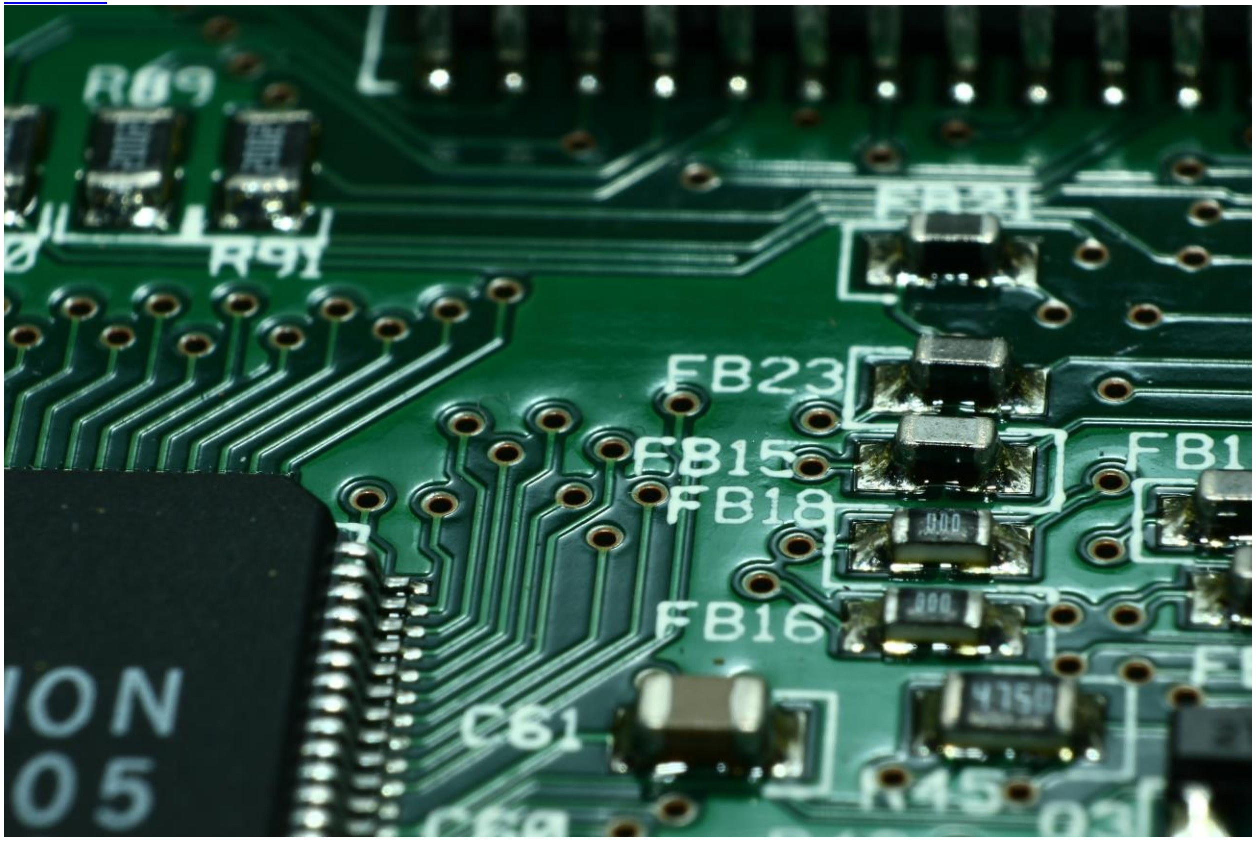
Learning Assembly Language - A Helpful Guide [Part 2]

Tutorial



From part 1, we have now got a firm grasp of the fundamental elements of assembly language (x86 specifically). The next step is to develop an understanding of the components that make up assembly language's program flow.

Assembly Program Flow

How do we control program flow in standard programming languages such as python? We use conditional statements (**if, else**), loops (**for, while**) as well as function calls.

These things simply don't exist in assembly, the core program flow is based around **jumping & branching**.

FLAGS

In order to understand how **jumping & branching** works, you must understand what **flags** are. **Flags** are a way of storing the result from an action. Most importantly, they help store the result when we compare two variables. All the flags are stored in a specific register called **EFLAGS** - each flag can only have two values, on or off.

The three key flags for you to know about are the following:

Sign

A flag that is set (turned on) when the result from the last operation was *negative.*

E.g. 4 - 20 would produce -16, a negative value, meaning the flag is set.

