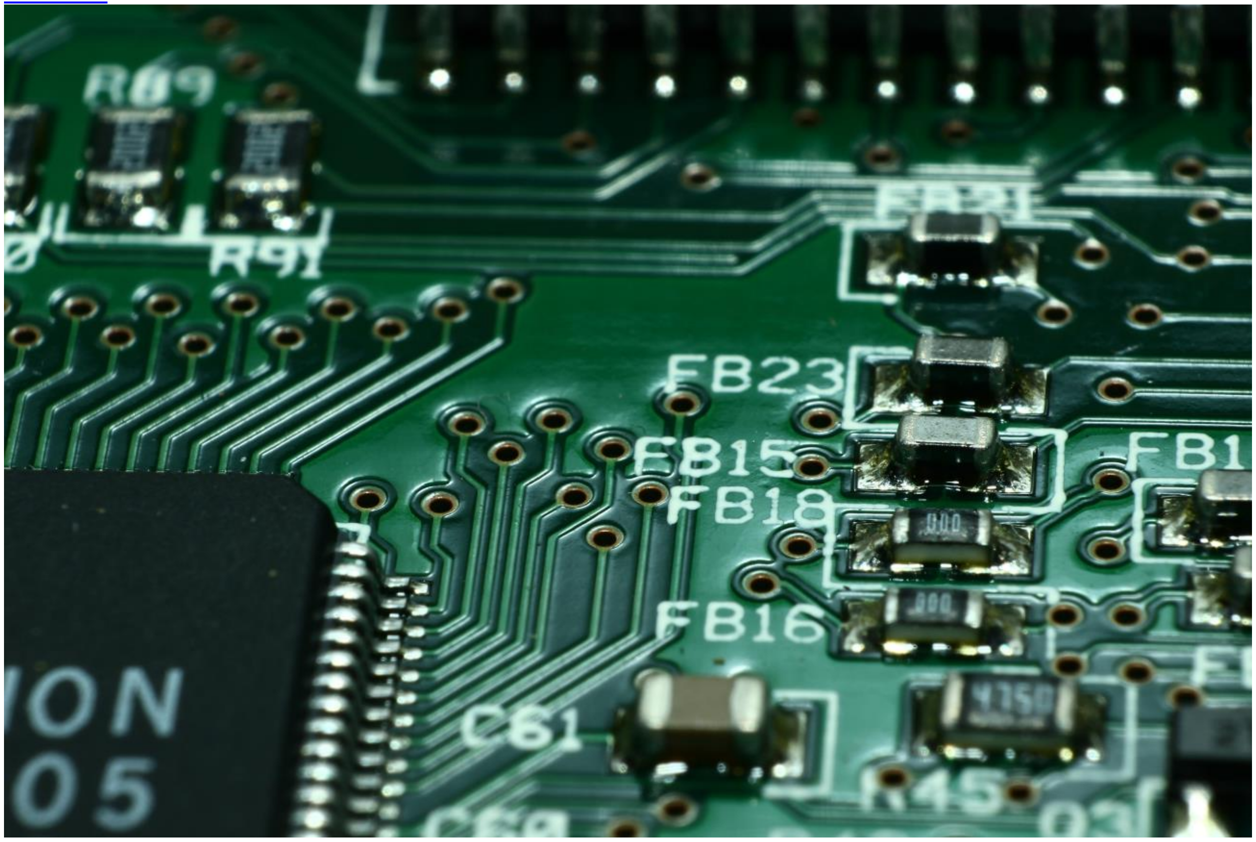
Learning Assembly Language - A Helpful Guide [Part 1]

Tutorial



What is Assembly?

Assembly language is a programming language just like any other. Well nearly... every line in assembly language actually represents a single CPU instruction. This is different from a language like **python** where a single line of code could represent hundreds of CPU instructions.

As a result of this, we have a lot more control over what we can do, but, it is a lot more complicated and time-consuming to create and debug programs.

Why Learn Assembly?

So if assembly is so much more complicated and time-consuming, you are probably thinking "what is the point?".

Well, by being able to understand assembly, you get an understanding for how your CPU behaves, allowing you to understand all programs you write/inspect to a much deeper level than those who don't have this skill. It will teach you more about memory management and the inner workings of what you are asking of the CPU when you make certain function calls, allowing you to improve you coding across all forms of programming exponentially.

