Machine-Level Programming: Control, Procedures

CSCI3240: Lecture 7 and 8

Dr. Arpan Man Sainju

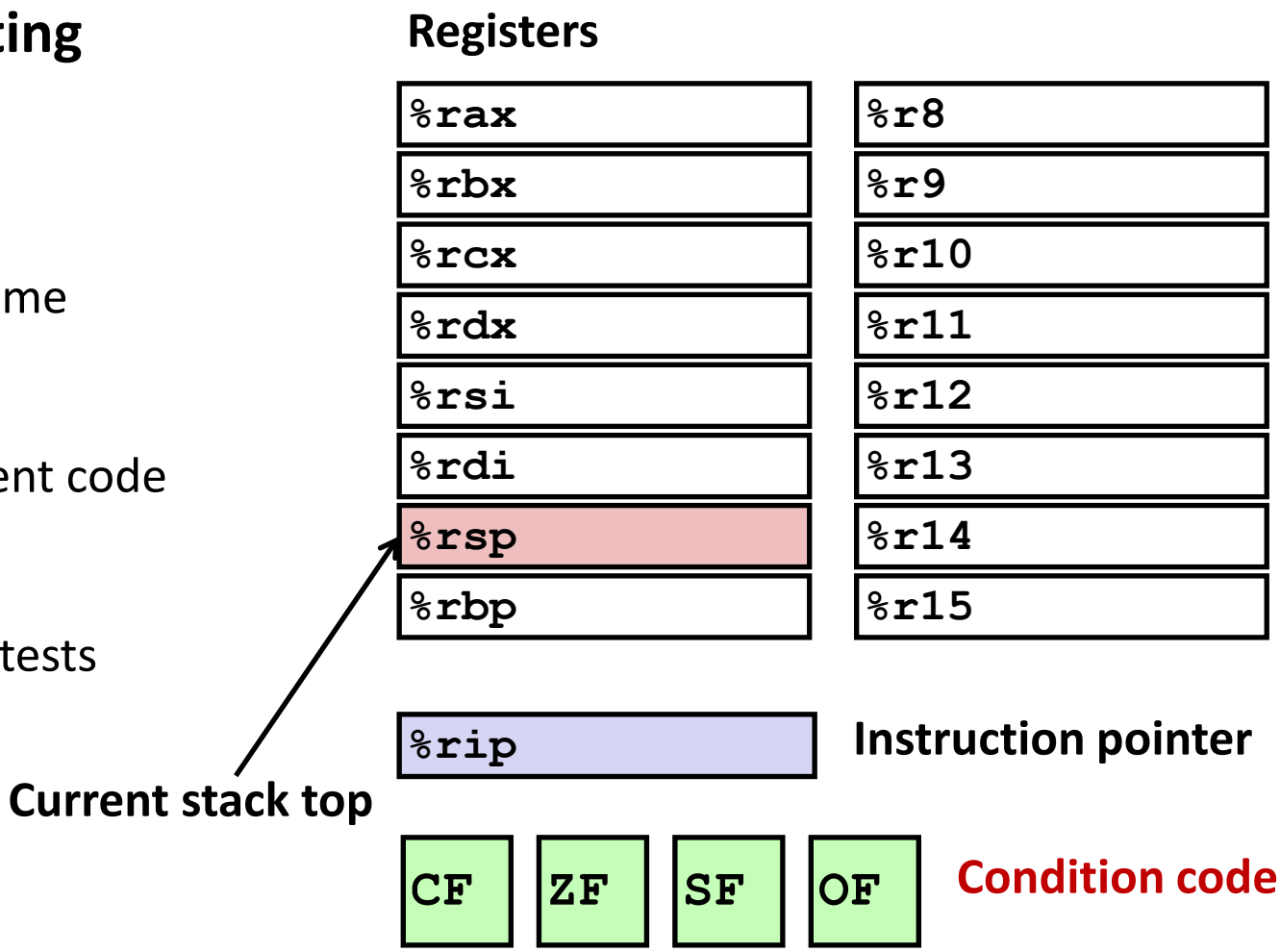
Middle Tennessee State University

Today

- **Control: Condition codes**
- Conditional branches
- Loops
- Switch Statements

Processor State (x86-64, Partial)

- **Information about currently executing program**
  - Temporary data (**`%rax`**, …)
  - Location of runtime stack (**`%rsp`**)
  - Location of current code control point (**`%rip`**, …)
  - Status of recent tests (**CF, ZF, SF, OF**)



Condition Codes

- **In addition to the integer registers, the CPU maintains a set of single-bit condition codes registers describing attributes of the most recent arithmetic or logical operation.**
  - These registers can be then be tested to perform conditional branches.
  - **CF: Carry Flag**: sets (becomes 1) when unsigned overflow occurs.
  - **ZF: Zero Flag**: The most recent operation yielded zero.
  - **SF: Sign Flag**: The most recent operation yielded a negative value.
  - **OF: Overflow Flag**: The most recent caused a two's-complement overflow.

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - **`cmpq`** *Src2*, *Src1*
  - **`cmpq b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Example: rax = -3 rdi = 2

```
cmpq %rax, %rdi
```

ZF = ?	SF = ?	OF = ?	Rax = ?	Rdi = ?
ZF = 0	SF = 0	OF = 0	Rax = -3	Rdi = 2

cmpq Exercise

- **A. rsi = Tmin r15 = -5**
  - `cmpq %rsi, %r15`

ZF =?	SF =?	OF =?	rsi =?	r15 =?

- **B. r12 = TMax, r13 =1**
  - `cmpq %r12, %r13`

ZF =?	SF =?	OF =?	r12 =?	r13 =?

Cmpq Exercise Answers

- **A. rsi = Tmin r15 = -5**
  - `cmpq %rsi, %r15` computes r15-rsi (without storing result)

ZF =0	SF =0	OF =1	rsi =TMin	r15 =-5

- **B. r12 = TMin, r13 =TMin**
  - `cmpq %r12, %r13` computes r13-r12 (without storing result)

ZF = 1	SF =0	OF =1	r12 =Tmin	r13 =Tmin

Carry Flag Examples (Working with Unsigned integers)

- **A. rsi = UMax r15 = 1**
  - `addq %rsi, %r15`

CF =?	ZF =?	rsi =?	r15 =?

- **B. r12 = TMax, r13 =TMax**
  - `addq %r12, %r13`

CF = ?	ZF = ?	r12 =?	r13 = ?

Note: SF and OF are ignored.

Carry Flag Examples (Working with Unsigned integers)

- **A. rsi = UMax r15 = 1**
  - `addq %rsi, %r15`

CF =1	ZF =1	rsi =UMax	r15 =0

- **B. r12 = TMax, r13 =TMax**
  - `addq %r12, %r13`

CF = 0	ZF = 0	r12 =Tmax	r13 = Umax -1

Note: SF and OF are ignored.

Carry Flag Example (Working with Unsigned integers)

- **A. rsi = UMax r15 = Umax**
  - `addq %rsi, %r15`

CF =?	ZF =?	rsi = ?	r15 =?

- **C. r13 =UMax**
  - `shlq $1, %r13`

CF = ?	ZF = ?	r13 =?

Note: SF and OF are ignored.

Carry Flag Example (Working with Unsigned integers)

- **A. rsi = UMax r15 = Umax**
  - `addq %rsi, %r15`

CF =1	ZF =0	rsi = UMax	r15 =Umax-1

- **C. r13 =UMax**
  - `shlq $1, %r13`

CF = 1	ZF = 0	r13 =Umax-1

Note: SF and OF are ignored.

Overflow Flag Examples (Working with Signed numbers)

- **A. rsi = TMin r15 = -1**
  - `addq %rsi, %r15`

CF =Ignored	ZF =0	SF =0	OF =1	rsi =Tmin	r15 =Tmax

- **B. r12 = TMax, r13 =TMax**
  - `addq %r12, %r13`

CF = Ignored	ZF = 0	SF =1	OF =1	r12 =Tmax	r13 = -2

Note: A negative sum of positive operands (or vice versa) is an overflow.

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - **`testq`** *Src2*, *Src1*
    - **`testq b,a`** like computing **`a&b`** without altering the destination
    - Typically, same operands are repeated:
      - **`Testq %rax, %rax`** (to check if rax is 0, +ve or -ve)
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
  - **ZF set** when **`a&a == 0`**
  - **SF set** when **`a&a < 0`**

Testq Exercise

- **A. rsi = -5**
  - `testq %rsi, %rsi`

ZF =?	SF =?	OF =?	rsi =?

- **B. r12 = 1234**
  - `testq %r12, %r12`

ZF =?	SF =?	OF =?	r12 =?

- **C. r13 = 0**
  - `testq %r13, %r13`

ZF =?	SF =?	OF =?	r13 =?

Testq Exercise Answers

- **A. rsi = -5**
  - `testq %rsi, %rsi`

ZF =0	SF =1	OF =0	rsi =-5

- **B. r12 = 1234**
  - `testq %r12, %r12`

ZF =0	SF =0	OF =0	r12 =1234

- **C. r13 = 0**
  - `testq %r13, %r13`

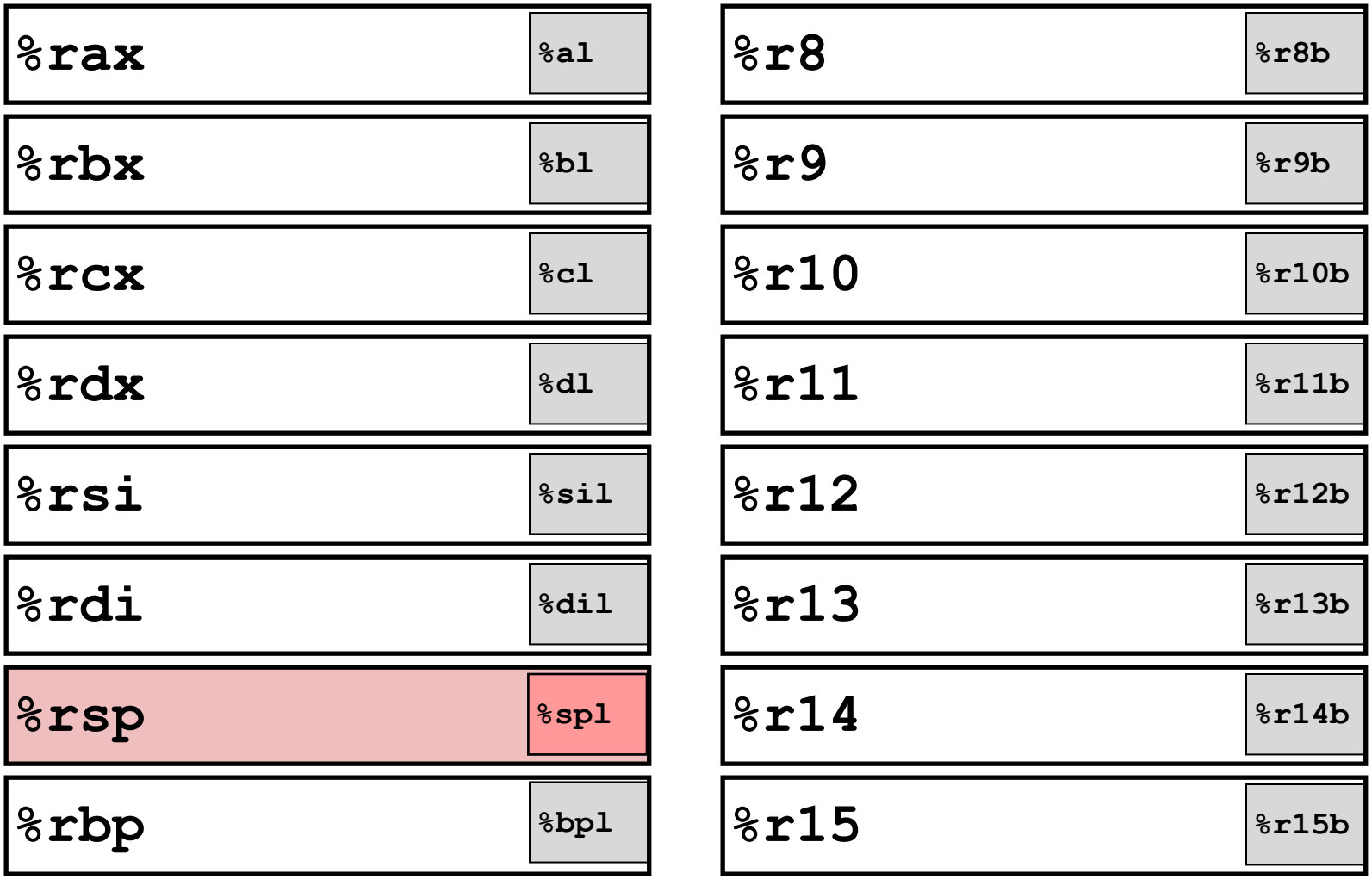
ZF =1	SF =0	OF =0	r13 =0

Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

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

x86-64 Integer Registers



- Can reference low-order byte

Reading Condition Codes (Cont.)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
    cmpq    %rsi, %rdi    # Compare x:y
    setg    %al           # Set when >
    movzbl  %al, %eax     # Zero rest of %rax
    ret
```

Exercise

