

## 04 Assembly Language and Dissassembly Primer

When analysing a malicious program, you only have it's executable, without it's source code. To gain the understanding of the malware's inner workings and to understand the critical aspects of a malicious binary, code analysis needs to be performed

We will cover the following topics from a code analysis (reverse engineering) perspective.

- Computer basics, memory and the CPU
- Data transfer, arithmetic, and bitwise operations
- Branching and Looping
- Functions and Stack
- Arrays, Strings and Structures
- Concepts of the x64 architecture

### 1. Computer Basics

All information is represented in *bits*. A bit, can be either a 0 or a 1. The collection of bits can represent a number, a character, or any other piece of information.

#### Fundamental Data Types

8 bits makes a *byte*. A single byte is represented in two hex digits. Each hexadecimal digit is made up of 4 bits, and is called a *nibble*. A *word* is two bytes in size. A *double word (dword)* is four bytes in size. A *quadword (qword)* is eight bytes in size.

#### 1.1 Memory

- The RAM stores the machine code and data of the computer.
- RAM is an array of bytes with each byte labeled in a unique number, known as it's address.
- The first address starts at 0, and the last is defined by the computer's HW and SW.
- The address and values are represented in hexadecimal.

##### 1.1.1 How Data Resides in Memory

- Data is stored in *little-endian* format
- Low-order byte is stored at the low address, and subsequent bytes are stored in successively higher addresses in the memory

#### 1.2 CPU

- CPU executes instructions (Stored in memory, as a sequence of bytes)

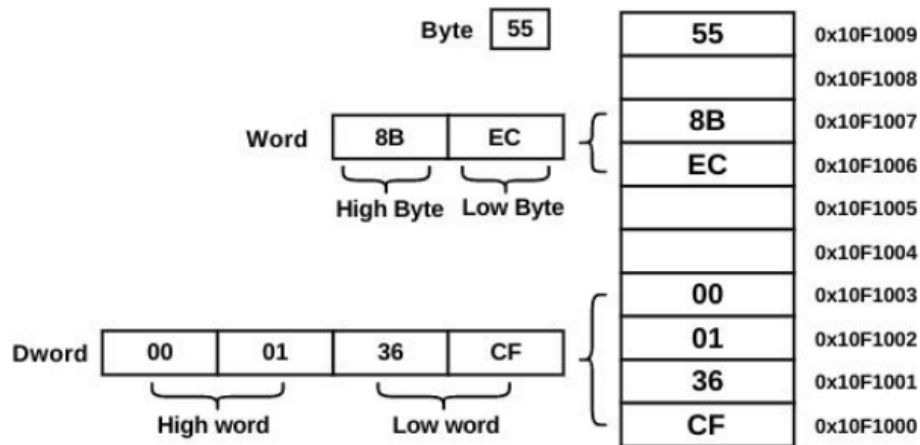


Figure 1: How Data Resides in Memory

- While executing the instructions, the required data is fetched from memory
- CPU contains a register set, which stores values fetched from memory during execution

### 1.2.1 Machine Language

- Each CPU has a set of instructions that it can execute (These make up the CPU's Machine Language)
- A compiler translates a program (like C or C++) into machine language

## 1.3 Program Basics

### Program Compilation

1. Source code is written in a high level language
2. Source code is run through a compiler
3. Object code is passed through a linker, which links the object code with its required libraries

### 1.3.2 Program On Disk

When a program is compiled, it generates a **.exe** file, which, if viewed by *pe-internals* displays the 5 sections generated by the compiler (**.text**, **.rdata**, **.data**, **.rsrc**, **.reloc**)

- In **.data**, we store the data, used by our program
- In **.rdata**, we store read-only data and sometimes, import-export information
- In **.rsrc**, we store resources used by the executable

- In `.text`, we store the machine code (Our program translated to machine code by the compiler)

### 1.3.3 Program in Memory

When the executable is double-clicked a process memory is allocated by the operating system, and the executable is loaded into the allocated memory by the operation system loader.