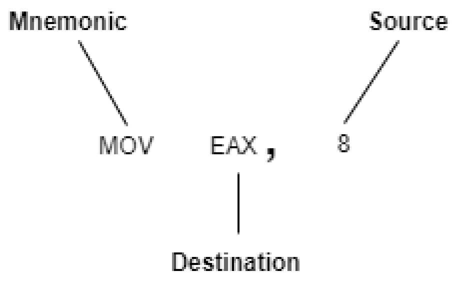
Moreover, it allows the ability to reverse engineer executables. If you compile a program, it goes from a human-readable form, into a form that is some mutation of assembly language - this means you can work backwards from a compiled and packed program, to find any possible hidden, unknown vulnerabilities, secret extra features, or just get a much more knowledgeable understanding.

And of course, if you have a niche, specific requirement to interact with low-level hardware or system internals, there is no more reliable method than assembly. It may be harder, but you have complete understanding of what your instructions are doing, being able to interact in ways that are exclusive to that only of assembly language.

Getting Started

To start with we will focus on x86 - the Intel specific assembly format. This flavour of assembly is used in the majority of desktop and laptop computers, being support by intel and AMDs main CPUs.

For x86, the format of all assembly instructions is roughly as follows:



The main 3 parts are the Mnemonic, Source and Destination.

The Mnemonic is simply the type of instruction that is being executed on that line. In our example here, we are using the **MOV** mnemonic, this means we are moving some value from the source into the destination.

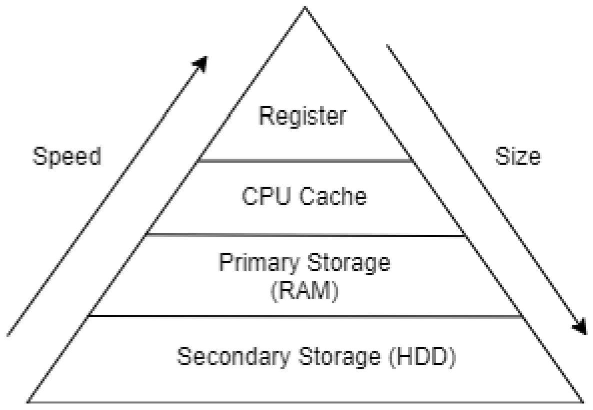
The Source is where our data is originating from, whether it is a constant, or some variable. Think of this as the data on the right of an equals sign, so here it would be EAX = **8**.

The Destination is where our result will be stored. As stated before, it can be thought of as the left side of an equation (left of the '='). In this case, it is **EAX**, this is a register, something we will come to later in the course, you can just think of it as a variable for now.

So all in all, the instruction above is moving the constant 8 into our "variable" EAX. That wasn't too bad, you just read your first bit of assembly!

What Are Registers?

As we mentioned before, and will be continuing to mention a lot, assembly uses something called registers. These are just fast local storage locations. They are easily accessible within the CPU, therefore it doesn't need to access any external memory, saving a lot of time. On the downside, the CPU only contains a few registers, and these registers can only hold a very small amount of data. In x86 the registers are only 32 bits in size. This means they can hold 4 bytes of data, or as little as 4 characters!



As you can see, we have a very clear data hierarchy that shows that in our computer we have a range of memory, from small and fast like registers, to large and slow like the HDD.

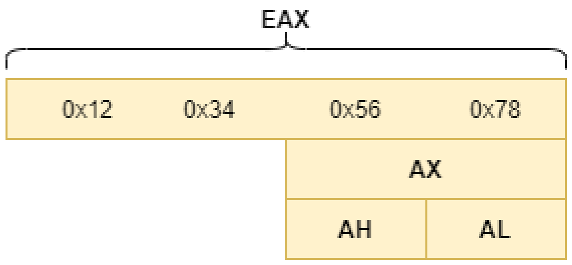
So how many registers does x86 have? In total there are 9 key registers:

EAX, EBX, ECX, EDX, EBP, ESP, EIP, ESI, EDI.

The reason for the naming convention is due to the fact that originally most registers had a specific purpose, for example, the **ECX** register was used exclusively for counting, so would typically be used as the counter when performing a loop. EAX, EBX, ECX and EDX are now thought of as general registers, they can be used for any general-purpose need. The other registers are still role-specific, however, we will come onto these later.

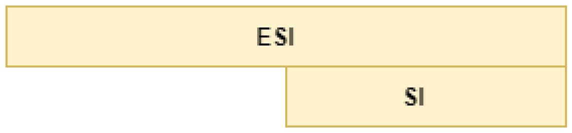
Register Breakdown

When dealing with registers, we don't have to reference it as a whole, the register can be broken down into smaller subsections for interacting with. This means that we can perform easier bit manipulations, and enables greater precision with potential for greater optimisation. When we reference a general register, we can pass a 4 byte value in, such as 0x12345678. To do this we can write the instruction mov EAX, 0x12345678. In this case, it is including all 4 bytes with the EAX reference. We can, however, break it down as shown:



As you can see from the diagram above, we can reference the whole value 0x12345678 with the name **EAX**. But if we want to just get the value of the third byte (0x56) we can use **AH** to retrieve it. **AX** would return 0x5678 and **AL** gives 0x78. All general registers (EAX, EBX, ECX, EDX) are broken down in the same way, with BX, CX, DX, etc...