```
int fun (long x, long y)
{
  return x ? y;
}
```

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

```
    cmpq    %rsi, %rdi    # Compare x:y
    setle    %al           # Set when ?
    movzbl %al, %eax     # Zero rest of %rax
    ret
```

SetX	Effect	Set Condition
sete *Dest*	*Dest* ← ZF	Equal / Zero
setne *Dest*	*Dest* ← ~ZF	Not Equal / Not Zero
sets *Dest*	*Dest* ← SF	Negative
setns *Dest*	*Dest* ← ~SF	Nonnegative
setg *Dest*	*Dest* ← ~(SF^OF) &~ZF	Greater (Signed)
setge *Dest*	*Dest* ← ~(SF^OF)	Greater or Equal (Signed)
setl *Dest*	*Dest* ← (SF^OF)	Less (Signed)
setle *Dest*	*Dest* ← (SF^OF) | ZF	Less or Equal (Signed)

Data Movement Instructions

Instruction	Effect	Description
MOV S, D	D <- S	Move with zero extension
movb		Move byte
movw		Move word (16 bit)
movl		Move double word (32 bit)
movq		Move zero extended **word** to double word (64 bits)

Data Movement Instructions

Instruction	Effect	Description
MOVZ Src , D	D <- ZeroExtended(S)	Move with zero extension
movzbw		Move zero extended byte to word (16 bits)
movzbl		Move zero extended byte to double word (32 bits)
movzbq		Move zero extended byte to quad word (64 bits)
movzwl		Move zero extended **word** to double word (32 bits)
movzwq		Move zero extended **word** to quad word (64 bits)

Today

- Control: Condition codes
- **Conditional branches**
- Loops
- Switch Statements

Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

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

Conditional Branch Example (Old Style)

- **Generation**

```
$ gcc -Og -S -fno-if-conversion control.c
```

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi #rdi-rsi
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                   # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

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

Expressing with Goto Code

- C allows `goto` statement
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;
eles_val = Else_Expr;
nt = !Test;
if (nt) result = else_val;
return result;
```

Conditional Move Example

```
long absdiff
   (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

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

Note: cmovle implies conditional move if the comparison is less than equal to

```
absdiff:
    movq    %rdi, %rax   # x
    subq    %rsi, %rax   # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx   # else_val = y-x
    cmpq    %rsi, %rdi   # x:y
    cmovle  %rdx, %rax   # if <=, result = else_val
    ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Today

- Control: Condition codes
- Conditional branches
- **Loops**
- Switch Statements

“Do-While” Loop Example

C Code

```
long pcount_do
   (unsigned long x) {
   long result = 0;
   do {
     result += x & 0x1;
     x >>= 1;
   } while (x);
   return result;
}
```

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

- Count number of 1’s in argument `x` (“popcount”)
- Use conditional branch to either continue looping or to exit loop

"Do-While" Loop Compilation

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
      movl    $0, %rax    #  result = 0
.L2:                      # loop:
      movq    %rdi, %rdx
      andl    $1, %rdx    #  t = x & 0x1
      addq    %rdx, %rax  #  result += t
      shrq    %rdi        #  x >>= 1
      jne     .L2         #  if (x) goto loop
      rep; ret
```

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

- **Body:**

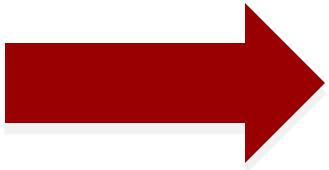
```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

General "While" Translation #1

- **"Jump-to-middle" translation**
- **Used with** `-Og`

While version

```
while (Test)
  Body
```



Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example #1

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Jump to Middle

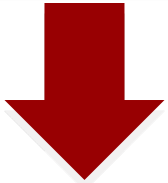
```
long pcount_goto_jtm
   (unsigned long x) {
   long result = 0;
   goto test;
 loop:
   result += x & 0x1;
   x >>= 1;
 test:
   if(x) goto loop;
   return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General "While" Translation #2

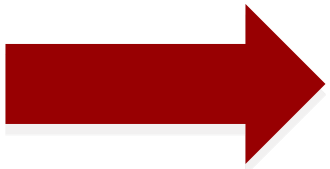
While version

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```



Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

- **"Do-while" conversion**
- **Used with –O1**

While Loop Example #2

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Do-While Version

```
long pcount_goto_dw
   (unsigned long x) {
   long result = 0;
   if (!x) goto done;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
 done:
   return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

“For” Loop Form

General Form

```
for (Init; Test; Update)
    Body
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =(x >> i) & 0x1;
  result += bit;
}
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

"For" Loop → While Loop

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =(x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =(x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

"For" Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;             Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

Today

- Control: Condition codes
- Conditional branches
- Loops
- **Switch Statements**

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

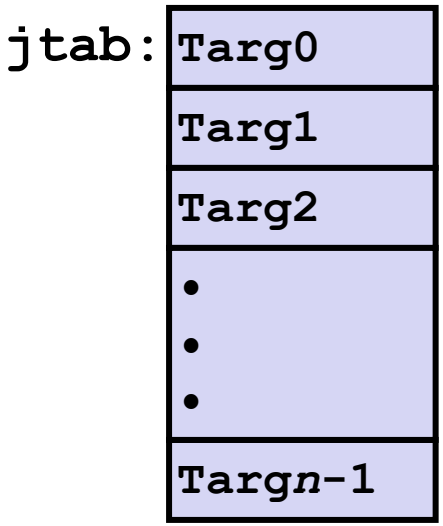
- **Multiple case labels**
  - Here: 5 & 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

Jump Table Structure

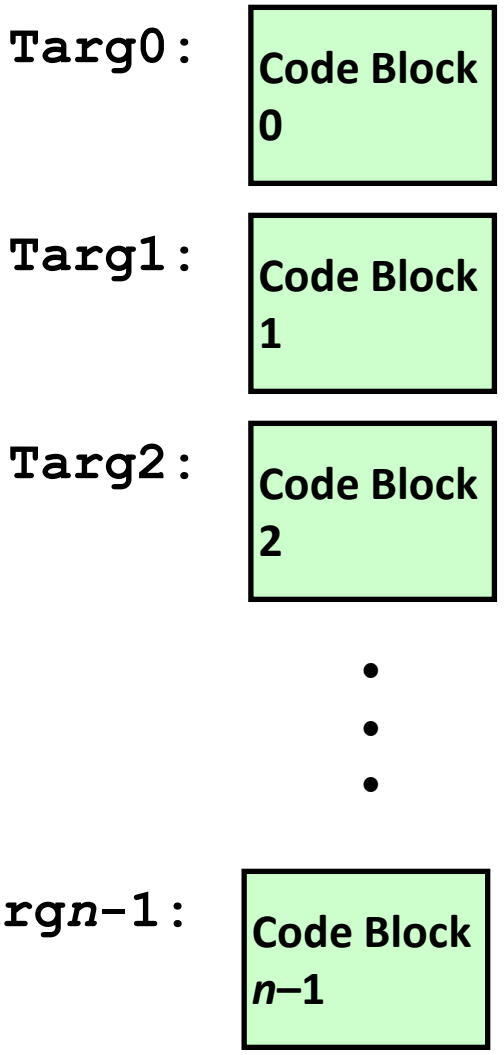
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Translation (Extended C)

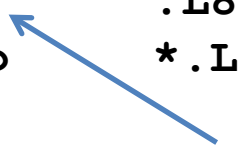
```
goto *JTab[x];
```

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```



What range of values takes default?

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

Note that **w** not initialized here

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes
  - Base address at **`.L4`**

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

- **Jumping**
  - **Direct:** **`jmp .L8`**
  - Jump target is denoted by label **`.L8`**
  - **Indirect:** **`jmp *.L4(,%rdi,8)`**
  - Start of jump table: **`.L4`**
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address **`.L4 + x*8`**
    - Only for $0 \le \mathbf{x} \le 6$

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

Code Blocks (x == 1)

```
  switch(x) {
  case 1:    // .L3
        w = y*z;
        break;
    . . .
}
```

```
.L3:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

Handling Fall-Through

```
long w = 1;
. . .
switch(x) {
. . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
. . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

Code Blocks (x == 2, x == 3)

```
    long w = 1;
. . .
    switch(x) {
. . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
. . .
    }
```

```
.L5:                        # Case 2
   movq     %rsi, %rax
   cqto     # sign extend rax to rdx:rax
   idivq    %rcx           #  y/z
   jmp      .L6            #  goto merge
.L9:                        # Case 3
   movl     $1, %eax       #  w = 1
.L6:                        # merge:
   addq     %rcx, %rax     #  w += z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Code Blocks (x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                    # Case 5,6
  movl   $1, %eax       #  w = 1
  subq   %rdx, %rax     #  w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #  2
  ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

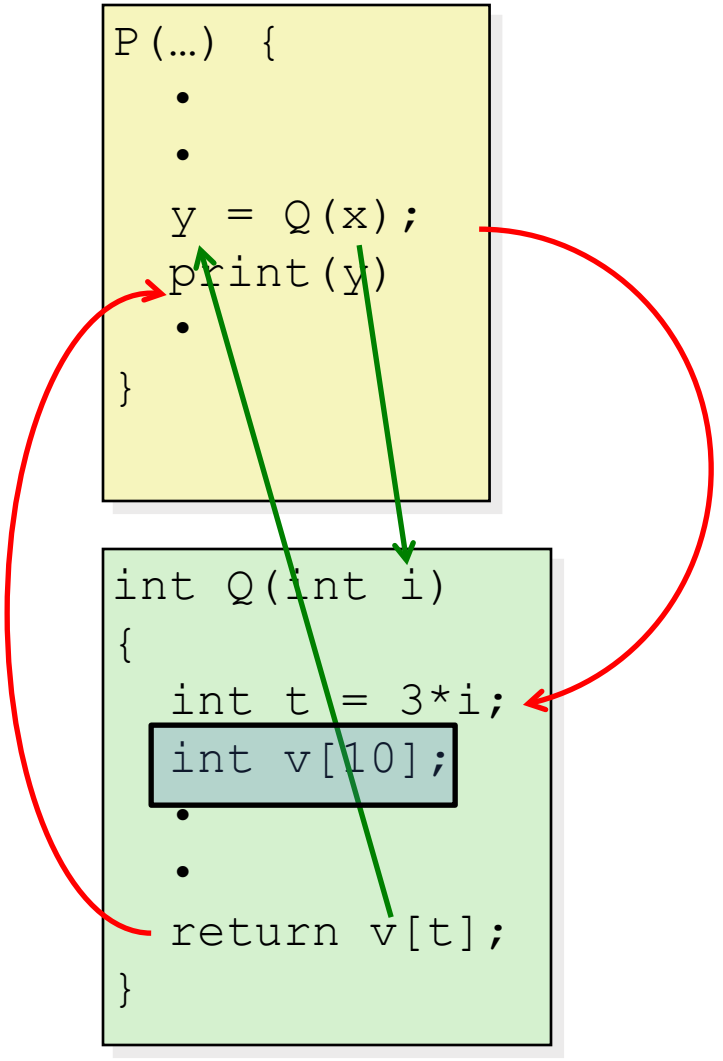
Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch
- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- **Standard Techniques**
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)

