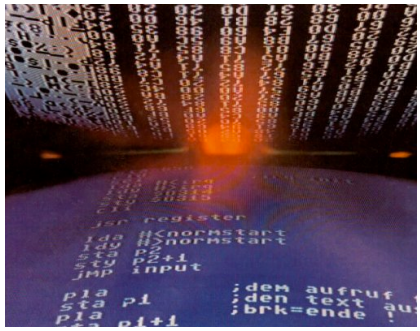
Assembly Language

```
Element tHis = Assembly_Language

while ( tHis.contains(“essential”) && (“fundamental”)) {
        if( you read this document ) {
                you will be understand Assembly
        }
        else {
                you will lose the chance
        }
```

Nick : Deok9
E-mail : DDeok9@gmail.com
HomePage : http://Deok9.org
Twitter :@DDeok9

Contents



0. Intro

Reversing 이라는 분야를 처음 접한것은 2010년 3월 이었다. 그 당시 서울에 올라와 학원을 다니면서 처음으로 Memory 구조, Assembly Language, Debugging 을 배웠다.
이때 당시만 해도 Window7 이 보편화 되지 않았었고 64bit 가 많이 사용되어 지지는 않았다. 그런데 요즈음 와서는 대부분의 사람들이 64bit Window7 을 사용하고 있고 이에 따라 64bit Application 도 등장하게 되었다.
이 문서에서는 32bit Intel 문법을 기준으로 각종 명령어, Register 를 자세하게 살펴본 후 64bit 에서 추가된 점과 달라진 점, AT&T 문법과 Intel 문법의 차이를 살펴볼 계획이다.
그리고 간단하게 Inline ASM 으로 계산기를 만들어 보고, Assembly 로 된 Code 를 C 언어로 분석, 변환하는 실습도 하겠다.

일반적인 Assembly 관련 문서에서 볼 수 있는 진수개념, Memory 개념, 16bit Register 들은 이미 잘 나온 문서들이 많기 때문에 생략할 것이다.

1. 32bit Register

Register 란 CPU 가 자체적으로 사용하는 일종의 Memory 공간이라 보면된다.
CPU 는 Memory 에 저장된 Data 나 그 위치를 Register 에 저장한 후 이를 읽어 들여 연산을 수행한다. 그럼 32bit Register 에 대해 살펴보도록 하겠다.

EAX (Accumulator)	산술, 논리 연산의 중심이 되는 Register 이다. RET 값을 전달해준다.
EBX (Base)	간접번지 지정시 사용된다.
ECX (Count)	반복 명령 수행시에 반복횟수 지정에 주로 사용된다. 4 Byte 지역 변수 선언시 PUSH ECX 로 사용되기도 한다.
EDX (Data)	간접번지 지정에 사용된다. 곱셈, 나눗셈을 할 때에는 보조 Accumulator 로 사용되기도 한다.

[표 1 - 1] General Register

➡ 일반적으로는 정수, Bool, 논리, Memory 연산 등에서 사용하며, 연산 외에도 여러가지 용도로 사용된다.

ESP (Stack Pointer)	가장 최근에 Stack 에 들어온 Data 를 가리킨다.
EBP (Base Pointer)	현재 실행 중인 함수의 Stack Frame 가장 첫 지점을 가리킨다. System 에 따라 주소 번지가 유동적으로 배당되는 것을 고정적인 상대 주소로 접근하게 한다.

[표 1 - 2] Pointer Register

➡ 말그대로 어딘가를 가리키는 Register 이며, Full Descending 방식을 사용하는 Intel Architecture 이기 때문에 위와 같이 사용이 가능하다.

EIP (Instruction Pointer)	다음에 실행할 명령이 들어 있는 Memory 번지를 가진다. CS Segment Register 와 한쌍이 되어 실행번지를 참조한다.

[표 1 - 3] Program Counter Register

➡ 항상 CPU 가 먼저 확인하고 실행하며, 범용 Register 와는 달리 사용자 임의로 조작할 수 없다. (call, jump, ret 등으로는 변경 가능)
➡ 만약 EIP 값을 조작할 수 있다면 **Exploit** 이 가능하다.

ESI (Source Index)	복사, 비교를 하는데 사용되는 Source 문자열을 가리킨다.
EDI (Destination Index)	복사, 비교를 하는데 사용되는 Destination 문자열을 가리킨다.

[표 1 - 4] Index Register

➡ 다른 범용 Register 와 마찬가지로 연산과, 간접번지 지정에 사용되며 이 외에도 문자열을 비교하거나 전송하는 Stream 명령에도 사용된다.

CS (Code Segment)	Code Segment 의 시작 번지를 가리키며 EIP Register 가 가진 Offset 값과 합쳐서 실행을 위한 명령어 주소를 참조하게 된다.
DS (Data Segment)	Data Segment 의 시작 번지를 가리키며 General, Index Register 와 합쳐서 Data 영역의 주소를 참조하게 된다.
SS (Stack Segment)	Stack Segment 의 시작 번지를 가리키며 Pointer Register 와 합쳐서 Stack 영역의 주소를 참조하게 된다.

[표 1 - 5] Segment Register

위에서 살펴본 Register 외에 산술, 논리 연산 시 상태 결과 값이 Setting 되는 Status Flag 가 있다.

CF (Carry Flag)	“연산된 결과 값” 이 “결과값이 들어갈 피 연산자의 크기” 보다 클 때 1로 Set 된다.
ZF (Zero Flag)	연산 결과가 0일 때 1로 Set 된다.
SF (Sign Flag)	연산 결과 최상위 bit (MSB)가 음수 (1) 일 때 1로 Set 된다.
OF (Overflow Flag)	피 연산자가 부호 있는 정수 라는 가정하에 “ 연산 결과 값”이 “결과값이 들어갈 피 연산자의 범위” 를 벗어났을 때 Set 된다.

