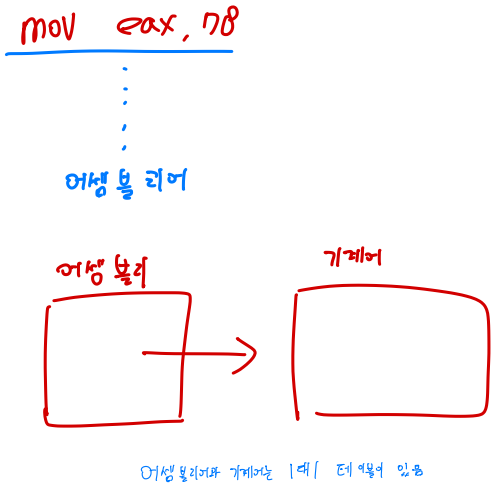
Assembly programming

(nasm, ~~intel syntax~~, 32-bit, Windows 환경)

X86

목차

- x86
- x86 어셈블리언어 개요
- x86 명령어 (32bit)
- x86 어셈블리언어 예시



x86

- **8086** → (1978년) 인텔이 설계한 16-bit 마이크로프로세서(microprocessor)
- **80386** → (1985년) 인텔이 설계한 32-bit 마이크로프로세서로 이후 i386으로 불림
- **x86** → 8086에 기반한 인텔의 CISC(Complex Instruction Set Computer) instruction set architecture
- **IA-32 (Intel Architecture, 32-bit)** → x86 instruction set의 32-bit 버전들의 통칭으로, 80386 마이크로프로세서에서 최초 구현
- **x86-64** → x86 instruction set의 64-bit 버전

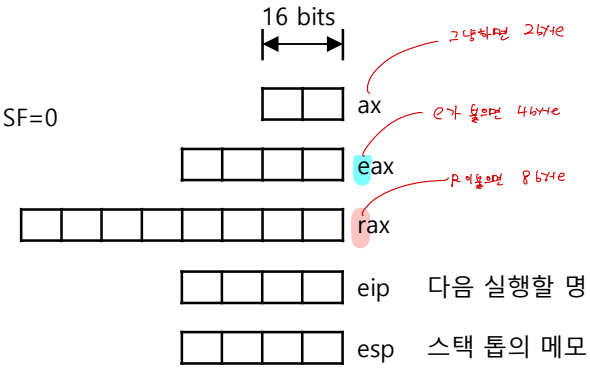
과거에 설계해둔 것을 이어서 사용하고 있다.

x86	x86-16 (16-bit)	8086 80286	범용 레지스터 (16-bit)	AX, BX, CX, DX, SI, DI, SP, BP
			Program counter	IP
			상태 레지스터	FLAGS
	IA-32 (32-bit)	80386 80486 i586 i686	범용 레지스터 (32-bit)	EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
			Program counter	EIP
			상태 레지스터	EFLAGS
	x86-64 (64-bit)		범용 레지스터 (64-bit)	RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP
			Program counter	RIP
			상태 레지스터	RFLAGS

8086 상태 레지스터(16-bit) FLAGS

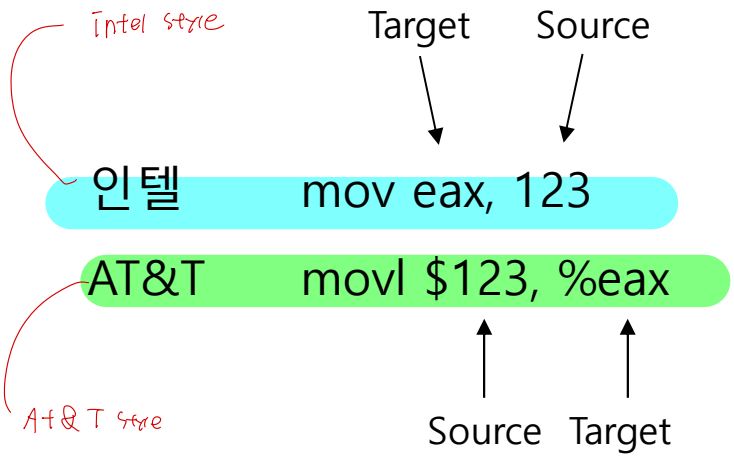
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

- SF(Sign Flag) → 가장 최근 연산 결과의 MSB가 1이면(음수이면) SF=1, 그렇지 않으면 SF=0
- ZF(Zero Flag) → 가장 최근 연산 결과의 0이면 ZF=1, 그렇지 않으면 ZF=0



x86 어셈블리언어

- x86 어셈블리언어는 Intel syntax와 AT&T syntax로 작성될 수 있음
- Intel syntax는 Windows 환경에서, AT&T syntax는 Unix 및 Linux 환경에서 주로 사용됨
- NASM(Netwide assembler)을 포함한 많은 x86 어셈블러들은 Intel 스타일을 따르며, GAS(GNU Assembler)는 디폴트로 AT&T 스타일을 따름
- 아래 두 명령어는 4바이트로 표현된 123을 eax 레지스터에 저장하는 동일한 동작을 수행하지만 그 작성 스타일은 다름

