

시스템 보안

어셈블리어의 이해 - 2



어셈블리어의 이해

- 어셈블리어 기본 명령어 - 산술 연산 명령(기본적인 정수 계산)

- ADD(Add) : 첫 번째 오퍼랜드와 두 번째 오퍼랜드 값을 더한 결과 값을 첫 번째 오퍼랜드에 저장

형식	ADD	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]
사용예	ADD	AX	4
	AX가 원래 3이었다면, 명령 실행 후 AX는 7		

- SUB(Subtract) : 첫 번째 오퍼랜드에서 두 번째 오퍼랜드 값을 뺀 결과 값을 첫 번째 오퍼랜드에 저장

형식	SUB	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]
사용예	SUB	AX	4
	AX가 원래 7이었다면, 명령 실행 후 AX는 3		

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 산술 연산 명령(기본적인 정수 계산)

- MUL(Multiplication) : 부호 없는 AL, AX, EAX 값을 오퍼랜드 값으로 나눈 결과를 지정된 레지스터에 저장(결과에 따라 OF, ZF 셋팅 될 수 있음)
8비트인 경우 AX에 저장, 16비트인 경우 DX(High):AX(Low)에 저장, 32비트인 경우 EDX(High):EAX(Low)

형식		MUL [오퍼랜드]
사용예	8bit	MUL BL
		AL = 10h, BL = 30h 일 때 명령 실행 후 AX = 0300h
	16bit	MUL BX
		AX = 100h, BX = 300h 일 때 명령 실행 후 결과는 30000h DX = 0003h, AX = 0000h
		※ AX 최대 값이 FFFFh인데 결과 값이 최대 값을 넘기 때문에 OF가 셋팅 되고 DX에 상위 값이 들어 감
	32bit	MUL EBX
		EAX = 10000h, EBX = 30001h 일 때 명령 실행 후 결과는 300010000h EDX = 00000003h, EAX = 00010000h

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 산술 연산 명령(기본적인 정수 계산)

- DIV(Division) : 부호 없는 AL, AX, EAX 값을 오퍼랜드 값으로 나눈 결과를 지정된 레지스터에 저장(결과에 따라 CF, OF, ZF 셋팅 될 수 있음)
8비트인 경우 AL(몫):AH(나머지), 16비트인 경우 AX (몫):DX(나머지), 32비트인 경우 EAX (몫):EDX(나머지)

형식		DIV [오퍼랜드]
사용예	8bit	DIV BL
		AL = 10h, BL = 3h 일 때 명령 실행 후 몫은 AL = 5h, 나머지는 AH = 1h에 저장
	16bit	DIV BX
		AX = 100h, BX = 30h 일 때 명령 실행 후 몫은 AX = 0005h, 나머지는 DX = 0010h에 저장
	32bit	DIV EBX
		EAX = 1000h, EBX = 300h 일 때 명령 실행 후 몫은 EAX = 0005h, 나머지는 EDX = 0100h에 저장

※ 주의 : 32bit 레지스터 연산 시 내부적으로 EAX의 값을 EDX:EAX 형식으로(64 비트로) 확장한 다음 연산이 수행됨

따라서 EDX를 초기화 하지 않으면 이전 작업의 결과에 영향을 받아서 원치 않는 결과 값이 나오기 때문에 DIV 사용 전에 EDX를 0h로 초기화 시켜 주어야 함
만약 이전 연산 작업 등의 이유로 EDX에 1h 값이 들어가 있었다면 위 32bit 연산의 결과는 EAX = 0200h, EDX = 55555ah로 OF가 발생

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 산술 연산 명령(기타 산술 연산 명령)

명령		설명
ADC	Add with Carry	캐리를 포함한 덧셈을 수행
SBB	Subtraction with borrow	캐리를 포함한 뺄셈을 수행
IMUL	Multiplication(Signed)	부호 있는 곱셈 연산을 수행
IDIV	Division(Signed)	부호 있는 나눗셈 연산을 수행
INC	Increment	오퍼랜드 내용을 하나 증가
DEC	Decrement	오퍼랜드 내용을 하나 감소
NEG	Change Sign	오퍼랜드의 2의 보수, 즉 부호를 반전

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 산술 연산 명령(기타 산술 연산 명령)

명령		설명
AAA	ASCII Adjust after Addition	덧셈 결과를 UNPACK 10진수로 변환
AAS	ASCII Adjust after Subtraction	뺄셈 결과를 UNPACK 10진수로 변환
AAD	ASCII Adjust after Division	나눗셈 결과를 UNPACK 10진수로 변환
AAM	ASCII Adjust after Multiplication	곱셈 결과를 UNPACK 10진수로 변환
DAA	Decimal Adjust After Addition	ADD or ADC 연산 결과를 PACK 10진수로 변환
CBW	Convert Byte to Word	AL의 데이터를 AX로 확장(부호 비트 포함)
CWD	Convert Word to Double word	AX의 데이터를 DX : AX에 걸쳐 저장(부호 비트 포함)
CWQ	Convert Double word to Quad word	EAX의 데이터를 EDX : EAX에 걸쳐 저장(부호 비트 포함)

어셈블리어의 이해

- 이진화 십진법(Binary-coded decimal, BCD)

- 이진수 네 자리(4bit)를 묶어 십진수 한 자리로 사용하는 기수법

- 이진화 십진법에는 팩(Pack) 형식과 언팩(Unpack 또는 Zone) 형식이 대표적

- 4bit를 십진수의 한 자리로 이용하고 부호 표현을 덧붙이는 등의 방식으로 십진수를 표현

- 자릿수 마다 4bit인 이유는 1bit는 2(0~1), 2bit는 4(0~3), 3bit는 8(0~7), 4bit는 16(0~15)만큼 표현이 가능한데, 10을 표현할 수 있는 최소한의 비트가 4bit이기 때문이다.

