nasm

ASSEMBLY

programming language

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com


https://www.facebook.com/tutorialspointindia


https://twitter.com/tutorialspoint

About the Tutorial

Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

Audience

This tutorial has been designed for those who want to learn the basics of assembly programming from scratch. This tutorial will give you enough understanding on assembly programming from where you can take yourself to higher levels of expertise.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages will help you in understanding the Assembly programming concepts and move fast on the learning track.

Copyright & Disclaimer

Table of Contents

1. ASSEMBLY – INTRODUCTION

What is Assembly Language?

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

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

Having an understanding of assembly language makes one aware of:

- How programs interface with OS, processor, and BIOS;
- 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 devices.

Other advantages of using assembly language are:

- It requires less memory and execution time;
- It 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.

Basic Features of PC Hardware

The main internal hardware of a PC consists of processor, memory, and 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.

The fundamental unit of computer storage is a bit; it could be ON (1) or OFF (0). A group of nine related bits makes a byte, out of which eight bits are used for data and the last one is used for parity. According to the rule of parity, the number of bits that are ON (1) in each byte should always be odd.

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

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

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.

The following table shows the positional values for an 8-bit binary number, where all bits are set ON.

Bit value	1	1	1	1	1	1	1	1
Position value as a power of base 2	128	64	32	16	8	4	2	1
Bit number	7	6	5	4	3	2	1	0

The value of a binary number is based on the presence of 1 bits and their positional value. So, the value of a given binary number is:

1 + 2 + 4 + 8 +16 + 32 + 64 + 128 = 255

which is same as 2^8 - 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. Basically, hexadecimal number system represents a binary data by dividing each byte in half and expressing the value of each half-byte. The following table provides the decimal, binary, and hexadecimal equivalents:

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

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.

Example: Binary number 1000 1100 1101 0001 is equivalent to hexadecimal - 8CD1

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

Example: Hexadecimal number FAD8 is equivalent to binary - 1111 1010 1101 1000

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

Rules (iii) and (iv) show a carry of a 1-bit into the next left position.

Example

Decimal	Binary
60	00111100
+42	00101010
102	01100110

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 its bit values and add 1*.

Example

Number 53	00110101
Reverse the bits	11001010
Add 1	1
Number -53	11001011

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

Example

Subtract 42 from 53.

Number 53	00110101
Number 42	00101010

Reverse the bits of 42	11010101
Add 1	1
Number -42	11010110
53 - 42 = 11	00001011

Overflow of the last 1 bit is lost.

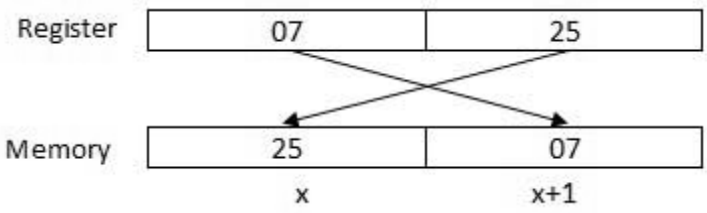
Addressing Data in Memory

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. Let us consider 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. So, if the processor brings the value 0725H from register to memory, it will transfer 25 first to the lower memory address and 07 to the next memory address.



x: memory address

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

2. ASSEMBLY – ENVIORNMENT SETUP

Try it Option Online

We already have set up NASM assembler to experiment with Assembly programming online, so that you can execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using our online compiler option available at http://www.compileonline.com/

```
section     .text
    global _start   ;must be declared for linker (ld)
_start:                 ;tells linker entry point
    mov     edx,len     ;message length
    mov     ecx,msg     ;message to write
    mov     ebx,1       ;file descriptor (stdout)
    mov     eax,4       ;system call number (sys_write)
    int     0x80        ;call kernel

    mov     eax,1       ;system call number (sys_exit)
    int     0x80        ;call kernel

section     .data
msg db 'Hello, world!', 0xa  ;our dear string
len equ $ - msg     ;length of our dear string
```

For most of the examples given in this tutorial, you will find a **Try it** option in our website code sections at the top right corner that will take you to the online compiler. So just make use of it and enjoy your learning.

Local Environment Setup

Assembly language is dependent upon the instruction set and the architecture of the processor. In this tutorial, we focus on Intel 32 processors like Pentium. To follow this tutorial, you will need:

- An IBM PC or any equivalent compatible computer

- A copy of Linux operating system
- A copy of NASM assembler program

There are many good assembler programs such as:

- Microsoft Assembler (MASM)
- Borland Turbo Assembler (TASM)
- The GNU assembler (GAS)

We will use the NASM assembler, as it is:

- Free. You can download it from various web sources.
- Well-documented and you will get lots of information on net.
- Could be used on both Linux and Windows.