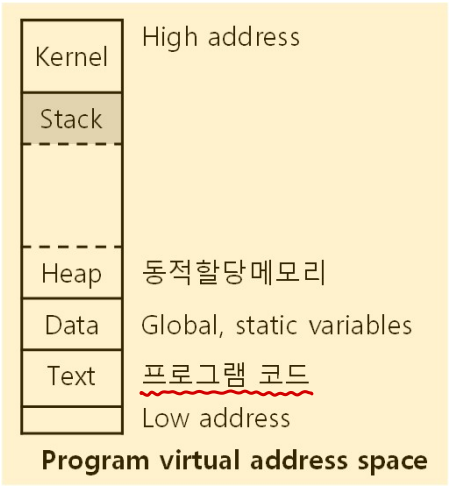


x86 instructions (32-bit)

명령어	설명
mov eax, 255	eax 레지스터에 4바이트의 255를 저장
mov ebp, esp	esp 레지스터에 저장된 값을 ebp 레지스터로 복사
mov eax, [ebp+8]	ebp에 저장된 메모리 주소에 8을 더한 메모리 주소에 저장된 4바이트를 eax로 복사
mov [ebp+8], eax	eax에 저장된 값을 ebp에 저장된 메모리 주소에 8을 더한 메모리 주소로 복사
add esp, 4	esp 레지스터에 저장된 값에 4를 더한 값을 esp에 저장
sub eax, 4	esp 레지스터에 저장된 값에서 4를 뺀 값을 esp에 저장
cmp eax, 4	eax 레지스터에 저장된 값에서 4를 빼는 연산 수행(연산 수행 결과에 따라 상태 레지스터 필드 변경됨)
imul eax, 4	eax 레지스터에 저장된 값에 4를 곱한 값을 eax에 저장 (signed **i**nteger **mul**tiplication)
jmp *label*	label에 해당하는 주소로 jump
je *label*	ZF=1이면 label에 해당하는 주소로 jump (**j**ump if **e**qual, ==)
jne *label*	ZF=0이면 label에 해당하는 주소로 jump (**j**ump if **n**ot **e**qual, !=)
jg *label*	~(SF^OF) & ~ZF이면(즉 SF=OF이고 ZF=0) label에 해당하는 주소로 jump (**j**ump if **g**reater than, signed >)
jge *label*	~(SF^OF)이면(즉 SF=OF) label에 해당하는 주소로 jump (**j**ump if **g**reater than or **e**qual, signed >=)
jl *label*	(SF^OF)이면(즉 SF≠OF) label에 해당하는 주소로 jump (**j**ump if **l**ess than, signed <)
jle *label*	(SF^OF) | ZF이면 (즉 SF≠OF 혹은 ZF=1) label에 해당하는 주소로 jump (**j**ump if **l**ess than or **e**qual, signed <=)
push 123	4바이트로 표현된 123을 스택에 push (esp의 값을 4 감소 후 esp가 가리키는 메모리 주소에 123을 저장)
push eax	eax의 값을 스택에 push (esp의 값을 4 감소 후 esp가 가리키는 메모리 주소에 eax의 내용을 저장)
pop eax	스택의 톱(top)의 내용을 eax로 복사 (esp가 가리키는 메모리 주소에 있는 값을 eax로 복사 후 esp의 값을 4 증가)
call *label*	eip에 저장된 (다음 실행될 명령어의) 주소를 스택에 push한 후 label에 해당하는 주소로 jump
ret	스택에서 pop한 주소로 jump

~	NOT
&	AND
|	OR
^	XOR

x86 어셈블리언어 스택 사용 예시 #0



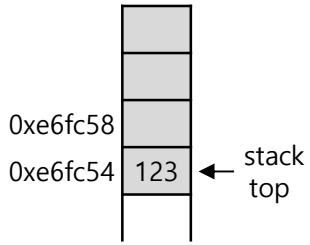
- esp는 스택의 톱 위치에 해당하는 메모리 주소를 저장하고 있음
- push eax → esp 값 4 감소 후, esp 값에 대응하는 메모리 주소에 eax 값 복사
- pop edx → esp 값에 대응하는 메모리 주소에 있는 값을 edx로 복사 후, esp 값 4 증가

pgm.asm NASM에서 어셈블러 작성법

```
section .text
    global _start
    extern _printf
_start:
    mov eax,123
    push eax
    pop edx
    push edx
    push msg
    call _printf
    add esp,8
    ret
section .data
msg:  db "Result=%d",0
```

