
WELCOME
TO
15-213
18-213
15-513

Activities are posted on the website (we might not have time today but you can review them after class)

```
wget http://www.cs.cmu.edu/~213/activities/machine-procedures.pdf
wget http://www.cs.cmu.edu/~213/activities/machine-procedures.tar
tar xf machine-procedures.tar
cd machine-procedures
```

Some things we need to review from last time…

- An x86 program’s view

An x86-64 program's view...

CPU

Program Counter (PC)

%rpi	0x00ff

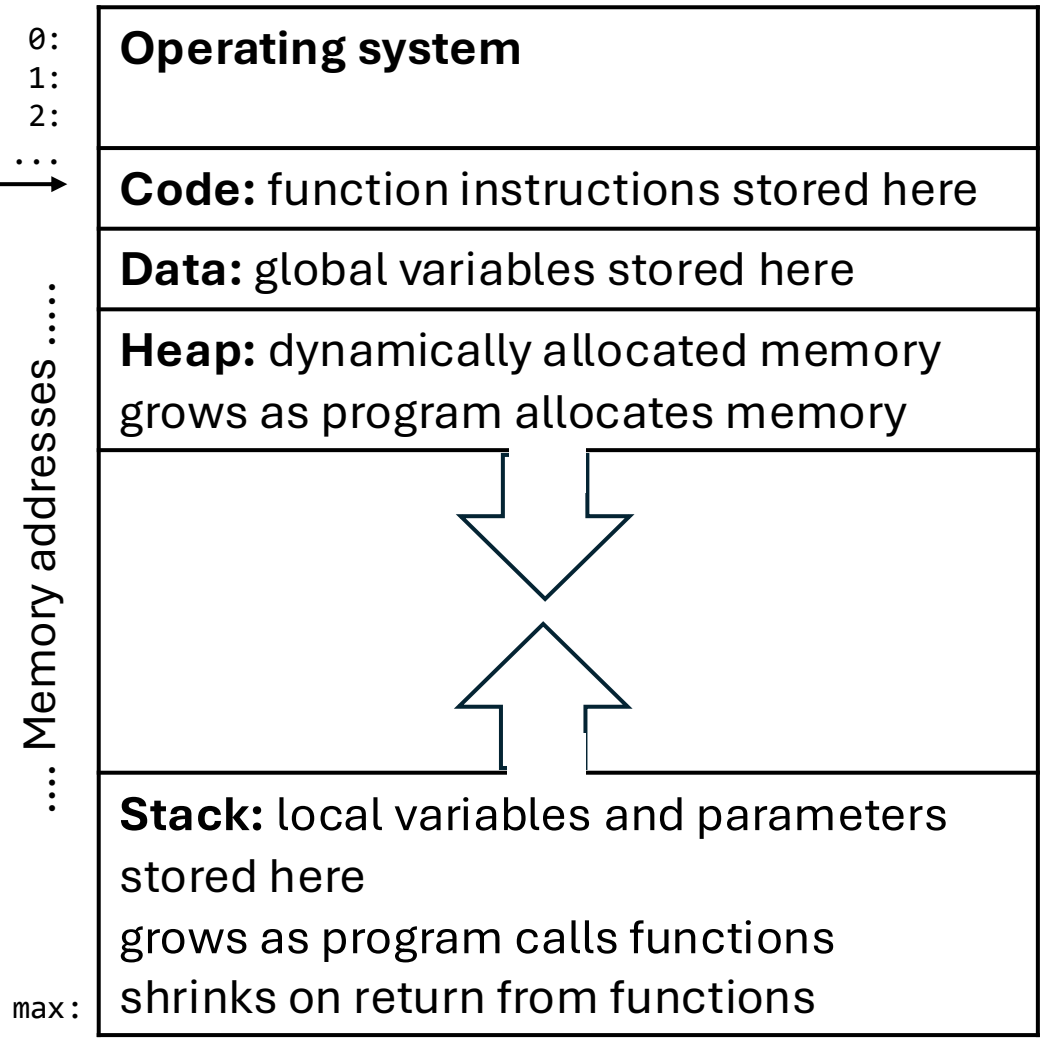
Condition Codes

ZF	0
SF	1
OF	1
CF	0

16 "General purpose" Registers

%rsp	0xff88
%rsi	1
%rdi	10
%rax	11
...	

Memory (Virtual)



An x86-64 program's view...

CPU

Program Counter (PC)

%rpi	0x00ff

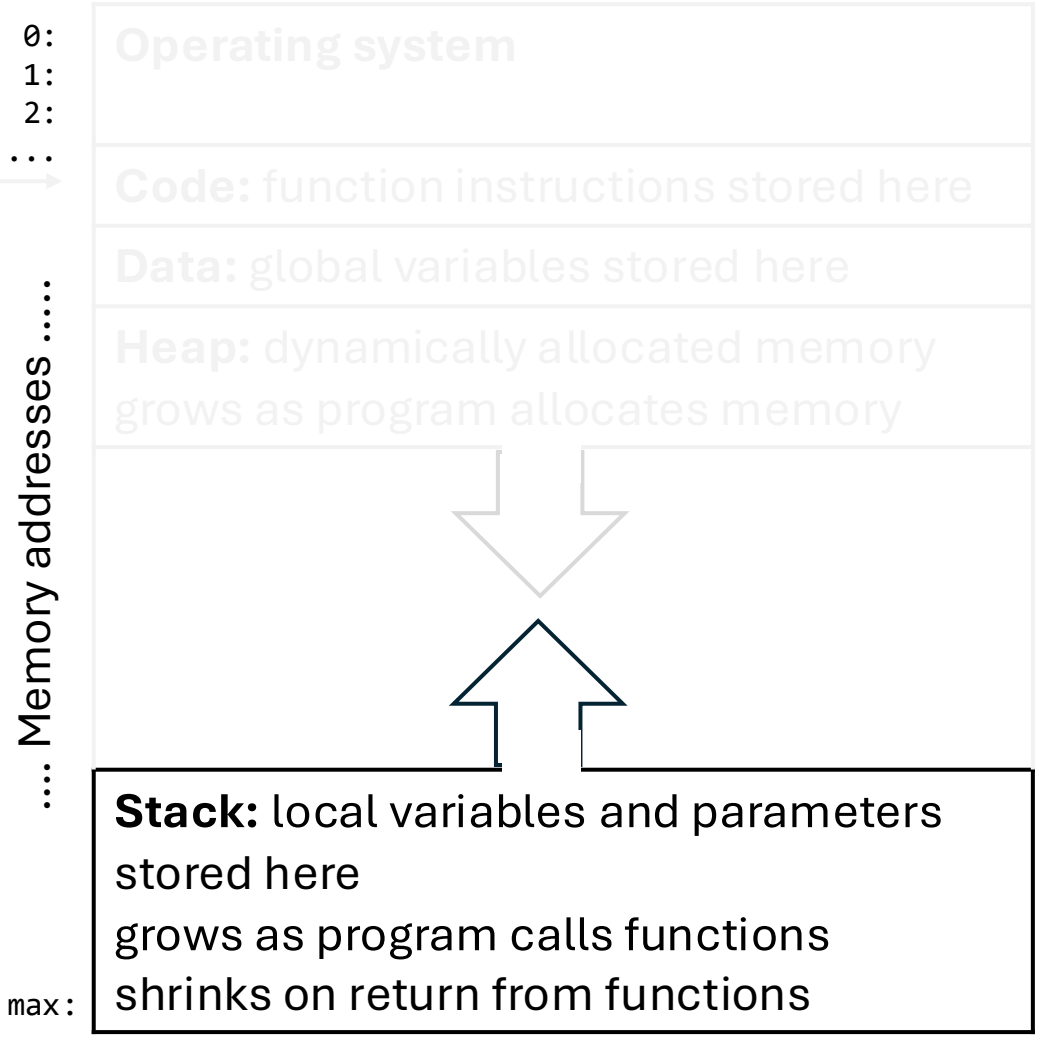
Condition Codes

ZF	0
SF	1
OF	1
CF	0

16 "General purpose" Registers

%rsp	0xff88
%rsi	1
%rdi	10
%rax	11
...	

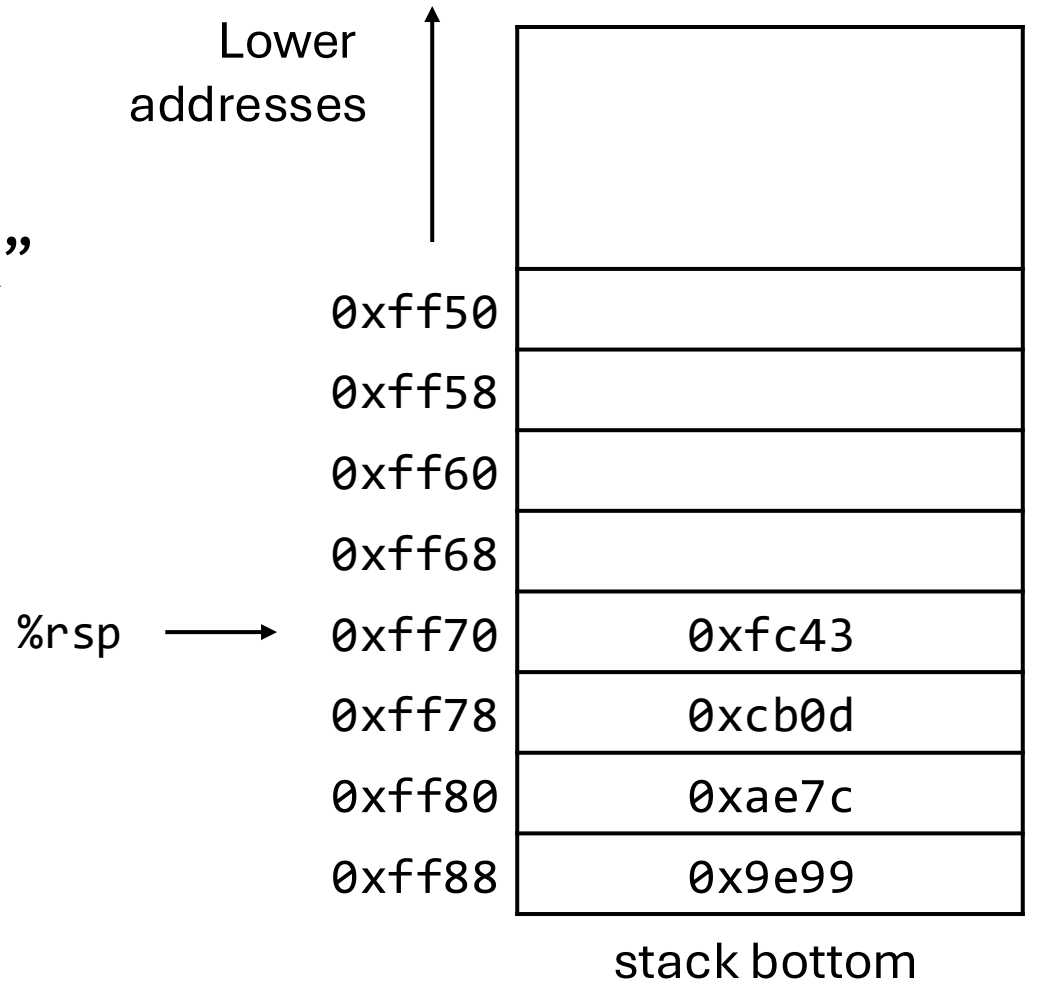
Memory (Virtual)



Traditional view of a stack

`%rsp` is the stack pointer

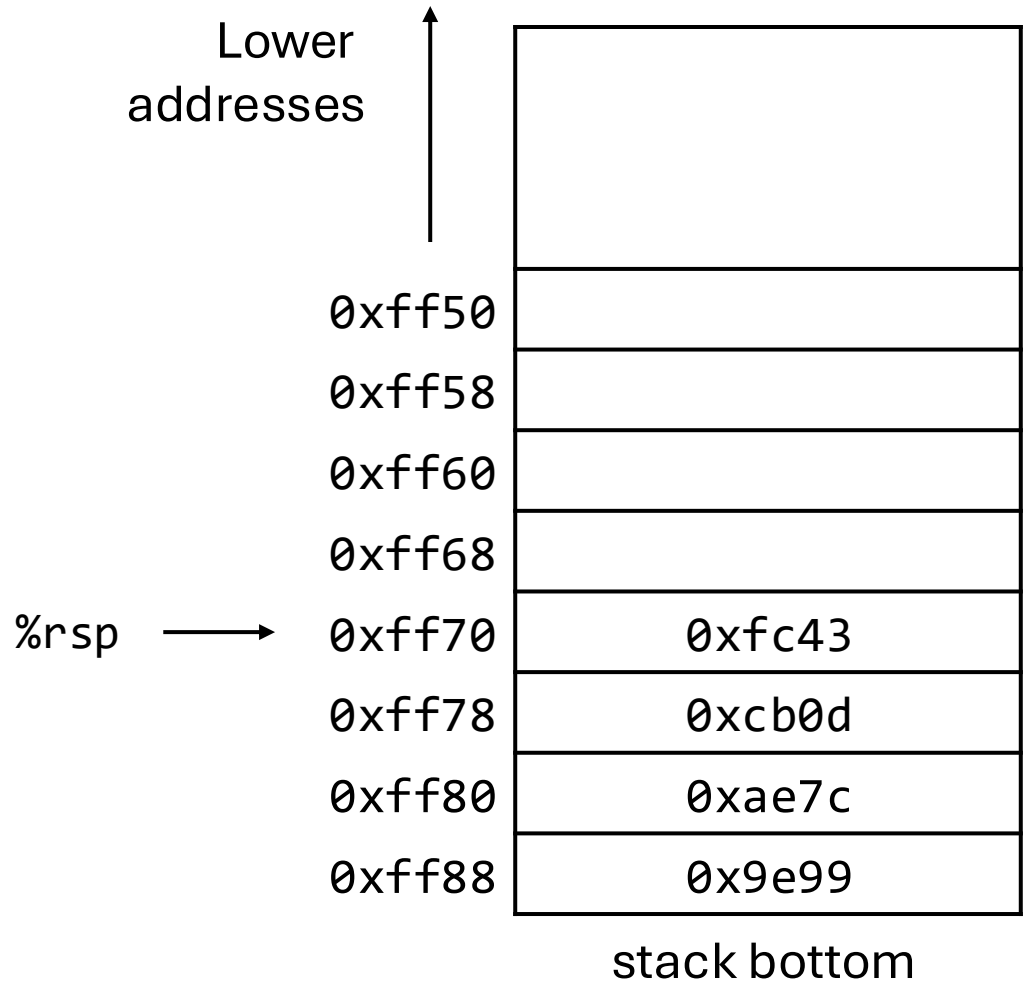
It points to the current “top of the stack”



Traditional view of a stack

A traditional view of the stack shows growing the stack from the "bottom up".

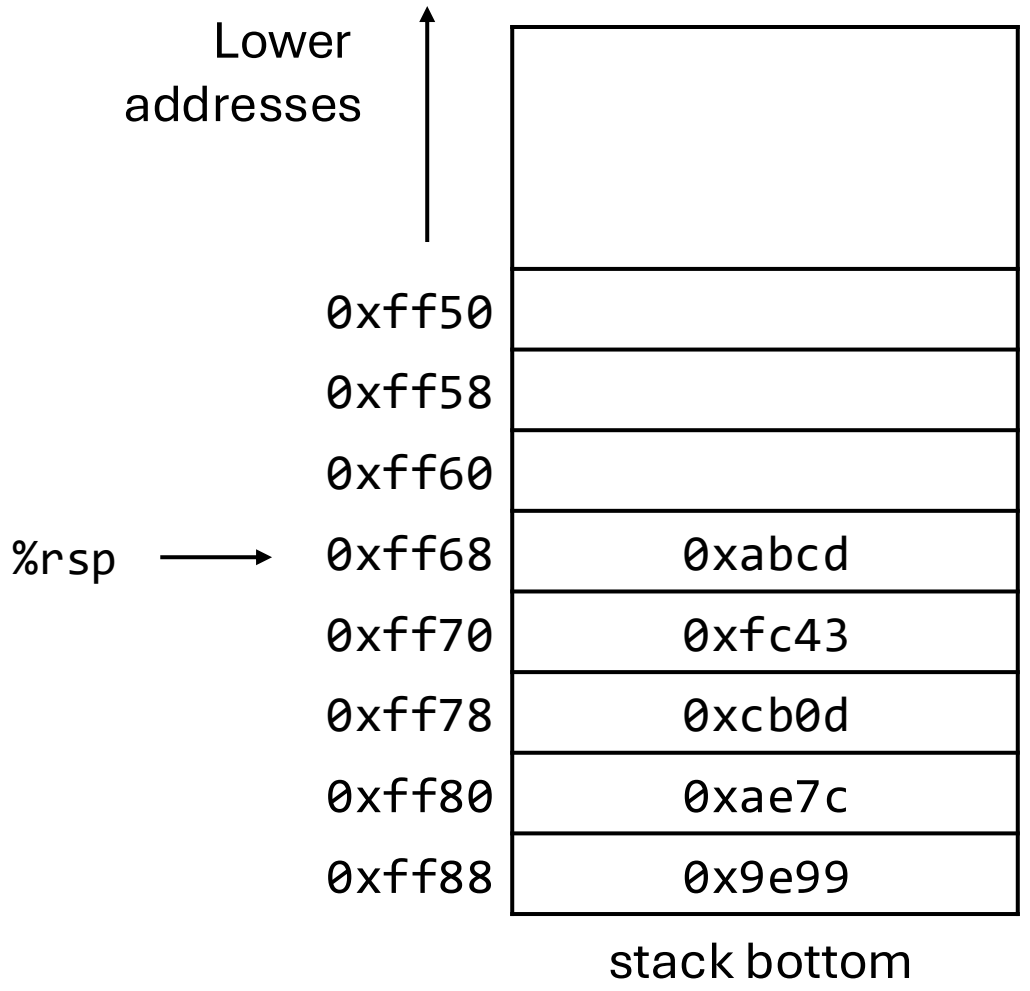
Pushing onto the stack puts an element on the top of the stack



Traditional view of a stack

A traditional view of the stack shows growing the stack from the "bottom up".

