Machine-Level Programming III: Procedures

CS2011: Introduction to Computer Systems
Lecture 8 (3.7)

Reminder: Condition Codes

Single bit registers

- **CF** Carry Flag (for unsigned)
- **ZF** Zero Flag
- **SF** Sign Flag (for signed)
- **OF** Overflow Flag (for signed)

jX and SetX isntructions

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Reminder: Machine Level Programming – Control

C Control

- if-then-else
- do-while
- while, for
- switch

Assembler Control

- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

Standard Techniques

- Loops converted to do-while/gaurded-do or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (if-elseif-elseif-else)

Mechanisms in Procedures

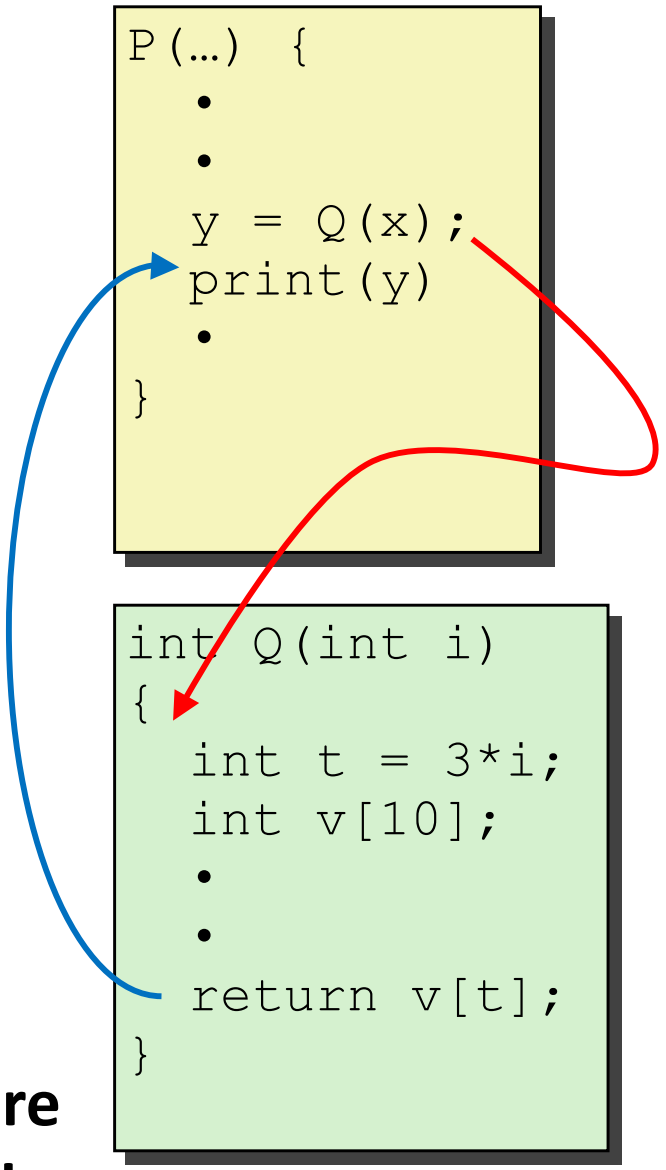
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

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

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

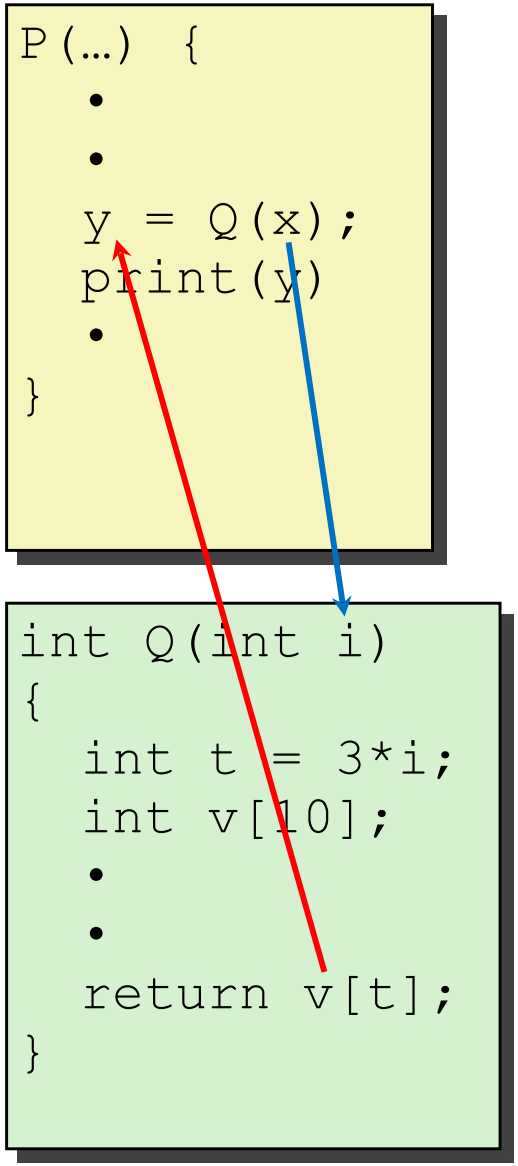
Mechanisms in Procedures

- **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
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



Mechanisms in Procedures

- **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
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**



Mechanisms in Procedures

- **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
- **Mechanisms all implemented with machine instructions**
- **x86-64 implementation of a procedure uses only those mechanisms required**

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

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

Mechanisms in Procedures

Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the **Application Binary Interface (ABI)**.

- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  int v[10];
  •
  •
  return v[t];
}
```

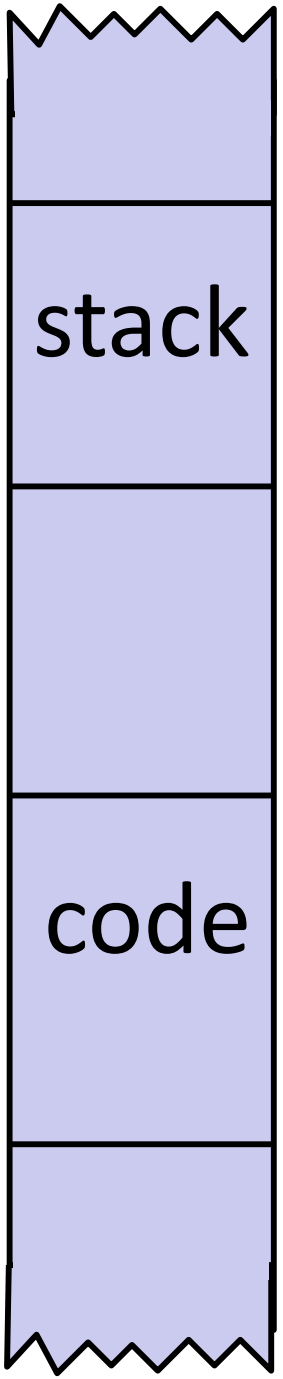
Machine-Level Programming III: Procedures

Procedures

- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Recursion

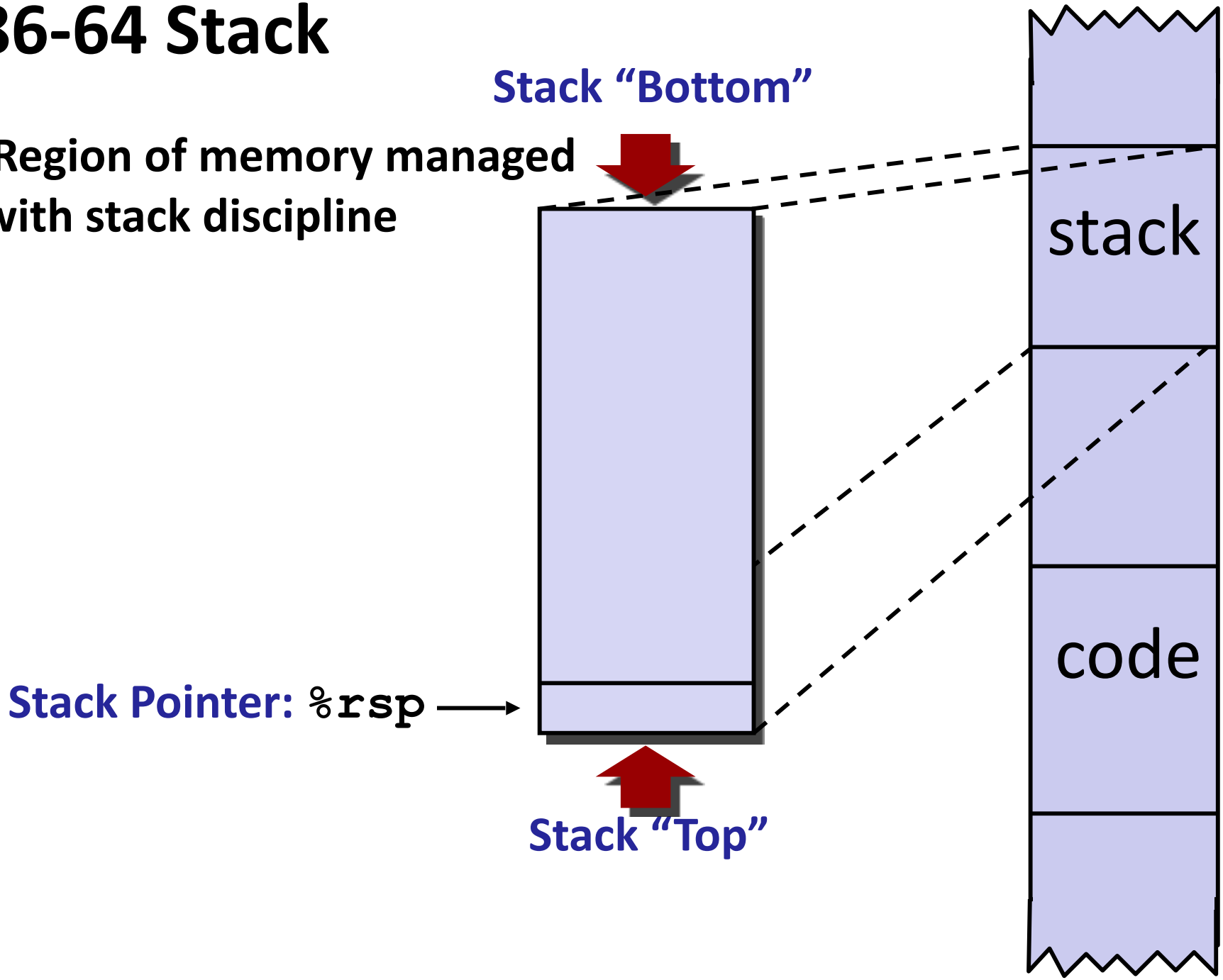
x86-64 Stack

- **Region of memory managed with stack discipline**
  - Memory viewed as array of bytes.
  - Different regions have different purposes.
  - (Like ABI, a policy decision)



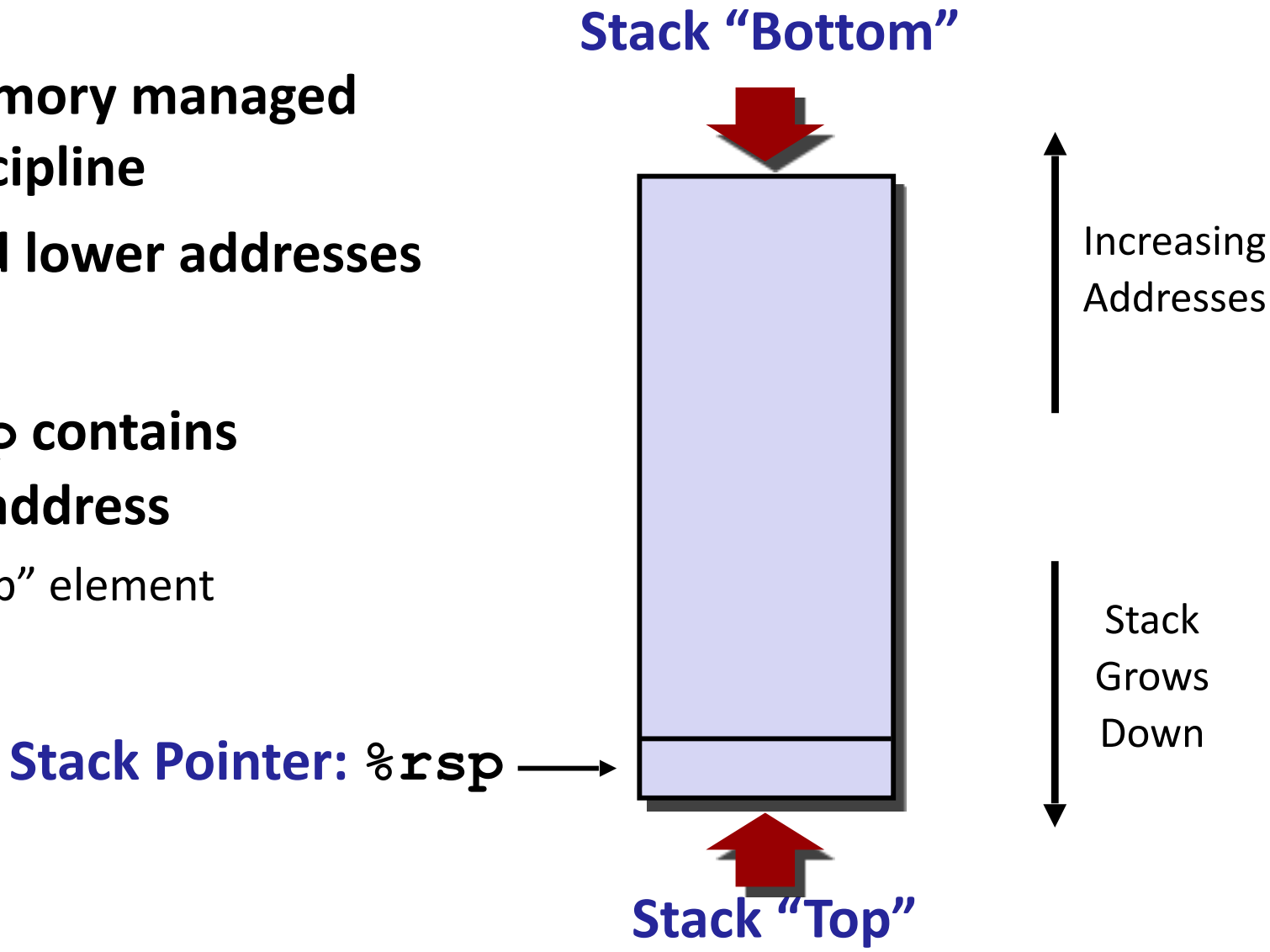
x86-64 Stack

- Region of memory managed with stack discipline

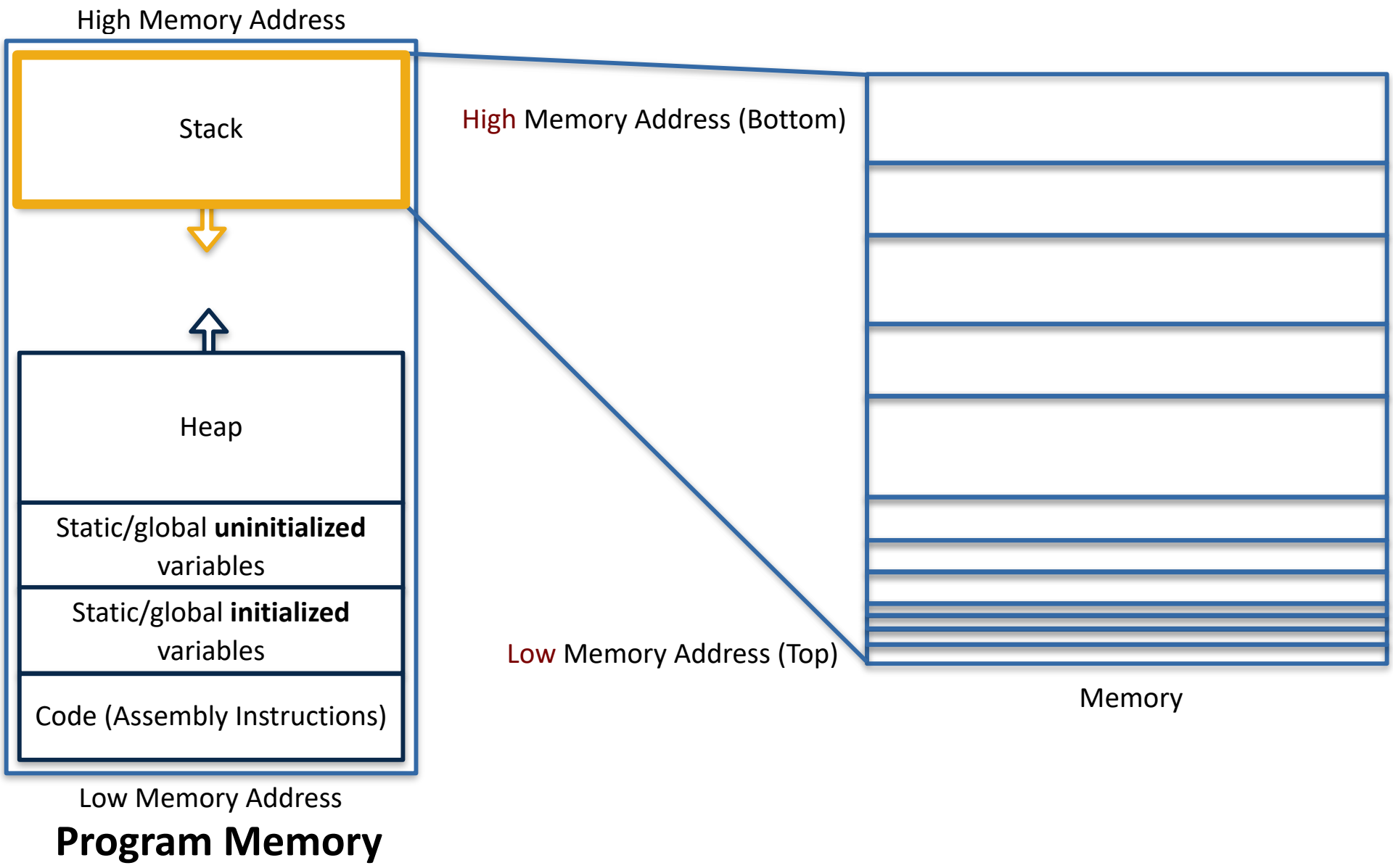


x86-64 Stack

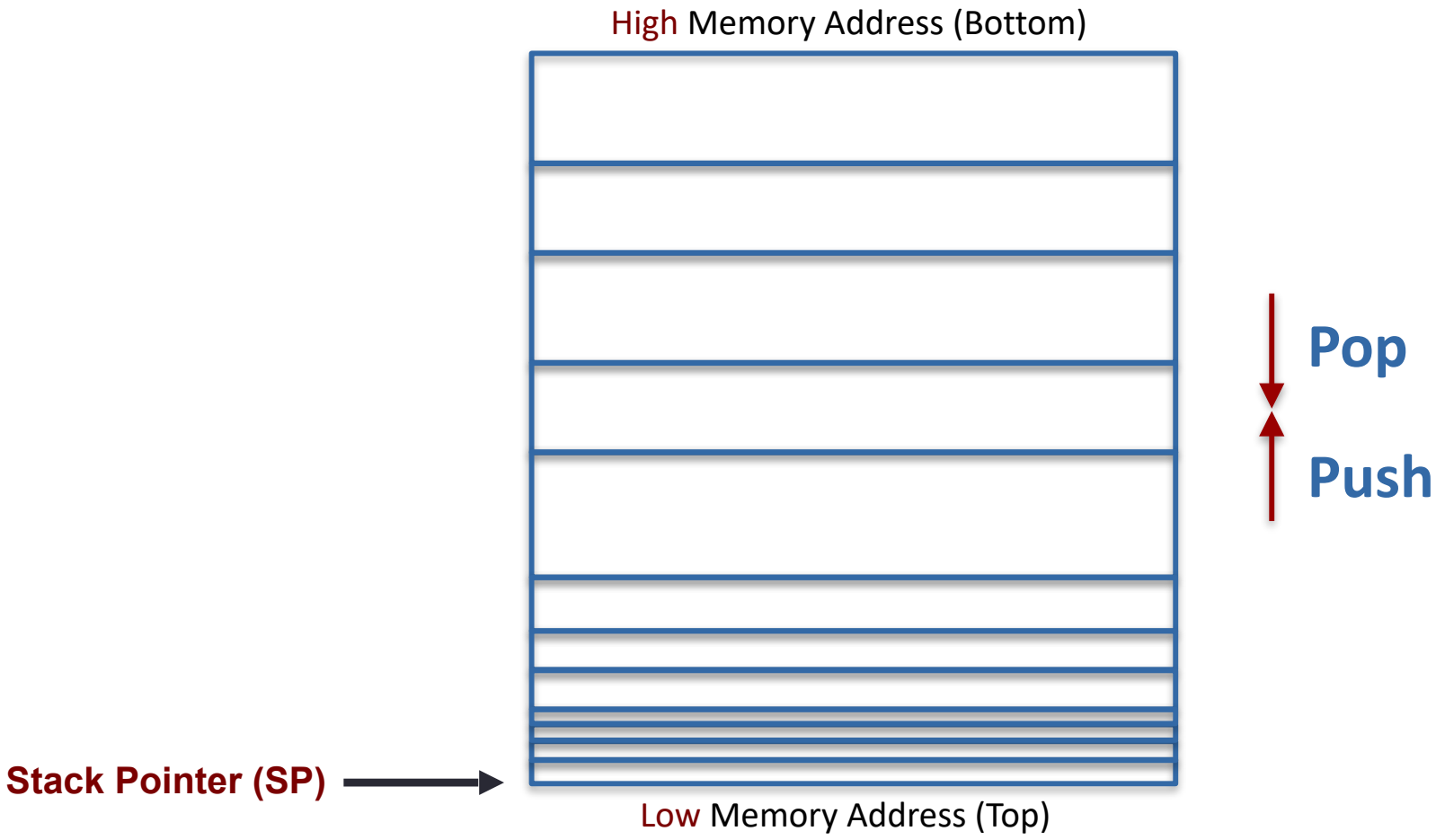
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of "top" element



x86-64 Stack (Alternative Illustration)

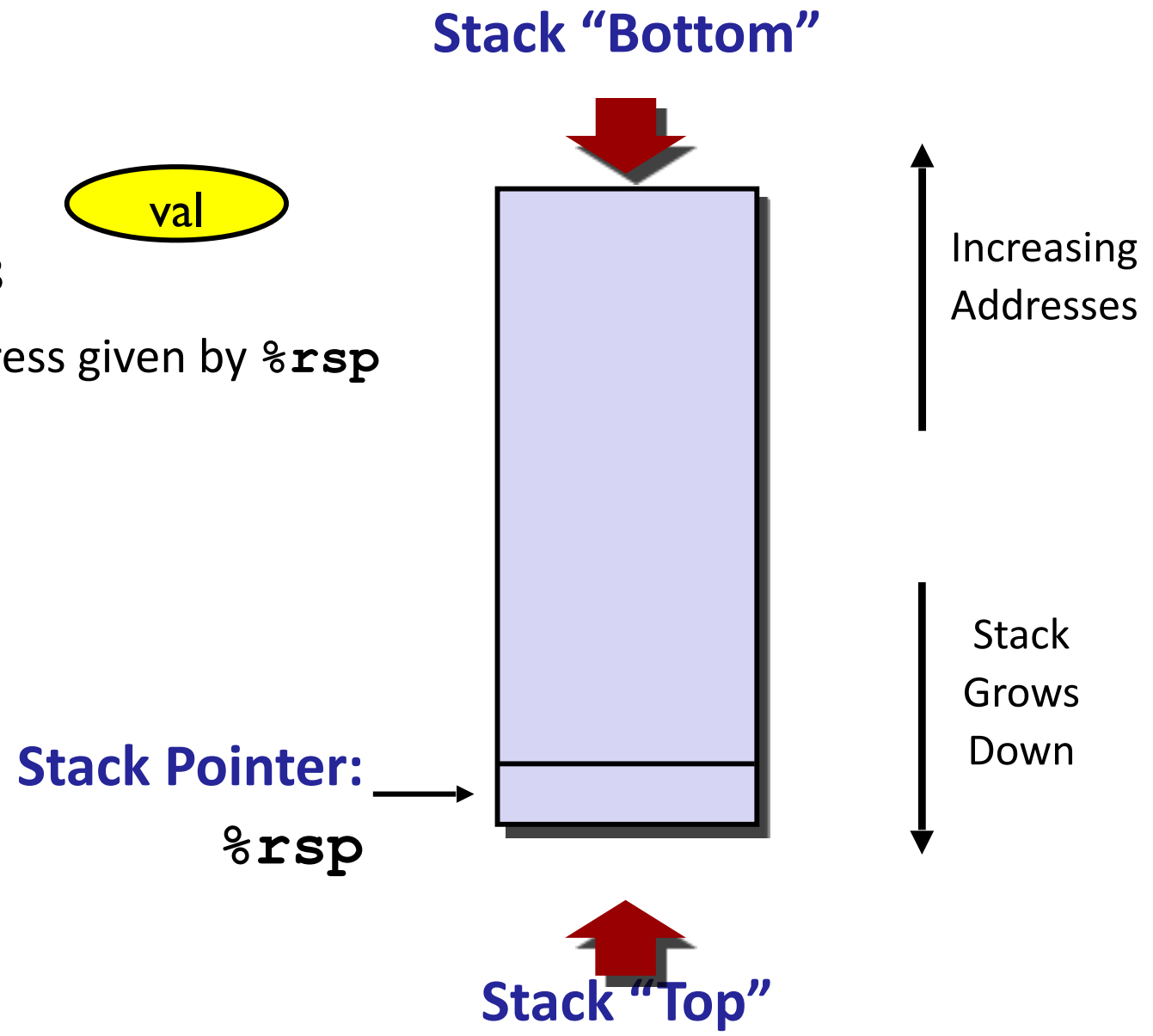


Stack Push/Pop Data Instructions



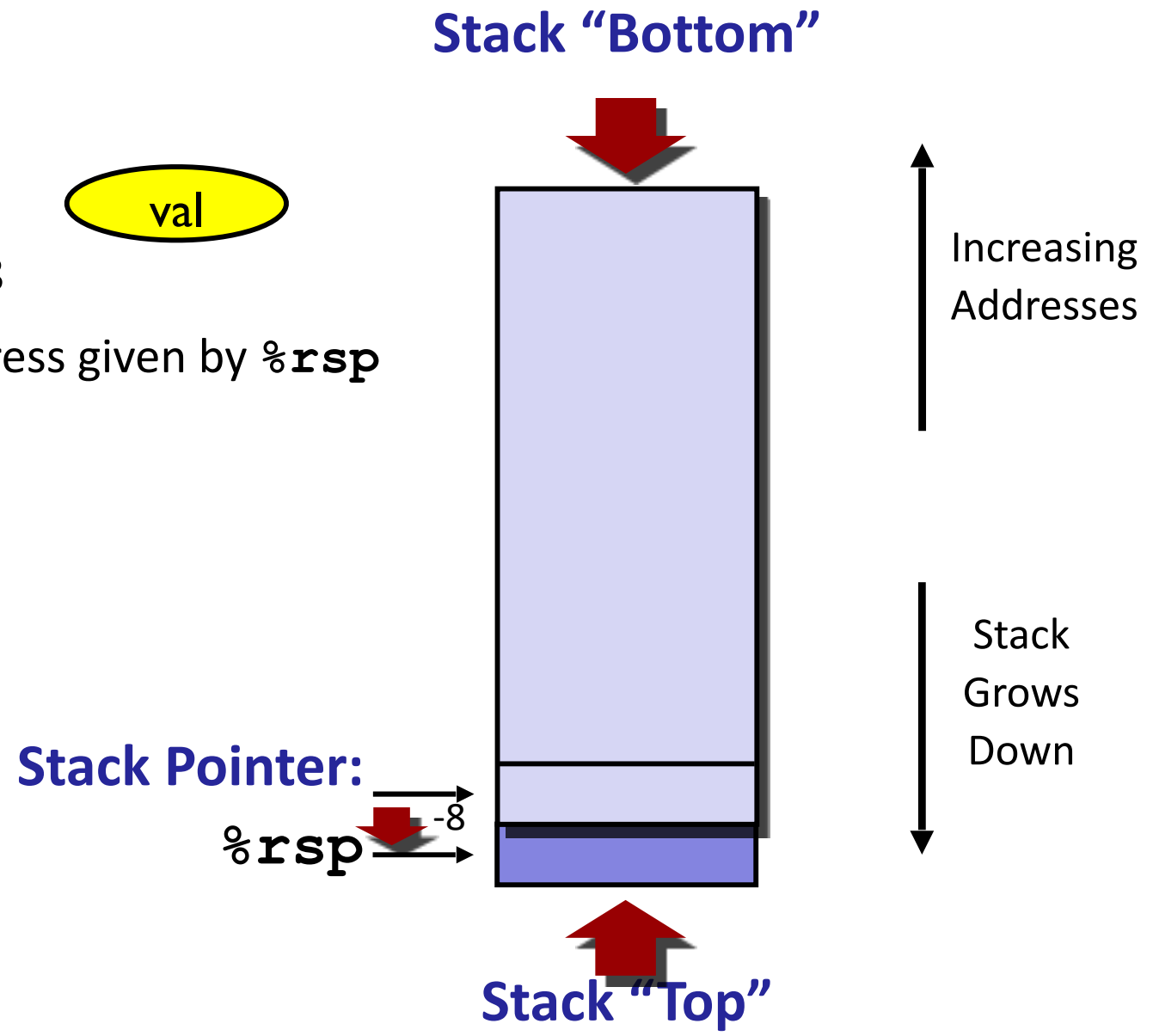
x86-64 Stack: Push

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



x86-64 Stack: Push

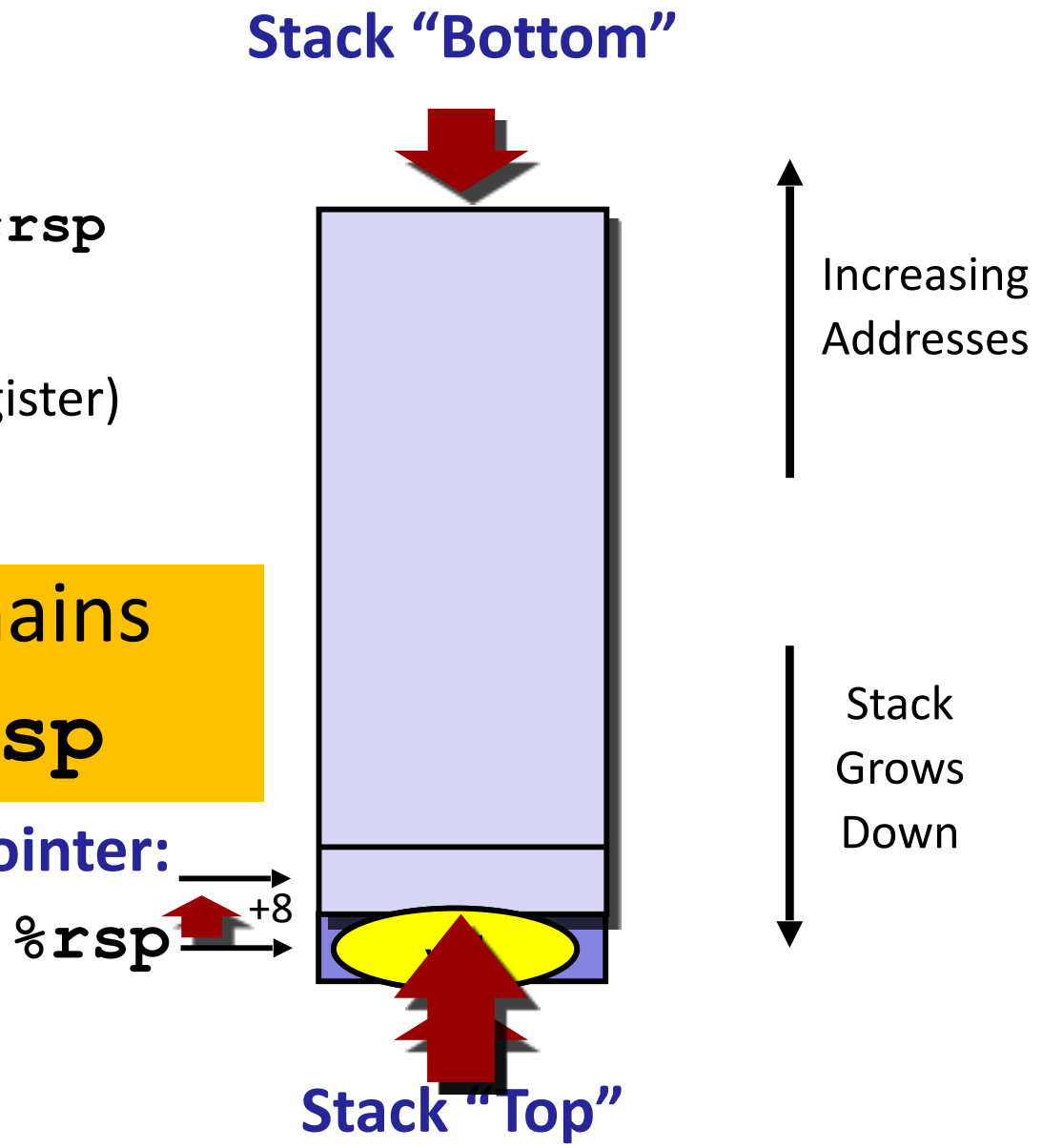
- **`pushq` *Src***
  - Fetch operand at *Src*
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



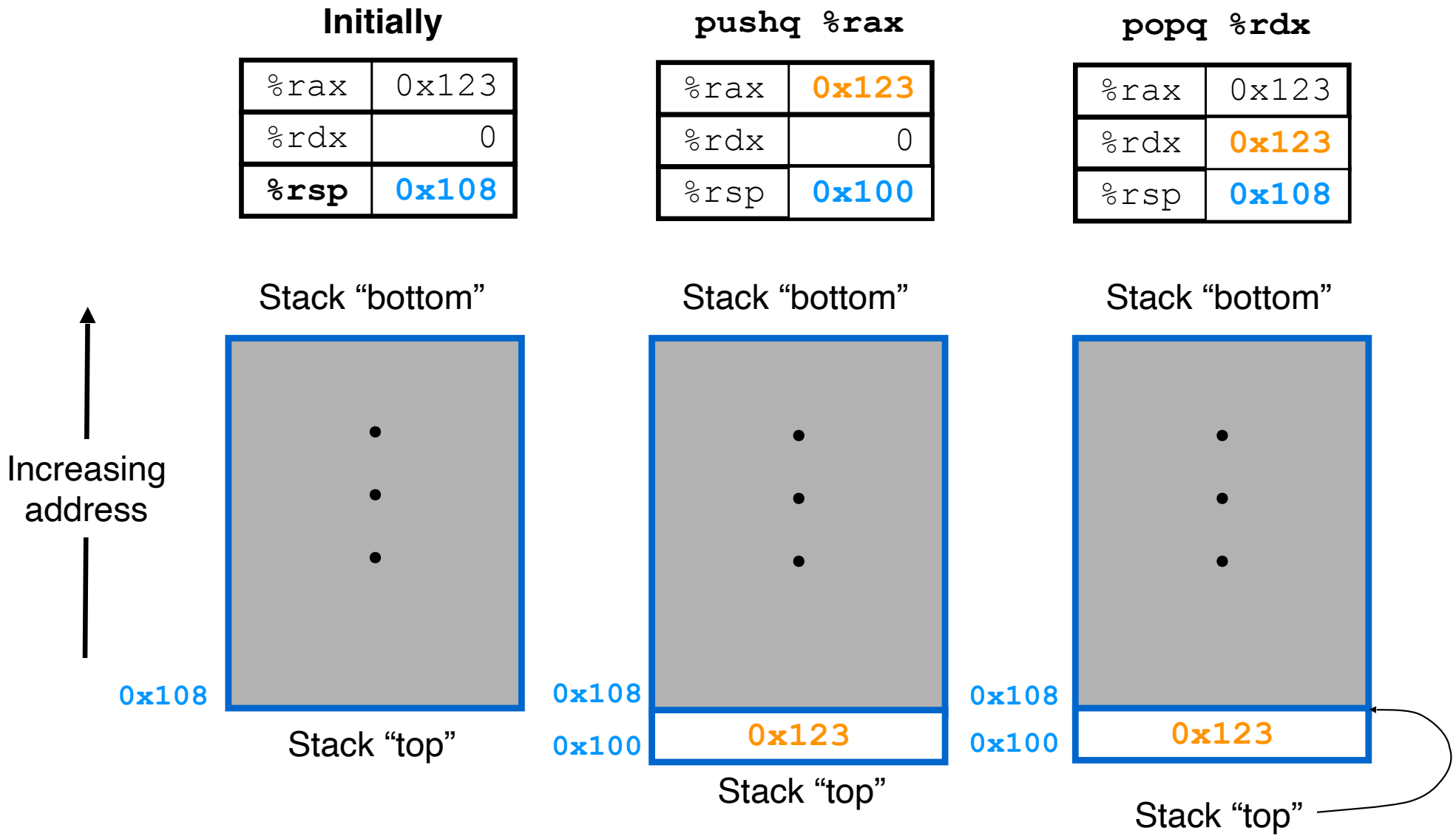
x86-64 Stack: Pop

- `popq` ***Dest***
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)

Value is **copied**; it remains in memory at old `%rsp`



Stack Push/Pop Data Instructions (Example)



Machine-Level Programming III: Procedures

Procedures

- Stack Structure
- Calling Conventions
  - **Passing control**
  - Passing data
  - Managing local data
- Recursion

Code Examples

