located at address 05104W this address becomes the return when DIV BL generates a divide-error interrupt.

The interrupt service routine for divide-error can do anything the user wishes to recover from the error. One programmer may wish simply to load the accumulator with 0 or some other number, while another may want to display an error message on the user's display screen.

Because divide error is a type-O interrupt its address vector is stored in memory locations 00000 through 00003.

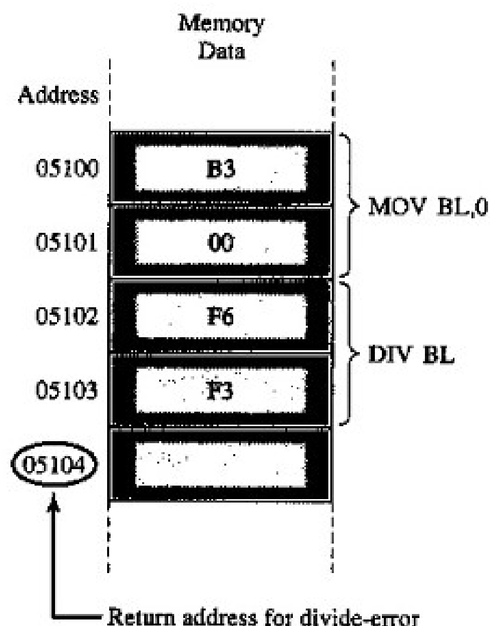


FIGURE 1.4 Instruction sequences causing a divide-error interrupt

1-6-2 Single –step

This interrupt relies on the setting of the trace flag in the flag register. When the trace flag is set, the processor will generate a type-i interrupt after each instruction executes. Remember from the interrupt processing sequence that after the flags are pushed the processor clears the trace flag, disabling single-stepping while it executes the trace ISR, An extremely useful debugging tool can be written and used within the trace ISR. This single- step debugger may be programmed to display the contents of each processor register, the state of the flags, and other useful information after execution of each instruction in a user program. Because the trace flag is cleared before the ISR is called, we need not worry about single-stepping through the trace ISR.

There are no instructions available that directly affect the state of the trace flag. There are other techniques that can be used to do this. For instance, a copy of the flags can be leaded into *AX* by first pushing a copy of the flag register onto the stack and then popping them into AX. Then an OR instruction can be used to set the trace flag. Once this is done, the flags are restored by pushing AX back onto the stack and then popping the flags. The instructions needed to accomplish this are:

PUSH F

POP AX

OR AX, 100H

PUSH AX

POPF

Once this is done, the processor will enter and remain in single-step mode until the trace flag is cleared. This can be done with the same set of instructions, replacing the OR with AND AX, 0FEFFH.

The ISR address vector for single-step must be stored in memory locations 00004 through 00007.

• EXAMPLE 1.2

Assume that the trace flag is set and that the trace [SR displays the contents of AX on the screen after each instruction executes. What do we expect to see when this group of instructions executes?

MOV AX, 1234H

INC AL

DEC AH

NOT AX

Solution: Because the trace flag is set, a single-step interrupt will be generated after each of the four instructions. When the first instruction completes, the trace ISR will display AX= 1234. The second instruction will increase AL to 35, causing the ISR to display AX= 1235 next. The third instruction will decrease AH to 11. The trace ISR will then display AX= 1135. Finally, after all bits in AX are inverted, the trace ISR will display AX=EECA. We will see that the personal computer has built-in routines capable of displaying messages and data on the screen, so writing a trace ISR is not as complicated a task as it appears.

1-6-3 NMI

Because NMI (non maskable interrupt) can never be ignored by the processor, it finds useful application in events that the computer absolutely must respond to. One such event is the disastrous power-fail. The processor unfortunately forgets the contents of its registers and flags when power is turned off and thus has no chance of getting back to the correct place in a program if its power is interrupted. One way to prevent this from happening and provide a way for the processor to resume execution is to use NMI to interrupt the processor at the beginning of a power-fail. Because the computer's power supply will continue to supply a stable voltage for a few milliseconds after it loses AC, the processor has plenty of time to execute the necessary instructions. Suppose that a certain system contains a small amount of non-volatile memory. This type of memory retains its data after it loses power and acts like RAM when power is applied. So, in the event of a power-fail, the NMI ISR should store the contents of each processor register in the NVM. These values can then be reloaded when power comes back up. In this fashion we can recover from a power-fail without loss of intelligence.

The ISR address vector for NMI is stored in memory locations 00002 through 0000BH.

1-6-4 Breakpoint

This interrupt is a type-3 interrupt but is coded as a single byte for reasons of efficiency. Breakpoint aids in debugging in the following way: a program being debugged will have the first byte of one of its instructions replaced by the code for breakpoint (CC). When the processor gets to this instruction, it will generate a type-3 interrupt. The [SR associated with breakpoint is similar to the trace ISR and should be

capable of displaying the processor register contents and also the address at which the breakpoint occurred. Before the ISR exits, it will replace the breakpoint byte with the original first byte of the instruction. Figure 1.5 shows how the breakpoint routine makes a copy of the first byte in the NOT AL instruction stored at location 06200H. The first byte of the instruction (F6H) is copied into a temporary location and then replaced by the breakpoint instruction code CC. Some people like to refer to this as *setting the breakpoint.* Once the breakpoint is set, a fetch from address 06200H will initiate a breakpoint interrupt.

Clearing the breakpoint is accomplished by copying the instruction byte from the temporary location back into its original location. The ISR address vector for breakpoint is stored in memory locations 0000CH through 0000FH.

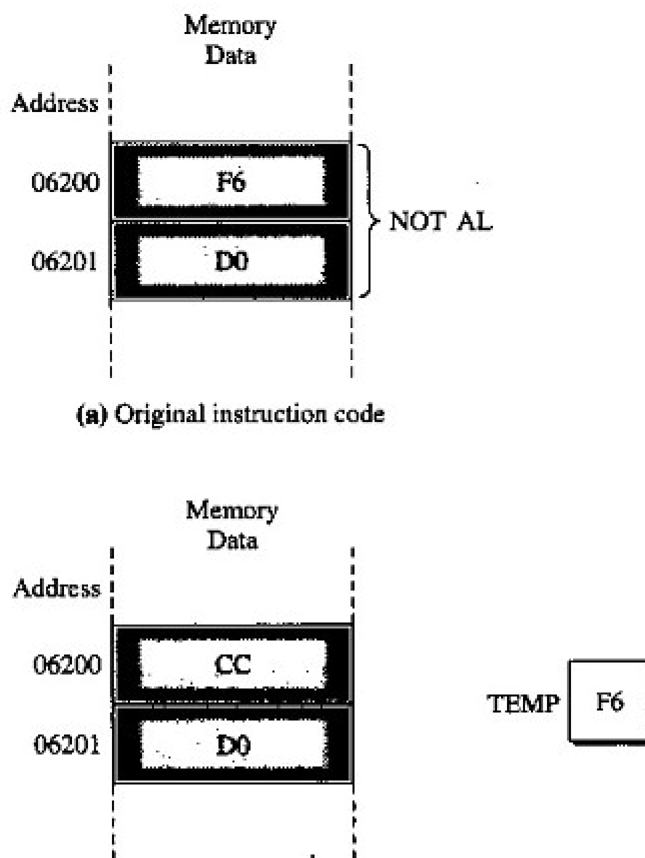


FIGURE 1.5 Setting a breakpoint

EXAMPLE 1.3

A programmer wishes to find out if a conditional jump takes place. Where should the programmer place the breakpoint instruction? The code being tested looks like this:

```
      CMP AL,0
      JNZ XYZ
      NOT AL
XYZ:  INC AL
```

Solution: The programmer has two choices for placement of the breakpoint instruction. It could be placed in the location occupied by NOT AL, which would cause a breakpoint when the JNZ does *not* jump to XYZ. It could also be placed in the location occupied by INC AL, activating when the JNZ *does* take place. Either way, the programmer will know the results of the CMP and .JNZ instructions (by the presence or absence of a breakpoint).

1-6-5 Overflow

This type-4 interrupt is initiated only when the INTO instruction is executed with the overflow flag set. Its applications, like divide-error, tend to be of a corrective nature. You may think of overflow as the watchdog for multi-bit addition and subtraction operations, much like divide-error watches out for division-by-zero. If the overflow flag is cleared, INTO will not generate an interrupt (essentially operating as a NOP in this case).

The ISR address vector for overflow is stored in memory locations 00010H through 0001 3H.

EXAMPLE 1.4

Will the following sequence of code generate an overflow interrupt?

```
MOV AL,70H
MOV BL,60H
ADD AL,BL
```

INTO

Solution: Yes. Although the numbers in AL and BL can both be interpreted as positive signed numbers, the sum (D0H) looks like a negative signed number. In this case the overflow flag is set and INTO will generate an interrupt.

1-6-6 INTR

Up to now we have only discussed the basics of this hardware interrupt signal. Let us take a closer look. No interrupt is generated by TNTR unless the interrupt-enable flag (IF) is set. This can easily be accomplished with the STI (set interrupt-enable flag) instruction. INTR must remain high until sampled by the processor, unlike NMI, which is a rising-edge triggered input. It is therefore necessary when using INTR to allow it to remain high only until an interrupt acknowledge cycle begins. For 8086/8088 systems, the 8259A programmable interrupt controller, automatically interfaces with INTR and $\overline{INTA}$. If the power of this peripheral is not needed, then custom interrupt circuitry must be designed, First, we need the INTR connection. The circuit shown in Figure 1.6 uses a flip- flop to condition the INTR input. A D-type flip-flop is used to convert the high-level requirement of INTR into an edge-sensitive request A rising edge on MINT (maskable interrupt) will clock a 1 through the flip-flop, placing a high level on INTR. When the 8086/8088 recognizes INTR and begins its interrupt acknowledge cycle, $\overline{INTA}$ will go low. This will clear the flip-flop and remove the INTR request. MINT must go low and back high again for another interrupt to be recognized.

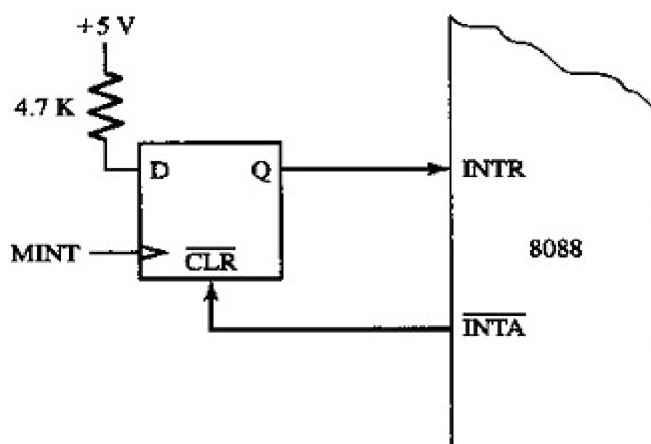


FIGURE 1.6 INTA conditioning circuitry

1.7 Interrupt service routines

The interrupt service routine referred to many times in this chapter is the actual section of code that takes care of processing a specific interrupt. The ISR for a divide-error is necessarily different from one designed to handle breakpoint or NMI interrupts.

Even though these interrupt service routines are **written** to accomplish different goals, there are portions of each that, for the sake of good programming, look and operate the same. Recall that any time an interrupt occurs, the processor pushes the flags and return address onto the stack before vectoring to the address of the ISR. Clearly, we must see that the ISR will change the data in various registers while it is processing the interrupt. Because we desire to return to the same point in our program where we left off *before* the interrupt occurred and resume processing, we insist that all prior conditions exist upon return. This means that we must return from the ISR with the state of all registers preserved. It is now the responsibility of the ISR to preserve the state of any registers that it alters. Figure 1.7 shows how this is done. In this example, DIV BL causes a divide-error interrupt. The first thing the ISR does is save the processor registers on the stack. These registers are saved with the PUSH instruction. You will need a PUSH for each register you need to save (or you can push all the registers with PUSHA). The contents of all processor registers, including the flags, are often called the environment or context of the machine. Putting a copy of everything onto the stack saves the environment that existed at the time of the interrupt.