[표 1 - 6] Status Flag

➡ CF 와 OF 는 모두 Overflow 가 발생했음을 알리기 위한 Flag 이다. 즉, 산술 연산의 결과가 피 연산자에 완전히 표현할 수 없을 때 Program 에게 그런 상황(Overflow) 를 알리기 위해서 사용된다. 두 Flag 의 차이점은 Program 이 처리하는 Data 의 Type (부호 유무) 과 관계가 있다.

✓ 부호 있는 정수 값은 동일한 부호 없는 정수 값 보다 1bit 작으므로(부호bit) 두 정수의 Overflow 는 서로 다르게 인지되어야 한다. 이런 이유로 CF 와 OF 는 각기 다르게 사용된다.

✓ MOV/XCHG 는 Flag 에 영향을 미치지 않으며, ADD/SUB 는 모든 Flag 에, INC/DEC 는 CF 를 제외한 모든 Flag 에 영향을 미친다.

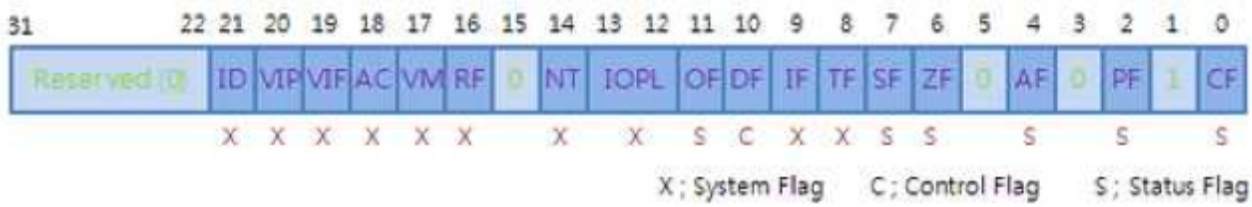
```
1    mov ax,0x1126          ; 10진수 4390
2    mov bx,0x7200          ; 10진수 29184
3    add ax,bx
```

[그림 1 - 1] Sample Code

➡ 결과 값 0x8326은, AX Register 값이 부호 없는 값이라면 10진수 33574가 되고 부호 있는 값이라면 Overflow 가 발생한다.

- ✓ 부호 있는 정수에서 최상위 bit 가 1이면 음수라는 것을 의미하므로 10진수 -31962가 된다.
- ✓ 이때 OF (부호 있는 인자에 대한 Overflow) = 1, CF (부호 없는 인자에 대한 Overflow) = 0

Status Flag 는 EFlags Register 에 속해 있다.



[그림 1 - 2] EFlags Register

➡ 각 Register bit 마다 의미를 가지고 있으며, 일부 bit 는 System 에서 직접 Setting 하고 일부 bit 는 Program 에서 사용된 명령의 수행 결과에 따라 Setting 된다.

2. 32bit Instruction

AND : destination operand 과 source operand 의 각 bit 를 AND 연산 한다.

OR : destination operand 와 source operand 의 각 bit 를 OR 연산한다.

TEST : source operand 와 destination operand 를 and 연산한 결과값의 상태정보만 저장
- ✓ 주로 값의 상태정보를 알고 싶을때 Flag Register 를 참조하여 이를 확인한다.

XOR : 비교하는 두 operand 가 같으면 0으로 초기화되고, 다르면 1로 초기화를 시킨다.

NEG : 피 연산자를 2의 보수로 만들어버린다.

XCHG : 두 연산자의 내용이 서로 교환된다.
- ✓ imm 값이 피연산자로 올 수 없다.

CMP : source operand 와 destination operand 값을 비교한다.
- ✓ destination operand - source operand 한 결과값의 상태정보만 Flag Register 에 저장한다.

MOV : source operand 의 내용을 destination operand 에 복사하는 명령어
- ✓ operand 의 사이즈 값이 동일 해야 하며, Memory 에서 Memory 로 복사는 불가능
- ✓ Segment register 에서 Segment register 로의 복사는 가능하다. (직접적인 값 복사가 아님)

MOVZX : 부호를 가지지 않고, 상위 bit 를 0으로 채워준다.
- ✓ Compiler 는 Unsign 형 에게만 이 명령어를 자동으로 설정해 준다.

MOVSZ : 부호를 가지고, 남는 상위 bit 를 sign bit 로 채워준다.
- ✓ Compiler 는 부호를 가지는 type 에게 이 명령어를 사용한다.

MOVS : ESI 가 가리키는 주소의 값을 EDI 가 가리키는 주소로 복사한다.
- ✓ operand 가 생략이 가능하며, 주로 문자열 비교 복사시 접두어로 REP 가 쓰인다.
- ✓ MOVS[ ] 에서 [ ] 안에 오는 단위에 따라 읽어들이는 크기가 다르다.

LEA : source operand 의 주소 값을 destination operand 에 저장한다.

INC : 피연산자에 1을 더한다.
- ✓ 연산 결과에 따라 ZF 나 OF 가 Set 될 수 있다.

DEC : 피연산자에 1을 빼는 명령이다.
- ✓ 연산 결과에 따라 ZF 나 OF 가 Set 될 수 있다.

ADD : 산술연산 중 덧셈을 시키는 명령어
- ✓ ESP 와 함께 사용하면 매개변수 해제에도 사용된다.

SUB : 산술연산 중 뺄셈을 시키는 명령어
- ✓ ESP 와 함께 사용하면 매개변수 선언에도 사용된다.

MUL : 산술연산 중 부호를 가지지 않는 곱셈을 하는 명령어

✓ 최상위 bit 가 Data 가 된다.

IMUL : 산술연산 중 부호를 가지는 곱셈을 하는 명령어

✓ 최상위 bit 가 부호 bit 가 된다.

연산하면	이 값과 연산하고	결과 저장
MUL CL	AL * CL	AX
MUL CX	AX * CX	DX : AX
MUL ECX	EAX * ECX	EDX : EAX

DIV : 산술 연산 중 부호를 가지지 않는 나눗셈을 하는 명령어

✓ 최상위 bit 가 Data 가 된다.