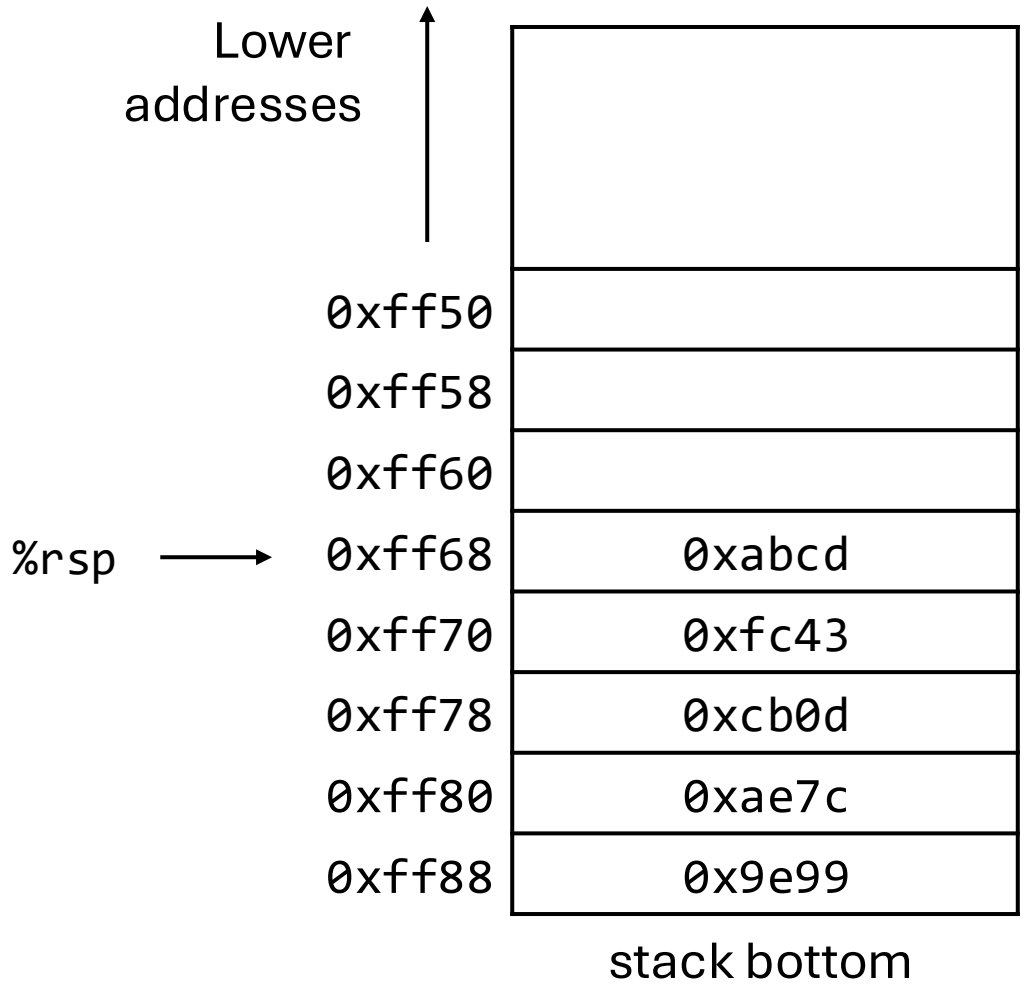
Pushing onto the stack puts an element on the top of the stack



Traditional view of a stack

A traditional view of the stack shows growing the stack from the "bottom up".

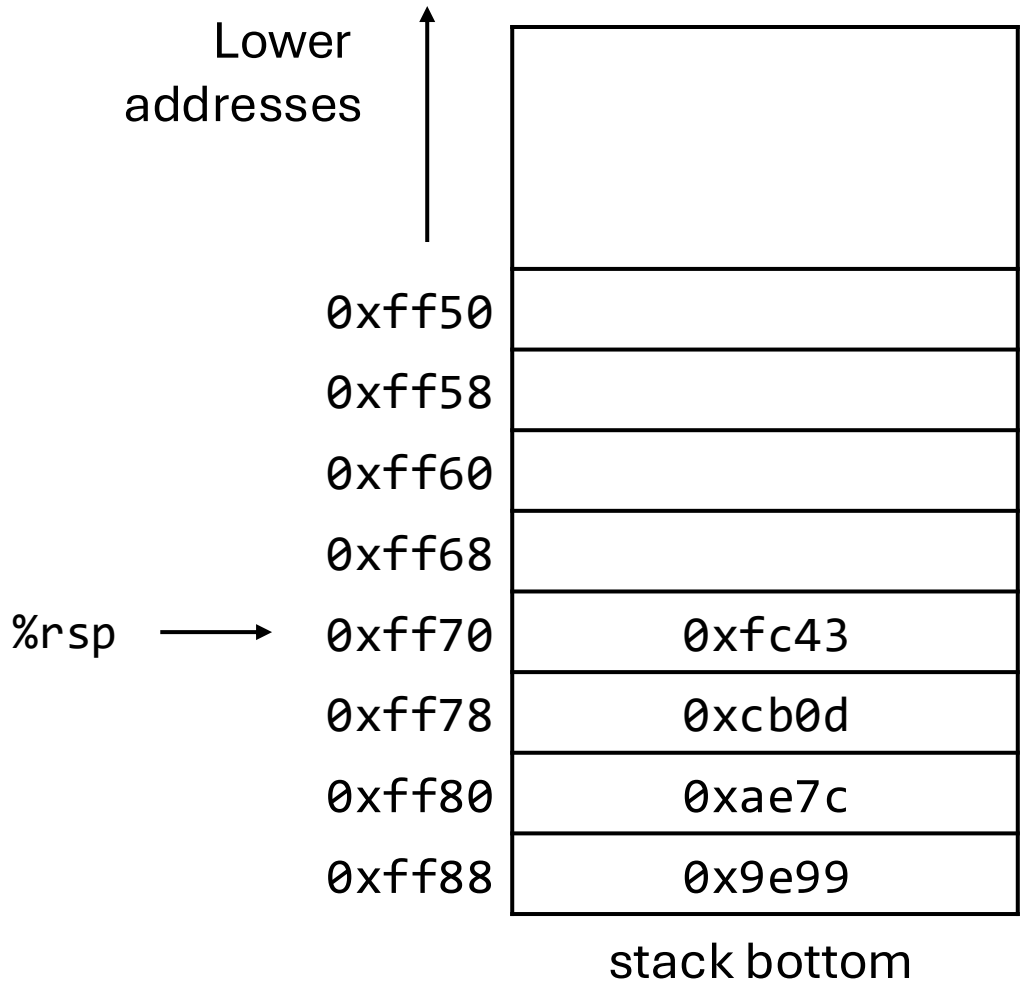
Popping elements off the stack removes the most recent thing added to the stack.



Traditional view of a stack

A traditional view of the stack shows growing the stack from the "bottom up".

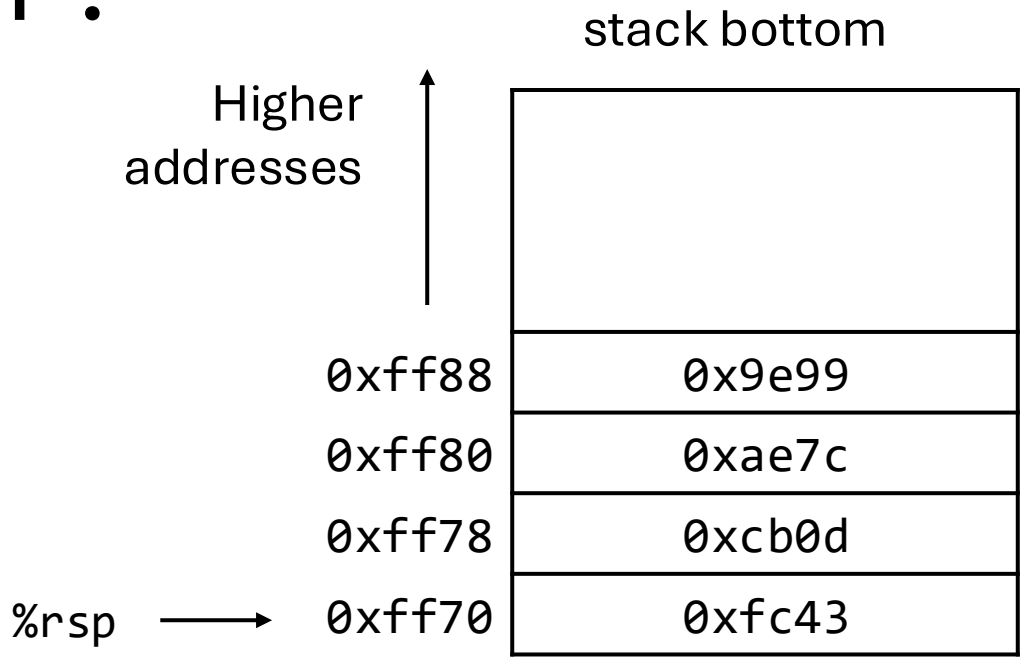
Popping elements off the stack removes the most recent thing added to the stack.



By convention in this class, we draw the x86-64 stack “upside down”!

The stack grows down towards lower addresses.

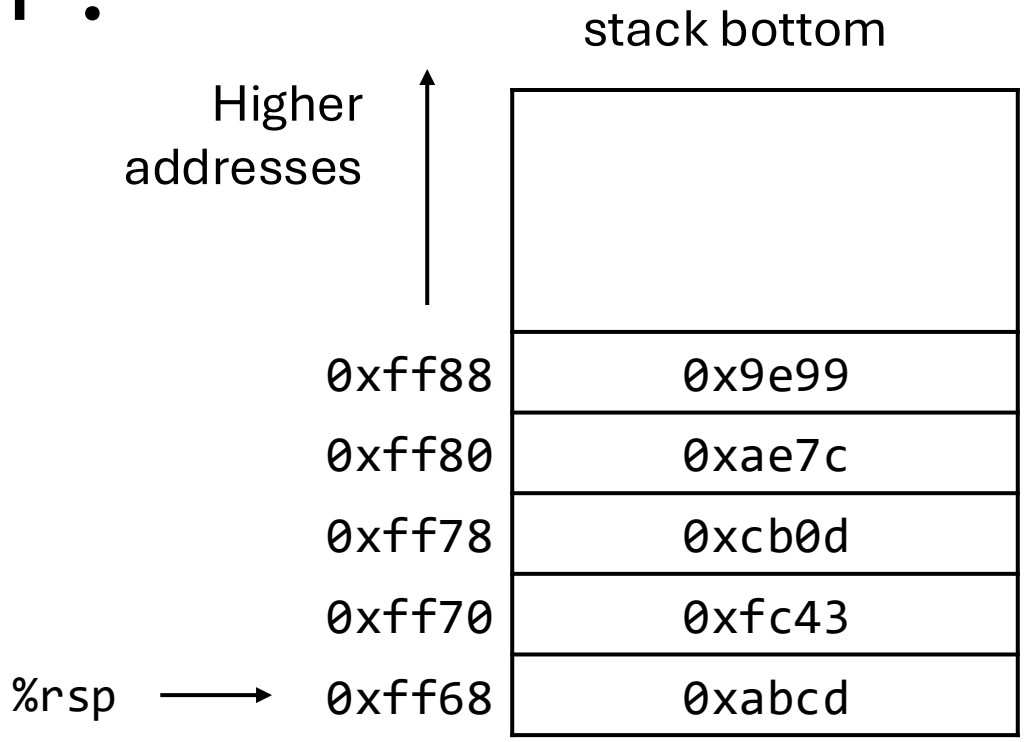
This is still “last-in-first-out” but we show the it growing down rather than up.



By convention in this class, we draw the x86-64 stack “upside down”!

The stack grows down towards lower addresses.

Pushing elements puts them on top of the stack (but we show this upside down).

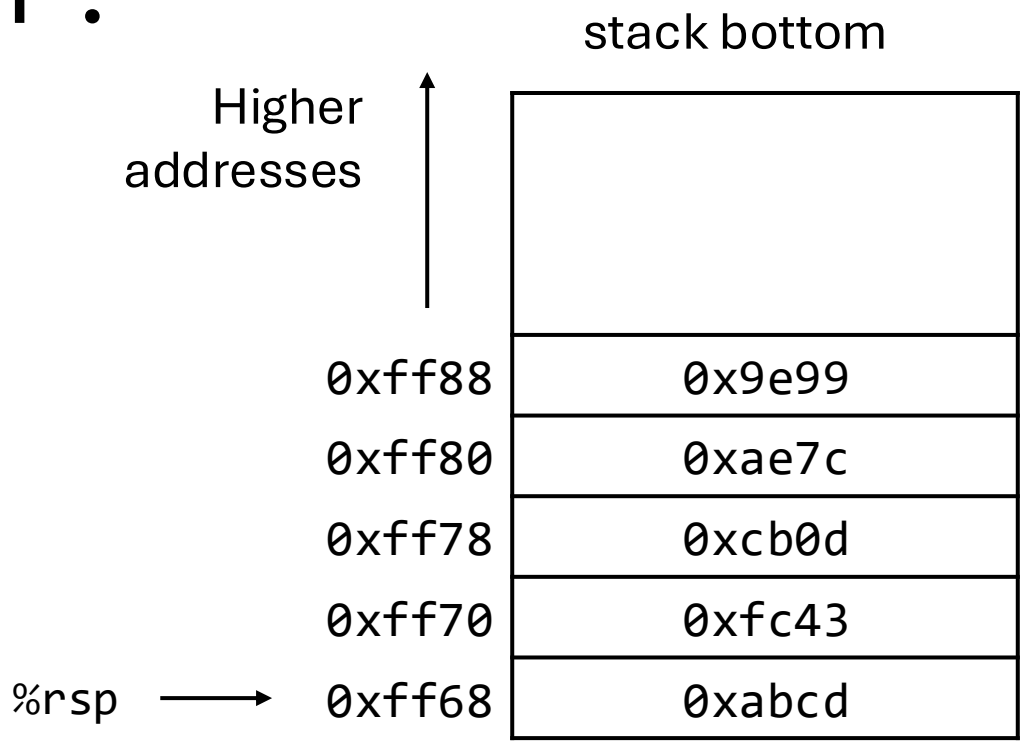


By convention in this class, we draw the x86-64 stack “upside down”!

The stack grows down towards lower addresses.

Pushing elements puts them on top of the stack (but we show this upside down).

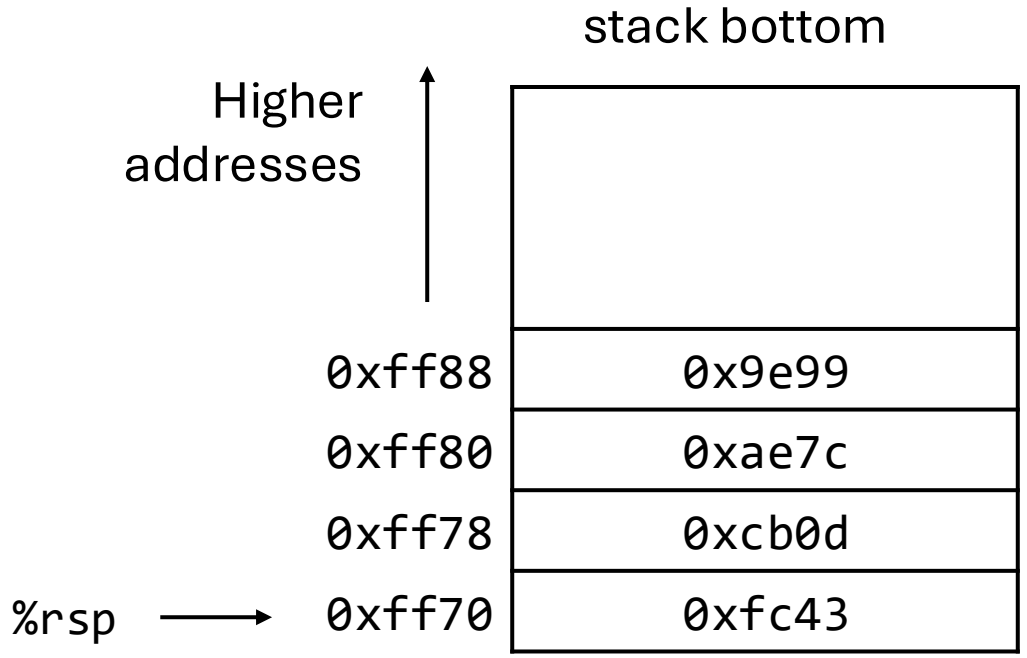
Popping is different than removing the element! Let’s look at the instructions and examples.



Pushing stack data with `pushq`

`pushq` *Src*

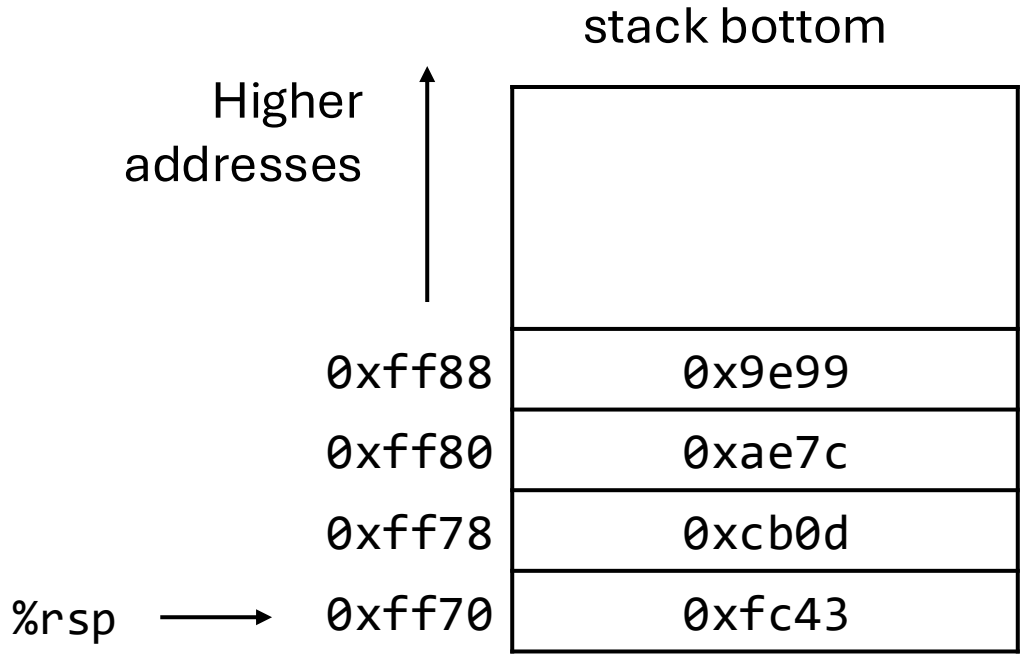
1. Decrement `%rsp` by 8 bytes
2. Write operand into the address given by `%rsp`



Pushing stack data with `pushq`

`pushq` *Src*

1. Decrement `%rsp` by 8 bytes
2. Write operand into the address given by `%rsp`



Example:

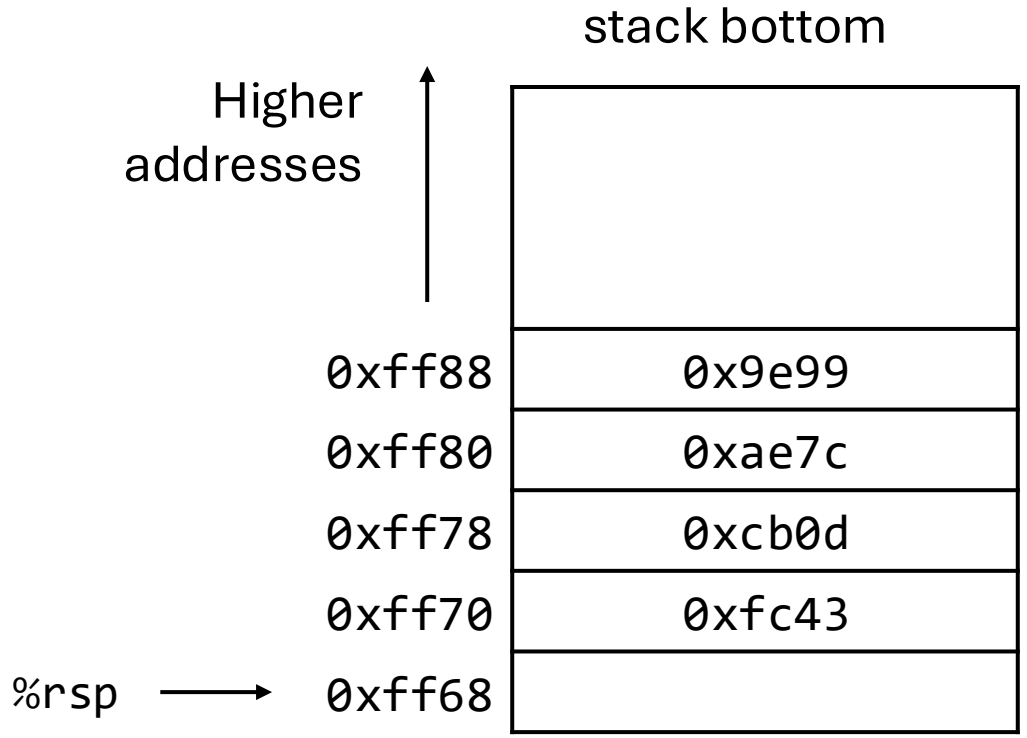
`pushq %rbp`

Registers	
`%rsp`	`0xff70`
`%rbp`	`0xabcd`

Pushing stack data with `pushq`

`pushq` *Src*

1. **Decrement `%rsp` by 8 bytes**
2. Write operand into the address given by `%rsp`