When the body of the ISR finishes execution, it is necessary to reload the registers that were saved at the beginning of the routine. The POP instruction is used for this purpose. POPs must be done in the reverse order of the PUSHes. A ISR that uses AX, BX, and CX would look something like this:

```
ANISR: PUSH AX              ; save register
       PUSH EX
       PUSH CX
       ; body of ISR
       POP CX               ; get registers back
       POP BX
       POP AX
       IRET                 ; return from interrupt
```

Saving all registers is preferable, and will save you much heartache in the future, when you find that saving one or two registers

was insufficient as the needs of the routine became more complex. In this case, use POPA to restore all registers.

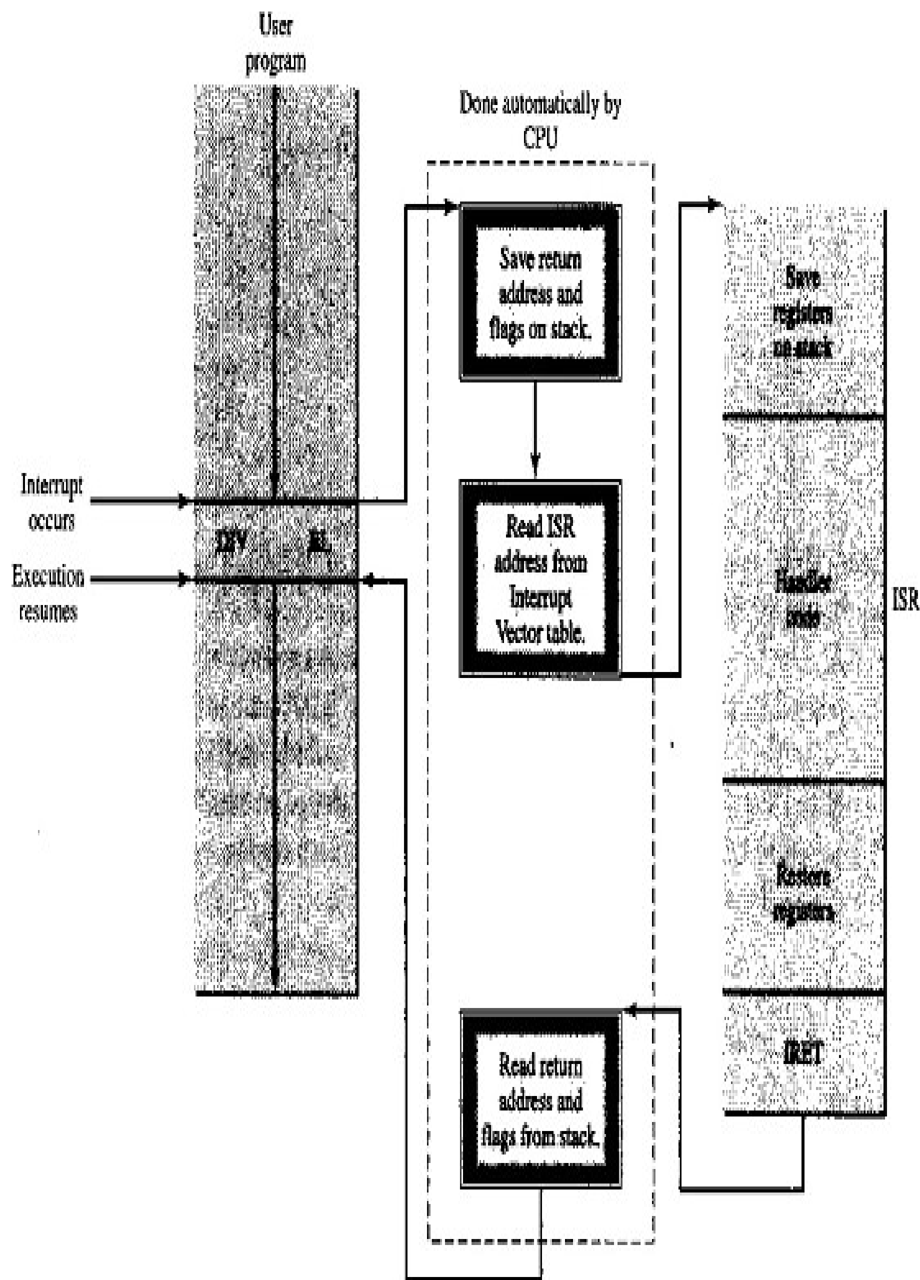


FIGURE 1.7 Storing environments during interrupt processing

1-8 Interrupt procedure of 8086

An 8086 interrupt can occur because of the following reasons:

1. Hardware interrupts, caused by some external hardware device.
2. Software interrupts, which can be invoked with the help of INT instruction.
3. Conditional interrupts, which are mainly caused due to some error condition generated in 8086 by the execution of an instruction.

When an interrupt can be serviced by a procedure, it is called as the Interrupt Service Routine (ISR). The starting addresses of the interrupt service routines are present in the first 1K addresses of the memory (Please refer to Unit 2 of this block). This table is called the interrupt vector table.

How can we write an Interrupt Servicing Routine? The following are the basic but rigid sequence of steps:

1. Save the system context (registers, flags etc. that will be modified by the ISR).
2. Disable the interrupts that may cause interference if allowed to occur during this ISR's processing
3. Enable those interrupts that may still be allowed to occur during this ISR processing.
4. Determine the cause of the interrupt.
5. Take the appropriate action for the interrupt such as – receive and store data from the serial port, set a flag to indicate the completion of the disk sector transfer, etc.
6. Restore the system context.
7. Re-enable any interrupt levels that were blocked during this ISR execution.
8. Resume the execution of the process that was interrupted on occurrence of the interrupt.

MS-DOS provides you facilities that enable you to install well-behaved interrupt

handlers such that they will not interfere with the operating system function or other interrupt handlers. These functions are:

Function	Action
Int 21h function 25h	Set interrupt vector
Int 21h function 35h	Get interrupt vector

Int 21h function 31h Terminate and stay residents 97

Here are a few rules that must be kept in mind while writing down your own Interrupt Service Routines:

1. Use Int 21h, function 35h to get the required IVT entry from the IVT. Save this entry, for later use.
2. Use Int 21h, function 25h to modify the IVT.
3. If your program is not going to stay resident, save the contents of the IVT, and later restore them when your program exits.
4. If your program is going to stay resident, use one of the terminate and stay resident functions, to reserve proper amount of memory for your handler.

Let us now write an interrupt routine to handle “division by zero”. This file can be loaded like a COM file, but makes itself permanently resident, till the system is running.

This ISR is divided into two major sections: the initialization and the interrupt handler. The initialization procedure (INIT) is executed only once, when the program is executed from the DOS level. INIT takes over the type zero interrupt vector, it also prints a sign-on message, and then performs terminate and “stay resident exit” to MS-DOS. This special exit reserves the memory occupied by the program, so that it is not overwritten by subsequent application programs. The interrupt handler (ZDIV) receives control when a divide-by-zero interrupt occurs.

```
CR              EQU   ODH          ; ASCII carriage return
LF              EQU   0Ah          ; ASCII line feed
BEEP            EQU   07h          ; ASCII beep code
BACKSP          EQU   08h          ; ASCII backspace code

CSEG            SEGMENT PARA PUBLIC  'CODE'
ORG 100h
 ASSUME CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
 INIT     PROC      NEAR
        MOV    DX,OFFSET ZDIV        ; reset interrupt 0 vector
                                     ; to address of new
                                   ; handler using function 25h, interrupt
```

```
        MOV     AX, 2500h               ; 0 handles divide-by-zero
        INT  21h
        MOV  AH,09                      ; print identification message
        INT  21h
                                ; DX assigns paragraphs of memory
                                ; to reserve
        MOV  DX,((OFFSET PGM_LEN + 15)/16) + 10h
        MOV  AX,3100h           ; exit and stay resident
        INT  21h                ; with a return code = 0
INIT    ENDP

ZDIV    PROC      FAR   ; this is the zero-divide
                                ; hardware interrupt handler.
          STI           ; enable interrupts.
            PUSH  AX           ; save general registers
            PUSH  BX
            PUSH  CX
            PUSH  DX
            PUSH  SI
            PUSH   DI
            PUSH  BP
            PUSH  DS
            PUSH   ES
            MOV  AX,CS
            MOV  DS,AX
            MOV   DX,OFFSET WARN    ; Print warning "divide by
            MOV  AH, 9              ; zero "and" continue or
            INT  21h                ; quit?"

ZDIV1:      MOV  AH,1               ; read keyboard
            INT  21h
            CMP  AL, 'C'            ; is it 'C' or 'Q'?
```

```
    JE  ZDIV3                      ; jump it is a 'C'.
    CMP  AL, 'Q'

    JE  ZDIV2                      ; jump it's a'Q'
    MOV  DX, OFFSET BAD            ; illegal entry, send a
    MOV  AH,9                      ; beep, erase the bad char
    INT  21h                       ; and try again
    JMP   ZDIV1

  ZDIV2: MOV   AX, 4CFFh           ; user wants to abort the
    INT  21h                       ; program, return with
                                   ; return code = 255
  ZDIV3: MOV  DX,OFFSET CRLF   ; user wishes to continue
    MOV   AH,9                     ; send CRLF
    INT  21h
    POP  ES                        ; restore general registers
    POP  DS                        ; and resume execution
    POP  BP
    POP  DI
    POP  SI
    POP  DX
    POP  CX
    POP  BX
    POP  AX
    IRET
   ZDIV   ENDP

SIGNON          DB    CR, LF, 'Divide by zero interrupt'
                DB    'Handler Installed'
                DB    CRLF,'$'
WARN            DB    CR, LF, 'Divide by zero detected:'
                DB    CR, LF 'Quit or Continue (C/Q) ?'
```

```
                DB      '$'
BAD             DB      BEEP, BACKSP, '',BACKSP,'$'
CRLF            DB      CR,LF,$'
PGM_LEN         EQU     $-INIT
CSEG            ENDS
                END
```

1-9 The 8259 programmable interrupt controller

Figure 1-8 shows a simplified block diagram of the 8259.

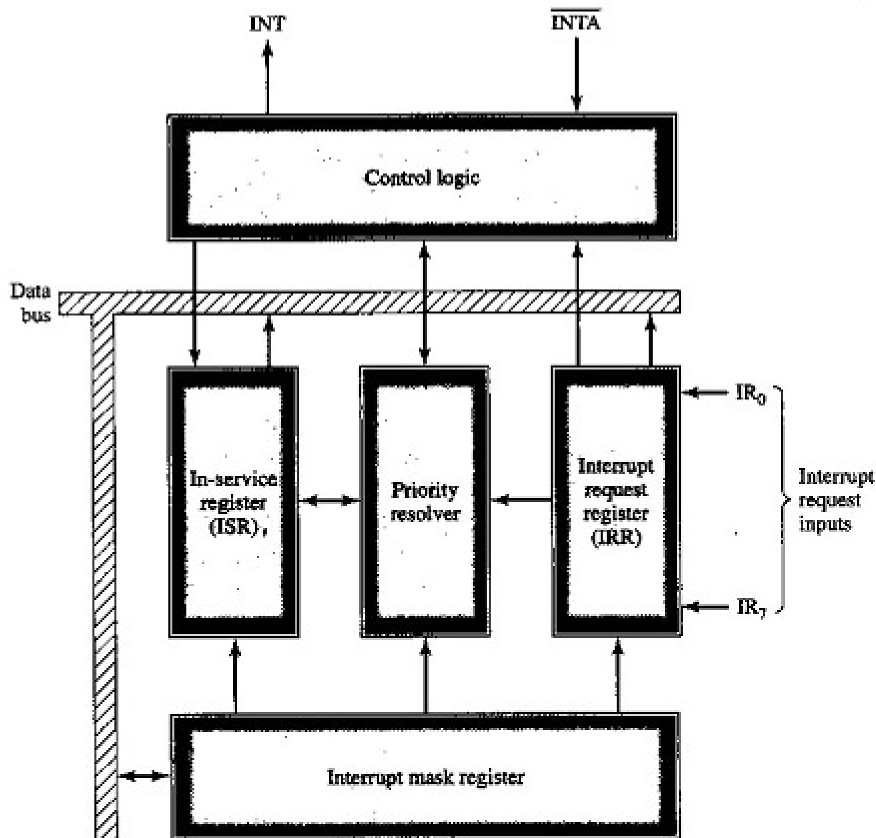


FIGURE 1-8 8259 block diagram

The functional units of 8259 are

Interrupt Request Register: The interrupts at IRQ input lines are handled by interrupt Request Register internally. IRR stores all the interrupt requests in it in order to serve them one by one on priority.

