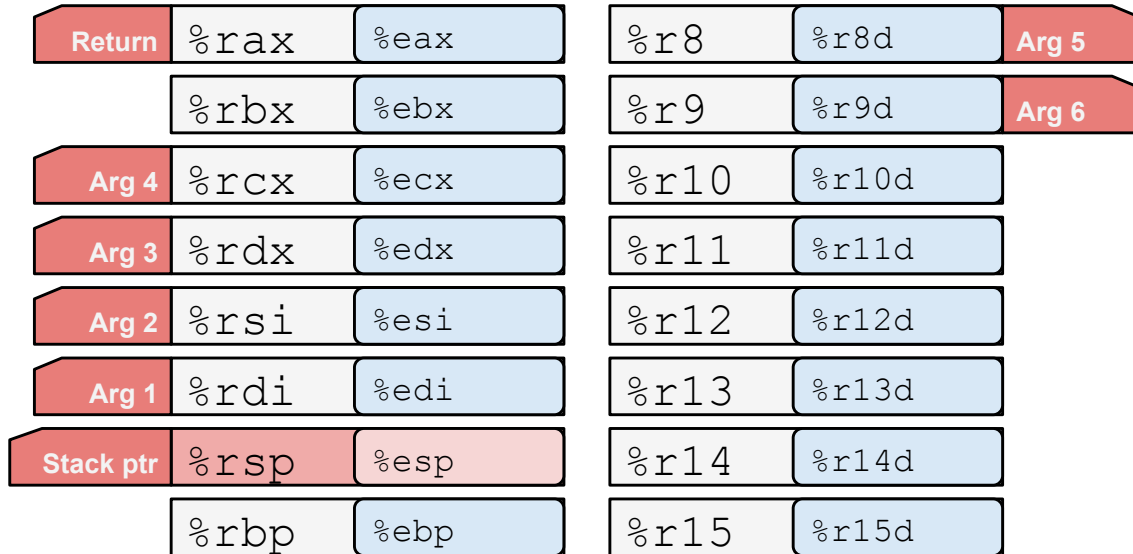
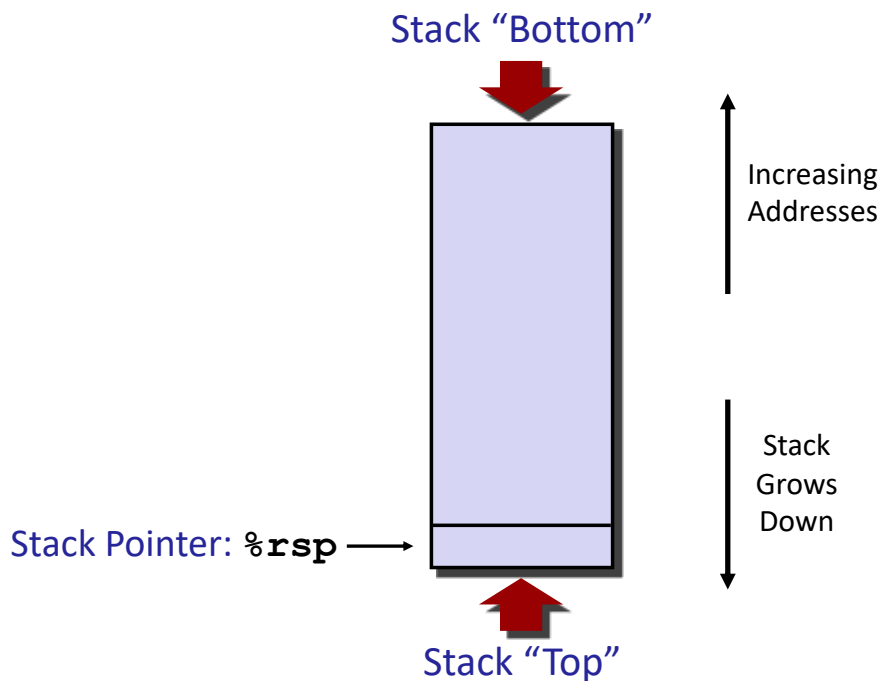


