

< x64 어셈블리 언어 >

1) 기본 구조 : eax에 3을 대입하라

2) 주요한 명령어들

명령 코드	
데이터 이동(Data Transfer)	<code>mov, lea</code>
산술 연산(Arithmetic)	<code>inc, dec, add, sub</code>
논리 연산(Logical)	<code>and, or, xor, not</code>
비교(Comparison)	<code>cmp, test</code>
분기(Branch)	<code>jmp, je, jg</code>
스택(Stack)	<code>push, pop</code>
프로시저(Procedure)	<code>call, ret, leave</code>
시스템 콜(System call)	<code>syscall</code>

3) 피연산자



* 메모리 피연산자 특징

- []으로 둘러싸인 것으로 표현됨
IA-16 당시 CPU의 Word가 16bit에서, Word가 16bit(=2byte)
- 앞에 크기 지정자가 붙음: BYTE, WORD, DWORD, QWORD
(1 byte) (2 byte) (4 byte) (8 byte)
- 사용법 예시

메모리 피연산자

QWORD PTR [0x8048000]	0x8048000의 데이터를 8바이트만큼 참조
DWORD PTR [0x8048000]	0x8048000의 데이터를 4바이트만큼 참조
WORD PTR [rax]	rax가 가르키는 주소에서 데이터를 2바이트 만큼 참조

< 데이터 이동 >

1) **MOV dst, src** src에 있는 값을 dst에 대입한다

ex) `mov rdi, rsi` : rsi에 있는 값을 rdi에 대입한다.

`mov QWORD PTR[rdi], rsi` : rsi에 있는 값을 rdi가 가리키는 주소에 대입 (QWORD: 8byte)

`mov Qword PTR[rdi+8*rcx], rsi` : rsi에 있는 값을 rdi+8*rcx 가 가리키는 주소에 대입

2) **lea dst, src** src의 유효주소(effective address)를 dst에 저장한다.

ex) `lea rsi, [rbx+8*rcx]` : rbx+8*rcx 를 rsi에 저장한다.

3) 예제

[Register]	rbx = 0x401A40
<hr/>	
[Memory]	0x401a40 0x0000000012345678 0x401a48 0x000000000000FFFF 0x401a50 0x00000000DEADBEF 0x401a58 0x00000000CAFEBABE 0x401a60 0x00000000887654321
<hr/>	
[Code]	1: mov rax, [rbx+8] // 결과: rax에 0xC0FFEE 저장됨 2: lea rax, [rbx+8] // 결과: rax에 0x401a48 저장됨

⇒ 1코드: rbx+8 주소에 저장된 값을 저장

⇒ 2코드: rbx+8의 주소를 rax에 저장

< 산술연산 > : 덧셈, 뺄셈

1) **add dst, src** dst에 src 값을 더한다

ex) `add eax, 3` eax += 3

`add ax, word PTR[rdi]` ax += *(word*)rdi

(rdi는 []) 라서 주소값 전부 word형 크기(2byte) 변환 후 *연산

= rdi의 값을 2byte 만큼 가져온 주소의 값을 ax에 더한다)

2) **sub dst, src** dst에서 src의 값을 뺀다

ex) `sub eax, 3` eax -= 3

[Register]	rax = 0x31337 rbx = 0x555555554000 rcx = 0x2 // rcx*8 = 0x10 또는 16
<hr/>	

[Memory]	0x555555554000 0x0000000000000000 0x555555554008 0x0000000000000001 0x555555554010 0x0000000000000003 0x555555554018 0x0000000000000005 0x555555554020 0x0000000000000133A
----------	--

코드1

[rbx+8] = [rbx + 0x10] = 0x3

⇒ rax += 0x3 ⇒ 0x3133A

코드2

`add rcx, 2` ⇒ rcx == 0x4

코드3

[rbx + rcx*8]

⇒ rax += 0x133A

sub ax, word PTR[rdi] ax = *(word*)rdi

- 3) inc op op의 값을 1증가시킴 ex) inc eax
dec op op의 값을 1감소시킴 ex) dec eax

```
[Code]
1: add rax, [rbx+rcx*8] rax + 0x0 == 03 = 0x3133A
2: add rcx, 2 0x4 3133A
3: sub rax, [rbx+rcx*8] rax - (rbx + 0x20) = 0
4: inc rax 0x1
```

