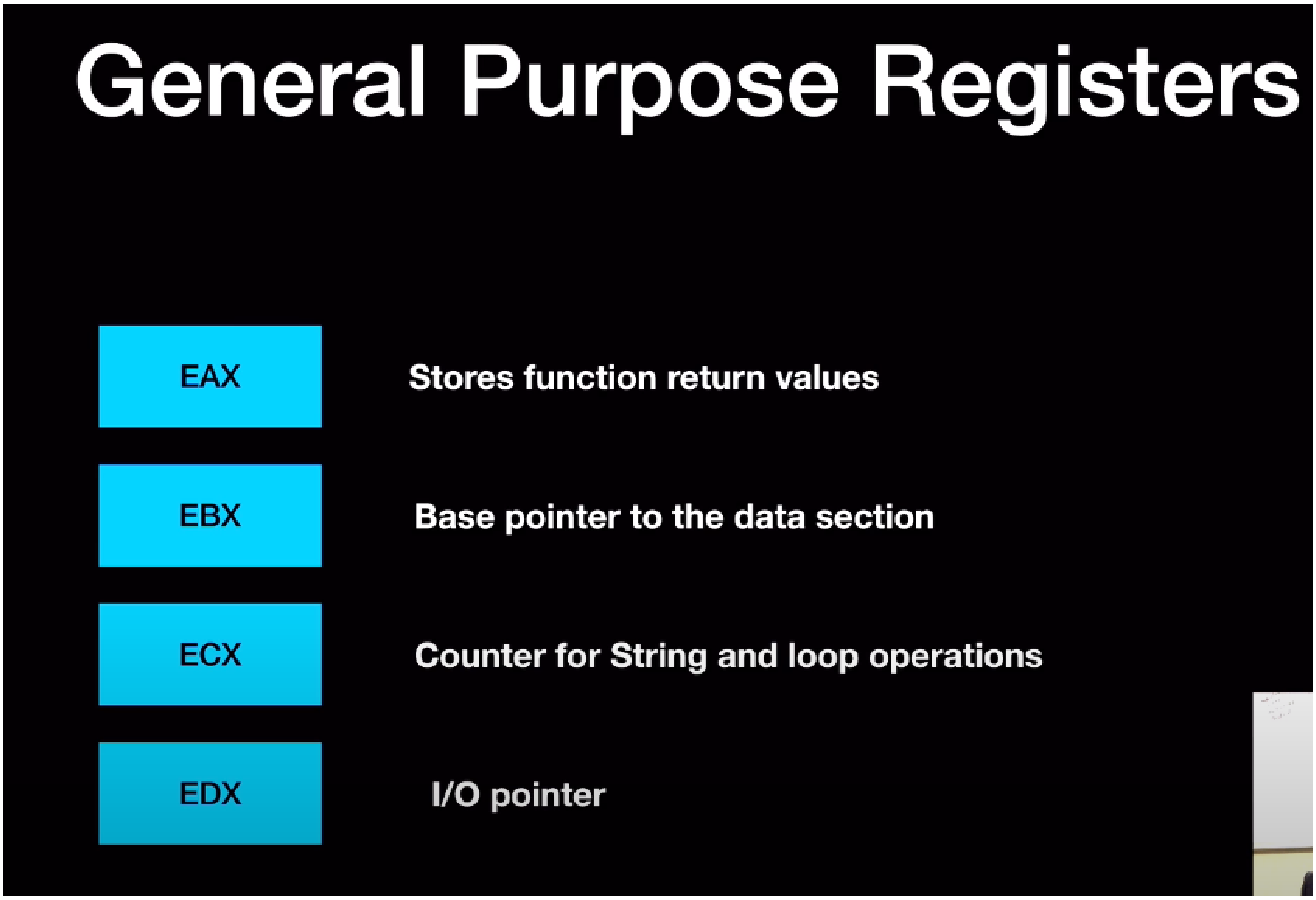
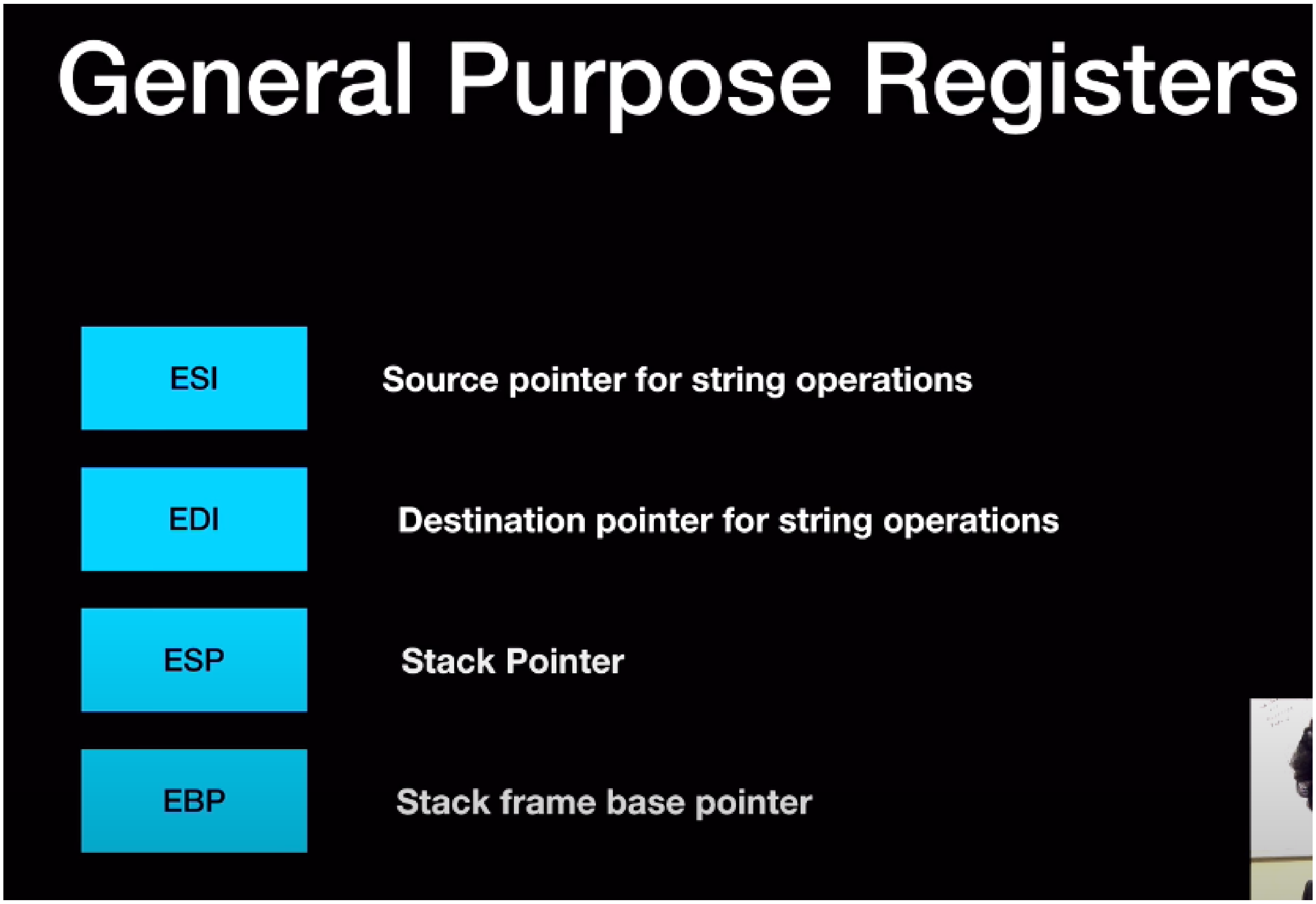
Assembly x86

General purpose registers:



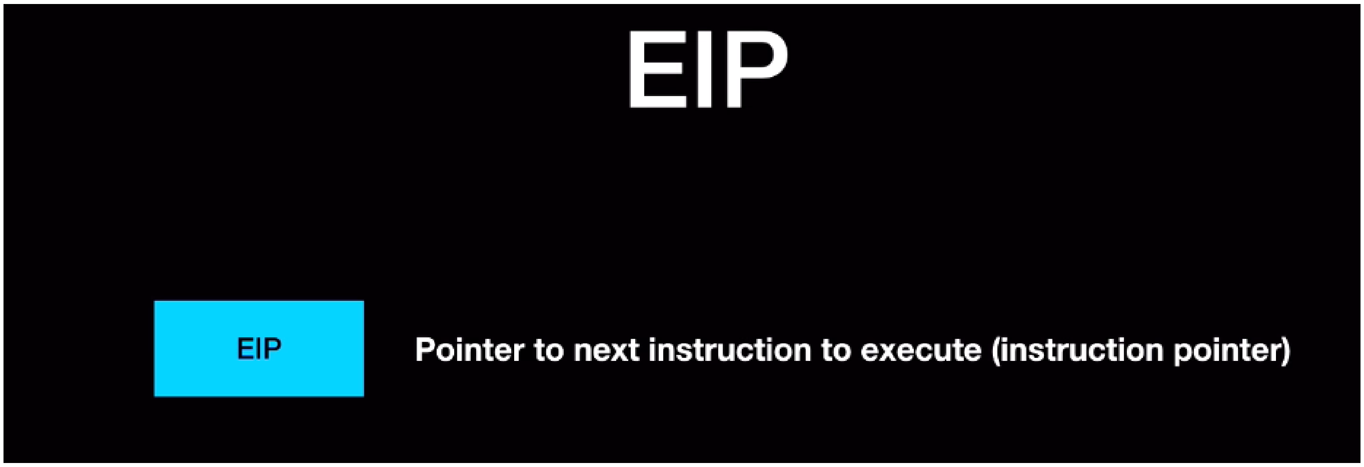
General Purpose Registers

EAX	Stores function return values
EBX	Base pointer to the data section
ECX	Counter for String and loop operations
EDX	I/O pointer



General Purpose Registers

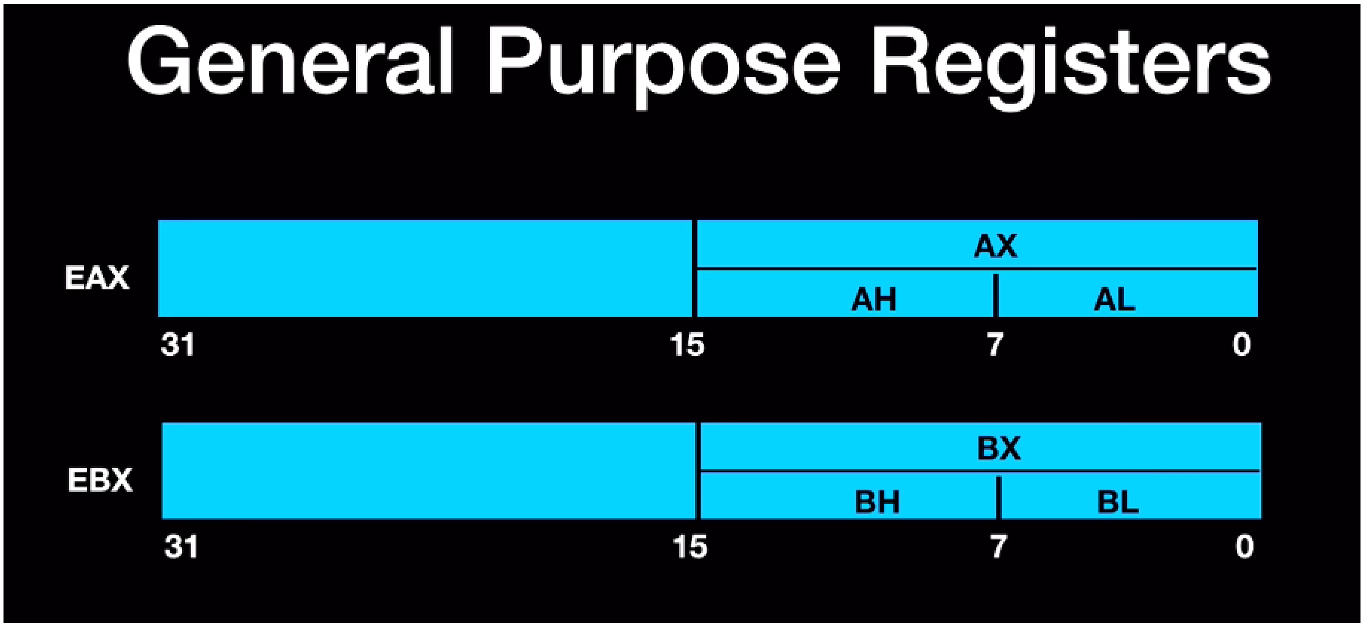
ESI	Source pointer for string operations
EDI	Destination pointer for string operations
ESP	Stack Pointer
EBP	Stack frame base pointer



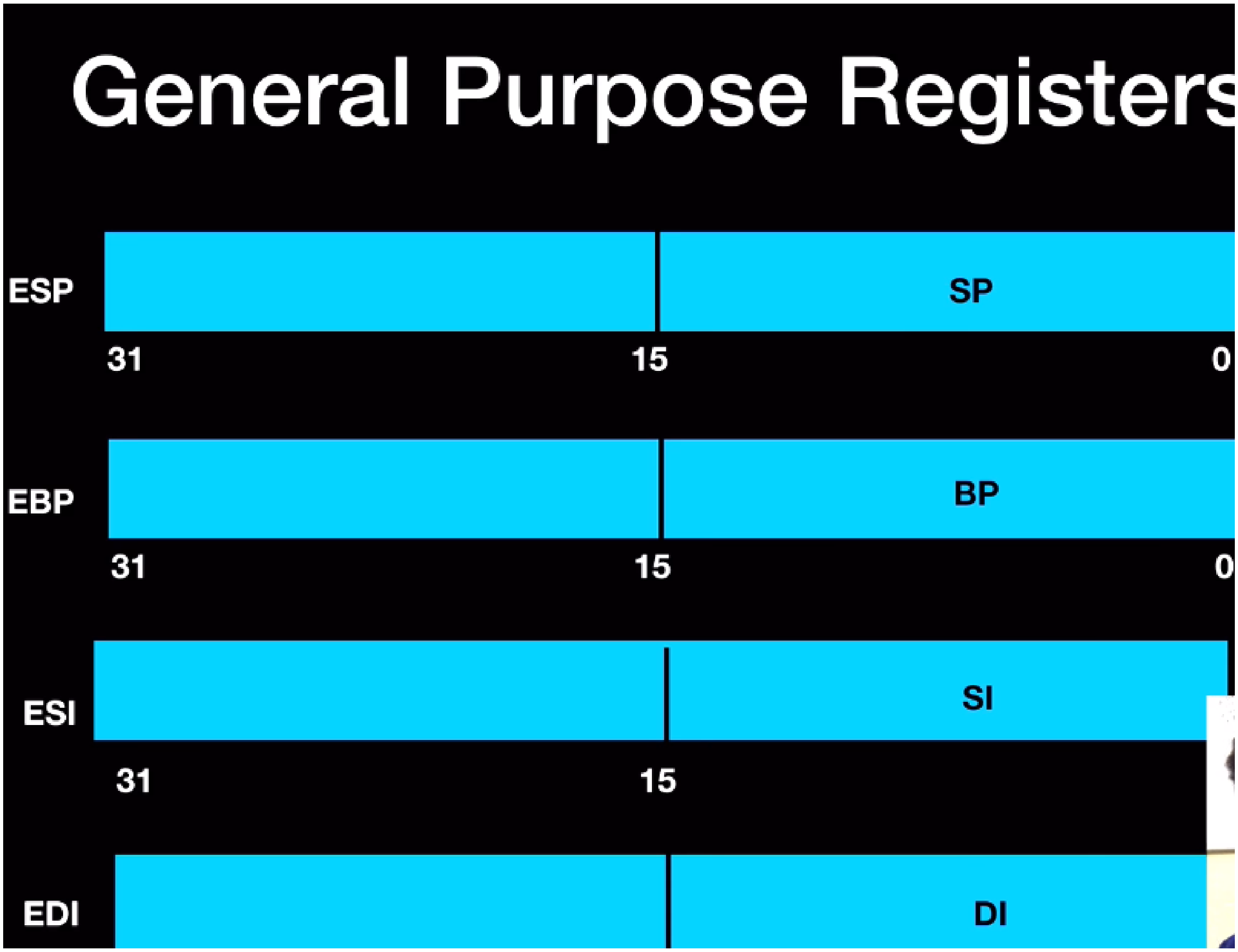
EIP

EIP	Pointer to next instruction to execute (instruction pointer)