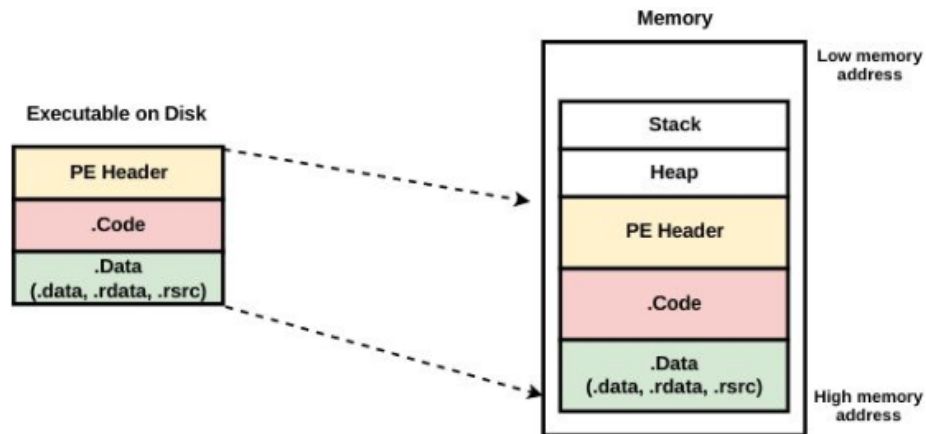


Figure 2: Loading executable from memory

Once the executable that contains the code is loaded into the memory, the CPU fetches the machine code from memory, interprets it, and executes it. While executing the machine instructions, the required data will also be fetched from memory.

### 1.3.4 Program Dissassembly (From Machine code To Assembly code)

A *dissassembler/debugger* is a program that translates machine code into a low-level code called *assembly* which can be read and analysed to determine the workings of a program.

## 2. CPU Registers

CPU can access data in registers much faster than data in Memory, this is why the values stored in memory are stored in these registers to perform operations

### 2.1 General Purpose Registers

- The x86 CPU has 8 general purpose registers:
  - `eax, ebx, ecx, edx, esp, ebp, esi, edi`

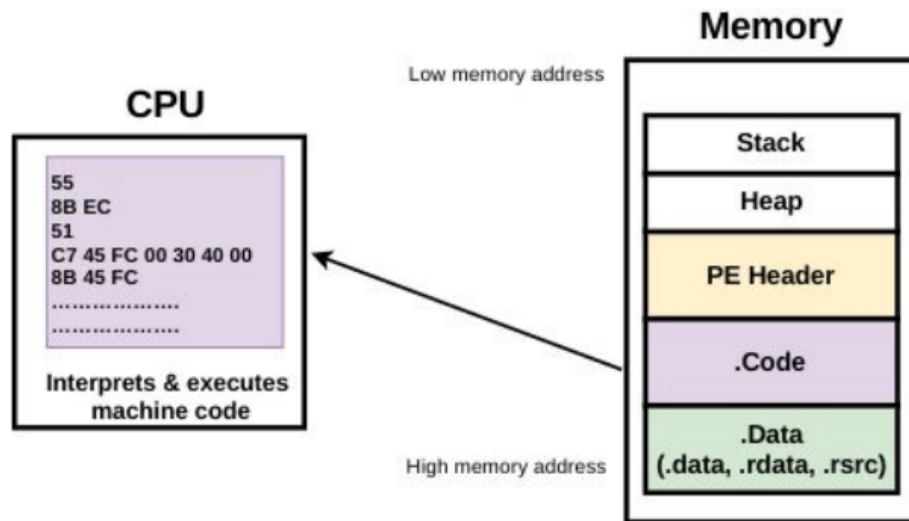


Figure 3: Interaction between the CPU and the memory-loaded program

- These registers are 32 bits (4 bytes) in size.
- A program can access registers as 32-bits, 16-bits or 8-bits
- The lower 16 bits of each of these registers can be accessed as `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si`, `di`
- The lower 8 bits of `eax`, `ebx`, `ecx`, `edx` can be referenced as `al`, `bl`, `cl`, `dl`
- The higher 8 bits can be accessed as `ah`, `bh`, `ch`, `dh`

As an example:

The `eax` register contains the 4-byte value `0xC6A93174`. A program can access the lower 2 bytes (`0x3174`) by accessing register `ax`. It can access the lower byte (`0x74`) by accessing register `al` and the next byte (`0x31`) can be accessed using register `ah`.

## 2.2 Instruction Pointer (EIP)

The CPU has a special register called the `eip`; it contains the address of the next instruction to execute. When the instruction is executed, the `eip` will point to the next instruction in the memory.

## 2.3 EFLAGS Register

The `eflags` register is a 32-bit register, each bit in this register is a flag. There are also additional registers called *segment registers* (`cs`, `ss`, `ds`, `es`, `fs`, `gs`) which keep track of sections in the memory.

