## 7- CONTROL FLOW AND THE JUMP INSTRUCTIONS:

The control flow relates to altering the execution path of instructions in a program. For example, a control flow decision may cause a sequence of instructions to be repeated or a group of instructions to not be executed at all. The jump instruction is provided in the 8086 instruction set for implementing control flow operations.

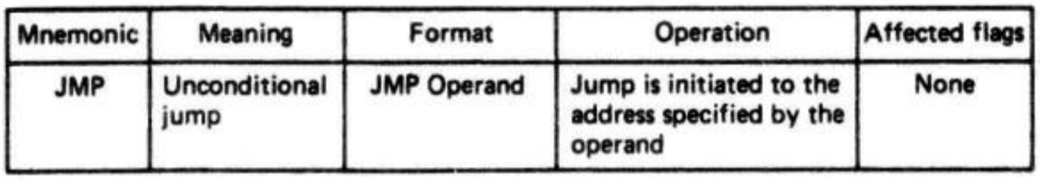
In the 8086 architecture, the code segment register and instruction pointer keep track of the next instruction to be fetched for execution. Thus, to initiate a change in control flow, a jump instruction must change the contents of these registers.

In this way, execution continues at an address other than that of the next sequential instruction. That is, a jump occurs to another part of the program.

### Unconditional and Conditional Jump

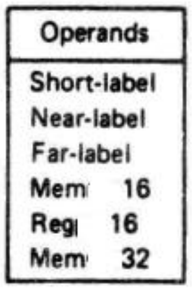
- **Unconditional Jump Instruction:**

Figure (a) shows the unconditional jump instruction of the 8086, together with its valid operand combinations in Fig.(b).



| Mnemonic | Meaning | Format | Operation | Affected flags |
|---|---|---|---|---|
| JMP | Unconditional jump | JMP Operand | Jump is initiated to the address specified by the operand | None |

(a)



| Operands |
|---|
| Short-label |
| Near-label |
| Far-label |
| Memptr16 |
| Regptr16 |
| Memptr32 |

(b)

**Unconditional Jump types:**

**a)** Intrasegment: this is a jump within the current segment .This type of jump is achieved by just modifying the value in IP.

1- Short Jump: Format → JMP **short Label (8 bit)**
2- Near Jump: Format → JMP **near Label (16 bit)**

**Ex:** Consider the following example of an unconditional jump instruction:

**JMP 1234H**

It means jump to address 1234H. However, the value of the address encoded in the instruction is not 1234H. Instead, it is the difference between the incremented value in IP and 1234H. This offset is encoded as either an 8-bit constant (short label)or a 16-bit constant (near label), depending on the size of the difference.

3- Mem.16: Format → **JMP Mem.16**
4- Reg.16:: Format → **JMP Reg.16**

the jump-to address can also be specified indirectly by the contents of a memory location or the contents of a register, corresponding to the Mem.16 and Reg.16 operand, respectively. Just as for the Near-label operand, they both permit a jump to any address in the current code segment.
**Ex**:

**JMP BX**

uses the contents of register BX for the offset in the current code segment that is, the value in BX is copied into IP.
To specify an operand as a pointer to memory, the various addressing modes of 8086 can be used, For instance:

**JMP [BX]**

uses the contents of BX as the offset address of them memory location that contains the value of IP (Mem.16 operand).

Ex:

**JMP [SI]**

will replace the IP with the contents of the memory locations pointed by DS:SI and DS:SI+1

**b)** Intersegment: this is a jump out of the current segment. This type of jump requires modification of the contents of both CS and IP.

1- Far Jump: Format → **JMP far Label (32 bit label)**

Ex:

**JMP label**(4-byteaddress)

**JMP 1234:5678**

Transfers control to another part of the program. 4-byte address may be entered in this form: 1234h:5678h, first value is a segment second value is an offset.

2- Mem.32: Format → **JMP Mem.32**

An indirect way to specify the offset and code-segment address for an intersegment jump is by using the Mem.32 operand. This time the four consecutive memory bytes starting at the specified address contain the offset address and the new code segment address respectively.

**Ex:**

**JMP DWORD[DI]**

It uses the contents of DS and DI to calculate the address of the memory location that contains the first word of the pointer that identifies the location to which the jump will take place. The two word pointer starting at this address is read into IP and CS to pass control to the new point in the program.