ESI, EDI, EBP, ESP can all be broken down, but only in a similar way as shown:

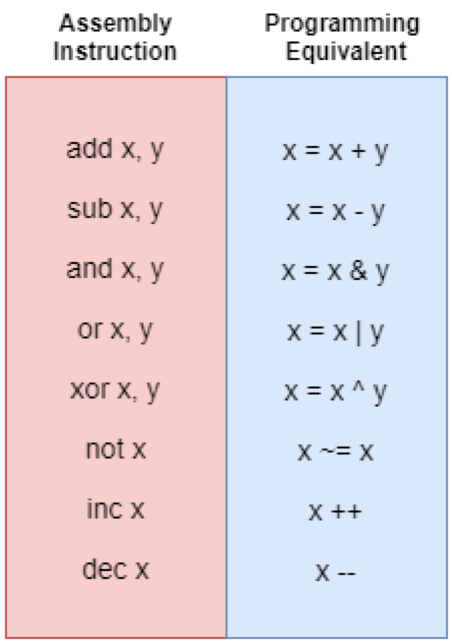


For example, EDI -> DI, EBP -> BP, ESP -> SP.

Getting started with Fundamental Instructions

Now we have an understanding of what registers are, and the way assembly instructions are laid out, we can begin to recognise some of the most frequent instructions.

Arithmetic instructions



Assembly Instruction	Programming Equivalent
add x, y	x = x + y
sub x, y	x = x - y
and x, y	x = x & y
or x, y	x = x | y
xor x, y	x = x ^ y
not x	x ~= x
inc x	x ++
dec x	x --

As you can see from these instructions, we can see how we can decipher to operation from the mnemonic. Then we store the result of the action on the source in the destination. There is also a new instruction format where there is only a destination - e.g. inc x. This is called a "unary operation" as it only uses one parameter. The reason it has no source is that the instruction has a fixed operation, it will always just add 1 to the register provided, there is, therefore, no need to pass in any other parameters.

Breaking down the cases above, we can replace X and Y in several different ways. There are 4 ways in particular that we can address the values X and Y.

Direct

This is when we pass an **immediate** value into a register / memory region.

E.g. **mov eax , 0x12345678**

Indirect

Moving some contents that is within memory into a register / another memory region.

E.g. **mov eax , [0x12345678]**

Direct Register

Similar to the "direct", this moves a value from one register to another.

E.g. **mov eax , ebx**

Indirect Register

Take the contents of memory, pointed to by one register, and move it into another register / memory region.

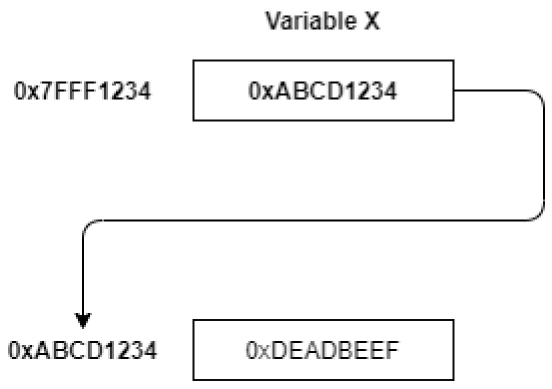
E.g. **mov eax , [ebx]**

Pointers Explained

When we put square brackets **- [ ] -** around an immediate value or register, what does this mean? This is the programming equivalent of defining it as a pointer. If you aren't familiar with pointers they are explained below:

When we write - **mov EAX, [EBX]** - it is the programming equivalent of - **EAX = *EBX**

This ***** is an operator that helps us get the data that the pointer is directing us towards. So how does this work? Say we have a variable 'X', if we assign Y = X, we take whatever value is inside the variable 'X', and put it straight into 'Y'. However, if we take variable 'X' as a pointer, we can instead say Y = *X. if this is the case, we treat the value stored in X as an address to get a value from. Take the diagram below:



As we can see, X = 0xABCD1234. If we use Y = *X as we talked about, it takes 0xABCD1234, and looks into memory to see what is stored at this address. As you can see from the diagram, it would be 0xDEADBEEF. Therefore the result of Y = *X would be Y = 0xDEADBEEF. This will become more apparent as we work with more hands-on examples in later guides.

[Part 2]

Now we understand pointers, assembly format, and some simple instructions, we can begin to read/write some actual programs.