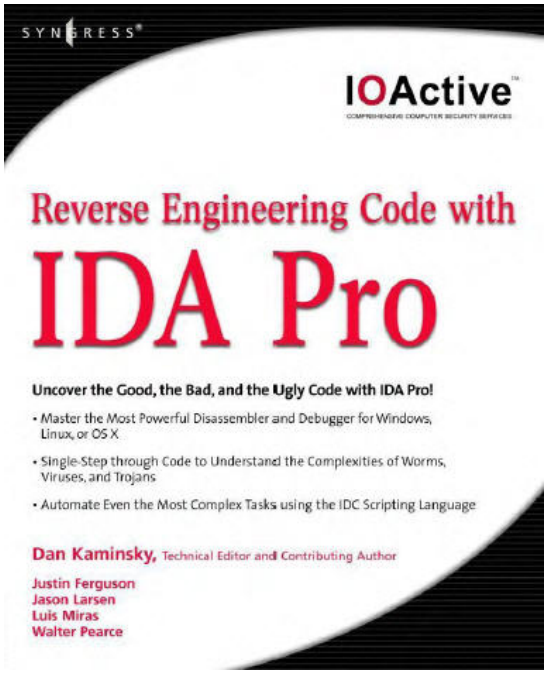
Reverse Engineering Code with IDA Pro



By Dan Kaminsky, Justin Ferguson, Jason Larsen, Luis Miras, Walter Pearce

정리: vangelis(securityproof@gmail.com)

이 글은 ***Reverse Engineering Code with IDA Pro***(2008년 출판)라는 책을 개인적으로 공부하면서 정리한 것입니다. 목적이 책 소개가 아니라 공부이므로 글의 내용은 형식에 국한되지 않고 경우에 따라 책의 내용 일부를 편역하거나 또는 철저하게 요약해서 올리고, 경우에 따라 필자가 내용을 추가할 것입니다. 내용에 오류 및 오타가 있다면 지적 부탁 드립니다.

2장 Assembly and Reversing Engineering Basics

개관[1]

이 장에서는 리버싱 엔지니어링을 위해 필요한 어셈블리어, Intel 아키텍처 프로세서, 명령어 등 기본적인 요소들을 소개하고 있다. 이 책은 32-bit Intel architecture(IA-32) 어셈블리어에 초점을 맞추고 있으며, 운영체계는 Windows와 Linux를 다루고 있다. 독자들은 IA-32 어셈블리어와 C/C++에 대한 이해도가 필요하다. 그렇다고 해서 이 장에서 관련 항목을 깊게 다루지는 않고 있다.

어셈블리어와 IA-32 프로세서[2]

어셈블러(assembler)들은 다른 문장 구조(syntax)를 가지고 있다. 가장 대표적인 Intel과 AT&T 문장 구조인데, 이 책에서는 Intel 문장 구조를 사용하고 있다. 이것은 IDA에 의해 사용되는 것이 Intel 문장 구조이기 때문인 것도 있고, 각종 문서나 white paper에서도 더 많이 사용되기 때문이다.

Intel과 AT&T 문장 구조의 차이점에 대해 다음 예를 보자. 다음은 간단한 "Hello, World" 프로그램이다. 첫 번째 것은 Red Hat 리눅스에서 gcc로 컴파일하고, gdb를 통해 디스어셈블링 한 것이다. Gdb를 통해 나오는 결과는 AT&T문장 구조이다.

```
(gdb) disas main
Dump of assembler code for function main:
0x8048328 <main>:      push   %ebp
0x8048329 <main+1>:    mov    %esp,%ebp
0x804832b <main+3>:    sub    $0x8,%esp
0x804832e <main+6>:    and    $0xfffffff0,%esp
0x8048331 <main+9>:    mov    $0x0,%eax
0x8048336 <main+14>:   sub    %eax,%esp
0x8048338 <main+16>:   sub    $0xc,%esp
0x804833b <main+19>:   push   $0x8048398
0x8048340 <main+24>:   call   0x8048268 <printf>
0x8048345 <main+29>:   add    $0x10,%esp
0x8048348 <main+32>:   mov    $0x0,%eax
```

---

[1] 2장에서는 시스템 해킹을 공부한 사람이라면 많이 접했던 분야이다. 그래서 아주 간단하게 정리를 할 생각이다.

[2] 이 섹션에서 "Assembly refers to the use of instruction mnemonics..."라는 부분이 나온다. 여기서 'mnemonics'는 같은 기능을 가진 instruction opcode 클래스의 예약된 이름이다. "mov eax, 0"라는 instruction의 포맷을 살펴보면, mov가 mnemocic이고, eax와 0은 operand가 되는 아규먼트들이다. 몇 개의 오퍼랜드를 가질 것인가는 opcode에 따라 결정되며, 오퍼랜드의 수는 0 ~3까지이다.

```
0x804834d <main+37>:    leave
0x804834e <main+38>:    ret
0x804834f <main+39>:    nop
End of assembler dump.
(gdb)
```

다음은 Dev-C++에서 컴파일 한 것을 IDA Pro로 본 것이다.

```
;
; +-------------------------------------------------------------------------+
; |    This file is generated by The Interactive Disassembler (IDA)       |
; |    Copyright (c) 2007 by DataRescue sa/nv, <ida@datarescue.com>       |
; | Licensed to: Mach EDV Dienstleistungen, Jan Mach, 1 user, adv, 11/2007|
; +-------------------------------------------------------------------------+
;
; Input MD5   : 88FEF2313AAAB6E089DCD55EF42C7804

; File Name   : C:\Documents and Settings\free2\바탕 화면\test.exe
; Format      : Portable executable for 80386 (PE)
; Imagebase   : 400000
; Section 1. (virtual address 00001000)
; Virtual size                : 000008D4 (   2260.)
; Section size in file        : 00000A00 (   2560.)
; Offset to raw data for section: 00000400
; Flags 60000060: Text Data Executable Readable
; Alignment     : default
; OS type        :  MS Windows
; Application type:  Executable 32bit

.686p
.mmx
.model flat
.intel_syntax noprefix

- 중략 -

push    ebp
mov     ebp, esp
sub     esp, 8
```

```
and     esp, 0FFFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    ___chkstk
call    ___main
mov     [esp+8+var_8], offset aHello ; "Hello"
call    printf
mov     eax, 0
leave
retn
_main endp
```

- 중략 -

결과를 보면 파란색으로 표시한 **.intel_syntax noprefix** 부분을 보면 Intel 문장 구조를 사용하고 있음을 알 수 있다. printf() 함수가 호출되는 부분부터 두 문장 구조를 비교해보자.

