4
Assembly Language and Disassembly Primer

Static analysis and dynamic analysis are great techniques to understand the basic functionality of malware, but these techniques do not provide all the required information regarding the malware's functionality. Malware authors write their malicious code in a high-level language, such as C or C++, which is compiled to an executable using a compiler. During your investigation, you will only have the malicious executable, without its source code. To gain a deeper understanding of a malware's inner workings and to understand the critical aspects of a malicious binary, code analysis needs to be performed.

This chapter will cover the concepts and skills required to perform code analysis. For a better understanding of the subject, this chapter will make use of relevant concepts from both C programming and assembly language programming. To understand the concepts covered in this chapter, you are expected to have a basic programming knowledge (preferably C programming). If you are not familiar with basic programming concepts, start with an introductory programming book (you can refer to the additional resources provided at the end of this chapter) and return to this chapter afterward.

The following topics will be covered from a code analysis (reverse engineering) perspective:

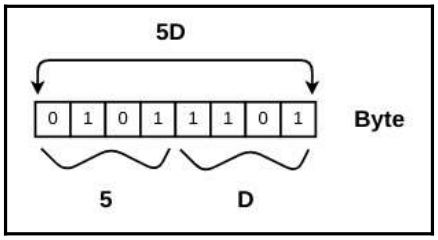
- Computer basics, memory, and the CPU
- Data transfer, arithmetic, and bitwise operations
- Branching and looping
- Functions and stack
- Arrays, strings, and structures
- Concepts of the x64 architecture

1. Computer Basics

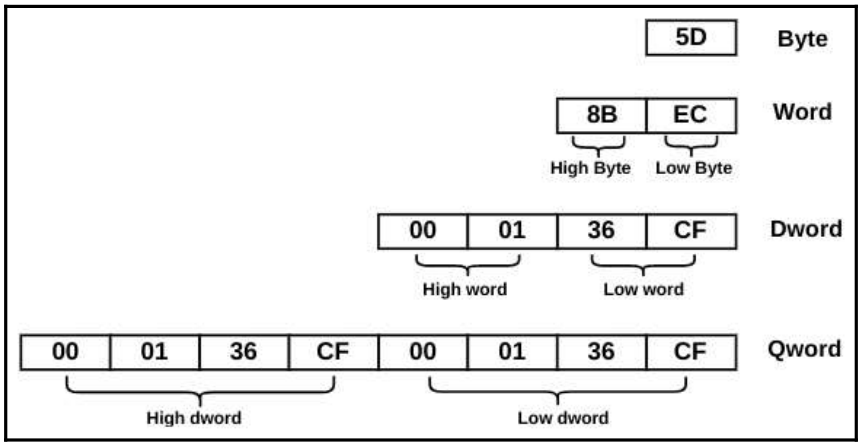
A computer is a machine that processes information. All of the information in the computer is represented in *bits*. A bit is an individual unit that can take either of the two values `0` or `1`. The collection of bits can represent a number, a character, or any other piece of information.

Fundamental data types:

A group of `8` bits makes a *byte*. A single byte is represented as two hexadecimal digits, and each hexadecimal digit is `4` bits in size and called a *nibble*. For example, the binary number `01011101` translates to `5D` in hexadecimal. The digit `5` (`0101`) and digit `D` (`1101`) are the `nibbles`:



Apart from bytes, there are other data types, such as a `word`, which is `2` bytes (`16 bits`) in size, a `double word (dword)` is `4` bytes (`32 bits`), and a `quadword (qword)` is `8` bytes (`64 bits`) in size:



Data Interpretation:

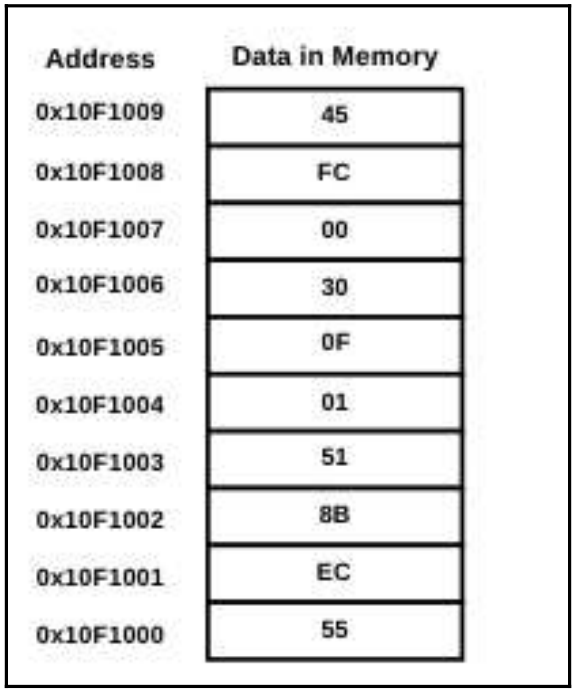
A byte, or sequence of bytes, can be interpreted differently. For example, `5D` can represent the binary number `01011101`, or the decimal number `93`, or the character `]`. The byte `5D` can also represent a machine instruction, `pop ebp`.

Similarly, the sequence of two bytes `8B EC` (`word`) can represent `short int 35820` or a machine instruction, `mov ebp,esp`.

The double word (`dword`) value `0x010F1000` can be interpreted as an integer value `17764352`, or a memory address. It's all a matter of interpretation, and what a byte or sequence of bytes means depends on how it is used.

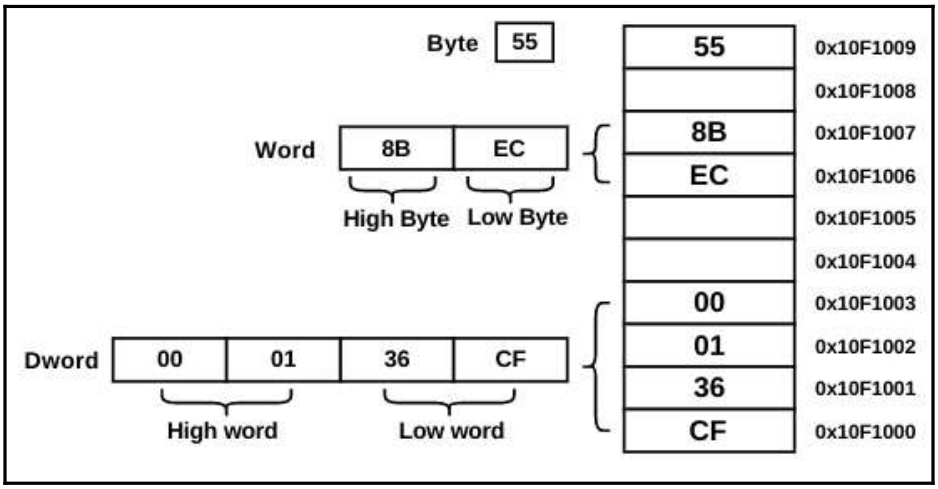
1.1 Memory

The *main memory (RAM)* stores the code (machine code) and data for the computer. A computer's main memory is an array of bytes (sequence of bytes in hex format), with each byte labeled with a unique number, known as its *address*. The first address starts at `0`, and the last address depends on the hardware and software in use. The addresses and values are represented in hexadecimal:



1.1.1 How Data Resides In Memory

In memory, the data is stored in the *little-endian* format; that is, a low-order byte is stored at the lower address, and subsequent bytes are stored in successively higher addresses in the memory:



1.2 CPU

The *Central Processing Unit (CPU)* executes instructions (also called *machine instructions*). The instructions that the CPU executes are stored in the memory as a sequence of bytes. While executing the instructions, the required data (which is also stored as a sequence of bytes) is fetched from memory.

The CPU itself contains a small collection of memory within its chip, called the *register set*. The registers are used to store values fetched from memory during execution.

1.2.1 Machine Language

Each CPU has a set of instructions that it can execute. The instructions that the CPU executes make up the CPU's machine language. These machine instructions are stored in the memory as a sequence of bytes that is fetched, interpreted, and executed by the CPU.

A *compiler* is a program that translates programs written in a programming language (like C or C++) into the machine language.

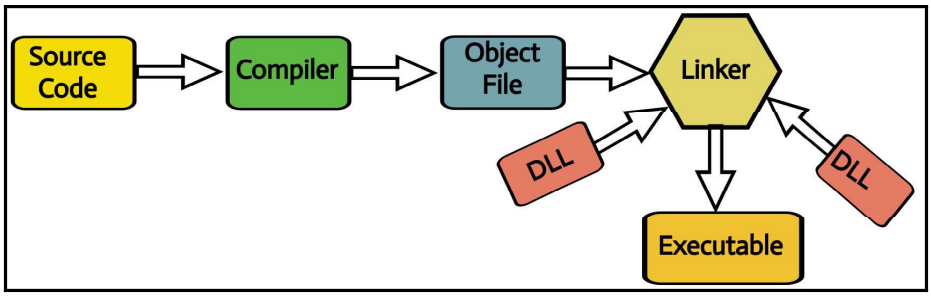
1.3 Program Basics

In this section, you will learn what happens during the compilation process and program execution, and how various computer components interact with each other while the program executes.

1.3.1 Program Compilation

The following list outlines the executable compilation process:

1. The source code is written in a high-level language, such as C or C++.
2. The source code of the program is run through the compiler. The compiler then translates the statements written in a high-level language into an intermediate form called an *object file* or *machine code*, which is not human-readable and is meant for execution by the processor.
3. The object code is then passed through the linker. The linker links the object code with the required libraries (DLLs) to produce an executable that can be run on a system:

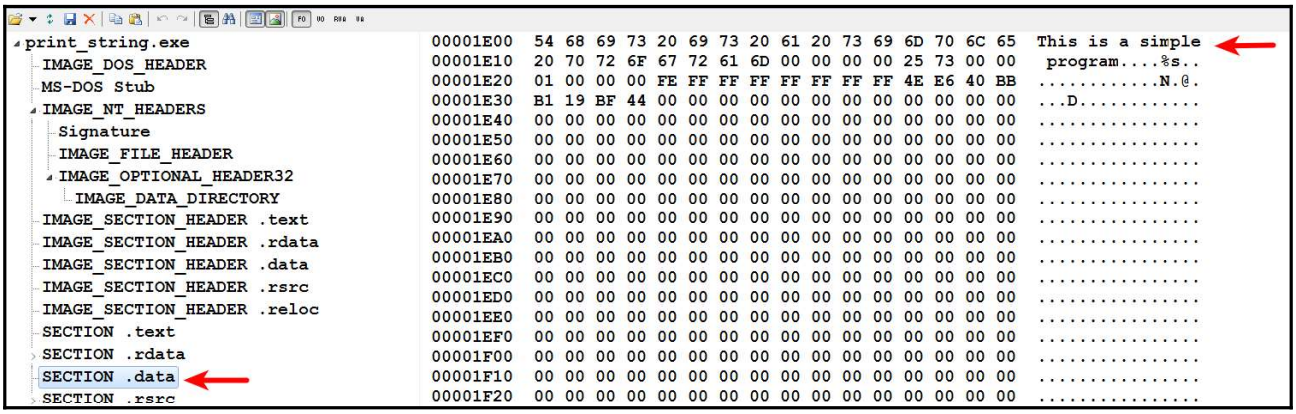


1.3.2 Program On Disk

Let's try to understand how a compiled program appears on the disk, with an example. Let's take an example of a simple C program that prints a string to the screen:

```
#include <stdio.h>
int main() {
   char *string = "This is a simple program";
   printf("%s",string);
   return 0;
}
```

The above program was passed through a compiler to generate an executable file (`print_string.exe`). Opening the compiled executable file in the PE Internals tool (`http://www.andreybazhan.com/pe-internals.html`) displays the five sections (`.text`, `.rdata`, `.data`, `.rsrc`, and `.reloc`) generated by the compiler. Information about the sections was provided in `Chapter 2`, *Static Analysis*. Here, we will mainly focus on two sections: `.text` and `.data`. The content of the `.data` section is shown in the following screenshot:

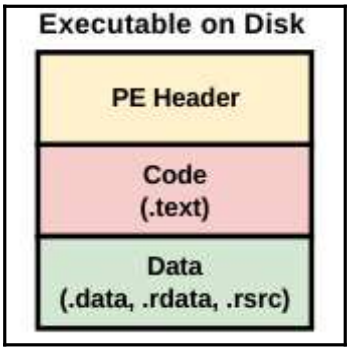


In the preceding screenshot, you can see that the string `This is a simple program`, which we used in our program, is stored in the `.data` section at the file offset `0x1E00`. This string is not a code, but it is the data required by the program. In the same manner, the `.rdata` section contains read-only data and sometimes contains *import/export* information. The `.rsrc` section contains resources used by the executable.

