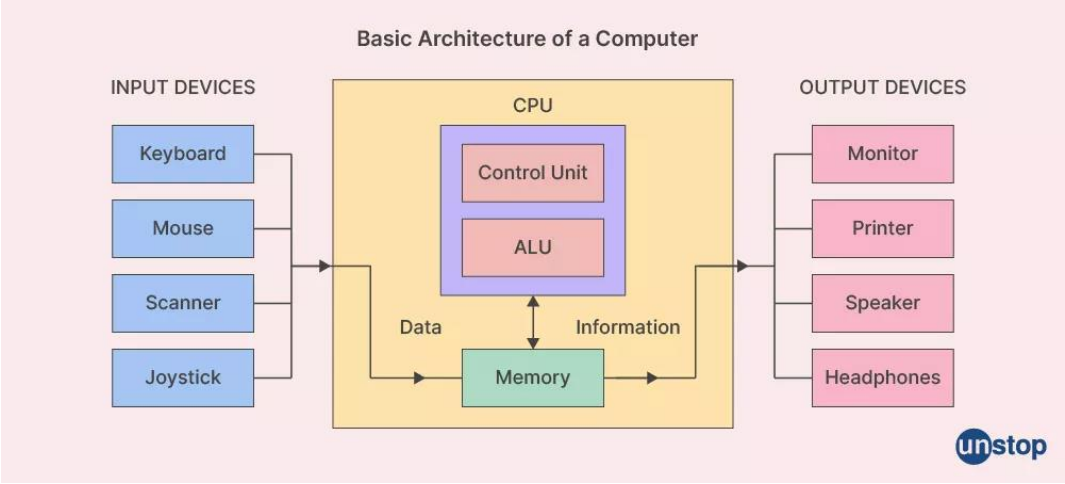
Module 1: Assembly Introduction

What is Assembly Language?

Assembly language is defined as a low-level programming language specific to a particular computer architecture. It serves as a bridge between the machine code that computers understand and high-level programming languages that humans use.



Basic Architecture of a Computer

Unlike high-level programming languages, which are designed to be portable across various systems, assembly language is tailored for a specific type of processor or architecture. Each family of processors has its own set of assembly instructions.

Assembly language instructions are symbolic representations of machine code, making it more readable for humans while maintaining a direct correspondence to the machine-level instructions.

```cpp
// C++ program to display "Hello World"

// Header file for input output functions
#include <iostream>
using namespace std;

// Main() function: where the execution of
// program begins
int main()
{
    // Prints hello world
    cout << "Hello World";

    return 0;
}
```

Output

```
Hello World
```

C++ Hello World Program (High Level PL)

```
section .data

    msg db  'Hello, world!',0xa
    len equ $ - msg

section .text
    global _start

_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg
    mov edx, len

    int 0x80
    mov eax, 1
    int 0x80
```

```
Hello, world!
```

Assembly language Hello World Program

Advantages of Assembly Language:

Efficient Use of Resources: Programs written in assembly language are typically more efficient in terms of memory usage and execution speed compared to programs written in high-level languages.

Greater Control: Assembly language provides programmers with granular control over hardware and memory, allowing for the precise management of system resources.

Suitable for Time-Critical Applications: Because of its low-level nature and efficiency, assembly language is ideal for applications that require fast and predictable response times, such as embedded systems, device drivers, and real-time computing tasks.

Understanding Hardware Interactions: Learning assembly language helps programmers understand how software interacts with the operating system, processor, and BIOS. This knowledge is crucial for tasks like optimizing performance, debugging at a low level, or developing system-level applications.

Dis-advantages of Assembly Language:

- It takes a lot of time and effort to write the code for the same.
- It is very complex and difficult to understand.
- The syntax is difficult to remember.
- It lacks portability of programs between different computer architectures as it is very much machine architecture dependent.
- It needs more size or memory of the computer to run the long programs written in Assembly Language.

Basic Features of PC Hardware:

It explains that the CPU executes instructions and manipulates data stored in memory, using registers as temporary storage locations.

