ADDRESSING MODES OF 8086

Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction. Here, we will present the addressing modes of the instructions depending upon their types. 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, logical, 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 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 explained as follows:

1. Immediate: In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

Example: 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.

Example: MOV AX, [5000H]

Here, data resides in a memory location in the data segment, whose effective address may be computed 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 it is referred using the particular register. All the registers, except IP, may be used in this mode.

Example: 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 registers. 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 registers. 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.

Example: 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 mode is a special case of the above discussed register indirect addressing mode.

Example: 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 10H*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. The example given before explains this mode.

Example: MOV Ax, 50H [BX]

Here, 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.

Example: 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].

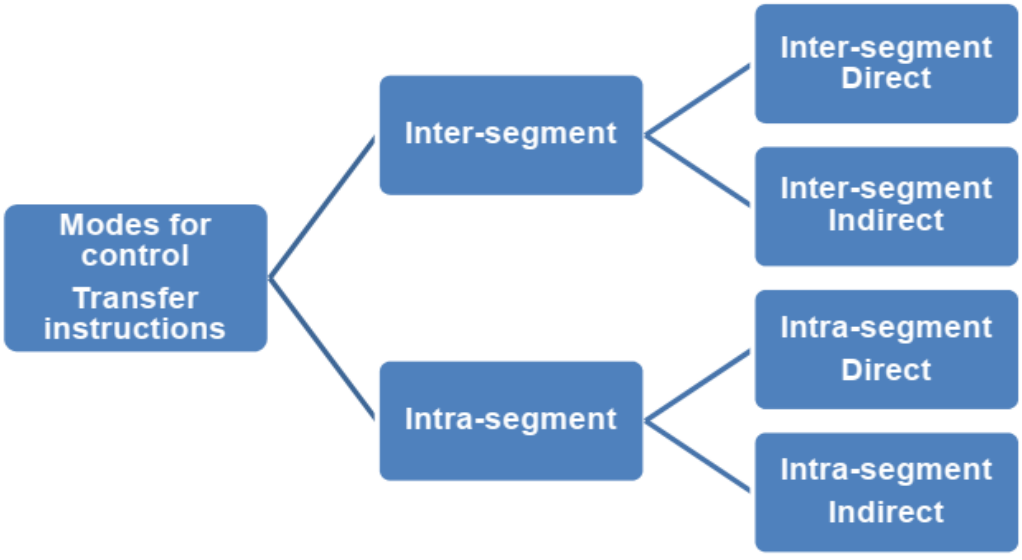
8. Relative Based Indexed: The effective address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the bases registers (BX or BP) and any one of the index registers, in a default segment.

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

Here, 50H is an immediate displacement, BX is a base register and SI is an index register. The effective address of data is computed as 160H*DS+ [BX] + [SI] + 50H.

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or a 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, viz. inter-segment and intra-segment addressing modes.

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



ADDRESSING MODES FOR CONTROL TRANSFER INSTRUCTION

1. **Intra-segment 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 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. –32768<+32768), it is termed as long jump.

2. **Intra-segment 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.

3. **Inter-segment Direct Mode:** 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.

4. **Inter-segment Indirect Mode:** 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.

8086 Instruction Set and Assembler Directives

The 8086 microprocessor supports 6 types of Instructions. They are

1. Data transfer instructions

2. Arithmetic instructions

3. Bit manipulation instructions

4. String instructions

5. Program Execution Transfer instructions (Branch & loop Instructions)

6. Processor control instructions

1. Data Transfer instructions: These instructions are used to transfer the data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belong to this group.

General purpose byte or word transfer instructions:

MOV : Copy byte or word from specified source to specified destination

PUSH : Push the specified word to top of the stack

POP : Pop the word from top of the stack to the specified location

PUSHA : Push all registers to the stack

POPA : Pop the words from stack to all registers

XCHG : Exchange the contents of the specified source and destination operands one of which may be a register or memory location.

XLAT : Translate a byte in AL using a table in memory

Simple input and output port transfer instructions

1. IN : Reads a byte or word from specified port to the accumulator
2. OUT : Sends out a byte or word from accumulator to a specified port

Special address transfer instructions

1. LEA : Load effective address of operand into specified register
2. LDS : Load DS register and other specified register from memory
3. LES : Load ES register and other specified register from memory.

