



Machine-Level Programming III: Procedures

18-213/18-613: Introduction to Computer Systems
6th Lecture, Feb 1, 2024

If you're struggling with assembly

■ Chapter 3 of the textbook is your friend

- More detailed explanations of everything in these lectures
- Work through the practice problems
- Ask for help with the practice problems
- Today's lecture will take us to the end of 3.7

■ Lots of tips, tricks, and guides have been posted on Piazza

■ Talk it out with a friend

- “Help me understand what `jmp *.L32(, %rdx, 8)` does in general” is an OK discussion to have; not an AIV
- “Help me understand what `jmp *.L32(, %rdx, 8)` does *in phase 8 of this binary bomb*”, on the other hand, *is* an AIV
- If you haven't got a friend handy, try a rubber duck. Really!

Recap: Finding Jump Table in Binary

```

00000000004005e0 <switch_eg>:
. . .
4005e9:      ff 24 fd f0 07 40 00      jmpq   *0x4007f0(,%rdi,8)
. . .

```

```

% gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:      0x0000000000400614      0x00000000004005f0
0x400800:      0x00000000004005f8      0x0000000000400602
0x400810:      0x0000000000400614      0x000000000040060b
0x400820:      0x000000000040060b      0x2c646c25203d2078
(gdb)

```

Switch Statements on the Shark Machines

- **Nuance: It is desirable for addresses to be relative rather than absolute.**
 - Since offsets can be smaller than whole addresses, the code takes up less memory
 - Relative vs absolute addresses also make it easier to link code by making it “position independent”. We’ll talk more about that later.
 - **To this end, x86-64 has an addressing mode which provides hardware support for managing addresses relative to the %rip.**
- **Nuance: Loads and stores with 64-bit displacement are available only via %eax.**
 - Why? Its all wires! (See Prof. Nace in 18-240 for details!)

Switch Statements on the Shark Machines

- To make things work nicely with the constraints on the prior page:
 - Rather than keeping absolute addresses, the jump table address is kept relative to the `%rip`.
 - The address of the target code is placed into the `%rax`, because only that register can contain a 64-bit target address

Switch Statements on the Shark Machines

- The code that does the jumping looks like this:

```
# Handle the cases too small or too large for the switch
# Negative cases look large to ja and offsets can be used
# to shift the smallest case to 0.
0x00000000004017e9 <+41>:    cmp     $0xc,%esi
0x00000000004017ec <+44>:    ja     0x401818 <foo+88>

# Get a pointer to the jump table
# %rip points to next addr, so %rdx becomes 0x498018 (jump table)
0x00000000004017ee <+46>:    lea   0x96823(%rip),%rdx

# Get offset from %rsi-th index of jump table
0x00000000004017f5 <+53>:    movslq (%rdx,%rsi,4),%rax

# Add that offset to the address of the jump table
0x00000000004017f9 <+57>:    add   %rdx,%rax

# Jump to that address, ultimately an offset from %rip
0x00000000004017fc <+60>:    jmp   *%rax
```

Switch Statements on the Shark Machines

- The jump table contains offsets relative to the start of the jump table

```
(gdb) x/20dw 0x498008
```

```
0x498018:      -616376 -616448 -616448 -616448
0x498028:      -616472 -616448 -616461 -616448
0x498038:      -616392 -616448 -616416 -616448
0x498048:      -616416 0          0          0
```

```
(gdb) x/20xw 0x498008
```

```
0x498008:      0x74636e75      0x206e6f69      0x216f6f66      0x00000000
0x498018:      0xffff69848     0xffff69800     0xffff69800     0xffff69800
0x498028:      0xffff697e8     0xffff69800     0xffff697f3     0xffff69800
0x498038:      0xffff69838     0xffff69800     0xffff69820     0xffff69800
0x498048:      0xffff69820     0x000000000     0x000000000     0x000000000
```


Switch Statements on the Shark Machines

- Ultimately, the target address is equal to the `%rip` + the offset to the start of jump table, plus an offset (possibly negative) from the start of the jump table to the target code.
 - This target address is computed, placed into the `%rax`, and jumped to

```
# Get a pointer to the jump table
# %rip points to next addr, so %rdx becomes 0x498018 (jump table)
0x00000000004017ee <+46>:    lea    0x96823(%rip),%rdx

# Get offset from %rsi-th index of jump table
0x00000000004017f5 <+53>:    movslq (%rdx,%rsi,4),%rax

# Add that offset to the address of the jump table
0x00000000004017f9 <+57>:    add    %rdx,%rax

# Jump to that address, ultimately an offset from %rip
0x00000000004017fc <+60>:    jmp    *%rax
```

Today

■ Procedures

- Mechanisms CSAPP 3.7 preamble
- Stack Structure CSAPP 3.7.1
- Calling Conventions
 - Passing control CSAPP 3.7.2
 - Passing data CSAPP 3.7.3
 - Managing local data CSAPP 3.7.4 – 3.7.5
- Illustration of Recursion CSAPP 3.7.6

Mechanisms in Procedures

What's needed?

■ 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

```
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

```
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

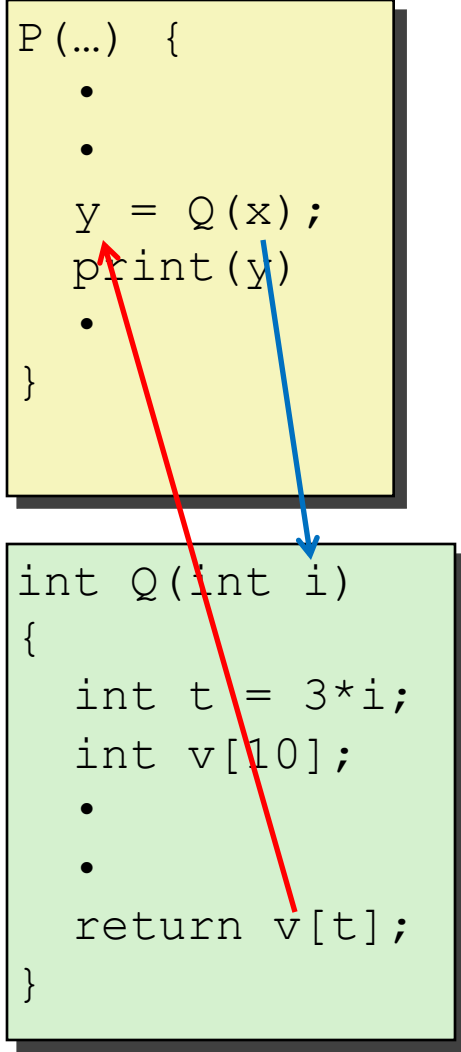
■ Passing data

- Procedure arguments
- Return value

■ Memory management

- Allocate during procedure execution
- Deallocate upon return

```
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

```
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

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

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


Mechanisms in Procedures

```
P (...) {
```

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**

```
int v[10];  
·  
·  
return v[t];  
}
```

Today

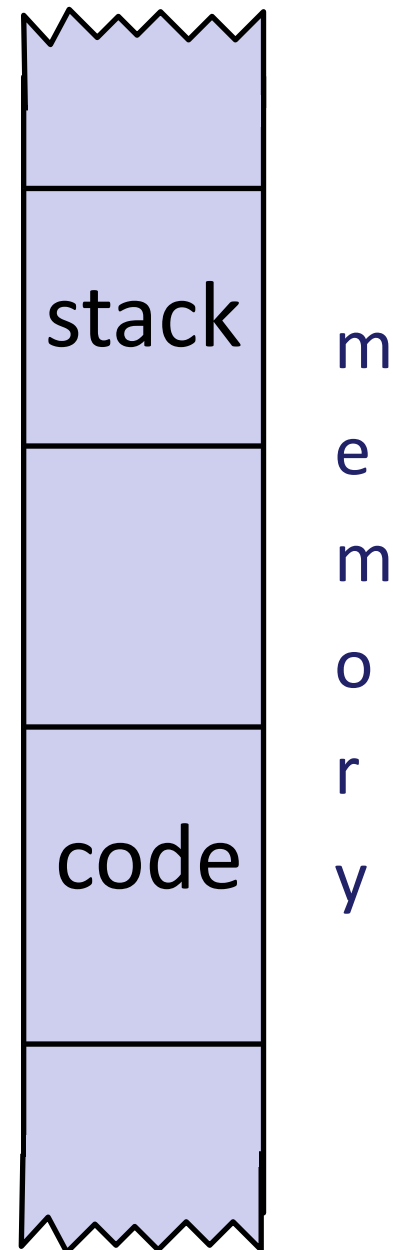
■ Procedures

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

x86-64 Stack

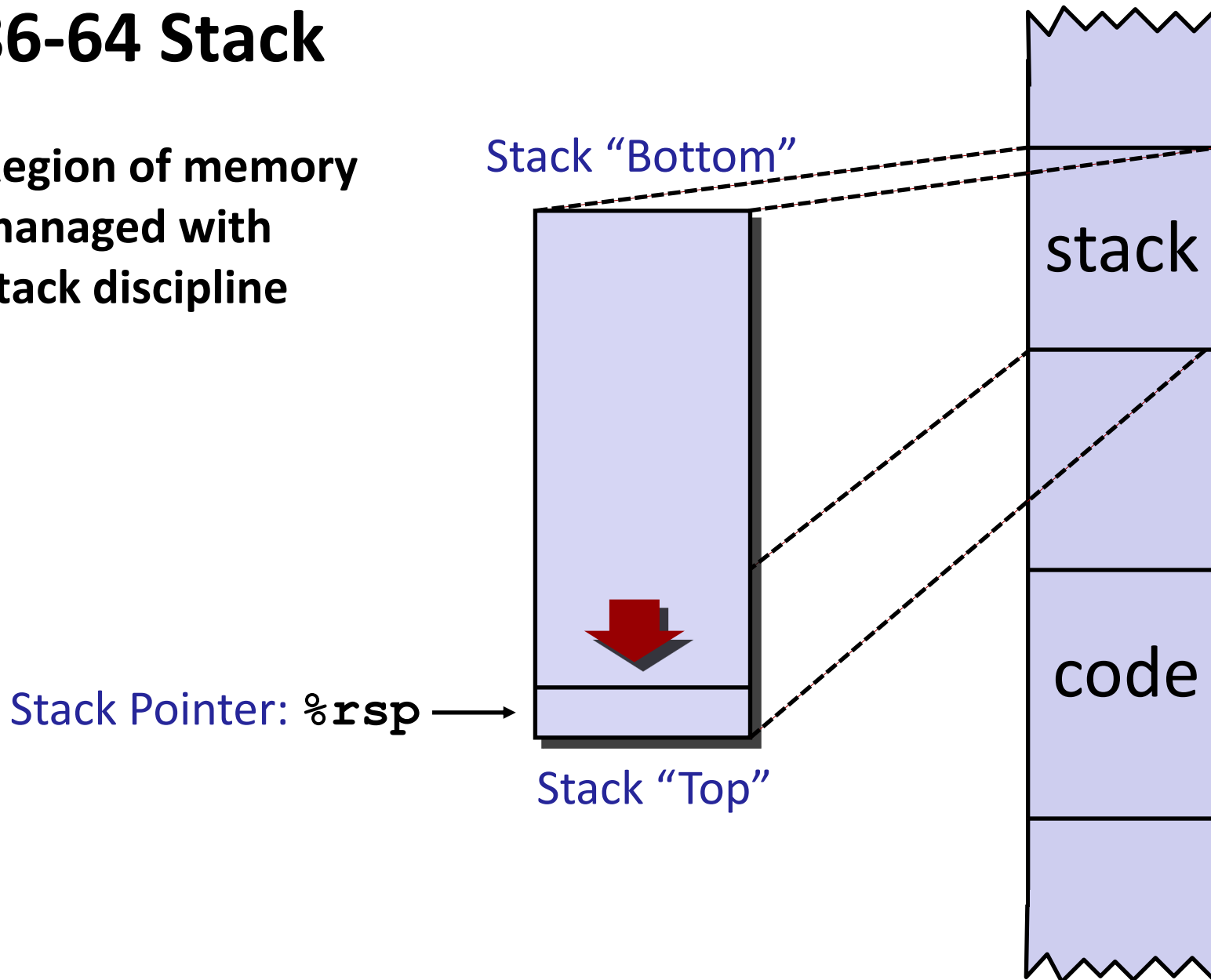
■ Region of memory managed with stack discipline