## Conditional Jump

The second type of jump instruction performs conditional jump operations. Figure (a) shows a general form of this instruction; Fig. (b) is a list of each of the conditional jump instructions in the 8086's instruction set.

| Mnemonic | Meaning | Format | Operation | Flags affected |
|---|---|---|---|---|
| Jcc | Conditional jump | Jcc Operand | If the specified condition cc is true the jump to the address specified by the operand is initiated; otherwise the next instruction is executed. | None |

(a)

| Mnemonic | Meaning | Condition |
|---|---|---|
| JA | above | CF = 0 and ZF = 0 |
| JAE | above or equal | CF = 0 |
| JB | below | CF = 1 |
| JBE | below or equal | CF = 1 or ZF = 1 |
| JC | carry | CF = 1 |
| JCXZ | CX register is zero | (CF or ZF) = 0 |
| JE | equal | ZF = 1 |
| JG | greater | ZF = 0 and SF = OF |
| JGE | greater or equal | SF = OF |
| JL | less | (SF xor OF) = 1 |
| JLE | less or equal | ((SF xor OF) or ZF) = 1 |
| JNA | not above | CF = 1 or ZF = 1 |
| JNAE | not above nor equal | CF = 1 |
| JNB | not below | CF = 0 |
| JNBE | not below nor equal | CF = 0 and ZF = 0 |
| JNC | not carry | CF = 0 |
| JNE | not equal | ZF = 0 |
| JNG | not greater | ((SF xor OF) or ZF) = 1 |
| JNGE | not greater nor equal | (SF xor OF) = 1 |
| JNL | not less | SF = OF |
| JNLE | not less nor equal | ZF = 0 and SF = OF |
| JNO | not overflow | OF = 0 |
| JNP | not parity | PF = 0 |
| JNS | not sign | SF = 0 |
| JNZ | not zero | ZF = 0 |
| JO | overflow | OF = 1 |
| JP | parity | PF = 1 |
| JPE | parity even | PF = 1 |
| JPO | parity odd | PF = 0 |
| JS | sign | SF = 1 |
| JZ | zero | ZF = 1 |

(b)

Note that each of these instructions tests for the presence or absence of certain status conditions.

For instance, the jump on carry (JC) instruction makes a test to determine if carry flag (CF) is set. Depending on the result of the test, the jump to the location specified by its operand either takes place or does not. If CF equals 0, the test fails and execution continues with the instruction at the address following the JC instruction. On the other hand, if CF equals 1, the test condition is satisfied and the jump is performed.

Note that for some of the instructions in Fig. (b), two different mnemonics can be used. This feature can be used to improve program readability. That is, for each occurrence of the instruction in the program, it can be identified with the mnemonic that best describes its function. For instance, the instruction jump on parity (JP) or jump on parity even (JPE) both test parity flag (PF) for logic 1. Since PF is set to one if the result from a computation has even parity, this instruction can initiate a jump based on the occurrence of even parity.

The reverse instruction JNP/JPO is also provided. it can be used to initiate a jump based on the occurrence of a result with odd instead of even parity.

In a similar manner, the instructions jump if equal (JE) and jump if zero (JZ) serve the same function. Either notation can be used in a program to determine if the result of a computation was zero.

All other conditional jump instructions work in a similar way except that they test different conditions to decide whether or not the jump is to take place. Examples of these conditions are: the contents of CX are zero, an overflow has occurred, or the result is negative.

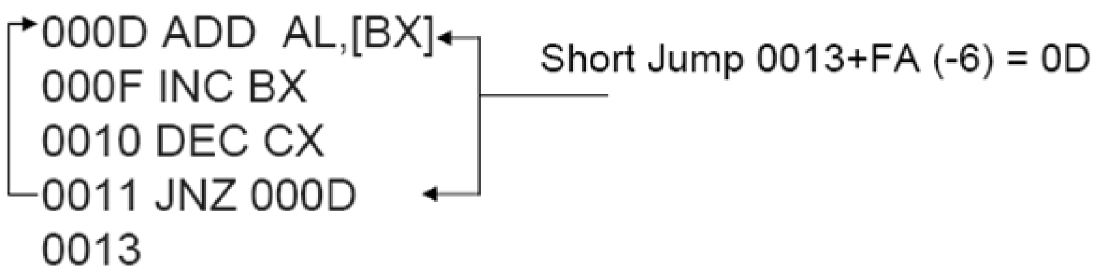
To distinguish between comparisons of signed and unsigned numbers by jump instructions, two different names, which seem to imply the same, have been devised.

They are above and below for comparison of unsigned numbers, and less and greater for comparison of signed numbers. For instance, the number ABCDH is above the number 1234H if they are considered to be unsigned numbers. On the other hand, if they are treated as signed numbers, ABCDH is negative and 1234H is positive. Therefore. ABCDH is less than 1234H.

So the Conditional jumps are always short jumps in the 8086 and these instructions will cause a jump to a label given in the instruction if the desired condition(s) occurs in the program before the execution of the instruction.

