Machine-Level Programming: Procedures

Prof. Hyuk-Yoon Kwon
https://sites.google.com/view/seoultech-bigdata

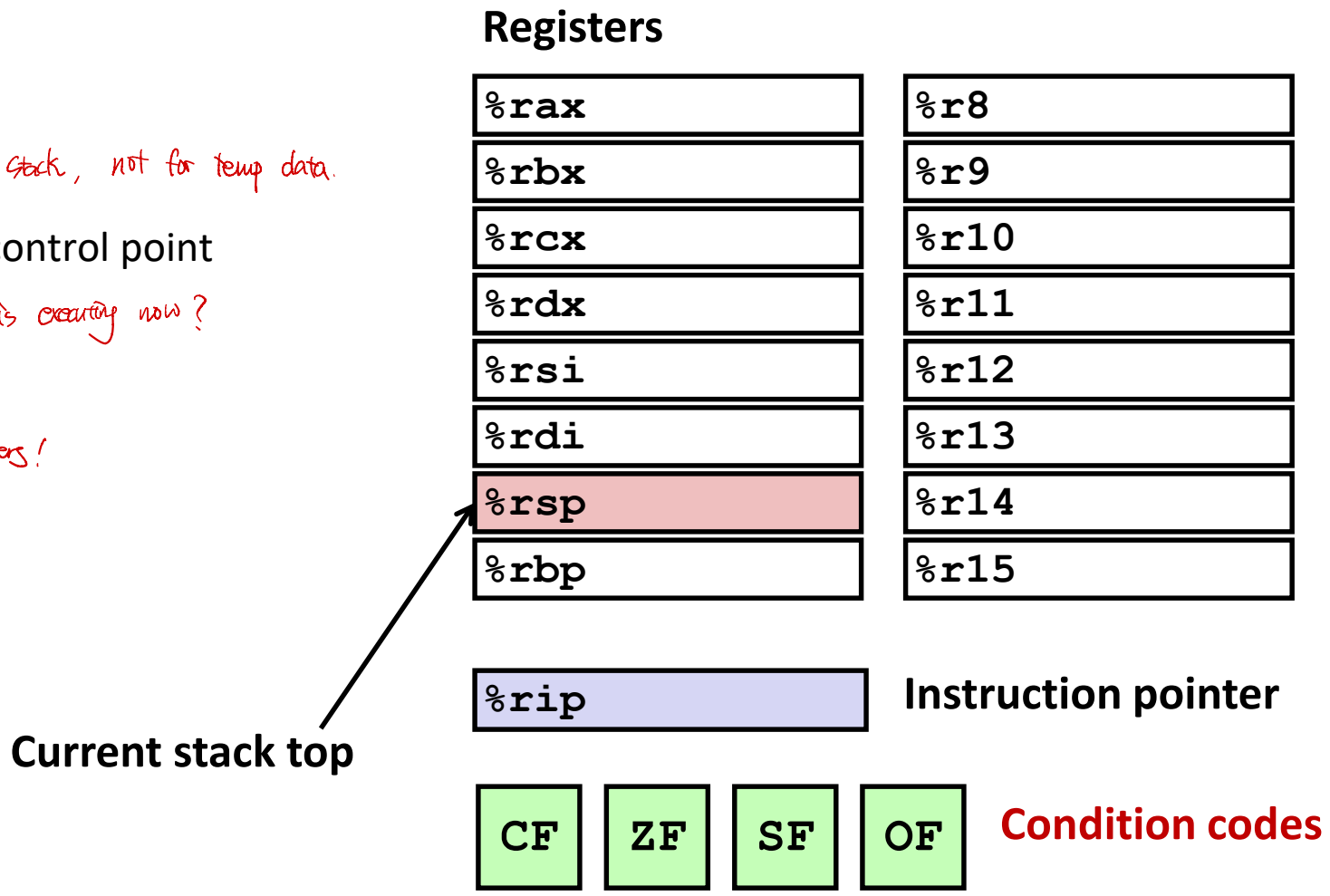
Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

Today

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

Processor State (x86-64, Partial)

- **Information about currently executing program**
  - Temporary data ... most registers!
    (`%rax`, …)
  - Location of runtime stack
    (`%rsp`) Current pointer into stack, not for temp data.
  - Location of current code control point
    (`%rip`, …) which instruction is executing now?
  - Status of recent tests
    (**CF, ZF, SF, OF**) ~> also registers!



Condition Codes (Implicit Setting)

condition codes are stored automatically, without explicit operations

Single bit registers

- **CF** Carry Flag (for unsigned) — mean 'overflow' when we handle unsigned
- **ZF** Zero Flag — result → Zero, ZF set

SF Sign Flag (for signed) — one evidence for check condition; sign bit set, → negative result, SF set

OF Overflow Flag (for signed)

Implicitly set (think of it as side effect) by arithmetic operations

Example: **`addq`** *Src*,*Dest* ↔ **`t = a+b`**

overflow

- **CF set** if carry out from most significant bit (unsigned overflow) ? how can find out ?
- **ZF set** if **`t == 0`**
- **SF set** if **`t < 0`** (as signed)
- **OF set** if two's-complement (signed) overflow

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

positive overflow — negative overflow

Not set by `leaq` instruction

not designed for arithmetic operation.
registers are not affected by 'lea'

Condition Codes (Explicit Setting: Compare)

instruction to set condition codes

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)
```

Condition Codes (Explicit Setting: Test)

Explicit Setting by Test instruction

- **`testq` *Src2*, *Src1***
  - **`testq b,a`** like computing **`a&b`** without setting destination

- Sets condition codes based on value of *Src1* & *Src2*

- **ZF set** when **`a&b == 0`** check equality
- **SF set** when **`a&b < 0`** Check one is less than the other.

Example:

```
testq  %rdx, %rdx
jle    .L3
```

→ ZF is not set
SF is depending on value of %rdx. (if %rdx < 0, SF is set
otherwise, SF is not set.

jump into a given label if the result is less than (SF) or equal to Zero (ZF)

check %rdx ≤ 0
if so, then jump to 'L3'
Compare given value & Zero

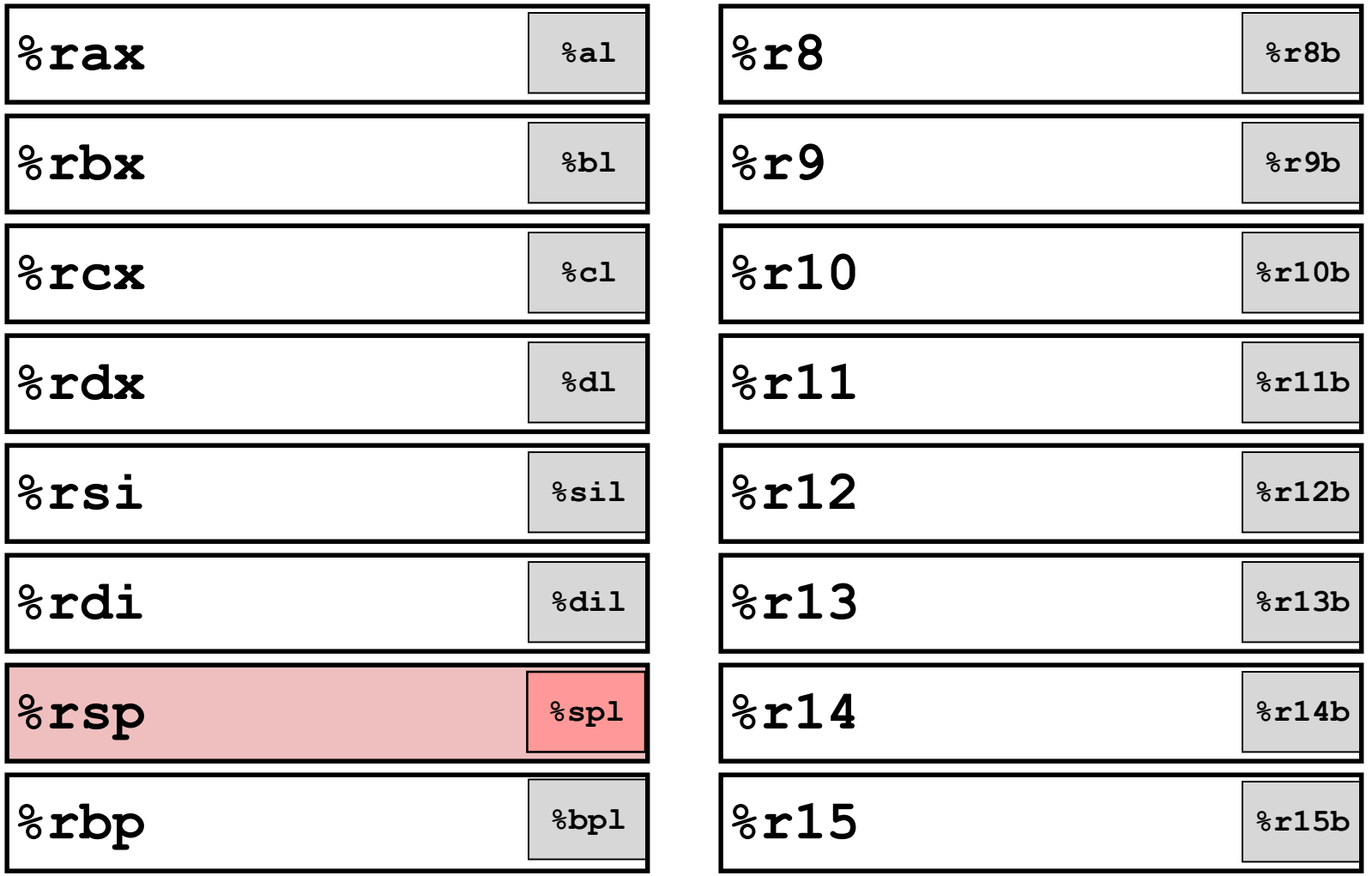
Reading Condition Codes

SetX Instructions

- Set the lowest byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

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

$2^8 - 1 = 255$ 1byte → 8 bit
$16^n - 1 = 255$ 4 bit

return 4 bytes

```
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 → (X-y)
setg    %al           # Set when >   X > Y
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Set one byte based on condition

%al 4 byte register

one byte from src to dst, rest of bytes remain zero

if %al = 1, X > y
else, X ≤ y

Today

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

Jumping

jX Instructions

set of jumping to specific part of code according to conditional codes

- Jump to different part of code depending on condition codes

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

Conditional Branch Example (Old Style)

- **Generation**

```
linux> gcc -Og -S -fno-if-conversion control.c
```

min optimization. get direct conversion of C into assembly, disable prediction of branch

predict cmp result → only keep codes to be executed removing un-used branch

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

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

(x-y) > 0 : %rax = %rdi ⋯ res = x ; %rax = %rax - %rsi ⋯ x = x-y

(x-y) ≤ 0 : %rax = %rsi ⋯ res = y ; %rax = %rax - %rdi ⋯ y = y-x

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

- GCC tries to use them
  - But, only when known to be safe

Why?

prevent advanced optimization techniques
∵ cannot execute upcoming codes continuously

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

expressed by one instruction that can keep original control flow

C Code

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

Goto Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
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

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

Calculate expression of each branch in advance.

conditional move instruction

(x-y) ≤ 0

Bad Cases for Conditional Move

secure the safety of code before using conditional move

Expensive Computations

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

- Both values get computed — regardless of condition.
- Only makes sense when computations are very simple

```
if ( )
  then ~
else if ( )
  then ~
else
  ~
```

Risky Computations

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

- Both values get computed
- May have undesirable effects — according to result of condition, in some case, then expression should not be executed.
  e.g., when 'p' is NULL

Computations with side effects

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

- Both values get computed — even if $x \leq 0$, $x *= 7$ is calculated.
  unexpected situation when compile
- 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;
}
```

if x=25, 0b11001 , -> return 3

- 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
```

X is unsigned
=> logical shift ok!

