초보자들을 위한 어셈블리어 기초-1

written by vangelis(vangelis@wowhacker.org)

쓸데 없는 말

어셈블리어는 저수준(low-level)의 프로그래밍 언어이다. 이것은 C/C++, Basic과는 다른 언어라는 것은 누구나 알고 있을 것이다. 일반적으로 어셈블리어를 배우는 이유는 컴퓨터의 작동 원리를 이해하는데 도움이 되기 때문이며, 또한 직업적인 목적을 위해 배우기도 한다. 이 글은 해킹을 공부할 때, 특히 시스템 해킹을 공부할 때마다 등장하는 어셈블리어 관련 정보를 초보자들도 최대한 이해할 수 있도록 도움을 주기 위한 것이다. 아마도 이 글을 읽고 있는 분들 중에서는 크랙 된 프로그램을 사용하고 있는 분들도 많이 있을 거라고 생각한다. 이것 역시 어셈블리어에서 출발한다고 생각하면 된다. 이 글이 프로그램 크랙에 목적을 두고 있지는 않기 때문에 프로그램 크랙에 관심이 많은 분들은 관련 사이트를 찾아 보길 바란다. 강조하건데 이 글은 생 초보자를 위한 것이다.

요즘은 인터넷의 발달로 컴퓨터가 일종의 계산기라는 생각은 많이 하지 않고 있는 것 같다. 계산기란 수를 다루는 것이므로 컴퓨터를 이해하기 위해서는 수의 체계에 대한 이해는 필수적이라 생각한다. 수의 체계 중에서 2진수, 10진수, 16진수에 대한 이해는 필수라는 것은 알고 있을 것이다. 그 중에서도 16진수에 대한 이해는 더욱 중요하다. 적어도 이 글을 읽는 사람이라면… 쓸데 없는 이야기 하지 말고 바로 본론으로 들어가보자.

수의 체계

십진수(Decimal)

10진수(Base 10)는 누구나 익숙해 있는 수 체계이다. 우리가 일상 생활에서 사용하는 것이 바로 이 10진수이기 때문이다. 10진수는 10개의 숫자(0~9)로 구성되어 있다. 10진수에 대해서는 간단한 예만 들고 넘어가도록 하겠다. 123라는 10진수를 예를 들어보자. 123은 다음과 같이 구성되어 있다. 이것은 초등학교만 나와도 이해할 수 있을 것이다.

$$123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

이진수(Binary)

2진수(Base 2)는 두 개의 숫자(0과 1)로 구성되어 있다. 2진수의 각각의 수는 bit라고 불린다. 2진수 또한 쉽게 이해할 수 있을 거라고 생각한다. 이 글을 읽는 사람이 적어도 중학생 이상이길 바란다. 초등학생들도 잘 이해할 테니까.. 이진수를 10진수로 전환하는

예를 하나 들어보자.

$11001_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$= 16 + 8 + 1$
$= 25$

이제 10진수 0 ~ 15까지를 2진수와 비교를 해보자.

10진수	2진수	10진수	2진수
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

심심한데, 산수 공부나 한번 하고 16진수로 넘어가자.

$11011_2 + 10001_2 = 101100_2$

16진수(Hexadecimal)

16진수는 base 16을 사용한다. 16진수(Hexadecimal 또는 줄여서 hex)는 당연히 16개의 수를 가지고 있다. 16개의 수를 가지고 있기 때문에 16진수를 표현하는 문제가 하나 생겼다. 그것은 인간이 사용하는 숫자는 0~9까지밖에 없기 때문이다. 그래서 0~9까지는 숫자를 그대로 사용하고, 그 다음부터는 A, B, C, D, E, F를 사용한다. 각 알파벳은 10진수의 10,11,12,13,14,15에 해당한다. 그럼 예를 하나 들어 16진수를 10진수로 변환해보자.

$2BD_{16} = 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0$
$= 512 + 176 + 13$
$= 701$

이제 16진수를 2진수로 변환해보자. 16진수를 2진수로 변환하기 위해 간단히 각각의 16진수를 4 bit의 이진수로 변환하면 된다. 예를 들어 16진수 24D는 2진수 001001001101로 변환된다. 즉, 2진수의 각 4bit 세그먼트를 16진수로 전환한다. 이때 왼쪽부터가 아니라 오른쪽부터 시작하는 것을 유의해야 한다. 다음 예를 보자.

```
110 0000 0101 1010 0111 1110₂
  6    0    5    A    7    E₁₆
```

다음으로 10진수를 16진수로 변환해보자. 1324라는 10진수는 52C가 된다. 이 값을 어떻게 나왔는가? 간단히 알아보자.

