Assembly & Disassembly Primer

Part 2

If Statement

From a reverse engineering perspective, it is important to understand how branching/conditional statements (like ***if, if-else and if-else if-else***) are translated into assembly language:

```
if (x == 0) {
    x = 5;
}
x = 2;
```

```
cmp     dword ptr [x], 0
jne     end_if
mov     dword ptr [x], 5

end_if:
mov     dword ptr [x], 2
```

The following screenshot shows the ***IF*** statement and the corresponding assembly instructions.

```
if (x == 0)          ──►  cmp  dword ptr [x], 0
                          jne  end_if ──────────┐
{                                               │
   x = 5;            ──►  mov  dword ptr [x], 5 │
}                                               │
                                                ▼
 x = 2;              ──►  end_if:
                          mov  dword ptr [x], 2
```

If-Else Statement

```c
if (x == 0) {
  x = 5;
}
else {
  x = 1;
}
```

```asm
cmp     dword ptr [x], 0
jne     else
mov     dword ptr [x], 5
jmp     end

else:
mov     dword ptr [x], 1

end:
```

If-Elseif-Else Statement

```
if (x == 0) {
  x = 5;
}
else if (x == 1) {
  x = 6;
}
else {
  x = 7;
}
```

```
cmp     dword ptr [ebp-4], 0
jnz     else_if
mov     dword ptr [ebp-4], 5
jmp     short end

else_if:
cmp     dword ptr [ebp-4], 1
jnz     else
mov     dword ptr [ebp-4], 6
jmp     short end

else:
mov     dword ptr [ebp-4], 7

end:
```

LAB 10

LAB 10 - Disassembly Challenge

The following is the disassembled output of a program; Can you translate the following code to its high-level equivalent?. Use the techniques and the concepts that you learned previously to solve this challenge

```
mov     dword ptr [ebp-4], 1
cmp     dword ptr [ebp-4], 0
jnz     loc_40101C
mov     eax, [ebp-4]
xor     eax, 2
mov     [ebp-4], eax
jmp     loc_401025

loc_40101C:
  mov     ecx, [ebp-4]
  xor     ecx, 3
  mov     [ebp-4], ecx

loc_401025:
```

Loops

- Loops execute a block of code until some condition is met.
- The loops jump backward.
- The two most common types of loops are for and while.

```
for (initialization; condition; update_statement ) {
    block of code
}
```

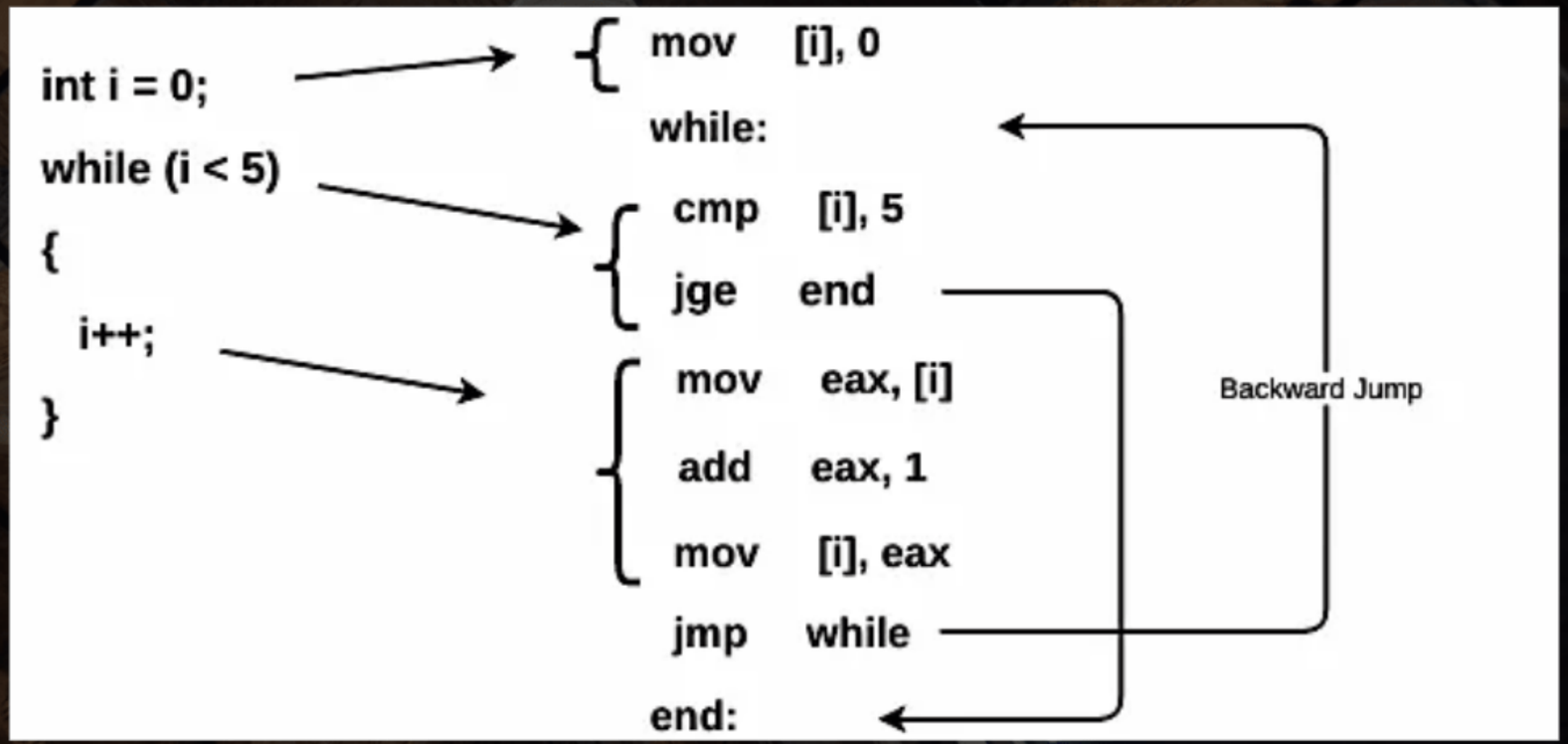
```
initialization
while (condition)
{
  block of code
  update_statement
}
```

```c
int i;
for (i = 0; i < 5; i++)
{
}
```