IDIV : 산술 연산 중 부호를 가지는 나눗셈을 하는 명령어

✓ 최상위 bit 가 부호 bit 가 된다.

연산하면	이 값과 연산하고	몫 : 나머지
DIV CL	AX / CL	AL : AH
DIV CX	DX : AX / CX	AX : DX
DIV ECX	EDX : EAX / ECX	EAX : EDX

SHL : 피 연산자의 bit 를 왼쪽으로 Shift (2의 배수 단위로 값 증가)

✓곱셈과 나눗셈에 관련된 연산, 암호화, 복호화에 주로 사용된다.

SHR : 피 연산자의 bit 를 오른쪽으로 Shift (2의 배수 단위로 값 감소)

✓ 곱셈과 나눗셈에 관련된 연산, 암호화, 복호화에 주로 사용된다.

SCAS : EAX 에 저장되어 있는 값과 ESI, EDI 가 가리키는 곳에 저장된 값을 비교한다.

✓ 결과에 따라서 Status Flag 값이 적절하게 Set 된다.

STOS : EAX 에 저장되어 있는 값을 EDI 가 가리키는 곳에 저장한다.

JMP : 해당 주소값으로 이동한다.

CALL : EIP 값은 변경하기 전에 Stack 에 RET 할 주소를 백업하고 함수를 호출한다.

RET : Return 을 수행하는 명령어

✓ 의미상 POP EIP 와 동일하다.

✓ 보통 EAX 가 Return 값을 전달한다.

LEAVE : Stack Frame 을 해제하는데 사용하는 명령어

✓ MOV ESP, EBP
POP EBP 와 동일하다.

PUSH : Stack 에 Data 를 저장하는데 쓰이며 이때 ESP 값은 자동으로 따라온다.

- ✓ 함수의 매개변수 전달
- ✓ 지역변수를 위한 공간 할당 (주로 ECX 로 4Byte 할당)
- ✓ 단순히 백업 목적

PUSHAD : General Register, Pointer Register, Index Register 의 값을 Stack 에 PUSH

- ✓ Backup 의 일종이라 볼 수 있다.
- ✓ 주로 Packer 에 사용됨

PUSHFD : Flag Register 를 Stack 에 PUSH 한다.

POP : ESP 가 가리키는 곳에 저장된 내용을 destination operand에 저장후 ESP 값을 증가

- ✓ ESP 값이 증가하게 되면 Stack Memory 영역상에서 한칸 밑을 가리키게 된다. (낮은 주소)
  실질적으로는 존재하지만 우리가 참조를 못하기 때문에 없는 것으로 생각한다.
  가장 최근을 가리키는 top point 역할을 ESP 가 하므로 ESP 이전 주소 (높은 주소)로 갈 수 없다.

POPAD : Stack 에 존재하는 값을 General Register, Pointer Register, Index Register 로 POP 한다.

- ✓ PUSHAD 로 Stack 에 Backup 해둔 Register 정보를 다시 이용할 때 사용한다.
- ✓ 주로 Packer 에 사용됨

POPFD : Stack 에 존재하는 값을 Flag Register 로 POP 한다.

REP : ECX 에 저장된 값만큼 해당 명령어를 반복 실행 한다.

PTR : 피 연산자의 크기를 재 설정 한다.

CDQ : Double word 에서 Quad word 로 변환하는 명령어

- ✓ 이 외에 CBW, CWDE, CWD 등이 있다.

INT : Inturrupt 를 나타내며, Software Interrupt 를 발생시켜 운영체제의 Subroutine 호출

STC : CF 를 1로 Set 한다.

CLC : CF 를 0으로 Set 한다.

STD : DF (Direction Flag) 를 1로 Set 한다.

CLD : DF (Direction Flag) 를 0으로 Set 한다.

STI : IF (Interrupt Flag) 를 1로 Set 한다.

CLI : IF (Interrupt Flag) 를 0으로 Set 한다.

NOP : 아무일도 하지 않는 명령어 이다.

- ✓ 주로 분기 구문을 우회하거나 Exploit 을 실행 할 때 사용한다.

현재까지 명령어의 간단한 의미를 살펴 보았다. 조금만 더 상세하게 살펴보도록 하자

더하기와 빼기 : ADD 와 SUB 명령은 여러가지 Type 의 인자를 사용할 수 있다. 즉, Register 이름, Hard Coding 된 값이나 Memory 주소를 사용할 수 있다. 그리고 명령의 수행결과는 destination operand 에 저장된다.

곱하기와 나누기 : IA-32 Processor 는 곱하기와 나누기 연산에 Binary Shift 연산을 사용한다. SHL 명령은 2의 제곱수로 곱하는 것과 동일한 결과를 SHR 명령은 2의 제곱수로 나누는 것과 동일한 결과를 만들어 낸다. 이 후에 Compiler 는 일반적으로 필요한 경우 결과값을 보정하기 위해서 추가적인 더하기나 빼기 연산을 수행한다.

```
1    lea eax,DWORD PTR [edx+edx]
2    add eax,eax
3    add eax,eax
4    add eax,eax
5    add eax,eax
```

[그림 2 - 1] LEA 와 ADD 명령을 조합한 32를 곱 연산

➡ Code 의 크기는 SHL 명령에 비해 코드가 크지만, 실질적으로 더 빠르다.

✓ 이는 LEA 명령과 ADD 명령이 모두 저지연, 고속 처리 명령이기 때문이다.

```
1    lea eax,DWORD PTR [esi + esi * 2]
2    sal eax,4
3    sub eax,esi
```

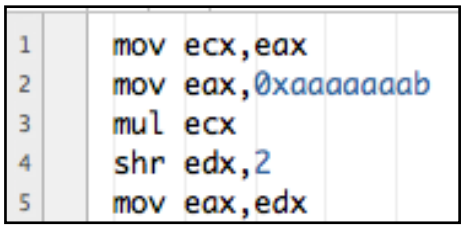
[그림 2 - 2] LEA 와 SHL 명령을 조합한 11을 곱 연산

➡ ESI Register 에 3을 곱하기 위해 LEA 명령을 사용하고 있다. 그리고 SHL 명령을 이용해서 4를 더 곱한다. 그 다음에는 곱한 결과의 값에서 원래의 값을 뺀다.

