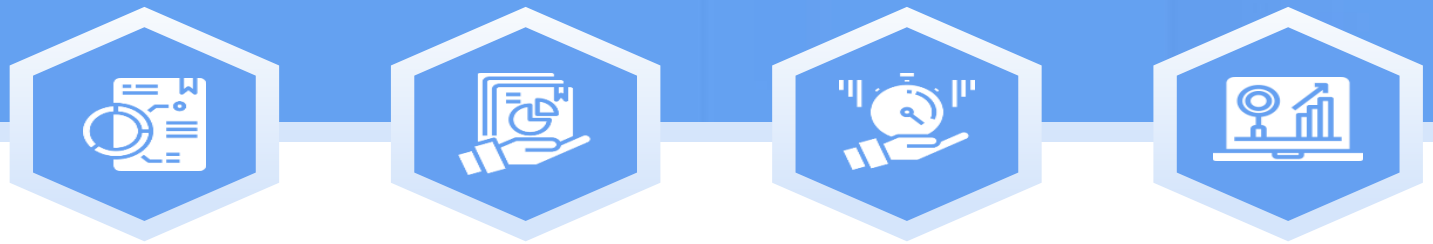
시스템 보안

어셈블리어 실습 - 1

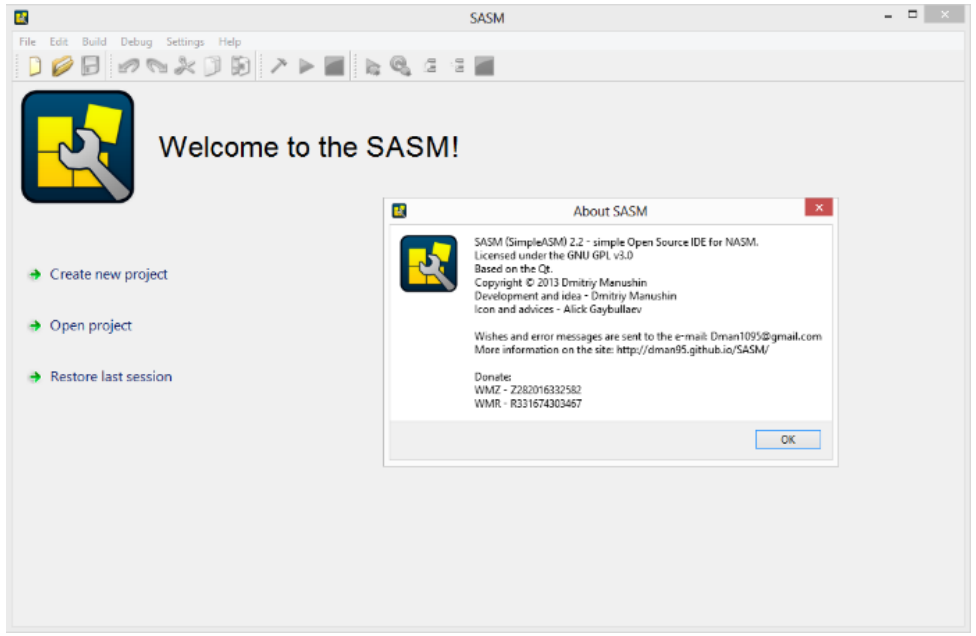


KOREA UNIVERSITY
SEJONG CAMPUS

어셈블리어 실습

- **실습 – 준비**

  - SASM(SimpleASM) 설치 : https://dman95.github.io/SASM/english.html



Download .exe for Windows

Download .zip for Windows

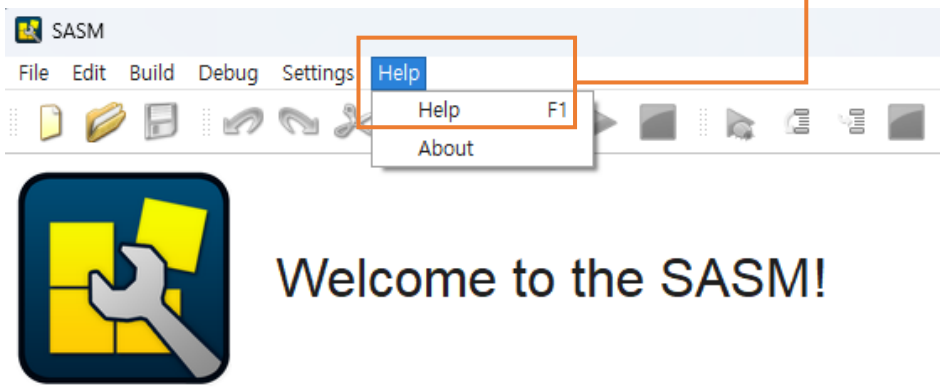
Download for Linux

Installing on Mac: link 1, link 2

어셈블리어 실습

- **실습 – 준비**

  - SASM Builtin functions



"io.inc" macro library for NASM

SASM includes crossplatform input/output library "io.inc". To use it you need to add directive %include "io.inc" (%include "io64.inc" for x64) to the beginning of your program.