1324 / 16 = 82.75

82 x 16   = 1312

1324 - 1312 = 12: C로 변환

82 / 16 = 5.125

5 x 16 = 80

82 – 80 = 2: 2로 변환

5 / 16 = 0.3125

0 x 16 = 0

5 – 0 = 5: 5로 변환

결국 52C가 된다. 이보다 더 쉬운 방법이 있으면 그것을 이용하자.

하나의 4bit 수는 nibble이라고 한다. 그래서 각각의 16진수는 하나의 nibble에 상응한다. 2 nibble은 1 byte에 해당하고, 1byte는 2자리 16진수로 나타난다. 1byte 값의 범위는 이진수로는 0 ~ 11111111, 16진수로는 FF, 10진수로는 0 ~ 255이다.
이제 2진수, 10진수, 16진수를 같이 써보자.

16 진수	10 진수	2 진수
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010

B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

메모리

메모리의 기본 단위는 1 byte이다. 32MB의 메모리를 가지고 있는 컴퓨터는 32백만 byte의 정보를 가질 수 있다. 메모리에서 각 byte는 메모리 주소라는 고유 숫자가 붙어 있다.

주 소	0	1	2	3	4	5	6	7
메모리	2A	45	B8	20	8F	CD	12	2E

종종 메모리는 1바이트보다는 더 큰 단위로 사용되기도 한다. 다음은 메모리의 단위이다.

word = 2 바이트, double word = 4 바이트, quad word = 8 바이트, paragraph = 16 바이트
메모리의 모든 데이터는 숫자로 되어 있다. 문자들은 숫자를 문자로 맵핑하는 문자 코드를 사용하여 저장된다. 가장 일반적인 문자 코드는 아마도 많이 들어보았을 ASCII(American Standard Code for Information Interchange)이다. ASCII를 대체하고 있는 더 완벽한 코드는 Unicode이다. 두 코드 사이의 주된 차이점은 ASCII는 하나의 문자를 인코딩하기 위해 1바이트를 사용하고, Unicode는 한 문자당 2바이트(1 word)를 사용한다는 것이다. 예를 들어, ASCII는 A라는 문자에 4116 (6510) 바이트를 맵핑하고, Unicode는 word 004116를 맵핑한다. ASCII가 1바이트를 사용하기 때문에 단지 256 개의 다른 문자에만 국한되어 있다. 이에 반해 Unicode는 더 많은 문자를 재현할 수 있다.

Segment:Offset

컴퓨터의 모든 것은 BUS라고 불리는 일련의 전선을 통해 연결되어 있다. RAM으로 연결된 BUS는 16비트이다. 그래서 프로세스가 RAM에 자료를 쓸 필요가 있을 때 BUS를 통해 16비트 로케이션을 보낸다. 이전에는 컴퓨터가 가질 수 있는 메모리에 한계가 있었다. 그러나 이 한계를 극복하기 위해 디자이너들은 20비트를 BUS를 통해 전달할 수 있는 방법을 고안해 냈다.

그래서 1MB의 메모리를 허용함으로써 메모리는 Segment라고 불리는 바이트의 집합으로 분리되었다. 그리고 그 세그먼트 내에 Offset 번호를 지정함으로써 메모리에 접근하게 되었

다. 그래서 만약 프로세서가 어떤 데이터에 접근하고자 한다면 먼저 세그먼트 번호를 보내며, Offset 번호가 뒤따른다. 예를 들어, 프로세서가 1234:4321(segment:offset)의 리퀘스트를 보내면 RAM은 세그먼트 번호 1234에 있는 4321 바이트를 보낸다. 다음 표를 보자.

```
             +--+--+--+--+--+
           5 |  |  |  |  |  |
             +--+--+--+--+--+
           4 |  |  |* |  |  |
  Y 축       +--+--+--+--+--+
(segment)  3 |  |  |  |  |  |
             +--+--+--+--+--+
           2 |  |  |  |  |  |
             +--+--+--+--+--+
           1 |  |  |  |  |  |
             +--+--+--+--+--+
               1  2  3  4  5
                 X 축(offset)
```