나누기 연산은 최적화된 나누기 연산 기술인 역곱을 사용한다. 역곱이란 나누기보다 4배 ~ 6배 정도 빠른 곱하기 연산을 사용해서 나누기 연산을 하는 것이다. 예를들어 30/3 의 경우 1/3 인 0.3333 을 30에 곱한다.

나눗수	32bit 역수	역수 값(분수)	나눗수와의 결합
3	0xAAAAAAAB	2/3	2
5	0xCCCCCCCD	4/5	4
6	0xAAAAAAAB	2/3	4

[표 2 - 1] 나누기 연산을 위한 역곱의 예



```
1    mov ecx,eax
2    mov eax,0xaaaaaaab
3    mul ecx
4    shr edx,2
5    mov eax,edx
```

[그림 2 - 3] 전형적인 역곱을 수행하는 Code

➡ 이 Code 는 ECX 에 0xAAAAAAAB 를 곱하고 있다. 이는 2/3 을 곱하는 것과 동일하며, 연산 후에 오른쪽으로 2bit Shift 를 시키고 있다. 2/3을 곱한 후 4로 나누는 연산은 결국 6으로 나누는 연산과 동일하다.

✓ 여기서 결과 값을 EDX 에서 가져 오는 이유는 MUL 명령의 64bit 결과 값중 상위 32bit 는 EDX에 하위 32bit 는 EAX 에 저장되기 때문이다.

✓ 역수의 정밀함에는 한계가 있기 때문에 Compiler 는 역수를 한번 더한 후 다시 역수를 뺀다.

Low-level 에서 비교와 뺄셈은 어떻게 할까?

dst operand	src operand	인자 간의 관계	Flag 변화
X >= 0	Y >= 0	X = Y	OF = 0, SF = 0, ZF = 1
X >= 0	Y >= 0	X > Y	OF = 0, SF = 0, ZF = 0
X < 0	Y < 0	X > Y	OF = 0, SF = 0, ZF = 0
X > 0	Y > 0	X < Y	OF = 0, SF = 1, ZF = 0
X < 0	Y >= 0	X < Y	OF = 0, SF = 1, ZF = 0
X < 0	Y > 0	X < Y	OF = 1, SF = 0, ZF = 0
X > 0	Y < 0	X > Y	OF = 1, SF = 1, ZF = 0

[표 2 - 2] 부호 있는 인자 값에 대한 CMP, SUB 연산 결과

➡ ZF 값이 1 이면 빼기 연산 결과 값이 0 이므로 두 인자 값이 동일

➡ 세 Flag 값이 모두 0인 경우는 연산 결과값이 Overflow 없는 양수이므로 X가 큼

➡ 연산 결과 값이 음수이고 Overflow 가 없으면 Y가 큼

➡ 연산 결과 값이 양수이고 Overflow 가 발생하면 Y가 큼

➡ 연산 결과 값이 음수이고 Overflow 가 발생하면 X가 큼

인자 간의 관계	Flag 변화
X = Y	CF = 0, ZF = 1
X < Y	CF = 1, ZF = 0
X > Y	CF = 0, ZF = 0

[표 2 - 3] 부호 없는 인자 값에 대한 CMP, SUB 연산 결과

➡ ZF 값이 1이면 빼기 연산 결과 값이 0 이므로 두 인자 값이 동일

➡ 두 Flag 값이 모두 0인 경우는 연산 결과 값이 Overflow 없는 양수 이므로 X가 큼

➡ Overflow 가 발생하는 경우는 Y가 X보다 큼

mnemonic	Flag	만족 조건
G, NLE	ZF = 0 & ((OF = 0 & SF = 0)	(OF = 1 & SF = 1)	X > Y
GE, NL	(OF = 0 & SF = 0)	(OF = 1 & SF = 1)	X >= Y
L, NGE	(OF = 1 & SF = 0)	(OF = 0 & SF = 1)	X < Y
LE, NG	ZF = 1	((OF = 1 & SF = 0)	(OF = 0 & SF = 1)	X <= Y

[표 2 - 4] CMP, SUB 에 대한 부호 있는 조건 Code

➡ G, NLE 의 경우 인자가 동일한지 확인하기 위해 ZF 를 사용하며, X > Y 비교를 위해 SF 를 사용해서 결과 값이 Overflow 없는 양수 이거나, Overflow 발생한 음수인지 판단
➡ GE, NL 의 경우 ZF 값이 0인지 확인하는 것을 제외하고는 위의 경우와 동일
➡ L, NGE 의 경우 X < Y 비교를 위해 Overflow 발생한 양수이거나, Overflow 없는 음수인지 판단
➡ LE, NG 의 경우 ZF 값이 0인지 확인하는 것을 제외하고는 위의 경우와 동일

mnemonic	Flag	만족 조건
A, NBE	CF = 0 & ZF = 0	X > Y
AE, NB, NC	CF = 0	X >= Y
B, NAE, C	CF = 1	X < Y
BE, NA	CF = 1	ZF = 1	X <= Y
E, Z	ZF = 1	X = Y
NE, NZ	ZF = 0	X != Y

[표 2 - 5] CMP, SUB 에 대한 부호 없는 조건 Code

➡ A, NBE 의 경우 Y 가 X 보다 크지 않다는 것을 확인하기 위해서 CF 를 사용며, ZF 를 사용해서 X, Y 가 동일한 지 확인
➡ AE, NB, NC 의 경우 CF 만 확인한다는 것을 제외하고는 위와 동일
➡ B, NAE, C 의 경우 CF 값이 1이면 X < Y 을 알 수 있음
➡ BE, NA 의 경우 ZF 를 확인한다는 것을 제외하고는 위와 동일
➡ E, Z 의 경우 ZF 값이 1이면 결과 값이 0 이고 X = Y 임을 알 수 있음
➡ NE, NZ 의 경우 ZF 값이 0 이면 결과 값이 0 이 아니고 X != Y 를 알 수 있음

3. 64bit Register & Instruction

32bit 를 살펴 보았으니 이제는 64bit Register 에 대해서 살펴보겠다.
앞서 32 bit 에 대해 살펴볼 때 다루지 않은 Register 도 간단하게 나올 것이다.