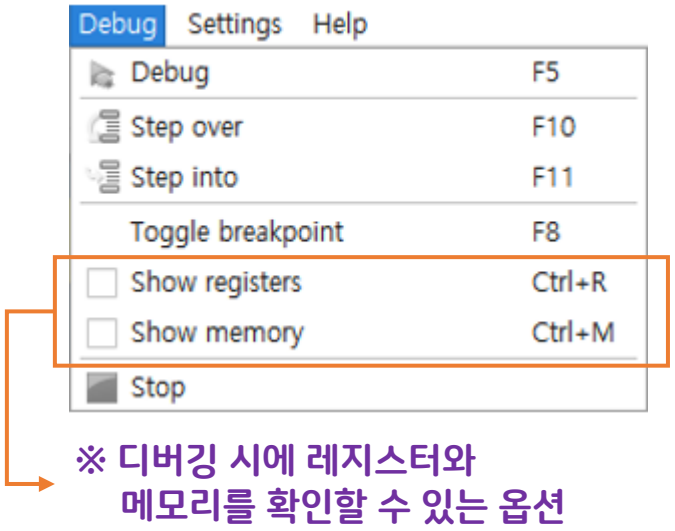
Macro name	Description
PRINT_UDEC *size, data* PRINT_DEC *size, data*	Print number *data* in decimal representation. *size* – number, giving size of *data* in bytes – 1, 2, 4 or 8 (x64). *data* must be number or symbol constant, name of variable, register or address expression without size qualifier (byte[], etc.). PRINT_UDEC print number as unsigned, PRINT_DEC — as signed.
PRINT_HEX *size, data*	Similarly previous, but data is printed in hexadecimal representation.
PRINT_CHAR *ch*	Print symbol *ch*. *ch* – number or symbol constant, name of variable, register or address expression without size qualifier (byte[], etc.).
PRINT_STRING *data*	Print null-terminated text string. *data* – string constant, name of variable or address expression without size qualifier (byte[], etc.).
NEWLINE	Print newline ('₩n').
GET_UDEC *size, data* GET_DEC *size, data*	Input number data in decimal representation from stdin. *size* – number, giving size of *data* in bytes – 1, 2, 4 or 8 (x64). *data* must be name of variable or register or address expression without size qualifier (byte[], etc.). GET_UDEC input number as unsigned, GET_DEC — as signed. It is not allowed to use esp register.
GET_HEX *size, data*	Similarly previous, but data is entered in hexadecimal representation with 0x prefix.
GET_CHAR *data*	Similarly previous, but macro reads one symbol only.
GET_STRING *data, maxsz*	Input string with length less than *maxsz*. Reading stop on EOF or newline and "₩n" writes in buffer. In the end of string 0 character is added to the end. *data* – name of variable or address expression without size qualifier (byte[], etc.). *maxsz* – register or number constant.

General purpose registers are not modified during execution of the above macros.

어셈블리어 실습

- **실습 – 준비**

  - SASM Debugging



※ 디버깅 시에 레지스터와
메모리를 확인할 수 있는 옵션

```
10  section .text
11  global CMAIN
12  CMAIN:
13      mov ebp, esp; for correct debugging
14      push ebp
15      mov ebp, esp
16
```

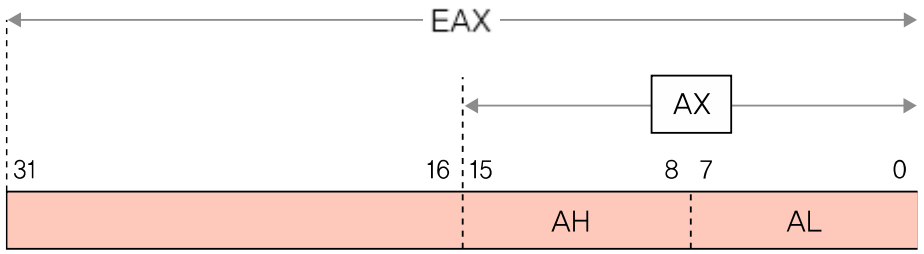
Registers

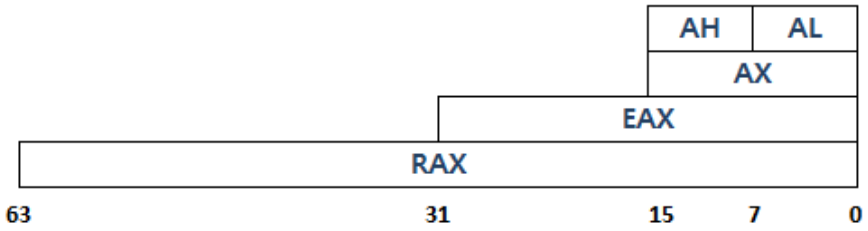
Register	Hex	Info
eax	0x1	1
ecx	0x401c60	4201568
edx	0x80	128
ebx	0x385000	3690496
esp	0x61ff0c	0x61ff0c
ebp	0x61ff54	0x61ff54