```
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: call   400550 <mult2>        # mult2(x,y)
  400549: mov    %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: ret                          # 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:  ret                         # Return
```

Procedure Control Flow

Use stack to support procedure call and return

Procedure call: `call label`

- Push return address on stack
- Jump to ***label*** ***(i.e., sets PC or %rip to label)***

Return address:

- Address of the next instruction right after call
- Example from disassembly

Procedure return: `ret`

- Pop return address from stack
- Jump to return address ***(i.e., sets PC or %rip to return address)***

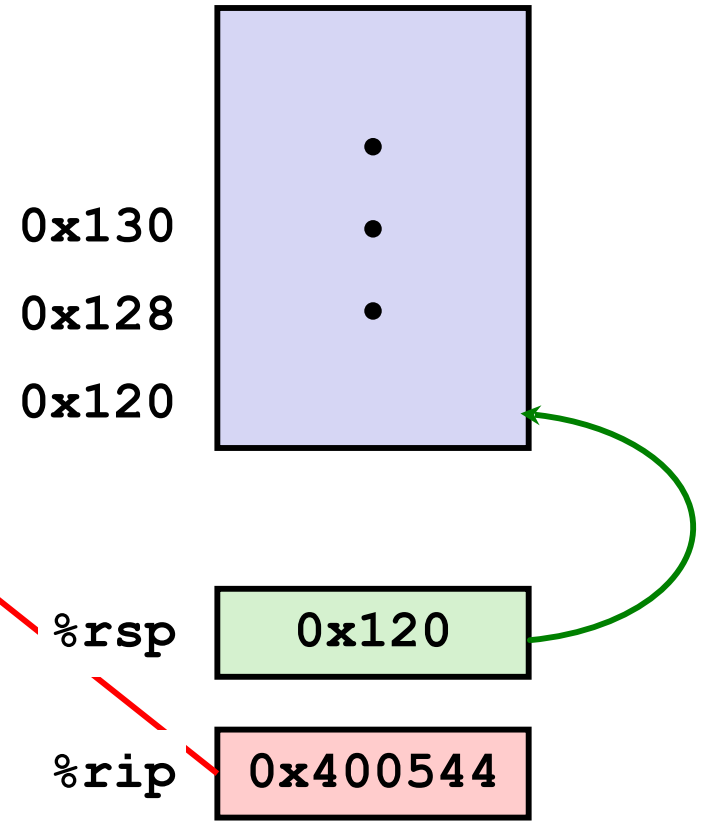
These instructions are sometimes printed with a `q` suffix

- This is just to remind you that you're looking at 64-bit code (call and callq are used interchangeably in x86-64)

Control Flow Example #1

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

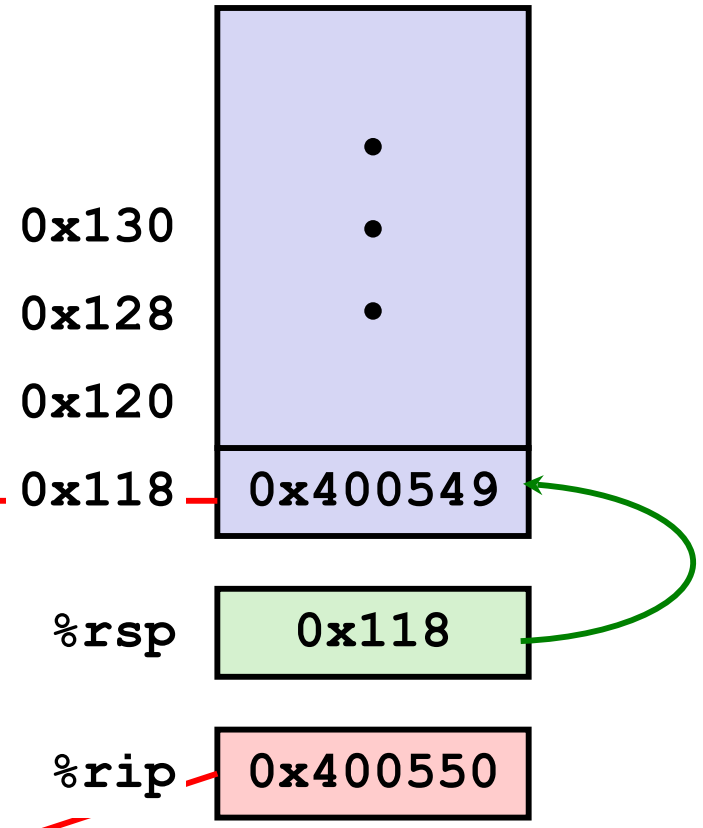
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  ret
```



Control Flow Example #2

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

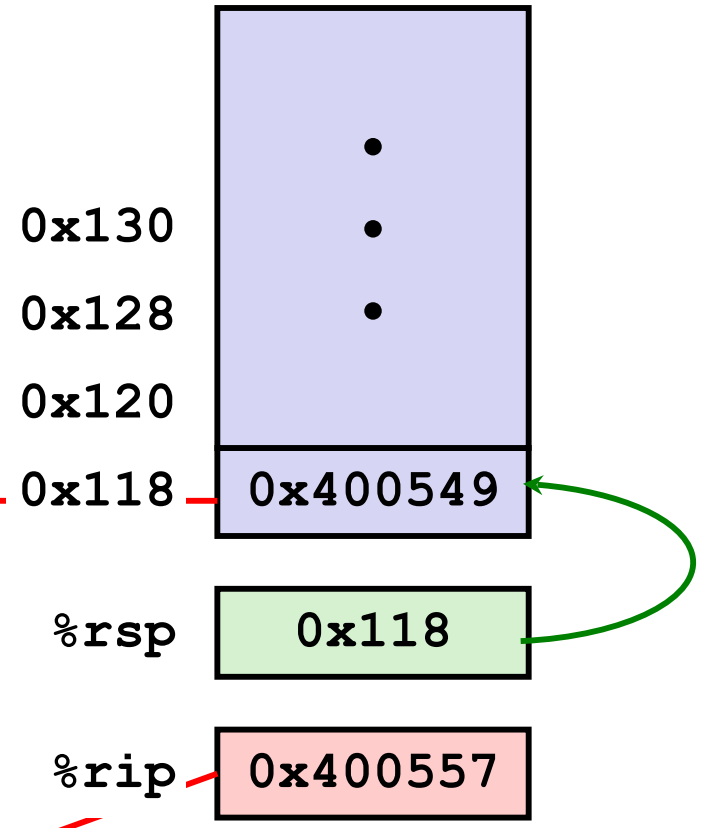
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  ret
```



Control Flow Example #3

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

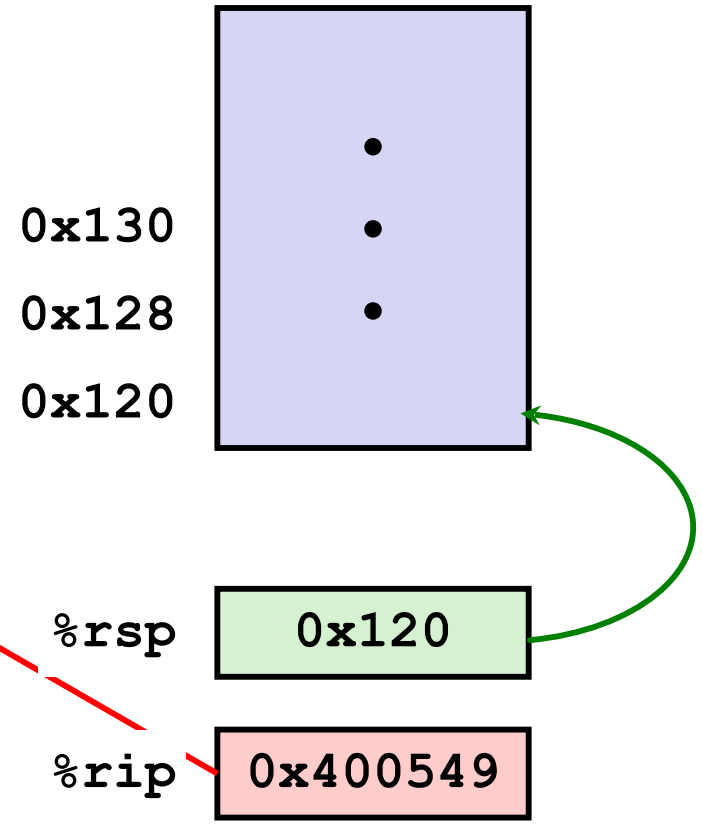
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  ret
```



Control Flow Example #4

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

```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  ret
```



Machine-Level Programming III: Procedures

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - **Passing data**
  - Managing local data
- Recursion

Procedure Data Flow (x86-64)

Registers

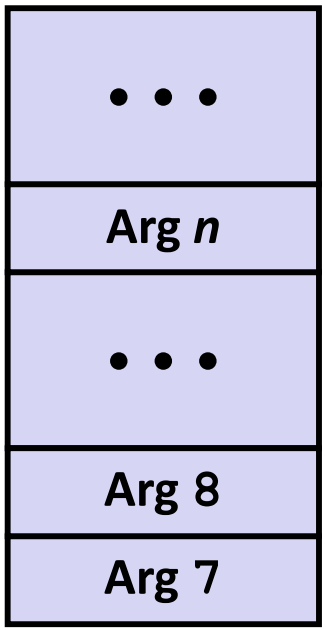
- First 6 arguments

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

The registers are used in this specified order.
Register **name** **depends on the size of the data type being passed.**

E.g.,
Operand size **64bits**: %rdi, %rdx, %r8, %r9
Operand size **32bits**: %edi, %edx, %r8**d**, %r9d
Operand size **16bits**: %di, %dx, %r8**w**, %r9w
Operand size **8bits**: %di**l**, %cl, %r8**b**, %r9b

Stack



- Return value

%rax

- Only allocate stack space when needed (much less efficient to allocate on stack)

Passing Data Example

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

dest in **%rdx** is moved in **%rbx** in case the **mult2** call actually needs **%rdx** (may call another function with 3 arguments).

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

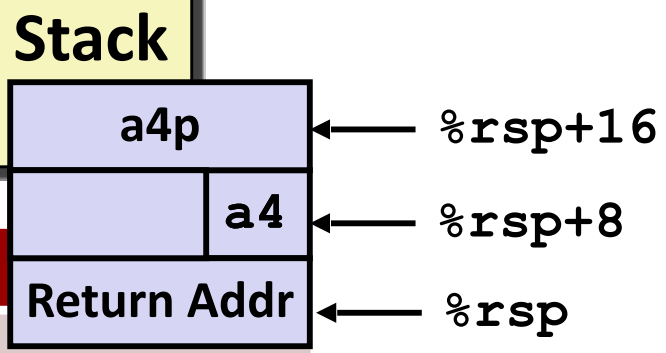
```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

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

Passing Data (Another x86-64 Example)

```
void proc(long a1, long *a1p, int a2, int *a2p,
short a3, short *a3p, char a4, char *a4p){
        *a1p += a1;
        *a2p += a2;
        *a3p += a3;
        *a4p += a4;
}
```

```
proc:
  movq     16(%rsp), %rax
  addq     %rdi, (%rsi)
  addl     %edx, (%rcx)
  addw     %r8w, (%r9)
  movl     8(%rsp), %edx
  addb     %dl, (%rax)
  ret
```



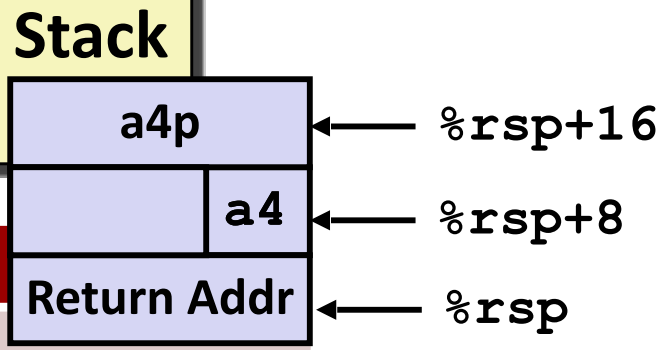
Register	Use(s)
%rdi	a1
%rsi	&a1p
%edx	a2
%rcx	&a2p
%r8w	a3
%r9	&a3p

Passing Data (Another x86-64 Example)

```
void proc(long a1, long *a1p, int a2, int *a2p,
short a3, short *a3p, char a4, char *a4p){
        *a1p += a1;
        *a2p += a2;
        *a3p += a3;
        *a4p += a4;
}
```

```
proc:
  movq    16(%rsp), %rax
  addq    %rdi, (%rsi)
  addl    %edx, (%rcx)
  addw    %r8w, (%r9)
  movl    8(%rsp), %edx
  addb    %dl, (%rax)
  ret
```

Register	Use(s)
%rdi	a1
%rsi	&a1p
%edx	a2
%rcx	&a2p
%r8w	a3
%r9	&a3p

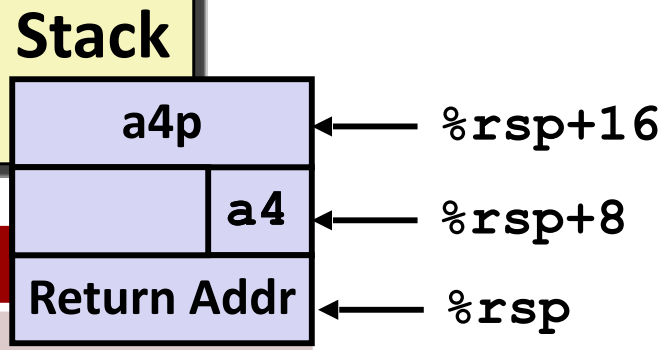


Passing Data (Another x86-64 Example)

```
void proc(long a1, long *a1p, int a2, int *a2p,
short a3, short *a3p, char a4, char *a4p){
        *a1p += a1;
        *a2p += a2;
        *a3p += a3;
        *a4p += a4;
}
```

```
proc:
  movq    16(%rsp), %rax
  addq    %rdi, (%rsi)
  addl    %edx, (%rcx)
  addw    %r8w, (%r9)
  movl    8(%rsp), %edx
  addb    %dl, (%rax)
  ret
```

Register	Use(s)
%rdi	a1
%rsi	&a1p
%edx	a2
%rcx	&a2p
%r8w	a3
%r9	&a3p

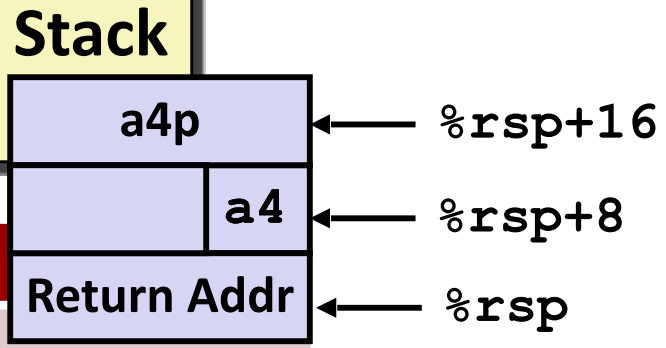


Passing Data (Another x86-64 Example)

```
void proc(long a1, long *a1p, int a2, int *a2p,
short a3, short *a3p, char a4, char *a4p){
        *a1p += a1;
        *a2p += a2;
        *a3p += a3;
        *a4p += a4;
}
```

```
proc:
  movq     16(%rsp), %rax
  addq     %rdi, (%rsi)
  addl     %edx, (%rcx)
  addw     %r8w, (%r9)
  movl     8(%rsp), %edx
  addb     %dl, (%rax)
  ret
```

Register	Use(s)
%rdi	a1
%rsi	&a1p
%edx	a2
%rcx	&a2p
%r8w	a3
%r9	&a3p

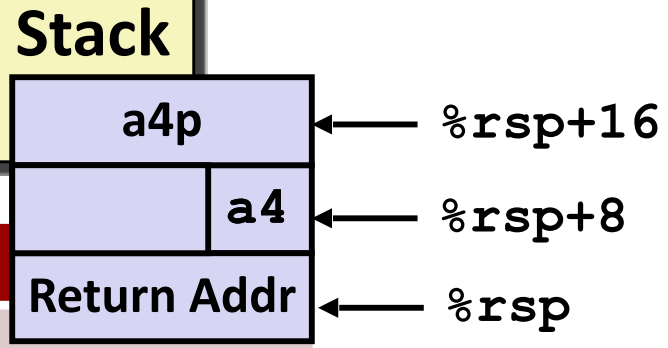


Passing Data (Another x86-64 Example)

```
void proc(long a1, long *a1p, int a2, int *a2p,
short a3, short *a3p, char a4, char *a4p){
        *a1p += a1;
        *a2p += a2;
        *a3p += a3;
        *a4p += a4;
}
```