- 팩(Pack) 형식은 4bit마다 십진수 한 자리씩 표현하고, 마지막 4bit에 부호를 표시(부호는 +일 경우 C, -일 경우 D, 부호가 없다면 F)

- 언팩(Unpack) 형식은 8bit를 4bit는 zone으로 나머지 4bit는 digit으로 표현

- digit에는 십진수 한 자리씩 들어가고 zone에는 특정 값을 채움

- Zone에는 EBCDIC의 경우 1111(마지막 zone에는 부호를 표시), ASCII decimal의 경우 0011, Unpacked decimal의 경우 0000

어셈블리어의 이해

- 이진화 십진법(Binary-coded decimal, BCD) 표현 : EBCDIC

- 확장 이진화 십진법 교환 부호(EBCDIC, Extended Binary Coded Decimal Interchange Code)

IBM에서 1963년과 1964년에 걸쳐 고안된 8비트 문자열 인코딩 방식

① 1357의 팩, 언팩형 표현

0001(1)		0011(3)		0101(5)		0111(7)		1111(F, 부호 없음)
1111(F)	0001(1)	1111(F)	0011(3)	1111(F)	0101(5)	1111(F, X)	0111(7)	

② +1357의 팩, 언팩형 표현

0001(1)		0011(3)		0101(5)		0111(7)		1100(C, +)
1111(F)	0001(1)	1111(F)	0011(3)	1111(F)	0101(5)	1100(C, +)	0111(7)	

③ -1357의 팩, 언팩형 표현

0001(1)		0011(3)		0101(5)		0111(7)		1101(D, -)
1111(F)	0001(1)	1111(F)	0011(3)	1111(F)	0101(5)	1101(D, -)	0111(7)	

어셈블리어의 이해

- 이진화 십진법(Binary-coded decimal, BCD) 표현 : ASCII and Unpacked BCD

- AAA(ASCII Adjust after Addition), AAS(ASCII Adjust after Subtraction) 등과 같은 ASCII 십진 연산에 활용

① 1357의 ASCII decimal 표현 : 8bit가 ASCII 값을 표현

0011(3)	0001(1)	0011(3)	0011(3)	0011(3)	0101(5)	0011(3)	0111(7)
---------	---------	---------	---------	---------	---------	---------	---------

② 1357의 Unpacked decimal 표현

0000(0)	0001(1)	0000(0)	0011(3)	0000(0)	0101(5)	0000(0)	0111(7)
---------	---------	---------	---------	---------	---------	---------	---------

어셈블리어의 이해

- 이진화 십진법(Binary-coded decimal, BCD) 사용 이유

- ASCII 10진 문자열을 이진수로의 변환 없이 바로 연산할 수 있음

예시) ASCII '8'과 '2' 더하기

```
mov ah, 0      ; 덧셈 수행하기 전에 ah를 0으로 만들  
mov al, '8'    ; ax = 0038h  
add al, '2'    ; ax = 006ah  
aaa           ; ax = 0100h  
or ax, 3030h   ; ax = 3130h = '10', ah = 31h = '1', al = 30h = '0'  
              ; UNPACK 10 진수(ASCII decimal)는 30h와 or 연산을 통해서 ascii로 쉽게 변환 가능
```

※ AAA(ASCII adjust after addition) 명령 실행 후 ax 값이 0100h가 되는 이유에 대한 보충 설명(인텔에서는 다음과 같이 설명)

al의 낮은 자릿수가 9보다 크거나 AF(auxiliary carry flag)가 1이면 al에 6을 더하고 ah에는 1을 더한 다음 모든 경우에 대해 al과 0fh를 AND 연산

즉, 위의 경우 9보다 크기 때문에 6을 더해 0070h가 되고 그 후 ah에 1을 더해 ax는 0170h가 되는데, 그 후 al과 0fh를 AND 연산하면 0100h라는 값이 나옴

어셈블리어의 이해

- 이진화 십진법(Binary-coded decimal, BCD) 사용 이유

- 10진수 체계로 사용자에게 친숙, BCD는 바이너리에 비해서 크기가 커지지만 직관적으로 알 수 있음

방법	표현	크기
185를 BCD로 변환 시(packed)	0001 1000 0101	12bit
185를 바이너리로 변환 시	1011 1001	8bit

- BCD는 자릿수가 증가할 때마다 4bit만 추가하면 되기 때문에 수를 표현하는 데 크기에 제한이 없음

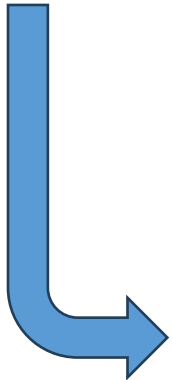
- 소수점 표현이 용이

10진수 0.1을 바이너리로 표현하면 0.0001100110011... 의 무한한 소수 값을 갖게 되지만,
이를 BCD코드로 표현하면 0.0001로 매우 심플하게 표현할 수 있음

어셈블리어의 이해

- 이진수 표현(정수)

- 십진수 55 이진수 변환



$$\begin{array}{r} 2 \overline{) 55} \\ 2 \overline{) 27} \cdots 1 \\ 2 \overline{) 13} \cdots 1 \\ 2 \overline{) 6} \cdots 1 \\ 2 \overline{) 3} \cdots 0 \\ \hline 1 \cdots 1 \end{array}$$

$55 = 110111_2$

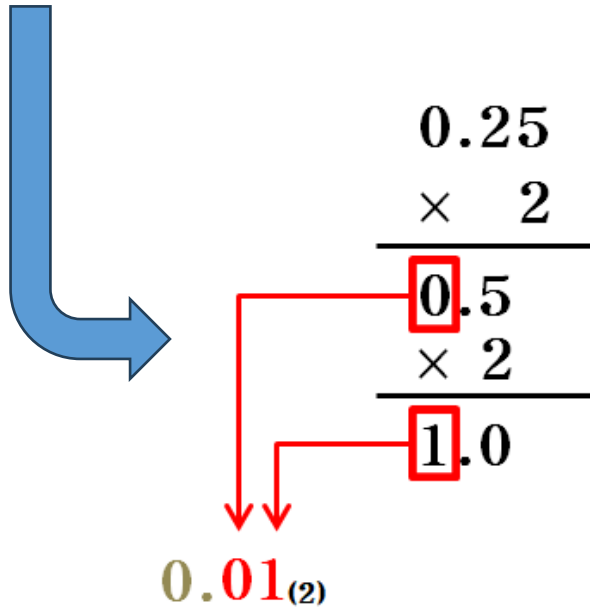
- 십진수 정수를 이진수로 변환하는 방법

- ① 십진수 값을 2로 나누는 과정을 반복하면서 나머지를 기록
- ② 마지막 1이 남으면 기록
- ③ 기록된 값을 역순으로 쓰면 해당 정수의 2진수 변환 값