The above code in for loop can be written using while loop as shown below.

```c
int i = 0;
while (i < 5) {
    i++;
}
```

The following screenshot shows the ***While loop*** and the corresponding assembly instructions.



LAB 11

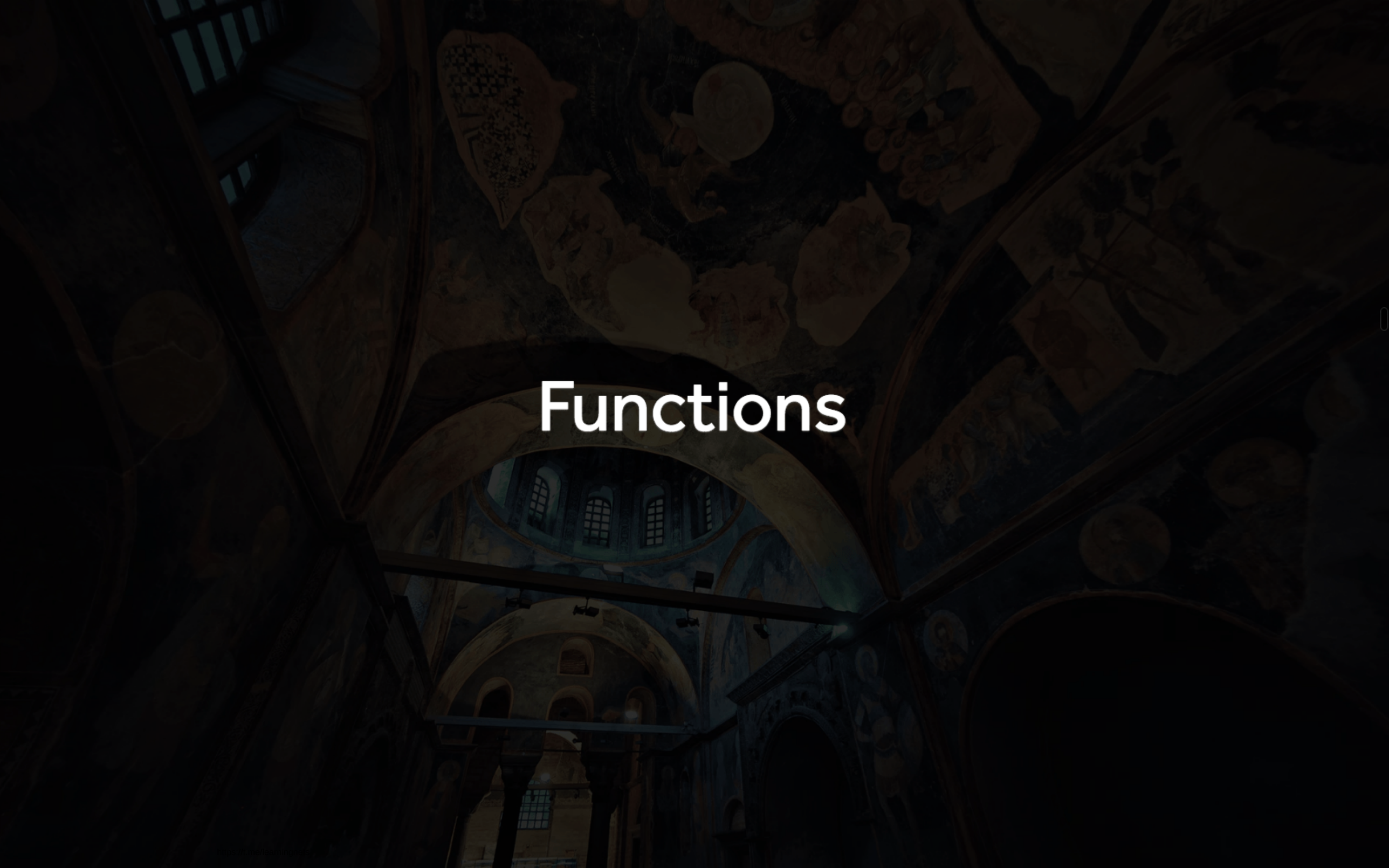
LAB 11 - Disassembly Challenge

The following is the disassembled output of a program; Can you translate the following code to its high-level equivalent?. Use the techniques and the concepts that you learned previously to solve this challenge

```
 mov  dword ptr [ebp-8], 1
 mov  dword ptr [ebp-4], 0

loc_401014:
 cmp     dword ptr [ebp-4], 4
 jge     short loc_40102E
 mov     eax, [ebp-8]
 add     eax, [ebp-4]
 mov     [ebp-8], eax
 mov     ecx, [ebp-4]
 add     ecx, 1
 mov     [ebp-4], ecx
 jmp     short loc_401014

loc_40102E:
```



Functions

- A function is a block of code that performs specific tasks; normally, a program contains many functions.

- When a function is called, the control is transferred to a different memory address. CPU then executes the code at that memory address, and it comes back (control is transferred back) after it finishes running the code.

