



CSCI-UA.0201

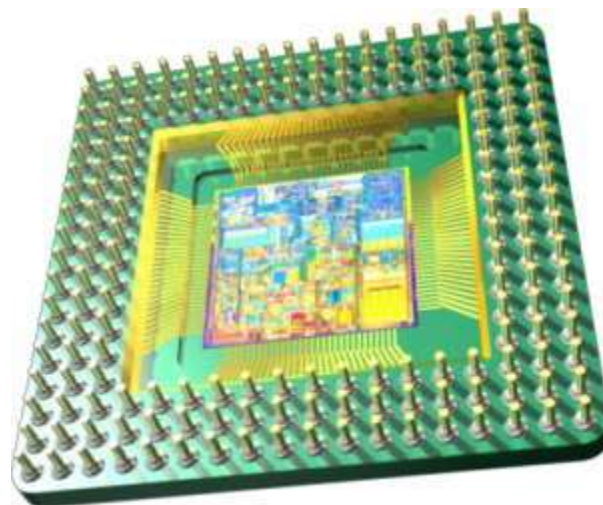
# Computer Systems Organization

## Machine-Level Programming IV

Mohamed Zahran (aka Z)  
mzahran@cs.nyu.edu  
<http://www.mzahran.com>

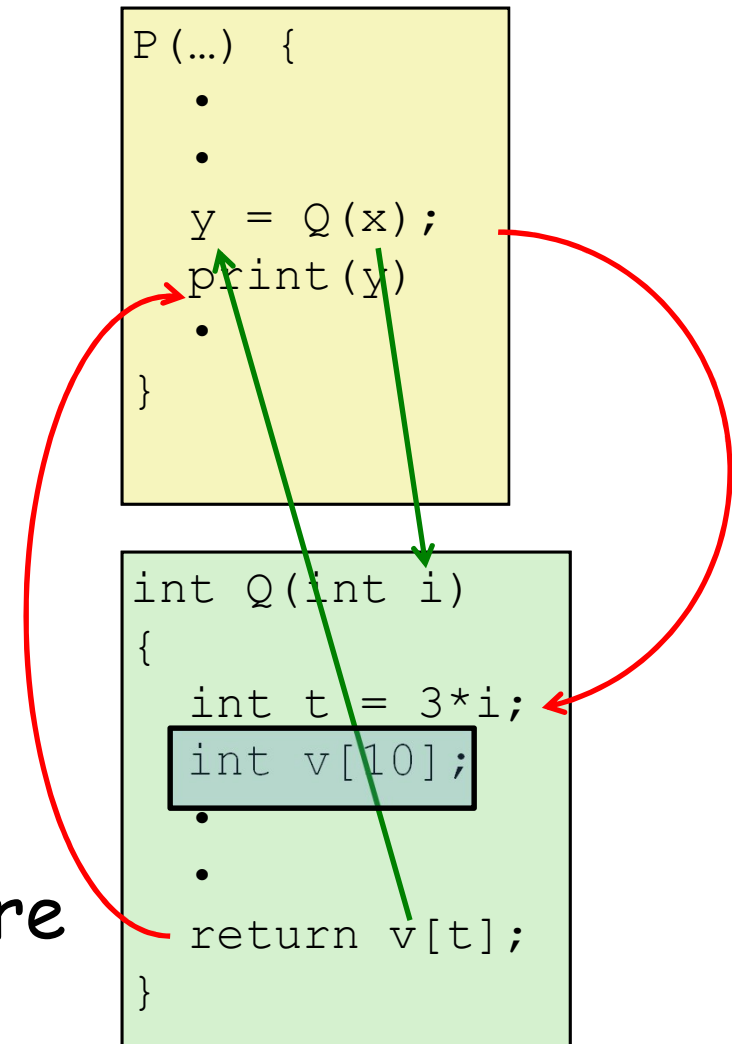
Some slides adapted  
(and slightly modified)  
from:

- Clark Barrett
- Jinyang Li
- Randy Bryant
- Dave O'Hallaron



## Suppose P calls Q

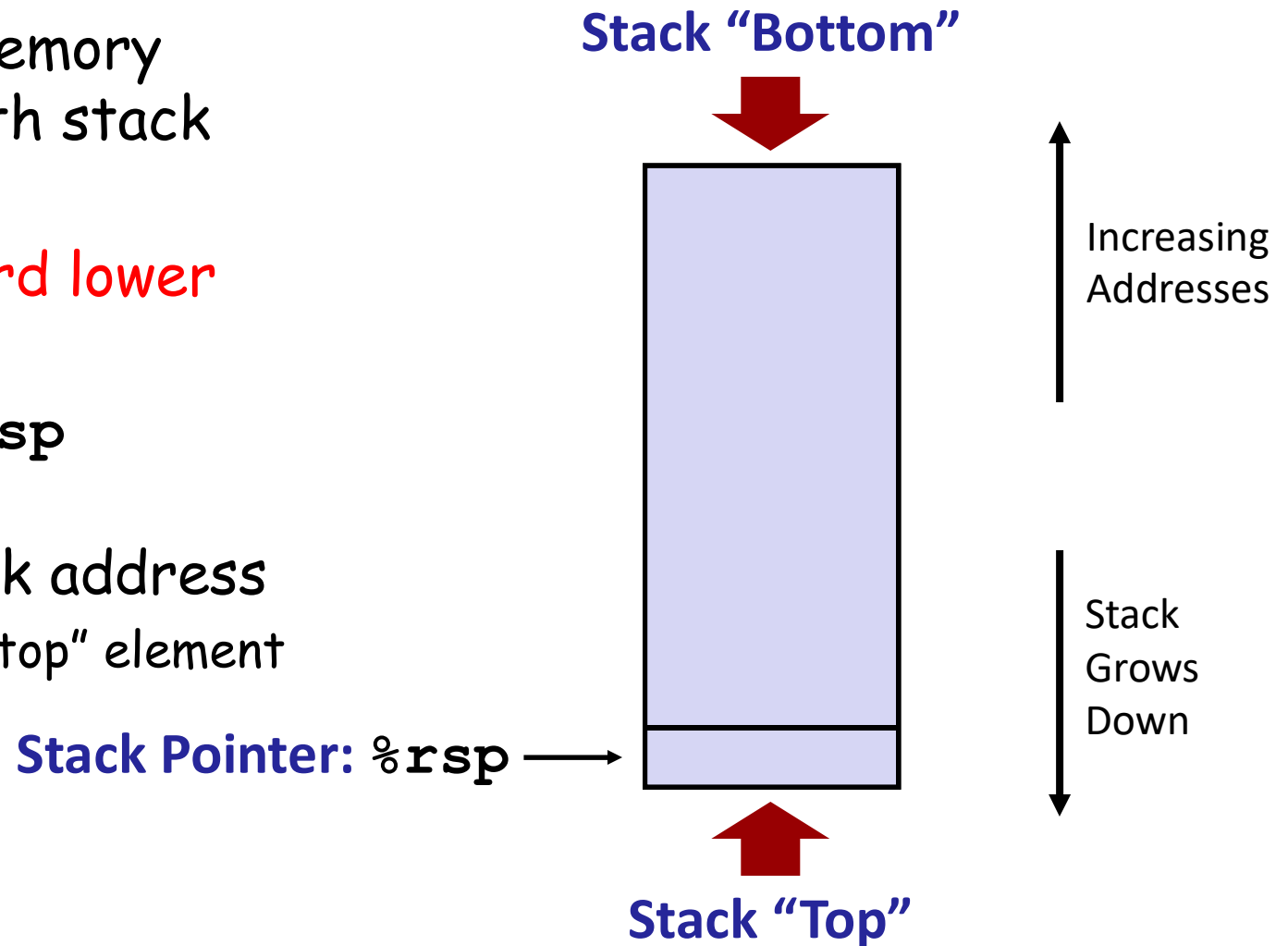
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return



A quick glimpse at how stack works...

# x86-64 Stack

- Region of memory managed with stack discipline
- **Grows toward lower addresses**
- Register `%rsp` contains lowest stack address
  - address of "top" element



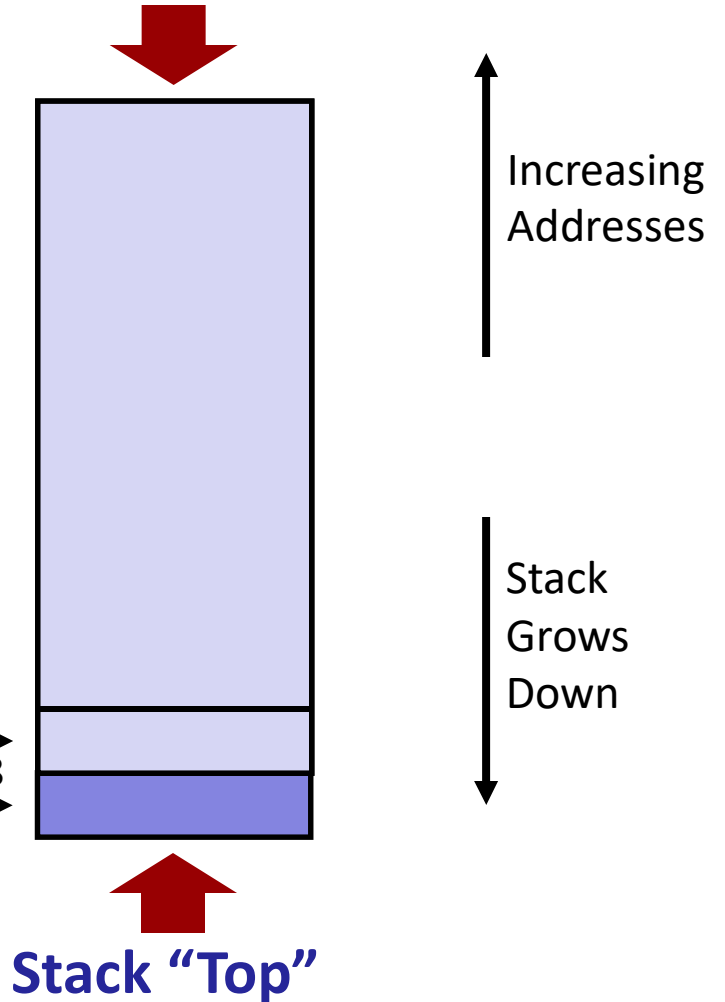
# x86-64 Stack: Push

Stack "Bottom"