[AT&T 문장구조]

```
0x8048340 <main+24>:    call   0x8048268 <printf>
0x8048345 <main+29>:    add    $0x10,%esp
0x8048348 <main+32>:    mov    $0x0,%eax  //책에서는 xor %eax, %eax
0x804834d <main+37>:    leave
0x804834e <main+38>:    ret
```

[Intel 문장구조]

```
mov     [esp+8+var_8], offset aHello ; "Hello"
call    printf
mov     eax, 0   //책에서는 xor eax, eax
leave
retn
```

책에서는 "xor %eax, %eax"를 사용하고 있다. 이 연산 결과는 잘 알겠지만 eax 레지스터의 값

이 0이 된다. xor를 사용하고, 두 오퍼랜드가 같으면 그 오퍼랜드의 값은 0이 된다. 그리고 "mov $0x0,%eax"의 결과를 보면 eax 레지스터에 0의 값을 복사한다. 결과는 당연 eax 레지스터의 값은 0이 된다. 책에 나오는 것과 필자의 테스트에서 나온 결과는 같은 것이다.

우리가 주목할 것은 위의 두 결과를 보면 source operand와 destination operand의 위치가 다르다는 것이다. 정리하면 다음과 같다.

* AT&T 문장구조 – **명령어 source operand, destination operand**

```
mov     $0x0,       %eax
```

* Intel 문장구조 – **명령어 destination operand, source operand**

```
mov     eax,        0
```

다음 부분은 C언어에서는 "return 0;"을 나타낸다.

```
mov    eax, 0   // mov    $0x0,%eax
leave
retn            // ret
```

프로세서는 'return 0;'이라는 것을 이해할 수 없다. 그런 의미에서 어셈블리어는 인간이 읽을 수 있는 마지막 코드 층이라고 할 수 있다. 컴파일링 후에 코드를 어셈블링하면 'opcode'를 출력하는데, opcode는 어셈블리어 명령을 이진수 형태로 나타낸 것 또는 개별 명령을 실행하기 위해 필요한 on-off의 연속(sequence)이다. 보통 opcode는 이진수 보다는 16진수로 나타낸다.

```
0x8048348 <main+32>:    mov    $0x0,%eax
0x804834d <main+37>:    leave
0x804834e <main+38>:    ret
0x804834f <main+39>:    nop
End of assembler dump.
(gdb) x/8b 0x8048348
0x8048348 <main+32>:    0xb8    0x00    0x00    0x00    0x00    0xc9    0xc3    0x90
(gdb)
```

이것을 objdump를 이용해 좀더 알기 쉽게 확인할 수 있다.

```
8048348:      b8 00 00 00 00       mov    $0x0,%eax
804834d:      c9                   leave
804834e:      c3                   ret
804834f:      90                   nop
```

하지만 결국 컴퓨터의 프로세서에 전달되는 것은 1과 0의 연속이다. Opcode에 대한 좀더 자세한 자료는 ***Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A***[3]의 **section 2.1.2**를 참고하길 바란다.[4] 이 책은 http://www.intel.com/products/processor/manuals/에서 pdf 파일로 구할 수 있으며, 책으로 받으려면 http://www.intel.com/design/literature.htm를 참고하면 된다. 참고로 이 책들은 무료이다. 신청하면 늦어도 2주 안으로 미국에서 한국으로 책이 배달된다. 필자의 경우 매뉴얼 페이지에 있는 6 종류의 책을 신청하여 2개의 박스로 받았던 적이 있다.

다음은 **섹션 2.1.2**의 원문 전체를 첨부한 것이다.

****

2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte, or
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet)

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix for SSE/SSE2/SSE3 instructions (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes, or
- A mandatory prefix (66H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet)

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

****

[3] http://download.intel.com/design/processor/manuals/253666.pdf

[4] 어떤 책을 공부하다 보면 각종 책이나 참고 자료들이 제시된다. 공부과정에서 그 참고 자료들도 같이 구해서 본다면 공부하고 있는 책 한 권을 제대로 이해할 수 있을 뿐만 아니라 더 많은 것을 이해할 수 있게 될 것이다.

opcode에 대한 더 자세한 내용은 위의 첨부한 원문 끝 줄에 나오듯 Appendix A와 B에 나와 있다. 그리고 ***Reversing Engineering Code with IDA Pro***라는 이 책을 잘 이해하기 위해서는 ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** Volume 2*A, 2B* 두 권이 늘 필요할 것이다. 이 두 권은 instruction set reference에 대한 것이다.

Tools & Traps…

Shellcode는 보통 C언어에서 문자열 배열로 저장되는 일련의 opcode이다. 'shellcode'라는 용어를 사용하게 된 것은 그 일련의 opcode가 쉘(/bin/sh 또는 cmd.exe)을 실행하는데 필요한 명령어(instruction)들이기 때문이다.

C언어에서 쉘코드를 사용하려면 다음과 같이 한다는 것을 이 글의 독자들은 다 알고 있을 것이다.

unsigned char shellcode[] = "\xc9\x31\xc0\xc9";

쉘코드는 해킹에서 필수적인 요소이므로 관련 지식을 반드시 알아둘 필요가 있다. 쉘코드를 만드는 방법에 대해서는 국내외에 많은 자료들이 있으므로 그 자료들을 참고하길 바란다.

어셈블리어 명령어는 명령어에 따라 **아규먼트로 상수, 레지스터[5], 또는 메모리 변수를 가진다**. 이 때 아규먼트를 오퍼랜더(operand)라고 한다. 상수는 소스 코드에 정적으로 정의되어 있다. 예를 들어 다음을 보자.

```
mov eax, 0x123
```

여기서 16진수 0x123이 상수이며, 특별하게 상수에 대해 설명한 것은 없다. 참고로 책에 나오는 내용을 하나 여기서 언급한다. 책을 보면 "`mov eax, 0`"와 "`xor eax, eax`"의 결과는 같다. 그러나 여기에 차이점이 있다고 한다. Eax란 레지스터에 0이라는 값을 직접 복사하는 것이 xor을 이용해 레지스터의 값을 지정하는 것이 더 큰 명령이라고 한다. 이것은 아마도 eax라는 레지스터에 오퍼랜더 0을 복사하는 것이 하나의 레지스터를 오퍼랜더로 사용하는 것보다 가볍지 않기 때문일 것이다.

