

COMPARING AND BRANCHING

Computer programs are of no particular worth unless decisions can be made in them. In IBM PC assembly Language (and most modern computers) decision-making is a two step process:

1. Two numbers are compared using **cmp**, the **compare** instruction, which sets several bits in a **16-bit register** of the CPU called the **flags register**, and
2. A **conditional jump** instruction is executed which does or does not go to a new location based on the values of those flags.

The **cmp** instruction has two operands just like the **mov** instruction:

cmp reg/mem, reg/mem/constant

with the usual limitation of at most **one memory operand per instruction**. The **operands** can either be bytes, words or double words but as usual **must be of the same size**.

The instruction

cmp op1, op2

performs the subtraction **op1 - op2**, sets the flags according to the result, and discards the result. The flags set by the **cmp** instruction are as follows:

Bit number:					O					S	Z					C
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

O – Overflow Flag (OF), S – Sign Flag (SF), Z – Zero Flag (ZF), C – Carry Flag (CF)

(The **S** flag is set to the sign of the result, the **Z** flag is set to 1 if the result is zero, and the **O** and **C** flags are set to the Overflow and Carry status of the result respectively. The shaded portion of the flags register represents other flags and unused bits.)

For each of these flags there are **two conditional jumps**. For instance, **jc** jumps if the Carry flag is set i.e. is equal to 1, and **jnc** jumps on no carry flag, i.e. **CF = 0**. The **>**, **<=**, etc, conditions require complicated combinations of these jumps. So a number of other conditional jumps are provided:

For SIGNED Numbers, after `cmp op1, op2`

je	address	:jump if equal	(op1 = op2)
jne	address	:jump if not equal	(op1 ≠ op2)
jg	address	:jump if greater	(op1 > op2)
jge	address	:jump if greater or equal	(op1 ≥ op2)
jl	address	:jump if less	(op1 < op2)
jle	address	:jump if less or equal	(op1 ≤ op2)

You can think of the relation in the conditional jump instruction as sitting between the operands.

cmp op1, "<" op2
jl address

If the condition of a conditional jump is true, then the jump is taken else we fall through the instructions following the conditional jump.

Therefore, **Jcondition address** is equivalent to executing if **condition is true** then **IP = address**

In addition, we have an **unconditional jump**, one that is taken in all circumstances:

jmp address ;jump unconditionally

Note: Labels of pseudo-instructions must not have a colon, all other must be followed by a colon

Example 1:

```

                cmp    ax, bx
                jl     axless
                mov    X, 1
                jmp    Both
axless:        mov    X, -1
Both:         nop

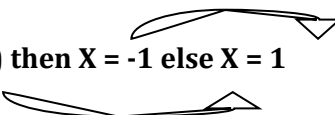
```

The equivalent pseudo-code is: **If (ax < bx) then X = -1 else X = 1**

Decoration

- ❖ If a conditional jump turns out to be **true**, it is drawn **above** the line
- ❖ If a conditional jump turns out to be **false**, it is drawn **below** the line

Hence



If (ax < bx) then X = -1 else X = 1

After you have drawn the lines, you can translate the various parts of the equation into assembly language, **leaving a blank corresponding to each start of an arrow** and **inserting a label for each arrow head**.

Step 1

```

                cmp    ax, bx          ; if (ax < bx) then ...
                mov    X, -1          ; X = -1
label1:        mov    X, 1          ; X = 1
label2:        nop

```

Step 2

Now we can fill in the blank lines by inserting appropriate jumps. We will enter **false jumps** i.e. for every instruction **JXX to jump on condition XX being true**, there is a corresponding **JNXX instruction, for (Jump on Not XX which is taken when the condition is false**. Therefore, our code becomes:

```

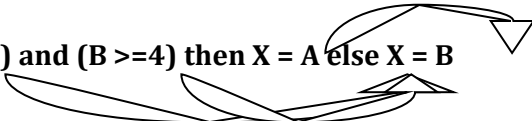
                cmp    ax, bx          ; if (ax < bx) then ...
                jnl    Label1          ; (jump on Not Less)
                mov    X, -1          ; X = -1
                jmp    Label2
label1:        mov    X, 1          ; X = 1
label2:        nop

```

Note: The **jmp Label2** is mandatory. Otherwise, **X** will always get set to 1, no matter what !!!

