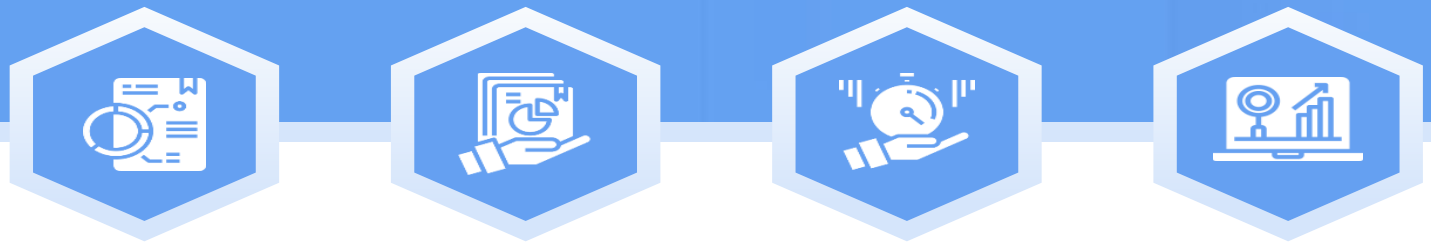
시스템 보안

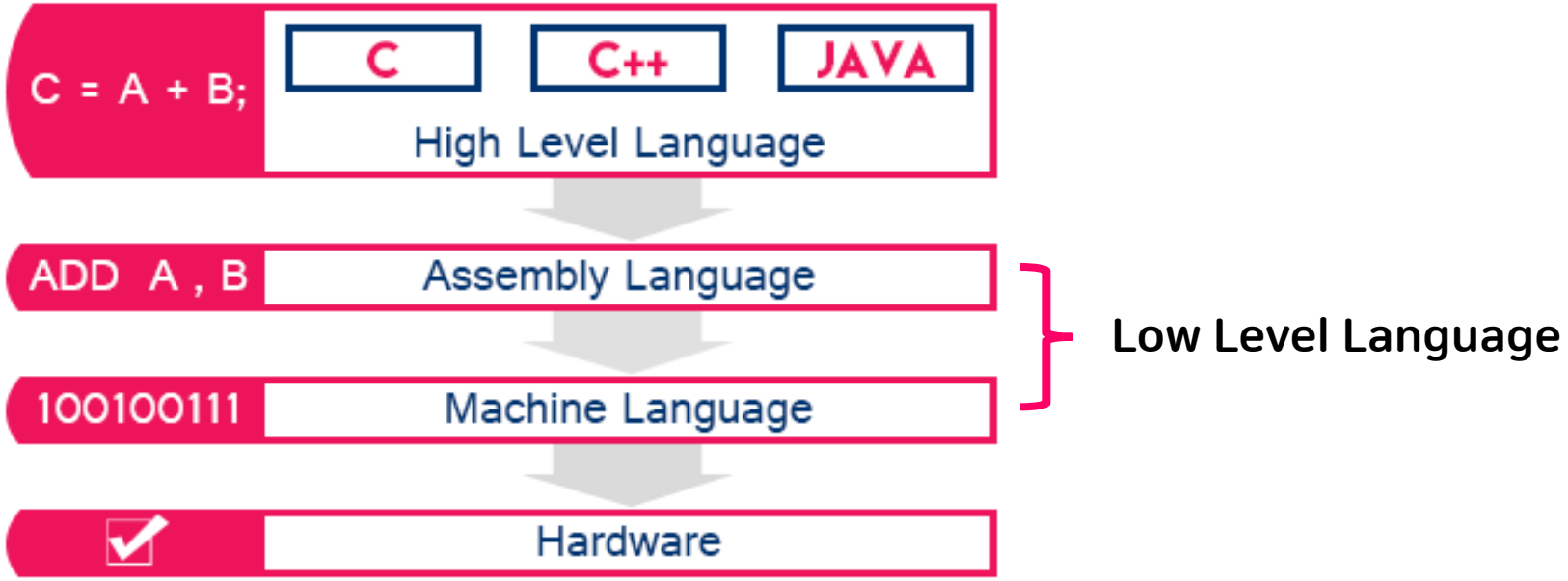
어셈블리어의 이해 - 1



KOREA UNIVERSITY
SEJONG CAMPUS

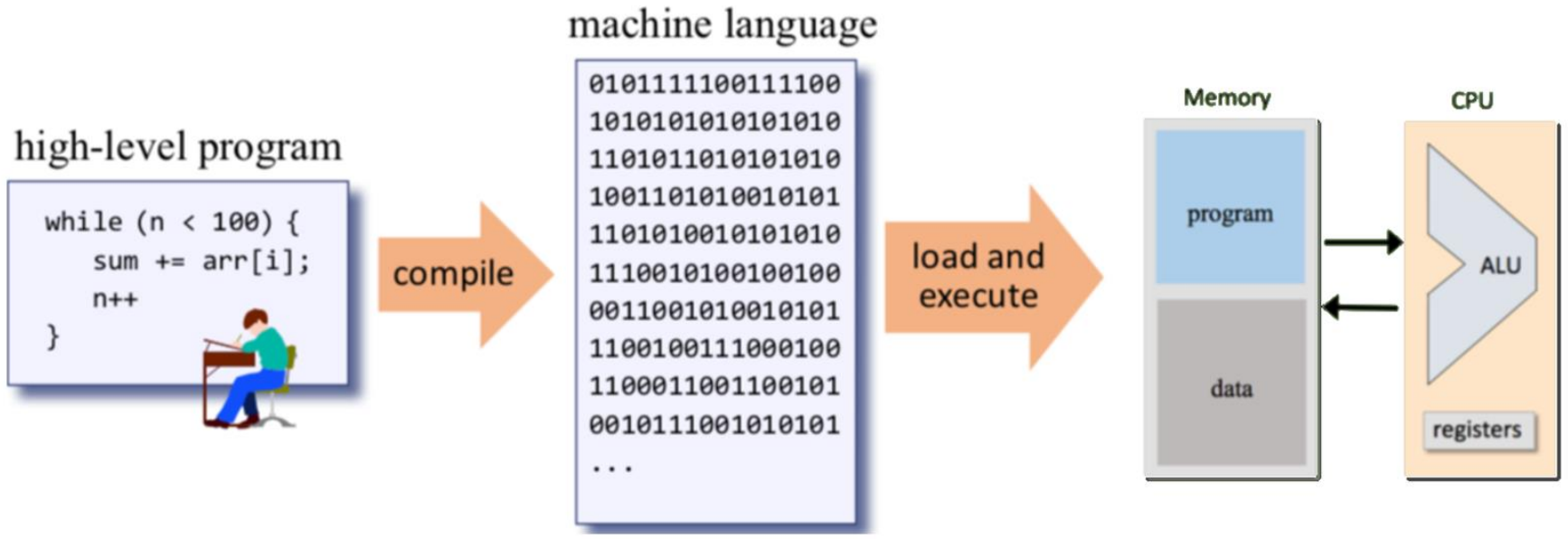
어셈블리어의 이해

- 컴퓨터 언어



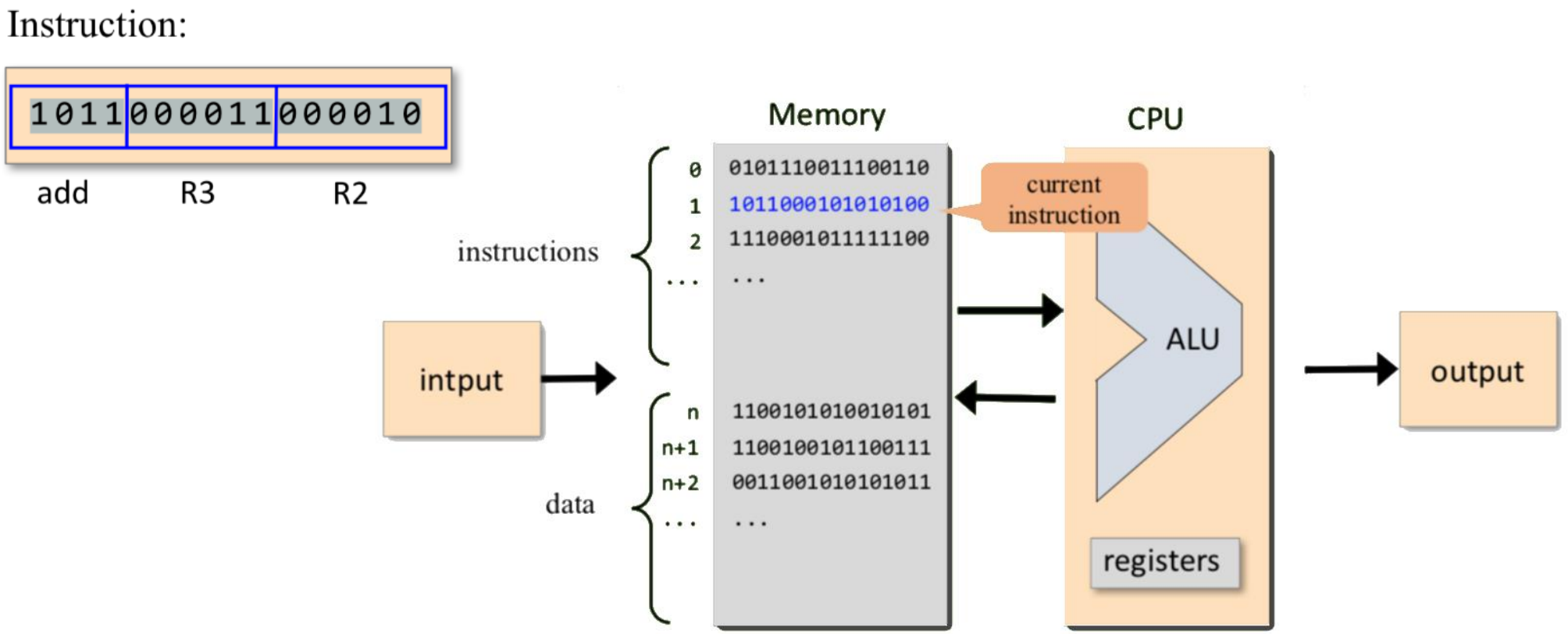
어셈블리어의 이해

- 프로그램 실행 과정



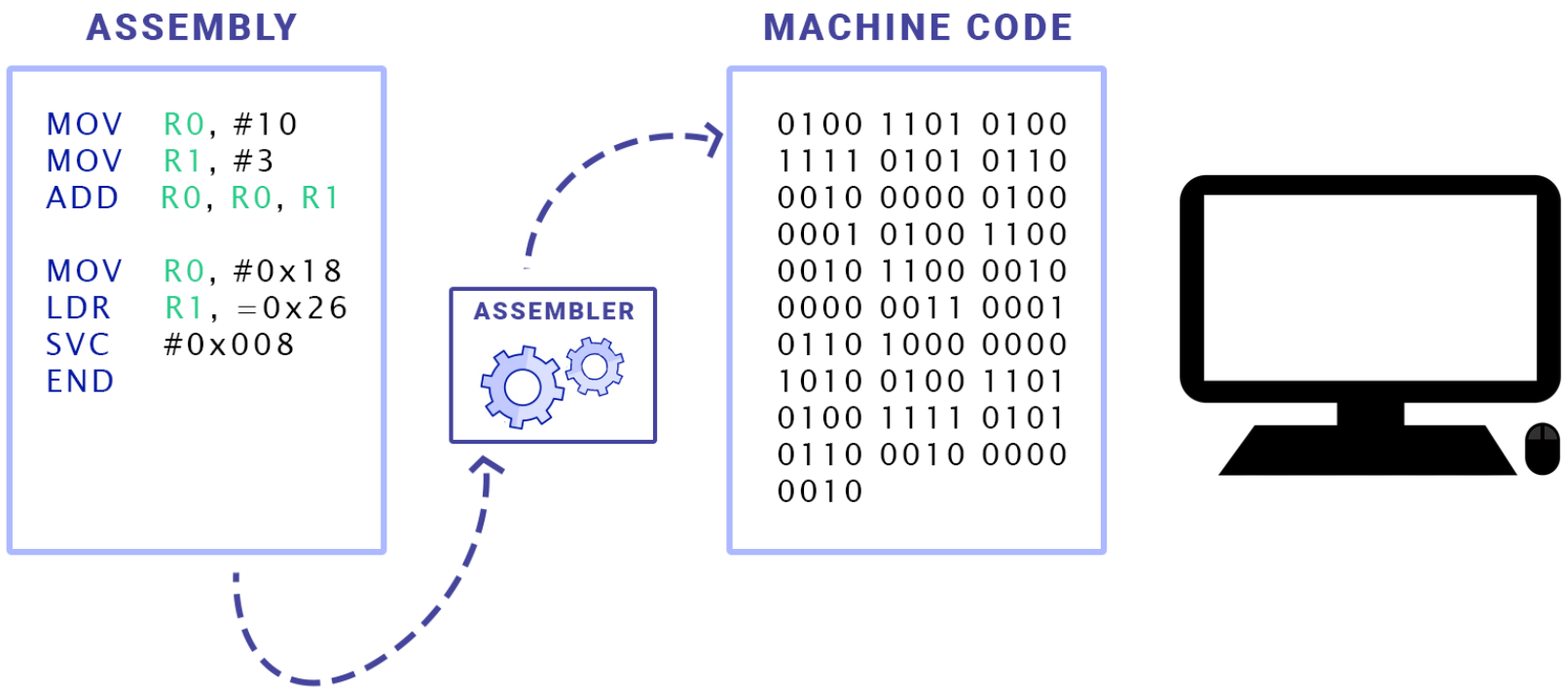
어셈블리어의 이해

- **기계어(Machine Language)란?**