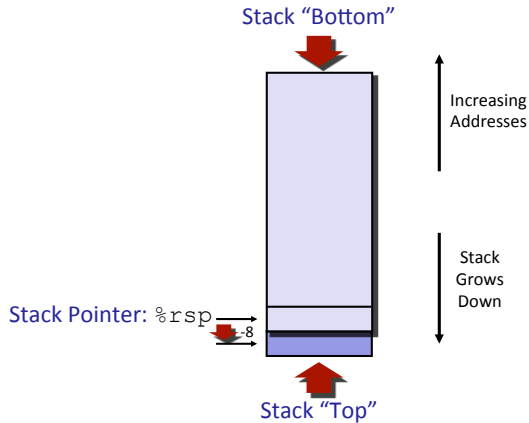
# Machine Code: Procedures



# Call Stack

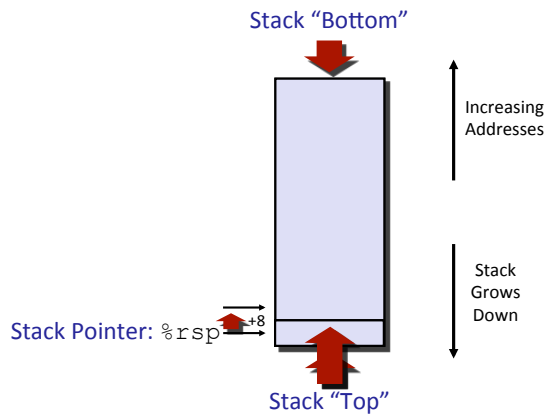


# Stack Operations



## pushq Src

1. Decrement `%rsp` by 8
2. Write `Src` to `(%rsp)`



## popq Dest

1. Read `(%rsp)` to `Dest`
2. Increment `%rsp` by 8

# Procedure Call Example

```
long x = add(3, 5);  
doSomething(x);
```

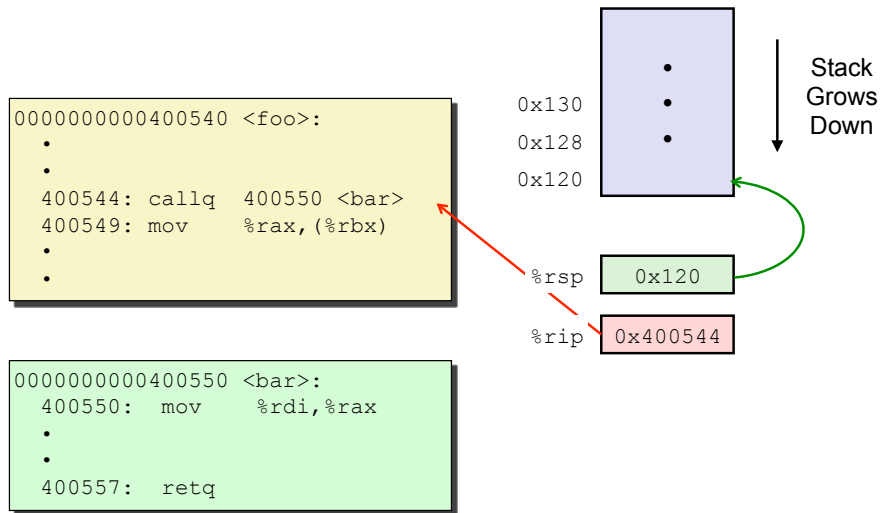
```
long add(long x, long y) {  
    return x + y;  
}
```

```
movq $3, %rdi  
movq $5, %rsi  
callq add push retaddr  
movq %rax, %rdi  
callq doSomething
```

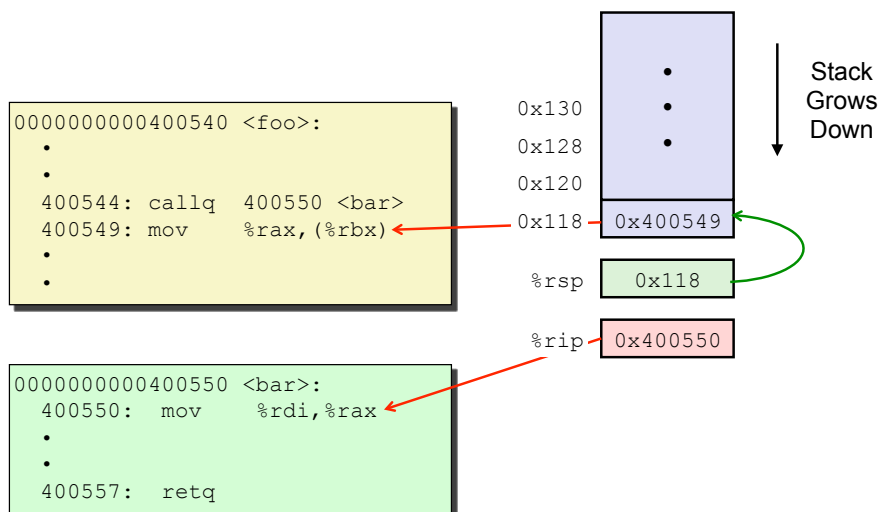
```
add:  
movq %rdi, %rax  
addq %rsi, %rax  
retq pop retaddr & jump
```

**return address**

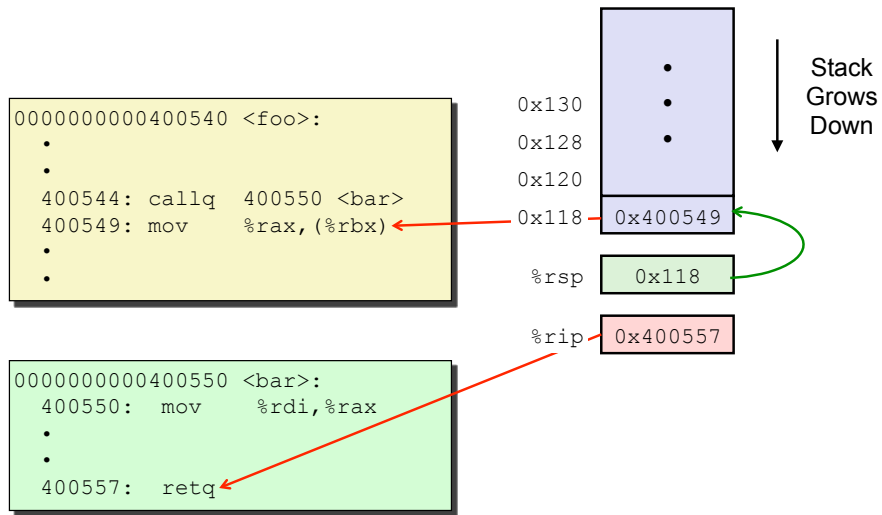
# Procedure Call Return Addresses (1)



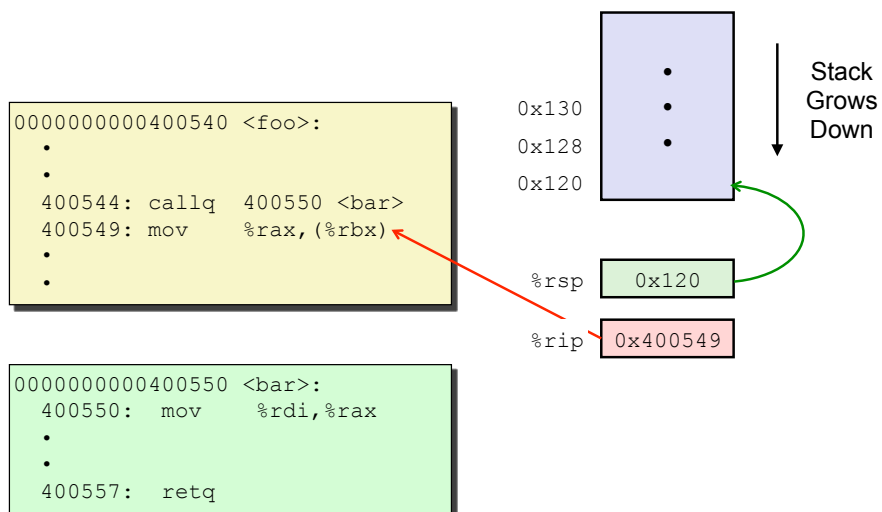
# Procedure Call Return Addresses (2)



# Procedure Call Return Addresses (3)



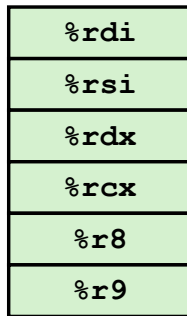
# Procedure Call Return Addresses (4)



