UNIT – 3: Assembly Language Instructions – II

Unconditional & Conditional JUMP instructions:

Conditional JUMP instructions:

- ✓ **JA/JNBE—Jump if above / Jump if not Below or Equal**
  These two mnemonics represent the same instruction. The term above and below are used when referring to the magnitude of unsigned numbers. The Number 0111 is above number 0010. If, after a compare or some other instruction which affects flags, the zero flag and the carry flag are both 0, this instruction will cause execution to jump to a label given in the instruction.

  Example:
  CMP AX, 4371H: compare by subtracting 4371H from AX.
  JA RUN_PRESS: Jump to label RUN_PRESS if AX above 4371H.

- ✓ **JAE/JNB/JNC—Jump if Above or Equal / Jump if not Below / Jump if no Carry**
  These three mnemonics represent the same instruction. The term above and below are used when referring to the magnitude of unsigned numbers. The Number 0111 is above number 0010. If, after a compare or some other instruction which affect flags, the carry flag is 0, this instruction will cause execution to jump to a label given in the instruction. If CF is 1, the instruction will have no effect on program execution.

  Example:
  CMP AX, 4371H: Compare (AX-4371H)
  JAE RUN_PRESS: Jump to label RUN_PRESS if AX above or Equal to 4371H

- ✓ **JB/JC/JNAE—jump if below/Jump if carry/jump if not above or equal**
  These three mnemonics represent the same instruction. The term above and below are used when referring to the magnitude of unsigned numbers. The Number 0111 is above number 0010. If, after a compare or some other instruction which affect flags, the carry flag is a 1, this instruction will cause execution to jump to label given in the instruction. If CF is 0, the instruction will have no effect on program execution.

  Example:
  CMP AX, 4371H: Compare (AX-4371H)
  JB RUN_PRESS: jump to label RUN_PRESS if AX below 4371H

- ✓ **JCXZ—Jump if the CX register is Zero**

This instruction will cause a jump to a label given in the instruction if the CX register contains all 0's. If CX does not contains all 0's, execution will simply proceed to the next instruction.

Example:
JCXZ SKIP_LOOP: If CX=0, skip the process.

✓ **JE/JZ—Jump if Equal/Jump if Zero**
These two mnemonics represent the same instruction. If the Zero flag is set, then this instruction will cause execution to jump to a label given in the instruction. If the Zero flag is not 1, the execution will simply go on to the next instruction after JE or JZ.

Example:
NXT: CMP BX,DX : compare (BX-DX).
JE DONE: Jump to DONE if BX=DX.

✓ **JG/JNLE—Jump if greater/Jump if not less than or equal**
These two mnemonics represent the same instruction. The terms greater and less are used to refer to the relationship of two signed numbers. Greater means more positive. The number 000000111 is greater than the number 11101010, because in signed notation the second number is negative. The instruction usually used after a compare instruction. The instruction will cause a jump to a label given in the instruction if the zero flag is 0 and the carry flag is the same as the overflow flag.

Example:
CMP BL, 39H: Compare by subtracting 39H from BL.
JG NEXT_1: Jump to label if BL more positive than 39H

✓ **JGE/JNL—Jump if greater than or Equal/Jump if not less then**

These two mnemonics represent the same instruction. The terms greater and less are used to refer to the relationship of two signed numbers. Greater means more positive. The number 000000111 is greater than the number 11101010, because in signed notation the second number is negative. The instruction usually used after a compare instruction. The instruction will cause a jump to a label given in the instruction if the sign flag is equal to overflow flag.

Example:

CMP BL, 39H: Compare by subtracting 39H from BL.
JGE NEXT_1: Jump to label if BL more positive than 39H or equal to 39H.

⇨ Unconditional JUMP instructions :

- ✓ **JMP—Unconditional Jump to Specified Destination**

  This instruction will always cause 8086 to fetch its next instruction from the location specified in the instruction rather than from the next location after the JMP instruction. If the destination is in the same code segment as JMP instruction, then only the instruction pointer will be changed to get to the destination location. This is referred to as a near jump. If the destination for the Jump instruction is in the segment with a name different from that of the segment containing the JMP instruction, then both the instruction pointer and the code segment register contents will be changed to get to the destination location.