  - 기계어란 CPU가 별도의 해석 과정을 거치지 않고 직접 실행할 수 있는 언어를 말하며, 0과 1로 구성된 2진 숫자로 이루어져 있음
  - 프로그램은 기계어로 번역되어야 CPU가 그 내용을 읽어서 실행시킬 수 있음



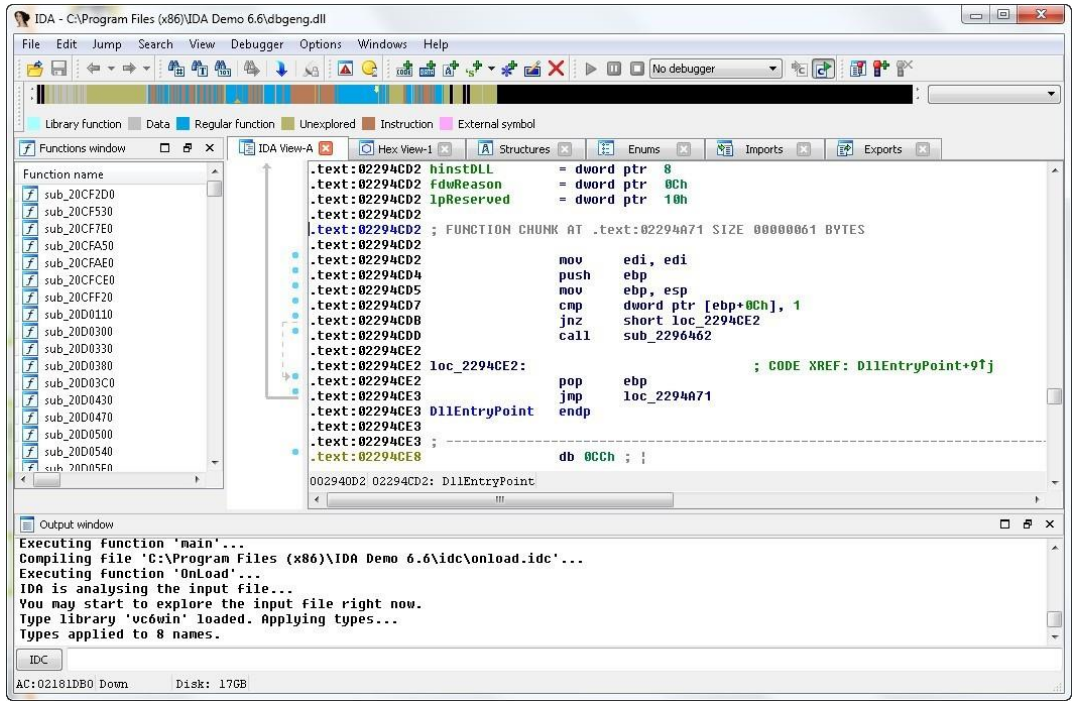
어셈블리어의 이해

- **어셈블리어(Assembly Language)란?**
  - 어셈블리어는 컴퓨터 프로그래밍 언어의 하나로 기계어를 1:1로 매핑할 수 있는 가장 저 레벨의 언어
  - CPU, 레지스터, 메모리 사이에 데이터를 조작하는 것이 주 기능이며, 사용하는 명령어는 CPU에 내장되어 있는 명령어를 사용



