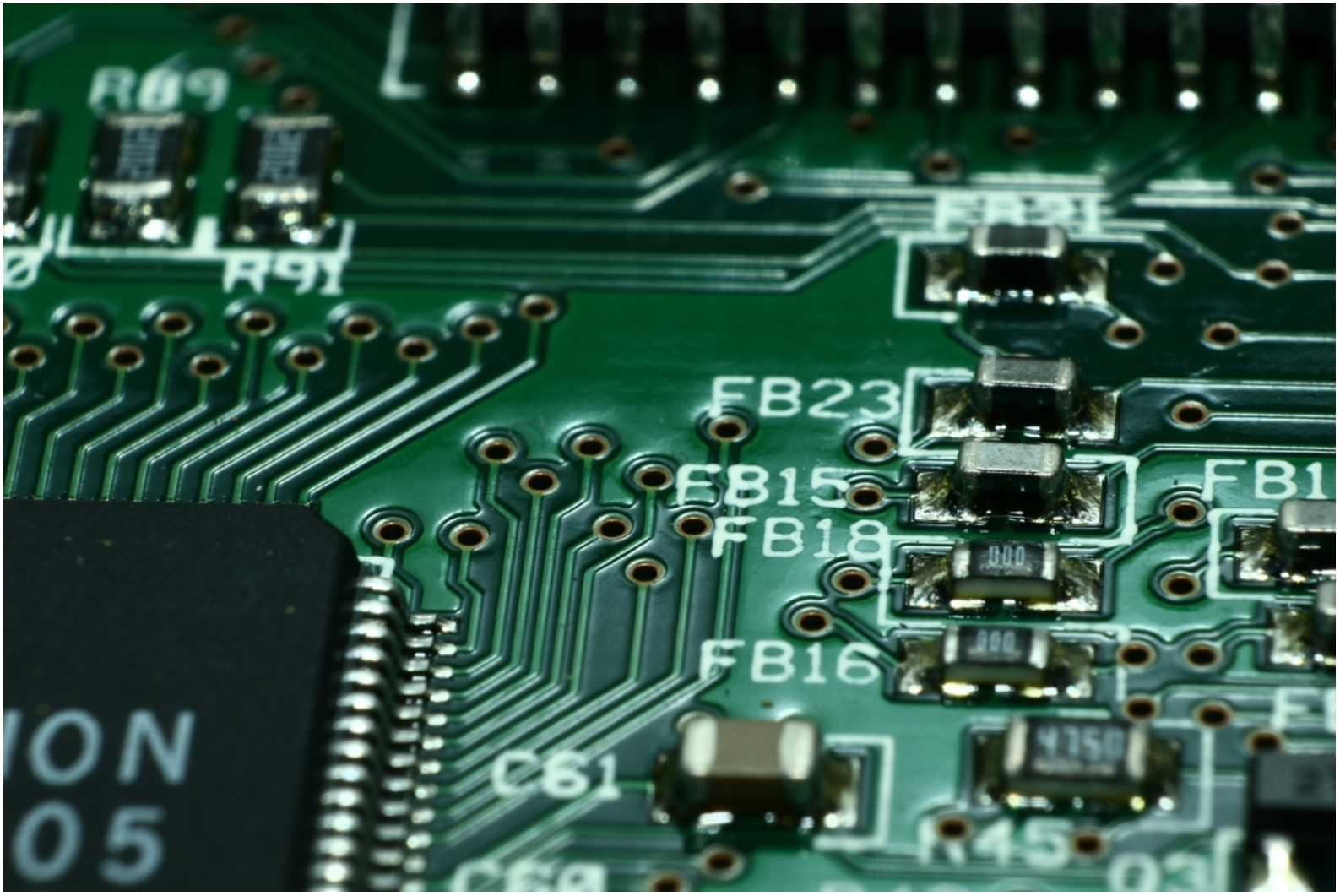
TUTORIAL

LEARNING ASSEMBLY LANGUAGE - A HELPFUL GUIDE [PART 3]

TUTORIAL



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. The

best place to start is by understanding how we make calls to functions, and one of the key elements of program execution - a STACK!

FUNCTION CALLS

Anyone who has programmed in higher-level programming languages such as Java and Python, understands the importance of using functions. These split up the code, organising it into reusable code that can perform a specific - required - purpose.

Covering **labels and jumps** in part 2, we know how we can easily move around our program, however it is useful to divide your program into functions instead, as they allow better understand of our code, and allow us to pass parameters for cleaner solutions.