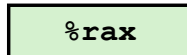
# Passing Data

## Registers

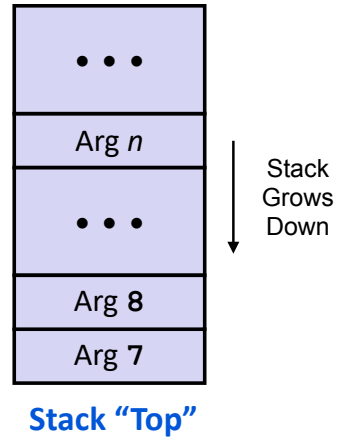
### ■ First 6 arguments



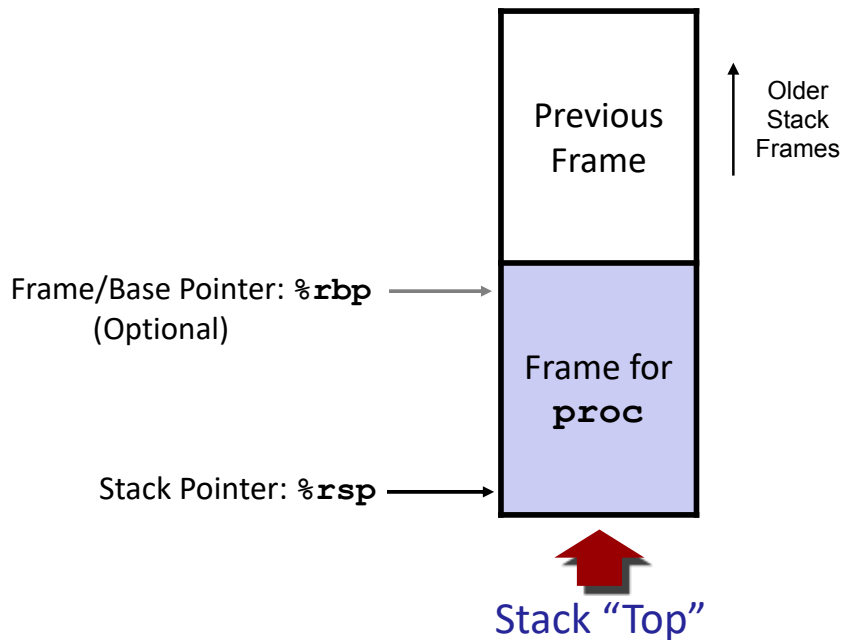
### ■ Return value



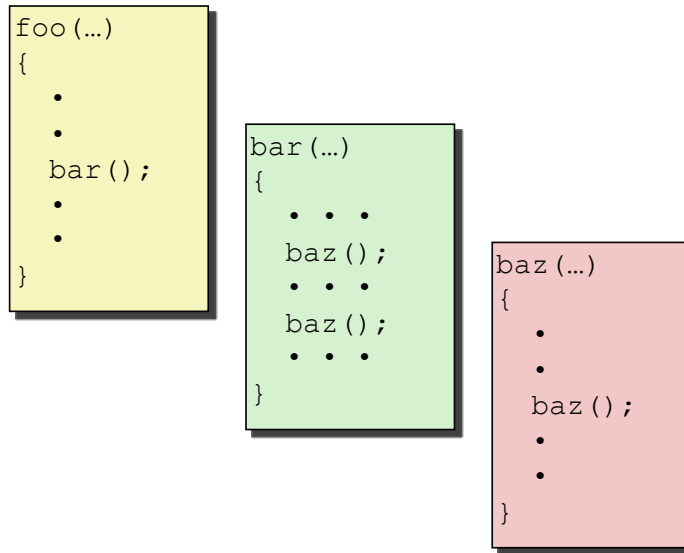
## Stack



# Stack Frames

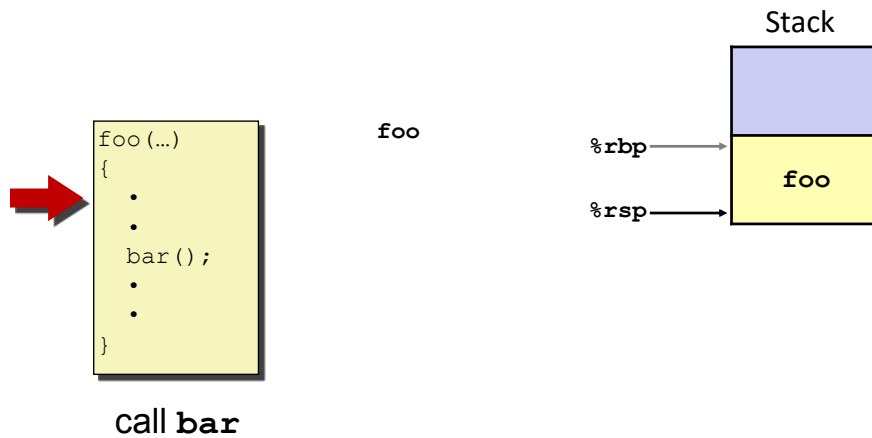