- The function contains multiple components: a function can take data as input via ***parameters***, it has a ***body*** that contains the code it executes, it contains ***local variables*** that are used to temporarily store values and it can output data.

- The parameters, local variables, and function flow controls are all stored in an important area of the memory called the ***stack***.

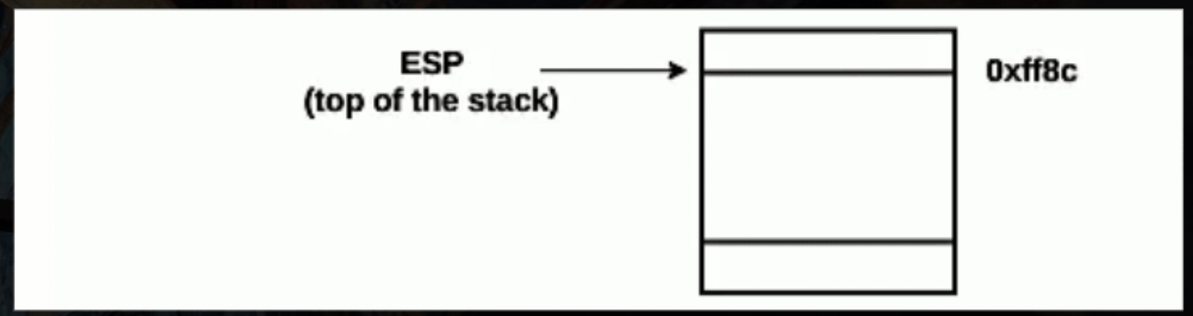
Stack

- Area of the memory that gets allocated by the operating system when the thread is created.
- The stack is organized in a Last-In-First-Out (LIFO) structure. The most recent data that you put in the stack will be the first one to be removed from the stack.
- You put data (called ***pushing***) into the stack using ***PUSH*** instruction, and you remove data (called ***popping***) from the stack using the ***POP*** instruction.
- The ***PUSH*** instruction pushes a 4-byte value onto the stack, and the ***POP*** instruction pops a 4-byte value from the top of the stack.
- The stack grows from higher addresses to lower addresses.
- when a stack is created, the ***ESP*** register (also called the ***stack pointer***) points to the top of the stack (higher address), and as you push data into the stack, the ***ESP*** register decrements by 4 (***ESP-4***) to a lower address.
- When you pop a value, the ***ESP*** increments by 4 (***ESP+4***).