위의 표에서 *는 4:3에 위치해 있다. 물리적 주소(physical address)를 구하기 위해서는 다음과 같이 하면 된다.

```
Segment x 10h(여기서 h는 이 수가 16진수라는 것을 가르킴) + offset = 물리적 주소
```

예를 들어, 1234:4321에서 물리적 주소는 다음과 같다.

```
1234 x 10h + 4321 = 16661
```

레지스터(Register)

CPU는 명령을 수행하는 물리적 장치이다. 아마도 펜티엄 시리즈라면 누구든 잘 알고 있는 것이다. 잘 모르겠으면 자신의 컴퓨터 케이스를 열어보자. 컴퓨터를 조립해 본 사람이라면 다들 쉽게 알고 있을 것이다. CPU를 꽂을 때 얼마나 손이 아프고 힘이 드는지.. CPU가 수행하는 명령은 보통 아주 단순하다. 명령은 register라고 불리는 CPU 그 자체에 있는 특별한 저장 영역에 있는 데이터를 요구한다. 보통 레지스터는 데이터를 저장할 수 있는 작은 영역이며, 파일을 저장하기엔 너무 작고, 대신 프로그램이 실행되는 동안 어떤 정보를

저장하기 위해 사용된다. CPU는 메모리에 있는 데이터보다 레지스터에 있는 데이터에 훨씬 더 빨리 접근할 수 있다. 하지만 레지스터에 있는 번호는 제한되어 있다. 이제 레지스터의 종류에 대해 알아보자. 시스템 해킹을 공부하는데 있어서 필수적인 부분이므로 반드시 완벽한 이해를 해야 한다.

일반적인 목적을 위한 레지스터:

[참고] 모든 일반적인 목적을 위한 레지스터는 16비트로 되어 있으며, 8비트 레지스터로 나누어질 수 있다. 예를 들어, AX는 AL 그리고 AH로 나누어질 수 있다. L은 low를 의미하고, H는 high를 의미한다. 만약 AX에 어떤 값을 부여하면 AH가 그 값의 첫 부분을 포함하고, AL은 마지막 부분을 포함한다. 예를 들어, 만약 AX에 DEAD를 부여하면 AH는 DE를, AL은 AD를 포함할 것이다. 마찬가지로 AH에 DE를, AL에 AD를 부여하면 AX는 DEAD를 포함할 것이다.

종류	구성	용도
AX	AH, AL	수학 연산, I/O 연산, INT 21
BX	BH, BL	Base 또는 Pointer
CX	CH, CL	루프 및 반복
DX	DH, DL	다양한 데이터, 문자 출력

386이 나왔을 때 4개의 새로운 레지스터가 추가되었는데, 그것은 EAX, EBX, ECX, EDX이다. 여기서 E는 32비트로 ‘Extended’(확장됨)를 의미한다. 다음 표를 보자. 각 공간은 8비트이다. 그리고 알아둘 것은 EAH 또는 EAL이라는 것은 없다.

```
|         EAX        |

+----+----+----+----+
|    |    | AH | AL |
+----+----+----+----+

          |   AX    |
```

세그먼트 레지스터:

CS(Code Segment) - 코드를 저장하는 메모리 블록

DS(Data Segment) - 데이터를 저장하는 메모리 블록

EX(Extra Segment) - 비디오와 관련된 것을 위해 사용됨

SS(Stack Segment) - 루틴으로부터 리턴 어드레스를 저장하기 위해 프로세서에 의해 사용되는 레지스터

인덱스 레지스터:

SI(Source Index) - 문자열/배열의 소스를 지정하기 위해 사용됨.

DI(Destination Index) - 문자열/배열의 목적지를 지정하기 위해 사용됨.

IP(Instruction Pointer) - 다음 명령의 주소를 저장하고, 그래서 직접적으로 변경될 수 없음.

스택 레지스터:

BP(Base Pointer) - 스택 오퍼레이션을 위해 SP와 연결되어 사용됨.

SP(Stack Pointer)

특별한 목적을 위한 레지스터:

IP(Instruction Pointer) - 실행된 명령의 offset를 가지고 있음.

Flag - 분기(branching)를 위해 사용됨. 플래그 레지스터는 크기가 1비트이다.