The concept of a “bit” as the fundamental unit of computer storage is introduced. A “bit” can be in one of two states: on (1) or off (0). A group of eight bits forms a byte, which is a common unit of data in computing.

The processor supports the following data sizes:

- Word: a 2-byte data item
- Doubleword: a 4-byte (32 bit) data item
- Quadword: an 8-byte (64 bit) data item
- Paragraph: a 16-byte (128 bit) area
- Kilobyte: 1024 bytes
- Megabyte: 1,048,576 bytes

Number Systems:

Binary Number System: This section explains the binary (base-2) number system, which uses only two digits, 0 and 1. Binary numbers are the most fundamental form of data representation in computers because they correspond directly to the on/off states of transistors in a computer’s processor and memory.

Positional Value: In binary, each bit position represents a power of 2. For example, in the binary number 1011, the rightmost bit represents 2^0, the next bit represents 2^1, then 2^2, and so on.

Binary Arithmetic: Operations such as addition, subtraction, multiplication, and division follow specific rules in binary, like their decimal counterparts but using only two digits (0 and 1).

Binary to Decimal Conversion:

To convert a binary number to decimal, sum the products of each binary digit (bit) and its corresponding power of 2.

Example: Convert 1101 to decimal.

Calculation: 11 x 2^3 + 1 x 2^2 + 0 x 2^1 + 1 x 2^0

= 8 + 4 + 0 + 1

Decimal equivalent: 13

Decimal to Binary Conversion:

To convert a decimal number to binary, repeatedly divide the number by 2 and record the remainder until the quotient is 0. The binary number is the sequence of remainders read from bottom to top.

Example: Convert 13 to binary.

Calculation:

13 / 2 = 6 remainder 1

6 / 2 = 3 remainder 0

3 / 2 = 1 remainder 1

1 / 2 = 0 remainder 1

Binary equivalent: 1101

Binary Addition:

- Binary addition follows these rules:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0 (with a carry of 1)

Example: Add 1011 and 1101.

1011

+ 1101

11000 (carry over occurs in the second column from the right)

Two's Complement for Negative Numbers:

Two's complement is a method for representing negative numbers in binary. To find the two's complement of a binary number, invert all the bits and add 1.

Example: Find the two's complement of 0101 (which is 5 in decimal).

Invert all bits: 1010

Add 1: 1010 + 1 = 1011

Two's complement result: 1011 (which represents -5 in an 8-bit signed integer)

Hexadecimal Number System:

The hexadecimal (base-16) number system is also introduced. Hexadecimal numbers use digits 0-9 and letters A-F to represent values from 0 to 15. Hexadecimal is often used in computing as a more human-readable representation of binary data. For example, a byte can be represented by two hexadecimal digits rather than eight binary digits.

Decimal number	Binary representation	Hexadecimal representation
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Hexadecimal to Decimal Conversion:

To convert a hexadecimal number to decimal, multiply each digit by its corresponding power of 16 and sum the results.

Example: Convert 1A3 to decimal.

Calculation: 1 x16^2 + 10 x 16^1 + 3 x 16^0 = 256 + 160 + 3 = 419

Decimal equivalent: 419

Decimal to Hexadecimal Conversion:

To convert a decimal number to hexadecimal, repeatedly divide the number by 16 and record the remainder until the quotient is 0. The hexadecimal number is the sequence of remainders read from bottom to top.

Example: Convert 419 to hexadecimal.

Calculation:

419 / 16 = 26 remainder 3

26 / 16 = 1 remainder 10 = A in hex

1 / 16 = 0 remainder 1

Hexadecimal equivalent: 1A3

Hexadecimal to Binary Conversion:

Each hex digit corresponds to a 4-bit binary equivalent.

Example: Convert 2F to binary.

Calculation:

2 = 0010 in binary

F = 1111 in binary

Binary equivalent: 0010 1111

Binary to Hexadecimal Conversion:

Group binary digits in sets of four (starting from the right) and convert each set to its hexadecimal equivalent.