Procedures

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**

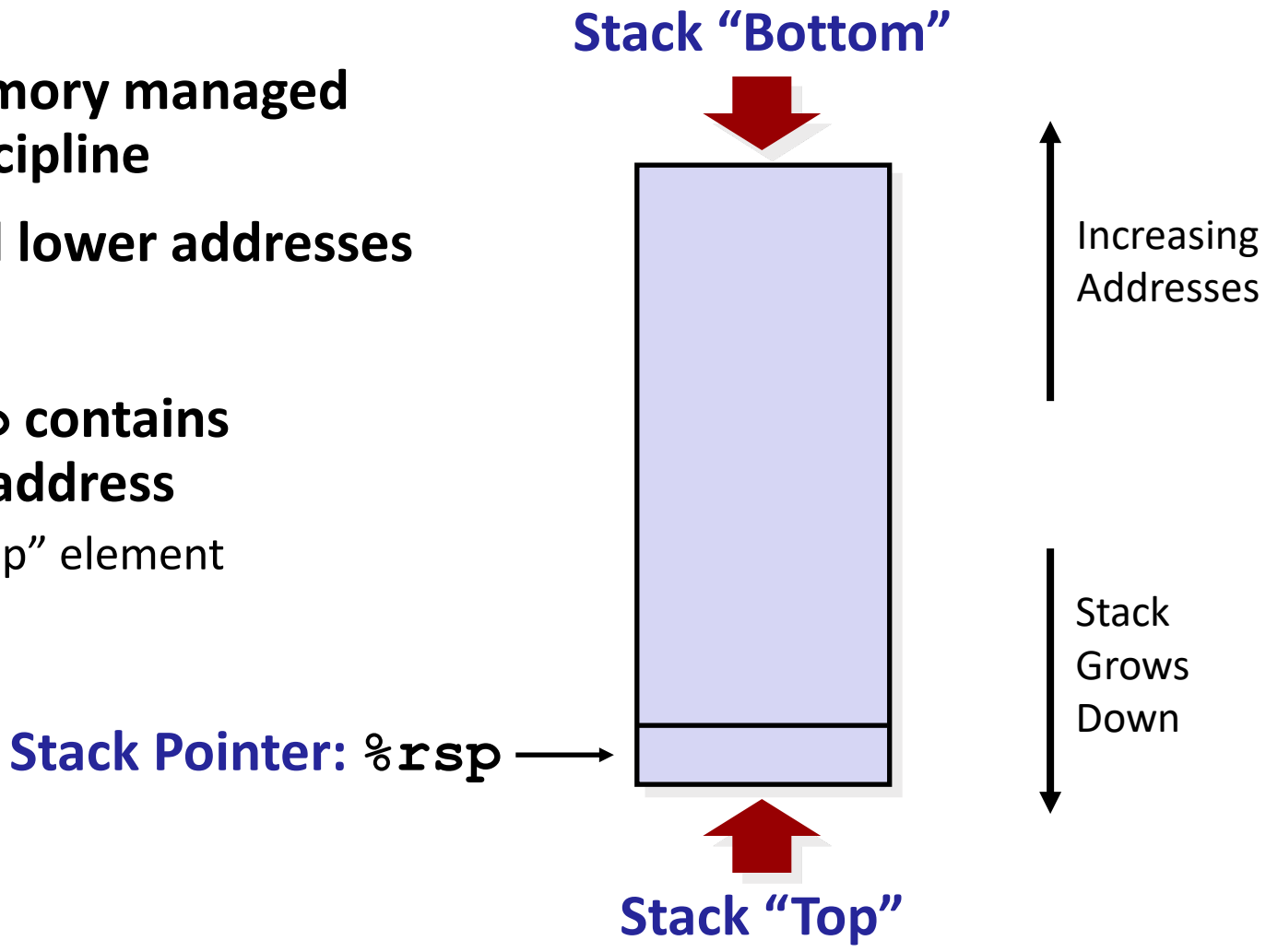


Today

- **Procedures**
  - **Stack Structure**
  - **Calling Conventions**
    - **Passing control**
    - **Passing data**

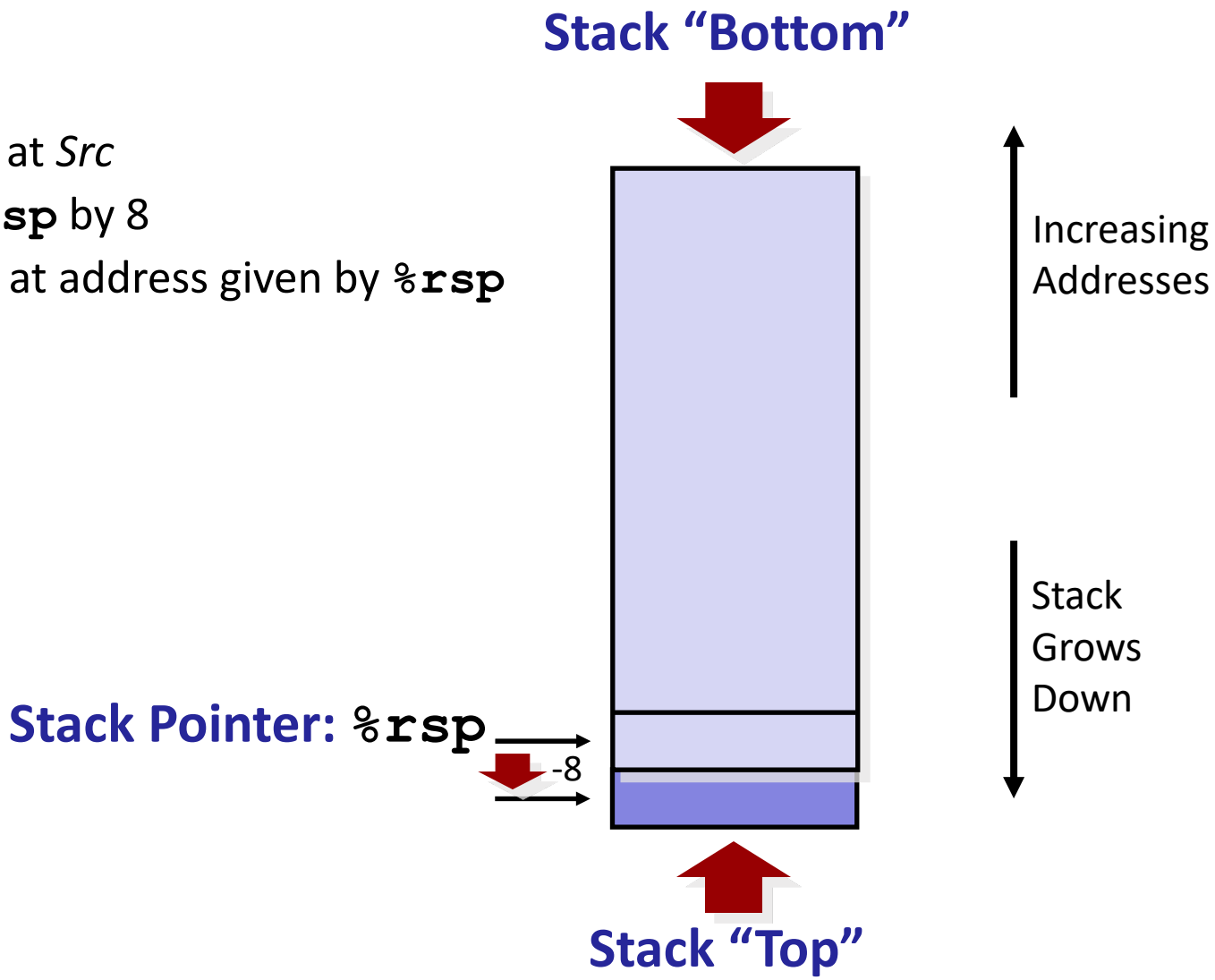
x86-64 Stack

- **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

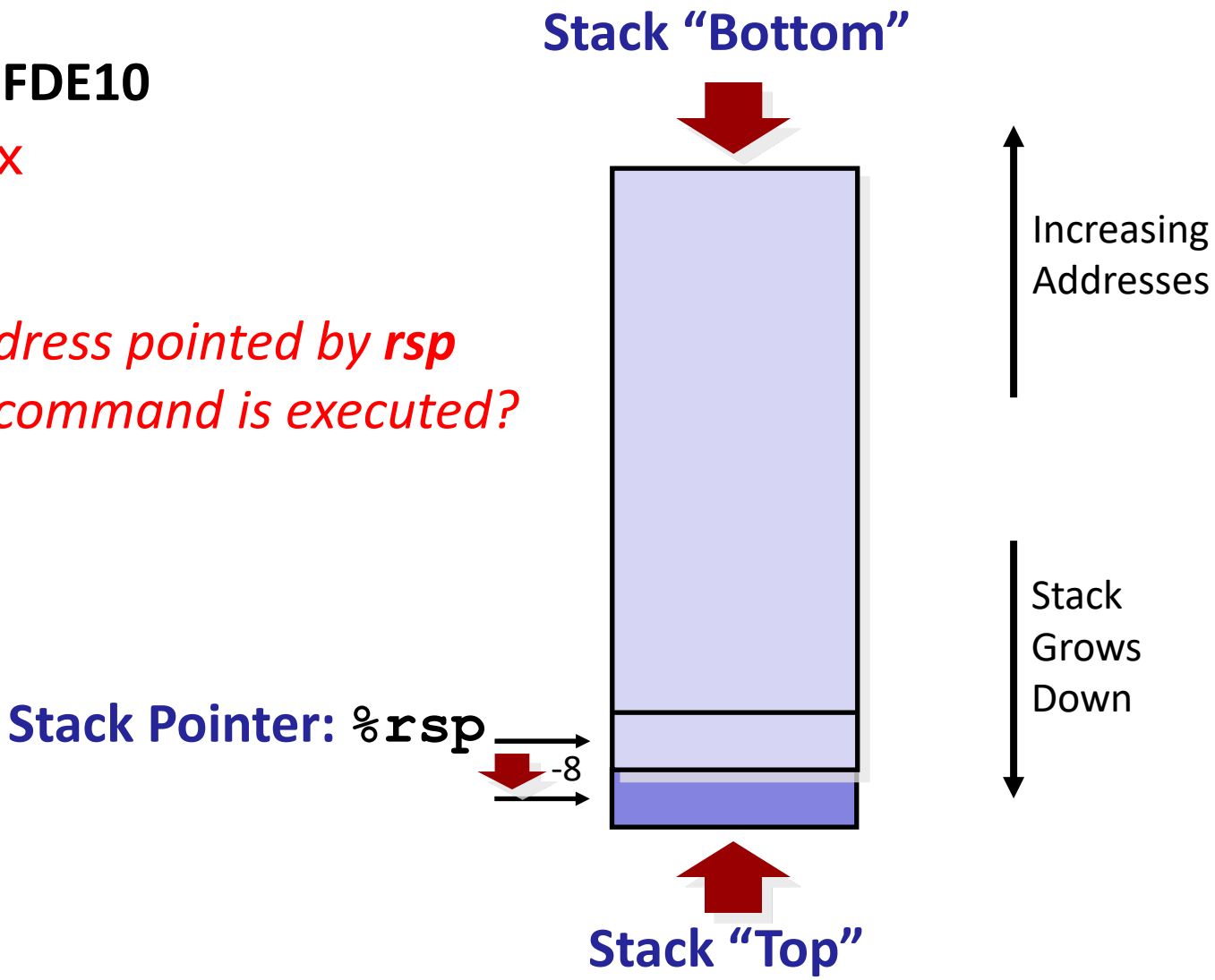
- **`pushq` *Src***
  - Fetch operand at *Src*
  - Decrement **`%rsp`** by 8
  - Write operand at address given by **`%rsp`**



Exercise

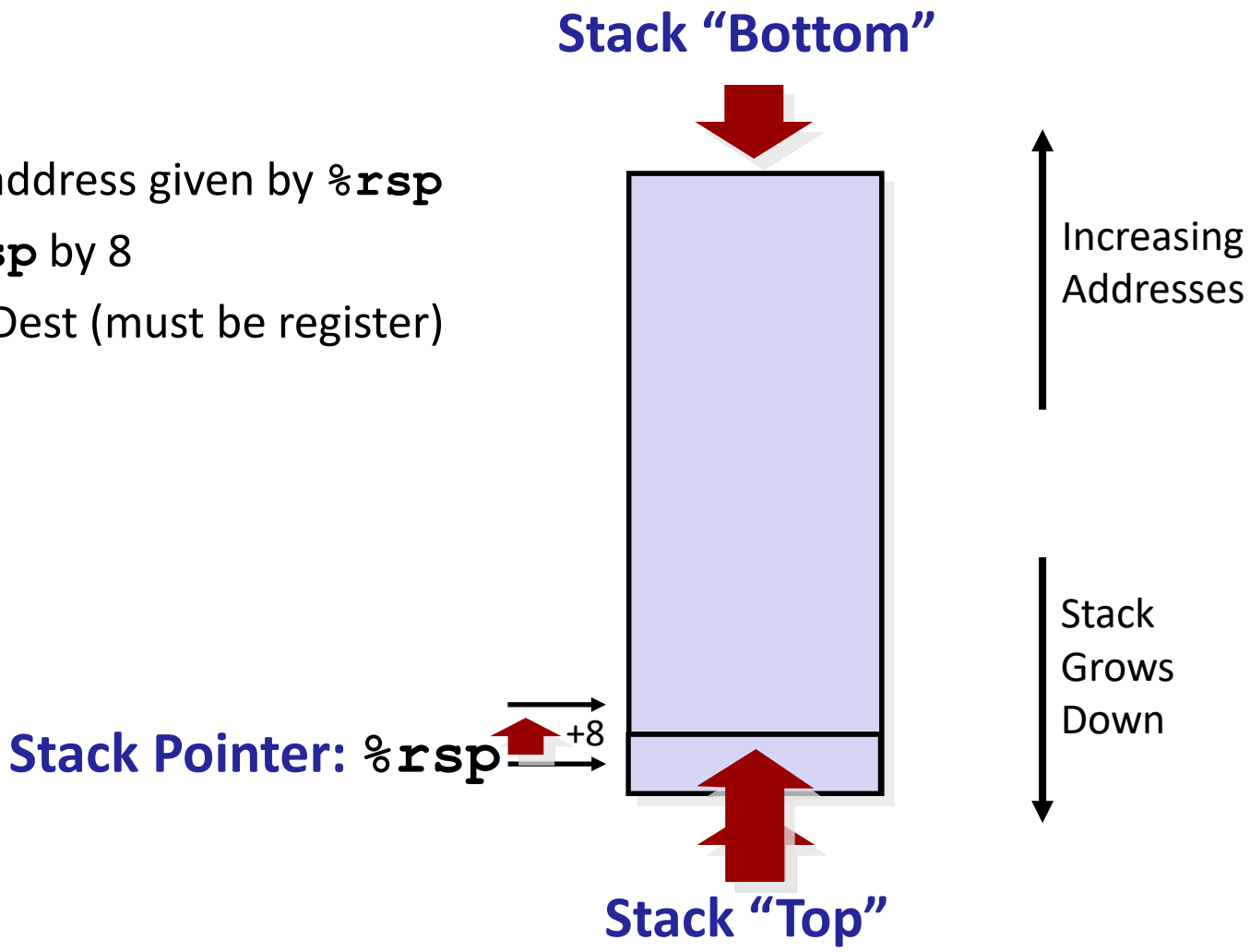
- **Let rsp = 0xFFFFDE10**
  - `push %rax`