Example:

`pushq %rbp`

Registers	
`%rsp`	`0xff68`
`%rbp`	`0xabcd`

Pushing stack data with `pushq`

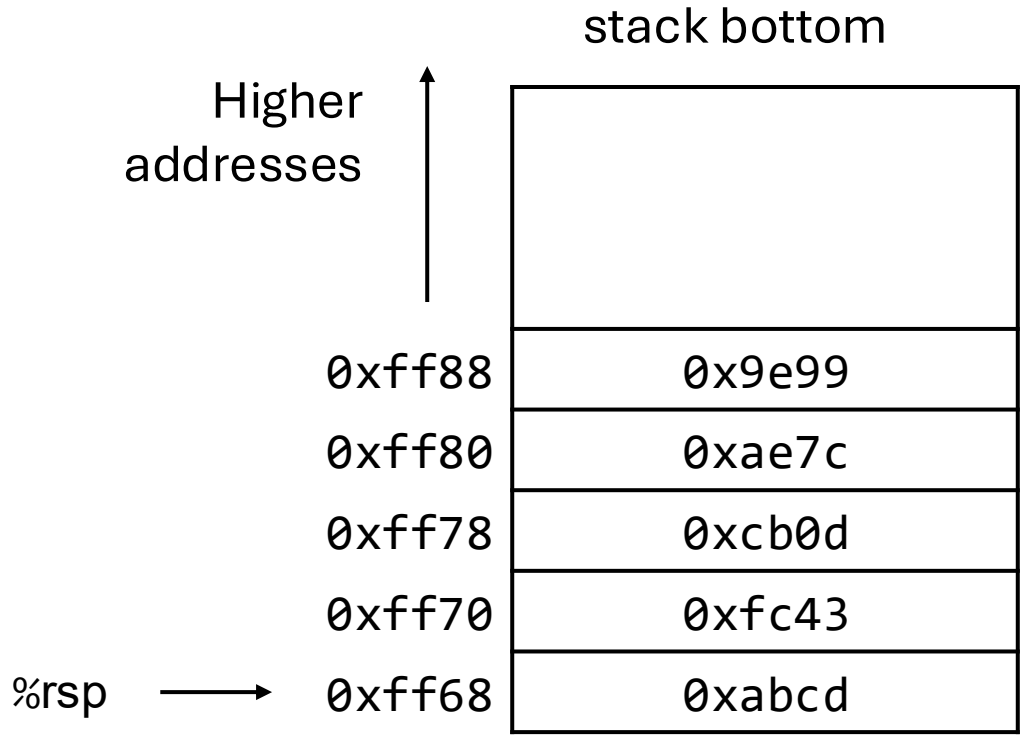
`pushq` *Src*

1. Decrement `%rsp` by 8 bytes
2. **Write operand into the address given by `%rsp`**

Example:

`pushq %rbp`

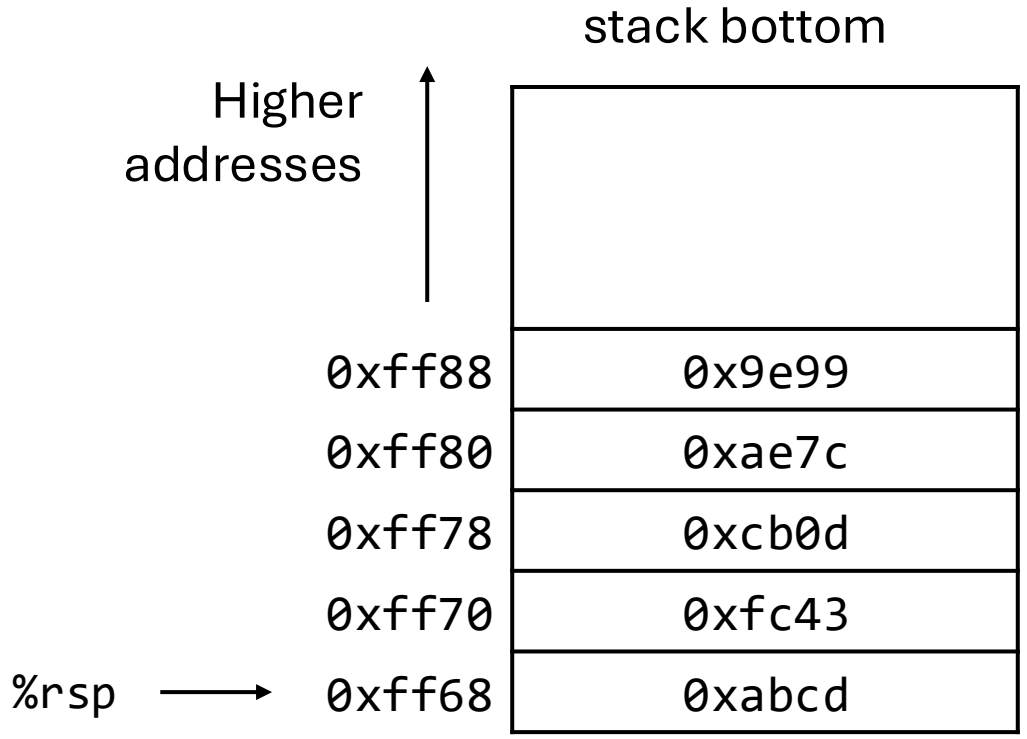
Registers	
`%rsp`	`0xff68`
`%rbp`	`0xabcd`



Popping stack data with popq

popq *Dest*

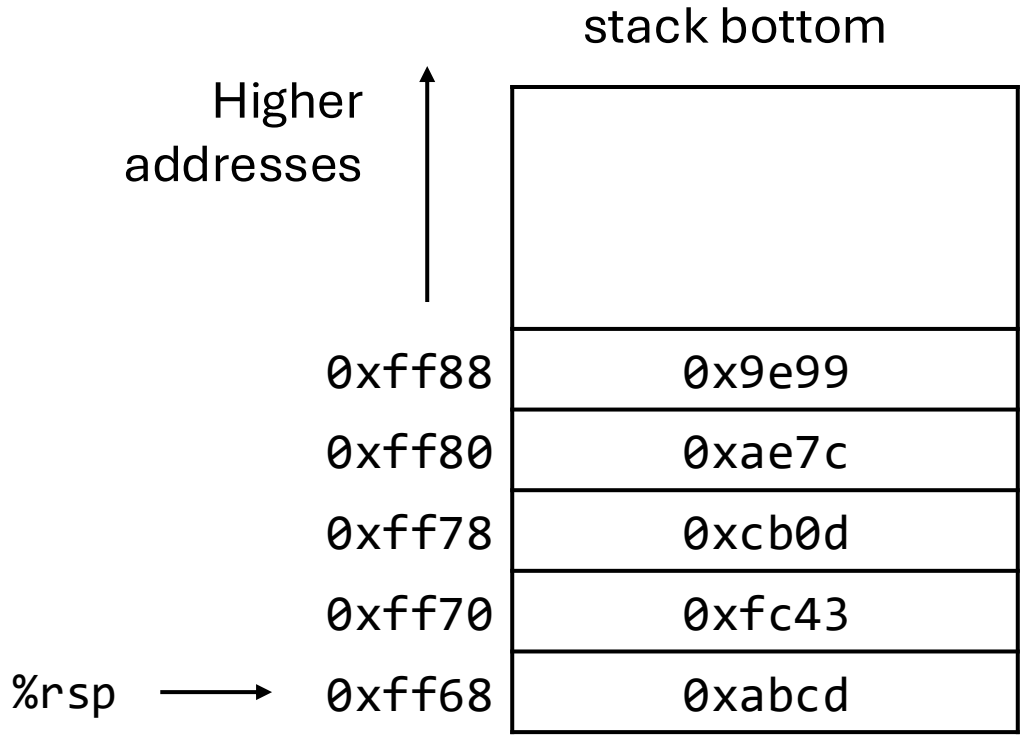
1. Read operand given by address given by `%rsp` and store in operand
2. Increment `%rsp` by 8 bytes



Popping stack data with popq

popq *Dest*

1. Read from address given by %rsp and store in operand
2. Increment %rsp by 8 bytes



Example:

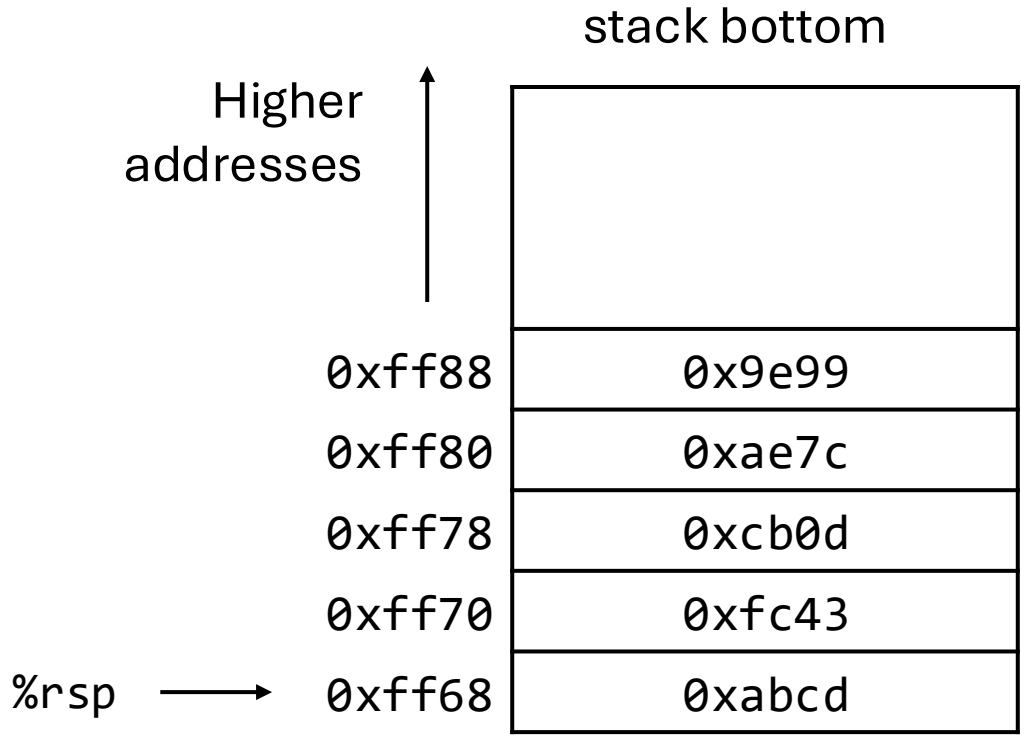
popq %rdx

Registers	
%rsp	0xff68
%rdx	0

Popping stack data with popq

popq *Dest*

1. **Read from address given by %rsp and store in operand**
2. Increment `%rsp` by 8 bytes



Example:

popq %rdx

Registers	
%rsp	0xff68
%rdx	0xabcd

Popping stack data with popq

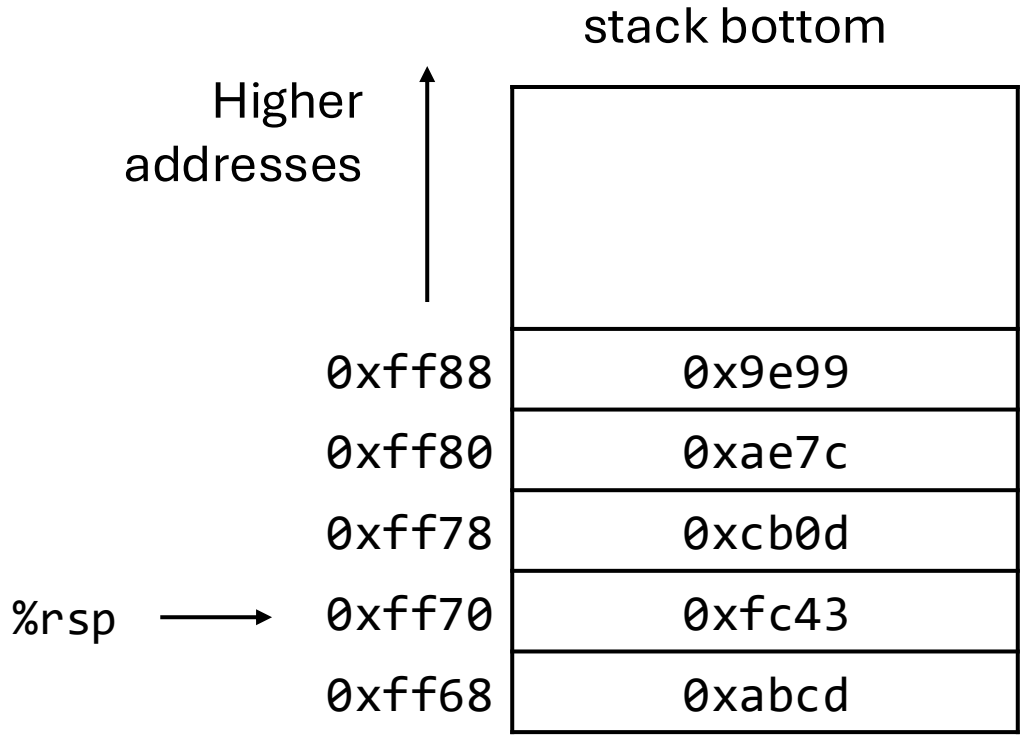
popq *Dest*

1. Read from address given by %rsp and store in operand
2. **Increment %rsp by 8 bytes**

Example:

popq %rdx

Registers	
%rsp	0xff68
%rdx	0xabcd



Important Note:
Value is **copied** into *Dest*, it is still in memory at old %rsp

Summary: Pushing and popping stack data

```
pushq %rbp
```

is equivalent to

```
subq $8, %rsp
movq %rbp, (%rsp)
```

```
popq %rdx
```

is equivalent to

```
movq (%rsp), %rdx
addq $8, %rsp
```

Today: How does x86-64 implement C procedure calls?

- Mechanisms in procedures
- Calling Conventions
  - Passing Control
  - Passing Data
  - Managing Local Data

Today: How does x86-64 implement C procedure calls?

- **Mechanisms in procedures**
- Calling Conventions
  - Passing Control
  - Passing Data
  - Managing Local Data

Today: How does x86-64 implement C procedure calls?

- Mechanisms in procedures
- Calling Conventions
  - **Passing Control**
  - Passing Data
  - Managing Local Data

What mechanisms do we need to implement procedures?

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

- Passing control
- Passing data
- Managing local data

What mechanisms do we need to implement procedures: **Passing Control**

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

Program needs to stop execution of caller (`multstore`) to execute callee (`mult2`) and then return to caller and continue execution.

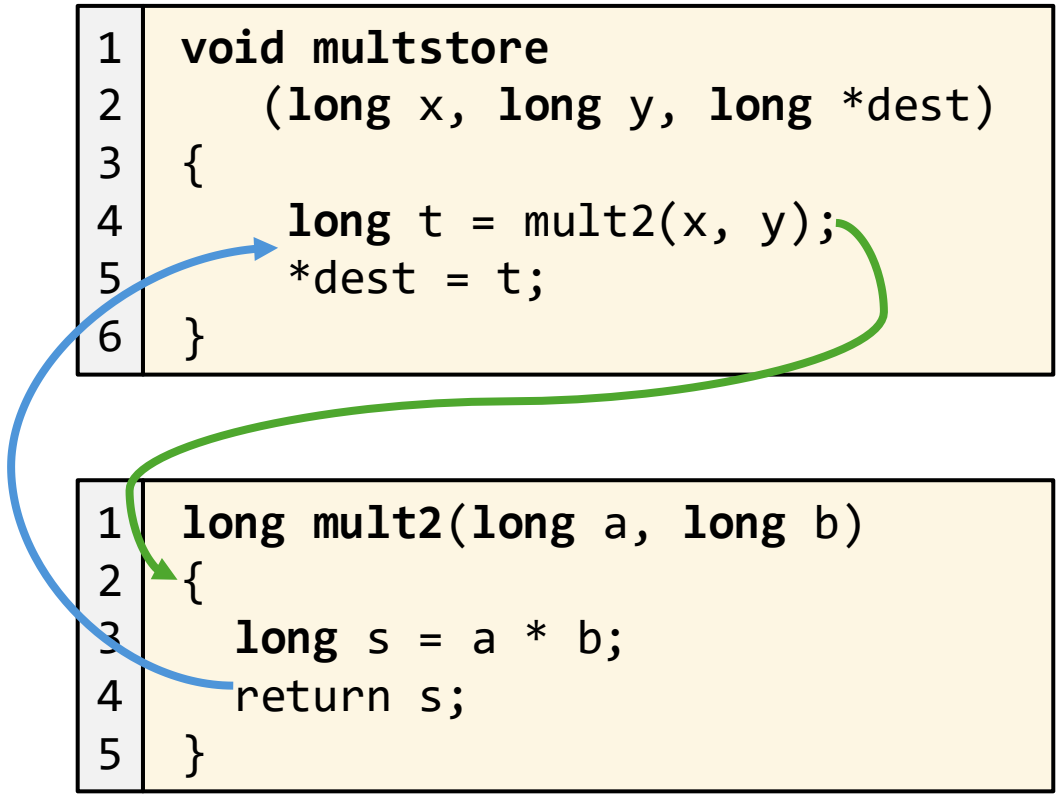
What mechanisms do we need to implement procedures: **Passing Control**

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

Program needs to stop execution of caller (`multstore`) to execute callee (`mult2`) and then return to caller and continue execution.