Example: Convert 11010111 to hexadecimal.

Group: 1101 and 0111

Calculation:

1101 = D

0111 = 7

Hexadecimal equivalent: D7

Addressing Data in Memory

"Addressing Data in Memory" is introduced as a fundamental concept for understanding how the processor accesses and manipulates data stored in a computer's memory. This concept is crucial for assembly language programming because it directly impacts how instructions are written and executed.

Key Concepts of Addressing Data in Memory:

Memory and Its Organization:

A computer's memory is a collection of storage locations, each with a unique address. Memory can store instructions, data, and variables that the CPU processes.

The memory model can be segmented or flat. In a segmented model, the memory is divided into different segments, each of which has a specific purpose (e.g., code, data, stack). This model is typical in x86 architecture, where the processor uses segment registers to access different parts of the memory.

Memory Addressing Modes:

Addressing mode refers to the way in which the operand of an instruction is specified. In assembly language, different addressing modes allow the programmer to access data stored in various locations (e.g., registers, memory, immediate values).

Types of Memory Addresses:

Absolute Addressing (Direct Addressing): This mode directly references a specific memory address. The instruction specifies the actual memory address where the data is located.

Example: MOV AX, [1234H] moves the contents from the memory address 1234H directly into the AX register.

Segmented Addressing (Offset Addressing): Memory is divided into segments, and each segment has a base address. An offset value is added to this base address to locate the exact position of data within the segment.

Example: The address 0725H may be interpreted in a segmented model where 07 is the high-order byte (most significant byte) and 25 is the low-order byte (least significant byte).

The Fetch-Decode-Execute Cycle:

The process through which the CPU executes instructions is known as the fetch-decode-execute cycle (or simply the execution cycle). It involves three continuous steps:

Fetching: The CPU retrieves the instruction from memory using the instruction pointer (IP) and loads it into the instruction register (IR).

Decoding: The CPU decodes the instruction to determine what actions are required. It identifies the opcode (operation code) and the addressing mode.

Executing: The CPU performs the operation specified by the instruction. This may involve reading from or writing to memory, performing arithmetic operations, or controlling hardware devices.

Memory addressing plays a critical role in the fetch step, as the CPU needs to know where in memory to find the next instruction or the data it needs.

Byte Ordering in Memory:

Little-endian vs. Big-endian: In little-endian systems, the least significant byte is stored first (at the lowest memory address), and the most significant byte is stored last. This is typical in x86 architectures.

For example, if a 16-bit word 0725H is stored in memory at address 1000H, the processor will store 25H at 1000H and 07H at 1001H.

Types of Memory Access:

Read Access: The CPU reads data from memory. This is a common operation for fetching instructions or loading data into registers for computation.

Write Access: The CPU writes data to memory. This is used when storing results of computations or updating program variables.

Address Calculation: To access a specific memory location, the CPU calculates the effective address using the segment and offset values, or directly uses the absolute address in some addressing modes.

Direct Addressing Example:

MOV AX, [2000H]

This instruction moves the data stored at memory address 2000H directly into the AX register.

The address 2000H is an absolute address that refers to a specific location in memory.

Register Indirect Addressing Example:

MOV AX, [BX]

In this instruction, the register BX holds the memory address of the data to be moved into the AX register.

This is an example of indirect addressing, where the address of the operand is specified by a register.

Base plus Index Addressing Example:

MOV AX, [BX + SI]

This instruction uses both the BX based register and the SI index register to calculate the effective address.

The sum of the contents of BX and SI is used as the address from which the data is fetched and moved into AX.

Segmented Memory Model Example:

In a segmented model, you might have a data segment (DS) register pointing to the start of the data segment, and the offset is specified in the instruction:

MOV AX, DS: [1234H]

DS contains the base address of the data segment, and 1234H is the offset within that segment where the data resides. The CPU calculates the effective address by combining DS and 1234H.