UNIT 1 THE 8086 MICROPROCESSOR

1. Introduction to 8086:

The word microprocessor comes from the combination micro and processor.

- Processor means a device that processes numbers, specifically binary numbers, 0's and 1's.
- Micro is a new addition.
- In the late 1960's, processors were built using discrete elements.
- These devices performed the required operation, but were too large and too slow.
- In the early 1970's the microchip was invented. All of the components that made up the processor were now placed on a single piece of silicon. The size became several thousand times smaller and the speed became several hundred times faster.
- The "Micro" Processor was born.

A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to instructions, and provides result as output.

History of Microprocessors:

Table presents the various microprocessor manufactured by Intel from 1971 to current date.

Processor	No. of bits	Clock speed (Hz)	Year of introduction
4004	4	740K	1971
8008	8	500K	1972
8080	8	2M	1974
8085	8	3M	1976
8086	16	5, 8 or 10M	1978
8088	16	5, 8 or 10M	1979
80186	16	6M	1982
80286	16	8M	1982
80386	32	16 to 33M	1986
80486	32	16 to 100M	1989
Pentium	32	66M	1993
Pentium II	32	233 to 500M	1997
Pentium III	32	500M to 1.4G	1999
Pentium IV	32	1.3 to 3.8G	2000

Dual core	32	1.2 to 3 G	2006
Core 2 Duo	64	1.2 to 3G	2006
i3, i5 and i7	64	2.4G to 3.6G	2010

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

Features:

- It is 16-bit microprocessor
- It has a 16-bit data bus, so it can read data from or write data to memory and ports either 16-bit or 8-bit at a time.
- It has 20 bit address bus and can access up to 2^{20} memory locations (1 MB).
- It can support up to 64K I/O ports
- It provides 14, 16-bit registers
- It has multiplexed address and data bus AD_0-AD_{15} & A_{16}-A_{19}
- It requires single phase clock with 33% duty cycle to provide internal timing.
- Pre-fetches up to 6 instruction bytes from memory and queues them in order to speed up the processing.
- 8086 supports 2 modes of operation
    - Minimum mode
    - Maximum mode

2. Microprocessor architecture:

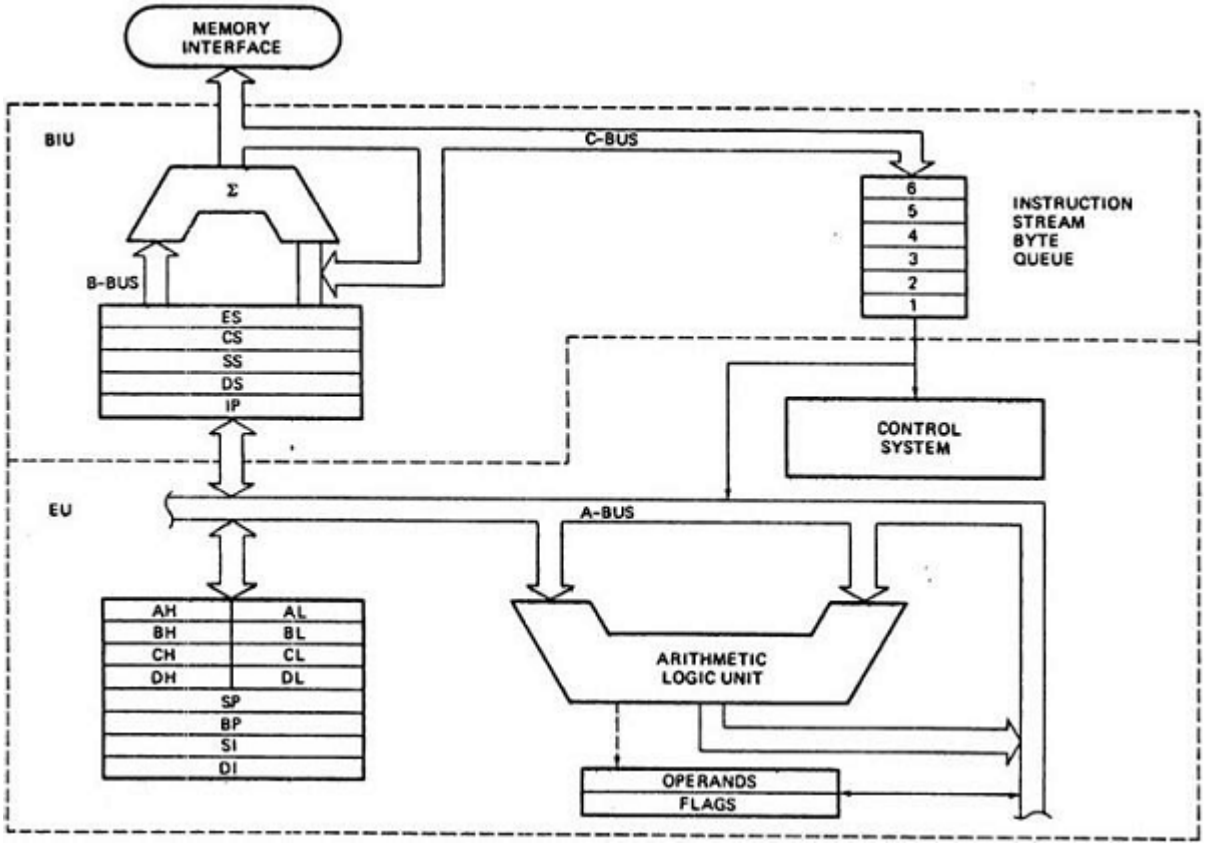
8086 Microprocessor is divided into two functional units, i.e., **EU** (Execution Unit) and **BIU** (Bus Interface Unit).

