**Assembly Language:**

* low-level programming language
* specific to a particular computer architecture
* converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.
* also called symbolic machine code
* close to processor
* great for speed optimization
* give complete control over the system’s resources

**Utility Program:**

Is system software that helps to maintain the proper and smooth functioning of a Computer System.

**Examples of Utiliy Program:**

antivirus software, file management tools, compression tools, disk management tools

**What is Assembly Language:**

Each personal computer has a **microprocessor** that manages the computer’s arithmetical, logical, and control activities.

**Machine Language Instructions:**

Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called **'machine language instructions'**

A **Processor** understands only machine language instructions, which are strings of 1’s and 0’s.

However, machine language is too obscure and complex for using in software development. So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

**Advantages of Assembly Language:**

* requires less memory and execution time
* allows hardware-specific complex jobs in an easier way
* It is suitable for time-critical jobs
* It is most suitable for writing interrupt service routines and other memory resident programs (e.g., Worms)

Having an understanding of assembly language makes one aware of –

* How programs interface with OS, processor, and BIOS (Basic Input Output System)
* How data is represented in memory and other external devices
* How the processor accesses and executes instruction
* How instructions access and process data
* How a program accesses external device

**Why assembly language is better?**

assembly language programs tend to be more efficient than programs written in other languages is because assembly language **forces the programmer to consider how the underlying hardware operates with each machine instruction they write**.

**Why assembly language is created?**

* Assembly language was created as an exact **shorthand** for machine level coding, so that you wouldn't have to count 0s and 1s all day.
* It works the same as machine level code: with instructions and operands.

**Uses:**

Today, assembly language is used primarily for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. Typical uses are device drivers, low-level embedded systems, and real-time systems.

**Real-Time Systems:**

* means the system that is subjected to real time
* i.e., response should be guaranteed within a specified timing constraint or system should meet the specified deadline
* E.g. Real-Time Monitors

**Basic Features of Pc Hardware:**

The main internal hardware of a PC consists of:

1. Processor
2. Memory
3. Registers

**Registers** are Processor components that hold data and address.

To execute a program, the system copies it from the external device into the internal memory. The processor executes the program instructions.

**Bit:**

The **fundamental unit** of computer storage is a bit; it could be ON (1) or OFF (0).

**Byte:**

A group of 8 related bits makes a byte.

**Parity Bit:**

the parity bit is used to make the number of bits in a byte odd.

**Parity Error:**

 If the parity is even, the system assumes that there had been a parity error (though rare), which might have been caused due to hardware fault or electrical disturbance.

**Data Sizes:**

The Processor supports the following data sizes:

* Word -> 2-byte data item
* Doubleword -> 4-byte (32 bit) data item
* Quadword -> 8-byte (64 bit) data item
* Paragraph ->16-byte (128 bit) data item
* Kilobyte -> 1024 bytes
* Megabytes -> 1,048,576 bytes

**Binary Number System:**

* Every number system uses positional notation, i.e., each position in which a digit is written has a different positional value.
* Each position is power of the base, which is 2 for binary number system, and these powers begin at 0 and increase by 1.

**Hexadecimal Number System:**

* Hexadecimal number system uses base 16
* The digits in this system range from 0 to 15
* By convention, the letters A through F is used to represent the hexadecimal digits corresponding to decimal values 10 through 15.
* Hexadecimal numbers in computing is used for abbreviating lengthy binary representations.
* hexadecimal number system represents a binary data by dividing each byte in half and expressing the value of each half-byte.

**Convert Binary Numbers into Hexadecimal Numbers:**

To convert a binary number to its hexadecimal equivalent, break it into groups of 4 consecutive groups each, starting from the right, and write those groups over the corresponding digits of the hexadecimal number.

**Convert Hexadecimal Number into Binary Numbers:**

To convert a hexadecimal number to binary, just write each hexadecimal digit into its 4-digit binary equivalent.

**Binary Arithmetic:**

The following table illustrates four simple rules for binary addition −

|  |  |  |  |
| --- | --- | --- | --- |
| **(i)** | **(ii)** | **(iii)** | **(iv)** |
|  |  |  | 1 |
| 0 | 1 | 1 | 1 |
| +0 | +0 | +1 | +1 |
| =0 | =1 | =10 | =11 |

**Two’s Complement Notation:**

* A negative binary value is expressed in two’s complement notation.
* According to this rule, to convert a binary number to its negative value is to reverse it’s bit values and add 1.

**Subtract one value from another:**

To subtract one value from another, *convert the number being subtracted to two's complement format and add the numbers*.

Overflow of the last 1 bit is lost.

|  |  |
| --- | --- |
| Number 53 | 00110101 |
| Number 42 | 00101010 |
| Reverse the bits of 42 | 11010101 |
| Add 1 | 00000001 |
| Number -42 | 11010110 |
| 53 - 42 = 11 | 00001011 |

**Addressing Data in Memory:**

**Fetch-Decode-Execute Cycle**

**Or**

**Execution Cycle:**

The process through which the processor controls the execution of instructions is referred as the **fetch-decode-execute cycle** or the **execution cycle**.