- **pushq Src**

- Fetch operand at *Src*
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`

Stack Pointer: `%rsp`

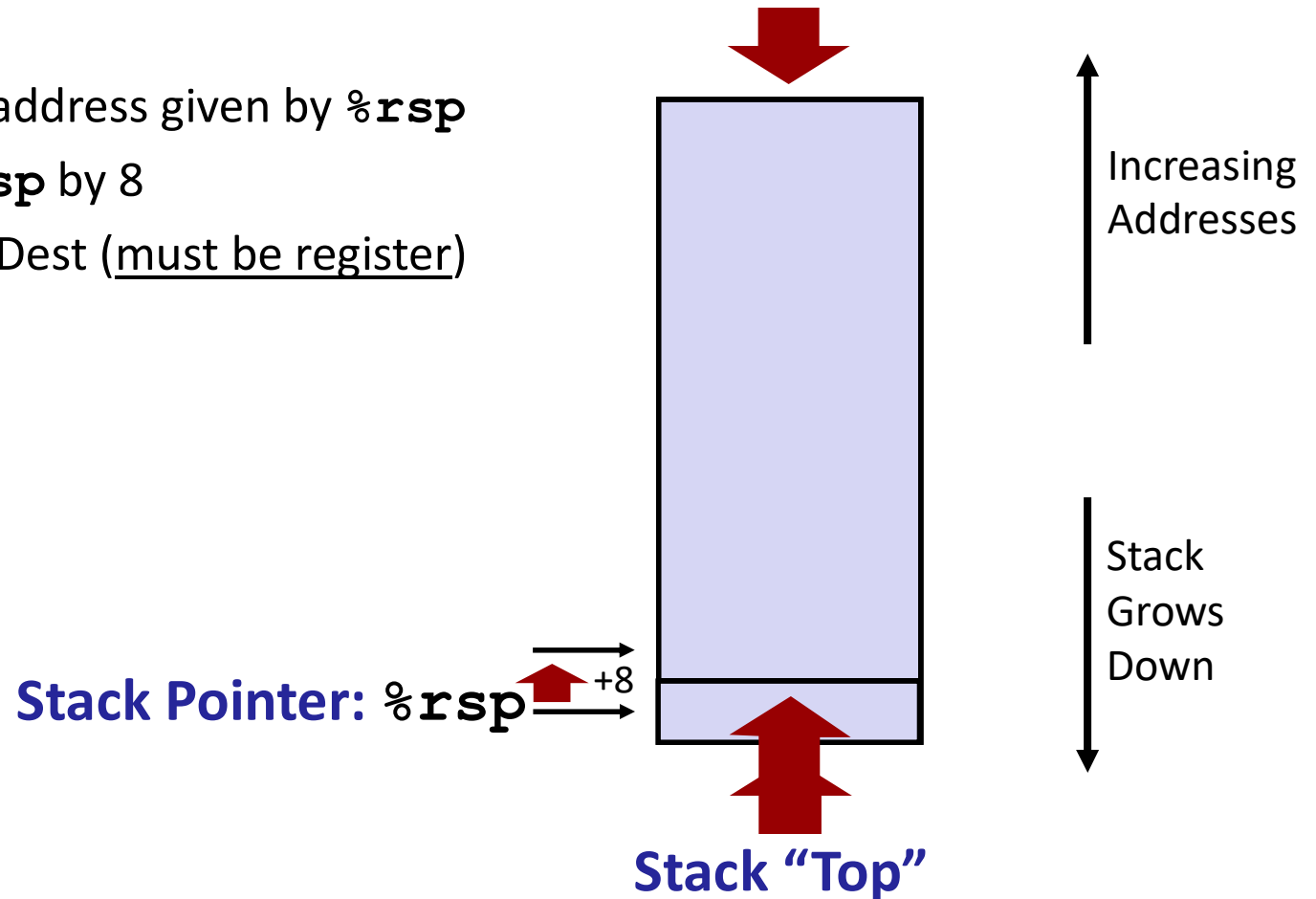


# x86-64 Stack: Pop

Stack "Bottom"

## ■ `popq Dest`

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at `Dest` (must be register)



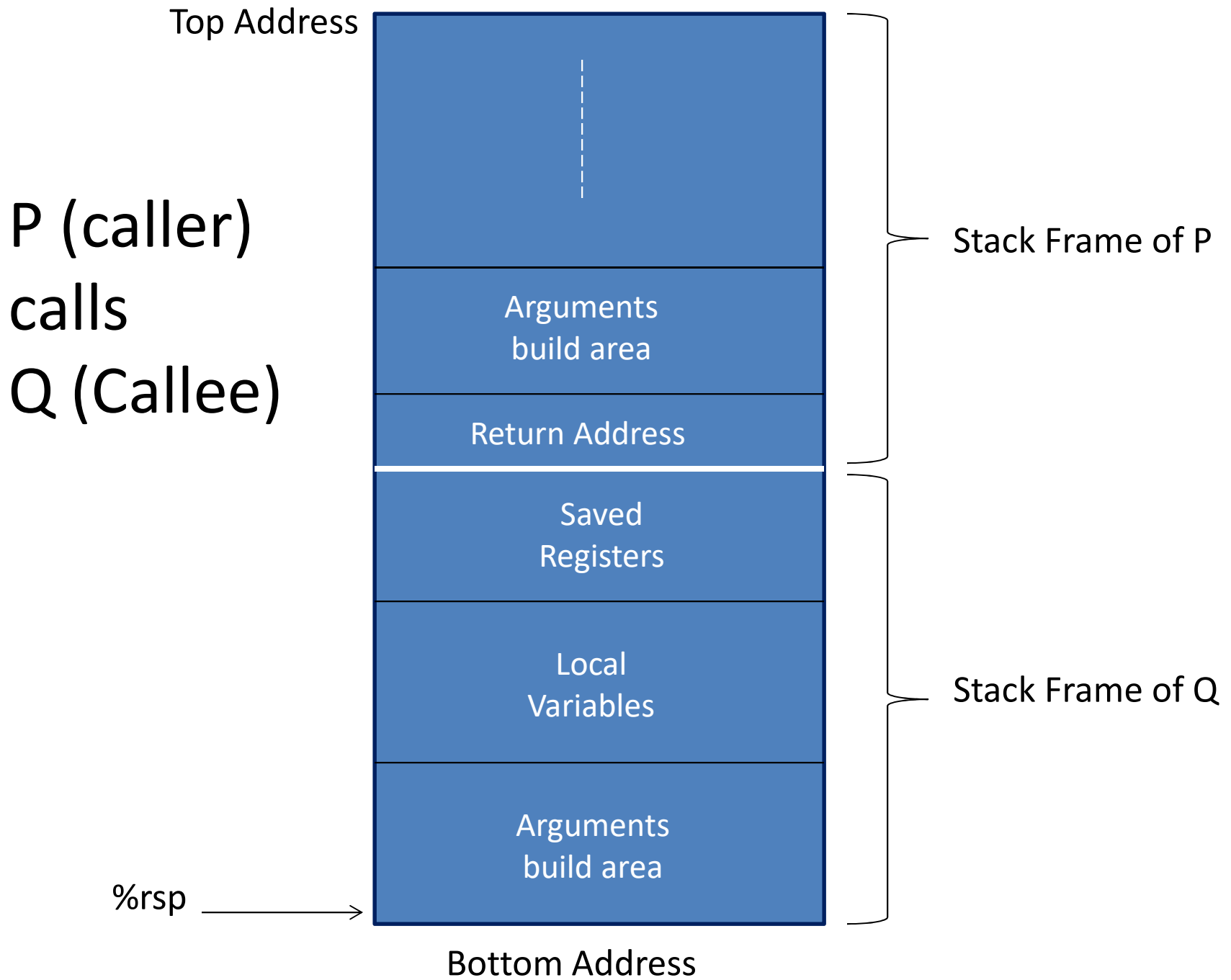
# Examples:

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

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

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

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





# When P calls Q

- P is suspended and control moves to Q.
- A **stack frame** is setup on top of the stack for Q
- That stack frame contains:
  - saved registers
  - local variables
  - arguments if Q is calling another function
- Some procedures may not need a stack frame (why?).

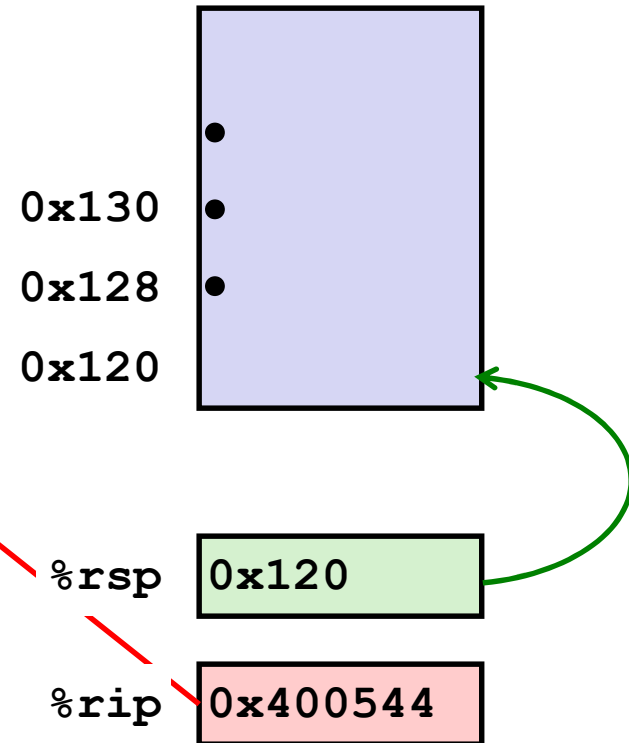
# Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **call label** [or **call \*op**]
  - Push return address on stack
  - Jump to *label*
- Return address:
  - Address of the next instruction right after call
- Procedure return: **ret**
  - Pop address from stack
  - Jump to address

# Example

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov  %rax, (%rbx)  
.  
.
```

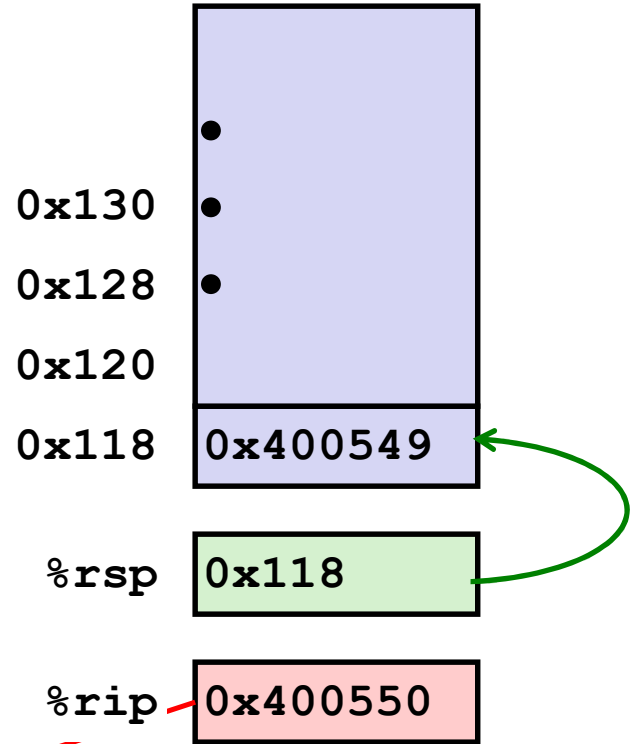
```
0000000000400550 <mult2>:  
400550: mov  %rdi, %rax  
.  
.  
400557: retq
```



# Example

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov  %rax, (%rbx)  
.  
.
```

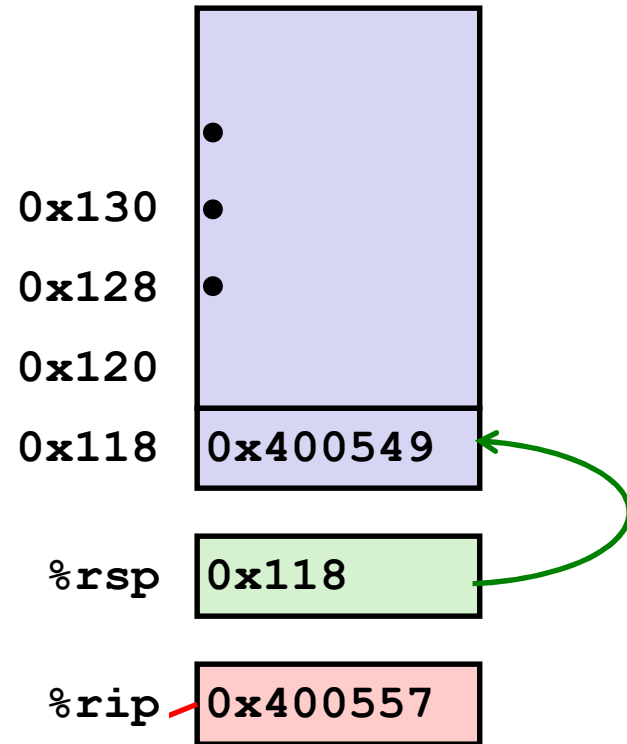
```
0000000000400550 <mult2>:  
400550: mov  %rdi,%rax  
.  
.  
400557: retq
```