EU (Execution Unit):

Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.

Let us now discuss the functional parts of 8086 microprocessors.

- **ALU**: It handles all arithmetic and logical operations, like +, −, ×, /, OR, AND, NOT operations.
- **Flag Register**: It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups − Conditional Flags and Control Flags.



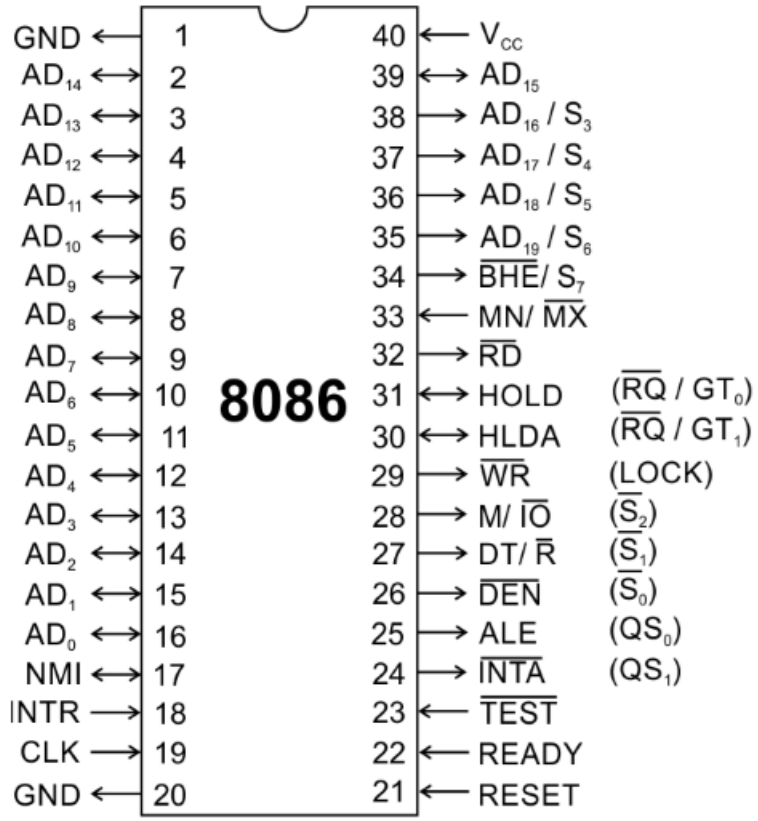
Architecture of IC 8086

Conditional Flags: It represents the result of the last arithmetic or logical instruction executed.

Following is the list of conditional flags:

- **Carry flag** – This flag indicates an overflow condition for arithmetic operations.
- **Auxiliary flag** – When an operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag. The processor uses this flag to perform binary to BCD conversion.
- **Parity flag** – This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.
- **Zero flag** – This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.
- **Sign flag** – This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.
- **Overflow flag** – This flag represents the result when the system capacity is exceeded.

Control Flags: Control flags controls the operations of the execution unit.



Pin diagram of IC 8086

Following is the list of control flags

- **Trap flag** – It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.
- **Interrupt flag** – It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.
- **Direction flag** – It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

General purpose registers:

There are 8 general purpose registers, i.e., AH, AL, BH, BL, CH, CL, DH, and DL. These

registers can be used individually to store 8-bit data and can be used in pairs to store 16bit data. The valid register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. It is referred to the AX, BX, CX, and DX respectively.

- **AX register** – It is also known as accumulator register. It is used to store operands for arithmetic operations.
- **BX register** – It is used as a base register. It is used to store the starting base address of the memory area within the data segment.
- **CX register** – It is referred to as counter. It is used in loop instruction to store the loop counter.
- **DX register** – This register is used to hold I/O port address for I/O instruction.

Stack pointer register

It is a 16-bit register, which holds the address from the start of the segment to the memory location, where a word was most recently stored on the stack.

BIU (Bus Interface Unit):

BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory. EU has no direction connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus.