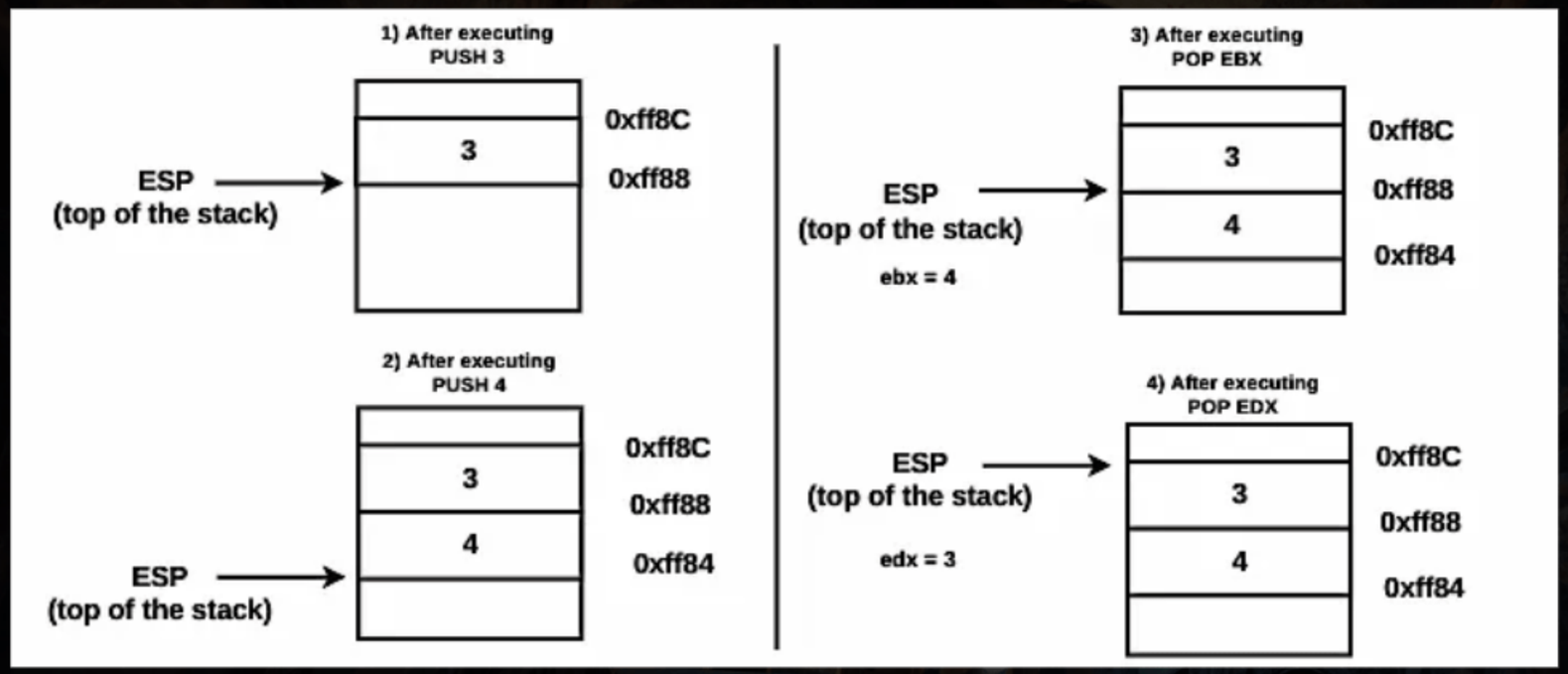
Stack Example

Before executing the following instructions, the ESP register points to the top of the stack (for example, at address 0xff8c), as shown here:

```
push 3
push 4
pop  ebx
pop  edx
```



```
push 3
push 4
pop  ebx
pop  edx
```



Calling & Returning From a Function

The ***CALL*** instruction in the assembly language can be used to call a function. The general form of the ***CALL*** looks as follows:

```
call <some_function>
```

When the ***CALL*** instruction is executed, the control is transferred to ***some_function***, before that, it stores the address of the next instruction by pushing it onto the stack.

The address following the ***CALL*** which is pushed onto the stack is called the ***return address***.

To return from a function, ***RET*** instruction is used. ***RET*** pops the address from the top of the stack; the popped address is placed in the ***EIP*** register, and the control is transferred to the popped address.

In the x86 architecture, the parameters that a function accepts are pushed onto the stack, and the return value is placed in the eax register.

Function Example

In order to understand the function, let's take an example of a simple C program. When the following program is executed, the ***main()*** function calls ***test*** function and passes two integer arguments: ***2*** and ***3***. Inside the ***test*** function, the value of arguments is copied to the local variables ***x*** and ***y***, and the ***test*** returns a value of ***0*** (return value):

```
  int test(int a, int b)
{
    int x, y;
    x = a;
    y = b;
        return 0;
}

int main()
{
    test(2, 3);
    return 0;
}
```

The statements inside the ***main()*** function are translated into assembly instructions as follows:

```
int main()
{
test(2, 3);
return 0;
}
```

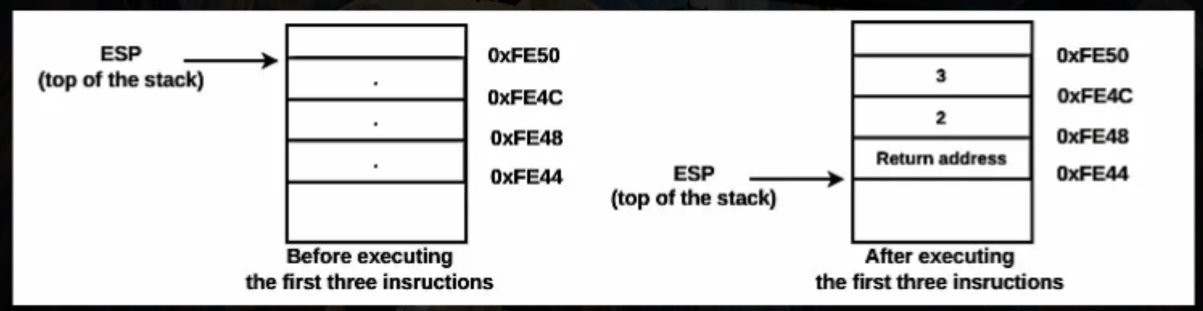
```
push 3 ❶
push 2 ❷
call test ❸
add esp, 8
xor eax, eax
```

