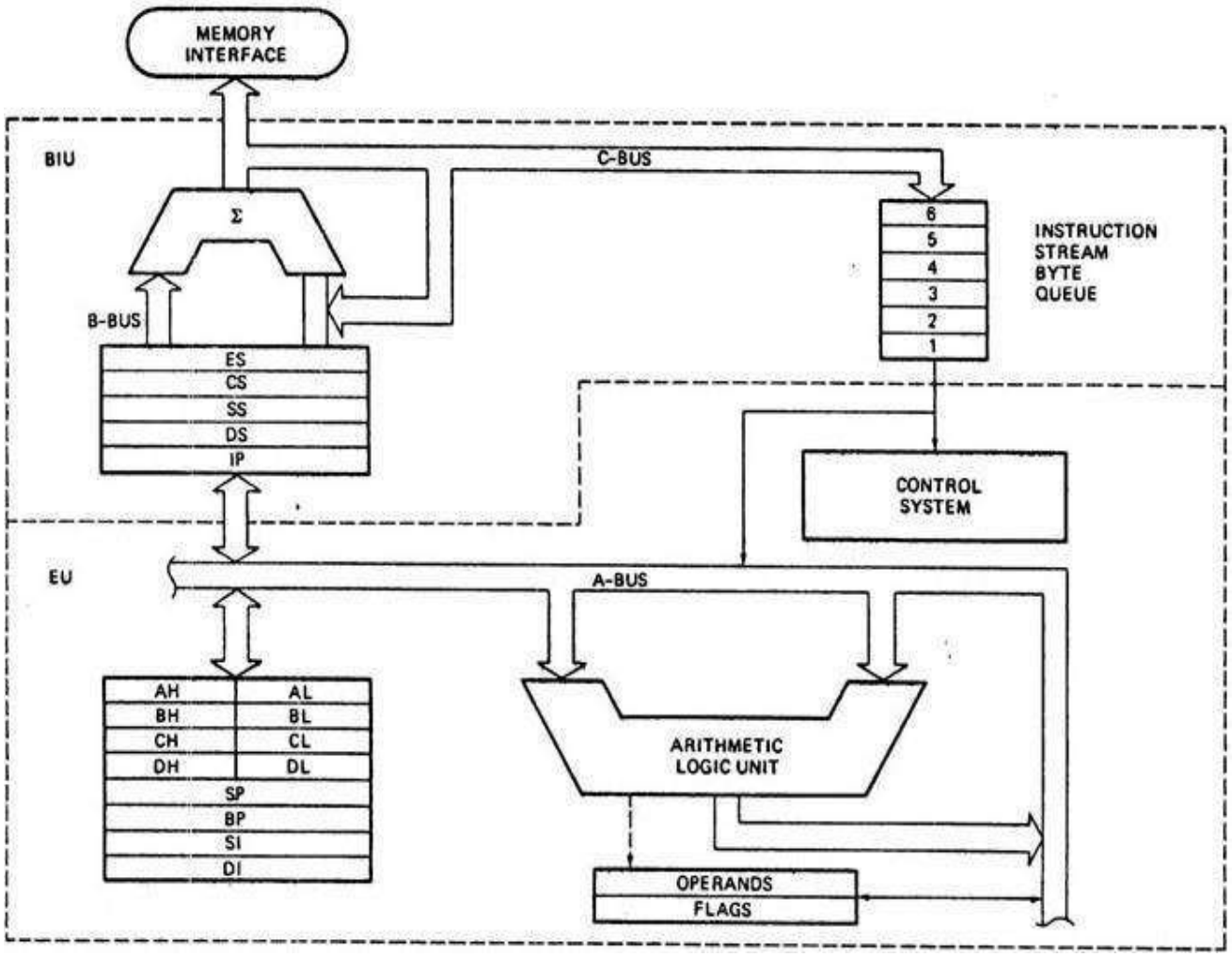
UNIT-V

8086 Microprocessors

Contents at a glance:

- ✓ Architecture of 8086 microprocessor
- ✓ Register organization
- ✓ 8086 flag register and its functions
- ✓ Addressing modes of 8086
- ✓ Pin diagram of 8086
- ✓ Minimum mode & Maximum mode system operation
- ✓ Instruction set



The Execution Unit (EU):

- The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions.
- The EU contains **control circuitry,** which directs internal operations.
- A decoder in the EU translates instructions fetched from memory into a series of actions, which the EU carries out.
- The EU has a 16-bit **arithmetic logic unit** (ALU) which can add, subtract, AND, OR, XOR, increment, decrement, complement or shift binary numbers.
- The main functions of EU are:
  - Decoding of Instructions
  - Execution of instructions
    - ✓ Steps
      - EU extracts instructions from top of queue in BIU
      - Decode the instructions
      - Generates operands if necessary
      - Passes operands to BIU & requests it to perform read or write bus cycles to memory or I/O
      - Perform the operation specified by the instruction on operands

Bus Interface Unit (BIU):

- The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory.

- In simple words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

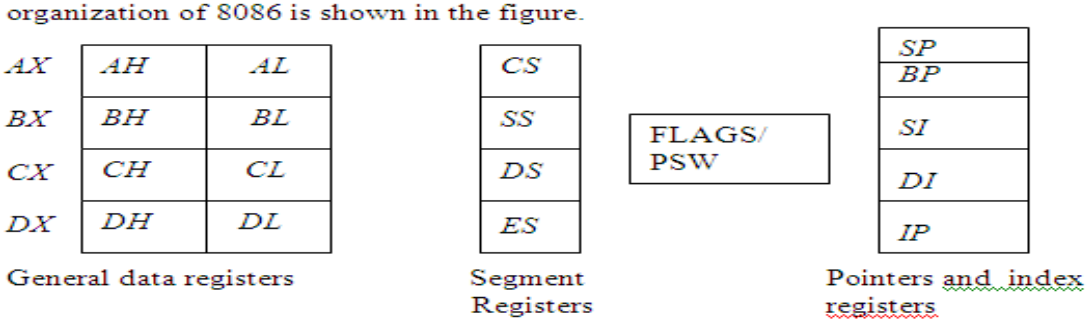
8086 HAS PIPELINING ARCHITECTURE:

- While the EU is decoding an instruction or executing an instruction, which does not require use of the buses, the BIU fetches up to six instruction bytes for the following instructions.
- The BIU stores these pre-fetched bytes in a first-in-first-out register set called a *queue.*
- When the EU is ready for its next instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes.
- Except in the case of JMP and CALL instructions, where the queue must be dumped and then reloaded starting from a new address, this pre-fetch and queue scheme greatly speeds up processing.
- Fetching the next instruction while the current instruction executes is called **pipelining**.

Register organization:

- 8086 has a powerful set of registers known as *general purpose registers* and *special purpose registers*.
- All of them are 16-bit registers.
- *General purpose registers:*
  - These registers can be used as either 8-bit registers or 16-bit registers.
  - They may be either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes etc.
- *Special purpose registers:*
  - These registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes.
- The 8086 registers are classified into the following types:
  - General Data Registers
  - Segment Registers
  - Pointers and Index Registers
  - Flag Register

The register set of 8086 can be categorized into 4 different groups. The register organization of 8086 is shown in the figure.



Register organization of 8086

General Data Registers:

- The registers *AX, BX, CX* and *DX* are the general purpose 16-bit registers.
- *AX* is used as 16-bit accumulator. The lower 8-bit is designated as *AL* and higher 8-bit is designated as *AH*. *AL*
  can be used as an 8-bit accumulator for 8-bit operation.
- All data register can be used as either 16 bit or 8 bit. *BX* is a 16 bit register, but *BL* indicates the lower 8-bit of
  BX and *BH* indicates the higher 8-bit of *BX*.
- The register *BX* is used as offset storage for forming physical address in case of certain addressing

modes.

- The register *CX* is used default counter in case of string and loop instructions.
- *DX* register is a general purpose register which may be used as an implicit operand or destination in case of a few instructions

Segment Registers:

- There are 4 segment registers. They are:
  - Code Segment Register(CS)
  - Data Segment Register(DS)
  - Extra Segment Register(ES)
  - Stack Segment Register(SS)
- The 8086 architecture uses the concept of **segmented memory**. 8086 able to address a memory capacity of 1 megabyte and it is byte organized. This 1 megabyte memory is divided into 16 logical segments. Each segment contains 64 kbytes of memory.
- Code segment register (CS): is used for addressing memory location in the code segment of the memory, where the executable program is stored.
- Data segment register (DS): points to the data segment of the memory where the data is stored.
- Extra Segment Register (ES) : also refers to a segment in the memory which is another data segment in the memory.
- Stack Segment Register (SS): is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.
- While addressing any location in the memory bank, the **physical address** is calculated from two parts:

  Physical address= segment address + offset address
- The first is segment address, the segment registers contain 16-bit segment base addresses, related to different segment.
- The second part is the offset value in that segment.