  Examples:
  JMP Continue:
  Fetch next instruction from address at label CONTINUE. If the Label is in the same segment, an offset coded as a part of the instruction will be added to the instruction pointer to produce the new fetch address.

  JMP BX:
  Replace the Contain of IP with the contain of BX. BX must first be loaded with the offset of the destination instruction in the CS.

- ✓ **JNE/JNZ—Jump if not equal/Jump if not Zero**

  These two mnemonics represent the same instruction. If the zero flag is 0, then this instruction will cause execution to jump to a label given in the instruction. If the zero flag is 1, then execution will simply go on to the next instruction after JNE or JNZ.

  Example:
  NXT: IN AL, 0F8H: Read data value from port.
  CMP AL, 72: Compare (AL-72).
  JNE NXT: Jump to next if AL! = 72

- ✓ **JNO—Jump if no Overflow**

  The Overflow flag will be set if the result of some signed arithmetic operation is too large to fit in the destination register or memory location. The JNO instruction will cause the 8086 to jump to a destination given in the instruction if the overflow flag is not set.

Example:
ADD AL, BL : add signed bytes in AL and BL.
JNO DONE: Process done if no overflow.

- ✓ **JNP/JPO—Jump if no parity/Jump if Parity odd**

  If the number of 1st left in the lower 8 bit of a data word after an instruction which affects the parity flag is odd, then the parity flag will be 0. The destination address must be in range of -128 to+127 bytes from the address of the instruction after the JNP/JPO instruction.

  Example:
  IN AL, 0F8H: Read ASCII character from UART
  OR AL, AL: set flags
  JPO ERROR1: Even parity excepted, send error message if parity found odd

- ✓ **JNS—JUMP if not Signed(Jump is Positive)**

  This instruction will cause execution to jump to a specified destination if the sign flag is 0. Since a 0 in the sign flag indicates a positive signed number, you can think of this instruction as saying "jump if positive". If the sign flag is set, indicating negative signed result, execution will simply go on to the next instruction after JNS.

  Example:
  DEC AL: decrement counter.
  JNS REDO: jump to label REDO if counter has not decremented to FFH.

- ✓ **JO—Jump if Overflow**

  The JO instruction will cause the 8086 to jump to a destination given in the instruction if the overflow flag is set. The Overflow flag will be set if the magnitude of the result produced by the some signed arithmetic operation is too large to fit in the destination register or memory location.

  Example:

  ADD AL, BL: add signed bytes in AL and BL.
  JO ERROR: jump to a label ERROR if overflow from add.

- ✓ **JP/JPE—Jump if parity/jump if parity Even**

  If the number of 1st left in the lower 8 bit of a data word after an instruction which affects the parity flag is even, then the parity flag will be set. If the Parity flag is set, the JP/JPE instruction will cause execution to jump to specified destination address.

Example:

IN AL, F8H: Read ASCII character from UART
OR AL, AL: set flags
JPE ERROR2: Odd parity expected, send error message if parity found even

- ✓ **JS—Jump if Signed(Jump is Negative)**
  This instruction will cause execution to jump to a specified destination if the sign flag is set. Since a 1 in the sign flag indicates a Negative signed number, you can think of this instruction as saying "jump if negative" or "Jump if minus". If the sign flag is 0, indicating a positive signed result, execution will simply go on to the next instruction after JS.

  Example:
  ADD BL, DH: Add signed byte in DH to Signed byte in BL.
  JS TOO_COLD: Jump to label TOO_COLD if result of addition is Negative Number.

Unconditional & Conditional LOOP instructions

- ✓ **LOOP—Jump to specified Label if CX != 0 after Auto decrement—Loop Label**
  This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded in to CX. Each time the loop instruction executes, CX is automatically decremented by 1. If the CX is not 0, execution will jump to destination specified by a label in the instruction. If CX=0 after the auto decrement, execution will simply go on to the next instruction after LOOP. The Destination address for the jump must be in the range of -128 to +127 bytes from the address of the instruction after the LOOP instruction. LOOP affects no flags.