Flag transfer registers

1. LAHF : Load AH with the low byte of the flag register
2. SAHF : Store AH register to low byte of flag register
3. PUSHF : Copy flag register to top of the stack
4. POPF : Copy word at top of the stack to flag register

2. Arithmetic instructions : These instructions are used to perform various mathematical operations like addition, subtraction, multiplication and division etc….

Addition instructions

1.ADD : Add specified byte to byte or word to word
2.ADC : Add with carry

3.INC : Increment specified byte or specified word by 1

4.AAA : ASCII adjust after addition

5.DAA : Decimal (BCD) adjust after addition

Subtraction instructions

1. SUB : Subtract byte from byte or word from word
2. SBB : Subtract with borrow
3. DEC : Decrement specified byte or word by 1
4. NEG : Negate or invert each bit of a specified byte or word and add 1(2's complement)
5. CMP : Compare two specified byte or two specified words
6. AAS : ASCII adjust after subtraction
7. DAS : Decimal adjust after subtraction

Multiplication instructions

1. MUL : Multiply unsigned byte by byte or unsigned word or word.
2. IMUL : Multiply signed bye by byte or signed word by word
3. AAM : ASCII adjust after multiplication

Division instructions

1. DIV : Divide unsigned word by byte or unsigned double word by word
2. IDIV : Divide signed word by byte or signed double word by word
3. AAD : ASCII adjust after division

4. CBW : Fill upper byte of word with copies of sign bit of lower byte
5. CWD : Fill upper word of double word with sign bit of lower word.

3. Bit Manipulation instructions : These instructions include logical , shift and rotate instructions in which a bit of the data is involved.

Logical instructions

1. NOT :Invert each bit of a byte or word.
2. AND : ANDing each bit in a byte or word with the corresponding bit in another byte or word.
3. OR : ORing each bit in a byte or word with the corresponding bit in another byte or word.
4. XOR : Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word.
5. TEST :AND operands to update flags, but don't change operands.

Shift instructions

1. SHL/SAL : Shift bits of a word or byte left, put zero(S) in LSBs.
2. SHR : Shift bits of a word or byte right, put zero(S) in MSBs.
3. SAR : Shift bits of a word or byte right, copy old MSB into new MSB.

Rotate instructions

1. ROL : Rotate bits of byte or word left, MSB to LSB and to Carry Flag [CF]
2. ROR : Rotate bits of byte or word right, LSB to MSB and to Carry Flag [CF]
3. RCR :Rotate bits of byte or word right, LSB TO CF and CF to MSB
4. RCL :Rotate bits of byte or word left, MSB TO CF and CF to LSB

4. String instructions

A string is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes.

1. REP : An instruction prefix. Repeat following instruction until CX=0
2. REPE/REPZ : Repeat following instruction until CX=0 or zero flag ZF=1
3. REPNE/REPNZ : Repeat following instruction until CX=0 or zero flag ZF=1
4. MOVS/MOVSB/MOVSW: Move byte or word from one string to another
5. COMS/COMPSB/COMPSW: Compare two string bytes or two string words
6. INS/INSB/INSW: Input string byte or word from port
7. OUTS/OUTSB/OUTSW : Output string byte or word to port
8. SCAS/SCASB/SCASW: Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX
9. LODS/LODSB/LODSW: Load string byte in to AL or string word into AX

5.Program Execution Transfer instructions

These instructions are similar to branching or looping instructions. These instructions include conditional & unconditional jump or loop instructions.

Unconditional transfer instructions

1. CALL : Call a procedure, save return address on stack
2. RET : Return from procedure to the main program.
3. JMP : Goto specified address to get next instruction

Conditional transfer instructions

1. JA/JNBE : Jump if above / jump if not below or equal
2. JAE/JNB : Jump if above /jump if not below
3. JBE/JNA : Jump if below or equal/ Jump if not above
4. JC : jump if carry flag CF=1
5. JE/JZ : jump if equal/jump if zero flag ZF=1
6. JG/JNLE : Jump if greater/ jump if not less than or equal
7. JGE/JNL : jump if greater than or equal/ jump if not less than
8. JL/JNGE : jump if less than/ jump if not greater than or equal
9. JLE/JNG : jump if less than or equal/ jump if not greater than
10. JNC : jump if no carry (CF=0)

11.JNE/JNZ : jump if not equal/ jump if not zero(ZF=0)
12.JNO : jump if no overflow(OF=0)
13.JNP/JPO : jump if not parity/ jump if parity odd(PF=0)
14.JNS : jump if not sign(SF=0)
15.JO : jump if overflow flag(OF=1)
16.JP/JPE : jump if parity/jump if parity even(PF=1)
17.JS : jump if sign(SF=1)

6. Iteration control instructions

These instructions are used to execute a series of instructions for certain number of times.