*What is the address pointed by **rsp** after the push command is executed?*



x86-64 Stack: Pop

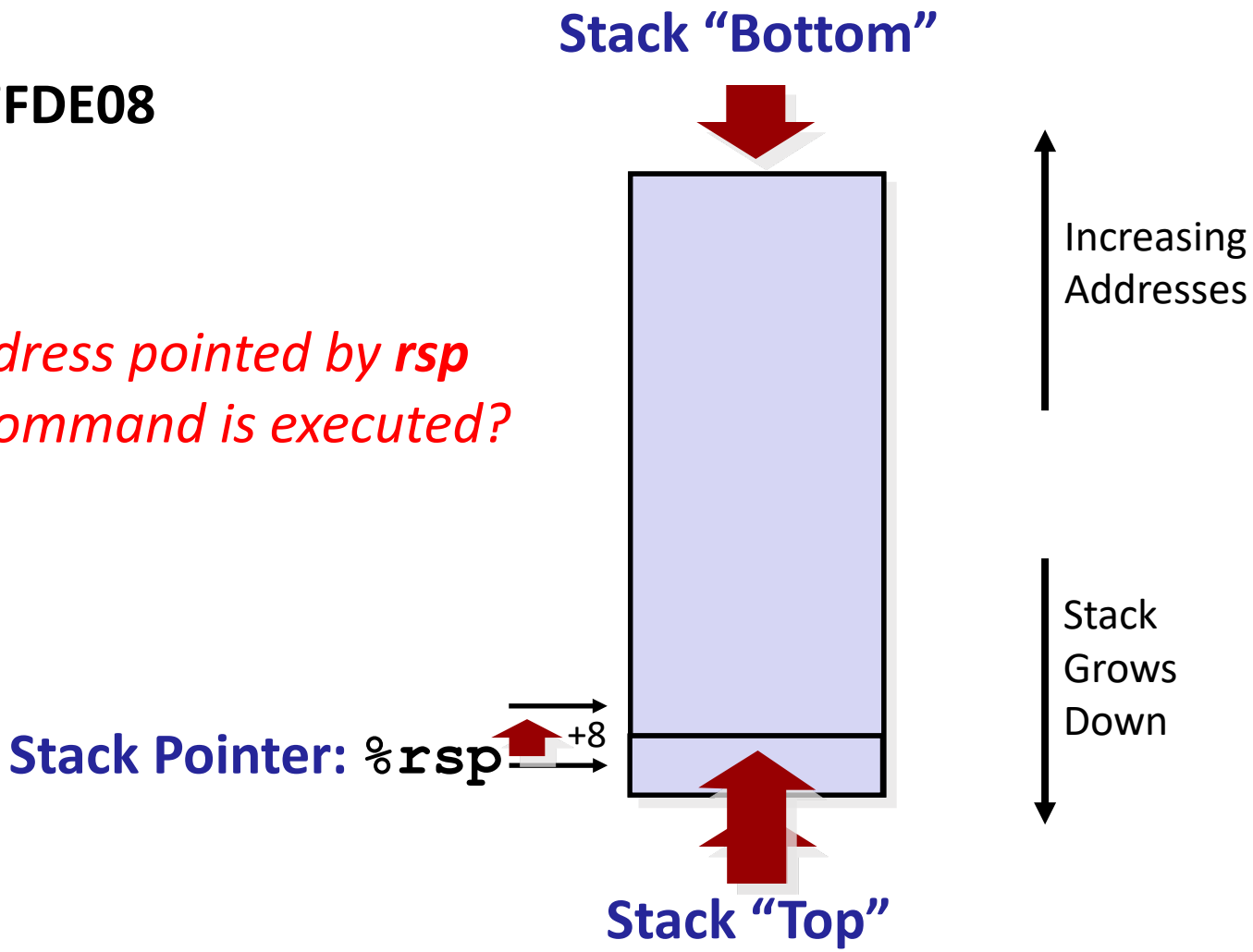
- **`popq` *Dest***
  - Read value at address given by **`%rsp`**
  - Increment **`%rsp`** by 8
  - Store value at Dest (must be register)



Exercise

- **Let rsp = 0xFFFFDE08**
  - `pop %rax`

*What is the address pointed by **rsp** after the pop command is executed?*



Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - **Passing control**
    - Passing data

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
```

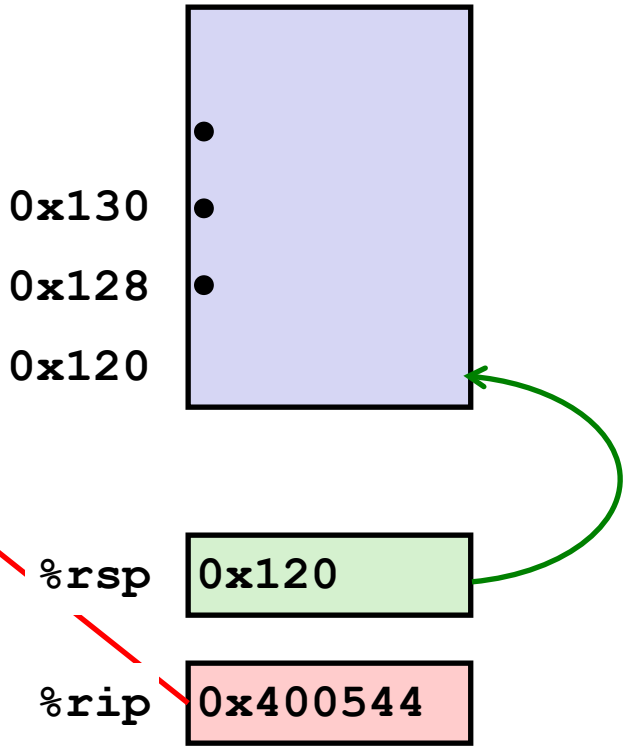
Procedure Control Flow

- **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)
  •
  •
```

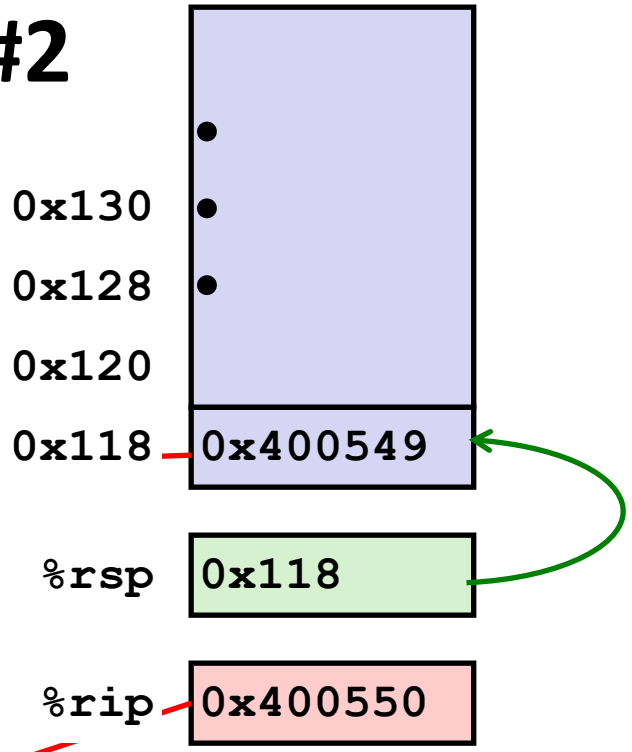
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2

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

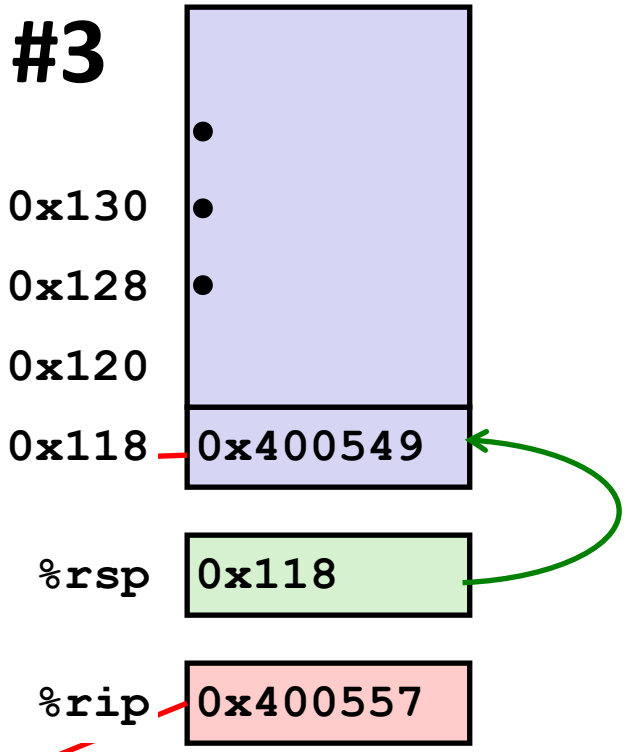
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3

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

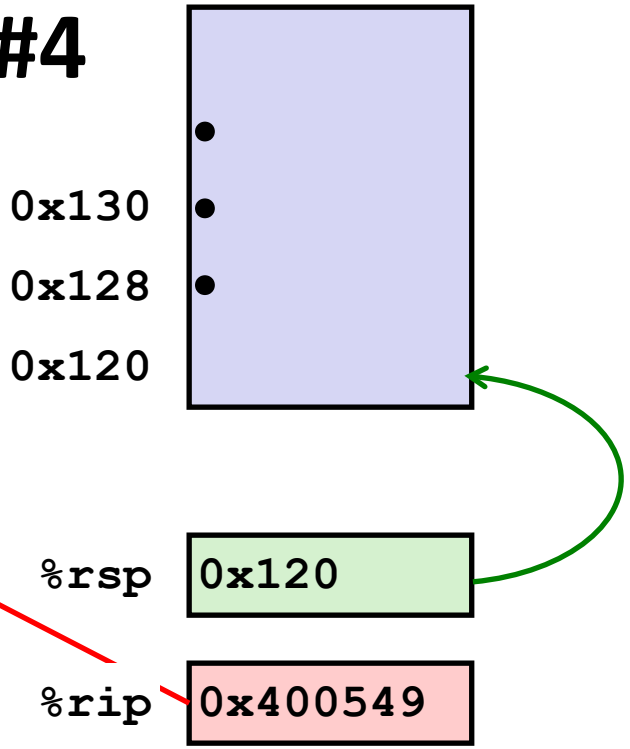
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #4

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

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



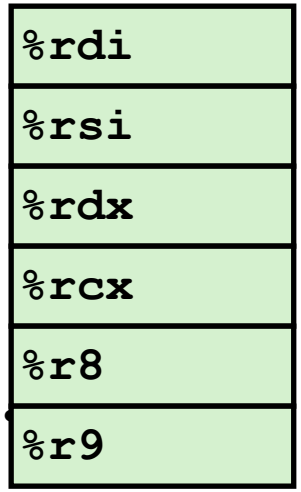
Today

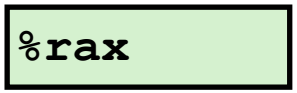
- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - **Passing data**

Procedure Data Flow

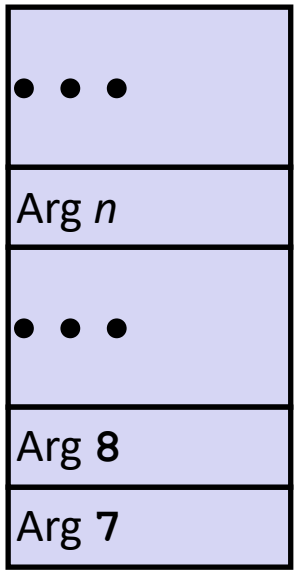
Registers

- First 6 arguments





Stack



- Only allocate stack space when needed

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

Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be "*Reentrant*"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- **Stack discipline**
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- **Stack allocated in *Frames***
  - state for single procedure instantiation