The content of the `.text` section is shown in the following screenshot:

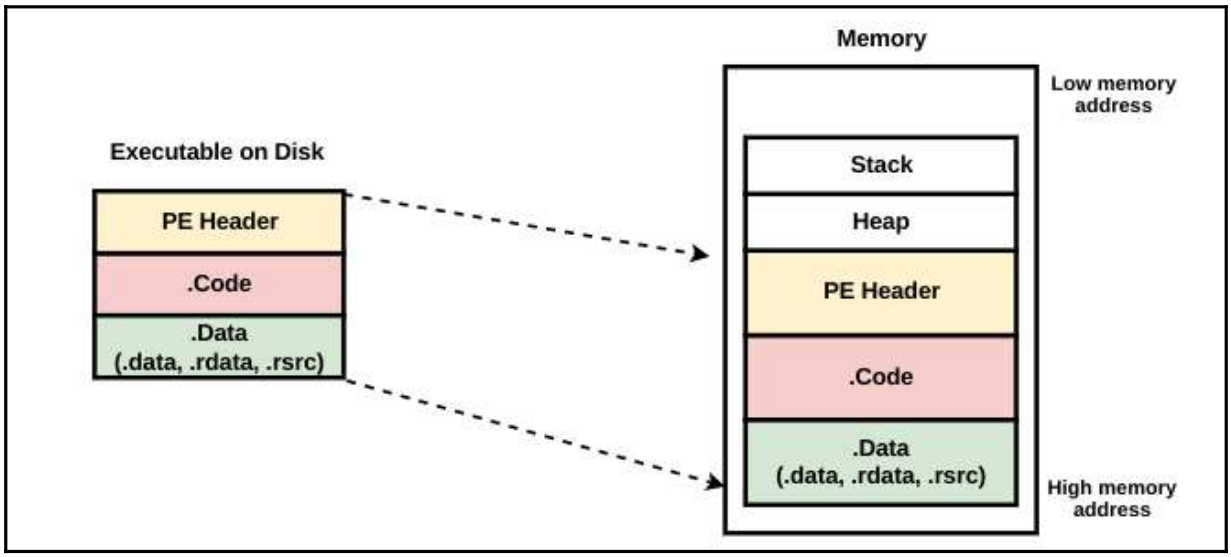
```
00000400  55 8B EC 51 C7 45 FC 00 30 40 00 8B 45 FC 50 68  U..Q.E..0@..E.Ph
00000410  1C 30 40 00 FF 15 98 20 40 00 83 C4 08 33 C0 8B  .0@.... @....3..
00000420  E5 5D C3 CC FF 25 98 20 40 00 CC CC CC CC CC CC  .]...%. @.......
00000430  55 8B EC E8 38 03 00 00 A3 40 30 40 00 6A 01 FF  U...8....@0@.j..
```

The sequence of bytes (`35` bytes to be specific) displayed in the `.text` section (starting from the file offset `0x400`) is the *machine code*. The source code that we had written was translated into machine code (or machine language program) by the compiler. The machine code is not easy for humans to read, but the processor (CPU) knows how to interpret those sequences of bytes. The machine code contains instructions that will be executed by the processor. The compiler segregated the data and the code in different sections on the disk. For the sake of simplicity, we can think of an executable as containing code (`.text`) and data (`.data`, `.rdata`, and so on):



1.3.3 Program In Memory

In the previous section, we examined the structure of the executable on the disk. Let's try to understand what happens when an executable is loaded into the 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 operating system loader. The following simplified memory layout should help you to visualize the concept; note that the structure of the executable on the disk is similar to the structure of the executable in the memory:



In the preceding diagram, the heap is used for dynamic memory allocation during program execution, and its contents can vary. The stack is used for storing the local variables, function arguments, and the return address. You will learn about the stack in detail in later sections.

The memory layout shown previously is greatly simplified, and the positions of components may be in any order. The memory also contains various *Dynamic Link Libraries (DLLs)*, which are not shown in the preceding diagram, to keep it simple. You will learn about the process memory in detail in the upcoming chapters.

Now, let's go back to our compiled executable (`print_string.exe`) and load it into the memory. The executable was opened in the *x64dbg* debugger, which loaded the executable in the memory (we will be covering *x64dbg* in a later chapter; for now, we will focus on the structure of the executable in memory). In the following screenshot, you can see that the executable was loaded at the memory address `0x010F0000`, and all the sections of the executable were also loaded into the memory. A point to remember is that the memory address that you are looking at is the virtual address, not the physical memory address. The virtual address will eventually be translated into a physical memory address (you will learn more about the virtual and physical address in later chapters):

Address	Info	Size	Content	Type	Protection
010F0000	print_string.exe	00001000		IMG	-R---
010F1000	".text"	00001000	Executable code	IMG	ER---
010F2000	".rdata"	00001000	Read-only initialized data	IMG	-R---
010F3000	".data"	00001000	Initialized data	IMG	-RWC-
010F4000	".rsrc"	00001000	Resources	IMG	-R---
010F5000	".reloc"	00001000	Base relocations	IMG	-R---

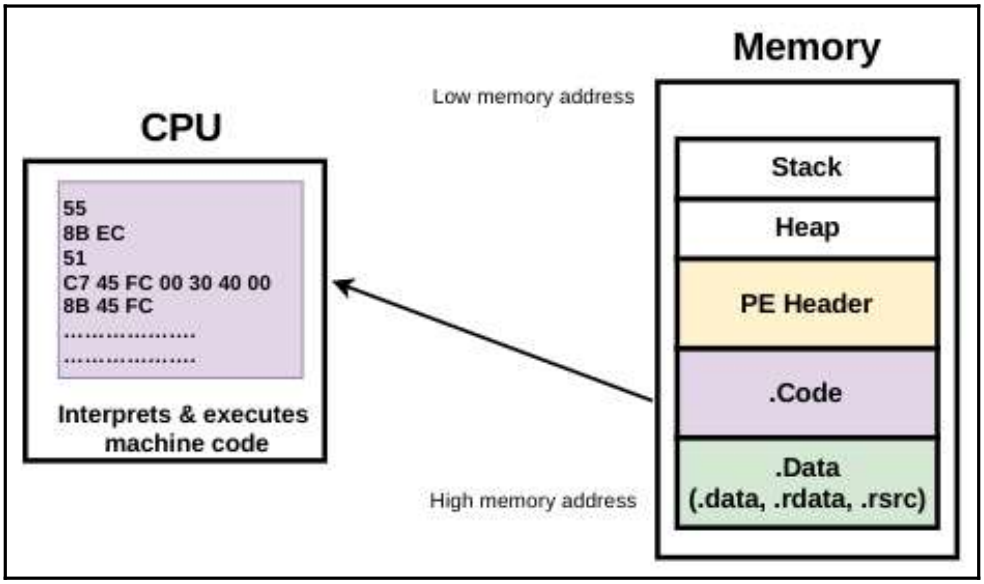
Examining the memory address of the `.data` section at `0x010F3000` displays the string `This is a simple program`.

Address	Hex				ASCII
010F3000	54 68 69 73	20 69 73 20	61 20 73 69	6D 70 6C 65	This is a simple
010F3010	20 70 72 6F	67 72 61 6D	00 00 00 00	25 73 00 00	program....%s..
010F3020	01 00 00 00	FE FF FF FF	FF FF FF FF	4E E6 40 BB	....þÿÿÿÿÿÿÿNæ@»
010F3030	B1 19 BF 44	00 00 00 00	00 00 00 00	00 00 00 00	±.¿D....

Examining the memory address of the `.text` section at `0x010F1000` displays the sequence of bytes, which is the machine code.

Address	Hex				ASCII
010F1000	55 8B EC 51	C7 45 FC 00	30 0F 01 8B	45 FC 50 68	U.ìQÇEü.0...EüPh
010F1010	1C 30 0F 01	FF 15 98 20	0F 01 83 C4	08 33 C0 8B	.0..ÿ.. ...Ä.3À.
010F1020	E5 5D C3	CC FF 25 98 20	0F 01 CC CC	CC CC CC CC	å]ÃÌÿ%. ..ÌÌÌÌÌÌ

Once the executable that contains the code and data 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. In our example, the CPU fetches the machine code containing the instructions (to print on the screen) from the `.text` section, and it fetches the string (data) `This is a simple program`, to be printed from the `.data` section. The following diagram should help you to visualize the interactions between the CPU and the memory:

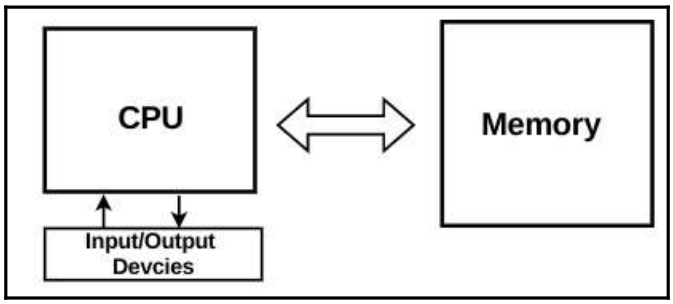


While executing instructions, the program may also interact with the input/output devices. In our example, when the program is executed, the string is printed onto the computer screen (output device). If the machine code had an instruction to receive input, the processor (CPU) would have interacted with the input device (such as the keyboard).

To summarize, the following steps are performed when a program is executed:

1. The program (which contains code and data) is loaded into the memory.
2. The CPU fetches the machine instruction, decodes it, and executes it.
3. The CPU fetches the required data from memory; the data can also be written to the memory.

4. The CPU may interact with the input/output system, as necessary:



1.3.4 Program Disassembly (From Machine code To Assembly code)

As you would expect, machine code contains information about the inner workings of the program. For example, in our program, the machine code included the instructions to print on the screen, but it would be painful for a human to try to understand the machine code (which is stored as a sequence of bytes).

A *disassembler/debugger* (like *IDA Pro* or *x64dbg*) is a program that translates machine code into a low-level code called *assembly code (assembly language program)*, which can be read and analyzed to determine the workings of a program. The following screenshot shows the machine code (a sequence of bytes in the `.text` section) translated into the assembly instructions representing `13` executable instructions (`push ebp`, `mov ebp,esp`, and so on). These translated instructions are called *assembly language instructions*.

You can see that the assembly instructions are much easier to read than the machine code. Notice how a disassembler translated the byte `55` into a readable assembly instruction `push ebp`, and the next two bytes `8B EC` into `mov ebp,esp`; and so on:

```
010F1000   55                 push ebp
010F1001   8B EC              mov ebp,esp
010F1003   51                 push ecx
010F1004   C7 45 FC 00 30 0F 01 mov dword ptr ss:[ebp-4],print_string.10F3000 10F3000:"This is a simple program"
010F100B   8B 45 FC           mov eax,dword ptr ss:[ebp-4]
010F100E   50                 push eax
010F100F   68 1C 30 0F 01     push print_string.10F301C                      10F301C:"%s"
010F1014   FF 15 98 20 0F 01  call dword ptr ds:[<&printf>]
010F101A   83 C4 08           add esp,8
010F101D   33 C0              xor eax,eax
010F101F   8B E5              mov esp,ebp
010F1021   5D                 pop ebp
010F1022   C3                 ret
```

From a code analysis perspective, determining the program's functionality mainly relies on understanding these assembly instructions and how to interpret them.

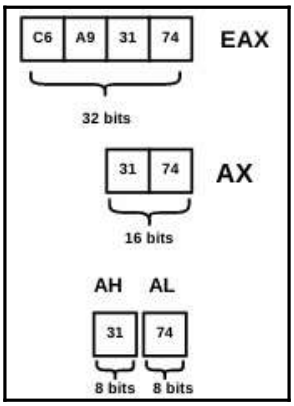
In the rest of the chapter, you will learn the skills required to understand the assembly code to reverse engineer the malicious binary. In the upcoming sections, you will learn the concepts of x86 assembly language instructions that are essential to perform code analysis; x86, also known as IA-32 (32-bit), is the most popular architecture for PCs. Microsoft Windows runs on an x86 (32-bit) architecture and Intel 64 (x64) architectures. Most malware that you will encounter are compiled for x86 (32 bit) architectures and can run on both 32 bit and 64 bit Windows. At the end of the chapter, you will understand the x64 architecture and the differences between x86 and x64.

2. CPU Registers

As mentioned previously, the CPU contains special storage called *registers*. The CPU can access data in registers much faster than data in memory, because of which the values fetched from the memory are temporarily stored in these registers to perform operations.