어셈블리어 실습

- **비트 수에 따른 레지스터 따른 명칭**

- EAX, EBX, ECX, EDX는 32비트 레지스터로 앞의 E는 '확장된(Extended)'을 의미
  64비트 레지스터는 RAX, RBX, RCX, RDX 등으로 앞의 R은 'Register'를 의미하는 것으로 알려져 있음
- EAX, EBX, ECX, EDX 레지스터는 오른쪽 16비트를 각각 AX, BX, CX, DX 16비트 레지스터로 사용 가능하고,
  16비트의 왼쪽 8비트 상위(high) 부분과 오른쪽 8비트 하위(low) 부분을 8비트 레지스터로 사용 가능
  AH/AL, BH/BL, CH/CL, DH/DL





어셈블리어 실습

- **함수(Function)? 프로시저(Procedure)?**

- C언어와 같은 프로그래밍 언어에서의 함수(Function)를 어셈블리언어에서는 프로시저(Procedure)라고 부름

C 언어에서의 함수

```c
int calc()
{
    int sum = 1;
    sum += 1;

    return sum;
}
```

NASM 에서의 프로시저

```
calc:
    mov eax, 1
    inc eax
    ret
```

MASM 에서의 프로시저

```
calc PROC
    mov eax, 1
    inc eax
    ret
calc ENDP
```

※ 어셈블리(Assembly)는 언어, 어셈블러(Assembler)는 컴파일러를 의미

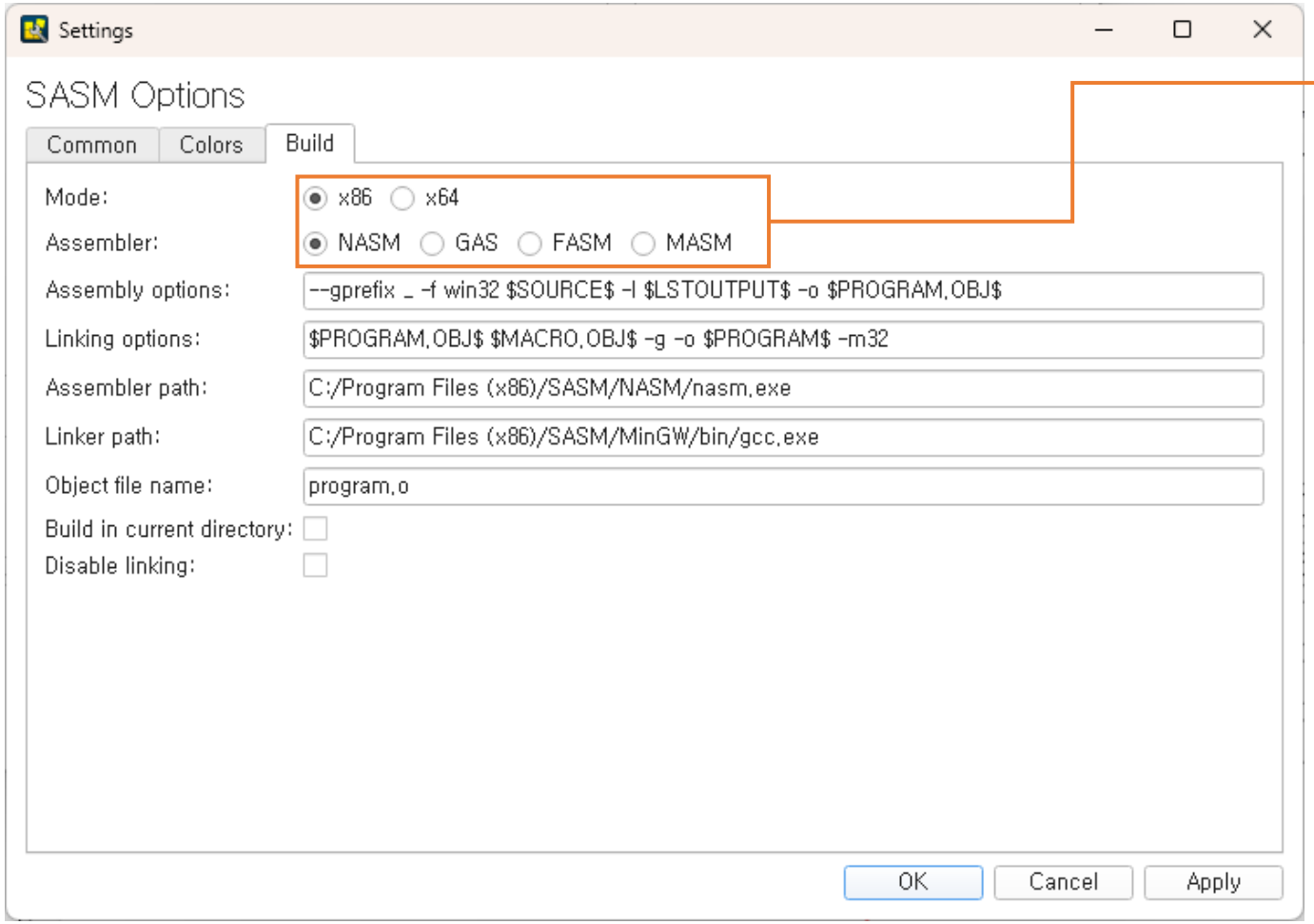
어셈블러에는 여러가지 종류가 있으며 각 어셈블러마다 지원 플랫폼과 문법이 다름

