3.3 Instructions

The ability to read and comprehend Assembly code is vital to reverse engineering. There are roughly 1,500 instructions, however, a majority of the instructions are not commonly used or they're just variations (such as MOV and MOVS). Just like in high-level programming, don't hesitate to look up something you don't know.

Before we get started there are three different terms you should know: **immediate**, **register**, and **memory**.

- An **immediate value** (or just immediate, sometimes IM) is something like the number 12. An immediate value is *not* a memory address or register, instead, it's some sort of constant data.
- A **register** is referring to something like RAX, RBX, R12, AL, etc.
- **Memory** or a **memory address** refers to a location in memory (a memory address) such as 0x7FFF842B.

> You may see a semicolon at the end of, or in-between, a few Assembly instructions. This is because the semicolon (😉 is used to write a comment in Assembly.

It's important to know the format of instructions which is as follows:
(Instruction/Opcode/Mnemonic) <Destination Operand>, <Source Operand>

> I will be referring to Instructions/Opcodes/Mnemonics as instructions, just note that some people call it different things.

Example:

```
mov RAX, 5
```

MOV is the instruction, RAX is the destination operand, and 5 is the source operand. Capitalization of instructions or operands does not matter. You will see me use a mixture of all letters capitalized and all letters lowercase. In the example given, 5 is an immediate value because it's not a valid memory address and it's certainly not a register.

Common Instructions

Data:

MOV is used to move/store the source operand into the destination. The source doesn't have to be an immediate value like it is in the following example. In the following example, the immediate value of 5 is being moved into RAX.
This is equivalent to RAX = 5.

```
mov RAX, 5
```

LEA is short for Load Effective Address. This is essentially the same as MOV except for addresses. It's also commonly used to compute addresses. In the following example, RAX will contain the memory address/location of num1.

```
lea RAX, num1
```

PUSH is used to push data onto the stack. Pushing refers to putting something on the top of the stack. In the following example, RAX is pushed onto the stack. Pushing will act as a copy so RAX will still contain the value it had before it was pushed.

```
push RAX
```

POP is used to take whatever is on the top of the stack and store it in the destination. In the following example whatever is on the top of the stack will be put into RAX.

```
pop RAX
```

Arithmetic:

INC will increment data by one. In the following example RAX is set to 8, then incremented. RAX will be 9 by the end.

```
mov RAX, 8
inc RAX
```

DEC decrements a value. In the following example, RAX ends with a value of 7.

```
mov RAX, 8
dec RAX
```

ADD adds a source to a destination and stores the result in the destination. In the following example, 2 is moved into RAX, 3 into RBX, then they are added together. The result (5) is then stored in RAX.
Same as RAX = RAX + RBX or RAX += RBX.

```
mov RAX, 2
mov RBX, 3
add RAX, RBX
```

SUB subtracts a source from a destination and stores the result in the destination. In the following example, RAX will end with a value of 2.
Same as RAX = RAX - RBX or RAX -= RBX.

```
mov RAX, 5
mov RBX, 3
sub RAX, RBX
```

MUL (unsigned) or **IMUL** (signed) multiplies the destination by the source. The result is stored in the destination. IMUL is used for signed and MUL is used for unsigned. In the following example, RAX will end with a value of 15.

```
mov RAX, 5
mov RBX, 3
mul RAX, RBX
```

DIV (unsigned) or **IDIV** (signed). - I'll put my own explanation here eventually (maybe). For now just follow these links:

- **DIV**: https://www.felixcloutier.com/x86/div
- **IDIV**: https://www.felixcloutier.com/x86/idiv

Flow:

CMP compares two operands and sets the appropriate flags depending on the result. The following would set the Zero Flag (ZF) to 1 which means the comparison determined that RAX was equal to five. Flags are talked about in the next lesson, 3.4 Flags.

```
mov RAX, 5
cmp RAX, 5
```

JCC instructions are conditional jumps that jump based on the flags that are currently set. JCC is not an instruction, but a set of instructions that includes JNE, JLE, JNZ, and many more. JNE is jump if not equal, and JLE is jump if less than or equal. This is often used in if statements. The following example will quit immediately if RAX isn't equal to 5. If it is equal to 5 then it will set RBX to 10, then quit.

```
mov RAX, 5
cmp RAX, 5
jne 5       ; Jump to line 5 (ret) if not equal.
mov RBX, 10
ret
```

RET is short for return. This will return execution to the previous function. The following example sets RAX to 10 then returns.

```
mov RAX, 10
ret
```

NOP is short for No Operation. This instruction effectively does nothing. It's typically used for padding. Padding is done because some parts of code like to be on specific boundaries such as 16 bit boundaries, or 32 bit boundaries.