어셈블리어의 이해

- 이진수 표현(소수)

- 십진수 0.25 이진수 변환



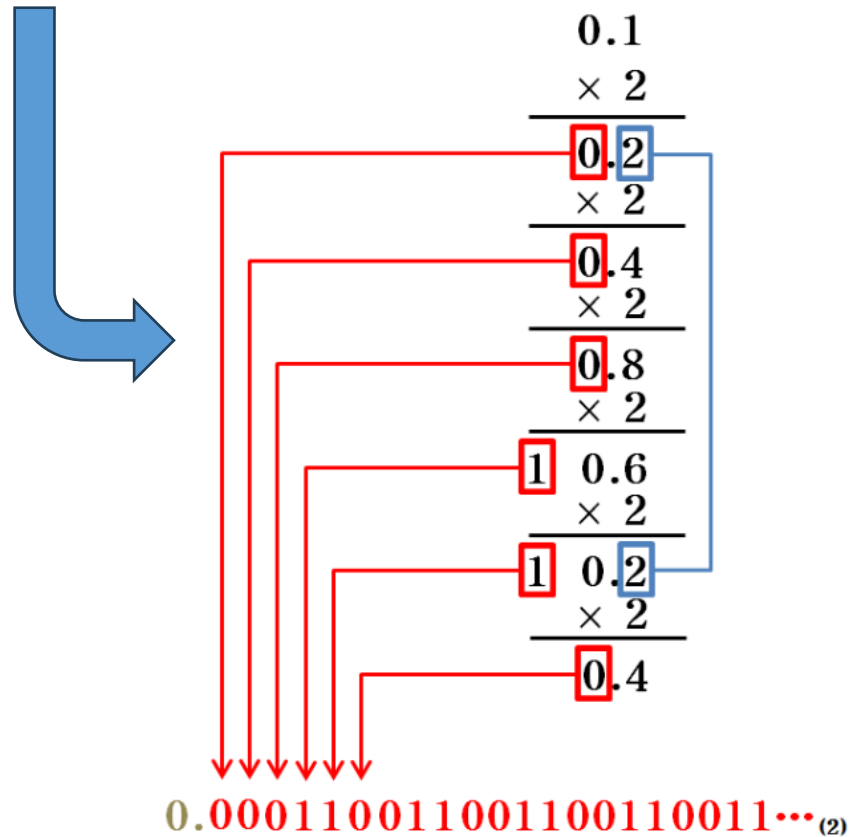
- 십진수 소수를 이진수로 변환하는 방법

- ① 10진수 소수에 2를 곱한 다음 소수점 위 정수를 기록
- ② 소수점 위 정수를 제외한 소수에 2를 곱하는 과정을 반복
- ③ 기록된 값을 순서대로 쓰면 해당 정수의 2진수 변환 값

어셈블리어의 이해

- 이진수 표현(소수)

- 십진수 0.1 이진수 변환



- 십진수 0.1은 이진수로 변환했을 때 무한 반복됨

어셈블리어의 이해

- 부동 소수점(Floating Point) 표현

- IEEE 754 : IEEE에서 개발한 컴퓨터에서 부동소수점을 표현하는 가장 널리 쓰이는 표준
- IEEE 754-1985에서 binary floating point에 대한 제정, IEEE 754-2008에서는 이진법과 십진법을 모두 포함

※ 부동 소수점의 정규화 형식(Normalized format)

$$(-1)^s \times c \times b^q$$

s : 부호부(sign)를 표현하며 0인 경우 +, 1인 경우 -를 의미

c : 가수부(significand, fraction, mantissa)를 나타내며 양의 정수로 표현, 정밀도(precision)에 따라 범위가 제한

b : 밑수(base)/기수(radix)를 나타내며 2 또는 10으로 각각 2진수, 10진수 표현이 되는 것을 의미

q : 지수부(exponent)를 나타내며 지수부는 소수점의 위치

예시)

12345를 밑수가 10인 경우로 표현을 하면 $(-1)^0 \times 1.2345 \times 10^4$

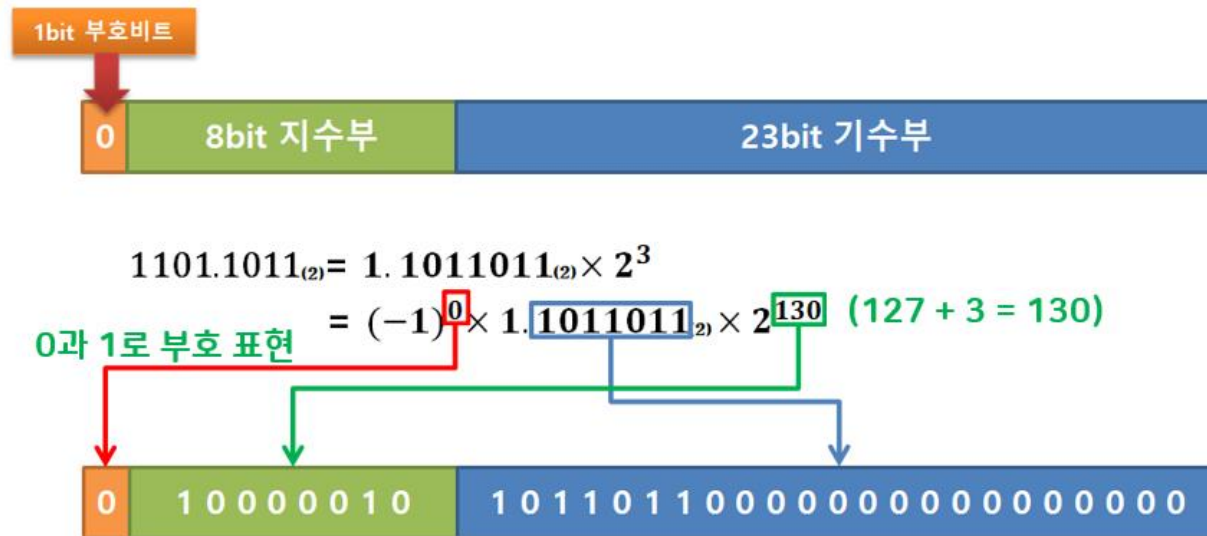
82.25(1011001.01)를 밑수가 2인 경우로 표현을 하면 $(-1)^0 \times 1.01100101 \times 2^6$