  Example:
  MOV BX, OFFSET PRICE: Point BX at first element in array.
  MOV CX, 40: Load CX with number of element in array.
  NEXT: MOVE AL, [BX]: get element from array.
  ADD AL, 07H: Add Correction Factor.
  DAA: Decimal Adjust Result.
  MOV [BX], AL: Put result back in array.
  INC BX:
  LOOP NEXT: Repeat until all elements Adjusted.

- ✓ **LOOPE/LOOPZ—Loop while CX != 0 and ZF=1**

LOOPE/ and LOOPZ are two Mnemonics for the same instruction. This instruction is used to repeat a group of instruction some number of times or until the zero flag becomes 0. The number of time the instruction sequence is to be repeated is loaded in to CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX != 0 and ZF = 1, the execution will jump to a destination specified by a label in the instruction. IF CX = 0 after the auto decrement or if ZF = 0, execution will simply go on to the next instruction after LOOPE/LOOPZ.

Example:
MOV BX, OFFSET ARRAY: Point BX to just before the start of array
DEC BX:
MOV CX, [BX] : Put number of array element in to CX
NEXT: INC BX : Point to the next element in the array
CMP[BX], OFFH : Compare Array element with FFH.

✓ **LOOPNE/LOOPNZ—Loop while CX !=0 and ZF=0**

LOOPNE/ and LOOPNZ are two Mnemonics for the same instruction. This instruction is used to repeat a group of instruction some number of times or until the zero flag becomes 1. The number of time the instruction sequence is to be repeated is loaded in to Count Register CX. Each time the LOOPNE and LOOPNZ instruction execute, CX is automatically decremented by 1. IF CX != 0 and ZF = 0, the execution will jump to a destination specified by a label in the instruction. IF CX = 0 after the auto decrement or if ZF = 1, execution will simply go on to the next instruction after LOOPNE/LOOPNZ.

Example:

MOV BX, OFFSET ARRAY: Point BX to just before the start of array
DEC BX:
MOV CX, [BX]: Put number of array element in to CX
NEXT: INC BX: Point to the next element in the array
CMP[BX], ODH : Compare Array element with 0DH.
LOOPNE NEXT

String Instructions

✓ **REP / REPE / REPZ / REPNE / REPNZ –**
(Prefix) Repeat String Instruction until Specified Conditions Exist

REP is a Prefix which is written before one of the string instructions. It will cause the CX register to be decremented and the string instruction to be repeated until CX=0. The instruction REP MOVSB , for example, will continue the copy string bytes until the number of bytes loaded into CX has been copied.

REP and REPZ are two mnemonics for the same Prefix. They stand for repeat if equal and Repeat if zero, respectively. REPE or REPZ is often used with the Compare String Instruction or with the Scan String Instruction. REPE or REPZ will cause the String Instructions to be repeated as long as the compared bytes or words are equal (ZF=1) and CX is not yet down to zero. In other Words, There are two Conditions that will stops the Repetition: CX=0 or String bytes or words not equal.

Example:
REPE CMPSB: Compare string bytes until end of string or until string bytes not equal.

✓ **MOVS / MOVB / MOVSW**

This Instruction copies a byte or a word from a location in the data segment to a location in the extra segment. The Offset of the Source byte or word in the data segment must be in the SI register. The offset of the destination in the extra segment must be contained in the DI register. For multiple bytes multiple word moves, the number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or word is moved, SI and DI are automatically adjusted to point to the next source and next destination. If the direction flag is 0, then SI and DI will be incremented by 1 after a byte moved and incremented by 2 after the word move. If the DF is 1, the SI and DI decremented by 1 after a byte move and decremented by 2 after a word move. MOVS affect no flag.

Example:
MOV SI, OFFSET SOURSE_STRING: load offset of source string in DS into SI.
MOV DI, OFFSET DESTINATION_STRING: Load offset of start of destination string ES into DI.

✓ **CMPS / CMPSB / CMPSW**

A string is the series of the same type of data items in sequential memory location. The CMPS instruction can be used to compare a byte in one string with a byte in another string or to compare a word in a one string with a word in another string. SI is used to store the offset of a byte or word in a source string, and DI is used to hold the offset of byte or word in the other String. The comparison is done by subtracting the byte or word pointed to by DI from the byte or word pointed to by SI.