# Call Chain Example

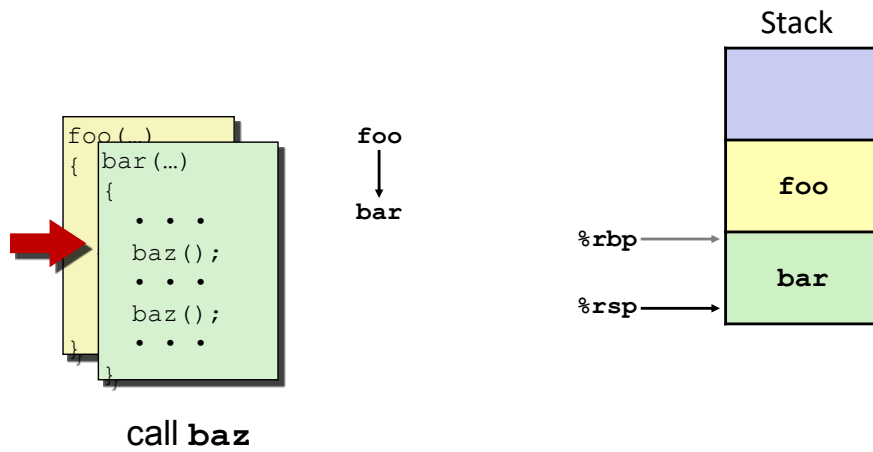


Procedure **baz ()** is recursive

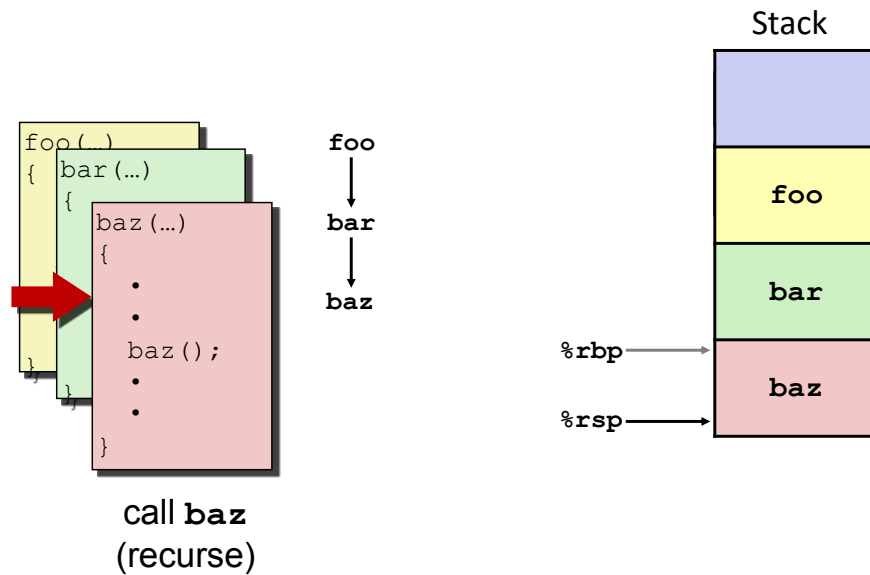
# Stack Frame Allocation (1)



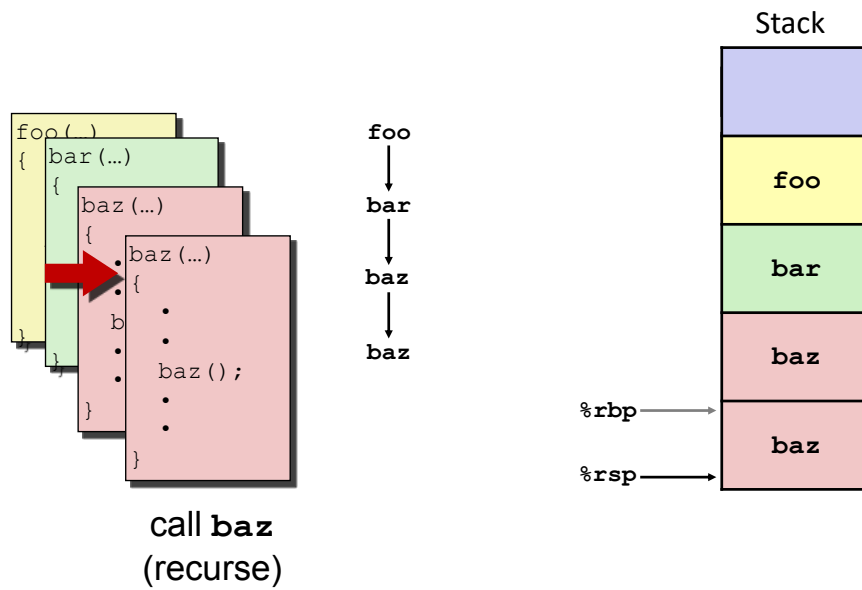
# Stack Frame Allocation (2)



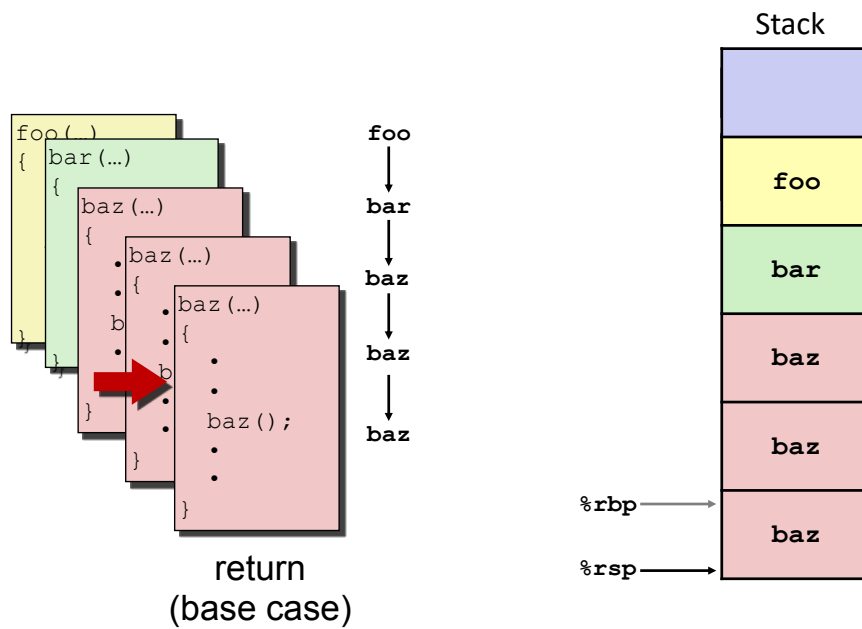
# Stack Frame Allocation (3)