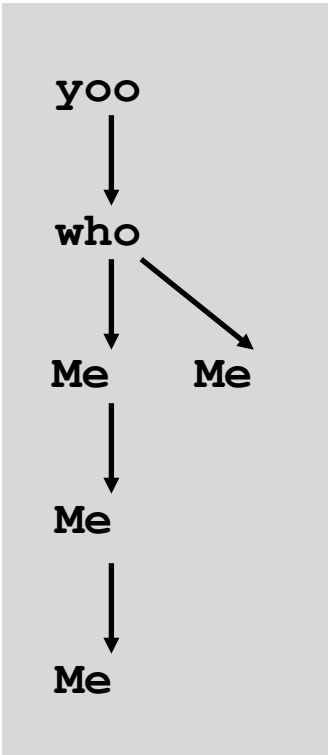
Call Chain Example

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

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

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

Example Call Chain

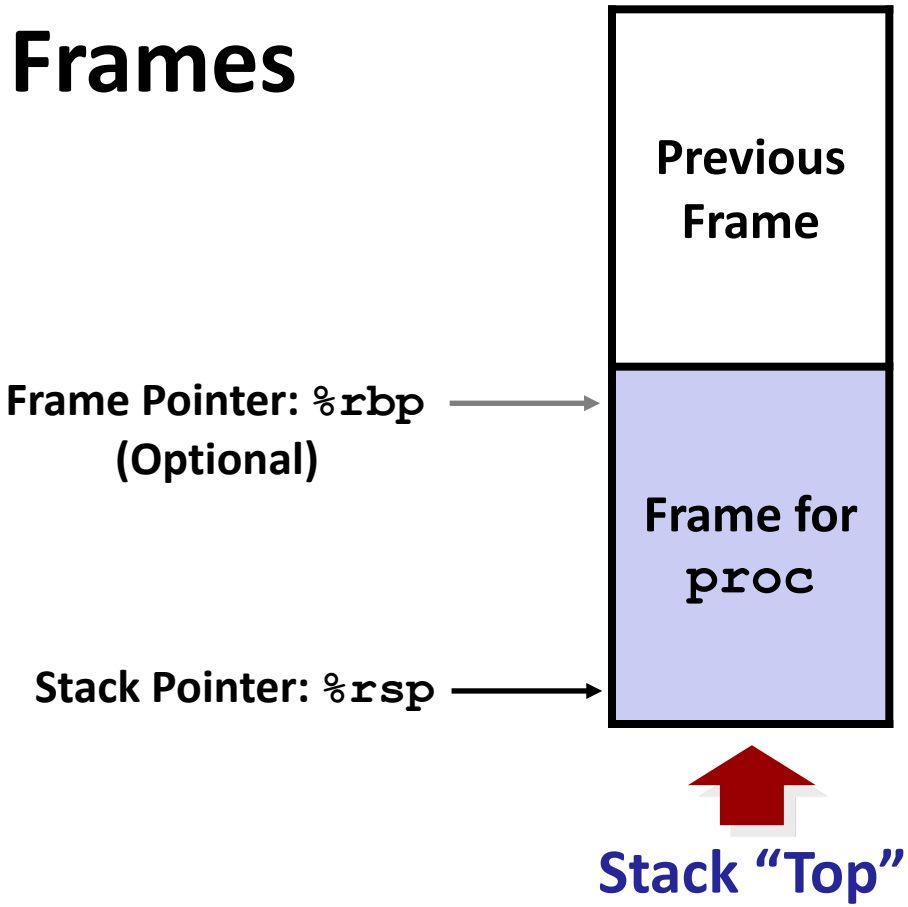


Procedure `Me()` is recursive

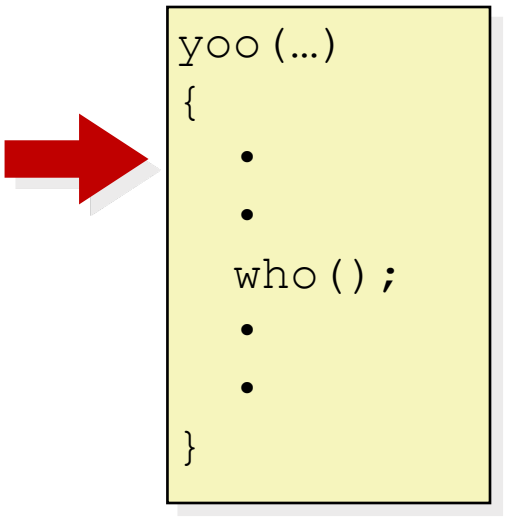
Stack Frames

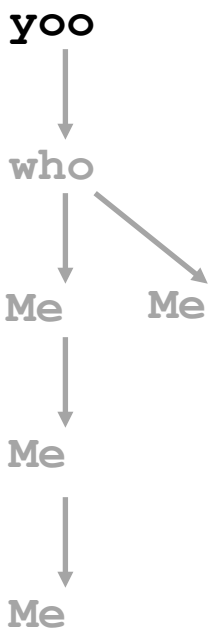
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

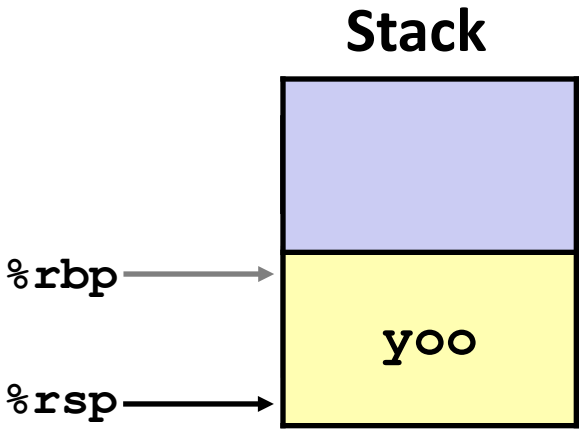
- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction



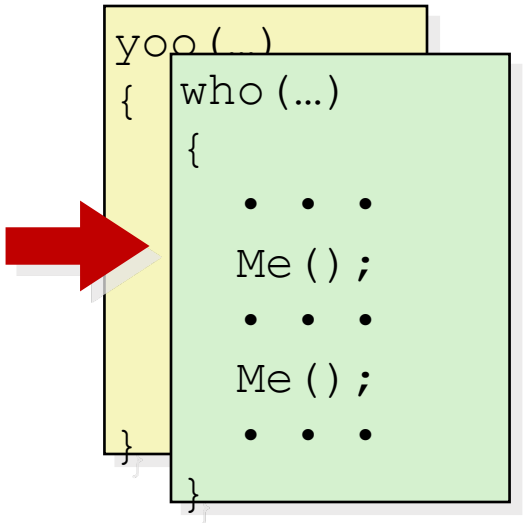
Example

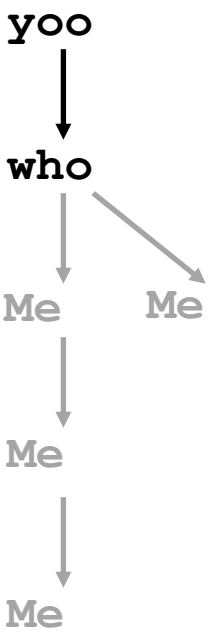


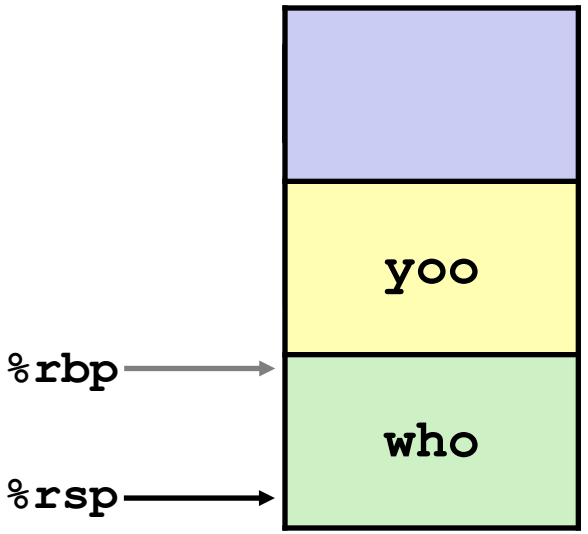




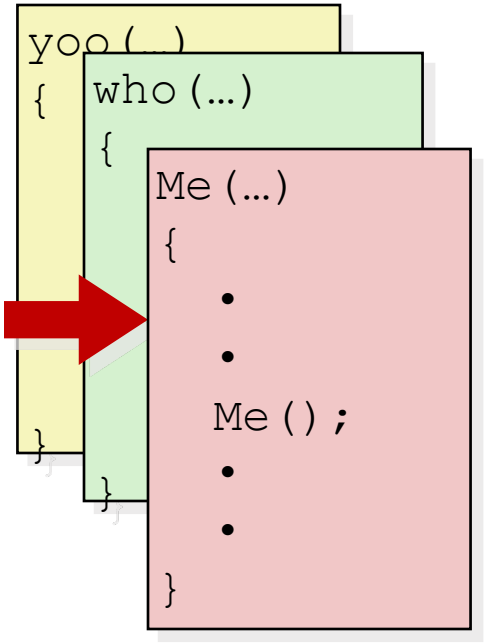
Example

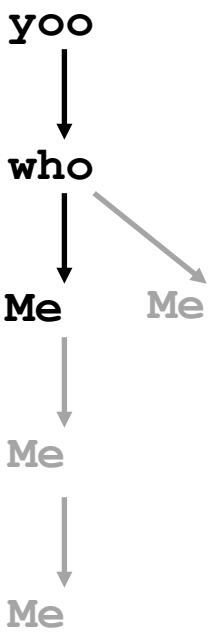


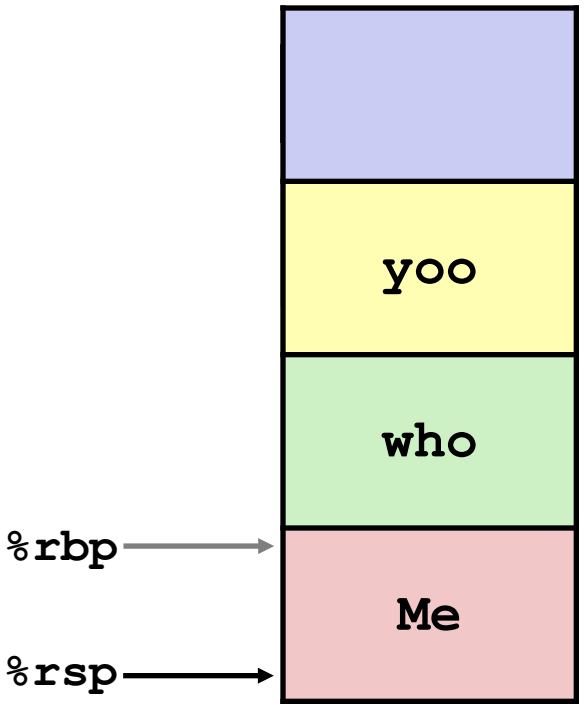




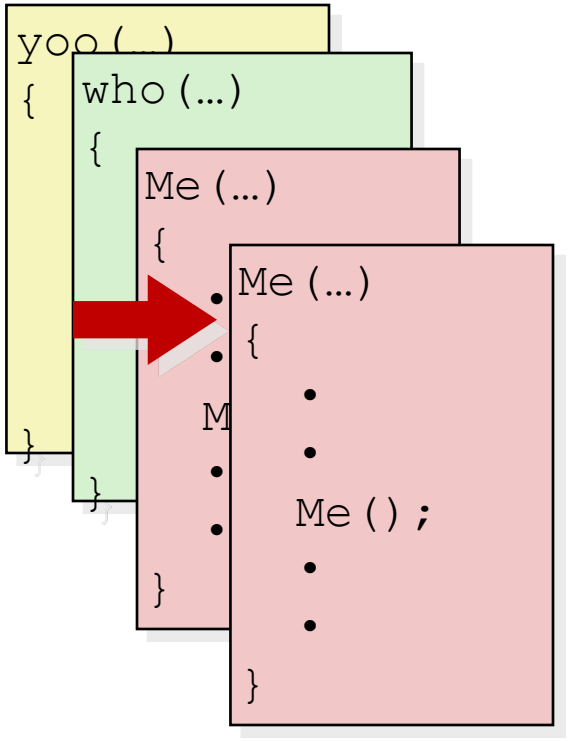
Example

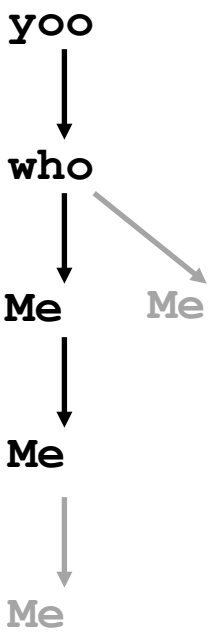


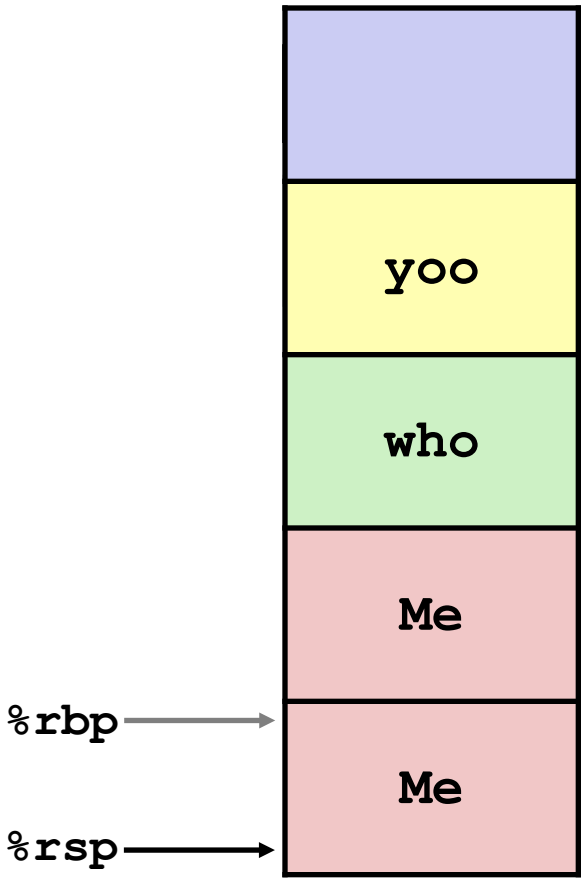




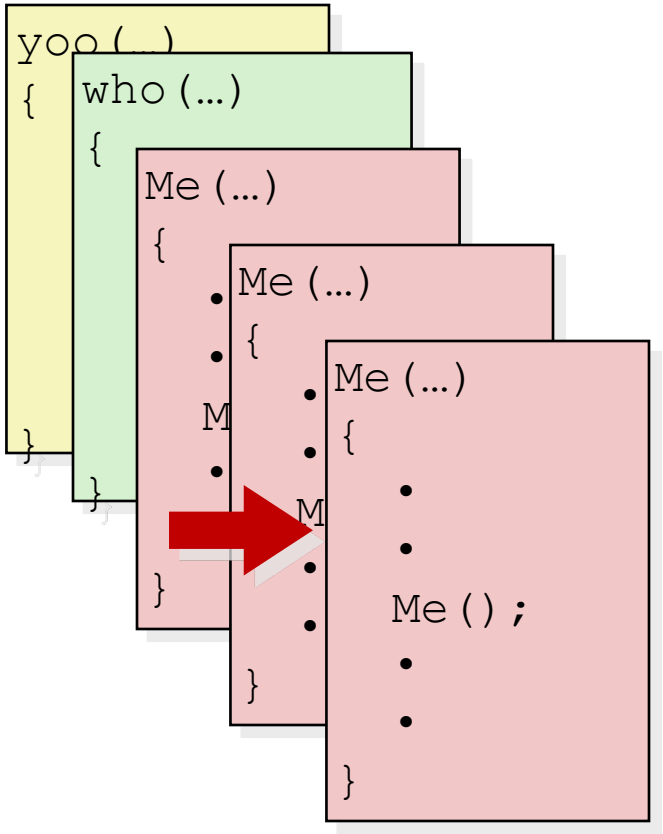
Example

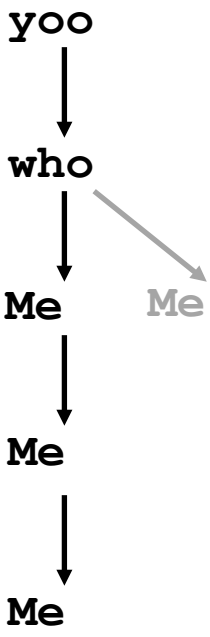


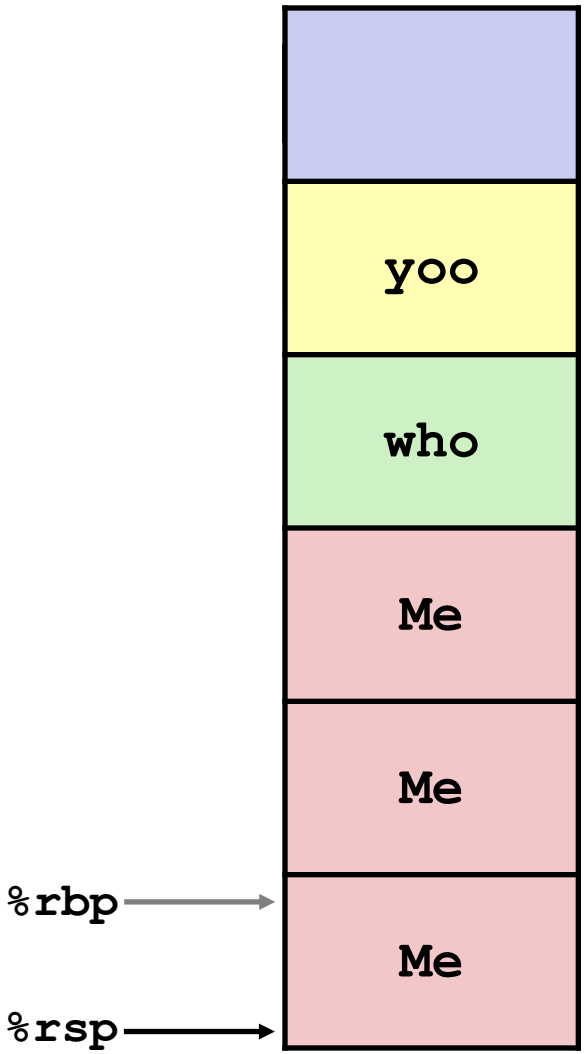




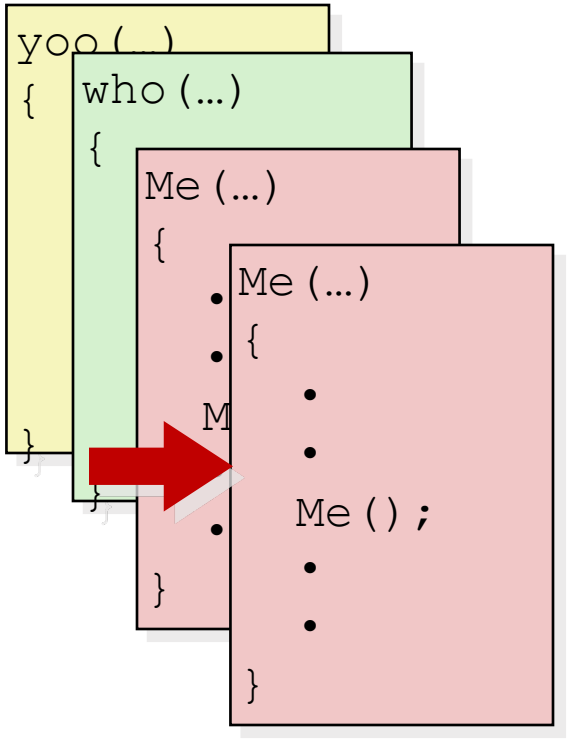
Example

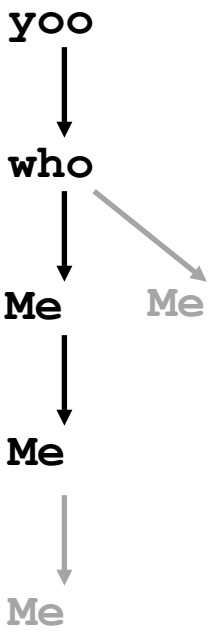

yoo(…)
{
who(…)
{
Me(…)
{
Me(…)
{
Me(…)
{
Me();
}

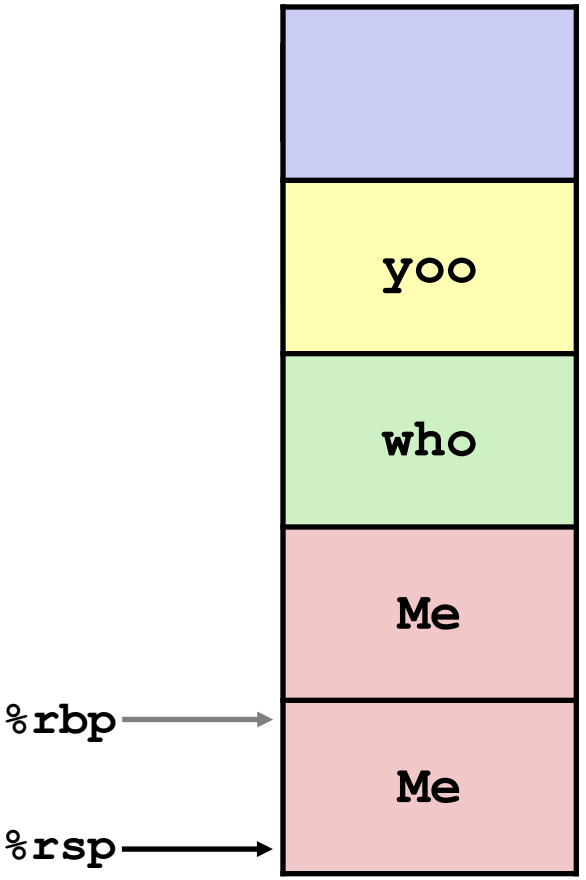

yoo
who
Me
Me
Me
Me


Stack
yoo
who
Me
Me
Me
%rbp
%rsp

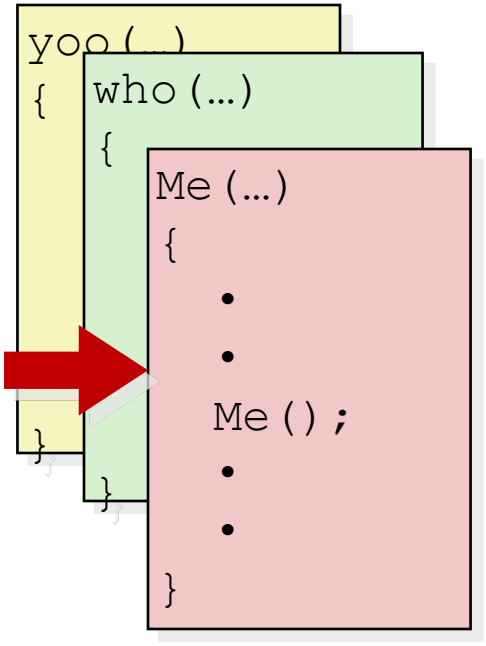
Example

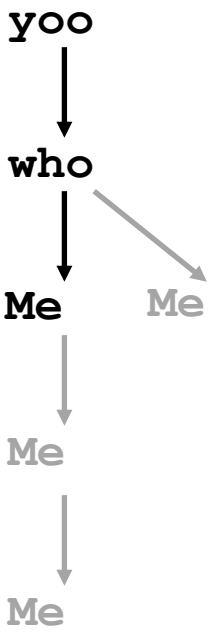


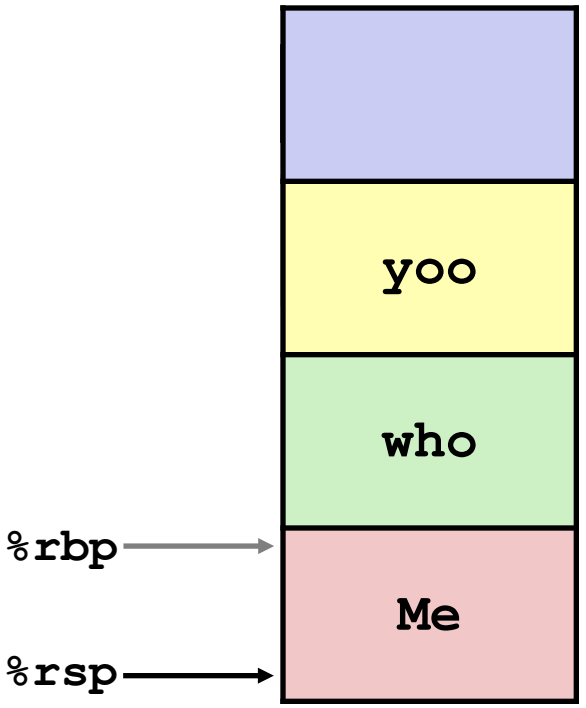




