Parse and Parse Assembly
(어셈블리어 입문자를 위한
어셈블리어 자료들의 모음)

목차

작성자 : b0BaNa
E-mail : zeroone2000@korea.com

작성일 : 2007년 3월 30일

0x01. Introduce.

이 문서의 목적은 저의 공부(시스템 이해)와 레포트작성에 있습니다. 또한, 인터넷 및 자료 수집을 통한 문서들을 종합적으로 모으고, 기술합니다. 시스템의 이해를 목적으로 하기 때문에 어셈블리 프로그래밍 관련 글이 아님을 밝힙니다. 어셈블리는 많은 명령어를 가지고 있지만 이 글에서는 기본적인 약간의 명령어만 다루겠습니다. 어셈블리 언어를 익히고자 함이 아닌 시스템분석을 위한, 시스템의 흐름을 이해하기 위해서는 내부적으로 컴퓨터가 고급 언어들을 어떤식으로 다루는지 파악해야 하기 때문입니다. 거의 [펌]수준이므로 저작권은 저에게 없다고 볼 수 있습니다.^^;; 전 일종의 편집자라고 생각해주세요. 이 문서에서의 어셈블리는 AT&T문법을 따릅니다. 이후의 문체는 편의상 반말로 진행하겠습니다. :-)

0x02. 어셈블리 언어란? & 배우는 목적

CPU 에는 해당 프로세서에 명령을 내리기 위해 고유의 명령어 세트가 마련되어 있는데 이 명령어 세트를 기계어라고 한다. 이 기계어는 숫자들의 규칙조합임으로 프로그래밍에 상당히 난해하다. 그래서 이 기계 명령어를 좀더 이해하기 쉬운 기호 코드로 나타낸것(기계어와 1:1로 대응된 명령을 기술하는 언어)이 어셈블리어이다. 어셈블리 언어는 그 코드가 어떤 일을 할지를 추상적이 아닌, 직접적으로 보여준다. 논리상의 오류나, 수행 속도, 수행 과정에 대해 명확히 해준다는 점에서 직관적인 언어이다. 어셈블리 언어를 사용하면 메모리에 대한 이해도도 높아진다. 어셈블리를 익히고, 배우는데 있어서는 여러 가지 목적이 있을 수 있다. 컴퓨터 시스템&구조를 좀 더 깊게 이해하고, 메모리상의 데이터나 I/O기기를 직접 액세스 하는등의 고급언어에서는 할 수 없는 조작을 위해서이다. 프로그램의 최적화 및 리버스 엔지니어링을 위해서도 필요하다.

+ 2줄 요약 +
- 어셈블리 언어는 기계어와 1:1 대응을 하는 언어이다.
- 어셈블리 언어를 배우면 시스템을 이해하는데 도움이 된다.

0x03. 어셈블리를 위한 기본 지식

(1) 기본적인 하드웨어

1) CPU

- 메모리에 있는 내용을 읽고, 쓰고 데이터를 메모리와 각 레지스터로 보낸다.
  프로그램의 명령을 해석하고 실행한다.하나의 프로세서는 12~14개의 레지스터를 가지고 있으며,CPU의 연산, 논리 장치는 숫자와 기호에 관한 연산자를 인식한다. 보통 이러한 장치들은 기본적인 연산만을 수행할 수 있다(덧셈, 뺄셈, 곱셈, 나눗셈, 숫자비교). 퍼스널 컴퓨터는 한번에 처리할 수있는 비트수(Word) 따라서 분류된다.