The first three instructions, ❶, ❷, and ❸, represent the function call ***test(2,3)***. The arguments (***2*** and ***3***) are pushed onto the stack before the function call in the reverse order (from right to left). The function, ***test()***, is called at ❸; as a result, the address of the next instruction, ***add esp,8***, is pushed onto the stack (this is the return address), and then the control is transferred to the start address of the ***test()*** function

Let's assume that before executing the instructions ❶, ❷, ❸, the esp (stack pointer) was pointing to the top of the stack at the address ***0xFE50***. The following screenshot depicts what happens before and after executing ❶, ❷, ❸

```
int main()
{
test(2, 3);
return 0;
}
```

```
push 3 ❶
push 2 ❷
call test ❸
add esp, 8
xor eax, eax
```



Now let's focus on the ***test()*** function. The following is the assembly translation of the ***test()*** function.

```
int test(int a, int b)
{
int x, y;
x = a;
y = b;
return 0;
}
```

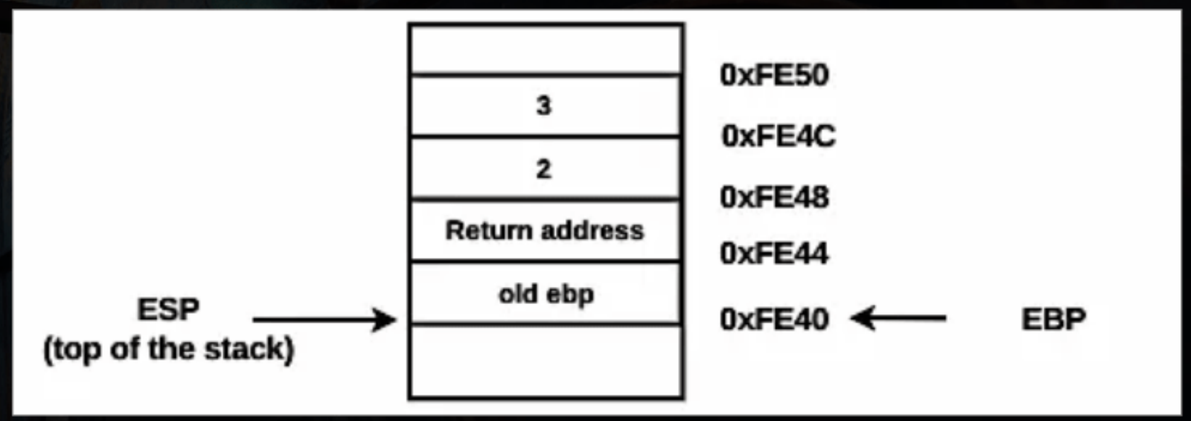
```
push ebp
mov ebp, esp
sub esp, 8
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
xor eax, eax
mov esp, ebp
pop ebp
ret
```

```
push ebp ❹
mov ebp, esp ❺
sub esp, 8
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
xor eax, eax
mov esp, ebp ❻
pop ebp ❼
ret
```

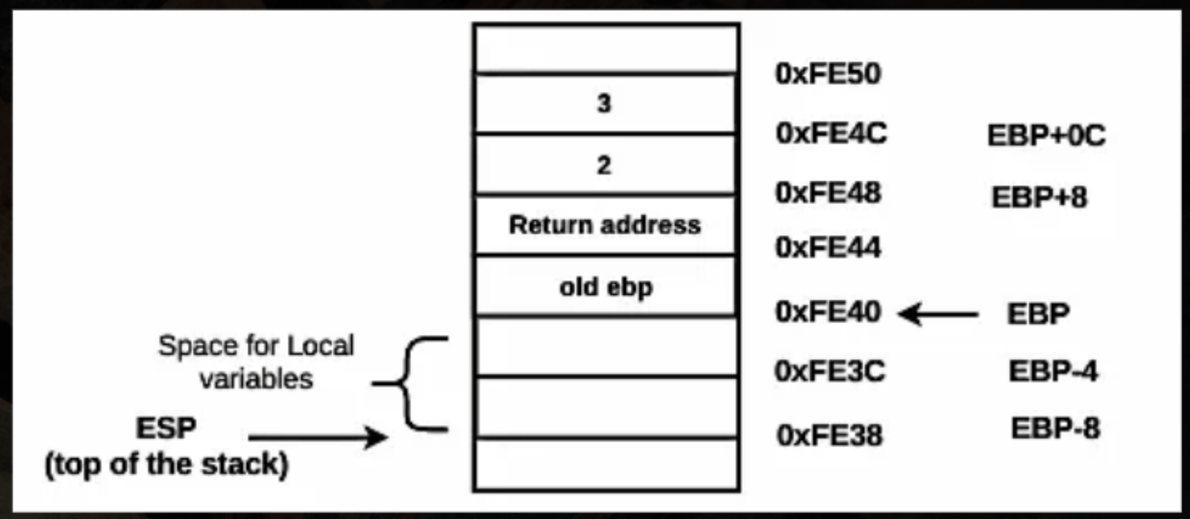
The first instruction ❹ saves the ***EBP*** (also called the ***frame pointer***) on the stack; this is done so that it can be restored when the function returns. In the next instruction, at ❺, the value of ***ESP*** is copied into ***EBP***; as a result, both ***ESP*** and ***EBP*** point at the top of the stack. The ***EBP*** from now on will be kept at a fixed position, & application uses ***EBP*** to reference function arguments & the local variables.

