X86 프로세서 어셈블리 언어 분석

X86 Processor Assembly Language Analysis

임준수

컴퓨터공학과

Jun Su. Lim

Department of Computer Science and Engineering, Dongguk University

요 약

어셈블리 언어는 가장 오래된 프로그래밍 언어로써 모든 고급 언어의 근간이 된다고 할 수 있다. 하지만 어셈블리 언어는 고급 언어에 비해 어렵다는 프로그래머들의 인식에 의해 현시대에 프로그래머들로부터 배척 받고 있다. 본 논문에서는 x86 어셈블리 언어의 중요한 핵심을 설명하여 본 논문을 읽는 사람들에게 어셈블리 언어가 가지는 중요성을 보여주고자 한다. 본 논문에서는 X86 어셈블리 언어의 구조, 데이터 전송, 주소지정과 산술연산, 프로시저, 실행시간 스택, 조건부 처리, 구조체 및 매크로를 설명한다. 결론에서는 본문의 내용을 바탕으로 어셈블리 언어의 장단점과 어셈블리 언어를 배워야 되는 이유를 말하고자 한다.

주제어 : 어셈블리 언어, 고급언어, X86 어셈블리 언어의 구조, 데이터 전송, 주소지정과 산술연산, 프로시저, 실행시간 스택, 조건부 처리, 구조체 및 매크로

Abstract

There assembly language can be said to be the foundation of any high-level language as the oldest programming languages. But assembly language programmers have been excluded from the modern world by the recognition of the difficult than in high-level language. In this paper, we want to show the importance of the assembly language to the people reading this paper, we describe the important key of the x86 assembly language. In this paper, the structure of the X86 assembly language, data transfer, addressing and arithmetic operation, procedure, runtime stack, conditional processing, macro and the structure will be described. In conclusion I would like to say why the pros and cons and learn assembly language, based on the contents of the text.

Key Words : assembly language, high-level language, the structure of the X86 assembly language, data transfer, addressing and arithmetic operation, procedure, runtime stack, conditional processing, macro, structure

I. 서 론

개요

어셈블리언어는 가장 오래된 프로그래밍 언어이고 모든 언어 중에서 고유의 기계어에 가장 가깝게 닮았다. 어셈블리 언어는 컴퓨터 하드웨어에 직접 접근할 수 있도록 도움을 준다. C 또는 C++ 개발자가 되려고 한다면 메모리, 주소와 명령어가 저수준에서 어떻게 동작하는지를 이해를 높일 필요가 있다. 많은 프로그래밍 오류들은 고급언어 수준에서는 쉽게 인식되지 못한다. 왜 동작하지 않는지를 알아내기 위해서 프로그램의 내부까지 자세히 볼 필요가 있다.

어셈블리 언어는 기계어와 일대일(one-to-one) 대응관계를 갖는다. 이는 각 어셈블리 언어 명령어는 하나의 기계어 명령어와 대응됨을 의미한다. C++와 자바와 같은 고급언어는 어셈블리언어와 기계어와 일대다(one-to-many)관계를 갖는다. C++에서의 한 문장은 여러 개의 어셈블리 언어나 기계어 명령어로 확장된다. C++나 Java는 저수준 데이터 접근을 못하는 대신에 예기치 않은 논리 오류를 줄이도록 하기 위한 구문 규칙을 사용하기 때문에 더 많은 규칙을 갖고 있다. 어셈블리 언어 프로그래머는 고급 언어가 갖고 있는 제한을 쉽게 넘어설 수 있다. 예를 들어 Java는 특정한 메모리 주소에 대한 접근을 허용하지 않는다. JNI(Java Native Interface) 클래스를 사용하여 C함수를 호출해서 Java의 제한을 피해서 작업을 할 수 있다. 그렇지만 작성된 프로그램은 유지보수하기가 불편할 수 있다. 반면에 어셈블리 언어는 어떠한 메모리 주소라도 접근할 수 있다.

본 논문에서는 시뮬레이션 어셈블러를 사용하는 컴퓨터용 프로그래밍에 대해 분석하지 않고 현업에 종사하는 전문가가 사용하는 매우 강력한 기능의 MASM 어셈블러의 x86 어셈블리언어를 설명할 것이다.