It has the following functional parts –

- **Instruction queue** – BIU contains the instruction queue. BIU gets upto 6 bytes of next instructions and stores them in the instruction queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.
  Fetching the next instruction while the current instruction executes is called **pipelining**.
- **Segment register** – BIU has 4 segment buses, i.e. CS, DS, SS& ES. It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations. It also contains 1 pointer register IP, which holds the address of the next instruction to executed by the EU.
  - **CS** – It stands for Code Segment. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.
  - **DS** – It stands for Data Segment. It consists of data used by the program and is accessed in the data segment by an offset address or the content of other register that holds the offset address.
  - **SS** – It stands for Stack Segment. It handles memory to store data and addresses during execution.
  - **ES** – It stands for Extra Segment. ES is additional data segment, which is used by the string to hold the extra destination data.
- **Instruction pointer** – It is a 16-bit register used to hold the address of the next instruction to be executed.

Power supply and frequency signals: It uses 5V DC supply at V_{CC} pin 40, and uses

ground at V_{SS} pin 1 and 20 for its operation.

Clock signal: Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

Address/data bus: AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

Address/status bus: A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

- **S7/BHE:** BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.
- **Read($\overline{RD}$):** It is available at pin 32 and is used to read signal for Read operation.
- **Ready :** It is available at pin 22. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.
- **RESET:** It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.
- **INTR:** It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.
- **NMI :** It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.
- $\overline{TEST}$: This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.
- **MN/$\overline{MX}$:** It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-aversa.
- **INTA:** It is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.
- **ALE:** It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.
- **DEN:** It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.
- **DT/R:** It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transreceiver. When it is high, data is transmitted out and vice-a-versa.

- **M/IO:** This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.
- **WR:** It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.
- **HLDA:** It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.
- **HOLD:** This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.
- **QS_1 and QS_0:** These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue.
  Their conditions are shown in the following table :

S_0	S_1	atus
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

S_0, S_1, S_2 are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status:

S2	S1	S0	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

LOCK

When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

RQ/GT_1 and RQ/GT_0

These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT_0 has a higher priority than RQ/GT_1.

Addressing modes of 8086:

- Addressing mode indicates a way of locating data or operands.
- The addressing modes describe the types of operands and the way they are accessed for executing an instruction.
- According to the flow of instruction execution, the instructions may be categorized as
  i) Sequential control flow instructions and
  ii) Control transfer instructions

Sequential control flow instructions are the instructions, which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logic, data transfer and processor control instructions are sequential control flow instructions.

The **control transfer instructions**, on the other hand, transfer control to some predefined address or the address somehow specified in the instruction, after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential control transfer instructions are:

- **Immediate:** In this type of addressing, immediate data is a part of instruction and appears in the form of successive byte or bytes.
  Ex: MOV AX, 0005H
  In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.
- **Direct:** In the direct addressing mode a 16-bit memory address (offset) is directly specified in the instruction as a part of it.
  Ex: MOV AX, [5000H]
  Here, data resides in a memory location in the data segment, whose effective address may be completed using 5000H as the offset address and content of DS as segment address. The effective address here, is 10H * DS + 5000H.
- **Register:** In register addressing mode, the data is stored in a register and is referred using the particular register. All the registers, except IP, may be used in this mode.
  Ex: MOV BX, AX
- **Register Indirect:** Sometimes, the address of the memory location, which contains data or operand, is determined in an indirect way, using the offset register. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI register. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above

registers in the default data segment.
Ex: MOV AX, [BX]
Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H * DS+[BX].

- **Indexed:** In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers, SI and DI respectively. This is a special case of register indirect addressing mode.
  Ex: MOV AX, [SI]
  Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10*DS+[SI].
- **Register Relative:** In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment.
  Ex: MOV AX, 50H[BX]
  Here, the effective address is given as 10H *DS+50H+[BX]
- **Based Indexed:** The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.
  Ex: MOV AX, [BX][SI]
  Here, BX is the base register and SI is the index register the effective address is computed as 10H * DS + [BX] + [SI].
- **Relative Based Indexed:** The effective address is formed by adding an 8 or 16-bit displacement with the sum of the contents of any one of the base register (BX or BP) and any one of the index register, in a default segment.
  Ex: MOV AX, 50H [BX] [SI]
  Here, 50H is an immediate displacement, BX is base register and SI is an index register the effective address of data is computed as
  10H * DS + [BX] + [SI] + 50H

For control transfer instructions, the addressing modes depend upon whether the destination is within the same segment or different one. It also depends upon the method of passing the destination address to the processor.

Basically, there are two addressing modes for the control transfer instructions, **inter-segment** addressing and **intra-segment** addressing modes.

3. Instruction set and assembler directives:

The 8086 microprocessor supports 8 types of instructions –

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions

- Iteration Control Instructions
- Interrupt Instructions

Let us now discuss these instruction sets in detail.