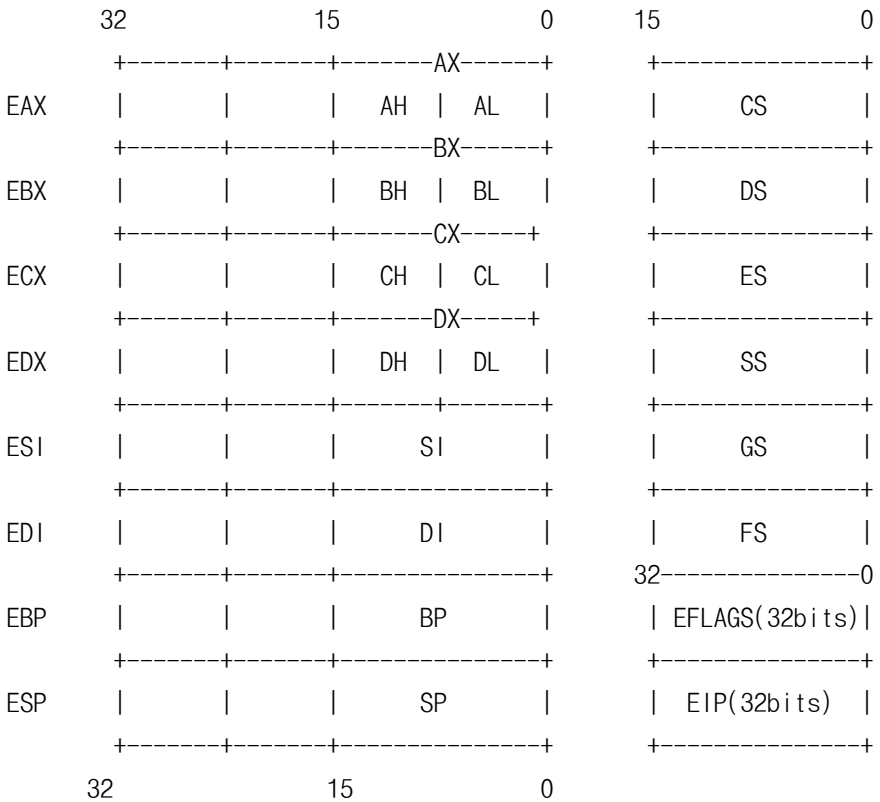
2) RAM

- 반도체로 조립된 셀들의 집합. 프로세스가 프로그램을 실행시키고 작동하기위해서 필요한 정보들을 저장하는데 쓰인다. 각각의 셀들은 숫자값을 포함하고 주소가 정해질 수 있는 형식이며 프로그램에서 흔히 메모리라고 하는 것들은 메인메모리, 즉, 램이라고 할 수 있다.

(2) 80x86 프로세서

1) CPU 레지스터 종류 : 범용 레지스터, 상태 레지스터, 플래그 레지스터

- 레지스터 : CPU내부의 기억장소로 PC가 정보를 처리하기 위해서는 정보가 특정한 셀에 저장되어 있어야 한다. 이러한 셀을 레지스터라고 불린다. 레지스터들은 8 또는 16비트 플립-플롭 회로들의 집합이다. 플립-플롭 회로란 두 단계의 전압으로 정보를 저장할 수 있는 장치이다. 낮은 전압은 0.5 볼트이고 높은 전압은 5볼트이다. 낮은단계의 에너지는 0으로 해석되고 높은 전압은 1이다. 이 상태는 보통 비트로 불리며 컴퓨터의 가장 작은 정보 단위이다.



< 레지스터의 구조 >

① 데이터 레지스터

- 데이터 레지스터는 각종 데이터 처리를 대상으로 하는 하는 32비트 레지스터 및 16비트 레지스터 일부를 프로그래머가 명령 중에서 자유롭게 지정을 할 수 있는 범용 레지스터이다.

  : EAX, EBX, ECX ,EDX

② 포인터 레지스터

: ESP, EBP,

③ 인덱스 레지스터 (Index register)

: ESI, EDI

④ 세그먼트 레지스터 (segment register)

: CS, DS, SS, ES

+ 범용 레지스터 +

32Bit	16Bit	상위8Bit	하위8Bit	기능
EAX	AX	AH	AL	누산기(Accumulator, 중간 결과를 저장해 놓음) 레지스터라 불리며, 곱셈이나 나눗셈 연산에 중요하게 사용
EBX	BX	BH	BL	베이스 레지스터라 불리며 메모리 주소 지정시에 사용
ECX	CX	CH	CL	계수기(Counter)레지스터라 불리며, Loop등의 반복 명령에 사용됩니다.
EDX	DX	DH	DL	데이터(Data)레지스터라 불리며 곱셈, 나눗셈에서 EAX와 함께 쓰이며 부호 확장 명령 등에 사용
ESI	SI			다량의 메모리를 옮기거나 비교할 때 그 소스(Source)의 주소를 가진다
EDI	DI			다량의 메모리를 옮기거나 비교할 때 그 목적지의 주소를 가리킨다.
ESP	SP			스택 포인터로 스택의 최종점을 저장한다. Push, Pop 명령에 의해 늘었다 줄었다 한다.
EBP	BP			ESP를 대신해 스택에 저장된 함수의 파라미터나 지역 변수의 주소를 가리키는 용도로 사용된다

.