II. 본 론

X86구조 세부 사항

CPU는 계산과 논리 동작이 일어나는 곳이다. CPU는 레지스터라고 하는 제한된 수의 저장 장소, 동작의 동기화를 위한 고주파수 클럭, 제어 장치, 산술 논리 장치가 있다. 메모리 저장 장치는 컴퓨터 프로그램이 실행되는 동안 명령어와 데이터를 저장하는 곳이다. 버스는 컴퓨터의 여러 부분 간에 데이터를 전송하는 일련의 병렬 와이어이다.

하나의 기계어 명령어의 실행은 명령어 실행 사이클이라고 하는 연속되는 개별 동작들로 나누

어진다. 인출, 해독, 실행이 세 개의 주요 동작이다. 명령어 사이클의 각 단계는 적어도 하나의 시스템 클럭이 소요된다. 적재와 실행은 프로그램이 어떻게 운영체제에 의해서 메모리를 배정받아 적재되고 실행되는지를 설명한다.

어셈블리언어는 적재와 실행을 하기 위해서 레지스터를 사용하는데 레지스터는 보통의 메모리보다 훨씬 더 빠르게 접근할 수 있는 CPU 내부의 이름이 부여된 저장장소이다. 다음은 레지스터 유형에 대한 설명이다.

- 범용 레지스터는 주로 산술, 데이터 전송, 논리 연산에 사용된다.
- 세그먼트 레지스터는 세그먼트라고 하는 미리 할당된 메모리 영역에 대한 시작 주소로 사용된다.
- EIP(instruction pointer)는 실행할 다음 명령어의 주소를 갖고 있다.
- EFLAGS(extended flags)는 CPU의 동작을 제어하고 ALU 연산 결과를 반영하는 개별적인 2진수 비트들로 구성되어 있다.

실제 주소 모드에서 16진수 주소 00000부터 FFFFF까지의 1MB의 메모리만 주소 지정할 수 있다. 보호 모드에서 프로세서는 동시에 여러 개의 프로그램을 실행할 수 있다. 프로세서는 각 프로세스에게 총 4GB의 가상 메모리를 할당한다. 또한 균일 세그먼트 모델에서 모든 세그먼트는 컴퓨터의 전체 물리적 주소 공간에 맵핑된다. 다중 세그먼트 모델에서 각 태스크는 지역 서술자 테이블(LDT)라고 하는 자신 만의 세그먼트 서술자 테이블을 갖는다. X86 프로세서는 세그먼트를 페이지라고 하는 4096바이트의 메모리 블록으로 나누는 페이징이라고 하는 특징을 지원한다. 페이징을 사용하면 동시에 실행되는 모든 프로그램이 사용하는 전체 메모리가 컴퓨터의 실제(물리적) 메모리보다 훨씬 더 커도 된다.

데이터 전송, 주소 지정과 산술연산

데이터 전송 명령어인 MOV는 소스 피연산자를 목적지 피연산자로 복사한다. MOVZX 명령어는 작은 피연산자를 큰 레지스터로 제로 확장한다. MOVSX명령어는 작은 피연산자를 큰 레지스터로 부호 확장한다. XCHG 명령어는 두 피연산자의 내용을 서로 교환한다. 적어도 하나의 피연산자는 레지스터이어야 한다. 어셈블리언어는 다음 유형의 피연산자를 갖는다.

- 직접 피연산자는 변수의 이름이고 변수의 주소를 나타낸다.
- 직접 오프셋 피연산자는 변수의 이름에 변위를 더하여 새로운 오프셋을 만든다. 이 새로운 오프셋은 메모리에 있는 데이터에 접근하는 데 사용될 수 있다.
- 간접 피연산자는 데이터의 주소를 포함한 레지스터이다. [esi]와 같이 레지스터를 대괄호 안에 넣어서 프로그램이 주소를 역참조하여 메모리 데이터를 꺼낸다.
- 인덱스 피연산자는 상수를 간접 피연산자와 결합한다. 상수와 레지스터 값을 더하여 계산된 오프셋이 역참조된다. 예를 들어 [array+esi]와 array[esi]는 인덱스 피연산자

이다.