# Stack Frame Allocation (4)

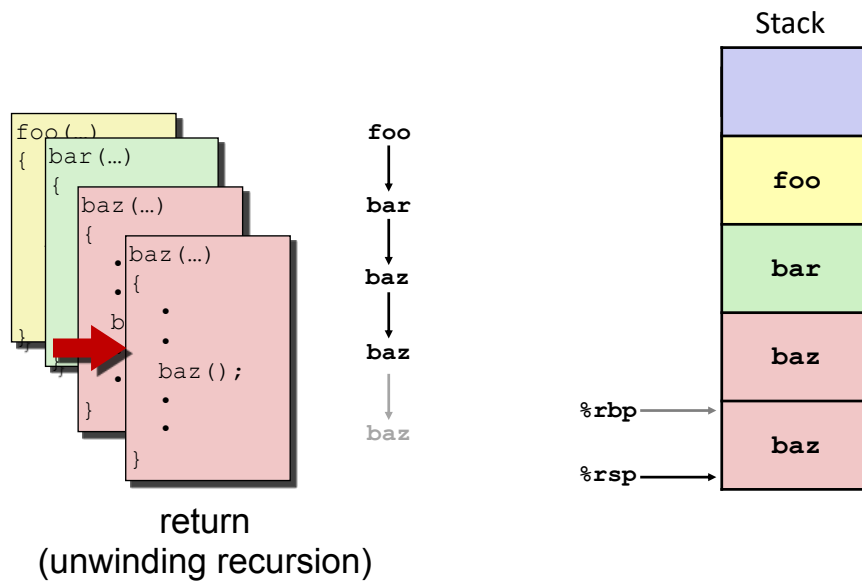


# Stack Frame Allocation (5)

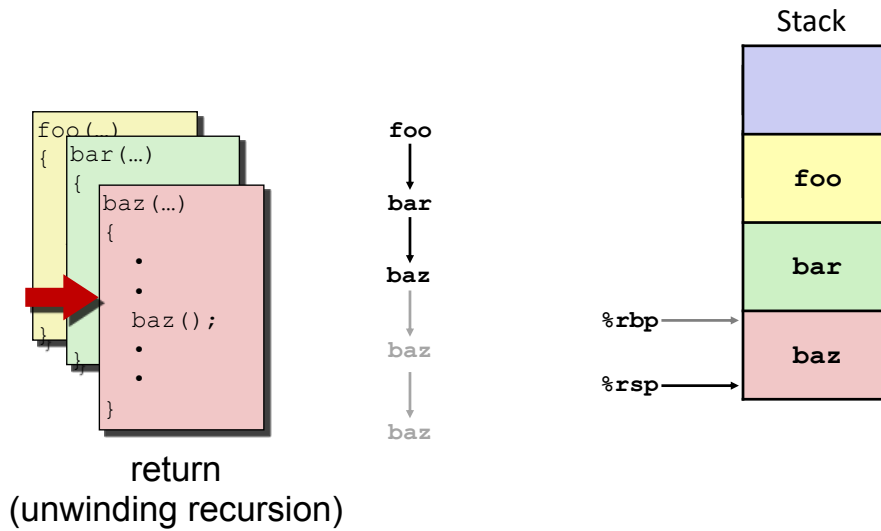




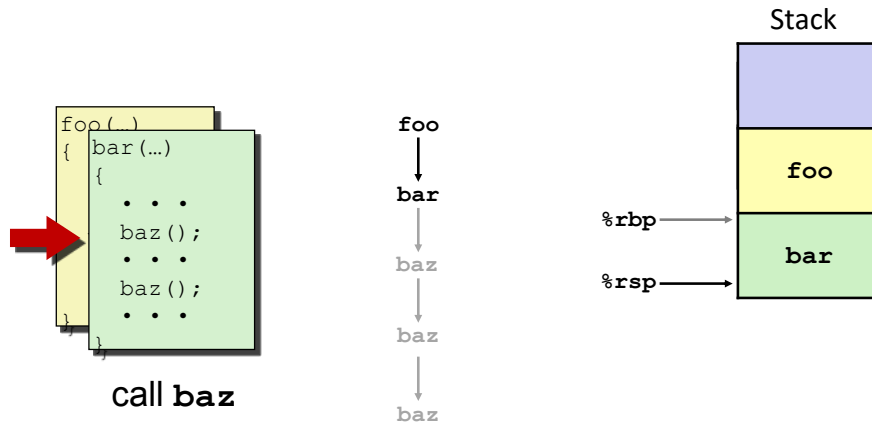
# Stack Frame Allocation (6)



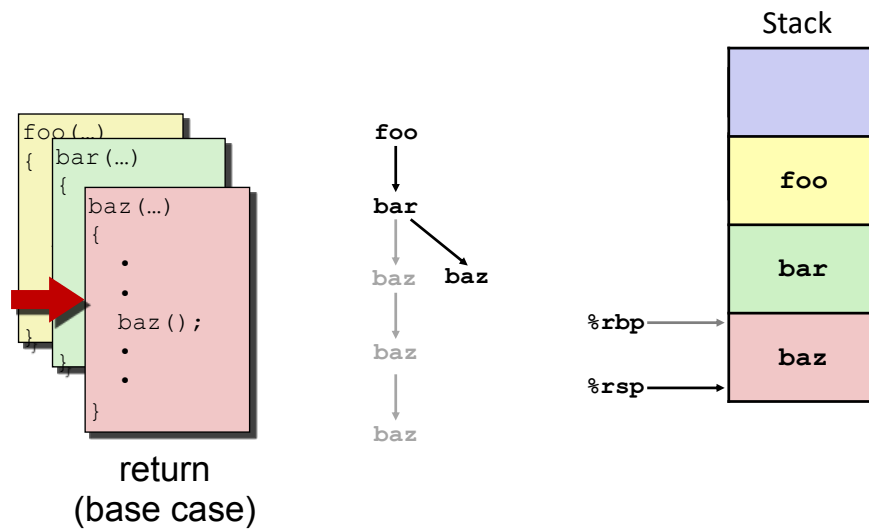
# Stack Frame Allocation (7)



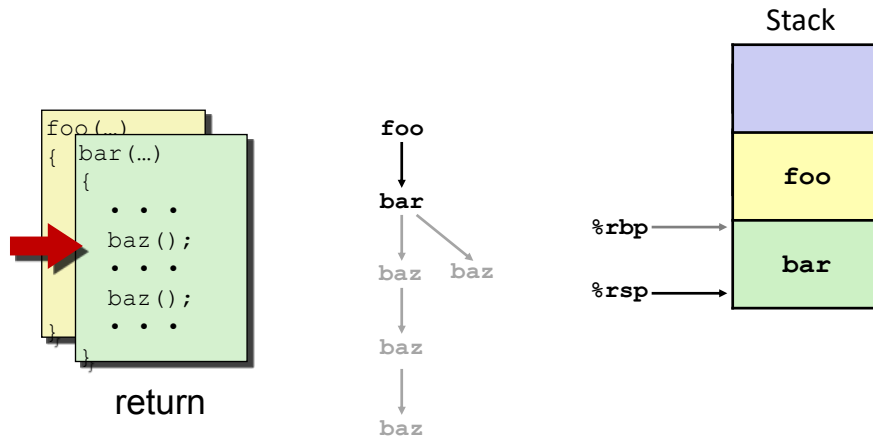
# Stack Frame Allocation (8)