The string pointed to by DI must be in the extra segment. The string pointed to by SI must be in the data segment.

Example:
MOV SI, OFFSET FIRST_STRING: point SI at source string.

MOV DI, OFFSET DESTINATION_STRING: point DI at destination String.
CLD: DF cleared. So SI and DI will auto increment after compare.

MOV CX, 100 : Put number of string element in CX.
REPE CMPSB: Repeat the comparison of string bytes until end of the string or until compared bytes is not equal.

- ✓ **SCAS / SCASB / SCASW**

SCAS compares a byte in AL or a word in AX with a byte or word pointed to by DI in ES. Therefore, the string to be scanned must be in extra segment, and DI must contain the offset of the byte or word to be compared. If the Direction flag is cleared (0), then DI will be incremented after SCAS. If the direction flag is set (1), then DI will be decremented after SCAS. For byte Strings, DI will be incremented or decremented by 1, and for word string, DI will be incremented or decremented by 2.

Example:
MOV DI, OFFSET TEXT_STRING: scan a text string of 80 character of a carriage return, **0DH**: Put offset of string into DI.
MOV AL, 0DH: Byte to be scanned for into AL.
MOV CX, 80: CX is used to element counter.
CLD: Clear DF so DI auto increments.
REPNE SCAS TEXT_STRING: Compare byte in string with byte in AL.

- ✓ **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. If the direction flag is cleared (0), SI will automatically be incremented to point to the next element of the string. For a string of bytes, SI will be incremented by 1. For a string of words, SI will be incremented by 2. If the Direction flag (DF) is set (1), Si will be automatically decremented to the point to the next string element. For a byte string, SI will be decremented by 1, and for a word string, SI will be decremented by 2.

Example:
CLD: Clear Direction flag so SI is auto incremented.
MOV SI, OFFSET SOURCE_STRING: Point SI at start of String.
LODS SOURCE_STRING: Copy byte or word from string AL or AX.

✓ **STOS / STOSB / STOSW**

The STOS instruction copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI. In effect, it replaces a string element with a byte from AL or a word from AX. After the copy, DI is automatically incremented or decremented to point to the next string element in memory. If the Direction flag (DF) is cleared, then DI will automatically be incremented by 1 for a byte string or incremented by 2 for a word string. If the direction flag is set, DI will automatically be decremented by 1 for a byte string or decremented by 2 for a word string.

Processor Control Instruction

✓ **STC**
STC- set the carry flag to a 1
STC does not affect any other flags.

✓ **CLC**
CLC-Clear the Carry Flag (CF)
This instruction resets the carry flag to 0. No other flags are affected.

✓ **CMC**
CMC-Complement the Carry Flag
If the Carry flag (CF) is a before this instruction, it will be set to a 1 after the instruction. If the carry flag is 1 before this instruction, it will be reset to a 0 after the instruction executes. CMC affects no other Flags.

✓ **STD**
STD-set the direction flag to a 1
STD is used to set the direction flag to a 1 so that SI and/or DI will automatically be decremented to the point to the next string element when one of the string instructions executes. If the direction flag is set, SI and/or DI will be decremented by 1 for byte strings, and 2 for word strings.

✓ **CLD**
CLD-Clear Direction Flag
This Instruction resets the direction flag to 0. No other flags are affected. If the direction flag is reset, SI and DI will automatically be incremented when one of the string instructions, such as MOVS, CMPS, or SCAS, executes.

✓ **STI**
STI-set interrupt flag (IF)

Setting the interrupt flag to 1 enables the INTR interrupt input for 8086. The instruction will not take effect until after the next instruction after STI. When the INTR input is enabled, an interrupt signal on this input will then cause the 8086 to interrupt program execution, push the return address and flags on stack, and execute an interrupt service procedure.

- ✓ **CLI**

  CLI-Clear interrupt Flag

  This instruction resets the interrupt flag to 0. No other flags are affected. If the interrupt flag is reset, the 8086will not respond to an interrupt signal on its INTR input.

- ✓ **NOP**

  NOP-Perform NO Operation

  This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction. NOP Affect NO Flag. The NOP instruction can be used to increase the delay of delay loop. When hand coding, a NOP can also be used to hold a place in a program for an instruction that will be added later.