- Memory viewed as array of bytes.
- Different regions have different purposes.
- (Like ABI, a policy decision)



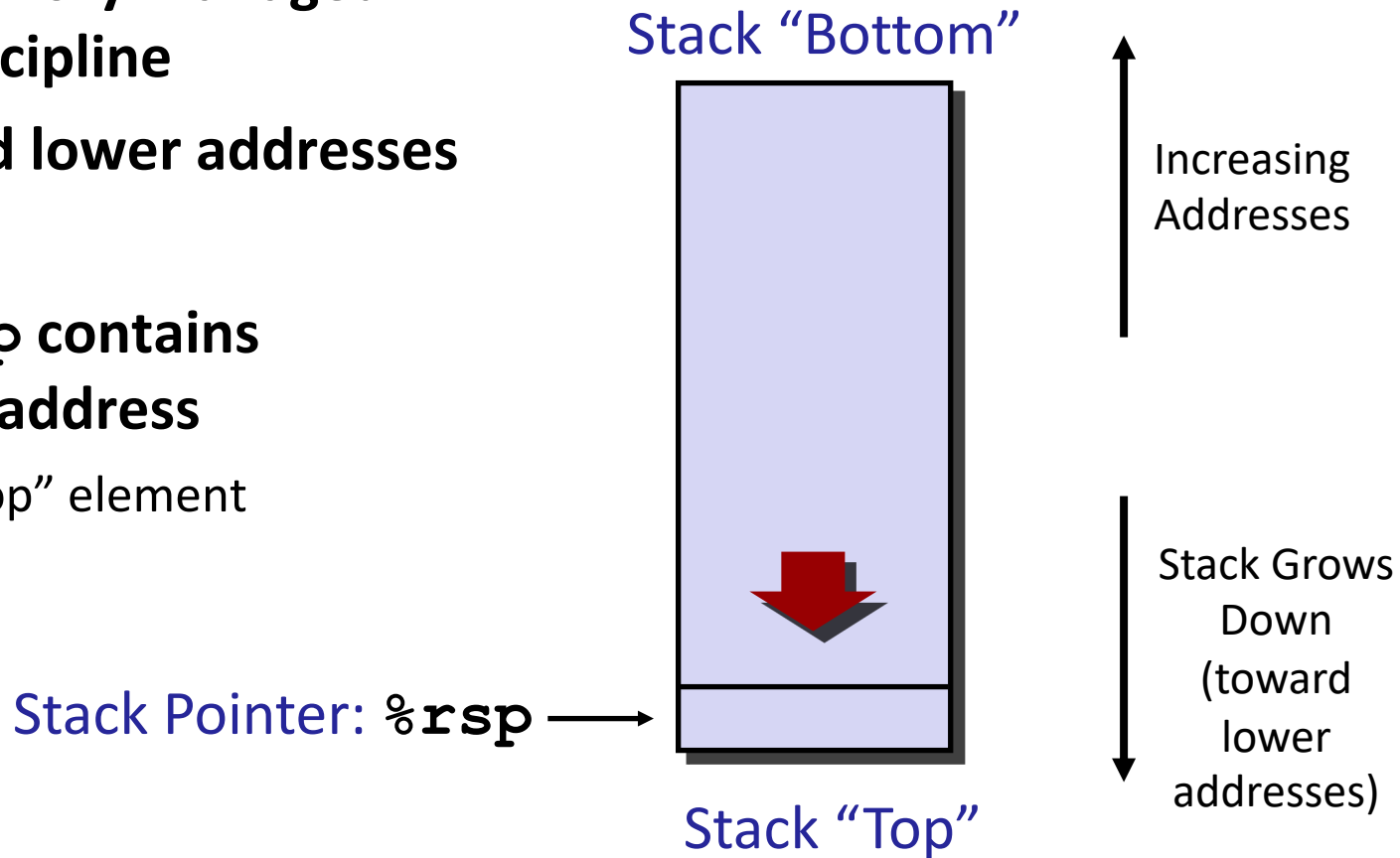
x86-64 Stack

- Region of memory managed with stack discipline



x86-64 Stack

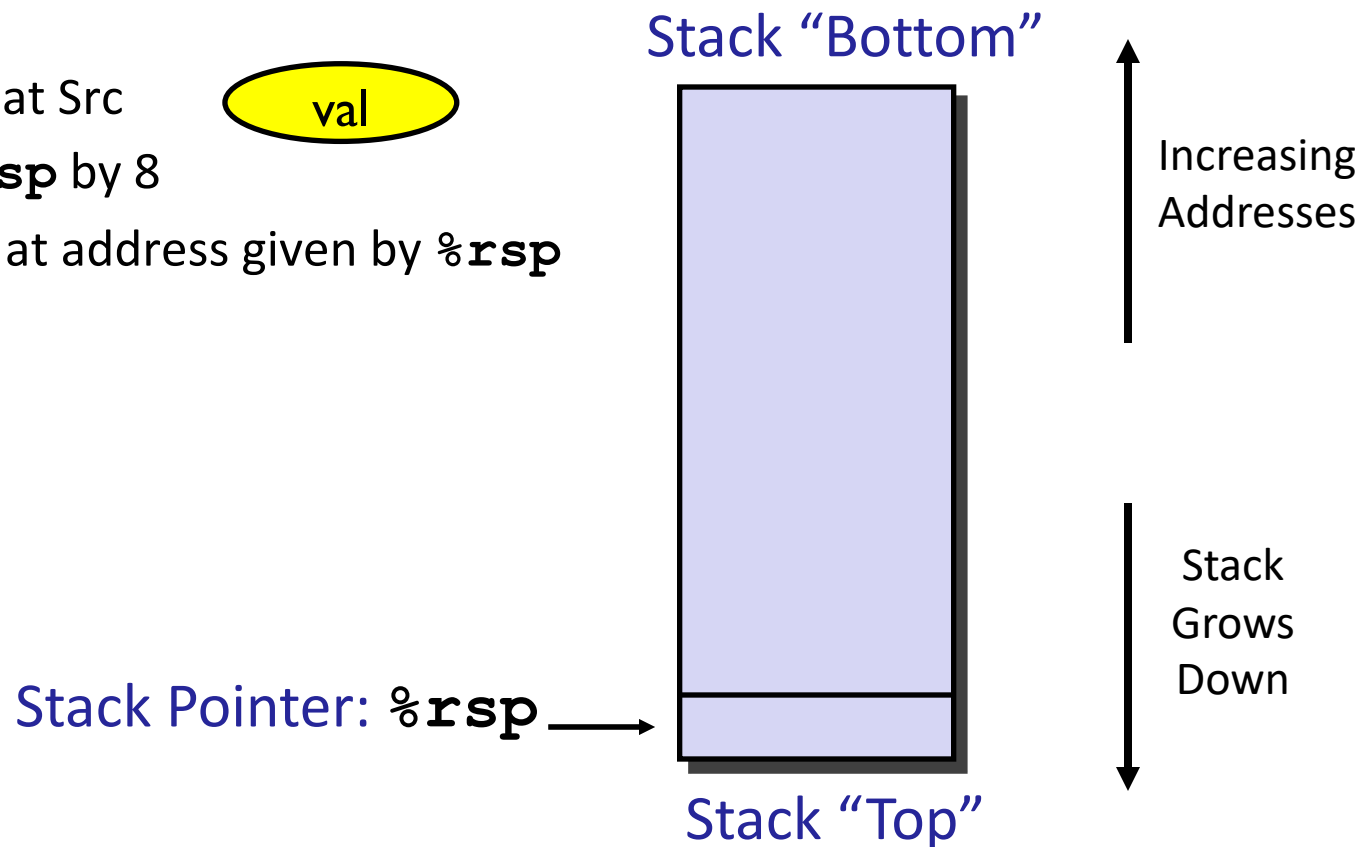
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
 - address of “top” element



x86-64 Stack: Push

■ `pushq Src`

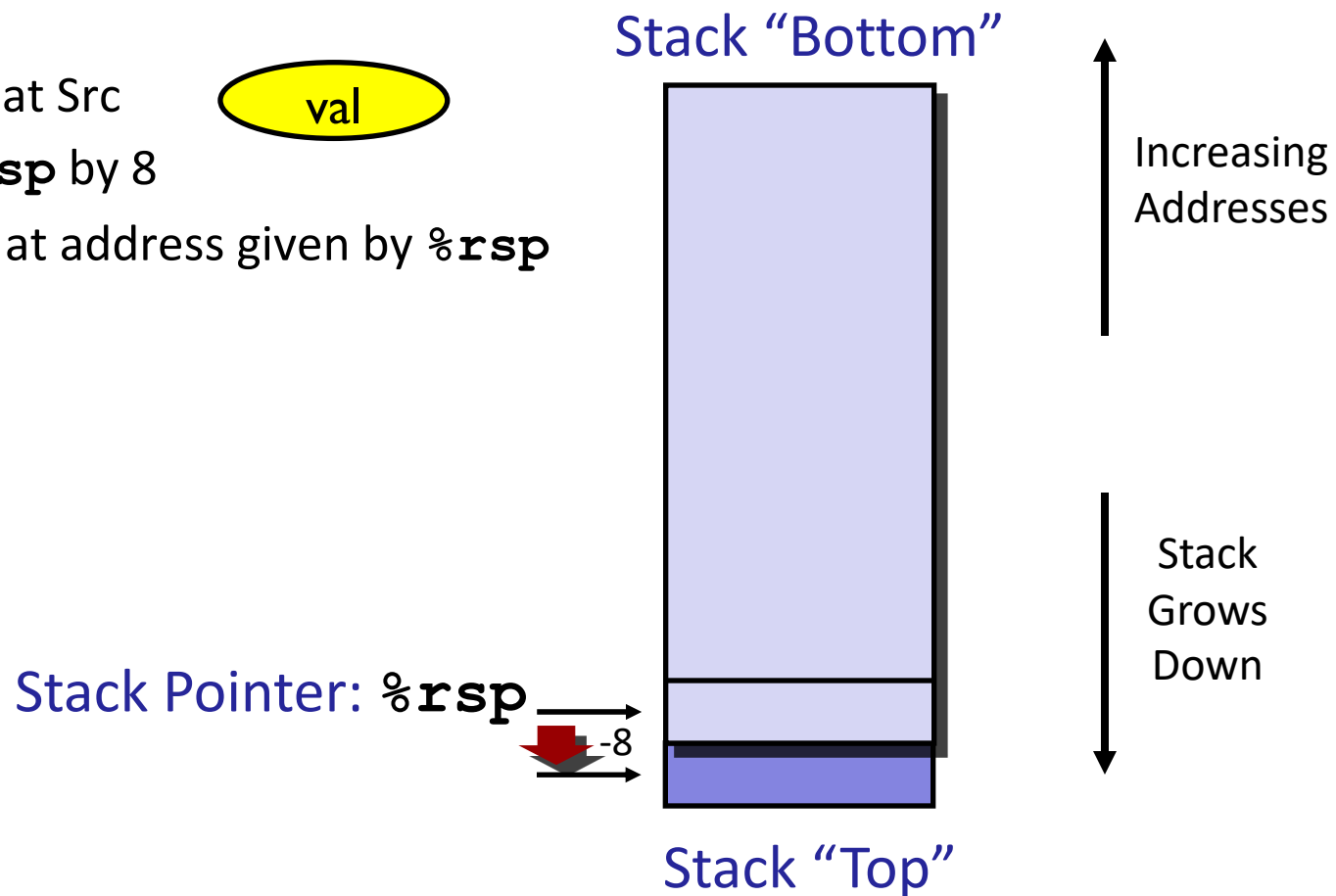
- Fetch operand at Src
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