It consists of three continuous steps:

* Fetching the instruction from memory
* Decoding or identifying the instruction
* Executing the instruction

**The processor may access one or more bytes of memory at a time.**

A hexadecimal number 0725H. This number will require two bytes of memory. The high-order byte or most significant byte is 07 and the low-order byte is 25.

The processor stores data in **reverse-byte sequence**, i.e., a low-order byte is stored in a low memory address and a high-order byte in high memory address.

When the processor gets the numeric data from memory to register, it again reverses the bytes.

**Kinds Of Memory Addresses:**

* **Absolute Address ->** a direct reference of specific location
* **Segment Address (or offset) ->** starting address of a memory segment with the offset value

**Assembly language is dependent upon the instruction set and the architecture of the processor.**

**BASIC SYNTAX:**

An assembly program can be divided into three sections:

* The **data** sections
* The **bss** section
* The **text** section

**The data section:**

* used for declaring initialized data or constants
* This data does not change at runtime.
* You can declare various constant values, file names, or buffer size, etc., in this section.
* The syntax for declaring data section is –

section.data

The **bss section:**

* The **bss** section is used for declaring variables.
* The syntax for declaring bss section is –

section.bss

**The text section:**

* The **text** section is used for keeping the actual code.
* This section must begin with the declaration **global \_start**, which tells the kernel where the program execution begins.
* The syntax for declaring text section is –

section.text

global \_start

\_start:

**Comments:**

* Assembly language comment begins with a semicolon (;).
* It may contain any printable character including blank
* It can appear on a line by itself, like –

; This program displays a message on screen

or, on the same line along with an instruction, like –

add eax, ebx ; adds ebx to eax

**Assembly Language Statements:**

Assembly language programs consist of three types of statements –

* Executable instructions or instructions
* Assembler directives or pseudo-ops
* Macros

**Executable Instructions:**

* The **executable instructions** or simply **instructions** tell the processor what to do.
* Each instruction consists of an **operation code** (opcode).
* Each executable instruction generates one machine language instruction

**Assembler Directives** or **Pseudo-ops:**

* The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process.
* These are non-executable
* do not generate machine language instructions

**Macros:**

**Macros** are basically a text substitution mechanism.

**Syntax of Assembly Language Statements:**

Assembly language statements are entered one statement per line.

Each statement follows the following format:

[label] mnemonic [operands] [ ;comment ]

The fields in the square brackets are optional.

 A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands or the parameters of the command.

INC COUNT ; Increment the memory variable COUNT

MOV TOTAL, 48 ; Transfer the value 48 in the

; memory variable TOTAL

ADD AH, BH ; Add the content of the

; BH register into the AH register

AND MASK1, 128 ; Perform AND operation on the

; variable MASK1 and 128

ADD MARKS, 10 ; Add 10 to the variable MARKS

MOV AL, 10 ; Transfer the value 10 to the AL register

**Memory Segments:**

A **segmented memory model** divides the system memory into groups of independent segments referenced by pointers located in the segment registers.

Each segment is used to contain a specific type of data.

* **Data segment:**
  + It is represented by **.data** section and the **.bss**.
  + used to declare the memory region, where data elements are stored for the program
  + cannot be expanded after the data elements are declared
  + it remains static throughout the program.
  + .bss section is also a static memory section that contains buffers for data to be declared later in the program. This buffer memory is zero-filled.
* **Code Segment:**
  + It is represented by **.text** section.
  + This defines an area in memory that stores the instruction codes.
  + This is also a fixed area.
* **Stack:**
  + This segment contains data values passed to functions and procedures within the program.

**Registers:**

Processor operations mostly involve processing data. This data can be stored in memory and accessed from thereon.

**Reading data from and storing data into memory slows down the processor as it involves complicated processes of sending the data request across the control bus and into the memory storage unit and getting the data through the same channel.**

To speed up the processor operations, the processor includes some internal memory storage locations, called **registers**.

The registers store data elements for processing without having to access the memory. A limited number of registers are built into the processor chip.

**Processor Registers:**

There are **ten 32-bit** and **six 16-bit** processor registers in IA-32 architecture.

**Categories:**

The registers are grouped into three categories –

* **General Registers**
* **Control Registers**
* **Segment Registers**

The general registers are further divided into the following groups –

* **Data Registers**
* **Pointer Registers**
* **Index Registers**

Data Registers:

* Four 32-bit registers are used for arithmetic, logical, and other operations.
  + 1. EAX
    2. EBX
    3. ECX
    4. EDX
* Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
* Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.

Some of these data registers have specific use in arithmetic operations.

* **AX is the primary accumulator**

it is used in input/output and most arithmetic instructions.

* **BX is known as the base register**

it could be used in indexed addressing.

* **CX is known as the count register**

As the ECX, CX registers store the loop count in iterative operations.

* **DX is known as the data register**

It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.