Data Transfer Instructions:

These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group –

Instruction to transfer a word:

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** – Used to put a word at the top of the stack.
- **POP** – Used to get a word from the top of the stack to the provided location.
- **PUSHA** – Used to put all the registers into the stack.
- **POPA** – Used to get words from the stack to all registers.
- **XCHG** – Used to exchange the data from two locations.
- **XLAT** – Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- **IN** – Used to read a byte or word from the provided port to the accumulator.
- **OUT** – Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

- **LEA** – Used to load the address of operand into the provided register.
- **LDS** – Used to load DS register and other provided register from the memory
- **LES** – Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

- **LAHF** – Used to load AH with the low byte of the flag register.
- **SAHF** – Used to store AH register to low byte of the flag register.
- **PUSHF** – Used to copy the flag register at the top of the stack.
- **POPF** – Used to copy a word at the top of the stack to the flag register.

Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Following is the list of instructions under this group –

Instructions to perform addition

- **ADD** – Used to add the provided byte to byte/word to word.
- **ADC** – Used to add with carry.
- **INC** – Used to increment the provided byte/word by 1.
- **AAA** – Used to adjust ASCII after addition.
- **DAA** – Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

- **SUB** – Used to subtract the byte from byte/word from word.
- **SBB** – Used to perform subtraction with borrow.
- **DEC** – Used to decrement the provided byte/word by 1.
- **NPG** – Used to negate each bit of the provided byte/word and add 1/2's

complement.

- **CMP** – Used to compare 2 provided byte/word.
- **AAS** – Used to adjust ASCII codes after subtraction.
- **DAS** – Used to adjust decimal after subtraction.

Instruction to perform multiplication

- **MUL** – Used to multiply unsigned byte by byte/word by word.
- **IMUL** – Used to multiply signed byte by byte/word by word.
- **AAM** – Used to adjust ASCII codes after multiplication.

Instructions to perform division

- **DIV** – Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** – Used to divide the signed word by byte or signed double word by word.
- **AAD** – Used to adjust ASCII codes after division.
- **CBW** – Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- **CWD** – Used to fill the upper word of the double word with the sign bit of the lower word.

Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Following is the list of instructions under this group –

Instructions to perform logical operation

- **NOT** – Used to invert each bit of a byte or word.
- **AND** – Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** – Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** – Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** – Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

- **SHL/SAL** – Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** – Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Instructions to perform rotate operations

- **ROL** – Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **ROR** – Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- **RCR** – Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** – Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group –

- **REP** – Used to repeat the given instruction till CX ≠ 0.
- **REPE/REPZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **REPNE/REPNZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **MOVS/MOVSB/MOVSW** – Used to move the byte/word from one string to another.
- **COMS/COMPSB/COMPSW** – Used to compare two string bytes/words.
- **INS/INSB/INSW** – Used as an input string/byte/word from the I/O port to the provided memory location.
- **OUTS/OUTSB/OUTSW** – Used as an output string/byte/word from the provided memory location to the I/O port.
- **SCAS/SCASB/SCASW** – Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- **LODS/LODSB/LODSW** – Used to store the string byte into AL or string word into AX.

Program Execution Transfer Instructions (Branch and Loop Instructions)

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions –

Instructions to transfer the instruction during an execution without any condition –

- **CALL** – Used to call a procedure and save their return address to the stack.
- **RET** – Used to return from the procedure to the main program.
- **JMP** – Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions –

- **JA/JNBE** – Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** – Used to jump if above/not below instruction satisfies.
- **JBE/JNA** – Used to jump if below/equal/ not above instruction satisfies.
- **JC** – Used to jump if carry flag CF = 1
- **JE/JZ** – Used to jump if equal/zero flag ZF = 1
- **JG/JNLE** – Used to jump if greater/not less than/equal instruction satisfies.
- **JGE/JNL** – Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** – Used to jump if less than/not greater than/equal instruction satisfies.
- **JLE/JNG** – Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNC** – Used to jump if no carry flag (CF = 0)
- **JNE/JNZ** – Used to jump if not equal/zero flag ZF = 0
- **JNO** – Used to jump if no overflow flag OF = 0
- **JNP/JPO** – Used to jump if not parity/parity odd PF = 0
- **JNS** – Used to jump if not sign SF = 0
- **JO** – Used to jump if overflow flag OF = 1
- **JP/JPE** – Used to jump if parity/parity even PF = 1
- **JS** – Used to jump if sign flag SF = 1

Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values.

Following are the instructions under this group –

- **STC** – Used to set carry flag CF to 1
- **CLC** – Used to clear/reset carry flag CF to 0
- **CMC** – Used to put complement at the state of carry flag CF.
- **STD** – Used to set the direction flag DF to 1
- **CLD** – Used to clear/reset the direction flag DF to 0
- **STI** – Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** – Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group –

- **LOOP** – Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** – Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** – Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** – Used to jump to the provided address if CX = 0

Interrupt Instructions

These instructions are used to call the interrupt during program execution.