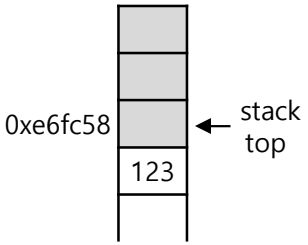
eax	123
edx	0
esp	0xe6fc58

push eax



eax	123
edx	0
esp	0xe6fc54

pop edx



eax	123
edx	123
esp	0xe6fc58

x86 어셈블리언어에서 printf 사용 예시 #1

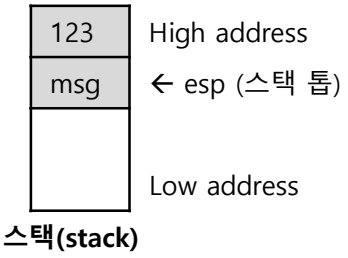
pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    push 123
    push msg
    call _printf
    add esp,8
    ret

section .data
msg: db "Result=%d",0
```

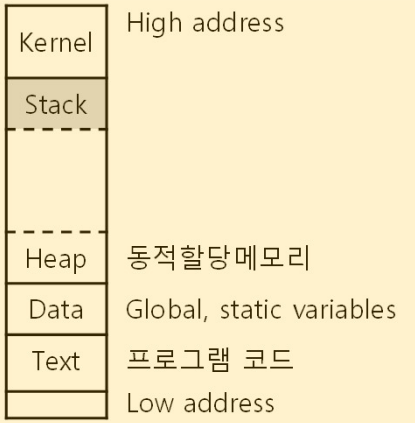
언더바(_)는 함수 호출 규약이다.

cdecl 함수호출규약 (일부)

- 함수에 전달할 argument들을 right to left 순으로 스택에 삽입(마지막 인자를 먼저 push, 첫 인자를 마지막으로 push)



스택(stack)



Program virtual address space

- 레이블(label) → 명령어나 데이터의 메모리 주소에 해당하며, 레이블명 뒤에 콜론(:)을 붙여 표시
- msg: db ... → **d**efine **b**yte
- global _start → 심볼 _start를 pgm.asm 외부 모듈에서 참조할 수 있도록 선언
- extern _printf → _printf가 pgm.asm 외부 모듈에서 정의된 심볼임을 선언
- esp → 스택 톱에 해당하는 메모리 주소를 저장하고 있는 레지스터
- push 123 → esp를 4 감소 후 esp가 가리키는 메모리의 4바이트 공간에 123을 저장
- push msg → esp를 4 감소 후 esp가 가리키는 메모리의 4바이트 공간에 msg(주소)를 저장
- call _printf → eip에 저장된 다음 명령어 주소를 스택에 push한 후 _printf 주소로 jump
- add esp, 8 → esp의 값을 8 증가(esp ← esp+8)시켜, esp 값을 함수 호출 전으로 복원
- 우측 코드의 printf(msg,123);에 대응하는 동작 수행

다음함수 주소를 스택에 push함 (즉 add 명령어를 stack에 메모)

```
#include <stdio.h>
char msg[]="Result=%d";
int main(){
    printf(msg,123);
    return 0;
}
```

실행방법

- **-f win32** → nasm의 출력파일포맷을 Microsoft extended COFF for Win32 (i386)으로 설정

```
nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=123
```

- C:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib₩libmsvcrt.a

x86 어셈블리언어 예시 #2 (덧셈)

실행방법

nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=33

```c
#include <stdio.h>
char msg[]="Result=%d";
int num1=11;
int num2=22;
int main(){
    num1=num1+num2;
    printf(msg,num1);
    return 0;
}
```

pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    mov eax, [num1]
    add eax, [num2]
    push eax
    push msg
    call _printf
    add esp,8
    ret

section .data
msg: db "Result=%d",0
num1: dd 11
num2: dd 22
```

이것은 메모리 주소값을 의미한다.

num1: dd 11
- num1 주소 위치의 4바이트 내용을 11로 초기화 (dd: define double)

mov eax, [num1]
- num1 주소에 있는 4바이트를 eax 레지스터로 복사
- eax ← [num1]

add eax, [num2]
- num2 주소에 있는 4바이트 내용과 eax 내용을 합산한 결과를 eax에 저장
- eax ← eax + [num2]

define directives
- db → define byte (1바이트 내용 정의)
- dw → define word (2바이트 내용 정의)
- dd → define double (4바이트 내용 정의)

db, dw, dd 중요하다!!

x86 어셈블리언어 예시 #2A (뺄셈, 곱셈)