2.1 General-Purpose Registers

The x86 CPU has eight general purpose registers: `eax`, `ebx`, `ecx`, `edx`, `esp`, `ebp`, `esi`, and `edi`. These registers are 32 bits (4 bytes) in size. A program can access registers as 32-bit (4 bytes), 16-bit (2 bytes), or 8-bit (1 byte) values. The lower 16 bits (2 bytes) of each of these registers can be accessed as `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si`, and `di`. The lower 8 bits (1 byte) of `eax`, `ebx`, `ecx`, and `edx` can be referenced as `al`, `bl`, `cl`, and `dl`. The higher set of 8 bits can be accessed as `ah`, `bh`, `ch`, and `dh`. In the following diagram, the `eax` register contains the 4-byte value `0xC6A93174`. A program can access the lower 2 bytes (`0x3174`) by accessing the `ax` register, and it can access the lower byte (`0x74`) by accessing the `al` register, and the next byte (`0x31`) can be accessed by using the `ah` register:



2.2 Instruction Pointer (EIP)

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

2.3 EFLAGS Register

The `eflags` register is a 32-bit register, and each bit in this register is a *flag*. The bits in `EFLAGS` registers are used to indicate the status of the computations and to control the CPU operations. The flag register is usually not referred to directly, but during the execution of computational or conditional instructions, each flag is set to either `1` or `0`. Apart from these registers, there are additional registers, which are called *segment registers* (`cs`, `ss`, `ds`, `es`, `fs`, and `gs`), which keep track of sections in the memory.

3. Data Transfer Instructions

One of the basic instructions in the assembly language is the `mov` instruction. As the name suggest, this instruction moves data from one location to another (from source to destination). The general form of the `mov` instruction is as follows; this is similar to the assignment operation in a high-level language:

```
mov dst,src
```

There are different variations of the `mov` instruction, which will be covered next.

3.1 Moving a Constant Into Register

The first variation of the `mov` instruction is to move a *constant (or immediate value)* into a register. In the following examples, `;` (a semicolon) indicates the start of the comment; anything after the semicolon is not part of the assembly instruction. This is just a brief description to help you understand this concept:

```
mov eax,10  ; moves 10 into EAX register, same as eax=10
mov bx,7    ; moves 7 in bx register, same as bx=7
mov eax,64h ; moves hex value 0x64 (i.e 100) into EAX
```

3.2 Moving Values From Register To Register

Moving a value from one register to another is done by placing the register names as operands to the `mov` instruction:

```
mov eax,ebx ; moves content of ebx into eax, i.e eax=ebx
```

Following is an example of two assembly instructions. The first instruction moves the constant value `10` into the `ebx` register. The second instruction moves the value of `ebx` (in other words, `10`) into the `eax` register; as a result, the `eax` register will contain the value `10`:

```
mov ebx,10  ; moves 10 into ebx, ebx = 10
mov eax,ebx ; moves value in ebx into eax, eax = ebx or eax = 10
```

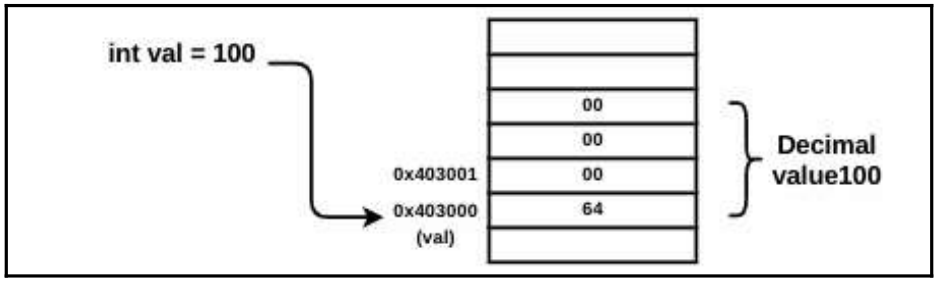
3.3 Moving Values From Memory To Registers

Before looking at the assembly instruction to move a value from the memory to a register, let's try to understand how values reside in the memory. Let's say you have defined a variable in your C program:

```
int val = 100;
```

The following list outlines what happens during the runtime of the program:

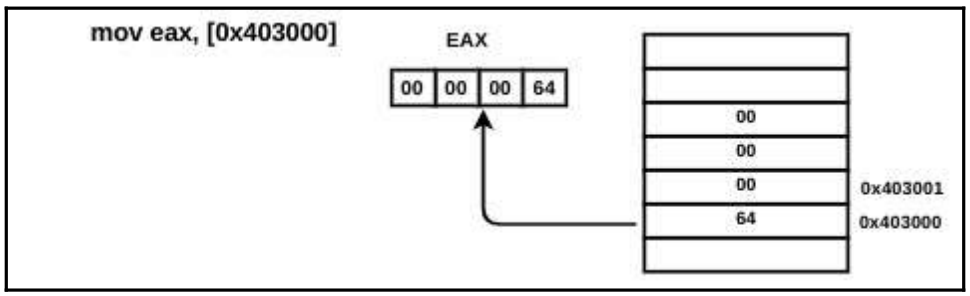
1. An integer is 4 bytes in length, so the integer `100` is stored as a sequence of 4 bytes (`00 00 00 64`) in the memory.
2. The sequence of four bytes is stored in the *little-endian* format mentioned previously.
3. The integer `100` is stored at some memory address. Let's assume that `100` was stored at the memory address starting at `0x403000`; you can think of this memory address labeled as `val`:



To move a value from the memory into a register in assembly language, you must use the address of the value. The following assembly instruction will move the 4 bytes stored at the memory address `0x403000` into the register `eax`. The square bracket specifies that you want the value stored at the memory location, rather than the address itself:

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

Notice that in the preceding instruction, you did not have to specify 4 bytes in the instruction; based on the size of the destination register (`eax`), it automatically determined how many bytes to move. The following screenshot will help you to understand what happens after executing the preceding instruction:



During reverse engineering, you will normally see instructions similar to the ones shown as below. The square brackets may contain a *register*, *a constant added to a register*, or *a register added to a register*. All of the following diagram instructions move values stored at the memory address specified within the square brackets to the register. The simplest thing to remember is that everything within the square brackets represents an address:

```
mov eax,[ebx]     ; moves value at address specifed by ebx register
mov eax,[ebx+ecx] ; moves value at address specified by ebx+ecx
mov ebx,[ebp-4]   ; moves value at address specified by ebp-4
```

Another instruction that you will normally come across is the `lea` instruction, which 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
```

Sometimes, you will come across instructions like the ones that follow. These instructions are the same as the previously mentioned instructions and transfer data stored in a memory address (specified by `ebp-4`) into the register. The `dword ptr` just indicates that a 4-byte (`dword`) value is moved from the memory address specified by `ebp-4` into the `eax`:

```
mov eax,dword ptr [ebp-4]  ; same as mov eax,[ebp-4]
```

3.4 Moving Values From Registers To Memory

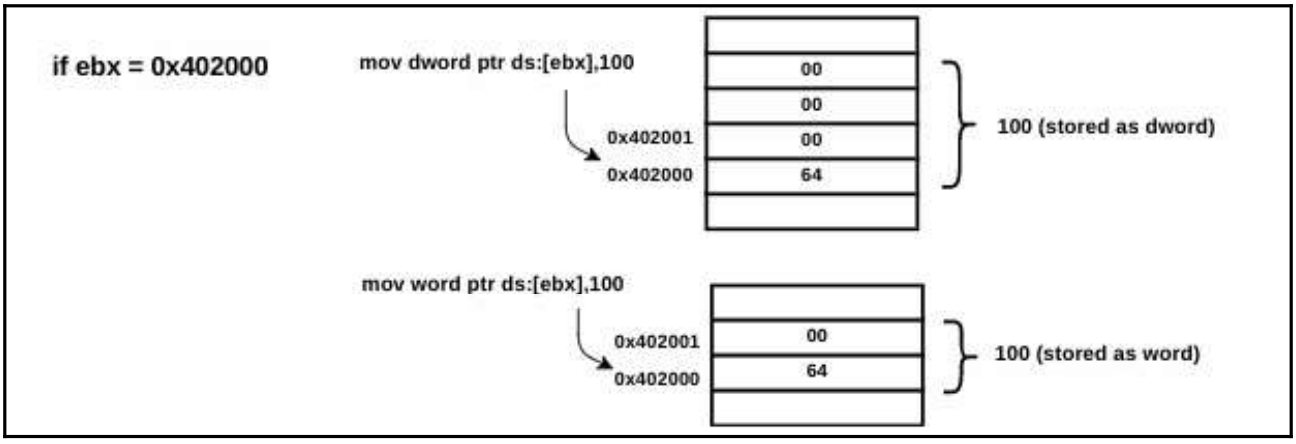
You can move a value from a register to memory by swapping operands so that the memory address is on the left-hand side (destination) and the register is on the right-hand side (source):

```
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
```

Sometimes, you will come across instructions like those that follow. These instructions move constant values into a memory location; `dword ptr` just specifies that a `dword` value (4 bytes) is moved into the memory location. Similarly, `word ptr` specifies that a `word` (2 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) into the memory location starting at the address `0x402000`, and the third instruction copies `100` as `00 64` (2 bytes) into the memory location starting at `0x40200`, as shown here:



Let's take a look at a simple challenge.

3.5 Disassembly Challenge

The following is a disassembled output of a simple C code snippet. Can you figure out what this code snippet does, and can you translate it back to a pseudocode (high-level language equivalent)? Use all of the concepts that you have learned so far to solve the challenge. The answer to the challenge will be covered in the next section, and we will also look at the original C code snippet after we solve this challenge:

```
mov dword ptr [ebp-4],1  ❶
mov eax,dword ptr [ebp-4]  ❷
mov dword ptr [ebp-8],eax  ❸
```

3.6 Disassembly Solution

The preceding program copies a value from one memory location to another. At ❶, the program copies a `dword` value `1` into a memory address (specified by `ebp-4`). At ❷, the same value is copied into the `eax` register, which is then copied into a different memory address, `ebp-8`, at ❸.

The disassembled code might be difficult to understand initially, so let me break it down to make it simple. We know that in a high-level language like C, a variable that you define (for example, `int val;`) is just a symbolic name for a memory address (as mentioned previously). Going by that logic, let's identify the memory address references and give them a symbolic name. In the disassembled program, we have two addresses (within square brackets): `ebp-4` and `ebp-8`. Let's label them and give them symbolic names; let's say, `ebp-4 = a` and `ebp-8 = b`. Now, the program should look like the one shown here:

```
mov dword ptr [a],1     ; treat it as mov [a],1
mov eax,dword ptr [a]   ; treat it as mov eax,[a]
mov dword ptr [b],eax   ; treat it as mov [b],eax
```

In a high-level language, when you assign a value to a variable, let's say `val = 1`, the value `1` is moved into the address represented by the `val` variable. In assembly, this can be represented as `mov [val], 1`. In other words, `val = 1` in a high-level language is the same as `mov [val],1` in assembly. Using this logic, the preceding program can be written into a high-level language equivalent:

```
a = 1
eax = a
b = eax ❹
```

Recall that, the registers are used by the CPU for temporary storage. So, let's replace all of the register names with their values on the right-hand side of the = sign (for example, replace `eax` with its value, `a`, at ❹). The resultant code is shown here:

```
a = 1
eax = a ❺
b = a
```

In the preceding program, the `eax` register is used to temporarily hold the value of `a`, so we can remove the entry at ❺ (that is remove the entry containing registers on the left side of the = sign). We are now left with the simplified code shown here:

```
a = 1
b = a
```

In high-level languages, variables have data types. Let's try to determine the data types of these variables: `a` and `b`. Sometimes, it is possible to determine the data type by understanding how the variables are accessed and used. From the disassembled code, we know that the `dword` value (4 bytes) `1` was moved into the variable `a`, which was then copied to `b`. Now that we know these variables are 4 bytes in size, it means that they could be of the type `int`, `float`, or `pointer`. To determine the exact data type, let's consider the following.

The variables `a` and `b` cannot be `float`, because, from the disassembled code, we know that `eax` was involved in the data transfer operation. If it was a floating point value, the floating point registers would have been used, instead of using a general purpose register such as `eax`.

The variables `a` and `b` cannot be a `pointer` in this case, because the value `1` is not a valid address. So, we can guess that `a` and `b` should be of the type `int`.

Based on these observations, we can now rewrite the program as follows:

```
int a;
int b;

a = 1;
b = a;
```

Now that we have solved the challenge, let's look at the original C code snippet of the disassembled output. The original C code snippet is shown as follows. Compare it with what we determined. Notice how it was possible to build a program similar to the original program (it is not always possible to get the exact C program back), and also, it's now much easier to determine the functionality of the program:

```
int x = 1;
int y;
y = x;
```

If you are disassembling a bigger program, it would be hard to label all of the memory addresses. Typically, you will use the features of the disassembler or debugger to rename memory addresses and to perform code analysis. You will learn the features offered by the disassembler and how to use it for code analysis in the next chapter. When you are dealing with bigger programs, it is a good idea to break the program into small blocks of code, translate it into some high-level language that you are familiar with, and then do the same thing for the rest of the blocks.

4. Arithmetic Operations

You can perform addition, subtraction, multiplication, and division in assembly language. A addition and subtraction are performed using the `add` and `sub` instructions, respectively. These instructions take two operands: *destination* and *source*. The `add` instruction adds the source and destination and stores the result in the destination. The `sub` instruction subtracts the source from the destination operand, and the result is stored in the destination. These instructions set or clear flags in the `eflags` register, based on the operation. These flags can be used in the 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. The following outlines a few variations of these instructions:

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

There is a special increment (`inc`) and decrement (`dec`) instruction, which can be used to add `1` or subtract `1` from either a register or a memory location:

```
inc eax    ; same as eax = eax+1
dec ebx    ; same as ebx = ebx-1
```

Multiplication is done using the `mul` instruction. The `mul` instruction takes only one operand; that operand is multiplied by the content of either the `al`, `ax`, or `eax` register. The result of the multiplication is stored in either the `ax`, `dx and ax`, or `edx and eax` register.

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 a *32-bit (4 bytes)*, then it is multiplied with the `eax` register, and the product is stored in the `edx` and `eax` register. The reason the product is stored in a register double the size is because when two values are multiplied, the output values can be much larger than the input values. The following outlines variations of `mul` instructions:

```
mul ebx  ;ebx is multiplied with eax and 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 can be either a register or a memory reference. To perform division, you place the dividend (number to divide) in the `edx and eax` register, 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
```

4.1 Disassembly Challenge

Let's take on another simple challenge. The following is a disassembled output of a simple C program. Can you figure out what this program does, and can you translate it back to a pseudocode?

```
mov dword ptr [ebp-4], 16h
mov dword ptr [ebp-8], 5
mov eax, [ebp-4]
add eax, [ebp-8]
mov [ebp-0Ch], eax
mov ecx, [ebp-4]
sub ecx, [ebp-8]
mov [ebp-10h], ecx
```

4.2 Disassembly Solution

You can read the code line by line and try to determine the program's logic, but it would be easier if you translate it back to some high-level language. To understand the preceding program, let's use the same logic that was covered previously. The preceding code contains four memory references. First, let's label these addresses - `ebp-4=a`, `ebp-8=b` , `ebp-0Ch=c`, and `ebp-10H=d`. After labeling the addresses, it translates to the following:

```
mov dword ptr [a], 16h
mov dword ptr [b], 5
mov eax, [a]
add eax, [b]
mov [c], eax
mov ecx, [a]
sub ecx, [b]
mov [d], ecx
```

Now, let's translate the preceding code into a pseudocode (high-level language equivalent). The code will as follows:

```
a = 16h    ; h represents hexadecmial, so 16h (0x16) is 22 in decimal
b = 5
eax = a
eax = eax + b  ❶
c = eax  ❶
ecx = a
ecx = ecx-b  ❶
d = ecx  ❶
```

Replacing all of the register names with their corresponding values on the right-hand side of the = operator (in other words, at ❶), we get the following code:

```
a = 22
b = 5
eax = a  ❷
eax = a+b  ❷
c = a+b
ecx = a  ❷
ecx = a-b  ❷
d = a-b
```

After removing all of the entries containing registers on the left-hand side of the = sign at ❷ (because registers are used for temporary calculations), we are left with the following code:

```
a = 22
b = 5
c = a+b
d = a-b
```

Now, we have reduced the eight lines of assembly code to four lines of pseudocode. At this point, you can tell that the code performs addition and subtraction operations and stores the results. You can determine the variable types based on the sizes and how they are used in the code (context), as mentioned earlier. The variables `a` and `b` are used in addition and subtraction, so these variables have to be of integer data types, and the variables `c` and `d` store the results of integer addition and subtraction, so it can be guessed that they are also integer types. Now, the preceding code can be written as follows:

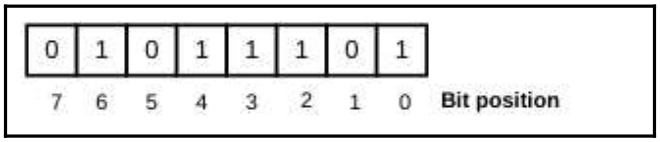
```
int a,b,c,d;
a = 22;
b = 5;
c = a+b;
d = a-b;
```

If you are curious about how the original C program of the disassembled output looks, then the following is the original C program to satisfy your curiosity. Notice how we were able to write an assembly code back to its equivalent high-level language:

```
int num1 = 22;
int num2 = 5;
int diff;
int sum;
sum = num1 + num2;
diff = num1 - num2;
```

5. Bitwise Operations

In this section, you will learn the assembly instructions that operate on the bits. The bits are numbered starting from the far right; the *rightmost bit (least significant bit)* has a bit position of `0`, and the bit position increases toward the left. The left-most bit is called the *most significant bit*. The following is an example showing the bits and the bit positions for a byte, `5D (0101 1101)`. The same logic applies to a `word`, `dword`, and `qword`:



One of the bitwise instructions is the `not` instruction; it takes only one operand (which serves as both the source and destination) and inverts all of the bits. If `eax` contained `FF FF 00 00 (11111111 11111111 00000000 00000000)`, then the following instruction would invert all of the bits and store it in the `eax` register. As a result, the `eax` would contain `00 00 FF FF (00000000 00000000 11111111 11111111)`:

```
not eax
```

The `and`, `or`, and `xor` instructions perform bitwise `and`, `or`, and `xor` operations and store the results in the destination. These operations are similar to `and (&)`, `or (|)`, and `xor (^)` operations in the C or Python programming languages. In the following example, the `and` operation is performed on bit `0` of the `bl` register and the bit `0` of `cl`, bit `1` of `bl` and the bit `1` of `cl`, and so on. The result is stored in the `bl` register:

```
and bl,cl  ; same as bl = bl & cl
```

In the preceding example, if `bl` contained `5 (0000 0101)` and `cl` contained `6 (0000 0110)`, then the result of the `and` operation would be `4 (0000 0100)`, as shown here:

```
                    bl: 0000 0101
                    cl: 0000 0110
-------------------------------------
After and operation bl: 0000 0100
```

Similarly, `or` and `xor` operations are performed on the corresponding bits of the operands. The following shows some of the example instructions:

```
or eax,ebx   ; same as eax = eax | ebx
xor eax,eax  ; same eax = eax^eax, this operation clears the eax register
```

The `shr` (shift right) and `shl` (shift left) instructions take two operands (the destination and the count). The destination can be either a register or a memory reference. The general form is shown as follows. Both of the instructions shift the bits in the destination to the right or left by the number of bits specified by the count operand; these instructions perform the same operations as `shift left (<<)` and `shift right(>>)` in the C or Python programming languages:

```
shl dst,count
```

In the following example, the first instruction (`xor eax, eax`) clears the `eax` register, after which `4` is moved into the `al` register, and the content of the `al` register (which is `4 (0000 0100)`) is shifted left by `2` bits. As a result of this operation (the two left-most bits are removed, and the two `0` bits are appended to the right), after the operation the `al` register will contain `0001 0000` (which is `0x10`):

```
xor eax,eax
mov al,4
shl al, 2
```

For detailed information on how bitwise operators work, refer to `https://en.wikipedia.org/wiki/Bitwise_operations_in_C` and `https://www.programiz.com/c-programming/bitwise-operators`.

The `rol` (rotate left) and `ror` (rotate right) instructions are similar to shift instructions. Instead of removing the shifted bits, as with the shift operation, they are rotated to the other end. Some of the example instructions are shown here:

```
rol al,2
```

In the preceding example, if `al` contained `0x44 (0100 0100)`, then the result of the `rol` operation would be `0x11 (0001 0001)`.

6. Branching And Conditionals

In this section, we will focus on branching instructions. So far, you have seen instructions that execute sequentially; but many times, your program will need to execute code at a different memory address (like an `if/else` statement, looping, functions, and so on). This is achieved by using branching instructions. Branching instructions transfer the control of execution to a different memory address. To perform branching, jump instructions are typically used in the assembly language. There are two kinds of jumps: *conditional* and *unconditional*.

6.1 Unconditional Jumps

In an *unconditional* jump, the jump is always taken. The `jmp` instruction tells the CPU to execute code at a different memory address. This is similar to the `goto` statement in the C programming language. When the following instruction is executed, the control is transferred to the jump address, and the execution starts from there:

```
jmp <jump address>
```

6.2 Conditional Jumps

In *conditional* jumps, the control is transferred to a memory address based on some condition. To use a conditional jump, you need instructions that can alter the flags (*set* or *clear*). These instructions can be performing an *arithmetic* operation or a *bitwise* operation. The x86 instruction provides the `cmp` instruction, which subtracts the *second operand (source operand)* from the *first operand (destination operation)* and alters the flags without storing the difference in the destination. In the following instruction, if the `eax` contained the value `5`, then `cmp eax,5` would set the zero flag (`zf=1`), because the result of this operation is zero:

```
cmp eax,5  ; subtracts eax from 5, sets the flags but result is not stored
```

Another instruction that alters the flags without storing the result is the `test` instruction. The `test` instruction performs a bitwise `and` operation and alters the flags without storing the result. In the following instruction, if the value of `eax` was zero, then the zero flag would be set (`zf=1`), because when you `and 0` with `0` you get `0`:

```
test eax,eax ; performs and operation, alters the flags but result in not
stored
```

Both `cmp` and `test` instructions are normally used along with the conditional `jump` instruction for decision making.

There are a few variations of conditional jump instructions; the general format is shown here:

```
jcc <address>
```

The `cc` in the preceding format represents conditions. These conditions are evaluated based on the bits in the `eflags` register. The following table outlines the different conditional jump instructions, their aliases, and the bits used in the `eflags` register to evaluate the condition:

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

6.3 If Statement

From a reverse engineering perspective, it is important to identify the branching/conditional statements. To do that, it is essential to understand how branching/conditional statements (like `if, if-else` and `if-else if-else`) are translated into assembly language. Let's look at an example of a simple C program and try to understand how the `if` statement is implemented at the assembly level:

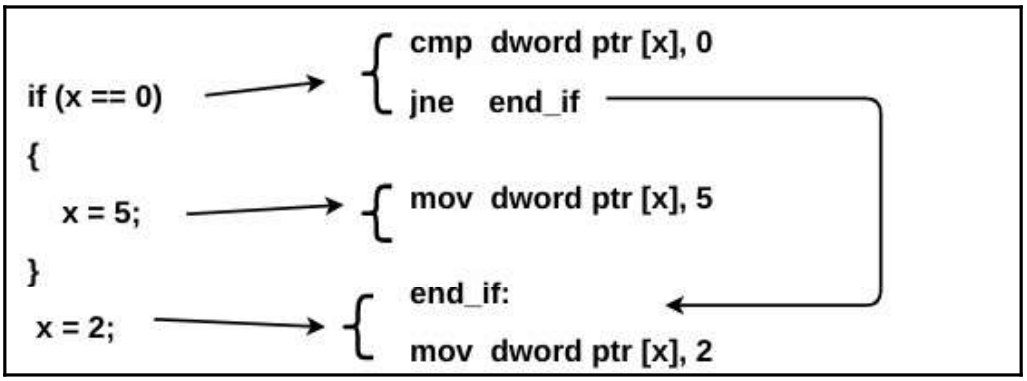
```
if (x == 0) {
   x = 5;
}
x = 2;
```

In the preceding C program, if the condition is true (`if x==0`), the code inside the `if` block is executed; otherwise, it will skip the `if` block and control is transferred to `x=2`. Think of a *control transfer* as a *jump*. Now, ask yourself: When will the jump be taken? The jump will be taken when `x` is not equal to `0`. That's exactly how the preceding code is implemented in assembly language (shown as follows); notice that in the first assembly instruction, the `x` is compared with `0`, and in the second instruction, the jump will be taken to `end_if` when `x` is not equal to `0` (in other words, it will skip `mov dword ptr [x],5` and execute `mov dword, ptr[x],2`). Notice how the equal to condition (`==`) in the C program was reversed to `not equal to` (`jne`) in the assembly language:

```
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 C programming statements and the corresponding assembly instructions:



6.4 If-Else Statement

Now, let's try to understand how the `if/else` statement is translated to assembly language. Let's take an example of the following C code:

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

In the preceding code, try to determine under what circumstances the jump would be taken (control would be transferred). There are two circumstances: the jump will be taken to the `else` block if the `x` is not equal to `0`, or, if `x` is equal to `0` (`if x == 0`), then after the execution of `x=5` (the end of the `if` block), a jump will be taken to bypass the `else` block, to execute the code after the `else` block.