```
proc:
  movq    16(%rsp), %rax
  addq    %rdi, (%rsi)
  addl    %edx, (%rcx)
  addw    %r8w, (%r9)
  movl    8(%rsp), %edx
  addb    %dl, (%rax)
  ret
```

Register	Use(s)
%rdi	a1
%rsi	&a1p
%edx	a2
%rcx	&a2p
%r8w	a3
%r9	&a3p



Machine-Level Programming III: Procedures

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - **Managing local data**
- Recursion

Stack-Based Languages

Languages that support recursion

- e.g., C, Pascal, Java
- Code must be "*Reentrant*"
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

Stack discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

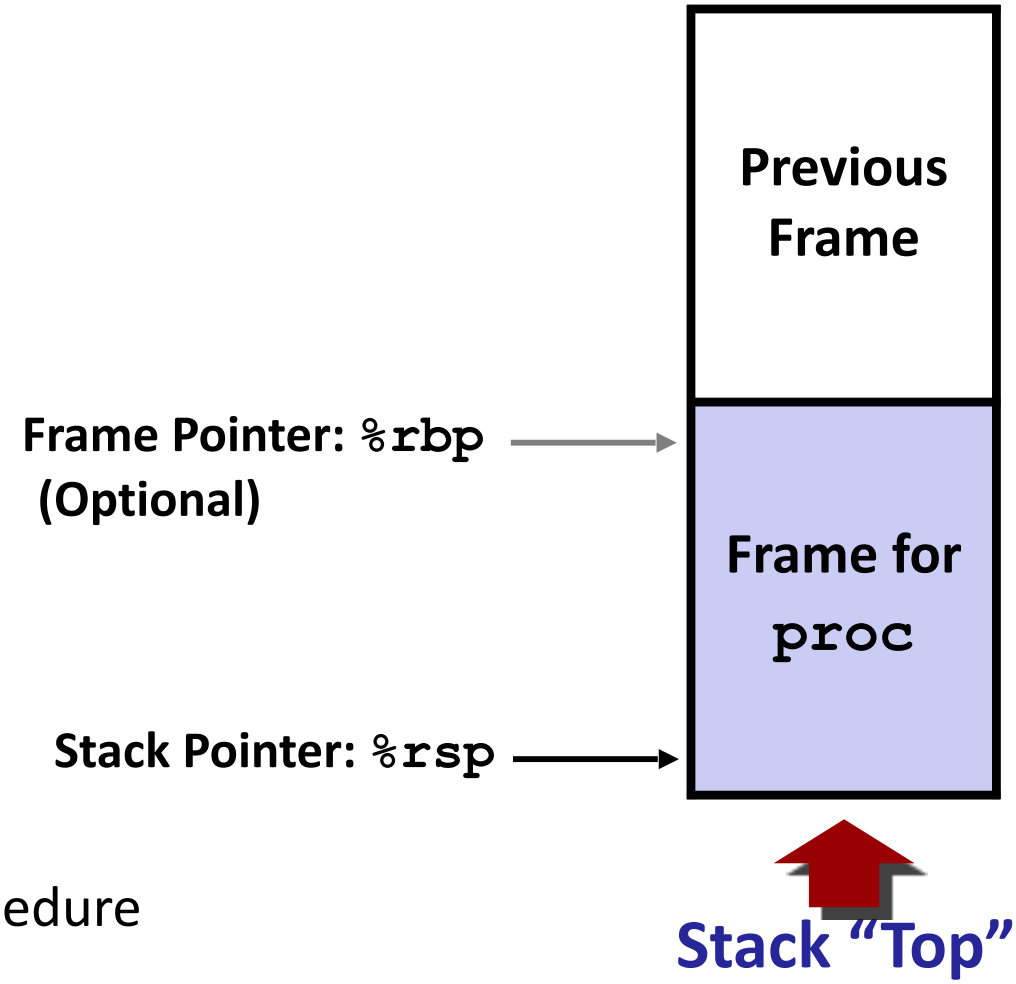
Stack allocated in *Frames*

- state for single procedure instantiation

Stack Frames

Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)

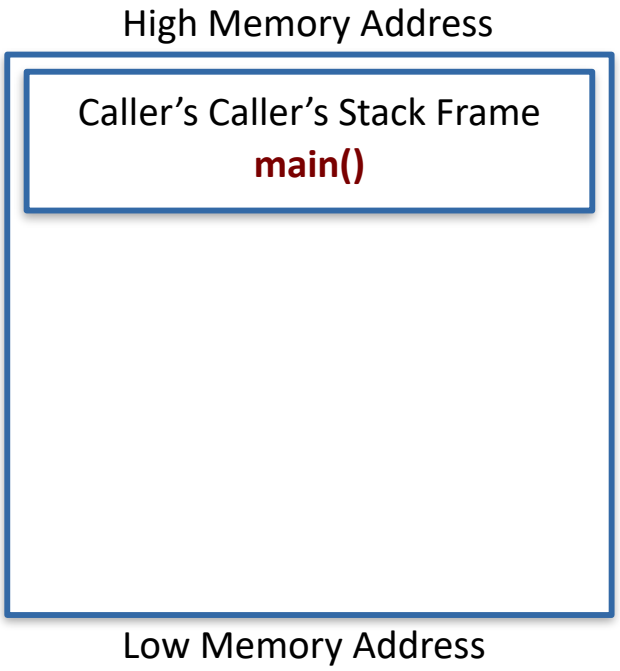


Management

- Space allocated when enter procedure
  - "Set-up" code
  - Includes push by **`call`** instruction
- Deallocated when return
  - "Finish" code
  - Includes pop by **`ret`** instruction

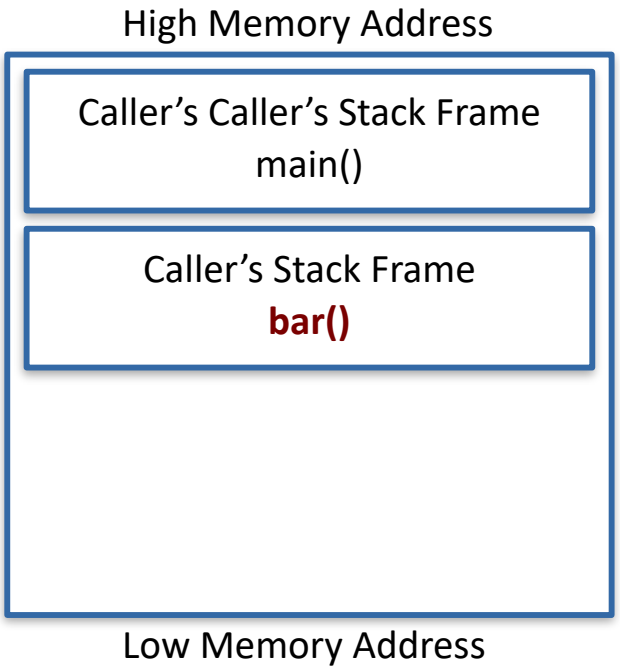
Call Chain Example (Stack Frames)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



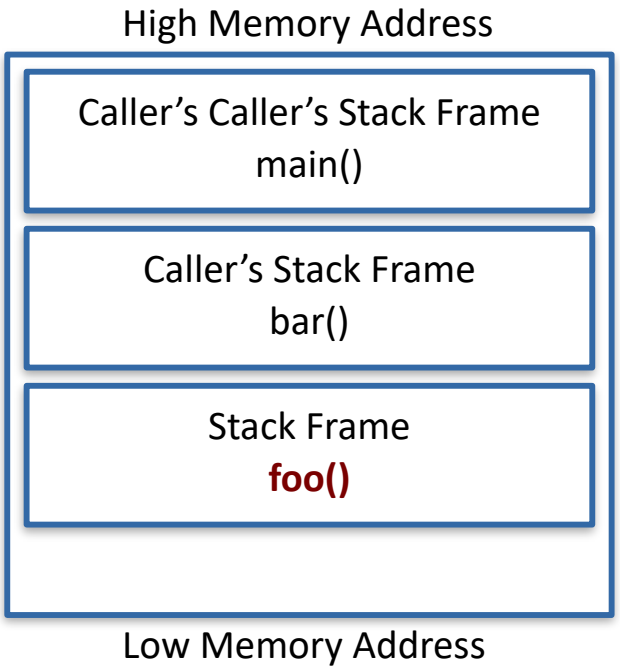
Call Chain Example (Stack Frames)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



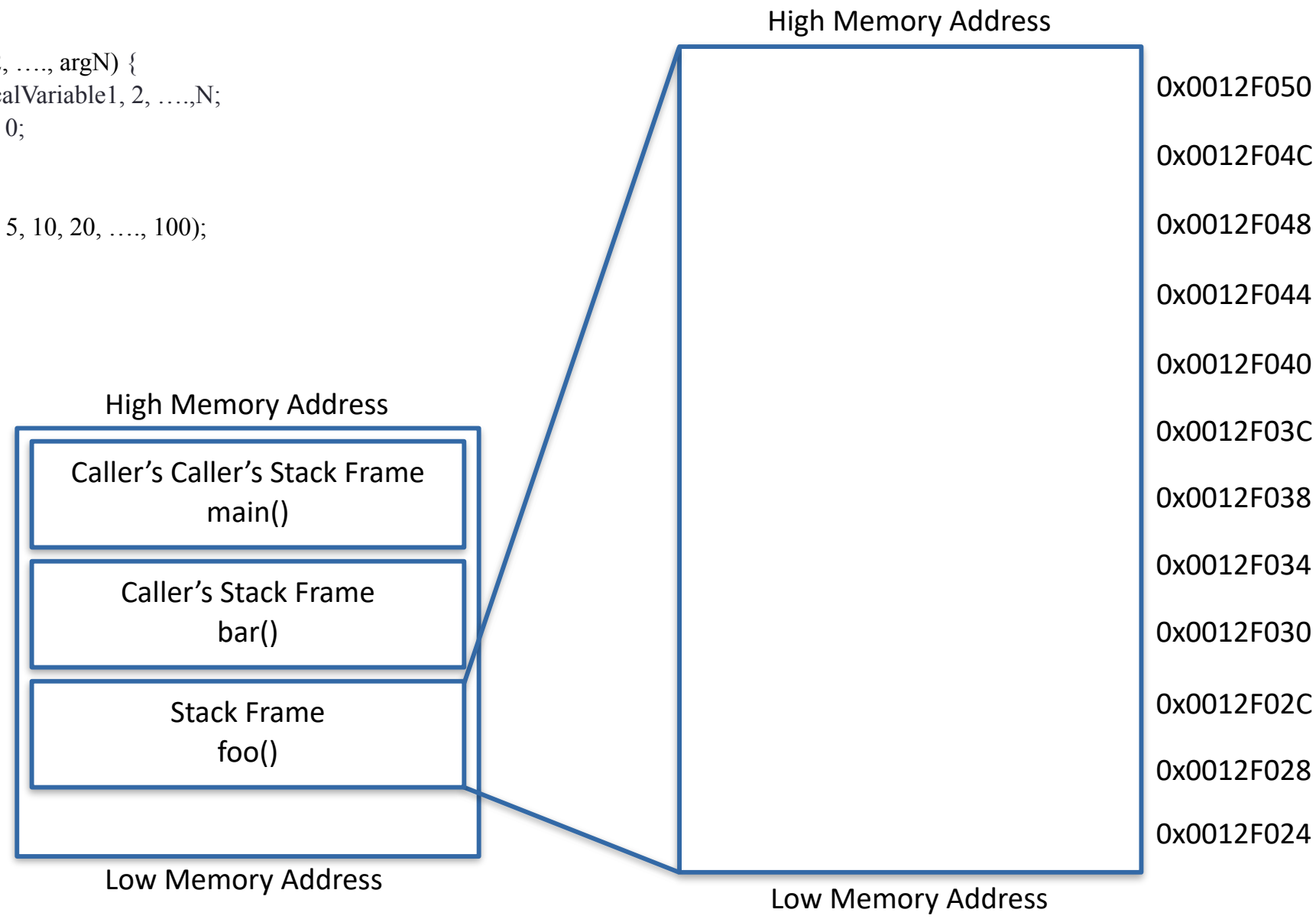
Call Chain Example (Stack Frames)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



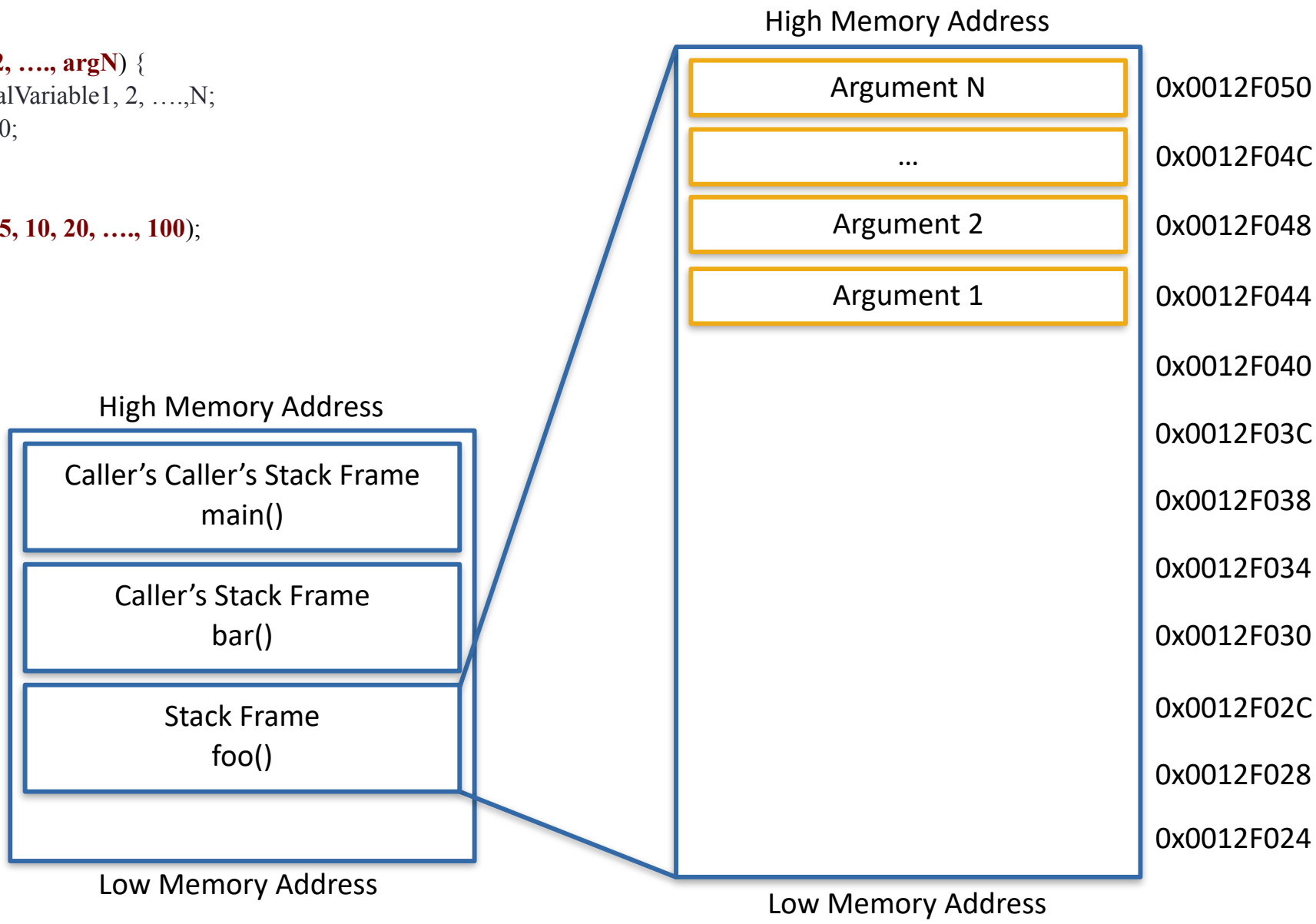
Stack Frame / Activation Record (x86)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



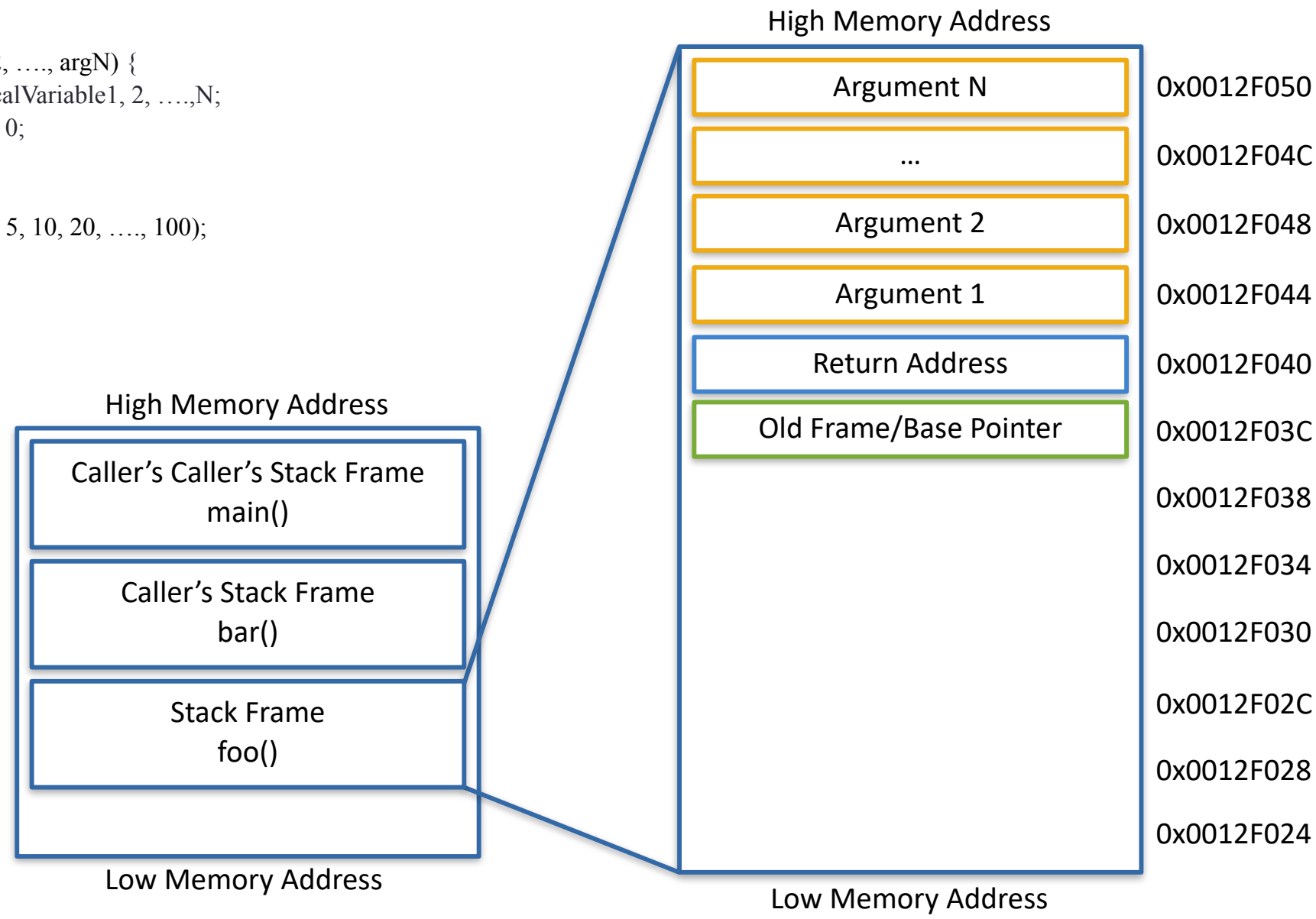
Stack Frame / Activation Record (x86)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



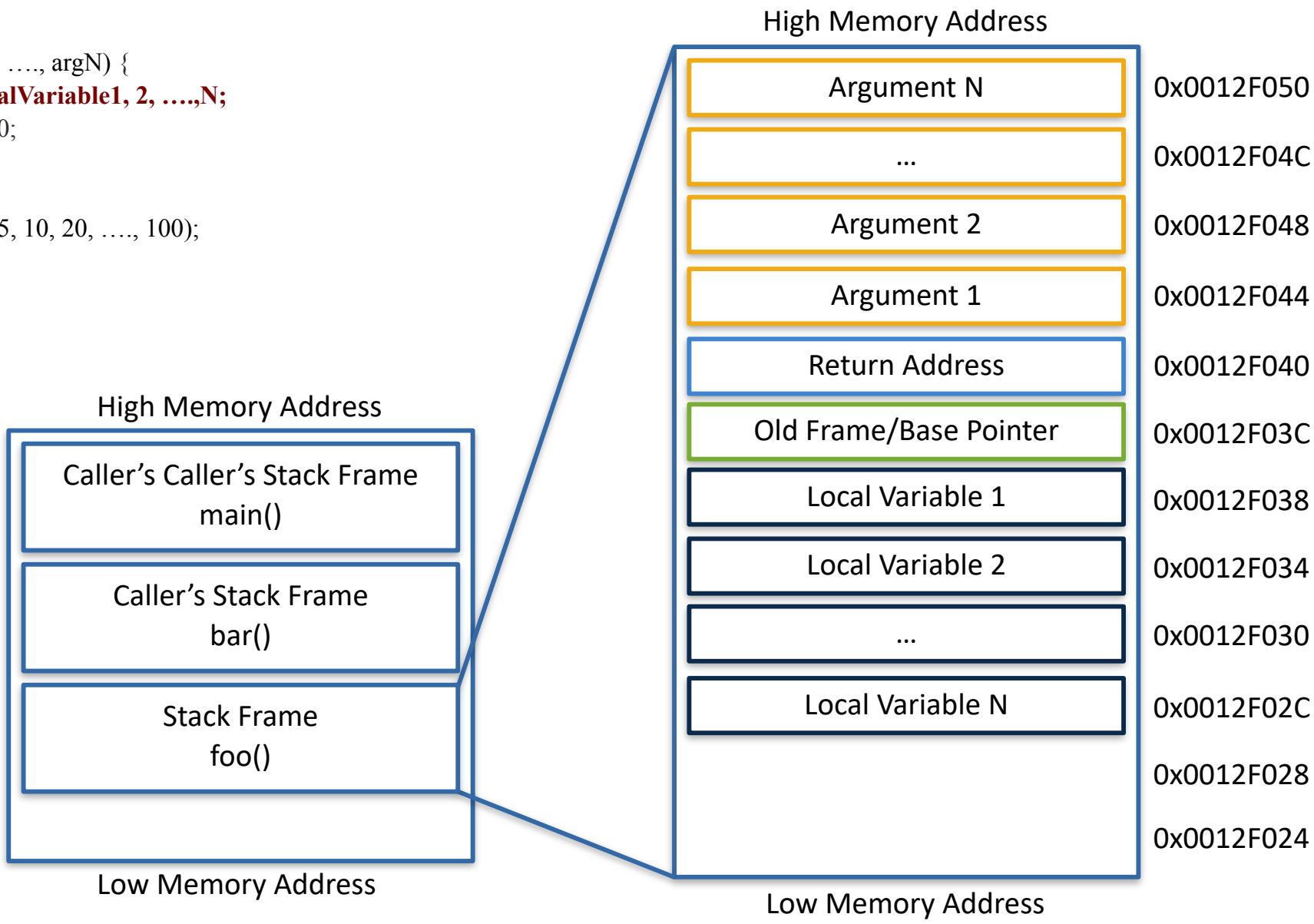
Stack Frame / Activation Record (x86)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



Stack Frame / Activation Record (x86)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



Stack/Base/Instruction Pointers (Summary)

SP - Stack Pointer

- Points to the current **top of the stack**

IP – Instruction Pointer

- Tells the CPU what to do **next**
  - Contains the memory address of the next program instruction to be executed

BP - Base Pointer (aka frame pointer)

- Used to access function parameters and local variables within current stack frame
- **In x86-64**: it is **optional**: most of the time, the **stack knows how much needs to be allocated** for local variables and additional arguments (beyond 6 arguments) and uses simple arithmetics to add or subtract constant number of bytes from SP

Example Stack Frame (x86 - Intel Format)

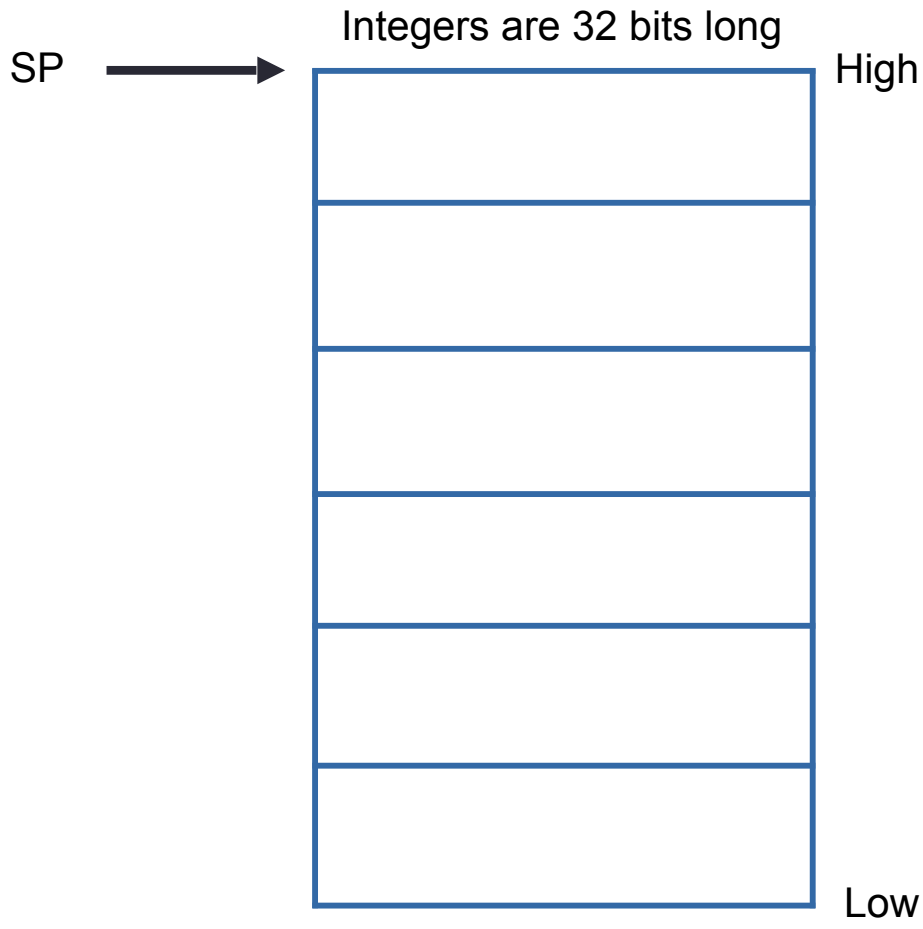
```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}

void main() {
        func(3, 6);
}
```

Example Stack Frame (x86 - Intel Format)