+ 세그먼트 레지스터 +

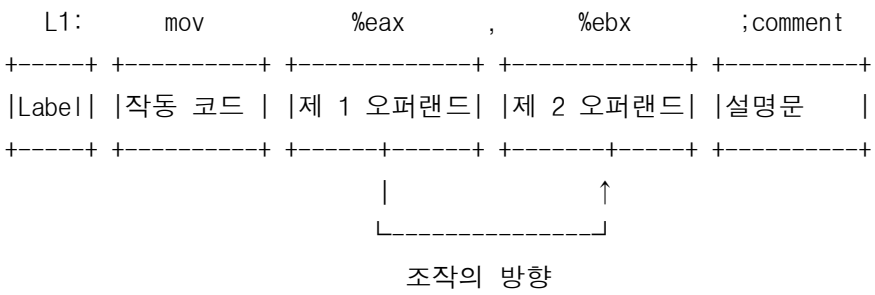
16Bit	기능
ES	보조 세그먼트 레지스터다. 두 곳 이상의 데이터 저장영역을 가리켜야 할 때 DS와 함께 사용된다. 하지만 32비트 프로그램에서는 DS와 ES가 같은 영역 역을 가리키고 있기 때문에 굳이 신경 쓰지 않아도 된다.
CS	코드 세그먼트를 가리키는 레지스터. 프로그래머의 코드의 시작주소를 가지고 있다.
SS	스택 세그먼트를 가리키는 레지스터. 스택의 시작 주소를 담고 있다. 스택 조작에 의해서 데이터를 처리하는 동작이 이루어진다.
DS	데이터 세그먼트를 가리키는 레지스터. 프로그래머가 정해놓은 데이터의 시작주소를 담고 있다.
FS	보조 세그먼트 레지스터. FS, GS는 286 이후에 추가된 것으로 운영체제를 작성하는 게 아니라면 없듯이 여겨도 된다
GS	

+ 상태 레지스터+

32Bit	16Bit	기능
EIP	IP	EIP는 현재 실행되고 있는 프로그램의 실행코드가 저장된 메모리의 주소를 가리키는 레지스터로 프로그램의 실행이 진행됨에 따라 자동으로 증가하고 프로그램의 실행 순서가 변경되는 제어문이 실행될 때 자동으로 변경된다. 그래서 직접 접근해서 값을 저장하거나 읽거나 하는 일이 없기 때문에 응용 프로그램에서는 손 댈 일이 없는 레지스터이다.
EFLAGS	FLAGS	비트 단위의 플래그 들을 저장하는 레지스터로 아주 특별한 용도로 사용된다

0x04. 어셈블리 명령어의 구성

- 어셈블리는 어셈블리어라고도 부르는데 이 어셈블리어는 명령어들의 조합이다. 인텔 CPU 안에는 이 명령어들이 회로로 구현되어 있어서 어셈블리 코드를 실행할 수 있다. CPU는 2진수로 모든 것을 처리하는데 어셈블리 명령어들도 2진수로 되어 있다. 하지만 2진수로 된 것 알아보기가 힘들어 mov, add와 같은 형태로 변환하여 보여진다. 아래 그림을 보자.



명령어 다음에 오는 레지스터 이름이나 값들은 operand라고 한다. mov %eax, %ebx에서 %eax를 제1오퍼랜드, %ebx를 제2오퍼랜드라고 한다. mov %eax, %ebx는 C언어로 보면 ebx = eax의 경우와 같다. eax에 저장된 값을 ebx에 할당(assignment)한다.(특정 장소(주로 메모리상에서)에서 특정 장소(주로 레지스터)로 데이터를 읽어 와서 적재(load)).

'L1: '과 같은 명령은 직접적으로 기계어 코드로 번역되지 않고 분기명령(jmp)등에서 참조될 때에, 번지의 계산에 사용된다.

0x05. 주소 지정 방식의 이해

- 어셈블리는 메모리를 직접 다룰 수 있다는 점에서 우리가 어셈블리를 배우는 큰 이유가 될 수 있다. 이 메모리를 다루기 위해서 다양한 주소 지정방식이 있는데 어떤식으로 주소를 사용하고, 참조하는지 확인할 필요가 있다. 참고로 '0x04'에서 예를 들었던 'mov %eax,%ebx'의 경우는 레지스터 어드레싱(register addressing)이다.