---

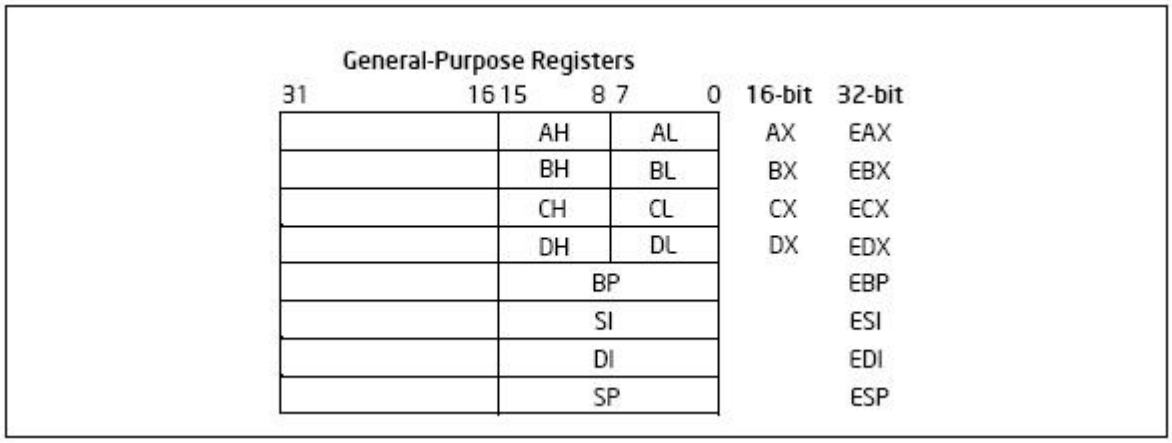
[5] 레지스터에 대해서는 ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** Volume 1의 **Chapter3**을 참고하길 바라며, 특히 레지스터에 대한 공부는 섹션 3.4 Basic Program Execution Register 이후 페이지를 정독하는 것이 필요하다. 그리고 이 책 2장의 나머지 부분에 대한 공부를 위해서는 ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** Volume 1의 Chapter 3부터 6까지의 공부가 필수인 것 같다.

이제는 아규먼트로 레지스터에 대해 알아보기로 한다. 그동안 시스템 해킹 공부를 많이 해본 경험이 있다면 익숙한 부분이다. 그렇더라도 복습과 초보자들을 위해 정리하기로 한다. 이 시점에서 ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** Volume 1[6]의 **Chapter3**을 읽을 필요가 있다.

레지스터는 C/C++언어에서 변수와 다소 유사하다. 범용 레지스터는 정수, 오프셋, 직접적인 값, 포인터 또는 32비트로 나타낼 수 있는 어떤 것을 포함할 수 있다. 범용 레지스터는 기본적으로 프로세서에 물리적으로 존재하는 사전에 할당된 변수이다. 범용 레지스터는 반복해서 재사용될 수 있고, C/C++ 프로그램에서는 변수는 한번만 사용된다는 점에서 우리가 '변수'라고 생각하는 것과는 약간 다르게 사용된다.

IA-32에서는 8개의 32-bit 범용 레지스터, 6개의 16-bit 세그먼트 레지스터, 1개의 32-bit 명령 포인터 레지스터(instruction pointer register), 1개의 32-bit 상태 레지스터(status register), 5개의 컨트롤 레지스터, 3개의 메모리 관리 레지스터, 8개의 디버그 레지스터 등이 있다. 리버싱에서 일반적인 경우에는 범용 레지스터, 세그먼트 레지스터, 명령 포인터 레지스터, 상태 레지스터만 다루게 된다. OS 드라이버 또는 그 유사한 것을 다룬다면 다른 레지스터들도 다루게 된다.

먼저 범용 레지스터에 대해 알아보자. 다음은 범용 레지스터를 간단하게 나타낸 것이다.



위의 그림을 보면 알 수 있듯이 32비트 범용 레지스터는 EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP 이상 8개이다. 책에서는 범용 레지스터에 대한 설명이 그렇게 잘 나와 있지 않다. 그래서 ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** Volume 1의 **Chapter3**을 참고할 것이다. 32비트 범용 레지스터는 다음 항목을 가진다.

- 논리적 그리고 수리적 오퍼레이션을 위한 오퍼랜드
- 주소 계산을 위한 오퍼랜드

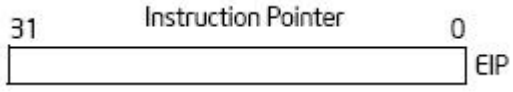
[6] http://download.intel.com/design/processor/manuals/253665.pdf

- 메모리 포인터

다음은 각 범용 레지스터의 특별한 용도에 대한 요약이다.

- **EAX** — 오퍼랜더와 결과 데이터에 대한 누산기(accumulator)
- **EBX** — DS segment에 있는 데이터에 대한 포인터
- **ECX** — 문자열과 loop 연산에 대한 counter
- **EDX** — I/O 포인터
- **ESI** — DS 레지스터가 가리키는 세그먼트에 있는 데이터에 대한 포인터; 문자열 오퍼레이션에 대한 source pointer
- **EDI** — ES 레지스터가 가리키는 세그먼트에 있는 데이터(또는 목적지)에 대한 포인터; 문자열 오퍼레이션에 대한 destination pointer
- **ESP** — Stack pointer (in the SS segment)

이제 명령 포인터(instruction pointer)인 EIP에 대해 알아보자.

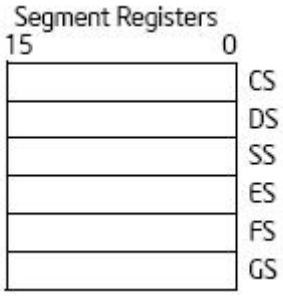


EIP 레지스터는 프로세서에 의해 실행될 다음 명령을 가리키며, 거의 대부분의 어플리케이션 기반의 공격의 목표는 이 EIP 레지스터를 통제하는 것이다. 아마도 스택 기반의 오버플로우 공격과 같은 시스템 해킹에 대해 공부를 한 사람이라면 이에 대해 잘 알 것이다. Intel 매뉴얼을 보면 "실행될 다음 명령에 대한 현재 코드 세그먼트에 있는 offset을 가지고 있다"라고 나와 있다.

EIP 레지스터는 소프트웨어 의해 직접적으로 접근이 가능하지는 않다. EIP 레지스터는 control-transfer 명령(jmp, jcc, call, ret과 같은), 인터럽트, exception에 의해서 절대적으로 통제된다. EIP 레지스터를 읽는 유일한 방법은 call 명령을 실행한 후 프로시저 스택으로부터 return 명령 포인터의 값을 읽는 것이다. EIP 레지스터는 프로시저 스택 상에서 return 명령 포인터의 값을 변경하고, return 명령(RET 또는 IRET)을 실행함으로써 간접적으로 로딩될 수 있다.