### 3. Data Transfer Instructions

The `mov` instruction is one of the basic instructions in the assembly language. It moves data from one location to another.

```
mov dst,src
```

There are also different variations of the `mov` instruction

#### 3.1 Moving a constant into register

A variation of the `mov` command. Moves a constant or a immediate value into a register.

```
mov eax,10 ; moves 10 into EAX register, same as eax=10
```

#### 3.2 Moving Values From Register to Register

Done by placing the names of the registers in the operands

```
mov eax,ebx ; moves content of ebx into eax
```

#### 3.3 Moving values from Memory to Registers

1. An integer is 4 bytes in length, so the integer 100 is stored as a sequence of 4 bytes (00 00 00 64) in the memory.
2. The sequence of 4 bytes is stored in *little-endian* format
3. The integer 100 is stored at some memory address.

To move a value from the memory into a register in the assembly language, you must use the address of the value. The dest (`eax`) will automatically determine how many bytes to move.

```
mov eax,[0x403000] ; eax will now contain 00 00 00 64 (i.e 100)
```

```
mov eax, [0x403000]
```

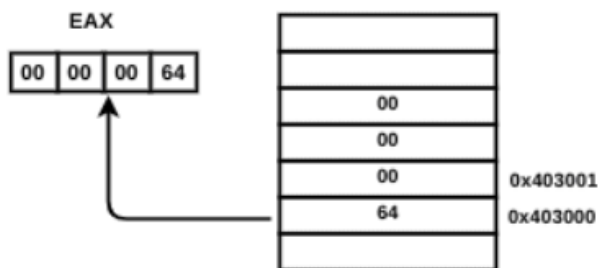


Figure 4: Moving value of register to register

The square brackets may contain a *register*, a *constant added to a register*, or a *register added to a register*.

Another common instruction is the `lea` instruction. This stands for *Load Effective Address*. This instruction will load the address instead of the value

```
lea ebx,[0x403000] ; loads the address 0x403000 into ebx
lea eax,[ebx] ; if ebx = 0x403000, then eax will also
                contain 0x403000
```

## Moving Values From Registers To Memory

Swapping the operands, you can move a value from a register to memory

```
mov [0x403000],eax ; moves 4 byte value in eax to memory
                    location starting at 0x403000
```

```
mov [ebx],eax ; moves 4 byte value in eax to the memory
               address specified by ebx
```

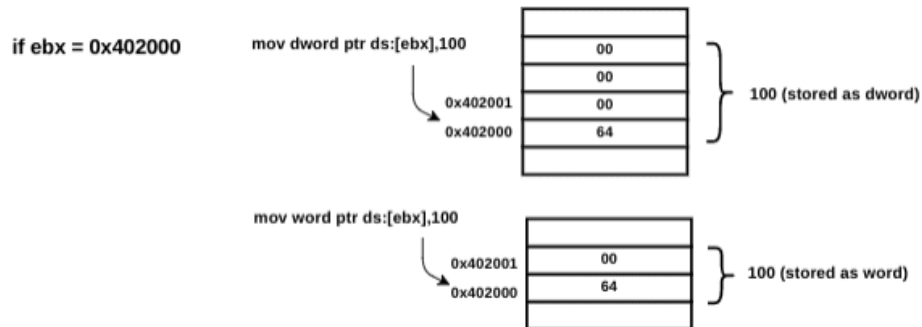
`dword ptr` just specifies that a `dword` value (4 bytes) is moved into the memory location.

```
mov dword ptr [402000],13498h ; moves dword value
                                0x13496 into the address 0x402000
```

```
mov dword ptr [ebx],100 ; moves dword value 100 into the
                          address specified by ebx
```

```
mov word ptr [ebx],100 ; moves a word 100 into the
                        address specified by ebx
```

In the preceding case, if `ebx` contained the memory address `0x402000`, then the second instruction copies 100 as `00 00 00 64` (4 bytes) and the third instruction copies 100 as `00 64` (2 bytes) into the memory location starting at `0x402000`, as shown below:



## 4. Arithmetic Operations

You can perform addition, subtraction, multiplication and division in assembly language. Addition adds `src + dest` and stores it in `dest`. Same with Subtraction. These instructions set or clear flags in the `eflags` register. These flags can be

used in conditional statements. The `sub` instruction sets the zero flag (`zf`) if the result is zero, and the carry flag (`cf`) if the destination value is less than the source.

```
add eax,42 ; same as eax = eax+42

add eax, ebx ; same as eax = eax+ebx

add [ebx],42 ; adds 42 to the value in the address
specified by ebx

sub eax, 64h ; subtracts hex value 0x64 from eax, same
as eax = eax-0x64
```

There are special increments (`inc`) and decrements (`dec`) instructions. These add 1 or subtract 1 from either a register or a memory location.

```
inc eax ; same as eax = eax + 1

dec eax ; same as eax = eax - 1
```