x86-64 Stack: Push

■ `pushq Src`

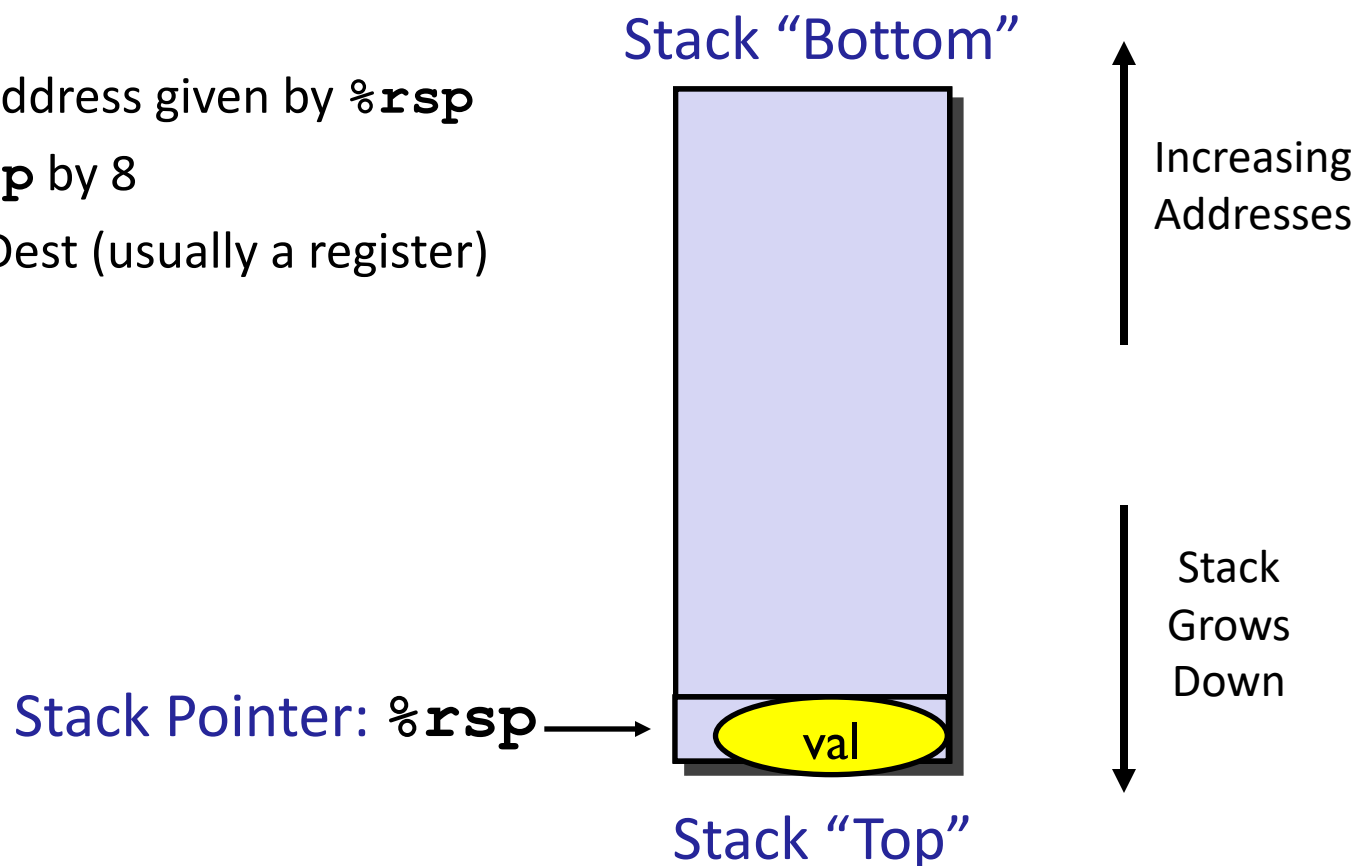
- Fetch operand at Src
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



x86-64 Stack: Pop

■ `popq Dest`

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



x86-64 Stack: Pop

■ `popq Dest`

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

val

Stack Pointer: `%rsp`



Stack "Bottom"



Stack "Top"

Increasing
Addresses

Stack
Grows
Down

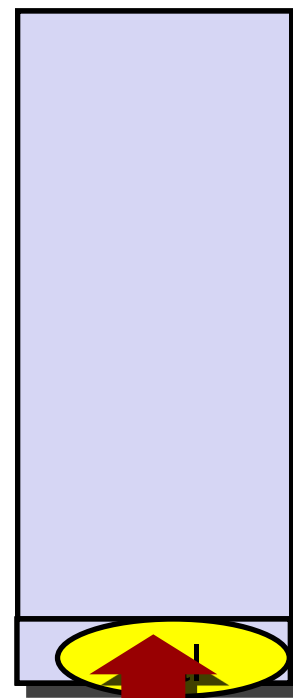
x86-64 Stack: Pop

■ `popq Dest`

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

Stack Pointer: `%rsp` →

Stack "Bottom"



Increasing
Addresses

Stack
Grows
Down

Stack "Top"

(The memory doesn't change,
only the value of `%rsp`)

Today

■ Procedures

- Mechanisms
- Stack Structure
- Calling Conventions
 - **Passing control**
 - Passing data
 - Managing local data
- Illustration of 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: callq  400550 <mult2>  # mult2(x,y)
400549: mov     %rax,(%rbx)    # Save at dest
40054c: pop     %rbx           # Restore %rbx
40054d: retq                               # Return
```

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

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

Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
 - Push return address on stack
 - Jump to label
- **Return address:**
 - Address of the next instruction right after call
 - Example from disassembly
- **Procedure return:** `ret`
 - Pop address from stack
 - Jump to address

Control Flow Example #1

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

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

0x130

0x128

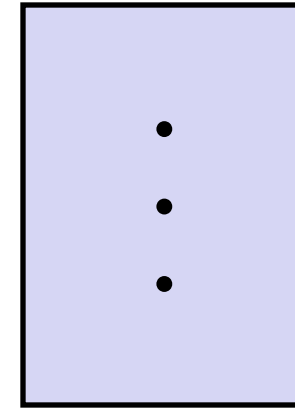
0x120

%rsp

%rip

0x120

0x400544



%rsp

%rip

0x120

0x400544

Control Flow Example #2

```
0000000000400540 <multstore>:
```

```
•
•
400544: callq 400550 <mult2>
400549: mov   %rax, (%rbx) ←
•
•
```

```
0000000000400550 <mult2>:
```

```
400550: mov   %rdi, %rax ←
•
•
400557: retq
```

0x130

0x128

0x120

0x118

%rsp

%rip

0x400549

0x118

0x400550

•

•

•



Control Flow Example #3

```
0000000000400540 <multstore>:
```

```
•
•
400544: callq 400550 <mult2>
400549: mov  %rax, (%rbx) ←
•
•
```

```
0000000000400550 <mult2>:
```

```
400550: mov  %rdi,%rax
•
•
400557: retq ←
```

0x130

0x128

0x120

0x118

0x400549

%rsp

0x118

%rip

0x400557



Control Flow Example #4

```

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

```

```

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

```

0x130

0x128

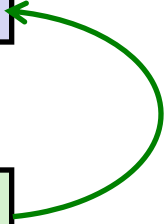
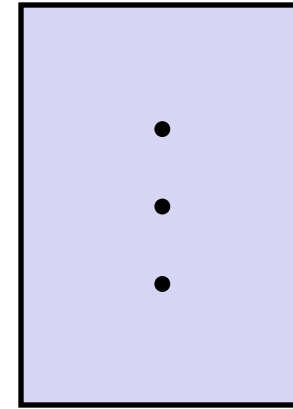
0x120

%rsp

0x120

%rip

0x400549



Today

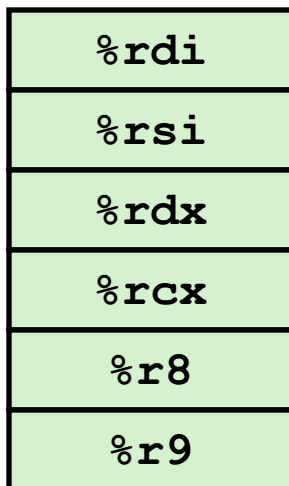
■ Procedures

- Mechanisms
- Stack Structure
- Calling Conventions
 - Passing control
 - **Passing data**
 - Managing local data
- Illustrations of Recursion & Pointers

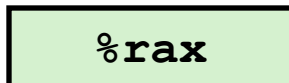
Procedure Data Flow

Registers

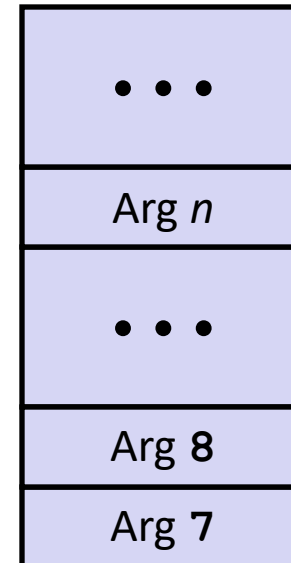
- First 6 integer arguments



- Return value



Stack



- Only allocate stack space when needed

Data Flow Examples

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

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

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

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

Today

■ Procedures

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

Stack-Based Languages

■ Languages that support recursion

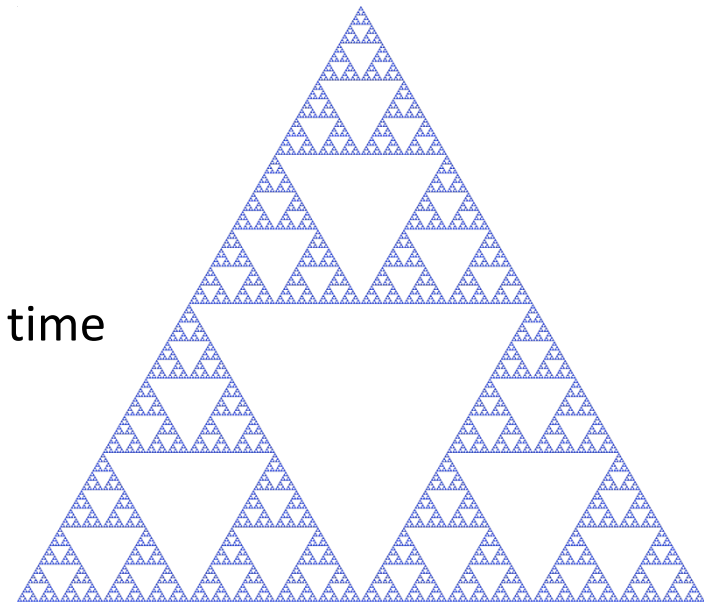
- e.g., C, Pascal, Java
- Code must be “Reentrant”
 - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return address

■ 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



Call Chain Example

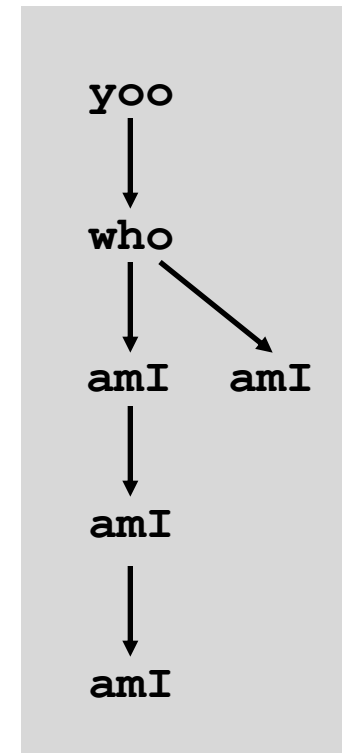
```
yoo (...)
{
  .
  .
  who ();
  .
  .
}
```

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

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

Procedure **amI ()** is recursive

Example
Call Chain



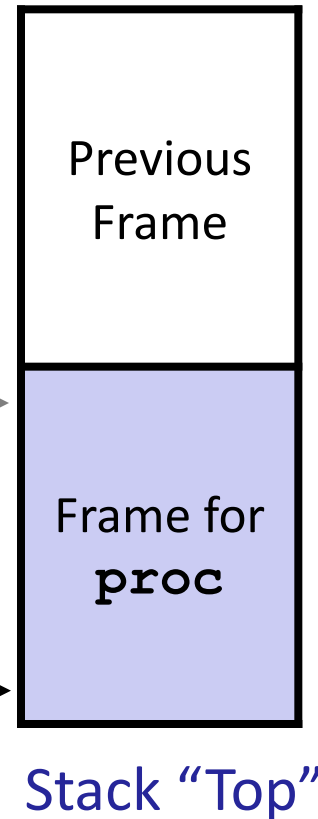
Stack Frames

■ Contents

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

Frame Pointer: `%rbp`
(Optional)


Stack Pointer: `%rsp`



■ Management

- Space allocated when enter procedure
 - “Set-up” code
 - Includes push by `call` instruction
- Deallocated when return
 - “Tear-down” code
 - Includes pop by `ret` instruction

Example

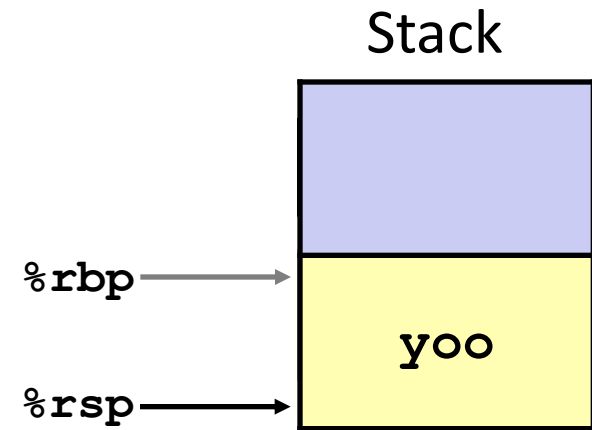