1. 즉시 지정방식(immediate addressing)
 - mov $0x1, %eax
  : eax에 (16진수)1을 값을 넣는(할당) 방식이다.
 - 이렇게 메모리(기억장치)의 주소의 내용을 꺼내지 않고 직접 값을 대응시키는 방식을 즉시지정방식이라고 한다.

2. 레지스터 지정방식(register addressing)
 - mov %esp, %ebp
  : 레지스터 ebp에 레지스터 esp의 값을 넣는다.(할당 개념)
  : 나중에 알 수 있겠지만, 위의 명령은 스택포인터를 베이스 포인터에 넣는 명령으로 함수가 시작될때 ebp의 값(일종의 시작기준점)을 정하는 명령이다.
 - 레지스터에서 직접 레지스터로 값을 대응시키는 방식을 레지스터 지정방식이라고 한다.
 - 속도는 빠르지만 레지스터의 크기(32비트)로 인해 크기가 제한된다.

3. 직접 주소 지정방식(directly addressing)
 - mov %eax, $0x80482f2
  : 주소 0x80482f2에 있는 값을 eax에 할당한다.
 - 가장 일반적인 주소지정방식이며, 메모리의 주소를 직접 지정해서 바로 찾아오는 방식이다. 즉 eax레지스터에 0x80482f2주소의 내용을 로드(load)한다는 의미이다.

4. 레지스터 간접 주소 지정 방식
 - mov (%ebx), %eax
  : ebx의 값을 주소로 하여(간접적으로) eax레지스터에 할당
 - ‘( )’가 들어간다면 간접 지정이라고 볼 수 있다. ‘( )’의 의미는 괄호 안에 들어간 값의 주소이다.

5. 베이스 상대 주소 지정 방식
 - mov 0x4(%esi), %eax
  : esi레지스터에서 4(byte)를 더한 주소의 값을 eax레지스터에 할당한다.
 - 보통 레지스터의 크기가 4byte이기 때문에 레지스터 다음 주소를 의미한다.
  문자열 열산이나 메모리 블록 전송등에 나오는 방식이다.

0x06. 어셈블리어 명령어 정리

명령어	예제	설명	분류
push	push %eax	eax의 값을 스택에 저장.	스택 조작
pop	pop %eax	스택 가장 상위에 있는 값을 꺼내서 eax에 저장	스택 조작
mov	mov %eax, %ebx	메모리나 레지스터의 값을 옮길때 사용	데이터 이동
lea	leal(%esi), %ecx	%esi의 주소값을 %ecx에 옮긴다.	주소 이동
inc	inc %eax	%eax의 값을 1 증가시킨다.	데이터 조작
dec	dec %eax	%eax의 값을 1 감소시킨다.	데이터 조작
add	add %eax, %ebx	레지스터나 메모리의 값을 덧셈할 때 쓰인다.	논리, 연산
sub	sub $0x8, %esp	레지스터나 메모리의 값을 뺄셈할 때 쓰인다.	논리, 연산
call	call proc	프로시져를 호출한다.	프로시져
ret	ret	호출했던 바로 다음 지점으로 이동.	프로시져
cmp	cmp %eax, %ebx	레지스터와 레지스터값을 비교	비교
jmp	jmp proc	특정한 곳으로 분기	분기
int	int $0x80	OS에 할당된 인터럽트 영역을 system call	인터럽트
nop	nop	아무 동작도 하지 않는다.(No Operation)	

0x07. 어셈블리 명령어 상세

- 명령어의 분류

1) 데이터 이동 : mov, lea
2) 논리, 연산 : add, sub, inc, dec
3) 흐름제어 : cmp,jmp
4) 프로시져 : call, ret
5) 스택조작 : push, pop
6) 인터럽트 : int

1) 데이터 전송

1. mov (move data)

- 형식 : mov SOURCE, DESTINATION
- 기능 : SOURCE위치에 들어있는 데이터를 복사하여 DESTINATION위치에 저장.
- 원칙 : 메모리와 레지스터(모든 연산은 레지스터에 저장된뒤 이루어진다.) 사이의 데이터 이동, 레지스터와 레지스터 사이의 데이터 이동이나 값을 메모리나 레지스터에 대입할 때 사용한다. (SOURCE와 DESTINATION의 크기가 동일해야 한다.)
  ! DESTINATION 레지스터가 CS가 될수 없다.(프로그램실행위치가 변경되기때문)
  (CS의 변경은 int, jmp, call, ret등의 명령으로 가능)
  ! SOURCE와 DESTINATION이 전부 메모리를 가르칠수 없다. (설계상 불가능)

! SOURCE가 직접지정방식일경우에는 DESTINATION은 CS일 수 없다.