어셈블리어 기본

- **어셈블리어(Assembly Language)란?**
  - 어셈블리어는 컴퓨터 프로그래밍 언어의 하나로 기계어를 1:1로 매핑할 수 있는 가장 저 레벨의 언어
  - CPU, 레지스터, 메모리 사이에 데이터를 조작하는 것이 주 기능이며, 사용하는 명령어는 CPU에 내장되어 있는 명령어를 사용

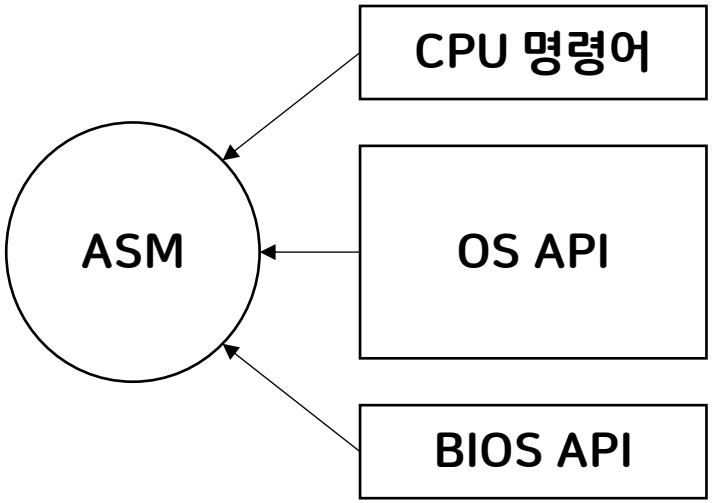



```
int main()
{
    int result = Func(4, 5, 2, 10, 9, 8);
    int a = 10;

    return 0;
}
```