Zero

A flag that is set only when the result from the last operation was exactly *zero.*

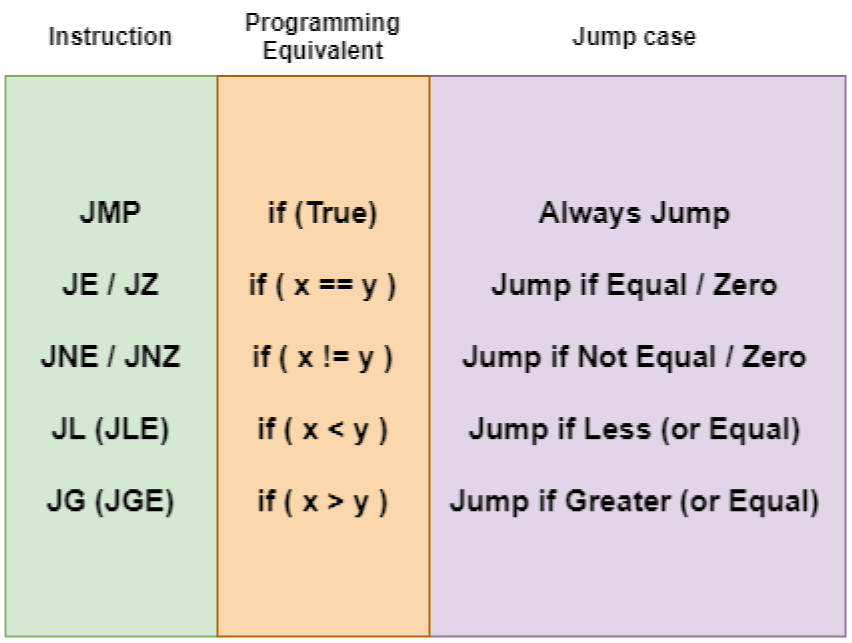
E.g. 7 - 7

Carry

This flag is set when we are adding/subtracting two numbers and require to "borrow" a bit as the two numbers were too large.

Jumping & Branching

Now we can use instructions to set flags, what is the purpose of this? Well, depending on these flags we can jump to different parts of the assembly. There are a number of "**jump"** instructions which will only trigger on certain flags, the list is as follows:



Instruction	Programming Equivalent	Jump case
JMP	if (True)	Always Jump
JE / JZ	if (x == y)	Jump if Equal / Zero
JNE / JNZ	if (x != y)	Jump if Not Equal / Zero
JL (JLE)	if (x < y)	Jump if Less (or Equal)
JG (JGE)	if (x > y)	Jump if Greater (or Equal)

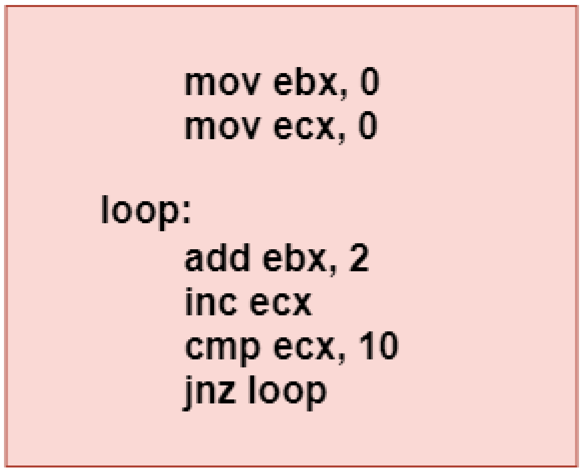
Before any jump instruction, there are typically two key instructions. These instructions are **CMP** and **TEST**.

CMP x, y - An instruction which does (x - y), and then set the flags. This way we can compare the parameters, then jump according to the **comparison**.

TEST x, y - A similar instruction, but it performs the logical AND (x & y) and then set's the flags. (If you don't understand bit-wise operations like AND, check out the explanation here -> https://en.wikipedia.org/wiki/Bitwise_operation)

Example

So now we have an understanding for all the individual components, let's do a simple example to piece it all together:



```
        mov ebx, 0
        mov ecx, 0

loop:
        add ebx, 2
        inc ecx
        cmp ecx, 10
        jnz loop
```

So let us step through this example to understand it.

The first two lines **mov ebx/ecx, 0** is setting the ebx and ecx registers to 0 - nice and simple so far!

Then we have a "loop" **label**. Labels are used in assembly to give us reference points, which we can then easily jump to.

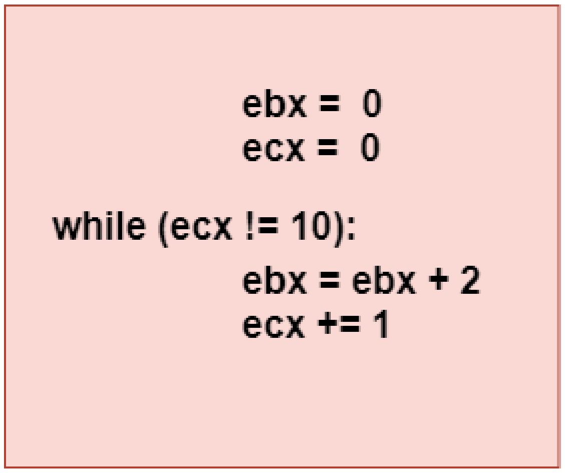
The next line then adds 2 into the ebx register. It was just set to 0, so now it becomes ebx = 0 + 2.

Next, we have an inc ecx, remember this is the same as ecx += 1. So now ecx = 1 as it was 0 before.

Now we compare ecx to the value 10. This is the equivalent of doing ecx - 10, then setting the flags.