어셈블리어의 이해

• 부동 소수점(Floating Point) 표현

- IEEE 754 : IEEE에서 개발한 컴퓨터에서 부동소수점을 표현하는 가장 널리 쓰이는 표준
- IEEE 754-1985에서 binary floating point에 대한 제정, IEEE 754-2008에서는 이진법과 십진법을 모두 포함

※ 32비트 부동 소수점 (Floating Point) 표현



기수부(Significand)

- 소수점 아래 부분을 그대로 쓰고 부족한 비트는 0으로 채움

지수부 (Exponent)

- E + bias 이며 unsigned

bias value = $2^{(e-1)} - 1$, e는 지수 비트의 개수

32비트 float의 경우 bias = $2^{(8-1)} - 1 = 127$

※ 참고 :

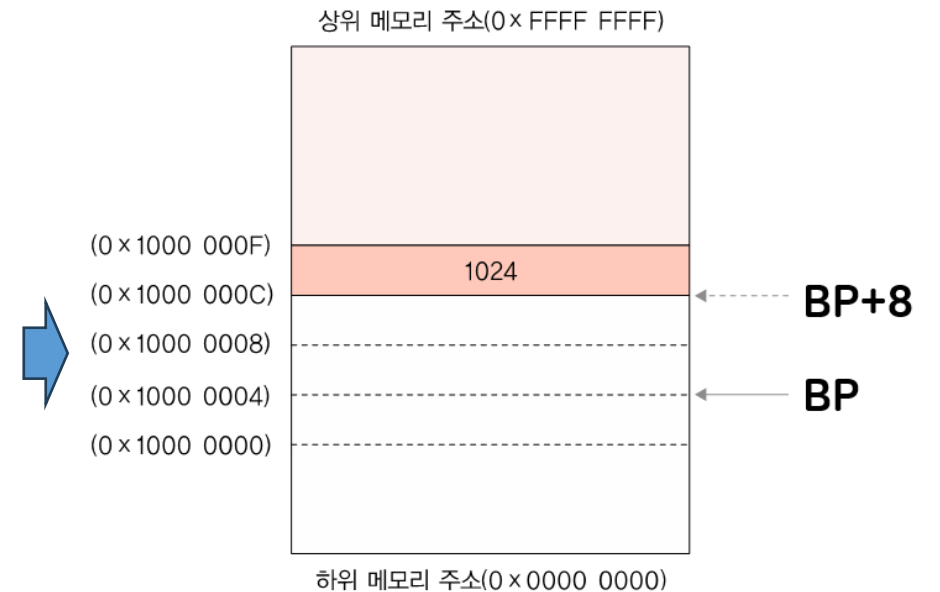
- 지수부는 8비트로 구성 되므로 256(0~255)까지 표현 가능하지만 0은 0을 표현하고 255는 무한대를 표현하기 위해 예약 되어 있음
- 1~124 범위의 값이 사용되는데 절반을 음수에 절반을 양수에 할당 부호를 표현하는 비트가 없기 때문에 bias를 사용해서 표현

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 데이터 전송 명령

- **MOV(Move)** : 데이터 저장할 때 사용

형식	MOV	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]
사용예	MOV	AX	[BP+8]
	BP의 주소에 8이 더해진 주소에 있는 데이터 값을 AX에 대입 BP의 현재 값이 0x10000004라면, BP+8은 0x1000000C 0x1000000C에 있는 값이 1024므로 AX에 1024를 대입		



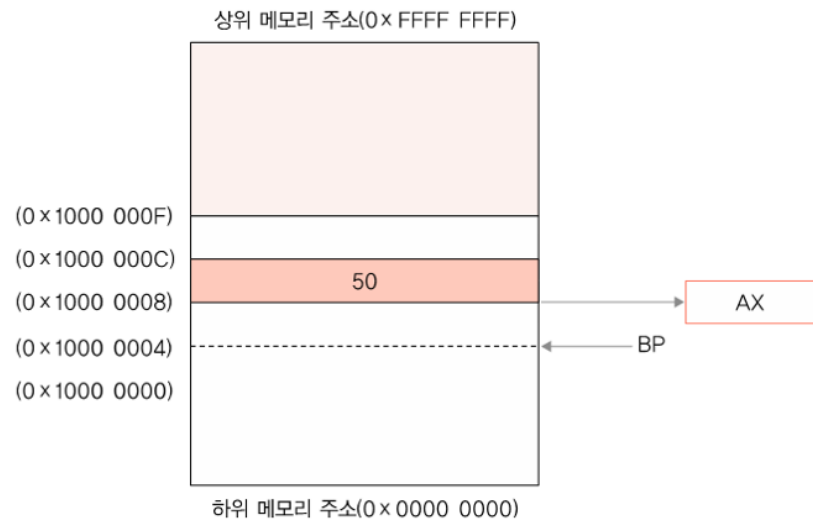
어셈블리어의 이해

• 어셈블리어 기본 명령어 - 데이터 전송 명령

- **LEA(Load effective address to register)** : 주로 주소 값을 처리하기 위해서 사용(MOV 명령어로 처리하려면 두 단계가 필요한 작업)

MOV 명령과 LEA 명령에서 첫 번째 오퍼랜드는 데이터 이동 공통, 두 번째 오퍼랜드에 대한 추가 연산의 처리 방식이 다름

두 번째 오퍼랜드에 'BP+4'가 있을 경우 MOV 명령은 'BP+4' 주소에 저장된 값을 처리하지만, LEA 명령은 'BP + 4' 값 자체를 처리