```
void main() {
        func(3, 6);
}
```

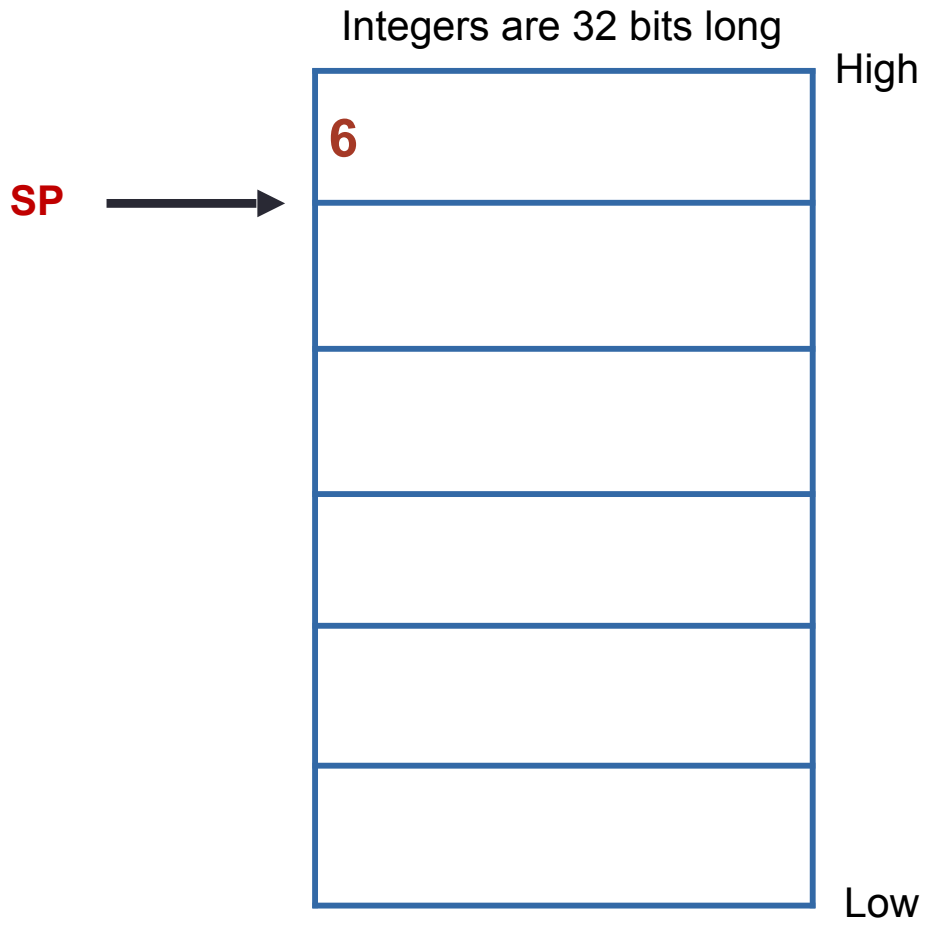
```
30:     6a 06           push   0x6
32:     6a 03           push   0x3
34:     e8 fc ff ff ff  call   35
39:     83 c4 08        add    sp,0x8
3c:     90              nop
3d:     c9              leave
3e:     c3              ret
```



Example Stack Frame (x86 - Intel Format)

```
void main() {
        func(3, 6);
}
```

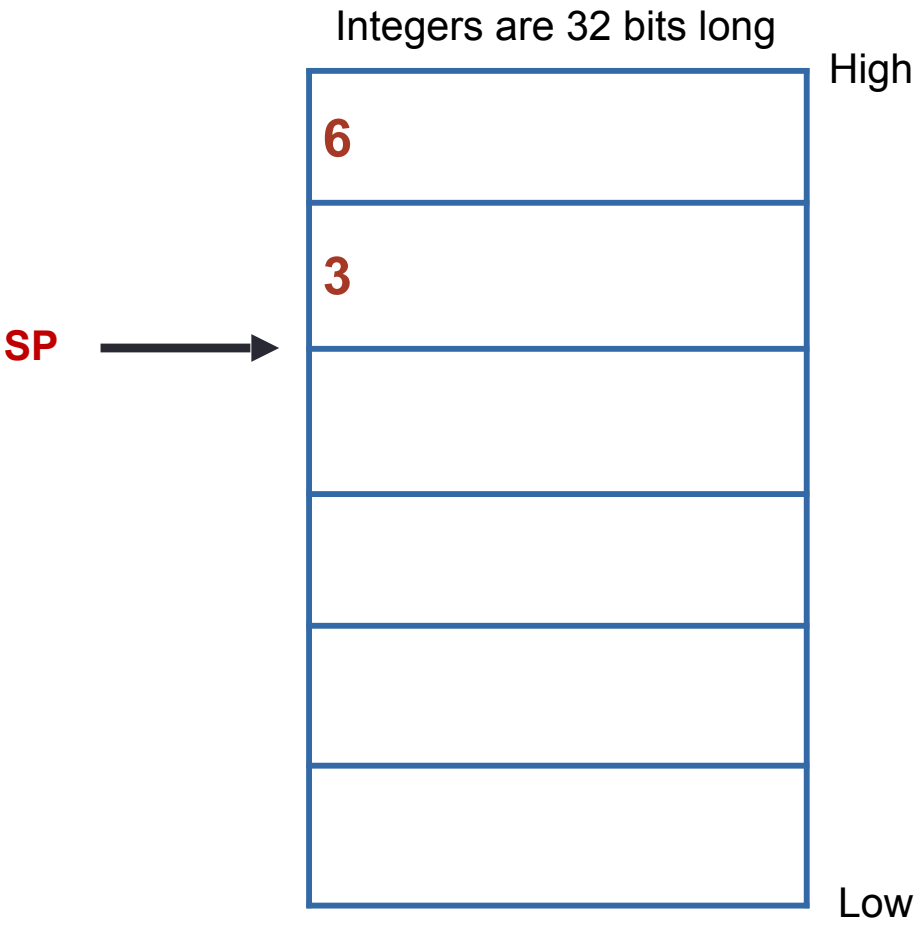
```
30:     6a 06           push   0x6
32:     6a 03           push   0x3
34:     e8 fc ff ff ff  call   35
39:     83 c4 08        add    sp,0x8
3c:     90              nop
3d:     c9              leave
3e:     c3              ret
```



Example Stack Frame (x86 - Intel Format)

```
void main() {
        func(3, 6);
}
```

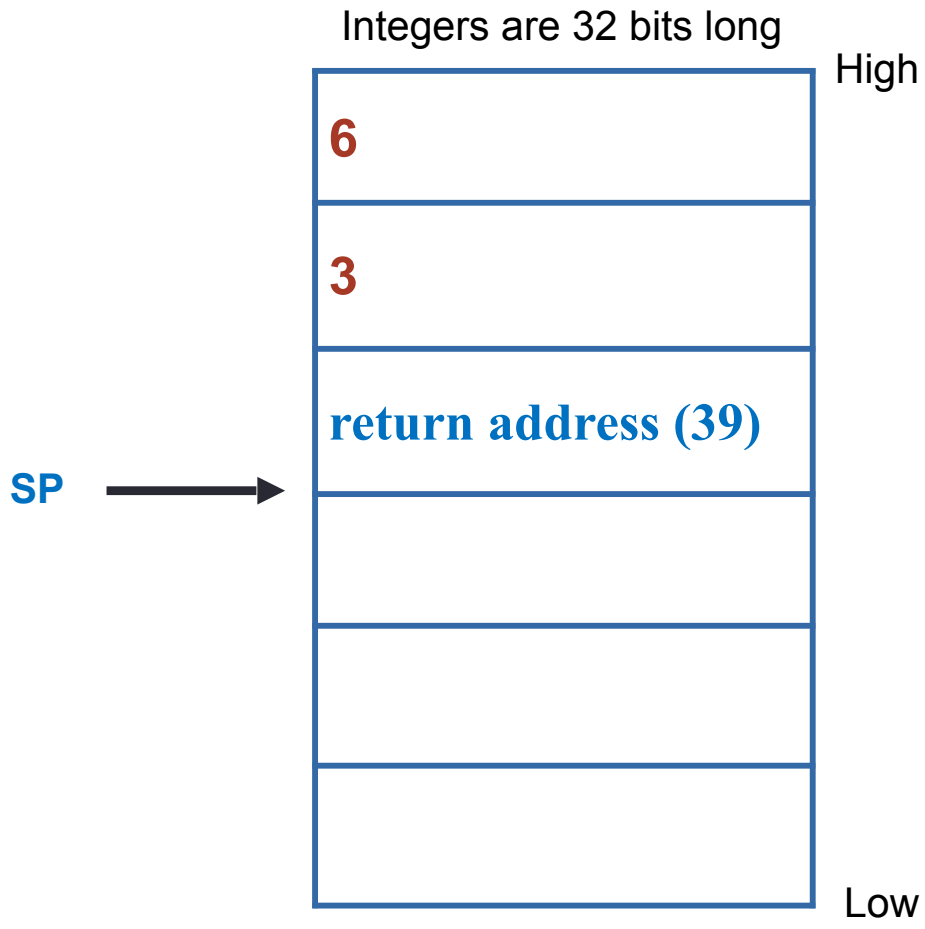
```
30:     6a 06           push   0x6
32:     6a 03           push   0x3
34:     e8 fc ff ff ff  call   35
39:     83 c4 08        add    sp,0x8
3c:     90              nop
3d:     c9              leave
3e:     c3              ret
```



Example Stack Frame (x86 - Intel Format)

```
void main() {
        func(3, 6);
}
```

```
30:     6a 06           push   0x6
32:     6a 03           push   0x3
34:     e8 fc ff ff ff  call   35
39:     83 c4 08        add    sp,0x8
3c:     90              nop
3d:     c9              leave
3e:     c3              ret
```



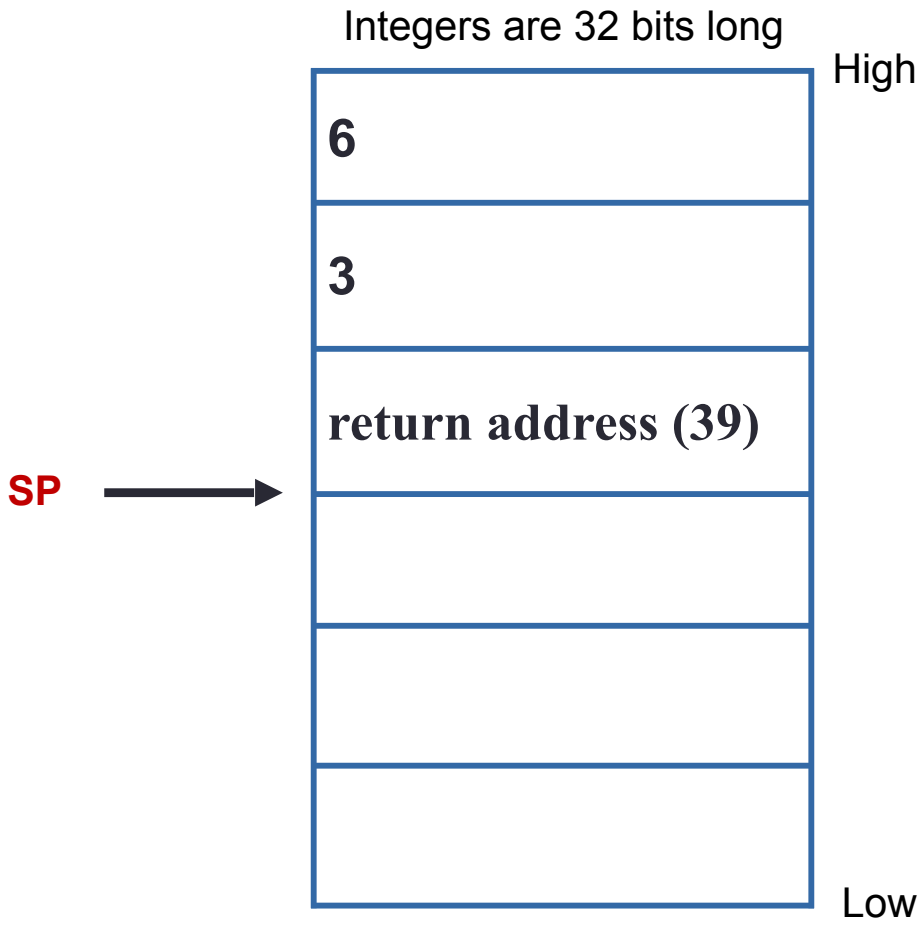
; IP updates to point to first instruction in func

Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Prologue:

```
  0:    55          push   bp
```

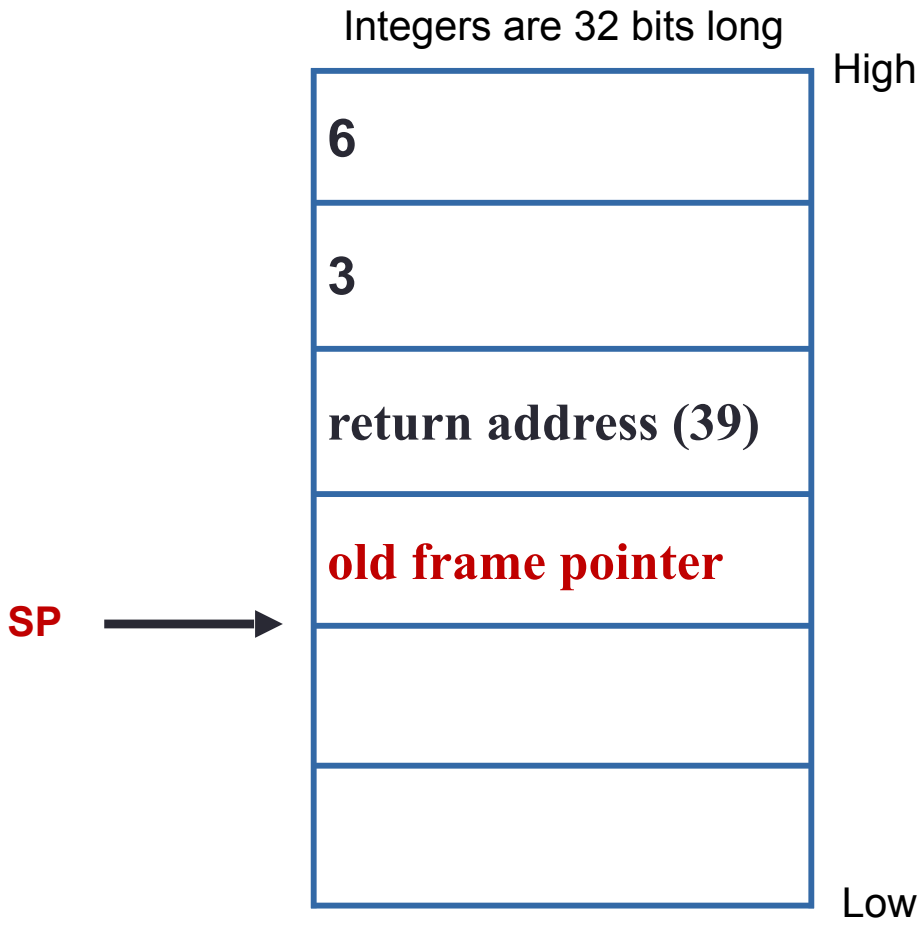


Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Prologue:

0: 55 push bp

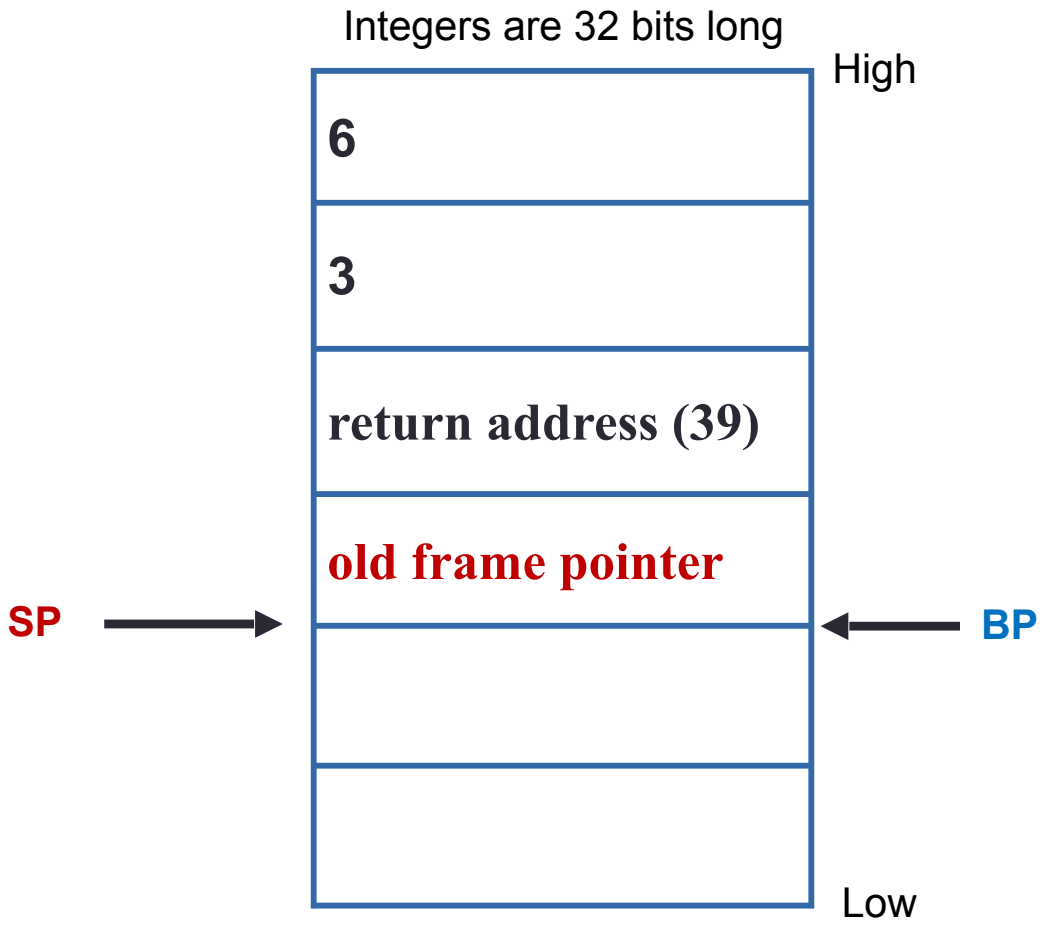


Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Prologue:

```
0:    55       push   bp
1:    89 e5    mov    bp,sp
```

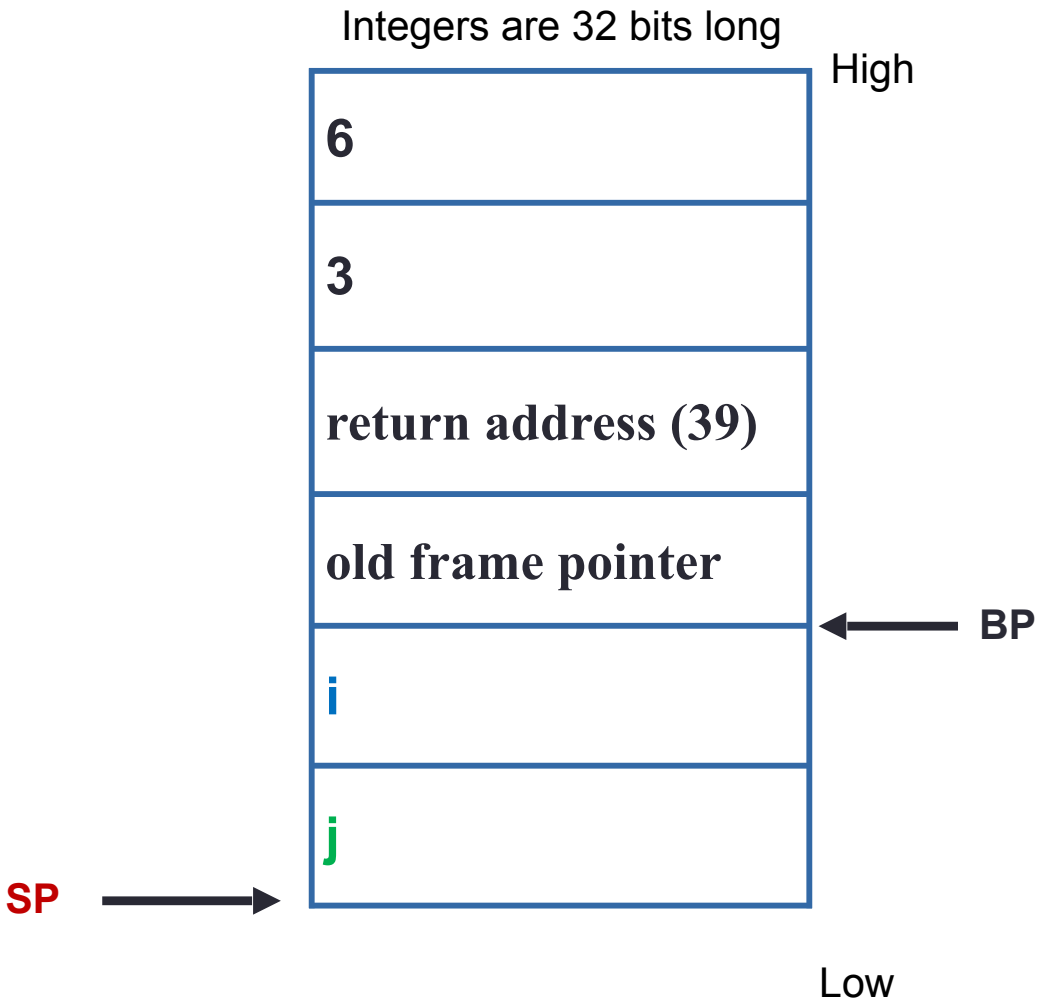


Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Execution:

```
sub   sp,0x8
mov   eax,DWORD PTR [bp+0x8]
mov   DWORD PTR [bp-0x4],eax
mov   eax,DWORD PTR [bp+0xC]
mov   DWORD PTR [bp-0x8],eax
```

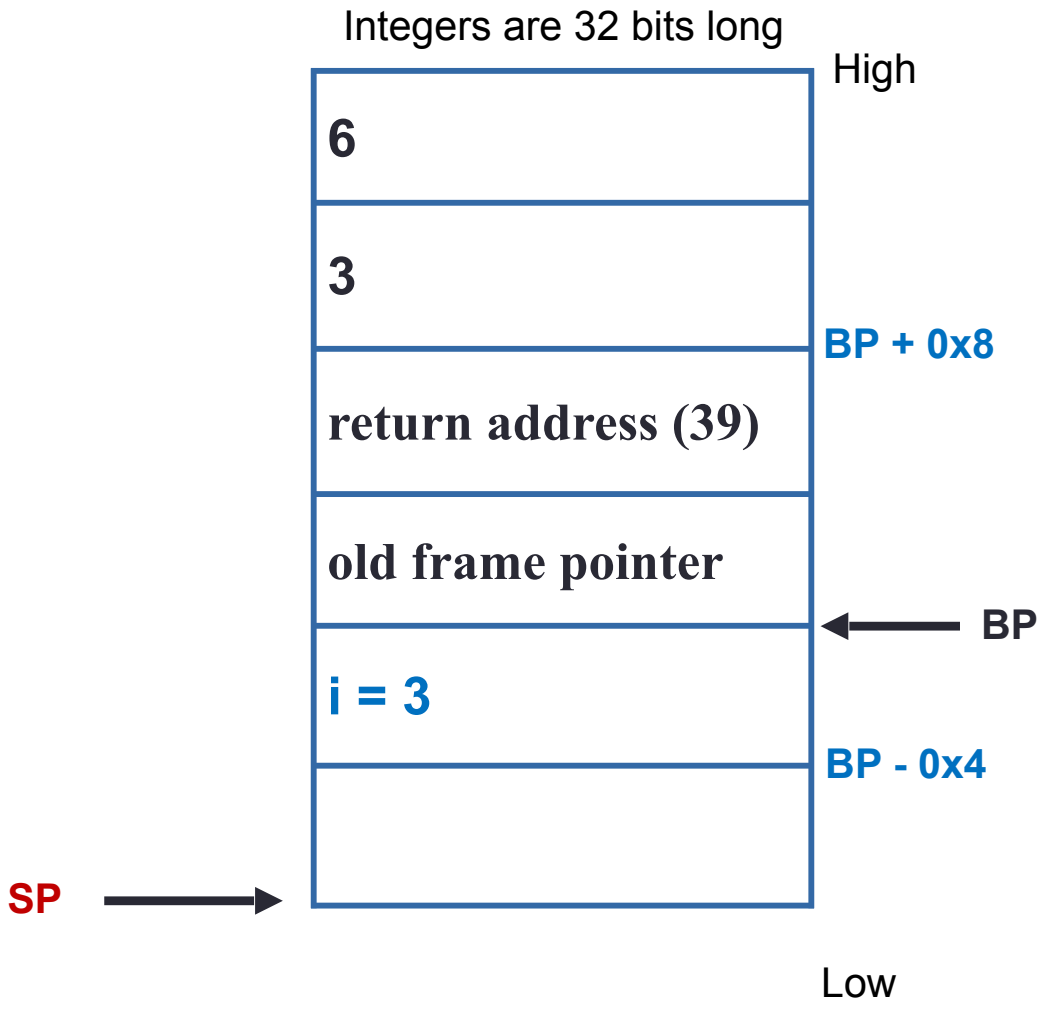


Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Execution:

```
sub   sp,0x8
mov   eax,DWORD PTR [bp+0x8]
mov   DWORD PTR [bp-0x4],eax
mov   eax,DWORD PTR [bp+0xC]
mov   DWORD PTR [bp-0x8],eax
```

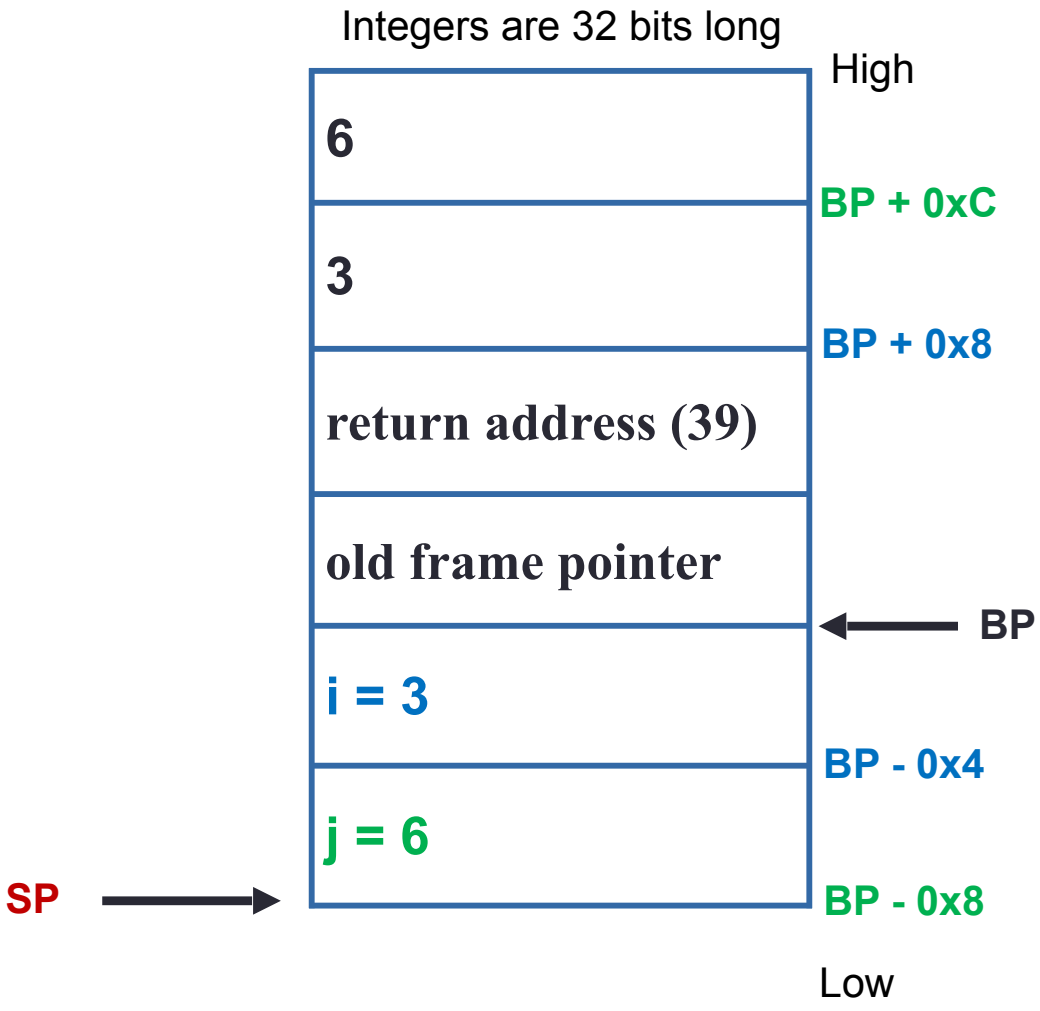


Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

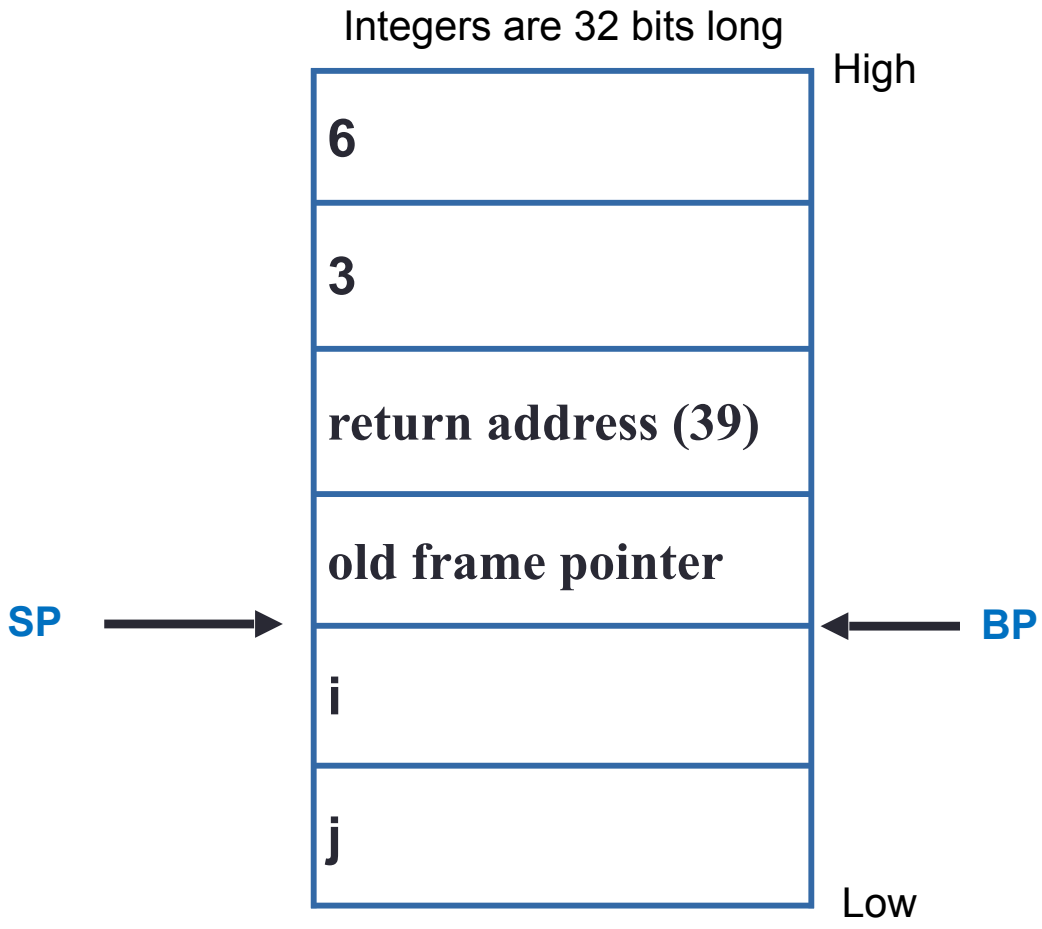
Function Execution:

```
sub   sp,0x8
mov   eax,DWORD PTR [bp+0x8]
mov   DWORD PTR [bp-0x4],eax
mov   eax,DWORD PTR [bp+0xC]
mov   DWORD PTR [bp-0x8],eax
```



Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

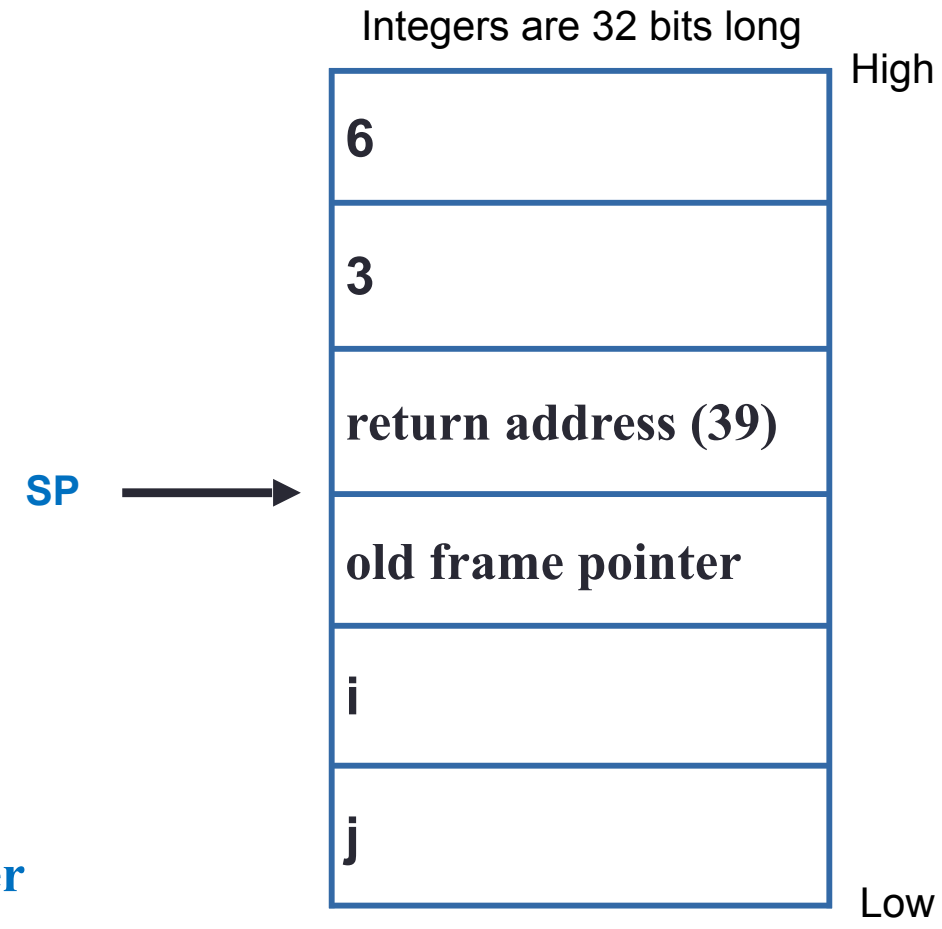


Function Epilogue:

```
mov sp, bp
pop bp
ret
```

Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```



Function Epilogue:

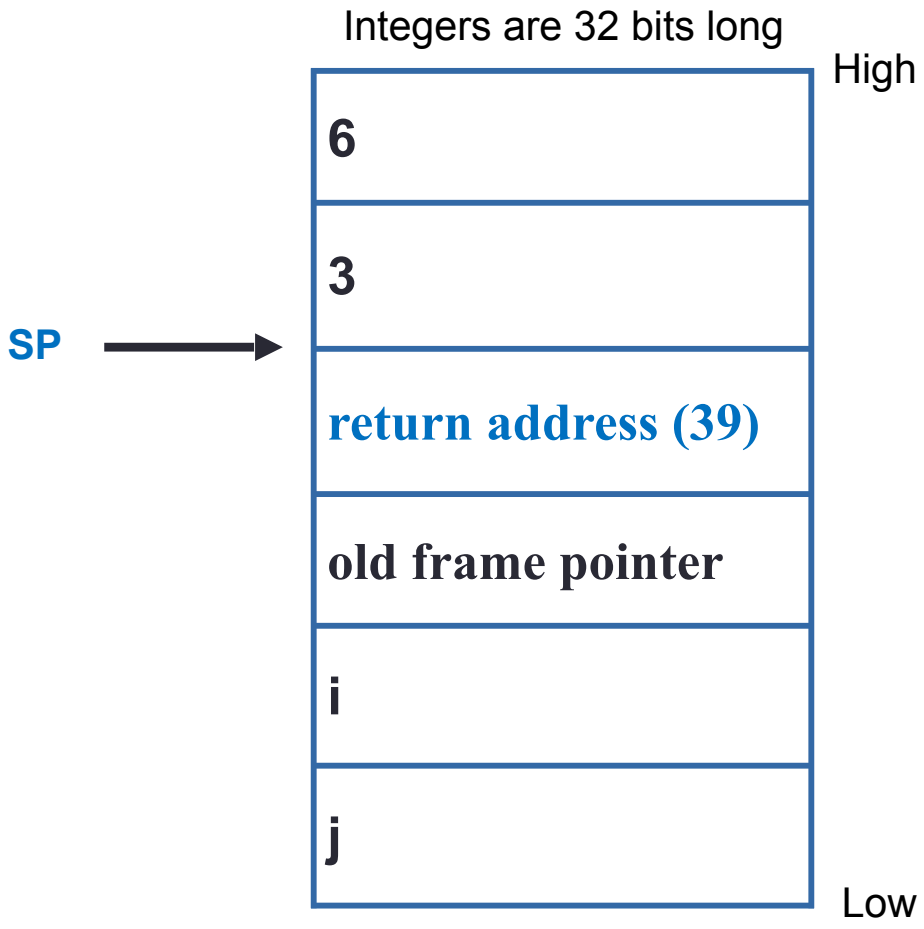
mov sp, bp

pop bp ; now bp has old frame pointer

ret

Example Stack Frame (x86 - Intel Format)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```



Function Epilogue:

mov sp, bp

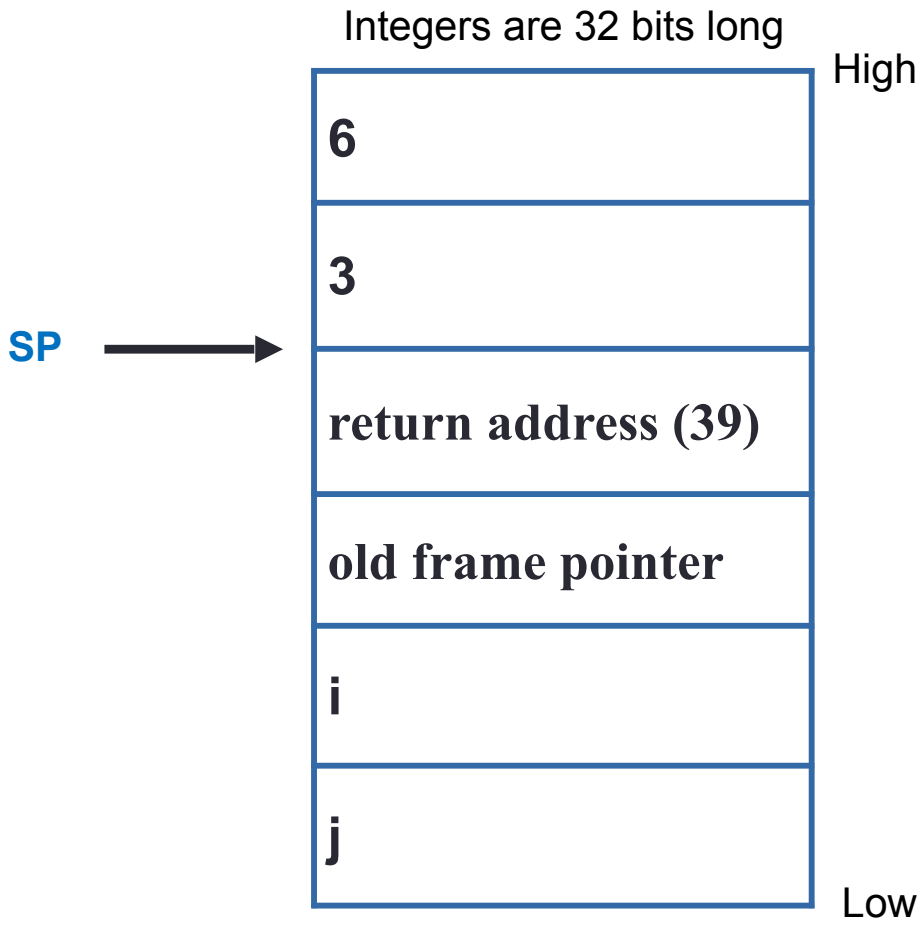
pop bp

ret ; pops return address (39) and places it in IP (i.e., instruction pointer)

Example Stack Frame (x86 - Intel Format)

```
void main() {
        func(3, 6);
}
```

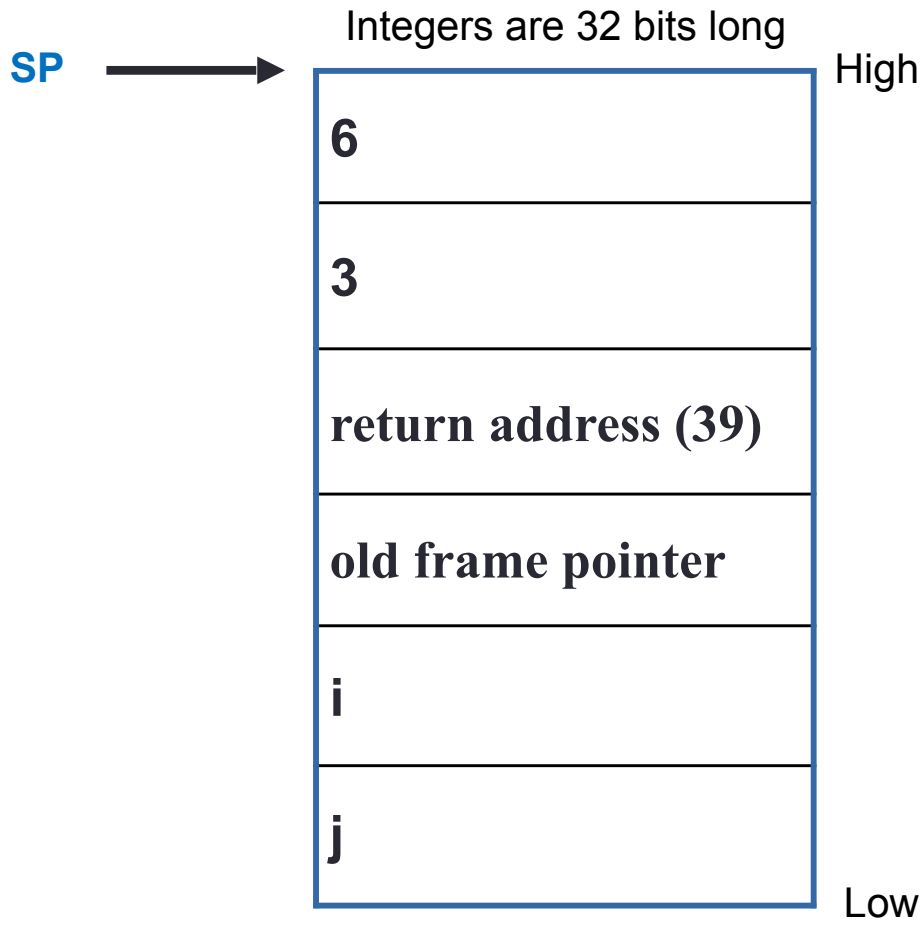
```
 30:    6a 06          push   0x6
 32:    6a 03          push   0x3
 34:    e8 fc ff ff ff call   35
 39:    83 c4 08       add    sp,0x8
 3c:    90             nop
 3d:    c9             leave
 3e:    c3             ret
```



Example Stack Frame (x86 - Intel Format)

```
void main() {
        func(3, 6);
}
```

```
 30:    6a 06          push   0x6
 32:    6a 03          push   0x3
 34:    e8 fc ff ff ff call   35
 39:    83 c4 08       add    sp,0x8
 3c:    90             nop
 3d:    c9             leave
 3e:    c3             ret
```



Another Call Chain Example

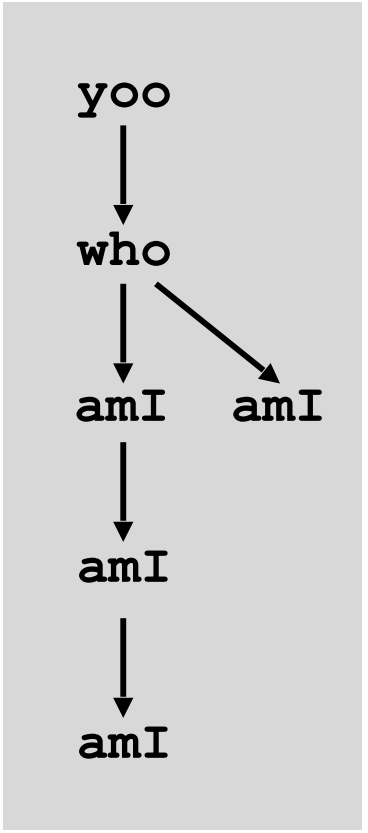
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

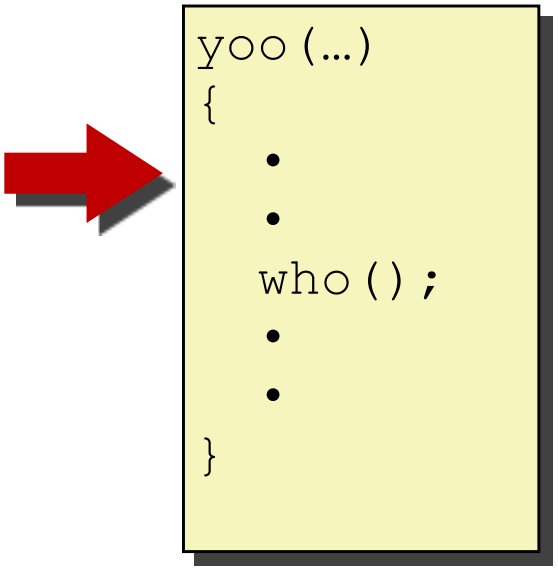
```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

