Programming Language Evaluation

ASSEMBLY LANGUAGE

PINAR KOK
20160808013
Computer Engineering Department
Akdeniz University

Introduction

In a computer, all processes are executed by a processor. Each processor has an individual language called *machine language.* The processor understands only that language, and you make contact with the processor by using that language. However, that machine language is hard to learn and use. That is why some software was developed to translate a language that is understood by people into the machine language. This software is called an *assembler.* The most important difference between assembler and compiler is that assembler makes the one-to-one conversion. They read the whole code and translate that code intoto a meaningful program.

One of these assembler languages is ***Assembly***. Assembly is a low-level programming language. That language, unlike high-level programming languages that are often platform-independent, is closely tied to a target computer architecture, and this is computer specific.

History

The first Assembly Language was developed in 1947 by *Kathleen Booth*, at the University of London the following work with *John von Neumann* and *Herman Goldstine* at the Institute for Advanced Study.

ENIAC (Electronic Numeric Integrator and Computer) was the first computer that was developed by *Mauchly* and *Eckert* in between 1944 and 1946. To run this enormous machine, it was necessary to be able to control 18000 valves.

After that machine, *EDVAC (Electronic Numeric Integrator and Computer)* was created by *von Neumann*. This computer was a real computer. Because program and data could be kept in memory at the same time. So, each process was executing by turns automatically. The unit that kept the information in the memory was *binary digit*. It could take the value 0 or 1. Bits came together to form *byte*s, and each byte has an address. Because of writing to addresses and reading from addresses could be one-by-one, these processes were compared to the passage through a bottleneck and the machine was called *von Neumann bottle*.

The machines with valves were first-generation computers (IBM 600 series). Second-generation computers were transistor computers. Five years later, the third-generation computer with integrated circuit technology was created. Fourth-generation computers created by *VLSI (Very Large-Scale Integration)* technology are today's computers.

Program and data are uploaded as a binary digit. The machine code does that. However, that is very hard. The emergence of card readers made this process much easier. Assembler appeared after this development. Programs written with assemblers became readable programs.

Today

Unfortunately, Assembly Language cannot catch the programmer's attention because of some reason. The most popular of these reasons are performance and productivity. Even so, there are still some reasons to use this language. For example;

- The situation that is necessary to perform without the need to refer to working time staff or libraries in connection with a high-level language.

- The use of processor-specific commands that are not used in or compiled by a device driver or compiler.
- In the case of extreme optimization.
- When a system with serious resource shortages needs to be coded manually to maximize the use of limited resources.

As a tool today, only a few programmers regularly work with assembly language, but concepts that are not easily seen but important continue to be important; like binary arithmetic. Since the behavior of a computer is mainly determined by its command set, the logical method of learning these concepts is to work on an Assembly Language.

Usage

Assembly code that was coded by hand is used in *BIOS* (*Basic Input/Output System*) of a system, typically. This low-level code is hidden in *ROM* (*Read Only Memory*) and used to run the system hardware before introduced the *Operating System.*

Most compilers are the first thing to do before compilation, enabling assembly code to be highlighted by enabling assembly code to be viewed for debugging and optimization purposes. Programs that use these possibilities, such as the *Linux* kernel, can then build abstractions using different assembly languages on each hardware platform. The portable code of the system can then use elements unique to this processor through a uniform interface.

There are many assemblers such as *MASM* (*Microsoft Assembler*), *TASM* (*Turbo Assembler*), or if you use *Linux Disto* then you can use *NASM* (*Netwide Assembler*). Assembly Language code snippets can be embedded into C codes, and in that case, *Visual C++* works fine. Of course, a converter between the processor groups has its own dialect. Sometimes, TASM can read the old MASM code, but the old MASM code cannot read the TASM. The basic is exactly the same, but the advanced features are different.

Some assemblers are components of a compiler system for a high-level language. Some assemblers are hosted on the target processor and operating system.