MOV 명령

형식	LEA	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]
사용예	LEA	AX	[BP+4]
	BP의 현재 값이 0x10000004라면, MOV 명령은 BP+4인 0x10000008 주소에 저장된 값 50을 AX에 대입 LEA 명령은 BP 값인 0x10000004에 4를 더해서 AX에 대입 (AX = 0x10000008)		
	LEA AX, [BP+4]	➡ MOV AX, BP ; [BP] 가 아님 ADD AX, 4	

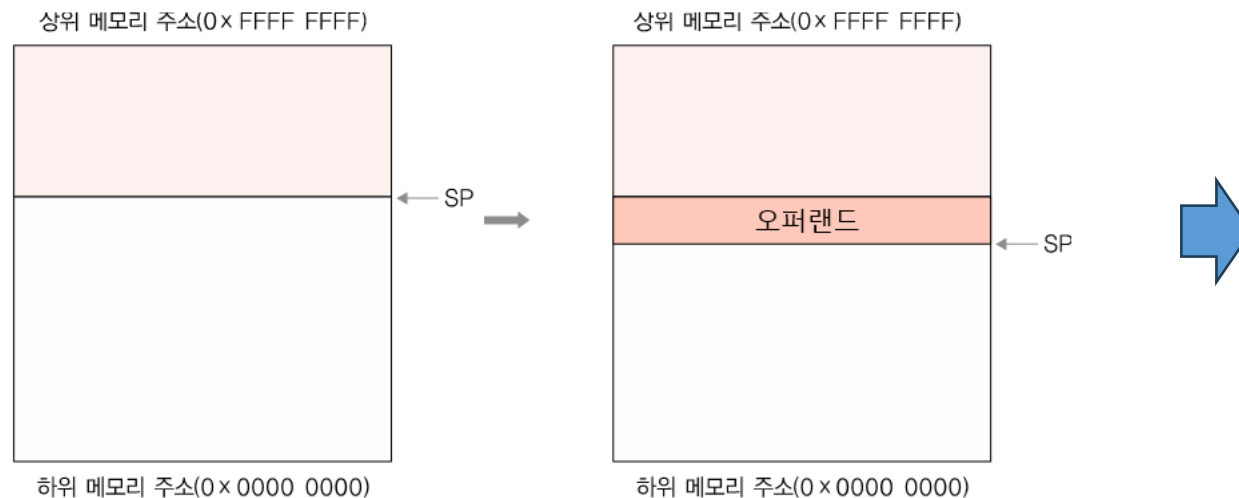
어셈블리어의 이해

- 어셈블리어 기본 명령어 - 데이터 전송 명령

- 스택(Stack)은 낮은 주소로 커짐(high address -> low address 방향으로)

- **PUSH(Push)** : 스택에 데이터를 삽입할 때 사용

형식	PUSH [오퍼랜드]
사용예	PUSH AX
	AX의 값을 스택에 저장



PUSH :
스택은 커지고,
스택 포인터(SP)는 데이터 크기만큼 감소

push eax는 아래 코드와 같은 의미

```
sub esp, 4  
mov [esp], eax
```

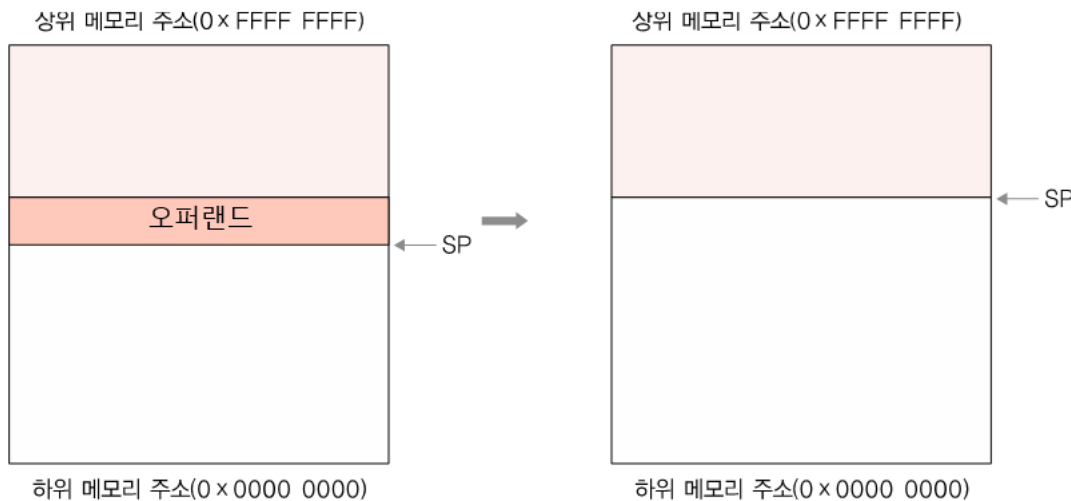
어셈블리어의 이해

- 어셈블리어 기본 명령어 - 데이터 전송 명령

- 스택(Stack)은 낮은 주소로 커짐(high address -> low address 방향으로)

- POP(Pop) : 스택에서 데이터를 꺼내오기 위해 사용

형식	POP [오퍼랜드]
사용예	POP AX
	스택 맨 위의 값(ESP가 가리키는 값)을 꺼내서 AX에 저장



POP :
스택은 감소하고,
스택 포인터(SP)는 삭제하는 데이터 크기만큼 증가

pop eax는 아래 코드와 같은 의미

```
mov eax, [esp]  
add esp, 4
```

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 데이터 전송 명령(기타 데이터 전송 명령)

명령		설명
XCHG	Exchange data	오퍼랜드1과 오퍼랜드2의 데이터를 교환
IN	Input from AL(AX to Fixed port)	오퍼랜드가 지시한 포트로부터 AX에 데이터를 입력
OUT	Output from AL(AX to Fixed port)	오퍼랜드가 지시한 포트에 AX의 데이터를 출력
XLAT	Translate byte to AL	BX:AL이 지시한 테이블의 내용을 AL로 로드
LDS	Load Pointer to DS	LEA 명령과 유사한 방식으로 다른 DS 데이터의 주소의 내용을 참조 시 사용
LES	Load Pointer to ES	LEA 명령과 유사한 방식으로 다른 ES 데이터의 주소의 내용을 참조 시 사용
LAHF	Load AH with Flags	플래그의 내용을 AH의 특정 비트로 로드
SAHF	Store AH into Flags	AH의 특정 비트를 플래그 레지스터로 전송
PUSHF	Push Flags	플래그 레지스터의 내용을 스택에 삽입
POPF	Pop Flags	스택에 저장되어 있던 플래그 레지스터 값을 삭제