The following is the assembly translation of the C program; notice that in the first line, the value of `x` is compared with `0`, and the jump (conditional jump) will be taken to the `else` block if the `x` is not equal to `0` (the condition was reversed, as mentioned previously). Before the `else` block, notice the unconditional jump to `end`. This jump ensures that if `x` is equal to `0`, after executing the code inside of the `if` block, the `else` block is skipped and the control reaches the end:

```
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 Statement

The following is a C code containing `if-ElseIf-else` statements:

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

From the preceding code, let's try to determine a situation when jumps (control transfers) will be taken. There are two conditional jump points; if `x` is not equal to `0`, it will jump to the `else_if` block, and if `x` is not equal to `1` (a condition check in `else if`), then the jump is taken to `else`. There are also two unconditional jumps: inside the `if` block after `x=5` (the end of the `if` block) and inside of the `else if` after `x=6` (the end of the `else if` block). Both of these unconditional jumps skip the `else` statement to reach the end.

The following is the translated assembly language showing the conditional and unconditional jumps:

```
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:
```

6.6 Disassembly Challenge

The following is the disassembled output of a program; let's 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:
```

6.7 Disassembly Solution

Let's start by assigning the symbolic names to the address (`ebp-4`). After assigning the symbolic names to the memory address references, we get the following code:

```
mov dword ptr [x], 1
cmp dword ptr [x], 0  ❶
jnz loc_40101C  ❷
mov eax, [x]  ❹
xor eax, 2
mov [x], eax
```

```
jmp loc_401025  ❸

loc_40101C:
mov ecx, [x]  ❺
xor ecx, 3
mov [x], ecx   ❻

loc_401025:
```

In the preceding code, notice the `cmp` and `jnz` instructions at ❶ and ❷ (this is a conditional statement) and note that `jnz` is the same as `jne` (`jump if not equal to`). Now that we have identified the conditional statement, let's try to determine what type of conditional statement this is (`if`, or `if/else`, or `if/else if/else`, and so on); to do that, focus on the jumps. The conditional jump at ❷ is taken to `loc_401010C`, and before the `loc_40101C`, there is an unconditional jump at ❸ to `loc_401025`. From what we learned previously, this has the characteristics of an `if-else` statement. To be precise, the code from ❹ to ❸ is part of the `if` block and the code from ❺ to ❻ is part of the `else` block. Let's rename `loc_40101C` to `else` and `loc_401025` to `end` for better readability:

```
mov dword ptr [x], 1  ❼
cmp dword ptr [x], 0  ❶
jnz else  ❷
mov eax, [x]  ❹
xor eax, 2
mov [x], eax  ❽
jmp end  ❸

else:
mov ecx, [x]  ❺
xor ecx, 3
mov [x], ecx  ❻
end:
```

In the preceding assembly code, `x` is assigned a value of `1` at ❼; the value of `x` is compared with `0`, and if it is equal to `0` (❶ and ❷), the value of `x` is `xored` with `2`, and the result is stored in `x` (❹ to ❽). If `x` is not equal to `0`, then the value of `x` is `xored` with `3` (❺ to ❻).

Reading the assembly code is slightly tricky, so let's write the preceding code in a high-level language equivalent. We know that ❶ and ❷ is an `if` statement, and you can read it as `jump is taken to else, if x is not equal to 0` (remember `jnz` is an alias for `jne`).

If you recall how the C code was translated to assembly, the condition in the `if` statement was reversed when translated to assembly code. Since we are now looking at the assembly code, to write these statements back to a high-level language, you need to reverse the condition. To do that, ask yourself this question, at ❷, when will the jump not be taken?. The jump will not be taken when `x` is equal to `0`, so you can write the preceding code to a pseudocode, as follows. Note that in the following code, the `cmp` and `jnz` instruction is translated to an `if` statement; also, note how the condition is reversed:

```
x = 1
if(x == 0)
{
  eax = x
  eax = eax ^ 2  ❾
  x = eax  ❾
}
else {
 ecx = x
 ecx = ecx ^ 3  ❾
 x = ecx  ❾
}
```

Now that we have identified the conditional statements, next let's replace all of the registers on the right-hand side of the = operator (at ❾) with their corresponding values. After doing that, we get the following code:

```
x = 1
if(x == 0)
{
  eax = x  ❿
  eax = x ^ 2  ❿
  x = x ^ 2
}
else {
  ecx = x  ❿
  ecx = x ^ 3  ❿
  x = x ^ 3
}
```

Removing all of the entries containing the registers on the left-hand side of the = operator (at ❿), we get the following code:

```
x = 1;
if(x == 0)
{
  x = x ^ 2;
}
```

```
else {
  x = x ^ 3;
}
```

If you are curious, the following is the original C program of the disassembled output used in the disassembly challenge; compare it with what we got in the preceding code snippet. As you can see, we were able to reduce multiple lines of assembly code back to their high-level language equivalent. Now, the code is much easier to understand, as compared to reading the assembly code:

```
int a = 1;
if (a == 0)
{
    a = a ^ 2;
}
else {
    a = a ^ 3;
}
```

7. Loops

Loops execute a block of code until some condition is met. The two most common types of loops are `for` and `while`. The jumps and conditional jumps that you have seen so far have been jumping forward. The loops jump backward. First, let's understand the functionality of a `for` loop. The general form of a `for` loop is shown here:

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

Here's how the `for` statement works. The `initialization` statement is executed only once, after which the `condition` is evaluated; if the condition is true, the block of code inside the `for` loop is executed, and then the `update_statement` is executed.

A `while` loop is the same as a `for` loop. In `for`, the `initialization`, `condition`, and `update_statment` are specified together, whereas in a `while` loop, the `initialization` is kept separate from the `condition` check, and the `update_statement` is specified inside the loop. The general form of a `while` loop is shown here:

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

```
}
```

Let's try to understand how the loop is implemented at the assembly level with the help of the following code snippet from a simple C program:

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

The preceding code can be written using a `while` loop, as shown here:

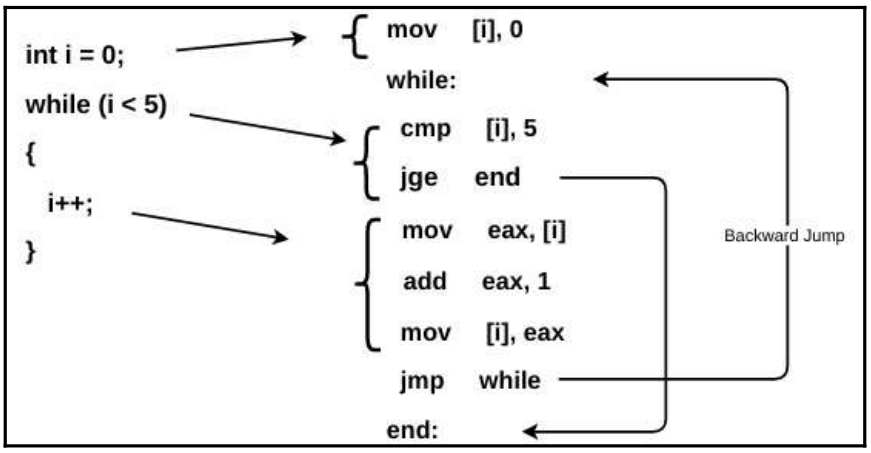
```
int i = 0;
while (i < 5) {
    i++;
}
```

We know that a jump is used to implement conditionals and loops, so let's think in terms of jumps. In the `while` and `for` loops, let's try to determine all the situations when the jumps will be taken. In both cases, when `i` becomes greater than or equal to `5`, a jump will be taken, which will transfer the control outside of the loop (in other words, after the loop). When `i` is less than `5`, the code inside the `while` loop is executed and after `i++` backward jump will be taken, to check the condition.

This is how the preceding code is implemented in assembly language (shown as follows). In the following assembly code, at ❶, notice a backward jump to an address (labeled as `while_start`); this indicates a loop. Inside of the loop, the condition is checked at ❷ and ❸ by using `cmp` and `jge` (jump if greater than or equal to) instructions; here, the code is checking if `i` is greater than or equal to `5`. If this condition is met, then the jump is taken to `end` (outside of the loop). Notice how the `less than` (<) condition in C programming is reversed to `greater than or equal to` (>=) at ❸, using the `jge` instruction. The initialization is performed at ❹, where `i` is assigned the value of `0`:

```
  mov [i],0  ❹

while_start:
  cmp [i], 5  ❷
  jge end  ❸
  mov eax, [i]
  add eax, 1
  mov [i], eax
  jmp while_start  ❶
end:
```

The following diagram shows the C programming statements and the corresponding assembly instructions:



7.1 Disassembly Challenge

Let's translate the following code into its high-level equivalent. Use the techniques and the concepts that you have learned so far 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:
```

7.2 Disassembly Solution

The preceding code consists of two memory addresses (`ebp-4` and `ebp-8`); let's rename `ebp-4` to `x` and `ebp-8` to `y`. The modified code is shown here:

```
 mov dword ptr [y], 1
 mov dword ptr [x], 0

loc_401014:
 cmp dword ptr [x], 4  ❷
 jge loc_40102E  ❸
 mov eax, [y]
 add eax, [x]
 mov [y], eax
 mov ecx, [x]  ❺
 add ecx, 1
 mov [x], ecx  ❻
 jmp loc_401014 ❶

loc_40102E:  ❹
```

In the preceding code, at ❶, there is a backward jump to `loc_401014`, indicating a loop. At ❷ and ❸, there is a condition check for the variable `x` (using `cmp` and `jge`); the code is checking whether `x` is greater than or equal to `4`. If the condition is met, it will jump outside of the loop to `loc_40102E` (at ❹). The value of `x` is incremented to `1` (from ❺ to ❻), which is the update statement. Based on all of this information, it can be deduced that `x` is the loop variable that controls the loop. Now, we can write the preceding code to a high-level language equivalent; but to do that, remember that we need to reverse the condition from `jge` (`jump if greater than or equal to`) to `jump if less than`. After the changes, the code looks as follows:

```
y = 1
x = 0
while (x<4) {
eax = y
eax = eax + x  ❼
y = eax  ❼
ecx = x
ecx = ecx + 1  ❼
x = ecx  ❼
}
```

Replacing all of the registers on the right-hand side of the = operator (at ❼) with their previous values, we get the following code:

```
y = 1
x = 0
while (x<4) {
eax = y  ❽
eax = y + x  ❽
y = y + x
ecx = x  ❽
ecx = x + 1  ❽
x = x + 1
}
```

Now, removing all of the entries containing registers on the left-hand side of the = sign (at ❽), we get the following code:

```
y = 1;
x = 0;
while (x<4) {
y = y + x;
x = x + 1;
}
```

If you are curious, the following is the original C program of the disassembled output. Compare the preceding code that we determined with the code that follows from the original program; notice how it was possible to reverse engineer and decompile the disassembled output to its original equivalent:

```
int a = 1;
int i = 0;
while (i < 4) {
a = a + i;
i++;
}
```

8. 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. The 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*.

8.1 Stack

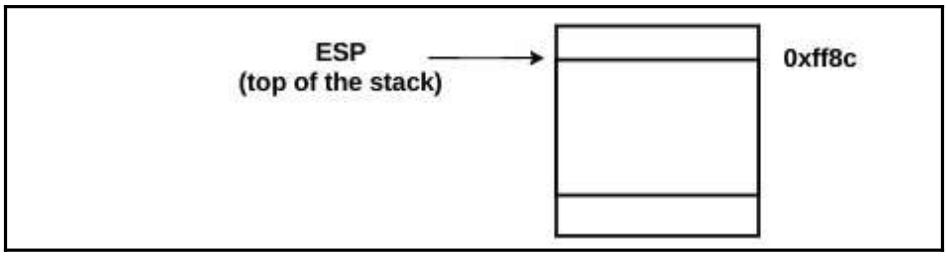
The stack is an area of the memory that gets allocated by the operating system when the thread is created. The stack is organized in a *Last-In-First-Out (LIFO)* structure, which means that the most recent data that you put in the stack will be the first one to be removed from the stack. You put data (called *pushing*) onto the stack by using the `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 general forms of the `push` and `pop` instructions are shown here:

```
push source     ; pushes source on top of the stack
pop destination ; copies value from the top of the stack to the destination
```

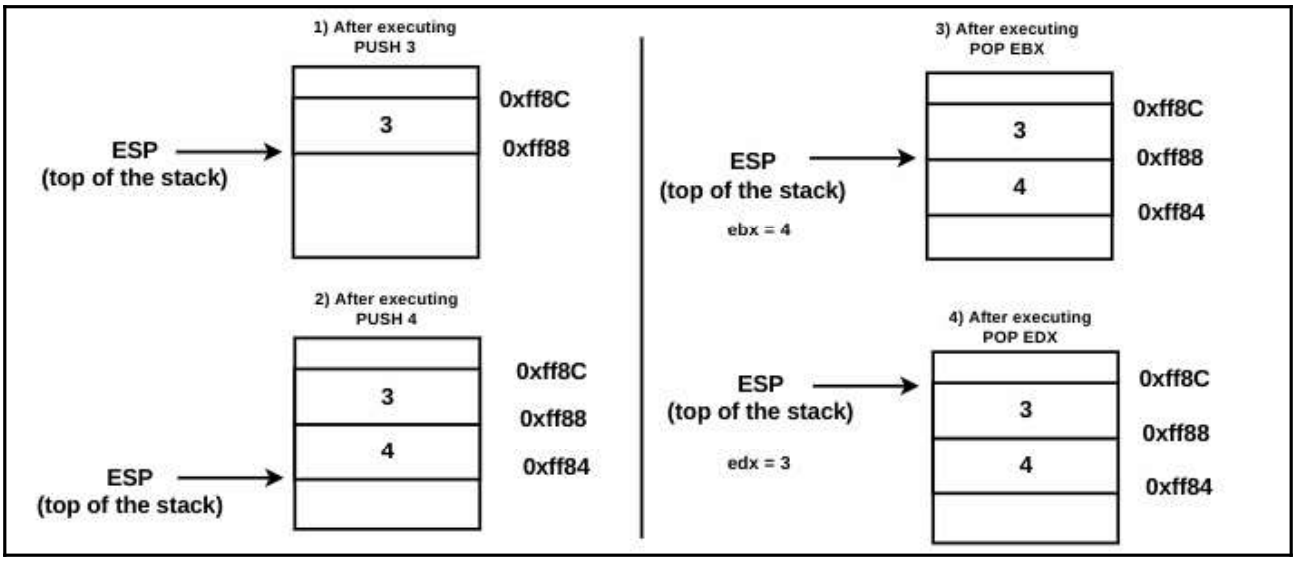
The stack grows from higher addresses to lower addresses. This means when a stack is created, the `esp` register (also called the *stack pointer*) points to the top of the stack (higher address), and as you `push` data into the stack, the `esp` register decrements by `4` (`esp-4`) to a lower address. When you `pop` a value, the `esp` increments by `4` (`esp+4`). Let's look at the following assembly code and try to understand the inner workings of the stack:

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

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



After the first instruction is executed (`push 3`), `ESP` is decremented by `4` (because the `push` instruction pushes a *4-byte* value onto the stack), and the value `3` is placed on the stack; now, `ESP` points to the top of the stack at `0xff88`. After the second instruction (`push 4`), `esp` is decremented by `4`; now, `esp` contains `0xff84`, which is now the top of the stack. When `pop ebx` is executed, the value `4` from the top of the stack is moved to the `ebx` register, and `esp` is incremented by `4` (because `pop` removes a *4-byte* value from the stack). So, `esp` now points to the stack at `0xff88`. Similarly, when the `pop edx` instruction is executed, the value `3` from the top of the stack is placed in the `edx` register, and `esp` comes back to its original position at `0xff8c`:



In the preceding diagram, the values popped from the stack are physically still present in memory, even though they are logically removed. Also, notice how the most recently pushed value (`4`) was the first to be removed.

8.2 Calling 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>
```

From a code analysis perspective, think of `some_function` as an address containing a block of code. When the `call` instruction is executed, the control is transferred to `some_function` (a block of code), but before that, it stores the address of the next instruction (the instruction following `call <some_function>`) by pushing it onto the stack. The address following the `call` which is pushed onto the stack is called the *return address*. Once `some_function` finishes executing, the return address that was stored on the stack is popped from the stack, and the execution continues from the popped address.

8.3 Returning From Function

In assembly language, to return from a function, you use the `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 the return value is placed in the `eax` register.

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 the `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`):

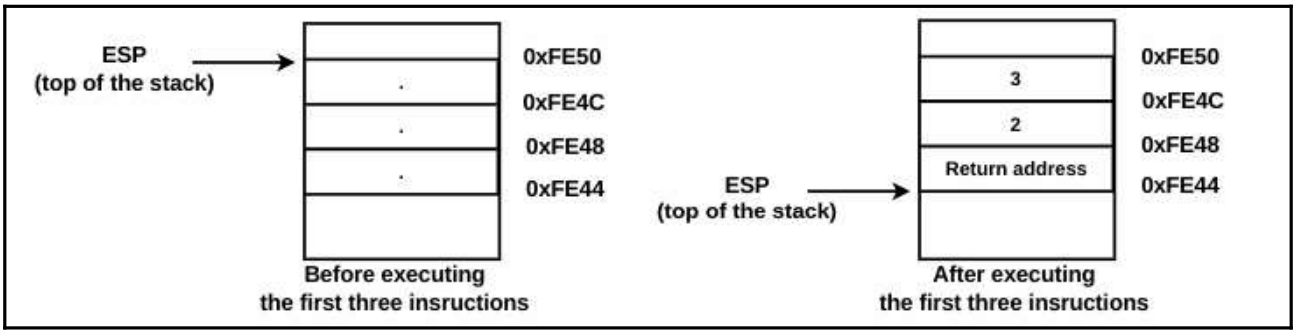
```
int test(int a, int b)
{
    int x, y;
    x = a;
    y = b;
    return 0;
}

int main()
{
```

```
    test(2, 3);
    return 0;
}
```

First, let's see how the statements inside the `main()` function are translated into assembly instructions:

```
push 3  ❶
push 2  ❷
call test  ❸
add esp, 8 ; after test is exectued, the control is returned here
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), and the second argument, `3`, is pushed before the first argument, `2`. After pushing the arguments, 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 diagram depicts what happens before and after executing ❶, ❷, and ❸:



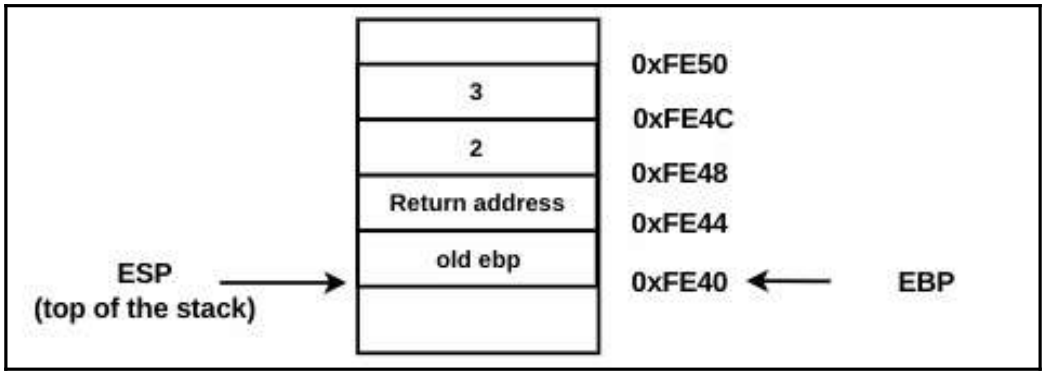
Now, let's focus on the `test` function, as shown here:

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

The following is the assembly translation of the `test` function:

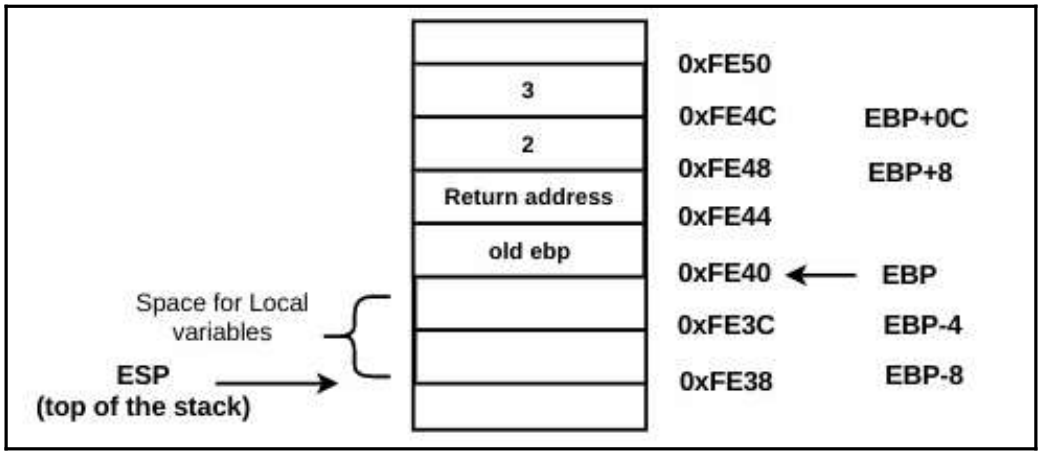
```
push ebp  ❹
mov ebp, esp  ❺
sub esp, 8  ❽
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
xor eax, eax  ❾
mov esp, ebp  ❻
pop ebp  ❼
ret  ❿
```

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. As a result of pushing the value of `ebp` onto the stack, the `esp` register will be decremented by `4`. 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, shown as follows. 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:



You will normally find `push ebp` and `mov ebp, esp` at the start of most functions; these two instructions are called *function prologue*. These instructions are responsible for setting up the environment for the function. 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*, and they restore the environment after the function is executed.

At ❽, `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 as follows:



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`). For example, in the preceding diagram, the first argument (`2`) can be accessed at the address `ebp+8` (which is the value of `a`), and the second argument can be accessed at the address `ebp+0xc` (which is the value of `b`). The local variables can be accessed at the addresses `ebp-4` (local variable `x`) and `ebp-8` (local variable `y`).

Most compilers (such as Microsoft Visual C/C++ compiler) make use of fixed `ebp` based stack frames to reference the function arguments and the local variables. The GNU compilers (such as gcc) do not use `ebp` based stack frames by default, but they make use of a different technique, where the `ESP` (stack pointer) register is used to reference the function parameters and local variables.

The actual code inside the function is between ❽ and ❻, which is shown here:

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

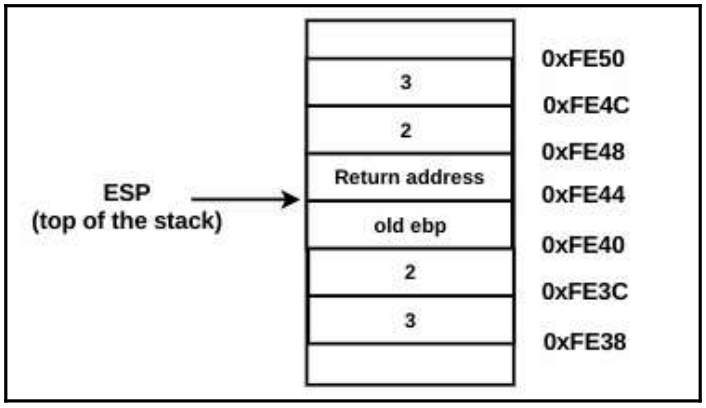
We can rename the argument `ebp+8` as `a` and `ebp+0Ch` as `b`. The address `ebp-4` can be renamed as the variable `x`, and `ebp-8` as the variable `y`, as shown here:

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

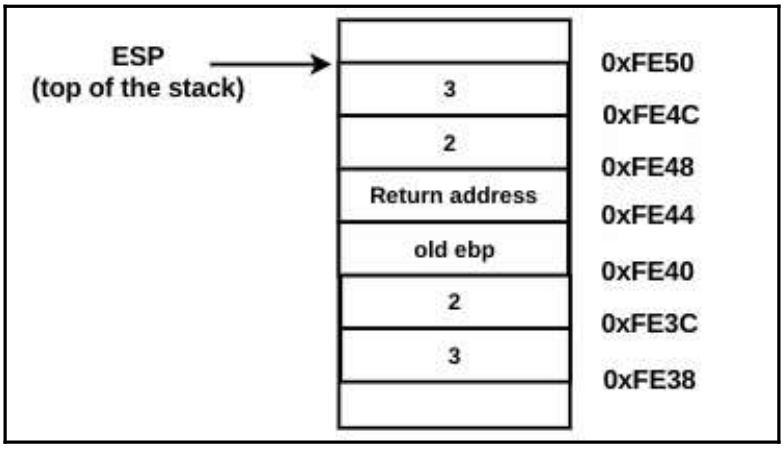
Using the techniques covered previously, the preceding statements can be translated to the following pseudocode:

```
x = a
y = b
```

At ❾, `xor eax,eax` sets the value of `eax` to `0`. This is the return value (`return 0`). The return value is always stored in the `eax` register. The *function epilogue* instructions at ❻ and ❼ restore the function environment. The instruction `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`. After the execution of the instructions at ❻ and ❼, the stack will look like the one shown here:



At ❿, 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 to its original position (the address `0xFE50`, from where we started), as follows. At this point, all of the values on the stack are logically removed, even though they are physically present. This is how the function works:



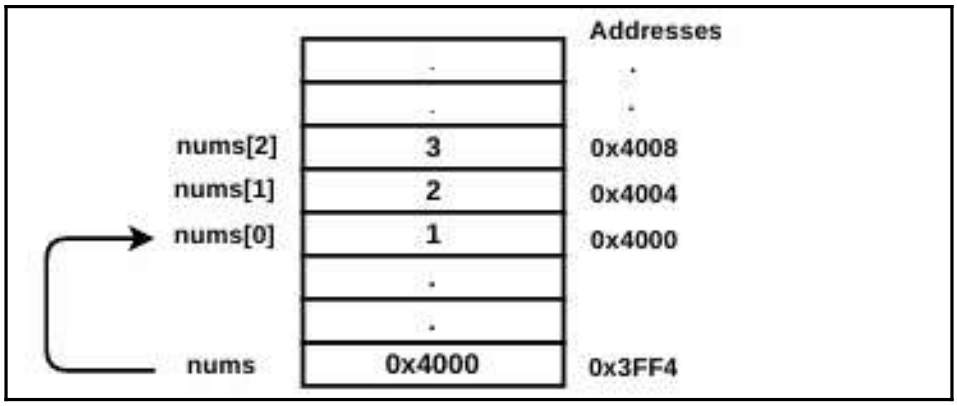
In the previous example, the `main` function called the `test` function and passed the parameters to the `test` function by pushing them onto the stack (in the right-to-left order). The `main` function is known as the *caller* (or the *calling function*) and `test` is the *callee* (or the *called functio*n). The `main` function (caller), after the function call, cleaned up the stack using `add esp,8` instruction. This instruction has the effect of removing the parameters that were pushed onto the stack and adjusts the stack pointer (`esp`) back to where it was before the function call; such a function is said to be using `cdecl` calling convention. The calling convention dictates how the parameters should be passed and who (*caller* or the *callee*) is responsible for removing them from the stack once the called function has completed. Most of the compiled C programs typically follow the `cdecl` calling convention. In the `cdecl` convention, the *caller* pushes the parameters in the right-to-left order on the stack and the *caller* itself cleans up the stack after the function call. There are other calling conventions such as `stdcall` and `fastcall`. In `stdcall`, parameters are pushed onto the stack (right-to-left order) by the *caller* and the *callee,* (*called function*) is responsible for cleaning up the stack. Microsoft Windows utilizes the `stdcall` convention for the functions (API) exported by the DLL files. In the `fastcall` calling convention, first few parameters are passed to a function by placing them in the registers, and any remaining parameters are placed on the stack in right-to-left order and the *callee* cleans up the stack similar to the `stdcall` convention. You will typically see 64-bit programs following the `fastcall` calling convention.

9. Arrays And 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. The following defines an integer array of three elements, and each element of this array occupies 4 bytes in the memory (because an integer is 4 bytes in length):

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

The array name `nums` is a pointer constant that points to the first element of the array (that is, the array name points to the `base address` of the array). In a high-level language, to access the elements of the array, you use the array name along with the `index`. For example, you can access the first element using `nums[0]`, the second element using `nums[1]`, and so on:



In assembly language, the address of any element in the array is computed using three things:

- The base address of the array
- The index of the element
- The size of each element in the array

When you use `nums[0]` in a high-level language, it is translated to `[nums+0*<size_of_each_element_in_bytes>]`, where `0` is the index and `nums` represents the base address of the array. From the preceding example, 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 for the `nums` integer array can be represented as follows:

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

The following shows the general format for accessing the elements of an array:

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

9.1 Disassembly Challenge

Translate the following code to its high-level equivalent. Use the techniques and the concepts that you have learned so far to solve this challenge:

```
push ebp
mov ebp, esp
sub esp, 14h
mov dword ptr [ebp-14h], 1
mov dword ptr [ebp-10h], 2
mov dword ptr [ebp-0Ch], 3
mov dword ptr [ebp-4], 0

loc_401022:
 cmp dword ptr [ebp-4], 3
 jge loc_40103D
 mov eax, [ebp-4]
 mov ecx, [ebp+eax*4-14h]
 mov [ebp-8], ecx
 mov edx, [ebp-4]
 add edx, 1
 mov [ebp-4], edx
 jmp loc_401022

loc_40103D:
 xor eax, eax
 mov esp, ebp
 pop ebp
 ret
```

9.2 Disassembly Solution

In the preceding code, the first two instructions (`push ebp` and `mov ebp, esp`) represent *function prologue*. Similarly, the two lines before the last instruction, `ret`, represent the *function epilogue* (`mov esp,ebp` and `pop ebp`). We know that the *function prologue* and *epilogue* are not part of the code, but they are used to set up the environment for the function, and hence they can be removed to simplify the code. The third instruction, `sub,14h`, suggests that `20 (14h)` bytes are allocated for local variables; we know that this instruction is also not part of the code (it's just used for allocating space for local variables), and can also be ignored. After removing the instructions that are not part of the actual code, we are left with the following:

```
1. mov dword ptr [ebp-14h], 1
2. mov dword ptr [ebp-10h], 2  ❼
3. mov dword ptr [ebp-0Ch], 3  ❽
4. mov dword ptr [ebp-4], 0  ❹

loc_401022:  ❷
5. cmp dword ptr [ebp-4], 3  ❸
6. jge loc_40103D  ❸
7. mov eax, [ebp-4]
8. mov ecx, [ebp+eax*4-14h]  ❻
9. mov [ebp-8], ecx
10. mov edx, [ebp-4]  ❺
11. add edx, 1  ❺
12. mov [ebp-4], edx  ❺
13. jmp loc_401022  ❶

loc_40103D:
14. xor eax, eax
15. ret
```

The backward jump at ❶, to `loc_401022`, indicates the loop, and the code between ❶ and ❷ is the part of the loop. Let's identify the `loop variable`, the `loop initialization`, the `condition check`, and the `update statement`. The two instructions at ❸ is a condition check that is checking whether the value of `[ebp-4]` is `greater than or equal to 3`; when this condition is met, a jump is taken outside of the loop. The same variable, `[ebp-4]`, is initialized to `0` at ❹ before the condition check at ❸, and the variable is incremented using the instructions at ❺. All of these details suggest that `ebp-4` is the loop variable, so we can rename `ebp-4` as `i` (`ebp-4=i`).

At ❻, the instruction `[ebp+eax*4-14h]` represents array access. Let's try to identify the components of the array (the `base address`, `index`, and the `size` of each element). We know that local variables (including elements of an array) are accessed as `ebp-<somevalue>` (in other words, the negative offset from `ebp`), so we can rewrite `[ebp+eax*4-14h]` as `[ebp-14h+eax*4]`. Here, `ebp-14h` represents the base address of the array on the stack, `eax` represents the `index`, and `4` is the size of each element of the array. Since `ebp-14h` is the base address, which means this address also represents the first element of the array, if we assume the array name is `val`, then `ebp-14h = val[0]`.

Now that we have determined the first element of the array, let's try to find the other elements. From the array notation, in this case, we know that the size of each element is `4` bytes. So, if `val[0] = ebp-14h`, then `val[1]` should be at the next highest address, which is `ebp-10h`, and `val[2]` should be at `ebp-0Ch`, and so on. Notice that `ebp-10h` and `ebp-0Ch` are referenced at ❼ and ❽. Let's rename `ebp-10h` as `val[1]` and `ebp-14h` as `val[2]`. We still haven't figured out how many elements this array contains. First, let's replace all of the determined values and write the preceding code in a high-level language equivalent. The last two instructions, `xor eax,eax` and `ret`, can be written as `return 0`, so the pseudocode now looks as follows:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
eax = i
ecx = [val+eax*4]  ❾
[ebp-8] = ecx  ❾
edx = i
edx = edx + 1  ❾
i = edx  ❾
}
return 0
```

Replacing all of the register names on the right-hand side of the = operator at ❾ with their corresponding values, we will get the following code:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
eax = i  ❿
ecx = [val+i*4]  ❿
```

```
[ebp-8] = [val+i*4]
edx = i  ❿
edx = i + 1  ❿
i = i + 1
}
return 0
```

Removing all of the entries containing register names on the left-hand side of the = operator at ❿, we get the following code:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
[ebp-8] = [val+i*4]
i = i + 1
}
return 0
```

From what we learned previously, when we access an element of the integer array using `nums[0]`, it is the same as `[nums+0*4]`, and `nums[1]` is the same as `[nums+1*4]`, which means that the general form of `nums[i]` can be represented as `[nums+i*4]` that is, `nums[i] = [nums+i*4]`. Going by that logic, we can replace `[val+i*4]` with `val[i]` in the preceding code.

Now, we are left with the address `ebp-8` in the preceding code; this could be a local variable, or it could be the fourth element in the array `val[3]` (it's really hard to say). If we assume it as a local variable and rename `ebp-8` as `x` (`ebp-8=x`), then the resultant code will look as shown below. From the following code, we can tell that the code probably iterates through each element of the array (using the index variable `i`) and assigns the value to the variable `x`. From the code, we can gather one extra piece of information: if the index `i` was used for iterating through each element of the array, then we can guess that the array probably has three elements (because the index `i` takes a maximum value of `2` before exiting the loop):

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
x = val[i]
i = i + 1
}
```

```
return 0
```

Instead of treating `ebp-8` as the local variable `x`, if you treat `ebp-8` as the array's fourth element (`ebp-8 = val[3]`), then the code will be translated to the following. Now, the code can be interpreted differently, that is, the array now has four elements and the code iterates through the first three elements. In every iteration, the value is assigned to the fourth element:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
val[3] = val[i]
i = i + 1
}
return 0
```

As you might have guessed from the preceding example, it is not always possible to decompile the assembly code to its original form accurately, because of the way the compiler generates code (and also, the code might not have all of the required information). However, this technique should help to determine the program's functionality. The original C program of the disassembled output is shown as follows; notice the similarities between what we determined previously and the original code here:

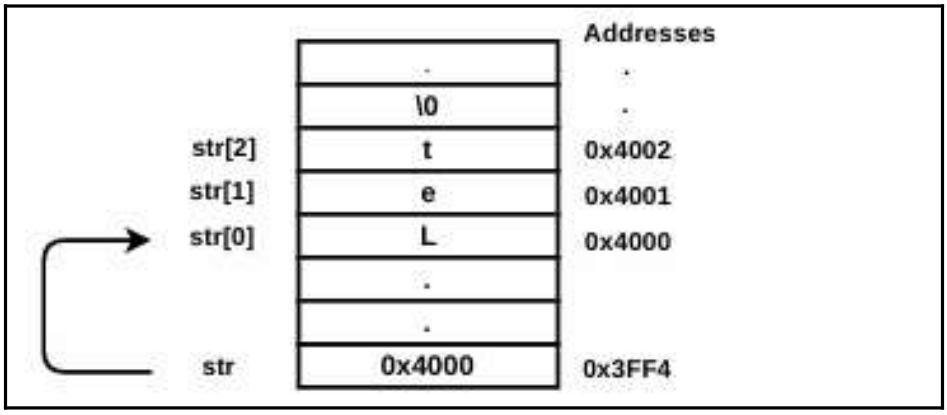
```
int main()
{
  int a[3] = { 1, 2, 3 };
  int b, i;
  i = 0;
   while (i < 3)
   {
     b = a[i];
     i++;
   }
  return 0;
}
```

9.3 Strings

A string is an array of characters. When you define a string, shown as follows, a *null terminator* (*string terminator*) is added at the end of every string. Each element occupies 1 byte of memory (in other words, each ASCII character is 1 byte in length):

```
char *str = "Let"
```

The string name `str` is a pointer variable that points to the first character in the string (in other words, it points to the base address of the character array). The following diagram shows how these characters reside in memory:



From the preceding example, you can access the elements of a character array (string), as shown here:

```
str[0] = [str+0] = [0x4000+0] = [0x4000] = L
str[1] = [str+1] = [0x4000+1] = [0x4001] = e
str[2] = [str+2] = [0x4000+2] = [0x4002] = t
```

The general form for the character array can be represented as follows:

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

9.3.1 String Instructions

The x86 family of processors provides string instructions, which operate on strings. These instructions step through the string (character array) and are suffixed with `b`, `w`, and `d`, which indicating 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* (it holds the address of the source string), and `edi` is the *destination address register* (it holds the address of the destination string).