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

Practice: While Loop Example #1

- Insert printf() appropriately to monitor the changed “result” and “x”

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;
}
```

General "While" Translation #2

Compile result could be different depending on optimization option

While version

```
while (Test)
  Body
```

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

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:
```

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;
}
```

1st approach in condition

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

“For” Loop Form

General Form

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

```
#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;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

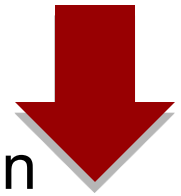
Body

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

“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;
  if (!(i < WSIZE))
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
  if (i < WSIZE)
    goto loop;
 done:
  return result;
}
```

Init

! Test

Body

Update

Test

compiler will remove this part,
i=0, WSIZE=8.
So, first test is always false.
=> optimization!

Today

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

Switch Statement Example

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

```
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;
}
```

Jump Table Structure

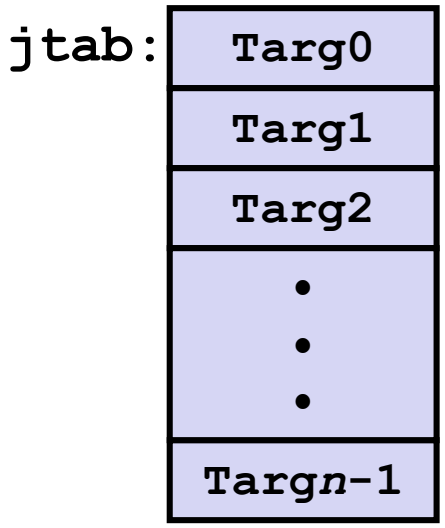
Switch Form

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

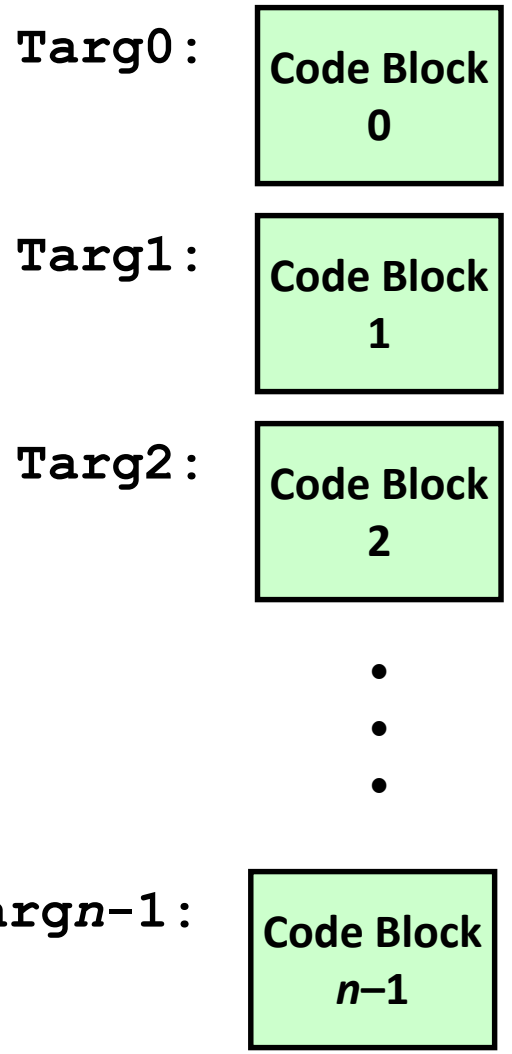
Translation (Extended C)

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

Jump Table



Jump Targets



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? $(x-6) > 0$

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]
```

what if %rdi < 0 ?

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 $0 \le x \le 6$

pointer in 64-bit

offset from the starting

cf) p.39 ④

Jump Table

Jump table

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

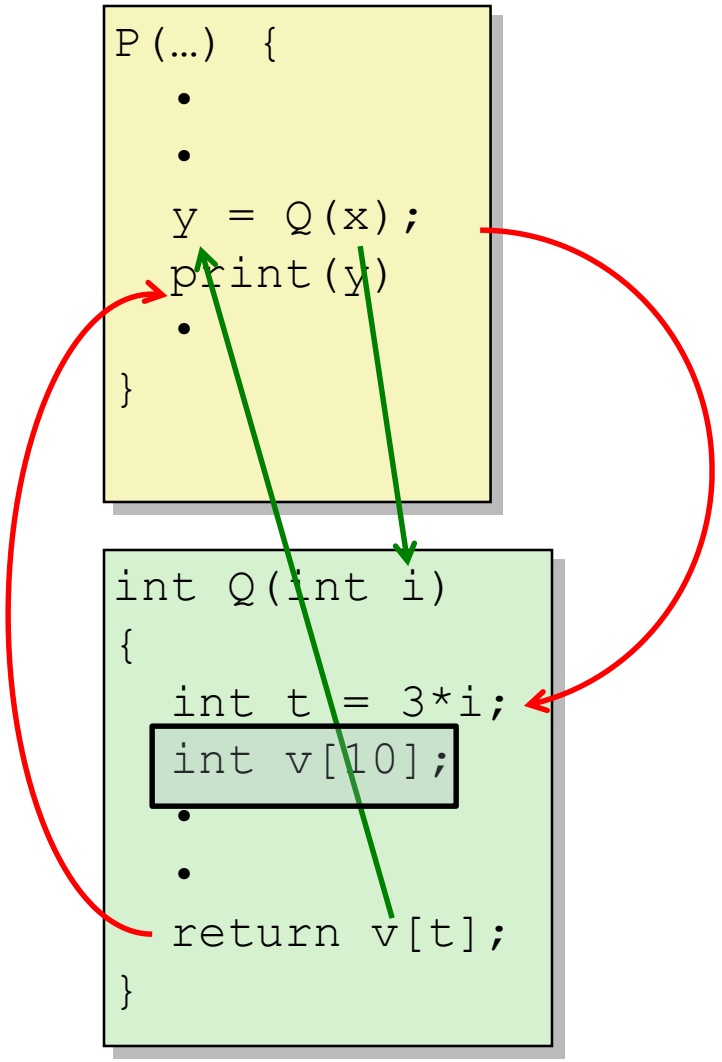
Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	Return value

Today

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

Mechanisms in Procedures

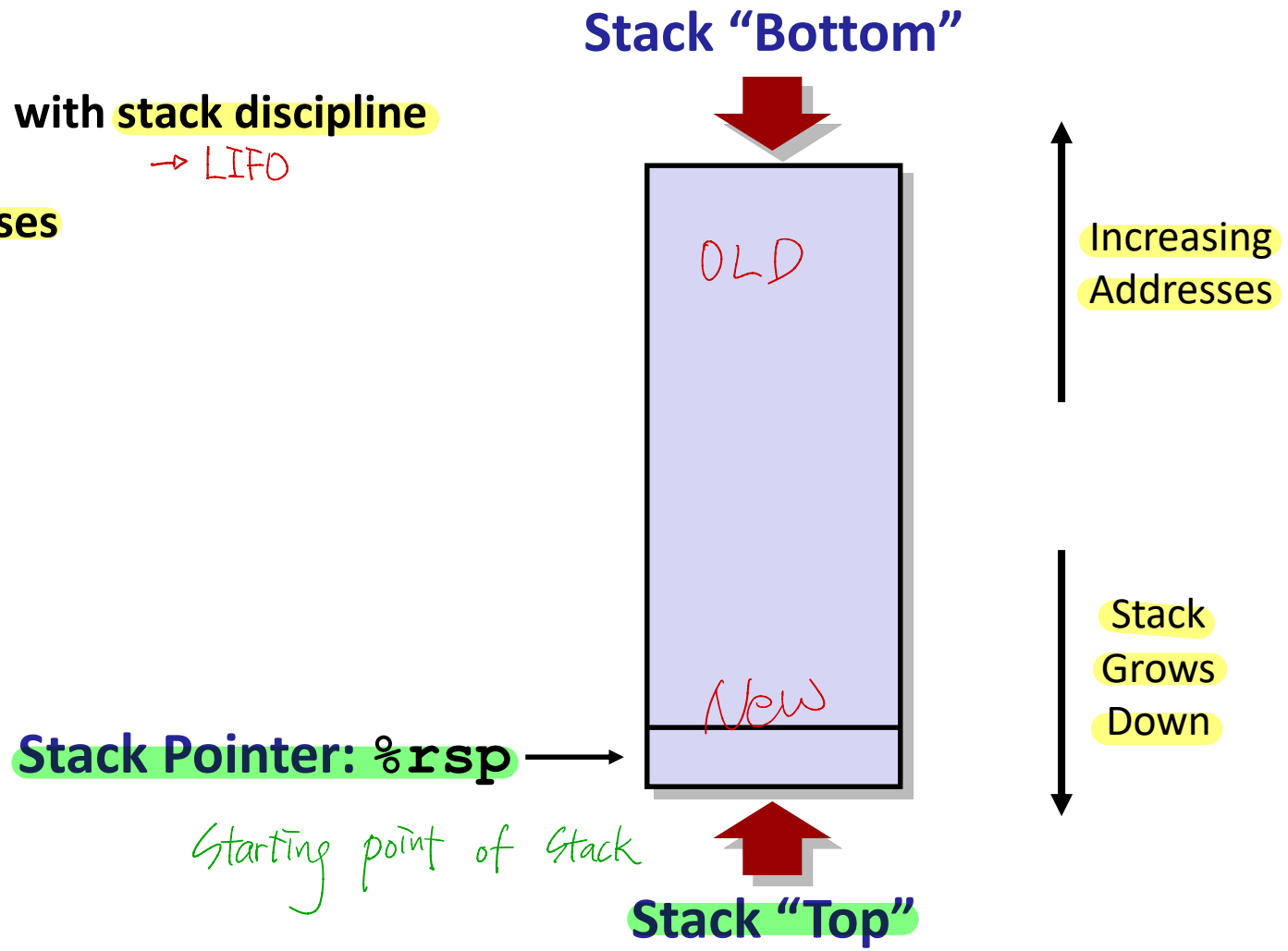
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**



x86-64 Stack

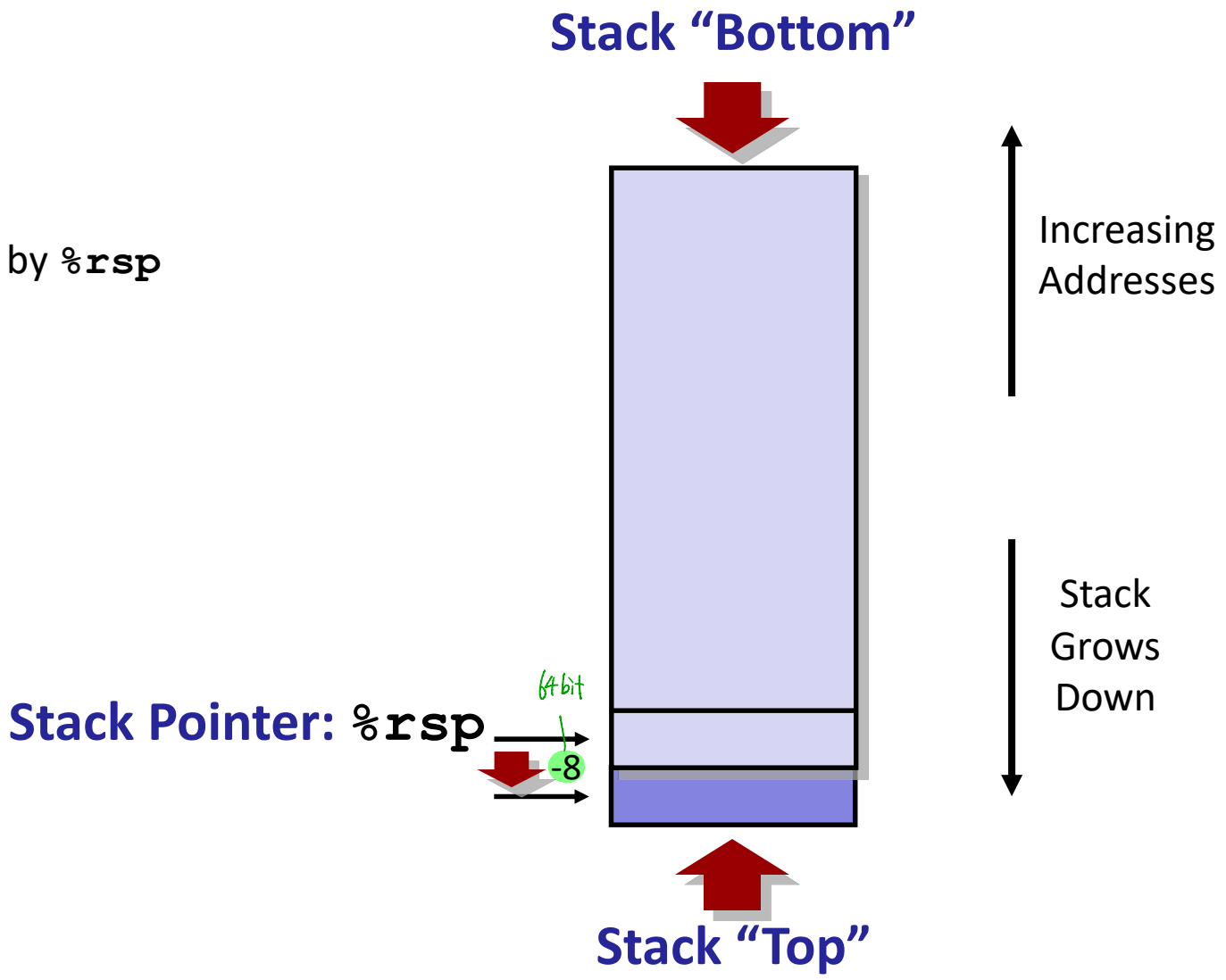
- Region of memory managed with stack discipline
  → LIFO
- Grows toward lower addresses
- Register `%rsp` contains

