Low Level Programming using Assembly Language

An assembly language program consists of statements. The syntax of an assembly language program statement which uses the following rules:

1. While writing assembly program one should take care to write one assembly statement/instruction is written per line (exception string repeat prefixes).
2. Each statement is either an instruction or an assembler directive.
3. Operands provide the information about the data to work with.

Assembly Language Statements: Assembly language programs consist of three types of statements –

- Executable instructions or 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 and do not generate machine language instructions.
- Macros: Macros are basically a text substitution mechanism.

We will be using the following format for an assembly language program statement

Label Name: mnemonic operand (destination), operand (source) ; comment

While writing an assembly program we will be using/diving the program in 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** (Block Started by Symbol) 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 system 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 is for computing factorial of given number

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

```
mov eax, ebx     ; moves ebx to eax
```

Some programmers may prefer to use an additional section (used in elf32) called rodata (read only data) in the program. This section contains all the values which are read only i.e. constants (Constant modifier) and strings.

Byte Order in Multi-Byte Entities (immediate values, address etc)

- Intel architecture is a little endian architecture. Least significant byte of multi-byte entity is stored at lowest memory address "**Little end goes first**".
- Some other systems use big endian. Most significant byte of multi-byte entity is stored at lowest memory address "**Big end goes first**".

Intel Registers: How to choose/use while writing IA-32 Assembly program

Most of the X-86 Intel family processors have 8 registers. These registers have 32 bits, 16-bit (are also accessible with a special one-byte instruction prefix) and 8-bit versions. Important to note here is that in 16-bit mode, the lower 16 bits are accessible by default, and the full registers are accessible only with a prefix byte.

32-Bit	16-Bit	08-Bit	
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		

EDI	**DI**		
ESP	**SP**		
EBP	**BP**		

The following is list containing the register names and their meanings:

- EAX - Accumulator Register
- EBX - Base Register
- ECX - Counter Register
- EDX - Data Register
- ESI - Source Index
- EDI - Destination Index
- EBP - Base Pointer (frame pointer)
- ESP - Stack Pointer

In addition to the full-sized general registers, the x-86 processor also has eight registers with 8 bit size. Important to note here is that these registers map directly into EAX, EBX, ECX, and EDX. From the design perspective of instruction set, the 8-bit registers are treated as separate entities.

EAX: The Accumulator

This register plays significant role in assembly language programming. Some instructions like mul and div requires one operand to be specifically in accumulator and result of multiplication (with assumption that result of multiplication is 32 bit) stores in accumulator. **EAX also contains the return value whenever an external function call is made. When used along with int 0x80 instruction; this instruction assumes that eax contains the System Call number.**

Few more instructions which move data in and out of the accumulator are LODS, STOS, IN, OUT, INS, OUTS, SCAS, and XLAT. The MOV instruction has a special one-byte opcode for moving data into the accumulator from a constant memory location.

In your code, try to perform as much work in the accumulator as possible. All other remaining general purpose registers apart from EAX are mostly used primarily to support the calculation occurring in the accumulator.

EDX: The Data Register

Another important register among the 8 GPR in Intel Architecture is the data register, EDX. This register is most closely tied to the accumulator. Most of the instructions which after execution deal with oversized data items, such as multiplication, division etc store the most significant bits in the data register and the least significant bits in the accumulator. One can say that EDX register is the 64-bit extension of the accumulator.

ECX: The Count Register

The count register, ECX, is the x86 equivalent of the variable I which most of us have used as loop controlling variable in programming languages like 'C'. Almost all counter-related instruction in the IA-32 instruction set use by default ECX. The instructions like LOOP, LOOPZ, LOOPNZ, JCXZ, consider ECX to hold the value. ECX register controls the string instructions through the repeat prefixes like REP, REPE, and REPNE. Most of these instructions use the count register to count downward instead of upward (as in C language loop where some times we use ++ operator for loop control variable).

EDI: The Destination Index

Every loop that generates data must store the result in memory, and doing so requires a moving pointer. The destination index, EDI, is that pointer. The destination index holds the implied write address of all string operations.

(Important to note here is that, many programmers treat EDI as no more than extra storage space. This is a mistake. All routines must store data, and some register must serve as the storage pointer. Since the destination index is designed for this job, using it for extra storage is a waste.)

ESI: The Source Index

The source index, ESI, has the same properties as the destination index. The only difference is that the *source index is for reading instead of writing*. In situations where your code does not read any sort of data, of course, using the source index for convenient storage space is acceptable.

