



CHUNGNAM NATIONAL UNIVERSITY



# 시스템 프로그래밍

강의 6 : 3.7 프로시저

<http://eslab.cnu.ac.kr>



\* Some slides are from Original slides of RBE

# 전달사항

---

중간고사 : 10월 31일 수요일 저녁 7시-9시

# 프로시저의 실행

## 제어의 전달

- 프로시저 코드의 시작부분으로
- 리턴 지점으로 돌아가기

## 데이터의 전달

- 프로시저 인자
- 리턴 값

## 메모리 관리

- 프로시저 실행중에 할당
- 리턴할 때 반환

모든 동작은 기계어로 구현

```
P (...) {  
    .  
    .  
    y = Q(x);  
    print(y)  
    .  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    .  
    .  
    return v[t];  
}
```

# 오늘의 내용

---

## 프로시저

- 스택의 구조
- 호출 관습
  - ▶ 제어의 전달
  - ▶ 데이터의 전달
  - ▶ 지역 데이터의 관리
- 재귀실행

# x86-64 스택

스택규약으로 관리되는 메모리  
영역

작은 주소 방향으로 성장한다

레지스터 **%rsp** 는 가장 작은  
스택 주소를 저장한다

- 스택 탑의 주소

Stack Pointer: **%rsp** →

Stack "Bottom"



주소증가방향

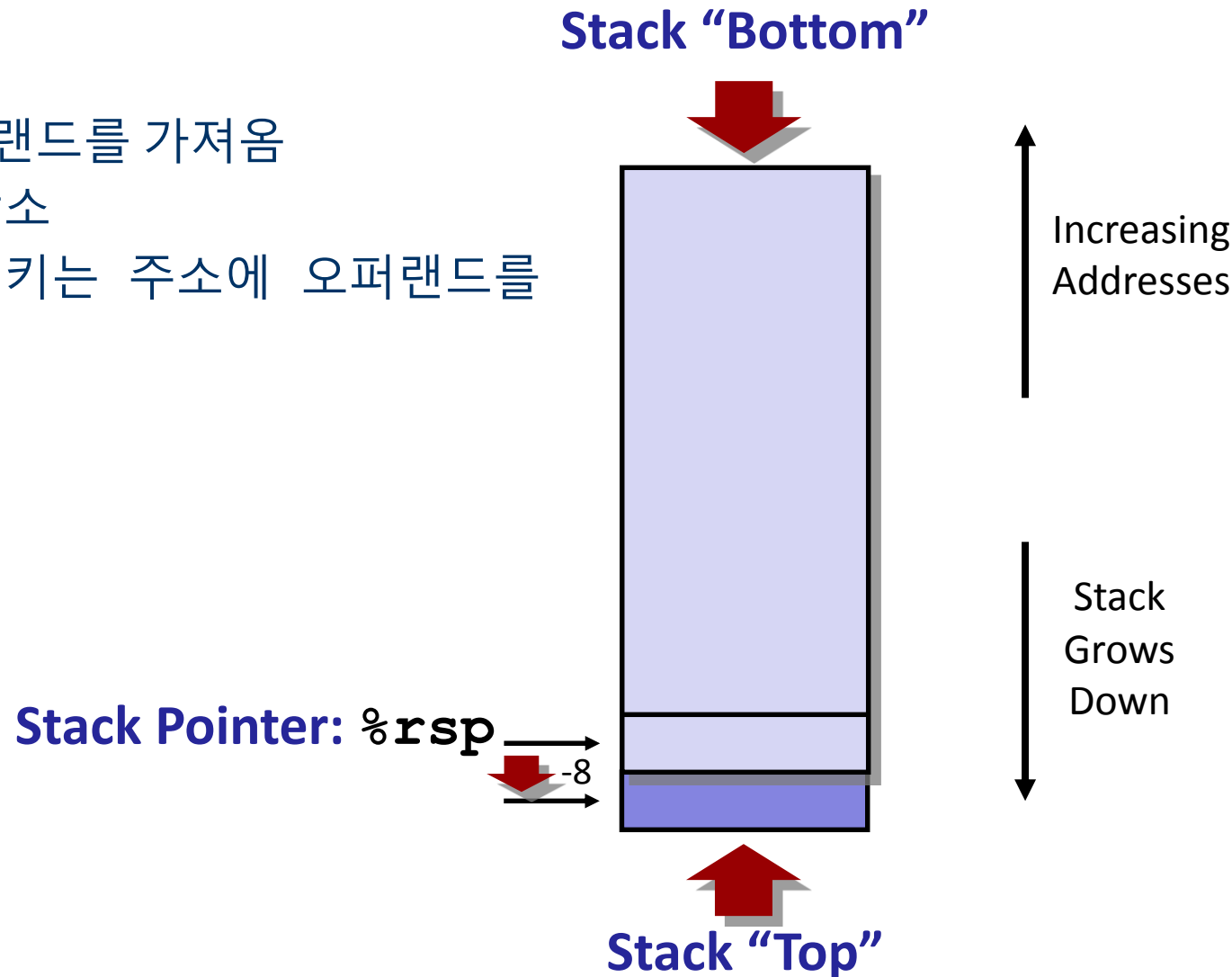
스택은  
아래로  
성장

Stack "Top"

# x86-64 Stack: Push

## `pushq Src`

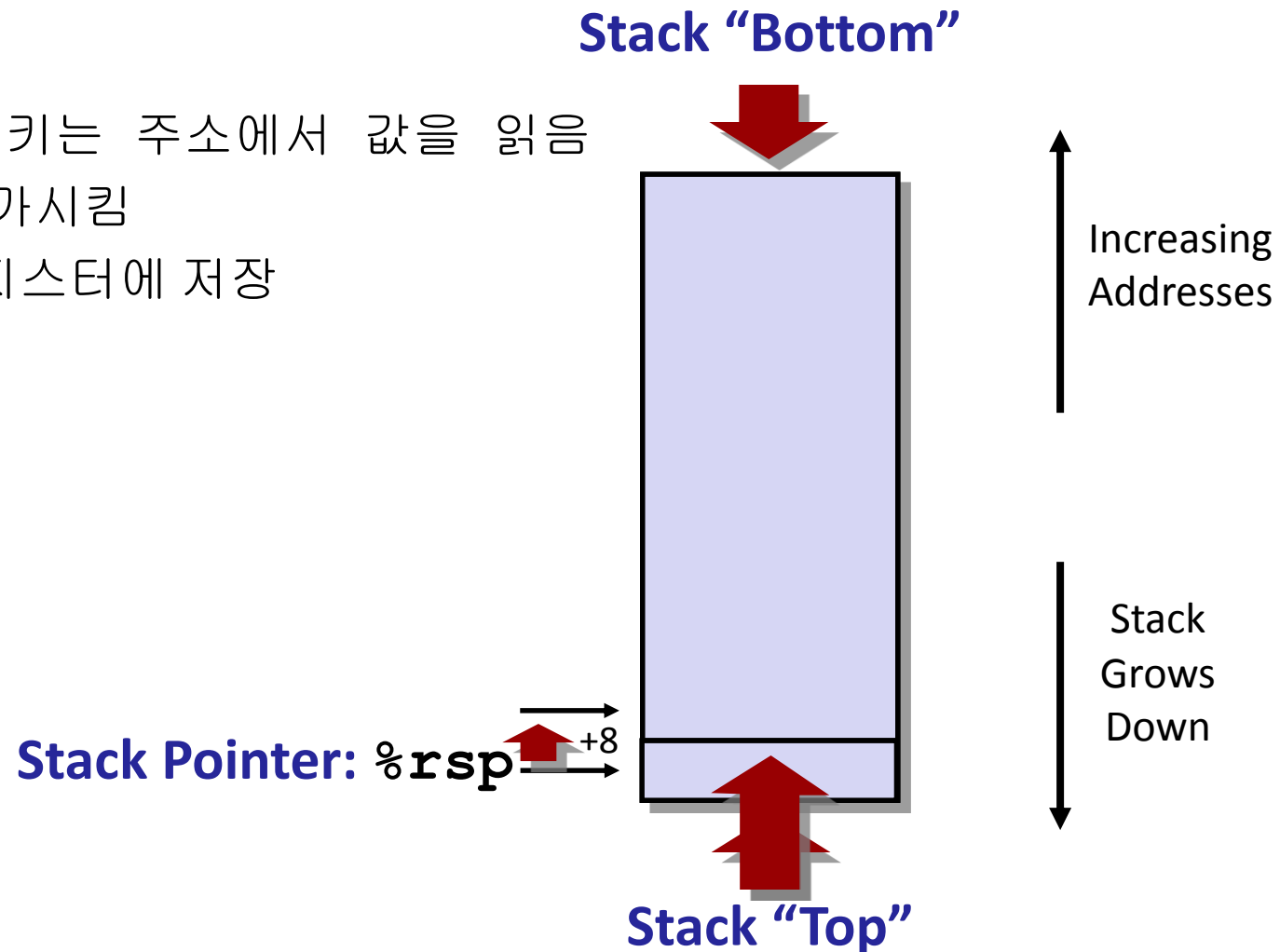
- `Src` 에서 오퍼랜드를 가져옴
- `%rsp` 를 8 감소
- `%rsp` 가 가리키는 주소에 오퍼랜드를 기록