  lowest stack address
  - address of "top" element



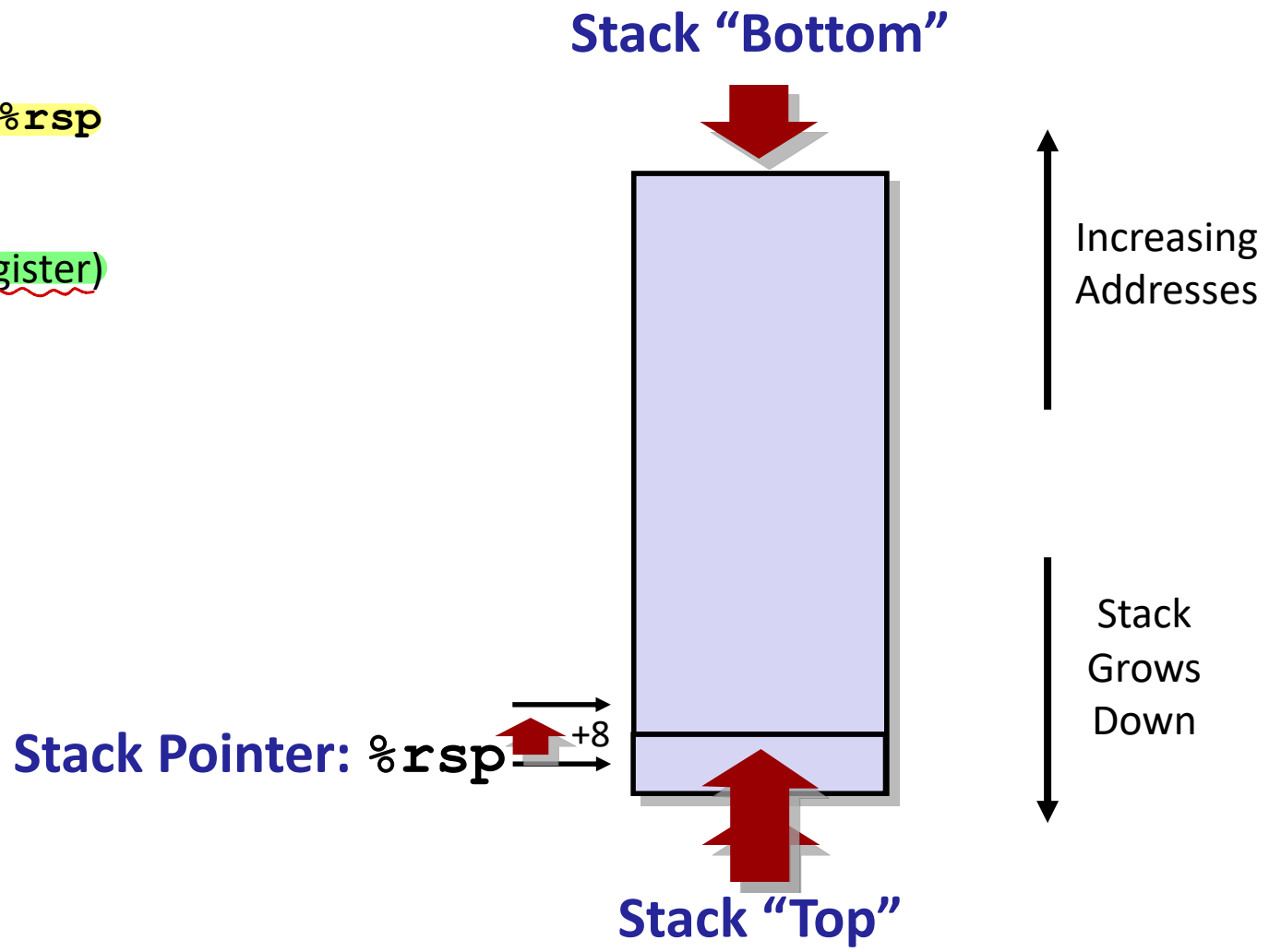
x86-64 Stack: Push

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



x86-64 Stack: Pop

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



Today

- **Procedures**
  - 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;
}
```

Call procedure, 'mult2'

```
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
```

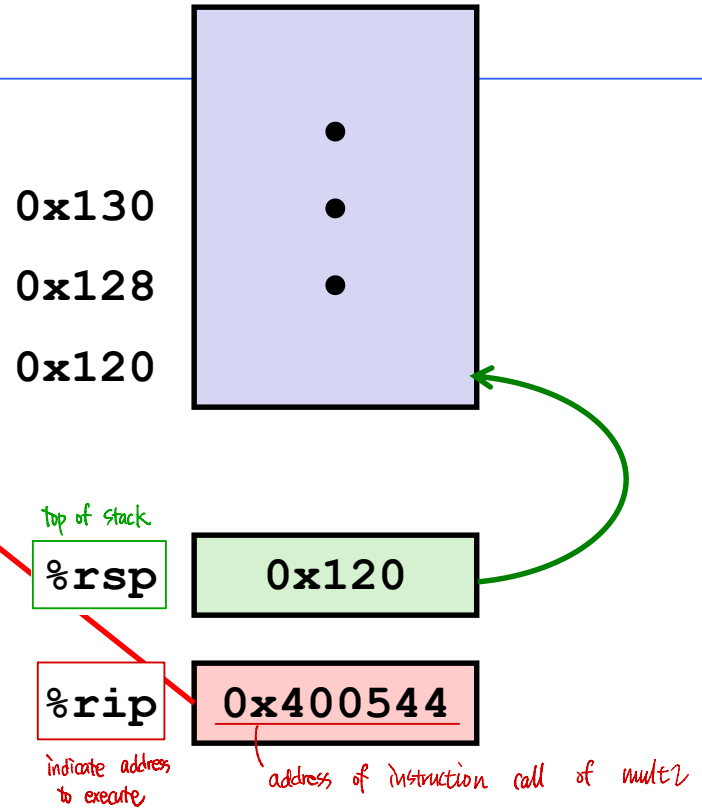
Procedure Control Flow

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

Control Flow Example #1

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

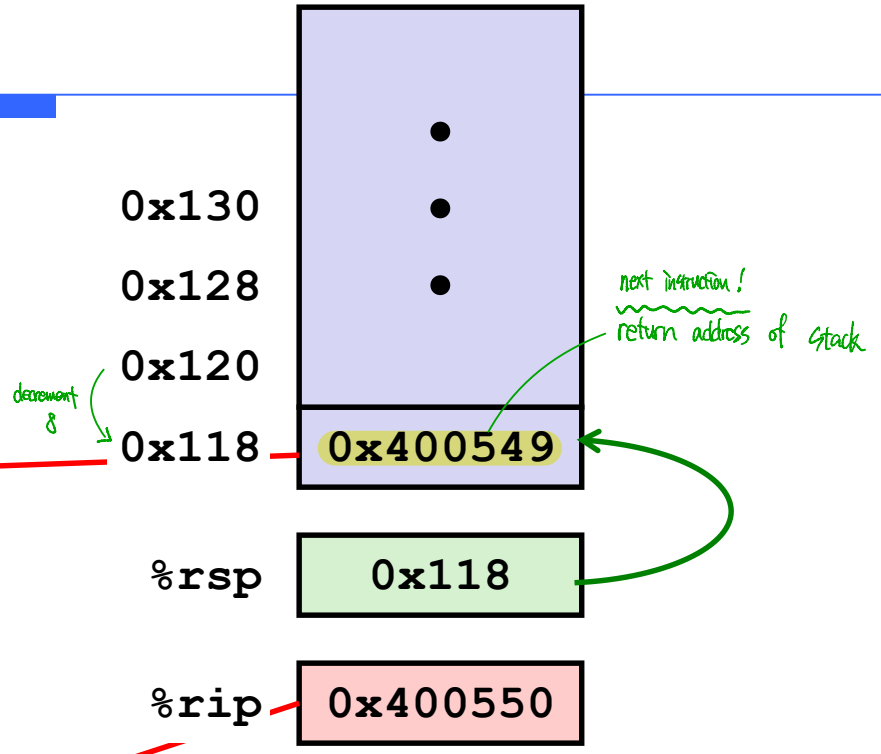
```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  retq
```



Control Flow Example #2

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

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

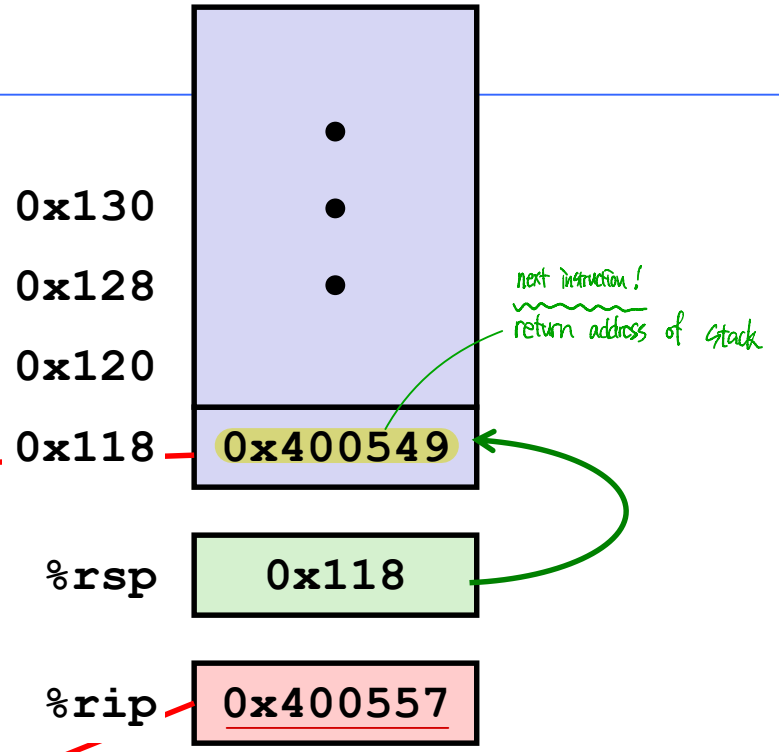


Control Flow Example #3

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

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

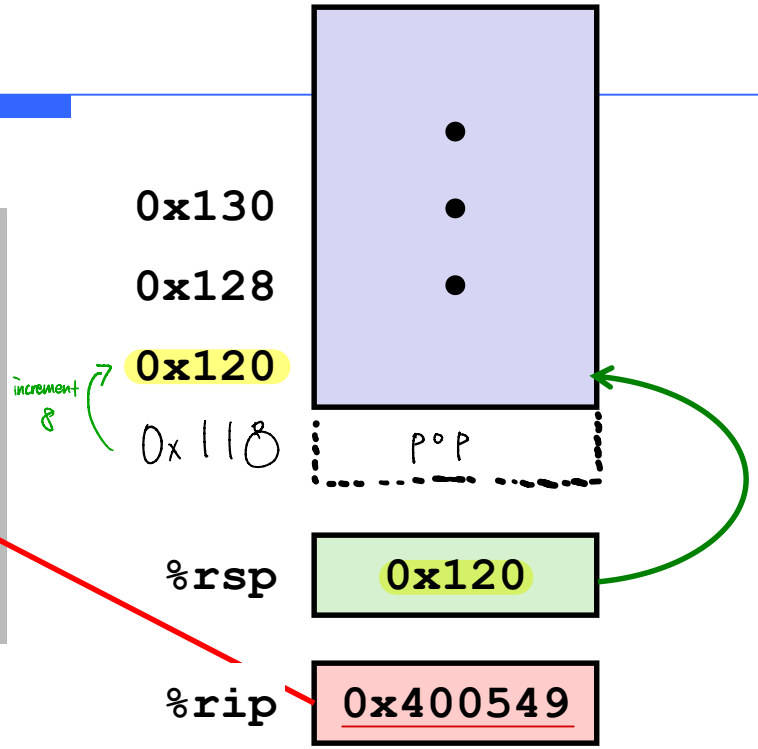
return to original procedure



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**
    - Managing local data
  - Illustrations of Recursion & Pointers

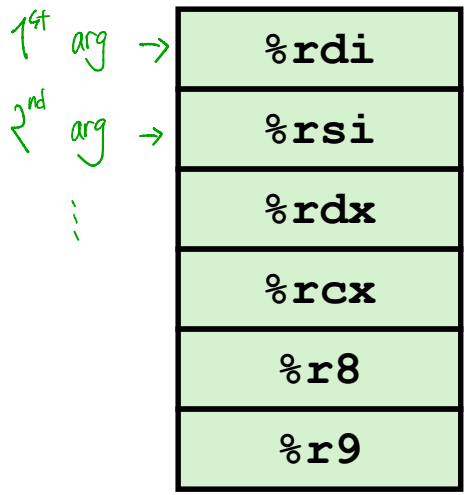
Procedure Data Flow

how to pass the data? Use register & stack

First 6 arguments

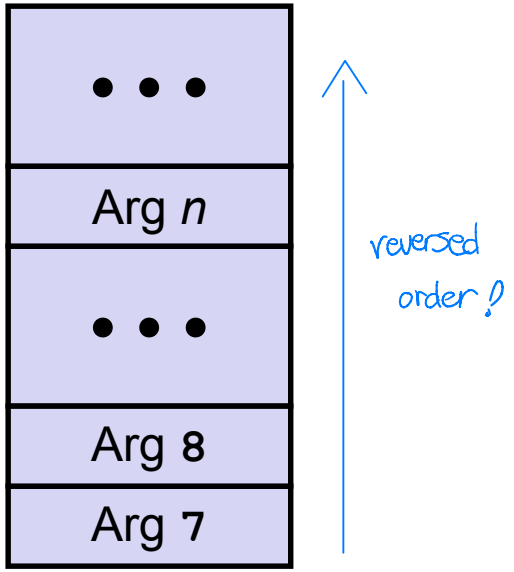
- **Registers**

in the procedures, it uses value stored in the register to access parameter.

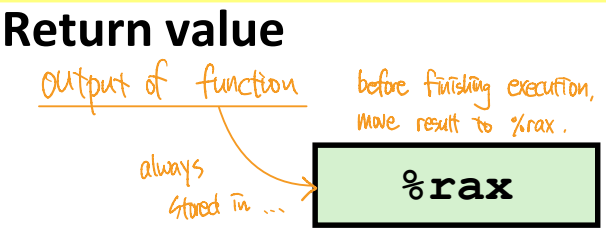


- Store argument before calling the procedure.
- reserved to store argument
- specific order to store!

- **Stack**



Return value



original procedure checks the result of called function's result by referring %rax.

what if 6+ arguments in the procedure?

finite registers ... 6!

☆(7th, ...) args are stored in stack in reversed order!

how to use: pop argument from the stack

- **Only allocate stack space when needed**

only register: finite # of registers

only stack: more cost to process

⇒ use both

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
```