기본 ASM

이제부터 나올 내용을 시스템에 해킹 공부에서 필수적인 부분이다. GDB를 이용해 리턴 어드레스를 찾고, 쉘코드를 작성할 때 등등 반드시 알고 있어야 한다. 이 부분을 모르고 있다면 시스템 해킹 공부는 거의 불가능할 것이다. 하지만 그렇게 어려운 것은 아니므로 겁먹지 말고 알아보자. 아마 ‘이렇게 쉬었단 말인가’라고 외칠지도 모르겠다. 혹시라도 외치지 못하면 외칠 수 있을 때까지 공부해야 한다. 먼저 기본적인 레지스터 조작에 대해 알아보자.

우리는 앞에서 레지스터가 무엇인지 알아보았고, 이제 그 레지스터를 수정하는 방법에 대해 알아볼 것이다. 먼저 MOV 연산자의 사용법에 대해 알아보자. 레지스터에 어떤 값을 부여하기 위해 다음과 같이 한다.

```
MOV 목적지, 값
```

몇 가지 예를 들어보자.

AX를 56h과 같게 하려면:

```
MOV AX,56h     ;AX의 값이 56h이 되었다.
```

값으로 다른 레지스터를 사용하려면:

```
MOV AX,BX      ;AX의 값이 BX의 값으로 대체 되었다.
               ;만약 BX의 값이 45h라면 AX의 값은 45h가 된다.
```

이제 XCHG 연산자의 용법에 대해 알아보자. XCHG 는 단순히 두 레지스터를 서로 바꾼다. XCHG 라는 말은 영어 단어 exchange 에서 나온 것이다. 용법은 다음과 같다.

XCHG 레지스터 1, 레지스터 2

예를 보자.

```
MOV DX,56h
MOV AX,3Fh
XCHG DX,AX
```

이 예에서 DX 는 56h 이고, AX 는 3Fh 인데, 서로의 값을 교환함으로써(XCHG) DX 는 3Fh 가 되었으며, AX 는 56h 가 되었다.

참고: 8 비트(h/l) 레지스터와 16 비트(X)와 교환하지 않도록 해야 한다. 다음 코드는 무효이다.

```
XCHG AH,BX
```

다음은 INC 와 DEC 에 대해 알아보자. 눈치가 빠른 사람은 벌써 알 수 있을 것이다. INC 는 영어 단어 increase 에서, DEC 는 decrease 에서 나온 것이다. 예를 보자.

```
MOV DX,50h      ;DX 의 값은 50h 으로 설정
INC DX          ;DX 의 값을 증가시켜 이제 51h 가 되었다. 즉, DX++

MOV DX,50h      ;DX 의 값은 50h 로 설정
DEC DX          ;DX 의 값을 감소시켜 4F 가 되었다.(50h - 1h = 4Fh) 즉, DX--
```

이제 스택 조작에 대해 알아보자. 스택 조작에 사용되는 스택 연산자에는 6 개가 있으며, 그 중 2 개가 가장 많이 사용된다. 그것은 POP 과 PUSH 이다. 이것들의 문법은 다음과 같다.

POP 레지스터
PUSH 레지스터

나중에 사용하기 위해 스택에 있는 AX 에 값을 일시적으로 저장하고자 한다고 가정하자. 이럴 경우 간단히 다음과 같이 하면 된다.

```
PUSH AX
```

그리고 원래의 값을 회복하고자 한다면 POP 을 사용하면 된다.

```
POP AX
```

주목할 것은 스택이 단지 16 비트 레지스터만 받아들인다는 것이다.
이제 여태까지 언급한 것을 다음 예를 통해 연습해보자. 최종 AX 와 BX 값이 얼마인지 확인해보아라.

```
MOV AX,51h
MOV BX,4Fh

XCHG AX,BX

PUSH AX
MOV AX,34h

POP BX

PUSH BX

POP AX
```

값이 얼마 나오는가? AX 는 4Fh, BX 는 4Fh 가 나와야 한다. 다시 확인해보자.

```
MOV AX,51h      ;AX 의 값이 51h 이 되었다.
MOV BX,4Fh      ;BX 의 값이 4Fh 가 되었다.

XCHG AX,BX      ;AX 의 값과 BX 의 값이 변경되었다. AX=4Fh 로, BX 의 값이 51h

PUSH AX         ;AX 의 값을 저장
MOV AX,34h      ;AX 의 값에 34h 를 넣어둠

POP BX          ;BX 의 원래 값을 회복. 즉, BX = 4Fh
```

```
PUSH BX        ;BX 의 값이 4Fh

POP AX         ;AX = 34h 에서 이전의 값을 회복. 즉, 다시 4Fh 가 됨.
```

이제 수학 연산자 조작에 대해 알아보자. 수학 연산자에 기본 4 가지 ADD, SUB, MUL, DIV 가 있다. 먼저 ADD 에 대해 알아보자. ADD 의 문장 구조는 다음과 같다.