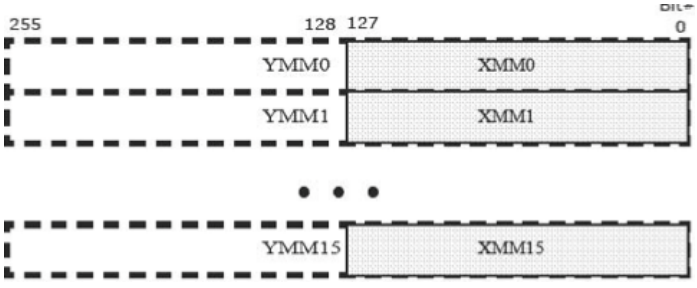
Register encoding	Not modified for 8-bit operands / Not modified for 16-bit operands / Zero-extended for 32-bit operands			Low 8-bit	16-bit	32-bit	64-bit
0			AH†	AL	AX	EAX	RAX
3			BH†	BL	BX	EBX	RBX
1			CH†	CL	CX	ECX	RCX
2			DH†	DL	DX	EDX	RDX
6				SIL‡	SI	ESI	RSI
7				DIL‡	DI	EDI	RDI
5				BPL‡	BP	EBP	RBP
4				SPL‡	SP	ESP	RSP
8				R8B	R8W	R8D	R8
9				R9B	R9W	R9D	R9
10				R10B	R10W	R10D	R10
11				R11B	R11W	R11D	R11
12				R12B	R12W	R12D	R12
13				R13B	R13W	R13D	R13
14				R14B	R14W	R14D	R14
15				R15B	R15W	R15D	R15
	63 32	31 16	15 8	7 0			

† Not legal with REX prefix ‡ Requires REX prefix

[그림 3 - 1] Register Operation

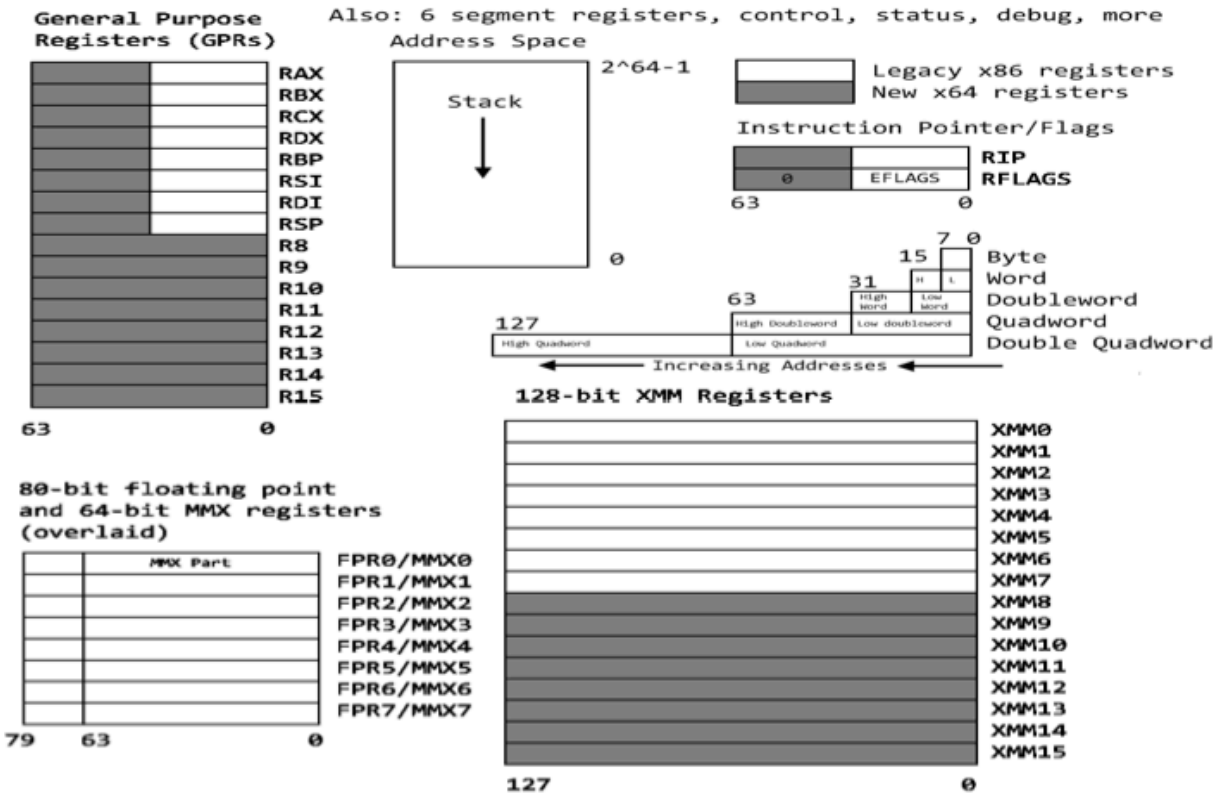
➡ 우선 Register 의 크기가 64bit 로 확장되면서 EAX -> RAX, ESP -> RSP, EIP -> RIP 와 같은 형식으로 Register 에 맨앞의 E 가 황제라는 의미인 R(ex) 로 바뀌었다.
➡ 그리고 R8 ~ R15 인 Register 가 8개 추가 되었다.
✓ R8 (QWORD), R8D (LOWER DWORD), R8W (LOWEST WORD) 와 같이 접근이 되어 진다.