- **INT** – Used to interrupt the program during execution and calling service specified.
- **INTO** – Used to interrupt the program during execution if OF = 1
- **IRET** – Used to return from interrupt service to the main program

4. Assembly language programming:

The assembly programming language is a low-level language which is developed by using mnemonics. The microcontroller or microprocessor can understand only the binary language like 0's or 1's therefore the assembler convert the assembly language to binary language and store it the memory to perform the tasks. Before writing the program the embedded designers must have sufficient knowledge on particular hardware of the controller or processor, so first we required to know hardware of 8086 processor.

The assembly language programming 8086 has some rules such as

- The assembly level programming 8086 code must be written in upper case letters
- The labels must be followed by a colon, for example: label:
- All labels and symbols must begin with a letter
- All comments are typed in lower case
- The last line of the program must be ended with the END directive

8086 processors have two other instructions to access the data, such as WORD PTR – for word (two bytes), BYTE PTR – for byte.

Op-code: A single instruction is called as an op-code that can be executed by the CPU. Here the 'MOV' instruction is called as an op-code.

Operands: A single piece data are called operands that can be operated by the op-code. Example, subtraction operation is performed by the operands that are subtracted by the operand.

Syntax: SUB b, c

8086 microprocessor assembly language programs (Example):

a. Write a Program For Read a Character From The Keyboard:

```
MOV ah, 1h          //keyboard input subprogram
  INT 21h       // character input
  // character is stored in al
  MOV c, al       //copy character from alto c
```

b. Write a Program For Reading and Displaying a Character

```
MOV ah, 1h       // keyboard input subprogram
  INT 21h       //read character into al
  MOV dl, al       //copy character to dl
  MOV ah, 2h       //character output subprogram
  INT 21h       // display character in dl
```

c. Write a Program Using General Purpose Registers

```
ORG 100h
```

```
MOV AL, VAR1        // check value of VAR1 by moving it to the AL.
LEA BX, VAR1        //get address of VAR1 in BX.
MOV BYTE PTR [BX], 44h // modify the contents of VAR1.
MOV AL, VAR1        //check value of VAR1 by moving it to the AL.
RET
VAR1 DB 22h
END
```

d. Write a Program For Displaying The String Using Library Functions

```
include emu8086.inc    //Macro declaration
ORG 100h
PRINT 'Hello World!'
GOTOXY 10, 5
PUTC 65          // 65 – is an ASCII code for 'A'
PUTC 'B'
RET              //return to the operating system.
END              //directive to stop the compiler.
```

e. Arithmetic and Logic Instructions

The 8086 processes of arithmetic and logic unit has separated into three groups such as addition, division, and increment operation. Most Arithmetic and Logic Instructions affect the processor status register.

The assembly language programming 8086 mnemonics are in the form of op-code, such as MOV, MUL, JMP, and so on, which are used to perform the operations.

Assembly language programming 8086 examples:

Addition

```
ORG0000h
MOV DX, #07H    // move the value 7 to the register AX//
MOV AX, #09H    // move the value 9 to accumulator AX//
Add AX, 00H     // add CX value with R0 value and stores the result in AX//
END
```

Multiplication

```
ORG0000h
MOV DX, #04H    // move the value 4 to the register DX//
MOV AX, #08H    // move the value 8 to accumulator AX//
MUL AX, 06H     // Multiplied result is stored in the Accumulator AX //
END
```

Subtraction

```
ORG 0000h
MOV DX, #02H    // move the value 2 to register DX//
MOV AX, #08H    // move the value 8 to accumulator AX//
SUBB AX, 09H    // Result value is stored in the Accumulator A X//
END
```

Division

```
ORG 0000h
MOV DX, #08H    // move the value 3 to register DX//
MOV AX, #19H    // move the value 5 to accumulator AX//
DIV AX, 08H     // final value is stored in the Accumulator AX //
END
```

Therefore, this is all about Assembly Level Programming 8086, 8086 Processor Architecture simple example programs for 8086 processors, Arithmetic and Logic Instructions. Furthermore, any queries regarding this article or electronics projects, you can contact us by commenting in the comment section below.

5. Modular Programming:

Many programs are too large to be developed by one programmer. Such programs are developed by team of programmers. They divide a large program into smaller modules. Then each Modular Programming is individually written, tested and debugged. When all modules are tested 'OK', they are linked together to form a large functioning program.

(a) Assembling Process:

As mentioned earlier, assembler translates a source file that was created using the editor into machine language such as binary or object code. The assembler reads the source file of our program from the disk where we saved it after editing. An assembler usually reads our source file more than once.

The assembler generates two files on the floppy or hard during these two passes. The first file is called the object file. The object file contains the binary codes for the instructions and information about the addresses of the instructions. The second file generated by the assembler is called assembler list file. This file contains the assembly language statements, the binary code for each instruction, and the offset for each instruction.

In the first pass, the assembler performs the following operations

- Reading the source program instructions.
- Creating a symbol table in which all symbols used in the program, together with their attributes, are stored.
- Replacing all mnemonic codes by their binary codes.
- Detecting any syntax errors in the source program.
- Assigning relative addresses to instructions and data.