Pointers and Index Registers:

- The index and pointer registers are given below:
  - IP—Instruction pointer-store memory location of next instruction to be executed
  - BP—Base pointer
  - SP—Stack pointer
  - SI—Source index
  - DI—Destination index
- The pointers registers contain offset within the particular segments.
  - The pointer register *IP* contains offset within the code segment.
  - The pointer register *BP* contains offset within the data segment.
  - Thee pointer register *SP* contains offset within the stack segment.
- The index registers are used as general purpose registers as well as for offset storage in case of indexed, base indexed and relative base indexed addressing modes.
- The register *SI* is used to store the offset of source data in data segment.
- The register *DI* is used to store the offset of destination in data or extra segment.
- The index registers are particularly useful for string manipulation.

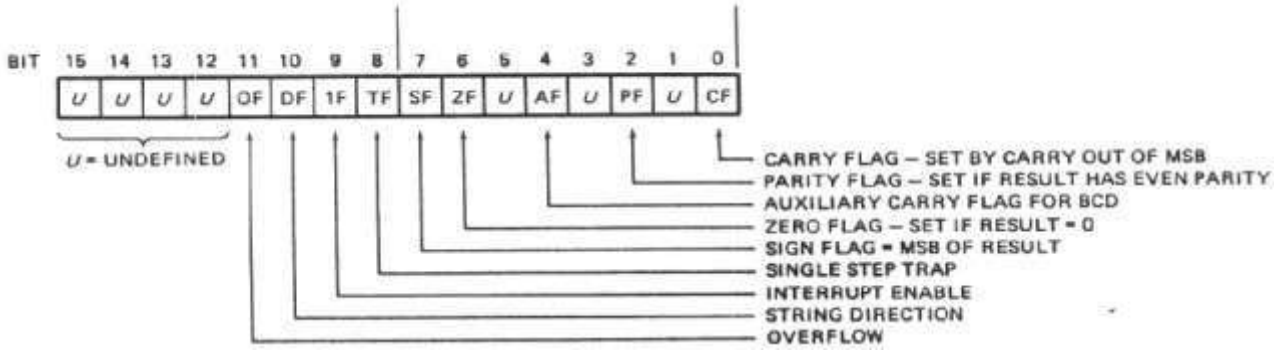
8086 flag register and its functions:

- The 8086 flag register contents indicate the results of computation in the *ALU*. It also contains some flag bits to control the *CPU* operations.
- A 16 bit flag register is used in 8086. It is divided into two parts .
  - Condition code or status flags
  - Machine control flags
- The **condition code flag register** is the lower byte of the 16-bit flag register. The condition code flag

register is identical to 8085 flag register, with an additional overflow flag.

- The **control flag register** is the higher byte of the flag register. It contains three flags namely direction flag (*D*), interrupt flag (*I*) and trap flag (*T*).

Flag register configuration



The description of each flag bit is as follows:

***SF*- Sign Flag:** This flag is set, when the result of any computation is negative. For signed computations the sign flag equals the MSB of the result.

***ZF*- Zero Flag:** This flag is set, if the result of the computation or comparison performed by the previous instruction is zero.

***PF*- Parity Flag:** This flag is set to 1, if the lower byte of the result contains even number of 1's.

***CF*- Carry Flag:** This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

***AF*-Auxilary Carry Flag:** This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

***OF*- Over flow Flag:** This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, and then the overflow will be set.

***TF*- Tarp Flag:** If this flag is set, the processor enters the single step execution mode. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

***IF*- Interrupt Flag:** If this flag is set, the mask able interrupts are recognized by the CPU, otherwise they are ignored.

***D*- Direction Flag:** This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

Memory Segmentation:

- The memory in an 8086 based system is organized as segmented memory.
- The CPU 8086 is able to access 1MB of physical memory. The complete 1MB of memory can be divided into 16 segments, each of 64KB size and is addressed by one of the segment register.
- The 16-bit contents of the segment register actually point to the starting location of a particular segment. The address of the segments may be assigned as 0000H to F000h respectively.

- To address a specific memory location within a segment, we need an offset address. The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH.

Physical address is calculated as below:

Ex: Segment address---→

1005H Offset address ----------

→ 5555H

Segment address -------→ 1005H0001 0000 0000 0101

Shifted left by 4 Positions0001 0000 0000 0101 0000

+

Offset address --- 5555H ------ 0101 0101 0101 0101

Physical address -------155A5H0001 0101 0101 1010 0101

Physical address = Segment address * 10H + Offset address.

The main advantages of the segmented memory scheme are as follows:

1. Allows the memory capacity to be 1MB although the actual addresses to be handled are of 16-bit size.
2. Allows the placing of code, data and stack portions of the same program in different parts (segments) of memory, for data and code protection.
3. Permits a program and/or its data to be put into different areas of memory each time the program is executed, i.e., provision for relocation is done.

Overlapping and Non-overlapping Memory segments:

- In the overlapping area locations physical address = CS1+IP1 = CS2+IP2. Where '+' indicates the procedure of physical address formation.

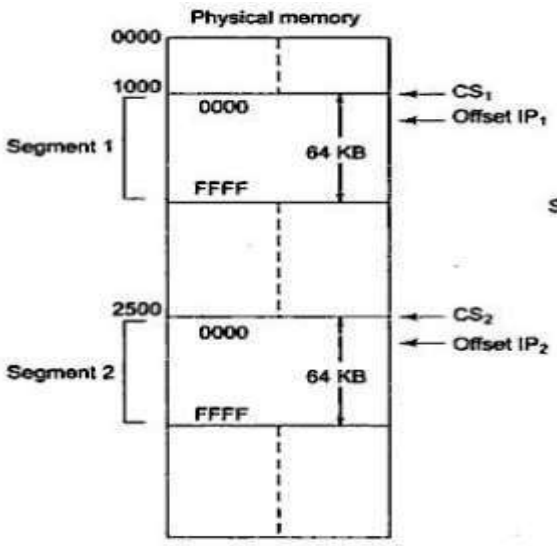


Fig. 1.3(a) *Non-overlapping Segments*

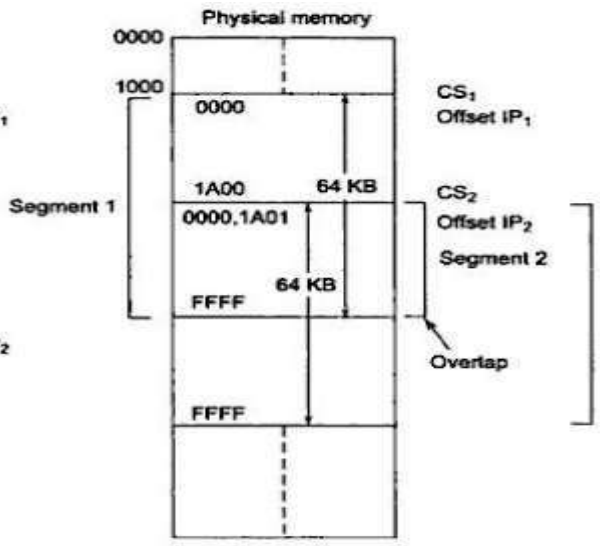


Fig. 1.3(b) *Overlapping Segments*

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:

1. **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.

2. **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.

3. **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

4. **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].

5. **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].

6. **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]

7. **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].