accessing the lower and higher part of each registers

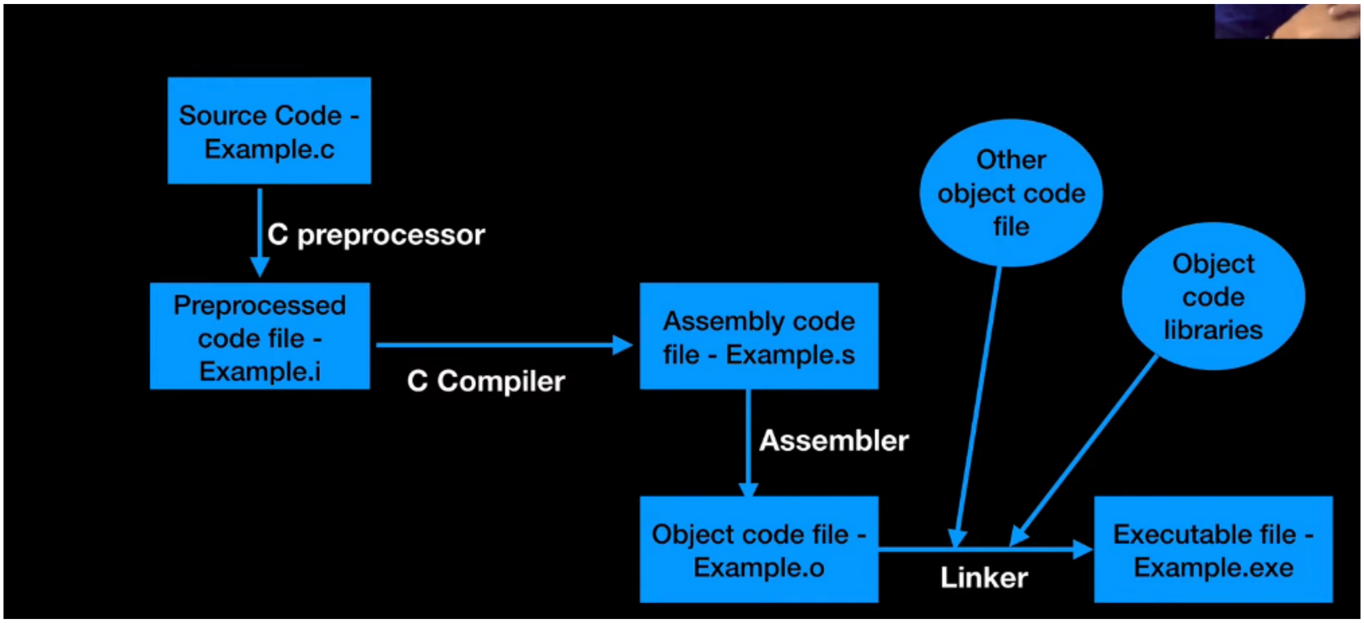


General Purpose Registers



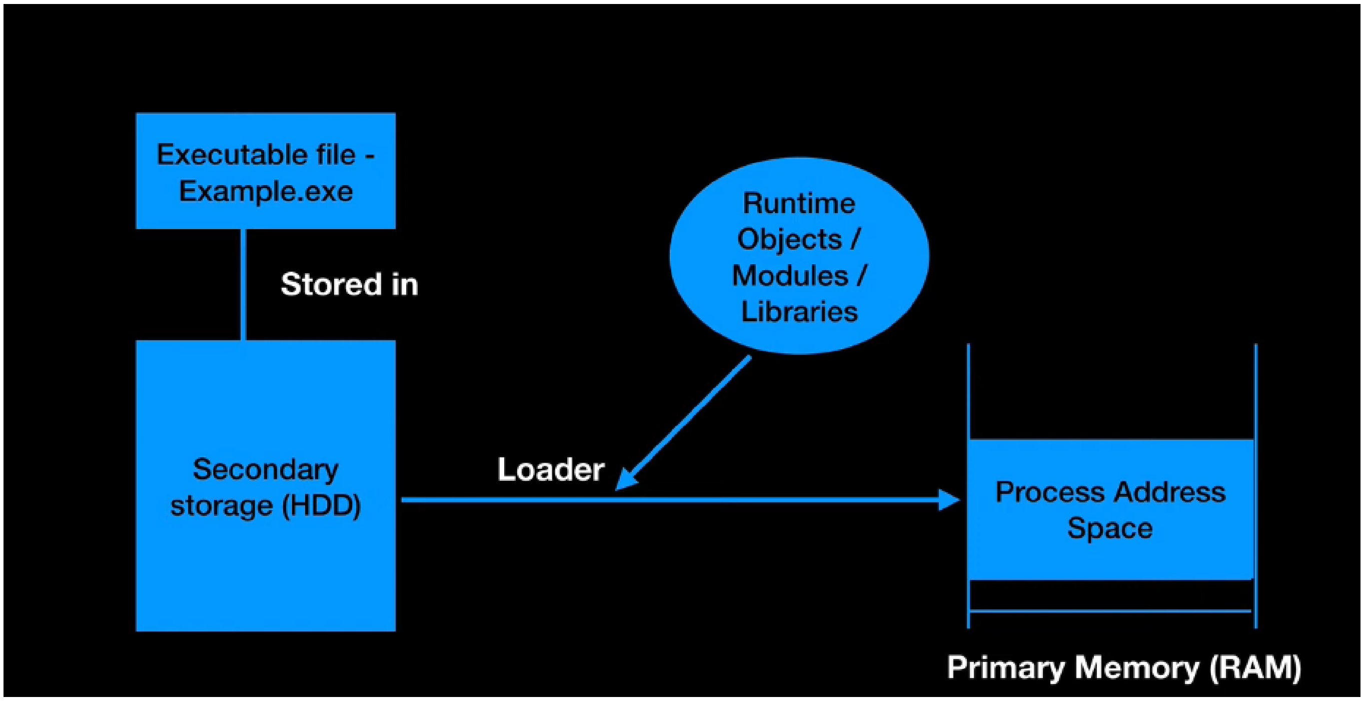
General Purpose Registers

How a C program Compiled:


Source Code - Example.c
C preprocessor
Preprocessed code file - Example.i
C Compiler
Assembly code file - Example.s
Assembler
Other object code file
Object code libraries
Object code file - Example.o
Linker
Executable file - Example.exe

1. C preprocessor will pre process the file
2. C compiler will compile the file and give us assembly code file
3. Assembler will convert the code to object file
4. Linker will convert the object file to an executable file such as .exe

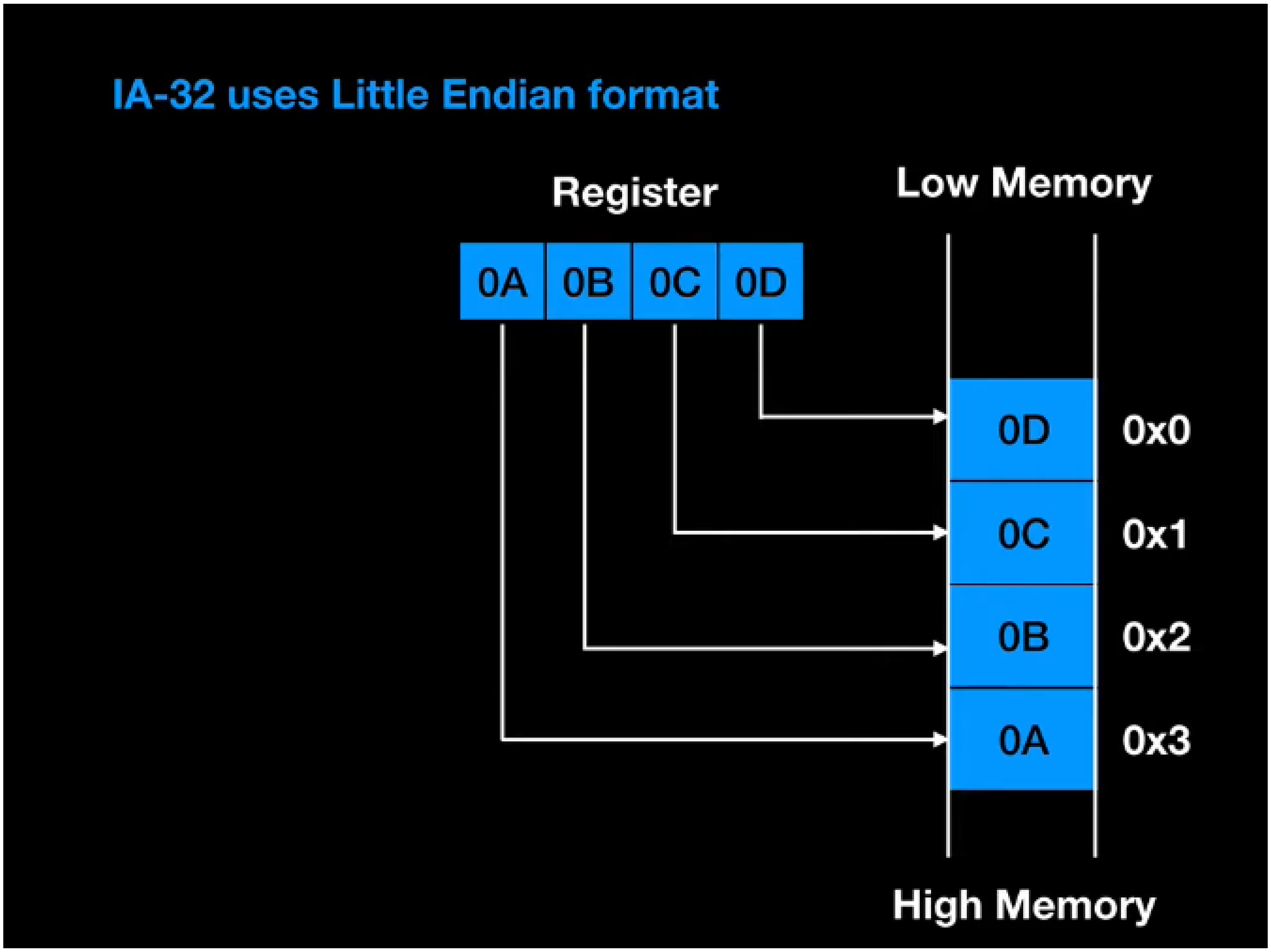
How a C program get executed:



1. Firs the executable file will be located in our secondary storage.
2. The loader will load all the modules, libraries and runtime objects.
3. Loader will allocate a space in RAM for our program, which is process address space.

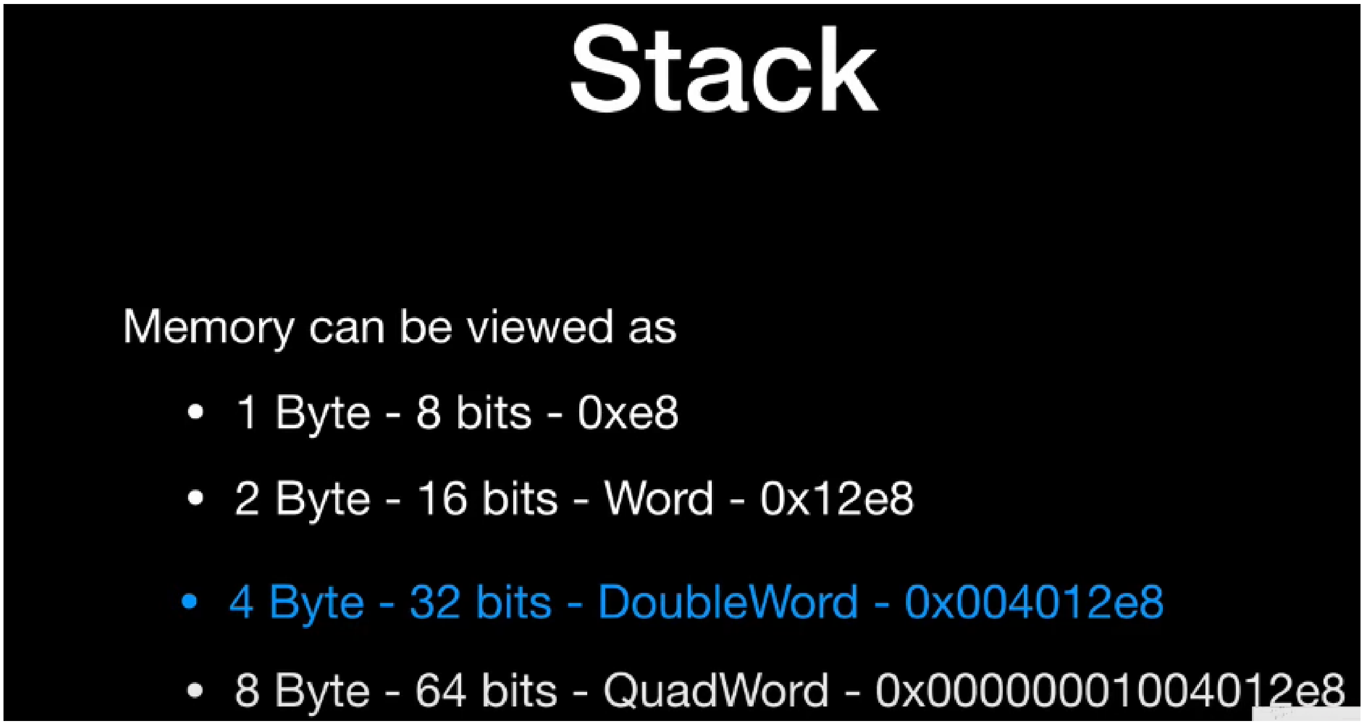
Little Endian Format

Intel Architecture 32 (IA 32) uses Little Endian Format.



IA-32 uses Little Endian format

This is how the register is stored in Memory. Its for Little Endian Format.

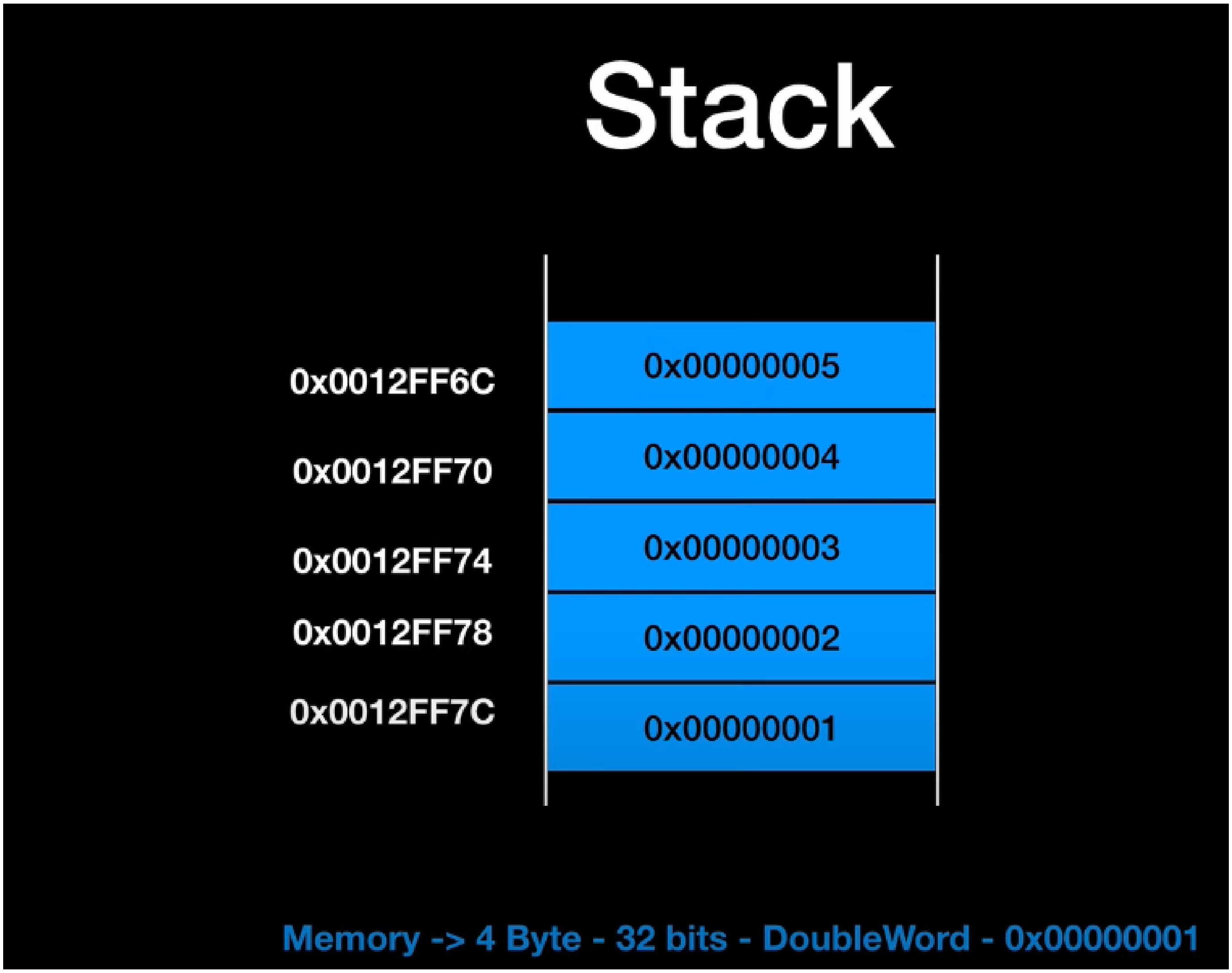


Stack

Memory can be viewed as

- 1 Byte - 8 bits - 0xe8
- 2 Byte - 16 bits - Word - 0x12e8
- 4 Byte - 32 bits - DoubleWord - 0x004012e8
- 8 Byte - 64 bits - QuadWord - 0x00000001004012e8