ADD 레지스터 1, 레지스터 2
ADD 레지스터, 값

```
MOV AX,5h      ;AX 의 값은 5h
MOV BX,4h      ;BX 의 값은 4h

ADD AX,BX      ;AX 에 BX 를 더하고, 그 값을 AX 에 저장(5h + 4h = 9h = AX)
```

다른 예를 들어보자.

```
MOV AX,5h

ADD AX,4h      ;AX = 5h, 여기에 4h 를 더함. 결국 AX 의 값은 5h + 4h 가 된다.
```

이제 SUB 에 대해 알아보자. 문장 구조는 다음과 같다.

SUB 레지스터 1, 레지스터 2
SUB 레지스터, 값

```
MOV BX,4Fh     ;BX 의  값은  4Fh

SUB BX,5h      ;BX 의 값에서 5h 를 뺌. 결국 4A 가 됨.
```

혹시 이 계산이 아직 어려운 사람은 다음과 같이 각 수를 먼저 10 진수로 전환한 후 그 10 진수를 계산하고, 그런 다음 16 진수로 전환해본다. 16 진수, 10 진수 전환이 가능한 간단한 계산기가 옆에 있으면 좋을 것이다.

$4F_{16} = 79_{10}$, $5_{16} = 5_{10}$, $79_{10} - 5_{10} = 74_{10} = 4A_{16}$
4Fh - 5h = 4A

이제 MUL 에 대해 알아보자. 문장 구조는 다음과 같다. 여기서는 단지 하나의 operand 만 필요하다. 이것은 프로세스가 주어진 레지스터를 AX 또는 AH 로 곱하기를 원한다고 추정하기 때문이다.

MUL 레지스터

```
MOV AX,5h
MOV BX,4Fh

MUL BX
```

이것은 AX 의 값을 결국 18B(4Fh x 5h = 18B)가 되게 한다.

마지막으로 DIV 연산자에 대해 알아보자. 문장 구조는 다음과 같다.

DIV 레지스터

MUL 과 마찬가지로 하나의 operand 가 있다. 예를 들어 보자.

```
MOV AX,5h
MOV BX,4Fh

DIV BX
```

이제 AX 의 값은 Fh 가 되었다(4Fh / 5h = Fh) 아래는 값이 나온 과정이다.

```
4Fh = 79
5h =  5
79 / 5 = 15.8

15 = Fh
```

이제 Bit 연산자들 AND, OR, XOR, NOT 에 대해 알아보자. 이 연산자들이 하는 것은 두 값을 bit 단위로 비교를 하는 것이다. 유용하니 꼭 알아두자. 혹시 오버플로우 공격을 위해 쉘코드를 만들어 본 사람이면 지금 이 부분이 얼마나 중요한지 알 것이다.

AND 의 문장 구조:

AND 레지스터 1, 레지스터 2

AND 레지스터, 값

AND 는 만약 두 오퍼랜드가 1 일 경우만 1 을 리턴한다. 이게 무슨 말인가? 다음 예를 들어보자.

```
MOV AX,5h
MOV BX,6h

AND AX,BX          ;AX 의 값이 4 가 됨.
```

AX 의 값이 최종 4 가 된 것은 다음과 같은 이유에서이다. 먼저 16 진수를 bit 단위로 비교하기 위해 2 진수로 변환한다.

```
5h = 101b
6h = 110b

101b
110b
---
100b   = 4h
```

이것은 아래의 AND truth 테이블을 보면 쉽게 알 수 있을 것이다. 앞에서도 말했듯이 오퍼랜드가 둘 다 1 일 때만 1 을 리턴 한다고 했었다.

```
AND truth table:
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1
```

OR 문장구조:

OR 레지스터 1, 레지스터 2

OR 레지스터, 값

OR 는 오퍼랜드 중 어느 하나라도 1 이면 1 을 리턴한다. 예를 보자.

```
MOV AX,5h
MOV BX,6h

OR AX,BX
```

AX 의 값은 이제 7h 가 되었다.

```
5h = 101b
6h = 110b

101b
110b
----
111b

111b = 7h
```

OR truth table:

```
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
```

XOR 문장 구조:

XOR 레지스터 1, 레지스터 2

XOR 레지스터, 값

XOR 는 오퍼랜드가 만약 어느 하나가 1 이면 1 을 리턴하고, 둘 다 1 이면 0 을 리턴한다. 예를 보자.

```
MOV AX,5h
MOV BX,6h
```

XOR AX,BX

최종 값이 나오는 과정을 보자.