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령

- **AND(And)** : 대응되는 비트가 둘 다 1일 때만 결과가 1 이고, 그 이외는 모두 0

형식	AND	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]
사용예	AND	AX	10h
	AX 값이 08h라면 이를 이진수로 표현하면 1000이 되고, 10h는 1010이므로 명령을 수행한 뒤에 AX 값은 1000 = 08h		
	AX	1	0
	0x10h	1	0
AND 연산 결과		1	0

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령

- **OR(Or)** : 대응되는 비트 중 하나만 1이어도 결과가 1이고, 둘 다 0인 경우에만 0

형식	OR	[첫 번째 오퍼랜드]		[두 번째 오퍼랜드]	
사용예	OR	AX		10h	
	AX 값이 08h라면 이를 이진수로 표현하면 1000이 되고, 10h는 1010이므로 명령을 수행한 뒤에 AX 값은 1010 = 10h				
	AX	1	0	0	0
	0x10h	1	0	1	0
OR 연산 결과		1	0	1	0

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령

- XOR(Exclusive Or) : 대응되는 비트 양쪽이 같으면 0, 다르면 1

형식	XOR	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]		
사용예	XOR	AX	10h		
	AX 값이 08h라면 이를 이진수로 표현하면 1000이 되고, 10h는 1010이므로 명령을 수행한 뒤에 AX 값은 0010 = 2h				
	AX	1	0	0	0
	0×10h	1	0	1	0
XOR 연산 결과		0	0	1	0

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령

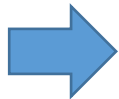
- NOT(Invert) : 오퍼랜드의 각 비트를 반전

형식	NOT	[오퍼랜드]												
사용예	NOT	AX												
	AX 값이 08h라면 이를 이진수로 표현하면 1000 명령을 수행한 뒤에 AX 값은 0111 = 7h													
	<table><tr><td>AX</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>NOT 연산 결과</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>					AX	1	0	0	0	NOT 연산 결과	0	1	1
AX	1	0	0	0										
NOT 연산 결과	0	1	1	1										

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 논리 명령

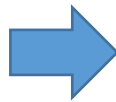
- **TEST(And function to flags, no result)** : 첫 번째 오퍼랜드와 두 번째 오퍼랜드를 AND 연산하는 방식으로 비교, 비교의 결과는 상태 플래그에만 반영

형식	TEST	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]										
사용예	TEST	AL	0000b										
	두 오퍼랜드의 값을 AND 연산 시키고 결과가 상태 플래그(Zero Flag)에 저장되기 때문에 두 오퍼랜드 값이 달라도 ZF가 셋팅 될 수 있어서 두 오퍼랜드가 같은 값인지를 판단하는데는 부적절												
	<table><tr><td>AL</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0000b</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>  Zero Flag = 1			AL	1	0	0	0	0000b	0	0	0	0
	AL	1	0	0	0								
0000b	0	0	0	0									
보통 TEST EAX, EAX 처럼 EAX가 0인지를 확인하는 용도로 많이 사용 ※ 두 오퍼랜드가 0이 아닌 경우를 제외하고는 값을 단정지을 수 없기 때문													

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 논리 명령

- **CMP(Sub function to flags, no result)** : 첫 번째 오퍼랜드에서 두 번째 오퍼랜드를 빼는 방식으로 두 오퍼랜드를 비교, 비교의 결과는 상태 플래그에만 반영 저장

형식	CMP	[첫 번째 오퍼랜드]	[두 번째 오퍼랜드]		
사용예	CMP	AL	0100b		
	두 오퍼랜드의 값 크기를 비교하고 결과가 상태 플래그(Zero Flag)에 저장되기 때문에 두 오퍼랜드가 같은 지, 두 오퍼랜드 중 어떤 값이 큰지 여부를 판단할 때 사용				
	AL	1	0	0	 Zero Flag = 0 Carry Flag = 0
	0100b	0	1	0	
	AL 값이 같은 경우는 결과가 0이기 때문에 Zero Flag = 1, Carry Flag = 0				
AL 값이 더 큰 경우는 결과가 0보다 크기(양수) 때문에 Zero Flag = 0, Carry Flag = 0					
AL 값이 더 작은 경우는 결과가 0보다 작기(음수) 때문에 Zero Flag = 0, Carry Flag = 1					
※ Carry Flag는 비교 결과가 음수이거나 unsigned 연산에서 값의 차이가 너무 커서 담을 수 없는 경우 설정 됨					

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령(기타 논리 명령)

명령		설명
SHL / SAL	Shift Left(Logical Shift, Arithmetic Shift)	왼쪽으로 오퍼랜드 만큼 자리 이동
SHR /SAR	Shift Right(Logical Shift, Arithmetic Shift)	오른쪽으로 오퍼랜드 만큼 자리 이동
ROL	Rotate Left	각 비트를 왼쪽으로 이동하고 최상위 비트는 캐리 플래그와 최하위 비트 위치로 복사
ROR	Rotate Right	각 비트를 오른쪽으로 이동하고 최하위 비트를 캐리 플래그와 최상위 비트 위치로 복사
RCL	Rotate through carry left	캐리를 포함하여 왼쪽으로 오퍼랜드 만큼 회전 이동
RCR	Rotate through carry Right	캐리를 포함하여 오른쪽으로 오퍼랜드 만큼 회전 이동

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령(기타 논리 명령)

- Logical Shift : SHL(Shift Left) / SHR(Shift Right)



SHL(Shift Left)

0 1000000 1b → 1 000000 10b

- 최상위 비트는 캐리 플래그로 복사하고 버려짐
캐리 플래그를 제외한 각 비트를 왼쪽으로 이동
빈자리인 최하위 비트는 0으로 채워짐



SHR(Shift Right)

