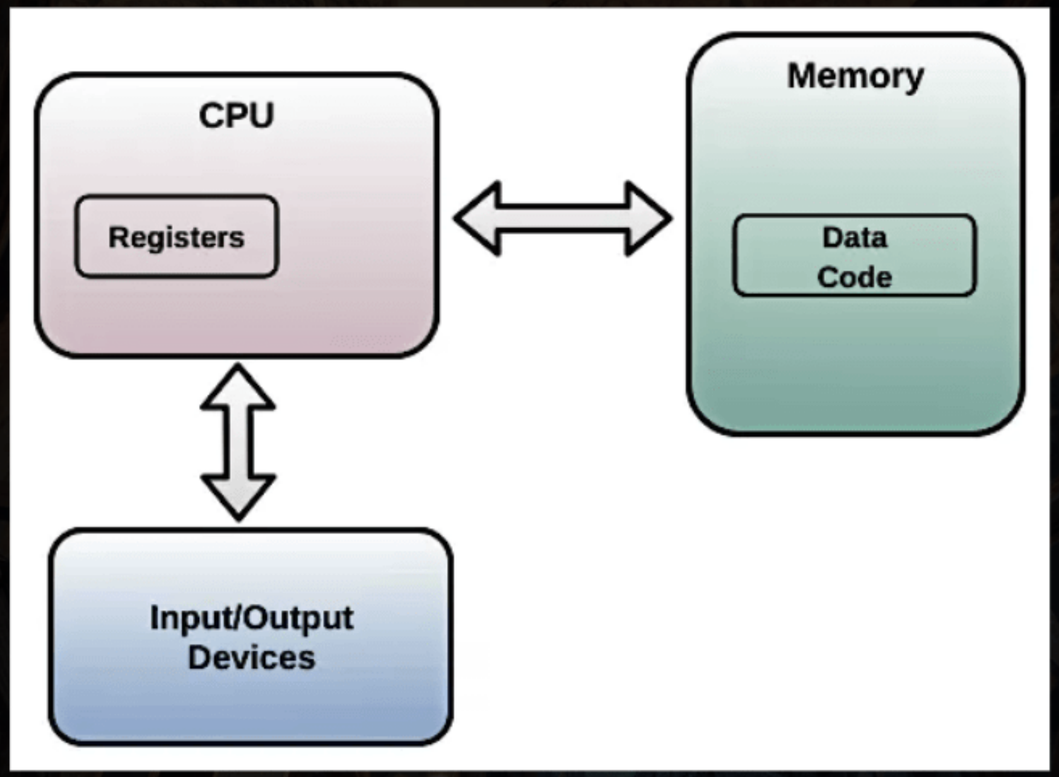
Assembly & Disassembly Primer

Part 1

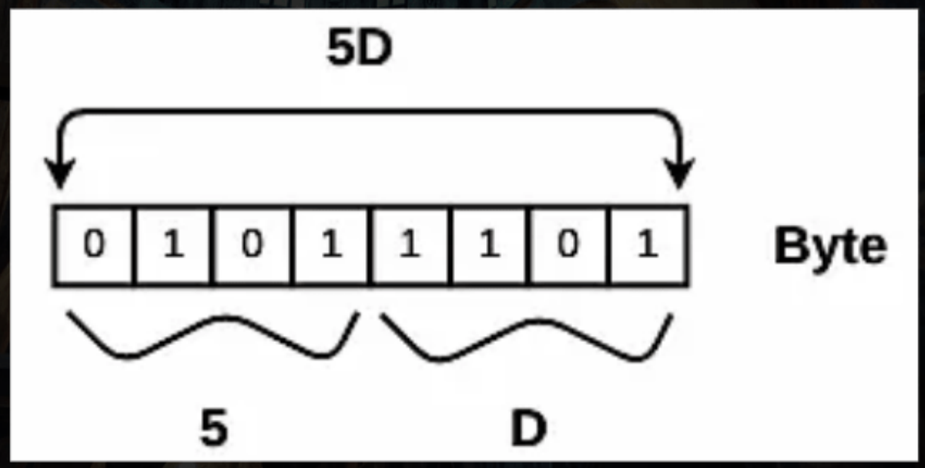
Computer Basics

Computer Components



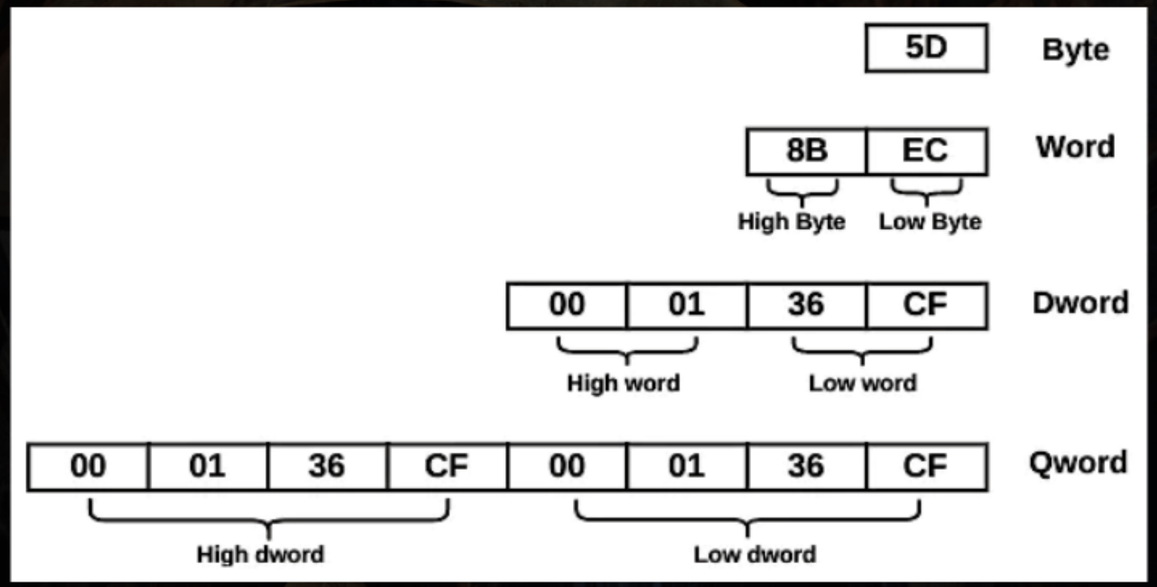
Fundamental Data Types (Bits and Bytes)

- 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
- 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.



Word, Dword & Qword

A *word* 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:

- ***Example 1:*** **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**.

- ***Example 2:*** The sequence of two bytes **8B EC** (word) can represent ***short int 35820*** or a machine instruction, ***mov ebp,esp***

- ***Example 3:*** *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

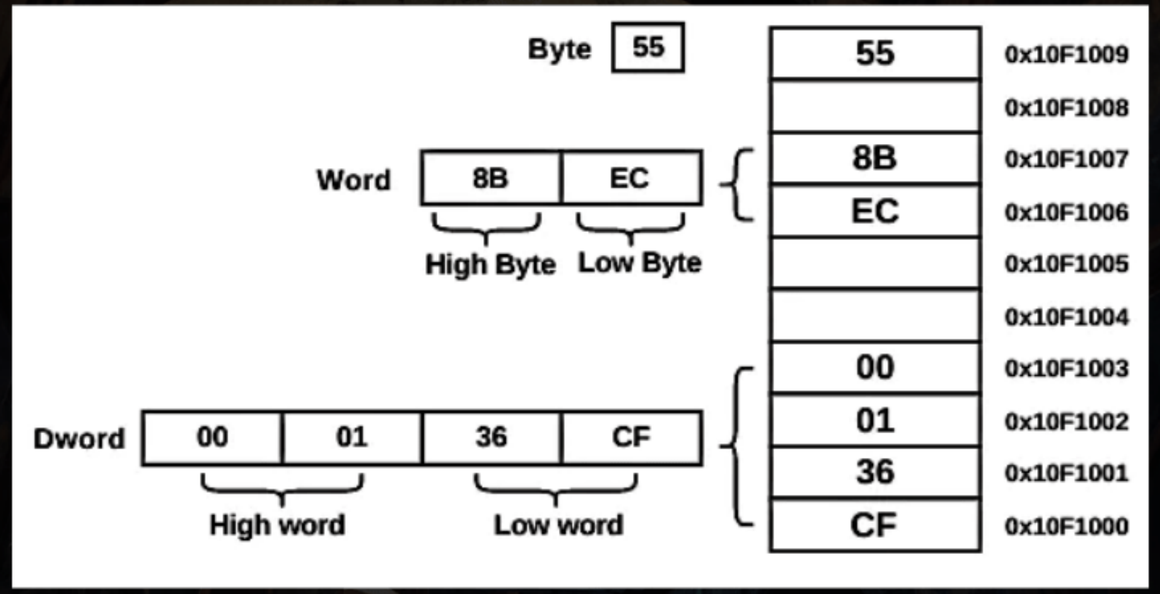
Memory

- 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.

Address	Data in Memory
0x10F1009	45
0x10F1008	FC
0x10F1007	00
0x10F1006	30
0x10F1005	0F
0x10F1004	01
0x10F1003	51
0x10F1002	8B
0x10F1001	EC
0x10F1000	55

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.



Central Processing Unit (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.

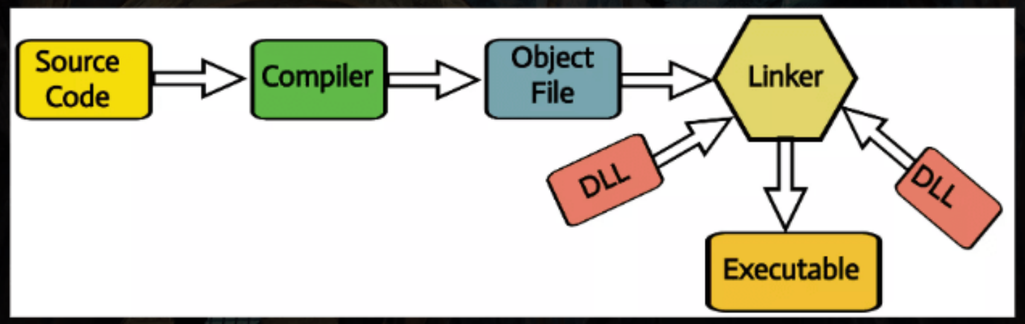
Program Basics

Program Compilation

- The source code is written in a high-level language, such as C or C++.

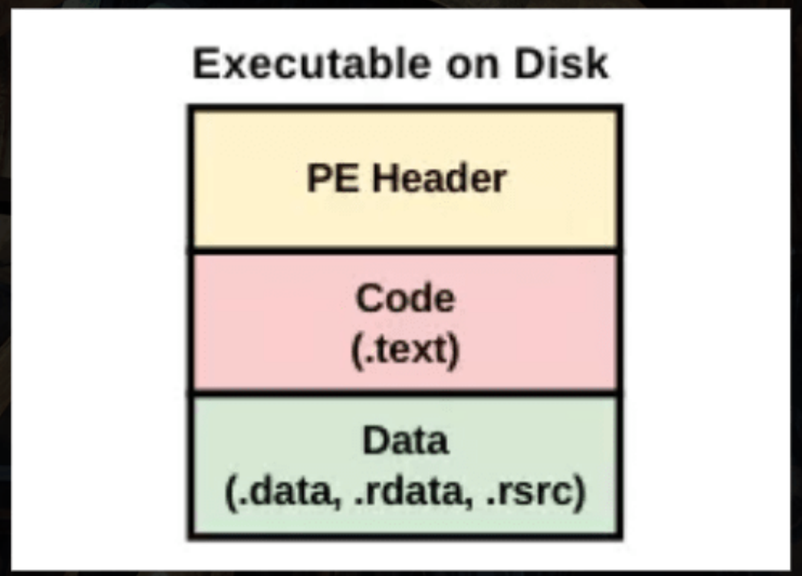
- The source code of the program is run through the compiler which translates the statements in a high-level language into an intermediate form called an *object file* or *machine code*, which is not human-readable and is meant for execution by the processor.

- The object code is then passed through the linker which links the object code with the required libraries (DLLs) to produce an executable.



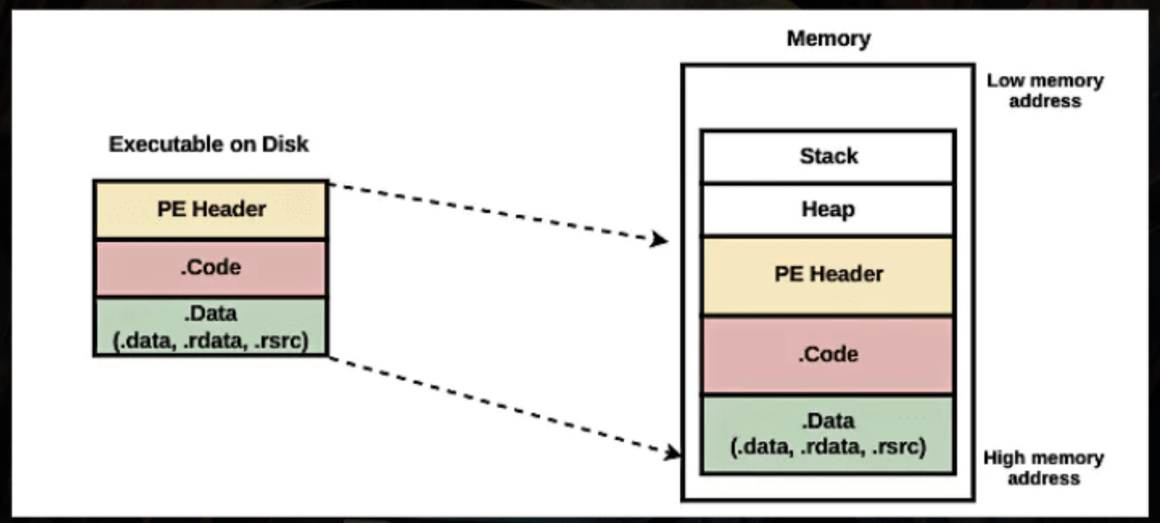
Program On Disk

When a program is compiled, the compiler segregates 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)



Program in Memory

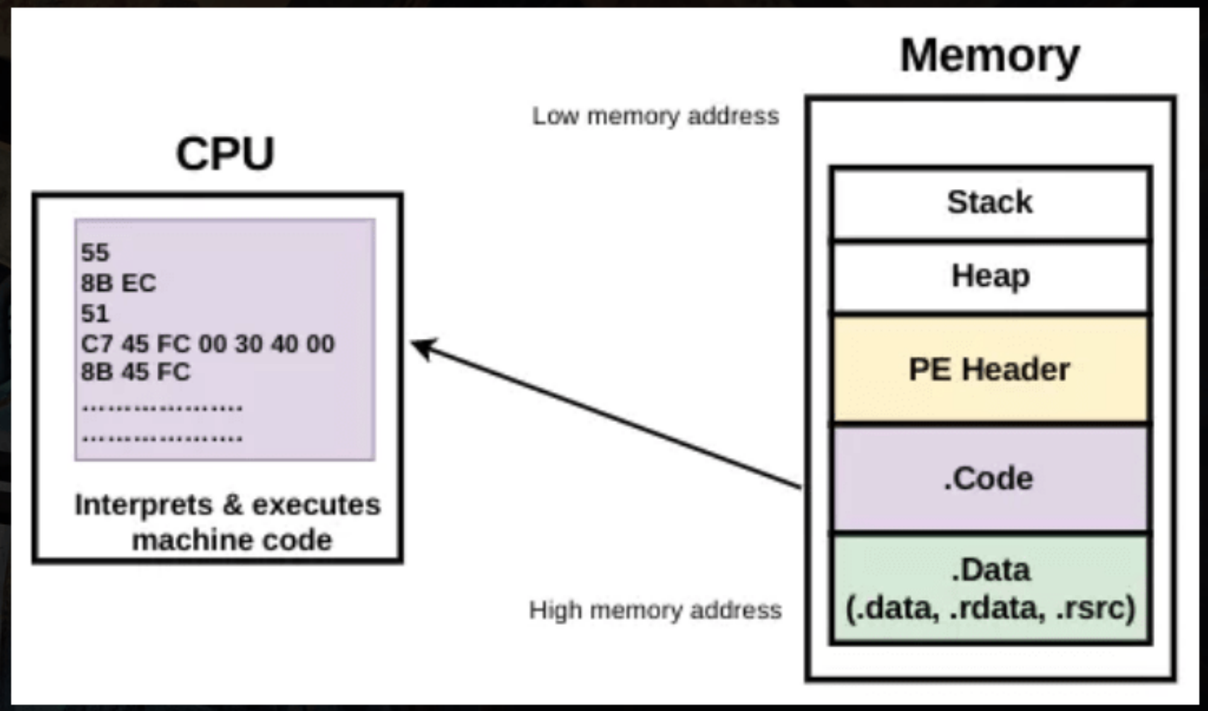
When the executable is double-clicked, a process memory is allocated by the operating system, and the executable is loaded into the allocated memory by the operating system loader. Notice that the structure of the executable on the disk and in the memory is similar.



Interaction Between the CPU and Memory

The following steps are performed when a program is executed:

- The program (which contains code and data) is loaded into the memory.
- The CPU fetches the machine instruction, decodes it, and executes it.
- The CPU fetches the required data from memory; the data can also be written to the memory.
- The CPU may interact with the input/output system, as necessary.



Live Demo

Program in Memory

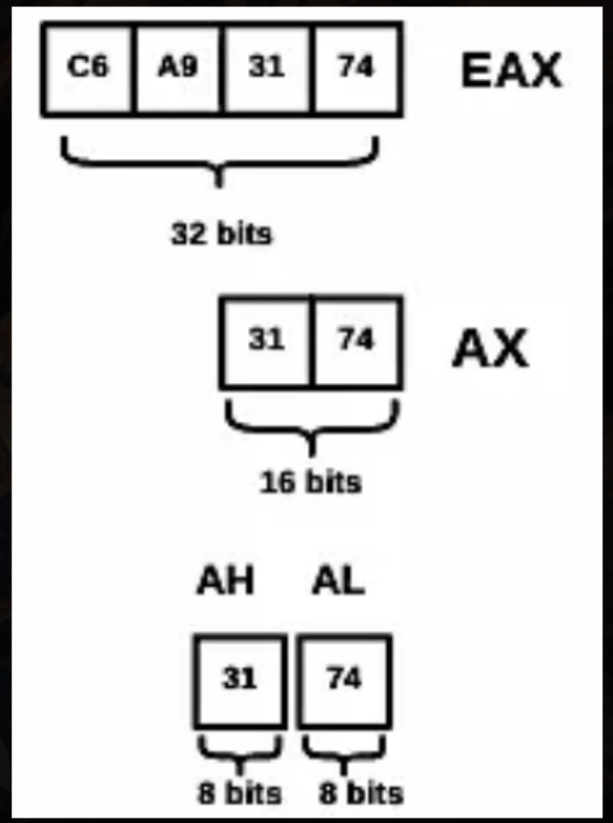
CPU Registers

General Purpose Registers

- The x86 CPU has eight general purpose 32-bit registers: *EAX, EBX, ECX, EDX, ESP, EBP, ESI,* and *EDI*

- *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*



Instruction Pointer (EIP) and EFLAGS

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

- 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.

Data Transfer Instructions

MOV Instruction

- One of the basic instructions in the assembly language is the *MOV* instruction
- This instruction moves data from one location to another (from source to destination)
- The general form of the mov instruction is shown as follows; this is similar to the assignment operation in a high-level language

```
mov dst,src        ; This is same as dst = src
```

Moving a Constant Into Register

The first variation of the *MOV* instruction is to move a constant (or immediate value) into a register

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

Moving Value 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 ebx,10        ;   ebx = 10
mov eax,ebx       ;   eax = ebx or eax = 10
```

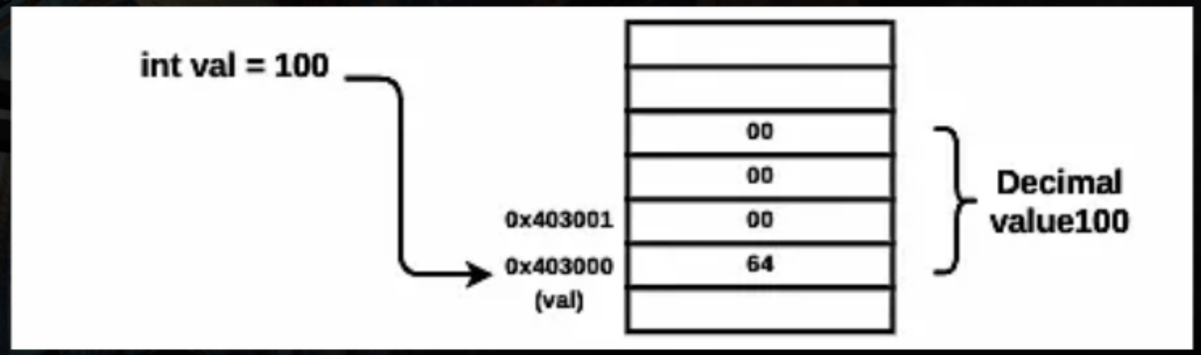
What Happens When You Define a Variable

```
int val = 100;
```

- An integer is 4 bytes in length, so 100 is stored as a sequence of 4 bytes (00 00 00 64) in the memory.

- The sequence of four bytes is stored in the *little-endian* format.

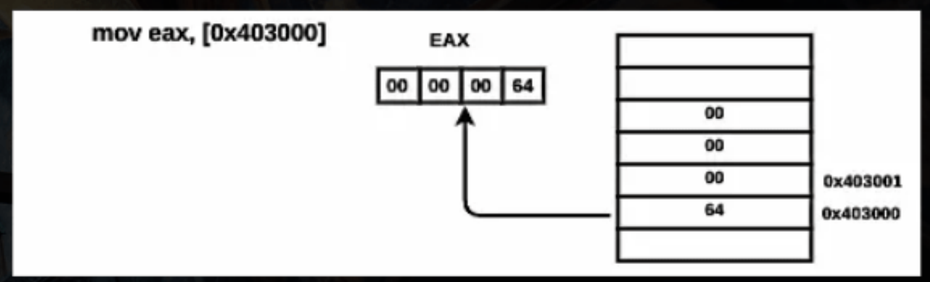
- The integer 100 is stored at some memory address. The memory address is labeled as *val*



Moving Value From Memory to Register

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

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.



During reverse engineering, you will normally see instructions similar to the ones shown as follows.

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

All of the following 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
```

Sometimes you will come across the below instructions:

1. ***LEA*** instruction stands for *load effective address;* this instruction loads the address instead of the value.
2. The ***dword*** *ptr* just indicates that a 4-byte (dword) value is moved from the memory address specified by ***EBP-4*** into the ***EAX***