What mechanisms do we need to implement procedures: **Passing Control**



```
1  void multstore
2     (long x, long y, long *dest)
3  {
4     long t = mult2(x, y);
5     *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

Program needs to stop execution of caller (`multstore`) to execute callee (`mult2`) and then return to caller and continue execution.

What mechanisms do we need to implement procedures: **Passing Data**

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

Program needs to pass arguments to callee (`mult2`) and return values to caller (`multstore`).

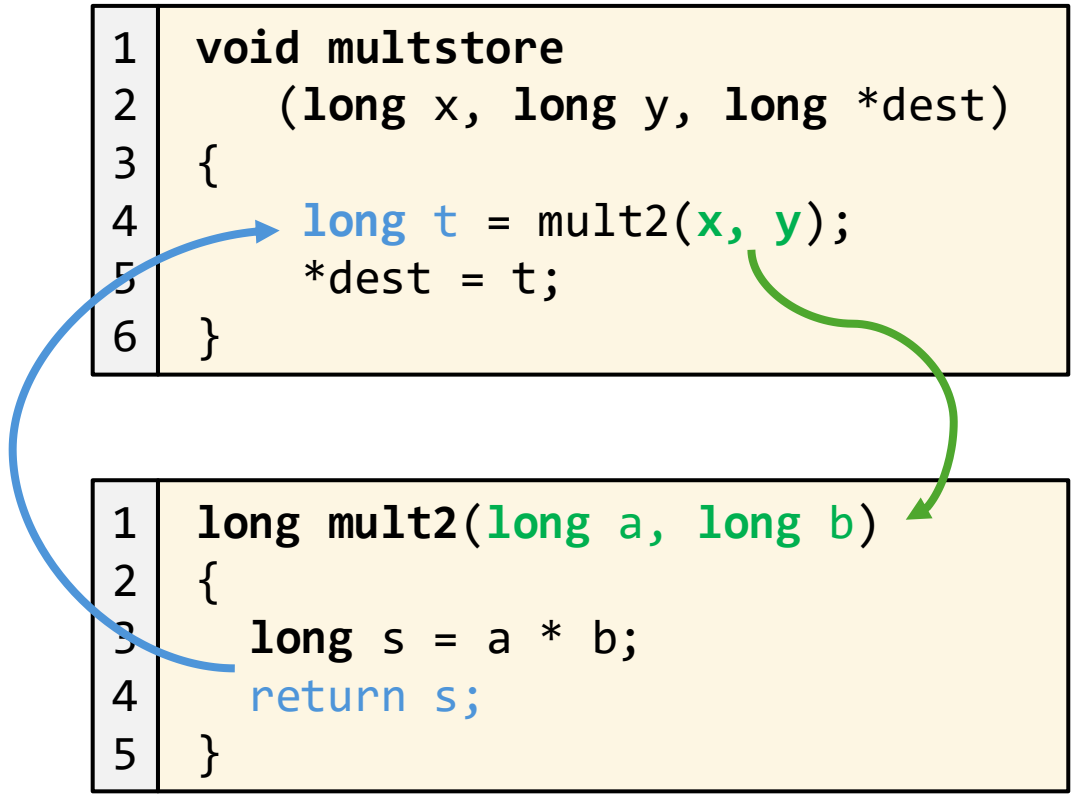
What mechanisms do we need to implement procedures: **Passing Data**

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

Program needs to pass arguments to callee (`mult2`) and return values to caller (`multstore`).

What mechanisms do we need to implement procedures: **Passing Data**



```
1  void multstore
2     (long x, long y, long *dest)
3  {
4     long t = mult2(x, y);
5     *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

Program needs to pass arguments to callee (`mult2`) and return values to caller (`multstore`).

What mechanisms do we need to implement procedures: **Managing Local Memory**

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

Program may need to allocate memory for local variables in callee (`mult2`) and then deallocate after returning to caller (`multstore`).

Today: How does x86-64 implement C procedure calls?

- Mechanisms in procedures
- Calling Conventions
  - **Passing Control**
  - Passing Data
  - Managing Local Data

Let's examine this translation of C to x86-64:

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>   # mult2(x,y)
5    400549: mov    %rax,(%rbx)      # Save at dest
6    40054c: pop    %rbx             # Restore %rbx
7    40054d: ret                     # Return
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

```
1  000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax       # a
3    400553:  imul   %rsi,%rax       # a * b
4    400557:  ret                    # Return
```

To implement passing control, programs use `call` and `ret` instructions which use the stack to store return address.

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx                # Save %rbx
3    400541: mov    %rdx,%rbx           # Save dest
4    400544: call   400550 <mult2>      # mult2(x,y)
5    400549: mov    %rax,(%rbx)         # Save at dest
6    40054c: pop    %rbx                # Restore %rbx
7    40054d: ret                        # Return
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax          # a
3    400553:  imul   %rsi,%rax          # a * b
4    400557:  ret                       # Return
```

`call` instruction pushes the return address on stack and `ret` instruction pops it

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

`call` *addr* <*label*>

1. Push return address on stack
2. Jump to *addr*

`ret`

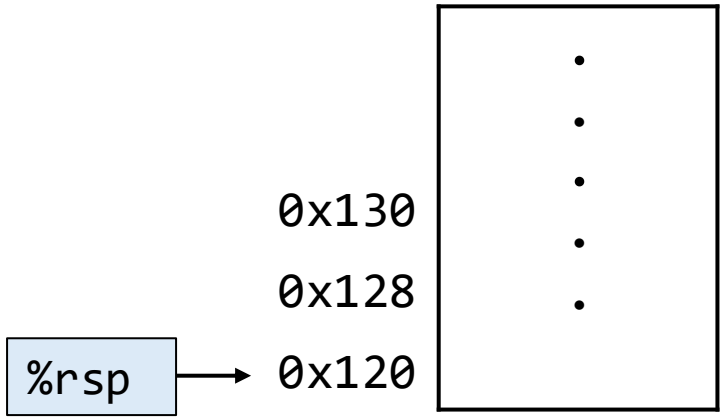
1. Pop return address on stack
2. Jump to return address

Before `call`

%rip →

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

Stack




```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

Registers	
%rsp	0x120
%rip	0x400544

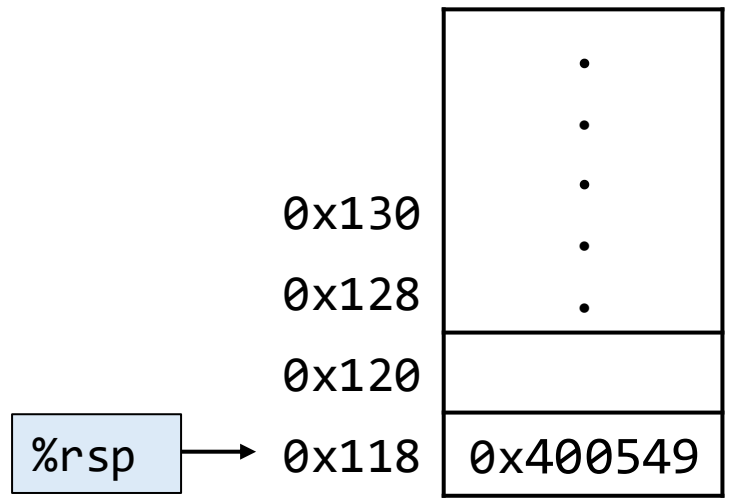
Executing `call`: push return address

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

%rip →

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

Stack



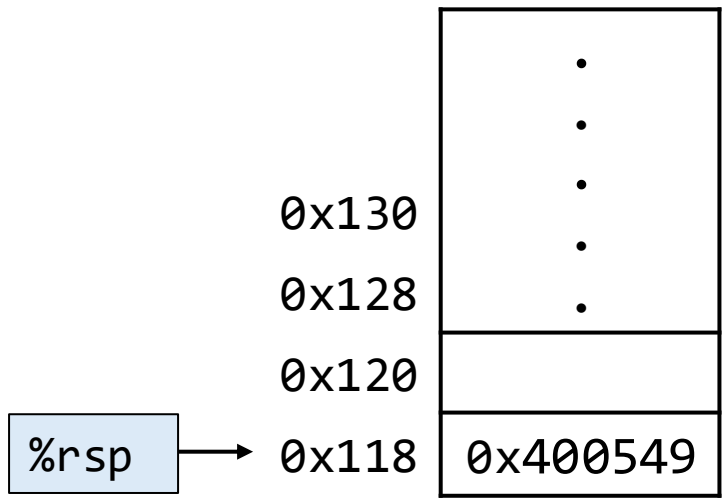
Registers	
%rsp	0x118
%rip	0x400544

Executing `call`: `jmp` to address of function to execute

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx                # Save %rbx
3    400541: mov    %rdx,%rbx           # Save dest
4    400544: call   400550 <mult2>      # mult2(x,y)
5    400549: mov    %rax,(%rbx)         # Save at dest
6    40054c: pop    %rbx                # Restore %rbx
7    40054d: ret                        # Return
```

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax          # a
3    400553:  imul   %rsi,%rax          # a * b
4    400557:  ret                       # Return
```

%rip →



Registers	
%rsp	0x118
%rip	0x400550

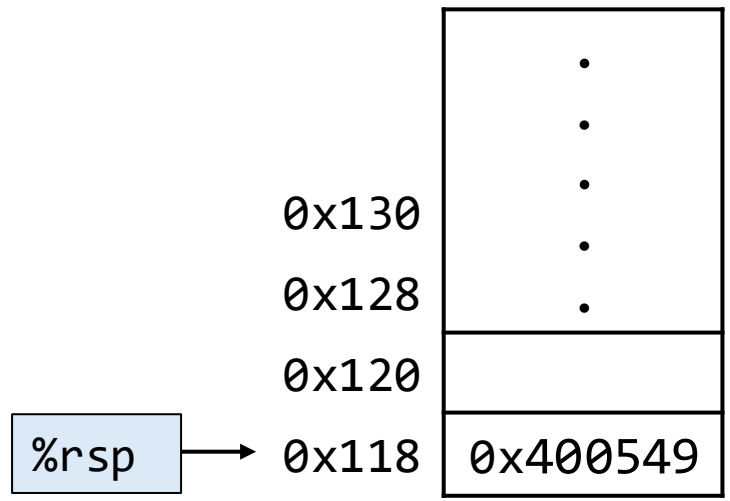
After `call`

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

%rip →

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

Stack



Registers	
%rsp	0x118
%rip	0x400550

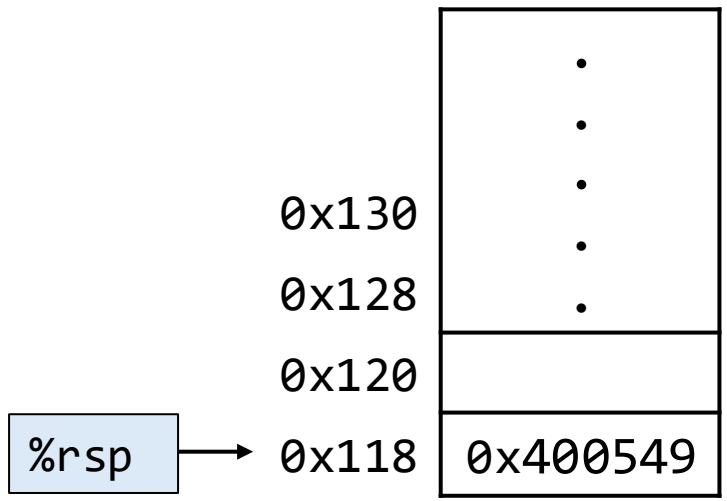
Before `ret`

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

%rip →

Stack



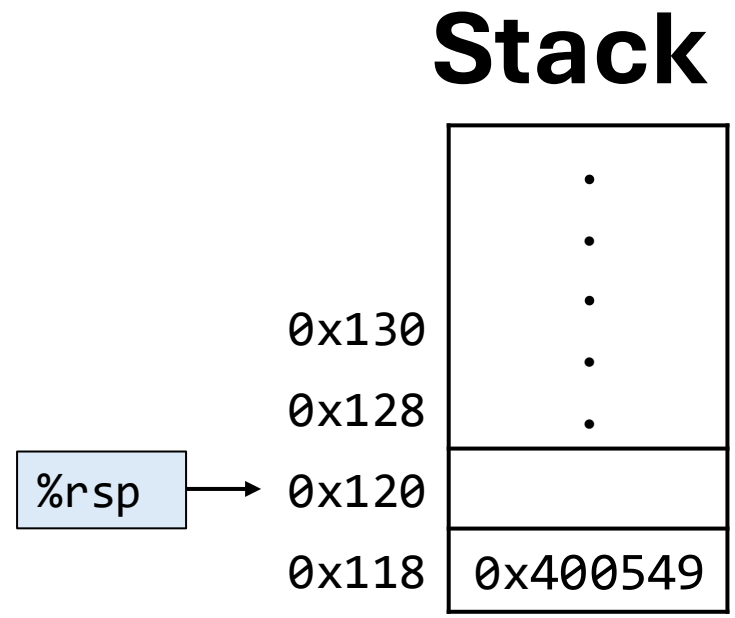
Registers	
%rsp	0x118
%rip	0x400557

Executing `ret`: pop return address

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

%rip →



Registers	
%rsp	0x120
%rip	0x400549

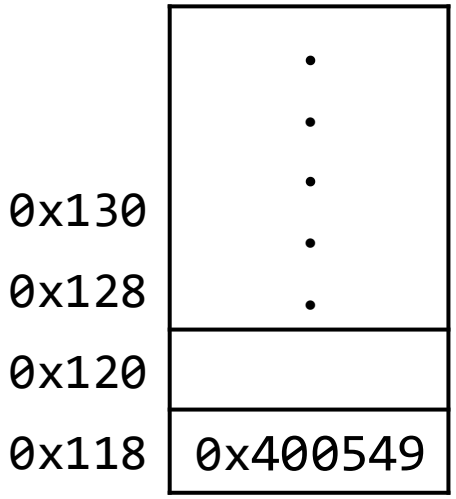
Executing `ret`: `jmp` to return address

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx              # Save %rbx
3    400541: mov    %rdx,%rbx         # Save dest
4    400544: call   400550 <mult2>    # mult2(x,y)
5    400549: mov    %rax,(%rbx)       # Save at dest
6    40054c: pop    %rbx              # Restore %rbx
7    40054d: ret                      # Return
```

%rip →

```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax        # a
3    400553:  imul   %rsi,%rax        # a * b
4    400557:  ret                     # Return
```

Stack



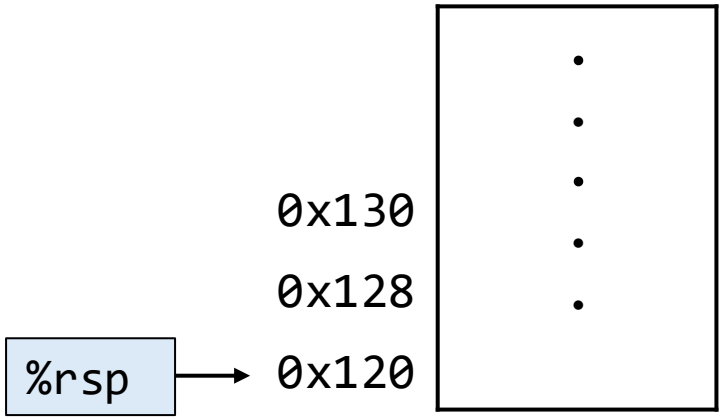
Registers	
%rsp	0x120
%rip	0x400549

After `ret`

%rip →

```
1  0000000000400540 <multstore>:
2    400540: push   %rbx                # Save %rbx
3    400541: mov    %rdx,%rbx           # Save dest
4    400544: call   400550 <mult2>      # mult2(x,y)
5    400549: mov    %rax,(%rbx)         # Save at dest
6    40054c: pop    %rbx                # Restore %rbx
7    40054d: ret                        # Return
```

Stack




```
1  0000000000400550 <mult2>:
2    400550:  mov    %rdi,%rax          # a
3    400553:  imul   %rsi,%rax          # a * b
4    400557:  ret                       # Return
```

Registers	
%rsp	0x120
%rip	0x400549

Today: How does x86-64 implement C procedure calls?

- Mechanisms in procedures
- Calling Conventions
  - Passing Control
  - **Passing Data**
  - Managing Local Data

Activity Time!

If you didn't do at the start of class:

Login to a shark machine, then type:

```
wget http://www.cs.cmu.edu/~213/activities/machine-procedures.pdf
wget http://www.cs.cmu.edu/~213/activities/machine-procedures.tar
tar xf machine-procedures.tar
cd machine-procedures
```

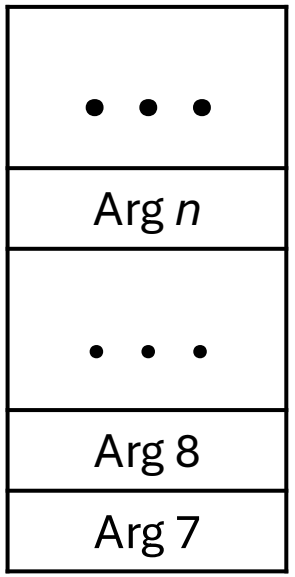
Do Activity: Problems 6-9

The first 6 arguments to a procedure are stored in registers, the remaining are stored on stack.

As illustrated in the activity:

Registers	
`%rdi`	Arg 1
`%rsi`	Arg 2
`%rdx`	Arg 3
`%rcx`	Arg 4
`%r8`	Arg 5
`%r9`	Arg 6

Stack



Important note: Programs only allocate stack space when it is needed.

Example of where arguments and return values are stored:

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

```
1  0000000000400540 <multstore>:
2  # x in %rdi , y in %rsi,  dest in %rdx
3    400540: push   %rbx              # Save %rbx
4    400541: mov    %rdx,%rbx         # Save dest
5    400544: call   400550 <mult2>   # mult2(x,y)
6  # t in %rax
7    400549: mov    %rax,(%rbx)      # Save at dest
8    40054c: pop    %rbx             # Restore %rbx
9    40054d: ret                     # Return
```

```
1  0000000000400550 <mult2>:
2  # a in %rdi, b in %rsi
3    400550:  mov    %rdi,%rax       # a
4    400553:  imul   %rsi,%rax       # a * b
5  # s in %rax
6    400557:  ret                    # Return
```

Arguments are “passed” in the argument registers.

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

```
1  000000000400540 <multstore>:
2  # x in %rdi , y in %rsi,  dest in %rdx
3    400540: push   %rbx              # Save %rbx
4    400541: mov    %rdx,%rbx         # Save dest
5    400544: call   400550 <mult2>   # mult2(x,y)
6  # t in %rax
7    400549: mov    %rax,(%rbx)      # Save at dest
8    40054c: pop    %rbx             # Restore %rbx
9    40054d: ret                     # Return
```

```
1  000000000400550 <mult2>:
2  # a in %rdi, b in %rsi
3    400550:  mov    %rdi,%rax       # a
4    400553:  imul   %rsi,%rax       # a * b
5  # s in %rax
6    400557:  ret                    # Return
```

Return values are “returned” in return register.

