Branch instructions

By default, the CPU loads and executes programs sequentially. But the current instruction might be conditional, meaning that it transfers control to a new location in the program based on the values of CPU status flags (Zero, Sign, Carry, etc.). Assembly language programs use conditional instructions to implement high-level statements such as IF statements and loops. Each of the conditional statements involves a possible transfer of control (jump) to a different memory address. A transfer of control, or branch (jump), is a way of altering the order in which statements are executed. There are two basic types of jumps (branch):

1.Unconditional jumps:

This is performed by the JMP instruction. It is just like a goto, it always makes the jump. In this case, the execution control is transferred to the specified location independent of any status or condition. Transfer of control may be forward, to execute a new set of instructions or backward, to re-execute the same steps.

The basic instruction that transfers control to another point in the program is JMP. The syntax of JMP instruction:

```
        JMP    label   ; go to ' label'
        ……
label:  inst.
```

To declare a label in your program, just type its name and add ":" to the end, label can be any character combination but it cannot start with a number, you remember the syntax of any instruction as:

Label can be declared on a separate line or before any other instruction, for example:

```
x1:
MOV AX, 1
```

or can write as:

```
x1: MOV AX, 1
```

Types of Jump Addresses

There are three types of addresses that are distinguished by their distance from currently executing address:

a) A short address , here jump operation within the same segment limited to a distance of 00 h to FF h or -128 to +127 bytes. This means It can only move up or down 128 bytes in memory. The advantage of this type is that it uses less memory than the others. The conditional jumps can only be short address.

b) A near address , also here jump operation within the same segment limited to a distance of 0000 h to FFFF h or -32,768 to +32,767 bytes. Means this address is 16 bits (2 bytes).

c) A far address , The previous two jumps were used to jump within a segment, but this far jump allows control to move may be at a distance over 32k or in another code segment. Both segment and offset must be given to a far address (two bytes for segment and two bytes for offset). The CALL and RET instructions for example are related to this type.

2.Conditional jumps : a conditional jump (***Jcond***) may or may not make the jump depending on the flags in the FLAGS register. If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction by breaking the sequential flow and they do it by changing the offset value in IP. There are numerous conditional jump instructions, depending upon the condition and data**.** The basic syntax of conditional jumps instruction:

Jcond **label**

cond refers to a flag condition identifying the state of one or more flags. CPU status flags are most commonly set by arithmetic, comparison, and logical (Boolean) instructions. Conditional jump instructions evaluate the flag states, using them to determine whether or not jumps should be taken.

The conditional jump instructions are divided in three groups, first group just test single flag as shown in table 1, second compares numbers as signed used for arithmetic operations as shown in table 2, and third compares numbers as unsigned used for logical operations as shown in table 3.

Table 1: Conditional jump instructions that test single flag

Instruction	Description	Condition	Opposite Instruction
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

Table 2:Conditional Jump instructions based on signed numbers for arithmetic operations

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JG , JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

<> - sign means not equal.

Table 3:Conditional Jump instructions based on unsigned numbers for logical operations

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JA , JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB
JBE , JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE, JA

Ex1: The CMP instruction in the following example compares AX with Zero. The JZ (jump if Zero) instruction jumps to label **L1** if the Zero flag was set by the CMP instruction:

```
CMP AX,0
JZ L1        ; JUMP IF ZF = 1  or  JE  L1
...
L1:
```

EX2. The CMP instruction in the following example compares AX with 4. The JG (jump if AX>4) instruction jumps to label **L2** else continue execute the sequential flow instructions. Here, the jump is taken because AX is greater than 4:

```
MOV  AX,5
CMP   AX,4    ; AX – 4
JG  L2          ; JUMP IF GREATER  (AX> 4)
…..
L2:
```

Loop Instructions

There are several instructions designed to implement for –like loops. Table 4 show the LOOP instructions. The basic syntax of LOOP instructions:

LOOP label

Means decrease CX, jump to label if CX not zero.