The main two instructions we need to know, which allow us to create functions, are as follows:

call - Similar to jump command, but once finished, returns to the next specified instruction

ret - Tells the program to resume execution from after the last **call** instruction.

The **call** instruction remembers where we need to return to through one important piece of information, the **return address**. This piece of information points at the address which we will jump to when the **ret** instruction is called. But how do we store this value?

Unlike previously, where all values have been stored in a register, we need to store this value in a safe location where it is less likely to be overwritten, so we use **the stack**!

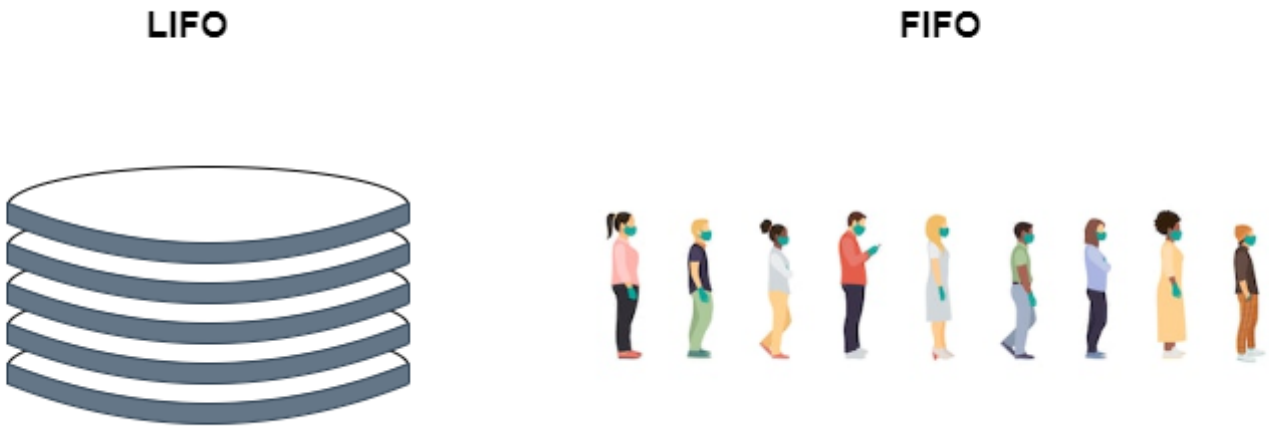
THE STACK

The stack is something you may have heard of before, but you really need to understand it well to excel at assembly programming!

So what is a stack?

A stack is a buffer, and more specifically a LIFO buffer (Last In First Out). This is different to the other standard buffer type - a queue - which is FIFO (First In First Out). The 'LIFO' type describes *how* you can insert and retrieve information, by saying it is LIFO, we are saying that whenever we want to get something from the buffer, the value we get will always be the last one we added!

I find a diagram often helps understand this:



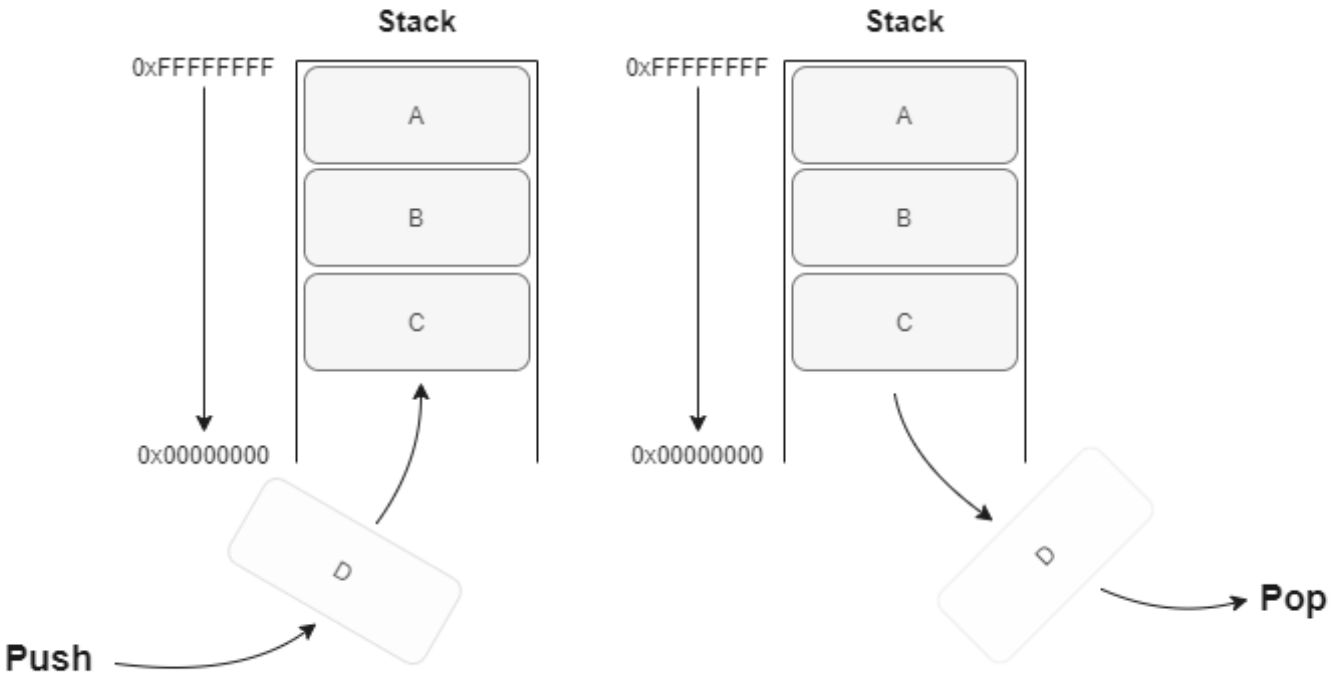
Queue Image - Shutterstock (1718731357)

As you can see, on the left we have a **stack** of plates. Every time we want to get a clean plate of the stack, we take the one from the top. However, if we add to the stack, we would place it on top of the stack. This means that the last plate we added to the pile, is the first one that is taken off the pile.

On the right we have a queue which is FIFO. If you join the queue and are the first person, then you are going to be served first too!

Stack Components

There are a number of crucial things to know when dealing with a stack. Using the image below for reference we can walk through them:



First thing first is the **layout**. When we are talking about a stack, it is really just a region in memory. This means that it has a start address, and an end address. Although we are unlikely to reach the maximum bounds for the stack, it is important to note one thing - the *direction the stack grows*. In our scenarios, the stack is **growing towards zero**. This means our stack will have a high start address, such as 0xFFFFFFFF as it is here, and every time we add something to the stack, it gets placed at a lower address. This may be confusing and hopefully the diagram helps - but essentially,

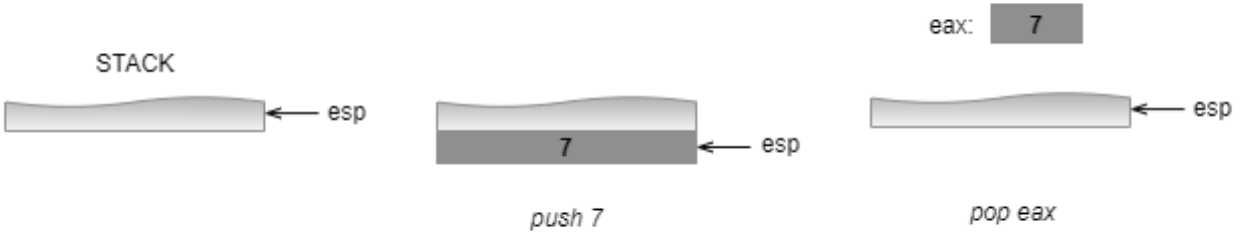
if we want to get element **A** from the diagram, we would access at the address 0xFFFFFFFF, whereas element **B** would be at 0xFFFFFFFB (even though it was placed on the stack *AFTER* element A). This may still be confusing, but as we work through more exercises it will become easier.

Next thing to learn about is the special purpose registers - ESP & EBP. As we spoke about before, the stack is a region of memory, but, how do we keep track of where the stack is (especially as each function has it's 'own' stack). This is where the clever introduction of EBP and ESP come in. EBP is known as the '**base pointer**', it will always point at the base (start) address of the stack. This allows us to know when our stack is empty and prevent us going out of bounds! The most useful part is that this now gives us an easy reference point, the ebp will never change within a function, so we can use it for reference at any point!

The next register is the ESP, the '**stack pointer**'. This pointer handily tells us where the top of the stack is - which means it points to the most recently added element in the stack. This is useful for getting the values from the stack and can also be used as a reference point (however it can change it's value throughout the function).