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main            proc near               ; CODE XREF: __scrt_common_main_seh+107↓p
                                        ; DATA XREF: .pdata:00007FF71034400C↓o

e               = dword ptr -28h
f               = dword ptr -20h
result          = dword ptr -18h
a               = dword ptr -14h

                sub     rsp, 48h
                mov     [rsp+48h+f], 8  ; f
                mov     [rsp+48h+e], 9  ; e
                mov     r9d, 0Ah        ; d
                mov     r8d, 2          ; c
                mov     edx, 5          ; b
                mov     ecx, 4          ; a
                call    ?Func@@YAHHHHHH@Z ; Func(int,int,int,int,int,int)
                mov     [rsp+48h+result], eax
                mov     [rsp+48h+a], 0Ah
                xor     eax, eax
                add     rsp, 48h
                retn
main            endp
```

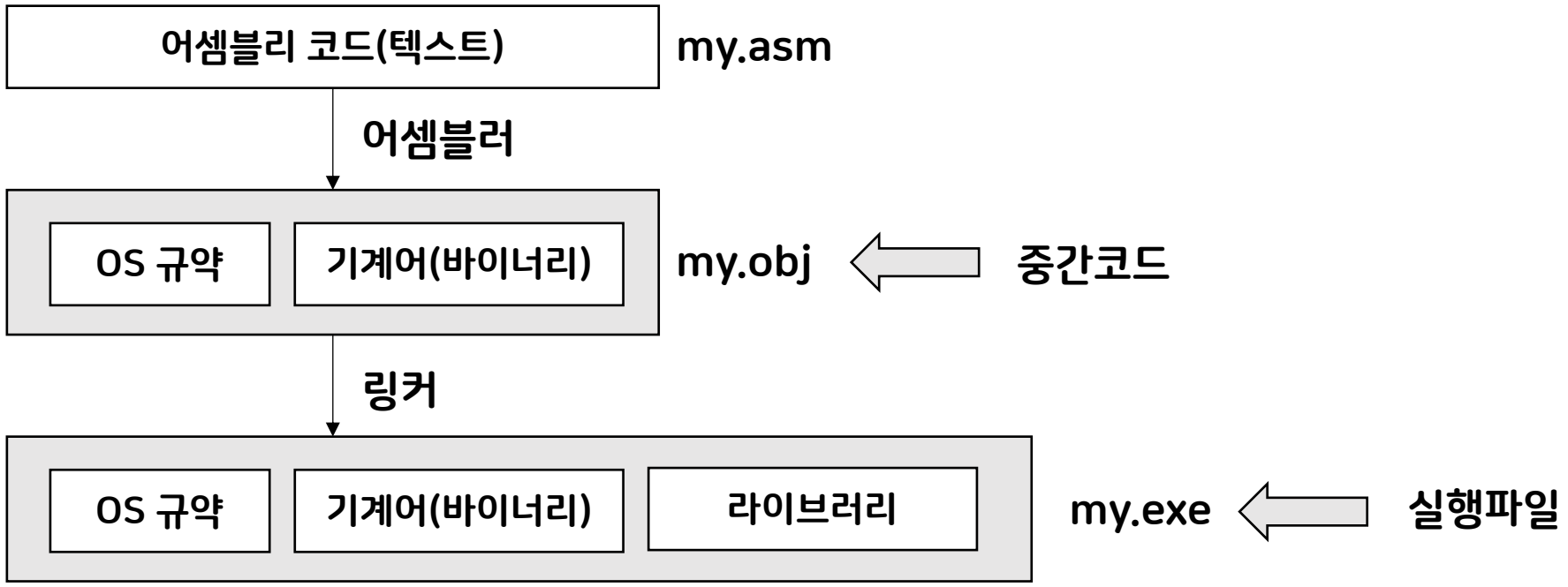
어셈블리어의 이해

- **어셈블리어(Assembly Language)**
  - 어셈블리어로 프로그래밍 한다는 것은 결국 CPU의 명령어를 사용하여 기능을 구현하는 것으로 어떤 로직을 구현하려면 반복적인 많은 작업이 필요
  - 이런 반복적인 작업들을 미리 구현해서 제공하는 것이 운영체제가 제공하는 함수(API)
  - 운영체제가 제공하는 API 뿐만 아니라 바이오스에 내장되어 있는 API도 사용할 수 있음



어셈블리어의 이해

- **어셈블리어(Assembly Language)**
  - 어셈블리어로 작성된 소스코드를 실행 가능한 바이너리(기계어)로 변환하는 프로그램이 어셈블러(assembler)
  - 어셈블러에 의해서 만들어진 바이너리가 실행되려면 컴파일을 통해서 변환된 기계어와 필요한 라이브러리들을 모두 결합하는 작업이 필요한데 이런 작업을 링킹(linking)이라고 하며, 이러한 기능을 수행하는 것이 링커(linker)



어셈블리어의 이해

- ## 어셈블리어(Assembly Language)의 표기법

  - 어셈블리 언어는 Intel과 AT&T 두 가지 문법(표기법)을 주로 사용하며 대소문자는 구분하지 않음
    윈도우에서는 Intel 문법을 리눅스에서는 AT&T 문법을 주로 사용

	Intel 문법	AT&T 문법
레지스터 표현	eax	%eax
상수 표현	16진수(h, 0x), 2진수(b, 0b) ex) 8h, 0x8, 1000b, 0b1000	$숫자 ex) $0x8, $0b1000
메모리 주소 참조	[eax]	(%eax)
레지스터 + offset 위치	[eax+숫자]	숫자(%eax)

어셈블리어의 이해

- **어셈블리어(Assembly Language)의 표기법**
  - 윈도우에서는 Intel 문법, 리눅스에서는 AT&T 문법을 주로 사용
  - Opcode : 명령어 , Operand : 피 연산자

- Intel 문법과 AT&T 문법의 오퍼랜드(Operand) 위치 차이
  Intel 문법에서는 목적지(destination)가 먼저 오고 출발지(source)이 뒤에 오지만, AT&T에서는 반대

Opcode	Operand1	Operand2
add	eax	ebx

Intel

Opcode	Operand1	Operand2
add	%eax	%ebx

AT&T

Intel의 경우 Operand2가 source고 Operand1이 destination이 되어 “ebx의 값을 eax에 더한다” 라는 의미가 되고,
AT&T의 경우에는 반대가 되어 “eax의 값을 ebx로 더한다”라는 의미

어셈블리어의 이해

- **어셈블리어(Assembly Language)의 표기법**
  - 윈도우에서는 Intel 문법, 리눅스에서는 AT&T 문법을 주로 사용
  - Opcode : 명령어 , Operand : 피 연산자

- Intel 문법에서의 어셈블리 명령 형식

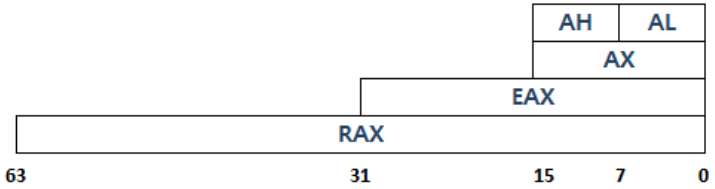
Label1:	mov	ax,	bx	; comment
라벨	명령 코드	첫 번째 오퍼랜드	두 번째 오퍼랜드	설명(주석)
	(opcode)	(operand1, destination)	(operand2, source)	

어셈블리어의 이해

- **어셈블리어(Assembly Language)의 표기법(예시)**
  - Intel 문법에서의 어셈블리어 예시 (이후 Intel 문법을 사용)