또한 Media Register 가 xmm0 ~ xmm7 에서 xmm15 까지 총 16개가 되었고, Vector 및 부동 소수 점에 사용된다. 그리고 AVX (Advanced Vector Extension) 라는 256bit Register 8개 (ymm0 ~ ymm7) 가 추가 되었다.
✓ Intel Sandy Bridge 에서 최초로 적용되었다.



[그림 3 - 2] YMM 은 XMM 을 Overlap

기타 64bit Register 의 전체적인 그림은 다음과 같다.



[그림 3 - 3] 64bit Register 전체적인 구조

➡ 검정색 음영 처리 된 부분이 64bit 에서 확장 및 새로 생긴 부분이다.

64 bit Instruction

CDQE : Quad word 에서 Double word 로 변환하는 명령어

- ✓ 기존의 CDQ 와는 반대개념이다.

CMPSQ : RSI 와 RDI 를 비교하는 명령어

CMPXCHG16B : RDX:RAX 와 m128을 비교하는 명령어

- ✓ m128 은 128 bit Register 를 가리킨다.
- ✓ 이를 통해 pointer 사이즈 보다 큰 data 를 비교와 swap 할 수 있다.

LODSQ : RSI 주소에 있는 값을 RAX 에 Load 하는 명령어

MOVSQ : RSI 의 주소값을 RDI 로 이동하는 명령어

STOSQ : RDI 의 주소를 RAX 에 보관하는 명령어

SYSCALL : SYSENTER 와 같은 의미

SYSRET : SYSEXIT 와 같은 의미

4. Intel vs AT&T

AT&T 문법은 UNIX 계열에서 일반적으로 사용되는 것으로 Linux 에서 GDB 를 사용해 보았다면 이를 한번쯤은 봤을 것이다. 이를 Intel 문법과 간단하게 비교해 보겠다.

Register 이름	모든 Register 이름에는 %가 앞에 붙으며, 이는 레지스터와 다른 Symbol들을 혼동하지 않게 하는 장점이 있다. ex) Intel : EAX <-> AT&T : %EAX
Operand 순서	source operand 가 왼쪽, destination operand 가 오른쪽에 위치한며, Intel 문법에서는 이와 반대로 되어 있다. ex) Intel : MOV EAX, ECX <-> AT&T : MOVL %ECX, %EAX
Operand 크기지정	operand의 크기를 지정할 때 크기에 따라 접미사를 명령어에 붙인다. operand로 지정된 Register 로 크기를 판단할 수 있을 시 생략 가능하며, Default로 32-bit 연산으로 하게 된다. Intel에서는 BYTE PTR, WORD PTR, DWORD PTR 같은 지시자(specifier)를 사용하여 이를 나타낸다. ex) Intel : MOV BX, WORD PTR aaa <-> AT&T : MOVW aaa, %BX
부호 확장 시	AT&T 문법에서는 extend 명령어에 원래의 크기 (S) 와 확장할 크기 (D) 를 지정하게 한다. ex) Intel : MOVSX < - > AT&T : MOVSSD Intel : MOVZX < - > AT&T : MOVZSD
상수 & immediate	모든 상수와 immediate 값에는 $가 붙으며, 변수의 주소를 나타낼 때에도 앞에 $ 를 붙인다. ex) Intel : MOV EAX, aaa <-> AT&T : MOVL $aaa, %EAX (주소 복사) Intel : MOV EAX, [aaa] < - > AT&T : MOVL aaa, %EAX (값을 복사)
Memory 참조	Intel : section:[base + index * scale + immed32] AT&T : section:immed32(base, index, scale) 위와 같은 방식으로 Memory 를 참조하며 이는 base + index * scale + immed32 주소를 나타내게 된다. 반드시 모두 지정해야 하는 것은 아니지만 immed32나 base 중의 하나는 반드시 지정해야 한다. ex) Intel : [EAX] <-> AT&T : (%EAX) (EAX 가 가리키는 주소의 값 참조) Intel : [EAX+VAR] <-> AT&T : VAR(%EAX) (EAX + Offset 주소의 값 참조) Intel : [eax * 4 + array] <-> AT&T : array(, %eax, 4) (4Byte 배열 참조) Intel : [ebx + eax * 4 + array] <-> AT&T : array(%ebx, %eax, 4)
JMP / CALL / RET	long jump나 long call에서는 다음과 같은 차이가 있다 ex) Intel : jmp/call far section:offset <-> AT&T : ljmp/lcall $section, $offset
Far RET	ex) Intel : ret far stack-adjust <-> AT&T : lret $stack-adjust

[그림 4 - 1] Intel vs AT&T 비교

➡ AT&T Assembler에서는 여러개의 section을 지원하지 않는다.

✓ UNIX 프로그램에서는 모든 프로그램이 하나의 section에 있다고 생각을 한다.

5. Before “with ASM”

우선 ASM (Assembly Language) 로 Programming 및 분석을 하기전에 필요한 기본 개념들에 대해 설명해 보도록 하겠다.

Stack Frame 이란 현재 실행되고 있는 함수가 사용하기 위해 할당한 Stack 안의 영역이다. 함수에 전달되는 Parameter 와 반환주소가 저장되며, 함수가 사용하는 내부 저장공간 역할을 수행한다.
대부분의 함수는 Stack Frame 을 설정하는 것으로 시작한다. Parameter 영역과 지역변수 영역 사이에 Pointer 를 유지하면서 빠르게 접근하기 위함이다.
Pointer 는 일반적으로 EBP 에 저장되고, 반면에 ESP 에는 현재의 Stack 위치를 가리키기 위해서 사용된다.

ENTER 와 LEAVE 명령은 특정 Type 의 Stack Frame 을 구현하기 위해서 CPU 가 제공하는 명령이다.
ENTER 명령은 EBP 를 Stack 에 PUSH 하고 EBP가 지역 변수 영역의 맨 위를 가리키게 설정한다. 또한 동일한 함수 내의 중첩 Stack Frame 도 관리할 수 있게 한다.

✓ 매우 복잡하므로 성능 문제가 발생하여 사용을 잘 하지 않는다.