GAS(GNU Assembler), MASM(Microsoft Macro Assembler), NASM(Netwide Assembler) 등

어셈블리어 실습

- **실습 – 준비(32bit)**

  - SASM Options



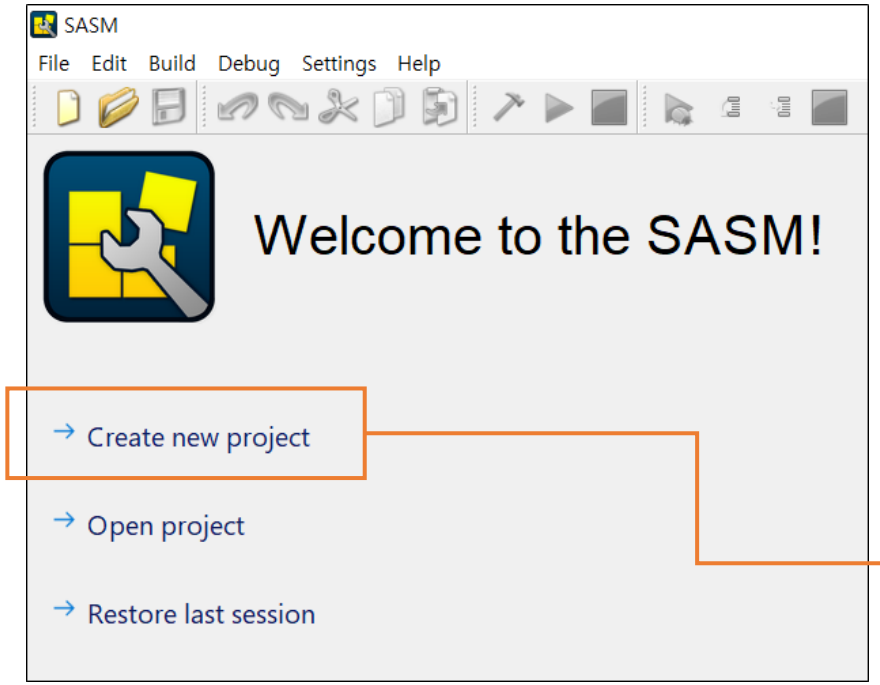
※ NASM(Netwide Assembler)