Memory format in Stack:



Stack

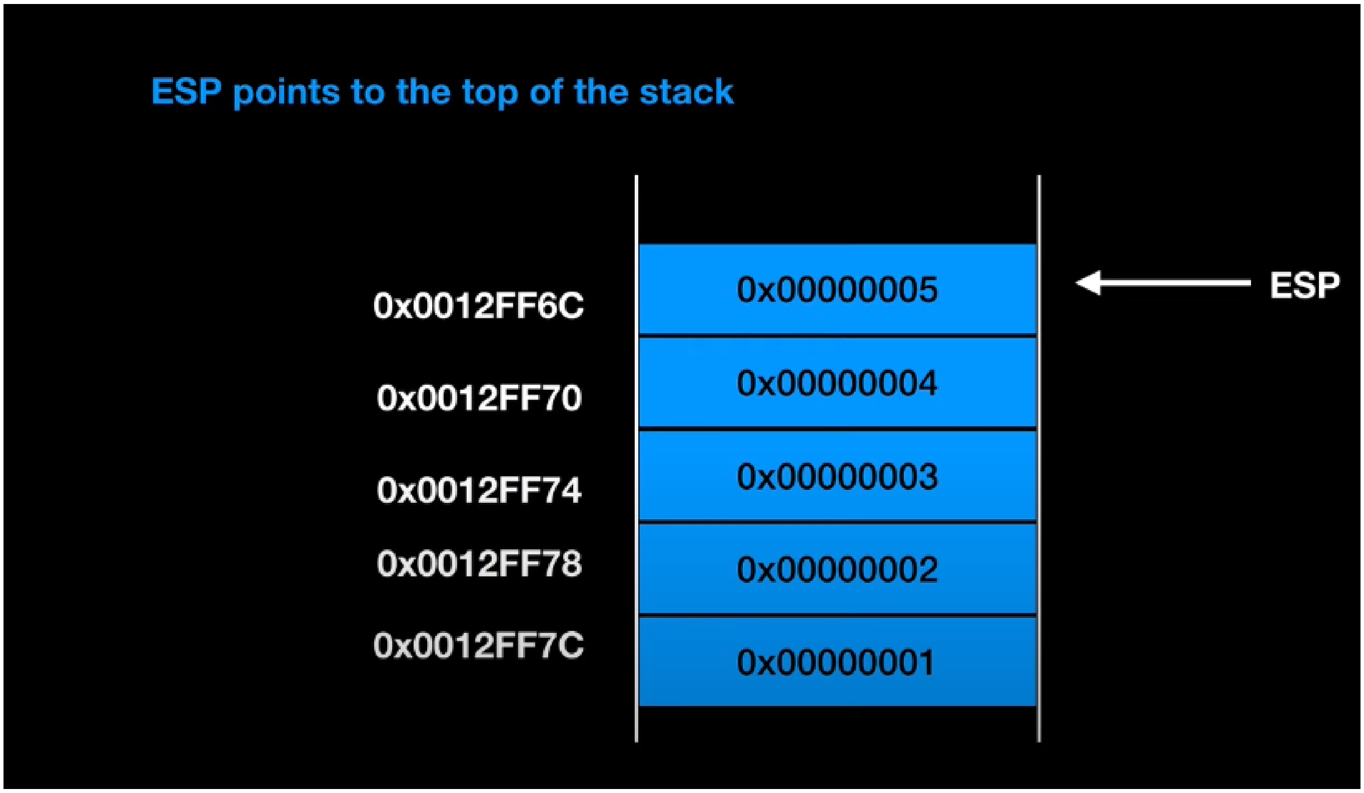
0x0012FF6C	0x00000005
0x0012FF70	0x00000004
0x0012FF74	0x00000003
0x0012FF78	0x00000002
0x0012FF7C	0x00000001

Memory -> 4 Byte - 32 bits - DoubleWord - 0x00000001

Here is an example data in stack. On the left side shows is the memory. Inside the stack the data is there.

ESP - Stack Pointer

It always points to the top of the stack, and that point will always have the lower memory address. For eg.

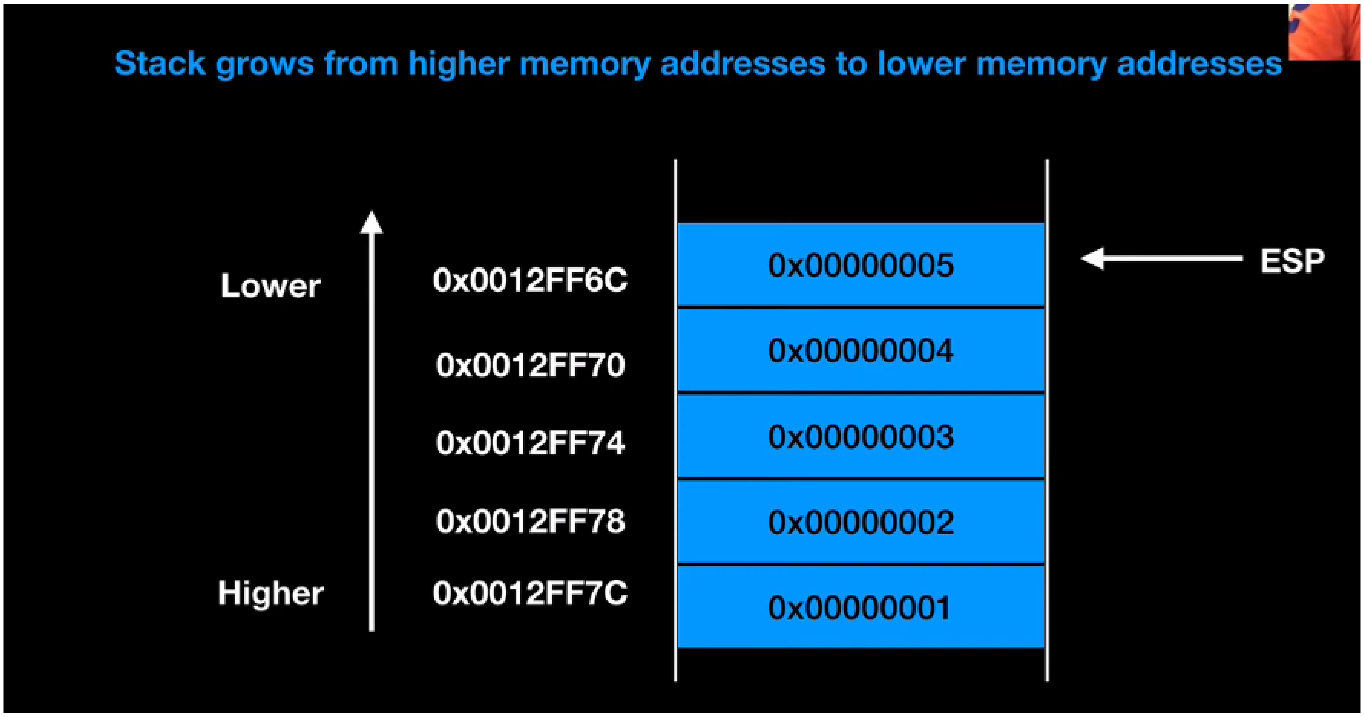


ESP points to the top of the stack

0x0012FF6C	0x00000005
0x0012FF70	0x00000004
0x0012FF74	0x00000003
0x0012FF78	0x00000002
0x0012FF7C	0x00000001

Here as you can see the lowest memory address is ESP -> 0x0012FF6C. And the ESP points to the lowest memory address.

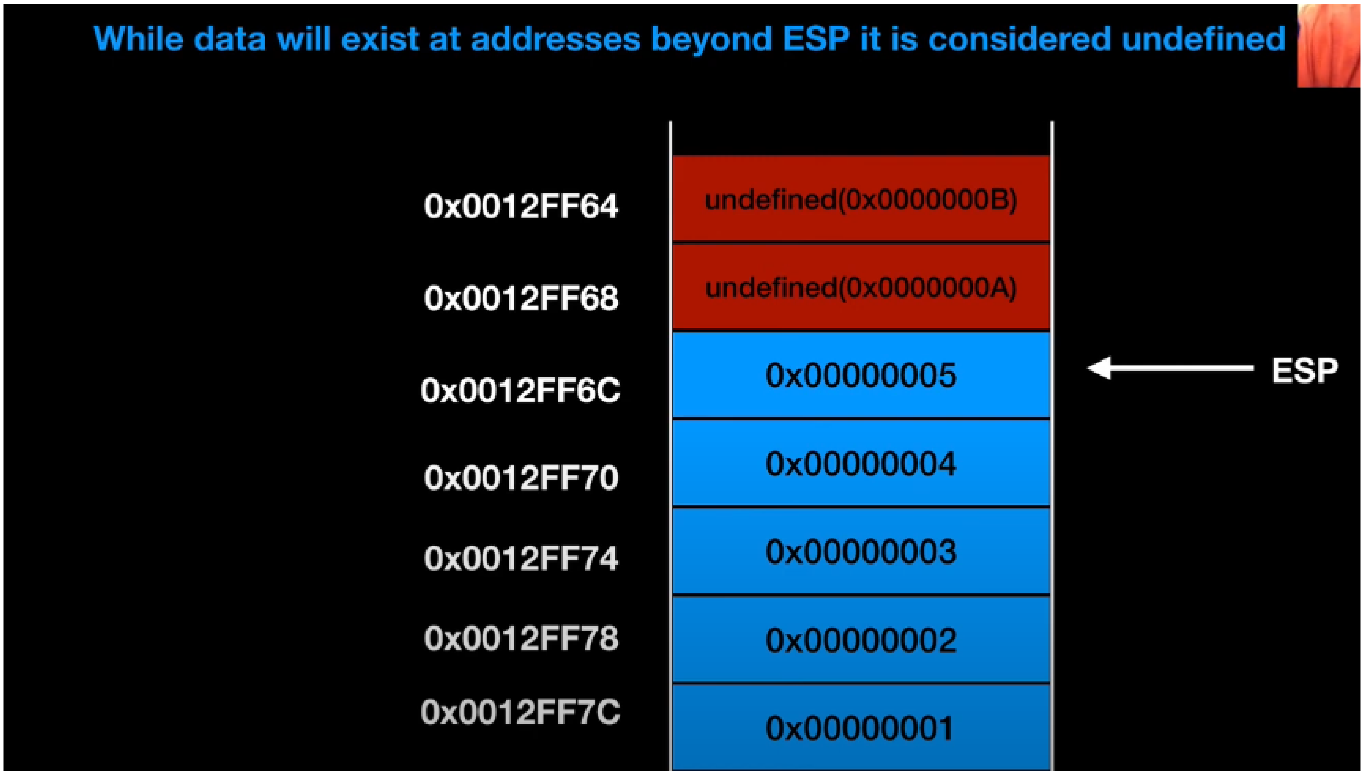
Stack grows from Higher mem address to Lower mem address.



Stack grows from higher memory addresses to lower memory addresses

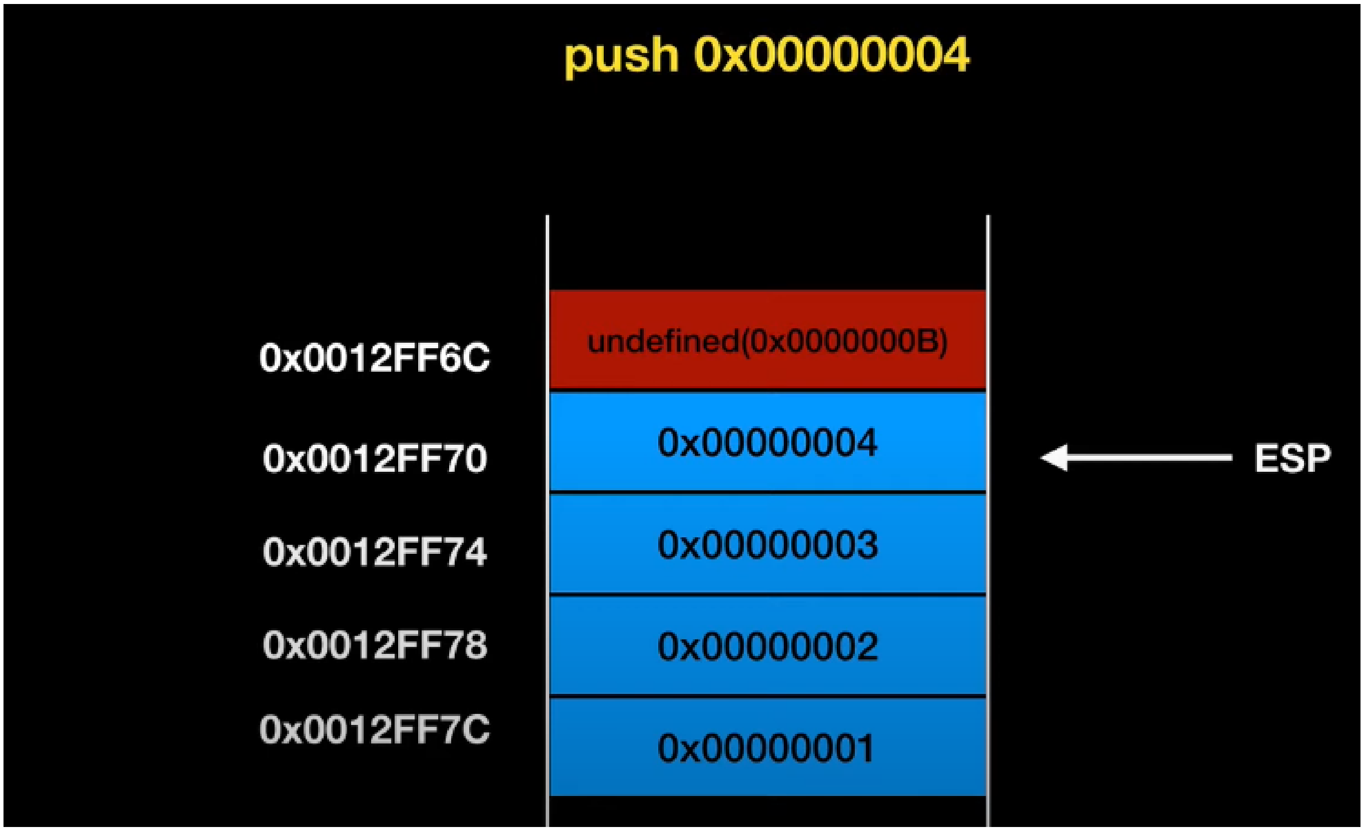
0x0012FF6C	0x00000005
0x0012FF70	0x00000004
0x0012FF74	0x00000003
0x0012FF78	0x00000002
0x0012FF7C	0x00000001

There will be some address above the ESP. Those will be referred as undefined. We can't access it.