Multiplication is done with the (`mul`) instruction. This instruction takes only one operand; which is multiplied by the content of the `al`, `ax`, `eax` register. The result of the multiplication is stored in either `ax`, `dx`, `edx`, `eax` registers. If the operand of the `mul` instruction is *8 bits (1 byte)*, then it is multiplied by the 8-bit `al` register, and the product is stored in the `ax` register. If the operand is *16 bits (2 bytes)*, then it is multiplied with the `ax` register, and the product is stored in the `dx` and `ax` register. If the operand is *32 bit (4 bytes)*, then it is multiplied with the `eax` register, and the product is stored in the `edx` and `eax` register.

```
mul ebx ; ebx is multiplied with eax and the result is
stored in EDX and EAX

mul bx ; bx is multiplied with ax and the result is
stored in DX and AX
```

Division is performed using the `div` instruction. The `div` takes only one operand, which is either a register or a memory reference. To perform division, you place the dividend (number to divide) in the `edx` and `eax` registers, with `edx` holding the most significant *dword*. After the `div` instruction is executed, the quotient is stored in `eax`, and the remainder is stored in the `edx` register:

```
div ebx ; divides the value in EDX:EAX by EBX
```

## 5. Bitwise operations

Assembly instructions that operate on the bits. The bits are numbered starting from the far right (rightmost (least significant bit) bit has a bit position of 0). The leftmost bit, is called the most significant bit.

**not instruction:**

Takes only one operand, serves as **src** and **dst** and inverts all of the bits.

```
not eax Converts 11100110 to 00011001 and stores it in the same
register
```

**and instruction:**

```
and bl,cl
bl: 0000 0101
cl: 0000 0110
```

After the operation:

```
bl: 0000 0100
```

**or instruction:**

```
or bl,cl
bl: 0000 0101
cl: 0000 0110
```

After the operation:

```
bl: 0000 0111
```

**xor instruction:**

```
xor bl,cl
bl: 0000 0101
cl: 0000 0110
```

After the operation:

```
bl: 0000 0011
```

**shr (Shift right)**

Takes two operands, the destination and the count The destination can be either a register or a memory reference

```
shr bl,4
bl: 0000 0101
```

After the operation:

```
bl: 0000 0000
```



### **shl (Shift left)**

Takes two operands, the destination and the count. The destination can be either a register or a memory reference.

```
shl bl,3
bl: 0000 0110
```

After the operation:

```
bl: 0011 0000
```

### **rol and ror (rotate left and rotate right)**

Are similar to shift, but instead of removing the shifter bits, they are rotated to the other end.

```
rol al,2
al: 0100 0100
```

After the operation:

```
al: 0001 0001
```

## **6. Branching and conditionals**

Branching instructions transfer the control of execution to a different memory address. To perform branching, jump instructions are typically used. There are two types: *conditional* and *unconditional*.

### **6.1 Unconditional jumps**

```
jmp <jump address>
```

The jump is always taken.

### **6.2 Conditional jumps**

Control is transferred to a memory address based on some condition. To use conditional jumps, we need instructions that can alter the flags (*set* or *clear*). These instructions can be performing a *arithmetic operation* or a *bitwise operation*.

#### **cmp instruction**

Subtracts the second operand from the first one without altering the dest (first operand).

```
cmp eax,5 ; if eax has a value of 5
```

Would set the *zero flag* (zf=1) because the result is 0. used with conditional jump instruction for decision-making.

### test instruction

Alters flags, without storing the result in `dst`. Performs a bitwise **and** and alters *zero flag* (`zf=1`) because when you and 0 with 0, you get 0. used with conditional `jump` instruction for decision-making

### Variations of conditional jumps

These conditions are evaluated based on the bits in the `eflags` register.

| Instruction      | Description              | Aliases               | Flags                      |
|------------------|--------------------------|-----------------------|----------------------------|
| <code>jz</code>  | jump if zero             | <code>je</code>       | <code>zf=1</code>          |
| <code>jnz</code> | jump if not zero         | <code>jne</code>      | <code>zf=0</code>          |
| <code>jl</code>  | jump if less             | <code>jnge</code>     | <code>sf=1</code>          |
| <code>jle</code> | jump if less or equal    | <code>jng</code>      | <code>zf=1 or sf=1</code>  |
| <code>jg</code>  | jump if greater          | <code>jnl</code>      | <code>zf=0 and sf=0</code> |
| <code>jge</code> | jump if greater or equal | <code>jnl</code>      | <code>sf=0</code>          |
| <code>jc</code>  | jump if carry            | <code>jb, jnae</code> | <code>cf=1</code>          |
| <code>jnc</code> | jump if not carry        | <code>jnb, jae</code> | .                          |

Figure 5: Conditional jumps

## 6.3 If statements

In order to reverse-engineer a program, we have to understand how the `if`, `if-else` and `if-else if-else` statements are translated into assembly.