5h = 101b

6h = 110b

101b

110b

011b

11b = 3h

XOR truth table:

0 XOR 0 = 0

1 XOR 0 = 1

0 XOR 1 = 1

1 XOR 1 = 0

이제 마지막으로 NOT 에 대해서 알아보자. NOT 은 값을 역전시킨다. 아주 쉬운 것이다. 문장 구조는 다음과 같다.

NOT 레지스터

NOT 값

예를 보자.

MOV AX,F0h

NOT AX

AX 는 F 가 되었다. F0h = 11110000 이고, 이것을 거꾸로 하면 00001111 가 될 것이고, 결국 F 가 된 것이다.

NOT truth table:

NOT 1 = 0

NOT 0 = 1

80x86 명령어들

Name	Description	Formats	Flags O S Z A P C
ADC	Add with Carry	O2	C C C C C C
ADD	Add Integers	O2	C C C C C C
AND	Bitwise AND	O2	0 C C ? C 0
CALL	Call Routine	R M I	
CBW	Convert Byte to Word		
CDQ	Convert Dword to Qword		
CLC	Clear Carry 0		
CLD	Clear Direction Flag		
CMC	Complement Carry C		
CMP	Compare Integers	O2	C C C C C C
CMPSB	Compare Bytes		C C C C C C
CMPSW	Compare Words		C C C C C C
CMPSD	Compare Dwords		C C C C C C
CWD	Convert Word to Dword into DX:AX		
CWDE	Convert Word to Dword into EAX		
DEC	Decrement Integer	R M	C C C C C
DIV	Unsigned Divide	R M	? ? ? ? ? ?
ENTER	Make stack frame I,0		
IDIV	Signed Divide	R M	? ? ? ? ? ?
IMUL	Signed Multiply	R M	C ? ? ? ? C
		R16,R/M16	
		R32,R/M32	
		R16,I	
		R32,I	
		R16,R/M16,I	
		R32,R/M32,I	
INC	Increment Integer	R M	C C C C C
INT	Generate Interrupt	I	
JA	Jump Above	I	
JAE	Jump Above or Equal	I	
JB	Jump Below	I	

JBE	Jump Below or Equal	I
JC	Jump Carry	I
JCXZ	Jump if CX = 0	I
JE	Jump Equal	I
JG	Jump Greater	I
JGE	Jump Greater or Equal	I
JL	Jump Less	I
JLE	Jump Less or Equal	I
JMP	Unconditional Jump R M	I
JNA	Jump Not Above	I
JNAE	Jump Not Above or Equal	I
JNB	Jump Not Below	I
JNBE	Jump Not Below or Equal	I
JNC	Jump No Carry	I
JNE	Jump Not Equal	I
JNG	Jump Not Greater	I
JNGE	Jump Not Greater or Equal	I
JNL	Jump Not Less	I
JNLE	Jump Not Less or Equal	I
JNO	Jump No Overflow	I
JNS	Jump No Sign	I
JNZ	Jump Not Zero	I
JO	Jump Overflow	I
JPE	Jump Parity Even	I
JPO	Jump Parity Odd	I
JS	Jump Sign	I
JZ	Jump Zero	I
LAHF	Load FLAGS into AH	
LEA	Load Effective Address	R32,M
LEAVE	Leave Stack Frame	
LODSB	Load Byte	
LODSW	Load Word	
LODSD	Load Dword	
LOOP	Loop	I
LOOPE/LOOPZ	Loop If Equal	I
LOOPNE/LOOPNZ	Loop If Not Equal	I