Back To The Example In 3.1

Remember the example from 3.1? Here it is:

```
if(x == 4){
    func1();
}else{
    return;
}
```

is the same as

```
mov RAX, x
cmp RAX, 4
jne 5       ; Line 5 (ret)
call func1
ret
```

Hopefully, you can now work out the assembly version on its own. It moves the variable `x` into RAX, then it compares `x` to 4. If they *are not equal* then it will return, if they *are equal* then it calls "func1".

Flipping Out

You may also notice that the comparison in the example above is flipped.
Instead of

```
if(x == 4){
    func1();
}else{
    return;
}
```

the Assembly version is:

```
if(x != 4){
    return;
}else{
    func1();
}
```

You could easily make the Assembly version match the original C version. The Assembly given in the example is what you will typically see in "the real world ". Although it may not seem like it, the reason for this is efficiency. Basically, the compiler can make the program look one of two main ways:

```
mov RAX, x
cmp RAX, 4
je 5       ; Line 5 (call func1)
ret
call func1
ret
```

```
mov RAX, x
cmp RAX, 4
jne 5       ; Line 5 (ret)
call func1
ret
```

This code wants to return once `func1` has returned, and/or if RAX is not equal to 4. In other words, it's returning either way. To reduce the number of instructions the compiler will choose the second option. You may have seen some programmers write if-statements that are more similar to `if(x != 4)` instead of `if(x == 4)`, and this is probably why. Although it really doesn't matter how the programmer writes it because the compiler will most likely choose the best option anyways.

Pointers

Assembly has it's ways of working with pointers and memory addresses like C/C++ does. In C/C++ you can use dereferencing to get the value inside of a memory address. For example:

```
int main(){
    int num = 10;
    int* ptr = &num;        //y is now the address of x
    /*
    ptr is dereferenced so it prints what's inside of the address
    that it's holding. The value inside of the address that it's
    holding is 10.
    */
    printf("%d", *ptr);
}
```

Two of the most important things to know when working with pointers and addresses in Assembly are **LEA** and **square brackets**.

- **Square Brackets** - Square brackets signify "address pointed to". For example, `[var]` is the address pointed to by var. In other words, when using `[var]` we want to access the memory address that `var` is holding.
- **LEA** - Ignore literally everything about square brackets when working with LEA. LEA is short for Load Effective Address and it's used for calculating and loading addresses.

It's important to note that when working with the LEA instruction, square brackets do *not* dereference.
It's important to note that when working with the LEA instruction, square brackets do *not* dereference.
It's important to note that when working with the LEA instruction, square brackets do *not* dereference.
Have I said it enough? Don't let this confuse you. People who don't know that LEA breaks the rules when working with square brackets get extremely confused very quickly. LEA is used to load and calculate addresses, NOT data. It doesn't matter if there are square brackets or not, it's dealing with addresses ONLY. LEA tends to mess with many peoples heads. Don't let her do that to you.

Here is a simple example of dereferencing and a pointer in Assembly:

```
lea RAX, [var]
mov [RAX], 12
```

In the example above the *address* of `var` is loaded into RAX. This is LEA we are working with, there is *no* dereferencing. RAX is essentially a pointer now. Then 12 is moved into the address that RAX holds (often said as the address pointed to by RAX). The address pointed to by RAX is the `var` variable. If that Assembly was executed, `var` would be 12.

Earlier I said that LEA can be used to calculate addresses, and it often is.

```
lea RAX, [RCX+8]    ;This will set RAX to the address of RCX+8.
```

```
mov RAX, [RCX+8]    ;This will set RAX to the value inside of RCX+8.
```

One more time:
It's important to note that when working with LEA square brackets do *not* dereference.
You'll see LEA and MOV used all the time so be sure you understand this. I know it can be a tad confusing but just remember that LEA is for addresses.

Final Note

There are many more Assembly instructions that I haven't covered. As we continue I will introduce more instructions as they come. Don't be afraid to look up instructions, because like I said, there are quite a few (hundreds or thousands).

<- Previous Lesson
Next Lesson ->

Chapter Home