- Conditional Jump is a two byte instruction.
- In a jump backward the second byte is the 2's complement of the displacement value.
- To calculate the target the second byte is added to the IP of the instruction right after the jump.

Example :

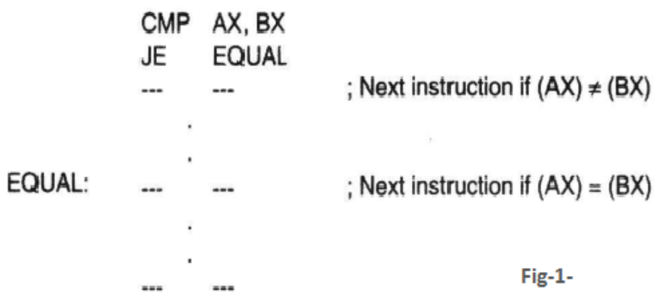


```
000D ADD  AL,[BX]
000F INC BX
0010 DEC CX
0011 JNZ 000D
0013
```

The JNZ instruction will encoded as :**75FA H**

- **Branch Program Structure**

For example, Fig -1- shows a program structure that implements to confirm whether or not two values are equal.



```
         CMP  AX, BX
         JE   EQUAL
         ---  ---        ; Next instruction if (AX) ≠ (BX)
            .
            .
EQUAL:   ---  ---        ; Next instruction if (AX) = (BX)
            .
            .
         ---  ---
```

Fig-1-

First, the CMP instruction subtracts the value in BX from that in AX and adjusts the flags based on the result. Next, the jump on equal instruction tests the zero flag to see if it is 1. If ZF is 1, it means that the contents of AX and BX are equal and a jump is made to the label EQUAL. Otherwise, if ZF is 0, which means that the contents of AX and BX are not equal, the instruction following the JE instruction is executed.

Another example use of a conditional jump is to branch based on the setting of a specific bit in a register. When this is done, a logic operation is normally used to mask off the values of all of the other bits in the register.

We may want to mask off all bits of the value in AL other than bit 2 and then make a conditional jump if the unmasked bit is logic 1.

```
            AND   AL, 04H
            JNZ   BIT2_ONE
            ---   ---          ; Next instruction if B2 of AL = 0
                .
                .
            ---   ---
BIT2_ONE:   ---   ---          ; Next instruction if B2 of AL = 1
                .
                .
            ---   ---
```

Fig-2-

This operation can be done with the instruction sequence in Fig-2-. First, the contents of AL are ANDed with 04H to give

(AL) = XXXXXXXX B · 00000100 B = 00000X00 B

Now the content of AL is 0 if bit 2 is 0 and the resulting value in the ZF is 1. On the other hand, if bit 2 is 1, the content of AL is nonzero and ZF is 0. Remember, we want to make the jump to the path when bit 2 is 1. Therefore, the conditional test is made with a jump on not zero instruction. When ZF is 1, the next instruction is executed, but if ZF is 0, the JNZ instruction passes control to the instruction identified by the label BIT2_ONE.

Let us look at how to perform this exact same branch operation in another way.

Instead of masking off all of the bits in AL, we could simply shift bit 2 into the carry flag and then make a conditional jump if CF equals 1. The program structure in Fig.-3- uses this method to test for logic 1 in bit 2 of AL. Notice that this implementation takes one extra instruction.

```
            MOV   CL,03H
            SHR   AL, CL
            JC    BIT2_ONE
            ---   ---          ; Next instruction if B2 of AL = 0
               .
               .
            ---   ---
BIT2_ONE:   ---   ---          ; Next instruction if B2 of AL = 1
               .
               .
            ---   ---
```

Fig-3-

- **The Loop Program Structure**

In many practical applications, we frequently need to repeat a part of the program many times-that is, a group of instructions may need to be executed over an over again until a condition is met. This type of control flow program structure is known as a loop.

Figure below show a typical assembly language implementation of a loop program structure.

```
         MOV   CL,COUNT    ;Set loop repeat count
AGAIN:   ---   ---         ;1st instruction of loop
         ---   ---         ;2nd instruction of loop
          .     .                .
          .     .                .
          .     .                .
         ---   ---         ;nth instruction of loop
         DEC   CL          ;Decrement repeat count by 1
         JNZ   AGAIN       ;Repeat from AGAIN if (CL) ≠ 00H or (ZF) = 0
         ---   ---         ;First instruction executed after the loop is
                           ;complete, (CL) = 00H, (ZF) = 1
```

Here we see that the sequence of instructions from label AGAIN to the conditional jump instruction JNZ represents the loop.

Note that the label for the instruction that is to be jumped to is located before the jump instruction that makes the conditional test. In this way, if the test result is true, program control returns to AGAIN and the segments of the program repeat. This continues until the condition specified by JNZ is false.