# Example

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov  %rax, (%rbx)  
.  
.
```

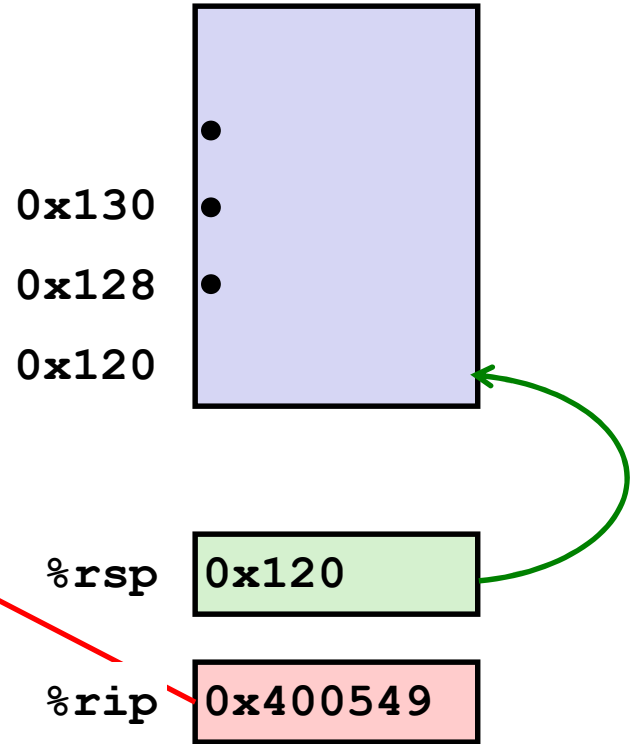
```
0000000000400550 <mult2>:  
400550: mov  %rdi,%rax  
.  
.  
400557: retq
```



# Example

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov  %rax, (%rbx)  
.  
.
```

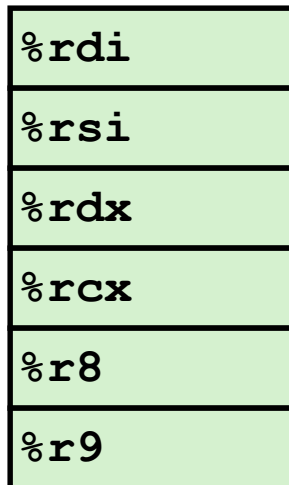
```
0000000000400550 <mult2>:  
400550: mov  %rdi,%rax  
.  
.  
400557: retq
```



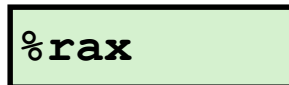
# Procedure Data Flow

## Registers

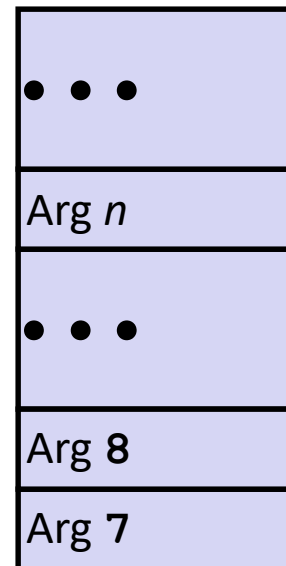
- First 6 arguments



- Return value



## Stack



- Only allocate stack space when needed
- When passing parameters on the stack, all data sizes are rounded up to be multiple of eight.

# Example: multstore calls mult2

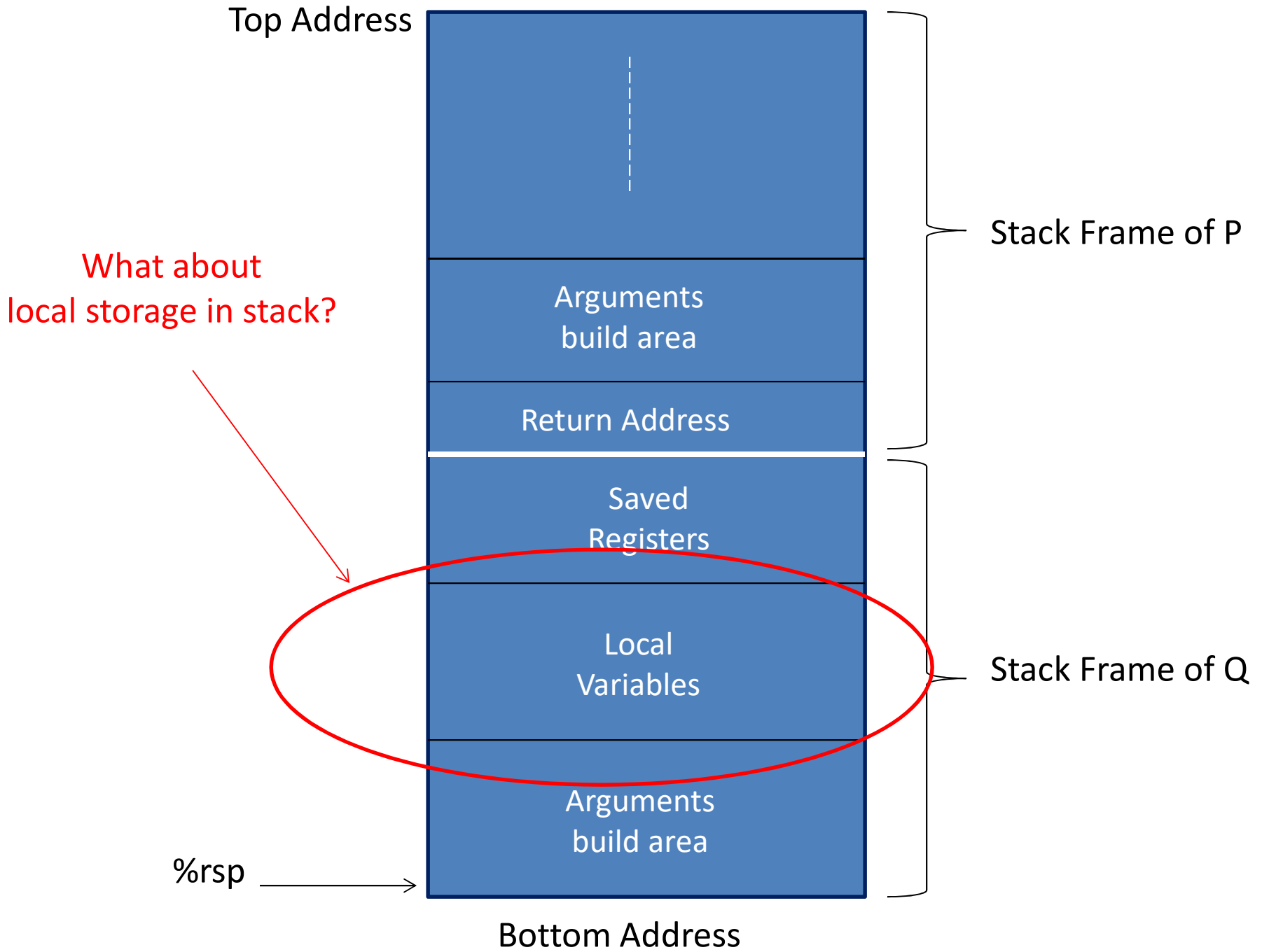
```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
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
```