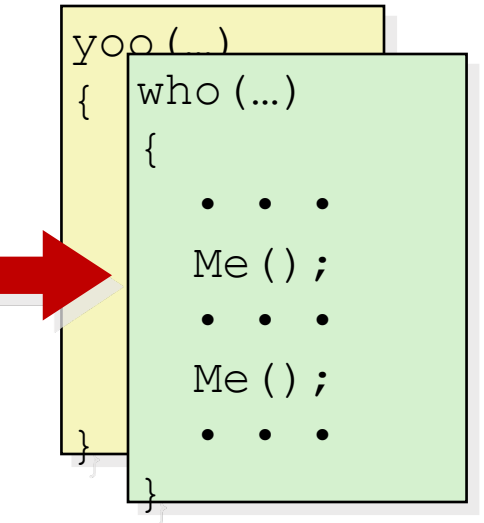
Example

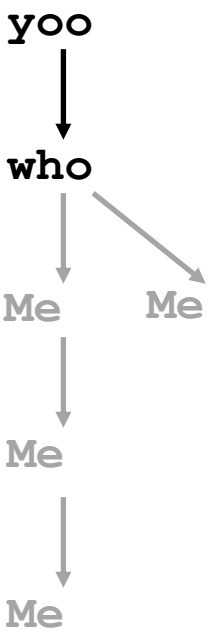


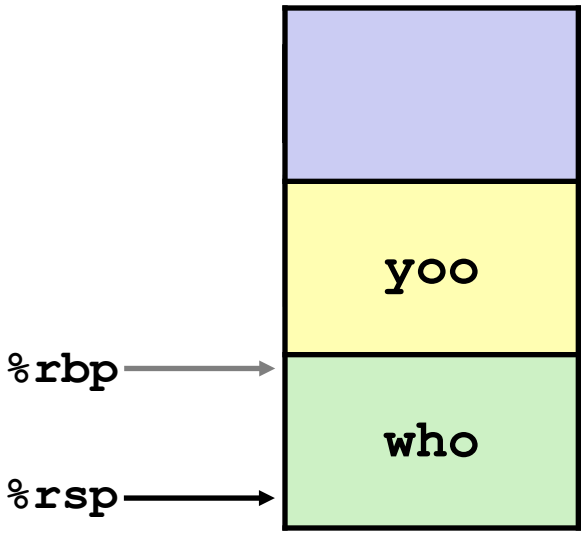




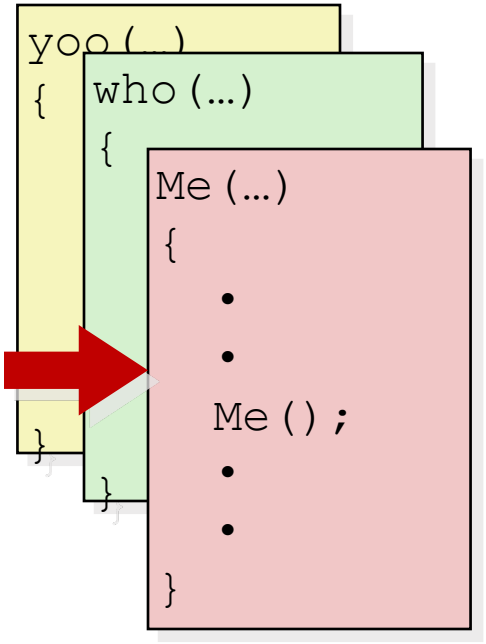
Example

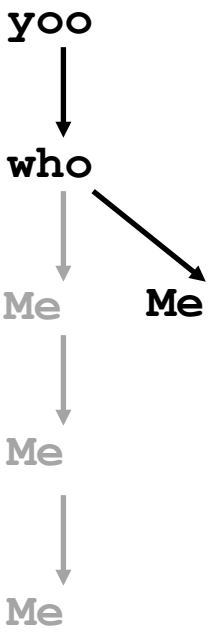


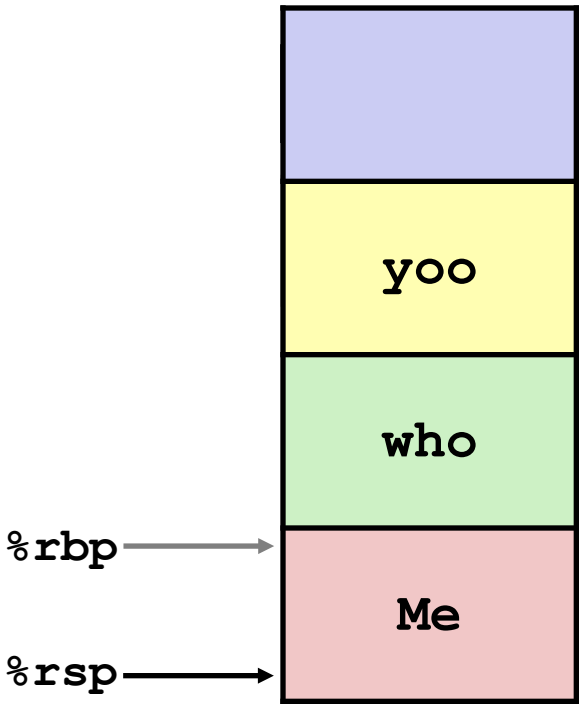




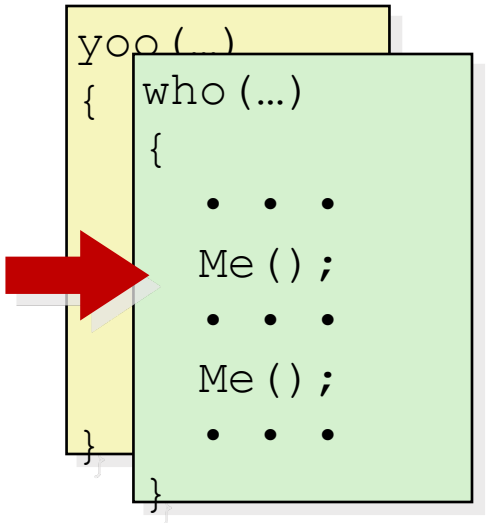
Example

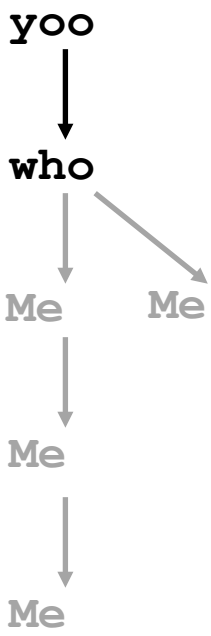


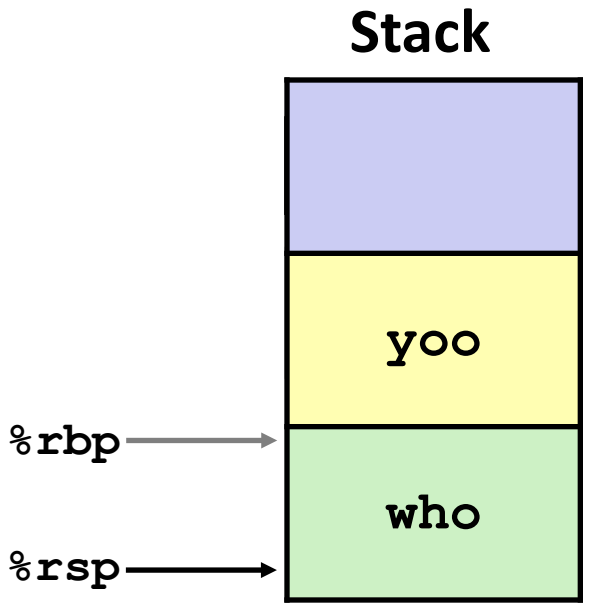




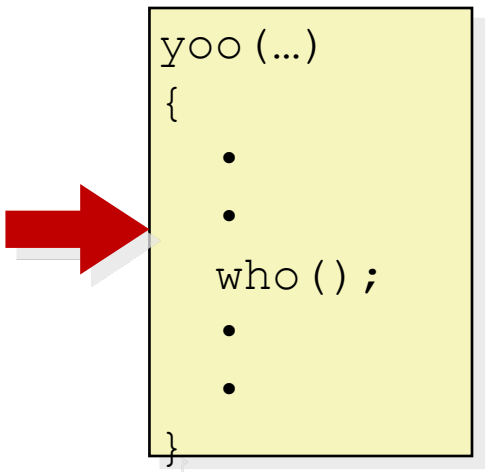
Example

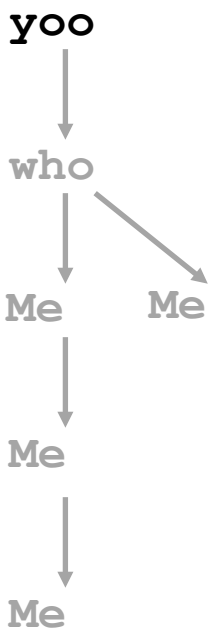


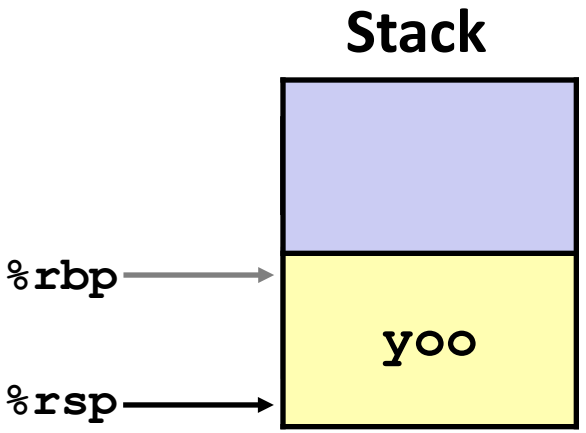




Example



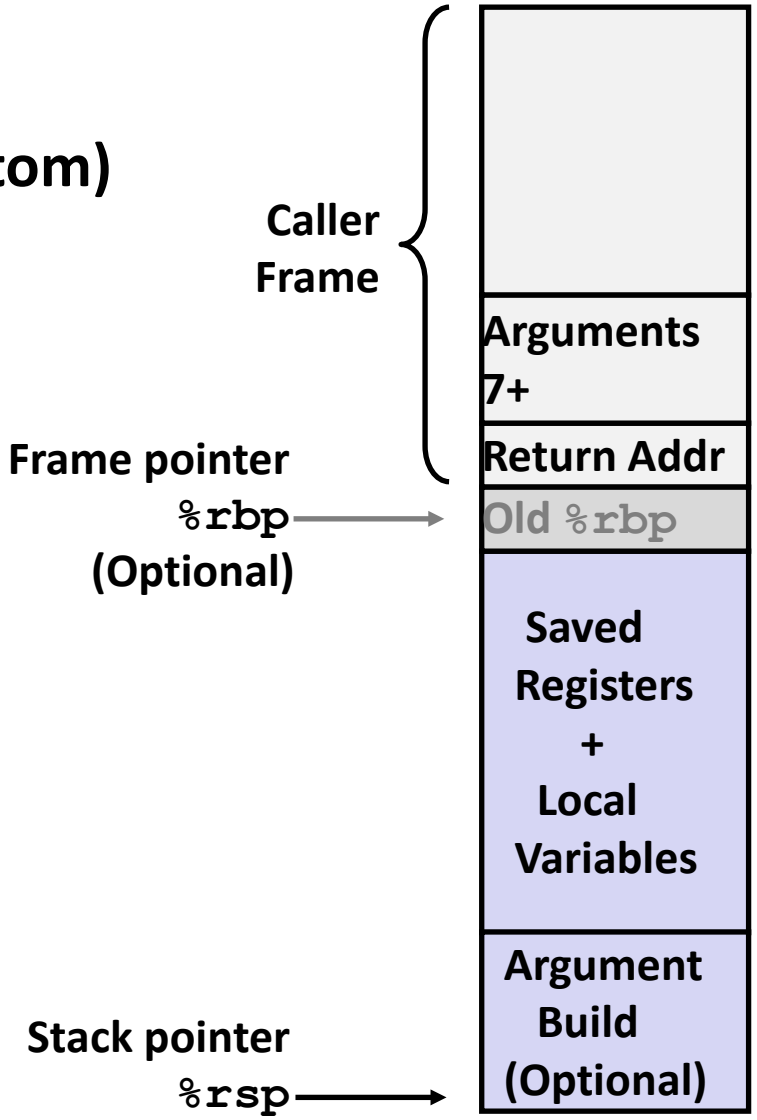




x86-64/Linux Stack Frame

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

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Example: `incr`

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

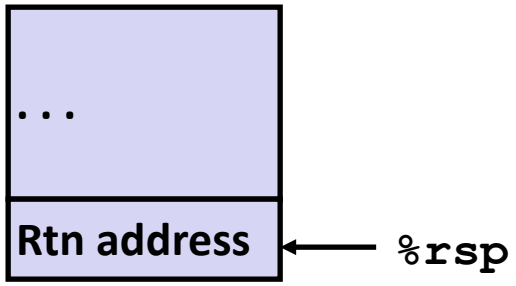
```
incr:
  movq     (%rdi), %rax
  addq     %rax, %rsi
  movq     %rsi, (%rdi)
  ret
```