Here is a list of x86-64 assemblers that assembly language source code into binary programs:

Assembler	Developer	Operating system
FASM	Tomasz Grysztar	Windows, DOS, Unix-like
GAS	GNU Project	Unix-like, Windows, DOS, OS/2
MASM	Microsoft	Windows, DOS, OS/2
NASM	Simon Tatham, Julian Hall, Hans Peter Anvin, et al.	Windows, Linux, macOS, DOS, OS/2
Open Watcom Assembler	Watcom	Windows, DOS, Linux, FreeBSD, OS/2
POASM		Windows, Windows Mobile
TCCASM	Fabrice Bellard	Unix-like, Windows
Yasm		Windows, DOS, Unix-like

However, the assembly code is the same if the processor is the same. x86 code compiled in Windows is binary compatible with x86 code on Linux. The compiler does not produce operating system dependent binary code, but it may package the code in a different format.

The difference is in which libraries are used. In order to use operating system stuff, such as I/O, you must link against the operating system's libraries. Therefore, executable files have to be in a format to be loaded into memory, and this varies from the operating system to the operating system. That is why you cannot take a Windows executable and associated libraries and run it on Linux. Windows system libraries are not available on a Linux machine, etc.

Besides, assembly can sometimes be carried on all systems running differently on the same type of *CPU* (*Control Processing Unit*), by connecting with a C library that does not change between usually running systems. Some high-level computer languages, such as C, support "*Inline Assembly*", where very short portions of the assembly code can be placed inside the high-level language code.

Difference of the "Hello World!" programs in *Assembly, Visual Basic* and *C* below. The code gets shorter as the language level increases:

Assembly

```
title   Hello World Program                              (hello.asm)

dosseg
.model small
.stack 100h

.data
hello_message db 'Hello World!',0dh,0ah,'$'

.code
main  proc
      mov    ax,@data
      mov    ds,ax

      mov    ah,9
      mov    dx,offset hello_message
      int    21h

      mov    ax,4C00h
      int    21h
main  endp
end   main
```

C

```
#include<stdio.h>

int main()
{
   printf("Hello World\n");
   return 0;
}
```

Visual Basic

```
Sub Main()
     MsgBox("Hello, World!") '
 End Sub
```

Code Structure

The main feature that distinguishes Assembly Language from others is that there are *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.

- **Data segment** is represented by *.data* section. 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.
- **Code segment** is represented by *.text* section. This defines an area in memory that stores the instruction codes. This is also a fixed area.
- **Stack segment** contains data values passed to functions and procedures within the program.

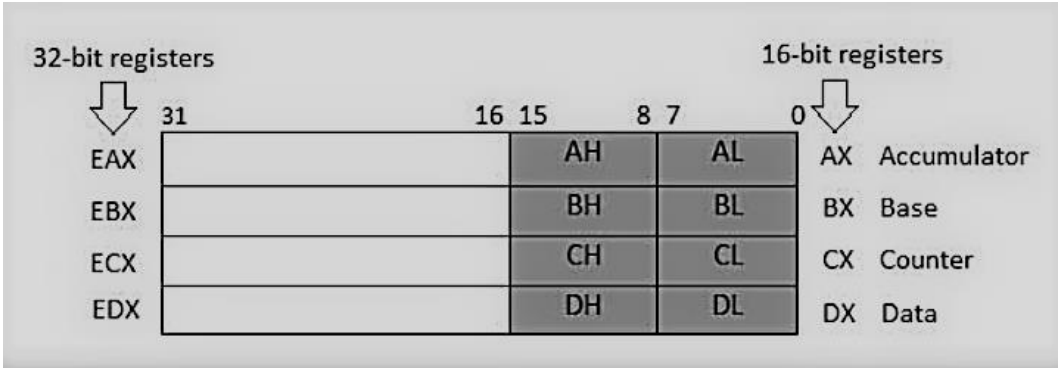
Of interest to us is the *flat memory model* that provides application programs with access to a continuous block of address space in memory. To implement a flat memory model at least data and code segments must be created in memory.