Practice: Passing Control and Data

- **Generate the assembly codes for the following two C codes and understand the passing control and passing data.**
  - Use –Og option: gcc –S –Og mult2.c

less optimization

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

```
long mult2
   (long a, long b)
{
   long s = a * b;
   return s;
}
void multstore
 (long x, long y, long *dest) {
    long p = x + 1; %rdi
    long q = y + 2; %rsi
    long t = mult2(p, q);
    t = t + 3;  %rax
    *dest = t;
}
```

Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - **Managing local data** → how to store variables
  - Illustration of Recursion

Stack-Based Languages

Languages that support recursion

- e.g., C, Pascal, Java
- Code must be "*Reentrant*" recursive call for the same procedure.
  - Multiple simultaneous instantiations of single procedure

A calls B
C calls B
} simultaneously ... maintain each B's own state.
different call, different value.
=> store info separately in stack

- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

} stored in stack to maintain its own status with different region.

Stack discipline

- State for given procedure needed for limited time
  - From when called to when return

call -> push info into stack
return -> pop info from stack

- Callee returns before caller does => separate region for callee & caller

maintain info separately by diff region in stack

procedure which is called

OE: Procedures which call other procedures

Stack allocated in ***Frames***

- state for single procedure instantiation

Stack consist of frames
assigned for each procedure.
info for each procedure in one frame respectively

Call Chain Example

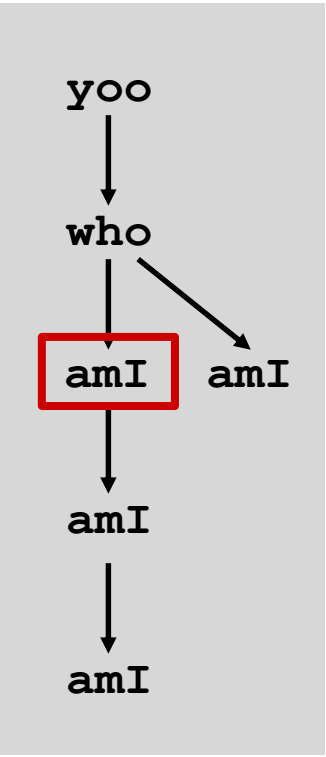
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

Procedure `amI()` is recursive

Example Call Chain



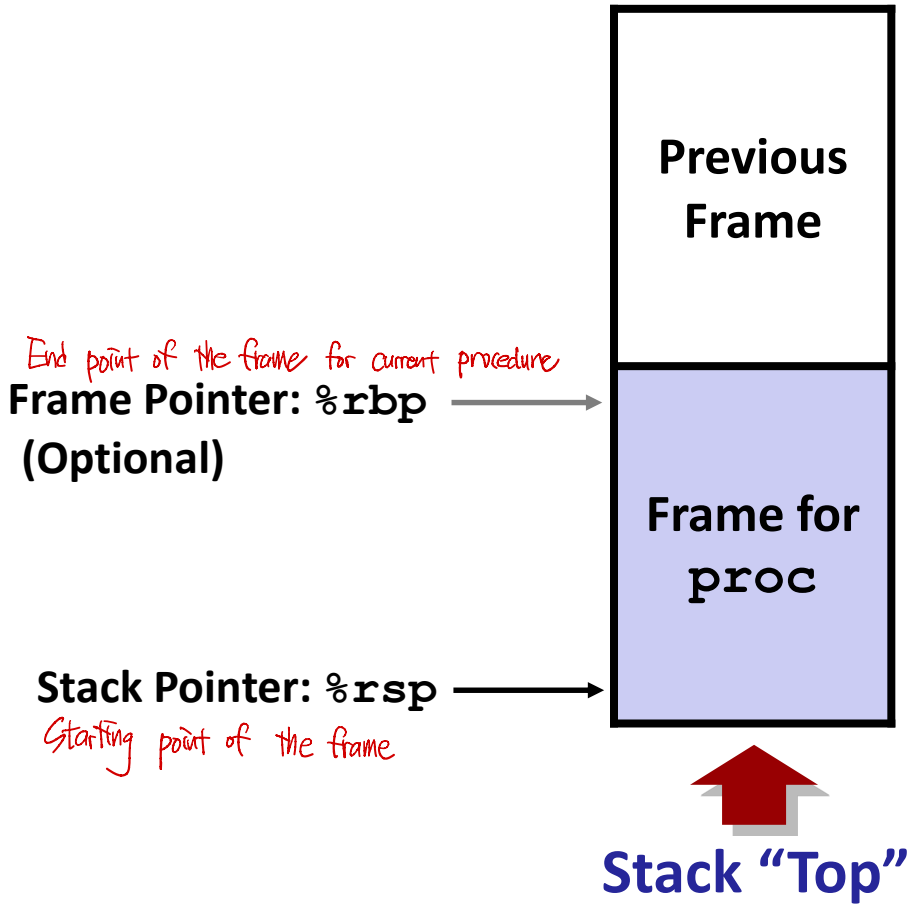
Stack Frames

Contents

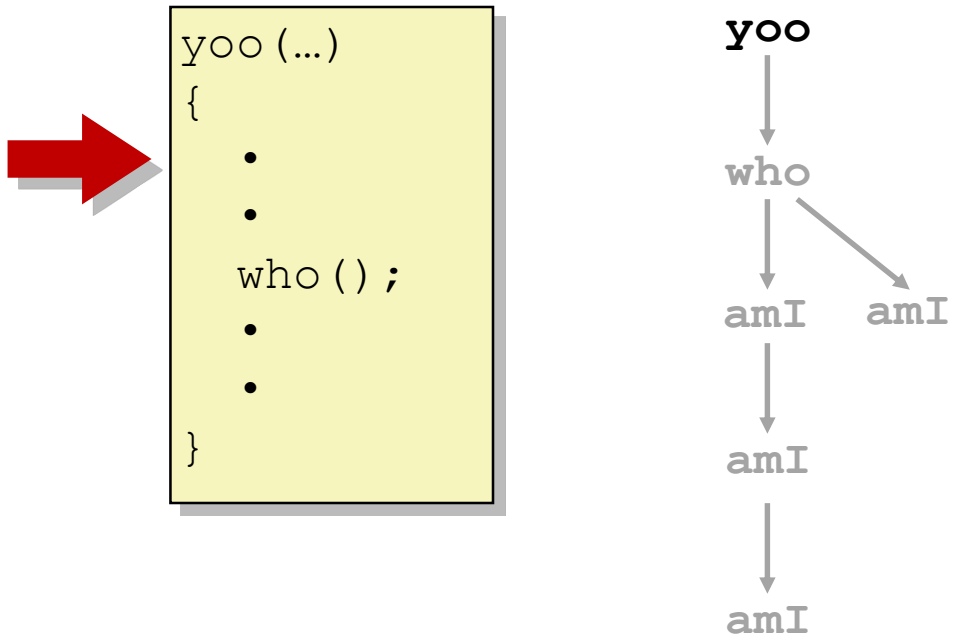
- Return information a.k.a. return address
- Local storage (if needed) for variables
- Temporary space (if needed)