You will normally find ***push ebp*** and ***mov ebp, esp*** at the start of most of the functions; these two instructions are called ***function prologue.***

At ❻ and ❼, the two instructions (***mov esp,ebp*** and ***pop ebp***) perform the reverse operation of ***function prologue***. These instructions are called ***function epilogue.***



```
push ebp
mov ebp, esp
sub esp, 8 ❽
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
xor eax, eax ❾
mov esp, ebp ❻
pop ebp
ret
```



At ❽, ***sub esp,8*** further decrements the ***esp*** register. This is done to allocate space for the local variables (***x*** and ***y***)

Notice that the ebp is still at a fixed position, and function arguments can be accessed at a positive offset from ebp (ebp + some value). The local variables can be accessed at a negative offset from ebp (ebp - some value)

The actual code of the ***test()*** function is between ❽ and ❻. At ❾, ***xor eax,eax*** sets the value of eax to 0. This is the return value (***return 0***)

Now let us translate the code inside the test() function to high level equivalent. In the below code, ***xor eax, eax*** is replaced with ***return 0***

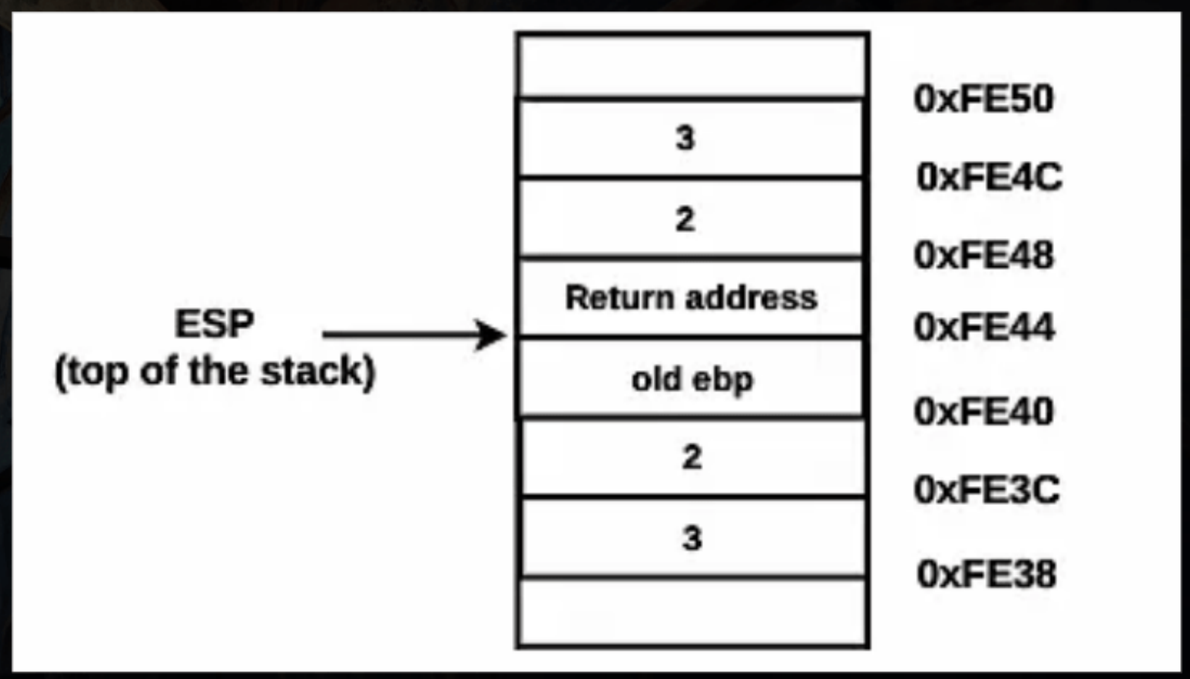
```
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
return 0
```

```
mov eax, [a]
mov [x], eax
mov ecx, [b]
mov [y], ecx
return 0
```

```
x = a
y = b
return 0
```