# x86-64 Stack: Pop

## ■ `popq Dest`

- `%rsp`가 가리키는 주소에서 값을 읽음
- `%rsp`를 8 증가시킴
- 값을 `Dest` 레지스터에 저장



# 오늘의 내용

---

## 프로시저

- 스택의 구조
- 호출 관습
  - ▶ 제어의 전달
  - ▶ 데이터의 전달
  - ▶ 지역 데이터의 관리
- 재귀실행



# 코드 예제

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
400540: push    %rbx                # Save %rbx
400541: mov     %rdx,%rbx           # Save dest
400544: callq   400550 <mult2>      # mult2(x,y)
400549: mov     %rax, (%rbx)        # Save at dest
40054c: pop     %rbx                # Restore %rbx
40054d: retq                      # Return
```

```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
400550: mov     %rdi,%rax           # a
400553: imul    %rsi,%rax           # a * b
400557: retq                      # Return
```

# 프로시저의 제어 흐름

스택을 이용하여 프로시저 호출과 리턴을 지원

프로시저 호출: **call label**

- 리턴주소를 스택에 푸시
- **label**로 점프

리턴주소:

- 콜 바로 다음 명령어의 주소

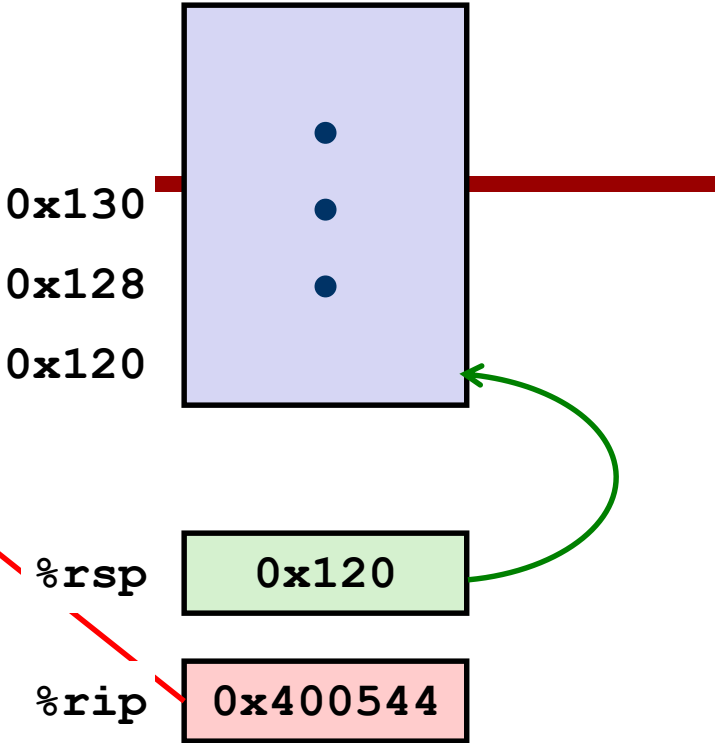
프로시저 리턴: **ret**

- 스택에서 리턴 주소를 팝
- 이 주소로 점프

# 제어흐름의 예 #1

```
00000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov   %rax, (%rbx)
.
.
```

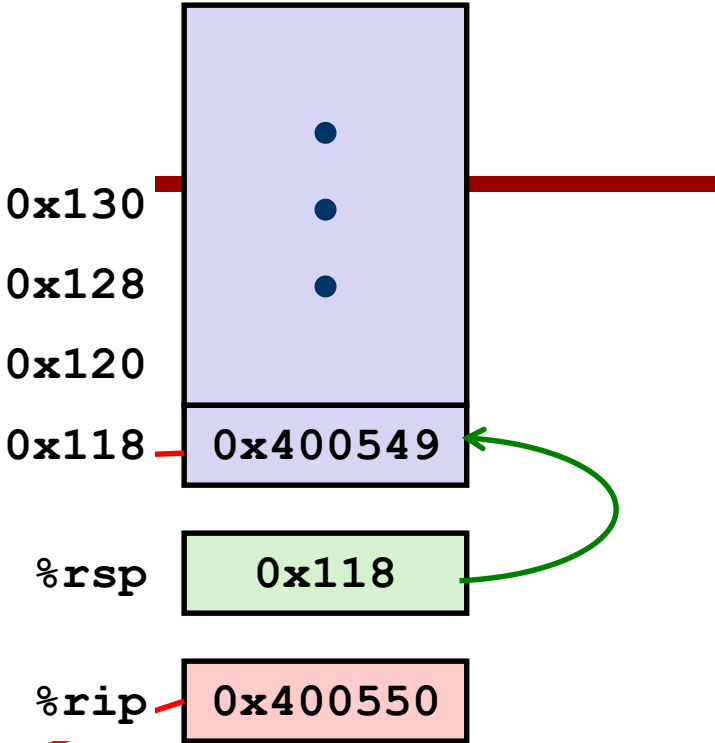
```
00000000000400550 <mult2>:
400550: mov   %rdi, %rax
.
.
400557: retq
```



# 제어흐름의 예 #2

```
00000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov   %rax, (%rbx)
.
.
```

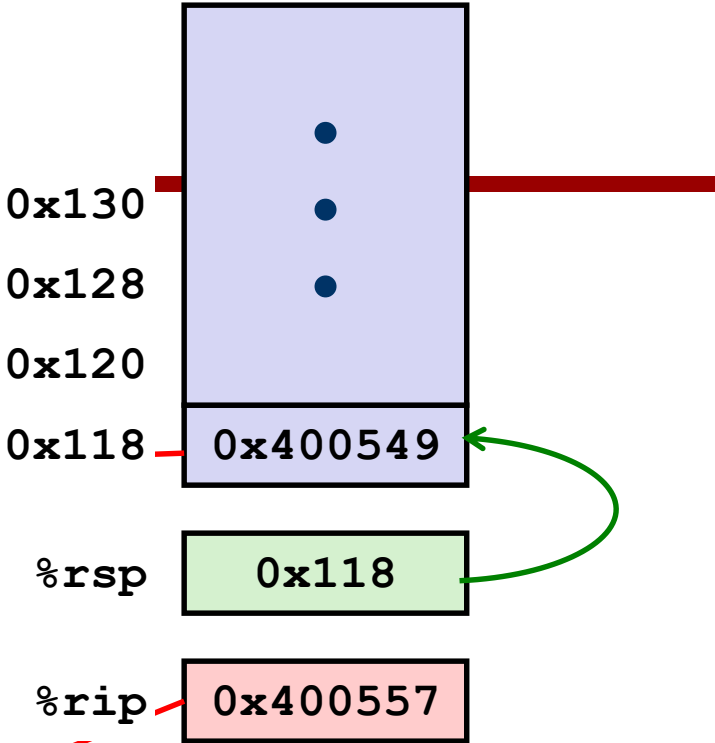
```
00000000000400550 <mult2>:
400550: mov   %rdi,%rax
.
.
400557: retq
```



# 제어흐름의 예 #3

```
00000000000400540 <multstore>:
.
.
400544: callq    400550 <mult2>
400549: mov     %rax, (%rbx)
.
.
```

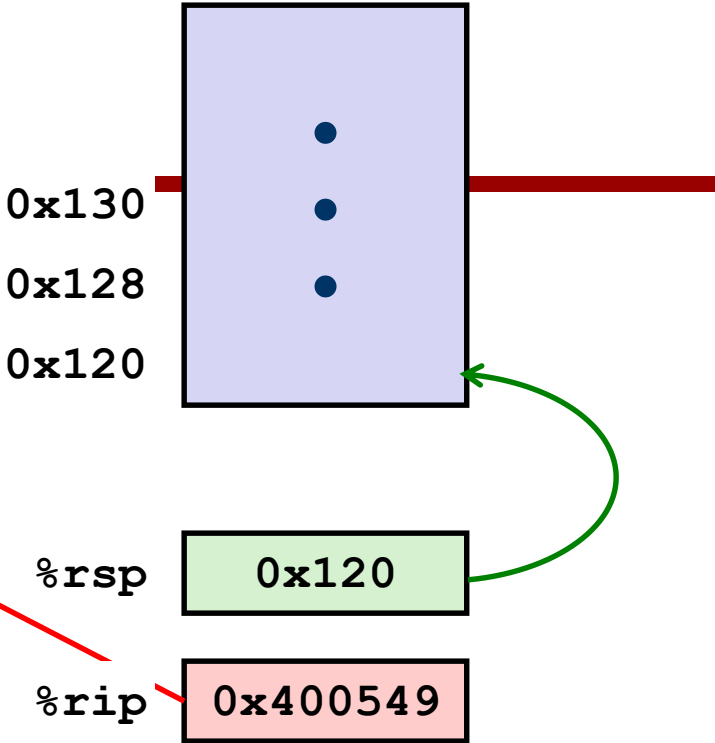
```
00000000000400550 <mult2>:
400550: mov     %rdi,%rax
.
.
400557: retq
```



# 제어흐름의 예 #4