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

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

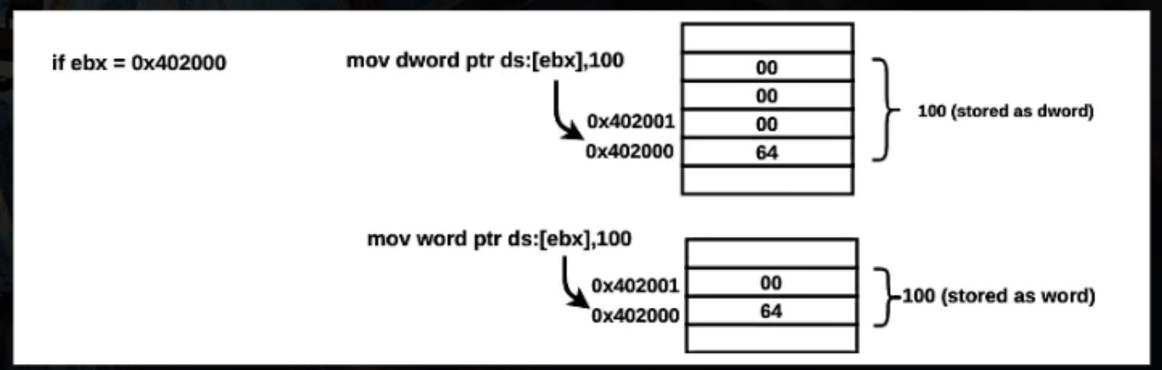
Moving Value from Register to Memory

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

```
mov [0x403000],eax   ; moves 4 byte from eax to memory location starting at 0x403000
mov [ebx],eax        ;  moves 4 byte value from eax to the memory address specified by ebx
```

Sometimes, you will come across instructions like the one shown. These instructions move constant values into a memory location; The ***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
mov dword ptr [ebx],100
mov word ptr [ebx], 100
```



LAB 8

Lab 8 - Disassembly Challenge

The following is a disassembled output of a simple C code snippet. Can you figure out what this code snippet does?. If possible translate it back to a pseudocode (high-level language equivalent). Use all of the concepts that you learned so far to solve the challenge.

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

Arithmetic Operations

Addition and Subtraction

The addition and subtraction are performed using the ***ADD*** and ***SUB*** instructions.

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

```
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
```

Increment and Decrement

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

- The multiplication is done using the ***MUL*** instruction.
- The ***MUL*** instruction takes only one operand; that operand is multiplied with the content of either the ***AL***, ***AX***, or ***EAX*** register.
- The result of the multiplication is stored in either the ***AX*** or ***DX:AX***, or ***EDX:EAX*** registers
- If the operand of the mul instruction is 8 bits (1 byte), then it is multiplied with the 8-bit ***AL*** register, and the product is stored in the ***AX*** register. If the operand is 16 bits (2 bytes), it is multiplied with the ***AX*** register, and the product is stored in the ***DX: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:EAX*** register

```
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

- The division is performed using the ***DIV*** instruction.
- It 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: 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***.

```
div ebx    ; divides the value in edx:eax by ebx
```

LAB 9

Lab 9 - Disassembly 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
```

Bitwise Operations

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, EAX would contain ***00 00 FF FF (00000000 00000000 11111111 11111111)***

```
not eax
```

AND, OR and XOR instruction

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

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 languages

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 CL: 0000 0100
```

Shift Left (SHL) and Shift Right (SHR)

- The ***SHR (shift right)*** & ***SHL (shift left)*** instructions take two operands (the destination and the count).

- Shift the bits in the destination to the right or left by the number of bits specified by the count operand.

In the below example, the content of the ***AL*** register ***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 (0x10 or 16)***

```
mov al,4      ; al = 4
shl al,2      ; al = al << 2
```

ROL & ROR instructions

- 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.

In the Following example, if ***AL*** contained ***0x44 (0100 0100)***, then the result of the ***ROL*** operation would be ***0x11 (0001 0001)***.

```
rol al,2
```

Branching & Conditionals

Branching Instructions

- A program will need to execute code at a different memory address (***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***.

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>
```

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 example, 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 ***0***:

```
cmp eax,5
```

- Another instruction that alters the flags without storing the result is the ***TEST*** instruction.

- The ***TEST*** instruction performs a bitwise ***AND*** operation and aplters the flags without storing the result.

- Both ***CMP*** and ***TEST*** instructions are normally used along with the conditional jump instruction for decision making.

In the following example, if the value of ***EAX = 0***, then the zero flag would be set (***ZF=1***), because when you ***AND 0*** with ***0*** you get ***0***

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

The general format of conditional jump is shown below:

```
JCC <address>
```

In the preceding format, The CC represents conditions. These conditions are evaluated based on the bits in the EFLAGS register.

The 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	