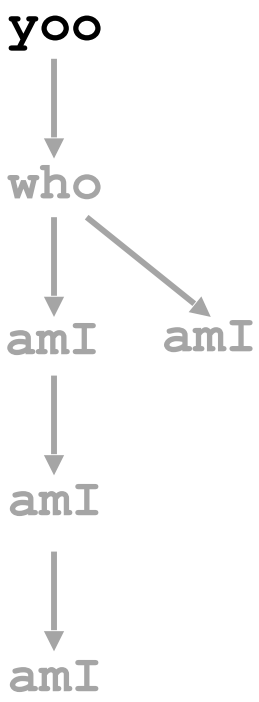
Procedure `amI()` is recursive

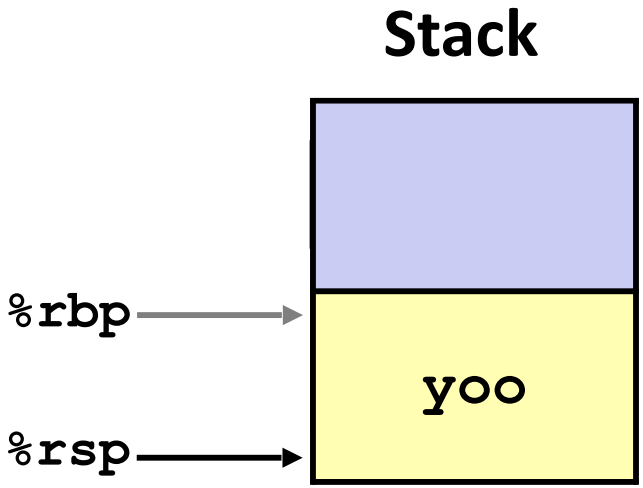
Example Scenario Call Chain



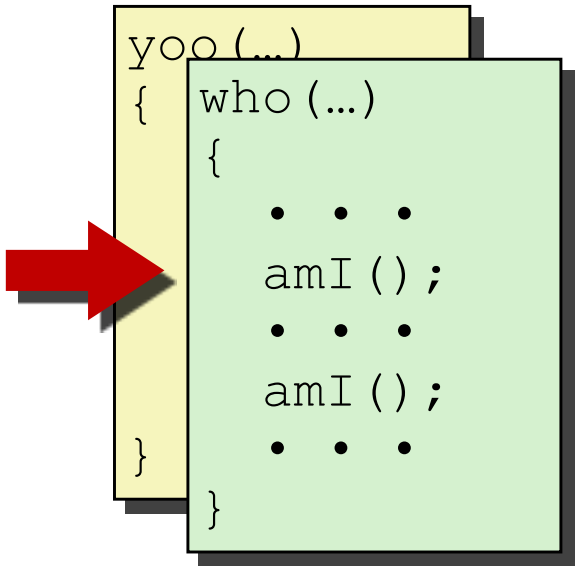
Another Example

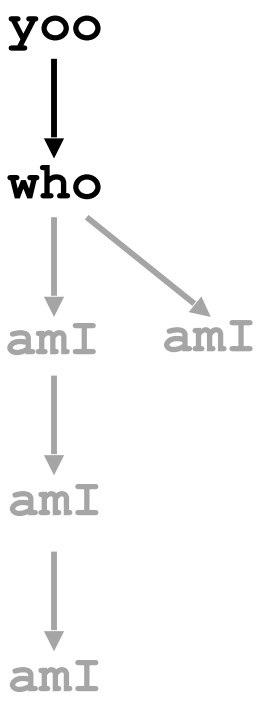


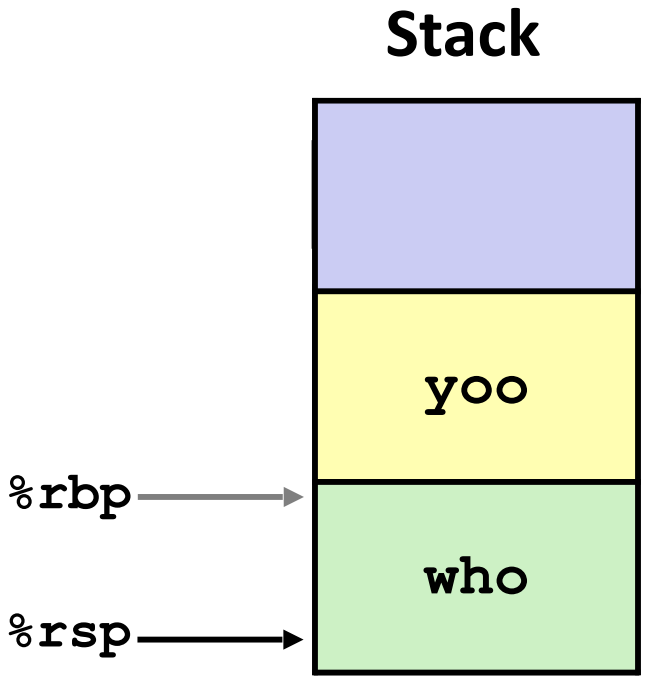




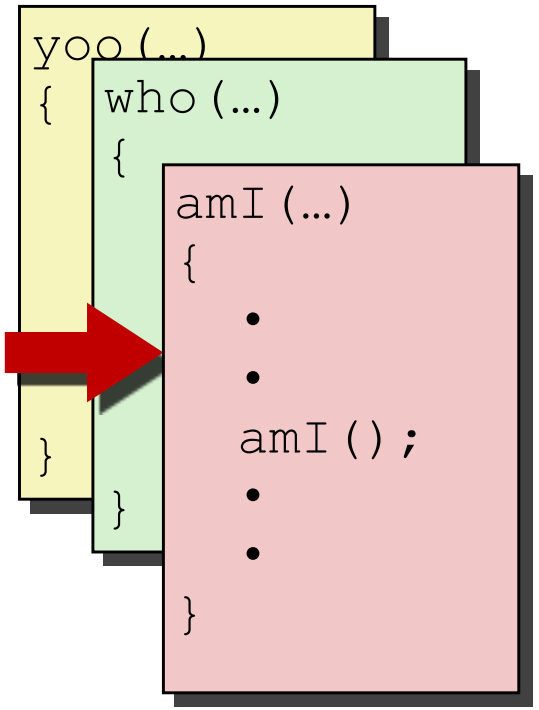
Another Example

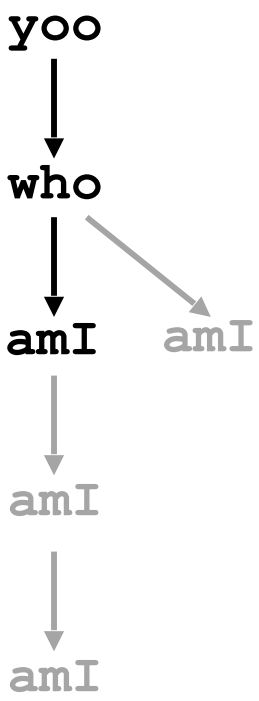


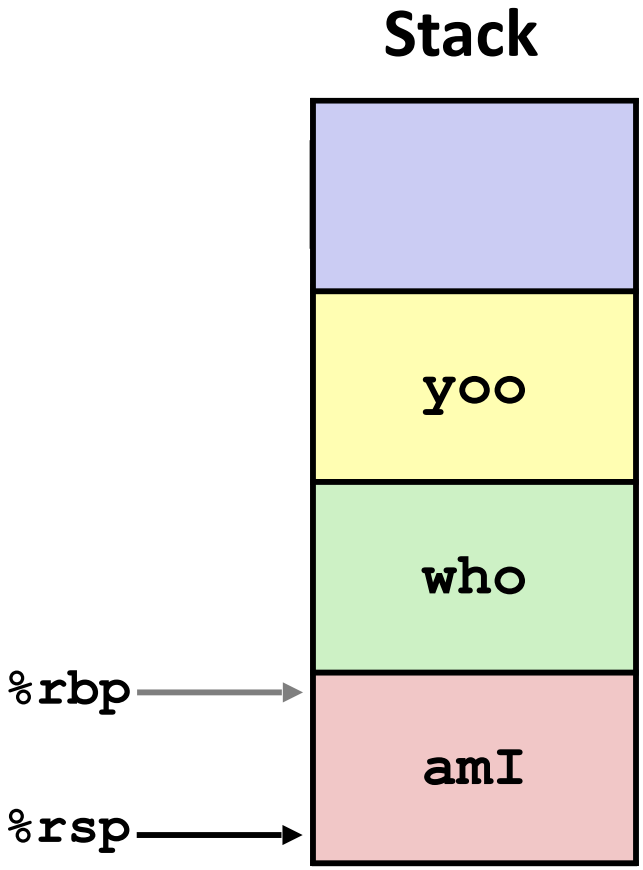




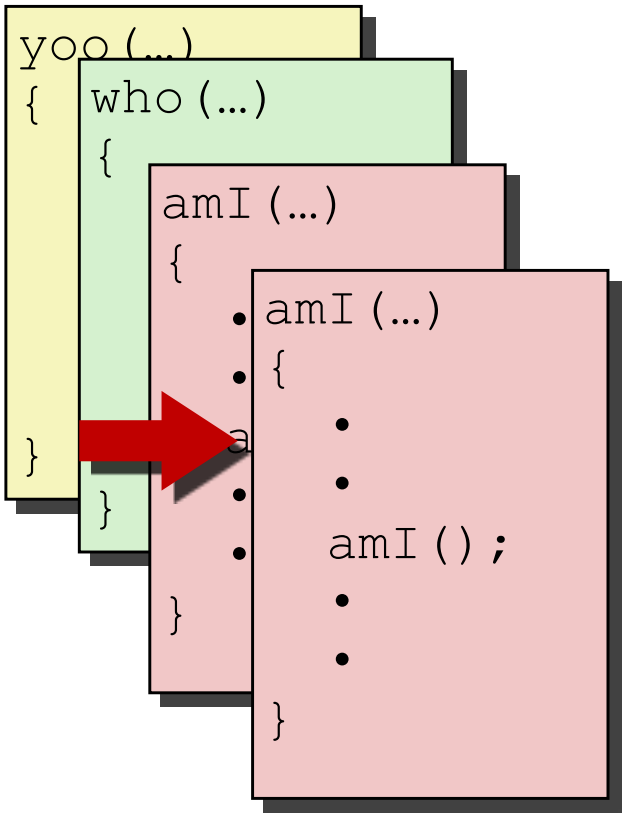
Another Example

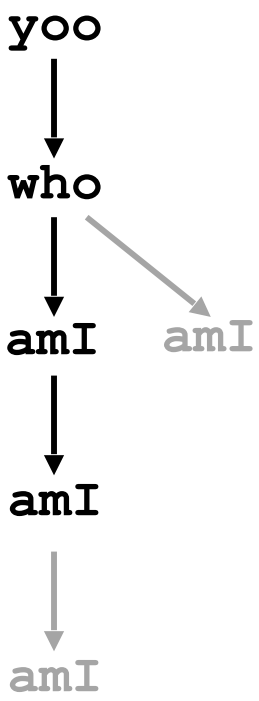


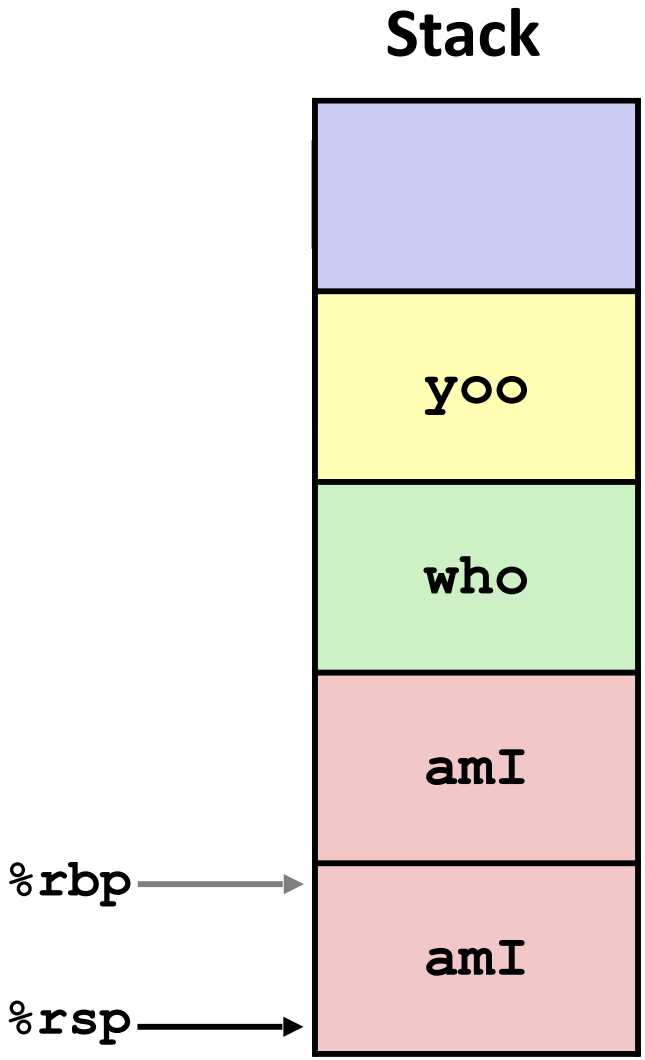




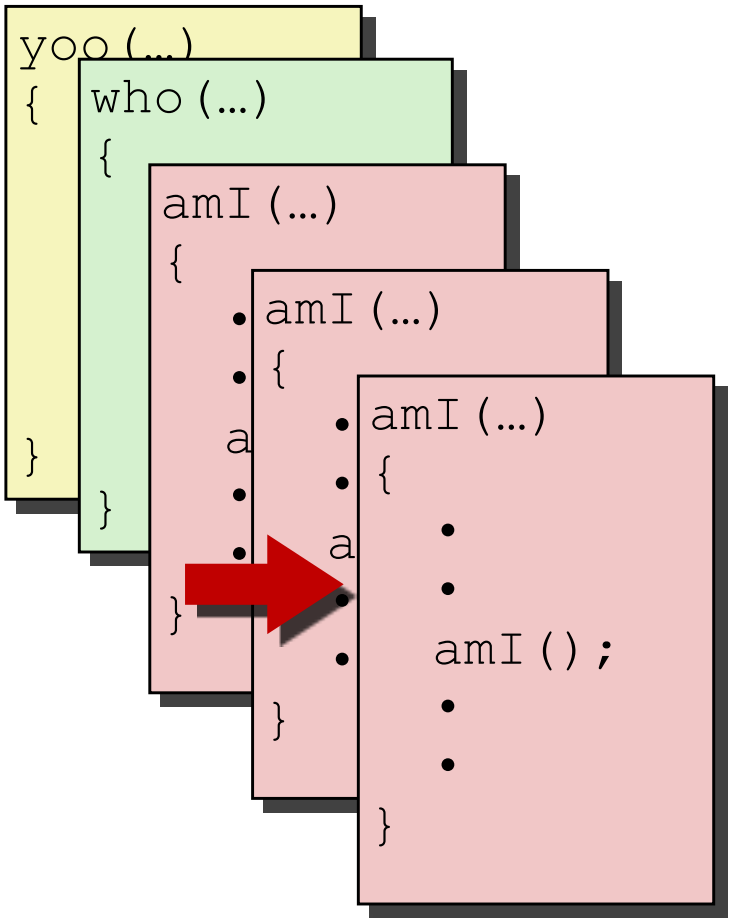
Another Example

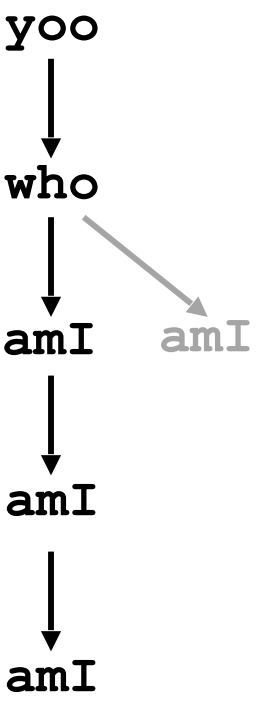


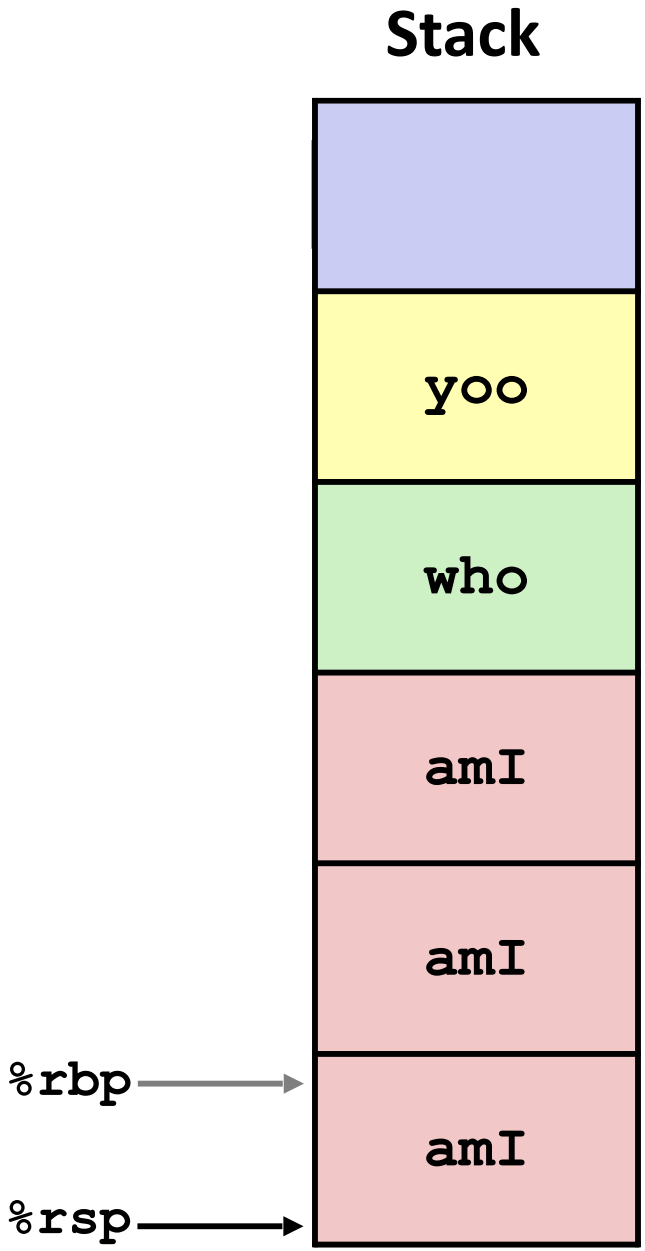




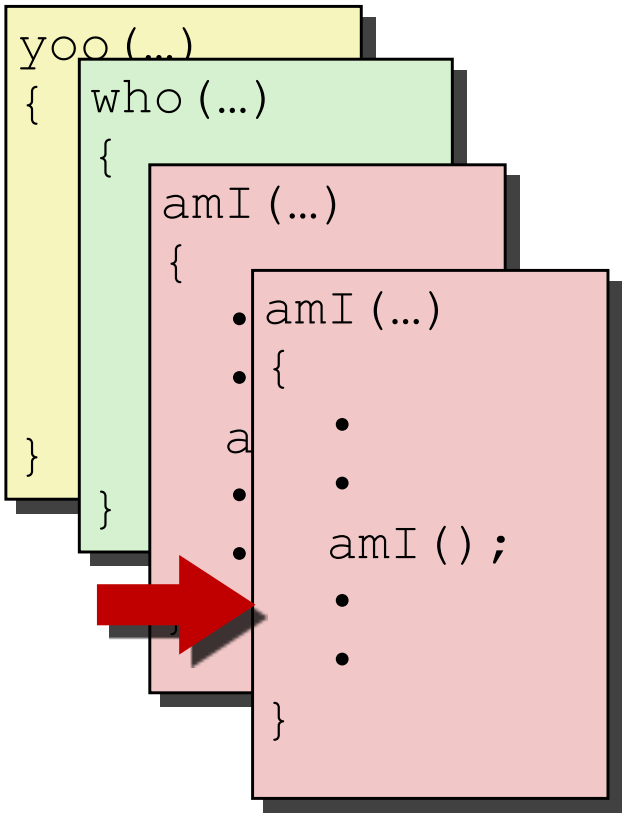
Another Example

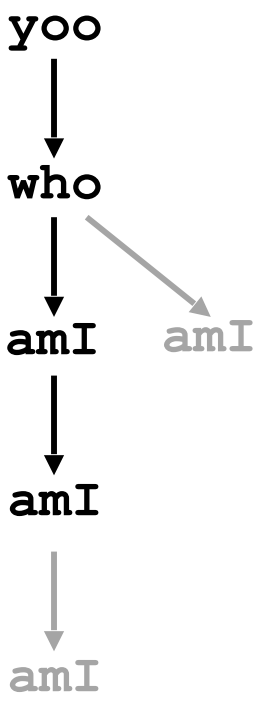


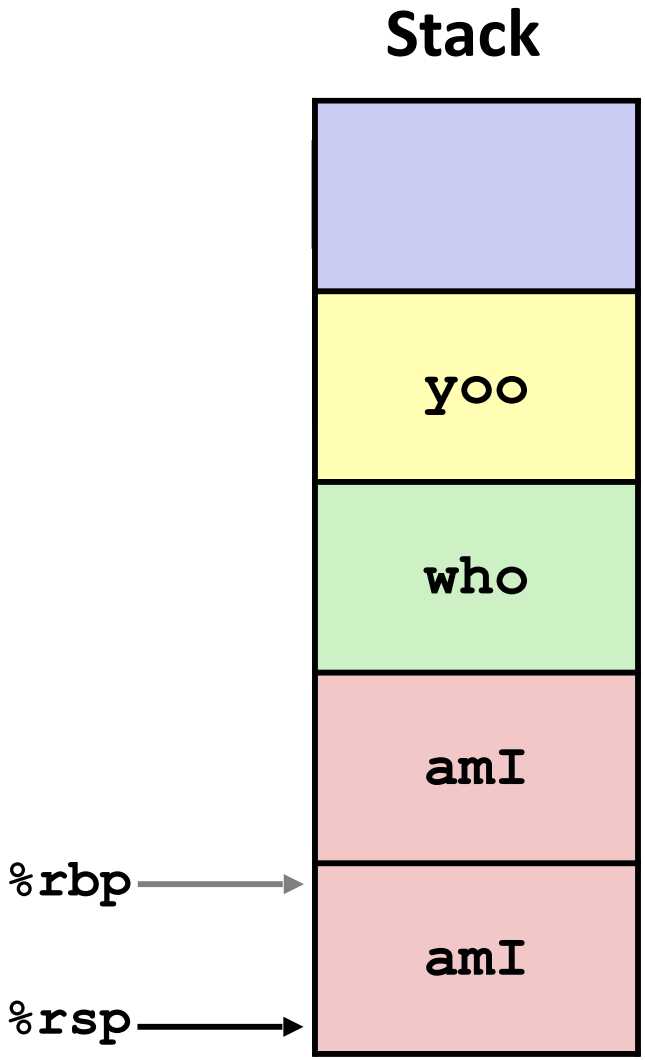




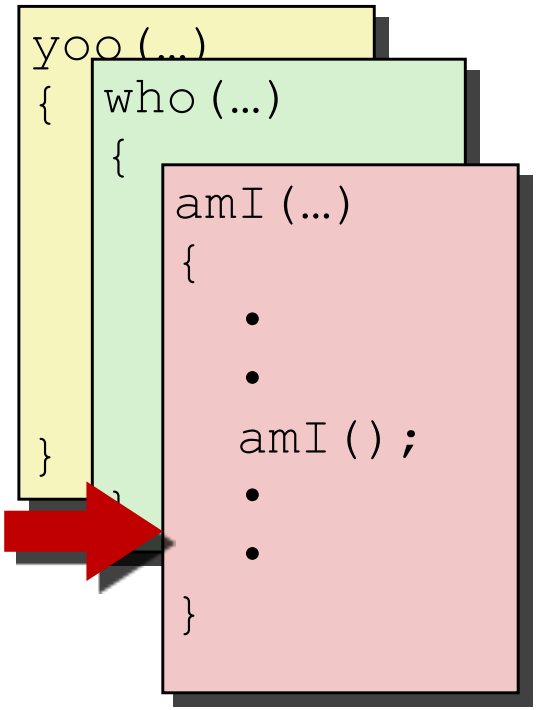
Another Example

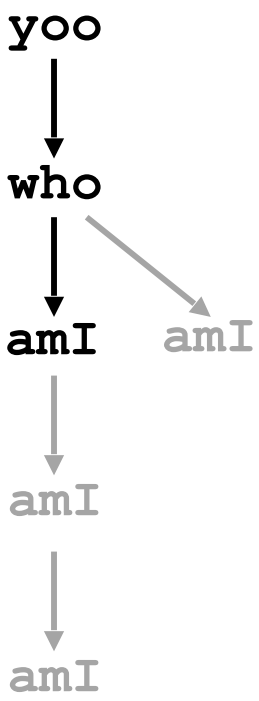


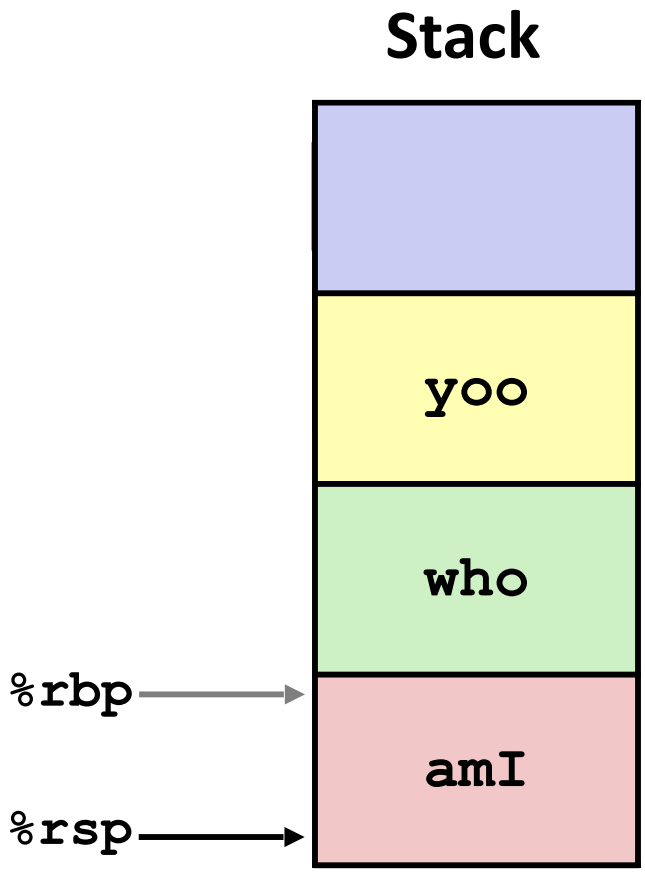




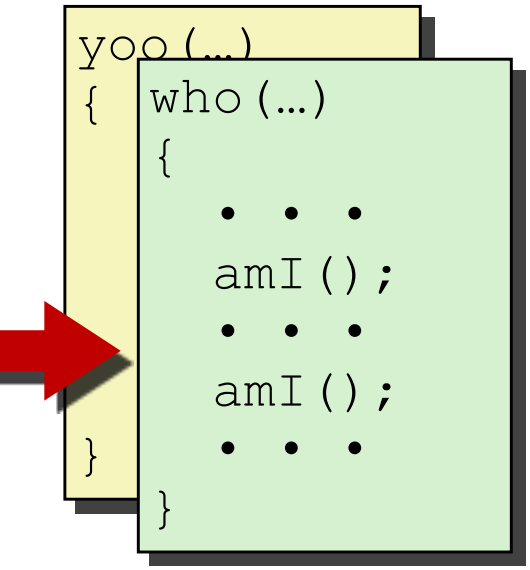
Another Example

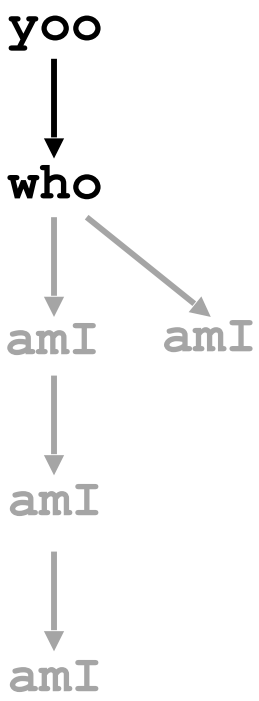


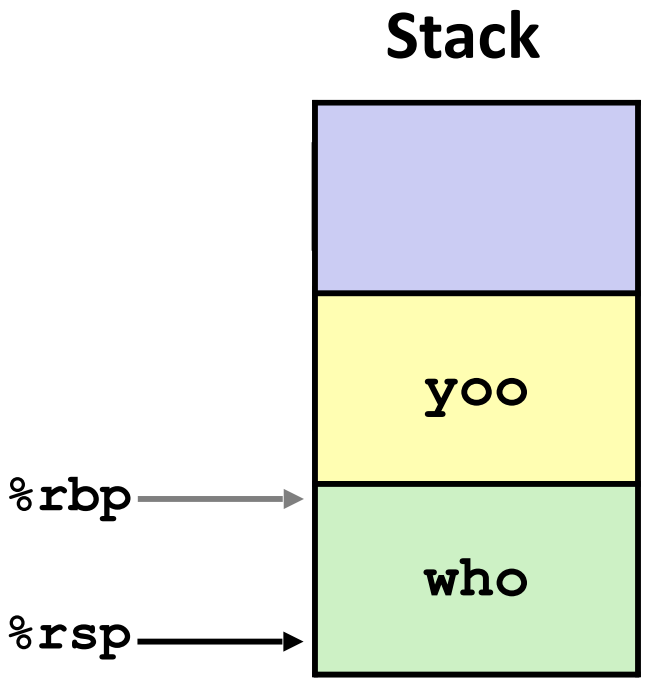




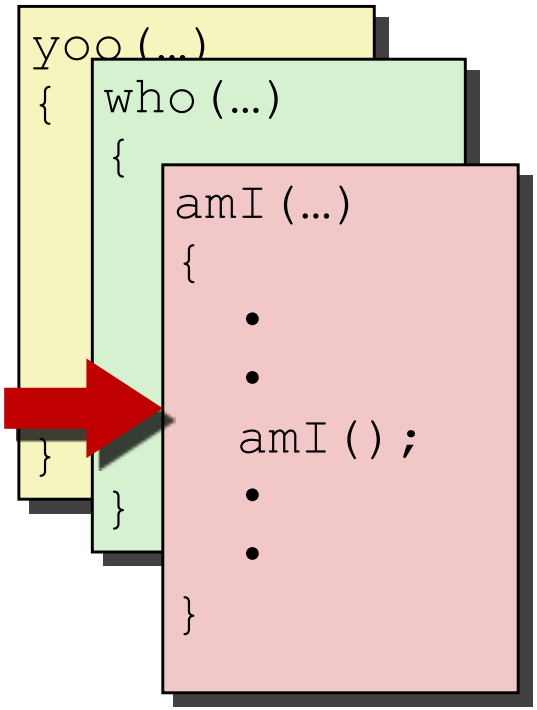
Another Example

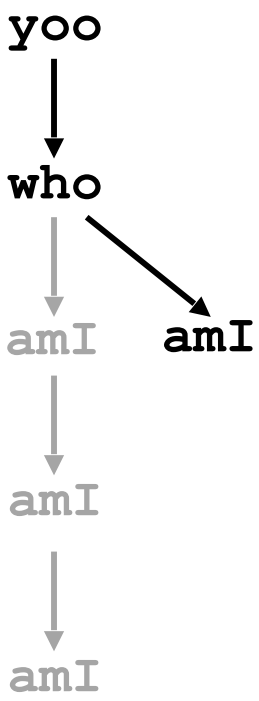


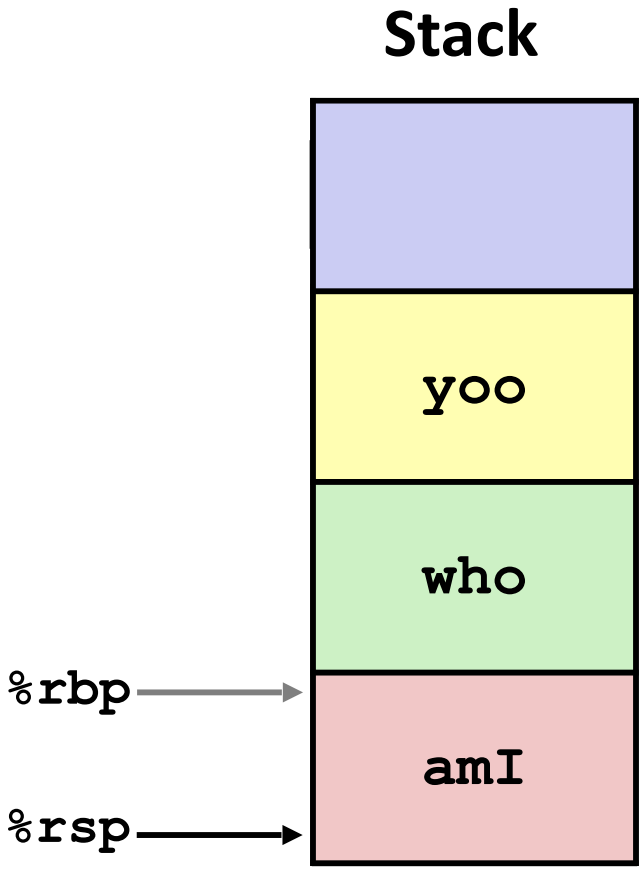




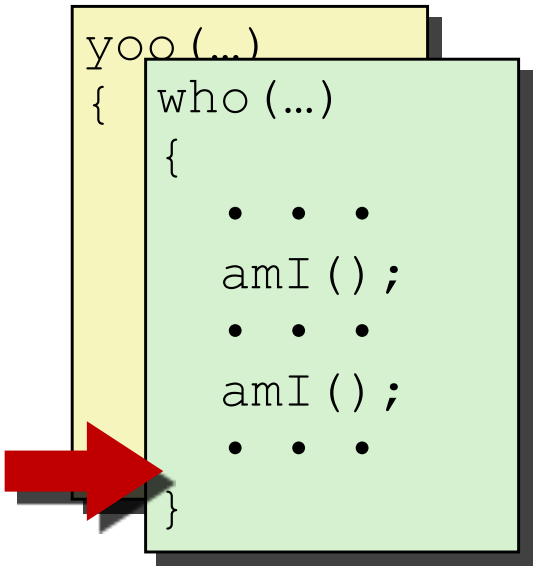
Another Example

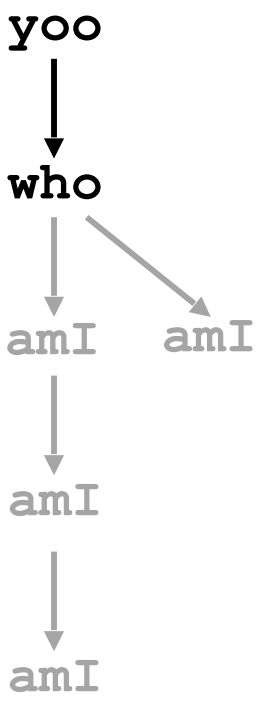


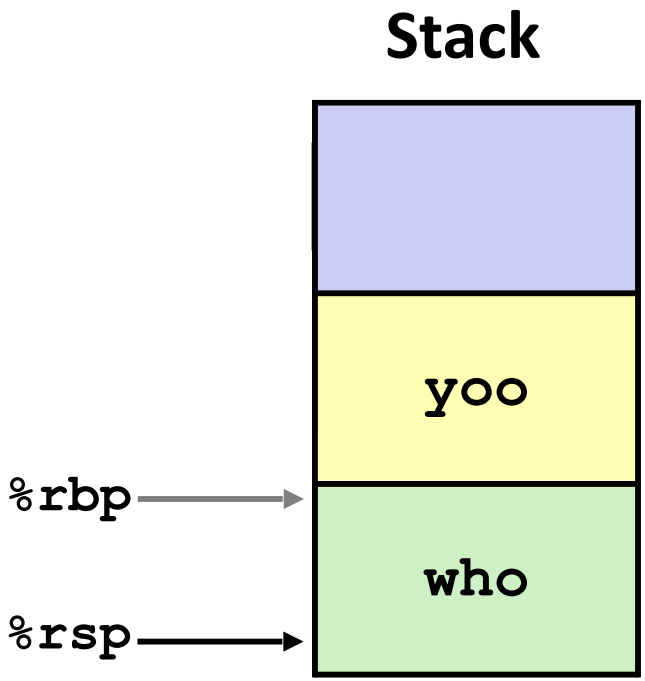




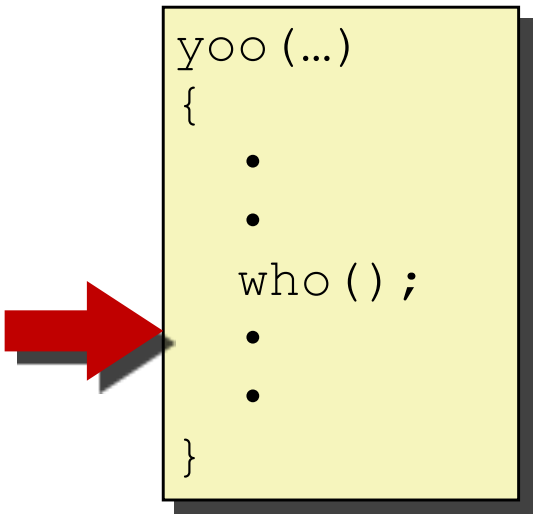
Another Example

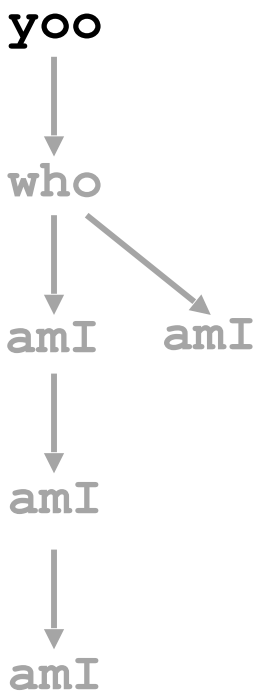


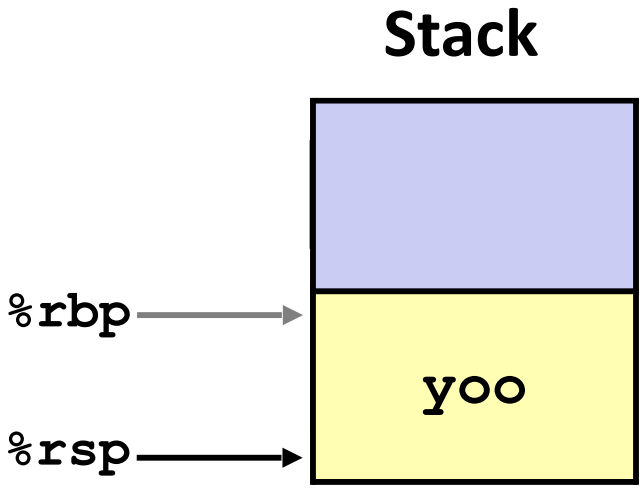




Another Example







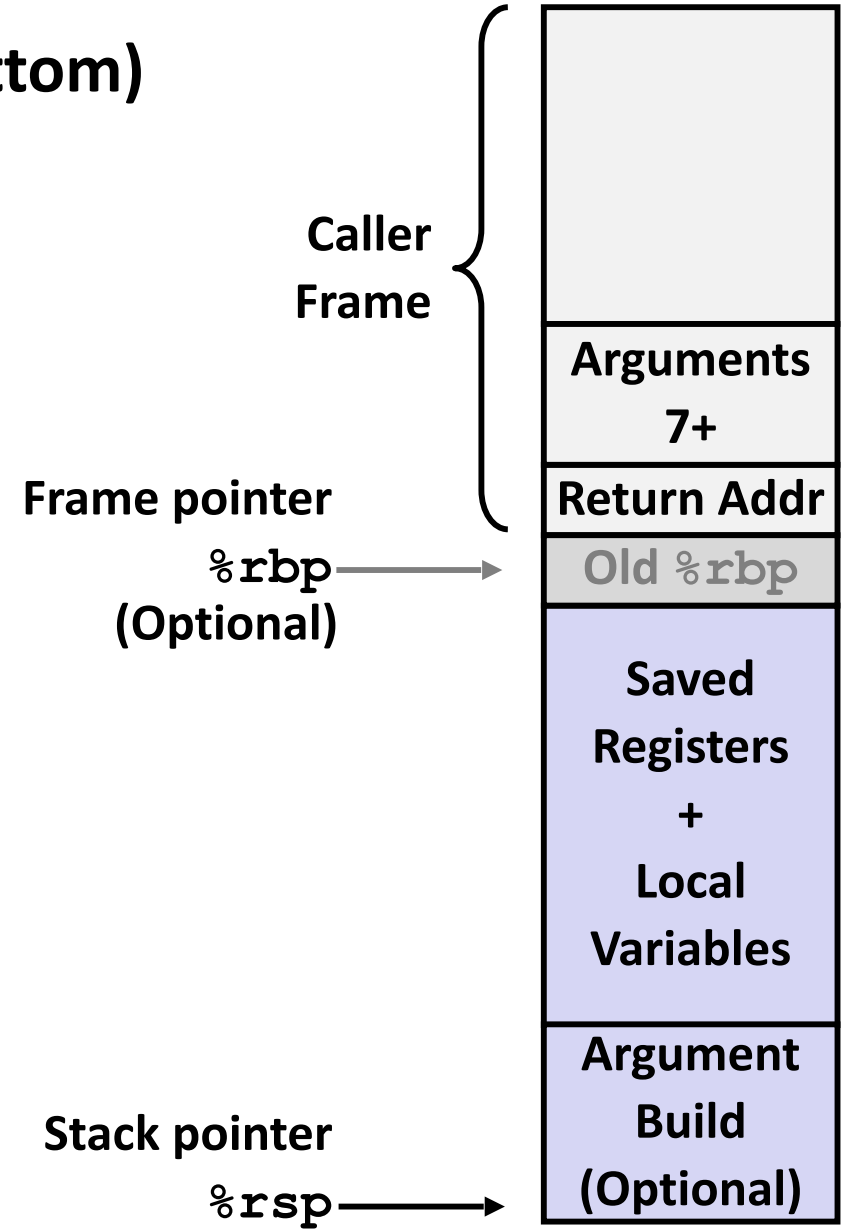
x86-64/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- "Argument build:"
  Parameters for function about to call
- Local variables
  If can't keep in registers
- Saved register context
- Old frame pointer (**optional**)

Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call



x86-64 Example: `incr`

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

Although there are **local variables** here, **nothing was allocated onto the stack** since the compiler managed to store all variables in registers (mainly rdi and rsi)
—> much more efficient

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

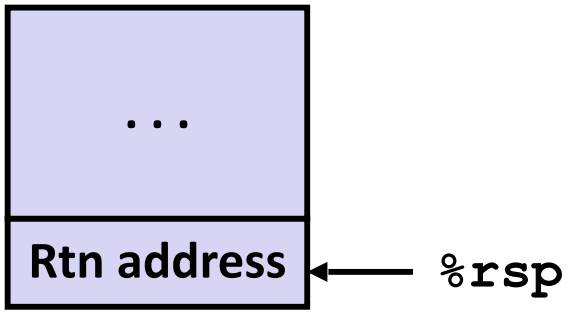
Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	`x`, Return value

x86-64 Example: Calling `incr` #1

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

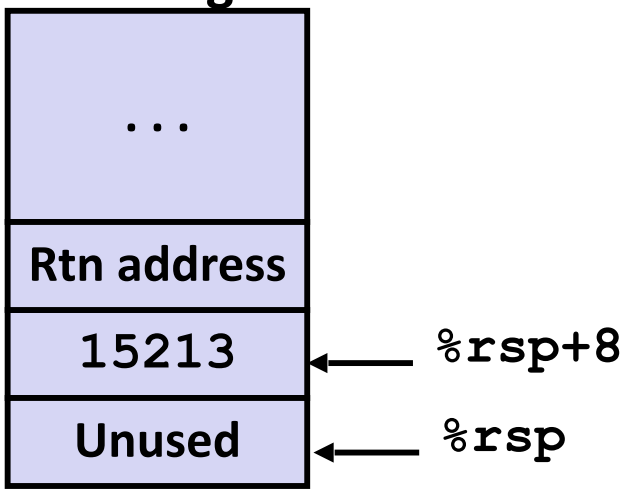
```
call_incr:
  subq    $16, %rsp ;make room
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Compiler decided to move stack pointer to somewhere useful (e.g., preparation for future uses) and use it to find locations of local variable later on.
Alternative: series of **push** instructions (more costly)

Resulting Stack Structure

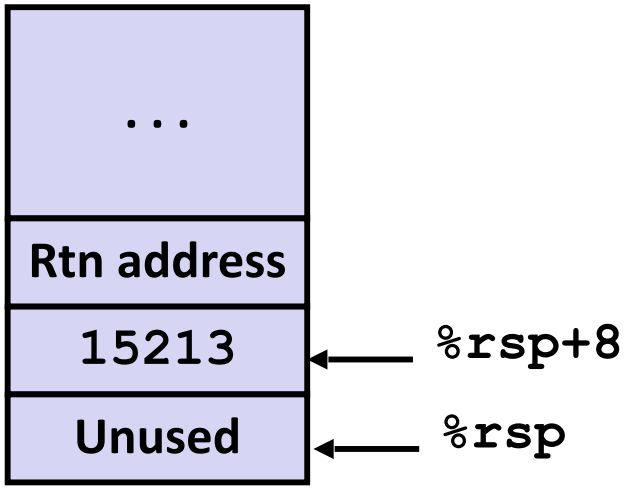


x86-64 Example: Calling `incr` #2

```
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

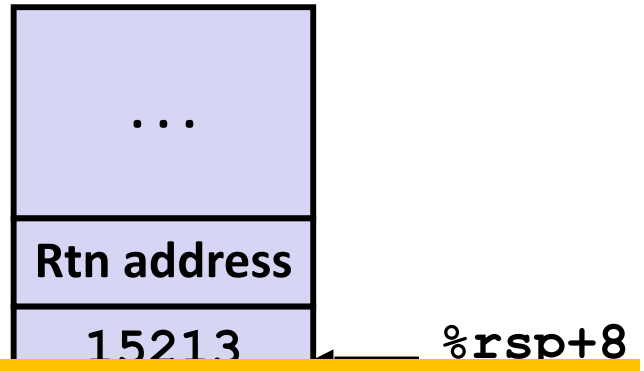