```
00000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov   %rax, (%rbx)
.
.
```

```
00000000000400550 <mult2>:
400550: mov   %rdi, %rax
.
.
400557: retq
```



# 오늘의 내용

---

## 프로시저

- 스택의 구조
- 호출 관습
  - ▶ 제어의 전달
  - ▶ **데이터의 전달**
  - ▶ 지역 데이터의 관리
- 재귀실행

# 프로시저 데이터 흐름

## Registers

첫 6개의 인자

%rdi
%rsi
%rdx
%rcx
%r8
%r9

리턴 값

%rax
------

## Stack

...
Arg $n$
...
Arg 8
Arg 7

스택공간은 필요할 때에만  
할당



# 데이터 흐름 예

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ● ● ●
400541: mov     %rdx,%rbx        # Save dest
400544: callq   400550 <mult2>    # mult2(x,y)
    # t in %rax
400549: mov     %rax, (%rbx)      # Save at dest
    ● ● ●
```

```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
400550: mov     %rdi,%rax        # a
400553: imul    %rsi,%rax        # a * b
    # s in %rax
400557: retq                      # Return
```

# 오늘의 내용

---

## 프로시저

- 스택의 구조
- 호출 관습
  - ▶ 제어의 전달
  - ▶ 데이터의 전달
  - ▶ 지역 데이터의 관리
- 재귀실행

# 스택기반 언어

## 재귀호출을 지원하는 언어

- C, Pascal, Java
- "Reentrant" 해야 하는 코드
  - ➔ 단일 프로시저로 생성한 다수의 동시성 실행개체
- 각 실행개체의 상태를 저장할 장소가 필요함
  - ➔ 인자
  - ➔ 지역변수
  - ➔ 리턴 포인터

## 스택 체제

- 제한된 시간 동안 필요한 프로시저의 상태
  - ➔ 호출 이후 부터 리턴까지
- 피호출자는 호출자가 리턴하기 전에 리턴한다 ?????!!!!

## 스택은 프레임으로 할당된다

- 단일 프로시저 실행에 대한 상태

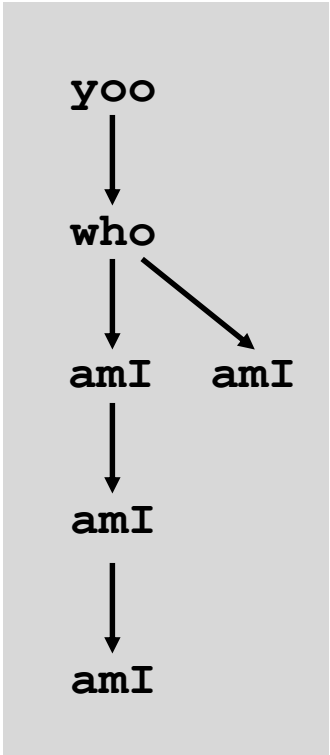
# 콜 체인 예

```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI ();  
  . . .  
  amI ();  
  . . .  
}
```

```
amI (...)  
{  
  .  
  .  
  amI ();  
  .  
  .  
}
```

## 콜 체인 예



프로시저 amI () 는 재귀함수

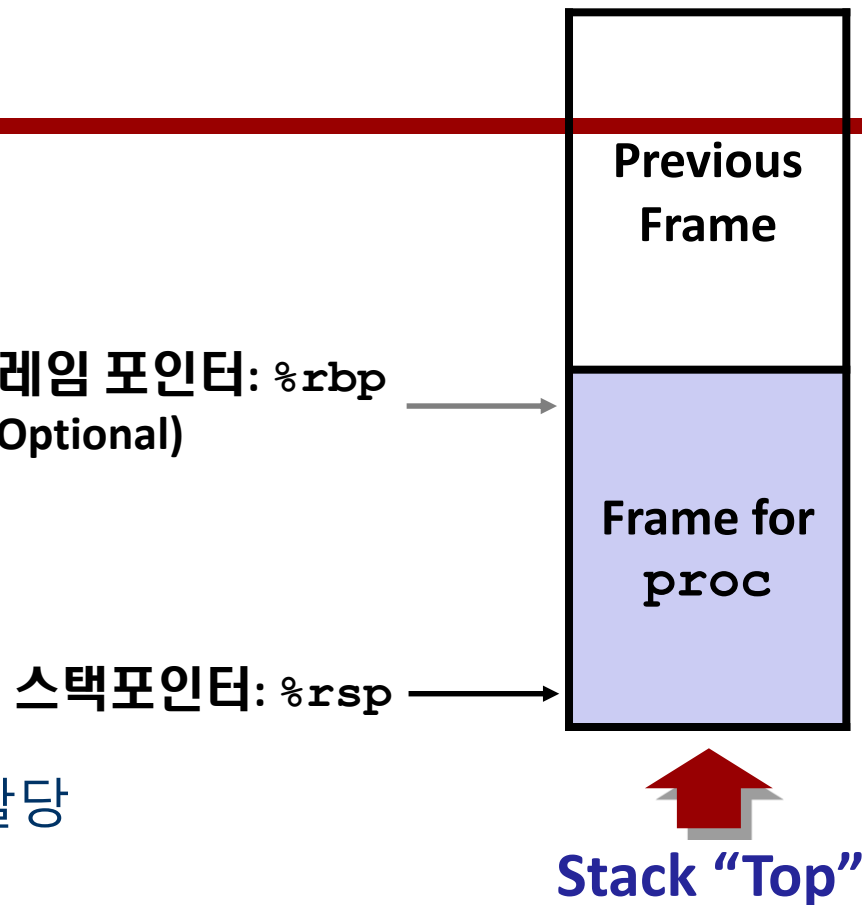
# 스택 프레임

## 내용


- 리턴정보
  - 로컬 스토리지(필요한 경우)
  - 임시공간 (필요한 경우)
- 프레임 포인터: `%rbp`  
(Optional)

## 운영

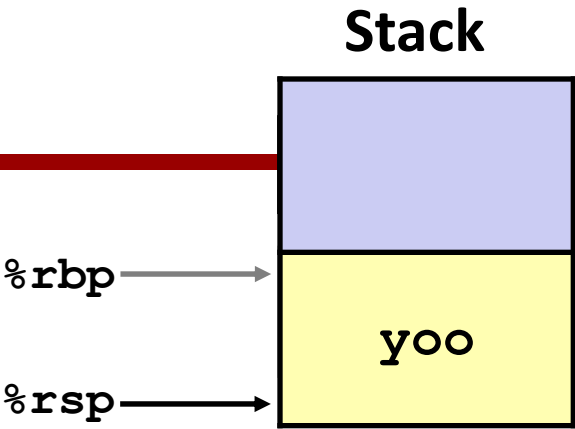
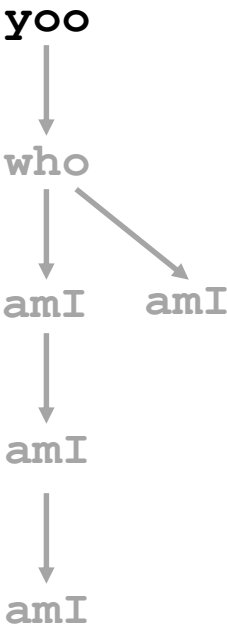
- 프로시저에 들어갈 때 프레임 할당
  - "Set-up" 코드에서 할당
  - `call` 명령으로 푸시가 발생
- 리턴시에 프레임 반환
  - "Finish" 코드에서 반환
  - `ret` 명령으로 팝 발생



