8086 assembler tutorial for beginners (part 7)

Program flow control

Controlling the program flow is a very important thing, this is where your program can make decisions according to certain conditions.

- **unconditional jumps**

  The basic instruction that transfers control to another point in the program is **JMP**.

  The basic syntax of **JMP** instruction:

  ```
  JMP label
  ```

  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, for example here are 3 legal label definitions:

  ```
  label1:
  label2:
  a:
  ```

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

  ```
  x1:
  MOV AX, 1

  x2: MOV AX, 2
  ```

  here's an example of **JMP** instruction:

```
org   100h

mov   ax, 5        ; set ax to 5.
mov   bx, 2        ; set bx to 2.

jmp   calc         ; go to 'calc'.

back: jmp stop     ; go to 'stop'.

calc:
add   ax, bx       ; add bx to ax.
jmp   back         ; go 'back'.

stop:
```

```
ret                 ; return to operating system.
```

Of course there is an easier way to calculate the some of two numbers, but it's still a good example of **JMP** instruction.
As you can see from this example **JMP** is able to transfer control both forward and backward. It can jump anywhere in current code segment (65,535 bytes).

- **Short Conditional Jumps**

Unlike **JMP** instruction that does an unconditional jump, there are instructions that do a conditional jumps (jump only when some conditions are in act). These instructions are divided in three groups, first group just test single flag, second compares numbers as signed, and third compares numbers as unsigned.

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

as you may already notice there are some instructions that do that same thing, that's correct, they even are assembled into the same machine code, so it's good to remember that when you compile **JE** instruction - you will get it disassembled as: **JZ**, **JC** is assembled the same as **JB** etc... different names are used to make programs easier to understand, to code and most importantly to remember. very offset dissembler has no clue what the original instruction was look like that's why it uses the most common name.

if you emulate this code you will see that all instructions are assembled into **JNB**, the operational

code (opcode) for this instruction is **73h** this instruction has fixed length of two bytes, the second byte is number of bytes to add to the **IP** register if the condition is true. because the instruction has only 1 byte to keep the offset it is limited to pass control to -128 bytes back or 127 bytes forward, this value is always signed.

```
   jnc a
   jnb a
   jae a

   mov ax, 4
a: mov ax, 5
   ret
```

Jump instructions for signed numbers

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.

Jump instructions for unsigned numbers

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

Generally, when it is required to compare numeric values **CMP** instruction is used (it does the same as **SUB** (subtract) instruction, but does not keep the result, just affects the flags).

The logic is very simple, for example:
it's required to compare 5 and 2,
5 - 2 = 3
the result is not zero (Zero Flag is set to 0).

Another example:
it's required to compare 7 and 7,
7 - 7 = 0
the result is zero! (Zero Flag is set to 1 and **JZ** or **JE** will do the jump).

here's an example of **CMP** instruction and conditional jump:

```
include "emu8086.inc"

org     100h

mov     al, 25      ; set al to 25.
mov     bl, 10      ; set bl to 10.

cmp     al, bl      ; compare al - bl.

je      equal       ; jump if al = bl (zf = 1).

putc    'n'         ; if it gets here, then al <> bl,
jmp     stop        ; so print 'n', and jump to stop.

equal:              ; if gets here,
putc    'y'         ; then al = bl, so print 'y'.

stop:

ret                 ; gets here no matter what.
```

try the above example with different numbers for **AL** and **BL**, open flags by clicking on flags button, use single step and see what happens. you can use **F5** hotkey to recompile and reload the program into the emulator.

---

loops

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:

```
org 100h

mov bx, 0  ; total step counter.

mov cx, 5
k1: add bx, 1
    mov al, '1'
    mov ah, 0eh
    int 10h
    push cx
    mov cx, 5
      k2: add bx, 1
      mov al, '2'
      mov ah, 0eh
      int 10h
      push cx
         mov cx, 5
         k3: add bx, 1
         mov al, '3'
         mov ah, 0eh
         int 10h
         loop k3    ; internal in internal loop.
      pop  cx
```

```
      loop  k2      ; internal loop.
    pop cx
loop k1             ; external loop.

ret
```

bx counts total number of steps, by default emulator shows values in hexadecimal, you can double click the register to see the value in all available bases.

just like all other conditional jumps loops have an opposite companion that can help to create workarounds, when the address of desired location is too far assemble automatically assembles reverse and long jump instruction, making total of 5 bytes instead of just 2, it can be seen in disassembler as well.

for more detailed description and examples refer to **complete 8086 instruction set**

All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward (note that most instructions are assembled into 3 or more bytes).

We can easily avoid this limitation using a cute trick:

- Get an opposite conditional jump instruction from the table above, make it jump to *label_x*.
- Use **JMP** instruction to jump to desired location.
- Define *label_x:* just after the **JMP** instruction.

label_x: - can be any valid label name, but there must not be two or more labels with the same name.

here's an example:

```
include "emu8086.inc"

org    100h

mov    al, 5
mov    bl, 5

cmp    al, bl     ; compare al - bl.


jne    not_equal  ; jump if al <> bl (zf = 0).
jmp    equal
not_equal:

add    bl, al
sub    al, 10
xor    al, bl

jmp skip_data
db 256 dup(0)     ; 256 bytes
skip_data:
```

```
putc    'n'         ; if it gets here, then al <> bl,
jmp     stop        ; so print 'n', and jump to stop.

equal:              ; if gets here,
putc    'y'         ; then al = bl, so print 'y'.

stop:

ret
```

Note: the latest version of the integrated 8086 assembler automatically creates a workaround by replacing the conditional jump with the opposite, and adding big unconditional jump. To check if you have the latest version of emu8086 click **help-> check for an update** from the menu.

Another, yet rarely used method is providing an immediate value instead of label. When immediate value starts with $ relative jump is performed, otherwise compiler calculates instruction that jumps directly to given offset. For example:

```
org    100h

; unconditional jump forward:
; skip over next 3 bytes + itself
; the machine code of short jmp instruction takes 2 bytes.
jmp $3+2
a db 3    ; 1 byte.
b db 4    ; 1 byte.
c db 4    ; 1 byte.

; conditional jump back 5 bytes:
mov bl,9
dec bl      ; 2 bytes.
cmp bl, 0   ; 3 bytes.
jne $-5     ; jump 5 bytes back

ret
```

<<< previous part <<< **>>> Next Part >>>**