In the following example, translated to assembly, the *jump* will be taken when the condition **is not met**.

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

This code, will be translated in assembly into:

```
cmp dword ptr [x], 0  
jne end_if  
mov dword ptr [x], 5  
end_if:  
mov dword ptr [x], 2
```

## 6.4 If-else statement

There are two conditions in this case.

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

If  $x == 0$  the code inside the if statement will be executed and then, the program will jump the else statement. If  $x \neq 0$  the code inside the if will be jumped and the code inside the else will be executed.

in assembly:

```
cmp dword ptr [x], 0  
jne else  
mov dword ptr [x], 5  
jmp end  
else:  
mov dword ptr [x], 1  
end:
```

## 6.5 If-Elseif-Else Statements

In the next example, there are two conditional statements, if `x != 0` it will jump to the second conditional statement, if this one is not met, it will jump this one, and go to the else.

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

wich translated into assambly:

```
    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:
```

## 7. Loops

To create a loop, the `goto` operation must jump backward Two of the most common loops are `for` loops, and `while` loops.

For example:

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

or

```

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

```

can be translated into assembly as follows:

```

mov [i], 0
while_start:
cmp [i], 5
jge end
mov eax, [i]
add eax, 1
mov [i], eax
jmp while_start
end:

```

## 8. Functions

When a function is called, the control is transferred to a different memory address. The code in that memory address is run, and once finished, goes back to the original memory address. The function's parameters, local variables, and function flow controls are stored in an important area of the memory called the *stack*.

### 8.1 Stack

The *stack* is an area of memory that gets allocated by the operating system when the thread is created. It is organized in a *Last-in-First-out (LIFO)* structure. To use the stack, there are two instructions:

- **push** which pushes a *4-byte* value onto the stack
  - **push source ; pushes source on top of the stack**
- **pop** which pops a *4-byte* value from the top of the stack
  - **pop destination ; copies value from the top of the stack to the destination**

The stack grows from higher address to lower address. When 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**).

### 8.2 Calling Function

Used to call a function.

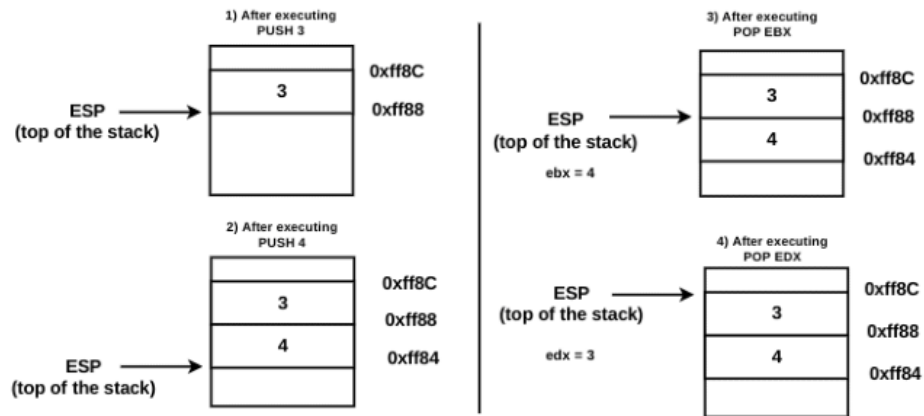


Figure 6: Push and Pop from stack

```
call <some_function>
```

The control is transferred to `some_function`, but before that, it stores the address (*return address*) of the next instruction (the one following `call <some_function>`) by pushing it to the stack. Once `<some_function>` has finished execution, the return address is popped from the stack, and the execution continues from the popped address.

### 8.3 Returning From Function

`ret` instruction

This instruction 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.