하지만 범용 레지스터와는 달리 직접적으로 수정될 수는 없다. 이것은 어떤 값을 레지스터로 이동시키기 위해 명령을 실행할 수 없고, ret 명령에 의해 따르는 스택 세그먼트에 push와 같은 것을 이용해 그것의 값을 간접적으로 수정하는 일련의 오퍼레이션을 수행해야 한다는 것을 의미한다.

이제 segment register에 대해 간단하게 알아보자.



위의 그림에서 보듯 6개의 16비트 세그먼트 레지스터들이 있는데, CS(code segment), DS(data segment), SS(stack segment), ES(extra segment), FS, GS가 있다. 세그먼트 레지스터들은 segment selector라고 부르는 것에 대한 포인터를 가지고 있으며, offset을 가지는 것으로부터 base address로써 사용되기도 한다. 다음 예를 보자.

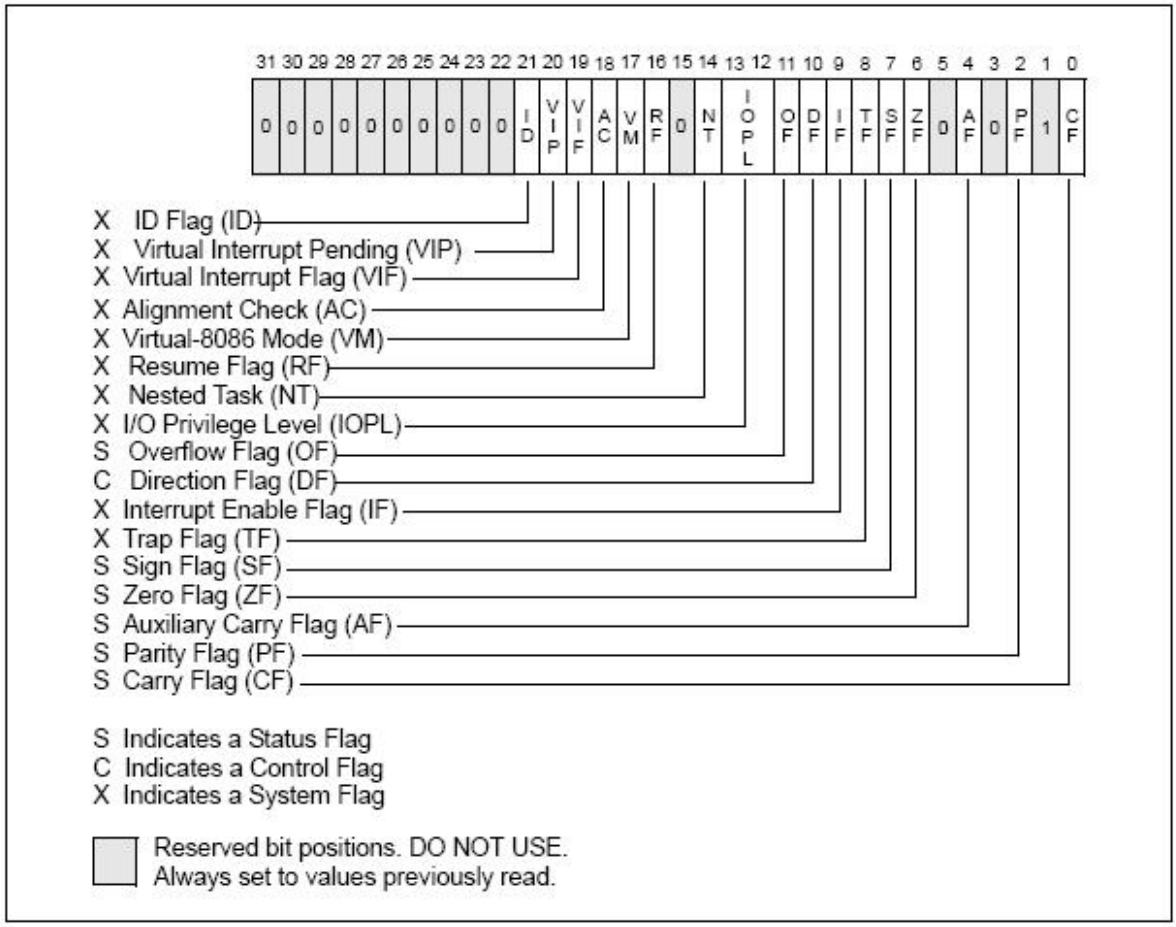
```
mov DS:[eax],  ebx
```

이 명령에서 ebx 레지스터의 내용이 eax에 의해 지정된 data segment에 대한 offset으로 복사된다. 쉽게 말하면, "`DS:[eax]`" 부분은 DS의 주소에 eax의 내용을 더한 것이라고 생각하면 된다. segment selector라는 것은 세그먼트들에 대한 16 bit로 된 식별자(identifier)이다. 세그먼트 레지스터는 세그먼트를 확인하는 8192개의 가능한 세그먼트 기술자(descriptor)들 중 하나를 확인하는데 사용되는 segment selector를 가리킨다.

segment selector는 상대적으로 단순한 구조체이다. 3에서 15까지 bit는 descriptor table(3개의 메모리 관리 레지스터들 중의 하나)들 중의 하나에 대한 index로 사용되고, bit 2는 어떤 descriptor table을 정확하게 지정할지, 그리고 bit 1과 0은 요청된 권한 레벨(0 ~ 3까지의 권한 레벨은 뒤에서 다루며, rootkit 공부에도 종종 등장하는 부분이기도 하다. 아마도 Phrack 65호의 "Stealth Hooking: another way to subvert the Windows kernel"를 보면 이에 대한 언급이 있었던 것 같다.)을 지정한다.

이제 레지스터 중에서 EFLAGS 레지스터에 대해 알아보자. 책에서는 한 문단으로만 설명을 하고 있어 내용이 많이 부실하다. 자세한 설명이 없는 이유는 현재 장에서는 그렇게 큰 의미를 가지지 못하기 때문이라고 한다. 그러나 공부를 위해서 Intel 매뉴얼을 참고하여 더 알아보기로 하겠다.