2. lea
- 형식 : lea SOURCE, DESTINATION
- 기능 : SOURCE OPERAND에서 지정된 주소를 DESTINATION으로 로드한다.
  LEA의 주된 용도는 매개변수나 지역변수의 주소를 얻어오는 것이다.
  예를 들어 C언어에서 지역변수나 매개변수에 &연산자를 사용한다면 컴파일러는
  lea명령어를 생성한다.
- 원칙 : SOURCE OPERAND는 메모리에 위치해야하며, 변경될 주소는 index register나
  DESTINATION에 정의된 주소여야 한다.

2) 논리, 연산 : add, sub, inc, dec
1. add
- 형식 : add opr1, opr2
- 기능 : opr2의 내용에 op1의 내용이 더해져서 그 결과를 opr2에 저장.
- 원칙 : ! 두 개의 오퍼랜드 모두에 메모리로 조합되는 것은 불가능.

2. sub (subtract)
- 형식 : sub opr1, opr2
- 기능 : 첫번째 오퍼랜드로 부터 2번째 오퍼랜드 의 내용을 뺀 다음 결과를 첫 번째 오퍼
- 원칙 : ! 메모리끼리는 뺄셈을 할수 없다.

3. inc (Increment)
- 형식 : inc DESTINATION
- 기능 : DESTINATION을 1 증가시키고 결과값을 다시 저장

4. dec (decrement)
- 형식 : dec DESTINATION
- 기능 : DESTINATION을 1 감소시키고 결과값을 다시 저장

3) 흐름 제어 : jmp, cmp
- 형식 : jmp proc
- 기능 : 프로그램의 흐름을 바꿀 때 사용. proc의 주소로 가서 그곳의 명령어를 실행.
  if/else문, loop문(루프가 아직 끝나지 않았을때, 처음위치로 돌아가기 위해)
  등에서 나타난다.
2. cmp
- 형식 : cmp value, value
  ex) cmp %eax, 0 (eax레지스터의 값을 0과 비교한다.)
  je start (비교 결과가 같다면 start로 분기한다.)
  (같지 않다면 je 다음에 오는 명령어를 실행한다.)
- 기능 : 두값을 비교하고 비교결과에 따라 분기한다. 보통 레지스터나 메모리 및 숫자의

크기를 비교한다. cmp 명령어는 Zero, Sign, Overflow 등의 플래그를 set or clear 한다. 이 플래그의 결과에 의해서 Jcc 명령어들은 분기할 것인지를 결정한다. 보통 CMP 명령어 다음에 JE, JNE 등의 jmp관련 명령어가 위치한다.

- 원칙 : cmp 명령은 혼자 사용되지 않고 언제나 조건 점프 명령어나 조건 이동(mov) 명령어와 함께 사용된다.

- 조건 점프 명령어 : cmp 명령어의 결과에 따라 점프하는 명령어.

* 참고 *

Unsigned 계열 (부호가 없는 값)

je : jump equal - 비교 결과가 같을 때 점프

jne : jump not equal - 비교 결과가 다를 때 점프

jz : jump zero - 결과가 0일 때 점프, je와 같음. (cmp 명령에서 결과가 같으면 0을 출력)

jnz : jump not zero - 결과가 0이 아닐 때 점프

ja : jump above - cmp a, b에서 a가 클 때 점프

jae : jump above or equal - 크거나 같을 때 점프

jna : jump not above - 크지 않을 때 점프

jnae : jump not above or equal - 크지 않거나 같지 않을 때 점프

jb : jump below - cmp a, b에서 a가 작을 때 점프

jbe : jump below or equal - 작거나 같을 때 점프

jnb : jump not below - 작지 않을 때 점프

jnbe : jump not below or equal - 작지 않거나 같지 않을 때 점프

jc : jump carry - 캐리 플래그가 1일 때 점프

jnc : jump not carry - 캐리 플래그가 0일 때 점프

jnp/jpo : jump not parity / parity odd - 패리티 플래그가 0일 때 / 홀수일 때 점프

jp/jpe : jump parity / parity even - 패리티 플래그가 1일 때 / 짝수일 때 점프

jecxz : jump ecx zero - ecx 레지스터가 0일때 점프

Signed 계열 (부호가 있는 값)