### 8.4 Function Parameters and Return Values

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

Given this program:

```
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 like so:

```

    push 3
    push 2
    call test
    add esp, 8 ; after test is executed, the control is
    returned here
    xor eax, eax

```

The first 3 instructions, represent the function call `test(2, 3)`. Note the parameters are pushed to the stack in reverse order. Then, the function `test()` is called

The assembly translation for `test()` function:

```

    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

```

The first instruction `push ebp` saves the `ebp` (also called the frame pointer) on the stack. This way, it can be restored when the function returns. In the next instruction, (`mov ebp, esp`) 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, and the application will use `ebp` to reference function arguments and the local variables.

In most functions, you will see instructions

```

    push ebp
    mov ebp, esp

```

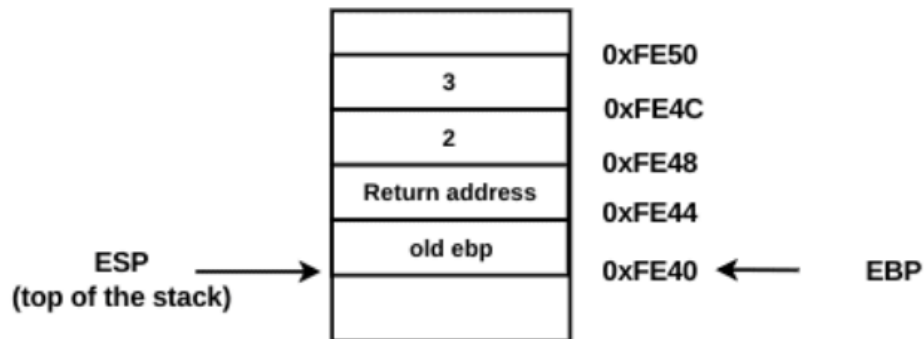


Figure 7: State of EBP and ESP

at the start, these two instructions are called *function prologue*. They setup the function environment.

The following two functions, perform the reverse operation. They are called *function epilogue*. They restore the environment to before the function was executed.

```
mov esp, ebp
```

```
pop ebp
```

The instruction `sub esp, 8` further decrements the `esp` register. This is done to allocate space for the local variables (`x` and `y`). Now, the stack looks like this:

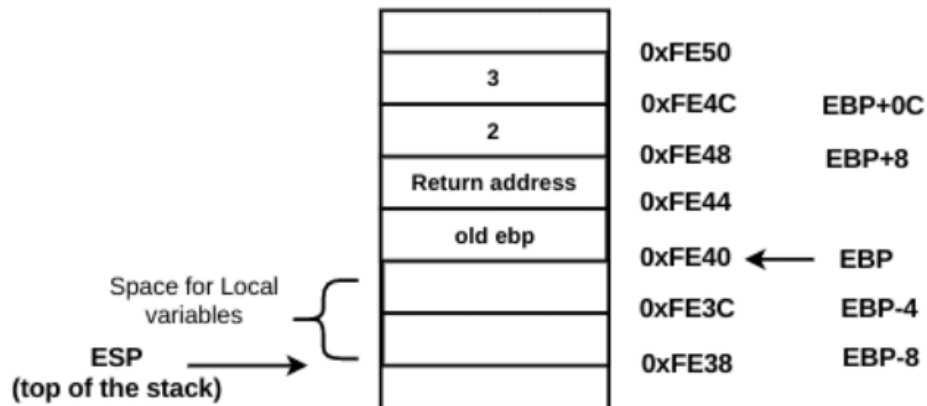


Figure 8: State of EBP and ESP after the function setup

`ebp` is at a fixed position, and **function arguments can be accessed at a positive offset** from `ebp` (`ebp + some value`).

**Local variables can be accessed at a negative offset** from `ebp` (`ebp -`



some value).

The actual code inside the function is:

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

The instruction `xor eax, eax` sets the return register `eax` (always the return register) to 0.

The function epilogue restore the function environment.