32-bit EFLAGS 레지스터는 status flag, control flag, 그리고 system flag의 그룹을 포함하고 있다. Status flag는 아래 그림에서 bit 0, 2, 4, 6, 7, 11이며, add, sub, mul, div 명령과 같은 계산 명령의 결과를 나타낸다. 이 status flag들 중 CF(bit 0)만이 stc, clc, cmc 명령을 사용해 직접 변경될 수 있다.



다음은 Intel 매뉴얼에서 발췌한 Status Flag의 설명 부분이며, 별도로 번역은 하지 않으니 참고하길 바란다.

CF (bit 0) Carry flag — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared

PF (bit 2) Parity flag — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

AF (bit 4) Adjust flag — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

ZF (bit 6) Zero flag — Set if the result is zero; cleared otherwise.

SF (bit 7) Sign flag — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

OF (bit 11) Overflow flag — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

DF 플래그(direction flag)는 EFLAG 레지스터의 bit 10에 위치하며, 문자열 명령들(movs, cmps, scas, lods, stos)을 통제한다. DF Flag를 설정하는 것은 문자열 명령들이 자동 감소(auto-decrement, 높은 주소에서 낮은 주소로 문자열을 처리)하도록 한다. DF Flag를 클리어하는 것은 문자열 명령들이 자동 증가(낮은 주소에서 높은 주소로 문자열을 처리)하도록 한다.

System Flag들과 IOPL 필더는 운영체제 또는 executive 오퍼레이션을 통제한다. 이 flag들은 어플리케이션 프로그램에 의해 수정되어서는 안된다. System flag들의 기능들은 다음과 같다. 역시 Intel 매뉴얼에서 발췌했다.

TF (bit 8) Trap flag — Set to enable single-step mode for debugging; clear to disable single-step mode.

IF (bit 9) Interrupt enable flag — Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.

IOPL (bits 12 and 13)

I/O privilege level field — Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. This field can only be modified by the POPF and IRET instructions when operating at a CPL of 0.

NT (bit 14) Nested task flag — Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when the current task is not linked to another task.

RF (bit 16) Resume flag — Controls the processor's response to debug exceptions.

VM (bit 17) Virtual-8086 mode flag — Set to enable virtual-8086 mode; clear to return to protected mode without virtual-8086 mode semantics.

AC (bit 18) Alignment check flag — Set this flag and the AM bit in the CR0 register to enable alignment checking of memory references; clear the AC flag and/or the AM bit to disable alignment checking.

VIF (bit 19) Virtual interrupt flag — Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)

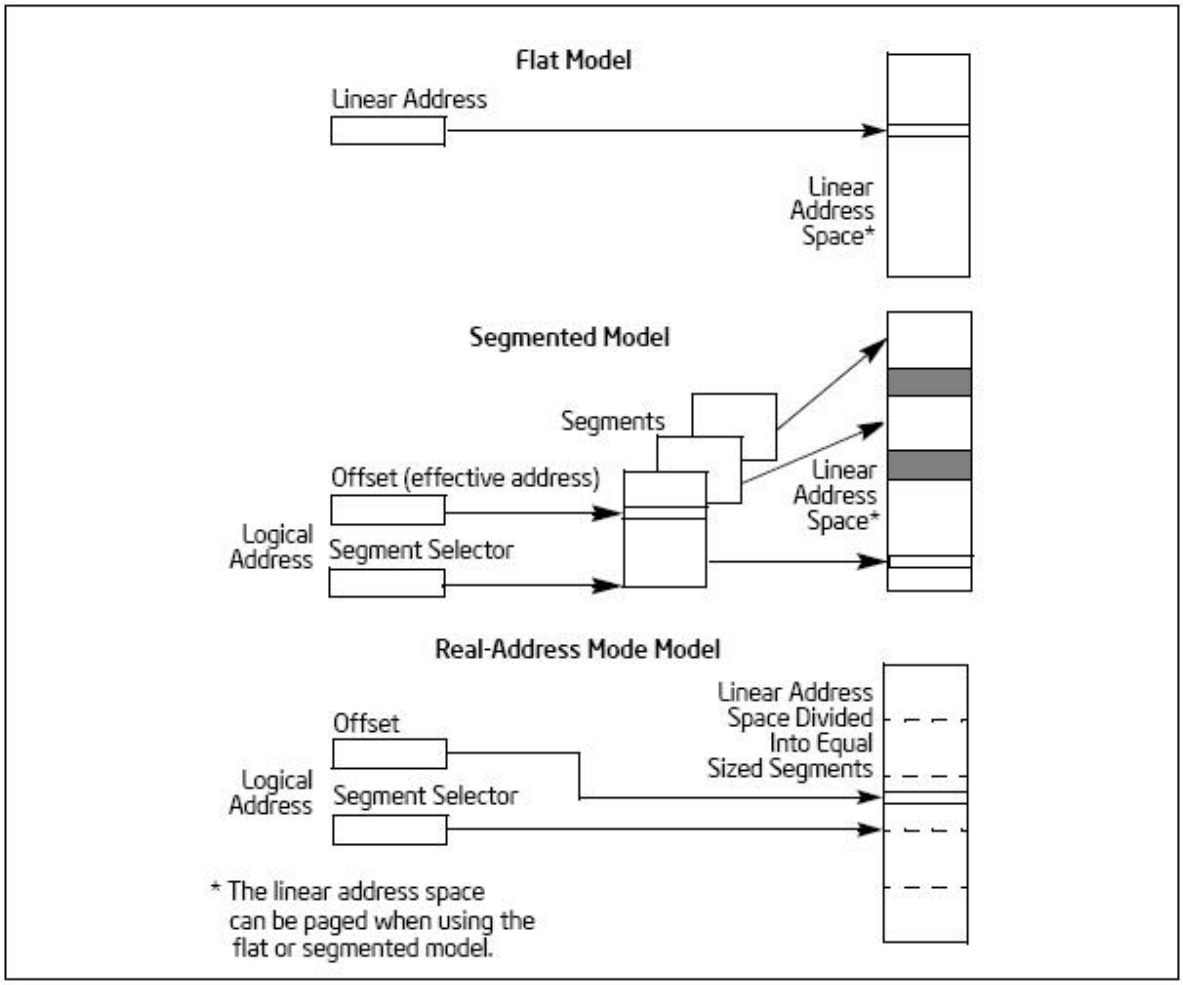
VIP (bit 20) Virtual interrupt pending flag — Set to indicate that an interrupt is pending; clear when no interrupt is pending. (Software sets and clears this flag; the processor only reads it.) Used in conjunction with the VIF flag.