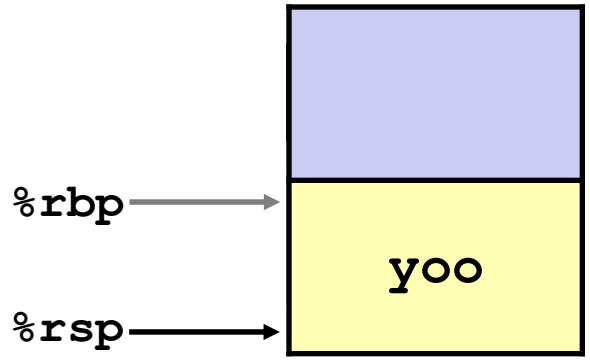
Management

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

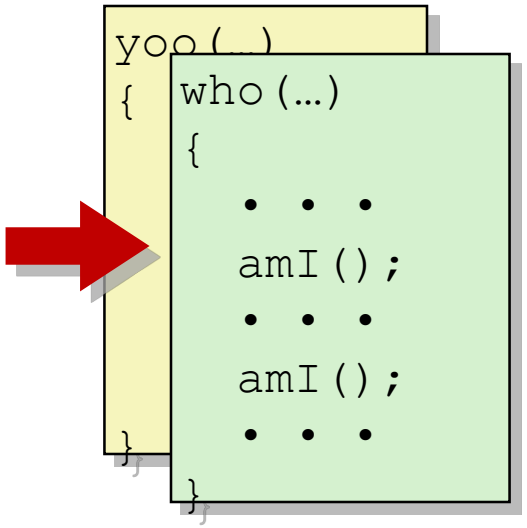


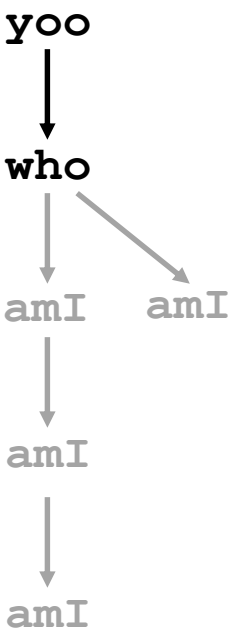
Example

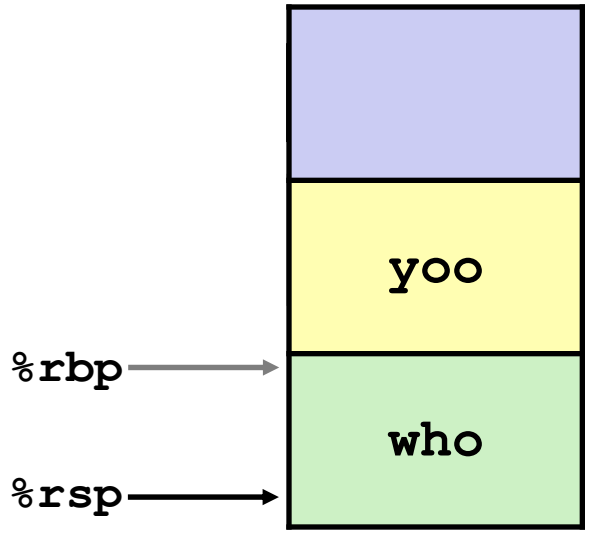




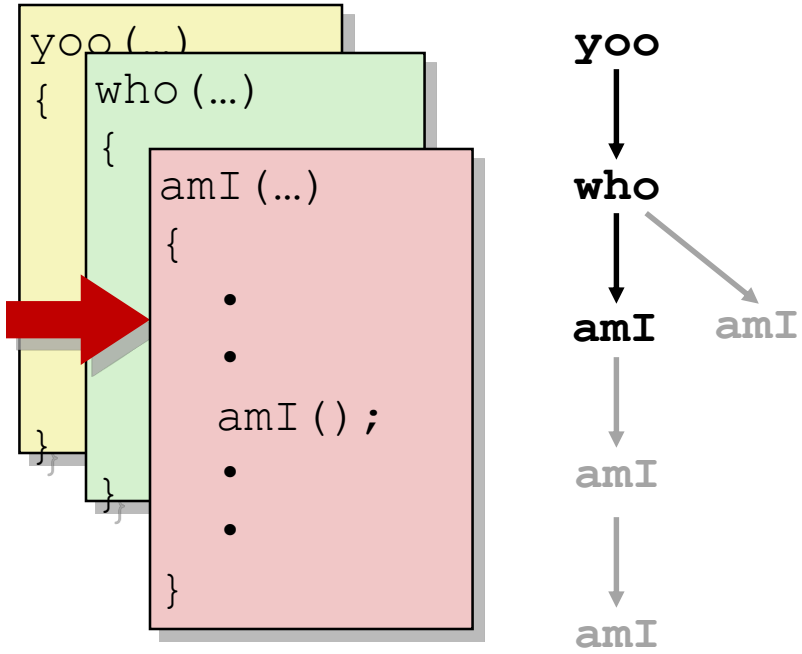
Example



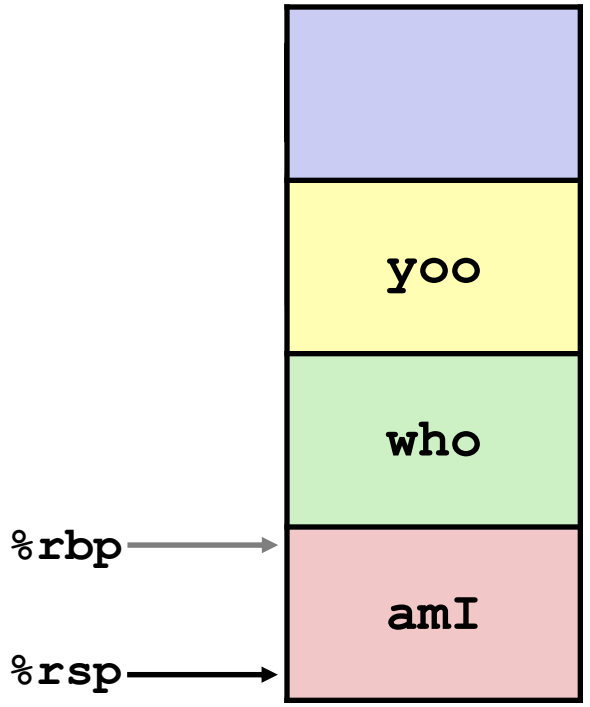




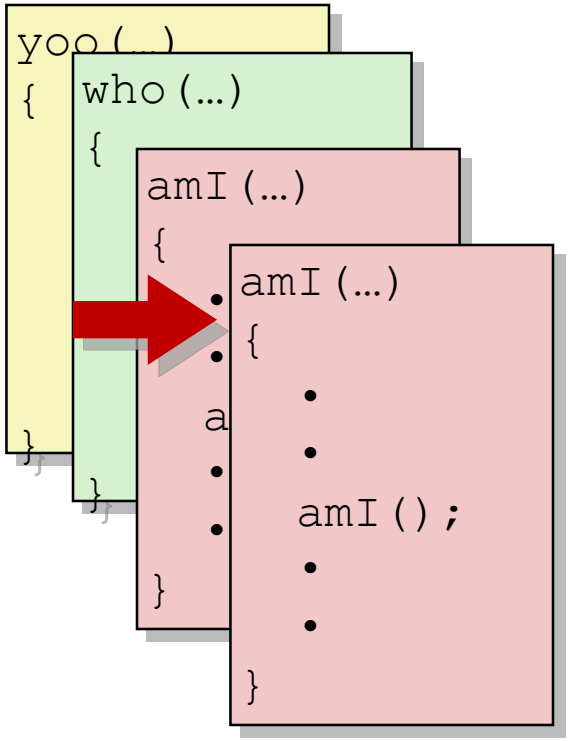
Example

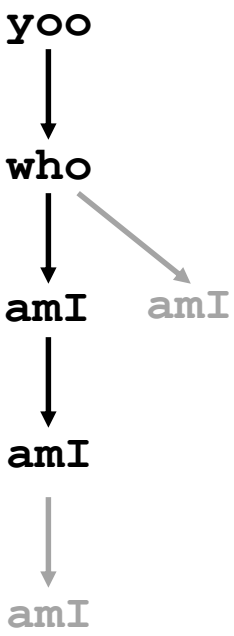

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

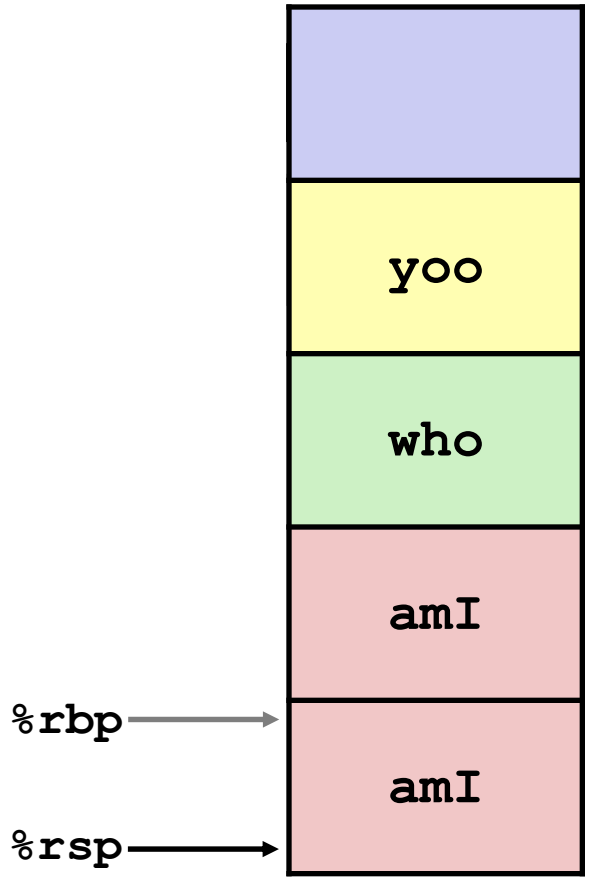
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