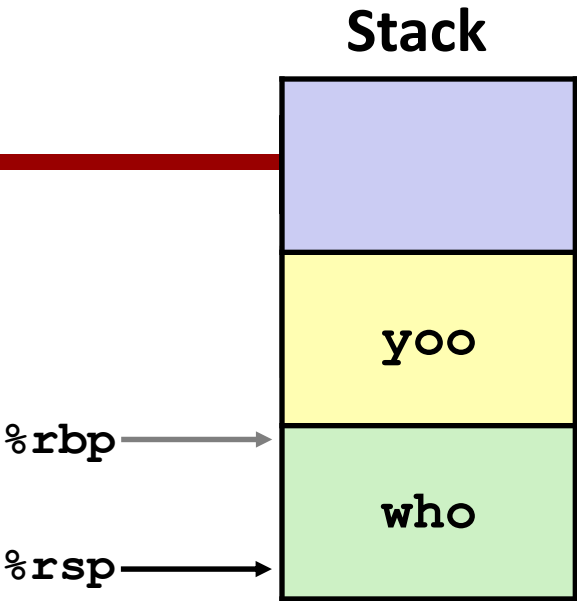
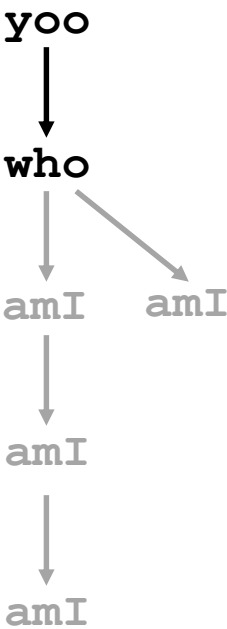
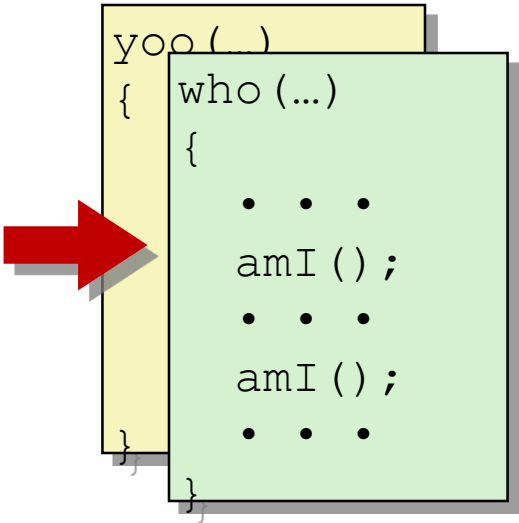
# 예제



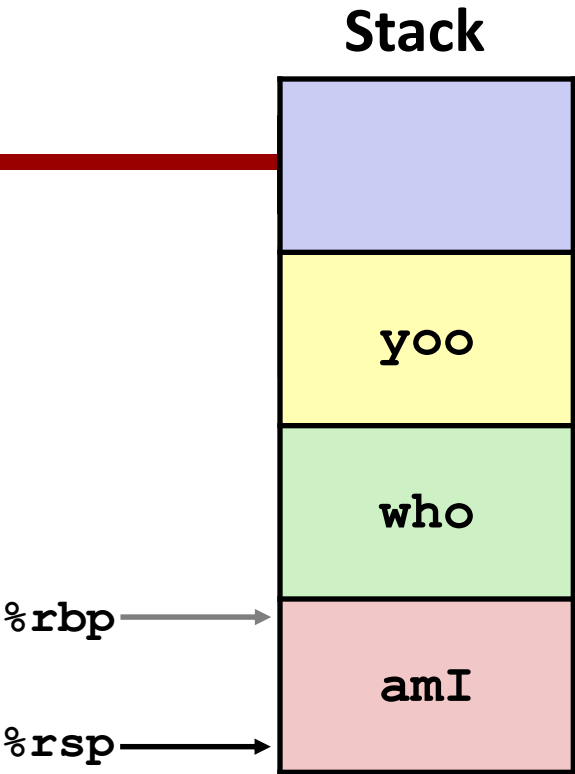
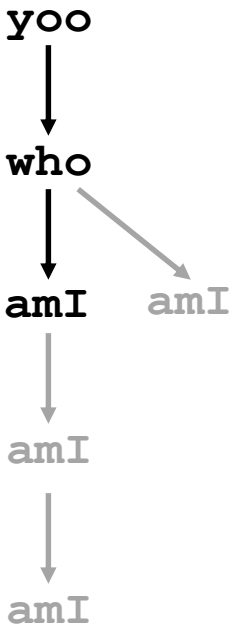
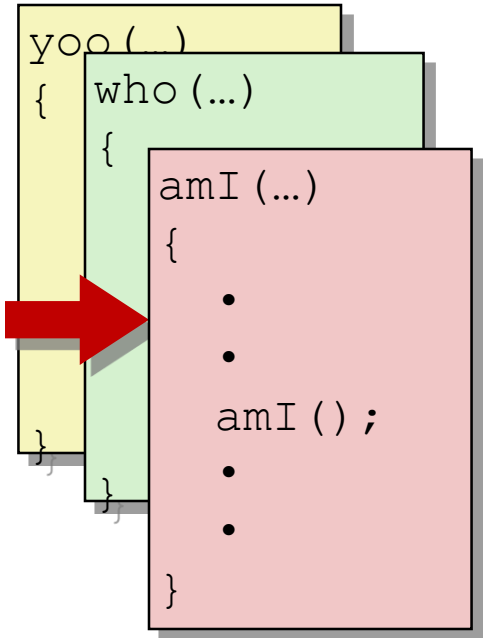
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```



# 예제

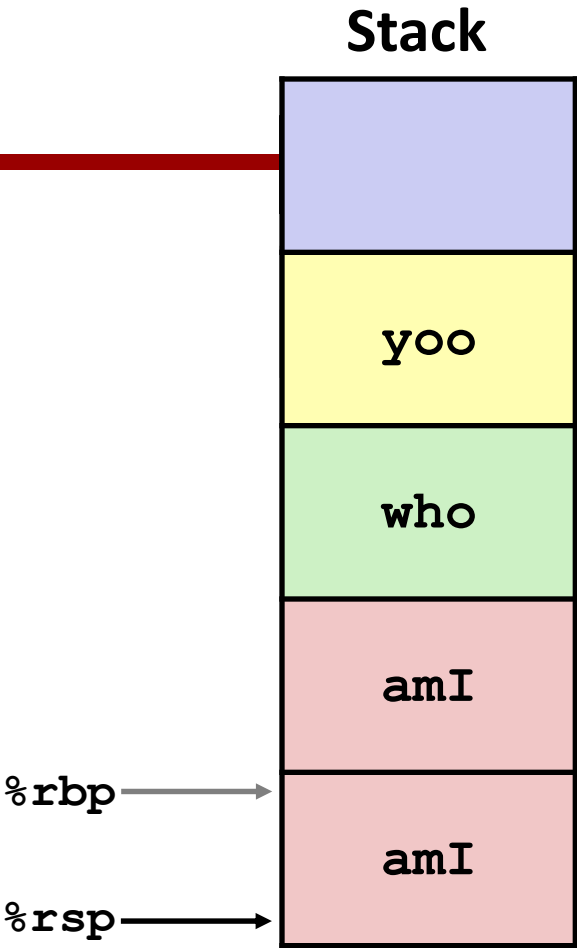
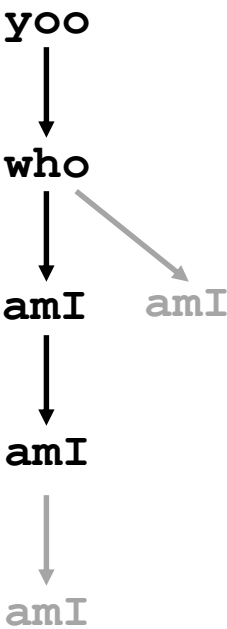
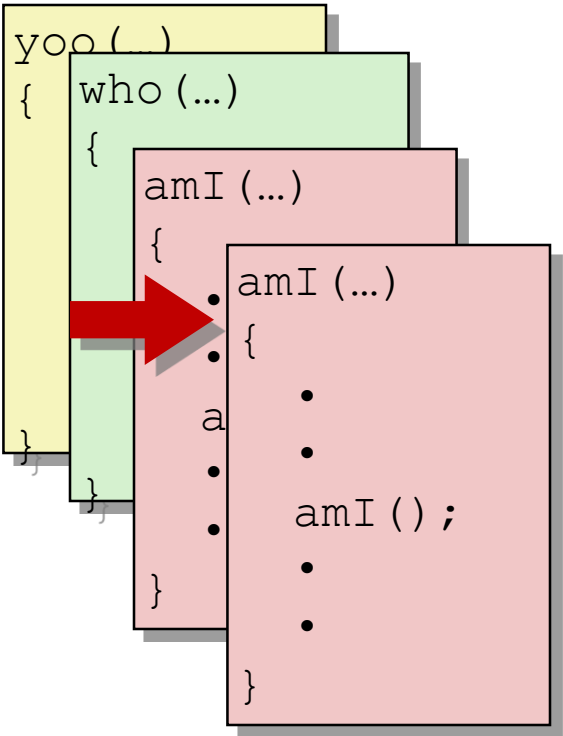