Algorithm:

- CX = CX – 1
- if CX <> 0 then
  jump

else

no jump, continue

Table 4: show the LOOP instructions.

instruction	operation and jump condition	opposite instruction
LOOP	decrease cx, jump to label if cx not zero.	DEC CX and JCXZ
LOOPE	decrease cx, jump to label if cx not zero and equal (zf = 1).	LOOPNE
LOOPNE	decrease cx, jump to label if cx not zero and not equal (zf = 0).	LOOPE
LOOPNZ	decrease cx, jump to label if cx not zero and zf = 0.	LOOPZ
LOOPZ	decrease cx, jump to label if cx not zero and zf = 1.	LOOPNZ
JCXZ	jump to label if cx is zero.	OR CX, CX and JNZ

Loops are basically the same jumps, it is possible to code loops without using the loop instruction, by just using conditional jumps and compare, and this is just what loop does. All loop instructions use CX register to count steps, as you know CX register has 16 bits and the maximum value it can hold is 65535 or FFFF, however with some agility it is possible to put one loop into another, and another into another two, and three and etc... and receive a nice value of 65535 * 65535 * 65535 .... till infinity.... or the end of ram or stack memory. It is possible store original value of cx register using ***push cx*** instruction and return it to original when the internal loop ends with ***pop cx***.

For example of loop , the following code snippet can be used for executing the loop-body 10 times (loop-body means the instructions that you want to repeated):

…..

```
Mov cx,10
Lab:
<Loop-body>
Loop   Lab
```

Write a program in assembly language to find min and max between two numbers (N1 and N2) assume these numbers (N1 and N2) are signed bytes

ملاحظة : عند ذكر متغير مثلا هنا (N1,N2,min, max) يجب تعريفه.

```
Org 100 h
Jmp start
N1  db  0b1h
N2  db   54h
max db  ?
min  db  ?
Start: Mov  al,N1
        Mov  bl,N2
       Cmp al,bl
       Jge  zzzz
       Mov  min,al
       Mov  max , bl
       Jmp eee
zzzz: mov  max,al
       mov  min,bl
eee:   ret
```

Note : b1h= -79 , 54h = 84

Flag manipulation and Processor Control Instructions

The Flag manipulation and Processor Control Instructions control the functioning of the available hardware inside the processor chip. These are categorized into 2 types:

a) Flag manipulation instructions
b) Machine control instructions.

The flag manipulation instructions directly modify same of the flags of 8086. The flag manipulation instructions and their functions are as follows in table 1:

Table 1: Flag manipulation instructions

CLC	**Clear Carry flag**	**CF=0**
CMC	**Complement Carry flag. Inverts value of CF.**	**If CF = 1 then CF = 0 If CF = 0 then CF = 1**
STC	**STC Carry flag**	**CF=1**
CLD	**Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW.**	**DF=0**
STD	**Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW.**	**DF=1**
CLI	**Clear Interrupt enable flag. This disables hardware interrupts.**	**IF=0**
STI	**Set Interrupt enable flag. This enables hardware interrupts.**	**IF=1**

These instructions modify the carry (CF), Direction (DF) and interrupt (IF) flags directly. The DF and IF, which may be modified using the flag mainipulation instructions, further control the processor operation; like interrupt responses and auto increment or auto-decrement modes. Thus the respective instructions may also be called as machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions. No direct instructions are available for modifying the status flags except carry flags.

The machine control instructions supported by 8086/8088 are listed as follows along with their functions. They don't require any operand:

Table 2: Machine control instructions.

NOP	**No operation**
HLT	**Halt the processor after interrupt is set**
WAIT	**Wait for TEST input pin to go low (active)**
ESC opcode mem/ reg	**Used to pass instruction to a coprocessor which shares the address and data bus with the 8086**
LOCK	**Lock bus during next instruction**