8. **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

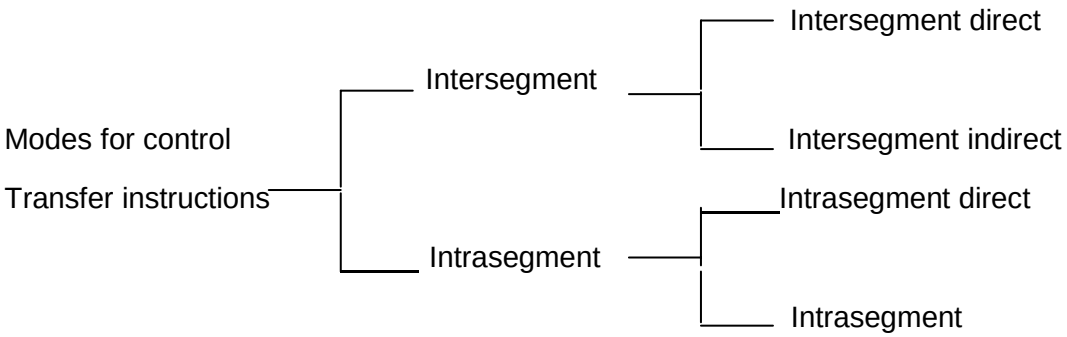
10H * DS + [BX] + [SI] + 50H

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, **intersegment** addressing and **intrasegment** 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 intersegment mode.

If the destination location lies in the same segment, the mode is called intrasegment mode.



indirect Addressing modes for Control Transfer Instructions

9. **Intrasegment Direct Mode:** In this 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. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

The effective address to which the control will be transferred is given by the sum of 8 or 16-bit displacement and current content of IP. In the case of jump instruction, if the signed displacement (d) is of 8-bits (i.e –128<d<+128) we term it as short jump and if it is of 16-bits (i.e-32, 768<d<+32,768) it is termed as long jump.

10. **Intrasegment Indirect Mode:** In this 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 indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode

may be used in unconditional branch instructions.

11. **Intersegment Direct:** In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

12. **Intersegment Indirect:** In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e contents of a memory block containing four bytes, i.e IP (LSB), IP(MSB), CS(LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

Forming the effective Addresses:

The following examples explain forming of the effective addresses in the different modes.

Ex: 1. The contents of different registers are given below. Form effective addresses for different addressing modes.

Offset (displacement)=5000H

[AX]-1000H, [BX]- 2000H, [SI]-3000H, [DI]-4000H, [BP]-5000H,

[SP]-6000H, [CS]-0000H, [DS]-1000H, [SS]-2000H, [IP]-7000H

Shifting segment address four bits to the left is equivalent to multiplying it by 16_D or 10_H

i. Direct addressing mode:

MOV AX,[5000H]

DS : OFFSET ⇔ 1000H : 5000H

10H*DS ⇔ 10000—segment address

offset ⇔ +5000---offset address

15000H – Effective address

ii. Register indirect:

MOV AX, [BX]

DS: BX ⇔ 1000H:

2000H

10H*DS ⇔ 10000—segment address

[BX] ⇔ +2000---offset address

12000H – Effective address

iii. Register relative:

MOV AX, 5000

[BX] DS : [5000+BX]

10H*DS ⇔ 10000

offset ⇔ +5000

[BX] ⇔ +2000

17000H – Effective address

iv. Based indexed:

MOV AX, [BX] [SI]

DS : [BX + SI]

10H*DS ⇔ 10000

[BX] ⇔ +2000

[SI] ⇔ +3000

15000H – Effective address

v. Relative based index:

MOV AX,

5000[BX][SI] DS : [BX+SI+5000]

10H*DS ⇔ 10000

[BX] ⇔ +2000

[SI] ⇔ +3000

+5000

1A000H – Effective address

Pin Diagram of 8086:

Signal description of 8086:

- The 8086 is a 16-bit microprocessor. This microprocessor operates in single processor or multiprocessor configurations to achieve high performance.
- The pin configuration of 8086 is shown in the figure. Some of the pins serve a particular function in minimum mode (single processor mode) and others function in maximum mode (multiprocessor mode).

The 8086 signals are categorized into 3 types:

1. Common signals for both minimum mode and maximum mode.
2. Special signals which are meant only for minimum mode
3. Special signals which are meant only for maximum mode

Common Signals for both Minimum mode and Maximum mode:

$AD_7 - AD_0$: The address/ data bus lines are the multiplexed address data bus and contain the right most eight bit of memory address or data. The address and data bits are separated by using *ALE* signal.

$AD_{15} - AD_8$: The address/data bus lines compose the upper multiplexed address/data bus. This lines contain address

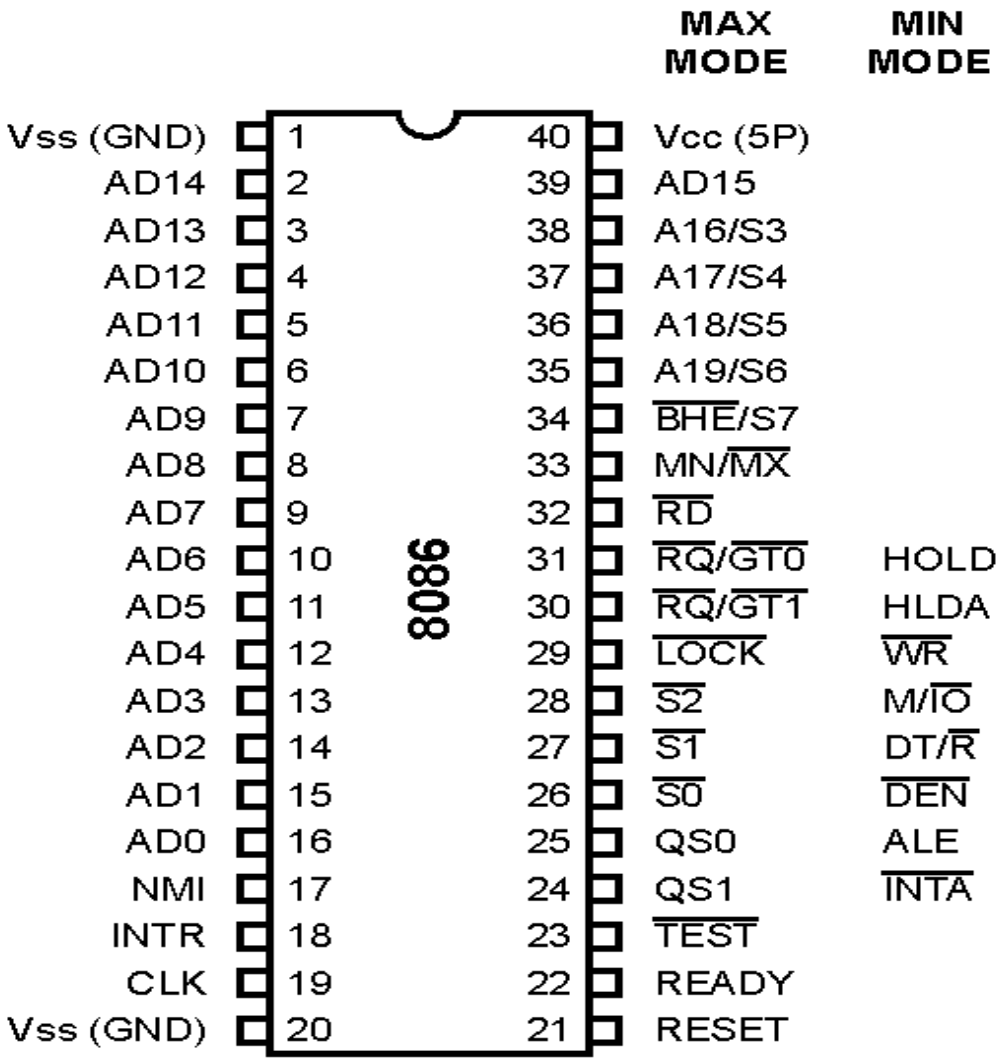
bit $A_{15} - A_8$ or data bus $D_{15} - D_8$. The address and data bits are separated by using *ALE* signal.

$A_{19}/S_6 - A_{18}/S_3$ The address/status bus bits are multiplexed to provide address signals $A_{19} - A_{16}$ and also status bits

$S_6 - S_3$. The address bits are separated from the status bits using the ALE signals. The status bit S_6 is always a logic 0, bit S_5 indicates the condition of the interrupt flag bit. The S_4 and S_3

being used for memory access.

S_4	S_3	Type of segment register used
0	0	Extra segment
0	1	Stack segment
1	0	Code segment
1	1	Data Segment



$\overline{BHE}/S_7$

The bus high enable (BHE) signal is used to indicate the transfer of data over the higher order $(D_{15}-D_8)$ data bus. It goes low for the data transfer over $D_{15}-D_8$ and is used to derive chip select of odd address memory bank or peripherals.