= Lrbx + 0x20 = 5133A
rax - 3133A == 0

코드4

inc rax == 1

<논리연산> and, or

- 1) and dst,src : dst와 src의 비트가 모두 1이면 1, 아니면 0

- 2) or dst,src : dst et src의 비트 중 하나라도 1이면 1, 아니면 0

[ex]

$\text{eax} = 0xfffff0000$ } and연산시 둘다 1이면 값이 작은쪽이 답 (cafe)
 $\text{ebx} = 0xcafebabe}$ } or 연산시 1이면 값이 큰쪽이 답 (ffff)

and eax, ebx

or eax, ebx

```
[Register]
rax = 0xfffffffff00000000
rbx = 0x00000000ffffffff
rcx = 0x123456789abcdef0

=====
[Code]
1: and rax, rcx // 0x1234567800000000
2: and rbx, rcx // 0x000000009abcdef0
3: or rax, rbx // 0x123456789abcdef0
```

and $\begin{array}{ccccccc} \text{1111} & \text{1111} & \text{1111} & \text{0000} & \text{0000} & \text{0000} & \text{0000} \\ \text{1100} & \text{1010} & \text{1110} & \text{1011} & \text{1010} & \text{1011} & \text{1110} \\ \text{1100} & \text{1010} & \text{1110} & \text{1011} & \text{1010} & \text{1011} & \text{1110} \\ \hline \text{C} & \text{a} & \text{f} & \text{e} & \text{0} & \text{0} & \text{c} & \text{0} \end{array}$	$\begin{array}{ccccccc} \text{1111} & \text{1111} & \text{1111} & \text{0000} & \text{0000} & \text{0000} & \text{0000} \\ \text{1100} & \text{1010} & \text{1110} & \text{1011} & \text{1010} & \text{1011} & \text{1110} \\ \text{1100} & \text{1010} & \text{1110} & \text{1011} & \text{1010} & \text{1011} & \text{1110} \\ \hline \text{f} & \text{f} & \text{f} & \text{f} & \text{b} & \text{a} & \text{b} & \text{e} \end{array}$
---	--

<논리연산> xor, not

- 1) xor dst,src : dst와 src의 비트가 다른면 1, 같으면 0

- 2) not eax : eax의 비트 전부 반전

- 3) 연습문제

$$\begin{aligned} a &= 10 & e &= 14 \\ b &= 11 & f &= 15 \\ c &= 12 & & \\ d &= 13 & & \end{aligned}$$

$$* 0101 \Rightarrow 2^2 + 2^0 = 4 + 1 = 5$$

```
[Register]
rax = 0x35014541
rbx = 0xdeadbeef

=====
[Code]
1: xor rax, rbx
2: xor rax, rbx
3: not eax
```

$\text{rax: } 0011\ 0101\ 0000\ 0001\ 0100\ 0101\ 0100\ 0001$ $\text{rbx: } 1101\ 1110\ 1010\ 1101\ 1011\ 1110\ 1110\ 1111$	$\begin{array}{cccccccccc} & & & & & & & & & \\ 1) & \text{Xor: } & 1110 & 1011 & 1010 & 1100 & 1111 & 1011 & 1010 & 1110 \\ & & e & b & a & c & f & b & a & e \\ & & 3 & 5 & 0 & 1 & 4 & 5 & 4 & 1 \\ 2) & \text{Xor: } & 0011 & 0101 & 0000 & 0001 & 0100 & 0101 & 0100 & 0001 \\ & & 3 & 5 & 0 & 1 & 4 & 5 & 4 & 1 \\ 3) & \text{not: } & 1100 & 1010 & 1111 & 1110 & 1011 & 1010 & 1011 & 1110 \\ & & 12 & 10 & 15 & 14 & 11 & 10 & 11 & 14 \\ & & c & a & f & e & b & a & b & e \end{array}$
--	---

<비교> cmp, test, jmp, je, jg

- 1) cmp op1, op2 op1과 op2를 빼어서 결과를 비교하는데, 연산결과는 op1에 저장 X ZF가 1이면 두수가 같다.

- 2) test op1, op2 두 피연산자에 and 비트연산을 취함. 0인 rax을 test rax,rax 하면 ZF가 1로 되고 rax=0이다.