```
%include "io.inc"

section .text
global CMAIN
CMAIN:
    PRINT_HEX 4, ax
    NEWLINE
    mov ax, bx          ; bx (source) 레지스터의 값을 ax (destination) 레지스터에 저장
    PRINT_HEX 4, bx
    NEWLINE
    PRINT_HEX 4, ax
    NEWLINE
    jmp Label_1         ; Label_1으로 무조건 이동
    ret

Label_1:
    PRINT_HEX 4, 0x100
    NEWLINE
    ret
```

SASM(SimpleASM) 예시



어셈블리어의 이해

- **어셈블리어(Assembly Language)의 표기법(예시)**
  - 변수 선언은 초기화된 변수가 저장되는 ".data" 섹션과 초기화가 되지 않은 변수가 저장되는 ".bss" 섹션으로 구분
  - ".bss" 섹션에는 변수명, 크기, 개수를 지정하고 ".data" 섹션에는 초기값이 설정된 변수를 선언

크기 지시자	설명
resb	1 byte
resw	2 byte
resd	4 byte
resq	8 byte

".bss" 섹션

크기 지시자	설명
db	1 byte
dw	2 byte
dd	4 byte
dq	8 byte

".data" 섹션

```
section .bss
    bss1 resb 1
    bss2 resw 1
    bss3 resd 1

section .data
    var1 db 0x12
    var2 dw 0x1234
    var3 dd 0x12345678
```

SASM(SimpleASM) 예시

어셈블리어의 이해

- **어셈블리어(Assembly Language)의 표기법(예시)**
  - 상수는 프로그램 내에서 변하지 않는 값을 가지는 데이터를 정의할 때 사용
  - %define은 C 언어의 #define과 같은 의미로 사용(컴파일 시 대체)
  - %assign이나 %define은 재 정의를 통해서 변경 가능

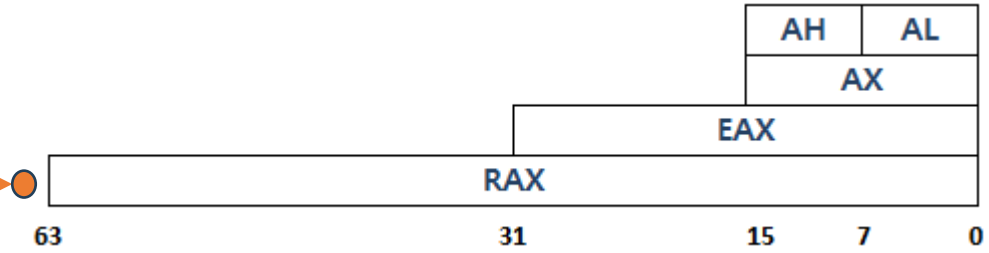
```
%include "io.inc"

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

; %assign 숫자 상수 선언, 코드 내에서 값을 변경하기 위한 재 정의 가능
%assign const_2 0x1001

; %define 숫자 또는 문자열 상수 선언, 코드 내에서 값을 변경하기 위한 재 정의 가능
%define const_3 0x10002

; %define은 c언어의 define과 같은 의미
%define PTR [EBP + 4]
%define MOV_ESI_300 mov esi, 300
```

SASM(SimpleASM) 예시

어셈블리어의 이해

- **어셈블리어(Assembly Language)**
  - 어셈블리어에서 가장 많이 사용하게 되는 범용 레지스터(General purpose register)
    32bit CPU에서는 8개, 64bit CPU에서는 16개(R8 ~ R15 추가)

Size (in Bits)			
64	32	16	8
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8~R15	R8D~R15D	R8W~R15W	R8L~R15L



어셈블리어의 이해

- ## 어셈블리어(Assembly Language)

  - 어셈블리어에서 가장 많이 사용하게 되는 범용 레지스터(General register)

  - EAX(Accumulation) : 산술, 논리 연산을 할 때 사용되며 함수의 리턴 값을 저장

  - EBX(Base Register) : ESI나 EDI와 결합하여 간접 번지 지정에 주로 사용

  - ECX(Counter Register) : 반복 명령어 사용하는 경우, 반복 카운터로 사용

  - EDX(Data Register) : 보통 EAX와 함께 연동해서 사용하며, 큰 수의 복잡한 연산 과 부호 확장 명령에 사용, 문자열을 출력할 때도 사용

    4byte * 4byte를 연산하면 EAX 만으로는 담을 수 없기 때문에 EDX(상위 bit), EAX(하위 bit)를 이용하여 8byte에 저장

    나누기를 진행할 경우 몫은 EAX 나머지는 EDX에 저장

어셈블리어의 이해

- **어셈블리어(Assembly Language)**

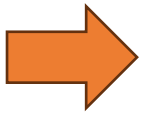
  - 어셈블리어에서 가장 많이 사용하게 되는 인덱스 레지스터(Index register)

  - ESI(Source Index) :
    데이터 복사나 조작 시 출발지(Source)의 주소를 저장

  - EDI(Destination Index) :
    데이터 복사나 조작 시 목적지(Destination)의 주소를 저장

  - MOVS(ESI 레지스터가 가리키는 주소에 있는 값을 EDI가 가리키는 주소로 복사하는 명령)를 사용할 때 마다 ESI와 EDI가 자동 증가
    MOVSB(1바이트 단위), MOVSW(2바이트 단위), MOVSD(4바이트 단위)

```
MOV ESI, OFFSET source
MOV EDI, OFFSET destination
MOV ECX, 10
REP MOVSB
```



source 주소의 값을 destination 주소로
ECX에 저장된 값 만큼(ECX가 0이 될 때 까지) MOVSB(바이트 단위) 명령 반복
(REP 명령은 ECX > 0 인 동안 반복하라는 명령, ECX 값은 반복을 하면서 1씩 감소)

어셈블리어의 이해

- **어셈블리어(Assembly Language)**

  - 어셈블리어에서 가장 많이 사용하게 되는 포인터 레지스터(Pointer register)

  - EBP(Base Pointer) :

    현재 스택 프레임의 베이스 주소를 담고 있는 레지스터로, 스택 프레임이란 함수 호출 과정에서 할당되는 스택 메모리 블록을 의미

  - ESP(Stack Pointer) :

    현재 스택의 최 상단 주소 값을 저장하고 있는 레지스터로 주로 스택에서 현재 위치를 나타내는데 사용

    PUSH(스택이 쌓일 때 마다 ESP 값이 증가) 나 POP(스택에서 제거 될 때 마다 ESP 값이 감소) 등의 명령어로 값이 변경됨