산술 명령어들과 연산자는 다음과 같다.

- INC명령어는 피연산자에 1을 더한다.
- DEC명령어는 피연산자에서 1을 뺀다.
- ADD명령어는 소스 피연산자를 목적지 피연산자에 더한다.
- SUB명령어는 목적지 피연산자에서 소스 피연산자를 뺀다.
- NEG명령어는 피연산자의 부호를 반전시킨다.
- OFFSET 연산자는 이 연산자를 포함하는 세그먼트의 시작으로부터의 변수의 거리를 반환한다.
- PTR 연산자는 변수의 선언된 크기를 바꾸어 지정할 수 있게 한다.
- TYPE 연산자는 변수 또는 배열 원소의 크기(바이트 단위)를 반환한다.
- LENGTHOF 연산자는 배열의 원소 개수를 반환한다.
- SIZEOF 연산자는 배열의 초기값이 설정된 바이트 수를 반환한다.
- TYPEDEF 연산자는 사용자 정의 자료형을 만든다.

다음 CPU상태 플래그는 산술 연산에 영향 받는다.

- Sign 플래그는 산술 연산의 결과가 음수일 때에 1로 설정된다.
- Carry 플래그는 부호없는 산술 연산의 결과가 목적지 피연산자에 저장되기에는 너무 클 때 1로 설정된다.
- Parity 플래그는 산술논리 연산을 수행한 직후에 목적지 피연산자의 최하위 바이트에 1인 비트의 개수가 짝수인지를 나타낸다.
- 보조 Carry 플래그는 목적지 피연산자의 비트 위치 3에서 캐리 또는 빌림수가 발생할 때에 1로 설정된다.
- Zero 플래그는 산술 연산의 결과가 0일 때 1로 설정된다.
- Overflow 플래그는 부호있는 산술 연산의 결과가 목적지 피연산자에 저장되기에는 너무 클 때에 1로 설정된다.

기본적으로 CPU는 프로그램을 적재하여 순차적으로 실행한다. 그렇지만 현재의 명령어는 조건부일 수 있으며, 조건부는 CPU의 상태 플래그(Zero, Sign, Carry 등)의 값에 따라서 새로운 위치로 프로그램의 제어가 이동됨을 의미한다. 어셈블리 언어 프로그램은 IF문과 루프와 같은 고급 문장을 구현하기 위하여 조건부 명령어를 사용한다. 각 조건부 문장은 다른 메모리 주소로 제어를 이동하는 것이 가능하다. 제어의 이동(transfer of control), 즉 분기(branch)는 문장들이 실행되는 순서를 변경하는 방법이다. 다음과 같은 두 가지 기본적인 유형의 이동이 있다.

- 무조건 이동(unconditional transfer) : 모든 경우에 제어는 새로운 위치로 이동된다.

새 주소는 명령어 포인터에 적재되어 새 주소에서 계속하여 실행되게 한다. JMP 명령어가 이 동작을 수행한다.

- 조건부 이동(conditional transfer) : 프로그램은 어떤 조건이 참이면 분기한다. 매우 다양한 조건부 분기 명령어가 조건부 논리 구조를 만들기 위해서 결합되어 사용될 수 있다. CPU는 ECX와 플래그 레지스터의 내용에 의하여 참/거짓 조건을 해석한다.

프로시저, 실행시간 스택

실행시간 스택은 스택 포인터 레지스터라고 하는 ESP레지스터를 사용하여 CPU가 직접 관리하는 메모리 배열이다. ESP 레지스터는 스택에 있는 어떤 위치에 대한 32비트 오프셋을 저장한다. ESP를 직접 조작하는 일은 거의 없으며 ESP는 CALL, RET, PUSH, POP과 같은 명령어를 사용하여 간접적으로 수정된다. ESP는 항상 스택의 맨 위에 추가된, 즉 push된 마지막을 가리킨다.