```
1  void multstore
2     (long x, long y, long *dest)
3  {
4      long t = mult2(x, y);
5      *dest = t;
6  }
```

```
1  000000000400540 <multstore>:
2  # x in %rdi , y in %rsi,  dest in %rdx
3    400540: push   %rbx              # Save %rbx
4    400541: mov    %rdx,%rbx         # Save dest
5    400544: call   400550 <mult2>   # mult2(x,y)
6  # t in %rax
7    400549: mov    %rax,(%rbx)      # Save at dest
8    40054c: pop    %rbx             # Restore %rbx
9    40054d: ret                     # Return
```

```
1  long mult2(long a, long b)
2  {
3    long s = a * b;
4    return s;
5  }
```

```
1  000000000400550 <mult2>:
2  # a in %rdi, b in %rsi
3    400550:  mov    %rdi,%rax       # a
4    400553:  imul   %rsi,%rax       # a * b
5  # s in %rax
6    400557:  ret                    # Return
```

Today: How does x86-64 implement C procedure calls?

- Mechanisms in procedures
- Calling Conventions
  - Passing Control
  - Passing Data
  - **Managing Local Data**

Languages that support recursion like C require some place to store state for each instantiation of the same procedure.

- Code must be "*reentrant*": Support multiple instances of a single procedure
- Requires needing some place to store state of each instantiation including arguments, local variables, and return value
- The stack allocated in **frames** where each frame contains state for a given procedure

Let's look at an example of calling a chain of functions.

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

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

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

Procedure `amI()` is recursive

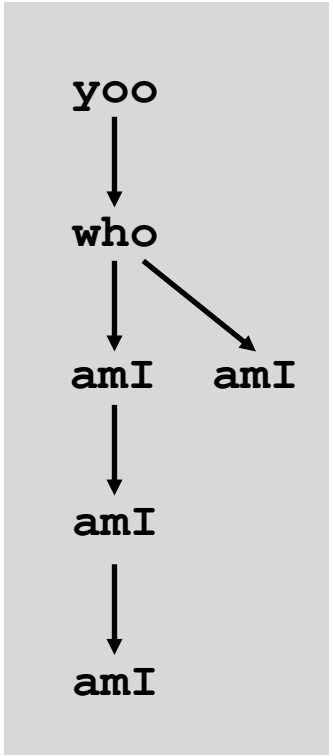
Let's look at an example of calling a chain of functions.

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

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

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

Example Call Chain



Procedure `amI()` is recursive

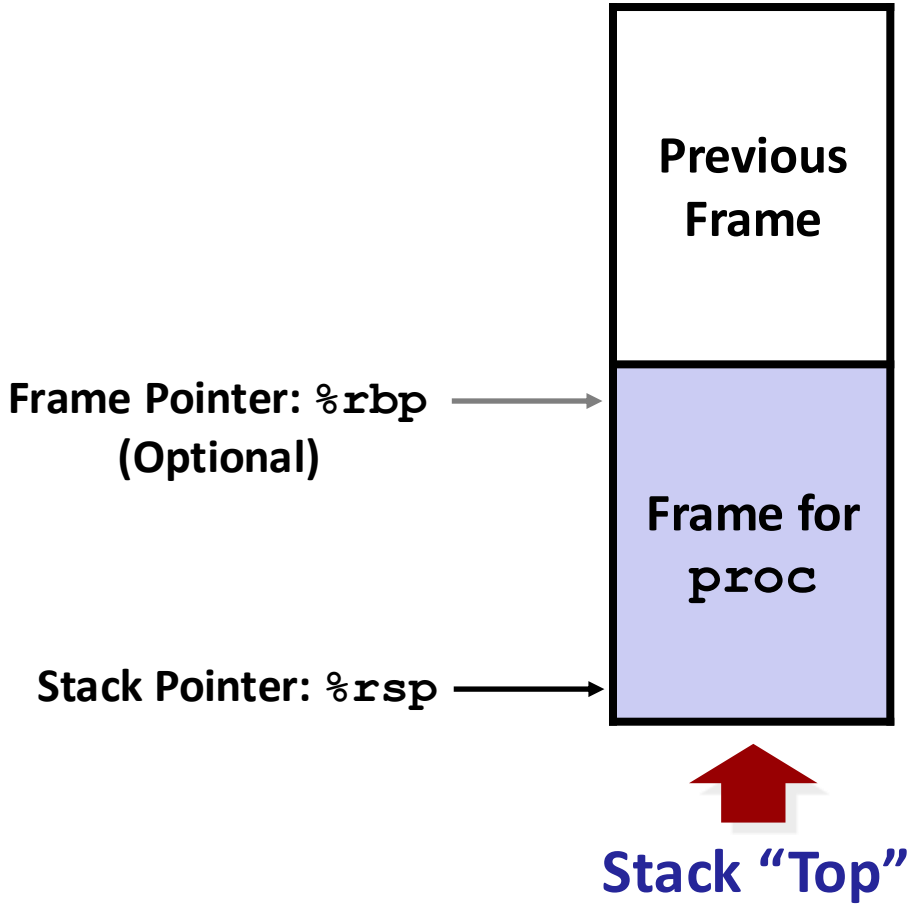
When calling a procedure, that procedure's stack frame is pushed onto the stack.

The "setup" code:

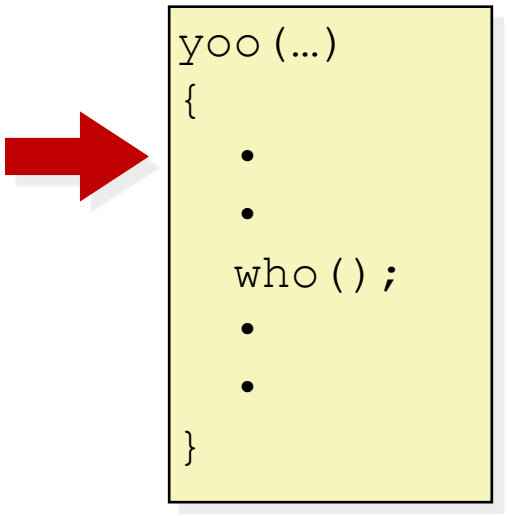
The program allocates space on the stack (if needed) and does `call` instruction.

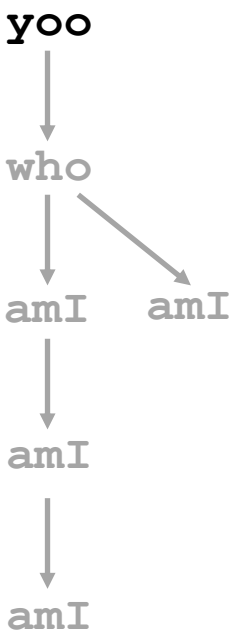
The "finish" code:

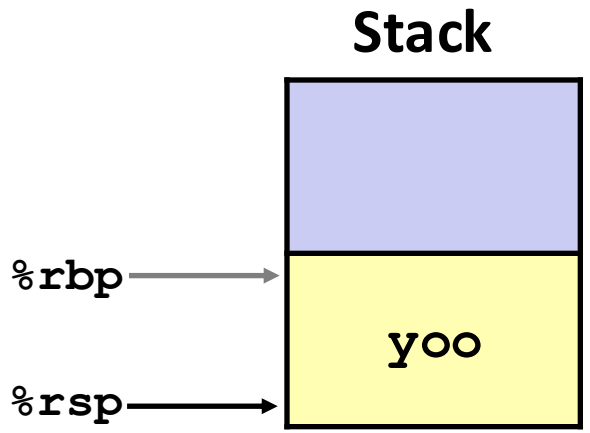
The program deallocates space on the stack (if needed) and does `ret` instruction



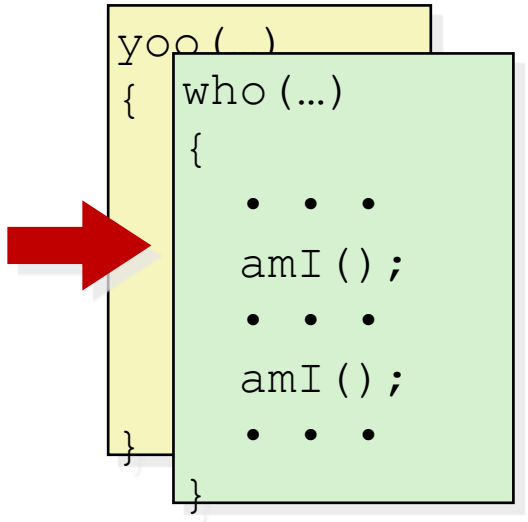