Mode는 x86(32bit)로 설정

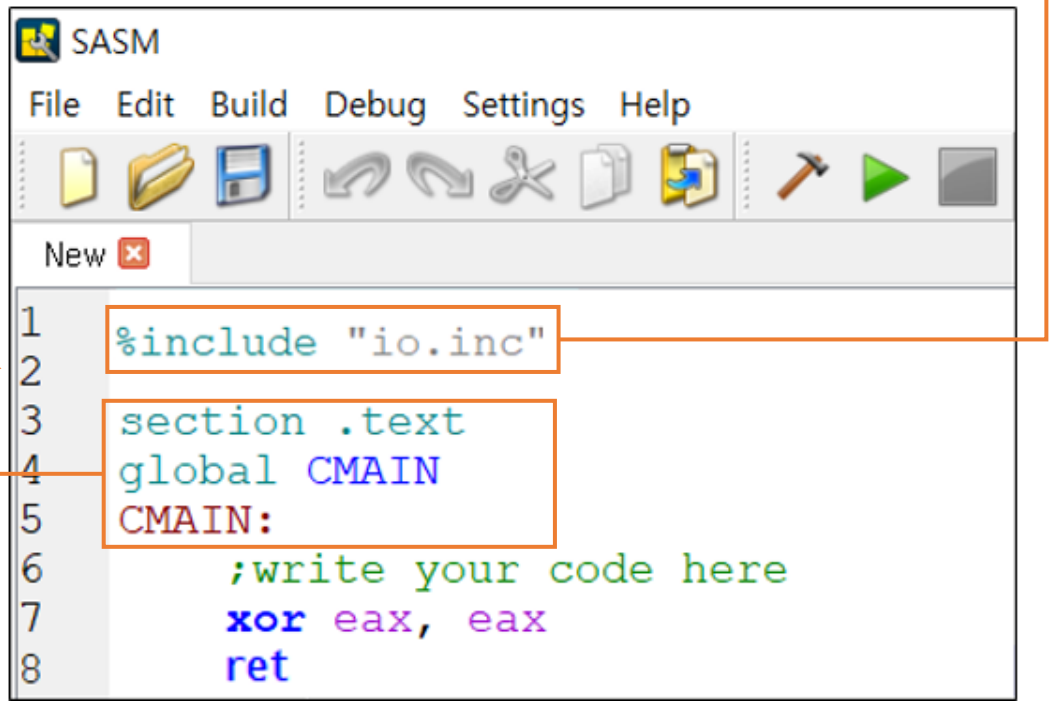
어셈블리어 실습

- **실습 – 준비(32bit)**

  - SASM Builtin functions & macro



※ "io.inc" macro library for NASM



※ .text 섹션에 C에서의 main() 함수인 CMAIN 선언

어셈블리어 실습

- **실습 – 상수 및 변수 정의 (1/2)**

  - SASM에서 아래 코드를 입력하여 결과를 확인, const_variable_define.asm

```
%include "io.inc"

; equ 숫자 상수 선언, 코드 내에서 변경 불가능
const_1 equ 0x1000

; %define은 c언어의 define과 같은 의미
%define PTR [ebx + 1]

section .bss              ; 데이터(초기화 되지 않은 데이터) 영역임을 선언
    bss1 resb 1           ; byte 변수 선언
    bss2 resw 1           ; word 변수 선언
    bss3 resd 1           ; dword 변수 선언

section .data
    var1 db 0x12          ; byte 변수 선언
    var2 dw 0x1234        ; word 변수 선언
    var3 dd 0x12345678    ; dword 변수 선언

section .text
global CMAIN
CMAIN:
1   PRINT_HEX 1, [var1]   ; 변수 크기에 따른 출력 방법
    NEWLINE
2   PRINT_HEX 2, [var2]
    NEWLINE
3   PRINT_HEX 4, [var3]
    NEWLINE
    NEWLINE
```

1. 12
2. 1234
3. 12345678

어셈블리어 실습

- **실습 – 상수 및 변수 정의 (2/2)**

```
1  PRINT_HEX 4, [bss3]   ; .bss에 선언된 변수들은 기본적으로 0으로 초기화 됨
   NEWLINE
   NEWLINE

   mov al, [var1]        ; .data에 선언된 변수들의 값으로 .bss 변수들의 값 설정
   mov [bss1], al
   mov ax, [var2]
   mov [bss2], ax
   mov eax, [var3]
   mov [bss3], eax

2  PRINT_HEX 1, [bss1]   ; .bss 변수들의 값 출력
   NEWLINE
3  PRINT_HEX 2, [bss2]
   NEWLINE
4  PRINT_HEX 4, [bss3]
   NEWLINE
   NEWLINE

   mov bx, const_1       ; .bss에 선언된 변수 값을 const_1 상수 값으로 설정
   mov [bss2], bx
5  PRINT_HEX 2, [bss2]
   NEWLINE
   NEWLINE

   mov ebx, bss3         ; ebx 레지스터에 bss3 변수의 주소를 설정
6  PRINT_HEX 2, PTR      ; define으로 정의한 PTR("[ebx + 1]")을 이용해서 출력
   NEWLINE

   xor eax, eax
   ret
```

1. 0
2. 12
3. 1234
4. 12345678
5. 1000
6. 3456

어셈블리어 실습

- **실습 – 메모리 저장 순서 확인**

  - SASM에서 아래 코드를 입력하여 결과를 확인, memory_littel_endian.asm

```
%include "io.inc"

section .data               ; 데이터(초기화 된 데이터) 영역임을 선언
    var1 dw 0x1234
    var2 dd 0x55555555

section .text               ; 코드 영역임을 선언
global CMAIN
CMAIN:
1   PRINT_HEX 1, [var1]     ; 1바이트 출력
    NEWLINE
2   PRINT_HEX 2, [var1]     ; 2바이트 출력
    NEWLINE
3   PRINT_HEX 4, [var1]     ; 4바이트 출력 (변수 경계를 넘어서 출력됨)
    NEWLINE
    mov eax, 0x56789abc
    mov [var2], eax         ; eax에 저장된 값으로 var2 변수 값 변경
4   PRINT_HEX 4, [var2]     ; var2의 값 4바이트 출력
    NEWLINE
5   PRINT_HEX 4, [var1]     ; var1의 값 4바이트 출력
    NEWLINE

    xor eax, eax            ; 종료 코드 설정 (0은 정상 종료)
    ret
```

- 실제 메모리에는 var1, var2 순서로 데이터가 저장되는데, 리틀 엔디안 방식으로 저장되기 때문에 아래와 같이 저장
- 초기 상태 : 34 12 55 55 55 55
- var2 변수의 값을 변경한 이후 상태 : 34 12 bc 9a 78 56