On a second pass through the source program, the assembler extracts the symbol from the operand field and searches for it in the symbol table. If the symbol does not appear in the table, the corresponding statement is obviously erroneous. If the symbol does appear in the table, the symbol is replaced by its address or value.

(b) Linking Process:

A linker is a program used to join together several object files into one large object file. When writing large programs, it is usually much more efficient to divide the large program into smaller modules. Each Modular Programming can be individually written, tested and debugged. When all the Modular Programming work, they can be linked together to form a large functioning program.

The linker produces a link file which contains the binary codes for all the combined modules. The linker also produces a link map which contains the address information about the link files. The linker, however, does not assign absolute addresses to the program, it only assigns relative addresses starting from zero. This form of the program is said to be relocatable, because it can be put anywhere in memory to be run.

(c) Debugging Process:

A debugger is a program which allows us to load our object code program into system memory, execute the program, and debug it.

- The debugger allows us to look at the contents of registers and memory locations after our program runs.
- It allows us to change the contents of register and memory locations and rerun the program.
- Some debugger allows us to stop execution after each instruction so we can check or alter memory and register contents.
- A debugger also allows us to set a breakout at any point in our program. When we run a program, the system will execute instructions up to this break point and stop. We can then examine register and memory contents to see if the results are correct at that point. If the results are correct, we can move the break point to a later point in our program. If results are not correct, we can check the program up to that point to find out why they are not correct.

In short, debugger tools can help us to isolate problems in our program.

6. Linking and Relocation:

The linking program links the different object modules of a source program and function library routines to generate an integrated executable code of the source

program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by the MASM. The linker program is invoked using the following options.

C> LINK or

C>LINK MS.OBJ

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first object may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The output of the link program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file.

In the advanced version of the MASM, the complete procedure of assembling and linking is combined under a single menu invokable compile function. The recent versions of MASM have much more sophisticated and user-friendly facilities and options. A linker links the machine codes with the other required assembled codes. Linking is necessary because of the number of codes to be linked for the final binary file.

The linked file in binary for run on a computer is commonly known as executable file or simply '.exe.' file. After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory.

The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant. The DOS linking program links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by the MASM. The linked file in binary for run on a computer is commonly known as executable file or simply '.exe.' file. After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory.

The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant. The loader is a part of the operating system and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at the different addresses during the run. The loader finds the appropriate start address. In a computer, the loader is used and it loads into a section of RAM the program that is ready to run. A program called locator reallocates the linked file and creates a file for permanent location of codes in a standard format.

Segment combination: In addition to the linker commands, the assembler provides a means of regulating the way segments in different object modules are organized by the linker.

Segments with same name are joined together by using the modifiers attached to the

SEGMENT directives. SEGMENT directive may have the form:

Segment name SEGMENT Combination-type where the combine-type indicates how the segment is to be located within the load module. Segments that have different names cannot be combined and segments with the same name but no combine-type will cause a linker error.

7. Stacks – Procedures – Macros:

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

The process of storing the data in the stack is called 'pushing into' the stack and the

reverse process of transferring the data back from the stack to the CPU register is known as

'popping off' the stack. The stack is essentially Last-In-First-Out (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first.

The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment. The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base address of the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

SS -> 5000H

SP -> 2050H

If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. The next 16 bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top 5204EH and the decremented contents of SP will be 204EH. This location will now be occupied by the recently pushed data. Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with

an initial value of FFFFH, it will be decremented by two whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.

After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure.

A Macro is a set of instructions grouped under a single unit. It is another method for implementing modular programming in the 8086 microprocessors (The first one was using Procedures).

The Macro is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the Macro is made.

A Macro can be defined in a program using the following assembler directives: MACRO (used after the name of Macro before starting the body of the Macro) and ENDM (at the end of the Macro). All the instructions that belong to the Macro lie within these two assembler directives. The following is the syntax for defining a Macro in the 8086 Microprocessor:

Macro in the 8086 Microprocessor:

```
Macro_name  MACRO  [ list of parameters ]
   Instruction 1
   Instruction 2
   - - - - - - - - - - -
   - - - - - - - - - - -
   - - - - - - - - - - -
   Instruction n
ENDM
```

And a call to Macro is made just by mentioning the name of the Macro:

```
Macro_name [ list of parameters]
```

It is optional to pass the parameters in the Macro. If you want to pass them to your macros, you can simply mention them all in the very first statement of the Macro just after the directive: MACRO.

The advantage of using Macro is that it avoids the overhead time involved in calling and returning (as in the procedures). Therefore, the execution of Macros is faster as compared to procedures. Another advantage is that there is no need for accessing stack or providing any separate memory to it for storing and returning the address

locations while shifting the processor controls in the program.

But it should be noted that every time you call a macro, the assembler of the microprocessor places the entire set of Macro instructions in the mainline program from where the call to Macro is being made. This is known as Macro expansion. Due to this, the program code (which uses Macros) takes more memory space than the code which uses procedures for implementing the same task using the same set of instructions.