1. LOOP :Loop through a sequence of instructions until CX=0
2. LOOPE/LOOPZ : Loop through a sequence of instructions while ZF=1 and CX = 0
3. LOOPNE/LOOPNZ : Loop through a sequence of instructions while ZF=0 and CX =0
4. JCXZ : jump to specified address if CX=0

7. Interrupt instructions

1. INT : Interrupt program execution, call service procedure
2. INTO : Interrupt program execution if OF=1
3. IRET : Return from interrupt service procedure to main program

8.High level language interface instructions

1. ENTER : enter procedure
2. LEAVE :Leave procedure
3. BOUND : Check if effective address within specified array bounds

9.Processor control instructions

Flag set/clear instructions

1. STC : Set carry flag CF to 1
2. CLC : Clear carry flag CF to 0
3. CMC : Complement the state of the carry flag CF
4. STD : Set direction flag DF to 1 (decrement string pointers)
5. CLD : Clear direction flag DF to 0
6. STI : Set interrupt enable flag to 1(enable INTR input)
7. CLI : Clear interrupt enable Flag to 0 (disable INTR input)

10. External Hardware synchronization instructions

1. HLT : Halt (do nothing) until interrupt or reset
2. WAIT : Wait (Do nothing) until signal on the test pin is low
3. ESC : Escape to external coprocessor such as 8087 or 8089
4. LOCK : An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes.

11. No operation instruction

1. NOP : No action except fetch and decode

Instruction Description

- **AAA** Instruction - ASCII Adjust after Addition
- **AAD** Instruction - ASCII adjust before Division
- **AAM** Instruction - ASCII adjust after Multiplication
- **AAS** Instruction - ASCII Adjust for Subtraction
- **ADC** Instruction - Add with carry.
- **ADD** Instruction - ADD destination, source
- **AND** Instruction - AND corresponding bits of two operands

Example

- **AAA** Instruction:

AAA converts the result of the addition of two valid unpacked BCD digits to a valid 2-digit BCD number and takes the AL register as its implicit operand.

Two operands of the addition must have its lower 4 bits contain a number in the range from 0-9.The AAA instruction then adjust AL so that it contains a correct BCD digit. If the addition produce carry (AF=1), the AH register is incremented and the carry CF and auxiliary carry AF flags are set to 1. If the addition did not produce a decimal carry, CF and AF are cleared to 0 and AH is not altered. In both cases the higher 4 bits of AL are cleared to 0.

AAA will adjust the result of the two ASCII characters that were in the range from 30h (“0”) to 39h(“9”).This is because the lower 4 bits of those character fall in the range of 0-9.The result of addition is not a ASCII character but it is a BCD digit.

Example:

MOV AH, 0 ; Clear AH for MSD

MOV AL, 6 ; BCD 6 in AL

ADD AL, 5 ; Add BCD 5 to digit in AL

AAA ; AH=1, AL=1 representing BCD 11.

- **AAD Instruction:** ADD converts unpacked BCD digits in the AH and AL register into a single binary number in the AX register in preparation for a division operation.

  Before executing AAD, place the Most significant BCD digit in the AH register and Last significant in the AL register. When AAD is executed, the two BCD digits are combined into a single binary number by setting AL=(AH*10)+AL and clearing AH to 0.

Example:

MOV AX, 0205h ; The unpacked BCD number 25

AAD ; After AAD, AH=0 and

; AL=19h (25)

After the division AL will then contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder.

Example:

; AX=0607 unpacked BCD for 67 decimal

; CH=09H

AAD ; Adjust to binary before division

; AX=0043 = 43H =67 decimal

DIV CH ; Divide AX by unpacked BCD in CH

; AL = quotient = 07 unpacked BCD

; AH = remainder = 04 unpacked BCD

- **AAM** Instruction - AAM converts the result of the multiplication of two valid unpacked BCD digits into a valid 2-digit unpacked BCD number and takes AX as an implicit operand.