실행시간 스택의 몇 가지 중요한 용도가 있다. 첫 번째, 스택은 레지스터가 한 가지 이상의 목적으로 사용될 때에 레지스터를 위한 간편한 임시 저장 영역으로 사용된다. 레지스터가 수정된 뒤에 원래의 값으로 복원될 수 있다. 두 번째, CALL 명령어를 수행할 때에 CPU는 현재 서브루틴(CALLER)의 복귀주소를 스택에 저장한다. 세 번째, 서브루틴을 호출할 때에 인수(argument)라고 하는 입력 값을 스택에 push하여 전달한다. 네 번째, 스택은 서브루틴 내의 지역변수를 위한 임시 저장 공간을 제공한다. 아래의 그림은 func(argument1, argumnet2)를 호출 했을 때에 스택 프레임의 모양이다.

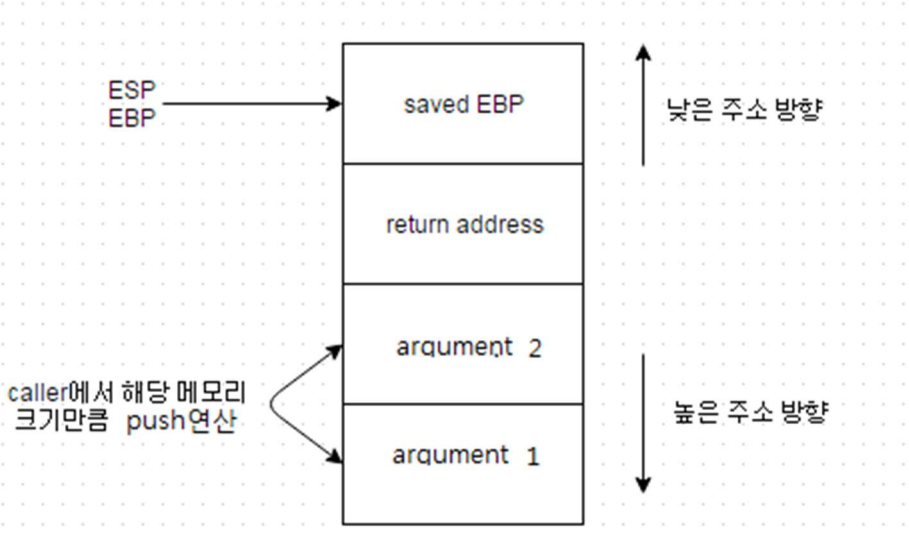

Figure 1. 스택 프레임 형태

MASM에서는 PUSH와 POP의 기능을 확장한 명령어가 존재한다. PUSHAD 명령어는 32비트 범용 레지스터를 스택에 push하고 PUSHA 명령어는 16비트 범용 레지스터에 대해서 같은 동작을

수행한다. POPAD 명령어는 스택을 32비트 범용 레지스터로 pop하고 POPA 명령어는 16비트 범용 레지스터에 대해서 같은 동작을 수행한다. PUSHFD 명령어는 32비트 EFLAGS 레지스터를 스택에 push하고 POPFD는 스택을 EFLAGS로 pop한다. PUSHF와 POPF는 16비트 EFLAGS 레지스터에 대해서 같은 동작을 수행한다.

프로시저는 PROC와 ENDP 디렉티브를 사용하여 선언되는 이름이 부여된 코드의 블록이다. 프로시저 실행은 RET명령어로 끝난다. CALL명령어는 프로시저의 주소를 명령어 포인터 레지스터로 넣어서 프로시저를 수행한다. 프로시저가 끝날 때에 RET(return from procedure) 명령어는 프로세서를 프로그램의 프로시저가 호출된 지점으로 되돌아오게 한다.