$\overline{BHE}$	A_0	Indication
0	0	Whole word
0	1	Upper byte from or to odd address
1	0	Lower byte from or to even address
1	1	None

$\overline{RD}$: Read:whenever the read signal is at logic0, the data bus receives the data from the memory or I/O devices connected to the system

READY: This is the acknowledgement from the slow devices or memory that they have completed the data transfer operation. This signal is active high.

INTR: Interrupt Request: Interrupt request is used to request a hardware interrupt of *INTR* is held high when interrupt enable flag is set, the 8086 enters an interrupt acknowledgement cycle after the current instruction has completed its execution.

$\overline{TEST}$: This input is tested by "*WAIT*" instruction. If the *TEST* input goes low; execution will continue. Else the processor remains in an idle state.

NMI- Non-maskable Interrupt: The non-maskable interrupt input is similar to *INTR* except that the *NMI* interrupt does not check for interrupt enable flag is at logic 1, i.e, *NMI* is not maskable internally by software. If *NMI* is activated, the interrupt input uses interrupt vector 2.

RESET: The reset input causes the microprocessor to reset itself. When 8086 reset, it restarts the execution from memory location *FFFF0H*. The reset signal is active high and must be active for at least four clock cycles.

CLK: Clock input: The clock input signal provides the basic timing input signal for processor and bus control operation.
It is asymmetric square wave with 33% duty cycle.

V_{CC} (+5V): Power supply for the operation of the internal circuit

GND: Ground for the internal circuit

$\overline{MN}/MX$: The minimum/maximum mode signal to select the mode of operation either in minimum or maximum

mode configuration. Logic 1 indicates minimum mode.

Minimum mode Signals: The following signals are for minimum mode operation of 8086.

M/IO - Memory/$\overline{IO}$ M/IO signal selects either memory operation or I/O operation. This line indicates that the microprocessor address bus contains either a memory address or an I/O port address. Signal high at this pin indicates a memory operation. This line is logically equivalent to S_2 in maximum mode.

$\overline{INTA}$ - Interrupt acknowledge: The interrupt acknowledge signal is a response to the INTR input signal. The $\overline{INTA}$ signal is normally used to gate the interrupt vector number onto the data bus in response to an interrupt request.

ALE- Address Latch Enable: This output signal indicates the availability of valid address on the address/data bus, and is connected to latch enable input of latches.

DT/R : Data transmit/Receive: This output signal is used to decide the direction of date flow through the bi-directional buffer. $DT/R = 1$ Indicates transmitting and $DT/R = 0$ indicates receiving the data.

$\overline{DEN}$ Data Enable: Data bus enable signal indicates the availability of valid data over the address/data lines.

□□ Write: whenever the write signal is at logic 0, the data bus transmits the data to the memory or I/O devices connected to the system.

HOLD: The hold input request a direct memory access (DMA). If the hold signal is at logic 1, the micro process stops its normal execution and places its address, data and control bus at the high impedance state.

HLDA: Hold acknowledgement indicates that 8086 has entered into the hold state.

Maximum mode signal: The following signals are for maximum mode operation of 8086.

$\overline{S_2}, \overline{S_1}, \overline{S_0}$ - Status lines: These are the status lines that reflect the type of operation being carried out by the processor.

These status lines are encoded as follows

S_2	S_1	S_0	Function
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O port

0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive (In active)

$\overline{LOCK}$: The lock output is used to lock peripherals off the system, i.e, the other system bus masters will be prevented from gaining the system bus.

QS_1 and QS_0 - Queue status: The queue status bits shows the status of the internal instruction queue. The encoding of these signals is as follows

QS_1	QS_0	Function
0	0	No operation, queue is idle
0	1	First byte of opcode
1	0	Queue is empty
1	1	Subsequent byte of opcode

$\overline{RQ}$ / $\overline{GT}1$ and $\overline{RQ}$ / $\overline{GT}$ 0 - request/Grant: The request/grant pins are used by other local bus masters to force the processor to release the local bus at the end of the processors current bus cycle. These lines are bi- directional and are used to both request and grant a *DMA* operation. RQ / GT 0 is having higher priority than RQ / $GT1$

8086 Minimum mode system operation

- In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX| pin to logic1.

- In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system. The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices.
- Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.
- The general system organization is shown in below figure.