In Service Register: It stores all the interrupt requests those are being served.

Priority Resolver: It determines the priorities of the interrupt requests appearing simultaneously. The highest priority is selected and stored into the corresponding bit of ISR.

Interrupt Mask Register: It stores the bits required to mask the interrupt inputs. IMR operated on IRR at the direction of the priority resolver.

Interrupt control Logic: It manages the interrupt and interrupt acknowledge signals to be sent to CPU for serving one of the eight interrupt requests.

Data Bus Buffer: It is a tri-state bi-directional buffer interfaces internal 8259 bus to the microprocessor to data bus. Control words, status and vector information passes through data buffer during read and write operations.

Read/Write Control Logic: It accepts and decodes command from the CPU

Cascade Buffer: It stores and compares the ID's of all 8259 s used in the system. The three IO pins CAS0-2 are outputs when the 8259 used as master. The same pins acts as inputs when the 8259 is in slave mode. The 8259 in master mode sends the ID of the interrupting slave device on those lines. The slave thus select will send its preprogrammed vector address on the data bus during the next INTA pulse.

The 8086 interrupt flag is set and the INTR input receives a high signal, then 8086 will send out interrupt acknowledge pulses on its INTA pin to the INTA pin of an 8259A PIC. The INTA pulses tell the 8259A to send the desired interrupt type to the 8086 on the data bus.

1) Multiply the interrupt type it receives from the 8259A by 4 to produce an address in the interrupt vector table.
2) Push the flags on the stack.
3) Clear IF and TF
4) Push the return address on the stack.
5) Get the starting address for the interrupt procedure from the interrupt vector table and load that address in CS and IP.
6) Execute the interrupt – service procedure.

The data bus allows the 8086 to send control words to the 8259A and read a status word from the 8259A. The RD and WR inputs control these transfers when the device is selected by asserting its chip select (CS) input low. The 8-bit data bus also allows the 8259A to send interrupt types to the 8086.

The eight interrupt inputs labeled IRO through IR7 on the right side of the diagram. If the 8259A is properly enabled, an interrupt signal applied to any one of these inputs will cause the 8259A to assert its INT output pin high. If this pin is connected to the INTR pin of an 8086 and if the 8086 interrupt flag is set, then this high signal will cause the previously described INTR response.

The INTA input of the 8259A is connected to the INTA output of the 8086. The 8259A uses the first INTA pulse from the 8086 to do some activities that depend on the mode in which it is programmed. When it receives the second INTA pulse from the 8086, the 8259A outputs an interrupt type on the 8-bit data bus, as shown in figure. The interrupt type that it sends to the 8086 is determined by the IR input that received an interrupt signal and by a number you send the 8259A when you initialize it.

In the block diagram note the four boxes labeled interrupt request register (IRR), interrupt mask register(IMR), in service register (ISR) and priority resolver.

The interrupt mask register is used to disable (mask) or enable (Unmask) individual interrupt inputs. Each bit in this register corresponds to the interrupt input with the same number. You unmask an interrupt input by sending a command word with a 0 in the bit position that corresponds to that input.

The interrupt request register keeps track of which interrupt inputs are asking for service. If an interrupt input has an interrupt signals on it, then the corresponding bit in the interrupt request register will be set.

The priority resolver acts as a “judge” that determines if an when an interrupt request on one of the IR inputs gets serviced.

As an example of how this works, suppose that IR2 and IR4 are unmasked and that an interrupt signal comes in on the IR4 in the interrupt request on the IR4 input will set bit 4 in the interrupt request register. The priority resolver will detect that this bit is set and check the bits in the interrupt service register (ISR) to see if a higher priority input is being serviced. If higher priority input is being serviced, as indicated by a bit being set for that input in the ISR, then the priority resolver will take no action. If no higher priority interrupt is being serviced. Then the priority resolver will active the circuitry which sends an interrupt signals to the

8086. When the 8086 responds with INTA pulses, the 8259A will send the interrupt type that was specified for the IR4 input when the 8259A was initialized.

Now, suppose that while the 8086 is executing the IR4 service procedure, an interrupt signal arrives at the IR2 input of the 8259A. This will set bit 2 of the interrupt request register. Since we assumed for this example that IR2 was unmasked, the priority resolver will detect that this bit in the IRR is set and make a decision whether to send another interrupt signal to the 8086.

1-9-1 Command words of 8259:

The command words of 8259 are classified into two groups, they are initialization command words (ICW) and operational command words (OCW), ICWs are used to set 8259 into normal operation, where as OCWs are used for selecting any one of the modes.

Modes of 8259: There are different modes of 8259, which can be selected by the help of setting and resetting OCWs.

Fully Nested Mode: It is the default mode and when interrupt requests will come, the higehst priority interrupt request is serviced. IR0 has highest priority and IR7 has lowest.

Cascade mode: 8259 connected in a system has one master and 8 slaves to handle upto 64 priority levels. Master controls the slaves using CAS0-CAS2 which act as chip select inputs for slaves.

Buffered Mode: When problem of enabling the buffer exists, 9259 buffer enable signal on SP/EN whenever the data is placed on the bus.

2. DIGITAL INTERFACING

Structure

2.1 Introduction
2-2 Need for I/O interface
2-3 Parallel Communication
2-4 8255 Programmable Peripheral interface
2-5 Modes of 8255
    2-5-1 BSR mode
    2-5-2 I/O modes
2-6 Interfacing of key board

2.1 Introduction

Interfacing: Designing logic circuits to enable them processor to communicate with peripherals by overcoming different income due to differences in speed, data format etc.

2-2 Need for I/O interface

When a microprocessor is enabled to with peripherals, there arise incompatibilities due to following reasons.

1) When the processor is working with multiple peripherals, it must interpret the address and M/I/O select signals to a peripheral to which it must communicate.

2) Difference in the timing according to transfer is to be taken place. When a high speed processor synchronized with a slow peripheral, there arises incompatibilities difference in speed.

3) Difference in format of data this is to be transferred between processor and peripherals. Processors send/receive parallelly where as some peripherals transmit data serially.

4) Some handshaking signals and interrupt are needed for reliable data transfer.

To over come all these problems circuit must include address decoders, buffers, and handshaking devices.

The interface logic that supports system bus must include

1) Bus drivers and receiver

2) Logic to transfer control signals to hand shaking signals

3) Logic to decode the address.

2-3 Parallel Communication

In parallel communication data is transferred simultaneously on several lines to increase data transfer rate. But the cost of parallel lines increases when it is used over long distances.

A parallel interface with handshaking lines and separate I/O connections to I/O device are shown below figure 2-1.

1) An input is carried out by putting data on data bus and a 1 on data in ready line.

2) Interface will latch the data into the data and putting a 1 on data in acknowledges line.

3) The device receiver acknowledges and drop data and ready signals.

4) After receiving data, interface sets ready bit and sends interrupt request.

5) CPU receives data, cleans ready status puts data lines on high impedance state.

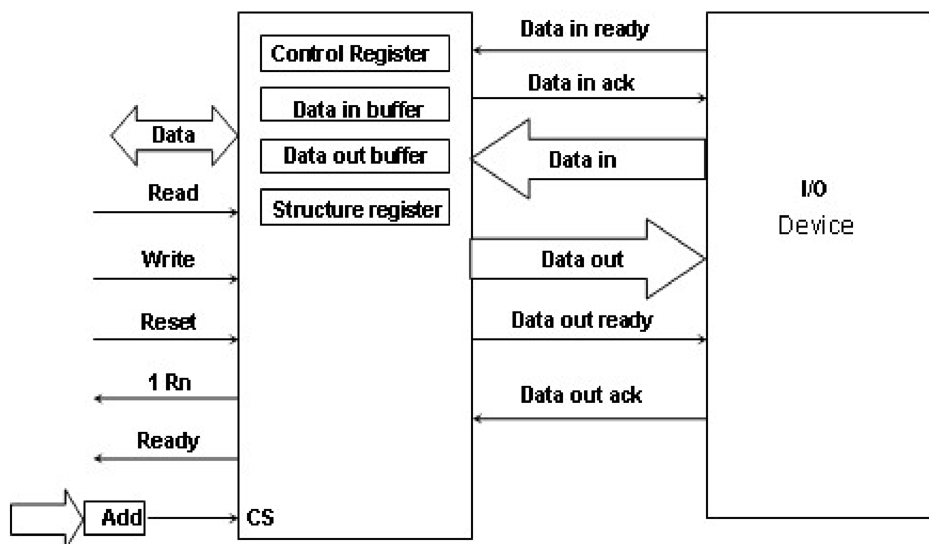
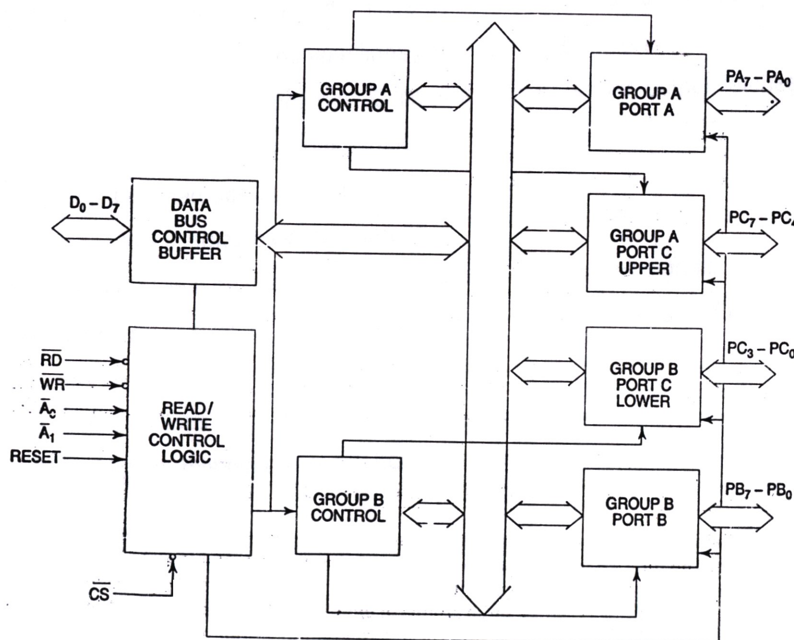


Figure 2-1 Parallel communication

2-4 8255 Programmable Peripheral interface

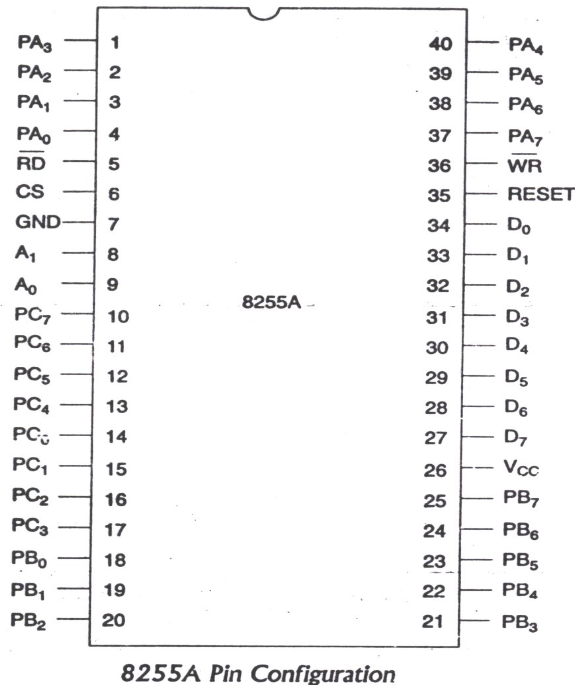
- It is a programmable peripheral parallel interface device.
- It provides parallel input and output ports
- It has 24 input/output lines which my be programmed into two groups of 12 lines or 3 groups
- It contains a control register and addressable ports A, B, C.
- These bits are divided into groups A and B.
- Group A contains port A and 4 MSB bits of port C.
- Group B contains port B and 4 LSB bits of port C

All the above ports can be operated as input or output ports by programming the control word register.



8255 Internal Architecture

Figure2-2



8255A Pin Configuration

Figure 2-3

PA0-PA7 : These are 8 port A lines acting as latched output or buffered input.

PB0-PB7 : These are 8 port B lines acting as latched output or buffered input.

PC0-PC3 : These are lower port C lines acting as inner output latches or input buffers. They also act as handshake lines in mode 1 and mode 2.

$\overline{RD}$**:** If it is low read operation is enabled

$\overline{WR}$: If it is low it indicates write operation.