조건부 처리

의사 결정을 허용하는 프로그래밍 언어는 조건부 분기라고 알려진 기법을 사용하여 제어의 흐름을 변경할 수 있다. 고급언어에서는 찾아볼 수 있는 모든 IF문, switch문 또는 조건부 루프는 자체의 분기 논리 구조를 가지고 있다. 어셈블리 언어는 그 자체가 기본이 되는 만큼 의사 결정 논리를 수행하는 데 필요한 모든 수단을 제공한다. X86 명령어 집합에는 명시적인 고급 논리 구조는 없지만, 비교와 점프 명령어를 조합하여 고급 논리 구조를 구현할 수 있다. 조건부 문장을 수행하는 데 다음의 두 단계가 포함된다. 첫째로 CMP, AND, SUB와 같은 연산은 CPU 상태 플래그를 수정한다. 둘째로 점프 명령어는 플래그를 검사하여 새로운 주소로 분기하게 한다.
예를 들어 아래의 코드는 C++ 고급언어를 어셈블리 언어로 변환한 모습이다.

clusterSize = 8192;
if gigabytes < 8
  clusterSize = 4096;
	mov clusterSize, 8192
	cmp gigabytes, 8
	jae next
	mov clusterSize, 4096

또한 어셈블리 언어는 AND 연산자를 포함한 복합 부울 수식을 쉽게 구현한다. 다음 의사코드를 보자

```
if( a1 > b1) AND ( b1 > c1 ) then
    X = 1
End if
```

해당 의사코드를 단축 평가를 사용하여 2가지 형태로 간단하게 구현할 수 있다. 첫 번째 형태는 수식이 거짓이면 둘째 수식은 계산되지 않는다. 두 번째 형태는 첫 번째 형태에서 사용한 JA 명령어를 JBE로 바꾸어서 코드를 5개의 명령어로 줄일 수 있다.

1.	**2.**
cmp a1 , b1 ja L1 jmp next L1: cmp b1 , c1 ja L2 jmp next L2: mov X, 1 next :	cmp a1 , b1 jbe next cmp b1 , c1 jbe next mov X, 1 next :

코드 크기 감소는 CPU가 처음의 JBE 명령어가 점프되지 않으면 두 번째 CMP 명령어로 곧바로 가게 하여 이루어진 것이다.

WHILE 루프는 문장 블록을 수행하기 전에 조건을 검사한다. 루프 조건이 참으로 지속되는 동안 문장들이 반복된다. 다음은 WHILE 루프의 어셈블리 언어 구현 모습이다.

while(val1 < val2)
{
val1++;
val2--;
}
	mov eax, val1
	beginwhile:
	cmp eax, val2
	jnl endwhile
	inc eax
	dec val2
	jmp beginwhile
	endwhile:
	mov val1, eax

EAX는 루프 안에서 val1 대신에 사용된다. Val1에 대한 참조는 EAX를 통해야 한다. JNL이 사용되었으며, 이는 val1과 val2가 부호 있는 정수라는 것을 의미한다.

요약하자면 AND, OR, XOR, NOT, TEST 명령어는 비트 수준에서 동작하기 때문에 비트 단위 명령어라고 부른다. 소스 피연산자의 각 비트는 목적지 피연산자의 같은 위치의 비트와 상대가 된다. CMP명령어는 목적지 피연산자를 소스 피연산자와 비교한다. 이 명령어는 내부적으로 목적지 피연산자에서 소스 피연산자를 빼고 결과에 따라서 CPU상태 플래그를 수정한다. CMP는 대개 코드 레이블로 제어를 이동하도록 조건부 점프 명령어가 뒤따른다.

구조체 및 매크로

구조체는 데이터를 묶고 이를 하나의 프로시저에서 다른 프로시저로 전달하는 쉬운 방법을 제공한다. 기존 프로시저를 호출할 때 레지스터로 인자를 전달하는 방법은 순서가 뒤섞이거나 매개변수와 일치하지 않는 인자를 전달할 수 있다. 하지만 구조체를 사용함으로써 전달할 인자를 구조체에 집어넣고 프로시저에 전달하면 프로그램의 오류가 발생하지 않는다.