실행방법
nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=54

```c
#include <stdio.h>
char msg[]="Result=%d";
int x=3;
int y=5;
int z=2;
int w=9;
int main(){
    int t;
    t=x;
    t=t+y;
    t=t-z;
    t=t*w;
    printf(msg,t);
    return 0;
}
```

pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    mov eax,[x]
    add eax,[y]
    sub eax,[z]
    imul eax,[w]
    push eax
    push msg
    call _printf
    add esp,8
    ret
section .data
msg:  db "Result=%d",0
x:    dd 3
y:    dd 5
z:    dd 2
w:    dd 9
```

명령어들을 어셈으로 사용한다.

mov eax, [x]
- x에 대응하는 메모리 주소에 있는 4바이트를 eax 레지스터로 복사

add eax, [y]
- y에 대응하는 메모리 주소에 있는 4바이트 값과 eax 값을 덧셈한 결과를 eax에 저장

sub eax, [z]
- z에 대응하는 메모리 주소에 있는 4바이트 값을 eax 값에서 감산한 결과를 eax에 저장

imul eax, [w]
- w에 대응하는 메모리 주소에 있는 4바이트 값과 eax 값을 곱셈한 결과를 eax에 저장

x86 어셈블리언어 조건문 예시 #2B

실행방법
nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=98

if 문을 어셈블리 언어로 구현

pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    mov eax,[x]
    cmp eax,100
    jl Label_LT
    add eax,1
    jmp Label_Next
Label_LT:
    sub eax,1
Label_Next:
    push eax
    push msg
    call _printf
    add esp,8
    ret
section .data
msg:  db "Result=%d",0
x:    dd 99
```

```
#include <stdio.h>
char msg[]="Result=%d";
int x=99;
int main(){
    int t;
    t=x;
    if(t>=100) t=t+1;
    else t=t-1;
    printf(msg,t);
    return 0;
}
```

cmp eax, 100
- eax 값에서 100을 감산하는 연산 수행 및 그 결과에 따라 EFLAGS의 플래그 비트(들) 설정

jl Label_LT
- eax의 값이 100 미만인 경우 Label_LT에 해당하는 주소로 jump
- 직전 수행한 cmp eax,100의 결과로 설정된 EFLAGS 비트들로부터 eax 값이 100 미만인지 아닌지 알 수 있음

jmp Label_Next
- Label_Next에 해당하는 주소로 jump

x86 어셈블리언어 반복문 예시 #2C

실행방법
nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=55

```c
#include <stdio.h>
char msg[]="Result=%d";
int main(){
    int v=0;
    int n=1;
    while(n<=10){
        v=v+n;
        n=n+1;
    }
    printf(msg,v);
    return 0;
}
```

반복문 구현

pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    mov eax,0
    mov ecx,1
Label_Cond:
    cmp ecx,10
    jg Label_Next
    add eax,ecx
    add ecx,1
    jmp Label_Cond
Label_Next:
    push eax
    push msg
    call _printf
    add esp,8
    ret
section .data
msg:  db "Result=%d",0
```

ECX는 반복문의 인덱스처럼 사용됨

cmp ecx, 10
- ecx 값에서 10을 감산하는 연산 수행 및 그 결과에 따라 EFLAGS의 플래그 비트(들) 설정

jg Label_Next
- ecx의 값이 10 초과인 경우 Label_Next에 해당하는 주소로 jump
- 직전 수행한 cmp ecx,10의 결과로 설정된 EFLAGS 비트들로부터 ecx 값이 10 초과인지 아닌지 알 수 있음

jmp Label_Cond
- Label_Cond에 해당하는 주소로 jump

x86 어셈블리언어 예시 #3

실행방법

nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=222

pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    mov eax, [num1]
    sub eax, [num2]
    mov [num1], eax
    push dword [num1]
    push msg
    call _printf
    add esp,8
    ret

section .data
msg: db "Result=%d",0
num1: dd 333
num2: dd 111
```

```
#include <stdio.h>
char msg[]="Result=%d";
int num1=333;
int num2=111;
int main(){
    num1=num1-num2;
    printf(msg,num1);
    return 0;
}
```

- push eax와 동일한 동작 수행
- dword 사용 예시 위해 작성

DWORD를 쓴 이유 num1은 주소값만 가르키게 DWORD를 정해줘서 4바이트로 명시 해야함

mov [num1], eax
- eax 레지스터의 내용(4바이트)을 num1 주소에 복사
- [num1] ← eax

sub eax, [num1]
- eax 내용에서 num1 주소에 있는 4바이트 내용을 감한 결과를 eax에 저장
- eax ← eax - [num1]

push dword [num1]
- num1 주소에 있는 4바이트를 스택에 push
- 4바이트를 명시하기 위해 **dword** 사용 (**d**ouble **word**)
- push [num1]라고만 적으면 오류 출력(operation size not specified)
- mov [num1], eax 혹은 mov eax, [num1]의 경우 eax가 4바이트이므로 [num1]이 4바이트 공간임을 알 수 있으므로 dword 불필요
- 2바이트를 명시하는 경우라면 push **word** [num1]
- 1바이트를 명시하는 경우라면 push **byte** [num1]