ESP and EBP: The Stack Pointer and the Base Pointer

Of the eight general purpose registers, only the stack pointer, ESP, and the base pointer, EBP, are widely used for their original purpose. These two registers play a very important role and treated as the heart of the IA-32 function-call processing. When a block of code calls a function, it pushes the parameters and the return address on the stack. Once inside, function sets the base pointer equal to the stack pointer and then places its own internal variables on the stack. From that point on, the function refers to its parameters and variables relative to the base pointer rather than the stack pointer.

In your code, there is never a reason to use the stack pointer for anything other than the stack. The base pointer, however, is available for use. If your routines pass parameters by register instead of by stack (they should), there is no reason to copy the stack pointer into the base pointer. The base pointer becomes a free register for whatever you need.

EBX: The Base Register

In 16-bit mode, the base register, EBX, acts as a general-purpose pointer. Besides the specialized ESI, EDI, and EBP registers, it is the only general-purpose register that can appear in a square-bracket memory access (For example, MOV [BX], AX). In the 32-bit world, however, any register may serve as a memory offset, so the base register is no longer special. So, among all of the general-purpose registers, EBX is the only register without an important dedicated purpose. It is a good place to store an extra pointer or calculation step

Let us summarize the above details

- **EAX** - All major calculations take place in EAX, making it similar to a dedicated accumulator register.

- **EDX** - The data register is an extension to the accumulator. It is most useful for storing data related to the accumulator's current calculation.
- **ECX** - Like the variable i in high-level languages, the count register is the universal loop counter.
- **EDI** - Every loop must store its result somewhere, and the destination index points to that place. With a single-byte STOS instruction to write data out of the accumulator, this register makes data operations much more size-efficient.
- **ESI** - In loops that process data, the source index holds the location of the input data stream. Like the destination index, EDI has a convenient one-byte instruction for loading data out of memory into the accumulator.
- **ESP** - ESP is the sacred stack pointer. With the important PUSH, POP, CALL, and RET instructions requiring it's value; you have to always make sure to avoid using the stack pointer for anything else.
- **EBP** - In functions that store parameters or variables on the stack, the base pointer holds the location of the current stack frame. In other situations, however, EBP is a free data-storage register.
- **EBX** – EBX is completely free for extra storage space.

Data Access Methods :

Immediate addressing: data stored in the instruction itself

e.g. mov ecx,10

Register addressing: data stored in a register e.g. mov eax,ecx

Direct addressing: address stored in instruction e.g. mov ecx,var

Indirect addressing: address stored in a register e.g. mov ecx, [eax]

Base pointer addressing: includes an offset. e.g. mov ecx,[eax+4]

Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8) e.g. mov ecx, [eax + edi * 2]

Compiling/Assembling and Linking an Assembly Program

We will be using NASM assembler throughout this course. You have to type the assembly program using any text editor (like vi, emacs, gedit) and save it as programname.asm. One has to follow steps for creating an executable code.

- To assemble the program, type **nasm -felf32 programname.asm**
- If there is any error, you will be shown the errors. Otherwise, an object file of your program named programname.o will be created.
- To link the object file and create an executable file named hello, type **gcc –m32 programname.o** This will generate a.out file which is elf (executable and linkable file) file
- To execute the program by typing **./a.out**
- To generate a code so as to debug the same use following method/steps

  1. **nasm -felf32 –g –Fdwarf programname.asm**
  2. **gcc –m32 –g –Fdwarf programname.o**

Sample program: 01 (First.asm)

```
section .data
   FirstNum dd 10
   SecondNum dd 20
section .rodata
   msg db 'Result is %d',10,0
section .bss
   Result resd 1
section .text
   global main
   extern printf
main:
   mov eax, dword[FirstNum]
   mov ebx, dword[SecondNum]
   add eax,ebx
   mov dword[Result], eax
   push dword[Result]
   push msg
   call printf
   add esp,8
```

Here you may see that we have written global main, it is mandatory here as we will be using gcc (ld) to create the executable and moreover to communicate the linker an entry point in an executable program/file.

We have data section in above code. The data elements can be declared in terms of byte size. Data section means Declaring Initialized Data. For declaring the same one can use DB, DW, DD, DQ and DT.

Define Directive	Used to	How much
DB	Define Byte	define 1 byte
DW	Define Word	define 2 bytes
DD	Define Double	define 4 bytes
DQ	Define Quad	define 8 bytes
DT	Define Ten Bytes	define 10 bytes

We also have bss section in above code. The data elements in bss can be declared in terms of byte size requirement. BSS section uses the reserve directives which are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units/bytes of space to be reserved. Each define directive has a related reserve directive. For reserving the space we use RESB, RESW, RESD, RESQ and REST directives

Reserve Directive	How much?
RESB	Reserve Byte
RESW	Reserve Word
RESD	Reserve Double
RESQ	Reserve Quad
REST	Reserve Ten Bytes