```

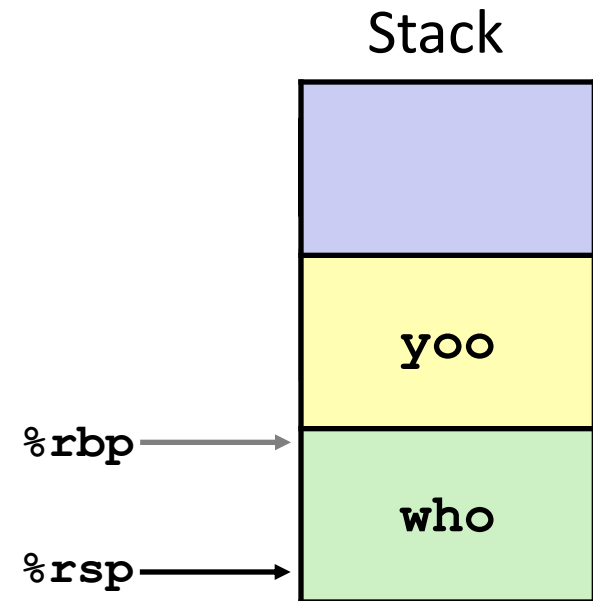
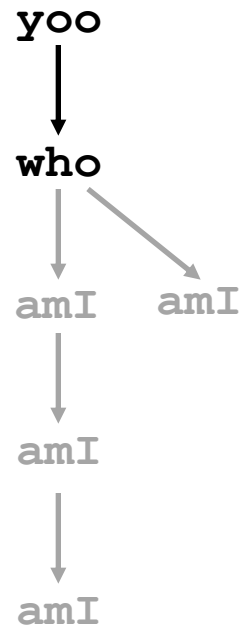
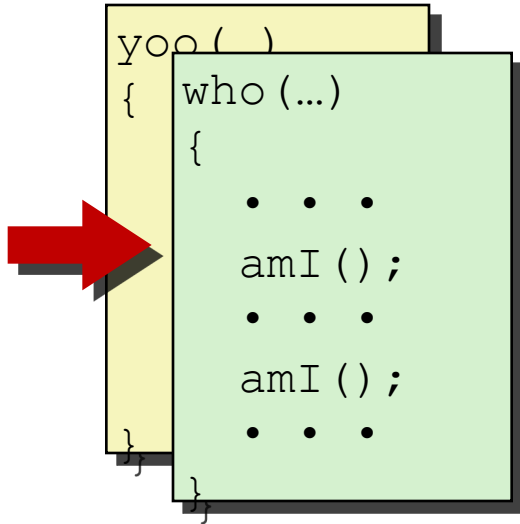
yoo (...)
{
    .
    .
    who ();
    .
    .
}
  
```