# When is local storage needed?

- Not enough registers
- A variable in high-level language is referred by its ("&" in C) so needs to have address!
- Arrays, structures, ...

# Example: 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
```

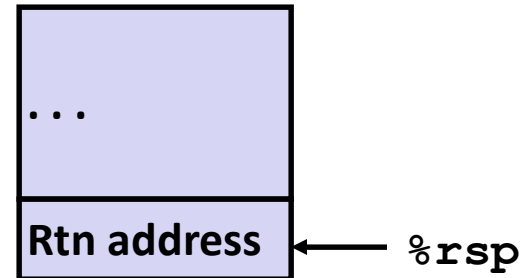
Register	Use(s)
%rdi	Argument <b>p</b>
%rsi	Argument <b>val</b> , <b>y</b>
%rax	<b>x</b> , Return value

# Example: Calling `incr` #1

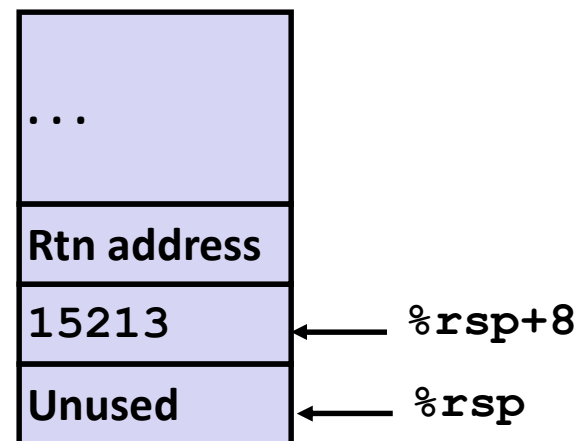
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure

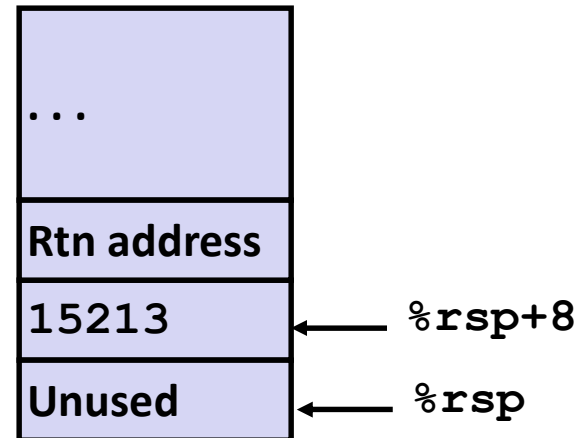


# Example: Calling `incr` #2