Finally, we have a **JNZ** instruction. This will be triggered if the last instruction did **NOT** set the **zero flag**. As ecx - 10 was the last comparison, and ecx only equalled 1, then the result was not zero, so the flag was **NOT** set. This means we need to jump back to the "loop" **label** and carry out the instructions again. This will repeat until ecx == 10.

The python equivalent of this would be:



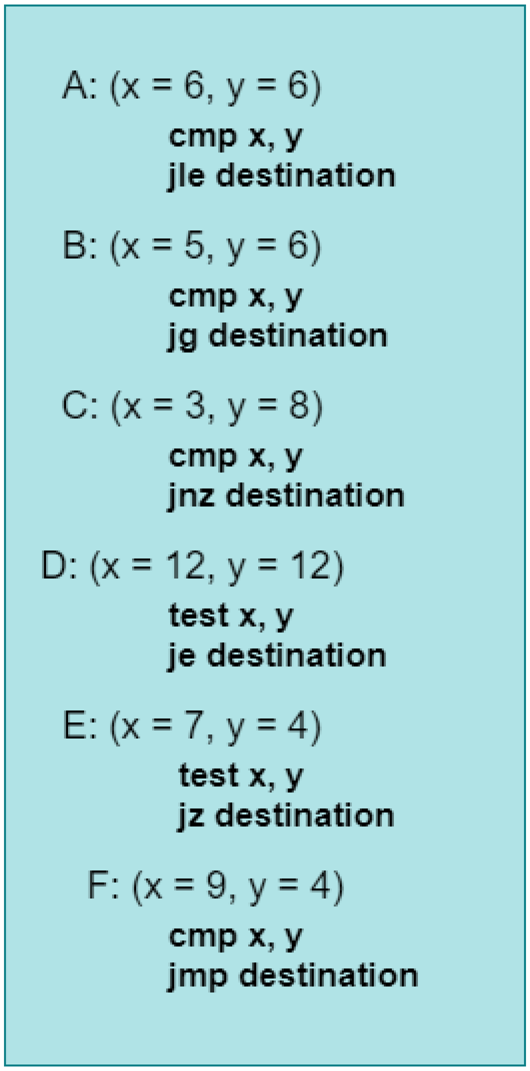
```
            ebx =  0
            ecx =  0

while (ecx != 10):
            ebx = ebx + 2
            ecx += 1
```

I hope that made sense so far. Now you can try out some checks to see if you have understood how jumps and flags work.

Mini Quiz

I will reveal the answer at the end of the post, answer TRUE if it will jump to the destination, and FALSE otherwise.



```
A: (x = 6, y = 6)
        cmp x, y
        jle destination
B: (x = 5, y = 6)
        cmp x, y
        jg destination
C: (x = 3, y = 8)
        cmp x, y
        jnz destination
D: (x = 12, y = 12)
        test x, y
        je destination
E: (x = 7, y = 4)
        test x, y
        jz destination
F: (x = 9, y = 4)
        cmp x, y
        jmp destination
```

Endianness

Endianness is something that is of huge benefit to understanding when dealing with assembly language.

Endianness is the way in which a sequence of bytes are stored in memory.

When we have a value such as 0x12345678 it can be represented in one of two ways - Big or Little Endian.

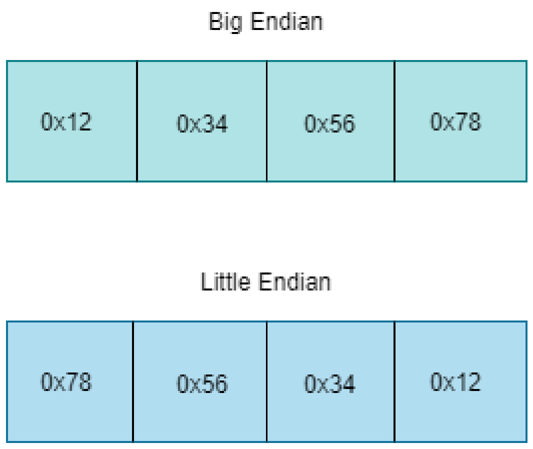
In all numbers, each digit has a "significance". For example with decimal numbers, like 1234, the number **1** here is the most significant, as it represents 1000s, then 2 represents 100s so is the next most significant.

This is the same with hex numbers, for 0x12345678 the byte 0x12 has the most significance, so it is classed as the most significant byte. Endianness is simple the ordering of the bytes by significance:

Big Endian (the human-readable format) - **most significant byte -> least significant**

Little Endian (the common assembly format) - **least significant byte -> most significant**

Take the number 0x12345678, we can visualise this with the following diagram:



Next Time - [Part 3]

Now we have covered jumping, flags and endianness, we will move onto working with functions. So far we have focused mainly on instructions and registers, and how these interact, but we will start to understand more about how memory and complete programs work within assembly.

This will introduce a much greater level of understanding and will give you a chance to write or reverse-engineer assembly independently.

Quiz Answers

A. True

B. False

C. True

D. True

E. False

F. True