```
PUSH    EBP             ⇒ 기존의 EBP값을 스택에 집어 넣어라
MOV     EBP, ESP        ⇒ ESP의 값을 EBP로 옮겨라
```

스택 프레임(Stack Frame) 생성

```
MOV     ESP, EBP        ⇒ ESP 값을 원래대로 복원시켜라
POP     EBP             ⇒ 스택에 백업한 EBP 값을 복원시켜라
RETN                    ⇒ 스택에 저장된 복귀 주소로 리턴하여 함수 종료
```

스택 프레임 (Stack Frame) 해제

어셈블리어의 이해

- 어셈블리어(Assembly Language)

  - EIP(Instruction Pointer)

    CPU가 처리할 명령어의 주소를 나타내는 레지스터로

    CPU는 EIP에 저장된 메모리 주소의 명령어(Instruction)을 하나 처리하고 난 후 자동으로 그 명령어 길이만큼 EIP를 증가

    범용 레지스터들과는 다르게 EIP는 값을 직접 변경할 수 없도록 되어 있어 다른 명령어를 통하여 간접적으로 변경해야 함

    예) 특정 명령어(JMP, CALL, RET)를 사용하거나 Interrupt, Exception을 발생 시켜야 함

어셈블리어의 이해

- ## 어셈블리어(Assembly Language) – 데이터 타입

  - 어셈블리어에서 사용하는 정수형 데이터 타입

타입	설명
BYTE	8비트 부호 없는 정수
SBYTE	8비트 부호 있는 정수
WORD	16비트 부호 없는 정수
SWORD	16비트 부호 있는 정수
DWORD	32비트 부호 없는 정수
SDWORD	32비트 부호 있는 정수
FWORD	48비트 정수
QWORD	64비트 정수

- 접두사 의미

  S : Signed

  F : Far

  Q : Quad

- 인텔 프로세서가 초기 16bit 시기부터

  8bit 데이터 타입을 BYTE, 16bit 데이터 타입을 WORD 로 명명

  이후 데이터 크기가 커짐에 따라서 WORD에서 확장된 형태로 명명

어셈블리어의 이해

- **어셈블리어(Assembly Language) – 데이터 타입**
  - 어셈블리어에서 사용하는 데이터 타입과 C언어에서의 데이터 타입 비교

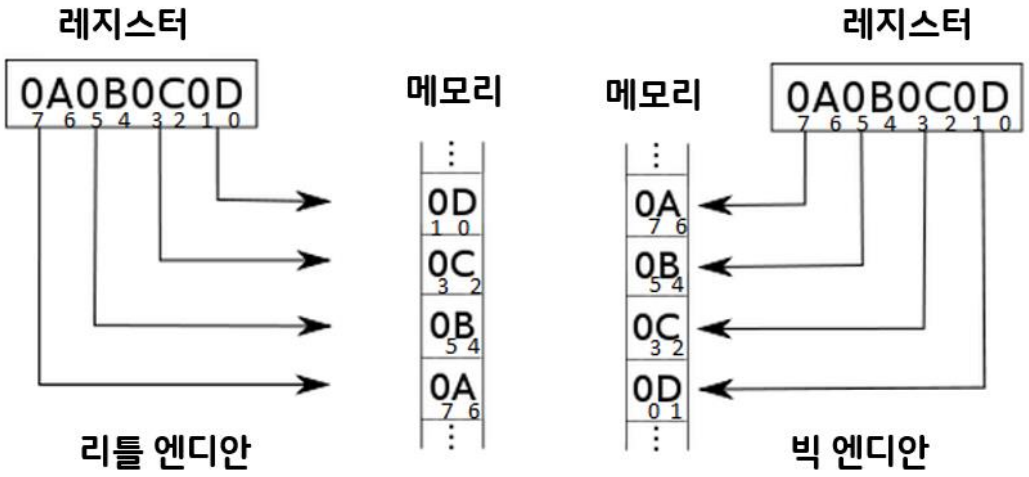
C 선언	Intel 데이터 타입	어셈블리어 접미사	사이즈(바이트)
char	BYTE(Byte)	b	1
short	WORD(Word)	w	2
int	DWORD(Double word)	d	4
long	QWORD(Quad word)	q	8
char *	QWORD(Quad word)	q	8

- 8, 16, 32, 64비트를 어셈블리어 접미사로 각각 b, w, d, q로 표시 (NASM 기준)

- 다른 어셈블러에서는 접미사로 각각 b, w, l, q로 표시하며 mov 명령어의 경우에도 movb, movw, movl, movq로 사용하기도 함

어셈블리어의 이해

- **어셈블리어(Assembly Language) – 데이터 저장 방식**
  - 메모리에 데이터를 저장하는 바이트 순서를 바이트 정렬(byte ordering)이라고 하며, 바이트 정렬 방식은 CPU에 따라서 달라짐
  - 인텔 계열에서는 리틀 엔디안(little endian) 방식을 사용하고, SPARK이나 RISC 계열의 CPU에서는 빅 엔디안(big endian) 방식을 사용
  - 빅 엔디안은 최상위 바이트(MSB, Most Significant Byte) 부터 차례로 저장하고,
    리틀 엔디안은 반대로 최하위 바이트(LSB, Least Significant Byte) 부터 차례로 저장하는 방식
  - 어떤 방식을 선택할 것인지는 CPU 내부에서의 연산을 어떻게 효율적으로 할 것인지에 대한 CPU 제조사의 전략에 따름



- 리틀 엔디안이나 빅 엔디안은
  메모리에 데이터가 저장되는 방식

어셈블리어의 이해

- **어셈블리어(Assembly Language) – 데이터 저장 방식**
  - 빅 엔디안은 최상위 바이트(MSB, Most Significant Byte) 부터 차례로 저장하고,
    리틀 엔디안은 반대로 최하위 바이트(LSB, Least Significant Byte) 부터 차례로 저장하는 방식

- 1바이트 크기의
  0x12 데이터 저장

주소	1000
저장 데이터 (리틀 엔디안)	0x12

주소	1000
저장 데이터 (빅 엔디안)	0x12

- 2바이트(워드) 크기의
  0x1234 데이터 저장

주소	1000	1001
저장 데이터 (리틀 엔디안)	0x34	0x12

주소	1000	1001
저장 데이터 (빅 엔디안)	0x12	0x34

- 4바이트(더블 워드) 크기의
  0x12345678 데이터 저장

주소	1000	1001	1002	1003
저장 데이터 (리틀 엔디안)	0x78	0x56	0x34	0x12

주소	1000	1001	1002	1003
저장 데이터 (빅 엔디안)	0x12	0x34	0x56	0x78

어셈블리어의 이해

- **어셈블리어(Assembly Language) – 주소 지정 방식**

  - OS 환경과 어셈블러 종류에 따라서 달라짐

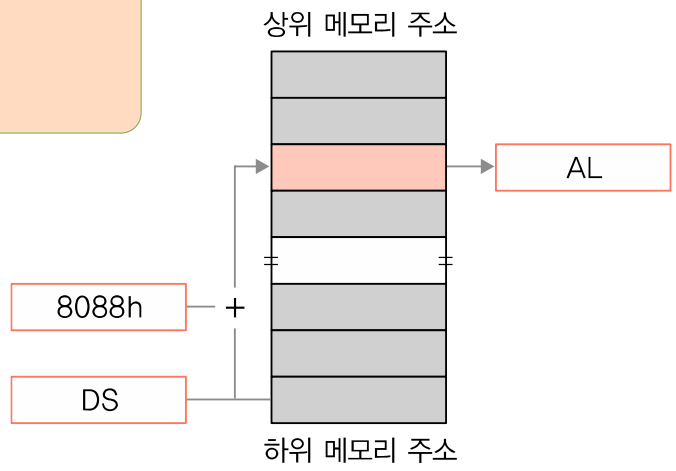
  - 레지스터 주소 지정 : 레지스터의 주소 값을 직접 지정 복사, 처리 속도 가장 빠름

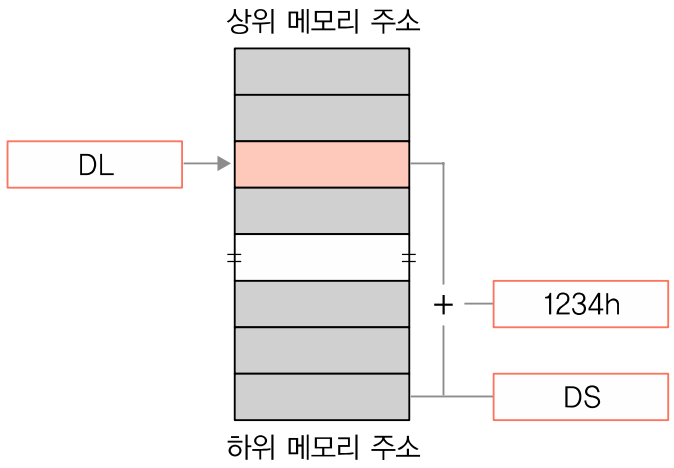
```
MOV [DX], [BX]
```

  - 직접 메모리 주소 지정 : 일반적인 주소 지정 방식, 오퍼랜드 하나가 메모리 위치를 참조하고 다른 하나는 레지스터를 참조

    예) DS:[8088h]와 DS:[1234h]는 각각 '세그먼트:오프셋' 형식의 메모리에 직접 접근하는 방식