좋은 메모리 I/O 성능을 위해서는 구조체 멤버는 짝수 배 데이터 타입에 맞는 주소에 정렬되어

야 한다. 그렇지 않으면 구조체 멤버에 접근하는데 CPU는 더 많은 시간을 소비한다. 예를 들어 더블워드 멤버는 더블워드 경계에 정렬되어야 하고 워드 멤버는 워드 경계에 정렬되어야 한다. 어셈블리 언어에서는 구조체 멤버의 정렬을 지원하기 위해서 ALIGN 디렉티브를 제공한다. 아래의 코드는 구조체를 정의한 모습이다. ALIGN을 사용해서 주소 정렬을 보여준다.

```
Employee STRUCT
    IdNum       BYTE   "000000000"          ; 9
    LastName    BYTE   30   DUP(0)          ; 30
    ALIGN       WORD                        ; 1 byte added
    Years       WORD   0                    ; 2
    ALIGH       DWORD                       ; 2 bytes added
    SalaryHistory   DWORD 0, 0, 0, 0        ; 16
Employee ENDS                               ; total 60 bytes
```

Employee 구조체를 정의하는데 Years는 ALIGH을 사용해서 WORD 경계에 정렬하고 SalaryHistory는 DWORD 경계에 정렬한다.

아래의 코드는 정렬된 구조체 멤버의 성능을 확인하기 위한 코드이다. 해당 코드는 프로세서가 올바르게 정렬된 구조체 멤버에 대하여 좀 더 효과적으로 접근할 수 있다는 걸 보여준다.

```
EmployeeBad STRUCT
        IdNum    BYTE "000000000"
        LastName BYTE  30  DUP(0)
        Years    WORD     0
        SalaryHistory DWORD  0, 0, 0, 0
EmployeeBad ENDS

Employee STRUCT
        IdNum    BYTE "000000000"
        LastName BYTE  30  DUP(0)
        ALIGN    WORD
        Years    WORD     0
        ALIGN    DWORD
        SalaryHistory DWORD  0, 0, 0, 0
Employee ENDS

INCLUDE Irvine32.inc
.data
ALIGN  DWORD
startTime  DWORD  ?      ;align startTime
emp Employee <>          ;or : emp EmployeeBad<>

.code
main PROC
        call    GetMSeconds    ;get starting time
        mov     startTime, eax

        mov     ecx, 0ffffffffh  ;loop counter
L1:     mov     emp.Years, 5
```

```
        mov     emp.SalaryHistory, 35000
        loop L1

        call    GetMSeconds;    ;get ending time
        sub     eax, startTime
        call    WriteDec        ;display elapsed time
main ENDP
END main
```

Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz 프로세서에서 반복문을 4294967295번 실행하였을 때 정렬된 Employee 구조체는 11668 밀리초가 걸렸고 비 정렬된 EmployeeBad 구조체는 11809 밀리초가 걸렸다. 실행할 때마다 밀리초는 바뀌지만 정렬된 구조체가 비 정렬된 구조체보다 프로세서가 효과적으로 접근하고 있는 것을 확인할 수 있다.

구조체 멤버를 참조하기 위해 간접 피연산자는 레지스터(ESI등)의 사용을 허용한다. 간접 주소 지정은 유연성을 제공하는데 특히 구조체 주소를 프로시저에 전달할 때나 구조체 배열을 사용할 때 그렇다. 간접 피연산자를 참조할 때는 PTR연산자가 요구된다. 아래의 코드는 간접 피연산자를 사용한 모습이다.

```
mov   esi, OFFSET worker
mov   ax, ( Employee PTR [esi]). Years
```

매크로 프로시저는 어셈블리 언어 문장들 중 하나의 블록에 이름을 붙인 것이다. 일단 정의되면 매크로는 같은 프로그램에서 여러 번 호출될 수 잇다. 매크로 프로시저를 호출할 때 매크로 코드의 복사본이 매크로 프로시저를 호출한 위치에 직접 삽입된다. 이러한 자동 코드 삽입은 인라인 확장(inline expansion)이라 한다. 매크로는 어셈블러의 전처리 단계 동안 인라인 확장된다. 이 같은 단계에서 전처리기는 매크로 정의를 읽고 프로그램에 남아 있는 소스 코드를 스캔한다. 매크로가 호출되는 모든 점에서 어셈블러는 매크로의 소스 코드 복사본을 삽입한다. 매크로의 어떠한 호출을 어셈블하기 전에 매크로 정의는 어셈블러에 의해 발견되어야 한다. 만약 프로그램이 매크로를 정의하긴 하지만 전혀 호출하지 않는다면 매크로 코드는 컴파일된 프로그램에 나타나지 않는다.