0 1000000 1b → 1 0100000 0b

- 최하위 비트는 캐리 플래그로 복사하고 버려짐
캐리 플래그를 제외한 각 비트를 오른쪽으로 이동
빈자리인 최상위 비트는 0으로 채워짐

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령(기타 논리 명령)

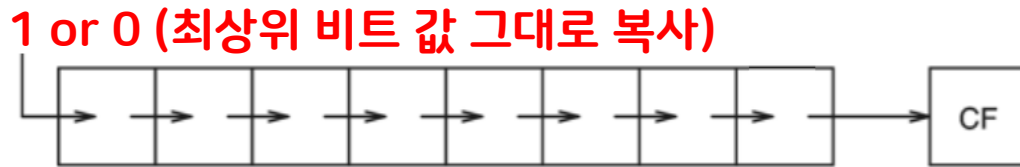
- Arithmetic Shift : SAL(Shift Arithmetic Left) / SAR(Shift Arithmetic Right)
- SAL은 SHL과 동일한 동작을 수행하며 어셈블리어에서는 구분을 하지만 실제 기계어 코드는 동일한 명령어



SAL(Shift Arithmetic Left)

0 1000000 1b → 1 000000 10b

- 최상위 비트는 캐리 플래그로 복사하고 버려짐
캐리 플래그를 제외한 각 비트를 왼쪽으로 이동
빈자리인 최하위 비트는 0으로 채워짐



SAR(Shift Arithmetic Right)

0 1000000 1b → 1 1100000 0b

- 최하위 비트는 캐리 플래그로 복사하고 버려짐
캐리 플래그를 제외한 각 비트를 오른쪽으로 이동
빈자리인 최상위 비트는 원래 값 그대로 복사
(부호 비트를 유지함으로써 부호 있는 연산에 활용 가능)

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령(기타 논리 명령)

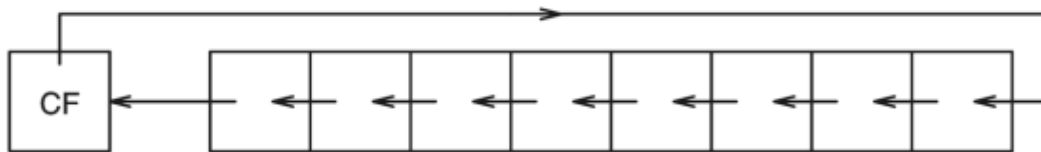
- ROR/ROL과 RCL/RCR의 차이점은 캐리 플래그(Carry flag)에 대한 처리 방식에 있음



ROL(Rotate Left, without carry)

0 10000000b → 1 00000001b

- 최상위 비트는 캐리 플래그로 복사하고
캐리 플래그를 제외한 각 비트를 왼쪽으로 회전



RCL(Rotate through carry left)

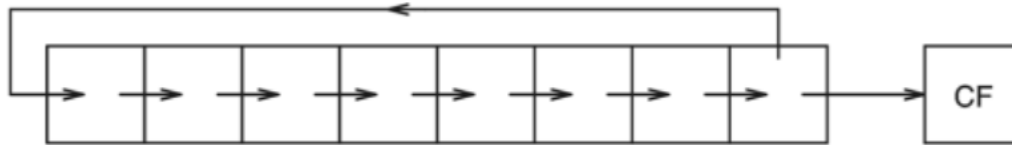
0 10000000b → 1 00000000b

- 캐리 플래그를 포함하여 각 비트를 왼쪽으로 회전

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 논리 명령(기타 논리 명령)

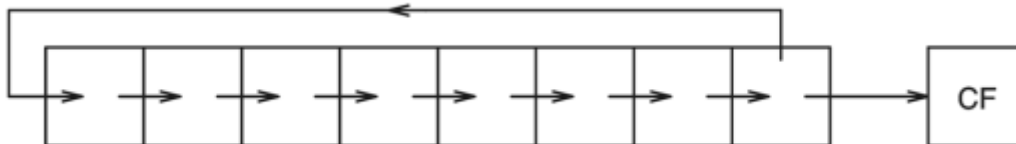
- ROR/ROL과 RCL/RCR의 차이점은 캐리 플래그(Carry flag)에 대한 처리 방식에 있음



ROR(Rotate Right, without carry)

0 0000000 1b → 1 10000000b

- 최하위 비트는 캐리 플래그 위치로 복사하고
캐리 플래그를 제외한 각 비트를 오른쪽으로 회전



RCR(Rotate through carry Right)

0 0000000 1b → 1 00000000b

- 캐리 플래그를 포함하여 각 비트를 왼쪽으로 회전

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 문자열 명령

- **REP(Repeat)** : MOVSB와 같은 명령어 앞에 위치, CX가 0이 될 때까지 뒤에 오는 명령 반복
- **MOVSB(Move String)** : 바이트(BYTE)나 워드(WORD), 더블워드(DWORD) 옮기는 명령
MOVSB, MOVSW, MOVSD가 있는데, SI에 저장된 메모리 주소의 데이터를 DI에 저장된 주소로 전송

형식	REP	[작동 코드]
사용예		CLD LEA SI, String_1 LEA DI, String_2 MOV CX, 384 REP MOVSB
		- CLD (Clear Direction) : 디렉션 플래그를 지움 - LEA SI, String_1 : String 1의 주소 값을 SI(Source Index)에 저장 - LEA DI, String_2 : String 2의 주소 값을 DI(Destination Index)에 저장 - MOV CX, 384 : CX에 384 저장 - REP MOVSB : SI로부터 DI까지 CX가 0이 될 때까지 1바이트씩 복사

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 문자열 명령(기타 문자열 명령)

명령문		설명
CMPS	Compare String	SI와 DI에 저장된 주소의 데이터를 비교한 결과에 따라 플래그를 설정
SCAS	Scan String	AL 또는 AX와 DI에 저장된 주소의 값을 비교한 결과에 따라 플래그를 설정
LODS	Load String	SI에 저장된 주소의 값을 AL 또는 AX로 로드
STOS	Store String	AL 또는 AX를 DI에 저장된 주소에 저장