Hence, it is better to use Macros where we have small instruction sets containing less number of instructions to execute.

Differences between Procedures and Macros:

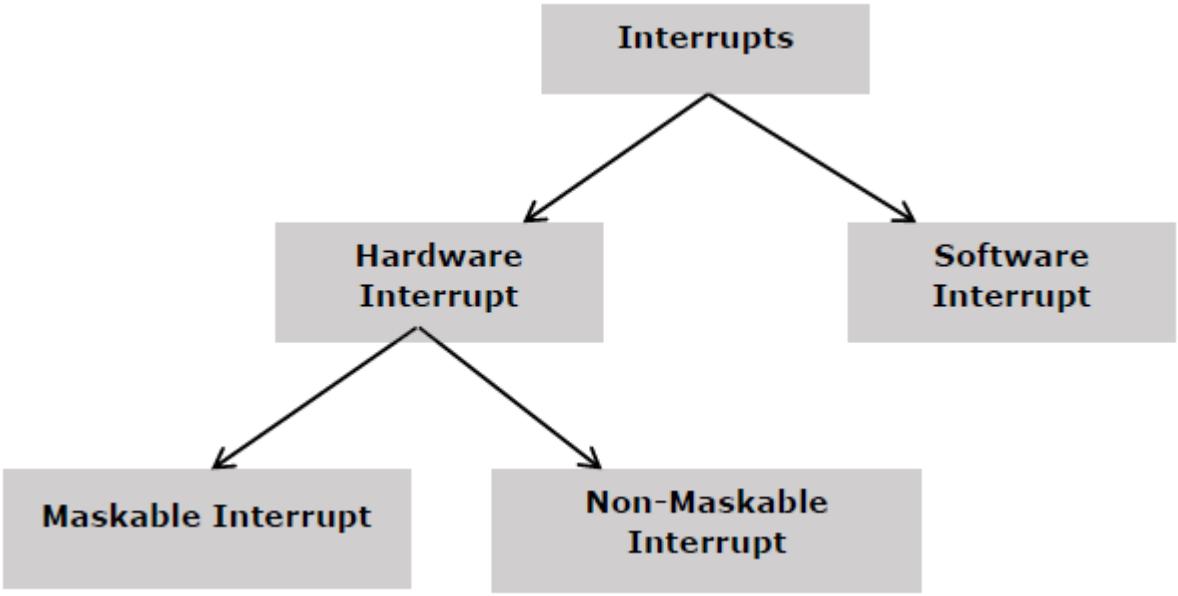
Characteristic	Procedure	Macro
Number of Instructions that can be effectively handled by the microprocessor	It is better to use Procedures for a set of a large number of instructions. Hence, it is optimal to use Procedures when the number of instructions is more than 10.	Macros are useful over Procedures when the number of instructions in the set is less. Therefore, when the subprogram contains less than 10 instructions, Macros are more efficient to use in such cases.
Assembler Directives used	The assembler directive - PROC is used to define a Procedure. And the assembler directive - ENDP is used to indicate that the body of the procedure has ended.	The assembler directive- MACRO is used to define a Macro, And to indicate that the body of the procedure has ended, the assembler directive- ENDM is used.
Execution Process	Every time a procedure is called, the CALL and RET instructions are required for shifting the control of instruction execution.	Every time a Macro is called, the assembler of the microprocessor places the entire set of instructions of the Macros in the mainline program form where the call to the macro is made.
Execution Time	The Procedures execute slower than the Macros because every time a procedure is called, it is	The execution of macros is faster as compared to procedures because there is no need to integrate or link the macros with

	necessary to integrate and link it with the calling program and this takes time.	the calling program. It is simply loaded into the main memory every time it is called.
Overhead time	Overhead time occurs while calling the procedure and returning the control to the calling program.	Overhead time is avoided as calling and returning does not take place.
Amount of memory required	The Procedures require less amount of memory than the Macros because a Procedure is written and loaded into the main memory only once, and is linked to the calling program when called.	The Macros require a large amount of memory because it is loaded into the main memory every time it is called.
Number of times machine code generated	The machine code (containing the instructions within the Procedure) is generated only once when the procedure is defined.	The machine code (containing the instructions within the Macros) is generated every time the macro is called.
Passing of parameters	In procedures, we cannot pass the parameter to id directly. However, the values can be passed to it using registers and also via stack.	The macros are capable of handling parameters within their definition and we can pass them in the statement which calls the macro.

8. Interrupts and interrupt service routines:

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

The following image shows the types of interrupts we have in a 8086 microprocessor :



Hardware Interrupts:

Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor.

The 8086 has two hardware interrupt pins, i.e. NMI and INTR. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is INTA called interrupt acknowledge.

NMI:

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR)and it is of type 2 interrupt.