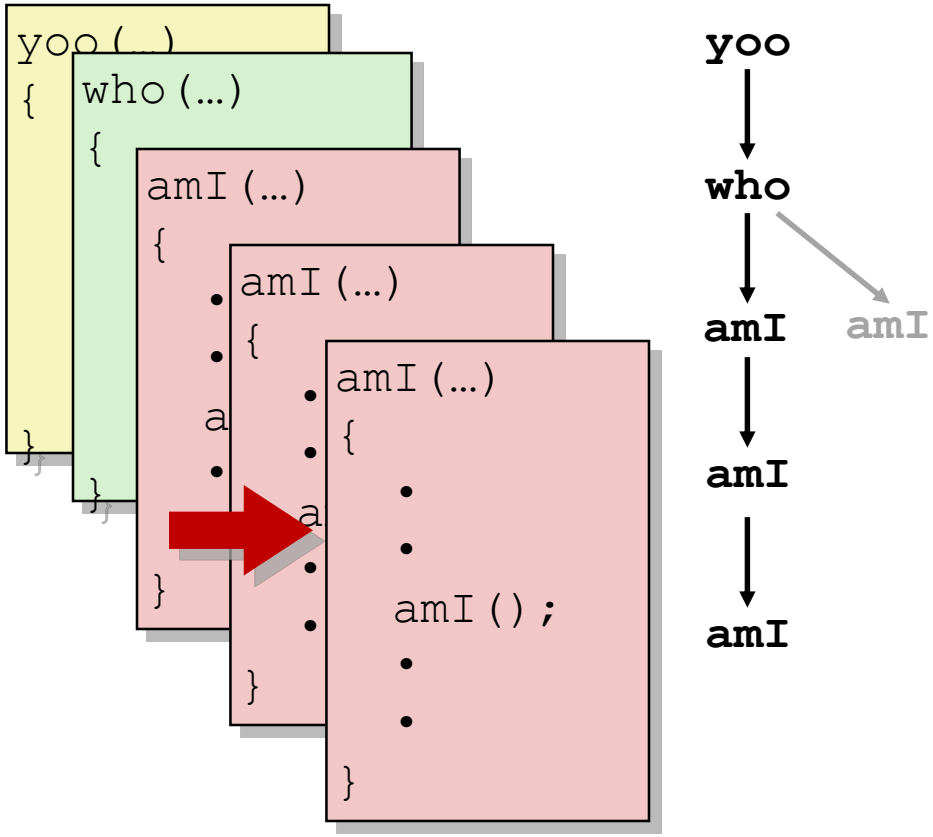
Example


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

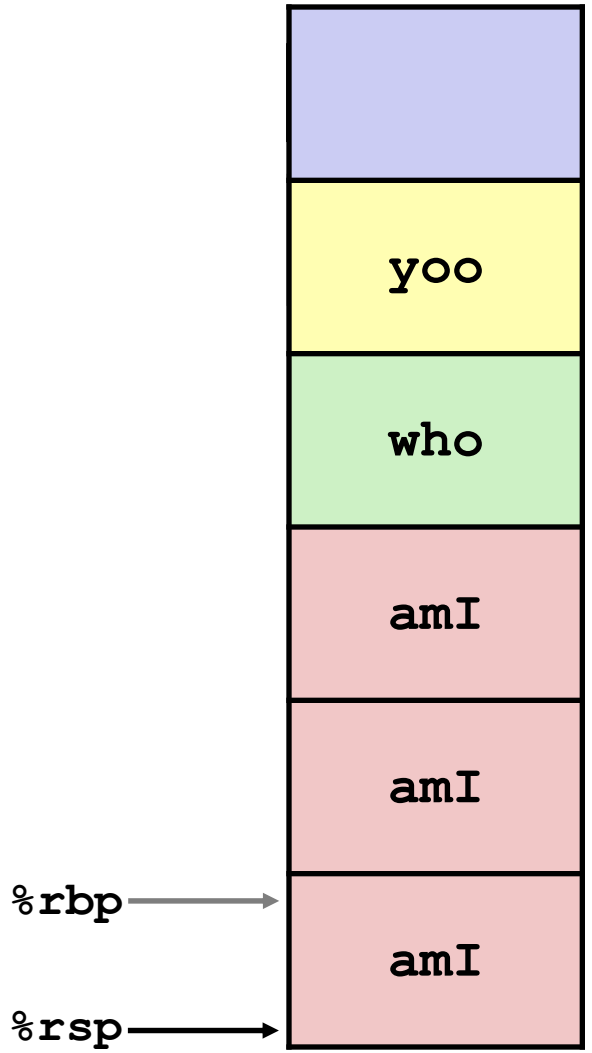

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

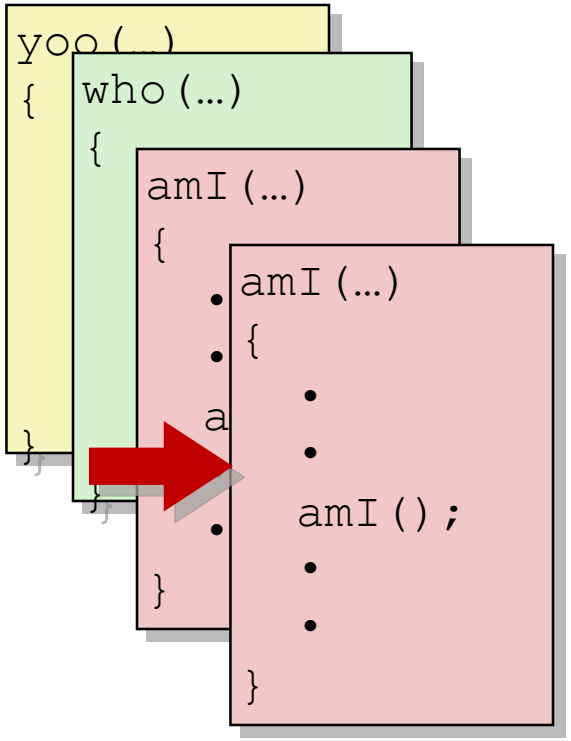
Example

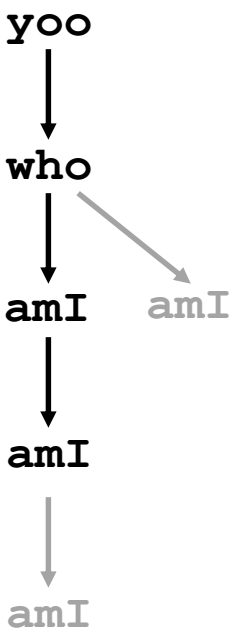


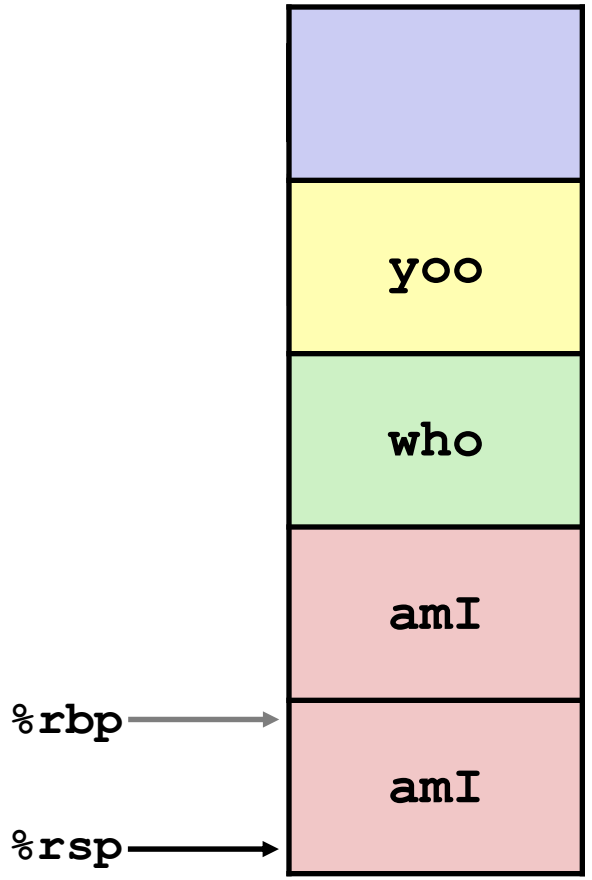
Stack



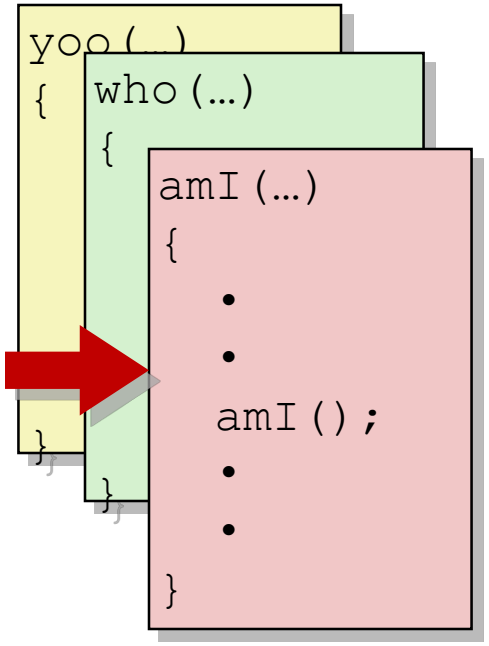
Example

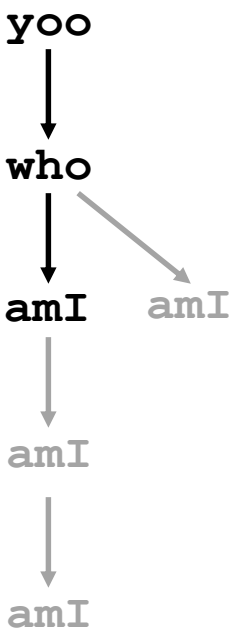

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

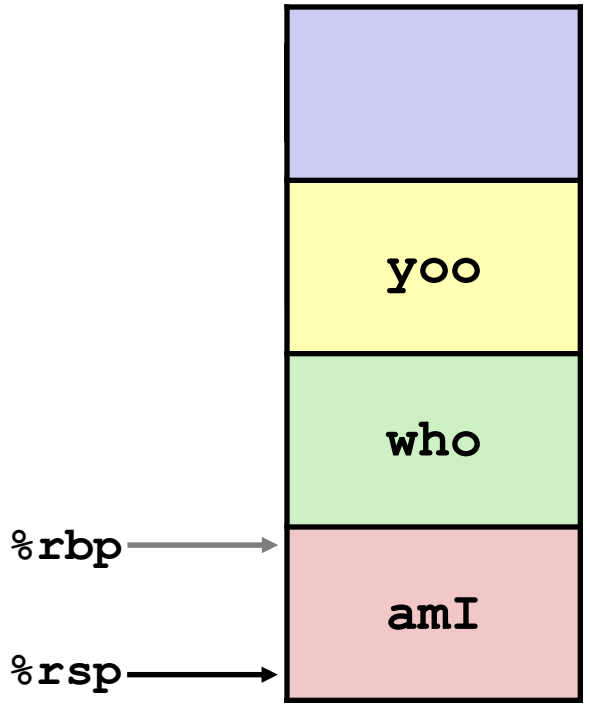

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

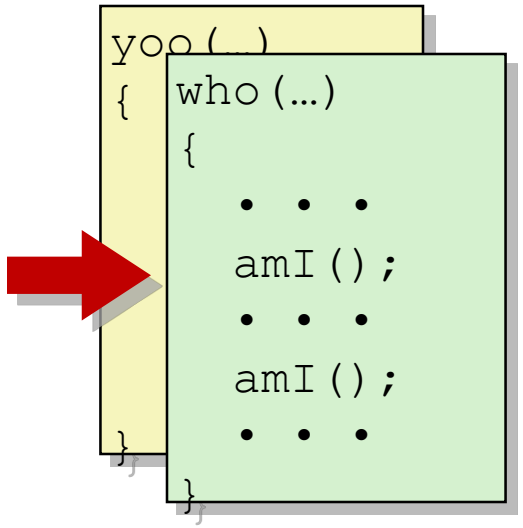
Example

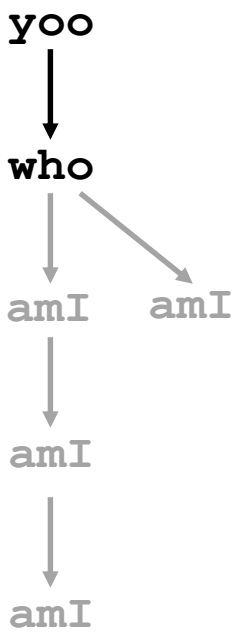

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

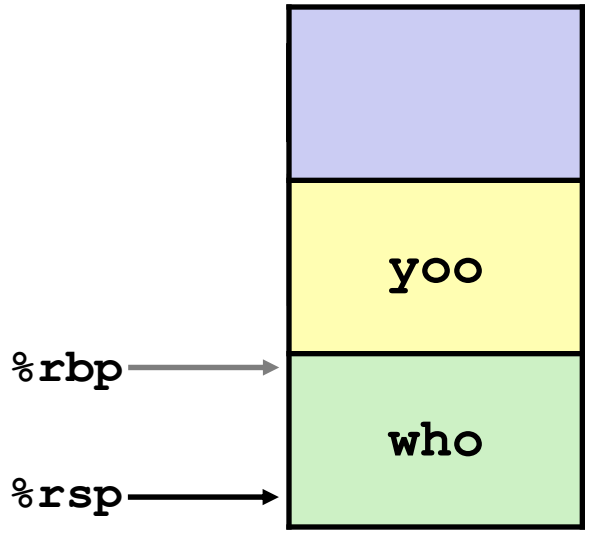

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

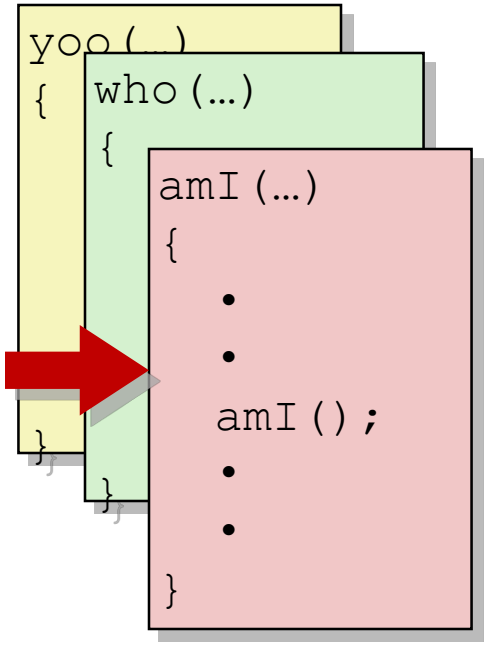
Example

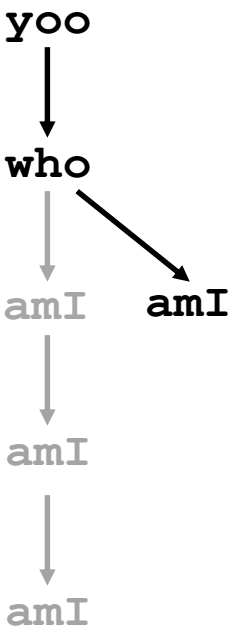


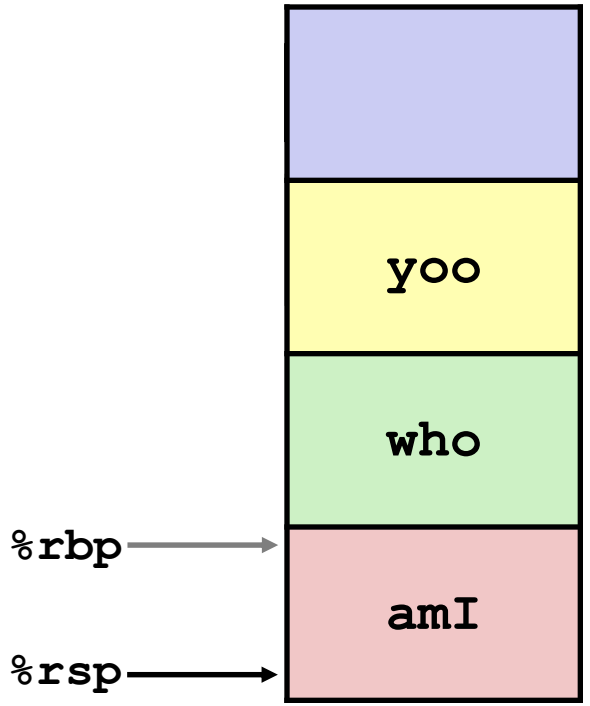




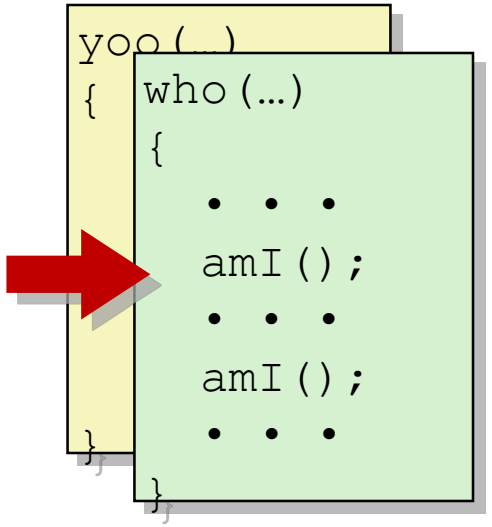
Example

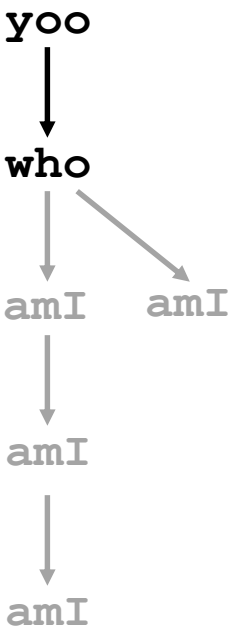

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

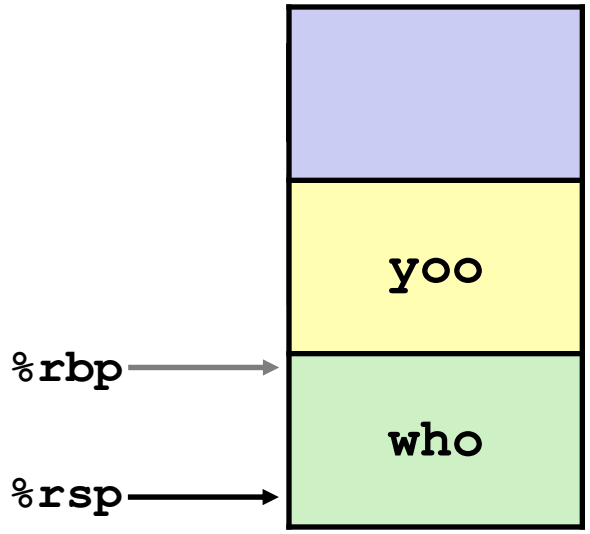

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

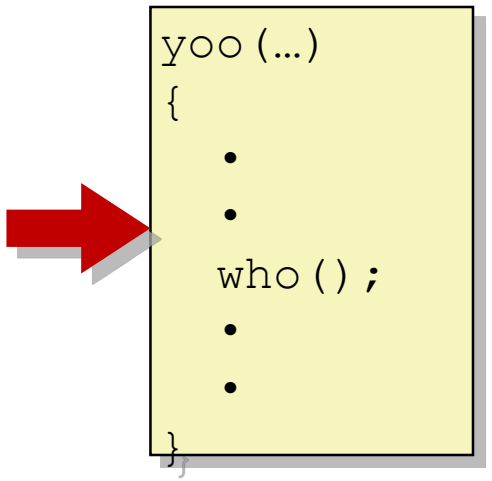
Example