$\overline{CS}$: If it goes low it enables 8255 to respond to RD and WR signals.

A1 – A0: Address lines used to address any one of the four registers (three ports and control register).

D0-D7: Data bus lines carry data or control word.

RESET: A high on this pin clears control word registers.

The 8-bit data bus buffer is controlled by read/write logic. Read/write logic manages all internal and external transfer of both data and control words.

The ports of the 8255 can be used as input output ports depending on address pins A0 and A1, RD and WR signals

A1	A0	RD	WR	
0	0	0	1	Port A to data
0	1	0	1	Port B to data
1	0	0	1	Port C to data
0	0	1	0	Data bus to port
0	1	1	0	Data bus to port
1	0	1	0	Data bus to port

The control register of 8255 is of 8-bit length and the format of control word is given below.

0/1 D6 D5 D4 D3 D2 D1 D0

0 for Mode select Associated with Mode select Associated with BSR For group A group A for group B group B

2-5 Modes of 8255

There are two basic modes of operation, they are I/O mode and Bit set reset (BSR) mode. Under I/O mode of operation there are 3 modes. They are mode 0, mode 1, mode 2.

2-5-1 BSR mode:

Any of the 8 bits of port C can be set depending on B0 of the control word. The bit to be set or reset is select select flags B3, B2 and B1 of CWR.

D3	D2	D1	Selected Bits
0	0	0	C0
0	0	1	C1
0	1	0	C2

0	1	1	C3
1	0	0	C4
1	0	1	C5
1	1	0	C6
1	1	1	C7

2-5-2 I/O modes:

Mode 0:

This is called as basic input/output mode in this mode each group is divided into two sets, they are Port A and Port C, Port B and Port C which can be used for inputting and outputting.

Two 8 bit ports A and B, two 4 bit ports C upper and lower are available. C lines can be combined as third 8-bit port. Any port can be used as input or output port.

Output port is latched and input are not. A maximum of four ports are available so that overall 16 I/O configurations are possible.

D4 Port A

D3 Port C(Upper half)

D2 Port B

D1 Port 1 (Lower half)

Mode 1:

This mode is called as strobed input/output mode. Two groups A and B are available for strobed data transfer. Each group one 8bit I/O port and one 4 bit control/data port.

When group A is in this mode, port A is used input and output and port C(upper half) is used for handshaking and control signals.

For Inputting:

PC4 $\overline{STB}_A$ A0 to this pin latches PA7-PA0 into port A.

PC5 $\overline{IBF}_A$ If it is 1 port A is full with data that is not taken by CPU otherwise input can send a new data

PC6, PC7 Used as output control signals to the device or input status signals.

For outputting:

PC4, PC5 Used as output control signals to the device or input status signals.

PC7 $\overline{OBF}_A$ If it is 0 port A is sending data to output device.

PC6 $\overline{ACK}_B$ Device sends a 0 on this pin when it data from port A.

When group B is in this mode, port B is used for input and output and port C(lower half) is used for handshaking and control signals.

For Inputting:

PC2 $\overline{STB}_B$ A0 to this pin latches PB7-PB0 into port B.

PC1 $\overline{IBF}_B$ If it is 1 port A is full with data that is not taken by CPU otherwise input can send a new data

For outputting:

PC4, PC5 Used as output control signals to the advice or input status signals

PC1 $\overline{OBF}_B$ If it is 0 port B is sending data to output device.

PC3 is used for INTR and associated with group PC0 is used as INTR for group B.

MODE 2:

This mode is also called as strobe bidirectional I/O. This mode is only applicable group A. IN this mode it acts as bidirectional port.

Only 8-bit port in group A is available. A 5-bit control port is available.

I/O lines are provided by port C, PC2-PC0.

PC4 $\overline{STB}_A$ A0 to this pin latches PA7-PA0.

PC5 $\overline{IBF}_A$ If it is 1 port A is full with data that by CPU otherwise input can send a new data.

PC6 $\overline{ACK}_A$ indicates that the device is ready data from PA7-PA.

PC7 $\overline{OBF}_A$ It is 0 when port A is filled with new data form the CPU and is set to 1 when data are taken by the device.

When group A is in mode 0 group B may be in mode 0 or mode 1.

2-6 Interfacing of key board

Problem: Interface 4*4 key boards with 8086 using 8255. Port A is used for selecting rows of keys while port B is used as an input port for sensing a closed key. The key lines are selected one by one through port A and port B lines are polled continuously till a key closure is ensued.

The address of port A and port B will be 8000 and 8002 while the address of CWR will be 8086. The control word of the problem is 82.

Program:

```
        MOV AL, 82 LOAD CWR WITH CONTROLWORD
        MOV DX.8006
        OUT DX, AL
        MOV BL, 00 INITIALIZE BL FOR KWY CODE
        XOR AX, AX CLEAR ALL FALGS
        MOV DX, 8000      ;PORT ADDRESS IN AX
        OUT DX, AL        ;GROUND ALL ROWS
        ADD DX, 02,       ;PORT b ADDRESS IN DX
WAIT:   IN AL, DX         ;READ ALL COLUMNS
        AND AL, 0F        ;MASK DATA LINES D7-D4
        CMP AL, 0F        ;ANY KEY CLOSED
        JZ WAIT
```

```
          CALL DEBOUNCE
          MOV AL, F7          ;LOAD DATA BYTE TO GROUND
          MOV BH, 04          ;A ROW AND SET ROW COUNTER
NXTROW:   ROL AL, 01      ;ROTATE AL TO GROUND NEXT ROW
          MOV CH, AL
          SUB DX, 02
          OUT DX, AL
          ADD DX, 02
          IN AL, DX
          AND AL, 0F
          MOV CL, 04
NXTCOL:   RCR
          JNC CODEKEY
          INC BL
          DCR CL
          JNZ NXTCOL
          MOV AL, CH
          DCR BH
          JNZ NXTROW
          JMP WAIT
CODEKEY:  MOV AL, BL
          MOV AH, 4CH
          INT 21
```

3. ANALOG INTERFACING

Structure

3-1 Introduction
3-2 Interfacing with switches and 7-sigment display unit
  3-2-1 Hardware Setup
  3-2-2 Multiplexed 7-seg Displays
3-3 Interfacing analog to digital data converters
3-4 Interfacing digital to analog converter
3-5 Interfacing stepper motor
3-6 Interfacing of key board

3-1 Introduction

The data bus of microprocessor is connected to I/O port. The address of the device is used as chip select of the device. By the help of control signals such as IORD and IOWR device operations are carried out.

3-2 Interfacing with switches and 7-sigment display unit

An interface is to be designed to read the status of switches SW1 to SW8 and to display the number of a key that is pressed.

Solution:

The status of the switches is first read into the register AL. The bit corresponding to the switch is checked by rotating AL through carry and then checking the carry flag. If the carry flag is 1 after one left rotation it means SW1 is on. If the carry flag sets after two rotations then it is SW2 and so on. For each rotation count CL is incremented. The eight bit contents of register CL are converted to 7 segment code.

3-2-1 Hardware Setup

An 74LS245 buffer is used as input port to read the status of switches SW1 to SW8. An 74LS373 latch is used as output port where the 7-segment display unit is connected.

The address of the input port is xxx8, hence only A0-A3 are used as address lines, Similarly the address of the output port is xxxA.

Program:

```
        MOV BL, 00      CLEAR BL
        MOV CL, OO      CLEAR CL
        XOR AX, AX      CLEAR ACCUMULATOR
        IN AL, 08       READ THE STATUS OF
        INC CL          INCREMENT CL FOR 1ST SWITCH
BACK:   RCR AL          ROTATE SWITCH STATUS
        JC XX           IF CARY HALT
        INC CL          ELSE INCREMENT CL FOR NEXT
SW
                        JMP BACK
FRONT:  MOV AL, CL      READ SW NO INTO AL
                        OUT OA, AL
                        HLT
```

3-2-2 Multiplexed 7-seg Displays

To display a single digit one output port is required, if we want to display 5 digits we need 5 ports. This will increase complexity of hardware. Thus only 2 ports are needed, one for selecting the display unit one at a time and other to send data to it. Each of the unit is active for a short duration and the process continues in a loop. This is repeated with high frequency so that the complete display containing more that one 7-segment display appears to be stationary.

Problem

Display numbers 1 to 5 continuously using 5, 7-segment displays.

Let us select the two port addresses 0004 and 0008 for the output ports. The first port 0004 outputs 7-segments code while the second port 0008 selects the display unit by grounding the common cathode.

In a 7-segment unit for a particular LED to be ON that particular anode should be 1 and common cathode line should be grounded. The code to display the numbers is given by

NO	a	b	c	d	e	f	g	dp	
	A7	A6	A5	A4	A3	A2	A1	A0	
1	1	1	0	0	0	0	0	0	= CO
2	1	1	0	1	1	0	1	0	= DA
3	1	1	1	1	0	0	1	0	= F2
4	0	1	1	0	0	1	1	0	= 66
5	1	0	1	1	0	1	1	0	= B6

These codes are stored in a lookup table starting from 2000.

Program

```
        MOV AX, 2000        INITIALIZE POINTER TO
        MOV DX, AX CODE TBALE DS:BX
        MOV BX, 0000
NEXT:   MOV AL, 00 GET 1ST NUMBER
        MOV DHAL
        MOV CL, 05          COUNT FOR DISPLAY
        MOV DL E1           SELECTION CODE FOR 1ST
DISPLAY
AGAIN:  XLAT
        OUT 04, AL          SEND THE CODE FOR THE FIRST
                            NUMBER TO PORT 04
        MOV AL, DL
        OUT 08, AL          SELECT 1ST DISPLAY
        ROL DL              DECIDE CODE FOR SLECTING
NEXT
        INC DH              DISPLAY FOR NEXT NUMBER
        MOV AL, DH GET NUMBER TO BE DISPLAYED
        LOOP AGAIN
        JMP NEXT
```

3-3 Interfacing analog to digital data converters

An A/D converter is treated as input device. Microprocessor sends initializing signal to the A/D converter to start the analog to digital conversion. Microprocessor must wait until the conversion is over. After conversion A/D converter sends end of conversion signal and the result is

send to output buffer. Time taken between start of conversion and end of conversion is called conversion delay.

ADC0809

It is 8-bit CMOS successive approximation converter whose conversion delay 100 microsecs at a clock frequency of 640 KHz. They have 3 :8 analog multiplexer so that at a time eight different analog inputs can be connected to chips. Out of these 8 inputs only one can be selected for conversion by using address lines ADD A, ADD B and ADD C.

Problem: Interface 0809 with 8089 using 8255 ports. Port A of 8255 is used for transferring digital data output of ADC to the CPU and port C for control signals.

The analog input I/P2 is used, its address is A, B, C equal to 010 to select. Port C upper acts as the input port to receive the EOC signal while port C lower acts as the output port to send SOC to ADC. Port A acts as 8-bit input data port to receive the digital data output from ADC. For this purpose 8255 control word is 98.

Program:

```
        MOV AL, 98 INITIALISE 8255
        OUT CWR, AL
        MOV AL, 02 SELECT I/P2 AS ANALOG INPUT
        OUTPORTB, AL
        MOV AL, 00 START OF CONVERSION
        OUTPORT C, AL
        MOV AL, 01
        OUT PORTC, AL
        MOV AL, 00
        OUT PORT C, AL
WAIT:   IN AL.PORTC         CHECK FOR EOC
        RCR
        INC WAIT
        IN AL, PORTA        IF EOC READ DATA IN AL
        HLT
```

3-4 Interfacing digital to analog converter

Problem:

Interface AD 7523 (16-bit DAC containing R-2R ladder type) for to digital to analog conversion with 8086.

Port A is initialized as the output port for sending the digital data as input to DAC. The ramp starts at 0 and increases to F2. The ramp period is 1 ms. The count F2 is calculated by dividing the required delay of 1 ms by the time required.

For the execution of loop once. The ramp slope can be controlled by calling a controllable delay after the OUT instruction.

Program:

```
        MOV AL,80 INTIALISE PORT A AS OUTPUT
        OUT CWR, AL
AGAIN:  MOV AL, 00 START THE RAMP FROM 0
BACK:   OUT PORTA, AL INPUT 00 TO DAC
        INC AL
        CMP AL, F2
        JB BACK
        JMP AGAIN
        END
```

3-5 Interfacing stepper motor

A stepper motor is a DC motor which rotates in steps rather than continuously. It mainly consists two parts namely stator and rotor. Stator is a stationary element containing four poles on which windings are present. Rotor is a rotating shaft containing magnetic poles. To rotate the shaft of the stepper motor a sequence of pulses are applied to the windings of the motor.

A typical stepper motor have a torque of 3kg-cm operating at 12V 1.2A and a step angle of 1.8 degree with 200 rotor teeth.

The four windings of stator A, B, C, D are applied with the required pulses in cyclic fashion. By reversing the sequence of execution the direction of rotation of the motor shaft is reversed. The excitation sequence of windings is shown below

Clock wise	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	0	0	0
Anticlockwise				
1	1	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	1	0	0
5	1	0	0	0

Hardware Setup:

8255 Port A is used to interface stepper motor with 8086. The port A address is 0740. The portA bit PA0 drives winding A, PA1 drives B and so on.

Program:

```
          MOV    AL, 80
          OUT    CWR, AL
          MOV    AL, 88          ;Bit pattern 10001000 to start the
          MOV    CX, 1000        ;Sequence of excitation
AGAIN 1:  OUT    PORTA, AL
          CALL   DEALY     ;Excite A, B, C, D in sequence with delay
          ROL    AL, 01
          DEC    CX
          JNZ    AGAIN 1
          INT    21
```

The count for rotating the shaft of motor through a specified angle is calculated by rotor teeth.

Count C = number of teeth* angle/360

3-6 Interfacing of key board

Problem: Interface 4*4 key board with 8086 using 8255. Port A is used for selecting rows of keys while port B is used as an input port for sensing a closed key. The key lines are selected one by one through port A and port B lines are polled continuously till a key closure is ensued.

The address of port A and port B will be 8000 and 8002 while the address of CWR will be 8086. The control word of the problem is 82.

Program:

```
         MOV    AL, 82         ; LOAD CWR WITH CONTROLWORD
         MOV    DX.8006
         OUT    DX, AL
         MOV    BL, 00         ; INITIALIZE BL FOR KWY CODE
         XOR    AX, AX         ; CLEAR ALL FALGS
         MOV    DX, 8000       ; PORT ADDRESS IN AX
         OUT    DX, AL         ; GROUND ALL ROWS
         ADD    DX, 02,        ; PORT b ADDRESS IN DX
WAIT:    IN     AL, DX         ; READ ALL COLUMNS
         AND    AL, 0F         ; MASK DATA LINES D7-D4
         CMP    AL, 0F         ;ANY KEY CLOSED
         JZ     WAIT
         CALL   DEBOUNCE
         MOV    AL, F7         ; LOAD DATA BYTE TO GROUND
         MOV    BH, 04         ;A ROW AND SET ROW COUNTER
NXTROW:  ROL    AL, 01      ;ROTATE AL TO GROUND NEXT ROW
         MOV    CH, AL
         SUB    DX, 02
         OUT    DX, AL
         ADD    DX, 02
         IN     AL, DX
         AND    AL, 0F
         MOV CL, 04
NXTCOL:  RCR
         JNC    CODEKEY
         INC    BL
         DCR    CL
```

```
            JNZ     NXTCOL
            MOV     AL, CH
            DCR     BH
            JNZ     NXTROW
            JMP     WAIT
CODEKEY:    MOV     AL, BL
            MOV     AH, 4CH
            INT     21
```

4. SUMMARY

We have seen that there is a fixed process used by the S0x86 to implement and process an interrupt. The CPU, when interrupted, saves the flag register and program counter on the stack, clears the trace and interrupt-enable flags, and loads the interrupt service routine address from the interrupt vector table. The interrupt vector table occupies memory locations 00000 through 003FFH and contains pairs of words that represent the execution addresses for each of the 256 interrupts. These pairs correspond to IP and CS values of each ISR. The interrupt number used to access the table may be internally generated by the processor, or may be supplied by external hardware during an interrupt acknowledge cycle. The processor has only two hardware interrupts: NMI and INTR. NMI cannot be disabled, but INIR can.

Interrupts are caused through software or by an external hardware request. The software interrupt may be generated intentionally by the programmer via INT and INTO, or by accident, via a run-time error such as division-by-zero. All interrupt service routines should preserve the state of any registers used to allow a proper return.

Three examples of actual interrupt service routines were also presented, followed by a short set of troubleshooting tips.

Further discuss programmable peripheral interface 8255, and analog interface have been presented in significant detail along with the interface.

5. Test yourself

1. What is the processor's environment? Why is it important to save the environment during interrupt processing?

2. Explain the different ways interrupts are generated.

3. How is INTR disabled? How is it enabled?

4. What is the interrupt vector table address for an INT 21H?

5. The address of the ISR for INT 25H is 03C0:9AE2 (in CS: IP format). Show how this address is stored within the interrupt vector table.

6. Which interrupt is recognized first, NMI or single-step?

7. What high-priority event might require the use of NMI in a computer designed for aircraft engine control?

8. During an interrupt acknowledge cycle, 30H is placed on the data bus. Where is the ISR address fetched from?

9. Draw and discuss internal architecture of 8255.

10. Write a program for analog inter facing.

11. Write a program for keyboard interfacing.

UNIT - IV

1. 8254 PROGRAMMABLE INTERVAL TIMER

Structure

1.1 Introduction
1-2 Interfacing the 8254
1-3 Programming the 8254

OBJECTIVES

In this section you will learn about:

- Discuss about internal architecture of 8254.
- Discuss about interface with necessary programming.

1.1 Introduction

Many applications require the processor to perform an accurate time delay between a set of operations. For example, a microprocessor might be dedicated to reading a custom keypad or driving a multiplexed display. Both applications require a small time delay between it- pealed input and output operations. A programmer may decide to use a software delay loop, such as:

```
DELAY:      MOV   CX, 4000      ; init delay counter
WAIT:       LOOP  WAIT          ; loop until CX=0
```

The total amount of delay involves 4,000 executions of the LOOP instruction. This time can be estimated by multiplying the processor's clock period by the total number of states required to execute the LOOP instructions. This type of delay loop has two main disadvantages. While the loop is executing, the processor is not able to do anything else, such as execute instructions pot related to the loop. Also, the time delay becomes inaccurate if the I processor is interrupted. For these reasons some designers (and programmers) prefer to do their timing with hardware. Software is used to program the hardware for a specific time delay. At the end of the time delay, the processor is interrupted. This

frees up the processor for other kinds of execution while the hardware is performing the time delay.

A peripheral designed to implement the type of time delay just described (and many others) is the *8254* programmable interval (liner. Figure 1.1 shows the signal groups of the 8254. The *8254* is interfaced through a group of 110 ports. Three internal counters can be programmed in a variety of formats, including, 4-digit BCD or 16-bit binary counting, square-wave generation, and one-shot operation. These formats allow the 8254 to be used for a number of different timing purposes. A short list of these applications includes real time clocks (and/or calendars), specific time delay generation, frequency synthesis, frequency measurement, and pulse-width modulation.

First let us see how the 8254 is interfaced to the 8086/8088.

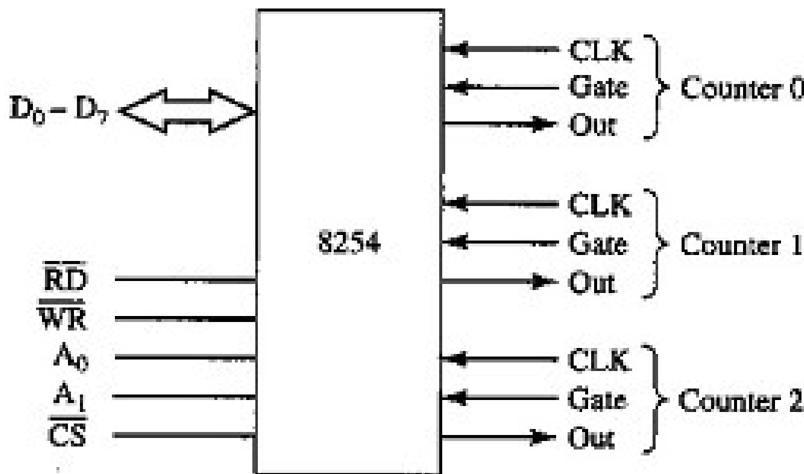


Figure1-1 The 8254 programmable interval timer

1-2 Interfacing the 8254

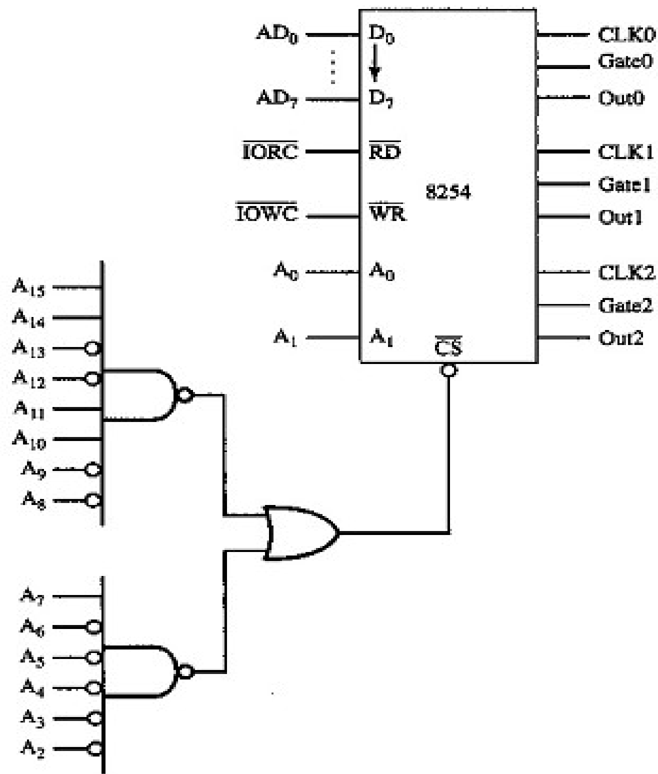


Figure 1.2 8254 Interfaced to the 8086/8088

The port-address decoder needed to connect the 8254 to the processor's address bus is shown in Figure 1. 2. Here, the 8254 is interfaced with an 8086/8088 operating in maximum mode (indicated by the IORC and IOWC command signals).

The two 8-input NAND gates in the port-address decoder map the 8254 into four I/O locations. CC8OH through CC83H. The first port (CC80H) is used to read and write counter 0. The second two ports (CC81H and CC82H) access counters 1 and 2, respectively.

The fourth port (CC83H) is used to control the 8254. Each counter contains CLX and GATE inputs, and one output, OUT. Counters may be cascaded by connecting the OUT output of one to the GATE of the other.

1-3 Programming the 8254

Each of the 8254's three counters can be programmed and operated independently of the others. This discussion will concentrate on programming and using counter 0 only.

Counter 0 is a 16-Sit synchronous down counter that can be preset to a specific count, and decremented to 0 by pulses on the CLK0 input. Counter 0 may operate in any of six different modes (selected with the use of a control word). Each mode supports four-digit BCD and 16-bit binary counting Thus, counts may range from *9999* to 0000 BCD or FFFF to 0000 hexadecimal. Programming the counter requires outputting a control word and an initial count to the 8254.

Figure 1.3 shows the bit assignments in the 8254's control word. Two bits (7 and 6) are used to select the counter being programmed. Another two bits (Sand 4) are used to control the way the counter is loaded with a new count. Bits 3, 2, and 1 are used to select one of six modes of operation. The least significant bit is used to select BCD or binary counting. To begin a counting operation, the control word must be output, followed by the 1- or 2-byte initial count. The initial count value output to the *8254* goes into a *count register* The count register is cleared when the counter is programmed (upon reception of the control word) and transferred to the actual down counter after it gets loaded with the 1- or 2-byte initial count. The counter may be loaded with a new count at any tints, without the need for a new control word.