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

Example: Calling `incr` #1

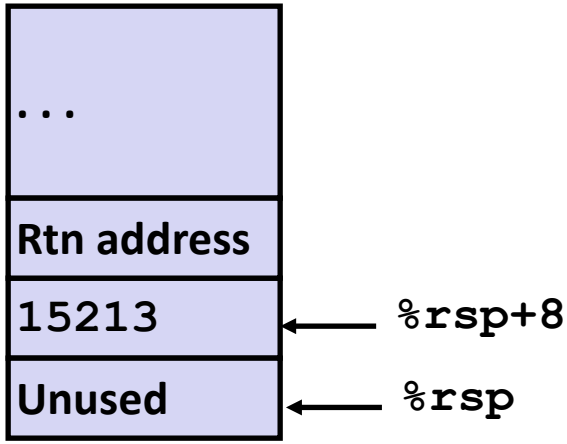
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure




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

Resulting Stack Structure

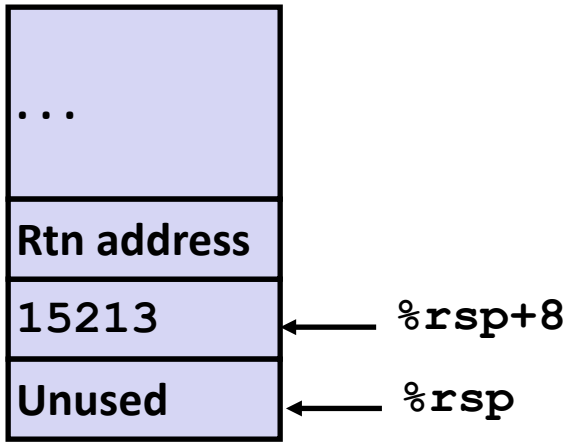


Example: Calling `incr` #2

```
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



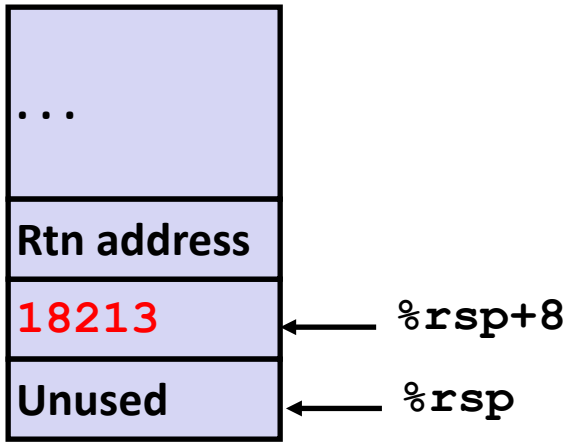
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #3

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

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

Stack Structure

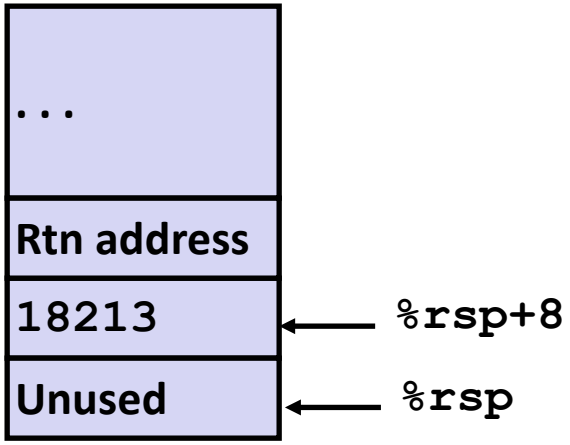


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

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

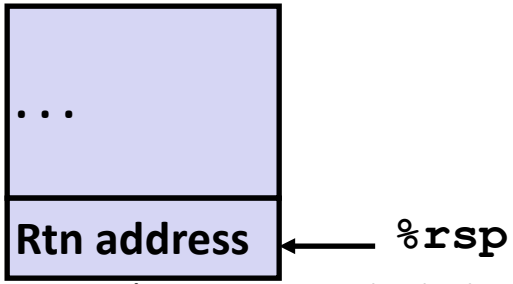
Stack Structure




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

Register	Use(s)
%rax	Return value

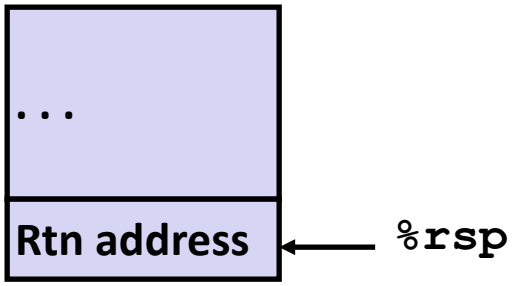
Updated Stack Structure



Example: Calling `incr` #5

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

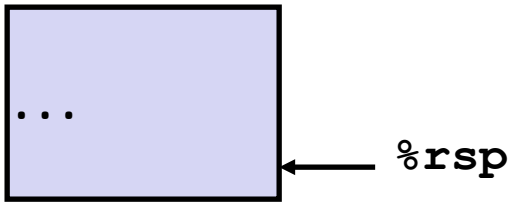
Updated Stack Structure




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

Register	Use(s)
%rax	Return value

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
```

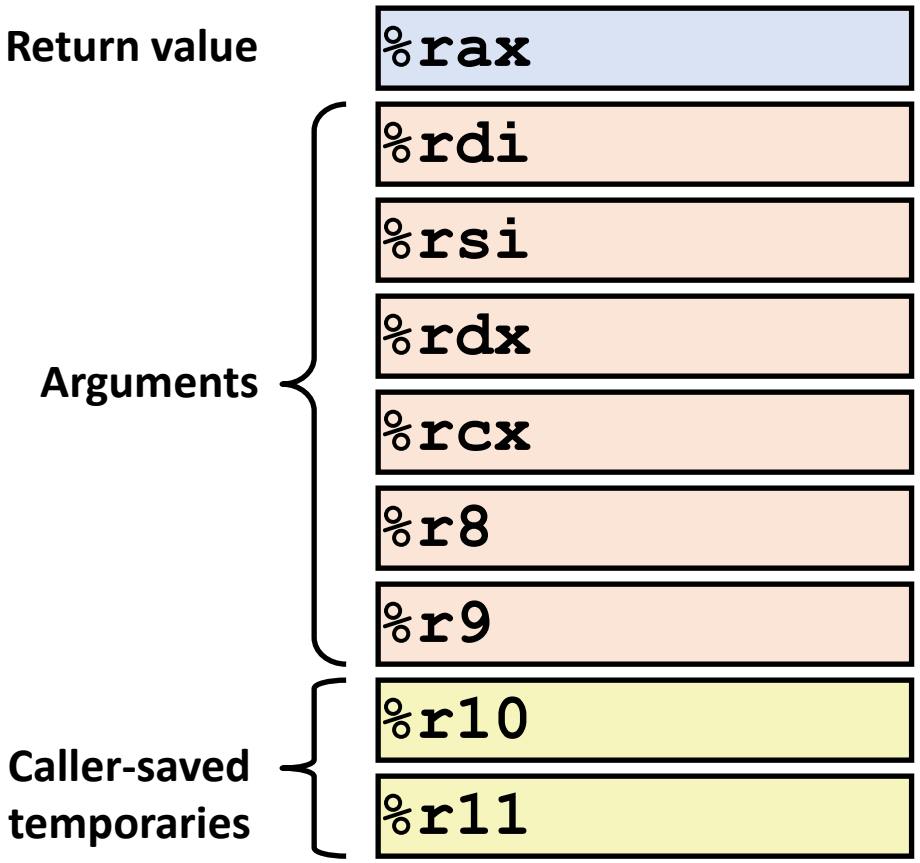
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble → something should be done!
    - Need some coordination

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"***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Saved"***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

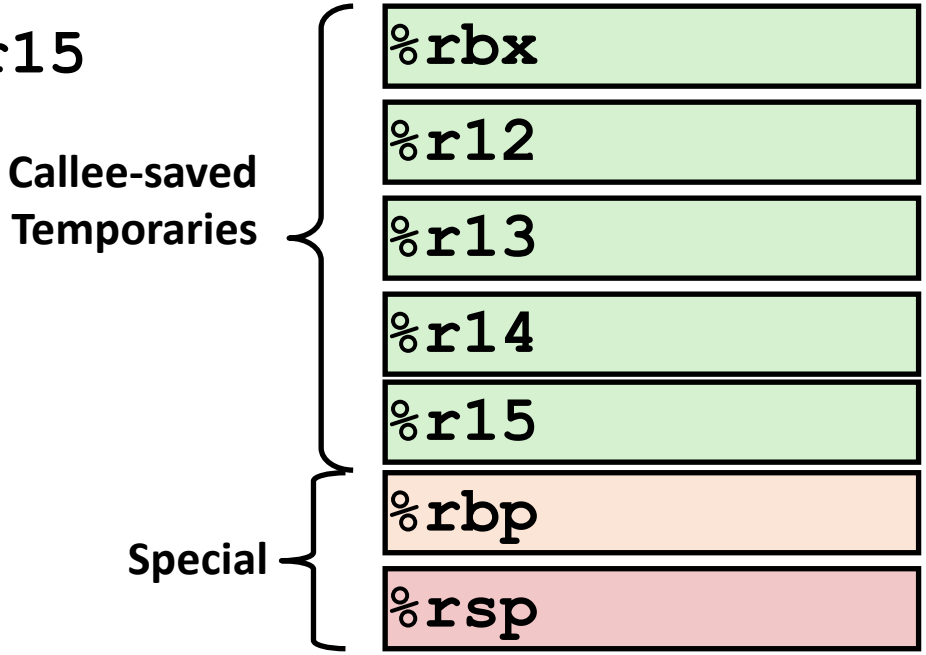
x86-64 Linux Register Usage #1

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



x86-64 Linux Register Usage #2

- `%rbx`, `%r12`, `%r13`, `%r14`, `%r15`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure

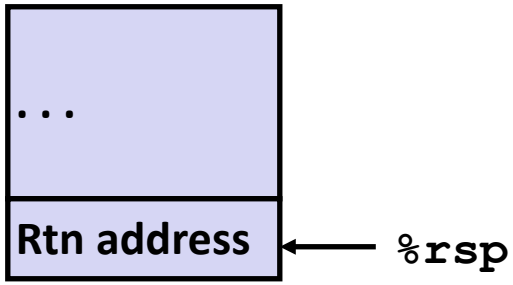


Callee-Saved Example #1

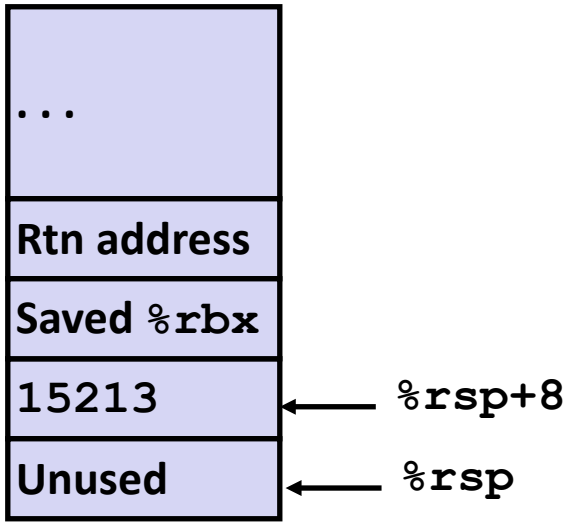
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

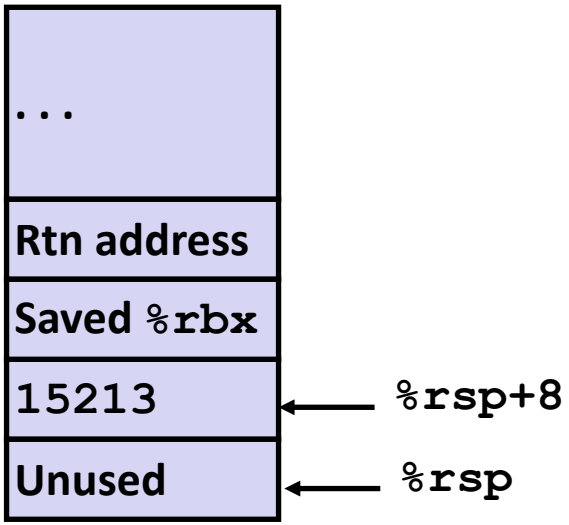


Callee-Saved Example #2

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

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure