MODULE 2 ADDRESSING MODES AND INSTRUCTION SETS

ADDRESSING MODES

The different ways in which a source operand is denoted in an instruction are known as addressing mode the addressing modes for sequential control flow instructions are

- Immediate Addressing Mode
- Direct Addressing mode
- Register Addressing mode
- Register Indirect Addressing mode
- Indexed Addressing Mode
- Register Relative addressing mode
- Base plus indexed addressing mode
- Base relative plus indexed Addressing mode

IMMEDIATE ADDRESSING MODE

The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode. E.g. MOV AX, $(10AB)_H$. Here 10AB is directly moved to AX register.

DIRECT ADDRESSING MODE

The addressing mode in which the effective address of the memory location at which the data operand is stored is given in the instruction. The effective address (Offset) is just a 16-bit number written directly in the instruction. E.g. MOV AX, [5000H].

The square brackets around the 5000H denote the contents of the memory location. When executed, this instruction will copy the contents of the memory location into AX register. This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.

REGISTER ADDRESSING MODE

The instruction will specify the name of the register which holds the data to be operated by the instruction. All registers except IP may be used in this mode. E.g. MOV AX, BX

REGISTER INDIRECT ADDRESSING MODE

This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI. E.g. MOV AX, [BX]

INDEXED ADDRESSING MODE

In this addressing mode, the operands offset address is found by adding the contents of SI or DI register and 8-bit/16-bit displacements. DS and ES are the default segments for index registers SI and DI respectively. This is the special case of the of register indirect addressing mode. E.g., MOV AX, [SI]

REGISTER RELATIVE ADDRESSING MODE

In register relative Addressing, BX, BP, SI and DI is used to hold the base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction. E.g. MOV AX, 50H[BX]. Here 50H will be added to 5000H to get the effective address.

BASE PLUS INDEXED ADDRESSING MODE

In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register. The default segment registers may be ES or DS. E.g. MOV AX, [BX],[SI]

BASE RELATIVE PLUS INDEXED ADDRESSING MODE

In this addressing mode, the operands offset is computed by adding the base register contents. An Index registers contents and 8 or 16-bit displacement. E.g. MOV AX, 50H [BX] [SI]

INSTRUCTION SETS OF 8086

The 8086/8088 instructions are categorised into the following main types. This section explains the function of each of the instructions with suitable examples wherever necessary.

(i) Data Copy/Transfer Instructions These types of instructions are used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this category.

(ii) Arithmetic and Logical Instructions All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

(iii) Branch Instructions These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this class.

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

(v) Machine Control Instructions These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

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

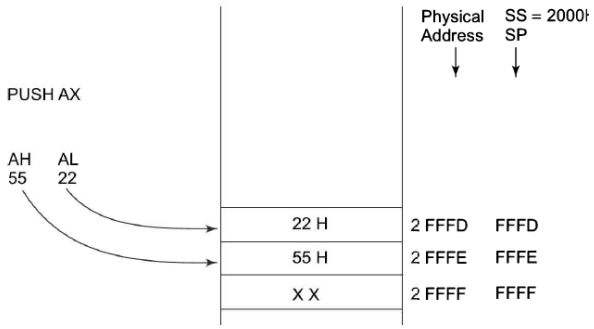
(vii) Shift and Rotate Instructions These instructions involve the bitwise shifting or rotation in either direction with or without a count in CX.

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

I. DATA COPY/ TRANSFER INSTRUCTIONS

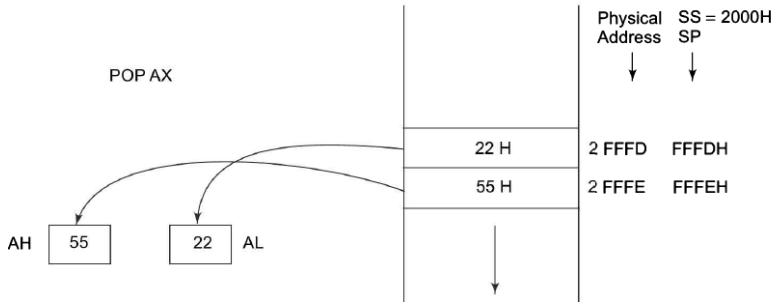
1) **MOV – MOVE**: This data transfer instruction transfers data from one register/memory location to another register/memory location. The source may be any one of the segment registers or other general or special purpose registers or a memory location and, another register or memory location may act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

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



3)

4) **POP: Pop from Stack** This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.



5)

6) **XCHG: Exchange** This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of contents of two memory locations is not permitted. Immediate data is also not allowed in these instructions.

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

8) **OUT: Output to the Port** This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on Dg-D,,; while that to an even addressed port is transferred on Do-D,. The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively. If the port address is of 16 bits it must be in DX.

9) **XLAT: Translate** The translate instruction is used for finding out the codes in case of code conversion problems, using look up table technique.

10) **LEA: Load Effective Address** = The load effective address instruction loads the effective address formed by destination operand into the specified source register. This instruction is more useful for assembly language rather than for machine language.

11) **LDS/LES: Load Pointer to DS/ES** This instruction loads the DS or ES register and the specific destination register in the instruction with the content of memory location specified as source in the instruction.

12) **LAHF : Load AH from Lower Byte of Flag** This instruction loads the AH register with the lower byte of the flag register. This command may be used to observe the status of all the condition code flags (except over flow) at a time.

13) **SAHF: Store AH to Lower Byte of Flag Register** = This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. Ifa bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

14) **PUSHF: Push Flags to Stack** The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte is pushed on to it. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

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

II. ARITHMETIC INSTRUCTIONS

1) **ADD: Add** This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location.
   1.ADD AX, O100H
   2.ADD AX, BX
   3.ADD AX, [SI]
   4.ADD AX, [5000H]
   5.ADD [5000H], O100H

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 calculations) to the result.
   1. ADC AX, BX Register
   2. ADC AX, [SI] Register indirect
   3. ADC AX, [5000H] Direct
   4. ADC [5000H], O100H Immediate

3) **INC: Increment** This instruction increases the contents of the specified register or memory location by
   1. All the condition code flags are affected except the carry flag CF.
   1. INC AX Register
   2. INC [BX] Register
   3. INC [5000H] Direct

4) **DEC: Decrement** = The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags, except the carry flag, are affected depending upon the result.
   1.DEC AX Register
   2.DEC [5000H] Direct

5) **SUB: Subtract** -The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register, memory location or immediate data and the destination operand may be a register or a memory location,
   1. SUB AX, O0100H Immediate [destination AX]
   2. SUB AX, BX Register
   3. SUB AX, [5000H] Direct
   4.SUB [5000H], 0100 Immediate

6) **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.
   1.SBB AX, 0100H Immediate [destination AX]
   2.SBB AX, BX Register
   3.SBB AX, [5000H] Direct
   4.SBB [5000H], 0100 Immediate

7) **CMP: Compare _** This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location.
   1.CMP BX, O10O0H Immediate
   2.CMP AX, QO100H Immediate
   3.CMP [5000H], O100H Direct
   4.CMP BX, [STI] Register indirect
   5.CMP BX, CX Register

8) **AAA: ASCII Adjust After Addition** The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4 high order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL.

9) **AAS: ASCII Adjust AL after Subtraction** AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. If the lower 4 bits of AL register are greater than 9 or if the AF flag is 1, the AL is decremented by 6 and AH register is decremented by 1

10) **AAM : ASCII Adjust after Multiplication** This instruction, after execution, converts the product available in AL into unpacked BCD format.

11) **AAD: ASCII Adjust before Division** The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL.

12) **DAA: Decimal Adjust Accumulator** This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL.

13) **DAS: Decimal Adjust after Subtraction** — This instruction converts the result of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only.

14) **NEG: Negate** The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand which may be a register or a memory location.

15) **MUL: Unsigned Multiplication Byte or Word** This instruction multiplies an unsigned byte or word by the contents of AL.

16) **IMUL: Signed Multiplication** — This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data.

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

18) **CWD: Convert Signed Word to Double Wor**d This instruction copies the sign bit of AX to all the bits of the DX register. This operation is to be done before signed division. It does not affect any flag.

19) **DIV: Unsigned Division _** This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate.

20) **IDIV: Signed Division _** This instruction performs the same operation as the DIV instruction, but with signed operands. The results are stored similarly as in case of DIV instruction in both cases of word and double word divisions.

III. LOGICAL INSTRUCTIONS

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

1) **AND: Logical AND** This instruction bit by bit ANDs the source operand that may be an immediate, a register or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand.
   1.AND AX, QOO8H
   2.AND AX, BX
   3.AND AX, [5000H]
   4.AND [5000H], DX

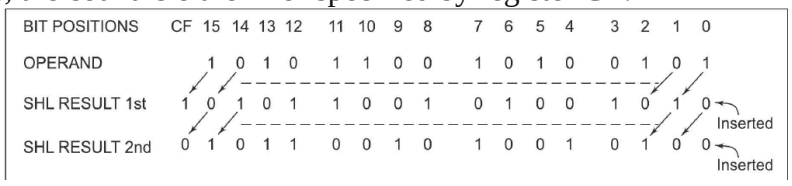
2) **OR: Logical OR** The OR instruction carries out the OR operation in the same way as described in case of the AND operation.
   1.OR AX, 0098H
   2.OR AX, BX
   3.OR AX, [5000H]
   4.OR [5000H], O008H

3) **NOT: Logical Invert** The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit.
   NOT AX
   NOT [5000H]

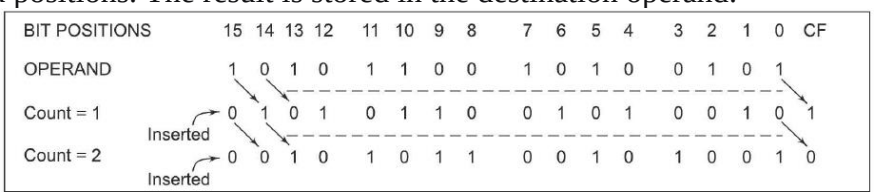
4) **XOR: Logical Exclusive OR** The XOR operation is again carried out in a similar way to the AND and OR operation.
   1.XOR AX, 0098H
   2.X0OR AX, BX
   3.XOR AX, [5000H]

5) **TEST: Logical Compare Instruction** The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to 1, if the corresponding bits of both operands are 1, else the result bit is reset to 0.
   1. TEST AX, BX
   2. TEST [0500], O6H
   3. TEST [BX] [DI], CX

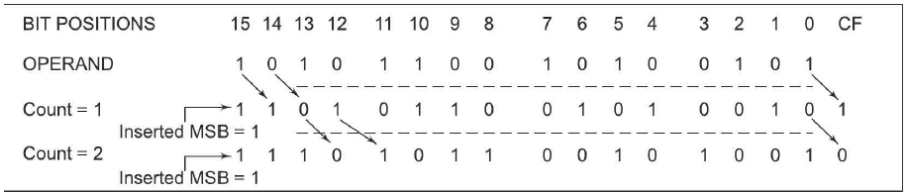
6) **SHL/SAL: Shift Logical/Arithmetic Left** These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL.



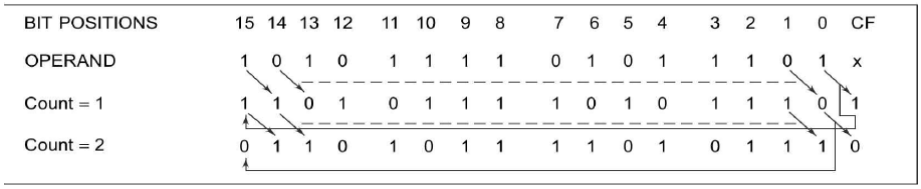
7) **SHR: Shift Logical Right** This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand.



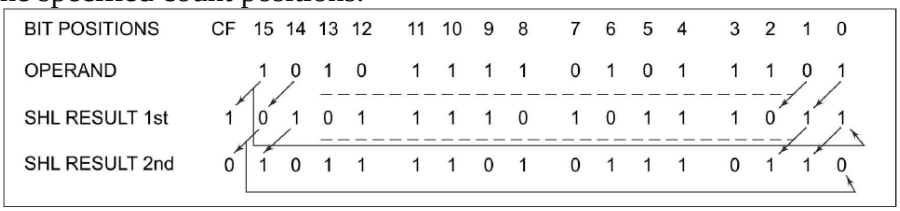
8) **SAR: Shift Arithmetic Right** — This instruction performs right shifts on the operand word or byte, that may be a register or a memory location by the specified count in the instruction. It inserts the most significant bit of the operand in the newly inserted positions.



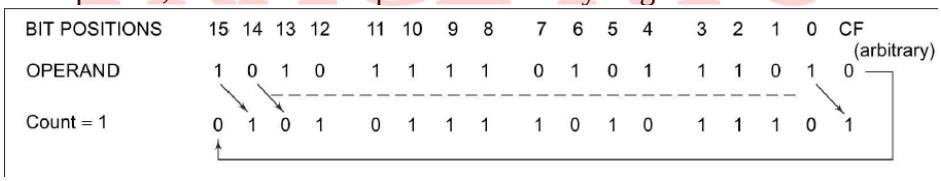
9) **ROR: Rotate Right without Carry** This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation.



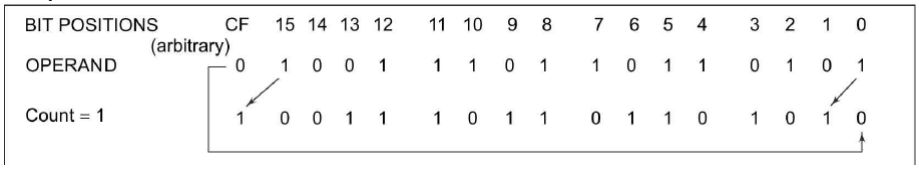
10) **ROL: Rotate Left without Carry** This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions.



11) **RCR: Rotate Right through Carry** This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF). For each operation, the carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag.



12) **RCL: Rotate Left through Carry** This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF). For each operation, the carry flag is pushed into LSB and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions.



IV. STRING MANIPULATION INSTRUCTIONS

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte strings or word strings. For example, a string of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent

1) **REP: Repeat Instruction Prefix** This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one). When CX becomes zero, the execution proceeds to the next instruction in sequence.

2) **MOVSB/MOVSW: Move String Byte or String Word** Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations.

3) **CMPS: Compare String Byte or String Word** The CMPS instruction can be used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set.

4) **SCAS: Scan String Byte or String Word** This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX.

5) **LODS: Load String Byte or String Word** The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending upon DF. The DF plays exactly the same role as in case of MOVSB/MOVSW instruction. If it is a byte transfer(LODSB), the SI is modified by one and if it is a word transfer(LODSW), the SI is modified by two. No other flags are affected by this instruction.

6) **STOS: Store String Byte or String Word** The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES: DI register pair. The DI is modified accordingly. No flags are affected by this instruction.

V. CONTROL TRANSFER OR BRANCH INSTRUCTIONS

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred.

Unconditional Control Transfer (Branch) Instructions —_ In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

Conditional Control Transfer (Branch) Instructions In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. The results of the previous operations are replicated by condition code flags. In other words, using this type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

Unconditional Branch Transfer

1) **CALL: Unconditional Call** — This instruction is used to call a subroutine from a main program. In case of assembly language programming, the term procedure is used interchangeably with subroutine. The address of the procedure may be specified directly or indirectly depending upon the addressing mode. There are again two types of procedures depending upon whether it is available in the same segment (Near CALL, i.e. +32K displacement) or in another segment (FAR CALL, i.e. anywhere outside the segment).

