x86 어셈블리 (1)

어셈블리어와 x86-64

어셈블리어

컴퓨터의 기계어와 치환되는 언어. 이후 내용은 x64 어셈블리어를 대상으로 함.

어셈블리어의 기본 규칙

- Intel 문법
  - NASM 에서 사용
- AT&T 문법
  - GAS에서 사용

x86-64 어셈블리어

기본 구조

```
명령어[opcode] 피연산자 1[op1] 피연산자 2[op2]
```

명령어

데이터 이동	mov, lea
산술 연산	inc, dec, add, sub
논리 연산	and, or, xor, not
비교	cmp, test
분기	jmp, je, jg
스택	push, pop
프로시저	call, ret, leave
시스템 콜	syscall

피연산자

- 상수
- 레지스터
- 메모리
  - []로 둘러싸인 것으로 표현되며, 앞에 크기 지정자인 `TYPE PTR` 올 수 있음
    - TYPE
      - `BYTE` 1바이트
      - `WORD` 2바이트
      - `DWORD` 4바이트
      - `QWORD` 8바이트

피연산자 표기법

r	레지스터. 이후 붙은 숫자는 비트 의미
imm	부호 있는 정수. 붙은 숫자는 비트
r/m	범용 레지스터 or 메모리. 붙은 숫자는 비트.

산술 연산과 논리 연산

add, sub, mul, div

사칙연산

- add : 덧셈
- sub : 뺄셈
- mul, imul : 곱셈
- div, idiv : 나눗셈

imul, idiv → 부호 있는 연산

add

```
add <destination>, <source>
```

dst에 src 더한 결과를 dst에 저장.

sub

```
sub <destination>, <source>
```

dst에 src 뺀 결과를 dst에 저장.

mul

`EAX` 레지스터에 보관된 값과 피연산자를 곱하는 명령어

mul 레지스터

```
mul <source>
```

mul 명령어는 임시적으로 RAX 레지스터를 첫 번째 피연산자로 사용한다. 그리고 source에 레지스터 혹은 메모리 주소를 두 번째 피연산자로 사용. 부호 없는 정수에 대해 곱셈 수행.

n비트와 n비트 곱셈의 값은 n비트를 넘을 수 있으므로 레지스터를 두 개 사용해 상위 n비트와 하위 n비트를 나눠 저장한다.

예를 들어 **EAX**와 **ESI** 두 레지스터를 곱한다고 가정했을 때, 32비트와 32비트의 곱셈 결과는 최대 64비트까지 나올 수 있기 때문에, 32비트 레지스터 하나에 결과를 보관하기엔 부족하므로 32비트 레지스터를 하나 더 사용해 상위 32비트와 하위 32비트를 나눠서 보관합니다.

- 8비트 x 8비트

AL 레지스터의 값과 `src` 피연산자의 값을 곱합니다. 두 숫자의 연산 결과는 최대 16비트 이기때문에, **AX (16비트)** 레지스터에 보관합니다.

- 16비트 x 16비트

AX 레지스터의 값과 `src` 피연산자의 값을 곱합니다. 곱한 결과의 상위 16비트는 **DX** 레지스터, 하위 16비트는 **AX** 레지스터에 보관합니다.

- 32비트 x 32비트

EAX 레지스터의 값과 `src` 피연산자의 값을 곱합니다. 곱한 결과의 상위 32비트는 **EDX** 레지스터, 하위 32비트는 **EAX** 레지스터에 보관합니다.

- 64비트 x 64비트

RAX 레지스터의 값과 `src` 피연산자의 값을 곱합니다. 곱한 결과의 상위 64비트는 **RDX** 레지스터, 하위 64비트는 **RAX** 레지스터에 보관합니다.

imul

```
imul <source> ; 피연산자가 1개
imul <destination>, <source> ; 피연산자가 2개
imul <destination>, <source>, <immediate> ; 피연산자가 3개
```

부호 있는 (signed) 정수에 대해 곱셈 수행.