```

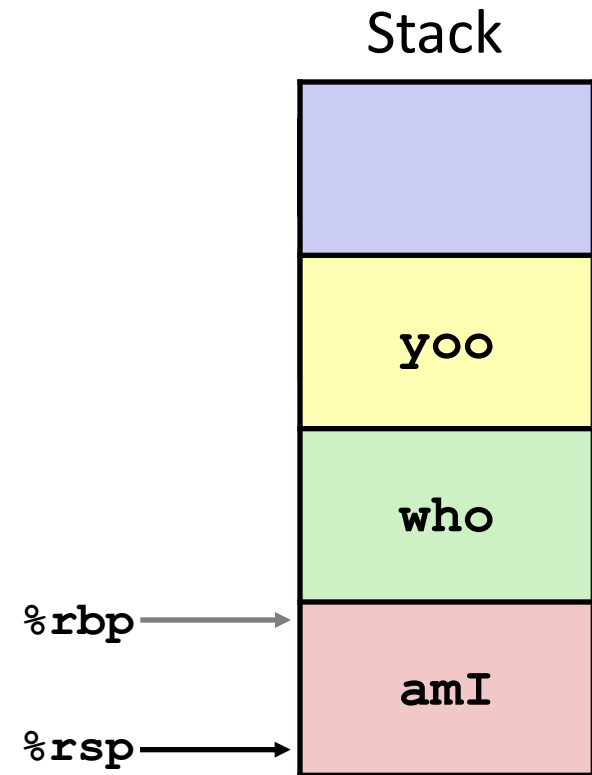
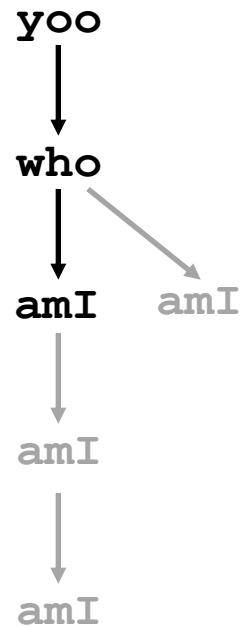
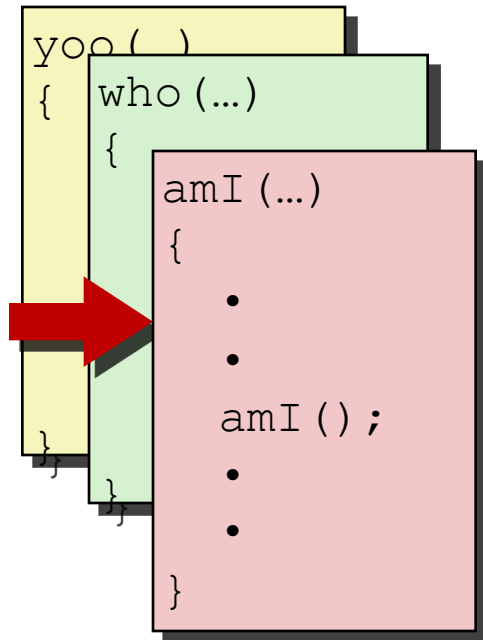
yoo
  |
  v
who
 /  \
amI  amI
  |
  v
amI
  |
  v
amI
  
```



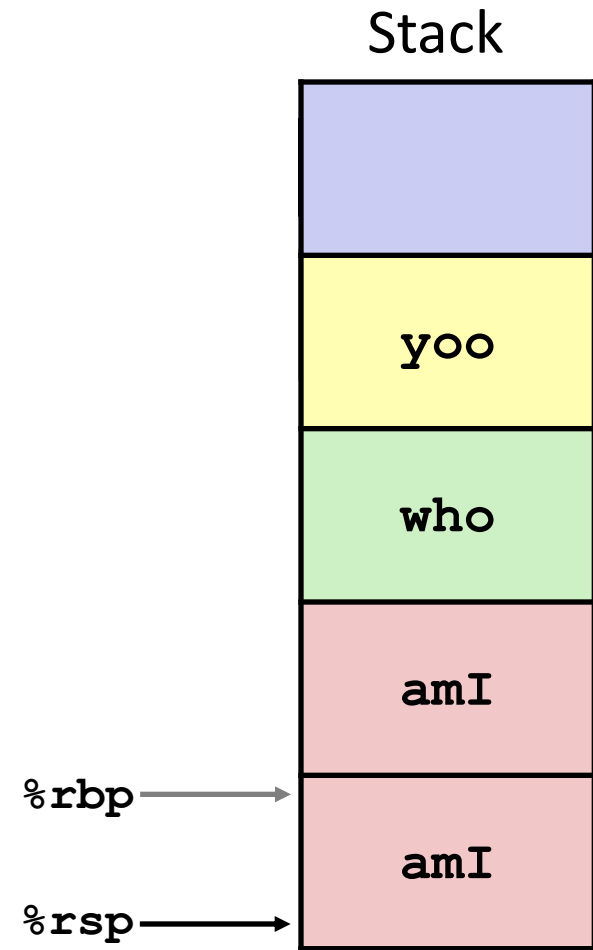
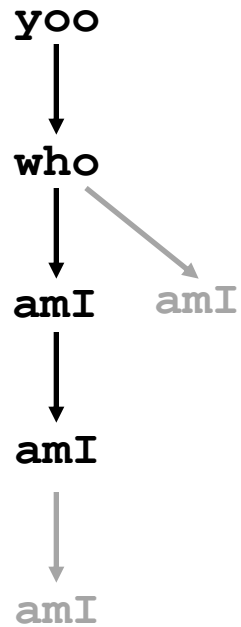
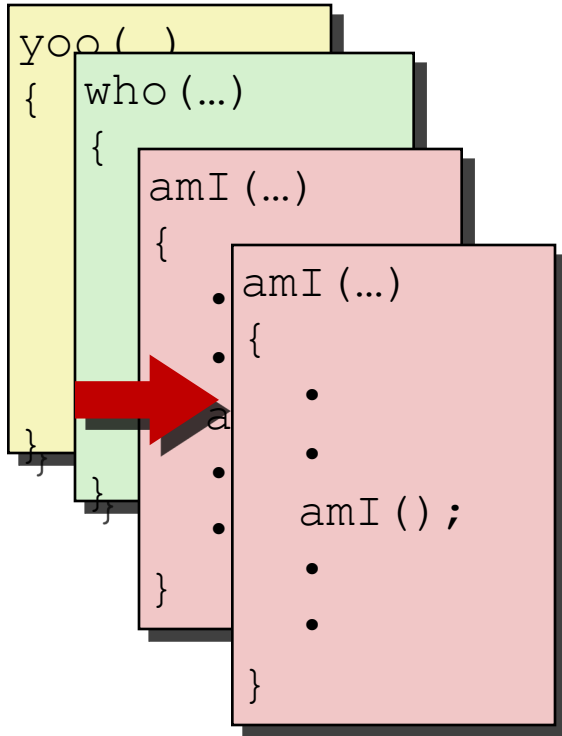
Example



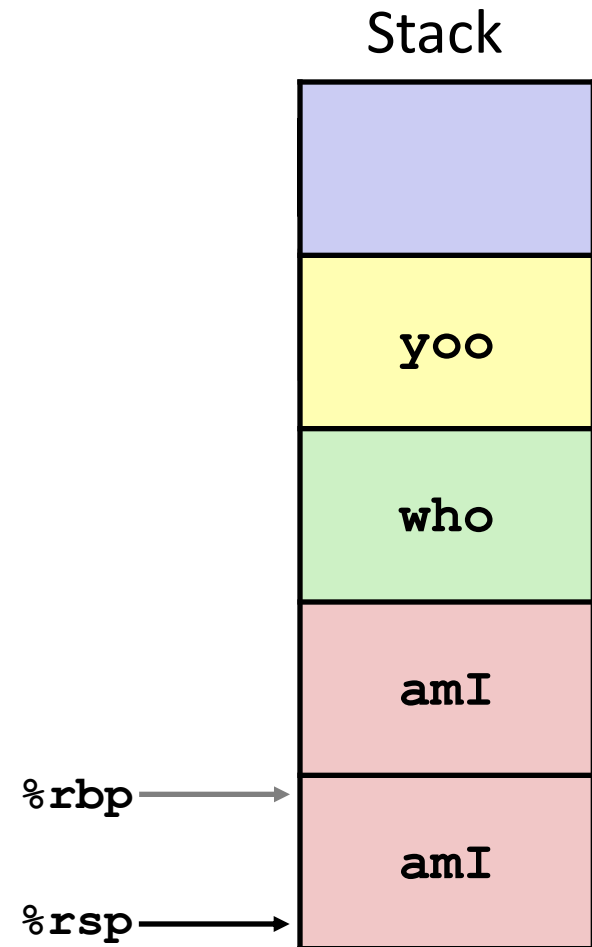
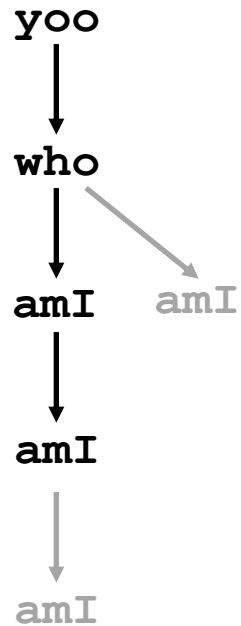
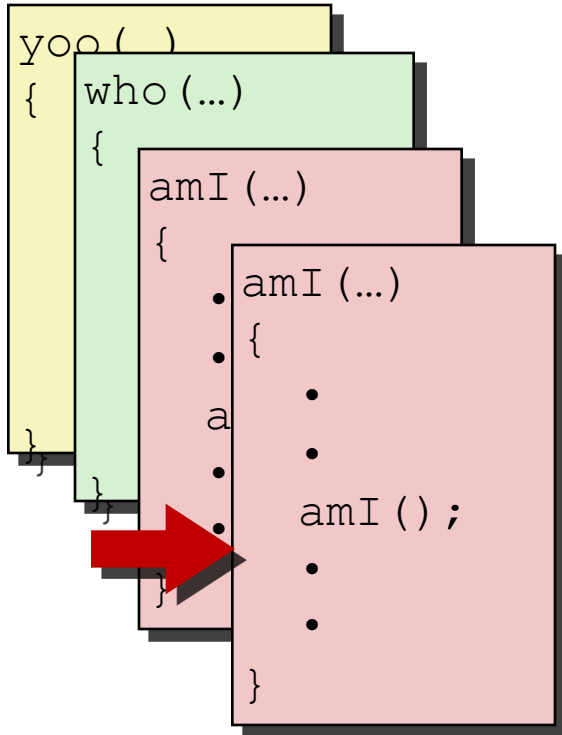
Example



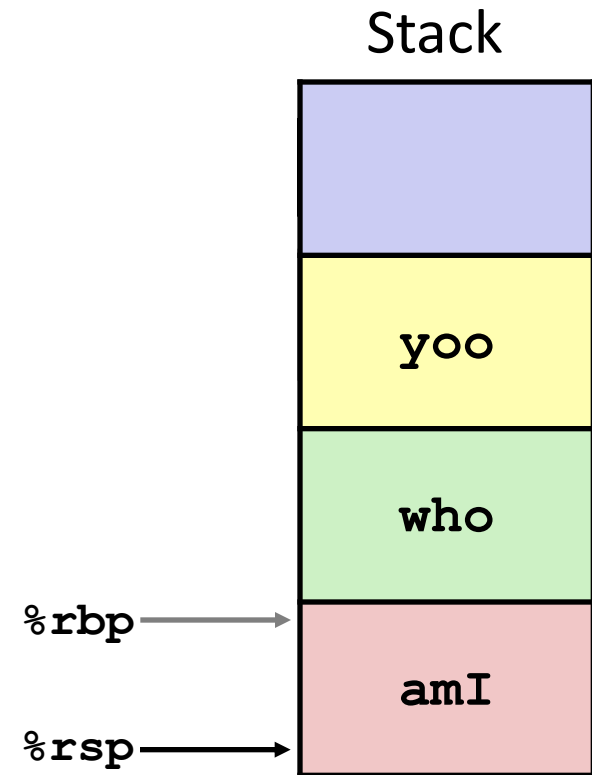
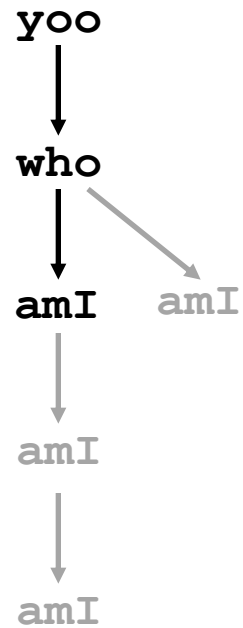
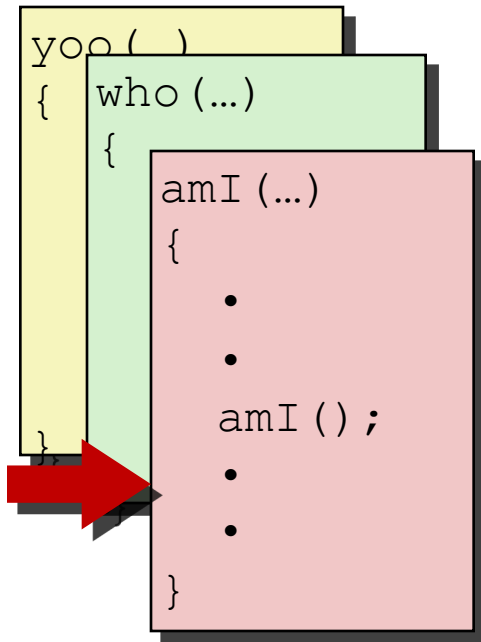
Example



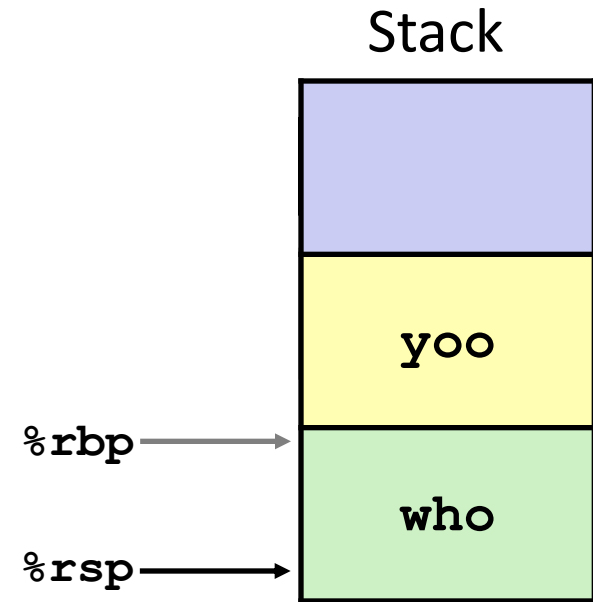
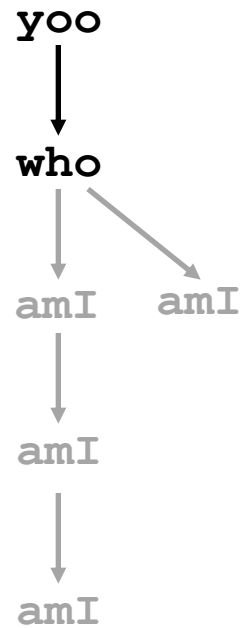
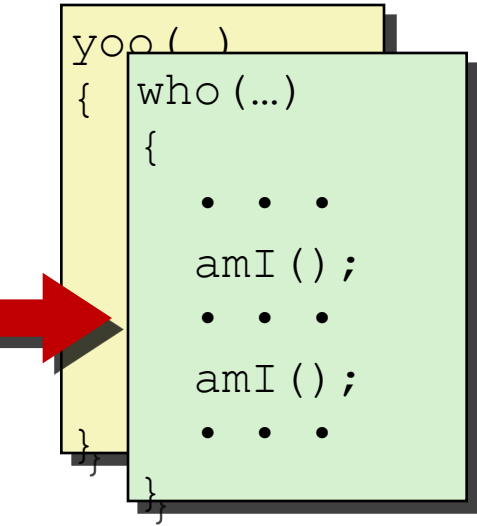
Example



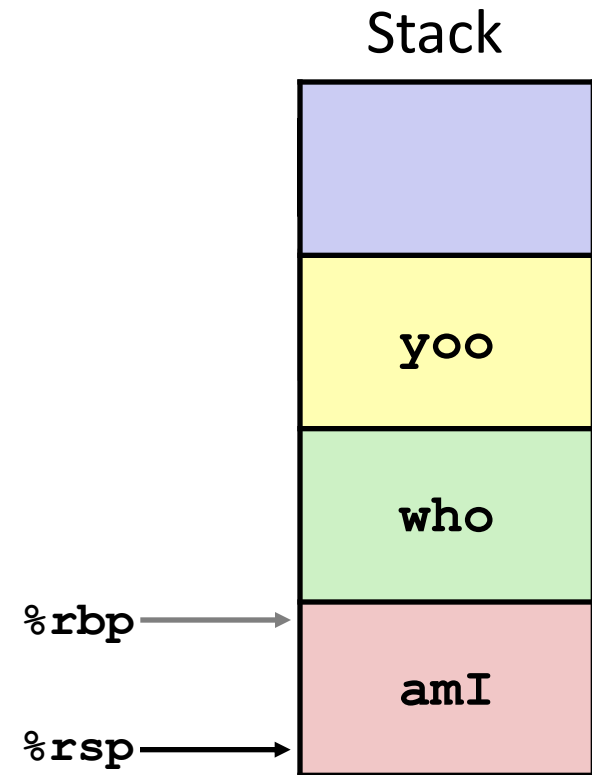
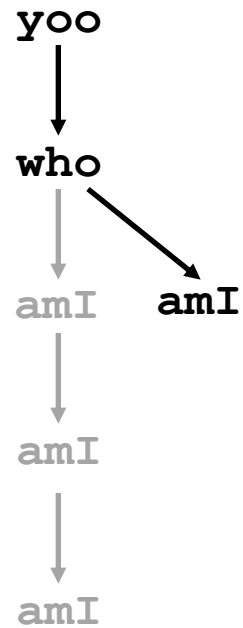
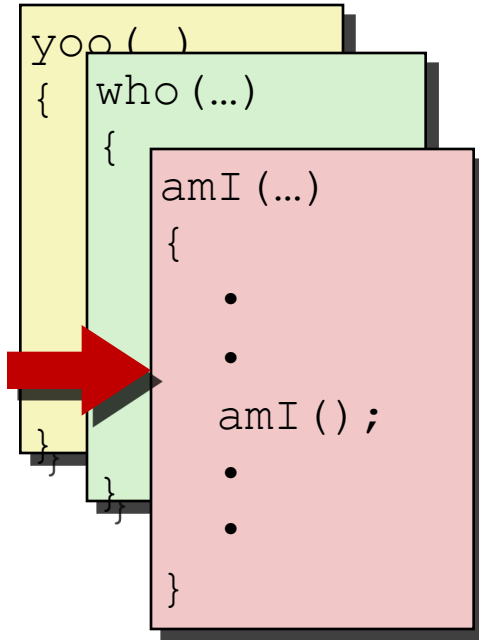
Example



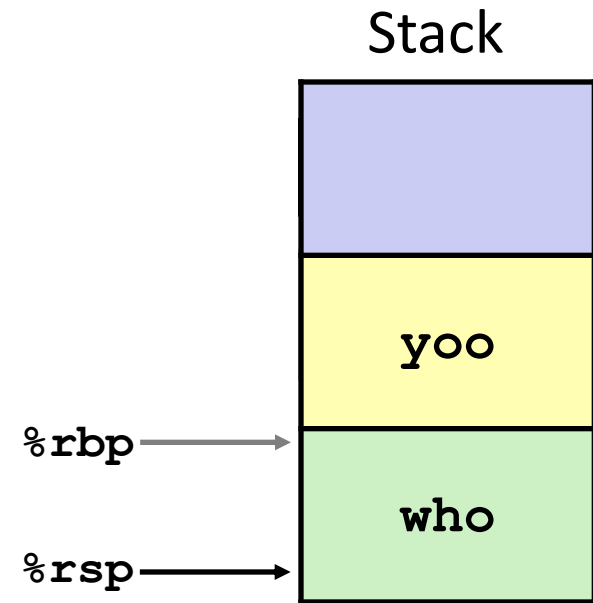
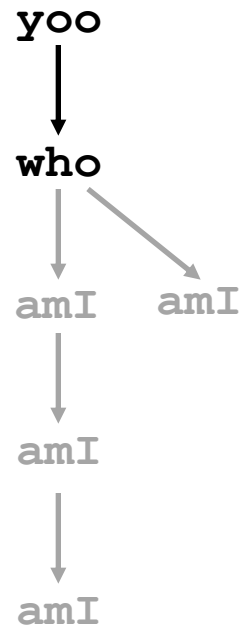
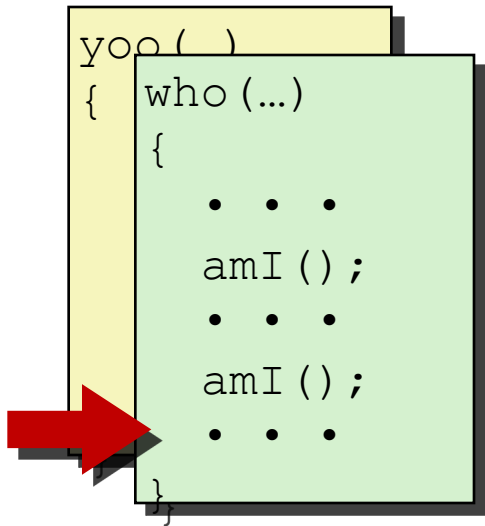
Example



Example



Example




Example