To give a valid result the digits that have been multiplied must be in the range of 0 – 9 and the result should have been placed in the AX register. Because both operands of multiply are required to be 9 or less, the result must be less than 81 and thus is completely contained in AL.

AAM unpacks the result by dividing AX by 10, placing the quotient (MSD) in AH and the remainder (LSD) in AL.

Example:

MOV AL, 5

MOV BL, 7

MUL BL ; Multiply AL by BL, result in AX

AAM ; After AAM, AX =0305h (BCD 35)

- **AAS** Instruction: AAS converts the result of the subtraction of two valid unpacked BCD digits to a single valid BCD number and takes the AL register as an implicit operand.

The two operands of the subtraction must have its lower 4 bit contain number in the range from 0 to 9.The AAS instruction then adjust AL so that it contain a correct BCD digit.

MOV AX, 0901H ; BCD 91

SUB AL, 9 ; Minus 9

AAS ; Give AX =0802 h (BCD 82)

(a)

; AL =0011 1001 =ASCII 9

; BL=0011 0101 =ASCII 5

SUB AL, BL ; (9 - 5) Result:

; AL = 00000100 = BCD 04, CF = 0

AAS ; Result:

; AL=00000100 =BCD 04

; CF = 0 NO Borrow required

(b)

; AL = 0011 0101 =ASCII 5

; BL = 0011 1001 = ASCII 9

SUB AL, BL ; (5 - 9) Result:

; AL = 1111 1100 = - 4

; in 2's complement CF = 1

AAS ; Results:

; AL = 0000 0100 =BCD 04

; CF = 1 borrow needed.

- **ADD** Instruction:

These instructions add a number from source to a number from some destination and put the result in the specified destination. The add with carry instruction ADC, also add the status of the carry flag into the result.

The source and destination must be of same type, means they must be a byte location or a word location. If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before adding.

EXAMPLE:

ADD AL, 74H ; Add immediate number 74H to content of AL

ADC CL, BL ; Add contents of BL plus

; carry status to contents of CL.

; Results in CL

ADD DX, BX ; Add contents of BX to contents ; of DX

ADD DX, [SI] ; Add word from memory at ; offset [SI] in DS to contents of DX

; Addition of Un Signed numbers

ADD CL, BL ; CL = 01110011 =115 decimal

; + BL = 01001111 = 79 decimal

; Result in CL = 11000010 = 194 decimal

; Addition of Signed numbers

ADD CL, BL ; CL = 01110011 = + 115 decimal

; + BL = 01001111 = +79 decimal

; Result in CL = 11000010 = - 62 decimal

; Incorrect because result is too large to fit in 7 bits.

- **AND** Instruction:

This Performs a bitwise Logical AND of two operands. The result of the operation is stored in the op1 and used to set the flags.

AND op1, op2

To perform a bitwise AND of the two operands, each bit of the result is set to 1 if and only if the corresponding bit in both of the operands is 1, otherwise the bit in the result I cleared to 0.

AND BH, CL ; AND byte in CL with byte in BH ; result in BH

AND BX, 00FFh ; AND word in BX with immediate ; 00FFH. Mask upper byte, leave ; lower unchanged

AND CX, [SI] ; AND word at offset [SI] in data ; segment with word in CX ; register. Result in CX register.

; BX = 10110011 01011110

AND BX, 00FFh ; Mask out upper 8 bits of BX

; Result BX = 00000000 01011110

; CF =0, OF = 0, PF = 0, SF = 0,

; ZF = 0

- **CALL** Instruction

•Direct within-segment (near or intrasegment)

•Indirect within-segment (near or intrasegment)

•Direct to another segment (far or intersegment)

•Indirect to another segment (far or intersegment)

- **CBW** Instruction - Convert signed Byte to signed word
- **CLC** Instruction - Clear the carry flag
- **CLD** Instruction - Clear direction flag
- **CLI** Instruction - Clear interrupt flag
- **CMC** Instruction - Complement the carry flag
- **CMP** Instruction - Compare byte or word - CMP destination, source.
- **CMPS/CMPSB/**

CMPSW Instruction - Compare string bytes or string words

- **CWD** Instruction - Convert Signed Word to - Signed Double word

Example

- **CALL** Instruction:

This Instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALL's: Near and Far.