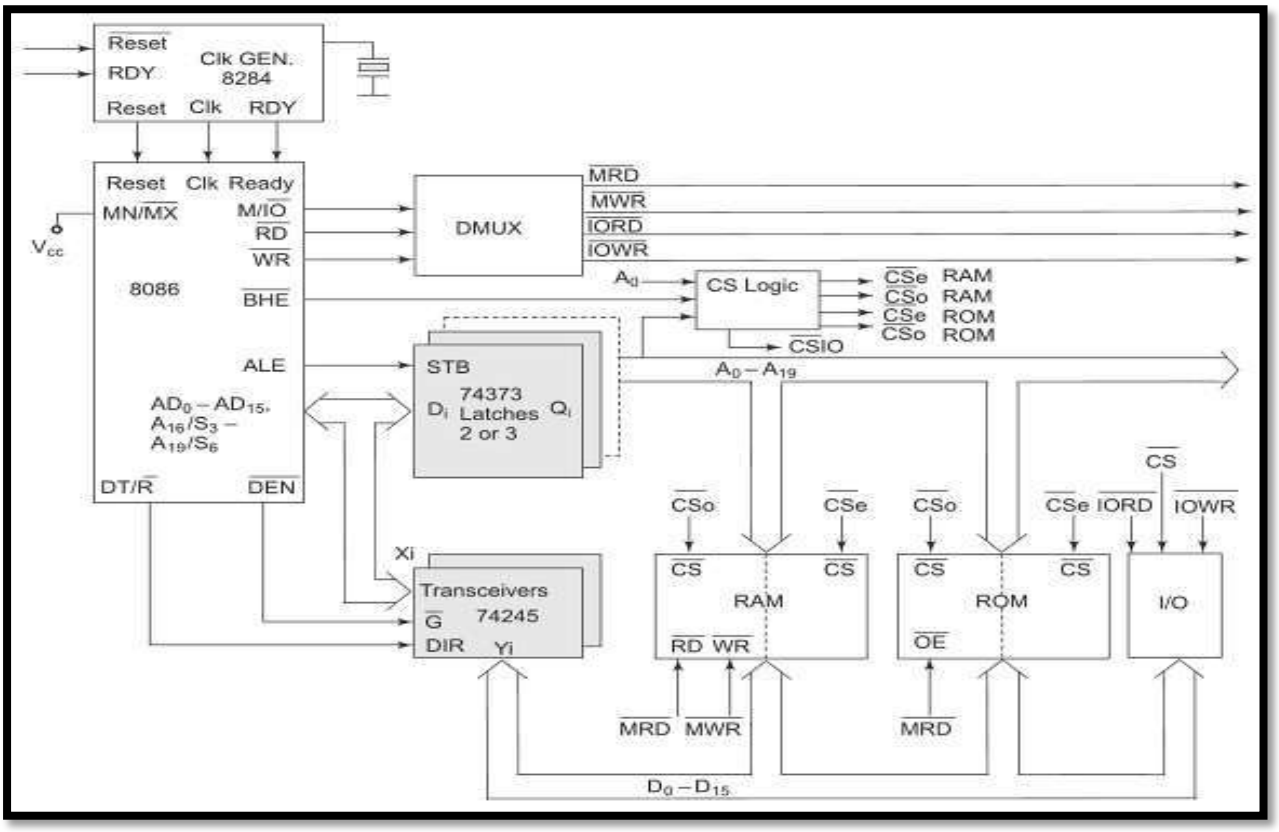


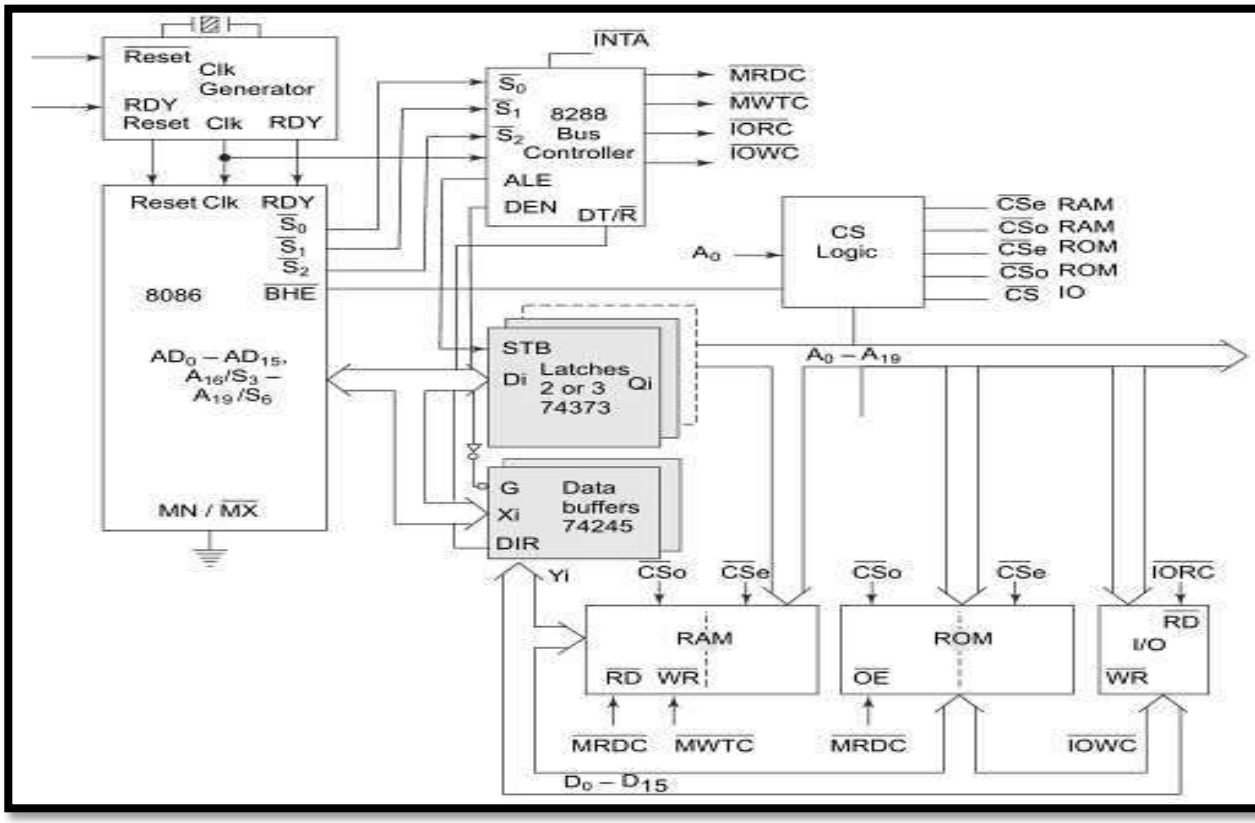
Figure: Minimum mode 8086 system

- The latches are generally buffered output D-type flip-flops, like, 74LS373 or 8282.
- They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.
- Since it has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation.
- Transreceivers are the bidirectional buffers and sometimes they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal.
- They are controlled by two signals, namely, DEN' and DT/R'. The DEN' signal indicates that the valid data is available on the data bus, while DT/R' indicates the direction of data, i.e. from or to the processor.
- The system contains memory for the monitor and users program storage. Usually, EPROMS are used for monitor storage, while RAMs for users program storage.
- A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices.
- The clock generator generates the clock from the crystal oscillator and then shapes it and divides to make it more precise so that it can be used as an accurate timing reference for the system.
- The clock generator also synchronizes some external signals with the system clock.
- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read

cycle and the second is the timing diagram for write cycle.

8086 Maximum mode system operation

- In the maximum mode, the 8086 is operated by strapping the MN/MX' pin to ground. In this mode, the processor derives the status signals S2', S1' and S0'. Another chip called bus controller derives the control signals using this status information.
- In the maximum mode, there may be more than one microprocessor in the system configuration. The other components in the system are the same as in the minimum mode system. The general system organization is as shown in the below figure.
- The basic functions of the bus controller chip IC8288, is to derive control signals like RD'and WR' (for memory and I/O devices), DEN, DT/R', ALE, etc. using the information made available by the processor on the status lines.
- The bus controller chip has input lines S2', S1' and S0' and CLK. These inputs to 8288 are driven by the CPU. It derives the outputs ALE, DEN, DT/R', MWTC', MRDC', IORC', IOWC' and INTA'.
- INTA' pin is used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.



- IORC*, IOWC* are I/O read command and I/O write command signals respectively. These signals enable an IO interface to read or write the data from or to the addressed port. The MRDC*, MWTC* are memory read command and memory write command signals respectively and may be used as memory read and write signals. All these command signals instruct the memory to accept or send data

from or to the bus.

INSTRUCTION SET OF 8086

The Instruction set of 8086 microprocessor is classified into 7, they are:-

- Data transfer instructions
- Arithmetic& logical instructions
- Program control transfer instructions
- Machine Control Instructions
- Shift / rotate instructions
- Flag manipulation instructions
- String instructions

Data Transfer instructions

Data transfer instruction, as the name suggests is for the transfer of data from memory to internal register, from internal register to memory, from one register to another register, from input port to internal register, from internal register to output port etc

1. MOV instruction

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

General Form:

MOV destination, source

Here the source and destination needs to be of the same size, that is both 8 bit or both 16 bit.

MOV instruction does not affect any flags.

Example:-

MOV BX, 00F2H ; load the immediate number 00F2H in BX register

MOV CL, [2000H] ; Copy 8 bit contents

nt

o t m
f h e
e m
o
r
y

location, at a displacement of 2000H from data segment base to the CL register

MOV [589H], BX ; Copy the 16 bit content of BX register on
to

the memory location, which at a displacement of 589H from the data segment base.

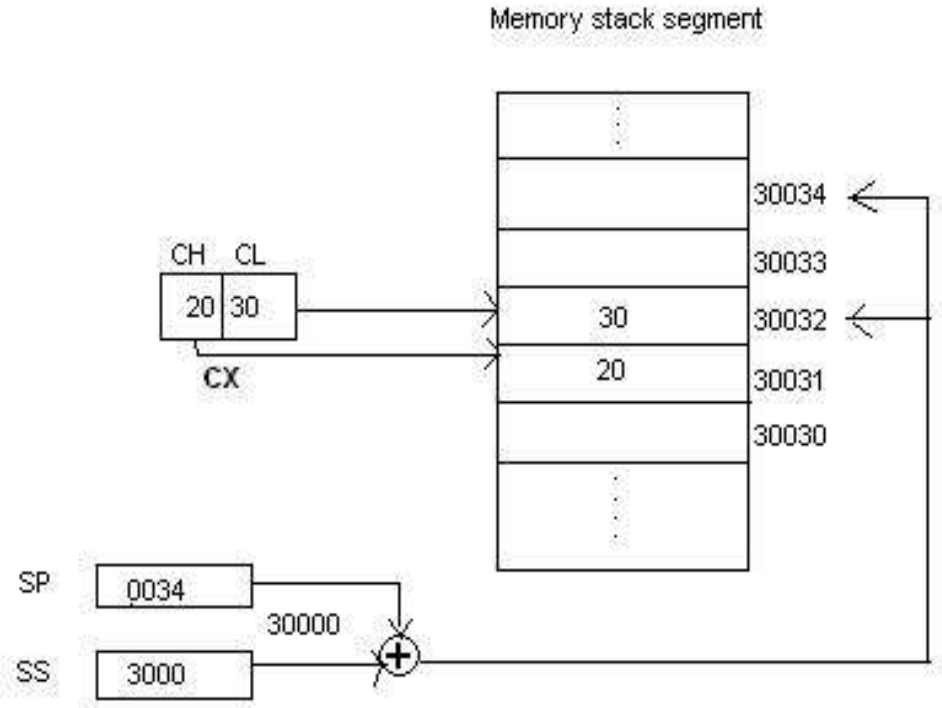
MOV DS, CX ; Move the content of CX to DS

2. PUSH instruction

The PUSH instruction decrements the stack pointer by two and copies the word from source to the location where stack pointer now points. Here the source must of word size data. Source can be a general purpose register, segment register or a memory location.

The PUSH instruction first pushes the most significant byte to sp-1, then the least significant to the sp-2.

Push instruction does not affect any flags.



