Introduction to Assembly Language

Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture, unlike most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program called 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 of assembly programming that 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. Basic knowledge of any programming language will help you understand the Assembly programming concepts and move fast on the learning track.

What is an 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 the keyboard, displaying information on the screen, and performing different other jobs. This set of instructions is called 'machine language instructions.

A processor understands only machine language instructions, which are strings of 1's and 0's. However, machine language needs to be more obscure and complex for software development. So, the low-level assembly language is designed for a specific family of processors representing various symbolic code instructions 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 instructions;
- How instructions to 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 more easily;
- It is suitable for time-critical jobs;
- It is most suitable for writing interrupt service routines and other memory resident programs.

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

The fundamental unit of computer storage is a bit; it could be ON (1) or OFF (0), and a group of 8 related bits makes a byte on most modern computers.

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 the power of the base, which is 2 for the 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 bit and its positional value. So, the value of a given binary number is –

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

which is the same as 28 – 1.

Hexadecimal Number System

The hexadecimal number system uses base 16. The digits in this system range from 0 to 15. By convention, the letters A through F represent the hexadecimal digits corresponding to decimal values 10 through 15.

Hexadecimal numbers in computing are used for abbreviating lengthy binary representations. The hexadecimal number system represents 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, write each hexadecimal digit into its 4-digit binary equivalent.

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

Local Environment Setup

Assembly language depends upon the instruction set architecture and the process. In this tutorial, we focus on Intel-32 processors like Pentium. To follow this tutorial, you will need

- An Intel processor (not M1 or M2)
- A copy of the Linux operating system (explain further in the lab)
- A copy of the 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 (Netwide Assemble) assembler as it is -

- Free. You can download it from NASM.
- Well documented, and you will get lots of information on the net.
- It could be used on Linux, Windows, and OSX.

> ***Alternatively, you can use an online NASM assembler if you do not have an Intel processor or a laptop. There are several online NASM assemblers that you can use, such as online NASM assemblers. However, I am in the process of testing these compilers.***

Basic Syntax

An assembly program can be divided into three sections –

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

The **data section** is used for declaring initialized data or constants. This data does not change at runtime. In this section, you can declare values, file names, buffer sizes, and various constant values.

The syntax for declaring the data section is –

```
section.data
```

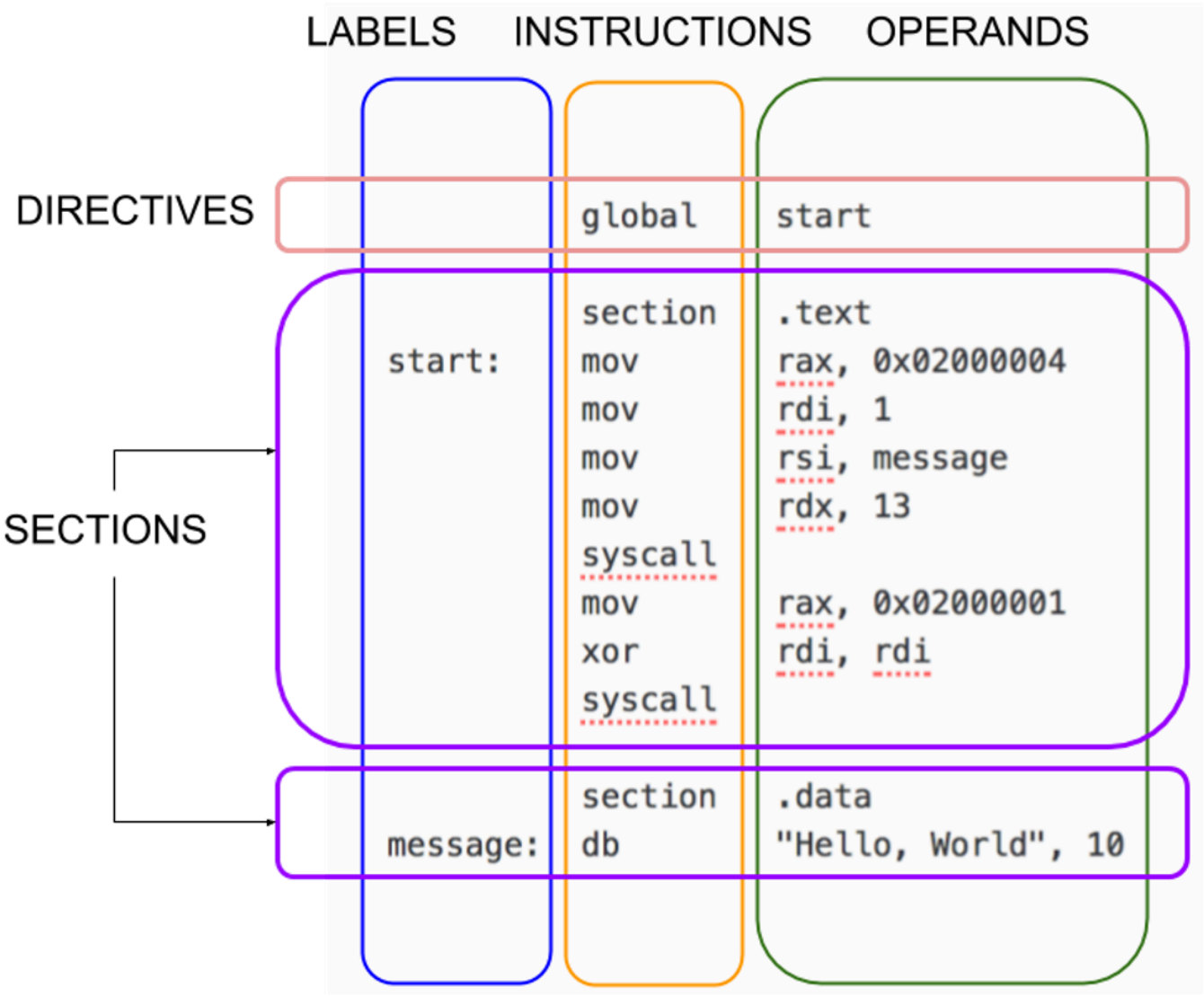
The bss section is used for declaring variables. The syntax for declaring bss section is

```
section.bss
```