- 3) jmp addr addr로 rip를 이동시킨다

- 4) je addr 직전에 비교한 두 피연산자가 같으면 점프 (jump if equal)

- 5) jg addr 직전에 비교한 두 피연산 // 중 전자가 크면 점프 (jump if greater)

< Opcode: 스택> // 원래 있던 스택의 공간에 넣기 X, 스택 공간 자체를 확장

1) push val : val을 스택 최상단에 쌓음

예제

```
1 [Register]
2 rax = 0x7fffffff400
3
4 [Stack]
5 0x7fffffff400 | 0x0 <= rsp
6 0x7fffffff408 | 0x0
7
8 [Code]
9 push 0x1337
```

결과

```
1 [Register]
2 rax = 0x7fffffff400
3
4 [Stack] // Stack rsp 자체가 바뀜
5 0x7fffffff408 | 0x31337 <= rsp
6 0x7fffffff400 | 0x0
7 0x7fffffff408 | 0x0
```

2) pop reg : 스택 최상단의 값을 꺼내서 reg에 대입

예제

```
1 [Register]
2 rax = 0
3
4 [Stack]
5 0x7fffffff400 | 0x31337 <= rsp
6 0x7fffffff408 | 0x31337 <= rsp
7 0x7fffffff400 | 0x0
8 0x7fffffff408 | 0x0
9
10 [Code]
11 pop rax
```

결과

```
1 [Register]
2 rax = 0x31337
3
4 [Stack]
5 0x7fffffff400 | 0x31337 <= rsp
6 0x7fffffff408 | 0x31337 <= rsp
7 0x7fffffff400 | 0x0
8 0x7fffffff408 | 0x0
9
10 [Code]
11 pop rax
```

< Opcode: 프로시저 >

* 프로시저란?: 특정 기능을 수행하는 코드 모음

↳ 장점1: 반복되는 연산을 프로시저 호출로 대체할 수 있어서 코드의 크기 ↓

↳ 장점2: 기능별로 코드 조작에 이름을 붙일 수 있어서 코드의 가독성 ↑

* call: 프로시저를 부르는 행위(호출)

{ 2) 프로시저를 호출할 때는, 프로시저 실행 후 원래의 흐름으로 돌아와야 하므로
 { 3) 다음의 명령어 주소(반환주소)를 stack에 저장하고 프로시저를 rip로 이동시킨다.

* return: 프로시저에서 돌아오는 것(반환)

1) call addr : addr에 위치한 프로시저 호출

연산 (push return_address
jmp addr)

2) leave 스택프레임 정리

연산 (mov rsp, rbp
pop rbp)

3) ret 반환주소로 반환

연산: pop rip

< 스택프레임의 할당과 해제 >

① 0x400000 call func ← rip
0x400005 mov esf, eax

main

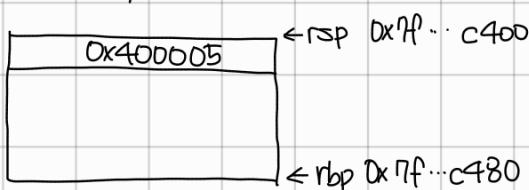
②

0x401000 push rbp ← rip
0x401001 mov rbp, rsp
0x401004 sub rsp, 0x30
0x401008 mov BYTE PTR [rsp], 0x3
...
0x401020 leave
0x401021 ret

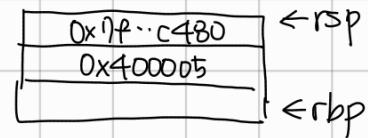
func

func 함수를 호출합니다. 다음 명령어 주소 0x40005는

stack에 push됩니다.



func 함수가 호출되었고, 기존 스택프레임 저장을 위해
rbp를 stack에 push합니다.



③

func

0x401000 push rbp

0x401001 mov rbp, rsp



```

0x401004 sub rsp, 0x30
0x401008 mov BYTE PTR [rsp], 0x3
...
0x401020 leave: mov rbp, rsp
0x401021 ret

```

rspl/rbp를 블리 공간 줄이기
상수의 값(줄여야 하는 값)을 rbp에 저장

mov rbp, rsp // 새 stack frame을 만들기 위해 rbp를 rsp로 설정합니다.
sub rsp, 0x30 // 공간 확장을 위해 rsp에서 0x30을 뺍니다
mov BYTE PTR [rsp], 0x3 // stack frame에 지역 변수 할당