Example:-

PUSH CX ; Decrements SP by 2, copy content of CX to the stack (figure shows execution of this instruction)

PUSH DS

; Decrement SP by 2 and copy DS to stack

3. POP instruction

The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a General purpose register, a segment register or a memory location. Here after the content is copied the stack pointer is automatically incremented by two.

The execution pattern is similar to that of the PUSH instruction.

Example:

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

4. IN & OUT instructions

The IN instruction will copy data from a port to the accumulator. If 8 bit is read the data will go to AL and if 16 bit then to AX. Similarly OUT instruction is used to copy data from accumulator to an output port.

Both IN and OUT instructions can be done using direct and indirect addressing modes.

Example:

```
                                  Copy a byte from the port 0F8H to
IN AL, 0F8H          ;              AL
MOV DX,
    30F8H            ;             Copy port address in DX

IN AL, DX            ;             Move 8 bit data from 30F8H port

IN AX, DX            ;             Move 16 bit data from 30F8H port
OUT 047H,                         Copy contents of AL to 8 bit port
    AL               ;              047H
MOV DX,
    30F8H            ;             Copy port address in DX

OUT DX, AL           ;             Move 8 bit data to the 30F8H port

OUT DX, AX           ;             Move 16 bit data to the 30F8H port
```

5. XCHG instruction

The XCHG instruction exchanges contents of the destination and source. Here destination and source can be register and register or register and memory location, but XCHG cannot interchange the value of 2 memory locations.

General Format

```
XCHG Destination,
    Source
```

Example:

```
XCHG BX, CX          ;     exchange word in CX with the word in BX

XCHG AL, CL          ;     exchange byte in CL with the byte in AL
XCHG AX,                   here physical address, which is
    SUM[BX]          ;      DS+SUM+[BX].
```

The content at physical address and the content
of AX are interchanged.

Arithmetic and Logic instructions

The arithmetic and logic logical group of instruction include,

1. ADD instruction

Add instruction is used to add the current contents of destination with that of source and store the result in destination. Here we can use register and/or memory locations. AF, CF, OF, PF, SF, and ZF flags are affected

General Format:

ADD Destination, Source

Example:

- ADD AL, 0FH ; Add the immediate content, 0FH to the content of AL and store the result in AL
- ADD AX, BX ; AX <= AX+BX
- ADD AX,0100H – IMMEDIATE
- ADD AX,BX – REGISTER
- ADD AX,[SI] – REGISTER INDIRECT OR INDEXED
- ADD AX, [5000H] – DIRECT
- ADD [5000H], 0100H – IMMEDIATE
- ADD 0100H – DESTINATION AX (IMPLICT)

2. 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 calculation) to the result. All the condition code flags are affected by this instruction. The examples of this instruction along with the modes are as follows:

Example:

- ADC AX,BX – REGISTER
- ADC AX,[SI] – REGISTER INDIRECT OR INDEXED
- ADC AX, [5000H] – DIRECT

- ADC [5000H], 0100H – IMMEDIATE
- ADC 0100H – IMMEDIATE (AX IMPLICT)

3. SUB instruction

SUB instruction is used to subtract the current contents of destination with that of source and store the result in destination. Here we can use register and/or memory locations. AF, CF, OF, PF, SF, and ZF flags are affected

General Format:

SUB Destination, Source

Example:

- SUB AL, 0FH ; subtract the immediate content, 0FH from the

content of AL and store the result in AL
- SUB AX, BX ; AX <= AX-BX
- SUB AX,0100H – IMMEDIATE (DESTINATION AX)
- SUB AX,BX – REGISTER
- SUB AX,[5000H] – DIRECT
- SUB [5000H], 0100H – IMMEDIATE

4. SBB: SUBTRACT WITH BORROW

The 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. The examples of this instruction are as follows:

Example:

- SBB AX,0100H – IMMEDIATE (DESTINATION AX)
- SBB AX,BX – REGISTER
- SBB AX,[5000H] – DIRECT
- SBB [5000H], 0100H – IMMEDIATE

5. CMP: COMPARE

The 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. The examples of this instruction are as follows:

Example:

- CMP BX,0100H – IMMEDIATE

- CMP AX,0100H – IMMEDIATE

- CMP [5000H], 0100H – DIRECT

- CMP BX,[SI] – REGISTER INDIRECT OR INDEXED

- CMP BX, CX – REGISTER

6. INC & DEC instructions

INC and DEC instructions are used to increment and decrement the content of the specified destination by one. AF, CF, OF, PF, SF, and ZF flags are affected.

Example:

```
INC     A       ;   AL<= AL + 1
INC     AX      ;   AX<=AX + 1
DEC     AL      ;   AL<= AL – 1
DEC     AX      ;   AX<=AX – 1
```

7. AND instruction

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

The CF and OF flags are both made zero, PF, ZF, SF are affected by the operation and AF is undefined.

General Format:

AND Destination, Source

Example:

- AND BL, AL ; suppose BL=1000 0110 and AL = 1100 1010 then after the operation BL would be BL= 1000 0010.

-
```
AND     CX,     AX ;
```

CX <= CX AND AX

AND CL, 08 ; CL<= CL AND (0000 1000)

8. OR instruction

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

The CF and OF flags are both made zero, PF, ZF, SF are affected by the operation and AF is undefined.

General Format:

OR Destination, Source

Example:

- OR BL, AL ; suppose BL=1000 0110 and AL = 1100 1010 then after the operation BL would be BL= 1100 1110.
- OR CX, AX ; CX <= CX AND AX
- OR CL, 08 ; CL<= CL AND (0000 1000)

9. NOT instruction

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 (BEFORE AX= (1011)2= (B) 16 AFTER EXECUTION AX= (0100)2= (4)16).
- NOT [5000H]

10. XOR instruction

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. The example instructions are as follows:

Example:

- XOR AX,0098H
- XOR AX,BX
- XOR AX,[5000H]

Shift / Rotate Instructions

Shift instructions move the binary data to the left or right by shifting them within the register or memory location. They also can perform multiplication of powers of 2^{+n} and division of powers of 2^{-n}.

There are two type of shifts logical shifting and arithmetic shifting, later is used with signed numbers while former with unsigned.

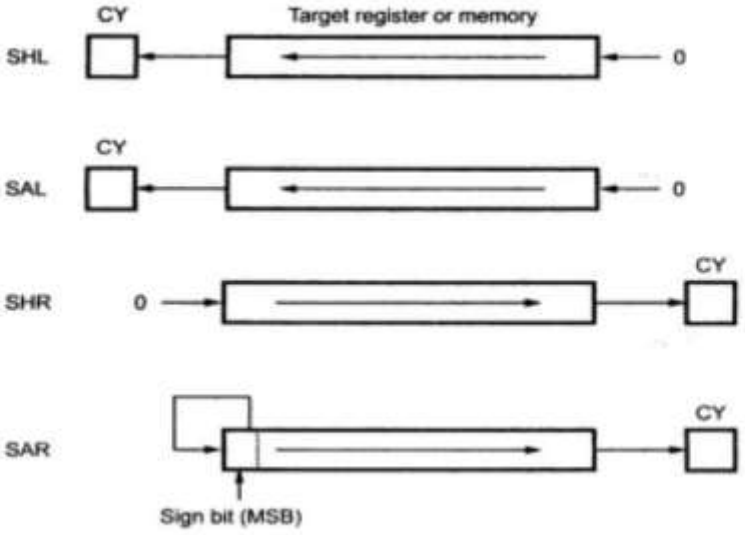


Fig.1 Shift operations

Rotate on the other hand rotates the information in a register or memory either from one end to another or through the carry flag.

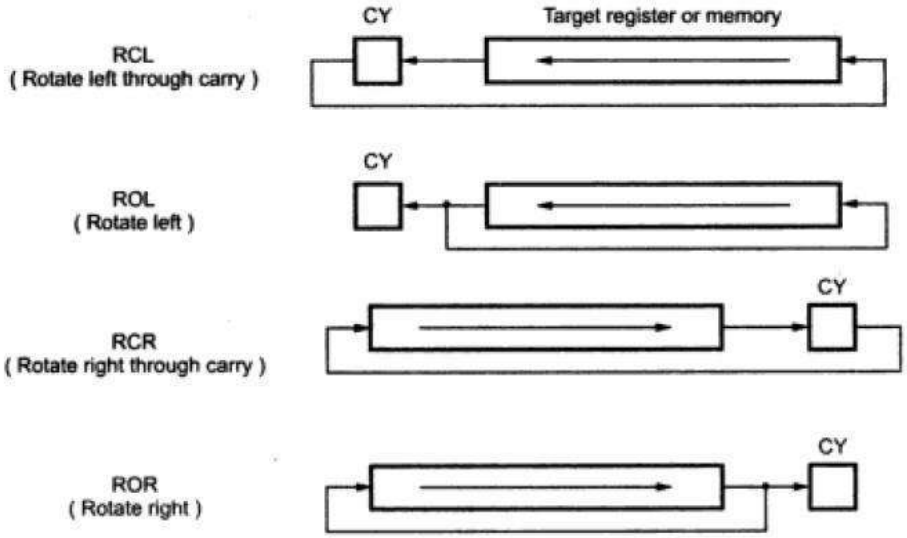


Fig.2 Rotate operations

SHL/SAL instruction

Both the instruction shifts each bit to left, and places the MSB in CF and LSB is made 0. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

All flags are affected.

General Format:

SAL/SHL destination, count

Example:

```
MOV BL, B7H      ;  BL is made B7H
SAL BL, 1        ;  shift the content of BL register one place to left.
```

Before execution,

CY	B
0	1

	B0
	1

After the execution,

	C
	Y
	0

1. SHR instruction

This instruction shifts each bit in the specified destination to the right and 0 is stored in the MSB position. The LSB is shifted into the carry flag. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

All flags are affected

General Format:

```
SHR destination, count
```

Example:

```
MOV BL, B7H      ;  BL is made B7H
```

SHR B L , 1 shift the content of BL register one place to the right.

Before execution,

CY

0

After execution,

CY

1

2. ROL instruction

This instruction rotates all the bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

All flags are affected

General Format:

ROL destination, count

Example:

```
MOV BL, B7H          ; BL is made B7H
ROL BL, 1            ; rotates the content of BL register one place
                     ; to the left.
```

Before execution,

B0

1

After the execution,

C

1

3. ROR instruction

This instruction rotates all the bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

All flags are affected

General Format:

ROR destination, count

Example:

MOV BL, B7H — BL is made B7H

ROR BL, 1 — shift the content of BL register one place to the right.

Before execution,

CY

1

(B0)

0

After execution,

CY

1

4. RCR instruction

This instruction rotates all the bits in a specified byte or word to the right some number of bit positions along with the carry flag. LSB is placed in a new CF and previous carry is placed in the new MSB. The destination can be of byte size or of word size, also it can be a register or a memory location. Number of shifts is indicated by the count.

All flags are affected

General Format: RCR destination, count

Example:

MOV BL, B7H ; BL is made B7H

RCR BL, 1 ; shift the content of BL register one place to the right.

Before execution,

CY

0

(CY)

After execution,

CY

1

Program control transfer instructions

There are 2 types of such instructions. They are:

1. Unconditional transfer instructions – CALL, RET, JMP
2. Conditional transfer instructions – J condition

1. CALL instruction

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two types of CALL instructions, near and far.

A **near CALL** is a call to a procedure which is in the same code segment as the

CALL instruction. 8086 when encountered a near call, it decrements the SP by 2 and copies the offset of the next instruction after the CALL on the stack. It loads the IP with the offset of the procedure then to start the execution of the procedure.

A **far CALL** is the call to a procedure residing in a different segment. Here value of CS and offset of the next instruction both are backed up in the stack. And then branches to the procedure by changing the content of CS with the segment base containing procedure and IP with the offset of the first instruction of the procedure.

Example:

Near call

```
CALL PRO          ;   PRO is the name of the procedure

CALL CX           ;   Here CX contains the offset of the first
                      instruction of
```

the procedure, that is replaces the content of IP with content of CX t h e

Far call

```
CALL DWORD PTR[8X]  ;   New values for CS and IP are fetched from
                        four
```

memory locations in the DS. The new value for CS is fetched from [8X] and [8X+1], the new IP is fetched from [8X+2] and [8X+3].

2. RET instruction

RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If it was a near call, then IP is replaced with the value at the top of the stack, if it had been a far call, then another POP of the stack is required. This second popped data from the stack is put in the CS, thus resuming the execution of the calling program. RET instruction can be followed by a number, to specify the parameters passed.

RET instruction does not affect any flags.

General format:

RET

Example:

```
p1 PROC                    procedure declaration.

MOV
AX,

RET                        return to caller.

p1 ENDP
```

3. JMP instruction

This is also called as unconditional jump instruction, because the processor jumps to the specified location rather than the instruction after the JMP instruction. Jumps can be **short jumps** when the target address is in the same segment as the JMP instruction or **far jumps** when it is in a different segment.

General Format:
JMP<targetaddress>

Example:

```
MOV AL,05H                                  ;

JMP label1                                  ;       jump over to label

MOV
AL, 00H ;

label1:   MOV
[2000H], AL ;
                                            ;
RET
```

4. Conditional Jump (J cond)

Conditional jumps are always short jumps in 8086. Here jump is done only if the condition specified is true/false. If the condition is not satisfied, then the execution proceeds in the normal way.

Ex

am

ple:

There are many conditional
jump instructions like JC :
Jump on carry (CF=set)

JNC : Jump on non carry (CF=reset)

JZ : Jump on zero (ZF=set)

JNO : Jump on overflow (OF=set)

Etc

5. Iteration control instructions

These instructions are used to execute a series of instructions some number of times. The number is specified in the CX register, which will be automatically decremented in course of iteration. But here the destination address for the jump must be in the range of -128 to 127 bytes.

Example:

Instructions here are:-

LOOP : loop through the set of instructions until CX is 0

LOOPE/LOOPZ: here the set of instructions are repeated until CX=0 or ZF=0

LOOPNE/LOOPNZ: here repeated until CX=0 or ZF=1 **Machine Control Instructions**

1. HLT instruction

The HLT instruction will cause the 8086 microprocessor to fetching and executing instructions.

The 8086 will enter a halt state. The processor gets out of this Halt signal upon an interrupt signal in INTR pin/NMI pin or a reset signal on RESET input.

General form:-

HLT

2. WAIT instruction

When this instruction is executed, the 8086 enters into an idle state. This idle state is continued till a high is received on the TEST input pin or a valid interrupt signal is received. Wait affects no flags. It generally is used to synchronize the 8086 with a peripheral device(s).

3. ESC instruction

This instruction is used to pass instruction to a coprocessor like 8087. There is a 6 bit instruction for the coprocessor embedded in the ESC instruction. In most cases the 8086 treats ESC and a NOP, but in some cases the 8086 will access data items in memory for the coprocessor

4. **LOCK instruction**

   In multiprocessor environments, the different microprocessors share a system bus, which is needed to access external devices like disks. LOCK

Instruction is given as prefix in the case when a processor needs exclusive access of the system bus for a particular instruction. It affects no flags.

Example:

LOCK XCHG SEMAPHORE, AL :The XCHG instruction requires two

bus accesses. The lock prefix prevents another processor from taking control of thesystem bus between the 2 accesses

5. NOP instruction

At the end of NOP instruction, no operation is done other than the fetching and decoding of the instruction. It takes 3 clock cycles. NOP is used to fill in time delays or to provide space for instructions while trouble shooting. NOP affects no flags.

Flag manipulation instructions

1. STC instruction

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

2. CLC instruction

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

3. CMC instruction

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

4. STD instruction

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

5. CLD instruction

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

6. STI instruction

This instruction sets the interrupt flag to 1. This enables INTR interrupt of the 8086. STI does not affect any other flag.

7. CLI instruction

This instruction resets the interrupt flag to 0. Due to this the 8086 will not respond to an interrupt signal on its INTR input. CLI does not affect any other flag.

String Instructions

1. MOVS/MOVSB/MOVSW

These instructions copy a word or byte from a location in the data segment to a location in the extra segment. The offset of the source is in SI and that of destination is in DI. For multiple word/byte transfers the count is stored in the CX register.

When direction flag is 0, SI and DI are incremented and when it is 1, SI and DI are decremented.

MOVS affect no flags. MOVSB is used for byte sized movements while MOVSW is for word sized.