A Near CALL is a call to a procedure which is in the same code segment as the CALL instruction.

When 8086 executes the near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. This offset saved on the stack is referred as the return address, because this is the address that execution will returns to after the procedure executes. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure.

A RET instruction at the end of the procedure will return execution to the instruction after the CALL by coping the offset saved on the stack back to IP.

A Far CALL is a call to a procedure which is in a different from that which contains the CALL instruction. When 8086 executes the Far CALL instruction it decrements the stack pointer by two again and copies the content of CS

register to the stack. It then decrements the stack pointer by two again and copies the offset contents offset of the instruction after the CALL to the stack.

Finally it loads CS with segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in segment. A RET instruction at end of procedure will return to the next instruction after the CALL by restoring the saved CS and IP from the stack.

; Direct within-segment (near or intrasegment)

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

; Indirect within-segment (near or intrasegment)

CALL BX ; BX contains the offset of the first instruction of the procedure. Replaces contents of word of IP with contents o register BX.

CALL WORD PTR [BX] ; Offset of first instruction of procedure is in two memory addresses in DS. Replaces contents of IP with contents of word memory location in DS pointed to by BX.

; Direct to another segment- far or intersegment.

CALL SMART ; SMART is the name of the Procedure

SMART PROC FAR; Procedure must be declare as an far

- **CBW** Instruction - CBW converts the signed value in the AL register into an equivalent 16 bit signed value in the AX register by duplicating the sign bit to the left.

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the sign extension of AL.

Example:

; AX = 00000000 10011011 = - 155 decimal

CBW ; Convert signed byte in AL to signed word in AX.

; Result in AX = 11111111 10011011

; = - 155 decimal

- **CLC** Instruction:

CLC clear the carry flag (CF) to 0 This instruction has no affect on the processor, registers, or other flags. It is often used to clear the CF before returning from a procedure to indicate a successful termination. It is also use to clear the CF during rotate operation involving the CF such as ADC, RCL, RCR.

Example:

CLC ; Clear carry flag.

- **CLD** Instruction:

This instruction reset the designation flag to zero. This instruction has no effect on the registers or other flags. When the direction flag is cleared / reset SI and DI will

automatically be incremented when one of the string instruction such as MOVS, CMPS, SCAS, MOVSB and STOSB executes.

Example:

CLD ; Clear direction flag so that string pointers auto increment

- **CLI** Instruction:

This instruction resets the interrupt flag to zero. No other flags are affected. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. This CLI instruction has no effect on the nonmaskable interrupt input, NMI

- **CMC** Instruction:

If the carry flag CF is a zero before this instruction, it will be set to a one after the instruction. If the carry flag is one before this instruction, it will be reset to a zero after the instruction executes. CMC has no effect on other flags.

Example:

CMC; Invert the carry flag.

- **CWD** Instruction:

CWD converts the 16 bit signed value in the AX register into an equivalent 32 bit signed value in DX: AX register pair by duplicating the sign bit to the left.

The CWD instruction sets all the bits in the DX register to the same sign bit of the AX register. The effect is to create a 32- bit signed result that has same integer value as the original 16 bit operand.

Example:

Assume AX contains C435h. If the CWD instruction is executed, DX will contain FFFFh since bit 15 (MSB) of AX was 1. Both the original value of AX (C435h) and resulting value of DX: AX (FFFFC435h) represents the same signed number.

Example:

; DX = 00000000 00000000

; AX = 11110000 11000111 = - 3897 decimal

CWD ; Convert signed word in AX to signed double

; word in DX:AX

; Result DX = 11111111 11111111

; AX = 11110000 11000111 = -3897 decimal.

- **DAA** Instruction - Decimal Adjust Accumulator
- **DAS** Instruction - Decimal Adjust after Subtraction
- **DEC** Instruction - Decrement destination register or memory DEC destination.
- **DIV** Instruction - Unsigned divide-Div source
- **ESC** Instruction

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

Example:

DIV CX ; (Quotient) AX= (DX: AX)/CX

: (Reminder) DX= (DX: AX)%CX

For DIV the dividend must always be in AX or DX and AX, but the source of the divisor can be a register or a memory location specified by one of the 24 addressing modes.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. The SUB AH, AH instruction is a quick way to do.