jg : jump greater - cmp a, b에서 a가 클 때 점프

jge : jump greater or equal - 크거나 같을 때 점프

jng : jump not greater - 크지 않을 때 점프

jnge : jump not greater or equal - 크지 않거나 같지 않을 때 점프

jl : jump less - cmp a, b에서 a가 작을 때 점프

jle : jump less or equal - 작거나 같을 때 점프
jnl : jump not less - 작지 않을 때 점프
jnle : jump not less or equal - 작지 않거나 같지 않을 때 점프

jo/jno : jump overflow / not overflow - 오버플로 플래그가 1일 때 / 0일 때 점프
js/jns : jump sign / not sign - 사인(부호) 플래그가 1일 때(음수) / 0일 때(양수) 점프

조건 점프 명령을 조합하여 if (a > b), for, while등의 조건문 구현

4) 프로시져 : call, ret
1. call
- 형식 : call Target
- 기능 : 스택 상에서 CS를 다음에 오는 명령의 오프셋 어드레스를 PUSH하고 target으로 이동한다. 즉, 다른 함수로 제어를 옮긴다는 뜻이다. CALL 명령어 다음에 오는 명령어의 주소를 스택에 PUSH하고 주어진 주소로 제어를 옮긴다는 뜻(EIP를 변경시킴)
CALL명령은 CALL명령 다음위치에 있는 명령의 주소를 스택에 push한다.
2. ret (return)
- 형식 : ret
- 기능 : 호출된 함수에서 호출한 함수로 복귀. esp에 있는 값을 꺼내서(pop) EIP레지스터에 할당한다. 즉, call명령 당시 push되었던 주소를 pop하여 eip에 넣는 것이다.

5) 스택조작 : push, pop
1. pop
- 형식 : pop DESTINATION
- 기능 : 스택 맨 윗부분(top)에서 하나의 워드를 DESTINATION에 로드(load)
그리고 스택포인터는 그 바로 전의 데이터를 가리킨다(point).

2. push
- 형식 : push DESTINATION
- 기능 : 메모리상에 설정된 스택이라는 공간에 데이터를 저장한다. 스택의 가장 윗부분에 데이터를 저장. 스택 포인터(esp)도 워드크기만큼 증가한다.

6) 인터럽트 : int
- 형식 : int (interrupt-type)
- 기능 : 운영체제에 할당된 인터럽트 영역을 system call.

0x08.참조 사이트 및 문서

1. “김병희” 님이 쓰신 문서. (제목이 나와있지 않네요)
2. Lecture Note in Assembly Language Written by Han, Kwang-Rok
3. 블루님의 매크로 어셈 이야기
4. Just Enough Assembly Language to Get By, Part 1,2 by Matt Pietrek
5. 김성훈님의 C 프로그래머가 알아야 할 것들 - Chapter 7 어셈블리
6. 유용수님의 어셈블리 강좌
7. 김영빈님의 “헥커가 되자”
8. C를 어셈블리어로
   (http://blog.naver.com/earlyhungki?Redirect=Log&logNo=20033888653)
9. Assembly Tutor
   (http://blog.naver.com/koreahjg?Redirect=Log&logNo=60034750188)
10. http://blog.naver.com/berg76?Redirect=Log&logNo=130001429195

0x09. 마치며...

분량은 그리 많지 않지만, 의외로 작업시간이 길었던것 같습니다. 그래도 많은 자료들을 모아서 나름대로 정리를 했다는데 의의를 두죠 ^^; C에서의 어셈블리 사용이라던지, 함수의 호출 및 복귀시 스택사용에서의 분석도 추가하고 싶었지만, 나름대로 할 일이 있어 하지 못했네요. 후에 기회가 된다면, 조금 더 내용을 추가시켜서 업데이트하도록 노력할게요. 다른 분들에게는 많은 도움은 못되겠지만, 그래도 조금이라도 도움이 되셨으면 합니다. 읽어주셔서 감사합니다. 위의 참조사이트 및 문서 저자분들에게 미리 말씀 못드렸는데 괜찮은지 모르겠네요. 모두 열심히 공부해요~ ^^)/