Example

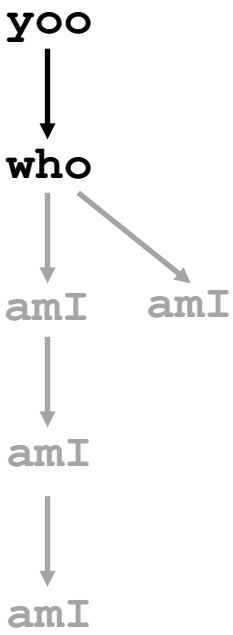


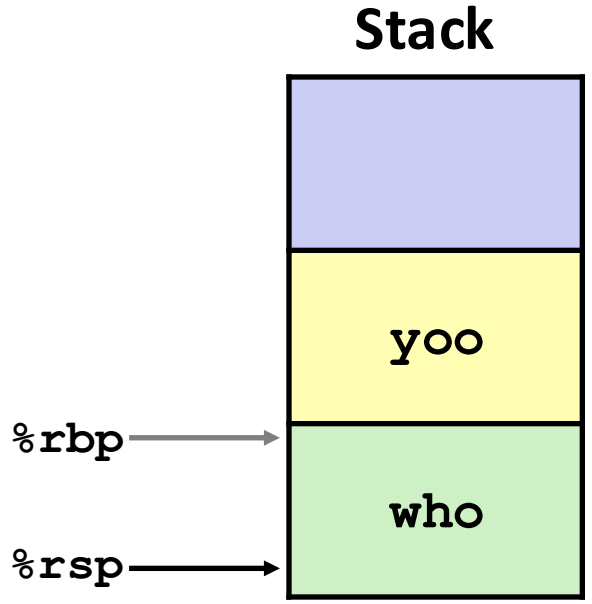




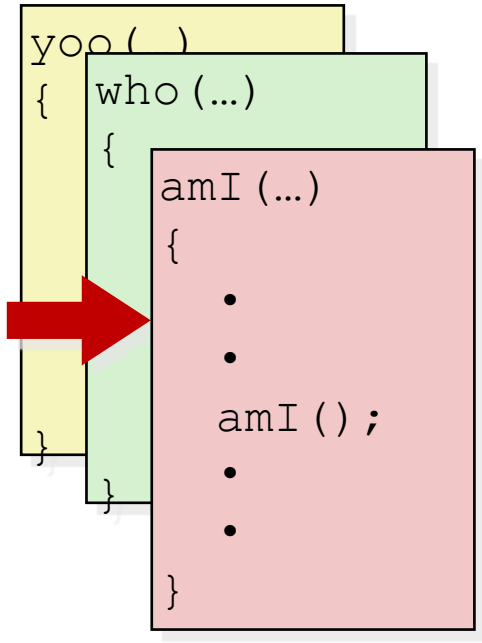
Example

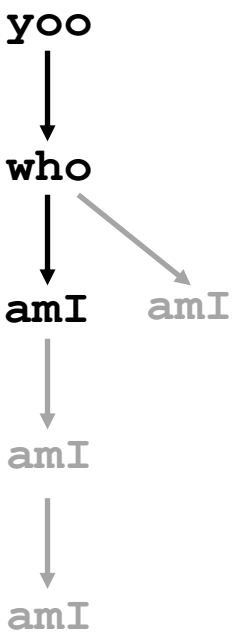


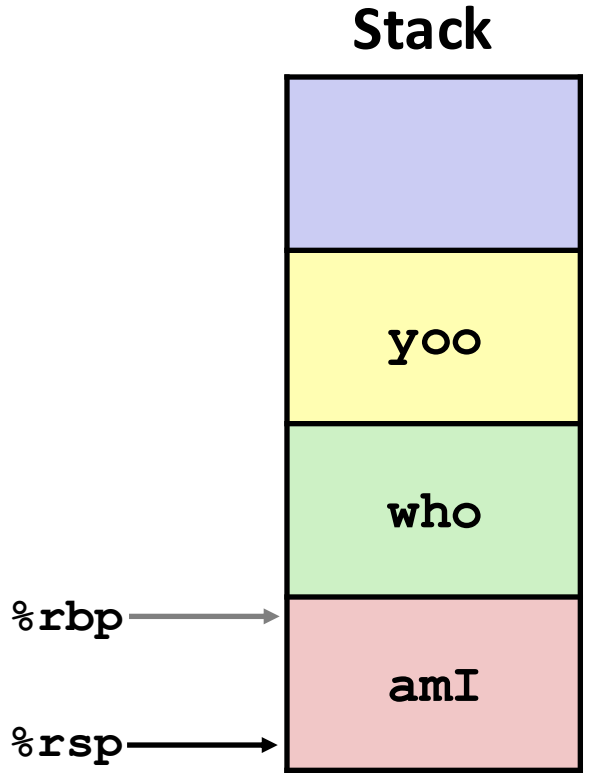




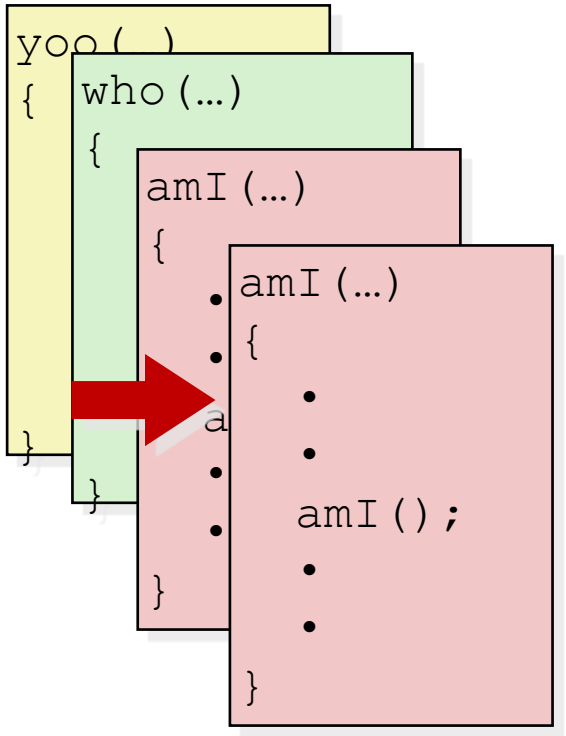
Example

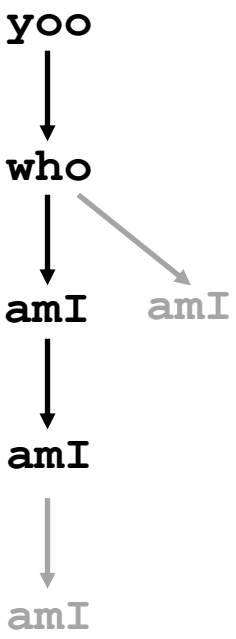

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

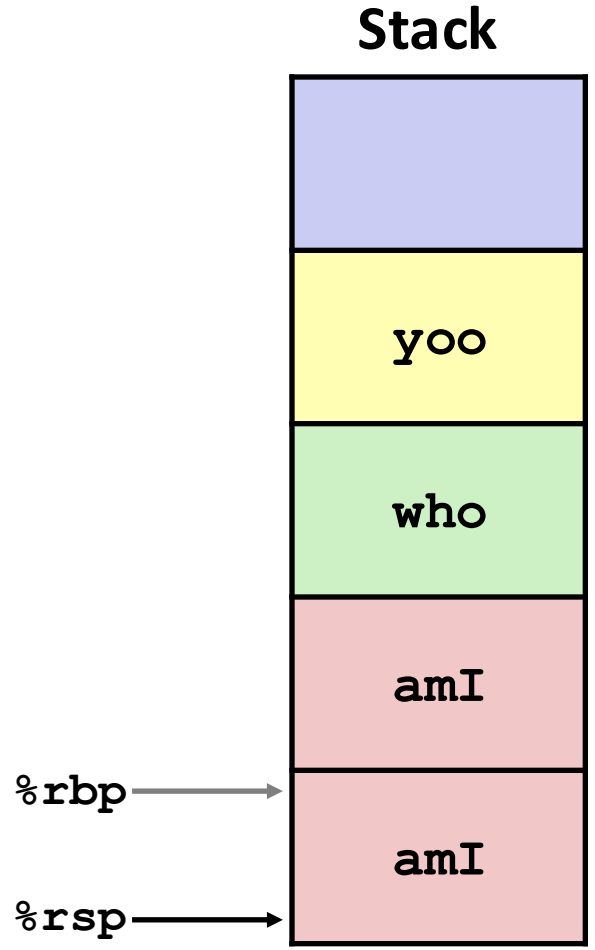

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

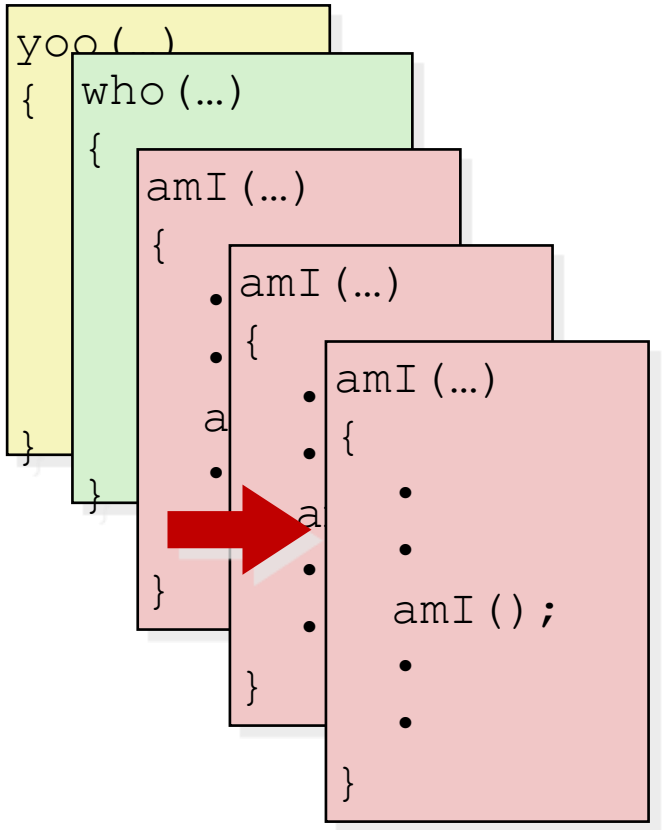
Example

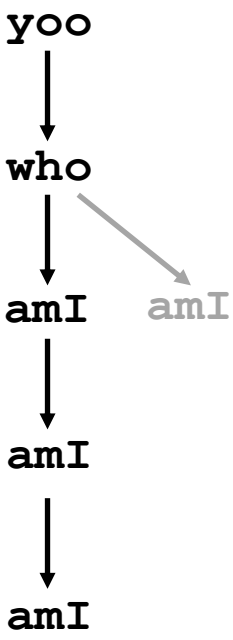

yoo( )
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}
}
}

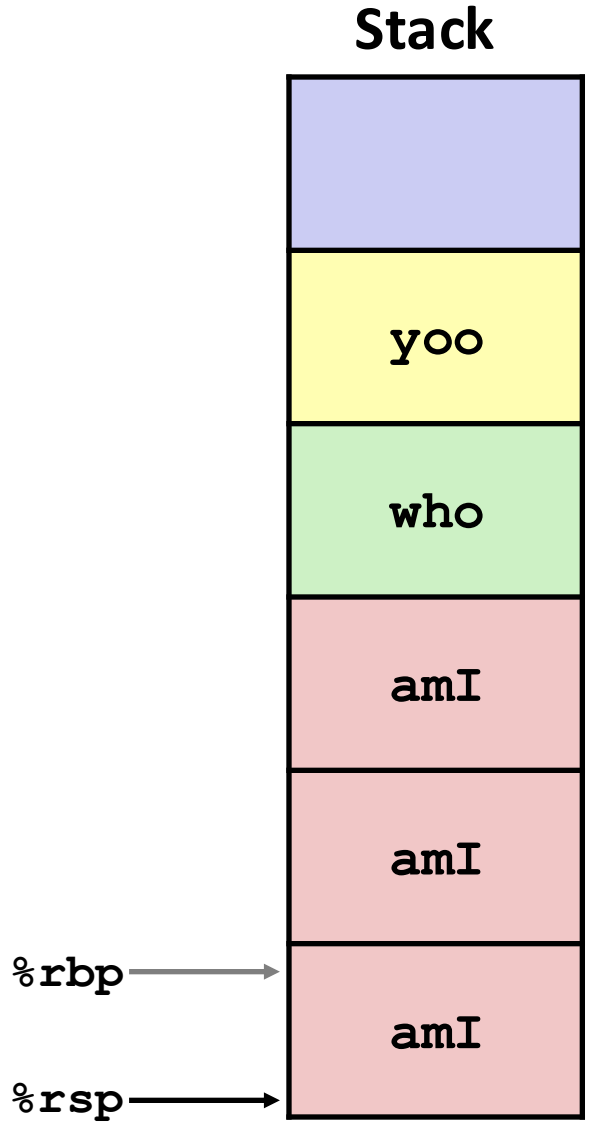

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

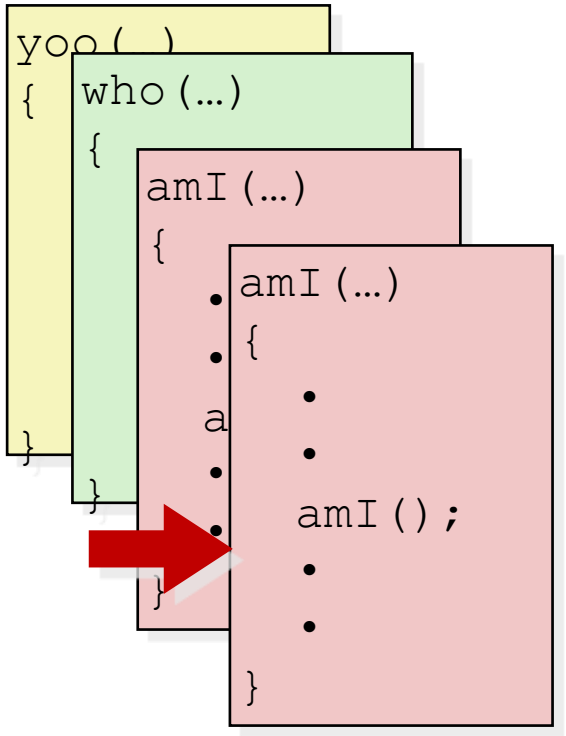
Example

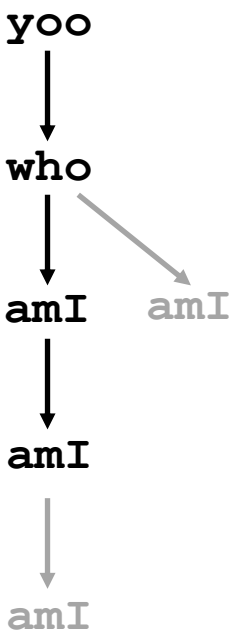

yoo( )
{
who(…)
{
amI(…)
{
amI(…)
{
amI(…)
{
amI();
}

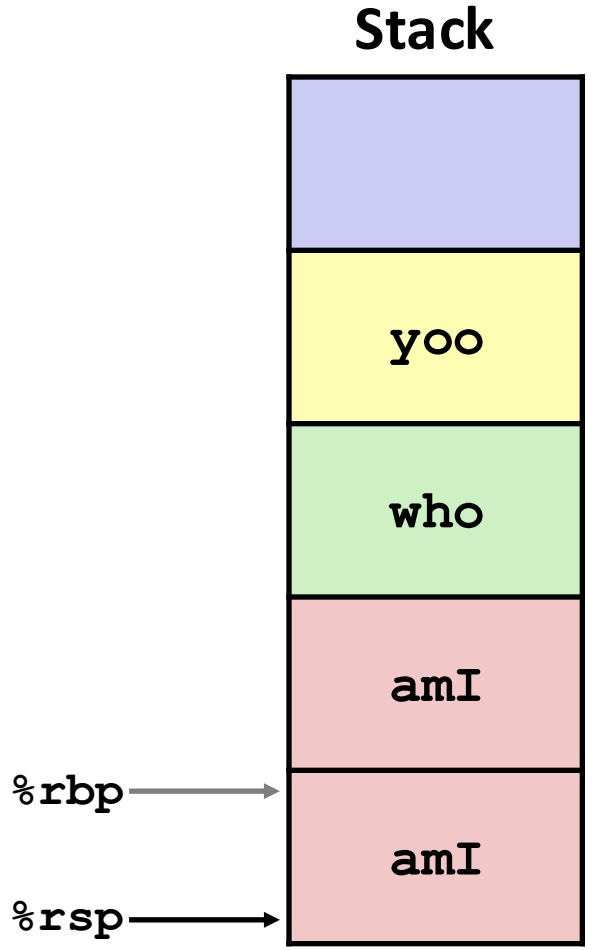

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

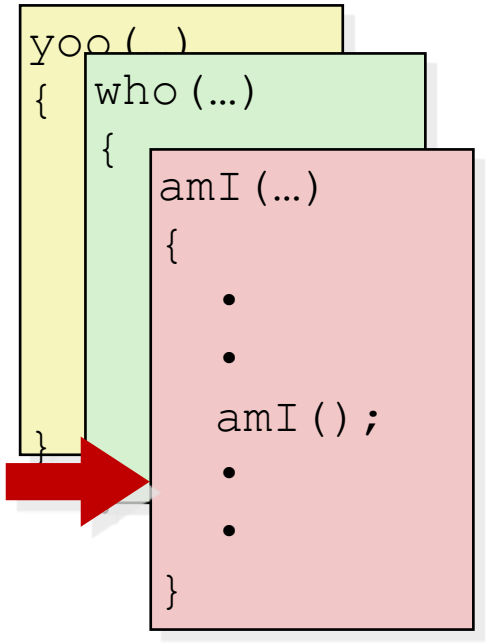
Example

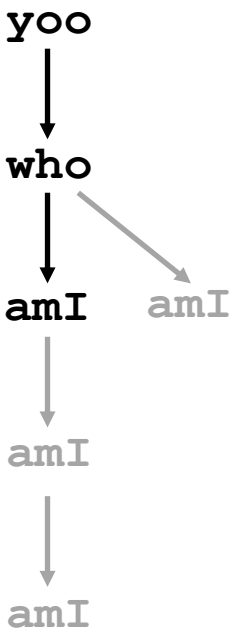

yoo( )
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

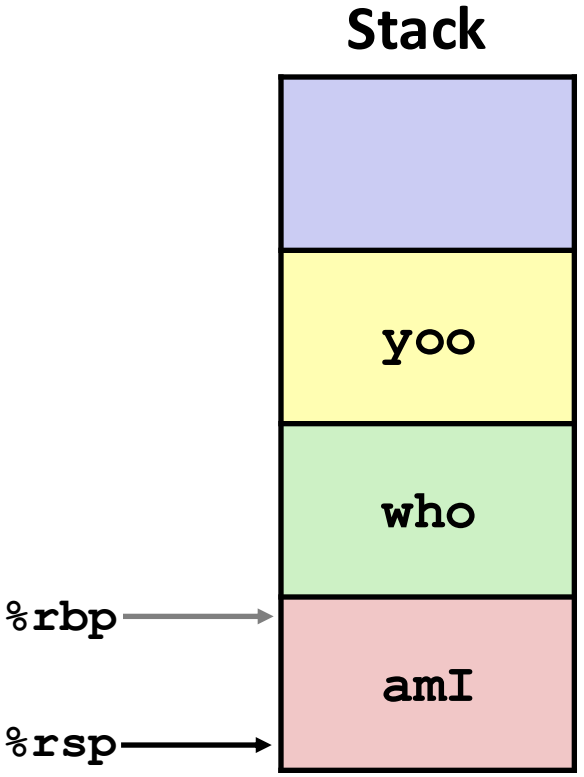

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

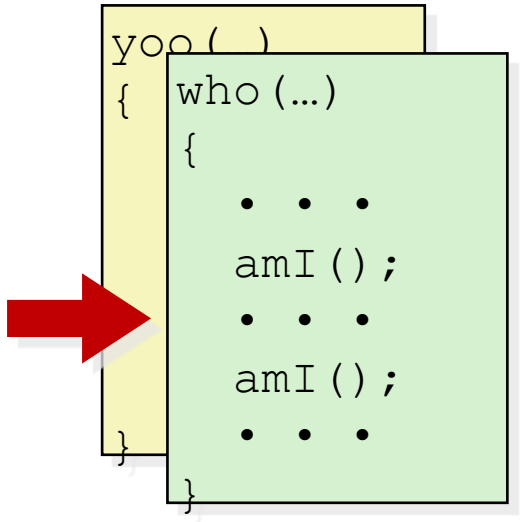
Example

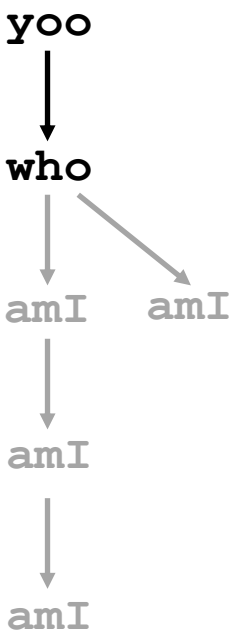

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}

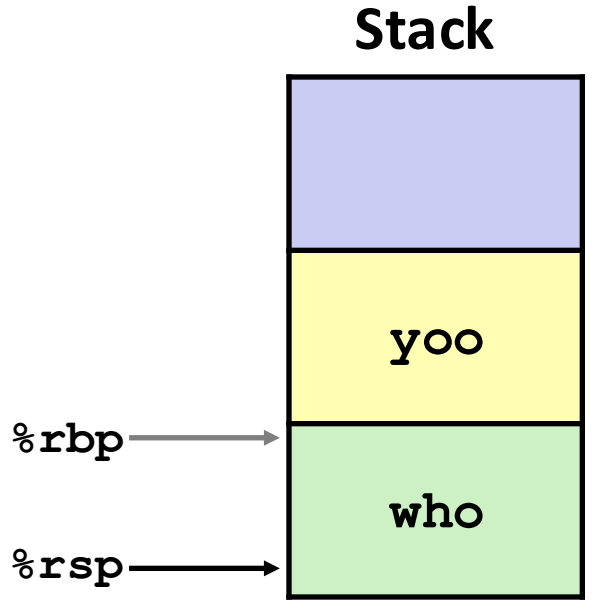

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

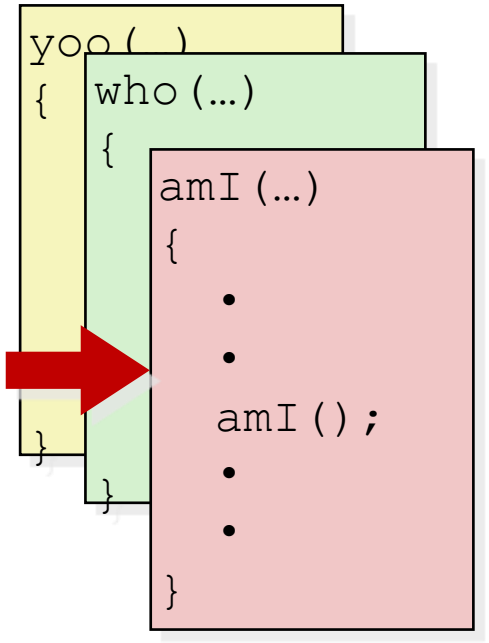
Example

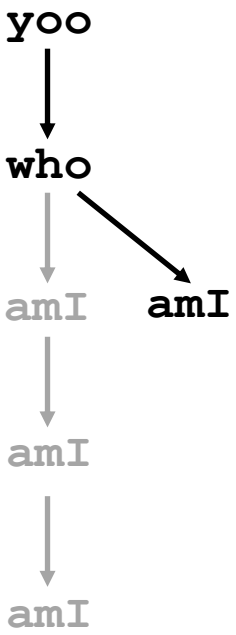


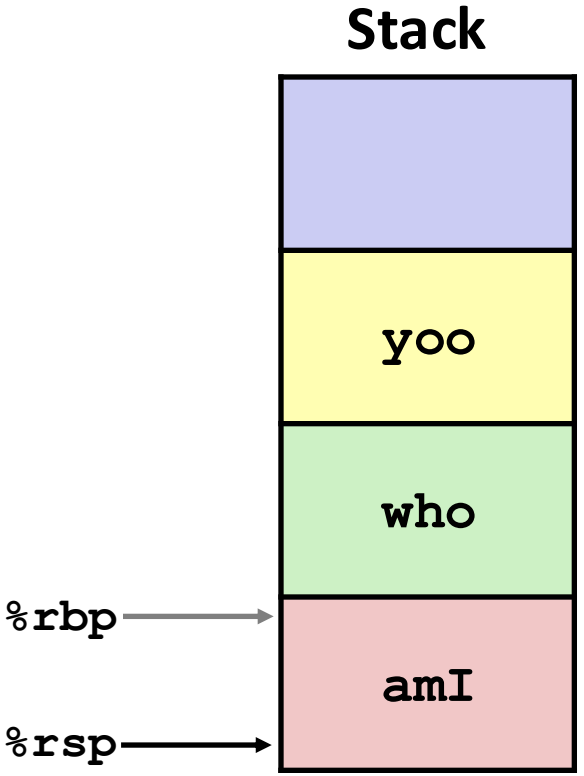




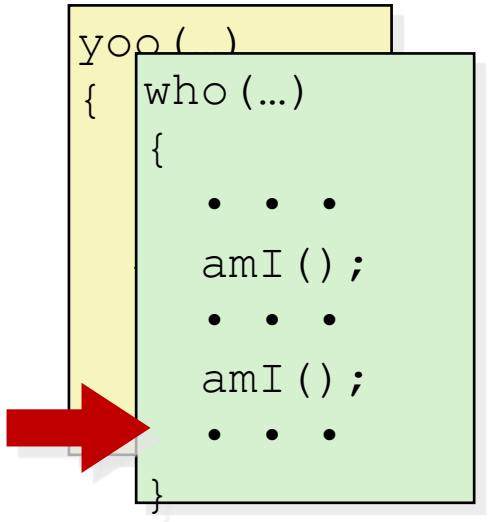
Example

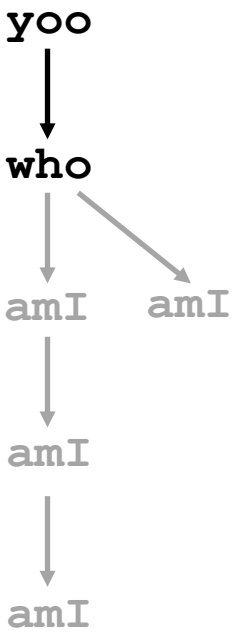

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

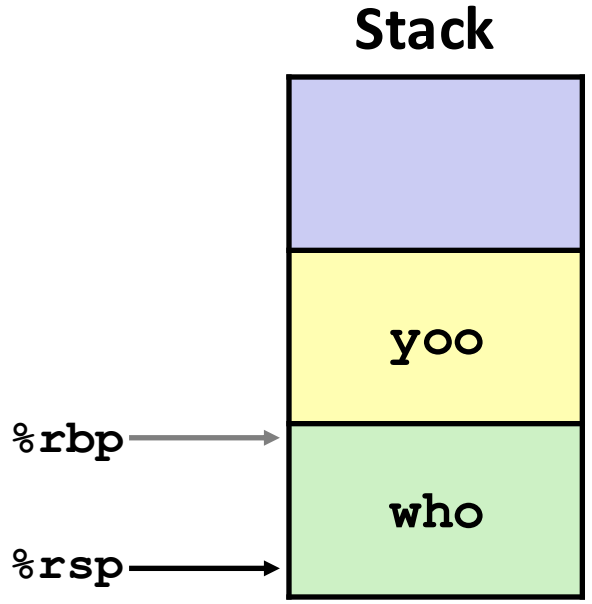

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

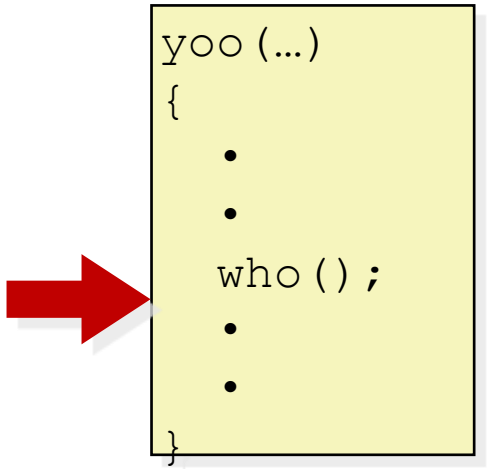
Example

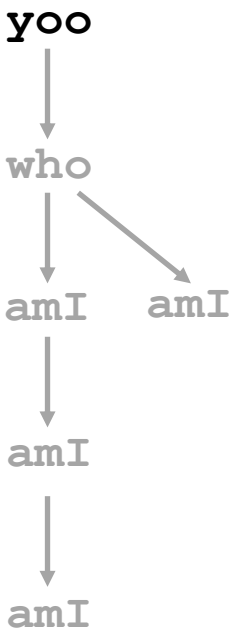


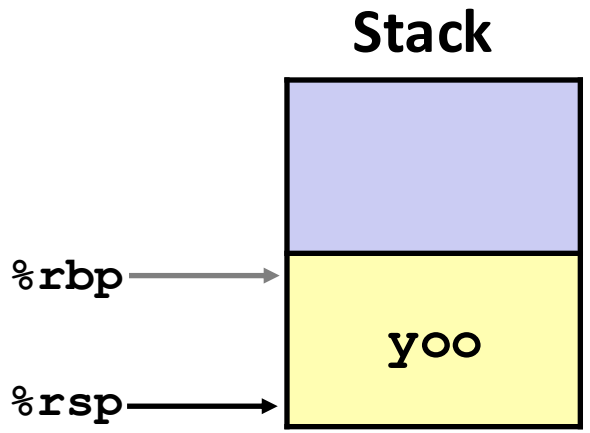




Example







Quiz Time!

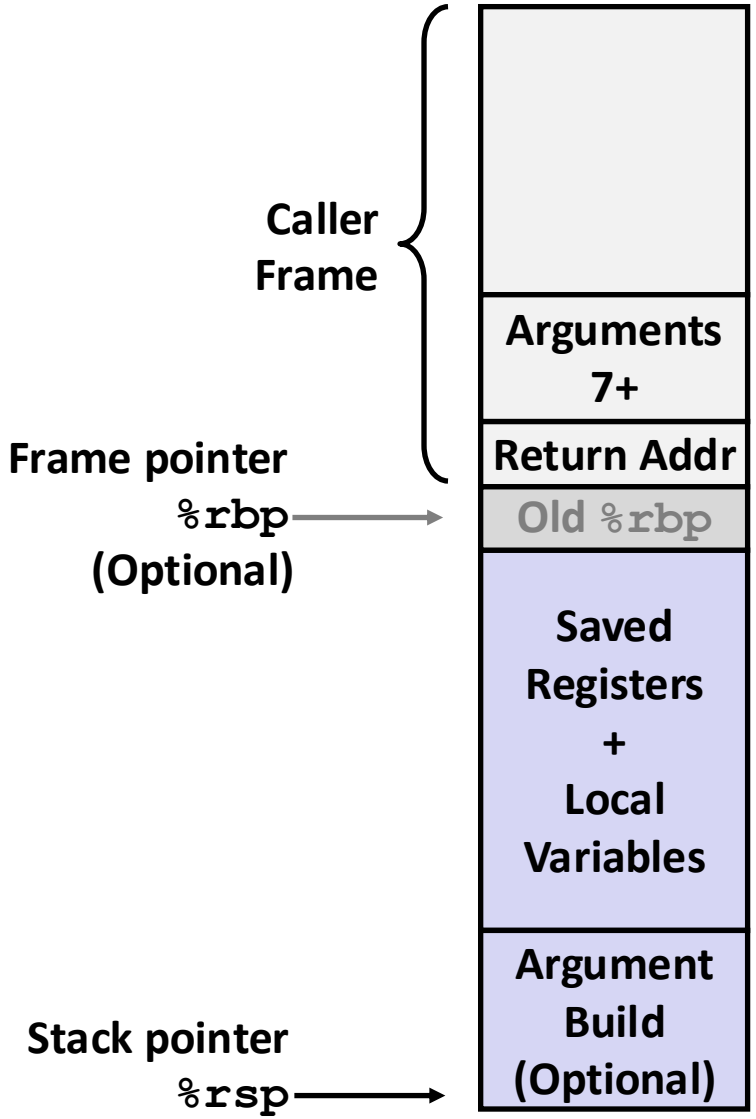
Canvas Quiz: Day 5 - Machine Procedures

https://tinyurl.com/213-lec5

x86-64/Linux Stack Frame

- Current stack frame is on the top of the stack and is the current executing procedure
- Caller stack frame has return address pushed by call instruction
  and the arguments for this call

Important Note:
Stack space only allocated if needed by procedure!



Example: `incr`