```
MOV AL, DS:[8088h]
MOV DS:[1234h], DL
```



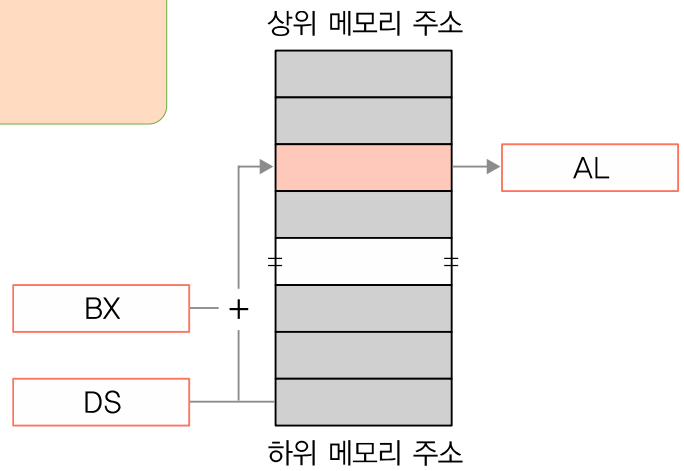


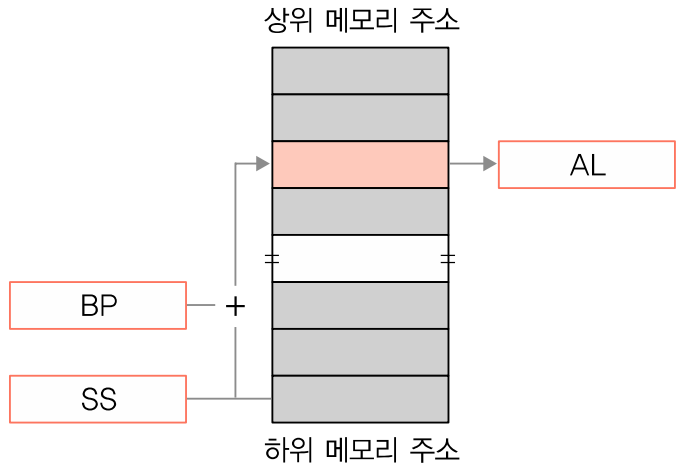
어셈블리어의 이해

- **어셈블리어(Assembly Language) – 주소 지정 방식**

  - 레지스터 간접 주소 지정 : '세그먼트:오프셋' 형식을 사용