# 예제

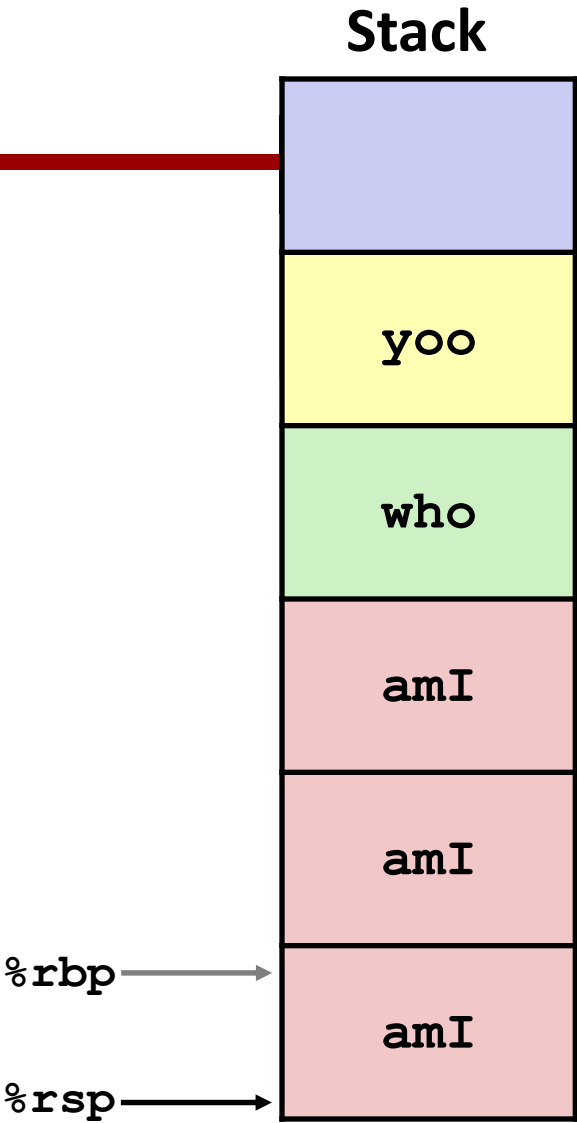
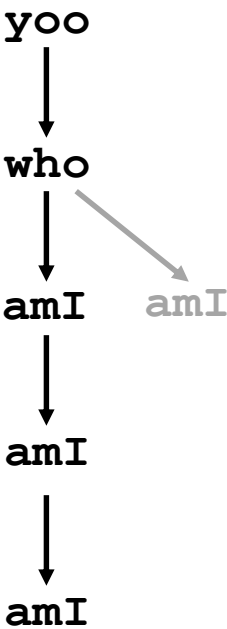
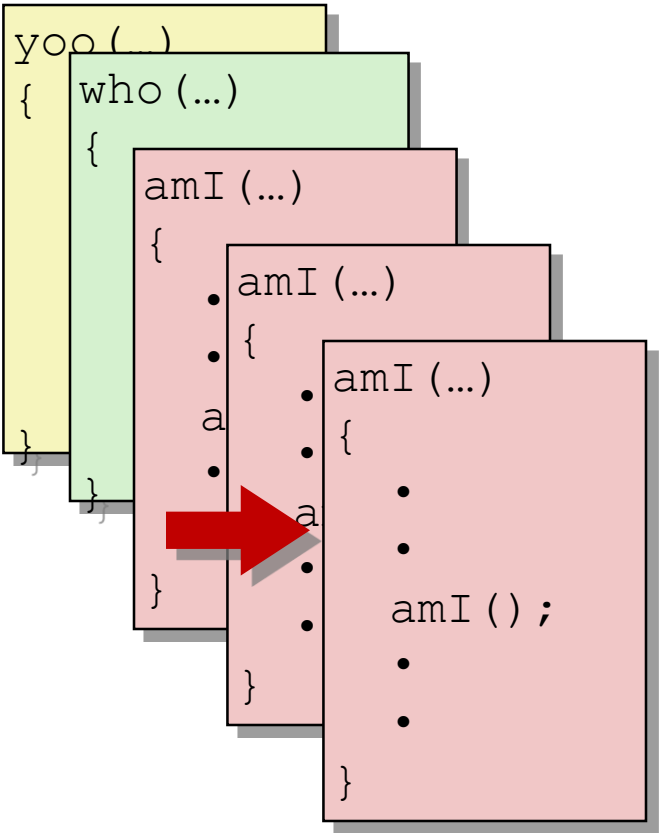




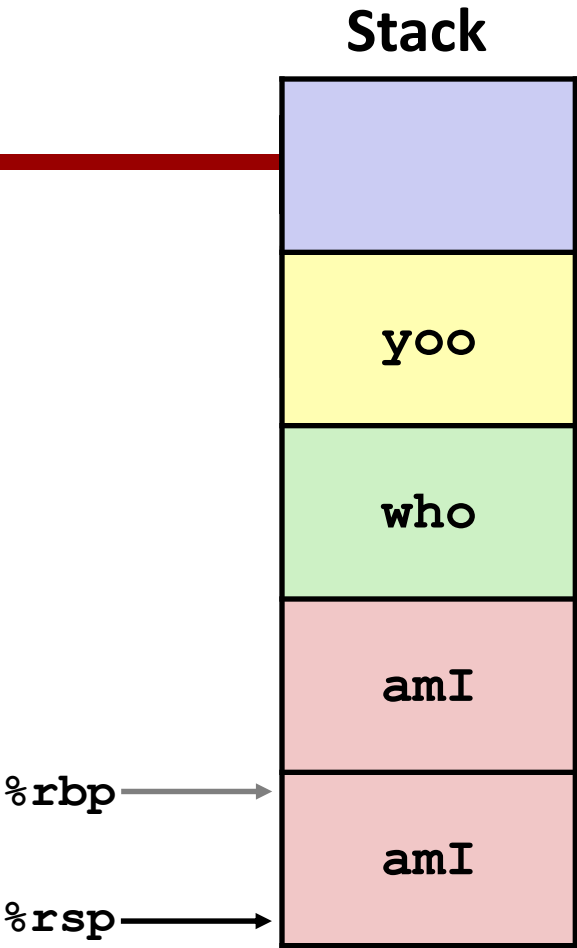
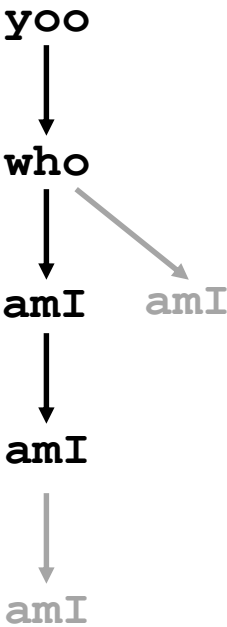
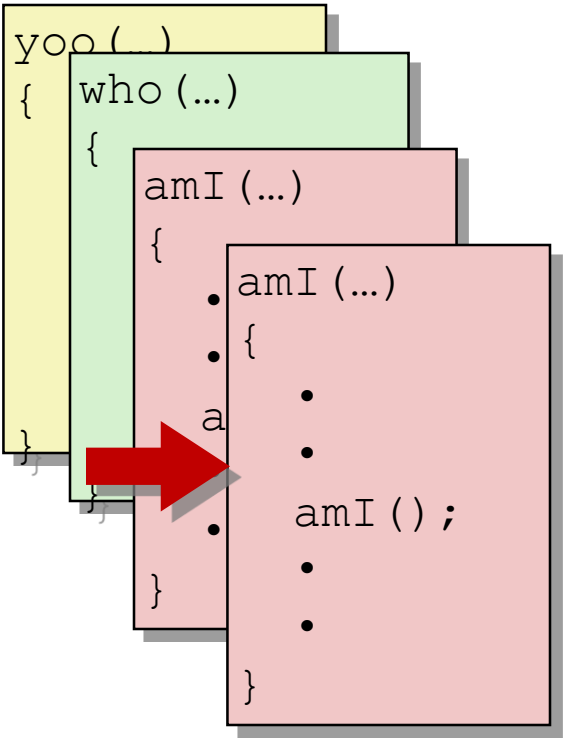
# 예제