```
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
```

## Stack Structure



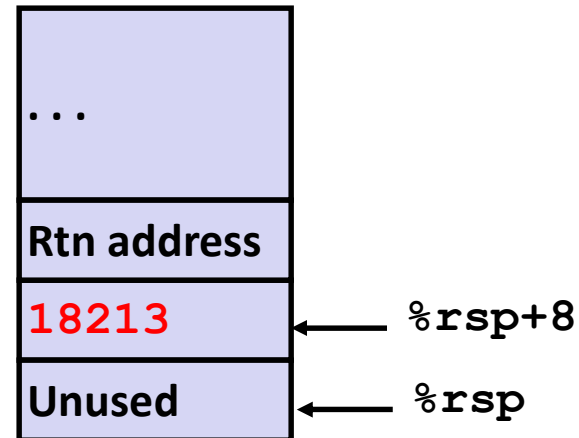
Register	Use(s)
<code>%rdi</code>	<code>&amp;v1</code>
<code>%rsi</code>	3000

# Example: Calling `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



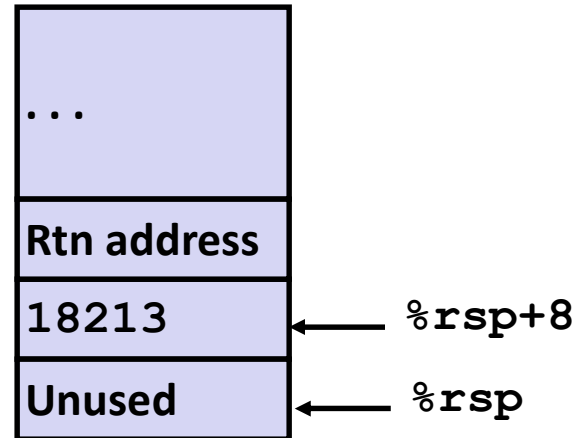
Register	Use(s)
<code>%rdi</code>	<code>&amp;v1</code>
<code>%rsi</code>	3000

# Example: Calling `incr` #4

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

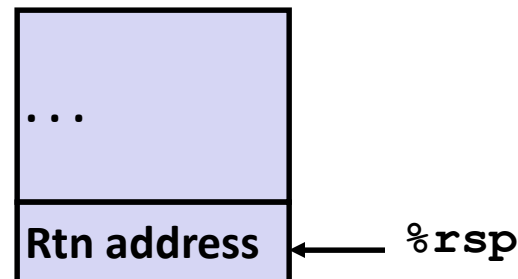
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure



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

Updated Stack Structure

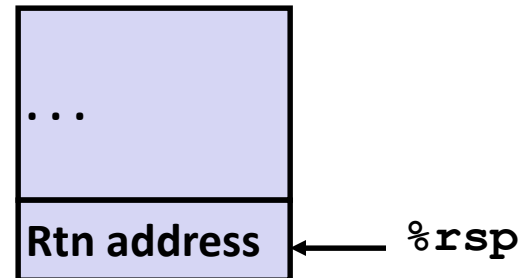


# Example: Calling `incr` #5

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

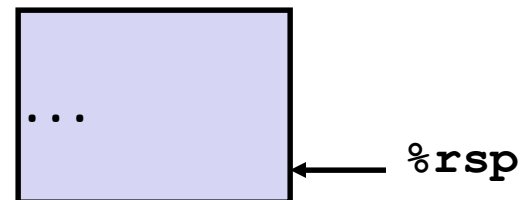
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

## Updated Stack Structure



Register	Use(s)
%rax	Return value

## Final Stack Structure





# Registers Usage Convention

# Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*
  - `who` is the *callee*
- Can register be used for temporary storage?

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

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

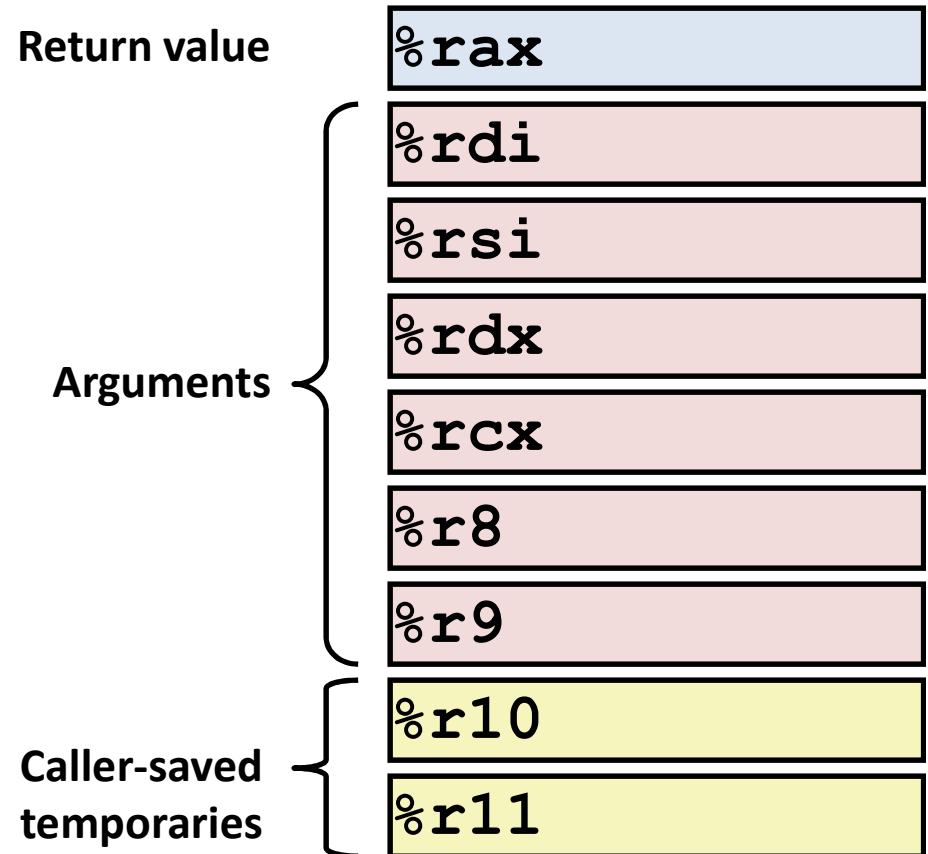
- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
  - Need some coordination

# Register Saving Conventions

- When procedure yoo calls who:
  - yoo is the *caller*
  - who is the *callee*
- Can register be used for temporary storage?
- Conventions
  - *“Caller Saved”*
    - Caller saves temporary values in its frame before the call
  - *“Callee Saved”*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

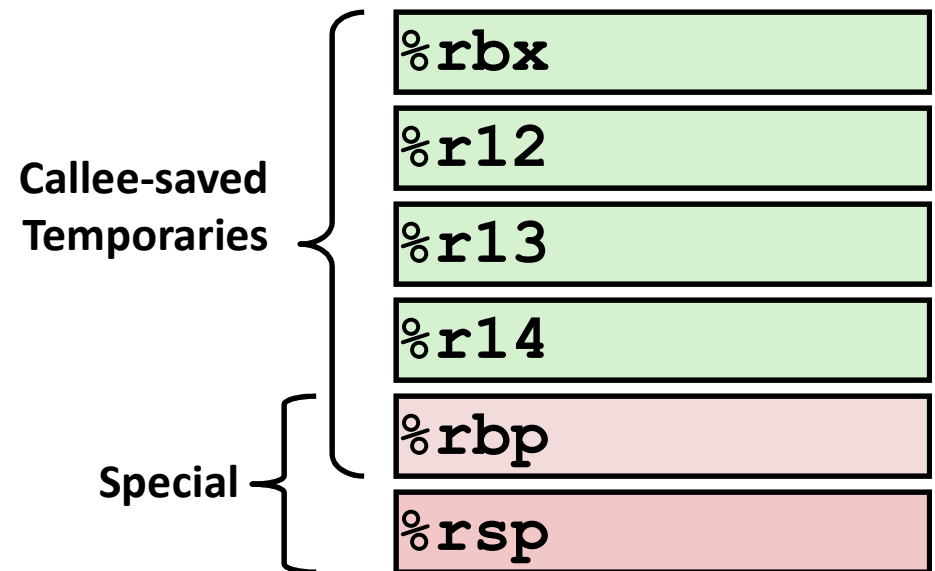
# x86-64 Linux Register Usage #1

- **%rax**
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- **%rdi, ..., %r9**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- **%r10, %r11**
  - Caller-saved
  - Can be modified by procedure



# x86-64 Linux Register Usage #2

- **%rbx, %r12, %r13, %r14**
  - Callee-saved
  - Callee must save & restore
- **%rbp**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- **%rsp**
  - Special form of callee save
  - Restored to original value upon exit from procedure

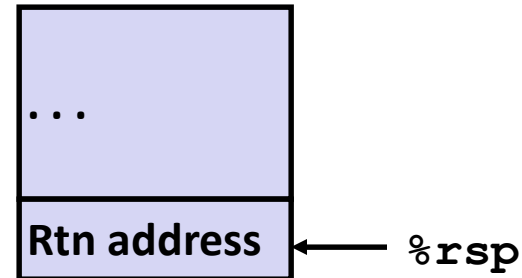


# Callee-Saved Example #1

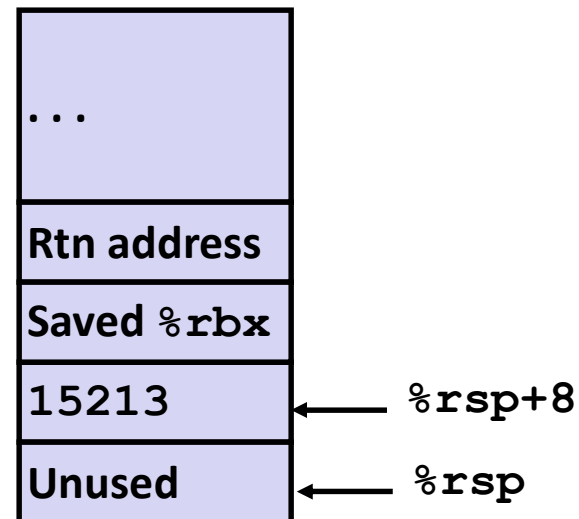
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    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

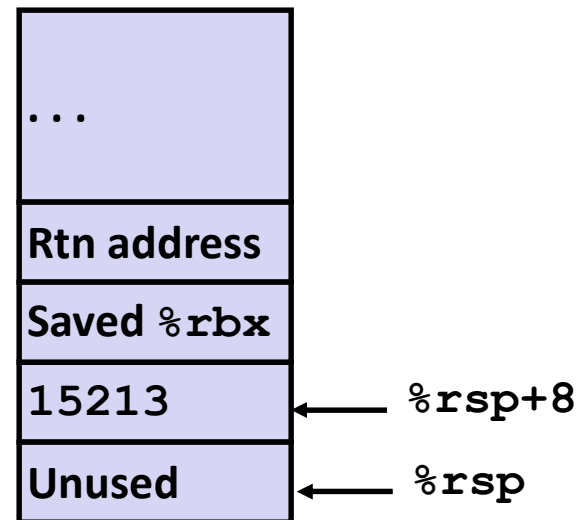


# Callee-Saved Example #2

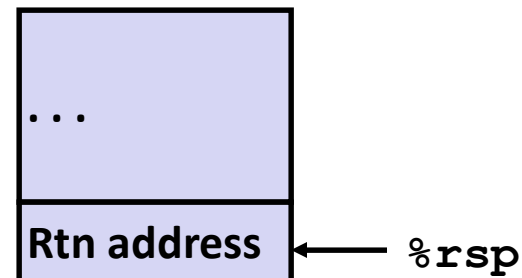
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    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



What About Recursion?



# Recursive Function

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je     .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andq   $1, %rbx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
.L6:
    ret
```