```
mov esp, ebp
```

copies the value of `ebp` into `esp`; as a result, `esp` will be pointing to the same address as `ebp`. `### pop ebp`

Restores the old `ebp` from the stack; After this, `esp` will be incremented by 4. To the state the program was before executing the function.

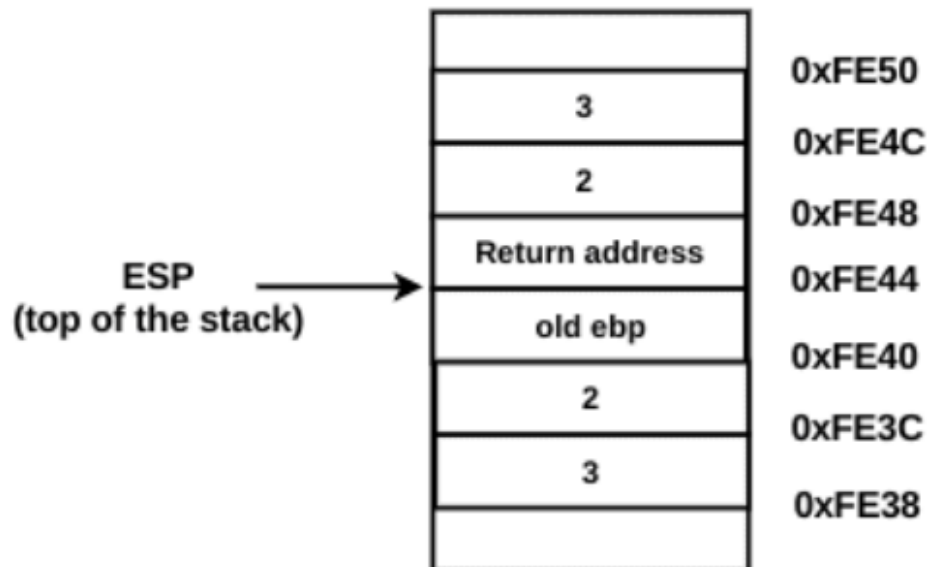


Figure 9: State of ESP after the epilogue

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 (`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 the `main` cleans up the stack, and the `esp` is returned to it's original position.

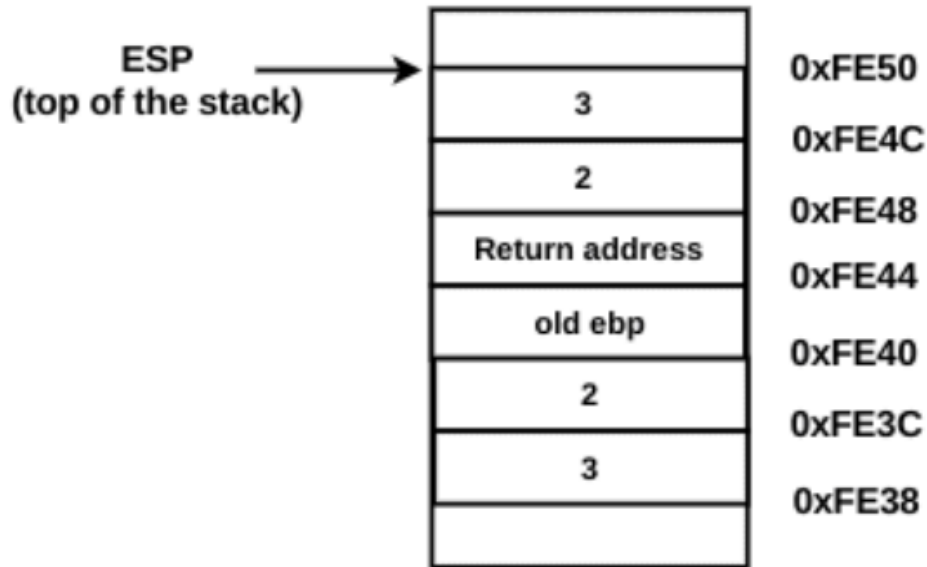


Figure 10: State of the ESP when the function call has ended

## Function call conventions

- `cdecl`
  - Used in the example above
  - The caller function passes parameters to the callee function right to left
  - The caller function is responsible of cleaning the stack after the callee function returns control to the caller function
- `stdcall`
  - Used by Windows for the functions (API) exported by the DLL files
  - The caller function passes parameters to the callee function right to left
  - The callee function is responsible of cleaning the stack
- `fastcall`
  - `x64bit` programs will use this convention

## 9. Arrays and Strings

A list consisting of the same data types. In assembly language, the address of any element in the array is computed using three things:

- Base address of the array

- Index of the element
- Size of each element in the array

To access the value of each array element:

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

## 9.3 Strings

A array of characters. Each element occupies 1 byte of memory. The string name is a pointer variable that points to the first character in the string.

To access the value of each string element:

```
str[i] = [str+i]
```

### 9.3.1 String Instructions

Instructions provided by the x86 architecture that operate on strings. These instructions step through the string and are suffixed with **b**, **w** and **d** which indicate the size of data to operate on (1, 2 or 4 bytes).

The string instructions make use of the registers **eax**, **esi** and **edi**.

- The register **eax** (or its sub-registers **ax** and **al**) are used to **hold values**
- The register **esi** acts as the **source address register** (holds the address of the source string)
- The register **edi** acts as the **destination address register** (holds the address of the destination string)

After performing a string operation, the **esi** and **edi** registers are either automatically incremented or decremented (Think of them as source and destination index registers). The direction flag (DF) in the **eflags** register determines whether **esi** and **edi** should be incremented or decremented. The **cld** instruction clears the direction flag (**df=0**); If **df=0**, then the index registers (**esi** and **edi**) are incremented. If **df=1**, then the index registers (**esi** and **edi**) are decremented.

### 9.3.2 Moving From Memory To Memory (movsx)

The **movsx** instructions are used to move a sequence of bytes from one memory location to another.

- The **movsb** instruction moves 1 byte from the address specified by the **esi** register to the address specified by the **edi** register.
- The **movsw** instruction moves 2 bytes from the address specified by the **esi** register to the address specified by the **edi** register.
- The **movsd** instruction moves 4 bytes from the address specified by the **esi** register to the address specified by the **edi** register.

After the value is moved, the **esi** and **edi** registers are incremented by 1, 2 or 4 bytes.

### 9.3.3 Repeat Instructions (rep)

To copy multi-byte content, the **rep** instruction is used, along with the string instruction.

The **rep** instruction depends on the **ecx** register, and it repeats the instruction as many times as indicated by the **ecx** register. After the **rep** instruction is executed, the value of the **ecx** is decremented.

In this example, the string “Good” is copied from **src** to **dst** along with the *null terminator*