1. 34
2. 1234
3. 55551234
4. 56789abc
5. 9abc1234

어셈블리어 실습

- **실습 – 문자열 처리**

  - SASM에서 아래 코드를 입력하여 결과를 확인, hello_world.asm

```
%include "io.inc"

section .data
    msg1 db 'Hello World!', 0xd, 0x00  ; char* msg = 'Hello World!\n'
    ; 문자열 길이를 자동 계산 하는 매크로 'equ $ -'는
    ; 계산하고자 하는 문자열 변수 정의 바로 다음 라인에 코드를 작성해야 함
    ; $ 문자가 시작하는 주소에서 문자열이 시작하는 주소를 뺀 값으로 계산하기 때문
    msg1_len equ $ - msg1
    msg2 db 'Hello', 0x0d, 'World!', 0x0d, 0x00

section .text
global CMAIN
CMAIN:
    PRINT_STRING msg1     ; msg1 변수의 문자열 출력
    mov ax, msg1_len
    PRINT_DEC 2, ax       ; msg1 변수의 문자열 길이 출력
    NEWLINE
    PRINT_STRING msg2     ; msg2 변수의 문자열 출력 (문자열 중간에 '\n' 문자 포함)

    xor eax, eax
    ret
```

- 문자열 길이를 계산해 주는 매크로 이용

  '₩n'과 NULL 문자까지 길이에 포함되기 때문에 문자열 길이 라기 보다는 문자열을 저장하고 있는 배열의 크기

```
hello world!
14
hello
world!
```

어셈블리어 실습

- **실습 – 배열과 반복 루프 (jmp 명령어 이용)**

  - SASM에서 아래 코드를 입력하여 결과를 확인, array_for_loop_jmp.asm

```
%include "io.inc"

section .data
    array db 1,2,3,4,5    ; byte array[5] = {1, 2, 3, 4, 5}

section .text
global CMAIN
CMAIN:
    ;초기화 부분
    mov eax, 0            ; 배열에서 데이터를 읽어올 때 사용할 레지스터 초기화
    mov edx, 0            ; 읽어온 데이터를 더해주기 위한 레지스터 초기화
    mov ecx, 5            ; 배열의 크기 설정.
    mov esi, array        ; array 배열의 주소 설정

for:
    mov al, [esi]         ; 배열에서 첫 번째 데이터를 읽음 (데이터 크기가 바이트기 때문에 al에 저장)
    add edx, eax          ; 읽어온 값을 더해줌
    add esi, 1            ; 다음 데이터를 읽기 위해서 주소 증가
    dec ecx               ; 반복 횟수 감소
    jnz for               ; 반복할 구간으로 점프 (ecx 값이 0이면 점프하지 않음)
    PRINT_DEC 2, edx      ; 합산된 최종 값 출력

    xor eax, eax          ; 종료 코드 설정
    ret
```

15

어셈블리어 실습

- **실습 – 배열과 반복 루프(loop 명령어 이용)**

  - SASM에서 아래 코드를 입력하여 결과를 확인, array_for_loop_loop.asm

```
%include "io.inc"

section .data
    array dd 1,2,3,4,5   ; int array[5] = {1, 2, 3, 4, 5}

section .text
global CMAIN
CMAIN:
    ;초기화 부분
    mov eax, 0           ; 배열에서 데이터를 읽어올 때 사용할 레지스터 초기화
    mov edx, 0           ; 읽어온 데이터를 더해주기 위한 레지스터 초기화
    mov ecx, 5           ; 배열의 크기 설정
    mov esi, array       ; array 배열의 주소 설정

for:
    mov eax, [esi]       ; 배열에서 첫 번째 데이터를 읽음
    add edx, eax         ; 읽어온 값을 더해줌
    add esi, 4           ; 다음 데이터를 읽기 위해서 주소 증가
    loop for             ; 반복 명령 (ecx에 지정된 값 만큼 반복)
    PRINT_DEC 2, edx     ; 합산된 최종 값 출력

    xor eax, eax         ; 종료 코드 설정
    ret
```

15

어셈블리어 실습

- **실습 – 함수 호출과 스택(프롤로그, 에필로그)**
  - 스택 프레임(Stack Frame)이란 함수가 호출될 때, 그 함수만의 스택 영역을 구분하기 위하여 생성되는 공간
    이 공간에는 함수와 관계되는 지역변수, 파라미터가 저장되며, 함수 호출 시 할당되며, 함수가 종료되면서 소멸
  - 이 영역을 표현하기 위해 함수 프롤로그(Prolog)와 함수 에필로그(Epilog)라는 것을 수행

- x86 함수 프롤로그와 에필로그

```
;
; Function Prologue
;
push  ebp          ; Save the stack-frame base pointer (of the calling function).
mov   ebp, esp     ; Set the stack-frame base pointer to the current location on the stack.
sub   esp, N       ; Grow the stack by N bytes to reserve space for local variables.
```

함수 프롤로그는 함수가 호출될 때 함수 시작 부분에서 실행되는데, 스택과 레지스터를 준비하는 몇 줄의 코드로 구성

```
;
; Function Epilogue
;
mov   esp, ebp     ; Put the stack pointer back where it was when this function was called.
pop   ebp          ; Restore the calling function's stack frame.
ret                ; Return to the calling function.
```