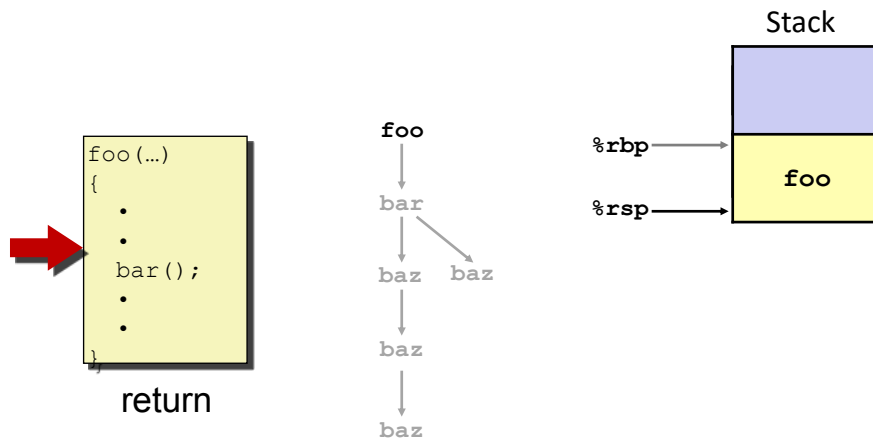
# Stack Frame Allocation (9)



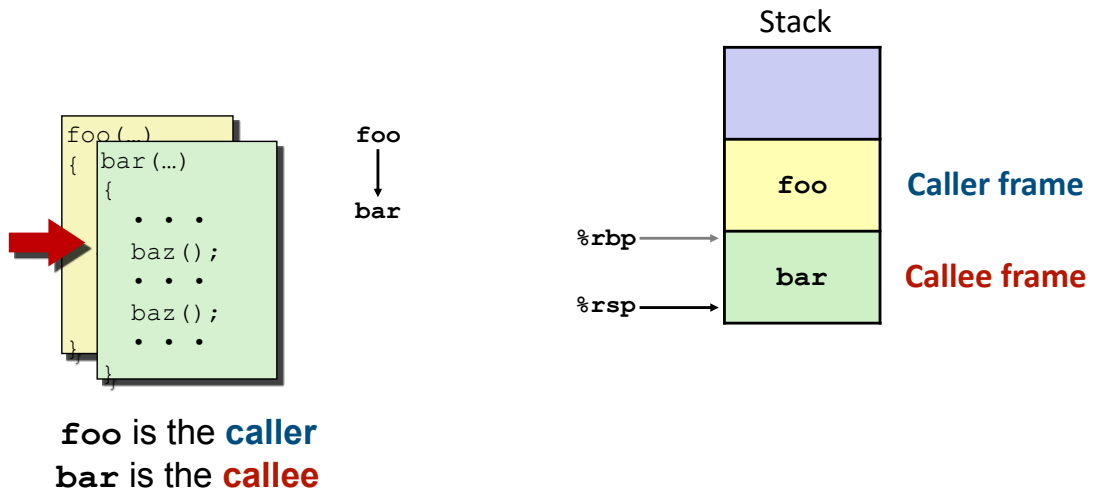
# Stack Frame Allocation (10)



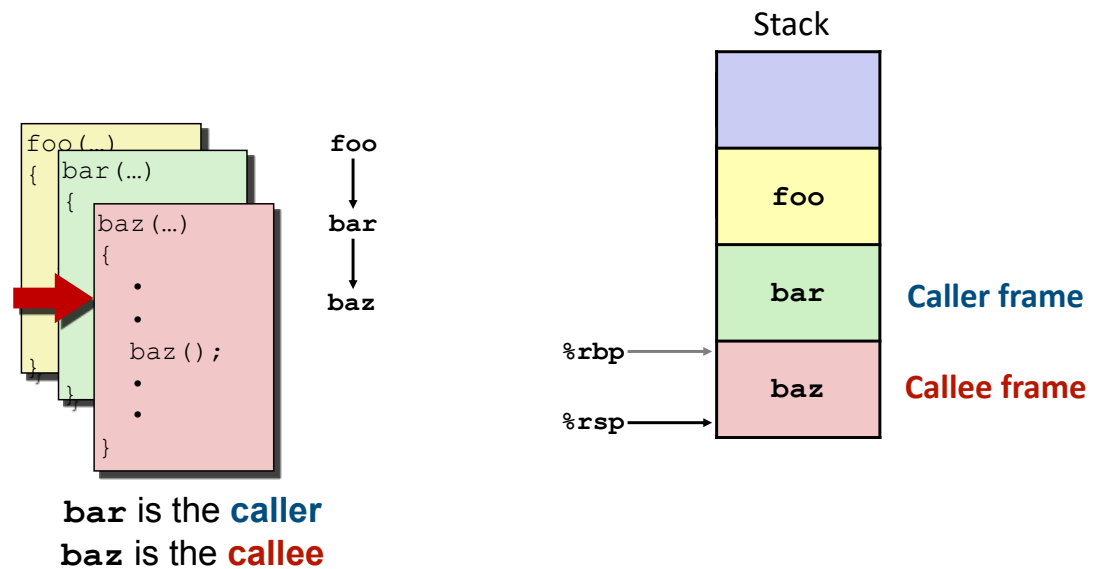
# Stack Frame Allocation (11)



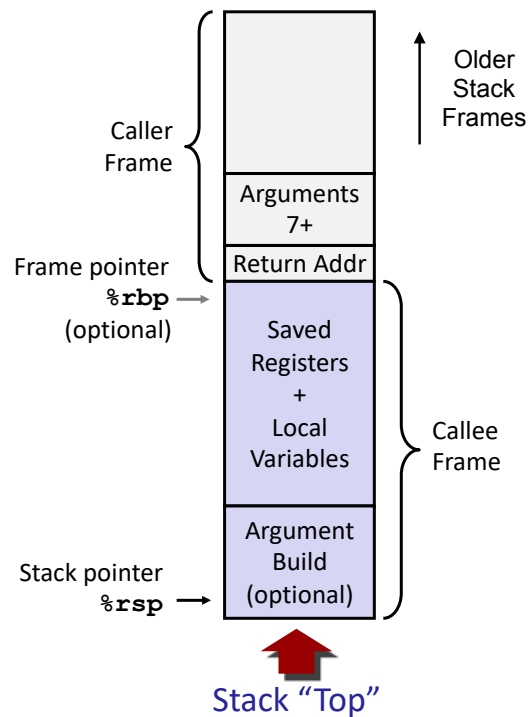
# Caller vs. Callee (1)