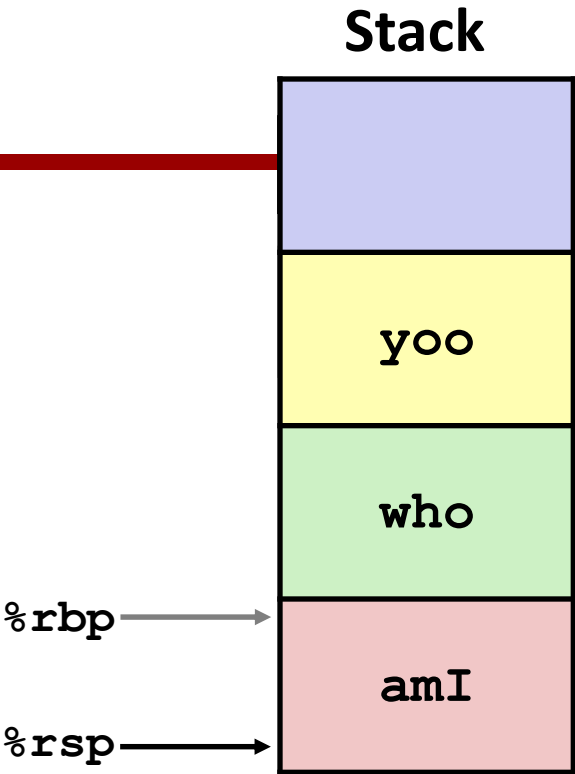
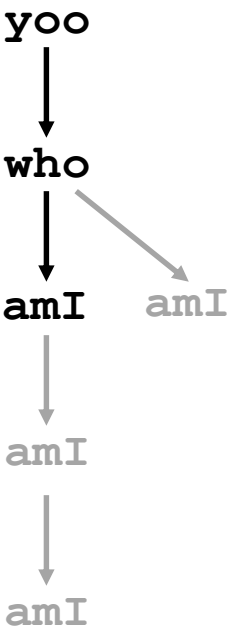
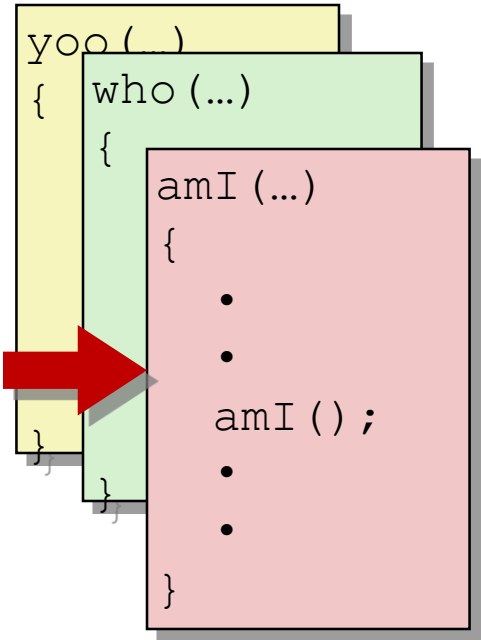
# 예제



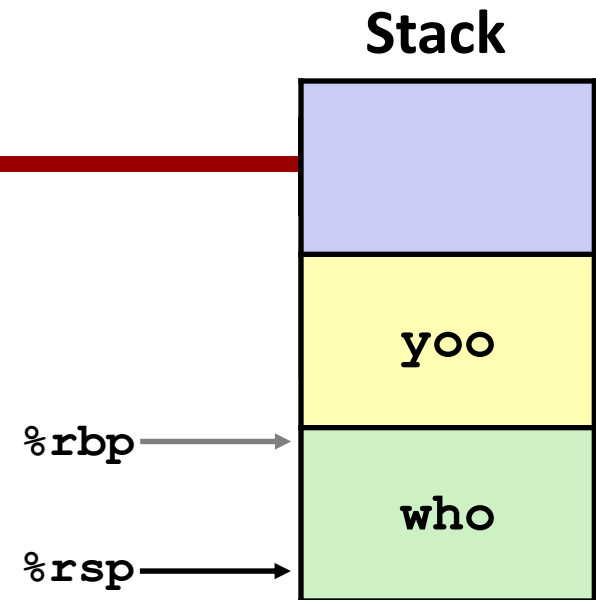
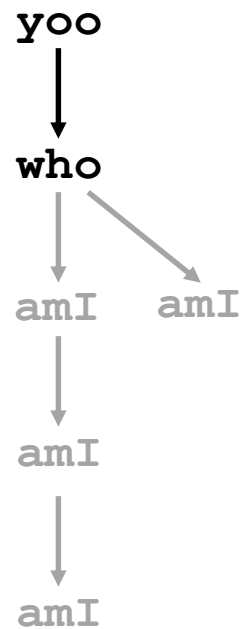
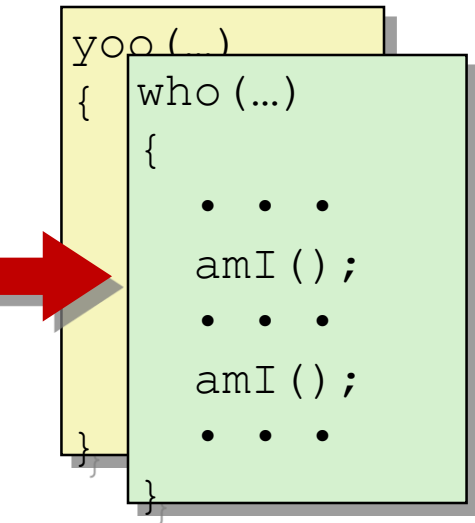
# 예제



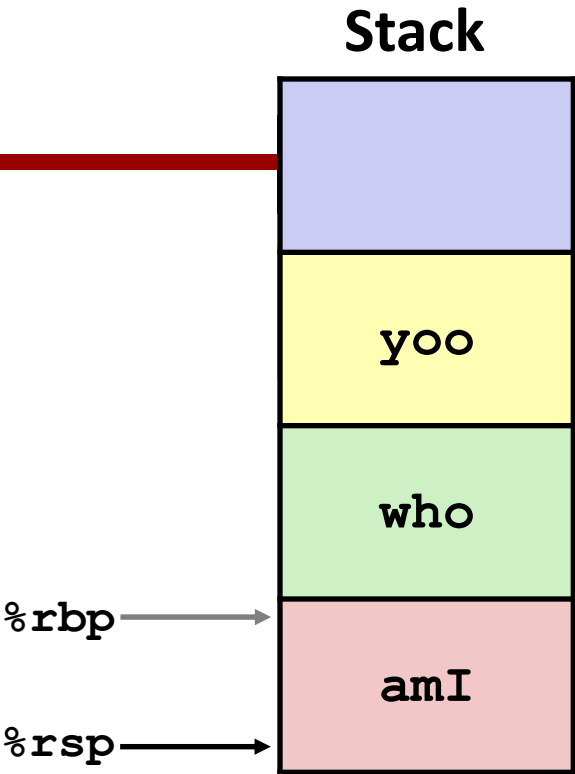
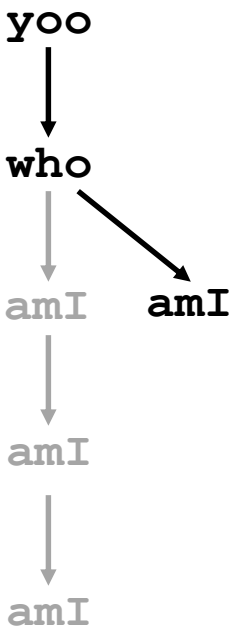
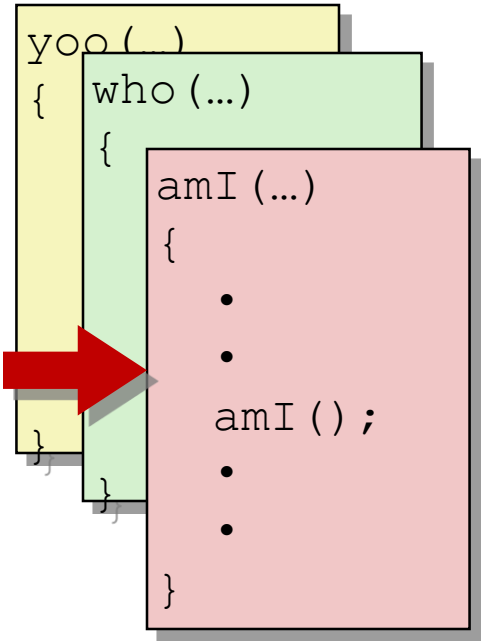
# 예제



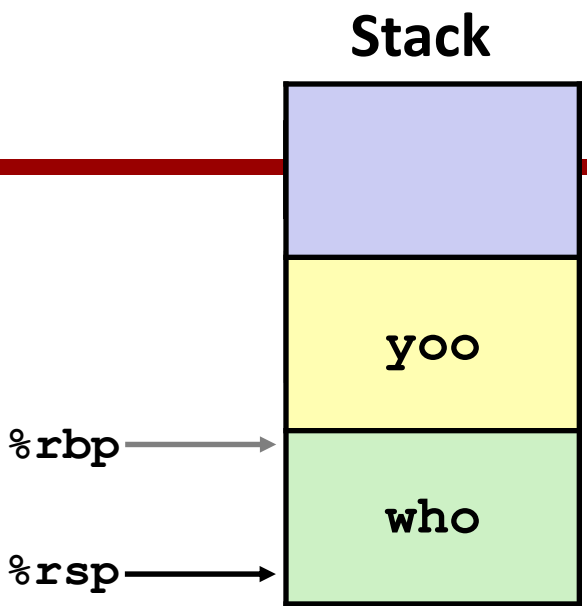
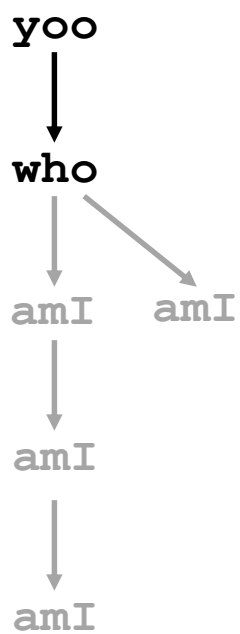
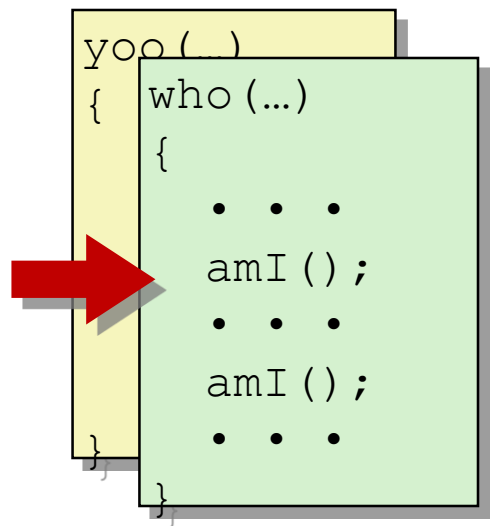
# 예제