When this interrupt is activated, these actions take place –

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.
- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
- IP is loaded from the contents of the word location 00008H.
- CS is loaded from the contents of the next word location 0000AH.
- Interrupt flag and trap flag are reset to 0.

INTR:

The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.

The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends '0' on INTA pin twice. The first '0' means INTA informs the external device to get ready and during the second '0' the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor –

- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value, CS value of the return address and IP value of the return address are pushed on to the stack.
- IP value is loaded from the contents of word location X × 4
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

Software Interrupts:

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes –

INT- Interrupt instruction with type number:

It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

Its execution includes the following steps –

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 'type number' × 4
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on. The first five pointers are dedicated interrupt pointers. i.e. –

- **TYPE 0** interrupt represents division by zero situation.
- **TYPE 1** interrupt represents single-step execution during the debugging of a program.
- **TYPE 2** interrupt represents non-maskable NMI interrupt.
- **TYPE 3** interrupt represents break-point interrupt.
- **TYPE 4** interrupt represents overflow interrupt.

The interrupts from Type 5 to Type 31 are reserved for other advanced microprocessors, and interrupts from 32 to Type 255 are available for hardware and software interrupts.

INT 3-Break Point Interrupt Instruction:

It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it stops the normal execution of program and follows the break-point procedure.

Its execution includes the following steps –

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 3×4 = 0000CH
- CS is loaded from the contents of the next word location.

- Interrupt Flag and Trap Flag are reset to 0

INTO - Interrupt on overflow instruction:

It is a 1-byte instruction and their mnemonic **INTO**. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next instruction.

Its execution includes the following steps –

- Flag register values are pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of word location 4×4 = 00010H
- CS is loaded from the contents of the next word location.
- Interrupt flag and Trap flag are reset to 0

9. **Byte and String Manipulation:**

String is s series of data byte or word available in memory at consecutive locations. It is either referred as byte string or word string. Their memory is always allocated in a sequential order. Instructions used to manipulate strings are called string manipulation instructions.

The String manipulation instructions are as follows.

Opcode	Operand	Description
REP	Instruction	Used to repeat the given instruction till CX ≠ 0.
REPE/REPZ	Instruction	Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
REPNE/REPNZ	Instruction	Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
MOVS/MOVSB/MOVSW	----	Used to move the byte/word from one string to another.
COMS/COMPSB/COMPSW	----	Used to compare two string bytes/words.
INS/INSB/INSW	----	Used as an input string/byte/word from the I/O

Opcode	Operand	Description
		port to the provided memory location.
OUTS/OUTSB/OUTSW	----	Used as an output string/byte/word from the provided memory location to the I/O port.
SCAS/SCASB/SCASW	----	Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
LODS/LODSB/LODSW	----	Used to store the string byte into AL or string word into AX.

The Strings can be defined as the collection of ASCII characters. Each ASCII character is of one byte, and each byte of a String is stored at successive memory locations.

Source Index (SI), Destination Index (DI) and the general-purpose register-CX (which functions as a counter) are the registers involved in performing string operations.

The following are the **various string manipulation instructions in the 8086 microprocessor**:

1) REP

Stands for **'Repeat'**. This instruction repeats the given instruction(s) till **CX** does not becomes zero, i.e. **CX!=0**

```
REP  instruction_to_be_repeated
```

2) REPE

This instruction repeats the given instruction(s) till **CX** remains zero, i.e. **CX=0**

```
REPE  instruction_to_be_repeated
```

3) REPZ

It repeats the given instruction(s) while **ZF** remains 1, i.e. **ZF=1**

```
REPZ  instruction_to_be_repeated
```

4) REPNE

This instruction is same as **REP**. It repeats the given instruction(s) till **CX** remains zero, i.e. **CX=0**

```
REPZ  instruction_to_be_repeated
```

5) REPNZ

It repeats the instructions while **ZF=0**

```
REPNZ  instruction_to_be_repeated
```

6) MOVSB

Stands for **'Move String Byte'**. It moves the contents of a byte (8 bits) from **DS:SI** to **ES:DI**

```
MOVSB
```

7) MOVSW

Stands for **'Move String Word'**. It moves the contents of a word (16 bits) from **DS:SI** to **ES:DI**

```
MOVSW
```

8) MOVSD

Stands for **'Move String Double Word'**. It moves the contents of a double word (32 bits) from **DS:SI** to **ES:DI**

```
MOVSD
```

9) CMPSB

Stands for **'Compare String Byte'**. It moves the compares the contents of a byte (8 bits) at **DS:SI** with the contents of a byte at **ES:DI** and sets the flag.

```
CMPSB
```

10) CMPSW

Stands for **'Compare String Word'**. It moves the compares the contents of the word (16 bits) at **DS:SI** with the contents of the word at **ES:DI** and sets the flag.

```
CMPSW
```

11) CMPSD

Stands for **'Compare String Double Word'**. It moves the compares the contents of the double word (32 bits) at **DS:SI** with the contents of the double word at **ES:DI** and sets the flag.

```
CMPSD
```