일반적으로 매크로는 프로시저보다 더 빨리 실행되는데 프로시저는 CALL, RET 명령어에 의한 별도의 부담이 있기 때문이다. 그러나 매크로를 사용하는 것은 한 가지 문제점이 있는데 커다란 매크로를 반복해서 사용하는 것은 프로그램 크기를 증가시킨다. 왜냐하면, 매크로에 대한 호출마다 매크로 문장들을 프로그램에 복사해 넣어야 하기 때문이다. 이러한 문제점을 보완하기 위해서는 매크로를 생성할 때 모듈적 접근방법을 사용하면 된다. 매크로를 짧게 하고 단순하게 하는 것이 더 복잡한 매크로를 만들 수 있기 때문이다. 이렇게 함으로써 프로그램의 중복 코드 양을 줄이고 프로그램의 크기를 줄일 수 있다.

III. 결 론

본 논문에서는 x86 어셈블리언어의 구조, 데이터 전송, 주소지정과 산술연산, 프로시저, 실행시간 스택, 조건부 처리, 구조체 및 매크로에 관하여 설명하였다. 본 주제들을 통해서 x86어셈블리언어의 내용을 설명했다. X86 어셈블리 언어는 다른 프로그래밍 언어에 비해 상대적으로 어렵고 복잡하게 구성되어 있다. 현재 프로그래밍의 패러다임은 저급언어보다 고급언어를 선호하는 경향이 강하다. 고급언어는 코드의 많은 부분을 쉽게 구성하고 유지할 수 있는 형식구조를 가짐으로써 어셈블리 언어의 난해한 코드 작성보다 훨씬 선호 받고 있다. 하지만 고급언어는 하드웨어의 직접 접근을 제공하지 않거나 제공한다고 할지라도 쉽지 않은 코딩 기술이 필요하기 때문에 유지보수가 힘들다. 그에 반해 어셈블리언어는 하드웨어 접근은 직접적이고 간단하며 하드웨어를 컨트롤하는 프로그램은 고급언어에 비해 상대적으로 짧고 유지보수하기가 쉽다. 또한 임베디드 시스템과 컴퓨터 게임 등에서 어셈블리언어는 실행 코드가 작고 빠르게 실행되지만 고급언어는 매우 많은 실행 코드를 만들어 시스템에 많은 리소스를 사용한다. 이와 같이 어셈블리 언어는 고급언어보다 뛰어난 성능 최적화와 하드웨어의 접근을 지원함으로써 프로그래머의 활동 영역을 넓히지만 반대로 너무나도 많은 자유를 가짐으로 인해 프로그래머의 부주의한 사용은 컴퓨터 시스템의 치명적 문제를 야기할 수 있다. 그러므로 어셈블리 언어는 양날의 칼과 같으며 프로그래머가 어떻게 사용하냐에 따라 다양한 역할을 할 수 있다. 그럼에도 어셈블리 언어는 컴퓨터 시스템과 하드웨어를 이해하는데 있어 매우 기본이 되기 때문에 C 또는 C++ 개발자가 되려고 한다면 꼭 배워야 된다고 생각한다.

참 고 문 헌

[1] KIP R. IRVINE,『ASSEMBLY LANGUAGE FOR x86 PROCESSORS』, 박명순, 윤상균(역), 서울: 퍼스트 북, 2011

[2] 한세경,『뇌를 자극하는 프로그래밍 원리 - CPU부터 OS까지』, 서울: 한빛미디어, 2007

[3] 브루스 댕, 알렉산더 가제트, 엘리어스 바찰러니,『역공학 – X86, X64, ARM, 윈도우 커널, 역공학 도구, 그리고 난독화』, 김종덕(역), 경기: 비제이퍼블릭, 2015

[4] Intel x86 assembly manual : http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (2015.12.07.)