# Recursive Function Terminal Case

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

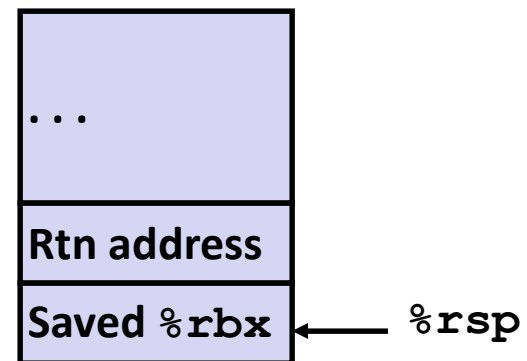
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

# Recursive Function Register Save

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
.L6:
    ret
```

Register	Use(s)	Type
%rdi	x	Argument



# Recursive Function Call Setup

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

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

# Recursive Function Call

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

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

# Recursive Function Result

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je     .L6
    pushq  %rbx
    movq   %rdi, %rbx
    andq   $1, %rbx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
.L6:
    ret
```

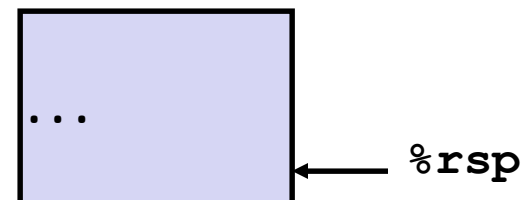
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

# Recursive Function Completion

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

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je     .L6
    pushq  %rbx
    movq   %rdi, %rbx
    andq   $1, %rbx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
.L6:
    ret
```

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



# Observations About Recursion

- Handled Without Special Consideration
  - Register saving conventions prevent one function call from corrupting another's data
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P



# Conclusions

- Important Points
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`