Example:

```
                        clear the direction flag to auto increment SI
CLD                         and DI
MOV AX, 0000H
    ;
MOV DS,
    AX                  initialize data segment register to 0
MOV ES,
    AX                  initialize extra segment register to 0
MOV SI,
    2000H               Load the offset of the string1 in SI
MOV DI,
    2400H               Load the offset of the string2 in DI
MOV CX,
    04H                 load length of the string in CX
REP                     decrement CX and MOVSB until CX will be
    MOVSB                   0
```

2. REP/REPE/REP2/REPNE/REPNZ

REP is used with string instruction; it repeats an instruction until the specified condition becomes false.

Example:

```
                    =
REP                         CX=0
REPE/REPZ           =       CX=0 OR
                                ZF=0
REPNE/REP           =       CX=0 OR
    NZ                          ZF=1
```

3. LODS/LODSB/LODSW

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

Example:

```
CLD                         ; clear direction flag to auto increment SI MOV SI,
                            ;  point SI at string
OFFSET
    S_STRING                ;

LODS S_STRING
```

4. STOS/STOSB/STOSW

The STOS instruction is used to store a byte/word contained in AL/AX to

the offset contained in the DI register. STOS does not affect any flags. After copying the content DI is automatically incremented or decremented, based on the value of direction flag.

Example:

```
MOV DL, OFFSET D_STRING ;        assign DI with destination address.

STOS D_STRING ;  assumed byte assembler    assumed string    byte or word determine
```

word, if byte then AL is used and if of word size, AX is used.

5. CMPS/CMPSB/CMPSW

CMPS is used to compare the strings, byte wise or word wise. The comparison is affected by subtraction of content pointed by DI from that pointed by

SI. The AF, CF, OF, PF, SF and ZF flags are affected by this instruction, but neither operand is affected.

Example:

```
MOV SI, OFFSET F_STRING          point first string
MOV DI, OFFSET S_STRING          point second string
MOV CX, 0AH
CLD                              set the counter as 0AH
REPE CMPSB                       clear direction flag to auto increment
                                 repeatedly compare till unequal or
                                     counter =0
```

ASSEMBLER DIRECTIVES

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

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

1. **ASSUME:**

   It is used to tell the name of the logical segment the assembler to use for a specified segment.

   E.g.: ASSUME CS: CODE tells that the instructions for a program are in a logical segment named CODE.

2. **DB -Define Byte:**

   The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initializes the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

1) RANKS DB 01H,02H,03H,04H

   This statement directs the assembler to reserve four memory locations for a list named RANKS and initialize them with the above specified four values.

2) MESSAGE DB „GOOD MORNING‟

   This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initializes those locations by the ASCII equivalent of these characters.

3) VALUE DB 50H

   This statement directs the assembler to reserve 50H memory bytes and leave them uninitialized for the variable named VALUE.

3. **DD -**Define Double word - used to declare a double word type variable or to reserve memory locations that can be accessed as double word.

E .g.: A

R
R
A
Y

_

P
O
I
N
T
E
R

D
D

2
5
6
2
9
2
6
1
H

declares a double word named ARRAY_POINTER.

4. DQ -Define Quad word

This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

E.g.: BIG_NUMBER DQ 2432987456292612H

declares a quad word named BIG_NUMBER.

5. **DT -Define Ten Bytes**:

   The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialize the 10-bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

   E.g.: PACKED_BCD 11223344556677889900 declares an array that is 10 bytes in length.

6. **DW -Define Word:**

   The DW directives serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

1) WORDS DW 1234H, 4567H, 78ABH, 045CH

   This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialization, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses.

2) NUMBER1 DW 1245H

This makes the assembler reserve one word in memory.

7. **END-End of Program:**

   The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. Also, no useful program statement should lie in the file, after the END statement.

8. **ENDP-**End Procedure - Used along with the name of the procedure to indicate the end of a procedure.

E.g.: SQUARE_ROOT PROC: start of procedure
SQUARE_ROOT ENDP: End of procedure

9. **ENDS-End of Segment:**

   This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments.

Whatever are the contents of the segments, they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment. The structure shown below explains the fact more clearly.

DATA SEGMENT

---------- DATA
ENDS

ASSUME CS: CODE, DS: DATA CODE
SEGMENT

---------- CODE
ENDS ENDS

10. EQU-Equate - Used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the vale.

E.g.: CORRECTION_FACTOR EQU 03H

MOV AL, CORRECTION_FACTOR

11. EVEN - Tells the assembler to increment the location counter to the next even address if it is not already at an even address.

Used because the processor can read even addressed data in one clock cycle

12. EXTRN - Tells the assembler that the names or labels following the directive are in some other assembly module.

For example if a procedure in a program module assembled at a different time from that which contains the CALL instruction ,this directive is used to tell the assembler that the procedure is external

13. GLOBAL - Can be used in place of a PUBLIC directive or in place of an EXTRN directive.

It is used to make a symbol defined in one module available to other modules.

E.g.: GLOBAL DIVISOR makes the variable DIVISOR public so that it can be accessed from other modules.

14. GROUP-Used to tell the assembler to group the logical statements named after the directive into one logical group segment, allowing the contents of all the segments to be accessed from the same group segment base.

E.g.: SMALL_SYSTEM GROUP CODE, DATA, STACK_SEG

15. INCLUDE - Used to tell the assembler to insert a block of source code from the named file into the current source module.

This will shorten the source code.

16. LABEL- Used to give a name to the current value in the location counter.

This directive is followed by a term that specifies the type you want associated with that name.

E.g: ENTRY_POINT LABEL FAR

NEXT: MOV AL, BL

17. NAME- Used to give a specific name to each assembly module when programs consisting of several modules are written.

E.g.: NAME PC_BOARD

18. OFFSET- Used to determine the offset or displacement of a named data item or procedure from the start of the segment which contains it.

E.g.: MOV BX, OFFSET PRICES

19. ORG- The location counter is set to 0000 when the assembler starts reading a segment. The ORG directive allows setting a desired value at any point in the program.

E.g.: ORG 2000H

20. PROC- Used to identify the start of a procedure.

E.g.: SMART_DIVIDE PROC FAR identifies named the start of a procedure

SMART_DIVIDE and tells the assembler that the procedure is far

21. PTR- Used to assign a specific type to a variable or to a label.

E.g.: INC BYTE PTR[BX] tells the assembler that we want to increment the byte pointed to by BX

22. PUBLIC- Used to tell the assembler that a specified name or label will be accessed from other modules.

E.g.: PUBLIC DIVISOR, DIVIDEND makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

23. SEGMENT- Used to indicate the start of a logical segment.

E.g.: CODE SEGMENT indicates to the assembler the start of a logical segment called CODE

24. SHORT- Used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction.

E.g.: JMP SHORT NEARBY_LABEL

25. TYPE - Used to tell the assembler to determine the type of a specified variable.

E.g.: ADD BX, TYPE WORD_ARRAY is used where we want to increment BX to point to the next word in an array of words.

Macros:

Macro is a group of instruction. The macro assembler generates the code in the program each time where the macro is “called”. Macros can be defined by MACROP and ENDM assembler directives. Creating macro is very similar to creating a new opcode that can used in the program, as shown below.

Example:

INIT MACRO MOV
AX,@DATA MOV DS

MOV ES, AX ENDM

It is important to note that macro sequences execute faster than procedures because there is no CALL and RET instructions to execute. The assembler places the macro instructions in the program each time when it is invoked. This procedure is known as Macro expansion.

WHILE:

In Macro, the WHILE statement is used to repeat macro sequence until the expression specified with it is true. Like REPEAT, end of loop is specified by ENDM statement. The WHILE statement allows to use relational operators in its expressions.

The table-1 shows the relational operators used with WHILE statements.

OPERATOR	FUNCTION
EQ	Equal
NE	Not equal
LE	Less than or equal
LT	Less than
GE	Greater than or equal
GT	Greater than
NOT	Logical inversion
AND	Logical AND
OR	Logical OR

Table-1: Relational operators used in WHILE statement.

FOR statement:

A FOR statement in the macro repeats the macro sequence for a list of data. For example, if we pass two arguments to the macro then in the first iteration the FOR statement gives the macro sequence using first argument and in the second iteration it gives the macro sequence using second argument. Like WHILE statement, end of FOR is indicated by ENDM statement. The program shows the use of FOR statement in the macro.

Example1:

```
DISP MACRO CHR MOV AH,
02H FOR ARG, <CHR>
MOV DL, ARG INT 21H

ENDM ENDM

. MODEL SMALL

. CODE

START: DISP „M", „A", „C", „R", „O" END START
```