# Caller vs. Callee (2)



# Stack Frame Components



## Using the Stack (1)

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

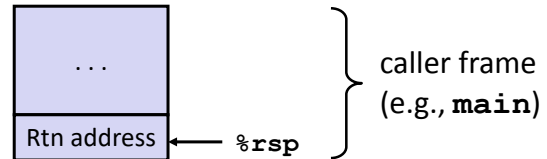
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

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

# Using the Stack (2)

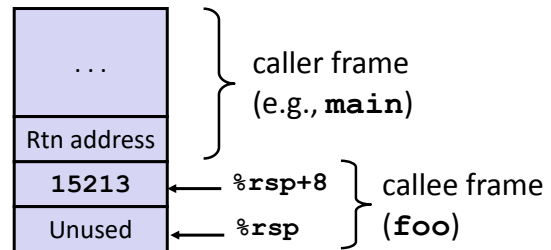
```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure



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

Resulting Stack Structure

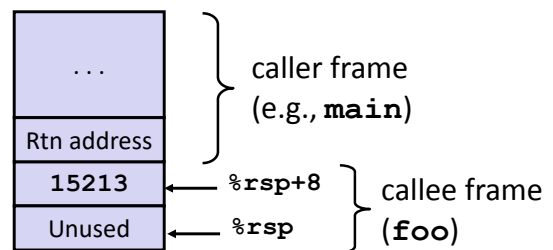


# Using the Stack (3)

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

```
foo:
    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

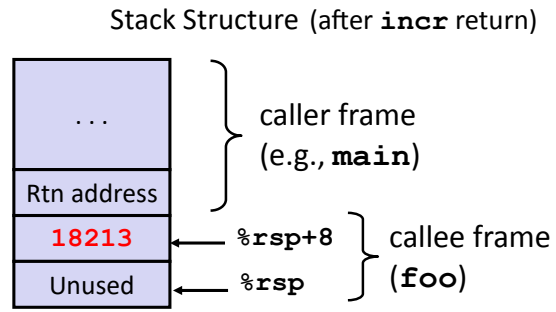


Register	Use(s)
%rdi	&v1
%rsi	3000

# Using the Stack (4)

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

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



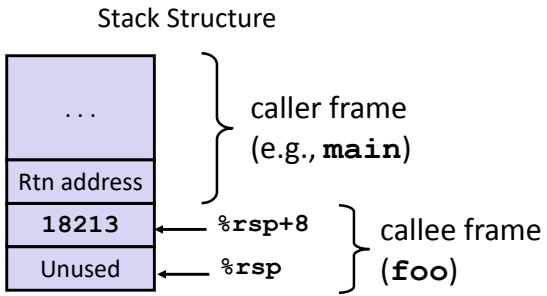
Register	Use(s)
<code>%rdi</code>	<code>&amp;v1</code>
<code>%rsi</code>	3000

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

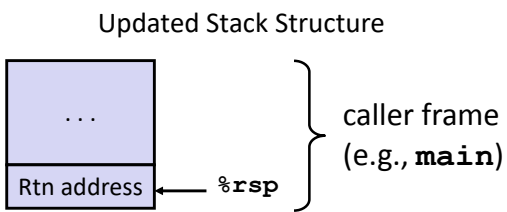
# Using the Stack (5)

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

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



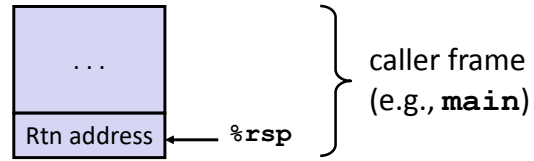
Register	Use(s)
<code>%rax</code>	Return value



# Using the Stack (6)

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

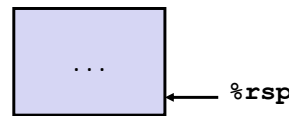
Updated Stack Structure



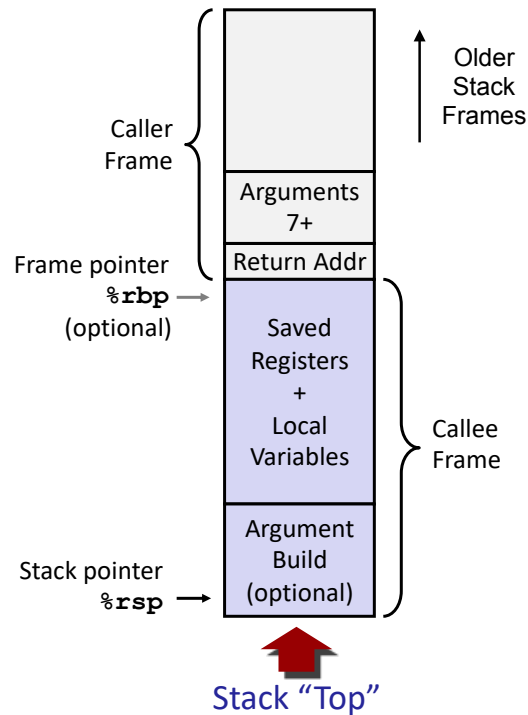
```
foo:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call   incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Register	Use(s)
%rax	Return value

Final Stack Structure



# Stack Frame Components





# Saved Registers

```
alice:
  movq X, %reg    # compute X
  callq bob
  movq %reg, ... # use X
```

← Could overwrite %reg!

## Caller-Saved %reg

```
alice:
  movq X, %reg # compute X
  pushq %reg  # save X
  callq bob   # might change X
  popq %reg   # restore X
  movq %reg, ... # use X
```

## Callee-Saved %reg

```
alice:
  pushq %reg # save caller's
  movq X, %reg # compute X
  callq bob   # preserves X
  movq %reg, ... # use X
  popq %reg   # restore caller
```

# Caller-Saved vs. Callee-Saved (1)

```
alice:
  movq X, %reg # compute X
  callq bob
  ... # don't need X again
```

## Caller-Saved %reg

```
alice:
  movq X, %reg # compute X
  callq bob   # might change X
  ... # don't need X again
```



Skip save/restore!

## Callee-Saved %reg

```
alice:
  pushq %reg # save caller's
  movq X, %reg # compute X
  callq bob   # preserves X
  ...
  popq %reg   # restore caller
```

# Caller-Saved vs. Callee-Saved (2)