```

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

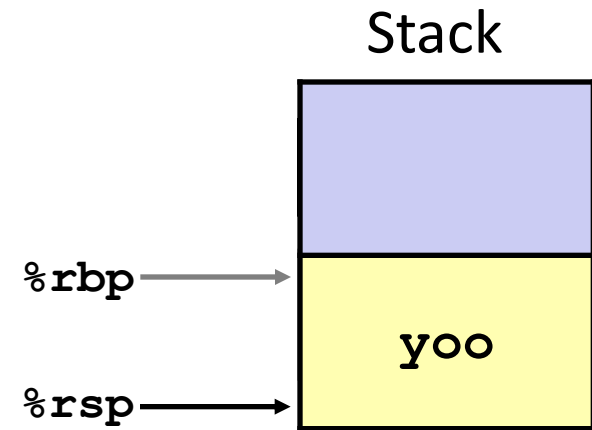
```



```

yoo
  ↓
who
  ↓  ↘
amI  amI
  ↓
amI
  ↓
amI

```



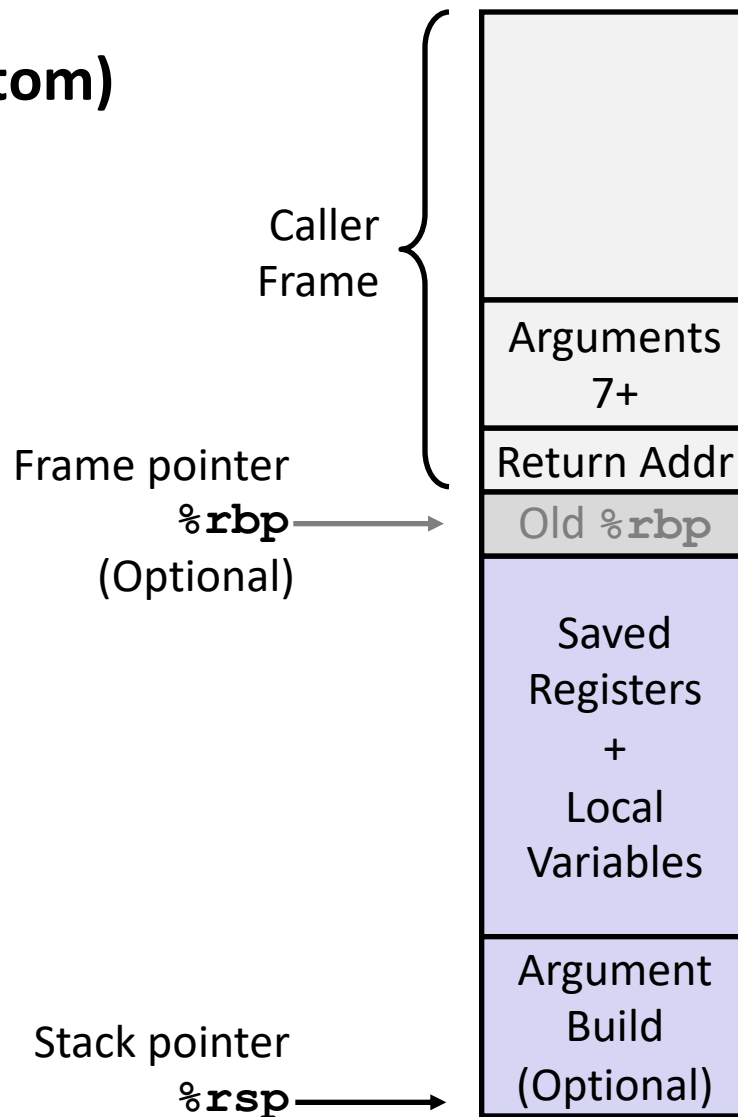
x86-64/Linux Stack Frame

■ 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



Example: `incr`

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

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

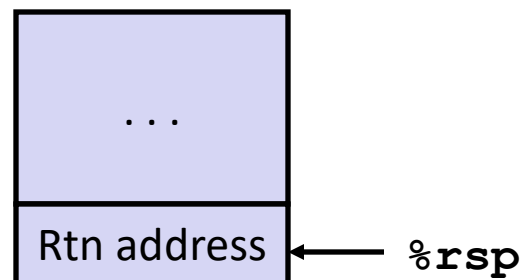
Register	Use(s)
<code>%rdi</code>	Argument <code>p</code>
<code>%rsi</code>	Argument <code>val</code> , <code>y</code>
<code>%rax</code>	<code>x</code> , Return value

Example: Calling `incr` #1

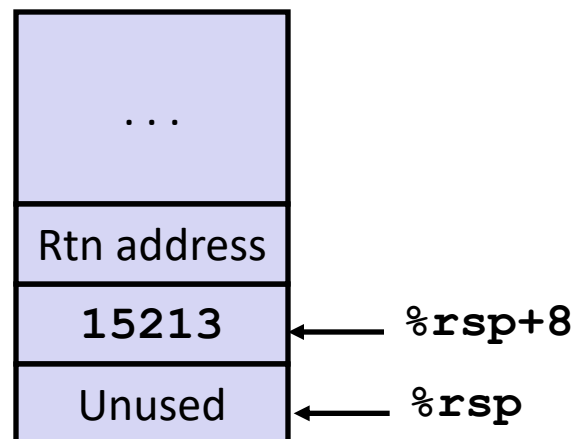
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

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

Initial Stack Structure



Resulting Stack Structure

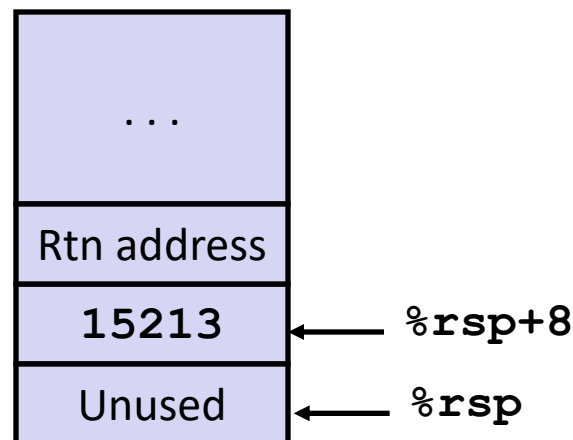


Example: Calling `incr` #2

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

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

Stack Structure

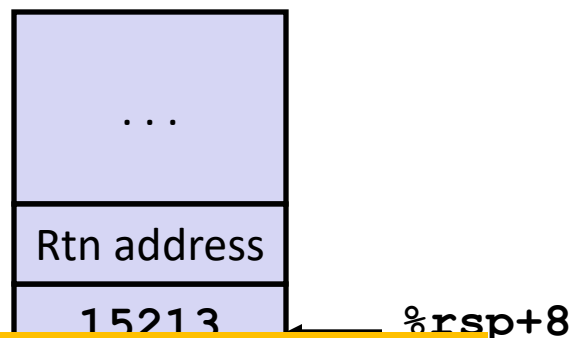


Register	Use(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	3000

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`

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

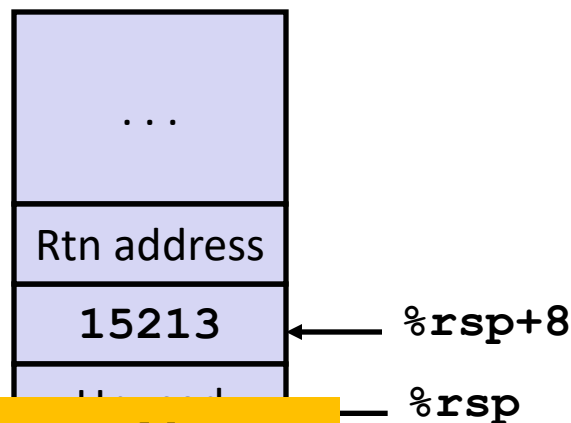
```
call_incr:
    subq    $8, %rsp
    movl    $3000, %esi
    leaq   8(%rsp), %rdi
    call   incr
    addq   8(%rsp), %rax
    addq   $16, %rsp
    ret
```

<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	<code>3000</code>

Example: Calling `incr` #2

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

Stack Structure



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

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

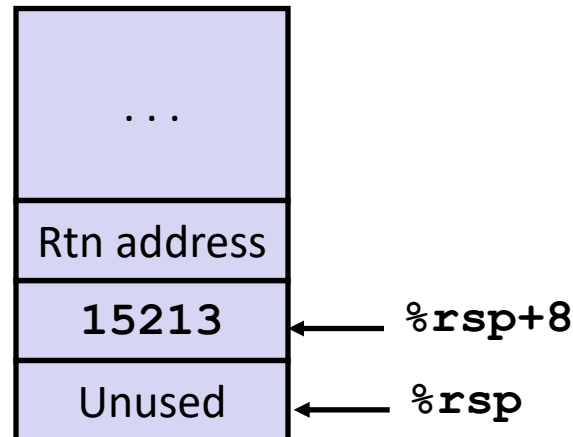
```
leaq    8(%rsp), %rdi
call    incr
addq    8(%rsp), %rax
addq    $16, %rsp
ret
```

	case(s)
	v1
%rsi	3000

Example: Calling `incr` #2

Stack Structure

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



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

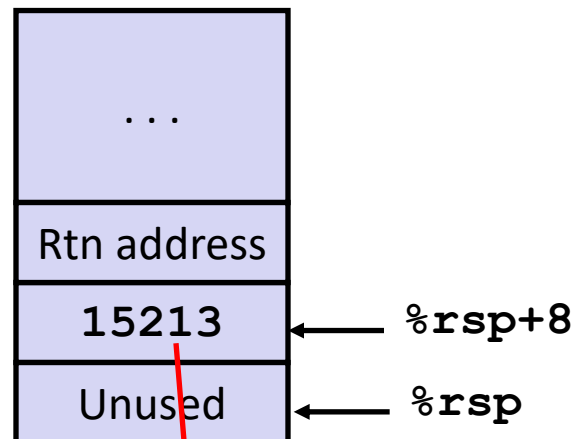
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr #3a`

Stack Structure

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

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



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

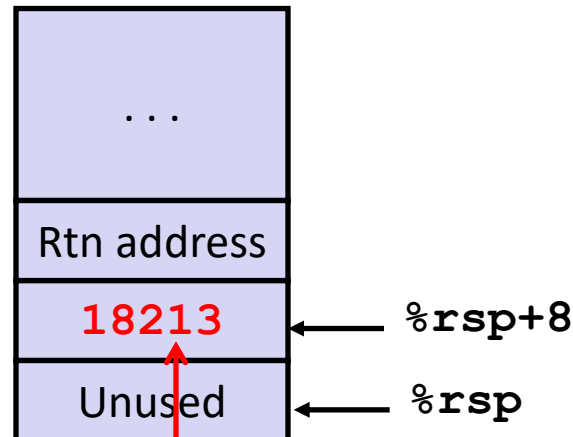
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Example: Calling `incr` #3b

Stack Structure

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

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



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

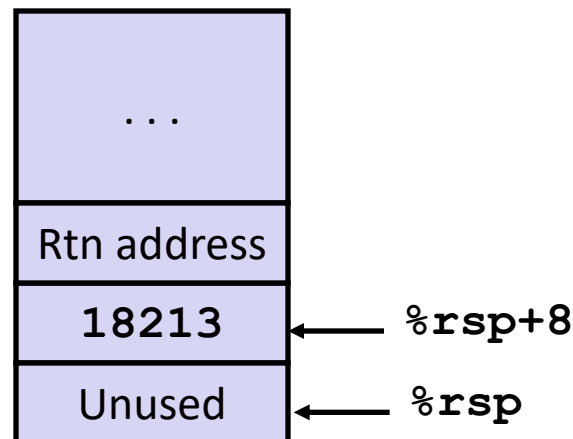