< Opcode: 시스템콜 >

* 커널 모드

: 운영체제가 전체 시스템을 제어하기 위해 system software에 부여하는 권한

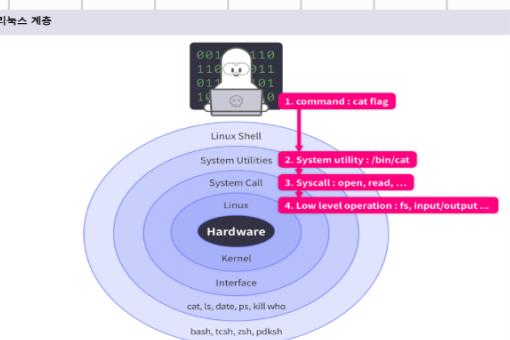
→ 모든 저수준 작업은 커널에서 진행

* 유저 모드

: 운영체제가 사용자에게 부여하는 권한

* system call

: 유저모드에서 커널모드의 시스템 소프트웨어에게 어떠한 동작을 요청하기 위해 사용



< Opcode: 시스템콜 >

시스템콜은 함수이다. ⇒ 필요한 기능과 인자에 대한 정보를 레지스터로 전달하면,

커널이 이를 읽어서 요청을 처리한다.

* rax: 무드번호인지? (syscall number)

* rdi rsi rdx rcx rb r9 stack 순서로 인자를 사용

< 퀴즈1 >

퀴즈 1 / 1
end로 점프하면 프로그램이 종료된다고 가정하자. 프로그램이 종료됐을 때, 0x400000부터 0x400019까지의 데이터를 대응되는 아스키 문자로 변환하면 어느 문자열이 나오는가?

```

Welcome[]
to[]assembly[]
world!
=====
[Memory]
0x51 0x66 0x6c 63 6f 6d 65 20
0x400000 | 0x67 0x55 0x5c 0x53 0x5d 0x55 0x10
0x400008 | 0x44 0x5f 20 0x51 0x52 0x53 0x55 0x5d
0x400010 | 62 6c 7a 20 0x71 6f 0x72 6c
0x400018 | 64 24 0x80 0x00 0x00 0x00 0x00 0x00
=====
[code]
1: mov dl, BYTE PTR[rsi+rcx] 0x67
2: xor dl, 0x30 0x51 0x67 ) 0~25나 26번
3: mov BYTE PTR[rsi+rcx], dl
4: inc rcx rax=1
5: cmp rcx, 0x19
6: jg end
7: jmp 1

```

$$1: \text{BYTE PTR}[rsi+rcx] = *(BYTEx*)rsi+rcx \\ = *(BYTEx) 0x400000 \Rightarrow \text{1byte} \text{로 } 0x00 \text{ or } 0xFF$$

$$dl = 0x67$$

$$2. \begin{array}{r} 0110 \quad 0111 \\ 0011 \quad 0000 \\ \hline \end{array} \\ \text{XOR } \begin{array}{r} 0101 \\ 4 \quad 1 \end{array} \quad \begin{array}{r} 0111 \\ 4 \quad 2 \end{array} = 0x57$$

$$3. 0x67 \rightarrow 0x57$$

$$4. rcx = 0 \rightarrow rcx = 1$$

$$5. 0x01 \text{과 } 0x19 \text{ 비교 dst < src } ZF = 0 \text{ (jg end 실행 X)}$$

〈퀴즈2〉

퀴즈 1 / 1

다음 어셈블리 코드를 실행했을 때 출력되는 결과로 올바른 것은?

다음 문제



```
[Code]
main:
    push rbp
    mov rbp, rsp
    mov esi, 0xf //esi=0xf
    mov rdi, 0x400500 //rdi=0x400500
    call 0x400497 <write_n>
    mov eax, 0x0
    pop rbp
    ret
```

```
write_n:
    push rbp //main의 rbp
    mov rbp, rsp
    mov QWORD PTR [rbp-0x8],rdi
    mov DWORD PTR [rbp-0xc],esi
    xor edx, edx //0 edx=0
    mov edx, DWORD PTR [rbp-0xc] edx=0xf
    mov rsi,QWORD PTR [rbp-0x8] rsi=0x400500
    mov rdi, 0x1 rdi=1
    mov rax, 0x1 rax=1
    syscall
    pop rbp
    ret
```

 1 0x400500 0xf(15권).

```
[Memory]
0x400500 | 0x387201454372 ↗ 렉스엔디안 72 33 34 64 79 20 37 30 ? ⇒ r34Ty 70 d3bub?
0x400508 | 0x0000000000000000
```