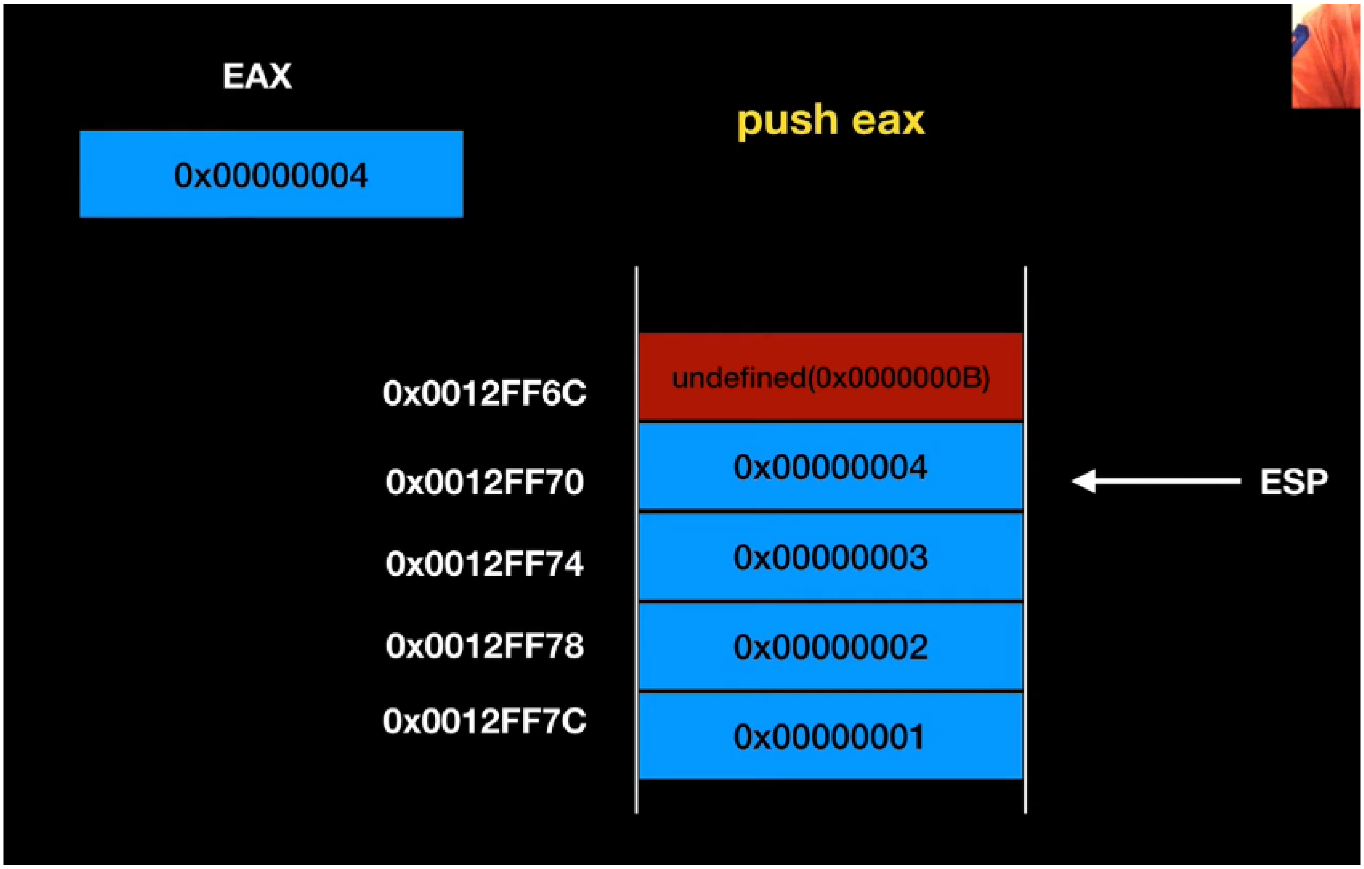
PUSH

Push Immediate value:



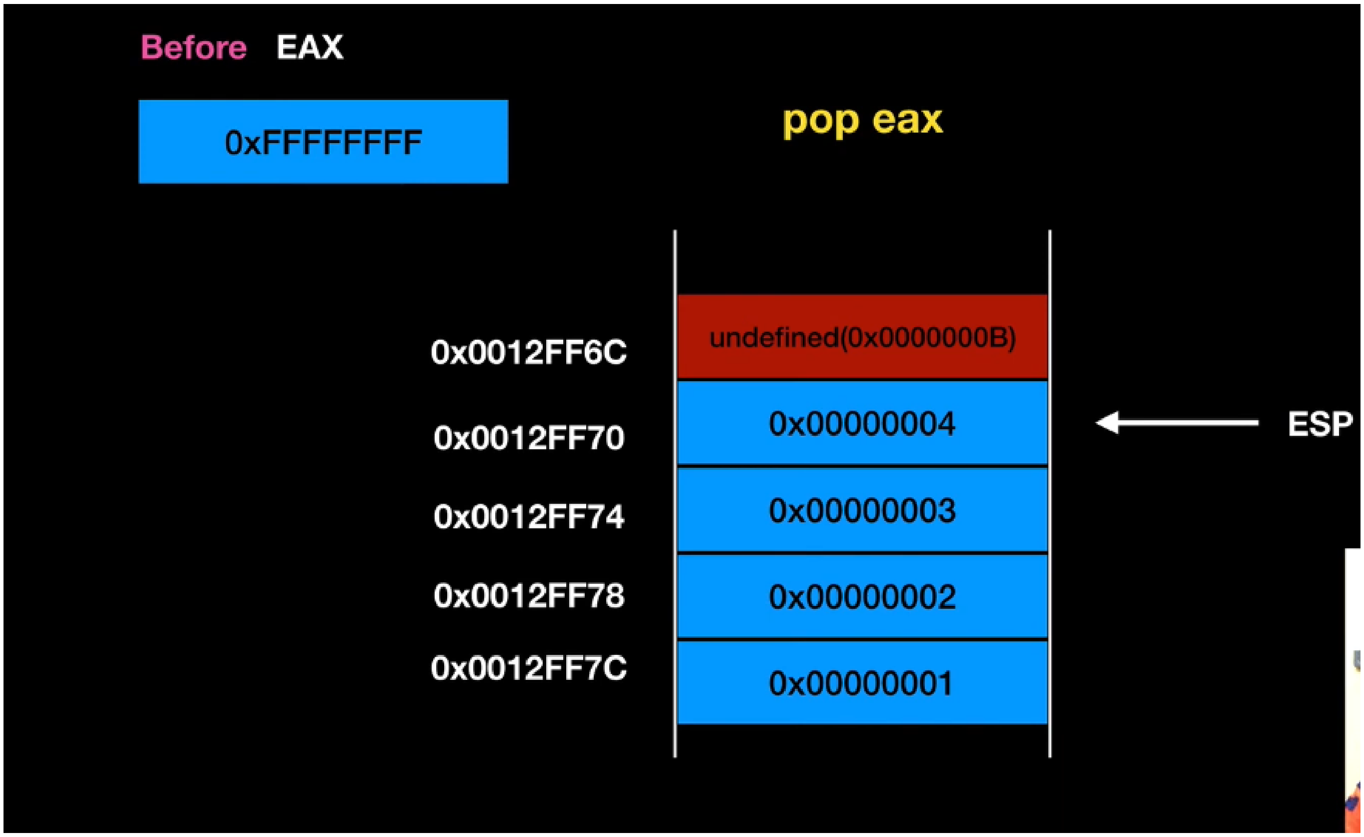
Here we have pushed 0x00000004 (value) into the stack. And now the ESP pointer will get reduced by 4, which will point to the current top ie) the last inserted value.

Push Register value:

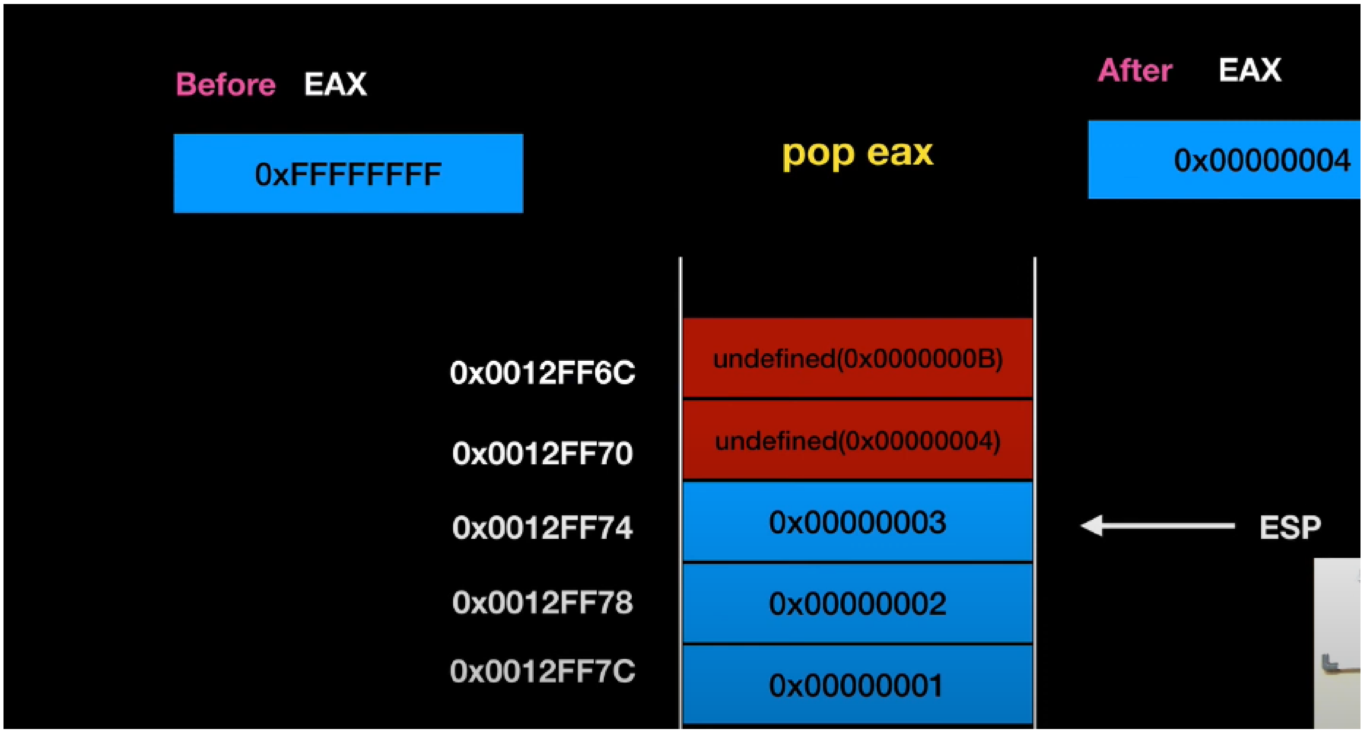


Here the EAX register is having the value 0x00000004. And now we are push EAX into stack. It will push the value stored in EAX into the stack. And the ESP will point to the Top.

POP

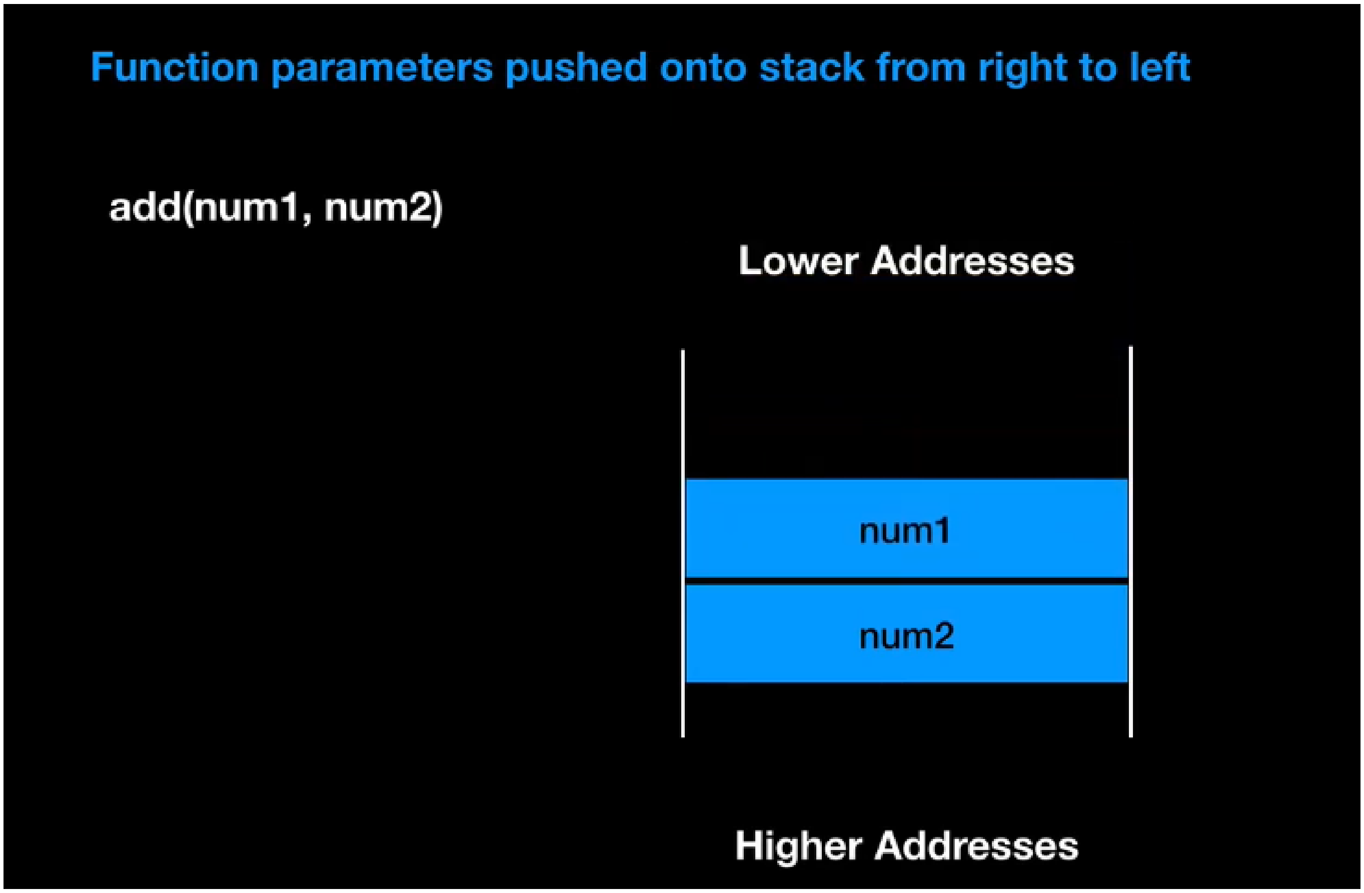


here we have a register EAX with no value.
Now we do the operation "pop eax".
This will pop the top value and stored it into EAX register and the ESP pointer will be incremented by 4.



Calling Conventions

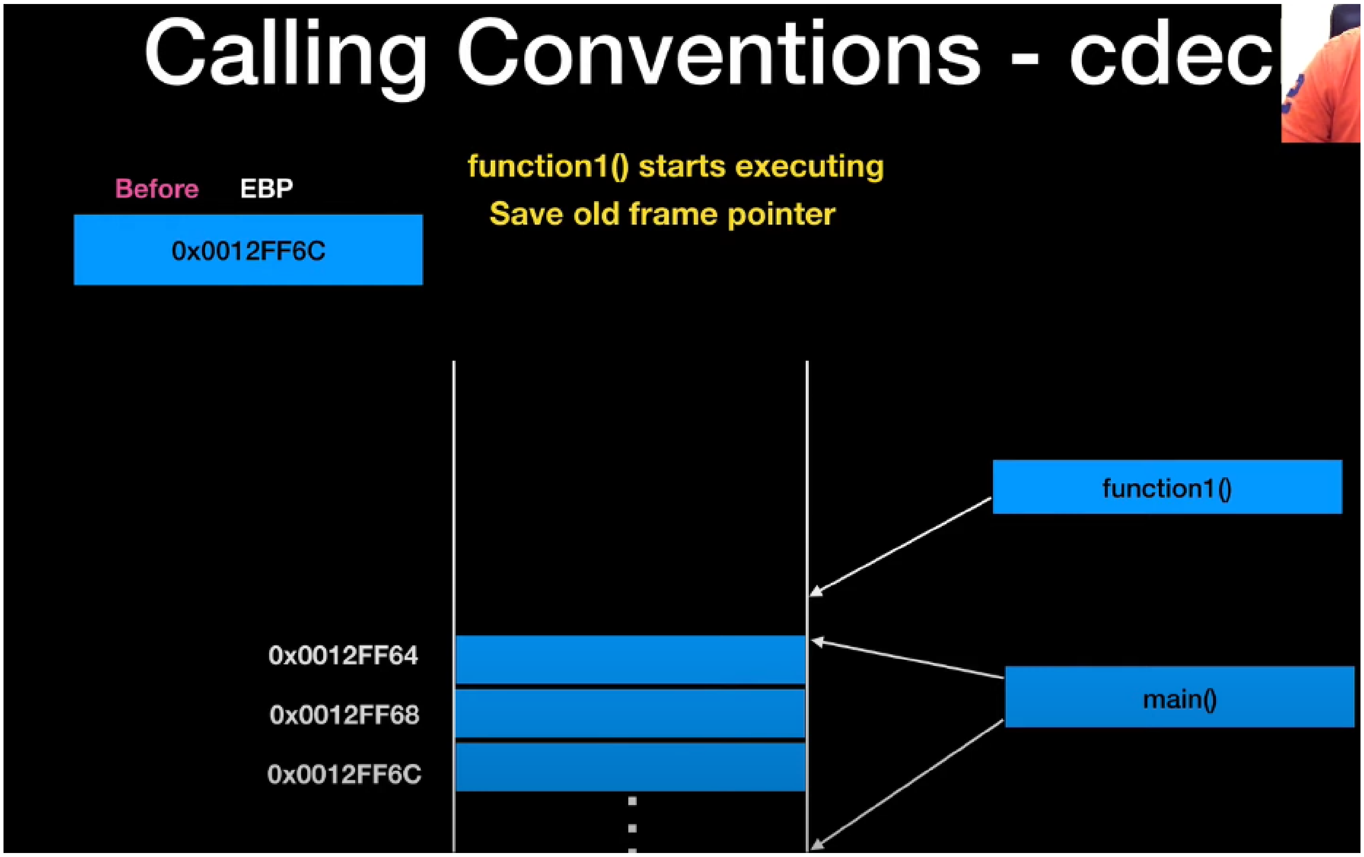
Callings conventions are about how one function calls the another function. It depends on compiler and can be configured. cdecl - This is one the most common calling convention. cdecl means C declaration.