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Example: Calling `incr` #4

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

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

Stack Structure



Register	Use(s)
%rax	Return value, 15213

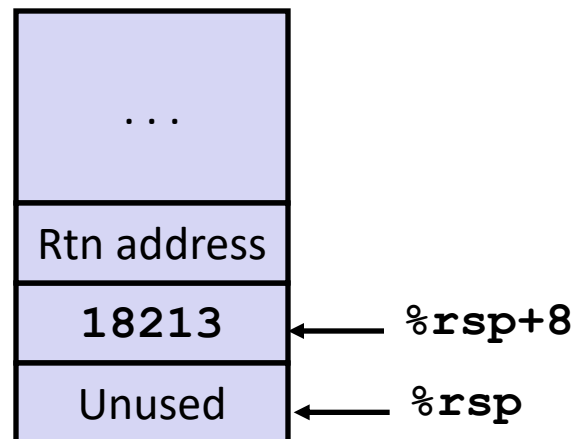
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Example: Calling `incr` #5a

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

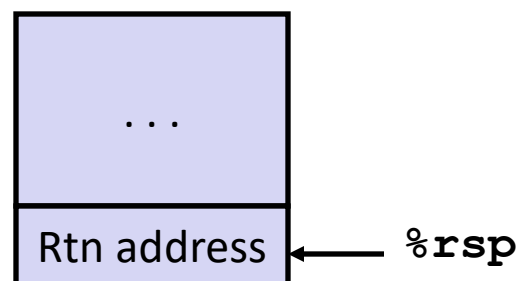
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



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

Updated Stack Structure

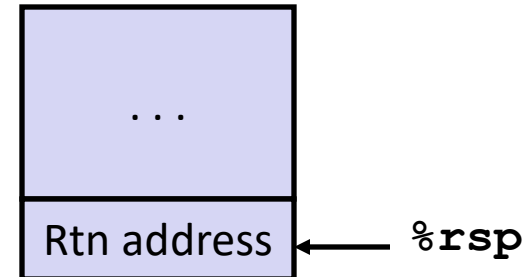


Example: Calling `incr` #5b

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

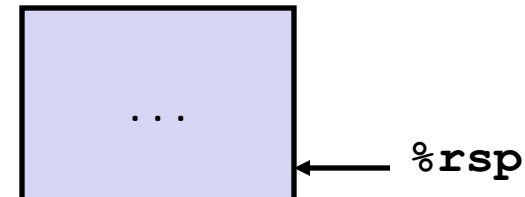
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Updated Stack Structure



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

Final Stack Structure



Register Saving Conventions

- When procedure `yoo` calls `who`:
 - `yoo` is the **caller**
 - `who` is the **callee**
- Can a 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`
- If a callee *clobbers* your register, its value is lost!
 - Need coordination between caller/callee

Register Saving Conventions

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

x86-64 Linux Register Usage #1

■ **%rax**

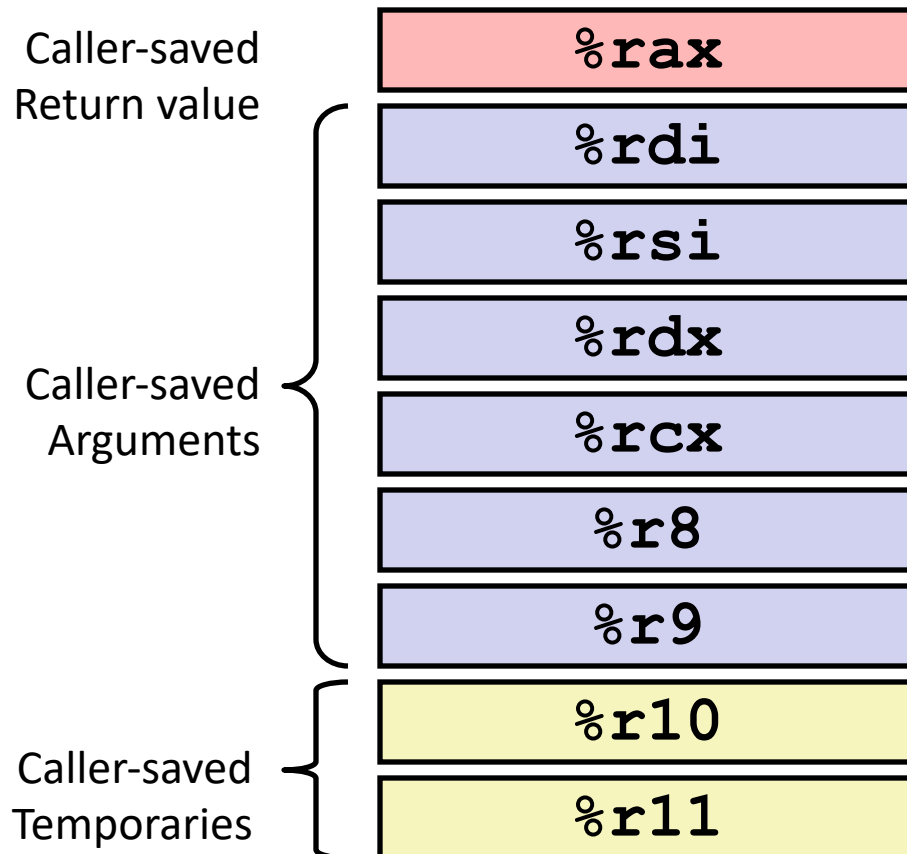
- Return value
- Also caller-saved
- Can be modified by procedure

■ **%rdi, ..., %r9**

- Integer arguments
- Also caller-saved
- Can be modified by procedure

■ **%r10, %r11**

- Caller-saved
- Can be modified by procedure



x86-64 Linux Register Usage #2

■ `%rbx`, `%r12`, `%r13`, `%r14`

- Callee-saved
- Callee must save & restore

■ `%rbp`

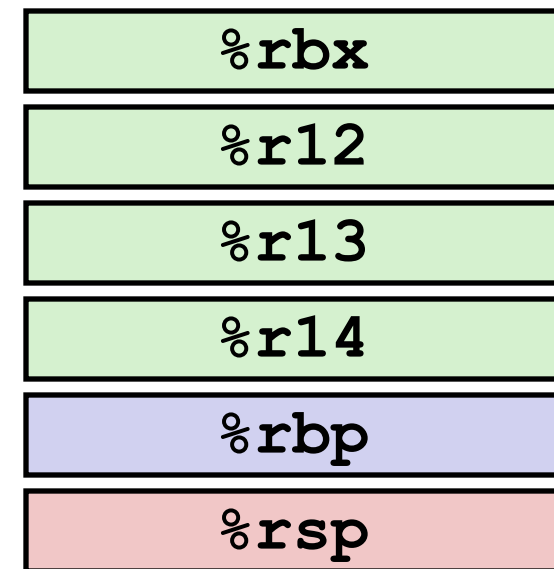
- Callee-saved
- Callee must save & restore
- May be used as frame pointer
- Compiler decides use of `rbp`

■ `%rsp`

- Special form of callee save
- Restored to original value upon exit from procedure

Callee-saved
Temporaries

Special



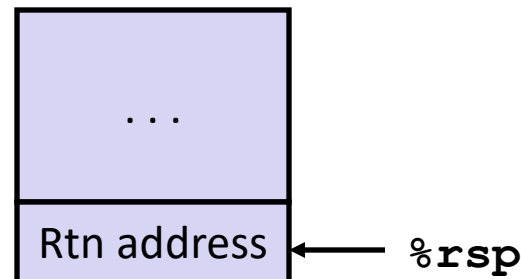
Quiz Time!

Canvas Quiz: Day 6 - Machine Procedures

Callee-Saved Example #1

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

Initial Stack Structure



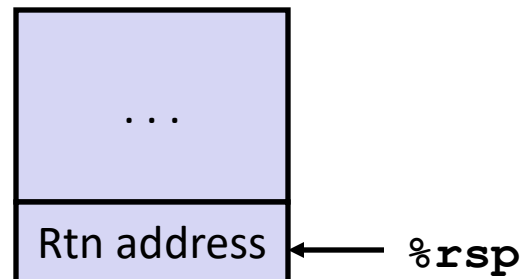
- **x** comes in register **%rdi**.
- We need **%rdi** for the call to **incr**.
- Where should be put **x**, so we can use it after the call to **incr**?

Callee-Saved Example #2

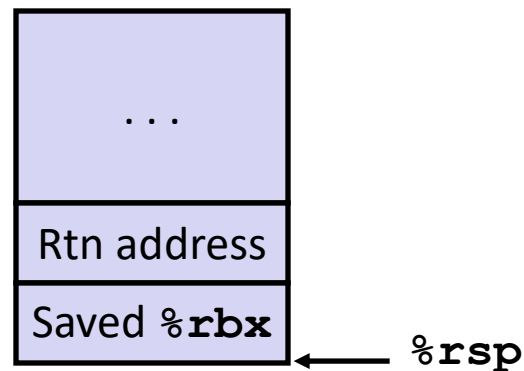
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

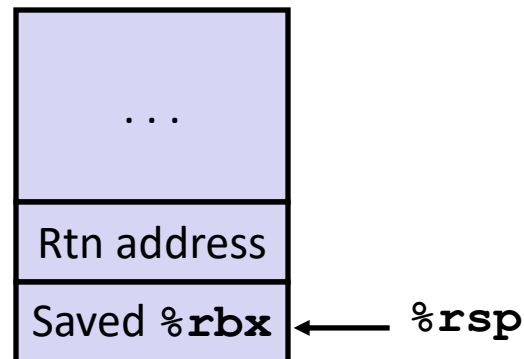


Callee-Saved Example #3

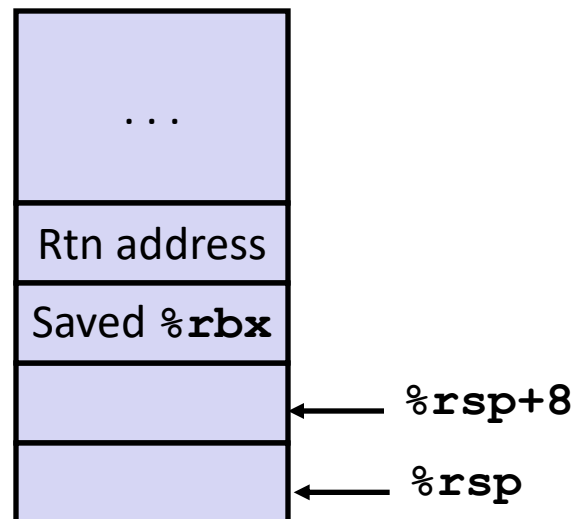
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

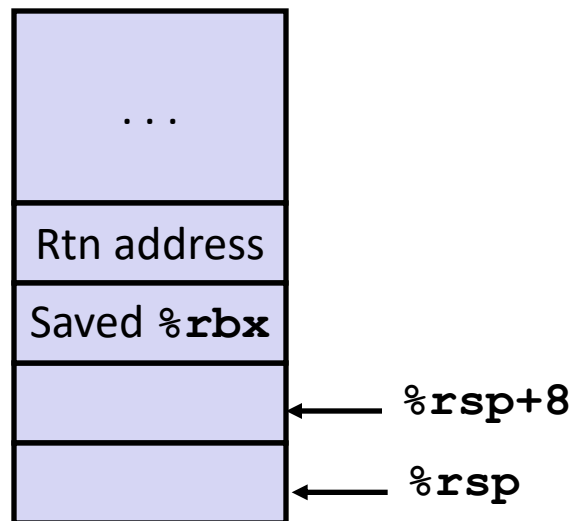


Callee-Saved Example #4