ID (bit 21) Identification flag — The ability of a program to set or clear this flag indicates support for the CPUID instruction.

이제까지 어셈블리어 명령어가 가지는 아규먼트로 상수와 레지스터 오퍼랜드에 대해 알아보았다. 이제 **메모리 오퍼랜드**에 대해 알아보자.

메모리 오퍼랜드는 높은 수준의 프로그래머는 대체로 변수로 생각한다. 즉, C나 C++ 같은 언어에서 변수를 선언하면 보통 메모리에 존재하고, 그래서 종종 메모리 오퍼랜드가 된다. 이것은 포인터를 통해 보통 접근한다. 개념 그 자체는 아주 간단한데, 어떤 일이 일어나는지 제대로 이해하기 위해서는 메모리의 주소 체계에 대한 더 깊은 지식이 필요한데, 이것은 메모리 모델(memory model), 오퍼레이션 모드(operation mode), 그리고 권한 레벨(privilege level)에 의존하게 된다.

IA-32 메모리 모델에는 flat memory model, segmented memory model, 그리고 real-address mode memory model이 있다. 다음은 메모리 모델의 도식도이다.



메모리 모델에 대해 좀더 자세한 것은 Intel 매뉴얼을 참고하길 바란다.

IA-32에서 3개의 기본 오퍼레이션 모드가 있다. 3개의 기본 모드는 protected mode, real-address mode, 그리고 system management mode(SMM)이다.

보호 모드에서 ring이라고 불리는 4개의 권한 레벨이 있으며, 4개의 ring은 0에서부터 3까지 확인 번호가 있다. 가장 낮은 번호가 가장 높은 권한을 가지고 있다. Ring-0은 운영체제를 위해 사용되

고, ring-3은 보통 어플리케이션을 위해 사용된다.

권한 레벨들은 시스템 데이터의 더 많은 신뢰관계가 필요한 구성요소들에 접근을 제한한다. 예를 들어, ring-3 어플리케이션은 ring-2 구성요소의 데이터에 접근할 수 없다. 하지만, ring-2 구성요소는 ring-3 구성요소의 데이터에 접근할 수 있다. 이것이 Windows나 Linux 커널로부터 데이터를 임의로 읽을 수는 없으나 커널은 사용자의 데이터를 읽을 수 있는가의 이유이다.

이제 알아보아야 할 것은 바이너리 실행파일의 스택, 힙, 그리고 다른 섹션에 대한 것이다. 해킹 공부를 한 사람이라면 수 없이 공부했을 것이다. 그래서 여기서는 책에 나오는 몇 가지 도식도만 제시하고 IA-32 Instruction Set에 대해 바로 알아보기로 한다. 그러나 충분한 이해가 부족하다면 개인적으로 반드시 공부하고 넘어가야 한다.