Function parameters pushed onto stack from right to left

add(num1, num2)

Here we can see the num2 is pushed into stack first, so it push the parameters from right to left.



Calling Conventions - cdec

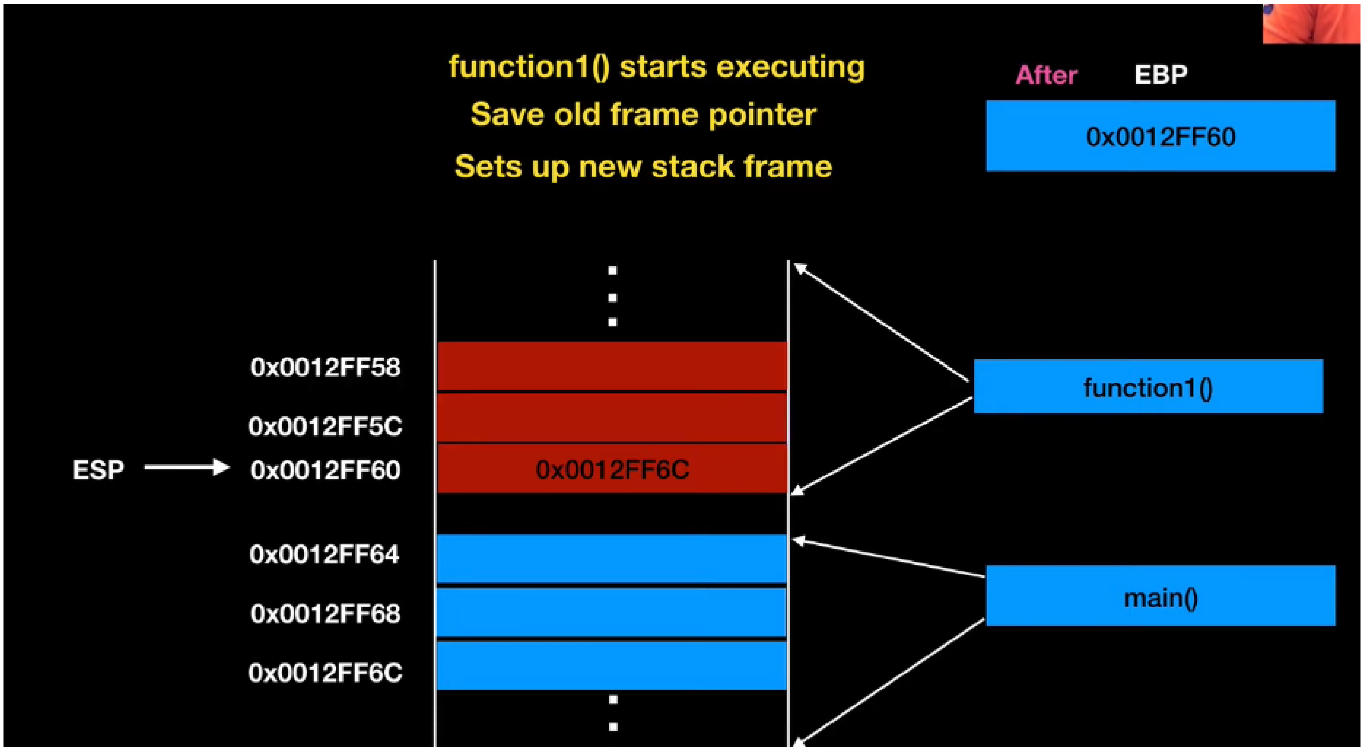
function1() starts executing

Save old frame pointer

We can see the stack frame of the main function. Above that we can see the stack frame of function1()

Now EBP will have the base address of the main();

Now the function1() is called.



function1() starts executing

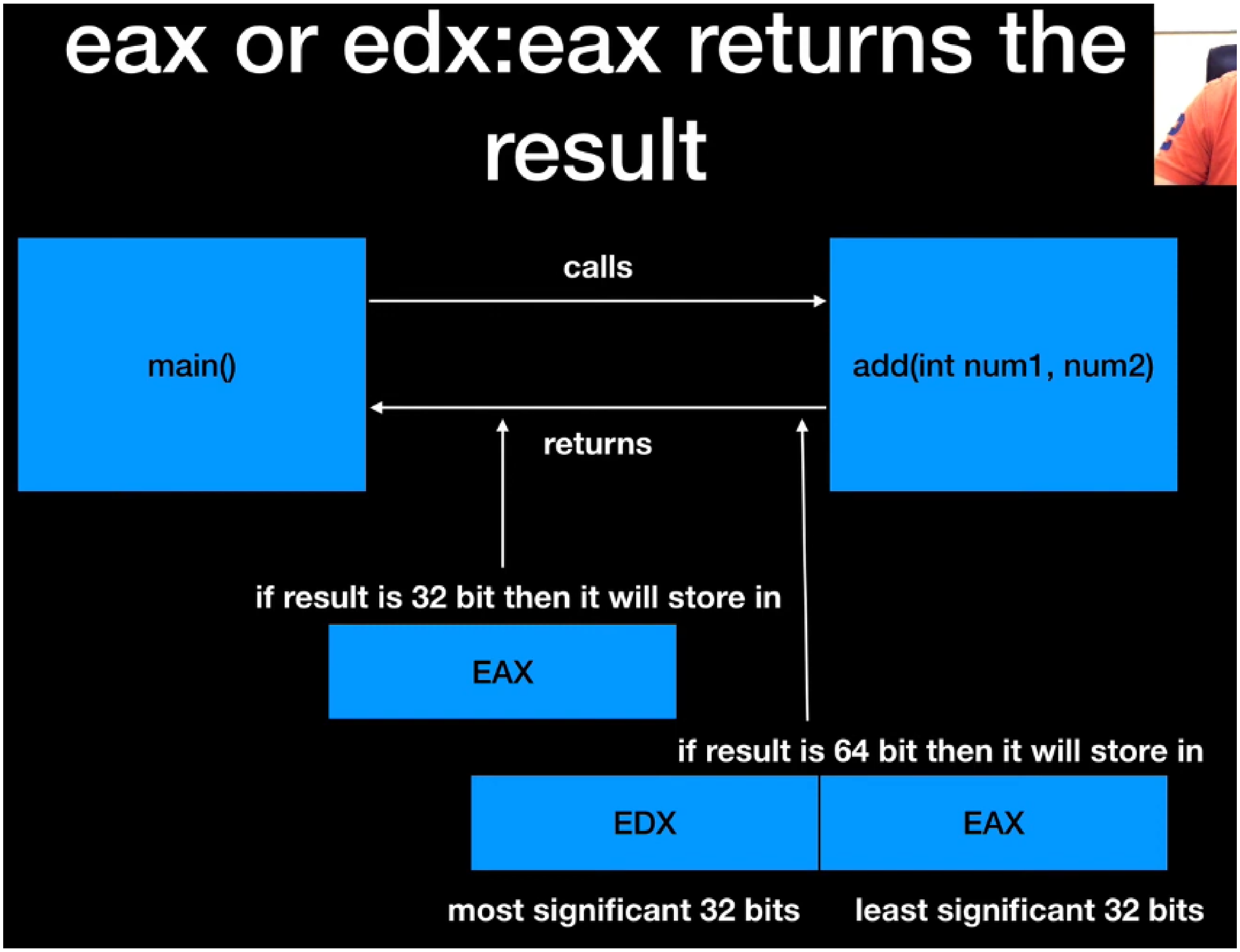
Save old frame pointer

Sets up new stack frame

Now the EBP will be pushed into the stack. And the EBP address is set to the current ESP address which is the function1()'s base address.

As you can see the function1()'s stack frame is set.

EAX



eax or edx:eax returns the result

main()

calls

add(int num1, num2)

returns

if result is 32 bit then it will store in

EAX

if result is 64 bit then it will store in

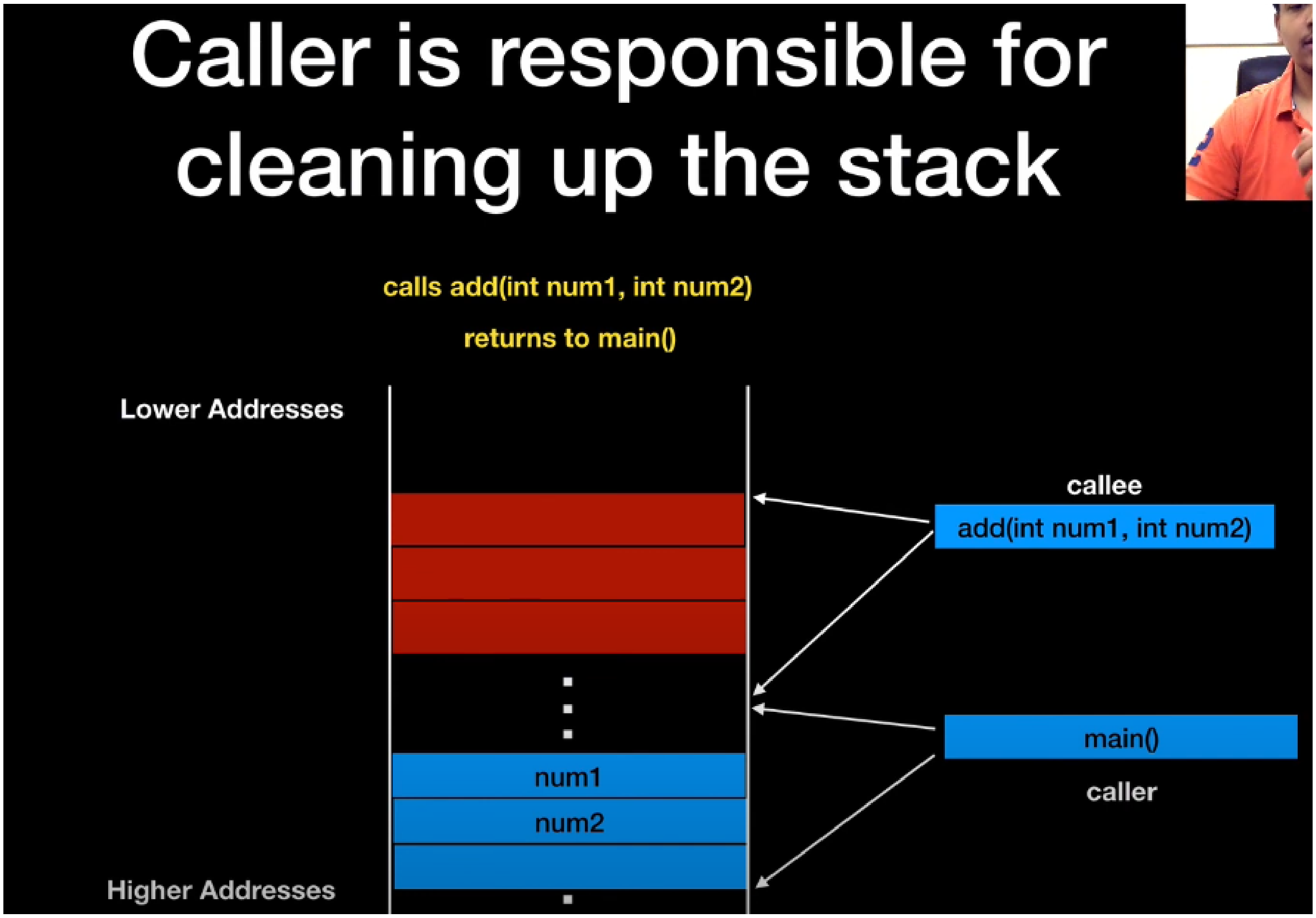
EDX

EAX

most significant 32 bits

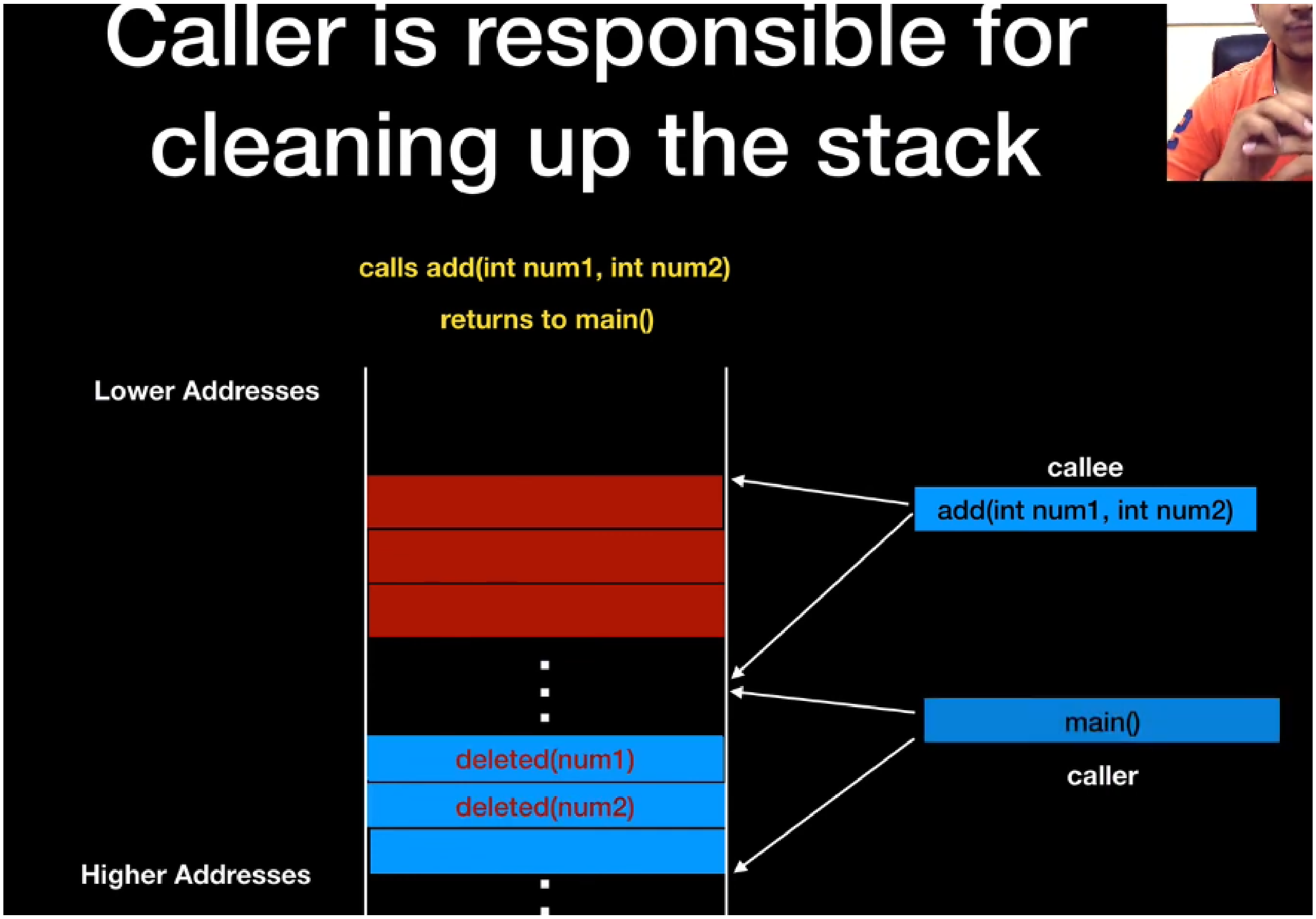
least significant 32 bits

EAX will store the the functions() result value. If result is 32 bits then eax will have the value.
If result is 64 bits EDX, EAX will have the value, where EDX will have MSB 32 bits and EAX will have LSB 32 bits.