The reason the ESP is particularly useful, is the two key assembly instructions for interacting with the stack - **pop** and **push**.

These are simple instructions, the **push** instruction (as can be seen in the diagram) just adds a new value to the stack - which updates the **esp** so it now points at the new value. **Pop** does the opposite, it takes the current top value from the stack, and then places it into a register you specified. Once done it updates the **esp** accordingly too.



Above we can see a simple starter example, where the instructions are simply, push 7 -> pop eax. This gives an idea as to how values can be added to the stack and placed into registers. Below are some more example instructions that deal with the stack, see if you can work through them, the answers are at the end of the blog:

Q1.

What are the values of **eax**, **ebx** and **ecx** after this has finished.

```
push 12
push 34
mov ebx, 18
push ebx
pop ecx
pop ebx
pop eax
```

Q2.

What are the values of **eax, ebx, ecx** and **edx** after this has finished.

```
push 1
push 8
push 2
pop eax
push 56
mov edx, 7
pop ecx
pop edx
push 7
push 19
pop ebx
add ebx, eax
```

Q3.

What are the values of **eax, ebx, ecx** and **edx** after this has finished.

For this question, it is best to know that, in our example, the stack is **empty** and starts at **0x40**. This means **ebp** = **esp** = 0x40 to start with.

Also, when you add to the stack, you always add 4 bytes, so esp would change by **0x4**. This is standard for x86. This is tricky so don't worry if it is confusing!

```
push 20
push 82
push 33
pop  eax
mov  ebx, esp
push 6
pop  ecx
pop  edx
add  ebx, ecx
sub  edx, ebp
```

SUMMARY SO FAR

So, at this point I will take a break from the stack, as it can be quite a daunting thing to learn about, especially as a beginner. There are a couple more concepts to learn about before we have completely understood all the fundamentals of a stack, however, we will continue these in the next part!

For those wanting to do some more research before the next guide, you can look up **stack prologue** and **epilogue**. As well as looking at **calling conventions** (particularly **stdcall** vs **cdecl**).

Next week I will go over the previously mentioned, as well as a how-to on assembling a program and running it! Check out the answers to the quiz below:

ANSWERS

Q1:

eax = 12 (0xC)

ebx = 34 (0x22)

ecx = 18 (0x12)

Q2:

eax = 2 (0x2)

ebx = 21 (0x15)

ecx = 56 (0x38)

edx = 8 (0x8)

Q3:

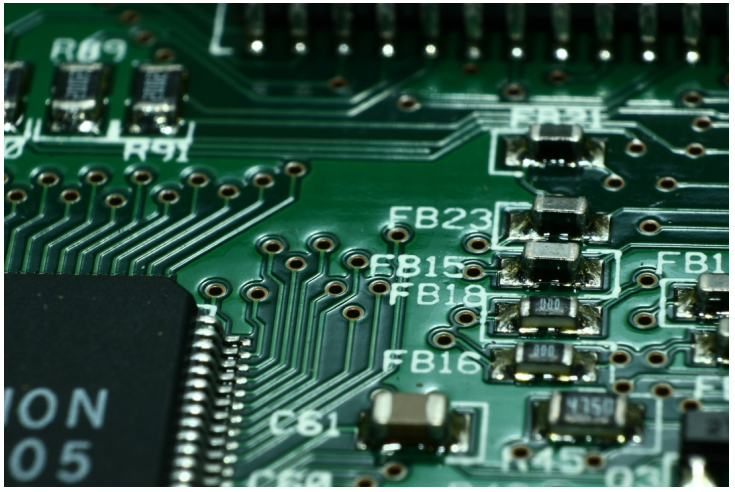
eax = 33 (0x21)

ebx = 62 (0x3E)

ecx = 6 (0x6)

edx = 18 (0x12)

LATEST POSTS



PENETRATION TESTING

PENTESTING FUNDAMENTALS & TECHNIQUES - BLIND SQL INJECTION

BY TUTORIAL

TUTORIAL

LEARNING ASSEMBLY LANGUAGE - A HELPFUL GUIDE [PART 3]

BY TUTORIAL

TECHTEACHING

SUBSCRIBE TO NEWSLETTER

Email address

SEND

CATEGORIES

CTF

Penetration Testing

Programming

Tutorial

INFORMATION

about

contact

terms

FOLLOW US

instagram

facebook

twitter

TEMPLATE

Image License Info

Powered by Webflow