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:
```



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 the 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 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. Unfortunately, these are non-executable and do not generate machine language instructions.

Macros are 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 is the name of the instruction (or the mnemonic), which is to be executed, and the second is the operands or the parameters of the command.

The Hello World program in assembly

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

```
1  section .text
2     global _start     ;must be declared for linker (ld)
3                       ;global is used to export the _start label.
4                       ;This will be the entry point to the program.
5
6
```

```
_start:             ;tells linker entry point
   mov  eax,4       ;system call number (sys_write)
   mov  ebx,1       ;file descriptor (stdout)
   mov  ecx,msg     ;message to write, A pointer to the variable 'msg'
   mov  edx,len     ;message length
   int  0x80        ;call kernel

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

section .data
msg db 'Hello, world!', 0xa   ;string to be printed
                              ;Declare a label "msg" which has
                              ;our string we want to print.
                              ;for reference: 0xa = "\n" (line feed)
                              ;db = define byte
len equ $ - msg     ;length of the string
                    ;len" will calculate the current
                    ;offset minus the "msg" offset.
                    ;this should give us the size of "msg".
                    ;len equals current offset - msg
```

Compiling and Linking an Assembly Program in NASM

> ***This section applies to students using NASM assemblers on an Intel-based processor and assembling code from scratch. If you are using an online NASM assembler, please skip this step._***

Ensure 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 -

- Please type the above code using a text editor and save it as `hello.asm`
- Ensure you are in the same directory where you saved `hello.asm`
- To assemble the program, type

```
nasm -f elf hello.asm
```

If there is any error, you will be prompted about that at this stage. Otherwise, an

- If there is any error, you will be prompted about that at this stage. Otherwise, an object file of your program is named `hello.o` will be created.
- To link the object file and create an executable file named hello, type

```
ld –m elf_i386 –s –o hello hello.o
```

- Execute the program by typing `./hello`

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

Making Syscalls on X86

Syscalls offer a method to invoke and use functionality in the operating system. For example, syscalls are launched on x86 Linux by invoking an interrupt instruction with the hex value of `0x80` (int 0x80). Before invoking this interrupt, however, you need to set up everything for the syscall. This is because the calling convention for syscalls differs on other architectures, including x86_64, which uses the SYSCALL instruction instead of INT 0x80.

> ***This tutorial is focused on 32-bit x86.***

After the syscall number is set and the registers are loaded with the parameters needed, calling the instruction INT 0x80 will execute the syscall. If the syscall returns a value, it will be placed in EAX. For example, opening a file should return a file descriptor or error code in EAX.

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 -

- Complete 32-bit data registers: EAX, EBX, ECX, EDX.
- The 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 abovementioned 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 a specific use in arithmetical operations

Some of these data registers have a specific use in arithmetical operations.

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

BX is the base register, which could be used in indexed addressing.

**CX is known as the count register; as the ECX, CX registers to 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 and DX for multiply and divide operations involving large values.

Code description

Line 7: We will first set EAX to the syscall number we want to invoke. As mentioned in our previous section, the syscall number on 32-bit x86 Linux for the write syscall was 4. Therefore, we use a MOV instruction (short for a move) to load the immediate value of 4 into the register EAX, containing the syscall number we want to run. The syntax for the move instruction in NASM is "MOV , ".

Lines 8-10: These instructions are set parameters for the write syscalls. The register EBX needs to contain the file descriptor integer that we want to write to, ECX needs to contain a pointer to the data buffer we want to write to the file descriptor, and EDX needs to contain the number of bytes from the data buffer we want to write out to the file descriptor. It's pretty straightforward using MOV instructions to do this.

Lines 13-14: Calling *sys_exit* at the end of all our programs will mean the kernel knows precisely when to terminate the process and return memory to the general pool, thus avoiding an error.

Lines 16+: We use the msg and len variables. These variables enable us to update the message, and the message length should also be automatically updated.

Last updated: Feb 2023 by Dr. Danish Khan.