Caller is responsible for cleaning up the stack

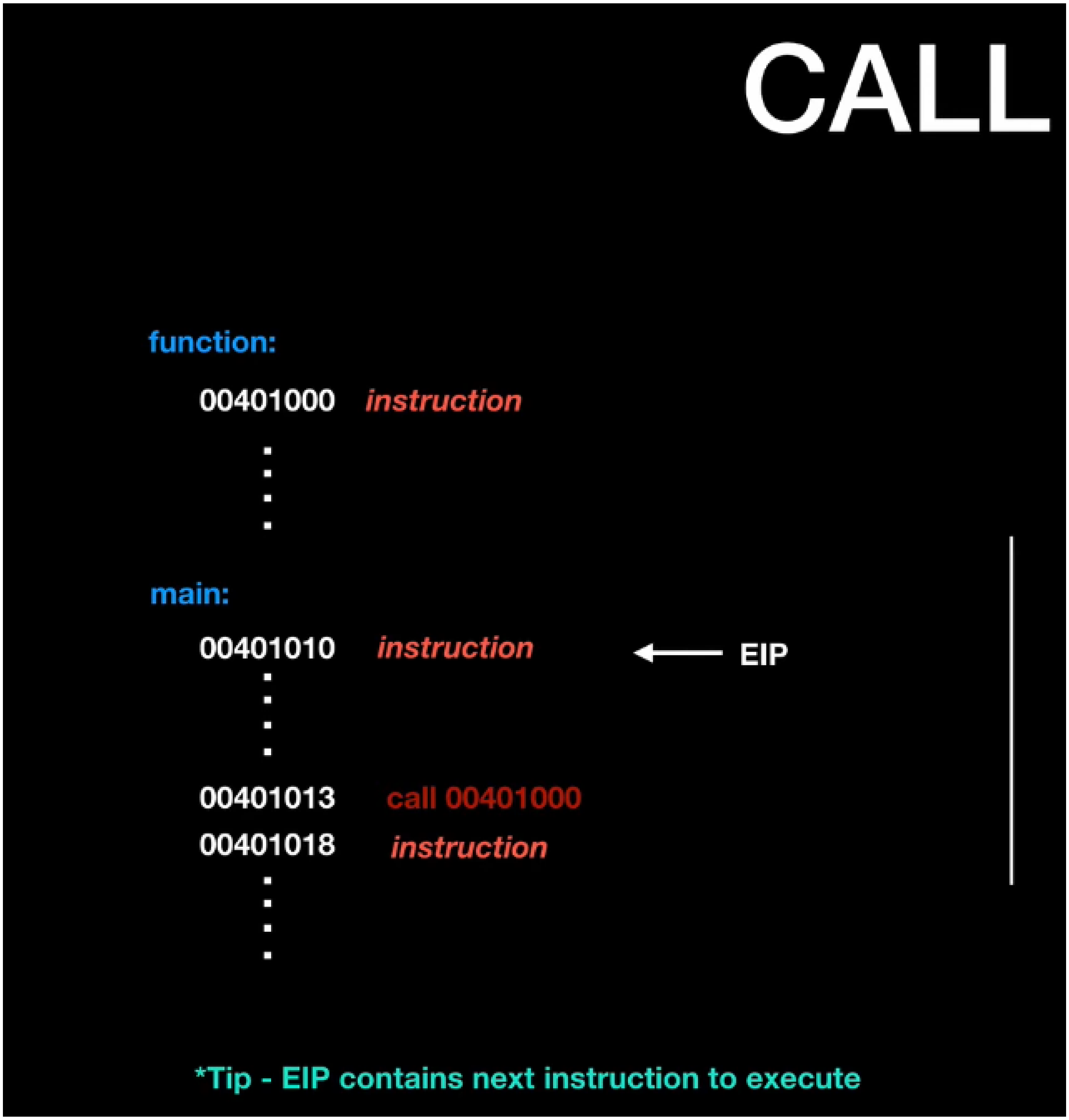
Here we have main() and add(). The parameters num1 and num2 are pushed into stack before it is called.



Caller is responsible for cleaning up the stack

after the function add() is called , the parameters inside the main stack frame are deleted. Its because in cdecl, caller is responsible for cleaning up the stack.

CALL

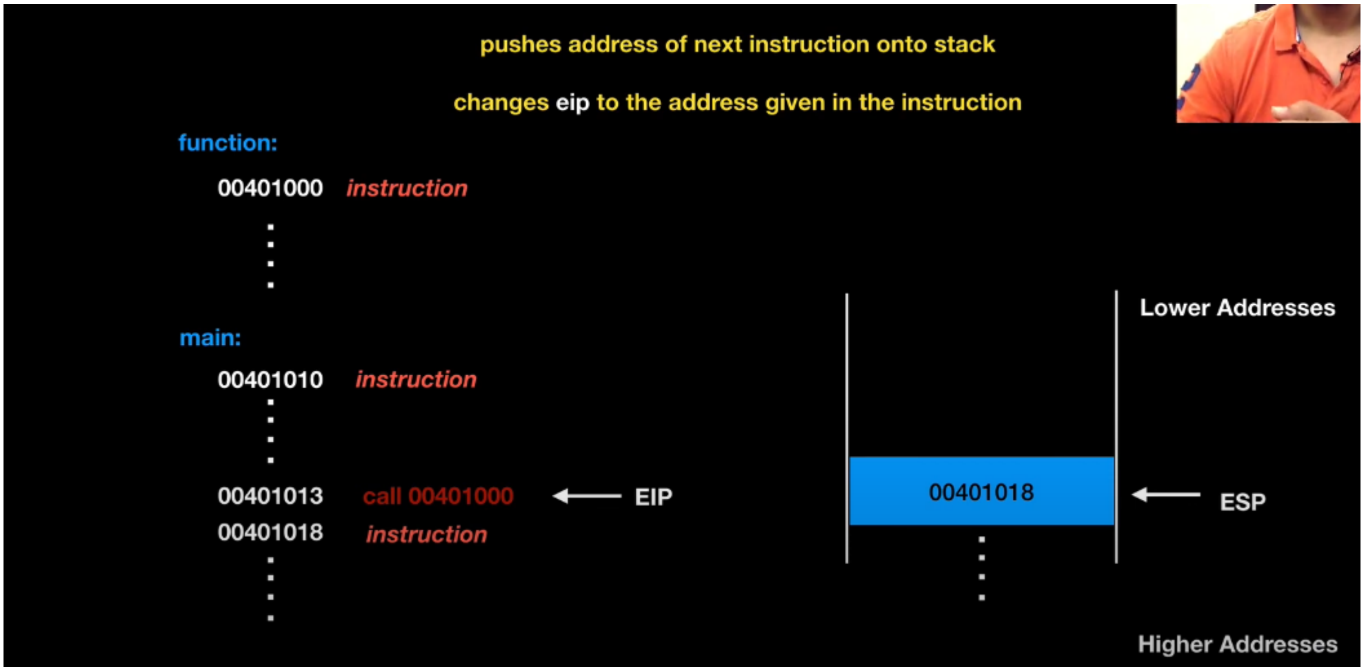


CALL

```
function:
    00401000   instruction
        ⋮
main:
    00401010   instruction        ← EIP
        ⋮
    00401013   call 00401000
    00401018   instruction
        ⋮
```

*Tip - EIP contains next instruction to execute

EIP is pointing the first instruction in main. Lets see what happens when EIP encounters call instruction. ie) main() calls a function().



pushes address of next instruction onto stack

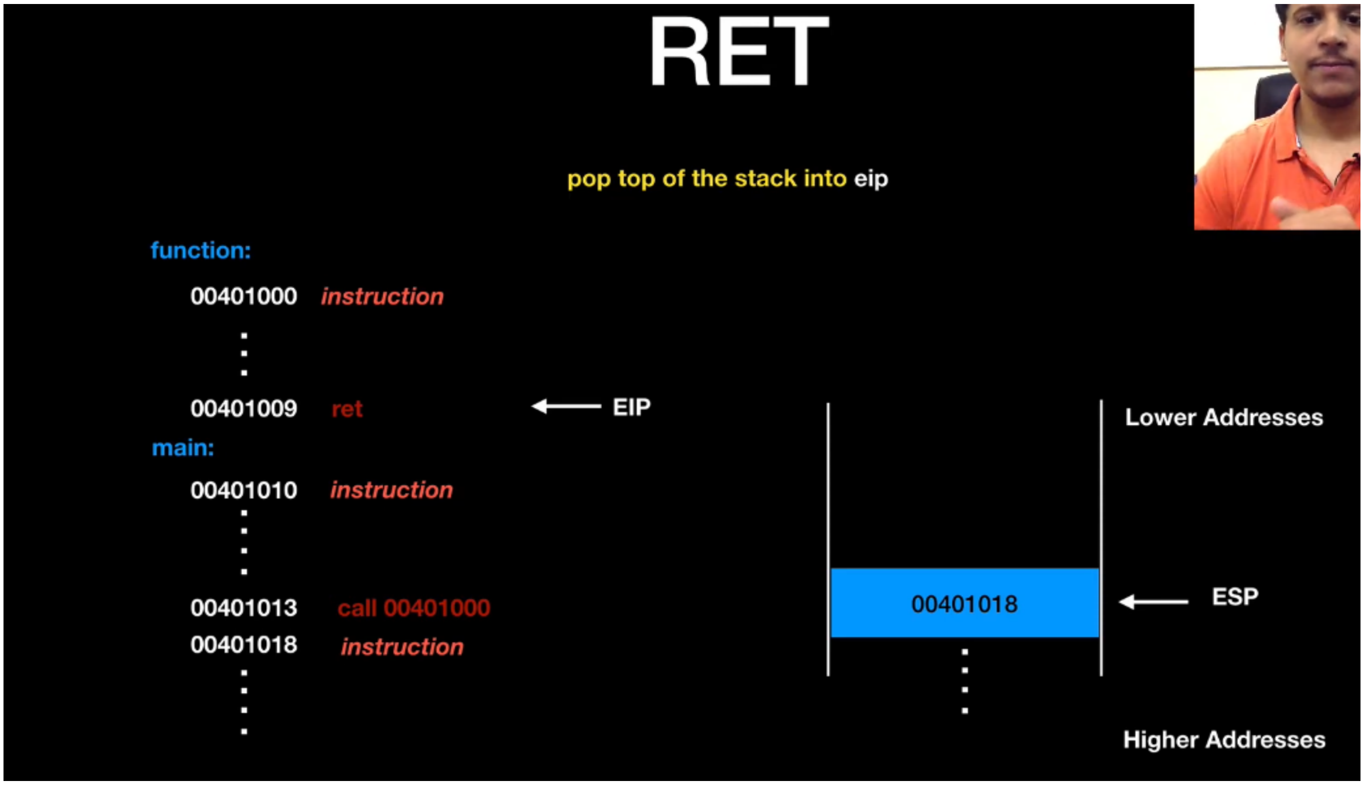
changes eip to the address given in the instruction

```
function:
    00401000   instruction
       :
main:
    00401010   instruction
       :
    00401013   call 00401000   <---- EIP
    00401018   instruction
       :
```

At that time, the next instruction address(00401018) is pushed into the stack .
And changes the EIP address to the address mentioned by call. In our case it is 00401000.
So EIP -> 00401000.

RET

It is the return instruction. When a function() is called inside main(). After the function() completed then the pointer will be pointed back to main() with the help of RET instruction.



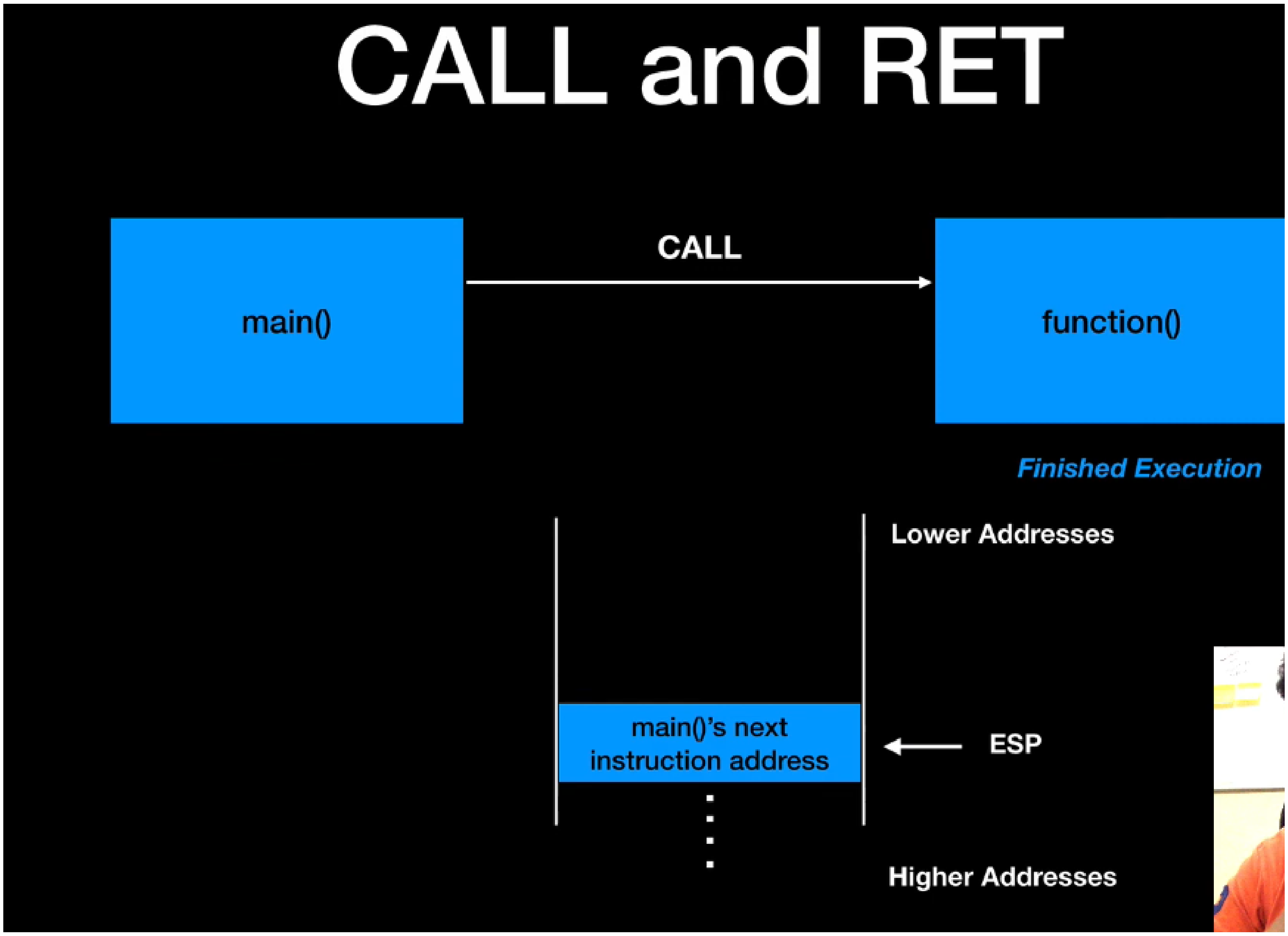
RET

pop top of the stack into eip

```
function:
    00401000   instruction
       :
    00401009   ret             <---- EIP
main:
    00401010   instruction
       :
    00401013   call 00401000
    00401018   instruction
       :
```

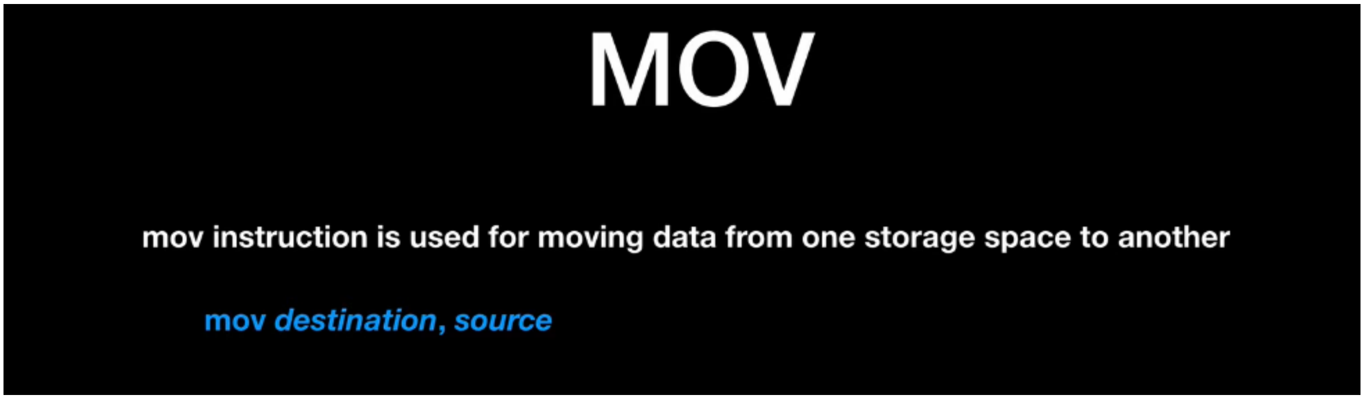
Here the call instruction is executed and all the instructions inside function are executed and now the EIP is encountering the ret.
It will pop the stack and put that value into EIP.