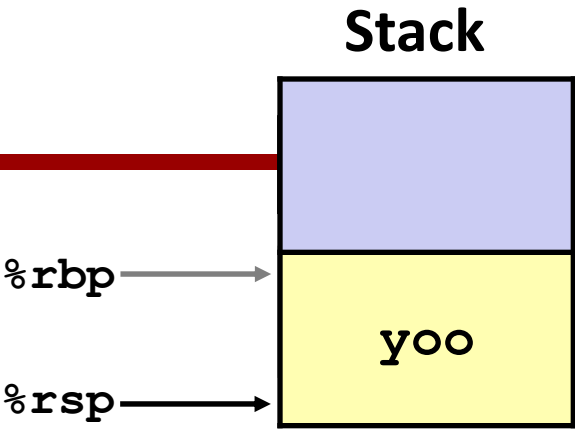
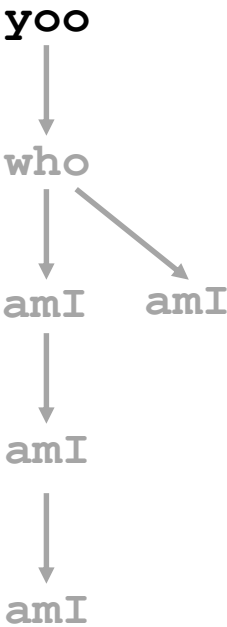
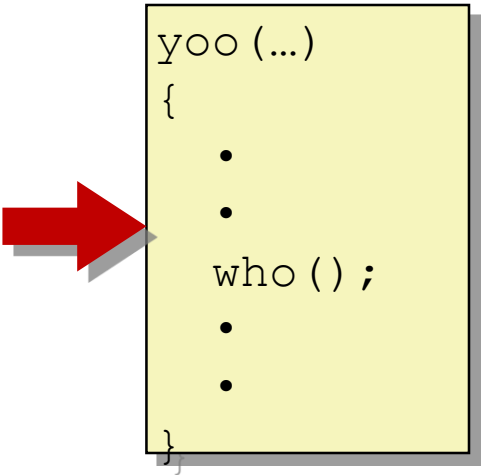
# 예제



# 예제



# 예제





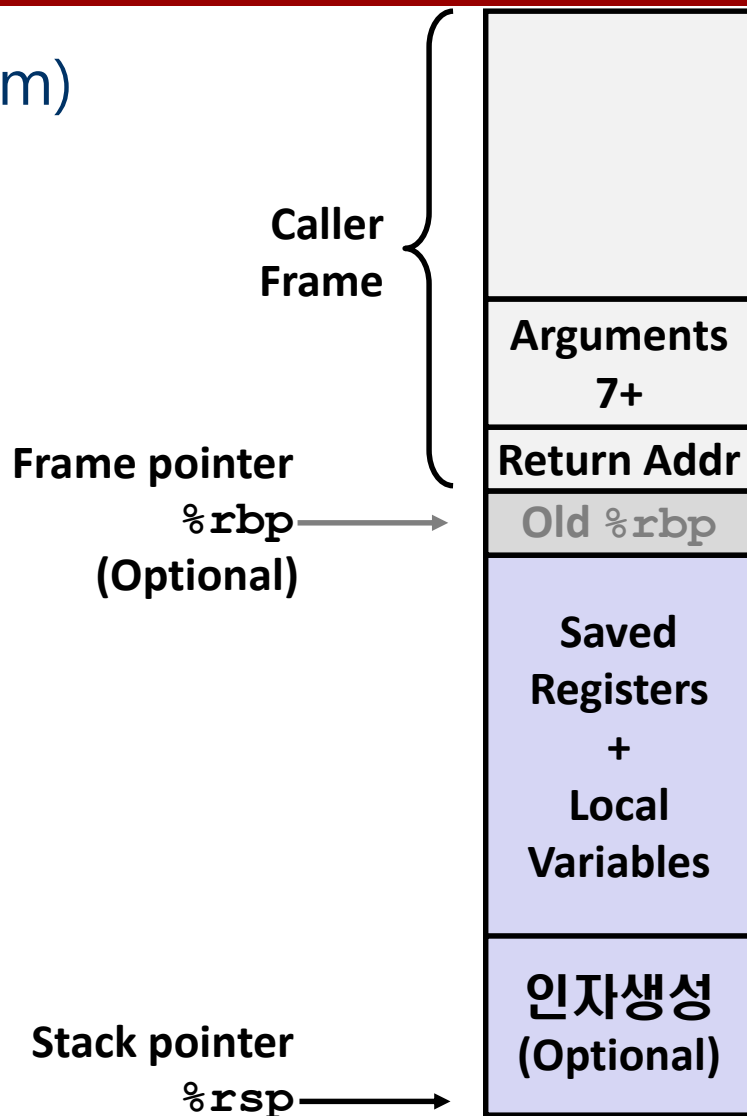
# x86-64/Linux 스택 프레임

## 현재 스택 프레임 ("Top" to Bottom)

- "인자 생성"  
호출하는 함수를 위한 매개변수 값
- 로컬변수  
레지스터에 넣을 수 없을 때
- 보관된 레지스터 컨텍스트
- 이전 프레임 포인터 (optional)

## 호출자 스택 프레임

- 리턴주소  
→ `call` 명령으로 푸시됨
- 이번 호출을 위한 인자들



## 예제: `incr`

```
long incr(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

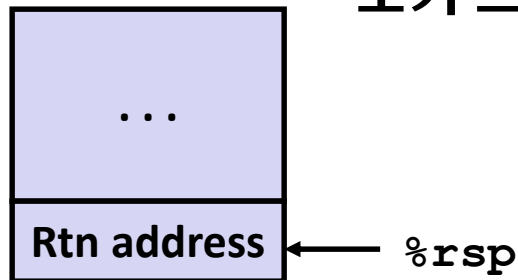
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument <code>p</code>
%rsi	Argument <code>val</code> , <code>y</code>
%rax	<code>x</code> , Return value

# 예제: `incr` #1

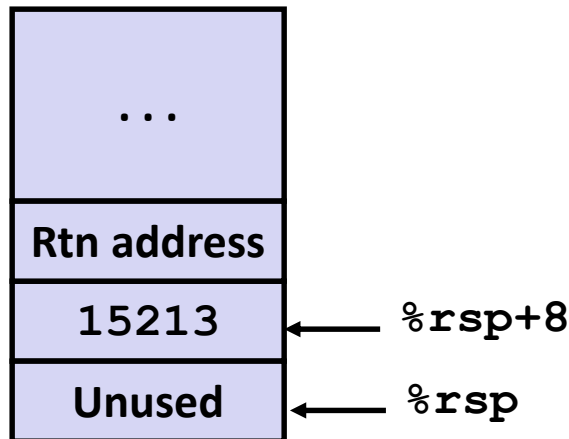
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

초기 스택 구조



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

호출결과 스택구조

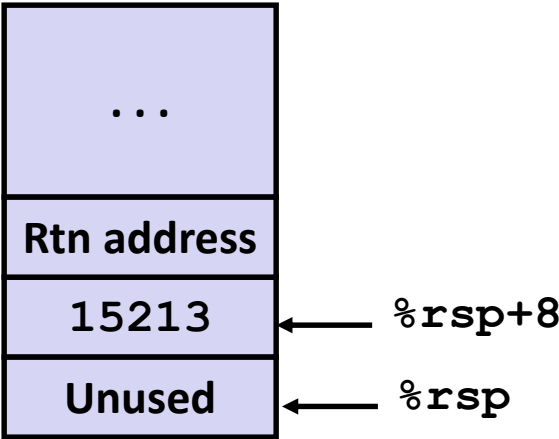


# 예제: `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

스택 구조



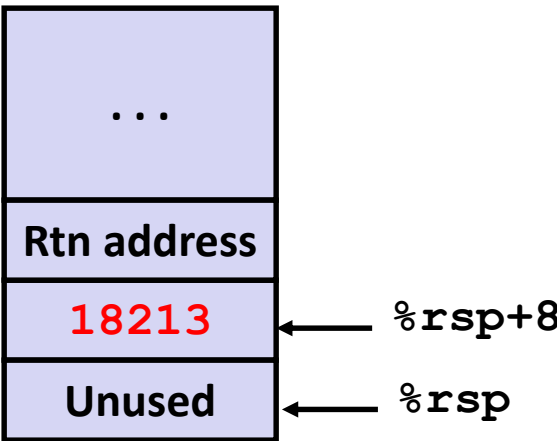
Register	Use(s)
%rdi	&v1
%rsi	3000

# 예제 : `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure

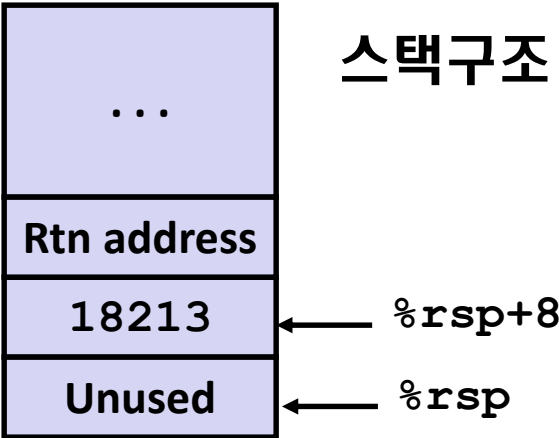


Register	Use(s)
%rdi	&v1
%rsi	3000

# Example: Calling `incr` #4

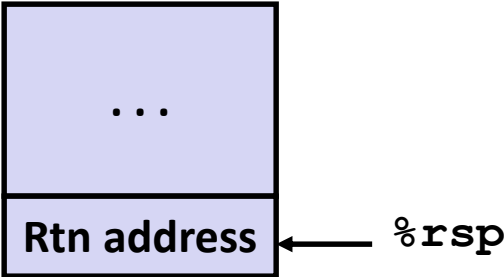
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```



Register	Use(s)
<code>%rax</code>	Return value

Updated Stack Structure

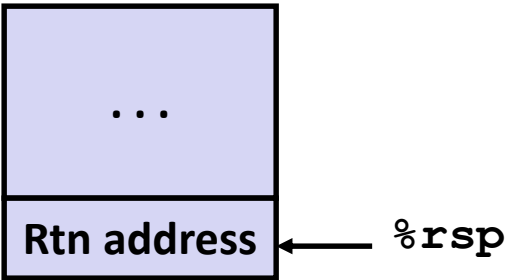


# 예제: `incr` #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

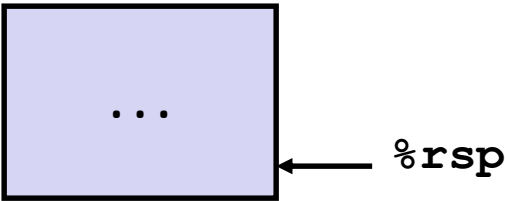
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



# 레지스터 보관 관습

yoo 가 who를 호출할 때

- yoo 는 호출자 *caller*
- who 는 피호출자 *callee*

레지스터는 임시저장공간으로 이용할 수 있는가?

