ALT-Assembly Language Tutorial

ASSEMBLY LANGUAGE TUTORIAL

Let's Learn in New Look

SHAIK BILAL AHMED

ABOUT THE TUTORIAL

Assembly Programming 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 software programmers with a need to understand the Assembly programming language starting from scratch. This tutorial will give you enough understanding on Assembly programming language from where you can take yourself at higher level 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 Notice

Table of Content

CHAPTER

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 like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction'.

Processor understands only machine language instructions which are strings of 1s and 0s. 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

An understanding of assembly language provides knowledge of:

- Interface of programs with OS, processor and BIOS;
- Representation of data in memory and other external devices;
- How processor accesses and executes instruction;
- How instructions accesses and process data;
- How a program access 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 the processor, memory and the registers. The 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. Eight bits are used for data and the last one is used for parity. According to the rule of parity, 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 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

The 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 the given binary number is: 1 + 2 + 4 + 8 +16 + 32 + 64 + 128 = 255, which is same as 2^8 - 1.

The Hexadecimal Number System

Hexadecimal number system uses base 16. The digits 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.

Main use of hexadecimal numbers in computing is 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) shows 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.

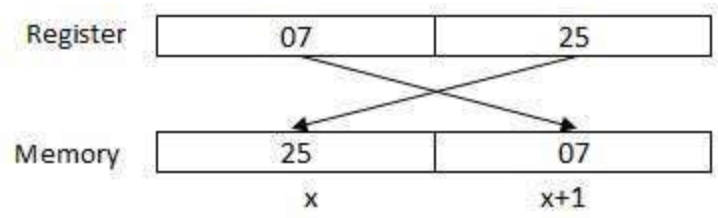
Addressing Data in Memory

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., the low-order byte is stored in low memory address and high-order byte in high memory address. So if 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:

- An absolute address - a direct reference of specific location.
- The segment address (or offset) - starting address of a memory segment with the offset value

CHAPTER

2

Assembly 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, like:

- 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 installed Linux, you may 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:

- Open a Linux terminal.
- Type ***whereis nasm*** and press ENTER.
- 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:

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

CHAPTER

3

Assembly Basic Syntax

An assembly program can be divided into three sections:

- The **data** section
- The **bss** section
- 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 main**, which tells the kernel where the program execution begins.

The syntax for declaring text section is:

```
section .text
   global main
main:
```

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

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 main      ;must be declared for linker (ld)
main:                ;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
```