CALL and RET - Summary



When a main() function calls another function(), the main()'s next instruction address will be pushed into stack and the EIP will point to the function()'s stack frame. After the execution complete the ret instruction inside the function() will pop that value from stack and put it into EIP. So it will again come to the main() and execute the remaining part of it.

MOV

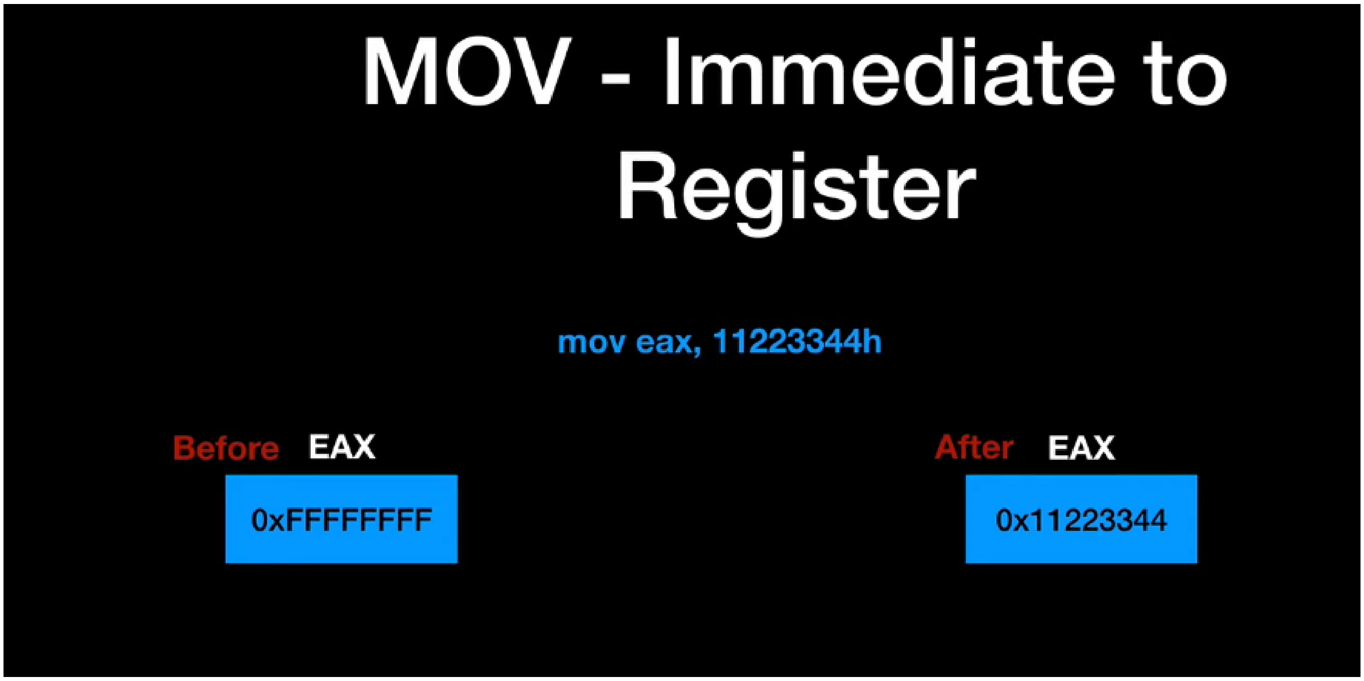


MOV

mov instruction is used for moving data from one storage space to another

mov *destination, source*

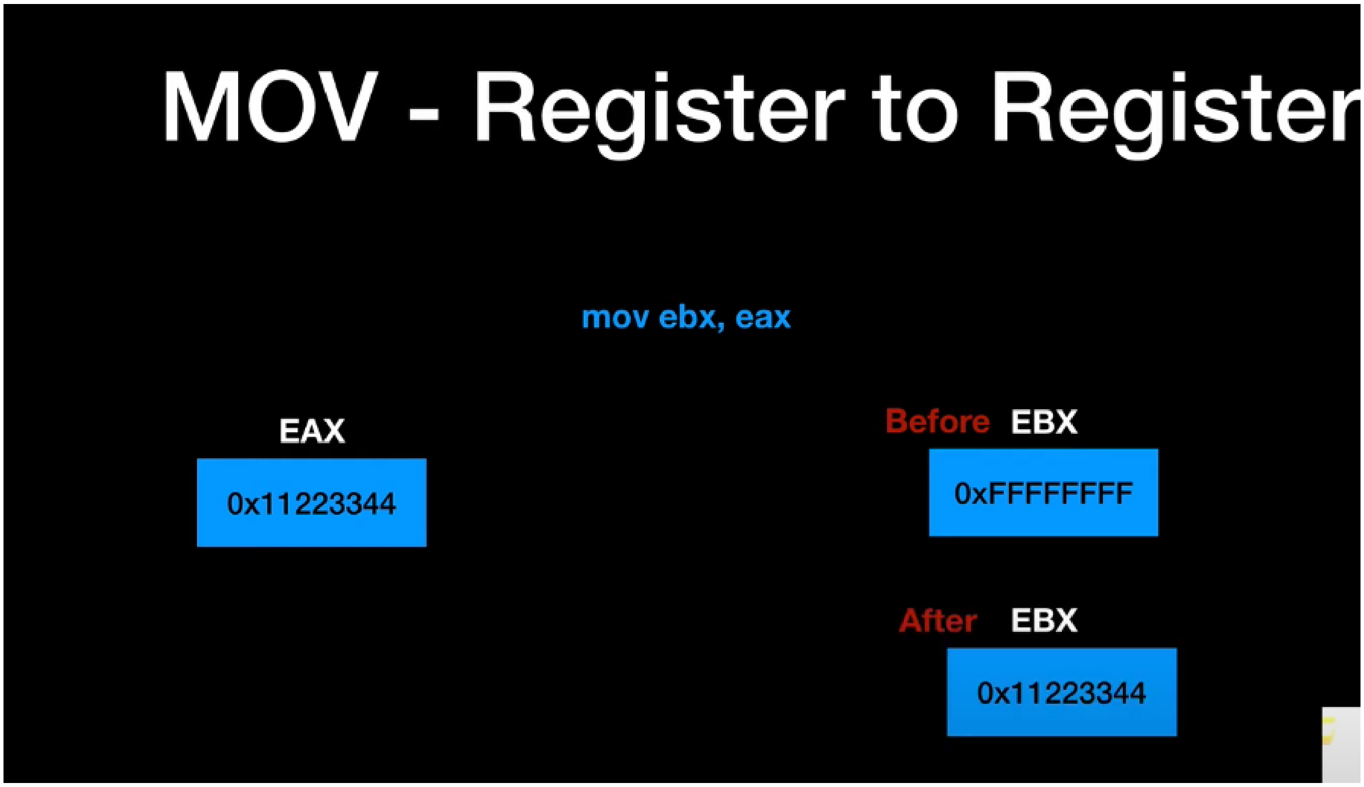
It will move data from one place to another, or copy the data and store it to a register.



MOV - Immediate to Register

mov eax, 11223344h

Here we have EAX which is empty. And after the mov instruction the value 11223344h is moved to EAX register.



MOV - Register to Register

mov ebx, eax

EAX

0x11223344

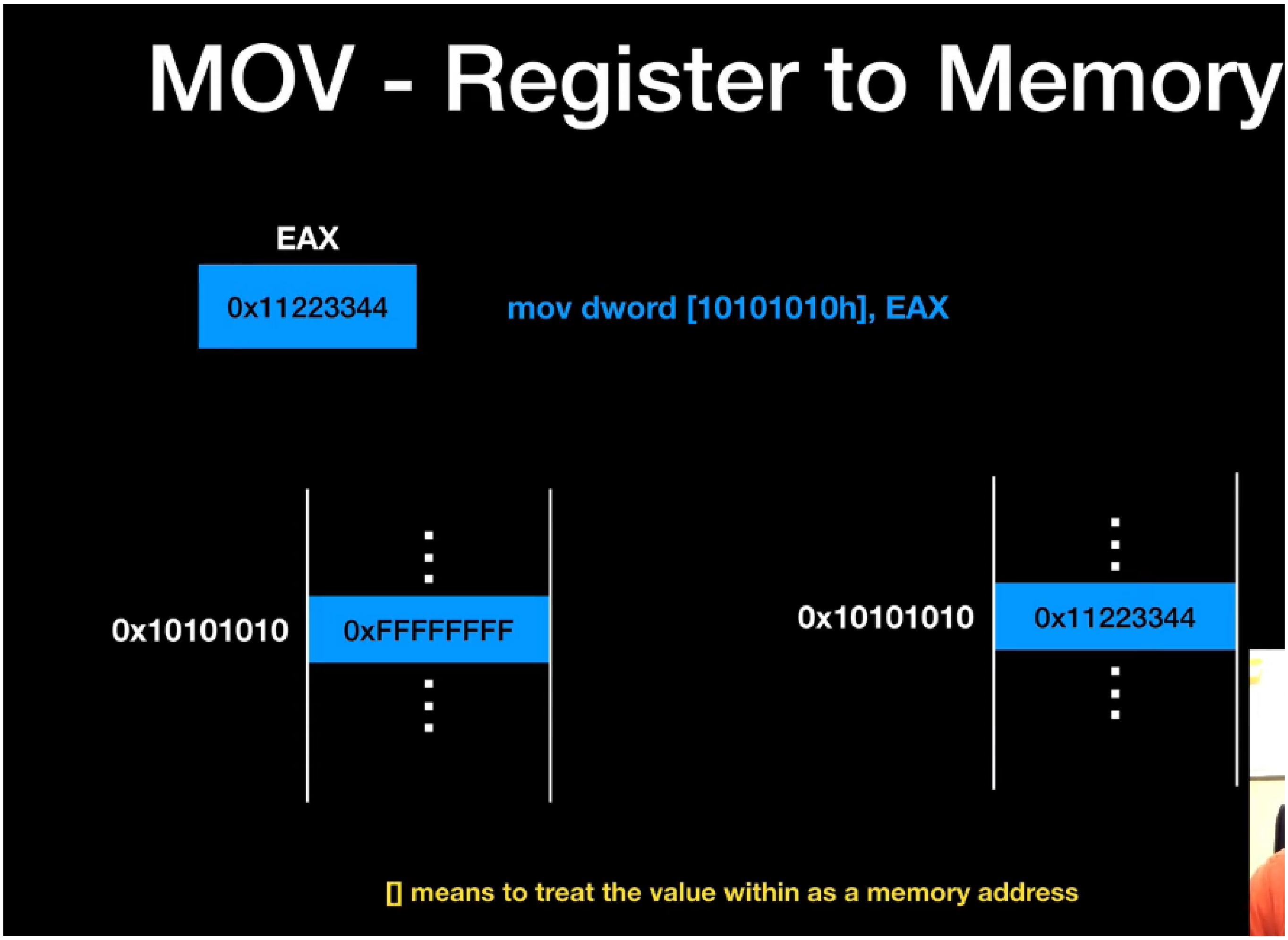
Before EBX

0xFFFFFFFF

After EBX

0x11223344

Here the value of EBX is initially empty. After the MOV operation EAX value is sent to EBX. (EBX=EAX) --> mov ebx,eax.



MOV - Register to Memory

EAX

0x11223344

```
mov dword [10101010h], EAX
```

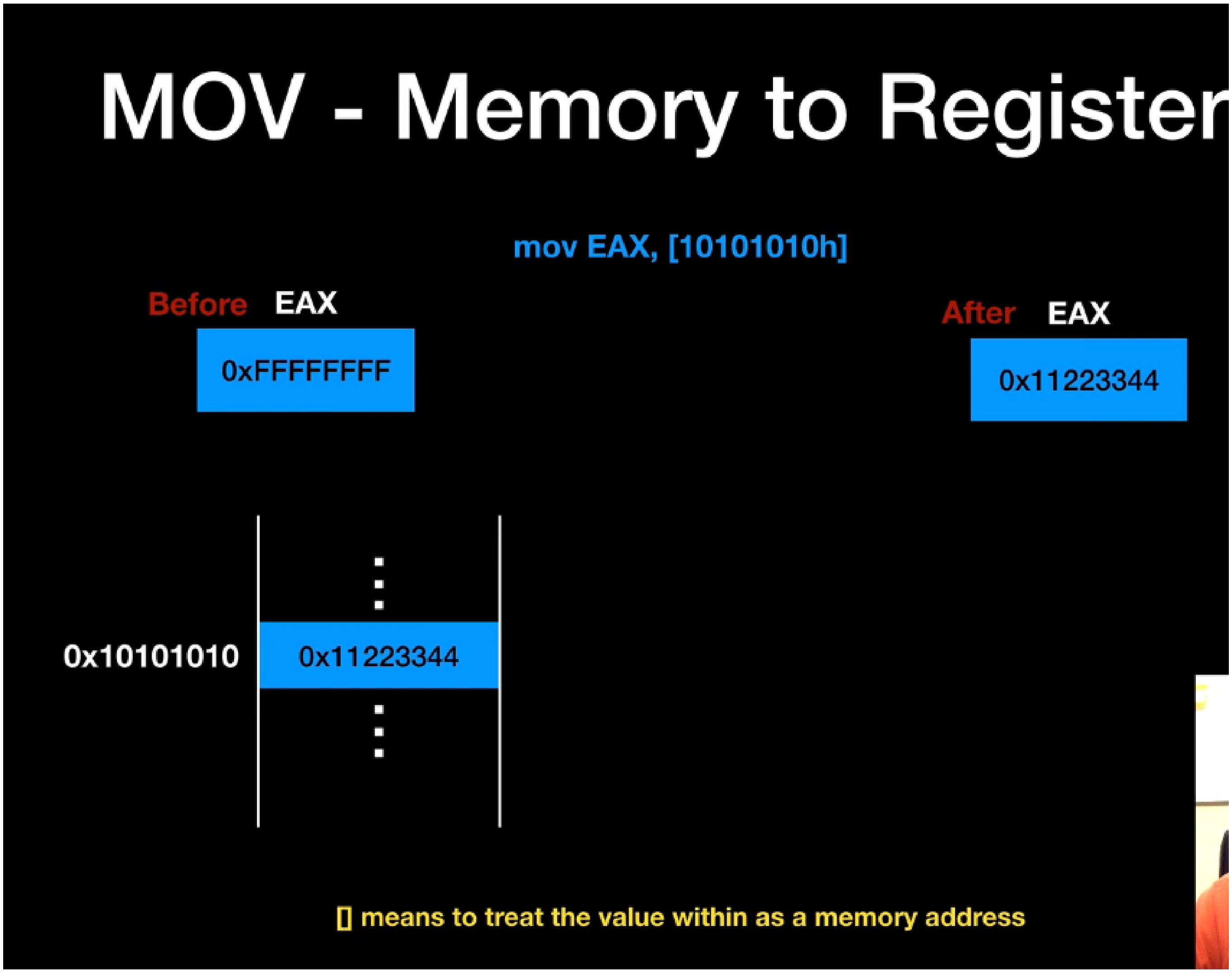
0x10101010 0xFFFFFFFF

0x10101010 0x11223344

[] means to treat the value within as a memory address

Here mov dword[address] , EAX means --> we are moving the value of EAX to the address we give inside dword. In the above example we are moving eax value to the address location 1010101010h.