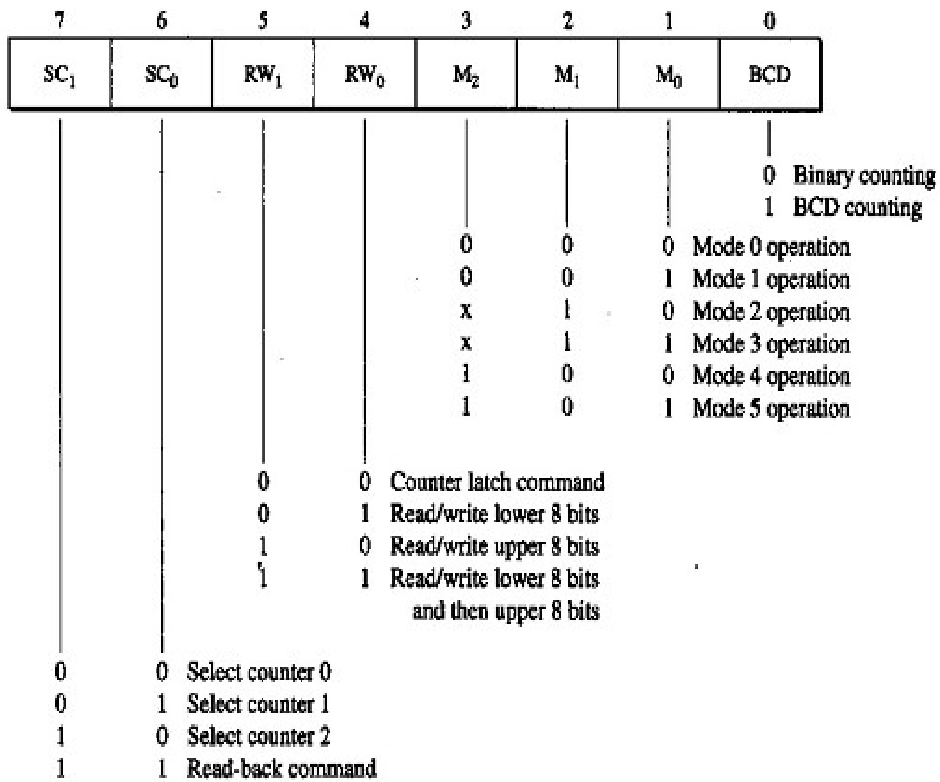


Figure 1.3 8254 control word

2. SERIAL COMMUNICATION INTERFACES

Structure

2-1 Introduction
2-2 Asynchronous communication
2-3 Synchronous Communication
2-4 USART (8251)
    2-4-1 Asynchronous mode (receive)
    2-4-2 Synchronous Mode (Transmission
    2-4-3 Synchronous Mode (receiver)
    2-4-4 Control words of 8251
    2-4-5 Command Instruction control Word
2-5 RS 232 Serial data standard
    2-5-1 Voltage Levels
    2-5-2 PIN (signal) Description
2-6 IEEE 488 BUS

OBJECTIVES

In this unit you will learn about:

- The differences between Asynchronous and synchronous communication
- Discuss internal architecture of USAR.
- Discuss internal configuration of RS 232.
- Discuss about futures of IEEE 488 bus

2-1 Introduction

Many I/O devices transmit data bit by bit i.e., one bit at a time over a single conductor or communication channel known as serial communication.

Serial communication is basically divided into two types, they are synchronous and asynchronous communication. When the speed of transmitter and receiver are same then it is said to be synchronous communication and if they are not same then it is said to be asynchronous communication.

Again these are subdivided into simplex, duplex and full duplex modes. In simplex mode, data is transmitted only in one direction over a single communication channel. In half duplex same communication channel is used for sending and receiving data. Since there is only one

channel it is not possible to send and receive data at a time. If separate lines are provided for sending and receiving data at a time then it is called full duplex.

2-2 Asynchronous communication

Since the speed of transmitter and receiver are not same.

Transmitter must inform receiver start of the transmission, end of transmission etc. details for reliable transmission.

When transmitting data, first bit is always 0 and is called start bit. It is followed by 5 and 8 information bits. These information bits are followed by parity bits and stop bits. The last bits are 1st called as stop bits.

2-3 Synchronous Communication

In this case there is no need to synchronize the transmitter and receivers since their speeds are same.

Rate of transmission is more in the case of synchronous communication when compared with that of asynchronous communication since in the later case extra bits are included for synchronization.

2-4 USART (8251)

It is a universal synchronous and asynchronous receiver and transmitter.

It is used for serial communication interface. It converts the parallel data into serial and serial data into parallel data.

The internal blocks of 8251 are transmitter and receiver buffers, transmit and receiver control units, modem control, read/write logic etc. The read write control logic controls the operation of the peripheral depending upon the operations initiated by CPU. This unit also selects one of the two internal addresses those are control address and data address at the behest of C/D signal. The modem and the USART transmit unit transmits data byte received by the data buffer from the CPU for further serial communication. The transmit buffer is a parallel to serial converter that receives a parallel byte for conversion into a serial signal. The receiver unit receives serial data and converts into parallel.

RESET: A high on this pin force 8251 into an idle state.

CLK: It is used to generate internal device timings and is connected clock generator.

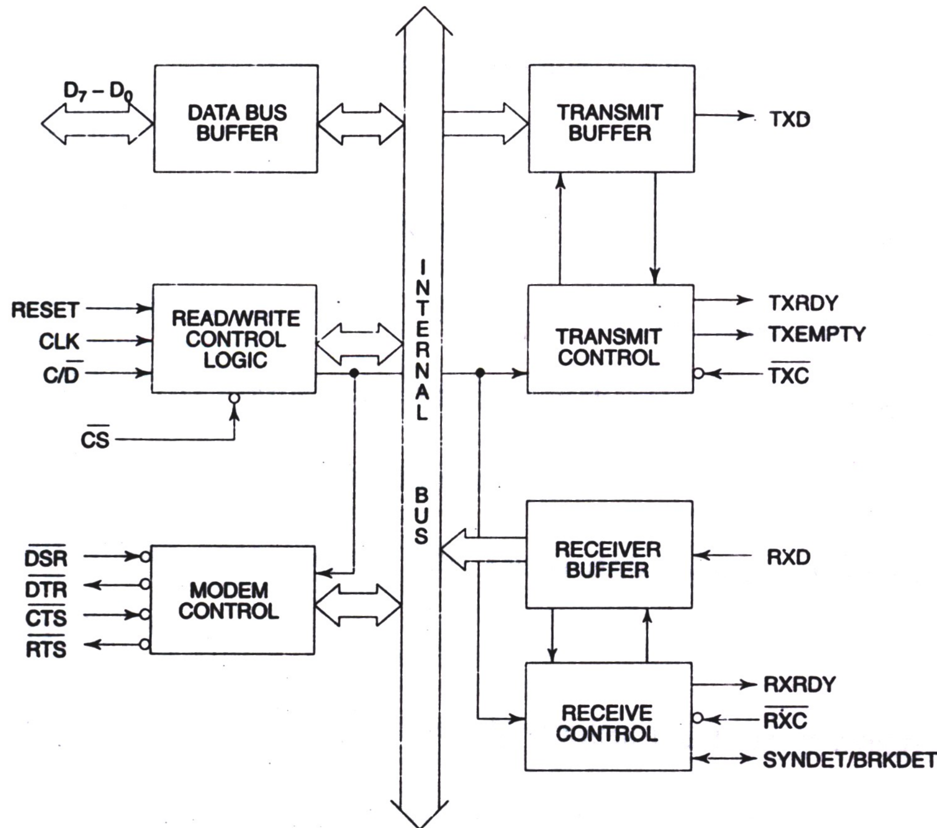
$\overline{TXC}$ **:** Transmitter clock input: It is used to control the rate at which the character is to be transmitted. In synchronous mode transmission rate is equal to TXC frequency and in asynchronous mode it is equal to 1 or 1/6 or 1/64 of the TXC frequency.

TXD: Transmitted data output: The serial data bits are transmitted through this pin.

$\overline{RXC}$ **:** Receiver clock input: It is used to control the rate at which the character is to be received. In synchronous mode transmission rate is equal to RXC frequency and in asynchronous mode it is equal to 1 or 1/6 or 1/64 of the RXC frequency.

RXD: Receiver ready output: The serial data bits are received through this pin.

RXRDY: Receiver ready output: This pin indicates that 8251 contains a character to be ready by CPU.



8251A Internal Architecture

TXRDY: Transmitter ready: This pin indicates CPU that the transmitter is ready to accept a new character for transmission from CPU.

$\overline{\text{DSR}}$: Data set ready: It is used to check if the data set is ready when communicating with a modem.

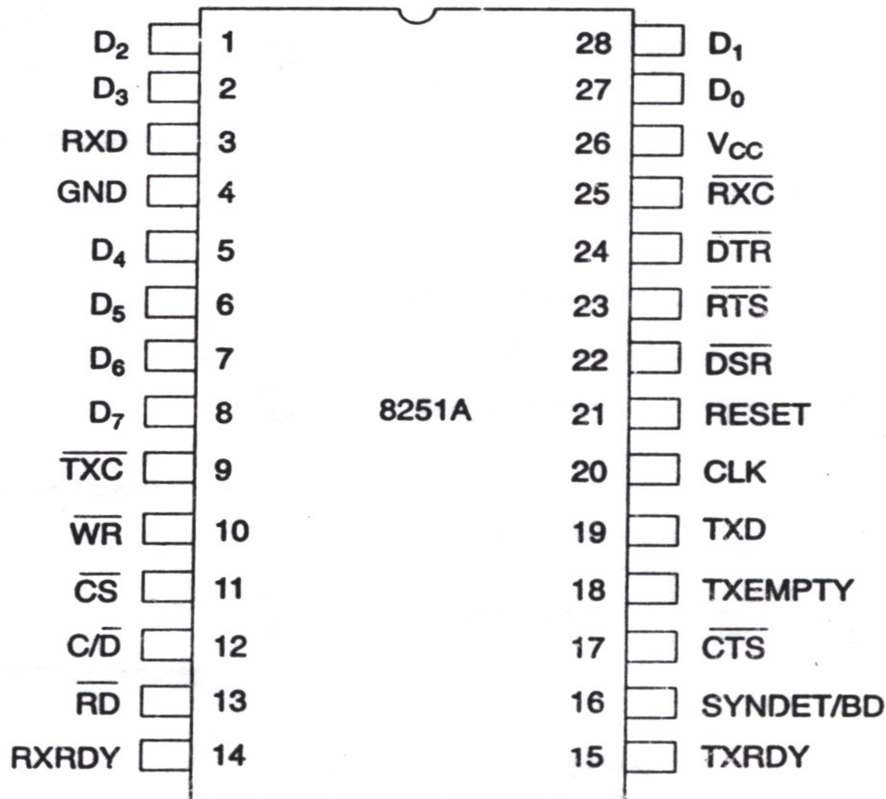
$\overline{\text{DTR}}$: Data terminal ready: A low on this pin indicates that the device is ready to accept data when the 8251 is communicating with a modem.

$\overline{\text{RTS}}$: Request to send data: A low on this pin indicates the modem that the receiver is ready to receive a data byte from the modem.

$\overline{\text{CTS}}$: Clear to send: A low on this pin enables 8251 to transmit the serial data provided the enable bit in the command byte is set to 1.

TXE: Transmitter Empty: This pin goes high when there are no characters to transmit.

SYNDET: This pin is used in the synchronous mode for detecting sync characters.

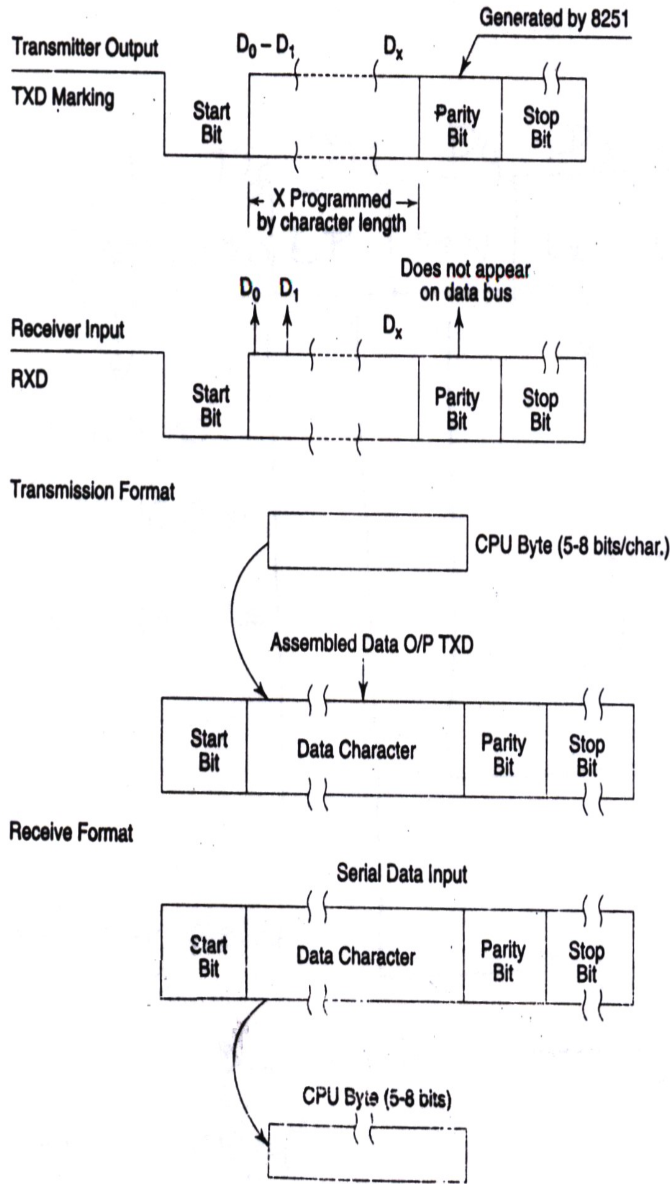


8251A Pin Configuration

When a data character is sent to 8251 by CPU is adds start bits prior to serial data bits, followed by parity bit and stop bits using the asynchronous mode instruction control word format.