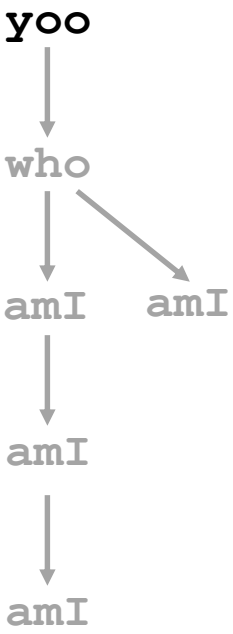


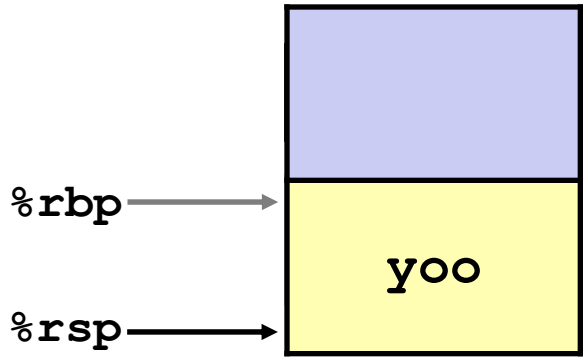




Example







x86-64/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- "Argument build:"

  Parameters for function about to call

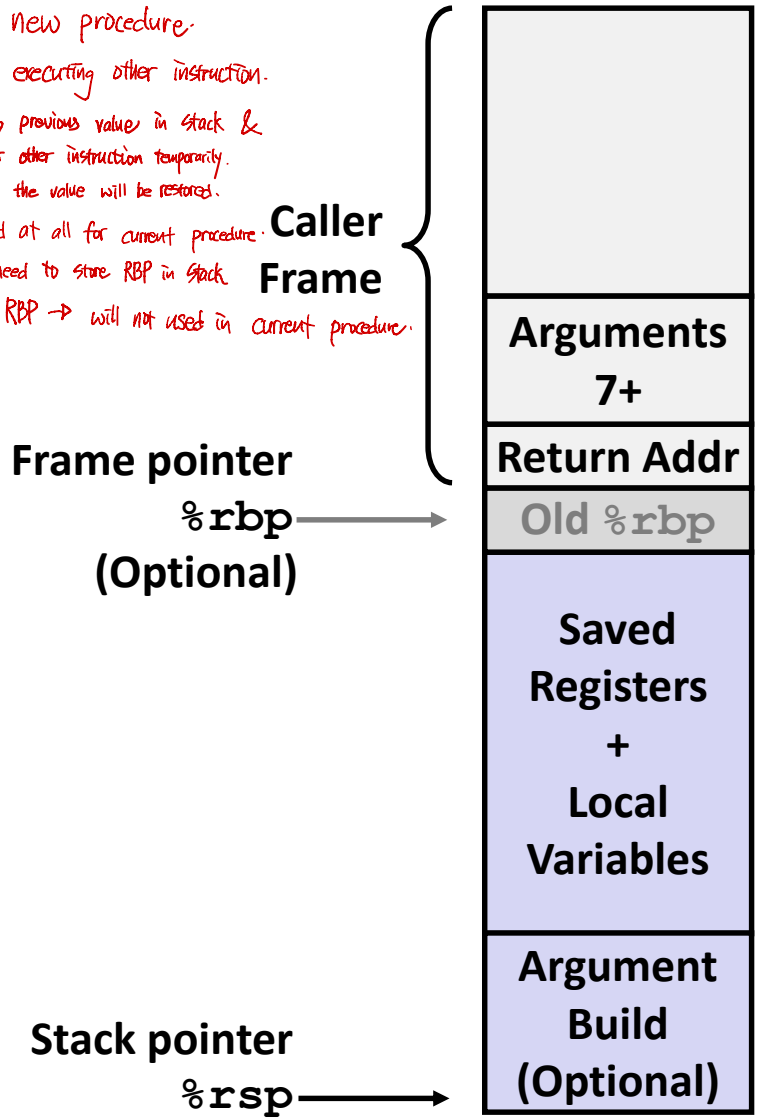
- Local variables

  If can't keep in registers

- Saved register context
- Old frame pointer (optional)

%RBP when handle new procedure.

why optional? { RBP for executing other instruction.
RBP stores previous value in stack & be used for other instruction temporarily. After using, the value will be restored.
⇒ not used at all for current procedure. ⋯ no need to store RBP in Stack
no need for RBP → will not used in current procedure.



Caller Stack Frame

Store { arguments to call the procedures
return address

- 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    X = *P
  addq    %rax, %rsi      Y = X+val
  movq    %rsi, (%rdi)    *P = Y
  ret
```

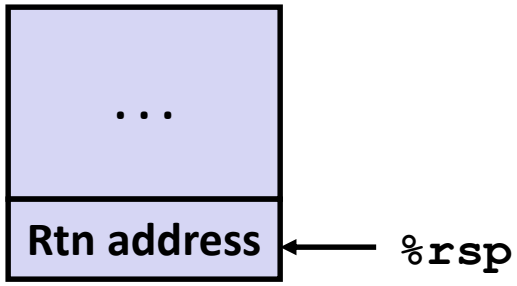
Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

Example: Calling `incr` #1

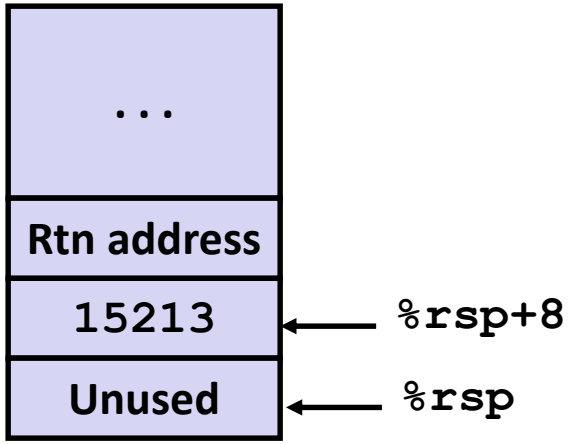
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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



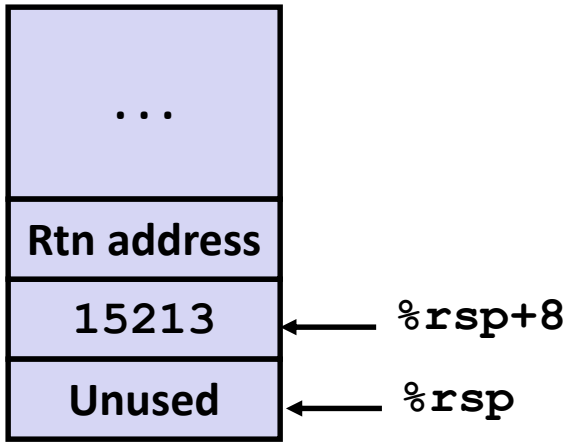
Example: Calling `incr` #2