Figure 2.5 A Stack

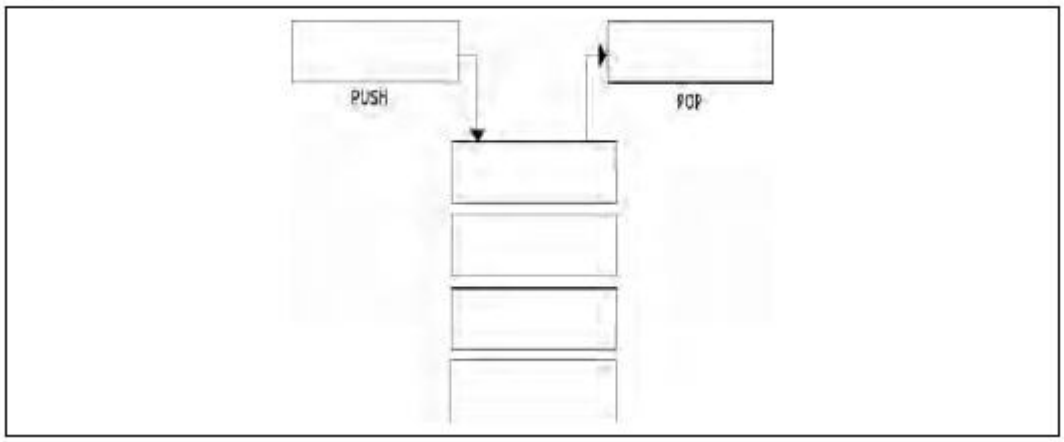


Figure 2.6 Stack Frame

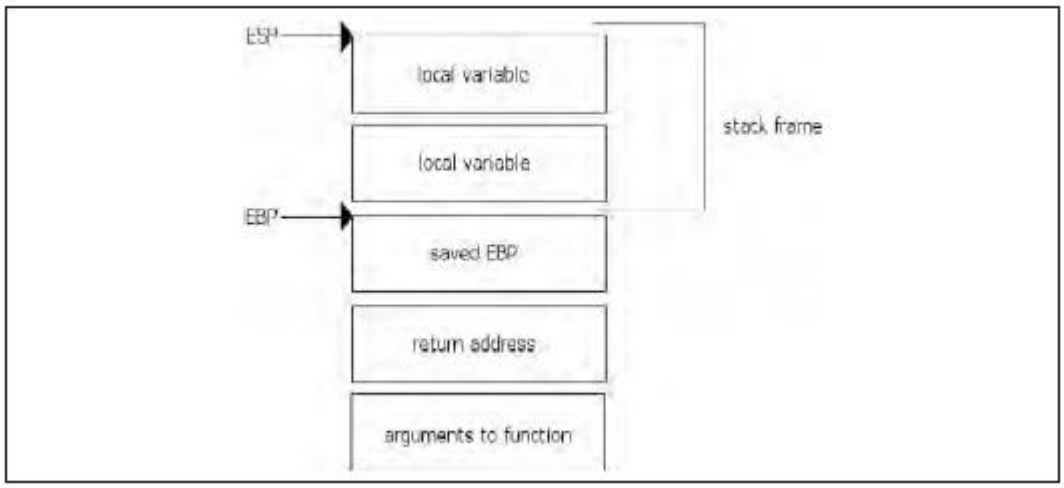


Figure 2.7 Glibc Allocated Chunk

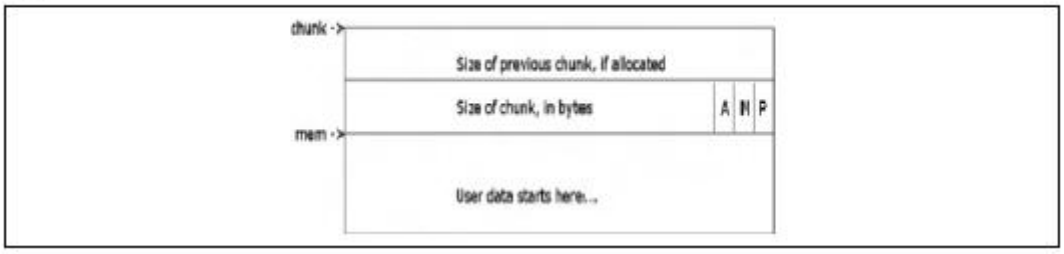
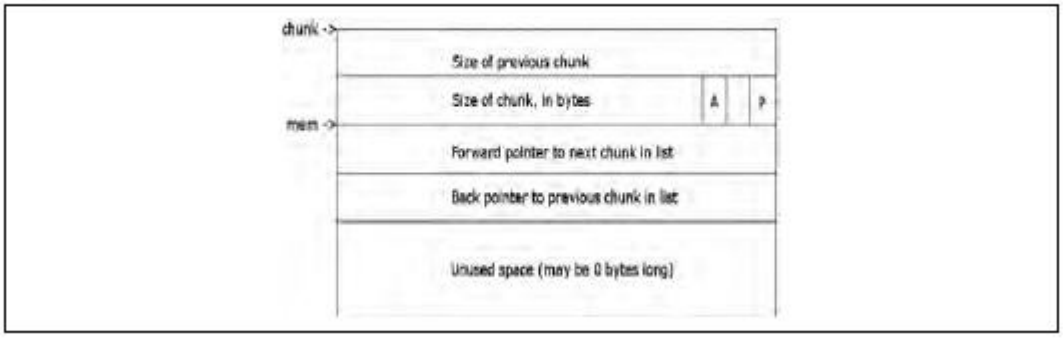


Figure 2.8 Glibc Free Chunk



IA-32 Instruction Set Refresher and Reference

이 섹션에서는 흔히 사용되는 명령들에 대해 알아본다. 이 섹션의 공부를 위해 가장 좋은 자료는 앞에서도 말했듯이 ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** Volume 2A와 2B이며, 명령어들에 대해 자세하게 설명이 나와 있다. 이 책에서는 Intel 매뉴얼의 내용 중 일부를 요약하여 올려놓고 있다. 필자도 책의 내용에 충실하고, 부족한 부분은 Intel 매뉴얼을 참고할 것이다. 먼저 전체적인 설명을 위해 책에 사용되고 있는 용어들을 먼저 확인하자.

Table 2.1 Terminology Employed

Term	Meaning
Reg32	Any 32-bit register
Reg16	Any 16-bit register
Reg8	Any 8-bit register
Mem32	32-bit memory operand
Mem16	16-bit memory operand
Mem8	8-bit memory operand
Sreg	Segment register
Memoffs8	8-bit memory offset
Memoffs16	16-bit memory offset
Memoffs32	32-bit memory offset

Imm8	8-bit immediate (constant)
Imm16	16-bit immediate
Imm32	32-bit immediate
ptr16:16	absolute address given in operand
ptr16:32	absolute address given in operand
mem16:16	absolute indirect address given in mem16:16
mem16:32	absolute indirect address given in mem16:32
rel8	8-bit relative displacement
rel16	16-bit relative displacement
rel32	32-bit relative displacement
Register name	Any name of a register that has already been introduced

제일 먼저 알아볼 명령어는 **mov**이다. 다들 잘 알고 있는 기본적인 명령어이다. mov는 한 오퍼랜드를 다른 오퍼랜드로 복사한다. 다음 mov 명령이 가질 수 있는 오퍼랜드의 형태이다.

Table 2.2 *mov* Instruction

Destination Operand	Source Operand
reg8/mem8	reg8
reg16/mem16	reg16
reg32/mem32	reg32
reg8	reg8/mem8
reg16	reg16/mem16
reg32	reg32/mem32
reg16/mem16	Sreg
Sreg	reg16/mem16
AL	memoffs8
AX	memoffs16
EAX	memoffs32
memoffs8	AL
memoffs16	AX
memoffs32	EAX
reg8	imm8
reg16	imm16
reg32	imm32
reg8/mem8	imm8
reg16/mem16	imm16
reg32/mem32	imm32

다음은 **and** 명령이다. and 명령은 source operand로 destination operand에 bitwise AND를 수행하고, 그 결과를 destination operand에 저장한다. 비교되는 두 오퍼랜드의 각 비트를 논리곱

(AND)한 결과를 destination operand에 반환한다. 따라서 만약 비교된 두 오퍼랜드의 bit가 1이라면 turned-on bit이고, 그렇지 않고 0이라면 turned-off bit이다.

Table 2.3 *and* Instruction

Destination Operand	Source Operand
reg8/mem8	reg8
reg16/mem16	reg16
reg32/mem32	reg32
reg8	reg8/mem8
reg16	reg16/mem16
reg32	reg32/mem32
AL	imm8
AX	imm16
EAX	imm32
reg8	imm8
reg16	imm16
reg32	imm32
reg8/mem8	imm8
reg16/mem16	imm16
reg32/mem32	imm32

다음은 **not** 명령이다. not 명령은 하나의 오퍼랜드에 대해 bitwise NOT 연산을 수행하는데, 오퍼랜드의 비트를 반전하여(1의 보수 – 0은 1로, 1은 0으로 바꾸는 방식) 반환한다.

Table 2.4 *not* Instruction

Destination Operand	Source Operand
reg8/mem8	N/A
reg16/mem16	N/A
reg32/mem32	N/A

다음은 **or** 명령이다. or 명령은 bitwise OR를 아규먼트에 실행하는데, bitwise OR은 두 오퍼랜드를 bit별로 비교하고, 비교된 bit 둘 다 0이면 출력 값도 0이 된다. 즉, 비트를 논리합(OR)한 결과 반환한다.

Table 2.5 *or* Instruction

Destination Operand	Source Operand
reg8/mem8	reg8
reg16/mem16	reg16
reg32/mem32	reg32
reg8	reg8/mem8
reg16	reg16/mem16
reg32	reg32/mem32
AL	imm8
AX	imm16
EAX	imm32
reg8	imm8
reg16	imm16
reg32	imm32
reg8/mem8	imm8
reg16/mem16	imm16
reg32/mem32	imm32

다음은 **xor** 명령이다. xor(exclusive-or)은 bitwise OR을 실행하며, source operand와 destination operand를 비교하고, 그 결과값을 destination operand에 저장한다. 비교된 두 개의 bit가 다르면 결과 값은 1이고, 같으면 0이 된다. 쉘코드를 만들어본 경험이 있는 사람은 잘 알겠지만 opcode에 0x00을 처리하기 위해 xor 명령을 이용한다.