2) **RET: Return from the Procedure** At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. the RET instruction is of four types.
   a. Return within segment
   b. Return within segment adding 16-bit immediate displacement to the SP contents.
   c. Return intersegment
   d. Return intersegment adding 16-bit immediate displacement to the SP contents.

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

4) **INTO: Interrupt on Overflow** This command is executed, when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0010 as explained in INT type instruction. This is equivalent to a Type 4 interrupt instruction.

5) **JMP: Unconditional Jump** This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement (intra segment relative, short or long) or CS: IP (intersegment direct far). No flags are affected by this instruction.

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

7) **LOOP: Loop Unconditionally** This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times.

Conditional Branch Transfer

- When these instructions are executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied. If not the execution continues sequentially.
- The conditions, here, means the status of condition code flags.
- These type of instructions do not affect any flag.
- The address has to be specified in the instruction relatively in terms of displacement which must lie within —80H to 7FH (or —128 to 127) bytes from the address of the branch instruction.
- In other words, only short jumps can be implemented using conditional branch instructions.
- A label may represent the displacement, if it lies within the above specified range.

	Mnemonic	*Displacement*	*Operation*
1.	JZ/JE	Label	Transfer execution control to address ‘Label’, if ZF=1.
2.	JNZ/JNE	Label	Transfer execution control to address ‘Label’, if ZF=0.
3.	JS	Label	Transfer execution control to address ‘Label’, if SF=1.
4.	JNS	Label	Transfer execution control to address ‘Label’, if SF=0.
5.	JO	Label	Transfer execution control to address ‘Label’, if OF=1.
6.	JNO	Label	Transfer execution control to address ‘Label’, if OF=0.
7.	JP/JPE	Label	Transfer execution control to address ‘Label’, if PF=1.
8.	JNP	Label	Transfer execution control to address ‘Label’, if PF=0.
9.	JB/JNAE/JC	Label	Transfer execution control to address ‘Label’, if CF=1.
10.	JNB/JAE/JNC	Label	Transfer execution control to address ‘Label’, if CF=0.
11.	JBE/JNA	Label	Transfer execution control to address ‘Label’, if CF=1 or ZF=1.
12.	JNBE/JA	Label	Transfer execution control to address ‘Label’, if CF=0 or ZF=0.
13.	JL/JNGE	Label	Transfer execution control to address ‘Label’, if neither SF=1 nor OF=1.
14.	JNL/JGE	Label	Transfer execution control to address ‘Label’, if neither SF=0 nor OF=0.
15.	JLE/JNC	Label	Transfer execution control to address ‘Label’, if ZF=1 or neither SF nor OF is 1.
16.	JNLE/JE	Label	Transfer execution control to address ‘Label’, if ZF=0 or at least any one of SF and OF is 1(Both SF and OF are not 0).

VI. FLAG MANIPULATION AND PROCESSOR INSTRUCTIONS

These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions. The flag

manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution.

Table 2.5 *Flag Manipulation Instructions*

CLC	-	Clear carry flag
CMC	-	Complement carry flag
STC	-	Set carry flag
CLD	-	Clear direction flag
STD	-	Set direction flag
CLI	-	Clear interrupt flag
STI	-	Set interrupt flag

These instructions modify the Carry (CF), Direction (DF) and Interrupt (IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and autoincrement or autodecrement modes.

The following are the machine control instructions supported by 8086 and 8088. They do not require any operand.

Table 2.6 *Machine Control Instructions*

WAIT	-	Wait for Test input pin to go low
HLT	-	Halt the processor
NOP	-	No operation
ESC	-	Escape to external device like NDP (numeric co-processor)
LOCK	-	Bus lock instruction prefix.