```
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
```

for 4 bytes, part of RSI

Stack Structure



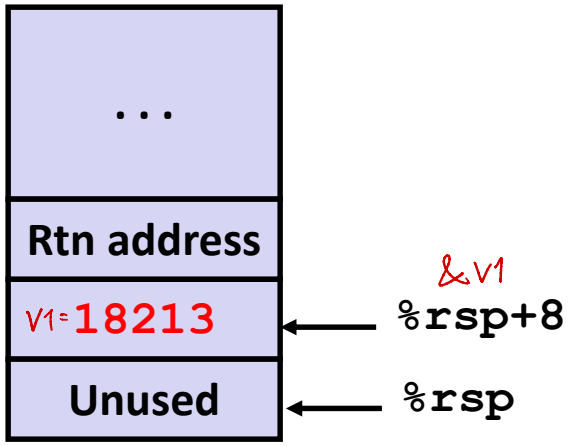
Register	Use(s)
%rdi	&v1
%rsi	3000

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



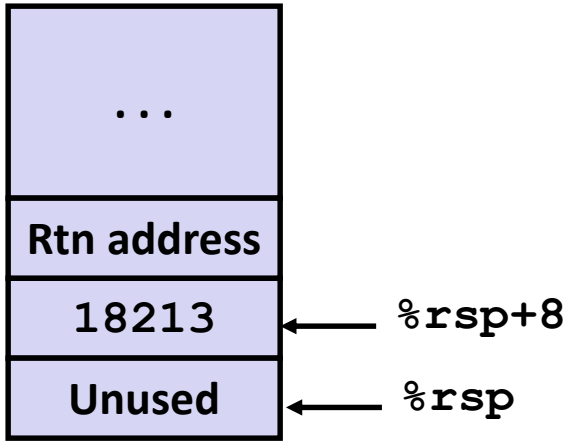
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

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

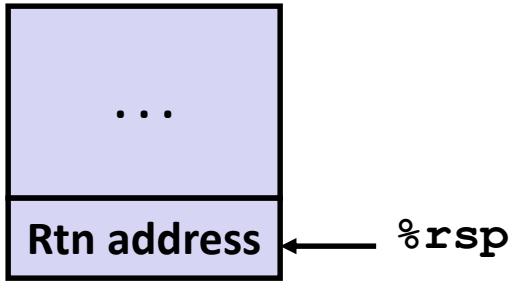
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure

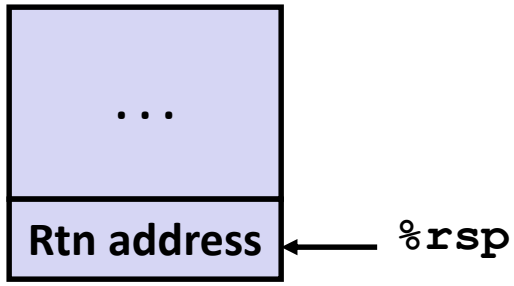


Example: Calling `incr` #5

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

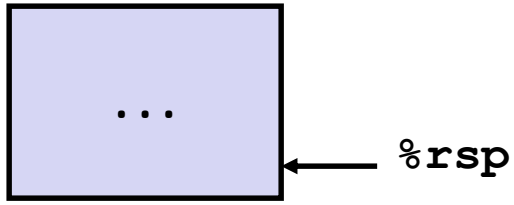
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



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

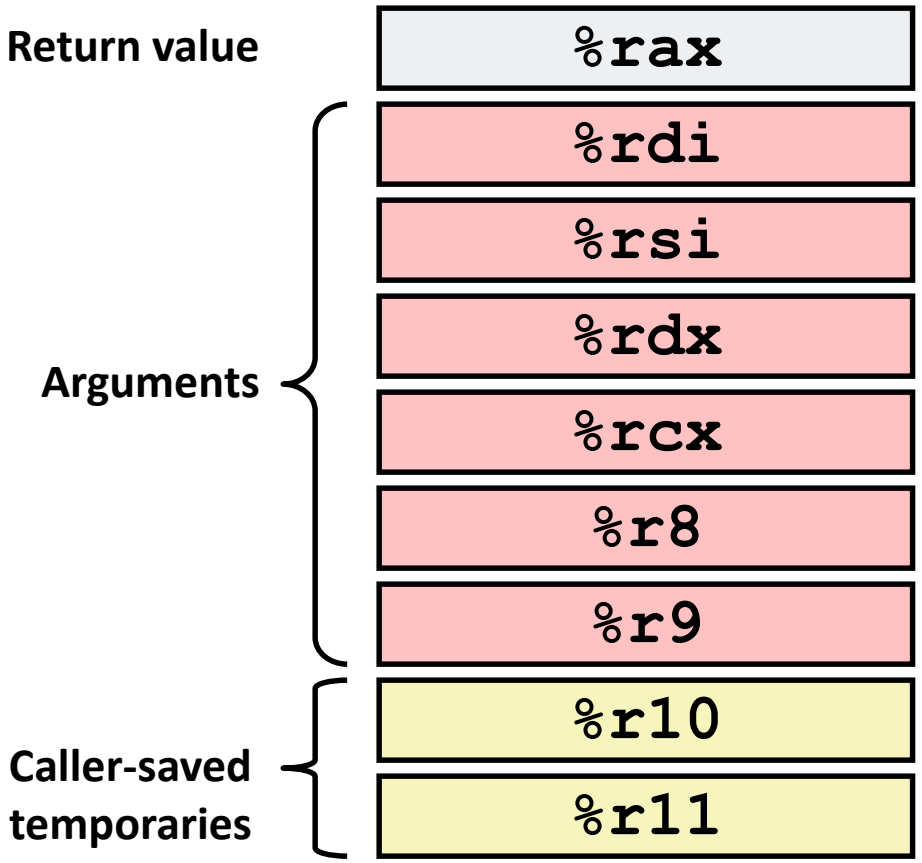
- **When procedure `yoo` calls `who`:**
  - `yoo` is the ***caller***
  - `who` is the ***callee***

- **Can register be used for temporary storage?**

- **Conventions** Caller or callee has responsibility to save & restore previous value in registers before using it as temporary storage.
  - ***"Caller Saved"***
    - Caller saves temporary values in its frame before the call (predefined registers) restore value! from other place.
  - ***"Callee Saved"***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

x86-64 Linux Register Usage #1

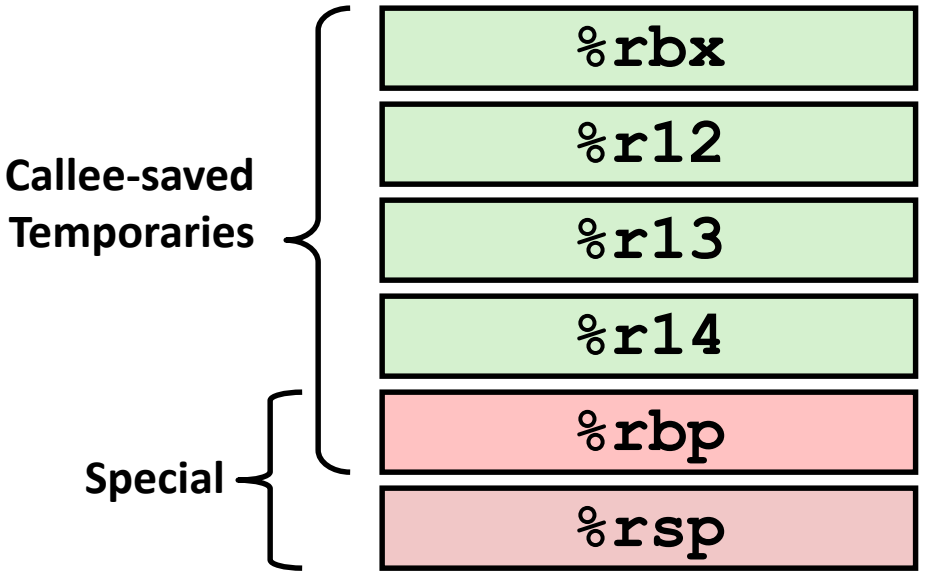
- **`%rax`**
  - Return value
  - Also caller-saved
  - Can be modified by procedure (callee) ∵ already backup by caller!
- **`%rdi`, ..., `%r9`**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure (callee) ∵ already backup by caller!
- **`%r10`, `%r11`**
  - Caller-saved
  - Can be modified by procedure (callee) ∵ already backup by caller!



x86-64 Linux Register Usage #2

- **`%rbx`, `%r12`, `%r13`, `%r14`**
  - Callee-saved
  - Callee must save & restore
- **`%rbp`** indicate starting point of stack frame in callee.
  → save value in %RBX & restore.
  - 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 to access variables stored in stack. When OG stack pointer is restored by callee.

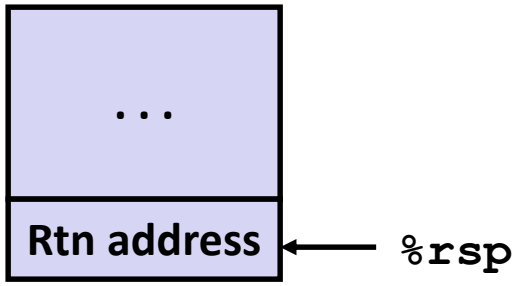


Callee-Saved Example #1

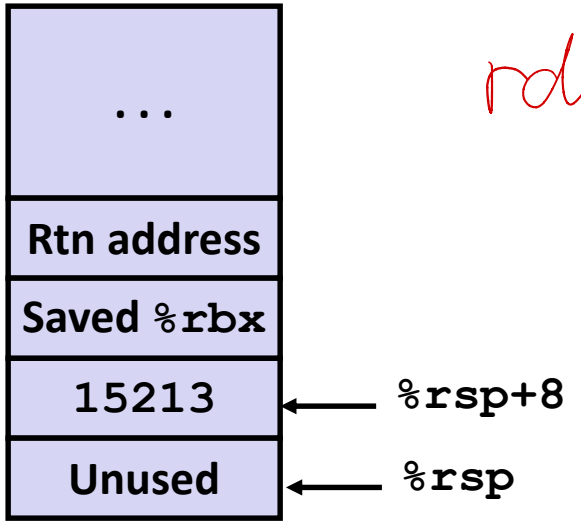
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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



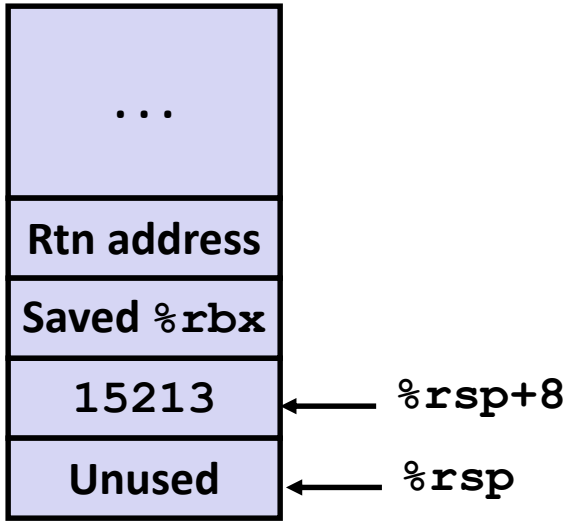
Callee-Saved Example #2

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

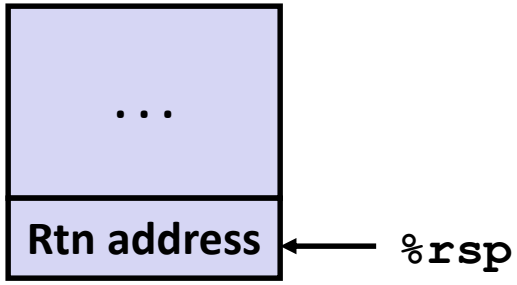
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

x+v2

Resulting Stack Structure



Pre-return Stack Structure



Today

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

Recursive Function

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

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

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

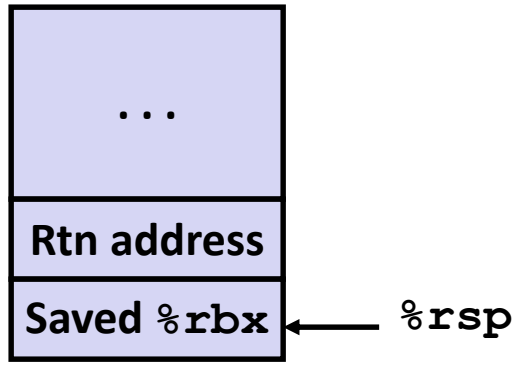
Recursive Function Register Save

```
/* 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 # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

pushq %rbx ⇒ callee saved registers, store variable temporarily

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 # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

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

Recursive Function Call

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

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

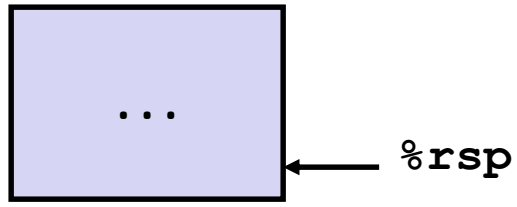
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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 # (by 1)
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx   restore into OG value
.L6:
   rep; ret   restore %RSP
```

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
- 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

Practice: Recursive Call

- **When we have the following C code, fill the blanks in assembly code**
  - Check your results based on the assembly code generated from GCC compiler.
  - Use –Og option: gcc –S –Og rfun.c

```
long rfun (unsigned long x) {
    if (X == 0)
        return 0;
    unsigned long nx = X >> 2;
    long rv = rfun(nx);
    return x +rv;
}
```

```
rfun:
    testq   1) %rdi, %rdi
    je      .L3
    pushq   %rbx
    movq    %rdi, %rbx
    shrq    $2, 2) %rdi
    call    rfun
    addq    %rbx, 3) %rax
    jmp .L2
.L3:
    movl $0, %eax
    ret
.L2:
    popq 4) %rbx
    ret
```

x86-64 Procedure Summary

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