```
1  long incr(long *p, long val) {
2      long x = *p;
3      long y = x + val;
4      *p = y;
5      return x;
6  }
```

```
1  incr:
2    movq    (%rdi), %rax
3    addq    %rax, %rsi
4    movq    %rsi, (%rdi)
5    ret
```

Registers	Use
%rdi	Argument **p**
%rsi	Argument **val**, **y**
%rax	**x**, Return value

Example: Calling `incr` with a local variable address

```
1  long incr(long *p, long val) {
2      long x = *p;
3      long y = x + val;
4      *p = y;
5      return x;
6  }
```

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  incr:
2    movq    (%rdi), %rax
3    addq    %rax, %rsi
4    movq    %rsi, (%rdi)
5    ret
```

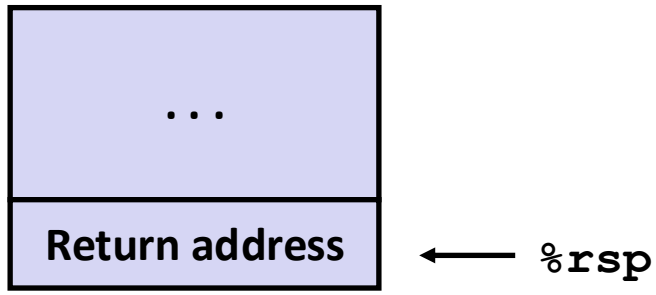
```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Before calling `incr`: Storing arguments

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Initial Stack Structure

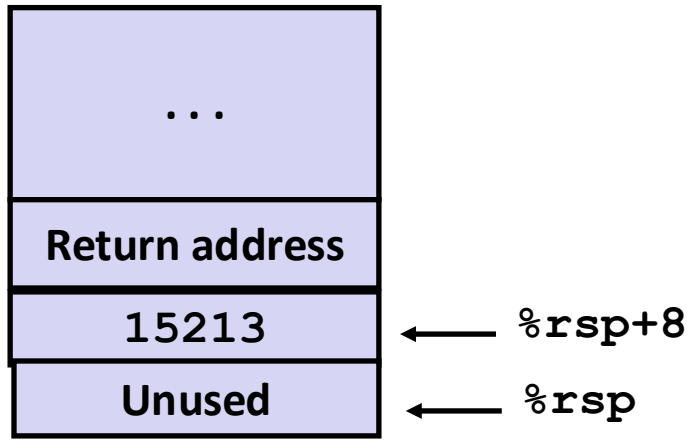


Before calling `incr`: Storing arguments

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Resulting Stack Structure

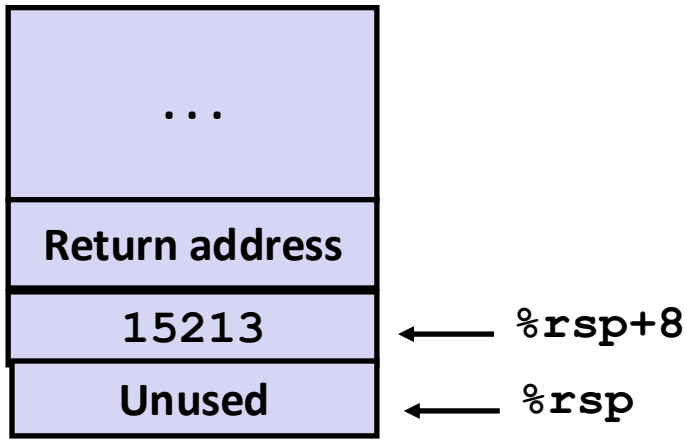


Before calling `incr`: Storing arguments

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Stack Structure



Registers	Use
%rdi	&v1
%rsi	3000

Before calling `incr`: Storing arguments

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_i
2    subq
3    movq
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Stack Structure

...sp+8

...sp

Aside 1: **`movl $3000, %esi`**

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

Registers	Use
%rdi	&v1
%rsi	3000

Before calling `incr`: Storing arguments

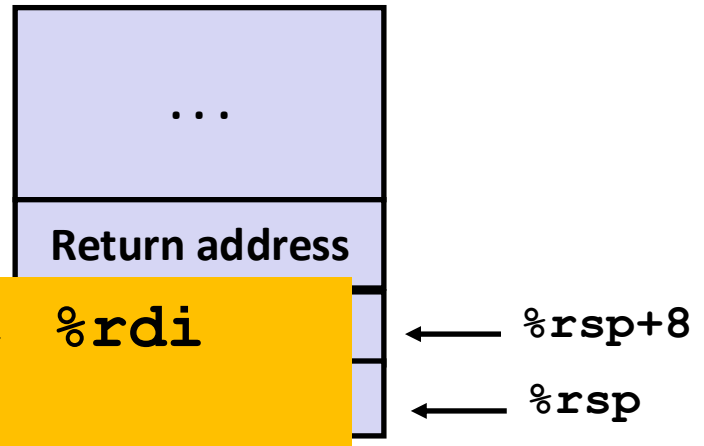
```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_ir
2    subq
3    movq
4    movl
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

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

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

Stack Structure



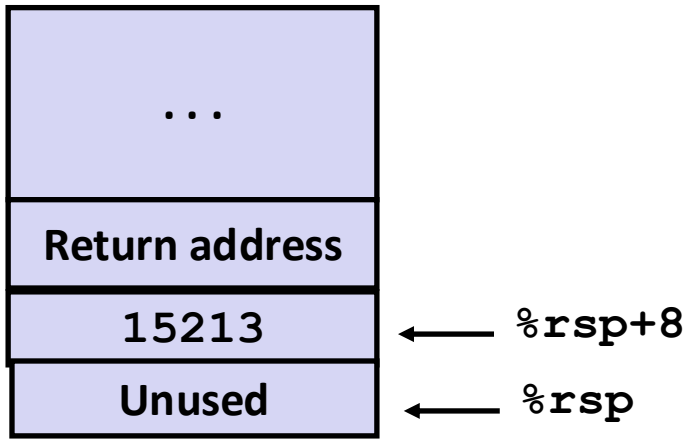
Registers	Use
%rdi	&v1
%rsi	3000

Before calling `incr`: Storing arguments

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Stack Structure



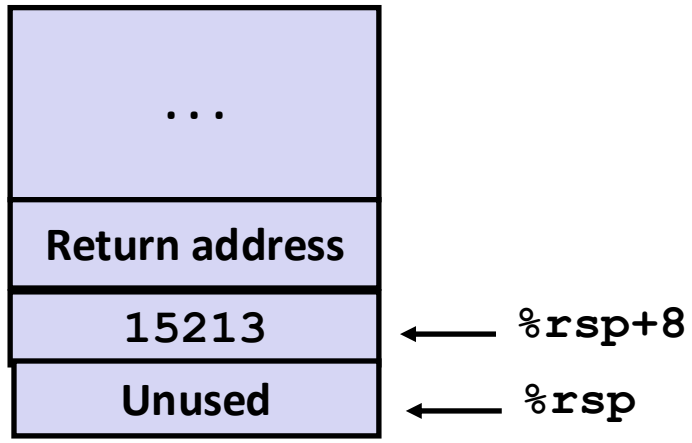
Registers	Use
%rdi	&v1
%rsi	3000

Before calling `incr`: Storing arguments

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Stack Structure



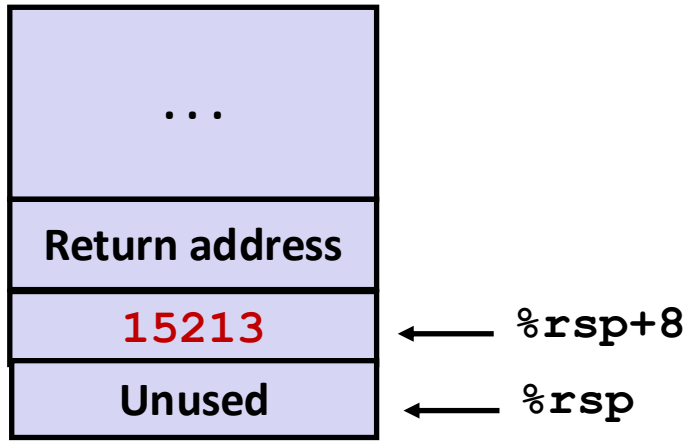
Registers	Use
%rdi	&v1
%rsi	3000

Calling `incr`:

```
1  long incr(long *p, long val) {
2      long x = *p;
3      long y = x + val;
4      *p = y;
5      return x;
6  }
```

```
1  incr:
2    movq    (%rdi), %rax
3    addq    %rax, %rsi
4    movq    %rsi, (%rdi)
5    ret
```

Stack Structure



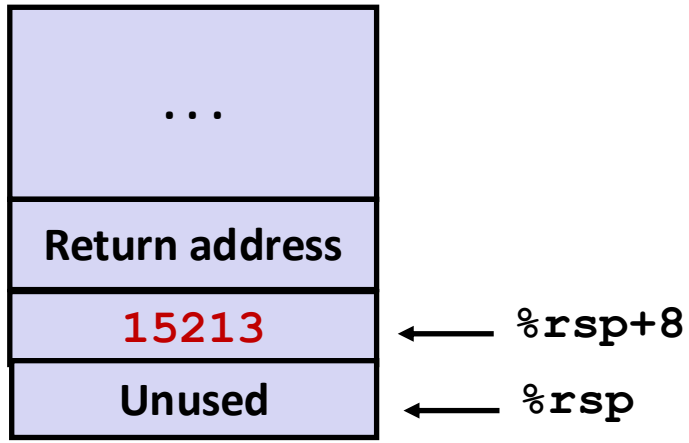
Registers	Use
%rdi	**&v1**
%rsi	3000
%rax	Return value

Calling `incr`:

```
1  long incr(long *p, long val) {
2      long x = *p;
3      long y = x + val;
4      *p = y;
5      return x;
6  }
```

```
1  incr:
2    movq    (%rdi), %rax
3    addq    %rax, %rsi
4    movq    %rsi, (%rdi)
5    ret
```

Stack Structure



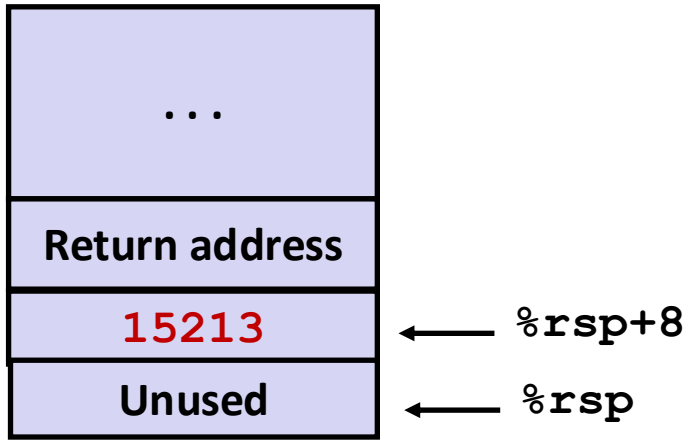
Registers	Use
%rdi	**&v1**
%rsi	3000
%rax	Return value

After calling `incr`: Getting return value

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Stack Structure



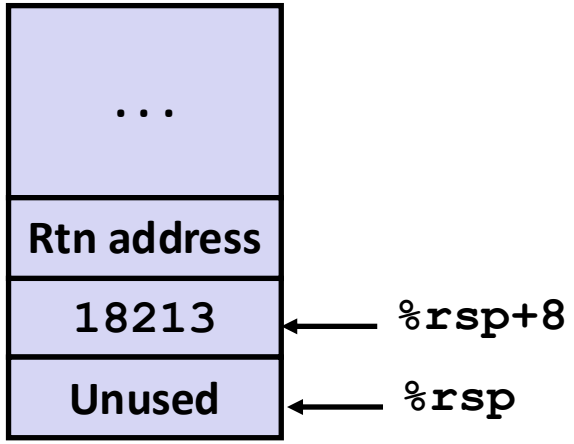
Registers	Use
%rax	Return value

After calling `incr`: Calculating return value

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

Stack Structure

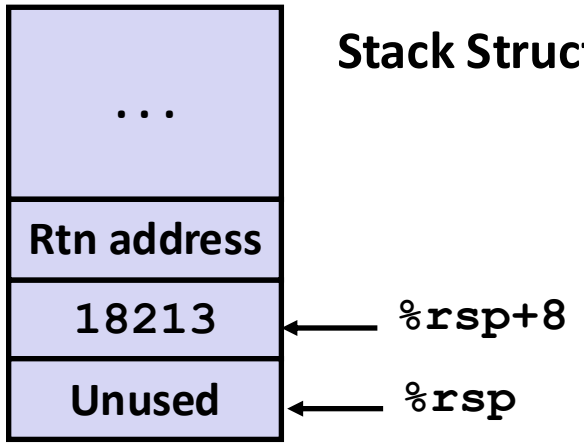


Registers	Use
%rax	Return value

After calling `incr`: Calculating return value

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```

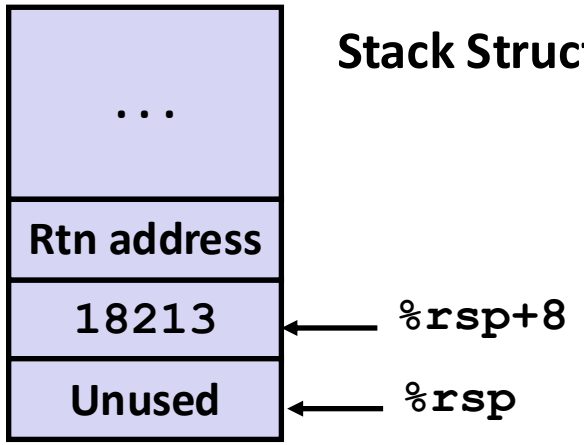


Registers	Use
%rax	Return value

After calling `incr`: Deallocate space

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

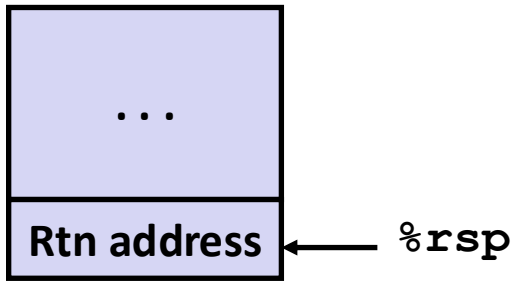
```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```



Stack Structure

Registers	Use
%rax	Return value

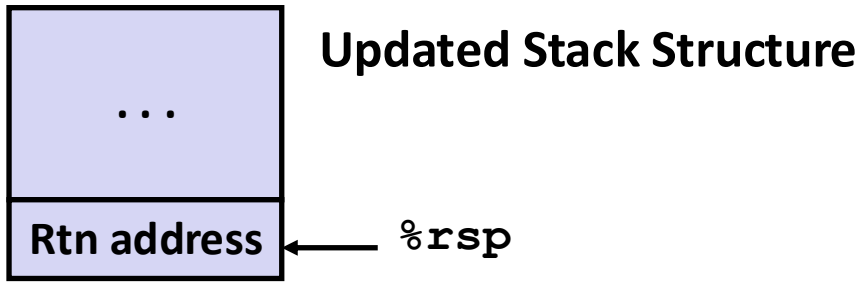
Updated Stack Structure



After calling `incr`: Returning

```
1  long call_incr() {
2      long v1 = 15213;
3      long v2 = incr(&v1, 3000);
4      return v1+v2;
5  }
```

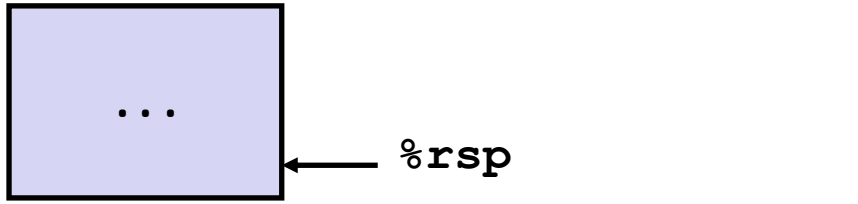
```
1  call_incr:
2    subq    $16, %rsp
3    movq    $15213, 8(%rsp)
4    movl    $3000, %esi
5    leaq    8(%rsp), %rdi
6    call    incr
7    addq    8(%rsp), %rax
8    addq    $16, %rsp
9    ret
```



Updated Stack Structure

Registers	Use
%rax	Return value

Final Stack Structure



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
```

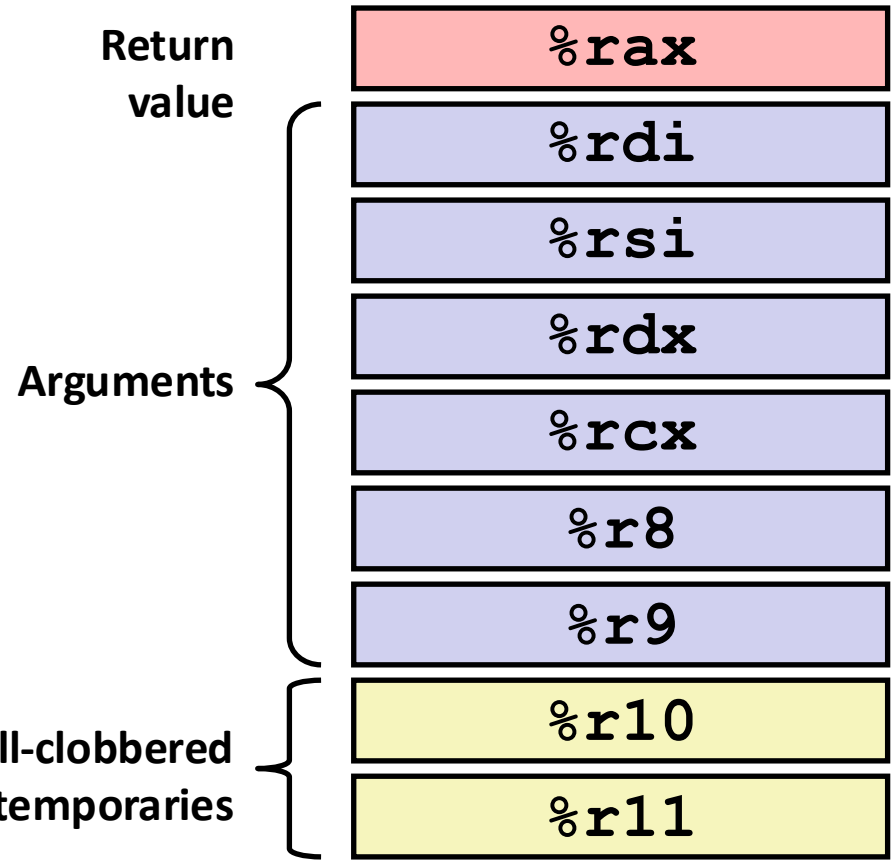
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble → something should be done!
    - Need some coordination