If you want to divide a word by a word, put the dividend word in AX and fill DX with all 0's. The SUB DX, DX instruction does this quickly.

Example: ; AX = 37D7H = 14, 295 decimal

; BH = 97H = 151 decimal

DIV BH ; AX / BH

; AX = Quotient = 5EH = 94 decimal

; AH = Remainder = 65H = 101 decimal

- **ESC** Instruction - Escape instruction is used to pass instruction to a coprocessor such as the 8087 math coprocessor which shares the address and data bus with an 8086. Instruction for the coprocessor is represented by a 6 bit code embedded in the escape instruction. As the 8086 fetches instruction byte, the coprocessor also catches these bytes from data bus and puts them in its queue. The coprocessor treats all of the 8086 instruction as an NOP. When 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6 bit code. In most of the case 8086 treats ESC instruction as an NOP.
- **HLT** Instruction - HALT processing
- **IDIV** Instruction - Divide by signed byte or word IDIV source
- **IMUL** Instruction - Multiply signed number-IMUL source
- **IN** Instruction - Copy data from a port IN accumulator, port
- **INC** Instruction - Increment - INC destination
- **HALT** Instruction - The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only way to get the processor out of the halt state are with an interrupt signal on the INTR pin or an interrupt signal on NMI pin or a reset signal on the RESET input.
- **IDIV** Instruction - This instruction is used to divide a signed word by a signed byte or to divide a signed double word by a signed word.

Example:

IDIV BL ; Signed word in AX is divided by signed byte in BL

- **IMUL** Instruction - This instruction performs a signed multiplication.

IMUL op ; In this form the accumulator is the multiplicand and op is the multiplier. op may be a register or a memory operand.

IMUL op1, op2 ; In this form op1 is always be a register operand and op2 may be a register or a memory operand.

Example:

IMUL BH ; Signed byte in AL times multiplied by ; signed byte in BH and result in AX.

Example:

; 69 * 14

; AL = 01000101 = 69 decimal

; BL = 00001110 = 14 decimal

IMUL BL ; AX = 03C6H = + 966 decimal

; MSB = 0 because positive result

; - 28 * 59

; AL = 11100100 = - 28 decimal

; BL = 00001110 = 14 decimal

IMUL BL ; AX = F98Ch = - 1652 decimal

; MSB = 1 because negative result

- **IN** Instruction: This IN instruction will copy data from a port to the AL or AX register.

For the Fixed port IN instruction type the 8 – bit port address of a port is specified directly in the instruction.

Example:

IN AL, 0C8H ; Input a byte from port 0C8H to AL

IN AX, 34H ; Input a word from port 34H to AX

A_TO_D EQU 4AH

IN AX, A_TO_D ; Input a word from port 4AH to AX

For a variable port IN instruction, the port address is loaded in DX register before IN instruction. DX is 16 bit. Port address range from 0000H – FFFFH.

Example:

MOV DX, 0FF78H ; Initialize DX point to port

IN AL, DX ; Input a byte from a 8 bit port ; 0FF78H to AL

IN AX, DX ; Input a word from 16 bit port to ; 0FF78H to AX.

- **INC** Instruction:

INC instruction adds one to the operand and sets the flag according to the result. INC instruction is treated as an unsigned binary number.

Example:

; AX = 7FFFh

INC AX ; After this instruction AX = 8000h

INC BL ; Add 1 to the contents of BL register

INC CL ; Add 1 to the contents of CX register.

- **INT** Instruction - Interrupt program
- **INTO** Instruction - Interrupt on overflow.
- **IRET** Instruction - Interrupt return
- **JA/JNBE** Instruction - Jump if above/Jump if not below nor equal.
- **JAE/JNB/JNC** Instructions- Jump if above or equal/ Jump if not below/
  Jump if no carry.
- **JA / JNBE** - This instruction performs the Jump if above (or) Jump if not below or equal operations according to the condition, if CF and ZF = 0.