Before initiating the program sequence, a parameter must be assigned to keep track of how many times the sequence of instructions has been repeated. This parameter. called the count, is tested each time the sequence is performed to verify whether or not it is to be repeated again. Note that register CL, which is used for the conditional test, is initialized with the value COUNT by a MOV instruction prior to entering the loop.

For example, to repeat a part of a program 10 times, we begin by loading the count register CL with COUNT equal to 0AH, then the operation of instructions 1 through n of the loop is performed. Next, the value in CL is decremented by 1 to indicate that the instructions in the loop are done; and after decrementing CL, ZF is tested with the JNZ instruction to see if it has reached 0. That is, the question "Should I repeat again?" is asked with software. If ZF is 0, which means that (CL) ≠ 0, the answer is yes, repeat again, and program control is returned to the instruction labeled AGAIN and the loop instruction sequence repeats. This continues until the value in CL reaches zero to identify that the loop is done. When this happens, the answer to the conditional test is no, do not repeat again, and the jump is not taken. Instead, the instruction following JNZ AGAIN is executed.

**Applications Using the Loop and Branch Software Structures**

As a practical application of the use of a conditional jump operation, let us write a program known as a block—move program. The purpose of this program is to move a block of N consecutive bytes of data starting at offset address BLK1ADDR in memory to another block of memory locations starting at offset address BLK2ADDR. We will assume that both blocks are in the same data segment, whose starting point is defined by the data segment value DATASEGADDR.

To solve this problem. It has four operations. The first operation is initialization.

Initialization involves establishing the initial address of the data segment. Loading the DS register with the value DATASEGADDR does this. Furthermore, source index register SI and destination index register DI are initialized with offset addresses BLK1ADDR and BLK2ADDR respectively. In this way, they point to the beginning of the source block and the beginning of the destination block, respectively. To keep track of the count of bytes transferred, register CX is initialized with N, the number of bytes to be moved. This leads us to the following assembly language statements:

```
MOV AX, DATASEGADDR
MOV DS, AX
MOV SI, BLK1ADDR
MOV DI, BLK2ADDR
MOV CX, N
```

Note that DS cannot be directly loaded by immediate data with a MOV instruction. Therefore, the segment address is first loaded into AX and then moved to DS. SI, DI, and CX load directly with immediate data.

The next operation that must be performed is the actual movement of data from the source block of memory to the destination block. The offset addresses are already loaded into SI and DI; therefore, move instructions that employ indirect addressing are used to accomplish the data—transfer operation. Remember that the 8086 does not allow direct memory-to-memory moves. For this reason, AX is used as a temporary storage location for data. The source byte is moved into AX with one instruction, and then another instruction moves it from AX to the destination location.

Thus, the data move is accomplished by the following instructions:

```
NXTPT:   MOV AH, [SI]
         MOV [DI] , AH
```

Note that for a byte move, only the higher eight bits of AX are used. Therefore, the operand is specified as AH instead of AX.

Now the pointers in SI and DI must be updated so that they are ready for the next byte-move operation. Also, the counter must be decremented so that it corresponds to the number of bytes that remain to be moved. These updates are done by the following sequence of instructions:

```
INC SI
INC DI
DEC CX
```

The test operation involves determining whether or not all the data points have been moved. The contents of CX represent this condition. When its value is not 0, there still are points to be moved, whereas a value of 0 indicates that the block move is complete. This 0 condition is reflected by 1 in ZF. The instruction needed to perform this test is

```
JNZ NXTPT
```

Here NXTPT is a label that corresponds to the first instruction in the data move operation. The last instruction in the program is a halt (HLT) instruction to indicate the end of the block move operation.

**Ex:** Write a program to add (50)H numbers stored at memory locations start at 4400:0100H , then store the result at address 200H in the same data segment. Ignored the carry.

**Solution:**

```
                MOV AX , 4400H
                MOV DS , AX
                MOV CX , 0050H ← counter
                MOV BX , 0100H ←  offset
    label → Again: ADD AL, [BX]
                INC BX
                DEC CX
                 JNZ Again
                MOV [0200], AL
```

**Ex**: Write a program to move a block of 100 consecutive bytes of data starting at offset address 400H in memory to another block of memory locations starting at offset address 600H. Assume both block at the same data segment F000H.

**Solution:**

```
        MOV AX, F000H
        MOV DS, AX
        MOV SI, 0400H
        MOV DI, 0600H
        MOV CX, 64H   →   64 Hexadecimal = 100 Decimal
LableX: MOV AH, [SI]
        MOV [DI], AH
        INC SI
        INC DI
        DEC CX
        JNZ LableX
        HLT           →   End of program
```