```
alice:
  movq X, %reg # compute X
  callq bob
  movq %reg, ... # use X
  callq charlie
  movq %reg, ... # use X
```

## Caller-Saved %reg

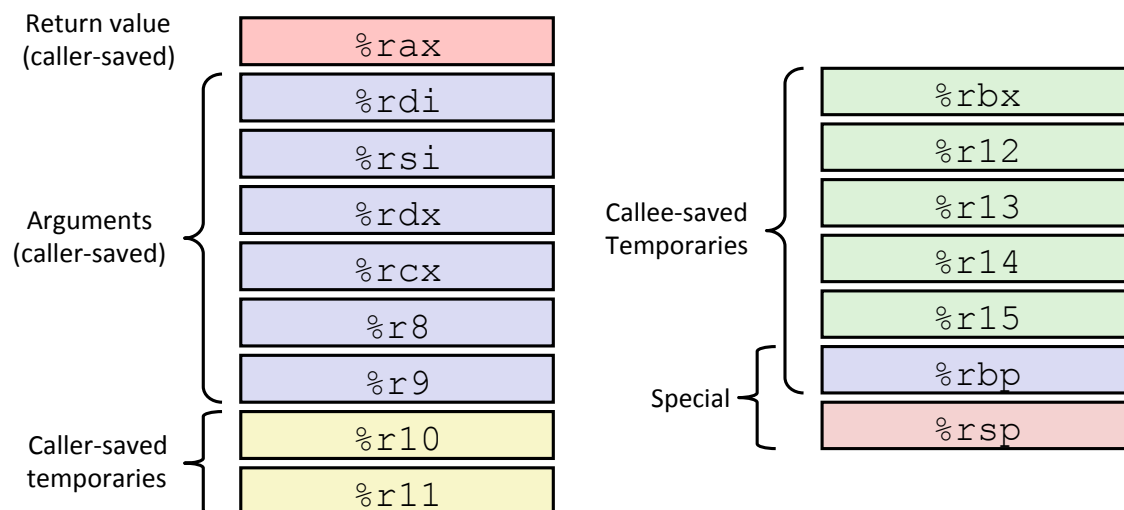
```
alice:
  movq X, %reg # compute X
  pushq %reg # save X
  callq bob # might change X
  popq %reg # restore X
  movq %reg, ... # use X
  pushq %reg # save X
  callq charlie # might change
  popq %reg # restore X
  movq %reg, ... # use X
```

## Callee-Saved %reg

```
alice:
  pushq %reg # save caller's
  movq X, %reg # compute
  callq bob # preserves X
  movq %reg, ... # use X
  callq charlie # preserves X
  movq %reg, ... # use X
  popq %reg # restore caller
```

← Only one save/restore!

# Register Saving Conventions

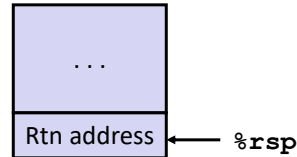


# Callee-Saved Example (1)

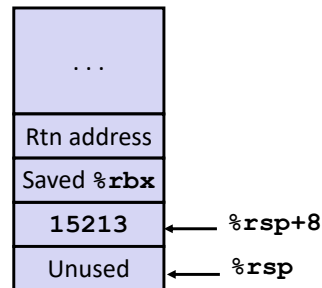
```
long foo2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
foo2:  
    pushq    %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```

Initial Stack Structure



Resulting Stack Structure

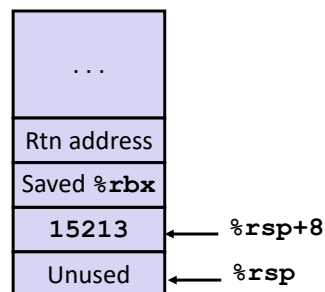


# Callee-Saved Example (2)

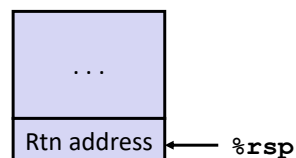
```
long foo2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
foo2:  
    pushq    %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```

Resulting Stack Structure



Pre-return Stack Structure



# Recursive Example

```
/* recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        long bit = x & 1;
        long rest = bitcount_r(x >> 1);
        return bit + rest;
    }
}
```

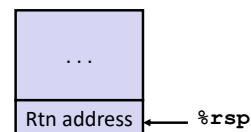
```
bitcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    bitcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

# Recursive Base Case

```
/* recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        long bit = x & 1;
        long rest = bitcount_r(x >> 1);
        return bit + rest;
    }
}
```

```
bitcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    bitcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

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



# Recursive Register Save

```

/* recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        long bit = x & 1;
        long rest = bitcount_r(x >> 1);
        return bit + rest;
    }
}

```

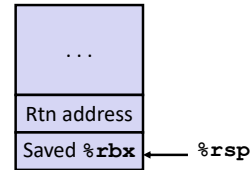
```

bitcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je     .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    bitcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret

```

Register	Use(s)	Type
%rbx	Caller's value	Callee-saved

Caller's value! →



# Recursive Call Setup

```

/* recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        long bit = x & 1;
        long rest = bitcount_r(x >> 1);
        return bit + rest;
    }
}

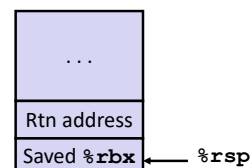
```

```

bitcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je     .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    bitcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret

```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rdi	x >> 1	Recursive arg

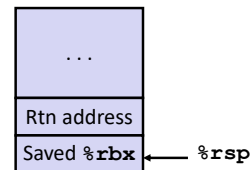


# Recursive Call

```
/* recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        long bit = x & 1;
        long rest = bitcount_r(x >> 1);
        return bit + rest;
    }
}
```

```
bitcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je     .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    bitcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

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

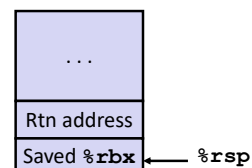


# Recursive Result

```
/* recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        long bit = x & 1;
        long rest = bitcount_r(x >> 1);
        return bit + rest;
    }
}
```

```
bitcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je     .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    bitcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

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



# Recursive Completion

```

/* recursive bitcount */
long bitcount_r(unsigned long x) {
    if (x == 0) {
        return 0;
    } else {
        long bit = x & 1;
        long rest = bitcount_r(x >> 1);
        return bit + rest;
    }
}

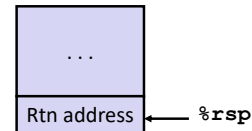
```

```

bitcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    bitcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret

```

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



# Stack Frame Components