Register	Use(s)
%rdi	&v1
%rsi	3000

x86-64 Example: Calling `incr` #2

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

Stack Structure



Aside 1: `movl $3000, %esi`

- Remember, movl -> %exx zeros out high order 32 bits.
- Why use movl instead of movq? 1 byte shorter.

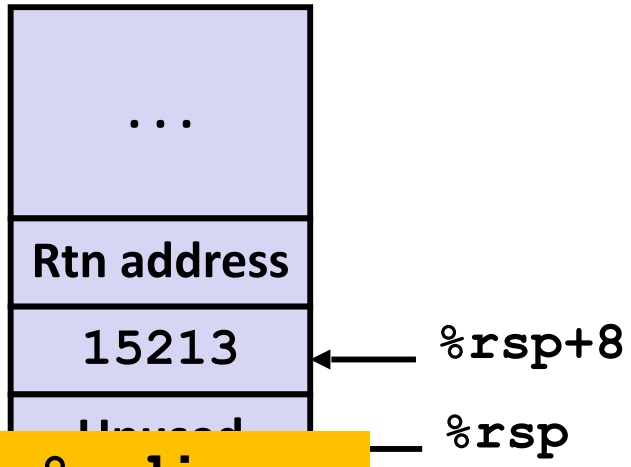
```
ca
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

%rdi	&v1
%rsi	3000

x86-64 Example: Calling `incr` #2

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

Stack Structure




```
ca
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Aside 2: `leaq 8(%rsp), %rdi`

- Computes %rsp+8
- Actually, used for what it is meant!

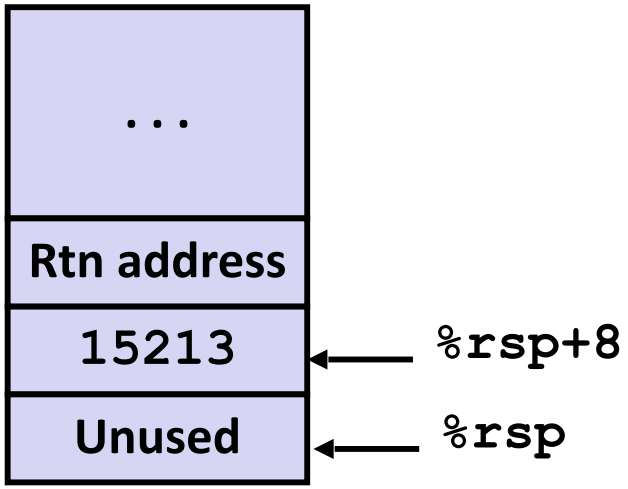
	se(s)
	v1
%rsi	3000

x86-64 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



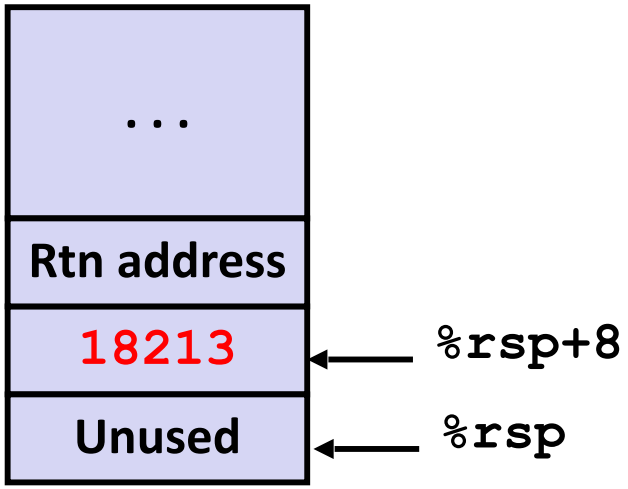
Register	Use(s)
%rdi	&v1
%rsi	3000

x86-64 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

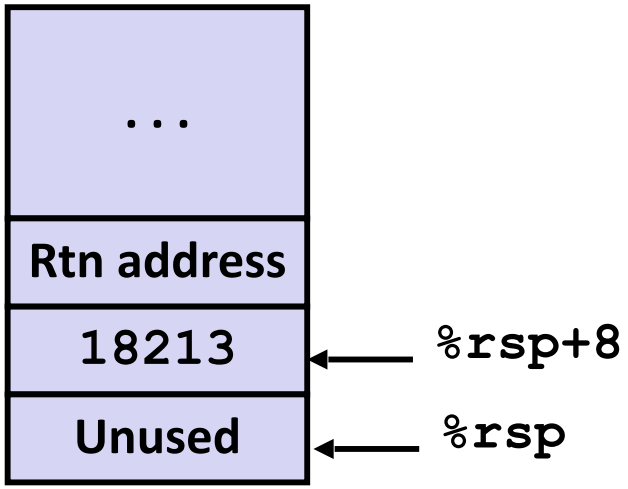


Register	Use(s)
%rdi	&v1
%rsi	3000

x86-64 Example: Calling `incr` #4

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

Stack Structure




```
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
```

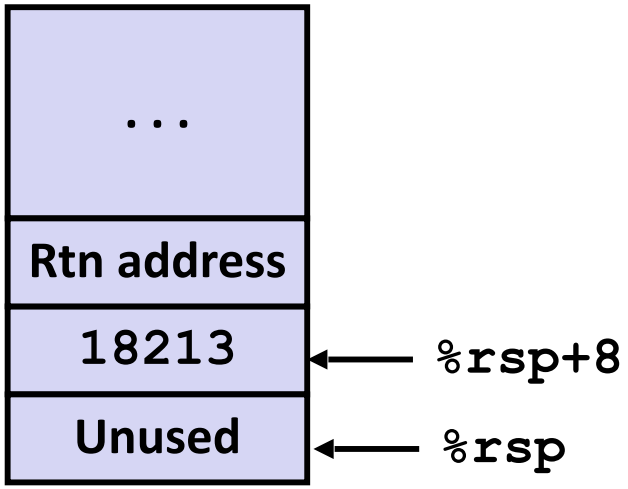
Register	Use(s)
%rax	Return value

x86-64 Example: Calling `incr` #5a

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

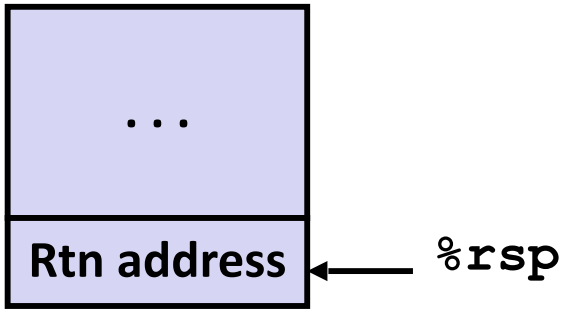
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure

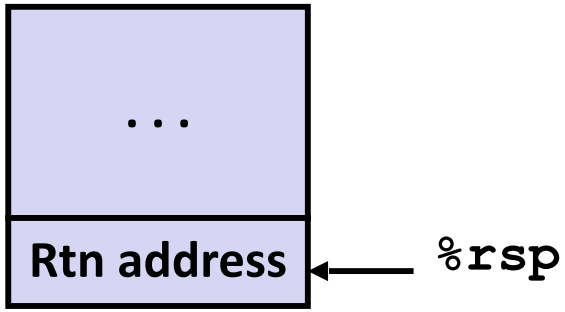


x86-64 Example: Calling `incr` #5b

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

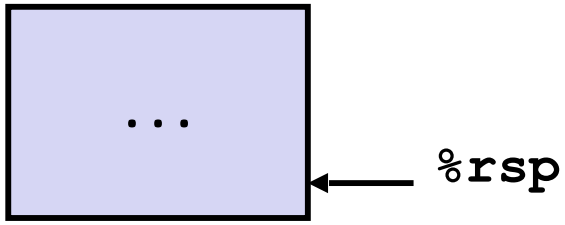
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

x86-64 Another Example (call_proc)

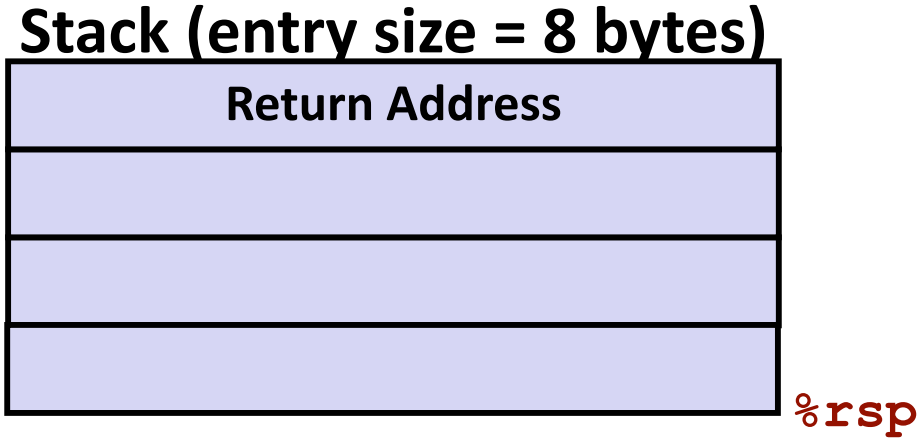
```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

Stack (entry size = 8 bytes)

x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

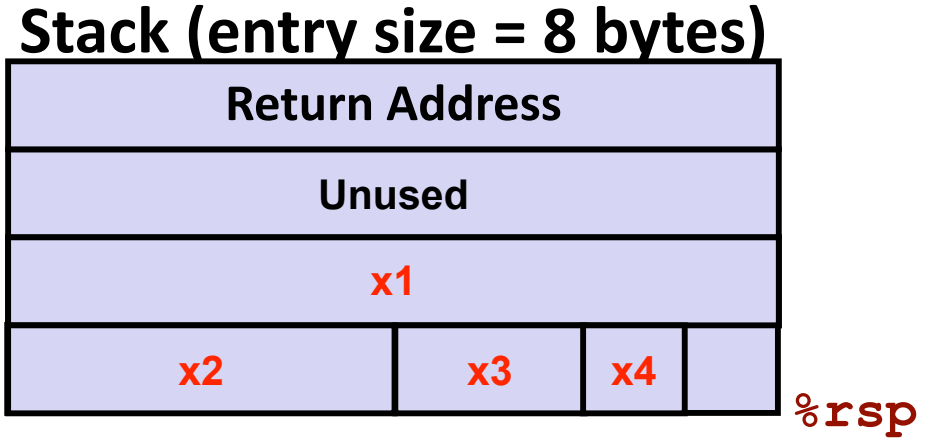



```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

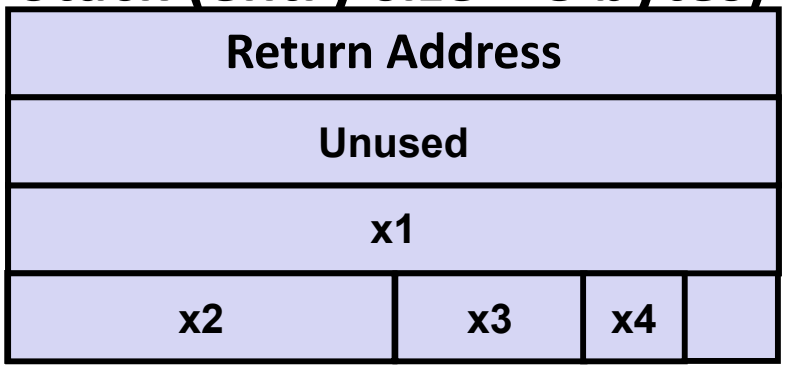
```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```



x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

Stack (entry size = 8 bytes)




```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

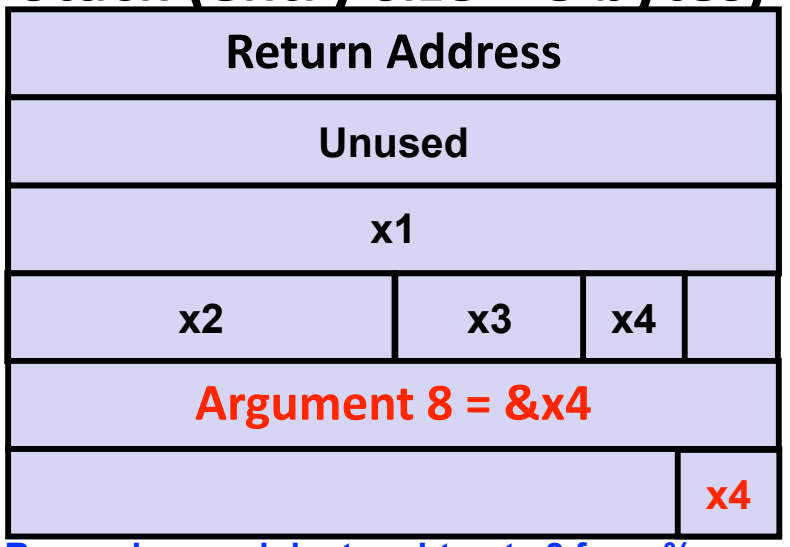
Register	Use(s)
%rdi	
%rsi	&x1
%edx	
%rcx	&x2
%r8w	
%r9	

x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

Stack (entry size = 8 bytes)



Remember push instr subtracts 8 from %rsp

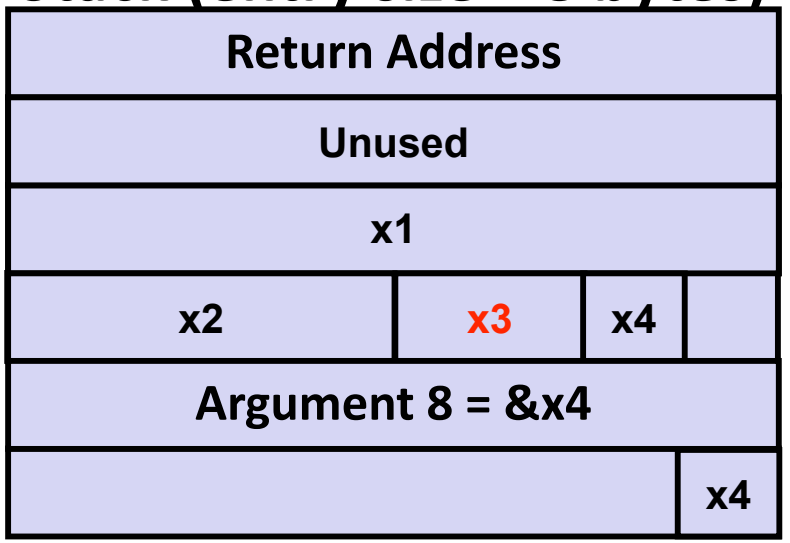
Register	Use(s)
%rdi	
%rsi	&x1
%edx	
%rcx	&x2
%r8w	
%r9	

x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

Stack (entry size = 8 bytes)



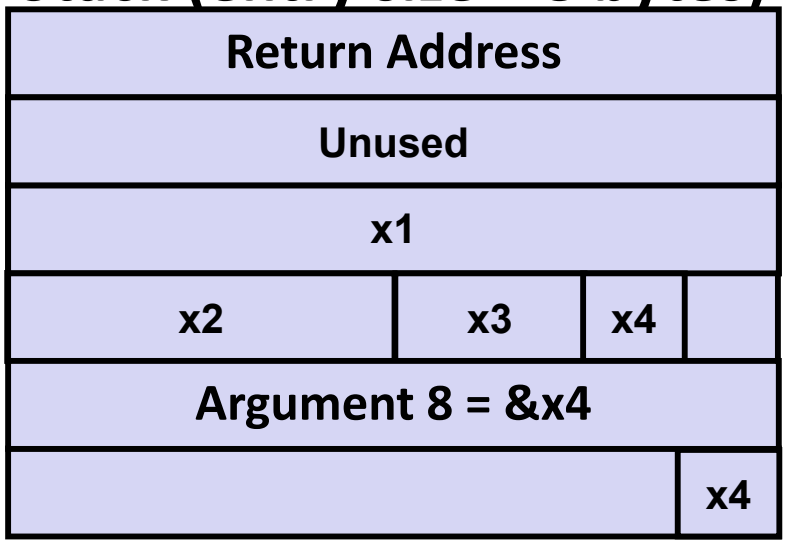
Register	Use(s)
%rdi	
%rsi	&x1
%edx	
%rcx	&x2
%r8w	
%r9	&x3

x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

Stack (entry size = 8 bytes)



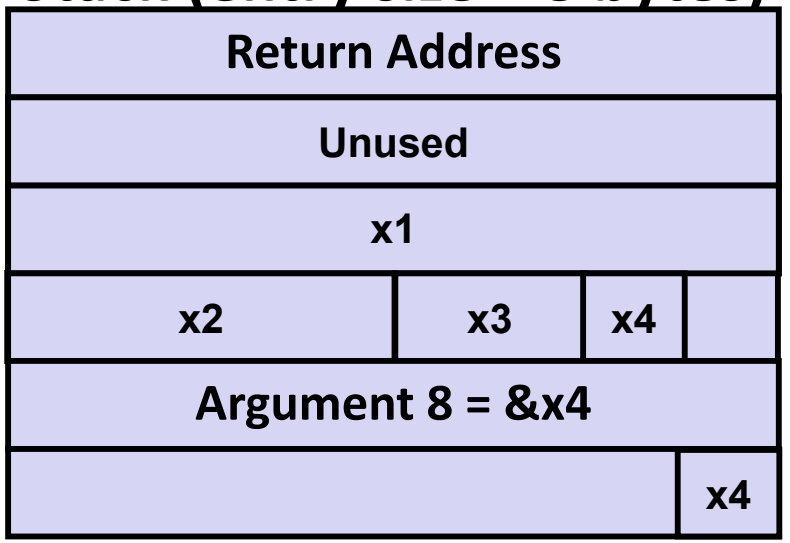
Register	Use(s)
%rdi	x1 = 1
%rsi	&x1
%edx	x2 = 2
%rcx	&x2
%r8w	x3 = 3
%r9	&x3

x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

Stack (entry size = 8 bytes)