Table 2.6 *xor* Instruction

Destination Operand	Source Operand
reg8/mem8	reg8
reg16/mem16	reg16
reg32/mem32	reg32
reg8	reg8/mem8
reg16	reg16/mem16
reg32	reg32/mem32
AL	imm8
AX	imm16
EAX	imm32
reg8	imm8
reg16	imm16
reg32	imm32
reg8/mem8	imm8
reg16/mem16	imm16
reg32/mem32	imm32

이제 **test** 명령에 대해 알아보자. test 명령은 특정 조건을 확인하고, 그 결과에 바탕을 두고 컨트롤 흐름을 수정한다. test 명령은 첫 번째와 두 번째 오퍼랜드의 bitwise AND를 수행하고, 그런 다음 EFLAGS 레지스터에 flag를 설정한다. 그 다음 bitwise AND 연산의 결과는 버려진다. 이 과정에서 SF, ZF, PF, CF, OF 플래그가 변경된다. Test는 CF와 OF 플래그를 0으로 설정한다.

Table 2.7 *test* Instruction

Destination Operand	Source Operand
reg8/mem8	reg8
reg16/mem16	reg16
reg32/mem32	reg32
AL	imm8
AX	imm16
EAX	imm32
reg8	imm8
reg16	imm16
reg32	imm32
reg8/mem8	imm8
reg16/mem16	imm16
reg32/mem32	imm32

이제 **cmp** 명령에 대해 알아본다. cmp 명령은 두 오퍼랜드를 비교하고, 이 비교는 destination operand로부터 source operand를 뺌으로써 수행되고, 이에 따라 EFLAAGS 레지스터에 flag를 설정한다.

Table 2.8 *cmp* Instruction

Destination Operand	Source Operand
reg8/mem8	reg8
reg16/mem16	reg16
reg32/mem32	reg32
reg8	reg8/mem8
reg16	reg16/mem16
reg32	reg32/mem32
AL	imm8
AX	imm16
EAX	imm32
reg8	imm8
reg16	imm16
reg32	imm32
reg8/mem8	imm8
reg16/mem16	imm16
reg32/mem32	imm32

이제 lea(load effective address)에 대해 알아본다. lea는 source 오퍼랜드에 의해 지정된 주소를 계산하여 그 값을 destination 오퍼랜드에 저장한다. 또한 source 오퍼랜드를 변경하지 않고 다수의 레지스터들 사이의 연산을 하기 위해 사용되기도 한다.

Table 2.9 *lea* Instruction

Destination Operand	Source Operand
reg8	mem8
reg16	mem16
reg32	mem32

이제 jmp 명령에 대해 알아보자. Jmp 명령은 실행 컨트롤을 오퍼랜드로 이동시킨다. 이 명령은 4개의 다른 타입의 jump(near jump, short jump, far jump, task switch)를 실행할 수 있다. Near jump는 현재 코드 세그먼트 내에서 발생하는 jump이고, short jump는 현재 주소로부터 -128 ~ 127까지 내의 주소로 jump하는 것이다. Far jump는 만약 현재 코드 세그먼트와 같은 권한 레벨을 가지고 있다면 주소 공간에서 어떤 세그먼트에 대한 통제고 가질 수 있다. 마지막으로 task switch jump는 다른 임무의 명령으로 jump하는 것이다.

Table 2.10 *jmp* Instruction

Destination Operand	Source Operand
rel8	N/A
rel16	N/A
rel32	N/A
reg16/mem16	N/A
reg32/mem32	N/A
ptr16:16	N/A
ptr16:32	N/A
mem16:16	N/A
mem16:32	N/A

다음은 **call**과 **ret** 명령에 대해 알아보자. call 명령은 새로운 메모리 위치에서 프로시저가 시작하도록 호출하는 것이다. call 명령은 스택에 그것의 리턴 어드레스를 push하고, 그 호출된 프로시저의 주소를 그 instruction pointer로 복사한다. 이와 반대로 **ret** 명령은 프로시저가 호출된 지점으로 리턴하는데, 프로시저가 작업을 끝내면 스택으로부터 리넡 어드레스를 instruction pointer로 pop한다. CPU는 항상 메모리 상에서 eip(instruction pointer 레지스터)가 가리키는 명령을 실행한다,

Table 2.13 *call* Instruction

Destination Operand	Source Operand
rel16	N/A
rel32	N/A
reg16/mem16	N/A
reg32/mem32	N/A
ptr16:16	N/A
ptr16:32	N/A
mem16:16	N/A
mem16:32	N/A

Table 2.14 *ret* Instruction

Destination Operand	Source Operand
N/A	N/A
imm16	N/A

이상으로 자주 사용되는 명령어들에 대해 아주 간단히 알아보았다. 앞에서도 이미 말했지만 Intel 매뉴얼을 참고하여 더 많은 공부를 스스로 할 수 있길 바란다. 구체적인 예를 통해 하나씩 알아보면 좋지만 이 글이 누군가를 위해 정리하는 것이 아니라 필자 개인이 공부하면서 정리하고, 그것을 그냥 올리는 것에 불과함으로 필자보다 부족한 사람(없을 것 같은데..)을 위해 구체적인 예까지 들어가면 정리할 시간을 갖는 것은 힘들다. 이 글을 쓰고 있는 지금 필자는 이미 4장을 보고 있다. 다른 분들을 위해 정리하는 것이 너무 힘들다.^^ 그래서 앞으로도 이기적인 방식으로 정리를 해서 그냥 올릴 것이다. 개인이 공부한 내용을 게시판에 올리는 것의 목적으로 가장 중요한 두 가지 이유는 공유와 자기억압이다.

급하지 않은 사람이라면 Kip R. Irvine의 ***Assembly Language for Intel-Based Computers***의 일부만이라도 읽어보길 권한다. 아마도 컴퓨터 공학과 출신들의 경우 이 책을 교재로 이미 보았을 수도 있을 것이다. 특히 ***Reversing Engineering Code with IDA Pro*** 이 책의 2장을 제대로 이해하기 위해 ***Assembly Language for Intel-Based Computers***의 3장 Assembly Language Fundamentals부터 8장 Advanced Procedures까지 총 6장의 내용을 읽어보는 것을 강력하게 권한다.

다소 어수선한 2장 정리였다.