x86 어셈블리언어 함수 호출 예시 #4 (1/2)

pgm.asm

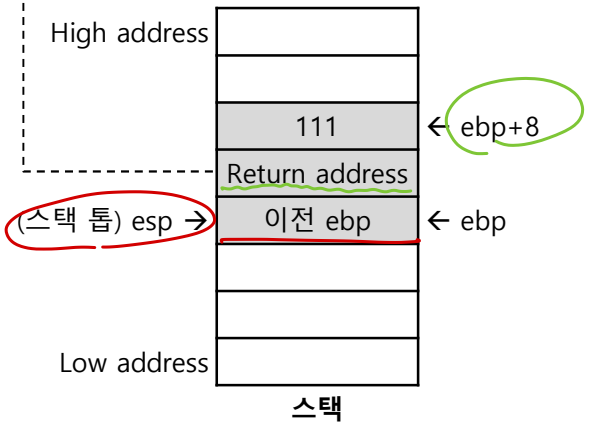
```
section .text
    global _start
    extern _printf
_start:
    push dword [num1]
    call ft
    add esp,4
    push eax
    push msg
    call _printf
    add esp,8
    ret

ft:
    push ebp
    mov ebp,esp
    mov eax,[ebp+8]
    add eax,eax
    mov esp,ebp
    pop ebp
    ret

section .data
msg: db "Result=%d",0
num1: dd 111
```

실행방법

```
nasm -f win64 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=222
```

cdecl 함수호출규약(calling convention)

- 호출측(caller) → 함수에 전달할 argument(들)을 right to left 순으로 스택에 삽입
- 호출측(caller) → call ft로 함수 ft 호출. EIP(반환주소)를 스택에 push하고 ft로 jump)
- 피호출측(callee) → 이전 ebp 보관(스택에 push) 및 ebp 갱신
- 피호출측(callee) → 함수 실행 동안 사용될 레지스터 값 있으면 (스택에 push하여) 보관
- 피호출측(callee) → (esp를 감소시켜) 스택에 지역변수(stack variable) 공간 할당
- 피호출측(callee) → **함수 코드 수행(반환 값 있으면 eax에 저장)**
- 피호출측(callee) → (esp를 증가시켜) 스택에 할당되었던 지역변수 공간 반환
- 피호출측(callee) → 함수 실행 동안 변경된 레지스터 원래 값 (스택에서 pop하여) 복원
- 피호출측(callee) → ebp를 이전 값(함수 호출 전 ebp 값)으로 복원
- 피호출측(callee) → (ret 명령을 통해) 반환주소를 스택에서 EIP로 pop하여 호출측으로 리턴
- 호출측(caller) → 스택에 push했던 argument(들)을 cleanup




```
#include <stdio.h>
char msg[]="Result=%d";
int num1=111;
int ft(int x){
    x=x+x;
    return x;
}
int main(){
    printf(msg,ft(num1));
    return 0;
}
```

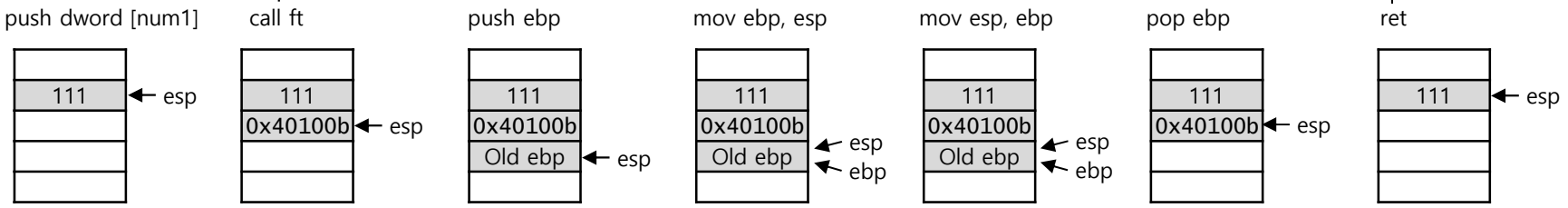
x86 어셈블리언어 함수 호출 예시 #4 (2/2)

```
section .text
    global _start
    extern _printf
_start:
    push dword [num1]
    call ft
    add esp,4
    push eax
    push msg
    call _printf
    add esp,8
    ret

ft:
    push ebp
    mov ebp,esp
    mov eax,[ebp+8]
    add eax,eax
    mov esp,ebp
    pop ebp
    ret

section .data
msg: db "Result=%d",0
num1: dd 111
```

```
0x401000: push   DWORD PTR ds:0x40200a
0x401006: call   0x40101d <ft>
0x40100b: add    esp,0x4
0x40100e: push   eax
0x40100f: push   0x402000
0x401014: call   0x40102c <printf>
0x401019: add    esp,0x8
0x40101c: ret
0x40101d: push   ebp
0x40101e: mov    ebp,esp
0x401020: mov    eax,DWORD PTR [ebp+0x8]
0x401023: add    eax,eax
0x401025: mov    esp,ebp
0x401027: pop    ebp
0x401028: ret
0x401029: xchg   ax,ax
0x40102b: nop
…
0x40200a: 0x0000006f
```