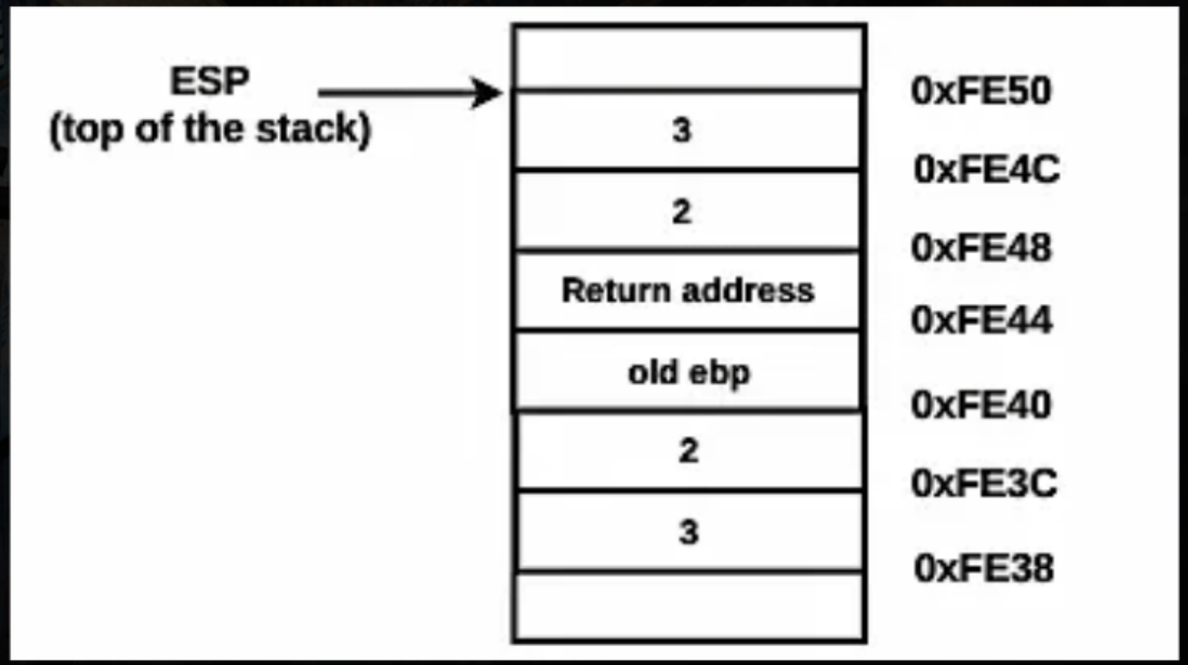
```
push ebp
mov ebp, esp
sub esp, 8
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
xor eax, eax
mov esp, ebp ❻
pop ebp ❼
ret
```

The function epilogue instructions, ***mov esp,ebp*** at ❻ copies the value of ***ebp*** into ***esp***; as a result, ***esp*** will point to the address where ***ebp*** is pointing. The ***pop ebp*** at ❼ restores the old ebp from the stack; after this operation, ***esp*** will be incremented by ***4***. The stack will now look like the one shown below:



```
push ebp
mov ebp, esp
sub esp, 8
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
xor eax, eax
mov esp, ebp
pop ebp
ret ⑩
```

At ⑩, when the ***ret*** instruction is executed, the return address on top of the stack is popped out and placed in the ***eip*** register. Also, the control is transferred to the return address (which is ***add esp,8*** in the main function). As a result of popping the return address, ***esp*** is incremented by ***4***. At this point, the control is returned to the ***main*** function from the ***test*** function. The instruction ***add esp,8*** inside of ***main*** cleans up the stack, and the ***esp*** is returned back to its original position (the address ***0xFE50***, from where we started).

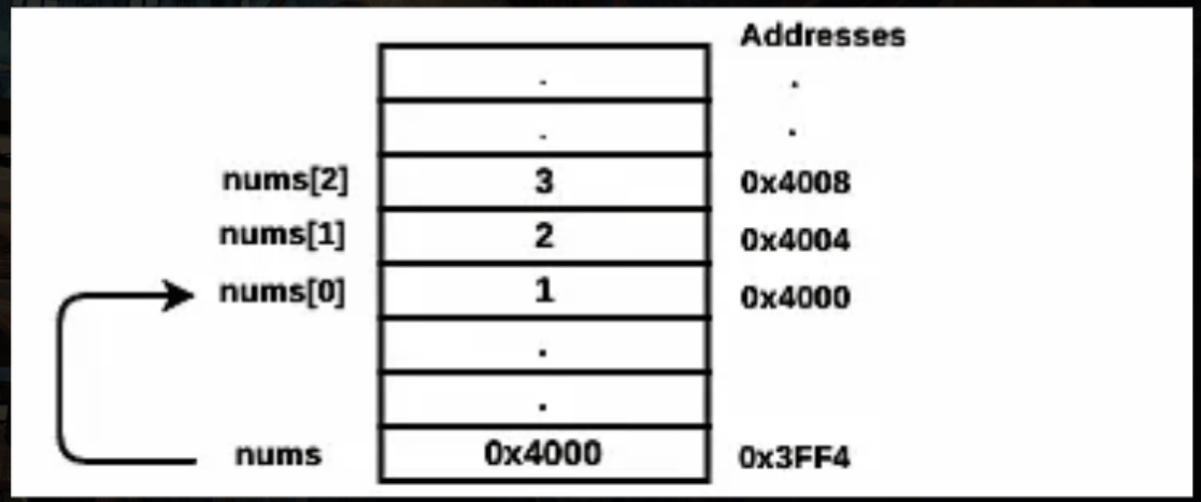


Arrays & Strings

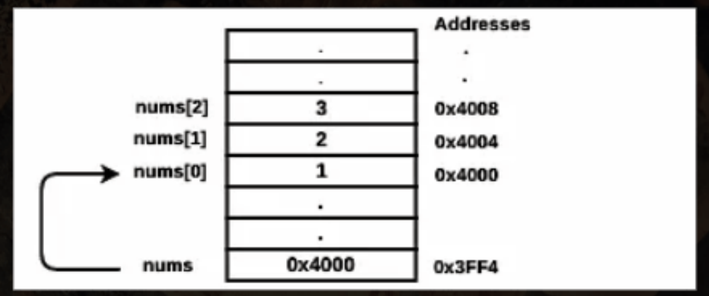
Arrays & Strings

An array is a list consisting of the same data types. The array elements are stored in contiguous locations in the memory, which makes it easy to access array elements

```
int nums[3] = {1, 2, 3}
```