2-4-1 Asynchronous mode (receive)

Falling edge of RXD marks as a start bit, and bit counter starts counting. The bit counter locates data bits, parity bit and stop bit. If a low level is detected as stop bit, the 8 bit character is located into parallel I/O buffer. Then RXRDY is raised high to indicate to CPU that a character is ready for it.



2-4-2 Synchronous Mode (Transmission)

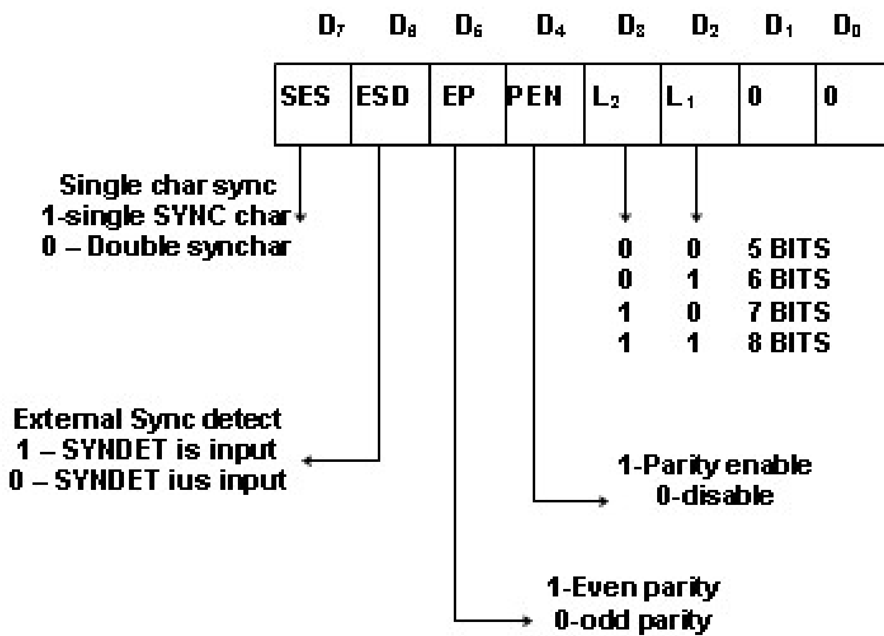
TXD output is high until CPU sends a character to 8251 Which usually is a sync character.

When $\overline{\text{CTS}}$ the first character goes low is serially transmitted out.

Data is shifted out at the rate as $\overline{\text{TXC}}$ over TXD output line.

2-4-3 Synchronous Mode (receiver)

In this mode character synchronization can be achieved internally or externally. In external SYNC mode synchronization is achieved by applying a high on SYNDET pin that forces 8251 out of HUNT mode. In internal SYNC mode the content of the receiver buffer is compared with the first SYNC character at every edge of RXC until it matches.

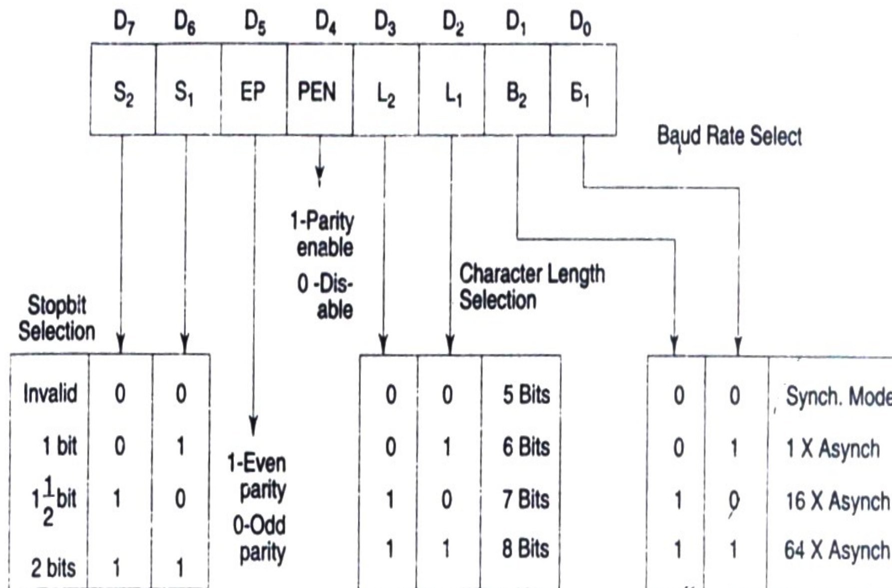


2-4-4 Control words of 8251

The control words of 8251 are divided into two types

Mode instruction control word

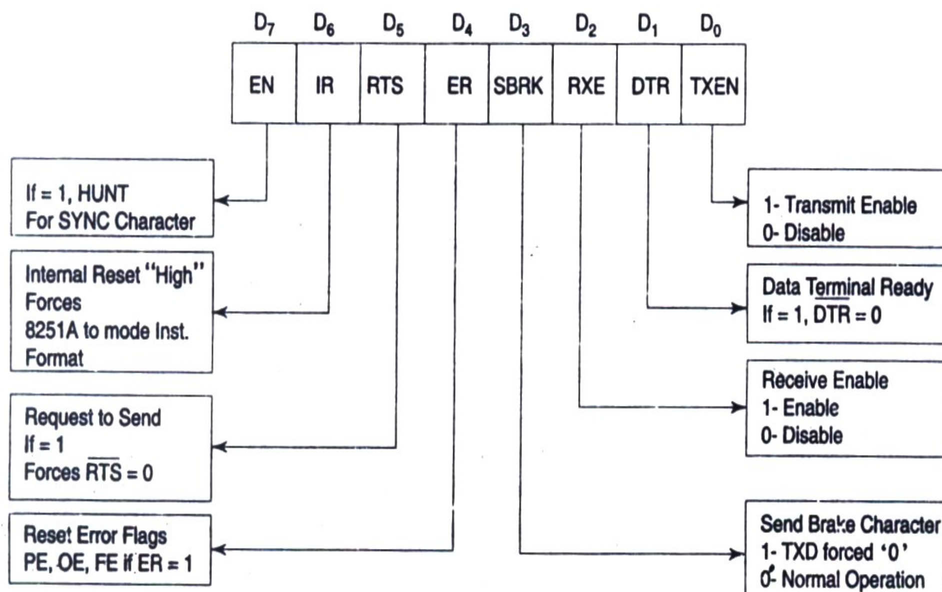
This defines general operational characteristics of 8251, when it is written into 8251 SYNC characters or command instructions may be programmed further as per the requirements.



Mode Instruction Format Asynchronous Mode

2-4-5 Command Instruction control Word

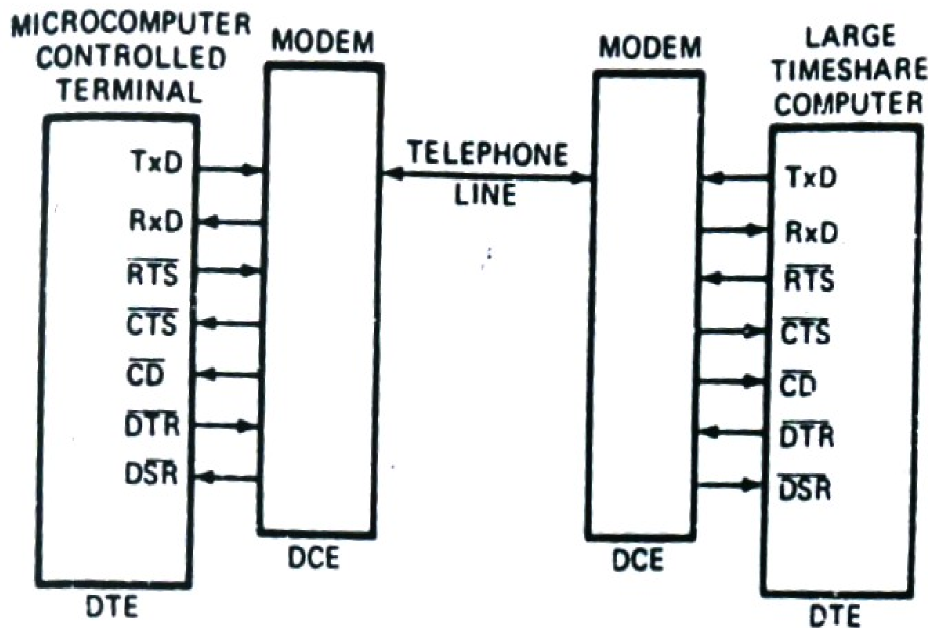
Command instruction controls the actual operations of the selected format like enable transmit/receive, error reset and modem control.



Command Instruction Format

2-5 RS 232 Serial data standard

The devices such as modems used to send data serially are known as data communication equipment DCE. The terminals that are sending or receiving the data are called as data terminal equipment TE. DCE connector is female and DTE connector is male. To send signals between DTE and DCE a standard was developed by Electronics Industries Association.



RS 232 describes 25 pins which are used to transfer signals and handshaking between connectors. It describes voltage levels, rise and fall times, bit rates etc.

2-5-1 Voltage Levels

The voltage levels for all RS232 signals are

A logic high is a voltage between – 3V and – 15V under load.

A logic high is a voltage between + 3V and + 15V under load.

2-5-2 PIN (signal) Description

PIN NUMBERS FOR 9 PINS	PIN NUMBERS FOR 25 PINS	COMMON NAME	RS-232C NAME	DESCRIPTION	SIGNAL DIRECTION ON DCE
	1		AA	PROTECTIVE GROUND	-
3	2	TXD	BA	TRANSMITTED DATA	IN
2	3	RXD	BB	RECEIVED DATA	OUT
7	4	$\overline{RTS}$	CA	REQUEST TO SEND	IN
8	5	$\overline{CTS}$	CB	CLEAR TO SEND	OUT
6	6	$\overline{DSR}$	CC	DATA SET READY	OUT
5	7	GND	AB	SIGNAL GROUND (COMMON RETURN)	-
1	8	$\overline{CD}$	CF	RECEIVED LINE SIGNAL DETECTOR	OUT
	9		-	(RESERVED FOR DATA SET TESTING)	-
	10		-	(RESERVED FOR DATA SET TESTING)	-
	11			UNASSIGNED	-
	12		SCF	SECONDARY RECEIVED LINE SIGNAL 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
4	20	$\overline{DTR}$	CD	DATA TERMINAL READY	IN
	21		CG	SIGNAL QUALITY DETECTOR	OUT
9	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	-

2-6 IEEE 488 BUS

General purpose interface bus GPIB also known as Hewlett-Packard interface bus or IEEE 488 bus.

The standard describes 3 types of devices connected on GPIB.

1) Listener: This can receive data from other instruments or controller.

   Example: Printers, display device etc.

2) Talker: This can send data to other instruments.

   Example: Tape recorder, Digital voltmeter and other measuring equipment.

3) Controller: This determines who talks or who listens on the bus.