ASSEMBLER DIRECTIVES AND OPERATORS

- An assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes.
- It decides the address of each label and substitutes the values for each of the constants and variables.
- It then forms the machine code for the mnemonics and data in the assembly language program.
- While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler.
- For completing all these tasks, an assembler needs some hints from the programmer, i.e. the required storage for a particular constant or a variable, logical names of the segments, types of the different routines and modules, end of file, etc.
- These types of hints are given to the assembler using some predefined alphabetical strings called assembler directives, which help the assembler to correctly understand the assembly language programs to prepare the codes.
- Another type of hint which helps the assembler to assign a particular constant with a label or initialise particular memory locations or labels with constants is an operator.
- In fact, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately.

1) **DB: Define Byte** —‘ The DB directive is used to reserve byte or bytes of memory locations in the available memory.

2) **DW: Define Word** The DW directive 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

3) **DQ: Define Quadword** This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialise it with the specified values.

4) **DT: Define Ten Bytes** The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialise 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.

5) **ASSUME: Assume Logical Segment Name** = The ASSUME directive is used to inform the assemble, the names of the logicals segments to be assumed for different segments used in the program. In the assembly language program, each segment is given a name.

- For example, the code segment may be given the name CODE, data segment may be given the name DATA etc. The statement ASSUME CS : CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE, while loading. Similarly, ASSUME DS : DATA indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS register is to be initialised by the segment address value decided by the operating system for the data segment, while loading.

6) **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.

7) **ENDP: END of Procedure** In assembly language programming, the subroutines are called procedures. They may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP. The statements, appearing in the same module but after the ENDP directive, are neglected from that procedure.

8) **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.


```
DATA        SEGMENT
              ⋮
DATA        ENDS
ASSUME      CS : CODE, DS : DATA
CODE        SEGMENT
              ⋮
CODE        ENDS
END
```

9) **EVEN: Align on Even Memory Address** The EVEN directive updates the location counter to the next even address, if the current location counter contents are not even, and assigns the following routine or variable or constant to that address.

10) **EQU: Equate** = The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitutes the numerical value for that label and finds out the equivalent code.

11) **EXTRN: External and PUBLIC**: Public The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive.

12) **GROUP: Group the Related Segments** This directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names. The assembler passes an information to the linker/loader to form the code such that the group declared segments or operands must lie within a 64K byte memory segment. Thus all such segments and labels can be addressed using the same segment base.

13) **LABEL: Label** The Label directive is used to assign a name to the current content of the location counter. When the assembly process starts, the assembler initialises a location counter to keep track of memory locations assigned to the program.

14) **LOCAL** The lables, variables, constants or procedures declared LOCAL in a module are to be used only by that particular module. After some time, some other module may declare a particular data type LOCAL, which was previously declared LOCAL by an other module or modules.

15) **NAME: Logical Name of a Module** The NAME directive is used to assign a name to an assembly language program module. The module, may now be referred to by its declared name. The names, if selected to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

16) **OFFSET: Offset of a Label** When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset inter changeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, labels and procedures to decide their offsets in their default segments.

17) **ORG: Origin** The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement.

18) **PROC: Procedure** The PROC directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e whether it is to be called by the main program located within 64K of physical memory or not

19) **PTR: Pointer** The POINTER operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD.

20) **PUBLIC** the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared lables, variables, constants or procedures the user must declare them externals using the EXTRN directive.

21) **SEGMENT: Logical Segment** The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program.

22) **SHORT** ~The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within —128 to +127 bytes from the address of the byte next to the jump opcode).

23) **GLOBAL** The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program.

24) **FAR PTR** This directive indicates the assembler that the label following FAR PTR is not available within the same segment and the address of the label is of 32-bits i.e. 2 bytes offset followed by 2 bytes segment address.

25) **NEAR PTR** This directive indicates that the label following NEAR PTR is in the same segment and