Example 2:

If (A < B) and (B >=4) then X = A else X = B


Step 1:

```

mov    ax, A
cmp    ax, B        ;    if (A < B) and ...

cmp    B, 14        ;    (B >=14) then ...

mov    ax, A
mov    X, ax        ;    X = A

label1:  mov    ax, B
        mov    X, ax        ;    X = B

```

Step 2:

```

mov    ax, A
cmp    ax, B        ;    if (A < B) and ...
jnl    Label1
cmp    B, 14        ;    (B >=14) then ...
jnge   Label1        ;    Note: jnge is equivalent to jl

mov    ax, A
mov    X, ax        ;    X = A
jmp    Label2

label1:  mov    ax, B
        mov    X, ax        ;    X = B

label2:

```

Simplify by Factoring

We notice that both halves of the IF branch ends with the same instruction: `mov X, ax`

We can replace these two instructions with a single occurrence outside the end of the two branches by deleting the first occurrence of `mov X, ax` and moving the `label2: label` to before the second occurrence.


```

mov    ax, A
cmp    ax, B        ;    if (A < B) and ...
jnl    Label1
cmp    B, 14        ;    (B >=14) then ...
jnge   Label1        ;    Note: jnge is equivalent to jl

mov    ax, A        ;    X = A
mov    X, ax
jmp    Label2

label1:  mov    ax, B
        mov    X, ax        ;    X = B
label2:

```



We see that the second `mov ax, A` instruction is superfluous since A is already in ax, so our code becomes:

```

mov    ax, A
cmp    ax, B        ;    if (A < B) and ...

```

```

        jnl     Label1
        cmp     B, 14      ; (B >=14) then ...
        jnge    Label1    ; Note: jnge is equivalent to jl
        mov     ax, A      ; X = A
        jmp     Label2
label1:  mov     ax, B
label2:  mov     X, ax      ; X = B

```

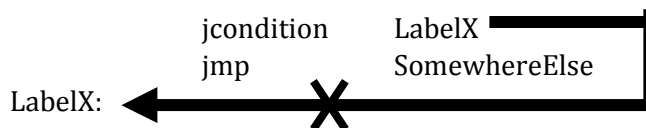
which is equivalent to

```

        mov     ax, A
        cmp     ax, B      ; if (A < B) and ...
        jnl     Label1
        cmp     B, 14      ; (B >=14) then ...
        jnge    Label1    ; Note: jnge is equivalent to jl
        jmp     Label2
label1:  mov     ax, B
label2:  mov     X, ax      ; X = B

```

After the deletion of the previous mov instruction, we now have a situation of a **jump around a jump** – a **conditional jump** followed immediately by an **unconditional jump** followed immediately by the **destination of the conditional jump**. That is:



This situation can always be replaced by the simple code:

```

        jNOTcondition      SomewhereElse

```

Final optimized code becomes:

```

        mov     ax, A
        cmp     ax, B      ; if (A < B) and ...
        jnl     Label1
        cmp     B, 14      ; (B >=14) then ...
        jge     Label2    ; X = A
label1:  mov     ax, B      ; X = B
label2:  mov     X, ax

```

UNSIGNED CONDITIONAL JUMPS

It is sometimes necessary to use unsigned conditional jumps. Subtraction is the same whether the numbers are signed or not.

For UNSIGNED Numbers, after cmp op1, op2

ja	address	:jump if above (op1 = op2)
jae	address	:jump if above or equal (op1 ≥ op2)
jb	address	:jump if below (op1 < op2)
jbe	address	:jump if below or equal (op1 ≤ op2)