```
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



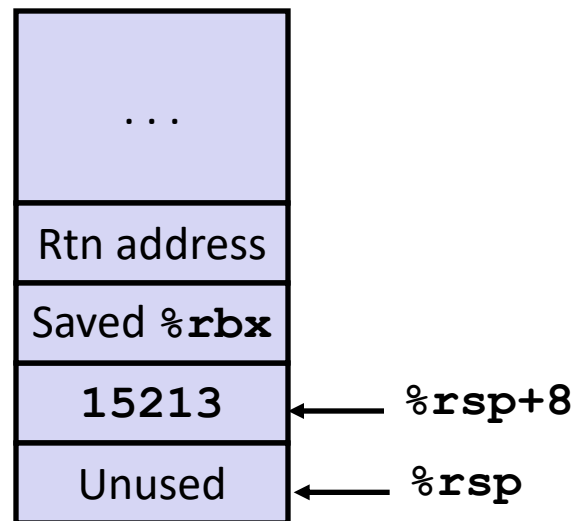
- `x` is saved in `%rbx`, a callee saved register

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



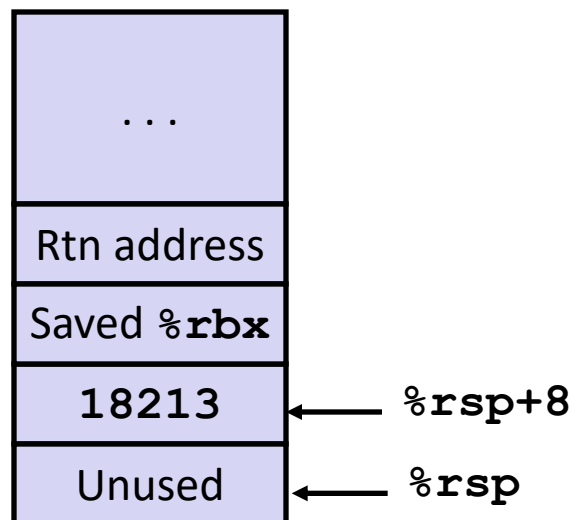
- **x** is saved in **%rbx**,
a callee saved register

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



Upon return from `incr`:

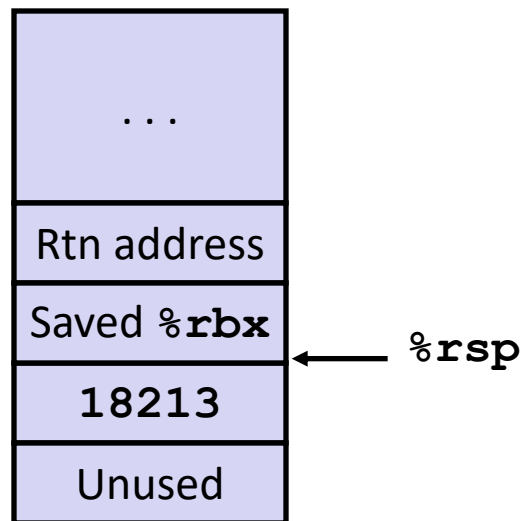
- `x` safe in `%rbx`
- Return val `v2` in `%rax`
- Compute `x+v2`:
`addq %rbx, %rax`

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



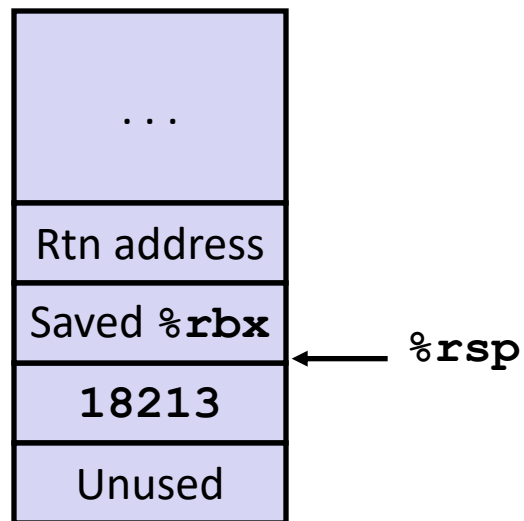
- Return result in **%rax**

Callee-Saved Example #8

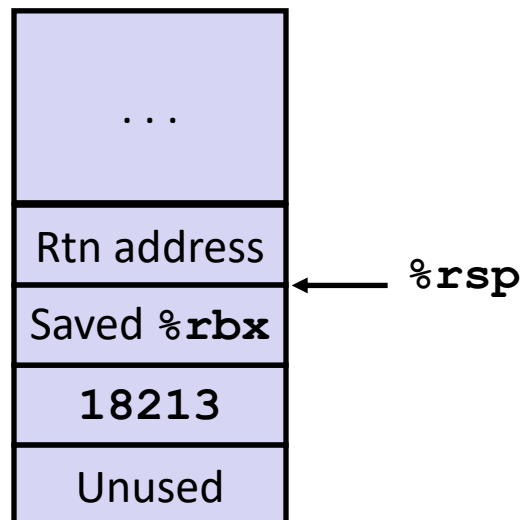
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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



Today

■ Procedures

- Mechanisms
- 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);
}

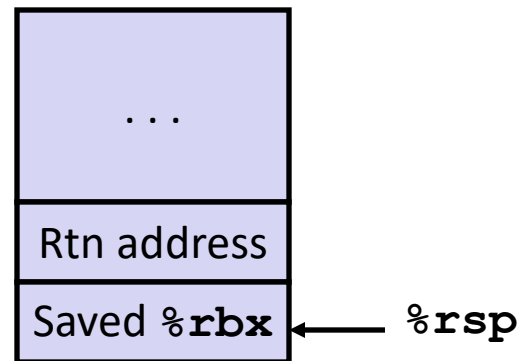
```

```

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	Recursive 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);
}

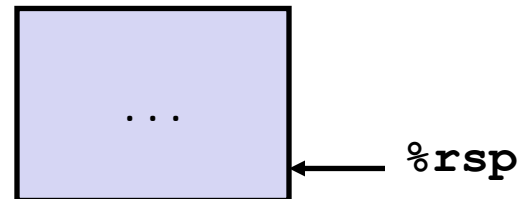
```

```

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 later Lecture)
- 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

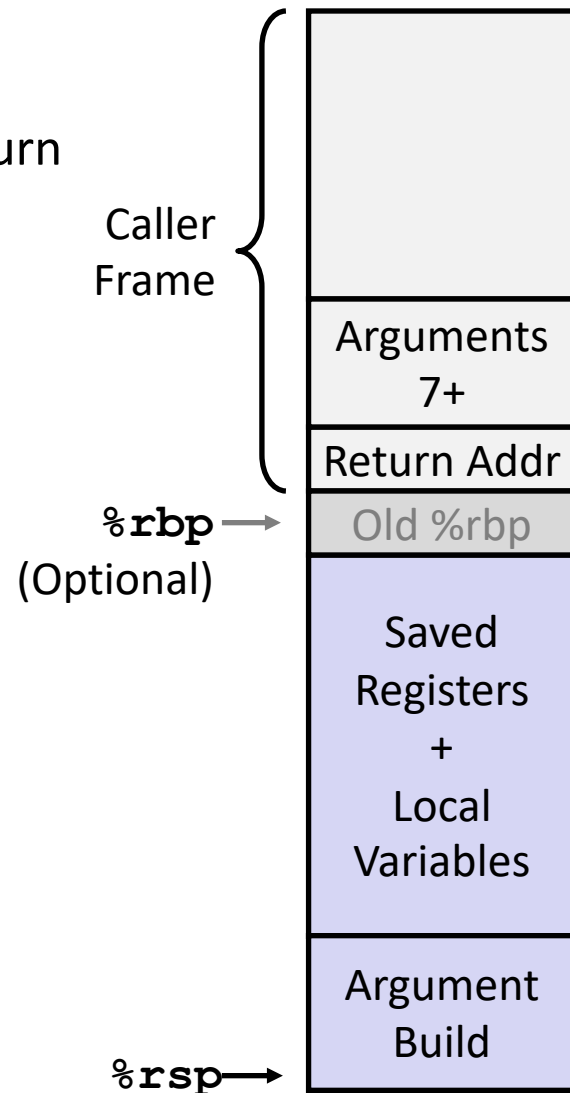
- Stack is the right data structure for procedure call/return
 - If P calls Q, then Q returns before P

■ Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments 7+ at top of stack
- Result return in **%rax**

■ Pointers are addresses of values

- On stack or global



Small Exercise

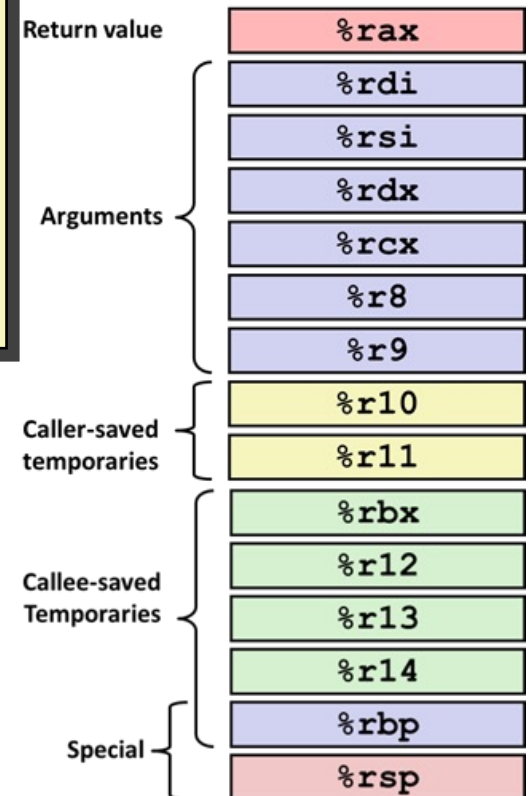
```

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.



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

```