- EIP를 스택에 push하고, ft 위치로 jump
- push(EIP); EIP := ft;

- 스택에서 pop한 주소로 jump
- EIP := pop();



x86 어셈블리언어 함수 호출 예시 #5

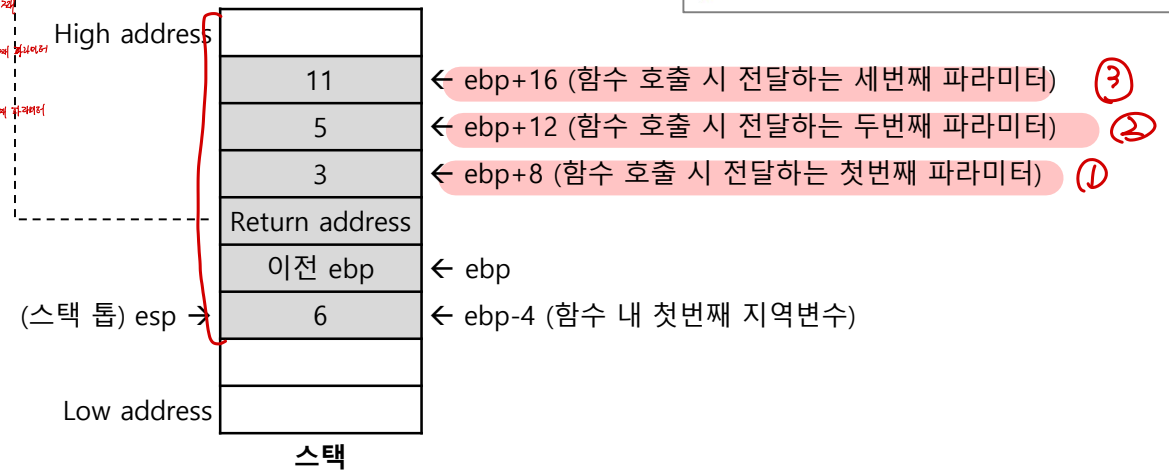
pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    push dword [num3]
    push dword [num2]
    push dword [num1]
    call ft
    add esp,12
    push eax
    push msg
    call _printf
    add esp,8
    ret
ft:
    push ebp
    mov ebp,esp
    sub esp,4
    mov eax,[ebp+16]
    sub eax,[ebp+12]
    mov [ebp-4],eax
    mov eax,[ebp+8]
    add eax,[ebp-4]
    mov esp,ebp
    pop ebp
    ret

section .data
msg: db "Result=%d",0
num1: dd 3
num2: dd 5
num3: dd 11
```

실행방법

nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=9