함수 에필로그는 함수가 종료되어 리턴 하려고 할 때 실행되며, 함수가 호출되기 전의 스택 및 레지스터를 복원하기 위한 몇 줄의 코드로 구성

어셈블리어 실습

- **실습 – 함수 호출과 스택 정리 (1/3)**

  - SASM에서 아래 코드를 입력하여 결과를 확인, function_stack.asm

```
%include "io.inc"

section .text
global CMAIN
CMAIN:
    ; 프롤로그
    push ebp
    mov ebp, esp

    mov eax, 3
    mov edx, 2

    ; 에필로그
    mov esp, ebp
    pop ebp

    xor eax, eax
    ret
```

Output

```
5
3
```

Caller

```
push edx
push eax

; add_proc(eax, edx)
; add_proc 프로시저는 __cdecl 호출 방식
call add_proc

; 함수를 호출한 쪽에서 스택을 정리해야 함
add esp, 8

; 결과 출력(호출 결과는 eax 레지스터로 전달)
PRINT_DEC 4, eax
NEWLINE
```

Caller

```
push edx
push eax

; subtract_proc(eax, edx)
; subtract_proc 프로저는 __stdcall 호출 방식
; 호출한 함수 내에서 스택을 정리
call subtract_proc

; 결과 출력(호출 결과는 eax 레지스터로 전달)
PRINT_DEC 4, eax
NEWLINE
```

Callee

```
; add_proc 프로시져 (__cdecl 호출 방식)
add_proc:
    ; 프롤로그
    push ebp
    mov ebp, esp

    ; 첫번째 파라메터 읽기(epb -> ret addr -> a)
    mov eax, [ebp + 8]
    ; 두번째 파라메터 읽기(epb -> ret addr -> a -> b)
    mov edx, [ebp + 8 + 4]
    add eax, edx

    ; 에필로그
    mov esp, ebp
    pop ebp

    ; 프로시저에서 스택 정리 안함
    ret
```

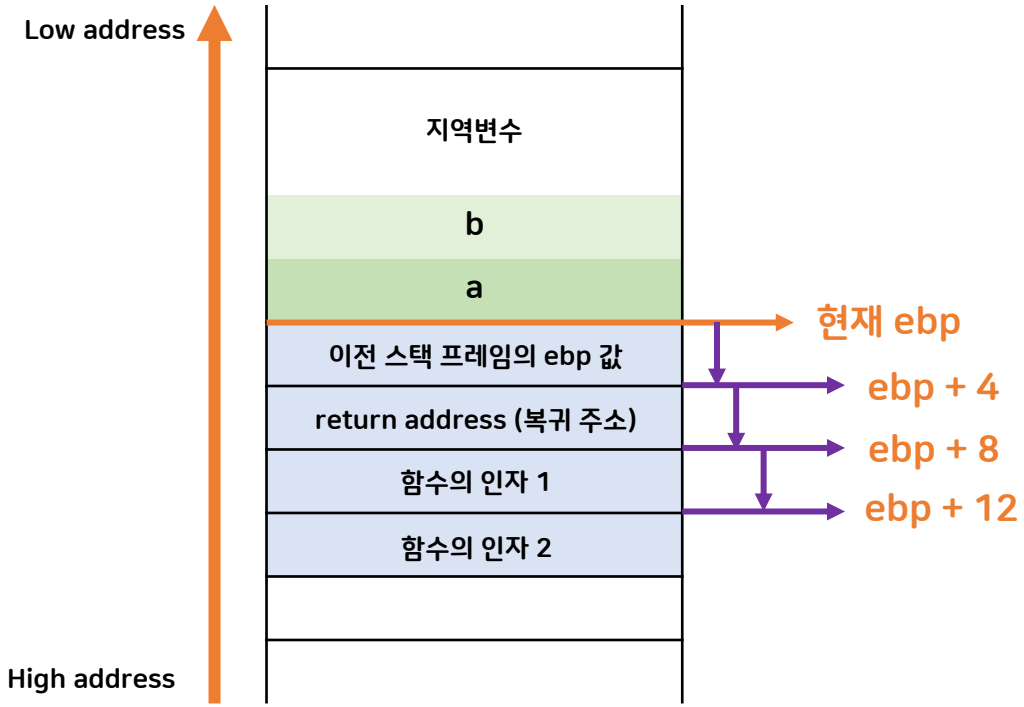
Callee

```
; subtract_proc 프로시져 (__stdcall 호출 방식)
subtract_proc:
    ; 프롤로그
    push ebp
    mov ebp, esp

    ; 첫번째 파라메터 읽기(epb -> ret addr -> a)
    mov eax, [ebp + 8]
    ; 두번째 파라메터 읽기(epb -> ret addr -> a -> b)
    mov edx, [ebp + 8 + 4]
    sub eax, edx

    ; 에필로그
    mov esp, ebp
    pop ebp

    ; 프로시저에서 스택 정리
    ; 8byte 만큼 스택을 정리하고 리턴
    ret 8
```