LEAVE 명령은 단순히 ESP 와 EBP 를 저장되기 이전 상태로 복원한다. 단순한 명령이므로 수많은 Compiler 가 함수의 끝 부분에 이 명령을 사용한다.

호출 규약이란 함수가 Program 내부에서 호출되는 방식을 정의한다. CALL 과 RET 로 기본적인 함수 호출을 하며 CALL 은 명령 Pointer 를 Stack 에 PUSH 하고 새로운 Code 로 JMP 한다. RET 명령은 반환주소를 EIP 로 POP 하고 EIP 가 가리키는 주소에서의 실행을 계속 수행한다.

cdecl	표준 C, C++ 호출 규약으로, 함수 호출자가 호출 이후의 Stack 복원을 담당하므로 Parameter 수를 동적으로 결정 가능하다. 하나이상의 Parameter 를 받는 함수가 반환할 때 RET 명령을 인자 없이 사용한다면 거의 확실히 cdecl 함수이다.
fastcall	Register 를 이용해 첫번째, 두번째 Parameter 를 함수에 전달하고, 나머지는 Stack 을 통해서 전달한다.
stdcall	가장 많이 사용되는 호출 규약으로, 함수 자체가 Stack 정리를 한다. RET 명령에 전달되는 인자의 값에서 해당 함수에 전달되는 Parameter 의 수를 알아낼 수 있다.
thiscall	함수 호출이 ECX 에 유효한 Pointer 를 Load 하고 Stack 에 Parameter를 Push 하고 EDX 를 사용하지 않는지 확인함으로써 알아낼 수 있다.

[표 5 - 1] 함수 호출 규약 비교

기본적인 **Stack** 에서의 동작을 살펴보았다. 아래에서는 기본적인 **Data** 구성 요소에 대해 살펴보도록 하겠다.

일반적으로 전역변수는 실행 Module 의 Data Section 의 고정된 주소에 위치한다. 그리고 전역 변수에 접근할 경우에는 Hard Coding 된 전역 변수 주소를 사용하므로 찾기 쉽다.
Compiler 는 Hard Coding 된 주소를 전역 변수 이외에는 거의 사용하지 않는다.
지역변수에 Static Keyword 를 사용하면 전역변수로 변환되어 모듈의 Data Section 에 저장된다. 이럴 때에는 해당 변수가 하나의 함수 내에서만 접근되는 지 확인함으로 판별 가능하다.

지역변수에는 일반적으로 Counter, Pointer, 다양한 종류의 일시적인 정보들이 들어간다. Compiler 는 지역변수를 Stack 에 저장하거나 Register 에 저장하는 방법으로 관리한다.

Import 된 변수는 다른 Binary Module (DLL) 에 저장되어 있는 전역 변수이다. Import 된 변수는 기타 다른 변수와는 다르게 이름을 가지므로 Reverser 에게 중요하다. 이는 가독성을 향상시킨다.

```
1    mov     eax,DWORD PTR [IATAddress]
2    mov     ebx,DWORD PTR [eax]
```

[그림 3 - 1] Import 변수에 접근하는 Code

➡ 다른 Pointer 를 가리키는 Pointer 를 통해 Data 를 간접적으로 읽어 들이고 있다. Import 된 변수인지 여부는 IATAddress 값으로 알 수 있다. 결국 첫번째 Pointer 가 IAT 를 가리키는 Pointer 가 되고 두번째가 Import 된 변수를 가리키는 Pointer 가 된다.

C 와 C++ 에서는 CONST Keyword 를 사용해서 상수를 정의 할 수 있다. 이렇게 하면 일반 전역 변수에 접근하는것 처럼 상수에 접근하는 Code 가 만들어진다.
다른 개발 tool 은 이러한 형태의 상수를 다른 전역 변수와 함께 Data Section 에 위치시킨다.

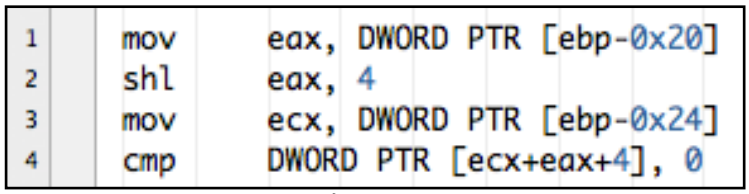
Window OS 는 TLS API (TlsAlloc, TlsGetValue, TlsSetValue) 로 TLS 저장소를 할당할 수 있다. 또 다른 방법으로는 전역 변수를 declspec (thread) 속성으로 정의함으로써, 실행 Image 의 Thread-local Section 에 해당 변수를 위치시킨다.

구조체란 여러 Data Type 의 Field 들이 모인 Memory Block 으로 각 Field 는 Memory Block 의 시작 위치에서부터 연속적으로 위치한다.
구조체의 마지막 Member 로 유동적인 Data 구조체를 만드는 것이 가능하며, 실행시에 특정 크기의 Data 구조체를 동적으로 할당하는 Code 를 만드는 것도 가능하다.

✓ 일반적으로 Processor 의 Word 크기로 정렬하여 성능 저하를 예방한다.

배열이란 Data 가 Memory 에 연속적으로 저장된 Data List 이다. Compiler 는 배열에 접근하기 위해서 거의 항상 어떤 종류의 변수를 객체의 Base 주소에 더하므로 배열에 접근하는 Code 를 쉽게 찾을 수 있다.