Example:

(1) CMP AX, 4371H ; Compare by subtracting 4371H ; from AX

JA RUN_PRESS ; Jump to label RUN_PRESS if ; AX above 4371H

(2) CMP AX, 4371H ; Compare (AX – 4371H)

JNBE RUN_PRESS ; Jump to label RUN_PRESS if ; AX not below or equal to 4371H

- **JAE / JNB / JNC** - This instructions performs the Jump if above or equal, Jump if not below, Jump if no carry operations according to the condition, if CF = 0.

Examples:

1. CMP AX, 4371H ; Compare (AX – 4371H)

JAE RUN ; Jump to the label RUN if AX is ; above or equal to 4371H.

2. CMP AX, 4371H ; Compare (AX – 4371H)

JNB RUN_1 ; Jump to the label RUN_1 if AX ; is not below than 4371H

3. ADD AL, BL ; Add AL, BL. If result is with in JNC OK ; acceptable range, continue

- **JB/JC/JNAE** Instruction - Jump if below/Jump if carry/ Jump if not above nor equal
- **JBE/JNA** Instructions- Jump if below or equal / Jump if not above
- **JCXZ** Instruction - Jump if the CX register is zero
- **JE/JZ** Instruction - Jump if equal/Jump if zero
- **JG/JNLE** Instruction- Jump if greater/Jump if not less than nor equal
- **JB/JC/JNAE** Instruction - This instruction performs the Jump if below (or) Jump if carry (or) Jump if not below/ equal operations according to the condition, if CF = 1

Example:

1. CMP AX, 4371H ; Compare (AX – 4371H)

JB RUN_P ; Jump to label RUN_P if AX is ; below 4371H

2. ADD BX, CX ; Add two words and Jump to

JC ERROR ; label ERROR if CF = 1

- **JBE/JNA** Instruction - This instruction performs the Jump if below or equal (or) Jump if not above operations according to the condition, if CF and ZF = 1

Example:

CMP AX, 4371H ; Compare (AX – 4371H)

JBA RUN ; Jump to label RUN if AX is ; below or equal to 4371H

CMP AX, 4371H ; Compare (AX – 4371H)

JNA RUN_R ; Jump to label RUN_R if AX is ; not above than 4371H

- **JCXZ** Instruction:

This instruction performs the Jump if CX register is zero. If CX does not contain all zeros, execution will simply proceed to the next instruction.

Example:

JCXZ SKIP_LOOP ; If CX = 0, skip the process

NXT: SUB [BX], 07H ; Subtract 7 from data value

INC BX ; BX point to next value

LOOP NXT ; Loop until CX = 0

SKIP_LOOP ; Next instruction

- **JE/JZ** Instruction:

This instruction performs the Jump if equal (or) Jump if zero operations according to the condition if ZF = 1

Example:

NXT: CMP BX, DX ; Compare (BX – DX)

JE DONE ; Jump to DONE if BX = DX,

SUB BX, AX ; Else subtract Ax

INC CX ; Increment counter

JUMP NXT ; Check again

DONE: MOV AX, CX; Copy count to AX

Example:

IN AL, 8FH ; read data from port 8FH

SUB AL, 30H ; Subtract minimum value

JZ STATR ; Jump to label if result of ; subtraction was 0

- **JG/JNLE** Instruction:

This instruction performs the Jump if greater (or) Jump if not less than or equal operations according to the condition if ZF =0 and SF = OF

Example:

CMP BL, 39H ; Compare by subtracting ; 39H from BL

JG NEXT1 ; Jump to label if BL is ; more positive than 39H

CMP BL, 39H ; Compare by subtracting ; 39H from BL

JNLE NEXT2 ; Jump to label if BL is not ; less than or equal 39H

- **JGE/JNL** Instruction - Jump if greater than or equal/ Jump if not less than
- **JL/JNGE** Instruction - Jump if less than/Jump if not greater than or equal
- **JLE/JNG** Instruction - Jump if less than or equal/ Jump if not greater
- **JMP** Instruction - Unconditional jump to - specified destination
- **JGE/JNL** Instruction - This instruction performs the Jump if greater than or equal / Jump if not less than operation according to the condition if SF = OF

Example:

CMP BL, 39H ; Compare by the ; subtracting 39H from BL

JGE NEXT11 ; Jump to label if BL is ; more positive than 39H ; or equal to 39H

CMP BL, 39H ; Compare by subtracting ; 39H from BL

JNL NEXT22 ; Jump to label if BL is not ; less than 39H

- **JL/JNGE** Instruction - This instruction performs the Jump if less than / Jump if not greater than or equal operation according to the condition, if SF ≠ OF

Example:

CMP BL, 39H ; Compare by subtracting 39H ; from BL

JL AGAIN ; Jump to the label if BL is more ; negative than 39H

CMP BL, 39H ; Compare by subtracting 39H ; from BL

JNGE AGAIN1 ; Jump to the label if BL is not ; more positive than 39H or ; not equal to 39H

- **JLE/JNG** Instruction - This instruction performs the Jump if less than or equal / Jump if not greater operation according to the condition, if ZF=1 and SF ≠ OF

Example:

CMP BL, 39h ; **Compare by subtracting 39h ; from BL**

JLE NXT1 ; Jump to the label if BL is more ; negative than 39h or equal to 39h

CMP BL, 39h ; Compare by subtracting 39h ; from BL

JNG AGAIN2 ; Jump to the label if BL is not ; more positive than 39h

- **JNA/JBE** Instruction - Jump if not above/Jump if below or equal
- **JNAE/JB** Instruction - Jump if not above or equal/ Jump if below
- **JNB/JNC/JAE** Instruction - Jump if not below/Jump if no carry/Jump if above or equal
- **JNE/JNZ** Instruction - Jump if not equal/Jump if not zero
- **JNE/JNZ** Instruction - This instruction performs the Jump if not equal / Jump if not zero operation according to the condition, if ZF=0

Example:

NXT: IN AL, 0F8H ; Read data value from port

CMP AL, 72 ; Compare (AL – 72)

JNE NXT ; Jump to NXT if AL ≠ 72

IN AL, 0F9H ; Read next port when AL = 72

MOV BX, 2734H ; Load BX as counter

NXT_1: ADD AX, 0002H ; Add count factor to AX

DEC BX ; Decrement BX

JNZ NXT_1 ; Repeat until BX = 0

- **JNG/JLE** Instruction - Jump if not greater/ Jump if less than or equal
- **JNGE/JL** Instruction - Jump if not greater than nor equal/Jump if less than
- **JNL/JGE** Instruction - Jump if not less than/ Jump if greater than or equal
- **JNLE/JG** Instruction - Jump if not less than nor equal to /Jump if greater than
- **JNO** Instruction – Jump if no overflow
- **JNP/JPO** Instruction – Jump if no parity/ Jump if parity odd
- **JNS** Instruction - Jump if not signed (Jump if positive)
- **JNZ/JNE** Instruction - Jump if not zero / jump if not equal
- **JO** Instruction - Jump if overflow
- **JNO** Instruction – This instruction performs the Jump if no overflow operation according to the condition, if OF=0

Example:

ADD AL, BL ; Add signed bytes in AL and BL

JNO DONE ; Process done if no overflow -

MOV AL, 00H ; Else load error code in AL

DONE: OUT 24H, AL ; Send result to display

- **JNP/JPO** Instruction – This instruction performs the Jump if not parity / Jump if parity odd operation according to the condition, if PF=0

Example:

IN AL, 0F8H ; Read ASCII char from UART

OR AL, AL ; Set flags

JPO ERROR1 ; If even parity executed, if not ; send error message

- **JNS** Instruction - This instruction performs the Jump if not signed (Jump if positive) operation according to the condition, if SF=0

Example:

DEC AL ; Decrement counter

JNS REDO ; Jump to label REDO if counter has not ; decremented to FFH

- **JO** Instruction - This instruction performs Jump if overflow operation according to the condition OF = 0

Example:

ADD AL, BL ; Add signed bits in AL and BL

JO ERROR ; Jump to label if overflow occur ; in addition

MOV SUM, AL ; else put the result in memory ; location named SUM

- **JPE/JP** Instruction - Jump if parity even/ Jump if parity
- **JPO/JNP** Instruction - Jump if parity odd/ Jump if no parity
- **JS** Instruction - Jump if signed (Jump if negative)
- **JZ/JE** Instruction - Jump if zero/Jump if equal
- **JPE/JP** Instruction - This instruction performs the Jump if parity even / Jump if parity operation according to the condition, if PF=1