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

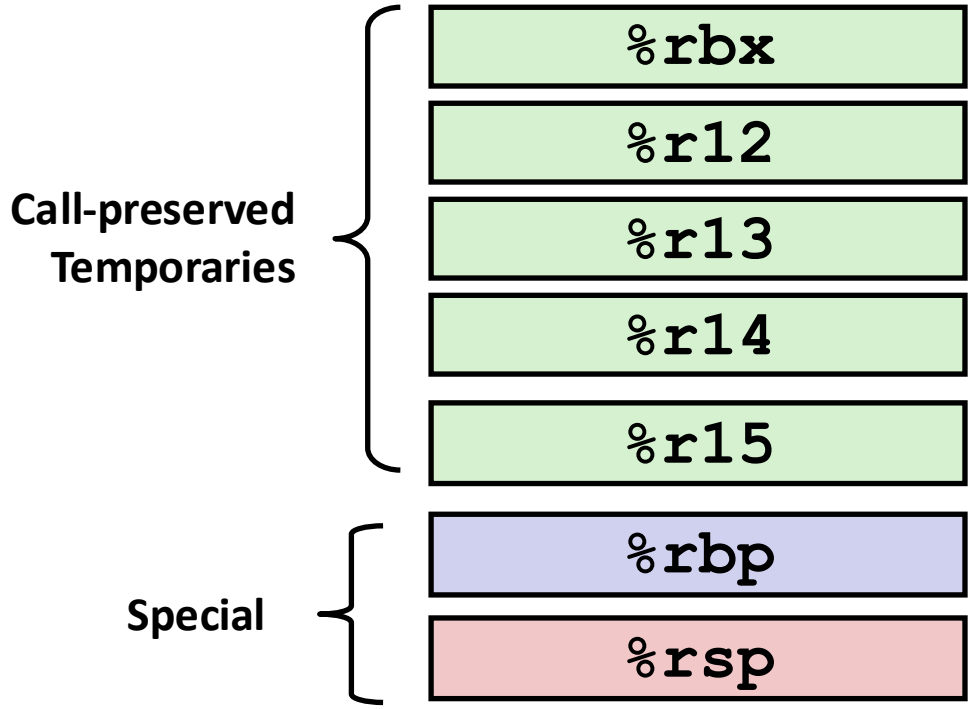
x86-64 Linux Register Usage #1

- `%rax`
  - Return value
  - Call-clobbered
    (i.e., caller must save&restore if value needed after the call)
- `%rdi`, …, `%r9`
  - Arguments
  - Call-clobbered
- `%r10`, `%r11`
  - Call-clobbered



x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14`, `%r15`
  - Call-preserved (i.e., Callee must save & restore)
- `%rbp`
  - Call-preserved
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of call-preserved
  - Restored to original value upon exit from procedure



x86-64 Procedure Summary

Important Points

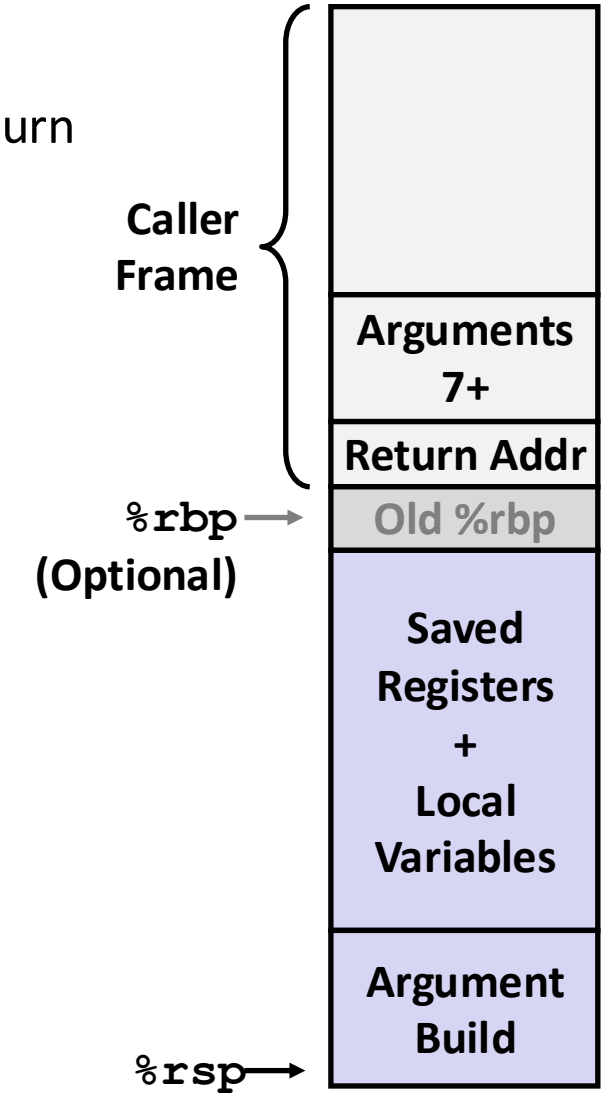
- Stack is the right data structure for procedure call/return
  - If P calls Q, then Q returns before P

Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in call-preserved registers
- Put function arguments at top of stack
- Result return in `%rax`

Pointers are addresses of values

- **On stack or global**



Additional Slides

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
```

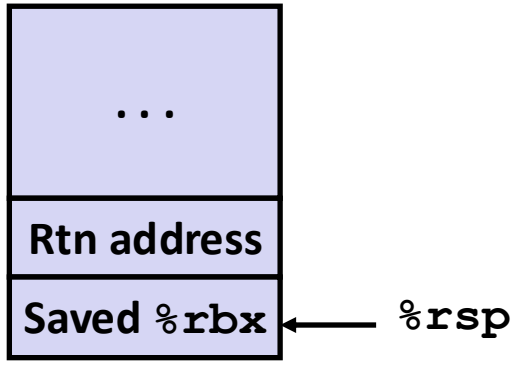
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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
```

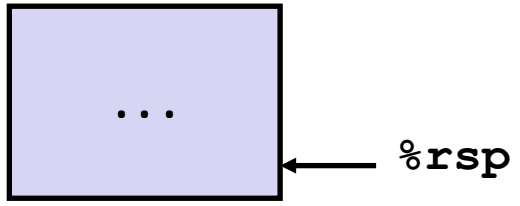
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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**



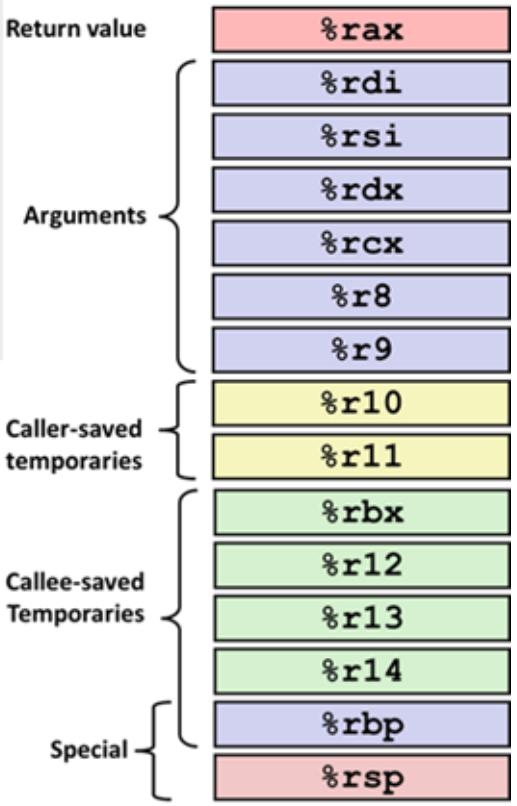
Observations About Recursion

- **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 in Lecture 9)
  - 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

Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4, long a5,
    long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4)+
        add5(a5, a6, a7, a8, a9);
}
```



- Where are `a0`,…, `a9` passed?

  `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, stack

- Where are `b0`,…, `b4` passed?

  `rdi`, `rsi`, `rdx`, `rcx`, `r8`

- Which registers do we need to save?

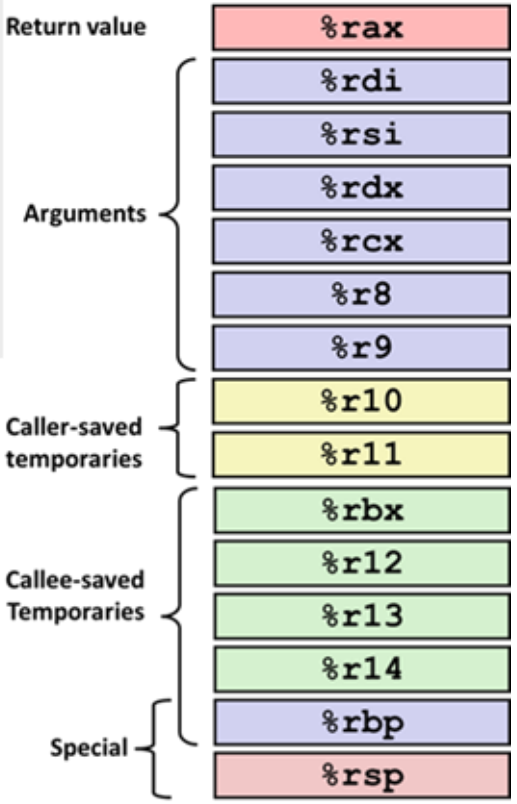
  Ill-posed question. Need assembly.

  `rbx`, `rbp`, `r9` (during first call to `add5`)

Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4, long a5,
    long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4)+
        add5(a5, a6, a7, a8, a9);
}
```

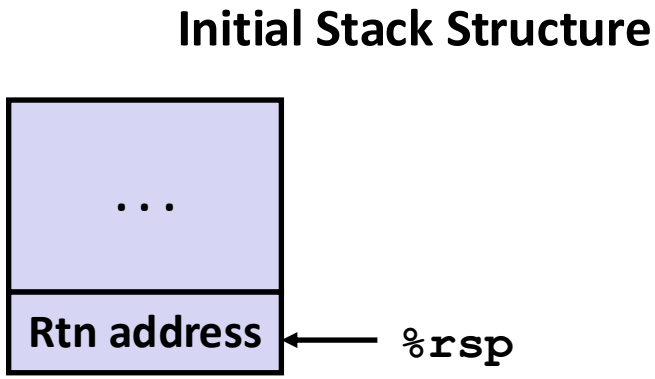



```
add10:
        pushq   %rbp
        pushq   %rbx
        movq    %r9, %rbp
        call    add5
        movq    %rax, %rbx
        movq    48(%rsp), %r8
        movq    40(%rsp), %rcx
        movq    32(%rsp), %rdx
        movq    24(%rsp), %rsi
        movq    %rbp, %rdi
        call    add5
        addq    %rbx, %rax
        popq    %rbx
        popq    %rbp
        ret
```

```
add5:
        addq    %rsi, %rdi
        addq    %rdi, %rdx
        addq    %rdx, %rcx
        leaq    (%rcx,%r8), %rax
        ret
```

Callee-Saved Example #1

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



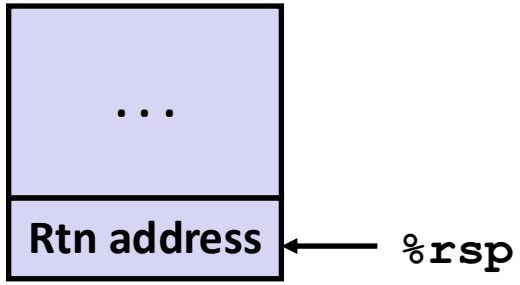
- X comes in register **`%rdi`**.
- We need **`%rdi`** for the call to incr.
- Where should be put x, so we can use it after the call to incr?

Callee-Saved Example #2

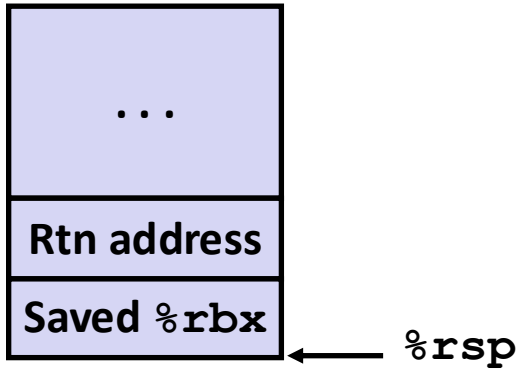
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

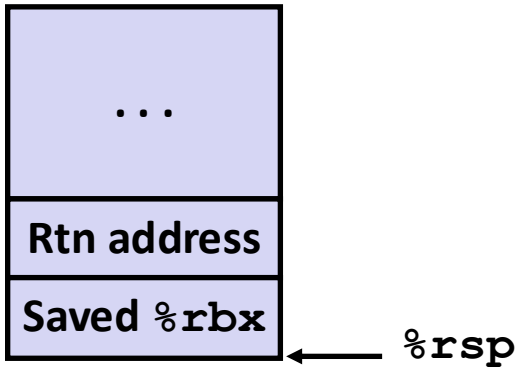


Callee-Saved Example #3

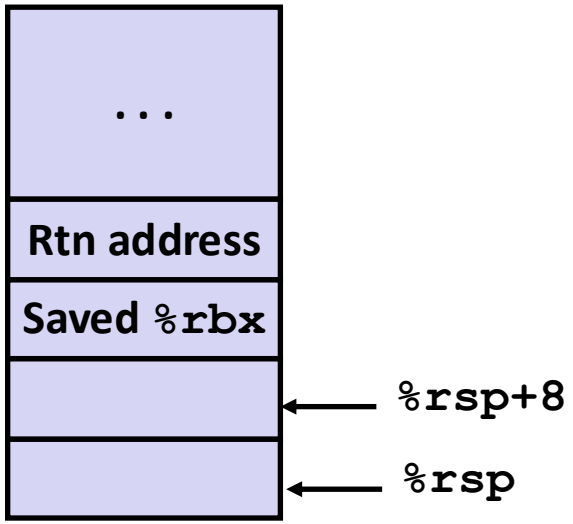
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

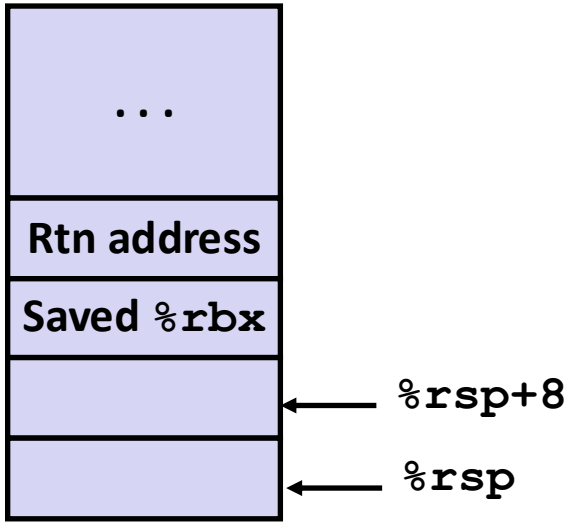


Callee-Saved Example #4

```
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



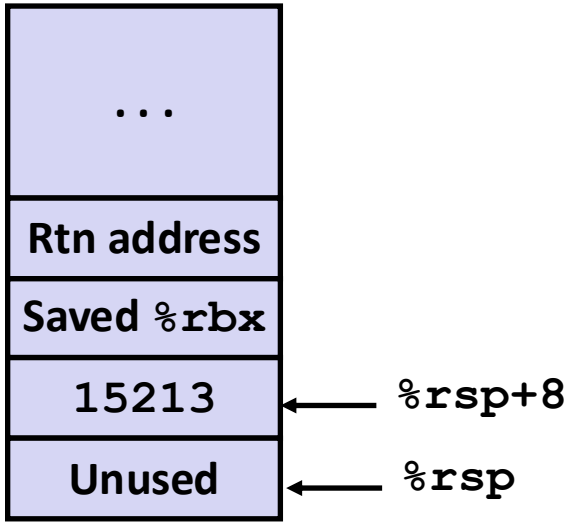
- X saved in **%rbx**.
- A callee saved register.

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



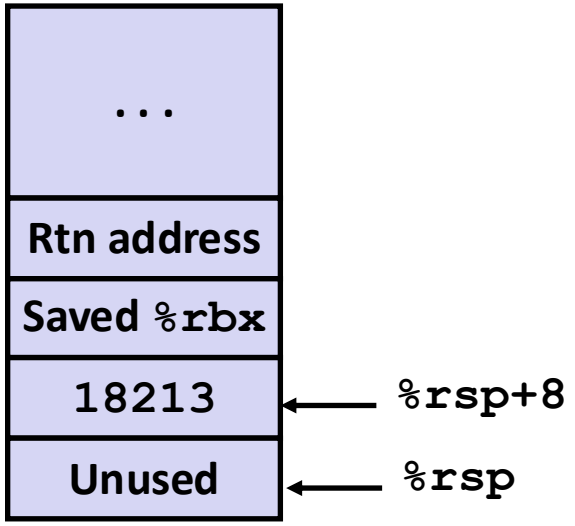
- X saved in **%rbx**.
- A callee saved register.

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



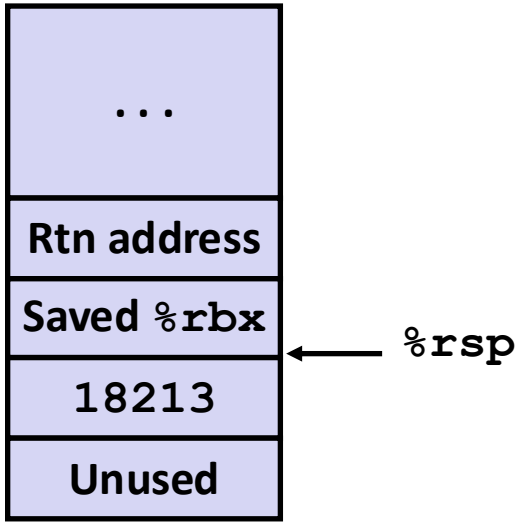
- X Is safe in **%rbx**
- Return result in **%rax**

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



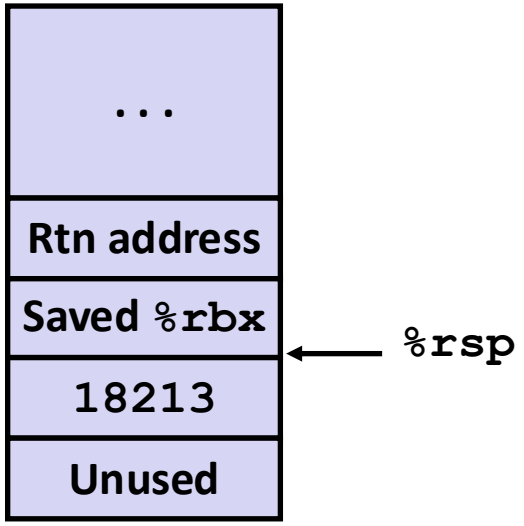
- Return result in **%rax**

Callee-Saved Example #8

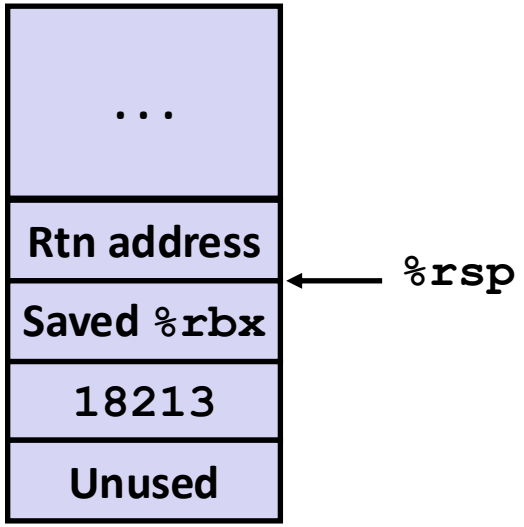
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

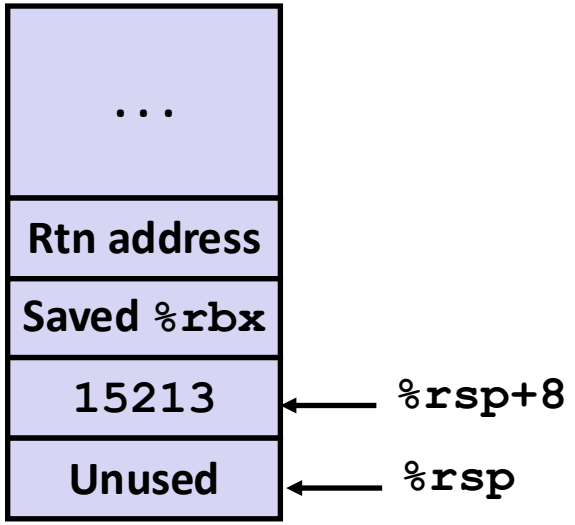


Callee-Saved Example #2

```
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