Installing NASM

If you select "Development Tools" while installing Linux, you may get NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:

1. Open a Linux terminal.
2. Type **whereis nasm** and press ENTER.
3. If it is already installed, then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just *nasm:*, then you need to install NASM.

To install NASM, take the following steps:

1. Check **The netwide assembler (NASM)** website for the latest version.
2. Download the Linux source archive nasm-X.XX.ta.gz, where X.XX is the NASM version number in the archive.
3. Unpack the archive into a directory which creates a subdirectory nasm-X. XX.
4. cd to *nasm-X. XX* and type **./configure** . This shell script will find the best C compiler to use and set up Makefiles accordingly.
5. Type **make** to build the nasm and ndisasm binaries.
6. Type **make install** to install nasm and ndisasm in /usr/local/bin and to install the man pages.

This should install NASM on your system. Alternatively, you can use an RPM distribution for the Fedora Linux. This version is simpler to install, just double-click the RPM file.

3. ASSEMBLY – BASIC SYNTAX

An assembly program can be divided into three sections:

- The **data** section,
- The **bss** section, and
- The **text** section.

The data Section

The **data** section is 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, and
- Macros.

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.

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

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.

Following are some examples of typical assembly language statements:

```
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
```

The Hello World Program in Assembly

The following assembly language code displays the string 'Hello World' on the screen:

```
section     .text
    global _start   ;must be declared for linker (ld)
_start:                 ;tells linker entry point
    mov     edx,len     ;message length
    mov     ecx,msg     ;message to write
    mov     ebx,1       ;file descriptor (stdout)
    mov     eax,4       ;system call number (sys_write)
    int     0x80        ;call kernel

    mov     eax,1       ;system call number (sys_exit)
    int     0x80        ;call kernel

section     .data
msg db 'Hello, world!', 0xa  ;our dear string
len equ $ - msg              ;length of our dear string
```

When the above code is compiled and executed, it produces the following result:

```
Hello, world!
```

Compiling and Linking an Assembly Program in NASM

Make sure you have set the path of **nasm** and **ld** binaries in your PATH environment variable. Now, take the following steps for compiling and linking the above program:

1. Type the above code using a text editor and save it as hello.asm.
2. Make sure that you are in the same directory as where you saved **hello.asm**.
3. To assemble the program, type **nasm -f elf hello.asm**
4. If there is any error, you will be prompted about that at this stage. Otherwise, an object file of your program named **hello.o** will be created.
5. To link the object file and create an executable file named hello, type **ld -m elf_i386 -s -o hello hello.o**

6. Execute the program by typing **./hello**

If you have done everything correctly, it will display 'Hello, world!' on the screen.

4. ASSEMBLY – MEMORY SEGMENTS

We have already discussed the three sections of an assembly program. These sections represent various memory segments as well.

Interestingly, if you replace the section keyword with segment, you will get the same result. Try the following code:

```
segment  .text                   ;code segment
    global _start          ;must be declared for linker
_start:      ;tell linker entry point
     mov edx,len         ;message length
     mov ecx,msg       ;message to write
     mov ebx,1           ;file descriptor (stdout)
     mov eax,4           ;system call number (sys_write)
     int 0x80            ;call kernel

     mov eax,1           ;system call number (sys_exit)
     int 0x80            ;call kernel

segment .data                  ;data segment
msg  db 'Hello, world!',0xa   ;our dear string
len  equ   $ - msg             ;length of our dear string
```

When the above code is compiled and executed, it produces the following result:

```
Hello, world!
```

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. One segment is used to contain instruction codes, another segment stores the data elements, and a third segment keeps the program stack.

In the light of the above discussion, we can specify various memory segments as:

- **Data segment** - It is represented by **.data** section and the **.bss**. The .data section is used to declare the memory region, where data elements are stored for the program.

This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

The .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.

5. ASSEMBLY – REGISTERS

Processor operations mostly involve processing data. This data can be stored in memory and accessed from thereon. However, 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. The registers are grouped into three categories:

- General registers,
- Control registers, and
- Segment registers.