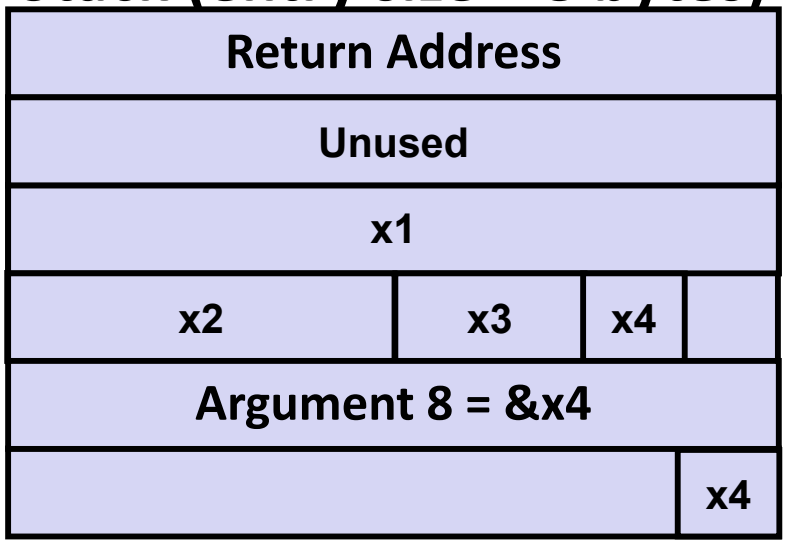
Register	Use(s)
%rdi	x1
%rsi	&x1
%edx	x2
%rcx	&x2
%r8w	x3
%r9	&x3

x86-64 Another Example (call_proc)

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

Stack (entry size = 8 bytes)



Register	Use(s)
%rdi	x1
%rsi	&x1
%edx	x2
%rcx	&x2
%r8w	x3
%r9	&x3

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
```

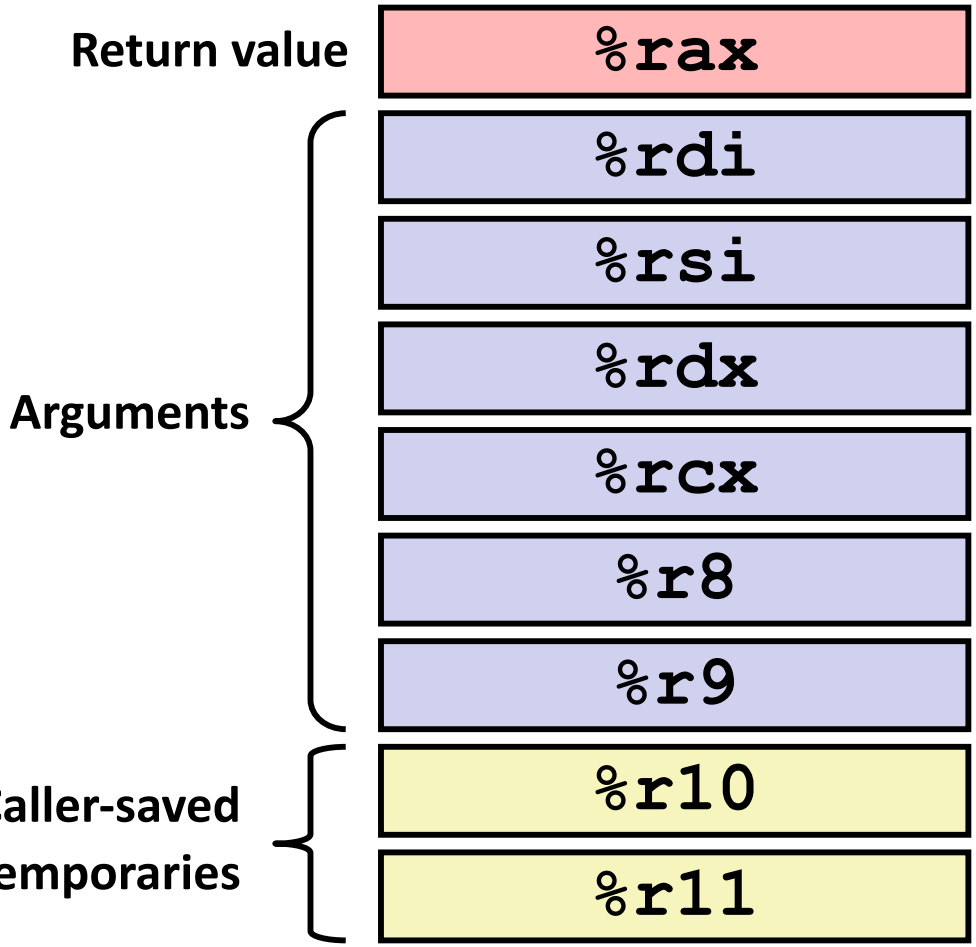
- Contents of register `%rdx` overwritten by `who`
- This could be trouble ➙ something should be done!
  - Need some coordination

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" (aka "Call-Clobbered")*
    - Caller saves temporary values in its frame before the call
  - *"Callee Saved" (aka "Call-Preserved")*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller
  - **All procedures (including library functions) must follow these conventions**

x86-64 Linux Register Usage #1

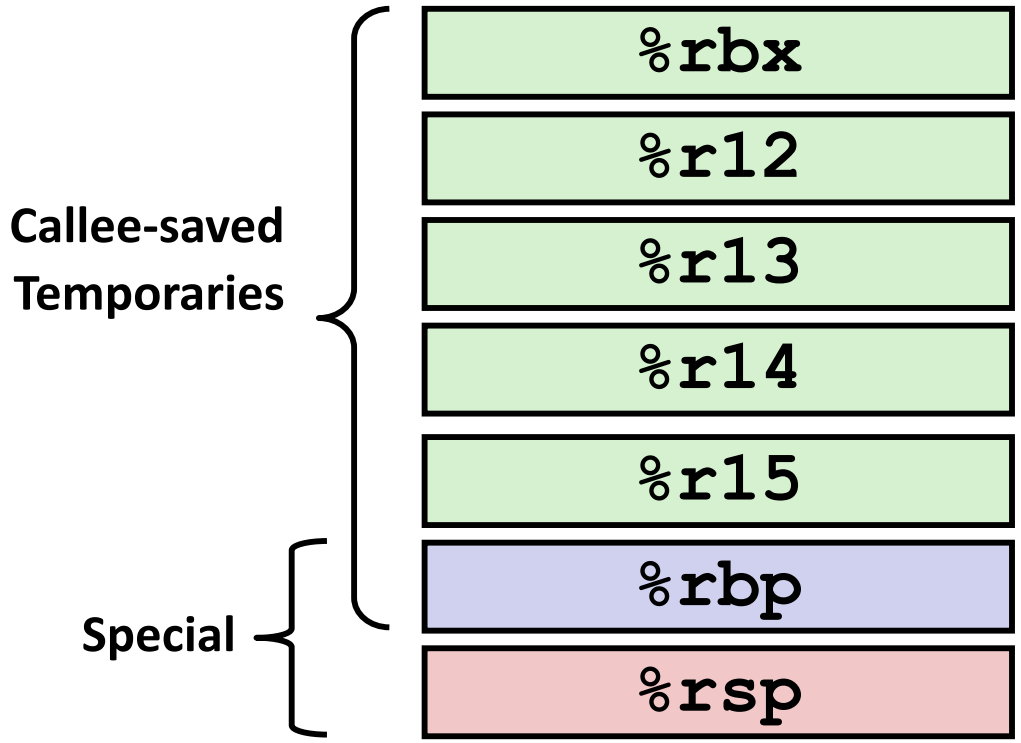
- `%rax`
  - Return value
  - Also **caller-saved**
  - Can be modified by procedure
- `%rdi`, …, `%r9`
  - Arguments
  - Also **caller-saved**
  - Can be modified by procedure
- `%r10`, `%r11`
  - **Caller-saved**
  - Can be modified by procedure



From **callee's** perspective, all these registers can be modified without any issues

x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14`, `%r15`
  - **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



From **caller's** perspective, all these registers are guaranteed to be **unchanged** after the call

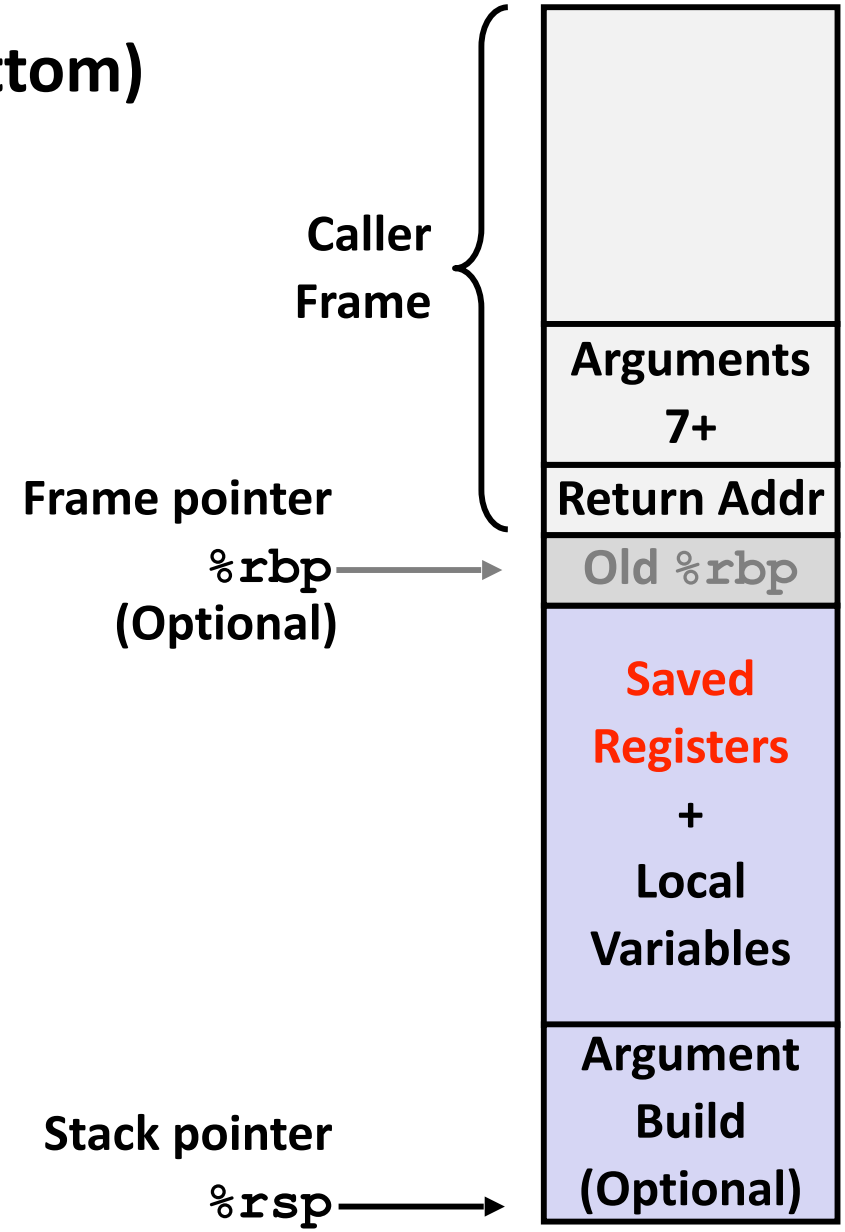
x86-64/Linux Stack Frame (Revisit)

Current Stack Frame ("Top" to Bottom)

- "Argument build:"
  Parameters for function about to call
- Local variables
  If can't keep in registers
- **Saved register context**
  - **Push old register value**
  - **Make change**
  - **Pop old register value**
- Old frame pointer (**optional**)

Caller Stack Frame

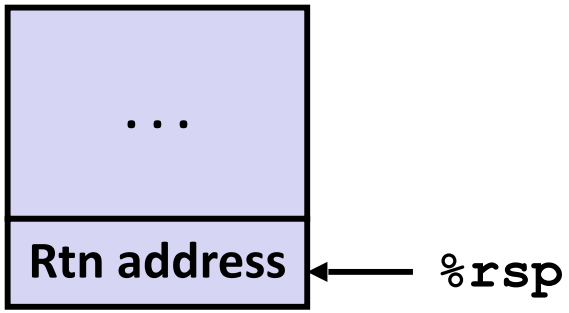
- Return address
  - Pushed by `call` instruction
- Arguments for this call



Callee-Saved Example #1

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

Initial Stack Structure



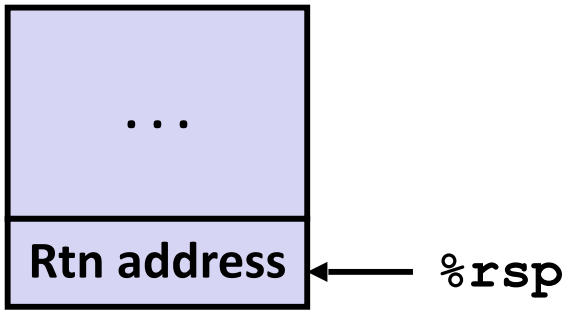
- X comes in register **`%rdi`**.
- We need **`%rdi`** for the call to incr.
- Where should we put x, so we can use it after the call to incr?
  In a **callee-saved register (e.g., %rbx)** since it is guaranteed not to be changed by the callee

Callee-Saved Example #2

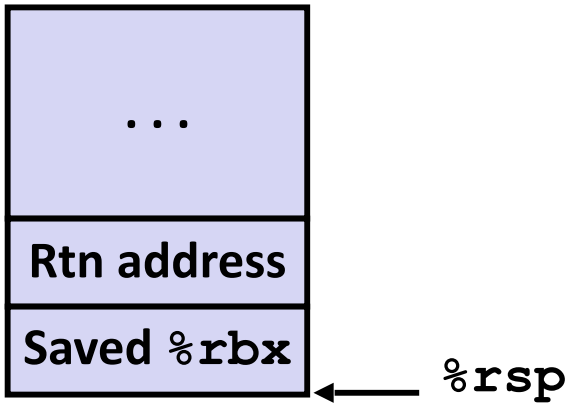
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure

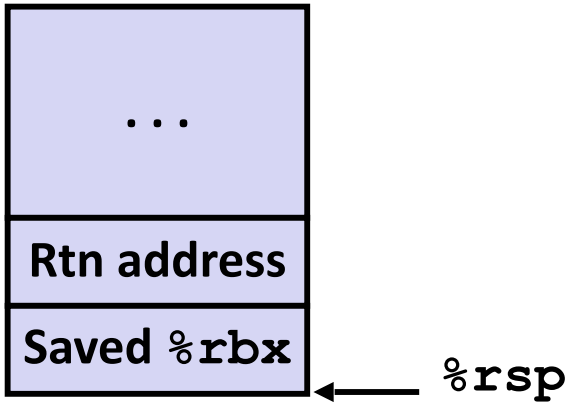


Callee-Saved Example #3

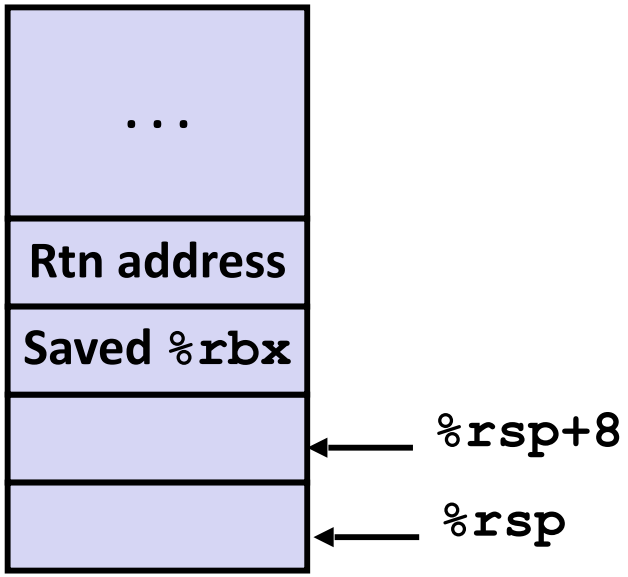
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

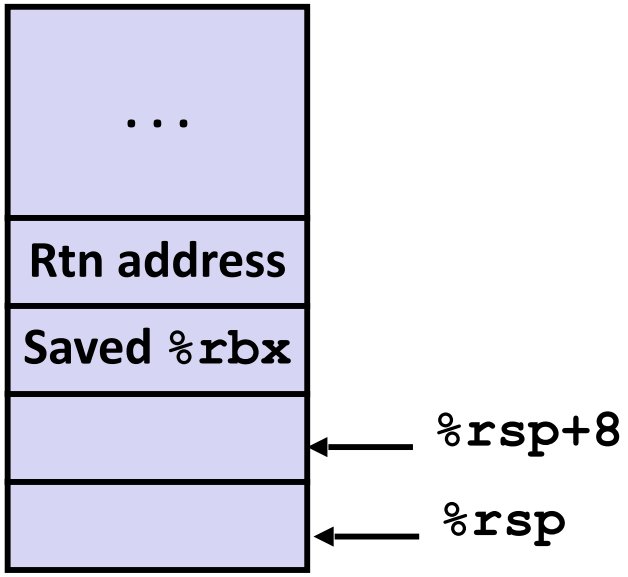


Callee-Saved Example #4

```
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
```

Stack Structure



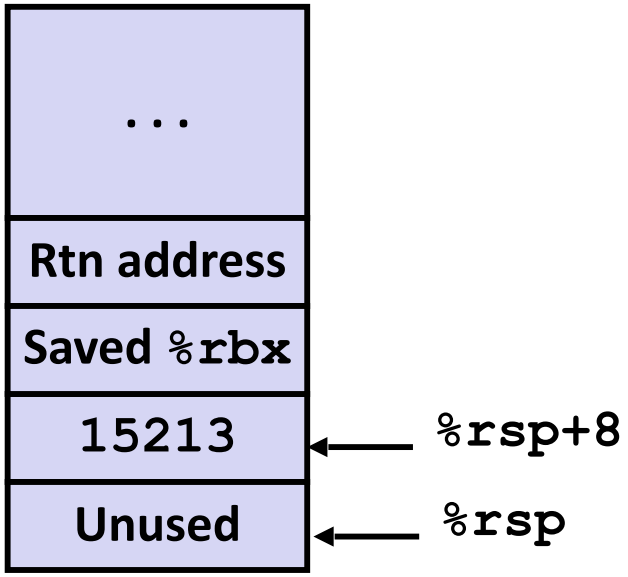
- X saved in **%rbx**.
- A callee saved register.

Callee-Saved Example #5

```
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
```

Stack Structure



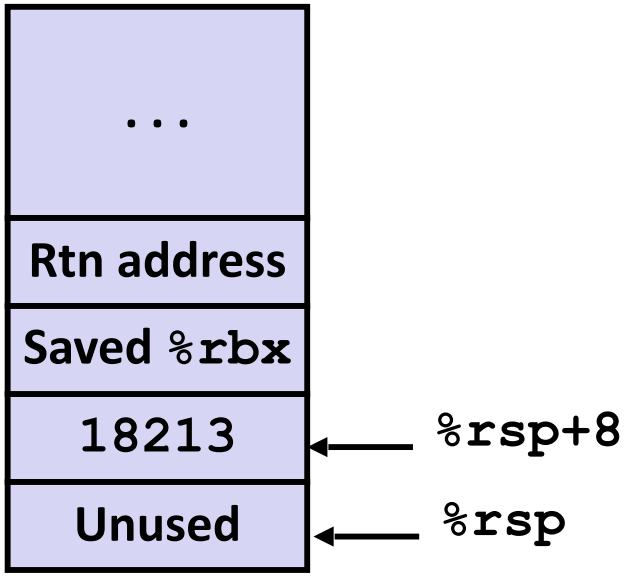
- X saved in **%rbx**.
- A callee saved register.

Callee-Saved Example #6

```
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
```

Stack Structure



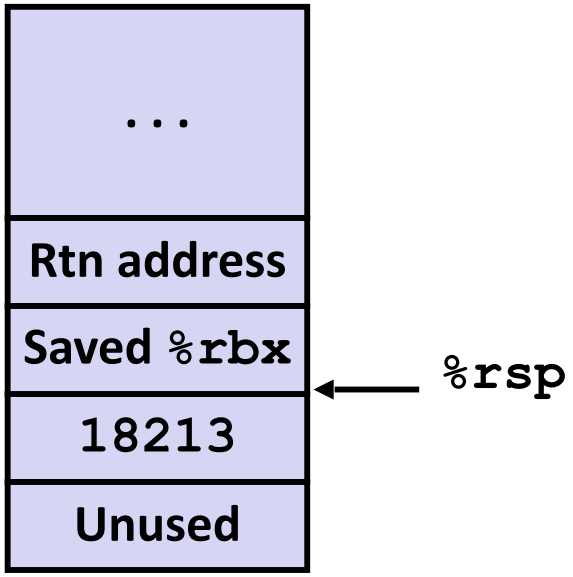
- x is safe in **%rbx**
- Return result in **%rax**

Callee-Saved Example #7

```
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
```

Stack Structure



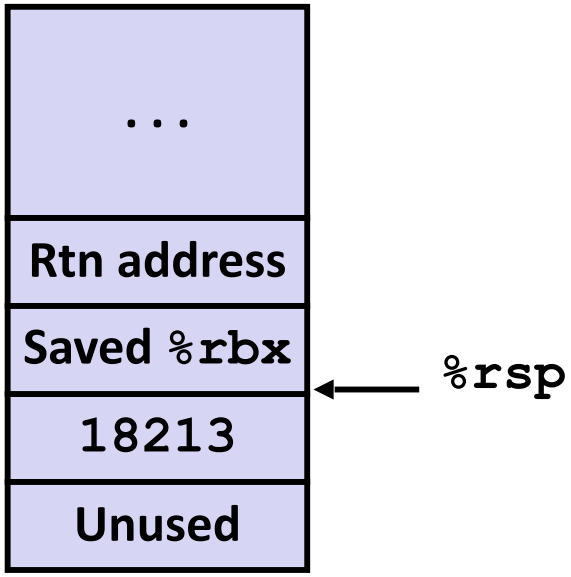
- Return result in **%rax**

Callee-Saved Example #8

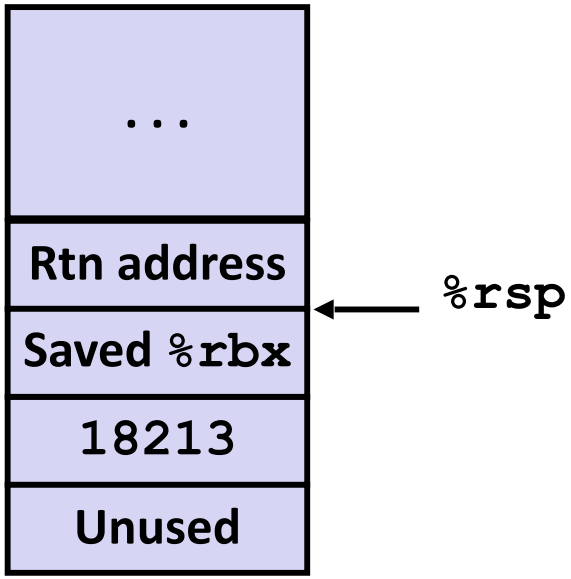
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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



final Stack Structure

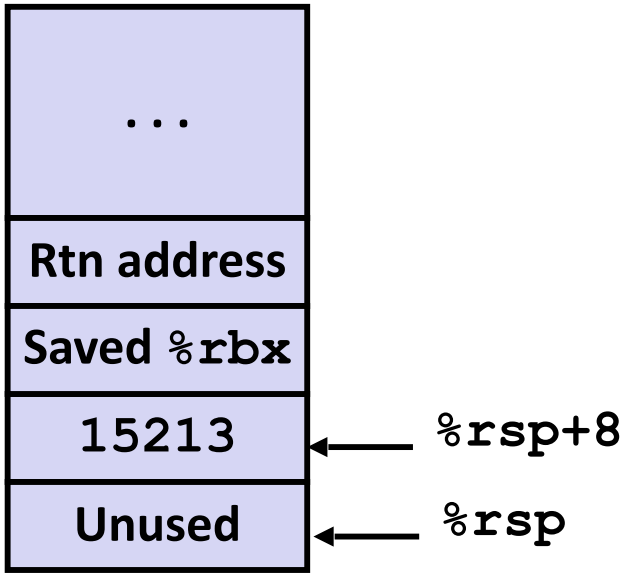


Callee-Saved Example #2

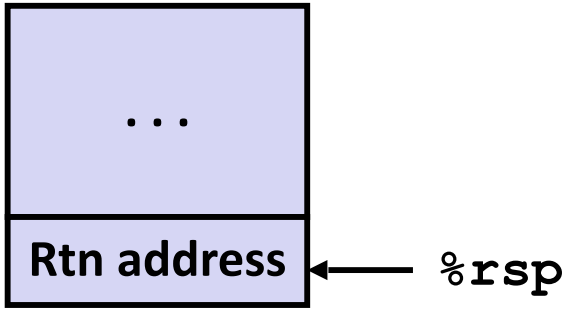
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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



Machine-Level Programming III: Procedures

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- **Illustration of 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:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; 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:
   movl    $0, %eax
   testq   %rdi, %rdi
   je      .L6
   pushq   %rbx
   movq    %rdi, %rbx
   andl    $1, %ebx
   shrq    %rdi
   call    pcount_r
   addq    %rbx, %rax
   popq    %rbx
.L6:
   rep; ret
```

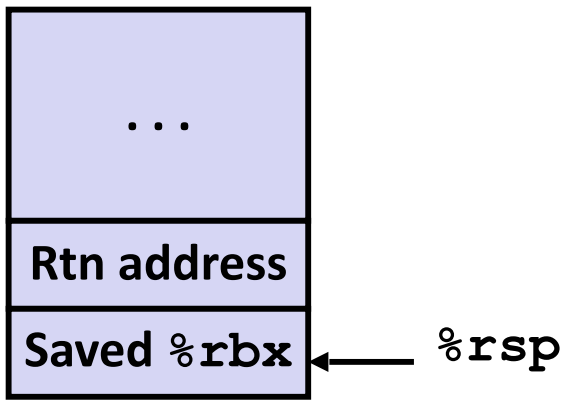
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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

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:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   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	

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

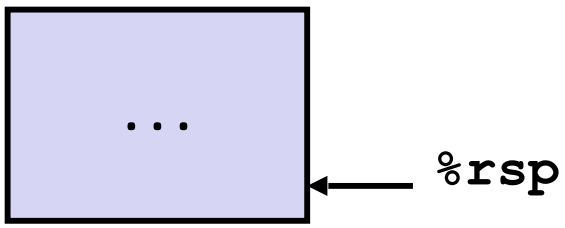
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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

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



Observations About Recursion

Handled Without Special Consideration

- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C code explicitly does so (e.g., buffer overflow)
- 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

x86-64 Procedure Summary

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`

Pointers are addresses of values

- **On stack or global**