3. Direct Memory Access (DMA) data transfer

Structure

3-1 Introduction
3-2 DMA Block diagram
    3-2-1 Mode Set Register
    3-2-2 Status register
    3-2-3 Priority Resolver
3-3 Pin Configuration of 8257

Objectives

• Discuss advantages of DMA internal

• Discuss about internal block diagram of DMA

• Discuss pin configuration of 8257

3-1 Introduction

In this mode device may transfer data directly to/from memory without any interference from the CPU. It is fastest among all the modes of data transfer.

The DMA controller temporally borrows the address bus, data bus, and control bus from the microprocessor and transfers the data bytes directly from the disk controller to a series of memory locations. Because the data transfer is handled totally in hardware, it is much faster than it would be if done by program instructions.

A DMA controller can also transfer data from memory to a port. Some DMA devices even can do memory-to-memory transfers to implement fast block transfers.

When the system is first turned on, the buses are connected from the microprocessor to system memory and peripherals.

We initialize all the programmable devices in the system and go on executing our program untills we need, for example, to read a file off a disk.

To read a disk file we send a series of commands to the smart disk controller device, telling it to find and read the desired block of data from the disk.

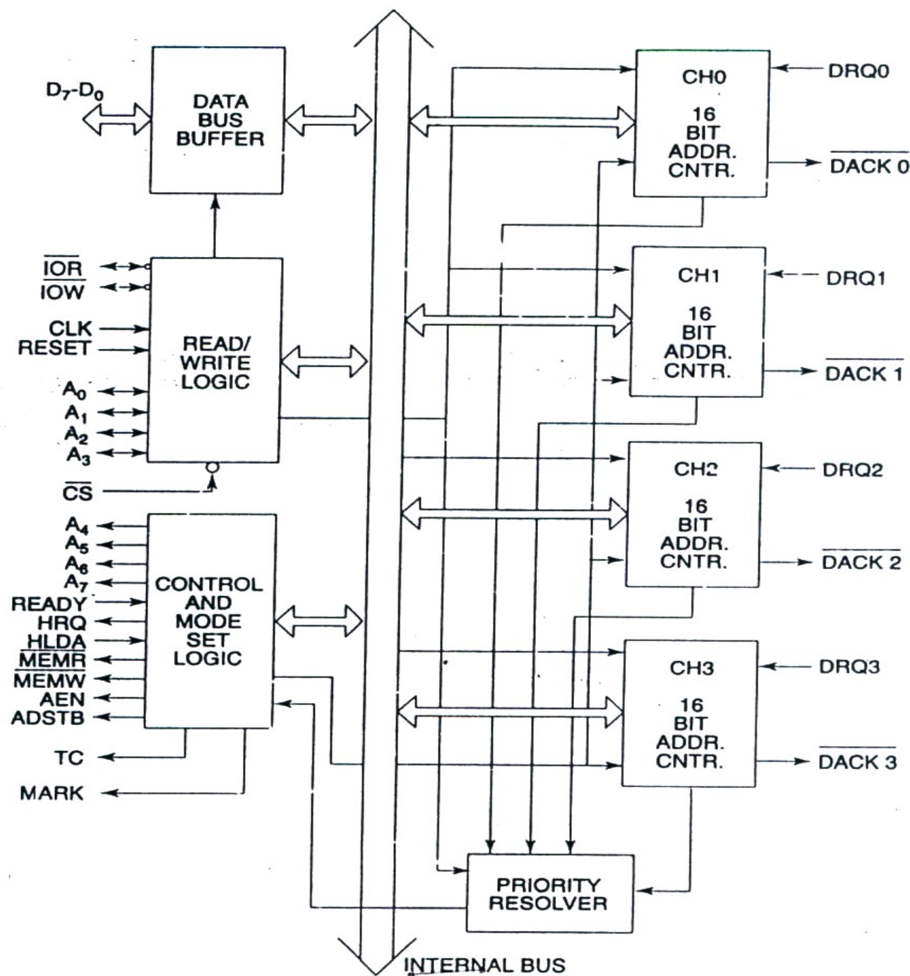
3-2 DMA Block diagram

When the controller needs to send the first byte of data from the disk block, it sends a DMA request, DREQ, signal to the DMA controller. If that input (channel) of the DMA controller is unmasked, the DMA controller will send a hold-request, HRQ, signal to the microprocessor HOLD input.

The microprocessor will respond to this input 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 will send out a control signal which throws the three bus switches down to the DMA position. This disconnects the processor from the buses and connects the DMA controller to the buses.

When the DMA controller gets control of the buses, it sends out the memory address where the first byte of data from the disk controller is to be written. Next the DMA controller sends a DMA-acknowledge, DACKO, signal to the disk controller device to tell it to get ready to output the byte. Finally, the DMA controller asserts both the MEMW and the IOR lines on the control bus.

Asserting the MEMW signal enables the addressed memory to accept data written to it. Asserting the IOR signal enables the disk controller to output the byte of data from the disk controller of output the byte of data from the disk on the data bus. The byte of data then 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 its hold request signal to the processor and releases the buses block which will be accessed by the device is the loaded in DMA address register of the channel.

Terminal count register: Each of the 4 DMA channels of 8257 has one terminal count register. It is a 16-bit register used to hold the number of DMA cycles after which data transfer stops. After each DMA cycle count decreases by one.

3-2-1 Mode Set Register

It is used to enable the DMA channels individually and also to set the various modes of operation. DMA channel should not be enabled till the DMA address register and terminal count regular contain valid

information otherwise an unwanted DMA request may initiate DMA cycle.

The format of mode set register is given below

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
↑ Auto load	↑ TC Stop	↑ Extende d write	↑ Rotating Priority	↑ Ch3	↑ Ch2	↑ Ch1	↑ Ch0

If TC STOP bit is set, the selected channel is after terminal count is reached. It it is 0 the channel is not disabled the count reaches 0 and further requests are allowed on the same channel.

If auto load bit is set, channel 2 is enabled for the repeat block chaining operations without immediate software intervention between two successive blocks.

If auto load bit is set duration of $\overline{\text{MEMW}}$ and $\overline{\text{IOW}}$ are extended.

3-2-2 Status register

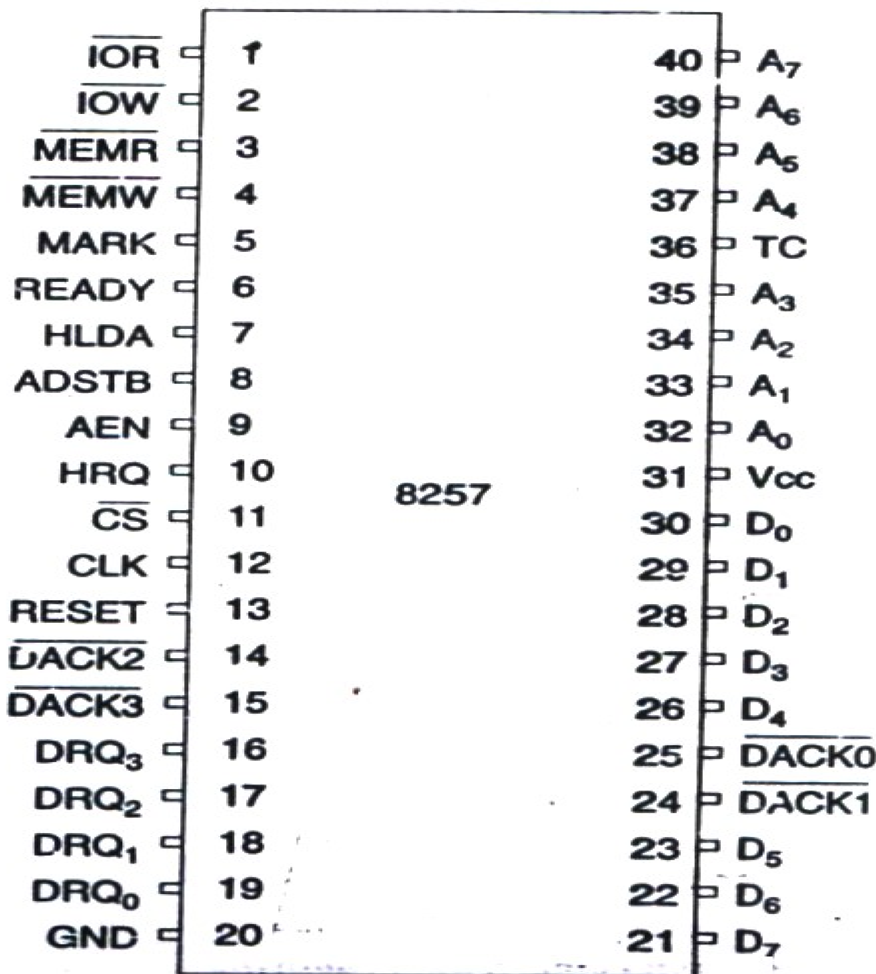
The lower order 4 bits of this register contain the terminal count status for the four individual channels. If any one these bits is set it indicates that specific channel has reached the terminal count.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
			↑ Update flag	↑ TC status of Ch3	↑ Ch2	↑ Ch1	↑ Ch0

3-2-3 Priority Resolver

The priority resolver resolves the priority of the 4 DMA channels depending upon whether normal priority or rotating priority is programmed.

3-3 Pin Configuration of 8257



Pin Diagram of 8257

DRQ0-DRQ3: These are used by 4 peripherals to request DMA service.

DACK0-DACK3: These are DMA acknowledge lines informing the requesting peripheral that requested has been accepted.

D0-D7: These are data lines used to interface the system bus with the internal data bus of 8257. They carry command words to 8257 and status word from 8257 in slave mode. In master mode they are used to send higher byte of generated address to the latched.

IOR: This pin is active low bi-directional tristate input line. It acts as input line used by CPU to read internal registers of 8257 in slave mode. It acts as output line in master mode and is used to read data from peripheral.

LOW: This pin is active low bi-directional tristate input line. It acts as input line used by CPU to load contents of data bus into internal registers of 8257 in slave mode. It acts as output line in master mode and is used to load data to a peripheral.

CLK: Provides clock frequency for providing system timings.

RESET: It disables all DMA channels by clearing mode register and tristates all the control lines.

A0-A3: In slave mode they act as input which selects one of the registers to be read or write. In master mode they are 4 least significant memory address output lines generated by 8257.

CS: This is active low chip select line that enables read and write operations.

A4-A7: This is the higher nibble of the lower byte address generated by 8257 during master mode of DMA operation.

READY: This is active high asynchronous input used to stretch memory read and write cycles.

HRQ: In non cascade 8257 systems this is connected with HOLD pin of CPU. In cascade mode this pin of a slave is connected with a DRQ input line of the master 8257 while that of the master is connected with HOLD input of the CPU.

HLDA: CPU drives this input to DMA controller high while granting the bus to the device. A high on this pin indicates that the bus has been granted to the requesting peripheral by CPU.

MEMR: This active low memory read output is used to read data from the addressed memory locations during DMA read cycles.

MEMW: This active low three state output is used to write data to the addressed memory location during DMA write operation.

ADSTB: This output from 8257 strobes the higher byte of the memory addressed generated by the DMA controller into the latches.

AEN: This is used to disable the system data bus and control bus driven by the CPU.

TC: Terminal count output indicates tot he currently selected peripheral that the present DMA cycle is the last for the previously programmed data block.

MARK: It indicates to the selected peripheral that the current DMA cycle is the 128th cycle since the previous mark output.

V_{cc}: This is +5V supply pin.

GND: This negative line for supply.

4. Summary

In this unit, we have presented a detailed account of the functioning of some of the important Intel peripherals. With recent advances in the field, the Intel family of the peripherals has been continuously added with a number of new peripherals. Those, which are frequently used in the industrial and general systems, are discussed in this unit. This unit started with discussion on the programmable interface timer 8254. The necessary functional details of 8254 have been discussed along with an interfacing. Further peripheral like USART, RS232, and DMA have been discussed along with architecture, signal descriptions, interfacing and programming examples.

5. Test Yourself

1. Draw and discuss internal architecture of 8254.

2. Draw and discuss the different modes of operation of 8254.

3. Draw and discuss internal architecture of USART.

4. Explain the mode instruction control word format of 8251.

5. Draw and discuss the asynchronous mode transmitter and receiver data formats of 8251.

6. Draw and discuss the status word format of 8251.

7. Draw and discuss pin configuration of RS232.

8. what is the advantage of DMA controlled data transfer over interrupt driven or program controlled data transfer?

9. Draw and discuss internal architecture of 8257.

10. Draw and discuss the status register of 8257.