```
lea esi, [src] ; "Good". 0x0
lea edi, [dst]
mov ecx, 5
rep movsb
```

This instruction has multiple forms, which allow early terminations based in the condition that occurs during the execution of the loop.

| Instruction         | Condition                                 |
|---------------------|---|
| <b>rep</b>          | Repeats until <b>ecx=0</b>                |
| <b>repe, repz</b>   | Repeats until <b>ecx=0</b> or <b>ZF=0</b> |
| <b>repne, repnz</b> | Repeat until <b>ecx=0</b> or <b>ZF=1</b>  |

Figure 11: Repeat Instruction Forms

### 9.3.4 Storing Value From Register to Memory (stosx)

- The **stosb** instruction moves 1 byte from the CPU's **al** register to the memory address specified by **edi** (Destination Index Register)
- The **stosw** instruction moves 2 byte from the CPU's **ax** register to the memory address specified by **edi** (Destination Index Register)
- The **stosd** instruction moves 4 byte from the CPU's **eax** register to the memory address specified by **edi** (Destination Index Register)

This example fills the destination buffer with 5 double words (**dword**), all of them equal to 0.

```
mov eax, 0 ; 4 bytes, because it uses the eax register
lea edi, [dest]
mov ecx, 5
rep stosd
```

### 9.3.5 Loading From Memory to Register (lodsx)

- The **lodsb** instruction moves 1 byte from memory address specified by **esi** (The source index register) to the **al** register.
- The **lodsw** instruction moves 2 byte from memory address specified by **esi** (The source index register) to the **ax** register.
- The **lodsd** instruction moves 4 byte from memory address specified by **esi** (The source index register) to the **eax** register.

### 9.3.6 Scanning Memory (scasx)

The **scasb** instruction is used to scan for the presence or absence of a byte value in a sequence of bytes. The byte to search for is placed in the **al** register, and the memory address is placed in the **edi** register.

### 9.3.7 Comparing Values in Memory (cmpsx)

The **cmpsb** instruction is used to compare a byte in the memory address specified by **esi** with a byte in the memory address specified by **edi**, to determine if they contain the same data.

## 10. Structures

A structure groups different types of data together; Each element of a structure is called a member. The structure members are accessed using constant offsets. Each member of the structure has its own offset and is accessed by adding the *constant offset* to the *base address*.

### 11. x64 Architecture

Is an extension of x86 architecture.

Key differences:

- 32-bit general purpose registers are extended to 64-bits
  - **eax, ebx, ecx, edx, esi, edi, ebp, esp** are extended to 64-bits, named **rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp**
  - The 8 new registers are named: **r8, r9, r10, r11, r12, r13, r14, r15**

- A program can access register as 64-bit (**RAX**, **RBX**, ...), 32-bit (**eax**, **ebx**, ...), 16-bit (**ax**, **bx**, ...), 8-bit (**al**, **bl**, ...)
- x64 architecture can handle 64-bit data and all of the addresses and pointers are 64 bits in size.
- x64 CPU has a 64-bit instruction pointer (**rip**) but currently, only the lower half are being used.
- x64 supports **rip-relative** addressing
- Uses fastcall convention

### 11.1 Analyzing 32-bit Executable on Windows

Windows OS can run 32-bit executable, because it developed a subsystem called WOW64 (Windows 32-bit on Windows 64-bit)

Key points when analyzing 32-bit binaries on 64-bit OS

- If you find malware accessing the **system32** directory, it is really accessing the **system64** directory. (The OS auto-redirects to windows64)
- If malware is writing a file in **\windows\system32** directory, you have to check **\windows\system64** directory
- Access to **%windir%\regedit.exe** is redirected to **%windir%\SysWOW64\regedit.exe**