It is important to remember that your code is placed at a physical memory address assigned by the system. The assigned physical address is stored in descriptor tables in memory. your program is then assigned linear memory space beginning at zero. So, your code always thinks its starting address is at the bottom of memory. The linear address is a combination of the 32-bit descriptor table indices referenced by the segment registers and the 32-bit offset located in the instruction pointer for the code segment and 32-bit offset assigned each variable in the data segment.

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, DX.*
- Lower and higher halves of the 16-bit registers can be used as eight 8-bit data registers: *AH, AL, BH, BL, CH, CL, DH, DL.*



Basic Examples

- *INC, DEC (increment and decrement):*

```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
  inc myWord        ; 1001h
  dec myWord        ; 1000h
  inc myDword       ; 10000001h

  mov ax,00FFh
  inc ax            ; AX = 0100h
  mov ax,00FFh
  inc al            ; AX = 0000h
```

- *ADD, SUB (addition and subtraction):*

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code                ; ---EAX---
  mov eax,var1       ; 00010000h
  add eax,var2       ; 00030000h
  add ax,0FFFFh      ; 0003FFFFh
  add eax,1          ; 00040000h
  sub ax,1           ; 0004FFFFh
```

- *NEG (reversing the sign of an operand):*

```
.data
valB BYTE -1
valW WORD +32767
.code
  mov al,valB       ; AL = -1
  neg al            ; AL = +1
  neg valW          ; valW = -32767
```

- *implementing an arithmetic expression (using ADD, SUB, NEG):*

```
    Rval = -Xval + (Yval - Zval)

Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
   mov eax,Xval
   neg eax                ; EAX = -26
   mov ebx,Yval
   sub ebx,Zval           ; EBX = -10
   add eax,ebx
   mov Rval,eax           ; -36
```

- *TYPE (returning the size of a single element of a data declaration):*

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.code
mov eax,TYPE var1    ; 1
mov eax,TYPE var2    ; 2
mov eax,TYPE var3    ; 4
mov eax,TYPE var4    ; 8
```

- *LENGTHOF (counting the number of elements in a single data declaration):*

```
.data                              LENGTHOF
byte1  BYTE 10,20,30               ; 3
array1 WORD 30 DUP(?),0,0          ; 32
array2 WORD 5 DUP(3 DUP(?))        ; 15
array3 DWORD 1,2,3,4               ; 4
digitStr BYTE "12345678",0         ; 9

.code
mov ecx,LENGTHOF array1            ; 32
```

- *SIZEOF (returning a value that equivalent to multiplying LENGTHOF by TYPE):*

```
.data                              SIZEOF
byte1  BYTE 10,20,30               ; 3
array1 WORD 30 DUP(?),0,0          ; 64
array2 WORD 5 DUP(3 DUP(?))        ; 30
array3 DWORD 1,2,3,4               ; 16
digitStr BYTE "12345678",0         ; 9

.code
mov ecx,SIZEOF array1              ; 64
```

- *indirect operands:*

```
.data
val1 BYTE 10h,20h,30h
.code
mov esi,OFFSET val1
mov al,[esi]            ; dereference ESI (AL = 10h)

inc esi
mov al,[esi]            ; AL = 20h

inc esi
mov al,[esi]            ; AL = 30h
```

- *printing the number 1 to 9 on the screen:*

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point
   mov ecx,10
   mov eax, '1'

l1:
   mov [num], eax
   mov eax, 4
   mov ebx, 1
   push ecx

   mov ecx, num
   mov edx, 1
   int 0x80

   mov eax, [num]
   sub eax, '0'
   inc eax
   add eax, '0'
   pop ecx
   loop l1

   mov eax,1             ;system call number (sys_exit)
   int 0x80              ;call kernel
section .bss
num resb 1
```

➢ As you can see, even simple and basic codes are too long to write!