- 피연산자 1개인 경우 : 정해진 레지스터와 source를 곱한 결과를 더 큰 레지스터에 저장
- 피연산자 2개인 경우 : src와 dst 곱한 결과를 dst에 대입
- 피연산자 3개인 경우 : src와 immediate(즉시값)을 곱한 결과를 dst에 대입

만약 연산 결과가 목적지 피연산자의 크기를 초과하여 데이터 손실이 발생하면 캐리 플래그(CF)와 오버 플로우 플래그(OF)를 1로 설정하여 이를 알린다.

div

부호 없는 나눗셈

```
div/idiv 제수
```

나눗셈을 수행할 피제수의 크기에 따라 AX, DX:AX, EDX:EAX, RDX:RAX에 저장하고, 수행한 결과의 몫과 나머지를 각 크기에 따라 AL:AH, AX:DX, RAX:RDX에 저장한다.

논리 연산

AND

두 개의 비트가 모두 1일 때만 결과가 1

```
AND destination, source
```

OR

두 개의 비트 중 하나라도 1이면 1

```
OR destination, source
```

XOR

```
XOR destination, source
```

두 개의 비트가 다를 때만 결과가 1

NOT

```
NOT destination
```

비트를 반전

INC, DEC

inc : 1씩 증가

dec : 1씩 감소

데이터 저장과 이동

mov

데이터를 복사/이동

```
mov <destination>, <source>
```

lea

메모리에 실제 접근을 하지 않고 **유효 주소**를 저장하기 위해 사용된다.

유효 주소 : 포인터. 연산의 대상이 되는 데이터가 저장된 위치

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = &arr[2]; // 유효 주소
```

```
mov rbx, &arr  ; 배열 arr의 시작 주소를 rbx 레지스터에 복사
mov rcx, 2     ; 배열의 인덱스 값을 rcx 레지스터에 복사
lea rax, [rbx + rcx * 4]  ; 배열의 세 번째 원소를 가리키는 주소 값 rax = rbx + (rcx * 4) = 1008
```

```
lea <destination>, <source>
```

mov와 lea의 차이

- mov

  c언어에서

  ```
  int val = arr[2]
  ```

  asm에서

  ```
  mov rbx, &arr ; rbx = arr의 주소
  mov rcx, 2 ; 배열의 인덱스
  mov rax, [rbx + rcx * 4] ; arr[2]에 접근해 값을 rax에 대입
  ```

- lea
  - C언어에서

  ```
  int *ptr = &arr[2];
  ```

```
mov rbx, &arr ; rbx = arr의 주소
mov rcx, 2 ; 배열의 인덱스
lea rax, [rbx + rcx * 4] ; arr[2]의 주소를 rax에 대입
```

비교 연산과 분기문

cmp

```
cmp <destination>, <source>
```

두 피연산자를 빼는 방식으로 플래그 갱신.

dst - src 값을 판단해 플래그 설정.

```
[Code]
1: mov rax, 0xA
2: mov rbx, 0xA
3: cmp rax, rbx ; ZF=1
```

test

```
test <destination>, <source>
```

dst와 src AND 연산한 뒤 플래그 레지스터 갱신.

보통 특정 비트가 0인지 아닌지 확인하거나 레지스터가 0인지 검사하는 데 사용.

두 연산자 중 하나라도 0이면 자동으로 ZF 플래그 1로 설정된다.

```
[Code]
1: xor rax, rax //0
2: test rax, rax  ; ZF=1 ; 레지스터 rax가 0인지 확인
```

분기

```
jmp <label>
je <label>
jz <label>
...
```

jmp

조건 없이 무조건 점프

je/jz

je(jump if equal) : 같으면 점프

jz(jump if zero) : 같으면 점프

jg/jge

jg: Jump if greater

jge : jump if greater or equal

jl/jle

jl : Jump if less

jle : Jump if less or equal

jne/jnz

jne : Jump if not equal

jnz : Jump if not zero

반복문

카운터 기반 반복문

```
    mov rcx, 10  ; 반복 횟수 설정
loop_start:
```

```
    ; 반복할 코드
    loop loop_start  ; RCX를 감소시키며 0이 아닐 때까지 반복
```

조건 기반 반복문

```
check_condition:
    cmp rax, 50   ; 특정 조건 확인
    jge loop_exit ; 조건이 만족되면 종료
    ; 반복할 코드
    jmp check_condition ; 다시 비교 연산으로 이동
loop_exit:
```

후조건 검사 반복문 : 최소 한 번 실행 뒤 조건 검사

```
do_loop:
    ; 반복할 코드 실행
    cmp rax, 10   ; 조건 검사
    jl do_loop    ; 조건이 만족하면 다시 실행반복문에서 주의할 점
```