✓ Source Code 에 Hard Coding 된 배열의 Index 가 포함되어 있다면 구조체와 구별이 불가능
✓ 배열의 경우 구조체와는 달리 정렬을 수행하지 않는다. (Memory 낭비, 순차접근 때문)



```
1    mov     eax, DWORD PTR [ebp-0x20]
2    shl     eax, 4
3    mov     ecx, DWORD PTR [ebp-0x24]
4    cmp     DWORD PTR [ecx+eax+4], 0
```

[그림 3 - 2] 배열에 접근하는 Code

➡ 지역 변수 ebp - 0x20 을 EAX 에 Load 해서 왼쪽을 4bit Shift 시키므로 ebp - 0x20 은 Loop 의 Counter 이다. 주로 배열의 Index 에 이와 같은 연산을 수행한다.
➡ Shift 연산을 수행 후 ECX 에 배열의 Base Pointer 로 보이는 ebp - 0x24 를 Load 한 후 곱하기를 수행한 값과 4를 더한다. 이는 전형적인 Data Type 구조체에 대한 접근 Code 이다.
➡ 첫 번째 변수 (ECX) 는 배열의 Base Pointer 이고, 두 번째 변수 (EAX) 는 배열에서의 Byte Offset 값이다. 이 값은 현재의 Index 값에 배열의 Item 크기를 곱해서 구한다. 마지막으로 4를 더함으로써 우리는 Data 구조체의 2번째 Item 에 접근함을 알 수 있다.

현재까지 Stack 에서의 동작, 기본적인 Data 구성요소에 대해 살펴보았다. 이제는 함수와 관련해서 더 살펴보도록 하겠다.

IA-32 Processor 에서는 거의 항상 Call 명령으로 함수를 호출한다. 이는 현재의 명령 Pointer 를 Stack 에 저장하고, 해당 함수 주소로 JMP 하는 명령어 이다.

내부 함수는 구현 Code 를 포함하고 있는 동일한 실행 Binary에서 호출된다. Compiler 는 일반적으로 호출할 함수의 주소를 Code 에 삽입함으로써 내부 함수 호출 Code 를 만들어 낸다. 따라서 Code 내에서 내부 함수 호출을 쉽게 찾아 낼 수 있다.

✓ 일반적으로 " Call CodeSectionAddress " 와 같은 방식을 띈다.

Import 된 함수는 Module 이 다른 실행 Binary 내에 구현된 함수를 호출할 때 발생한다. 이는 IAT (Import Address Table) 을 이용해서 호출을 구현하며, Import Directory 는 실행 시에 해당 실행 Binary 에 있는 함수와 Matching 시키기 위해 사용되며 IAT 는 해당 함수의 실제 주소를 저장한다.

✓ 전형적으로 " Call DWORD PTR [IAT_Pointer] " 와 같은 방식을 띄며, 여기에서 DWORD PTR 이 사용된다는 것은 IAT_Pointer 의 주소가 아닌 IAT_Pointer 가 가리키는 주소로 JMP 하라는 것이다.

여태까지의 이론들을 가지고 이제는 Programming 및 분석을 해보겠다.

6. “with ASM”

우선 Inline ASM 으로 계산기를 만들어 보겠다. Inline 으로 만드는 이유는 간단한 ASM 의 동작방식을 익히고 실제 Matching 되는 것을 확인하는 것이 현재 목표이고, Program 분석 시 해당 Routine 을 구현하고 싶을 때 굳이 C 언어로 변환하지 않고 Inline 으로 재 구현하는 경우가 많기 때문이다.

```
int Add(int num1,int num2)
{
	__asm // inline asm 을 적을 때에는__asm 을 붙여준다.
	{
		mov eax, num1; //eax 에num1 값을 넣음
		mov ebx, num2; //ebx 에num2 값을 넣음
		add eax, ebx;	//eax 에ebx 를 더함
		mov num1, eax; // num1 에eax 를 넣음
	}
	return num1; //num1 반환
}

int Sub(int num1,int num2)
{
	__asm
	{
		mov eax, num1; //eax 에num1 값을 넣음
		mov ebx, num2; //ebx 에num2 값을 넣음
		sub eax,ebx;	//eax 에서ebx 를 뺌
		mov num1,eax;  // num1 에eax 를 넣음
	}
	return num1; //num1 반환
}

int Mul(int num1,int num2)
{
	__asm
	{
		mov eax, num1; //eax 에num1 값을넣음
		mov ebx, num2; //ebx 에num2 값을넣음
		mul ebx;		//ebx 에eax 를 곱함( edx:eax 에저장)
		mov num1,eax;  //num1 에eax 를 넣음
	}
	return num1; //num1 반환
}

int Div(int num1,int num2)
{
	__asm
	{
		mov eax,num1;	//eax 에num1 값을 넣음
		mov ecx,num2;	//ecx 에num2 값을 넣음
		xor edx,edx;	//edx 를 초기화( 나머지값)
		div ecx;		//edx:eax 에서ecx 를 나눔
		mov num1,eax;  //num1 에eax 를 넣음
	}
	return num1; //num1 반환
}
```

```
0-> Finish
1 -> Add, 2 -> Sub, 3 -> Mul, 4 -> Div
Select : 1
        First number : 1234
        Second number : 5678
Result is 6912

0-> Finish
1 -> Add, 2 -> Sub, 3 -> Mul, 4 -> Div
Select : 2
        First number : 6912
        Second number : 5678
Result is 1234

0-> Finish
1 -> Add, 2 -> Sub, 3 -> Mul, 4 -> Div
Select : 3
        First number : 1234
        Second number : 2
Result is 2468

0-> Finish
1 -> Add, 2 -> Sub, 3 -> Mul, 4 -> Div
Select : 4
        First number : 8642
        Second number : 2
Result is 4321
```

원래 계획은 Assembly Language 로 된 Code 를 주석을 달아가며 C Code 화 시키는 것을 담으려고 했는데, 간단한 Lotto Program 이라도 Code 의 길이가 너무 길어 가독성을 해친다고 판단되어 넣지 않았다.
이 부분은 나중에 작성할 문서인 " **Hacking** 대회 **Binary Reversing** 문제 **All** 풀이 " 에서 다루도록 하겠다.

Assembly 라는 언어는 Computer 에서 없어서는 않 될 필수 언어이다.
이 문서를 통해 Assembly Language 의 기본과 Code 분석에 필요한 개념들을 살펴 보았다 문서를 읽고 많은 사람들이 **Computer** 를 파고드는 삽질인 Reversing 에 관심과 흥미를 가졌으면 좋겠다.