Note : The values within the square brackets [ ] are treated as address .



MOV - Memory to Register

mov EAX, [10101010h]

Before EAX

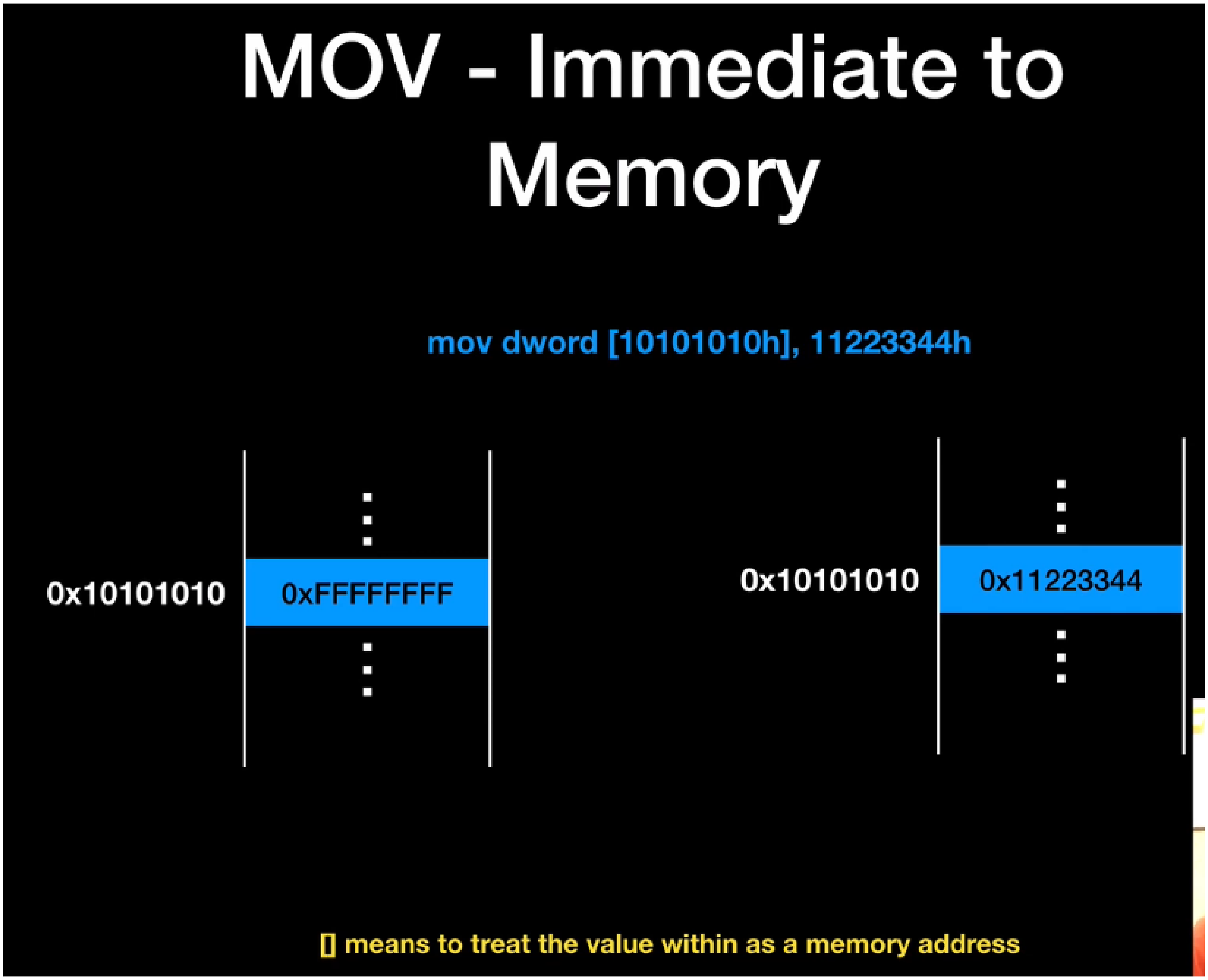
0xFFFFFFFF

After EAX

0x11223344

[] means to treat the value within as a memory address

Here we are copying the value located at the address 10101010h to the EAX register. This is the format.



MOV - Immediate to Memory

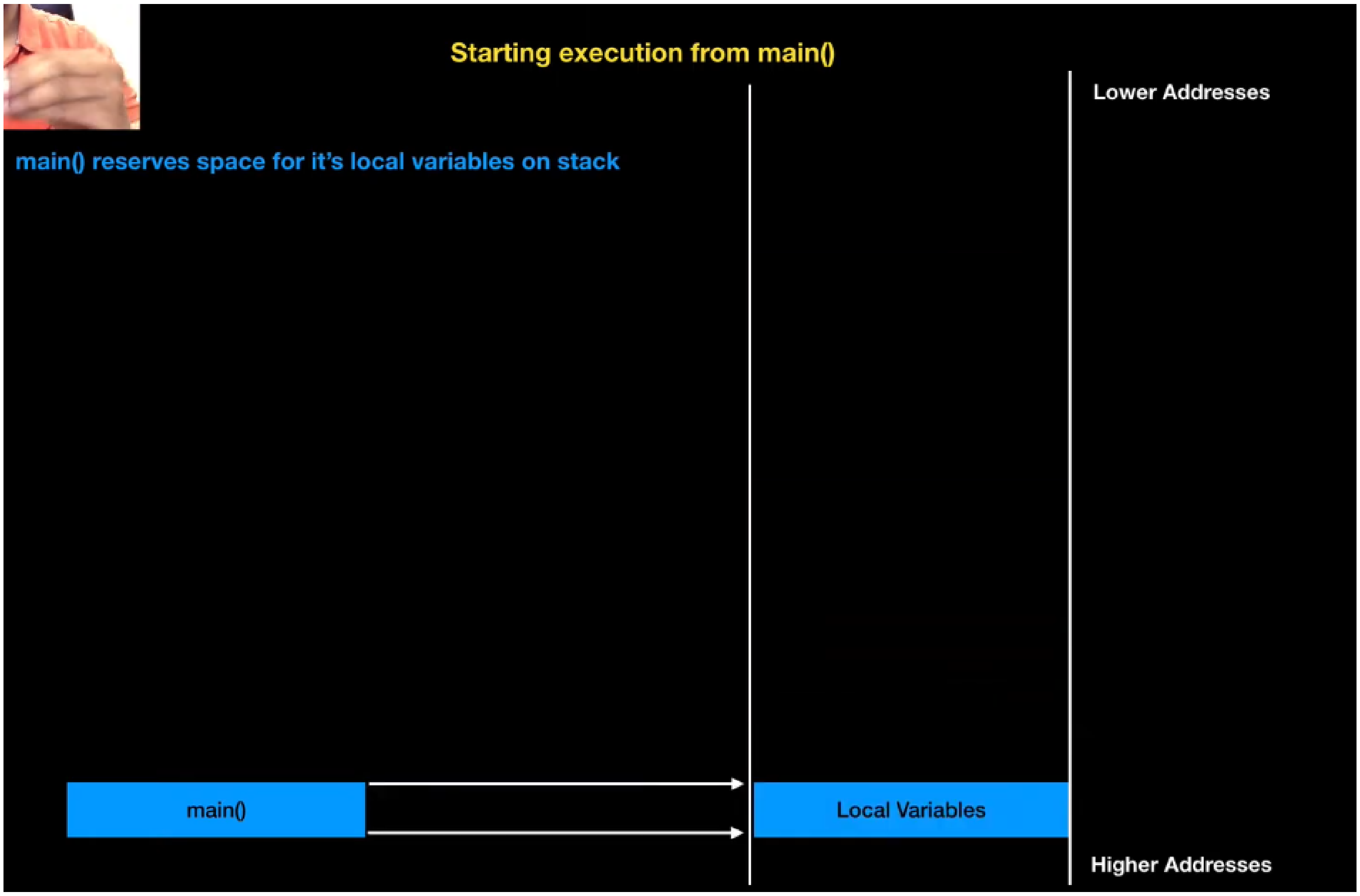
```
mov dword [10101010h], 11223344h
```

[] means to treat the value within as a memory address

Here we are copying the value 11223344h to the memory location 10101010h. Now that location will the specified value.

We can't mov values from memory address to memory address

Stack Frame Operation



Starting execution from main()

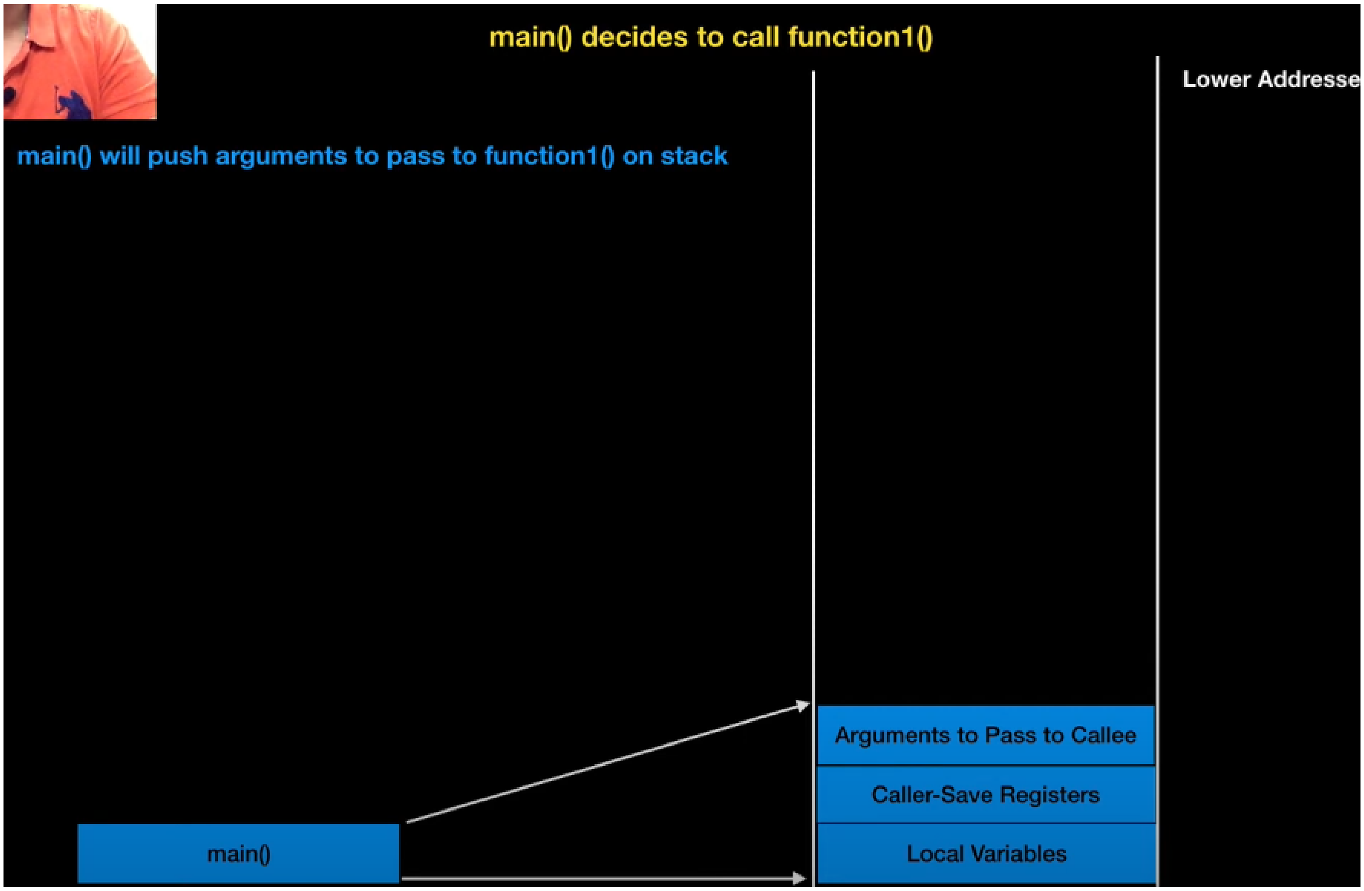
main() reserves space for it's local variables on stack

Lower Addresses

main()

Local Variables

Higher Addresses



main() decides to call function1()

main() will push arguments to pass to function1() on stack

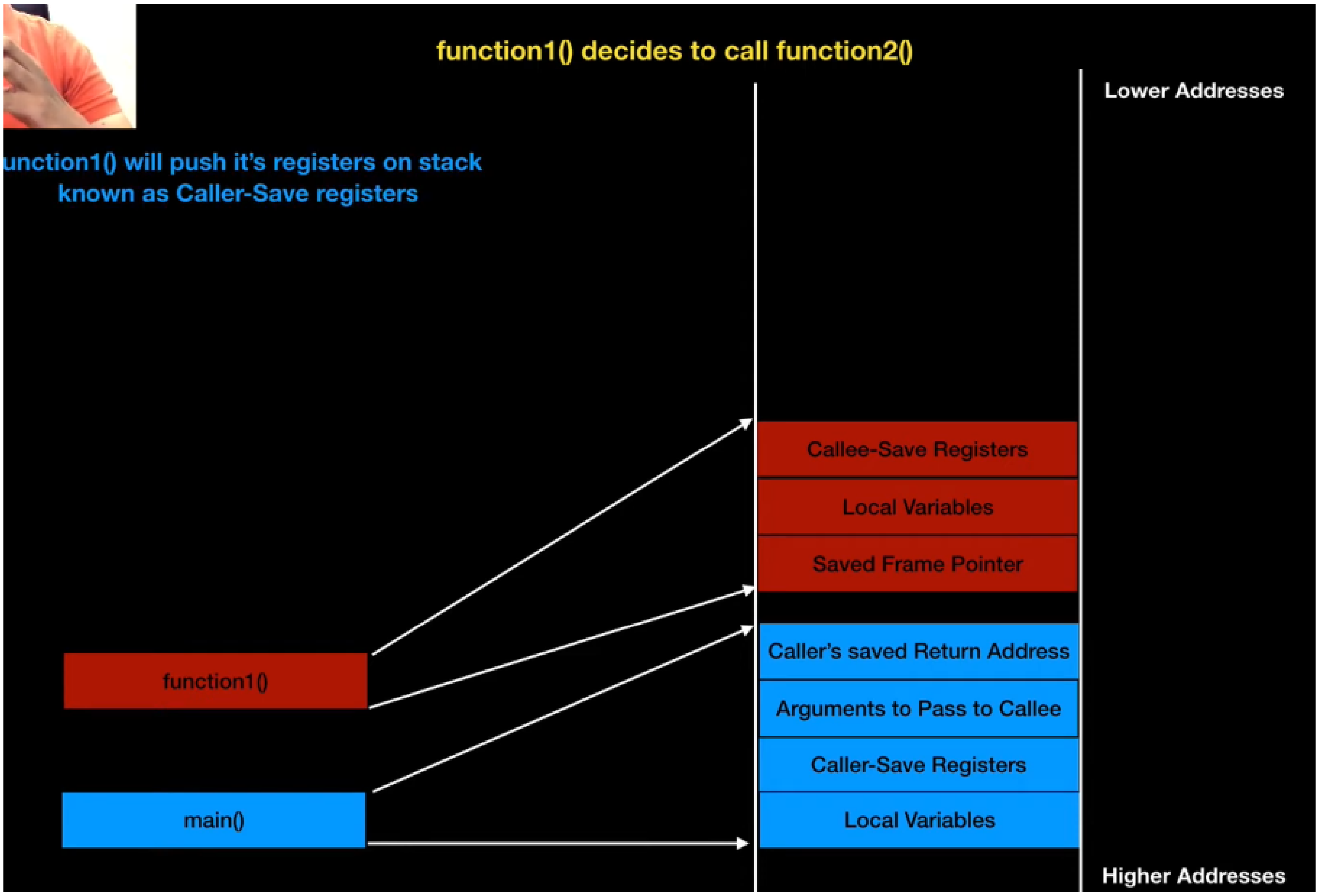
Lower Addresse

Arguments to Pass to Callee

Caller-Save Registers

Local Variables

main()



function1() decides to call function2()

unction1() will push it's registers on stack known as Caller-Save registers

1. At first we are calling the main()
2. Then the main try to call func1()
3. Now the main() next address will be pushed to stack (caller-save addr)
4. And then arguments of the func1() is pushed into the stack
5. Now func1() is executing. At that time func1() encounters func2().
6. Now func1() will push the func1() next address into stack.
7. same as the previous steps.

So this is how the stack frame works.