```
MOV AL,   DS:[BX]
MOV AL,   SS:[BP]
```





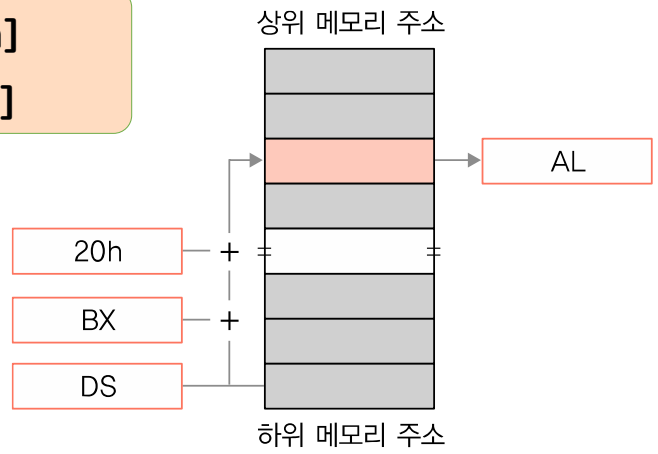
어셈블리어의 이해

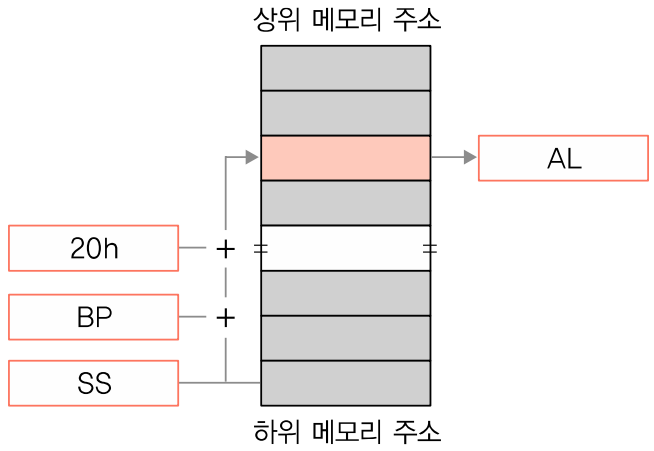
- **어셈블리어(Assembly Language) – 주소 지정 방식**

- 인덱스 주소 지정 : 레지스터 간접 지정 방식에 변위가 더해진 메모리 주소 지정 방식

예) 20h만큼 더해 메모리를 참조한 명령

```
MOV AL, DS:[BX+20h]
MOV AL, SS:[BP+20h]
```





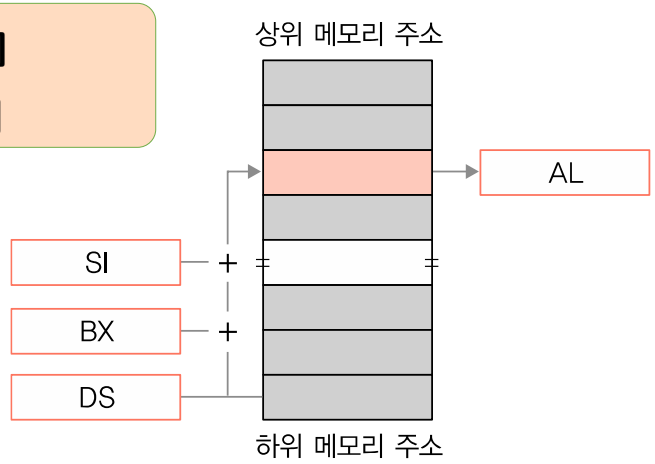
어셈블리어의 이해

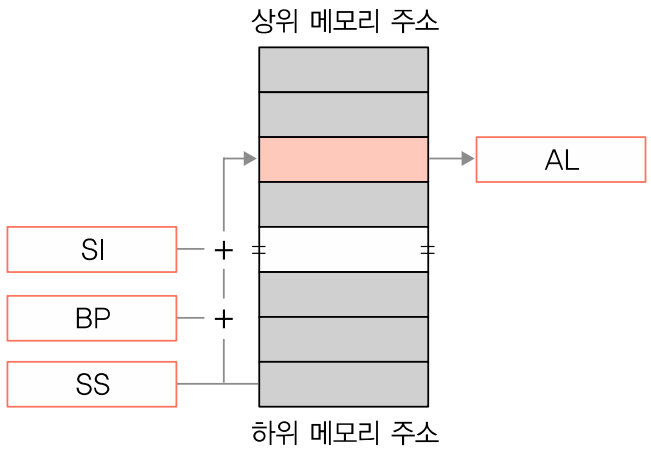
- 어셈블리어(Assembly Language) – 주소 지정 방식

- 베이스 인덱스 주소 지정 : 실제 주소 생성 위해 베이스 레지스터(BX 또는 BP)와 인덱스 레지스터(DI 또는 SI)를 결합한 주소지정 방식

MOV AL, DS:[BX+SI]

MOV AL, SS:[BP+SI]



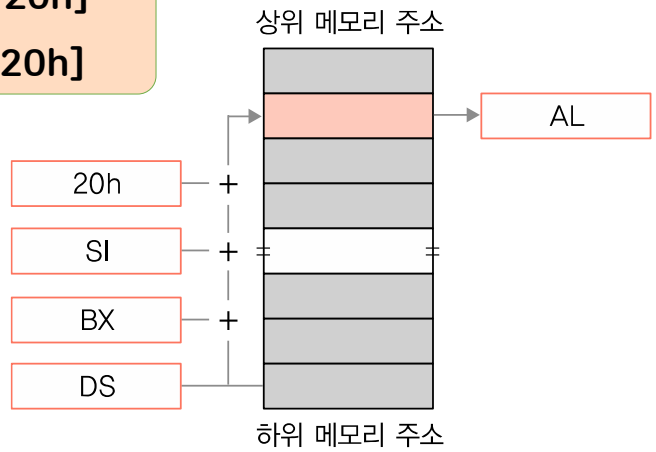


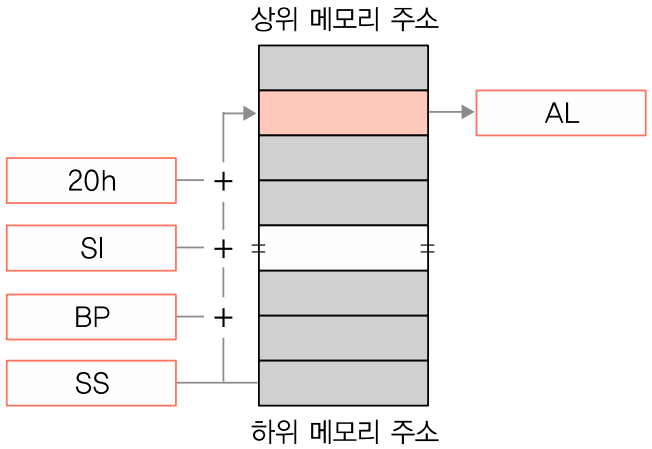
어셈블리어의 이해

- **어셈블리어(Assembly Language) – 주소 지정 방식**

- 변위를 갖는 베이스 인덱스 주소 지정 : 베이스-인덱스의 변형으로 실제 주소 생성 위해 베이스 레지스터, 인덱스 레지스터, 변위를 결합

```
MOV AL, DS:[BX+SI+20h]
MOV AL, SS:[BP+SI+20h]
```





Q A