```
#include <stdio.h>
int num1=3;
int num2=5;
int num3=11;
char msg[]="Result=%d\n";
int ft(int x, int y, int z){
    int t;
    t=z-y;
    return x+t;
}
int main(){
    printf(msg,ft(num1,num2,num3));
    return 0;
}
```



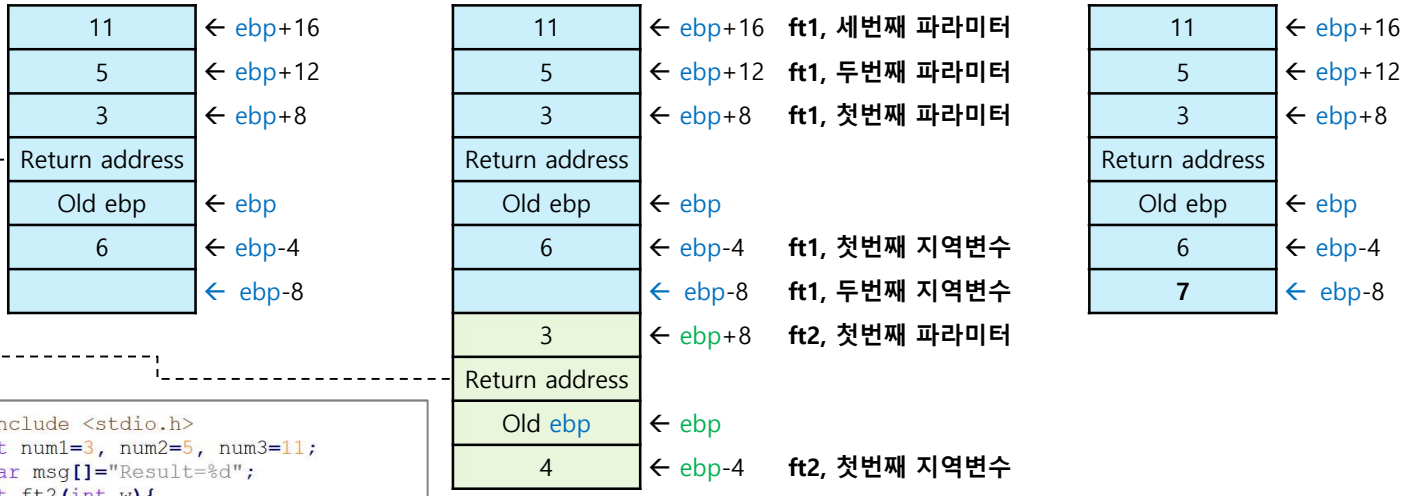
x86 어셈블리언어 함수 호출 예시 #6

pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    push dword [num3]
    push dword [num2]
    push dword [num1]
    call ft1
    add esp,12
    push eax
    push msg
    call _printf
    add esp,8
    ret
ft1:
    push ebp
    mov ebp,esp
    sub esp,8
    mov eax,[ebp+16]
    sub eax,[ebp+12]
    mov [ebp-4],eax
    push dword [ebp+8]
    call ft2
    add esp,4
    mov [ebp-8],eax
    mov eax,[ebp-4]
    sub eax,[ebp-8]
    mov esp,ebp
    pop ebp
    ret
ft2:
    push ebp
    mov ebp,esp
    sub esp,4
    mov eax,[ebp+8]
    sub eax,1
    add eax,eax
    mov [ebp-4],eax
    mov eax,[ebp+8]
    add eax,[ebp-4]
    mov esp,ebp
    pop ebp
    ret
section .data
msg: db "Result=%d",0
num1: dd 3
num2: dd 5
num3: dd 11
```

실행방법

```
nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=-1
```




```c
#include <stdio.h>
int num1=3, num2=5, num3=11;
char msg[]="Result=%d";
int ft2(int w){
    int t3;
    t3=w-1;
    t3=t3+t3;
    return w+t3;
}
int ft1(int x, int y, int z){
    int t1, t2;
    t1=z-y;
    t2=ft2(x);
    return t1-t2;
}
int main(){
    printf(msg,ft1(num1,num2,num3));
    return 0;
}
```

함수 안에서 다른 함수를 호출

cdecl 함수호출규약(calling convention)

- 호출측(caller) → 함수에 전달할 argument(들)을 right to left 순으로 스택에 삽입
- 호출측(caller) → call ft로 함수 ft 호출. EIP(반환주소)를 스택에 push하고 ft로 jump)
- 피호출측(callee) → 이전 ebp 보관(스택에 push) 및 ebp 갱신
- 피호출측(callee) → 함수 실행 동안 사용될 레지스터 값 있으면 (스택에 push하여) 보관
- 피호출측(callee) → (esp를 감소시켜) 스택에 지역변수(stack variable) 공간 할당
- 피호출측(callee) → **함수 코드 수행(반환 값 있으면 eax에 저장)**
- 피호출측(callee) → (esp를 증가시켜) 스택에 할당되었던 지역변수 공간 반환
- 피호출측(callee) → 함수 실행 동안 변경된 레지스터 원래 값 (스택에서 pop하여) 복원
- 피호출측(callee) → ebp를 이전 값(함수 호출 전 ebp 값)으로 복원
- 피호출측(callee) → (ret 명령을 통해) 반환주소를 스택에서 EIP로 pop하여 호출측으로 리턴
- 호출측(caller) → 스택에 push했던 argument(들)을 cleanup

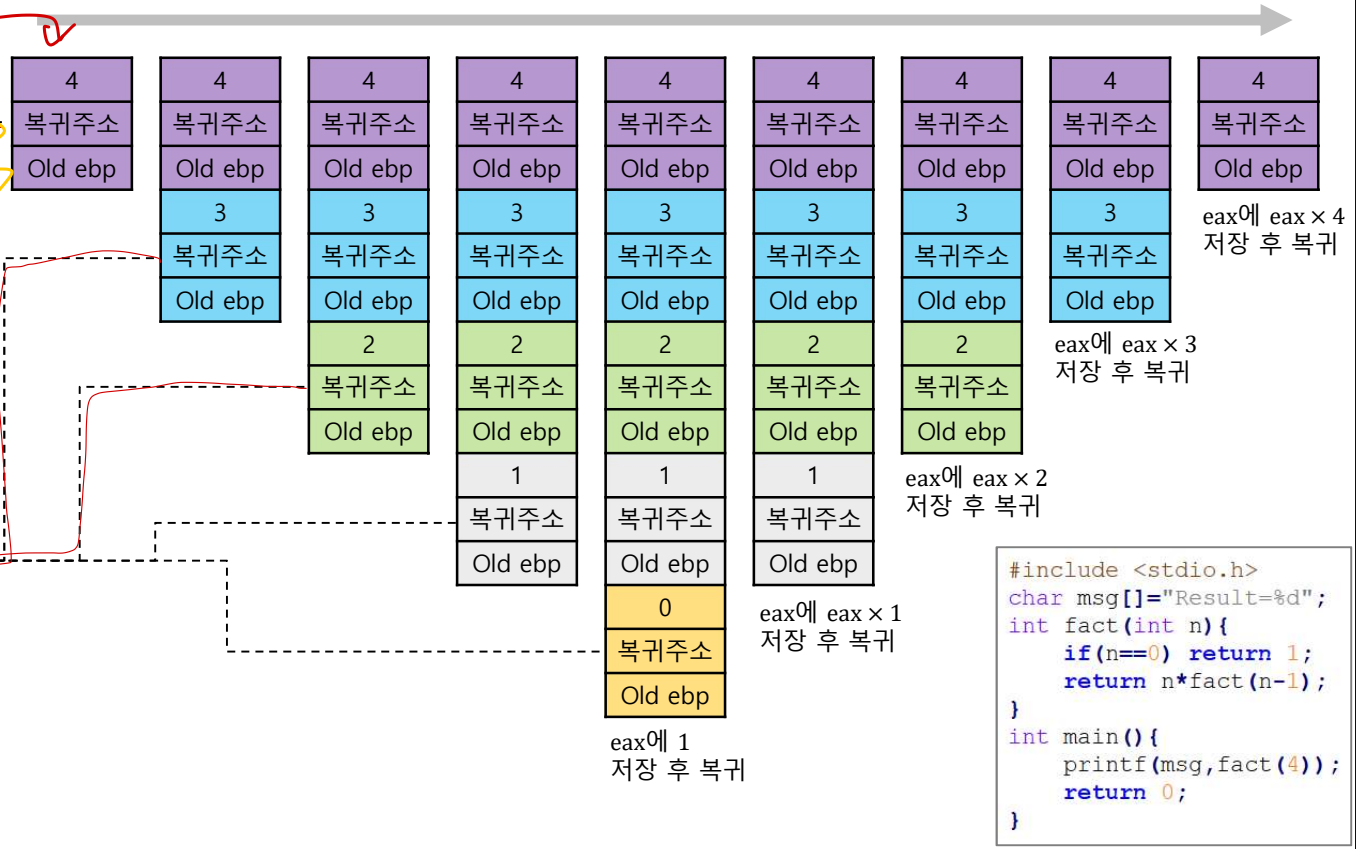
x86 어셈블리언어 재귀 함수 호출 예시 #7

pgm.asm

```
section .text
    global _start
    extern _printf
_start:
    push dword [num1]
    call fact
    add esp,4
    push eax
    push msg
    call _printf
    add esp,8
    ret
fact:
    push ebp
    mov ebp,esp
    cmp dword [ebp+8],0
    jne L1
    mov eax,1
    jmp L2
L1:
    mov eax,[ebp+8]
    sub eax,1
    push eax
    call fact
    add esp,4
    imul eax,[ebp+8]
L2:
    mov esp,ebp
    pop ebp
    ret
section .data
msg:  db "Result=%d",0
num1: dd 4
```

실행방법

nasm -f win32 pgm.asm -o pgm.o
ld -o pgm.exe pgm.o -LC:₩MinGW-w64₩mingw32₩i686-w64-mingw32₩lib -lmsvcrt
pgm.exe
Result=24




```
#include <stdio.h>
char msg[]="Result=%d";
int fact(int n){
    if(n==0) return 1;
    return n*fact(n-1);
}
int main(){
    printf(msg,fact(4));
    return 0;
}
```

References

- 박두순. (2016). (내공 있는 프로그래머로 길러주는)컴파일러의 이해. 한빛아카데미.
- 창병모. (2021). 프로그래밍 언어론 : 원리와 실제. 인피니티북스.
- Aho, A., Lam, M., Sethi, R., Ullman, J. (2006). Compilers: Principles, Techniques, and Tools (2nd Ed.). Addison Wesley.
- Nystrom, R. Crafting interpreter. https://craftinginterpreters.com/
- Sebesta, R. (2012). Concepts of Programming Languages (10th. ed.). Pearson.
- Thain, D. (2023). Introduction to Compilers and Language Design.
- Paxson, V. (1995). Flex, version 2.5.
- Donnelly, C., Stallman, R. (2008). Bison.
- LexAndYacc.pdf (epaperpress.com)
- Tom Niemann, LEX & YACC. http://epaperpress.com/lexandyacc