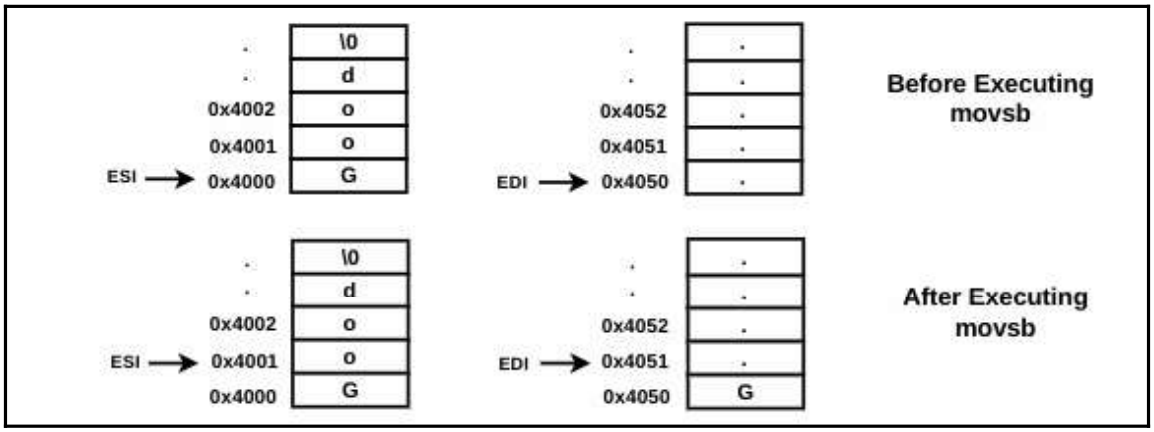
After performing a string operation, the `esi` and `edi` registers are either automatically incremented or decremented (you can think of `esi` and `edi` as source and destination index registers). The `direction flag (DF)` in the `eflags` register determines whether `esi` and `edi` should be incremented or decremented. The `cld` instruction clears the direction flag (`df=0`); if `df=0`, then the index registers (`esi` and `edi`) are incremented. The `std` instruction sets the direction flag (`df=1`); in such a case, `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 is used to move `1` byte from the address specified by the `esi` register to the address specified by the `edi` register. The `movsw, movsd` instructions move `2` and `4` bytes from the address specified by the `esi` to the address specified by `edi`. After the value is moved, the `esi` and `edi` registers are incremented/decremented by `1`, `2`, or `4` bytes, based on the size of the data item. In the following assembly code, let's assume that the address labeled as `src` contained the string `"Good"`, followed by a *null terminator* (`0x0`). After executing the first instruction at ❶, `esi` will contain the start address of the string `"Good"` (in other words, `esi` will contain the address of the first character, `G`), and the instruction at ❷ will set `EDI` to contain the address of a memory buffer (`dst`). The instruction at ❸ will copy 1 byte (the character `G`) from the address specified by `esi` to the address specified by `edi`. After executing the instruction at ❸, both `esi` and `edi` will be incremented by `1`, to contain the next address:

```
❶ lea esi,[src] ; "Good",0x0
❷ lea edi,[dst]
❸ movsb
```

The following screenshot will help you to understand what happens before and after executing the `movsb` instruction at ❸. Instead of `movsb`, if `movsw` is used, then 2 bytes will be copied from `src` to `dst`, and `esi` and `edi` will be incremented by `2`:



9.3.3 Repeat Instructions (rep)

The `movsx` instruction can only copy `1`, `2`, or `4` bytes, but to copy the 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 string instruction the number of times specified by the `ecx` register. After the `rep` instruction is executed, the value of `ecx` is decremented. The following assembly code copies the string `"Good"` (along with a *null terminator*) from `src` to `dst`:

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

The `rep` instruction, when used with the `movsx` instruction, is equivalent to the `memcpy()` function in C programming. The `rep` instruction has multiple forms, which allows early termination, based on the condition that occurs during the execution of the loop. The following table outlines different forms of `rep` instructions and their conditions:

Instruction	Condition
`rep`	Repeats until `ecx=0`
`repe, repz`	Repeats until `ecx=0` or `ZF=0`
`repne, repnz`	Repeat until `ecx=0` or `ZF=1`

9.3.4 Storing Value From Register to Memory (stosx)

The `stosb` instruction is used to move a byte from the CPU's `al` register to the memory address specified by `edi` (the *destination index register*). Similarly, the `stosw` and `stosd` instructions move data from `ax` (2 bytes) and `eax` (4 bytes) to the address specified by `edi`. Normally, the `stosb` instruction is used along with the `rep` instruction to initialize all of the bytes of the buffer to some value. The following assembly code fills the destination buffer with `5` double words (`dword`), all of them equal to `0` (in other words, it initializes `5*4 = 20` bytes of memory to `0`). The `rep` instruction, when used with `stosb`, is equivalent to the `memset()` function in C programming:

```
mov eax, 0
lea edi,[dest]
mov ecx,5
rep stosd
```

9.3.5 Loading From Memory to Register (lodsx)

The `lodsb` instruction moves a byte from a memory address specified by `esi` (the *source index register*) to the `al` register. Similarly, the `lodsw` and `lodsd` instructions move 2 bytes and 4 bytes of data from the memory address specified by `esi` to the `ax` and `eax` registers.

9.3.6 Scanning Memory (scasx)

The `scasb` instruction is used to search (or 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 (buffer) is placed in the `edi` register. The `scasb` instruction is mostly used with the `repne` instruction (`repne scasb`), with `ecx` set to the buffer length; this iterates through each byte until it finds the specified byte in the `al` register, or until `ecx` becomes `0`.

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. The `cmpsb` is normally used with `repe` (`repe cmpsb`) to compare two memory buffers; in this case, `ecx` will be set to the buffer length, and the comparison will continue until `ecx=0` or the buffers are not equal.

10. 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. To understand the concept, take a look at the following C program. The `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;
}
```

In order to understand how the members of the structures are accessed, let's look at the disassembled output of the `update` function. At ❸, the base address of the structure is moved into the `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:

```
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
```

From the preceding example, it can be seen that each member of the structure has its own *offset* and is accessed by adding the *constant offset* to the *base address*; so, the general form can be written as follows:

```
[base_address + constant_offset]
```

Structures may look very similar to arrays in the memory, but you need to remember a few points to distinguish between them:

- Array elements always have the same data types, whereas structures need not have the same data types.
- Array elements are mostly accessed by a variable offset from the base address (such as `[eax + ebx]` or `[eax+ebx*4]`), whereas structures are mostly accessed using constant offsets from the base address (for example, `[eax+4]`).

11. x64 Architecture

Once you understand the concepts of x86 architecture, it's much easier to understand x64 architecture. The x64 architecture was designed as an extension to x86 and has a strong resemblance with x86 instruction sets, but there are a few differences that you need to be aware of from a code analysis perspective. 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`. As you might expect, a program can access the register as 64-bit (`RAX`, `RBX`, and so on), 32-bit (`eax`, `ebx`, etc), 16-bit (`ax`, `bx`, and so on), or 8-bit (`al`, `bl`, and so on). For example, you can access the lower half of the `RAX` register as `EAX` and the lowest word as `AX`. You can access the registers `r8-r15` as `byte`, `word`, `dword`, or `qword` by appending `b, w, d` or `q` to the register name.
- 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*.
- Another major difference is that in the x86 architecture, the function parameters are pushed onto the stack as mentioned previously, whereas 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. Let's look at an example of simple C code (the `printf` function); this function takes six parameters:

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

The following is the disassembly of the C code compiled for a 32-bit (x86) processor; in this case, all of the parameters are pushed onto the stack (in reverse order), and after the call to `printf`, `add esp,18h` is used to clean up the stack. So, it is easy to tell that the `printf` function takes six parameters:

```
push 5
push 4
push 3
push 2
push 1
push offset Format ; "%d %d %d %d %d"
call ds:printf
add esp, 18h
```

The following is the disassembly of the C code compiled for a 64-bit (x64) processor. The first instruction, at ❶, allocates `0x38` (`56` bytes) of space 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 fifth and the sixth parameters are stored on the stack (in the allocated space), using instructions at ❻ and ❼. The `push` instruction was not used in this case, making it difficult to determine if the memory address is a *local variable* or a *parameter* to the function. In this case, the format string helps to determine the number of parameters passed to the `printf` function, but in other cases, it's not that easy:

```
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
```

Intel 64 (x64) and IA-32 (x86) architecture consist of many instructions. If you come across an assembly instruction that is not covered in this chapter, you can download the latest Intel architecture manuals from `https://software.intel.com/en-us/articles/intel-sdm`, and the instruction set reference (*volumes 2A, 2B, 2C, and 2D*) can be downloaded from `https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf`.

11.1 Analyzing 32-bit Executable On 64-bit Windows

The 64-bit Windows operating system can run a 32-bit executable; to do that, Windows developed a subsystem called *WOW64* (Windows 32-bit on Windows 64-bit). WOW64 subsystem allows for the execution of 32-bit binaries on 64-bit Windows. When you run an executable, it needs to load the DLLs to call the API functions to interact with the system. The 32-bit executable cannot load 64-bit DLLs (and a 64-bit process cannot load 32-bit DLLs), so Microsoft separated the DLLs for both 32-bit and 64-bit. The 64-bit binaries are stored in the `\Windows\system32` directory, and the 32-bit binaries are stored in the `\Windows\Syswow64` directory.

The 32-bit applications, when running under 64-bit Windows (Wow64), can behave differently, as compared to how they behave on the native 32-bit Windows. When you are analyzing a 32-bit malware on 64-bit Windows, if you find malware accessing the `system32` directory, it is really accessing the `syswow64` directory (the operating system automatically redirects it to the `Syswow64` directory). If a 32-bit malware (when executed on 64-bit Windows) is writing a file in the `\Windows\system32` directory, then you need to check the file in the `\Windows\Syswow64` directory. Similarly, access to `%windir%\regedit.exe` is redirected to `%windir%\SysWOW64\regedit.exe`. The difference in behavior can create confusion during analysis, so it is essential to understand this difference, and to avoid confusion during analysis, it is better to analyze a 32-bit binary in a 32-bit Windows environment.

To get an idea of how WOW64 subsystem can impact your analysis, refer to *The WOW-Effect* by Christian Wojner (`http://www.cert.at/static/downloads/papers/cert.at-the_wow_effect.pdf`)

12. Additional Resources

The following are some of the additional resources to help you gain a deeper understanding of C programming, x86, and x64 assembly language programming:

- *Learn C:* `https://www.programiz.com/c-programming`
- *C Programming Absolute Beginner's Guide* by Greg Perry and Dean Miller
- *x86 Assembly Programming Tutorial*: `https://www.tutorialspoint.com/assembly_programming/`
- Dr. Paul Carter's *PC Assembly Language*: `http://pacman128.github.io/pcasm/`
- *Introductory Intel x86 - Architecture, Assembly, Applications, and Alliteration*: `http://opensecuritytraining.info/IntroX86.html`
- *Assembly language Step by Step* by Jeff Duntemann
- *Introduction to 64-bit Windows Assembly Programming* by Ray Seyfarth
- *x86 Disassembly*: `https://en.wikibooks.org/wiki/X86_Disassembly`

Summary

In this chapter, you learned the concepts and techniques required to understand and interpret assembly code. This chapter also highlighted the key differences between the x32 and x64 architectures. The disassembly and decompiling (static code analysis) skills that you learned in this chapter will help you to gain a deeper understanding of how malicious code works, at a low level. In the next chapter, we will look at code analysis tools (disassemblers and debuggers), and you will learn how the various features offered by these tools can ease your analysis and help you inspect the code associated with the malicious binary.