어셈블리어 실습

- **실습 – 함수 호출과 스택 정리 (2/3)**

※ 프로시저로 전달되는 파라메터 참조 : 32bit에서는 주소가 4byte 크기



일반적인 함수의
스택 프레임 (Stack Frame) 구조 예시

edx(4byte), eax(4byte) 순서로 push

```
push edx
push eax

; add_proc(eax, edx)
call add_proc


add_proc:
mov eax, [ebp + 8]
mov edx, [ebp + 8 + 4]
```

eax(4byte), edx(4byte) 순서로 읽기

- 스택 구조이기 때문에 입력된 순서의 역순으로 참조

어셈블리어 실습

- **실습 – 함수 호출과 스택 정리 (3/3)**

Caller

```
push edx
push eax

; add_proc(eax, edx)
; add_proc 프로시저는 __cdecl 호출 방식
call add_proc

; 함수를 호출한 쪽에서 스택을 정리해야 함
add esp, 8

; 결과 출력(호출 결과는 eax 레지스터로 전달)
PRINT_DEC 4, eax
NEWLINE
```

Callee

```
; add_proc 프로시져 (__cdecl 호출 방식)
add_proc:
    ; 프롤로그
    push ebp
    mov ebp, esp

    ; 첫번째 파라메터 읽기(epb -> ret addr -> a)
    mov eax, [ebp + 8]
    ; 두번째 파라메터 읽기(epb -> ret addr -> a -> b)
    mov edx, [ebp + 8 + 4]
    add eax, edx

    ; 에필로그
    mov esp, ebp
    pop ebp

    ; 프로시저에서 스택 정리 안함
    ret
```

※ __cdecl 방식으로 호출하는 함수
- 호출 하는 쪽(Caller)에서 스택을 정리
- C 함수 : printf()

Caller

```
push edx
push eax

; subtract_proc(eax, edx)
; subtract_proc 프로저는 __stdcall 호출 방식
; 호출한 함수 내에서 스택을 정리
call subtract_proc

; 결과 출력(호출 결과는 eax 레지스터로 전달)
PRINT_DEC 4, eax
NEWLINE
```

Callee

```
; subtract_proc 프로시져 (__stdcall 호출 방식)
subtract_proc:
    ; 프롤로그
    push ebp
    mov ebp, esp

    ; 첫번째 파라메터 읽기(epb -> ret addr -> a)
    mov eax, [ebp + 8]
    ; 두번째 파라메터 읽기(epb -> ret addr -> a -> b)
    mov edx, [ebp + 8 + 4]
    sub eax, edx

    ; 에필로그
    mov esp, ebp
    pop ebp

    ; 프로시저에서 스택 정리
    ; 8byte 만큼 스택을 정리하고 리턴
    ret 8
```

※ __stdcall(WINAPI) 방식으로 호출하는 함수
- 호출 당하는 쪽(Callee)에서 스택을 정리
- ret 8 : 돌아갈 주소에 +8을 해서 스택을 복원
- Windows API : MessageBox()

어셈블리어 실습

- **실습 – 레지스터(Caller-save, Callee-save)**
  - 함수 내에서 또 다른 함수 호출 시에는 레지스터 간의 충돌을 고려하여 Windows의 컴파일러들은 아래와 같은 규칙을 사용
  - Caller-save 레지스터(scratch 레지스터 혹은 volatile 레지스터라고 부르기도 함)는 함수 내에서 제약 없이 마음대로 쓸 수 있는 레지스터 레지스터의 사용이 이를 호출한 이전 함수에 어떠한 영향도 미치지 않는 레지스터들
  - Callee-save 레지스터는 이를 호출한 함수에 영향을 미칠 수 있기 때문에 해당 레지스터를 사용 전에 반드시 저장하고 사용해야 하는 레지스터 해당 함수가 끝났을 때에는 반드시 이전 값으로 되돌려 주어 이 함수를 호출한 함수에 어떠한 영향을 미쳐서도 안 되는 레지스터들

Platform	Caller-save 레지스터	Callee-save 레지스터
x86 Windows	EAX, ECX, EDX, ST(0)~ST(7), XMM0~XMM7	EBX, ESI, EDI, EBP
x64 Windows	RAX, RCX, RDX, R8~R11, ST(0)~ST(7), XMM0~XMM5, High half of XMM6~XMM15	RBX, RSI, RDI, RBP, R12~R15, XMM6~XMM15

```
push ebp
mov ebp, esp
push ebx ; ebx, esi, edi 를 백업
push esi
push edi

mov esi, 1
mov edi, 2
lea ebx, [esi+edi]

pop edi ; 백업한 값을 다시 복구
pop esi
pop ebx
pop ebp

ret
```

※ 해당 규약을 따르지 않아도 코드는 작동을 하겠지만, 의도치 않은 오류가 발생할 가능성이 있으므로 규약을 지켜야 함

Q A