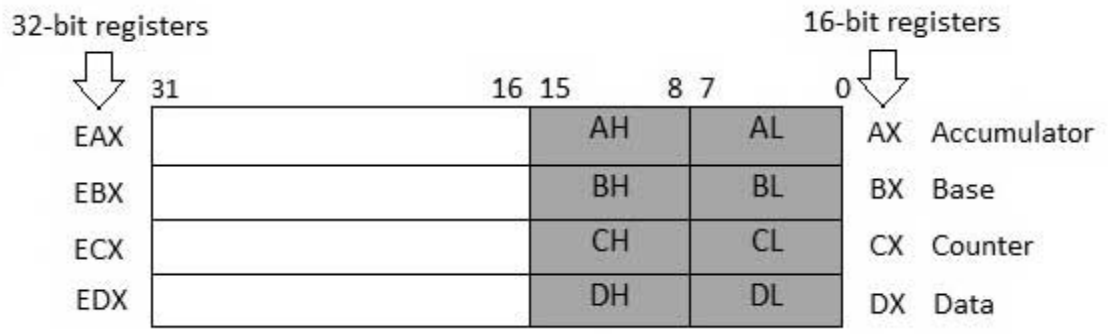
The general registers are further divided into the following groups:

- Data registers,
- Pointer registers, and
- Index registers.

Data Registers

Four 32-bit data registers are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways:

- As complete 32-bit data registers: EAX, EBX, ECX, 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 arithmetical operations.

AX is the primary accumulator; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

BX is known as the base register, as it could be used in indexed addressing.

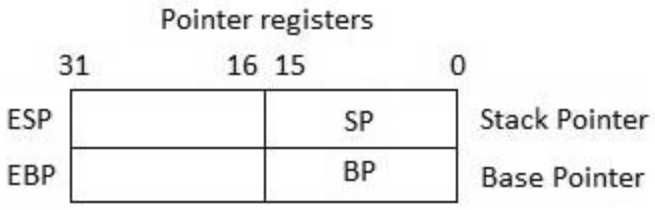
CX is known as the count register, as the ECX, CX registers store the loop count in iterative operations.

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.

Pointer Registers

The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers:

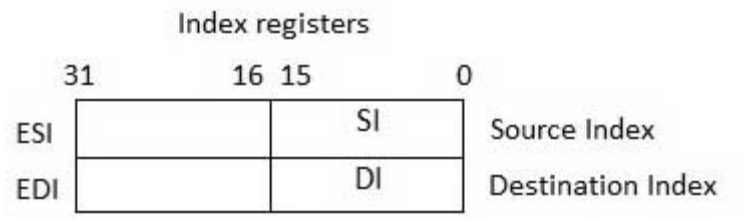
- **Instruction Pointer (IP)** - The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
- **Stack Pointer (SP)** - The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- **Base Pointer (BP)** - The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.



Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions, SI and DI, are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers:

- **Source Index (SI)** - It is used as source index for string operations.
- **Destination Index (DI)** - It is used as destination index for string operations.



Control Registers

The 32-bit instruction pointer register and the 32-bit flags register combined are considered as the control registers.

Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

The common flag bits are:

- **Overflow Flag (OF)**: It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
- **Direction Flag (DF)**: It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.
- **Interrupt Flag (IF)**: It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
- **Trap Flag (TF)**: It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.
- **Sign Flag (SF)**: It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.
- **Zero Flag (ZF)**: It indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.

- **Auxiliary Carry Flag (AF)**: It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
- **Parity Flag (PF)**: It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.
- **Carry Flag (CF)**: It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a *shift* or *rotate* operation.

The following table indicates the position of flag bits in the 16-bit Flags register:

Flag:					O	D	I	T	S	Z		A		P		C
Bit no:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Segment Registers

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments:

- **Code Segment**: It contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
- **Data Segment**: It contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
- **Stack Segment**: It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

Apart from the DS, CS and SS registers, there are other extra segment registers - ES (extra segment), FS and GS, which provide additional segments for storing data.

In assembly programming, a program needs to access the memory locations. All memory locations within a segment are relative to the starting address of the segment. A segment begins in an address evenly divisible by 16 or hexadecimal 10. So, the rightmost hex digit in all such memory addresses is 0, which is not generally stored in the segment registers.

The segment registers stores the starting addresses of a segment. To get the exact location of data or instruction within a segment, an offset value (or displacement) is required. To reference any memory location in a segment, the processor combines the segment address in the segment register with the offset value of the location.

Example:

Look at the following simple program to understand the use of registers in assembly programming. This program displays 9 stars on the screen along with a simple message:

End of ebook preview
If you liked what you saw…
Buy it from our store @ **https://store.tutorialspoint.com**