```
yoo:
    . . .
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    . . .
    ret
```

```
who:
    . . .
    subq $18213, %rdx
    . . .
    ret
```

- %rdx 는 who에 의해 지워진다
- 이것은 문제가 될수 있다 → 뭔가 대책이 필요!



# 레지스터 보관 관습

yoo 가 who를 호출할 때

- yoo 는 호출자 *caller*
- who 는 피호출자 *callee*

레지스터는 임시저장공간으로 이용할 수 있는가?

관습

- 호출자 보관 *"Caller Saved"*
  - 호출자가 임시 값들을 자신의 프레임에 호출 전에 보관
- 피호출자 보관 *"Callee Saved"*
  - 피호출자가 임시 값들을 사용하기 전에 자신의 프레임에 보관
  - 피호출자가 호출자로 리턴하기 전에 이들을 복원

# x86-64 Linux 레지스터 용법 #1

## **%rax**

- 리턴값
- 호출자-보관
- 프로시저에서 수정될 수 있음

## **%rdi, ..., %r9**

- 인자값
- 호출자-보관 caller-saved
- 프로시저에서 수정될 수 있음

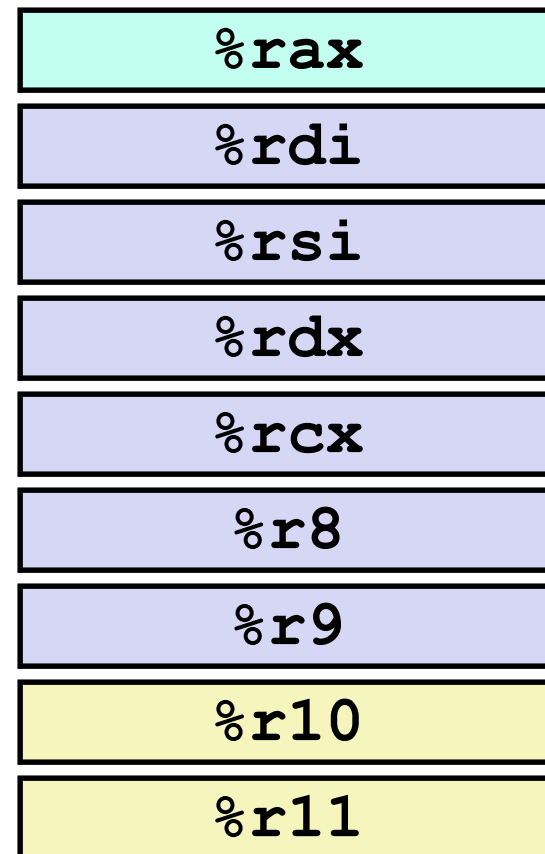
## **%r10, %r11**

- 호출자-보관 Caller-saved
- 프로시저에서 수정될 수 있음

Return value

Arguments

Caller-saved  
temporaries



# x86-64 Linux 레지스터 용법 #2

**%rbx, %r12, %r13, %r14**

- 피호출자-보관 Callee-saved
- 피호출자가 보관하고 복원해야 함

**%rbp**

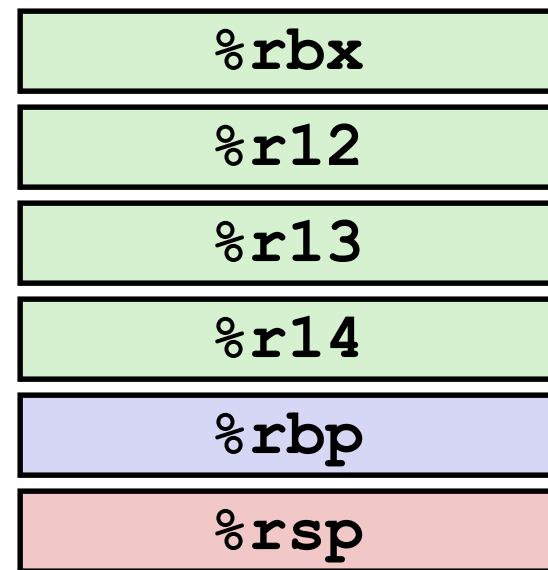
- 피호출자-보관
- 피호출자가 보관하고 복원해야 함
- 프레임포인터로 사용될 수 있음

**%rsp**

- 일종의 피호출자 보관형태
- 프로시저 리턴시에 원래값으로 복원

Callee-saved  
Temporaries

Special



# 오늘의 내용

---

## 프로시저

- 스택의 구조
- 호출 관습
  - ▶ 제어의 전달
  - ▶ 데이터의 전달
  - ▶ 지역 데이터의 관리
- 재귀실행

# 재귀함수

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

# 재귀함수 종료조건

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

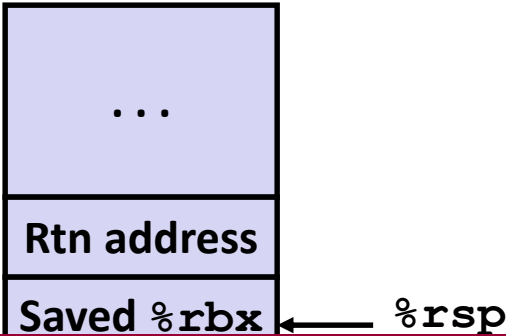
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

# 재귀함수 레지스터 보관

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



# 재귀함수 호출 설정

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved



# 재귀함수 호출

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

# 재귀함수 결과값

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

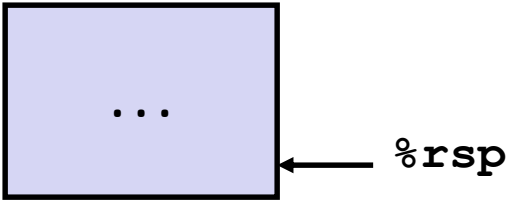
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

# 재귀함수 완료

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi # (by 1)
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value



# x86-64 프로시저 요약

## 주요 개념

- 스택은 프로시저 콜/리턴에 적합한 자료구조다

재귀함수는 일반 호출 방식으로 처리된다

포인터는 값들의 주소다