You can access the elements of the integer array (the size of each element is 4 bytes) as shown here:




```
nums[0] = [nums+0*4] = [0x4000+0*4] = [0x4000] = 1
nums[1] = [nums+1*4] = [0x4000+1*4] = [0x4004] = 2
nums[2] = [nums+2*4] = [0x4000+2*4] = [0x4008] = 3

A general form  is represented as follows:

nums[i] = nums+i*4
```

```
The general format for accessing the elements of an array:

[base_address + index * size of element]
```

Structures

A structure groups different types of data together; each element of the structure is called a ***member***. The structure members are accessed using constant offsets.

In the following program, ***simpleStruct*** definition contains three member variables (***a***, ***b***, and ***c***) of different data types. The ***main*** function defines the structure variable (***test_stru***) at ❶, and the address of the structure variable (***&test_stru***) is passed as the first argument at ❷ to the ***update*** function. Inside of the update function, the member variables are assigned values:

```
  struct simpleStruct
{
int a;
short int b;
char c;
};
void update(struct simpleStruct *test_stru_ptr) {
test_stru_ptr->a = 6;
test_stru_ptr->b = 7;
test_stru_ptr->c = 'A';
}
int main()
{
struct simpleStruct test_stru; ❶
update(&test_stru); ❷
return 0;
}
```

The translation of the ***update*** function is shown below

At ❸, the base address of the structure is moved into eax register (remember, ***ebp+8*** represents the first argument; in our case, the first argument contains the base address of the structure). At this stage, ***eax*** contains the base address of the structure. At ❹, the integer value ***6*** is assigned to the first member by adding the offset ***0*** to the base address ([***eax+0***] which is the same as [***eax***]). Because the integer occupies 4 bytes, notice at ❺ the ***short int value 7*** (in ***cx***) is assigned to the second member by adding the offset ***4*** to the base address. Similarly, the value ***41h (A)*** is assigned to the third member by adding ***6*** to the base address at ❻

```
void update(struct simpleStruct *test_stru_ptr) {
test_stru_ptr->a = 6;
test_stru_ptr->b = 7;
test_stru_ptr->c = 'A';
}
```

```
push ebp
mov ebp, esp
mov eax, [ebp+8] ❸
mov dword ptr [eax], 6 ❹
mov ecx, 7
mov [eax+4], cx ❺
mov byte ptr [eax+6], 41h ❻
mov esp,ebp
pop ebp
ret
```

x64 Architecture

This section covers some of the differences in the x64 architecture:

- The first difference is that the 32-bit (4 bytes) general purpose registers ***eax, ebx, ecx, edx, esi, edi, ebp,*** and ***esp*** are extended to ***64 bits*** (***8 bytes***); these registers are named ***rax, rbx, rcx, rdx, rsi, rdi, rbp, and rsp***. The eight new registers are named ***r8, r9, r10, r11, r12, r13, r14,*** and ***r15***

- ***x64 architecture can handle 64-bit (8 bytes) data and all of the addresses and pointers are 64 bits (8 bytes) in size.***

- The x64 CPU has a ***64-bit*** instruction pointer (***rip***) that contains the address of the next instruction to execute, and it also has a 64-bit flags register (***rflags***), but currently, only the lower 32 bits are used (***eflags***)

- The x64 architecture supports ***rip-relative*** addressing. The rip register can now be used to reference memory locations; that is, you can access data at a location which is at some offset from the current instruction pointer.

- In the x64 architecture, the first four parameters are passed in the ***rcx, rdx, r8, and r9*** registers, and if the program contains additional parameters they are stored on the stack.

64-bit Example

```
printf("%d %d %d %d %d", 1, 2, 3, 4, 5);
```

```
sub rsp, 38h ❶
mov dword ptr [rsp+28h], 5 ❼
mov dword ptr [rsp+20h], 4 ❻
mov r9d, 3 ❺
mov r8d, 2 ❹
mov edx, 1 ❸
lea rcx, Format ; "%d %d %d %d %d" ❷
call cs:printf
```

At ❶, ***0x38 (56 bytes)*** of space is allocated on the stack. The ***1st***, ***2nd***, ***3rd***, and ***4th*** parameters are stored in the rcx, rdx, r8 and r9 register (before the call to printf), at ❷, ❸, ❹, ❺. The ***5th*** and the ***6th*** parameters are stored on the stack (in the allocated space), using instructions at ❻ and ❼