MOV	Move Data	O2	
		SR,R/M16	
		R/M16,SR	
MOVSB	Move Byte		
MOVSW	Move Word		
MOVSD	Move Dword		
MOVSX	Move Signed	R16,R/M8	
		R32,R/M8	
		R32,R/M16	
MOVZX	Move Unsigned	R16,R/M8	
		R32,R/M8	
		R32,R/M16	
MUL	Unsigned Multiply	R M	C ? ? ? ? C
NEG	Negate	R M	C C C C C C
NOP	No Operation		
NOT	1’s Complement	R M	
OR	Bitwise OR	O2	0 C C ? C 0
POP	Pop From Stack	R/M16	
		R/M32	
POPA	Pop All		
POPF	Pop FLAGS		C C C C C C
PUSH	Push to Stack	R/M16	
		R/M32 I	
PUSHA	Push All		
PUSHF	Push FLAGS		
RCL	Rotate Left with Carry	R/M,I	
		R/M,CL	
			C C
RCR	Rotate Right with Carry	R/M,I	C C
		R/M,CL	
REP	Repeat		
REPE/REPZ	Repeat If Equal		
REPNE/REPNZ	Repeat If Not Equal		
RET	Return		
ROL	Rotate Left	R/M,I	C C
		R/M,CL	

ROR	Rotate Right	R/M,I R/M,CL	C C
SAHF	Copies AH into FLAGS		C C C C C
SAL	Shifts to Left	R/M,I R/M, CL	C
SBB	Subtract with Borrow	O2	C C C C C C
SCASB	Scan for Byte		C C C C C C
SCASW	Scan for Word		C C C C C C
SCASD	Scan for Dword		C C C C C C
SETA	Set Above	R/M8	
SETAE	Set Above or Equal	R/M8	
SETB	Set Below	R/M8	
SETBE	Set Below or Equal	R/M8	
SETC	Set Carry	R/M8	
SETE	Set Equal	R/M8	
SETG	Set Greater	R/M8	
SETGE	Set Greater or Equal	R/M8	
SETL	Set Less	R/M8	
SETLE	Set Less or Equal	R/M8	
SETNA	Set Not Above	R/M8	
SETNAE	Set Not Above or Equal	R/M8	
SETNB	Set Not Below	R/M8	
SETNBE	Set Not Below or Equal	R/M8	
SETNC	Set No Carry	R/M8	
SETNE	Set Not Equal	R/M8	
SETNG	Set Not Greater	R/M8	
SETNGE	Set Not Greater or Equal	R/M8	
SETNL	Set Not Less	R/M8	
SETNLE	Set Not LEss or Equal	R/M8	
SETNO	Set No Overflow	R/M8	
SETNS	Set No Sign	R/M8	
SETNZ	Set Not Zero	R/M8	
SETO	Set Overflow	R/M8	
SETPE	Set Parity Even	R/M8	
SETPO	Set Parity Odd	R/M8	
SETS	Set Sign	R/M8	

SETZ	Set Zero	R/M8		
SAR	Arithmetic Shift to Right	R/M,I		C
		R/M, CL		
SHR	Logical Shift to Right	R/M,I		C
		R/M, CL		
SHL	Logical Shift to Left	R/M,I		C
		R/M, CL		
STC	Set Carry 1			
STD	Set Direction Flag			
STOSB	Store Btye			
STOSW	Store Word			
STOSD	Store Dword			
SUB	Subtract	O2	C C C C C C	
TEST	Logical Compare	R/M,R	0 C C ? C 0	
		R/M,I		
XCHG	Exchange	R/M,R		
		R,R/M		
XOR	Bitwise XOR	O2	0 C C ? C 0	

이제까지 아주 어셈블리어 공부에서 아주 기본적인 것들만 알아보았다. 이 정도만 해도 디버깅 할 때나 디버깅한 내용을 살펴볼 때 어느 정도 도움은 될 것이라고 생각한다. 성질 급한 사람들은 스스로 다른 자료들을 찾아서 공부하자. 이제부터 좀더 깊은 분야로 들어가 보자. 이 글은 "초보자들을 위한 어셈블리어 기초-2" 에서 계속 된다. 아마도 이 글에서는 좀더 어셈블리어에 초점을 맞추기 보다는 시스템 해킹을 위한 부분에 더 초점을 맞춰질지도 모르겠다. 며칠 기다려보자. 무엇에 대해 쓸지... 마지막으로 이 글은 http://www.drpaulcarter.com/pcasm/를 많이 참조했다.

To Be Continued !!!