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 제어 명령

- **JMP(Unconditional Jump)** : 대표적인 점프 명령 프로그램을 실행할 주소 또는 라벨로 이동

형식	JMP [오퍼랜드]
사용 예	JMP 100h
	주소로 직접 지정한 100h번지로 점프
	label_1: MOV CX, 384 ... JMP label_1
	라벨로 JMP를 지정하는 경우

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 제어 명령

- 조건부 점프(산술, 논리 연산의 결과로 설정되는 상태 플래그에 따라서 점프 명령 수행)

조건 점프 명령	산술, 논리 연산	
JA	CMP A > B	CMP
JB	CMP A < B	
JE	CMP A == B	
JNE	CMP A != B	
JZ	TEST EAX, EAX (EAX = 0)	TEST
JNZ	TEST EAX, EAX (EAX != 0)	

명령어	의미	부등호	플래그 조건
JA	Jump if (unsigned) above	>	CF = 0 and ZF = 0
JAЕ	Jump if (unsigned) above or equal	>=	CF = 0 OR ZF = 1
JB	Jump if (unsigned) below	<	CF = 1
JBE	Jump if (unsigned) below or equal	<=	CF = 1 OR ZF = 1
JC	Jump if carry flag set		CF = 1
JCXZ	Jump if CX is 0 (CX = 0)		CX = 0
JE	Jump if equal	==	ZF = 1
JECXZ	Jump if ECX is 0 (ECX = 0)		ECX = 0
JG	Jump if (signed) greater <=> JL	>	ZF=0, and SF==OF
JZ	Jump if zero	==	ZF = 1

Flag	분기 명령
ZF (Zero Flag) : 연산결과가 0이면 1	JZ, JNZ
CF (Carry Flag) : 부호 없는 수 끼리 연산 후 연산결과가 비트 범위를 넘어서면 1 (8bit -> 9bit, 255[8bit] + 1 = 256[9bit])	JC, JNC
SF (Sign Flag) : 연산결과가 음수라면 1	JS, JNS
OF (Overflow Flag) : 부호 있는 수 끼리 연산 후 연산결과가 최대 크기를 넘어서면 1 (+ -> -, 127 + 1 = -128)	JO, JNO

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 제어 명령

- **CALL(Call)** : 프로시저(C 언어에서의 함수와 같은 개념) 호출, JMP처럼 오퍼랜드에 라벨을 지정
리턴 주소로 IP(Instruction Pointer) 주소 백업 'PUSH EIP + JMP'와 같은 의미
- **RET(Return from CALL)** : 함수에서 호출한 곳으로 돌아갈 때 사용하는 명령 'POP EIP'와 같은 의미

형식	CALL [오퍼랜드]
사용 예	<pre>MOV AX, 8h CALL SUBR ADD AX, 10h ... SUBR : INC AX RET</pre>
	<ul style="list-style-type: none">- SUBR 함수 호출하면, SUBR 라벨이 있는 곳에서 RET까지 실행(INC AX가 있으므로 AX 값은 0x09h)- RET 명령이 실행되면 CALL 다음 라인인 'ADD AX, 10h'이 실행되어 AX 값은 0x19h

어셈블리어의 이해

• 어셈블리어 기본 명령어 - 제어 명령

- **LOOP(Loop CX times)** : 특정 블록을 지정된 횟수만큼 반복 (오퍼랜드에는 라벨을 지정)
CX는 자동적으로 카운터로 사용되며 루프 반복할 때마다 감소

형식	LOOP [오퍼랜드]
사용 예	MOV AX, 0 MOV CX, 5 L1 : INC AX LOOP L1
	L1이 CX의 숫자만큼 5번 회전하므로 결과적으로 AX의 값은 5

- **INT(Interrupt)** : 인터럽트가 호출되면 CS:IP(Code Segment : Instruction Pointer)와 플래그를 스택에 저장하고,
그 인터럽트에 관련된 서브 루틴이 실행

형식	INT [오퍼랜드]
사용예	INT 21h

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 제어 명령(기타 제어 명령)

- STC(Set Carry) : 명령은 오퍼랜드 없이 사용, EFLAGS 레지스터의 CF 값을 세팅
- NOP(No Operation) : 아무 의미 없는 명령, 명령어 사이의 빈 공간을 채우는 등의 목적으로 사용

명령		설명
CLC	Clear Carry	캐리 플래그를 클리어
CMC	Complement Carry	캐리 플래그를 반전
HLT	Halt	정지
CLD	Clear Direction	디렉션 플래그를 클리어
CLI	Clear Interrupt	인터럽트 플래그를 클리어
STD	Set Direction	디렉션 플래그를 세팅
STI	Set Interrupt	인터럽트 인에이블 플래그를 세팅
WAIT	Wait	프로세스를 일시 정지 상태로 전환
ESC	Escape to External device	종료 명령

어셈블리어의 이해

- 어셈블리어 기본 명령어 - 자주 사용되는 명령어

명령어	예제	분류	설명
PUSH	PUSH EAX	스택 조작	EAX의 값을 스택에 저장
POP	POP EAX	스택 조작	스택 가장 상위에 있는 값을 꺼내서 EAX에 저장
MOV	MOV EAX, EBX	데이터 이동	메모리나 레지스터의 값을 옮길 때 사용
INC	INC EAX	데이터 조작	EAX의 값을 1증가(++)
DEC	DEC EAX	데이터 조작	EAX의 값을 1감소(--)
ADD	ADD EAX, EBX	논리, 연산	레지스터나 메모리의 값을 덧셈할 때 사용
SUB	SUB EAX, EBX	논리, 연산	레지스터나 메모리의 값을 뺄셈할 때 사용
CALL	CALL PROC	프로시저	프로시저를 호출
RET	RET	프로시저	호출했던 바로 다음 지점으로 이동
CMP	CMP EAX, EBX	비교	레지스터와 레지스터의 값을 비교
JMP	JMP PROC	분기	특정한 곳으로 분기
INT	INT \$0x80	인터럽트	OS에 할당된 인터럽트 영역을 System call
NOP	NOP		아무 동작도 하지 않음(No Operation)

QA

