Machine-Level Programming II

Computer Systems
Sep. 18, 2019

Michael Kirkedal Thomsen

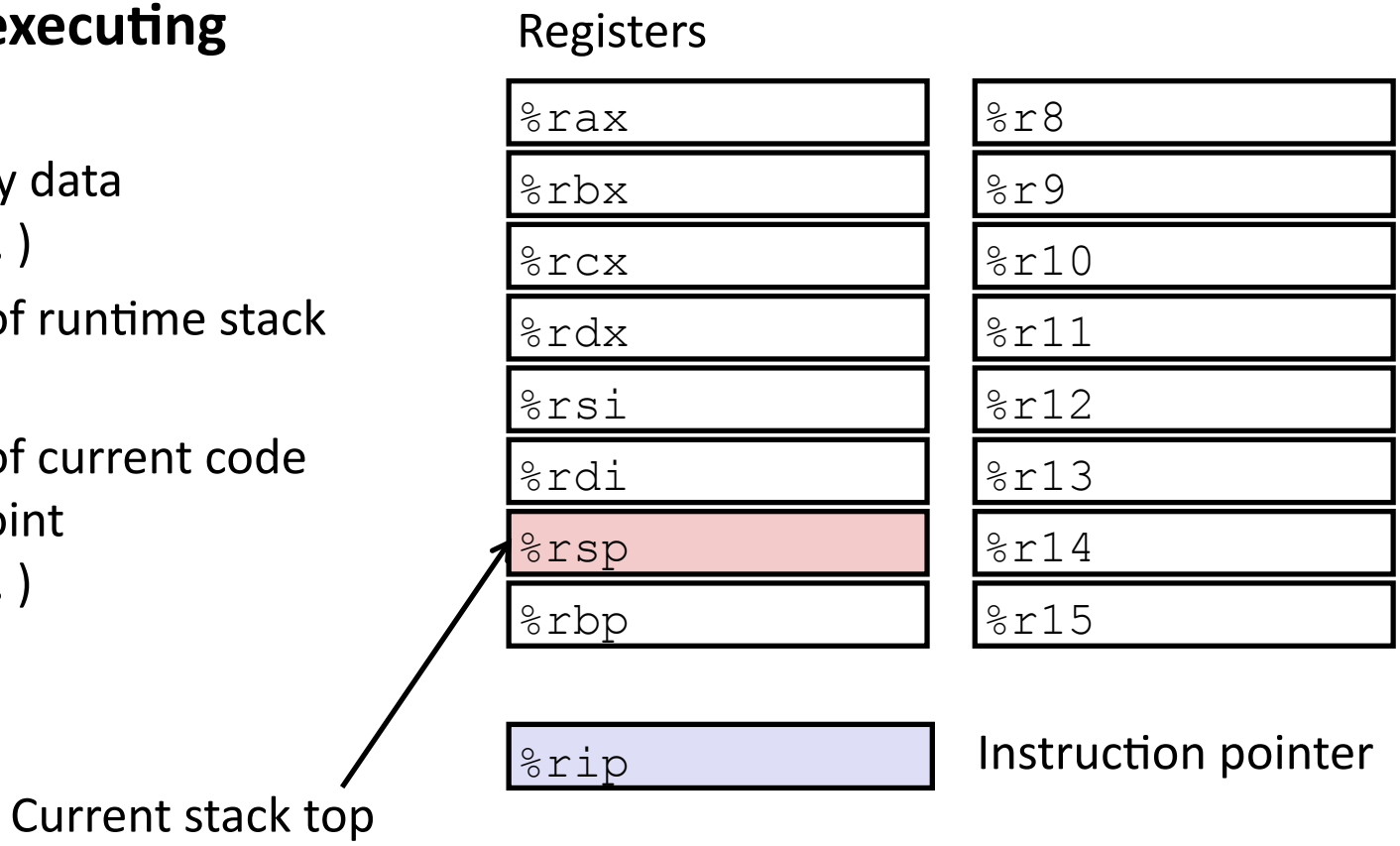
Based on slides by Randal E. Bryant and David R. O'Hallaron

What did you learn last Monday?

- What do you remember from Monday?

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`, …)



Today

- **Control**
  - **Condition branches (the x86prime way)**
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

Branching (x86prime)

- **jmp instruction**
  - Unconditionally continues execution at an address
  - jmp label
- **cbX Instructions**
  - Jump to different part of code depending on Boolean comparison
  - cbX r1 r2 label
  - cbX $i r1 label

Compare and branch (x86prime)

- **cbX Instructions**
  - Jump to different part of code depending on Boolean comparison

cbX	Condition	Description
`cbe`	`v1 = v2`	Equal
`cbne`	`v1 /= v2`	Not Equal
`cbg`	`v1 > v2`	Greater (Signed)
`cbge`	`v1 >= v2`	Greater (Signed)
`cbl`	`v1 < v2`	Less (Signed)
`cble`	`v1 <= v2`	Less or Equal (Signed)
`cba`	`..`	Above (unsigned)
`cbae`	`..`	Above or Equal (unsigned)
`cbb`	`..`	Below(unsigned)
`cbbe`	`..`	Below or Equal (unsigned)

  - Why do we have both signed and unsigned comparisons?

Conditional Branch Example

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

```
absdiff:
    jle     %rsi, %rdi, .L4 # x<y
    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;
    if (x <= y) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Conditional Branch Example from GoTo

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

```
absdiff:
   jle     %rsi, %rdi, .Else
   movq    %rdi, %rax
   subq    %rsi, %rax
   jmp .Done
.Else:        # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
.Done
   ret
```

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

General Conditional Expression Translation (Using Branches)

C Code

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

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

Goto Version

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

- Calculate advanced part of Test
- Create separate code regions for then & else expressions
- Execute appropriate one

Conditional Branch advanced expressions

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

```
func:
    subq    %rdi, %rdx
    jle     %rsi, %rdx, .L4 # z-x<y
    movq     %rdi, %rax
    subq     %rsi, %rax
    ret
.L4:         # x <= y
    movq     %rsi, %rax
    subq     %rdi, %rax
    ret
```

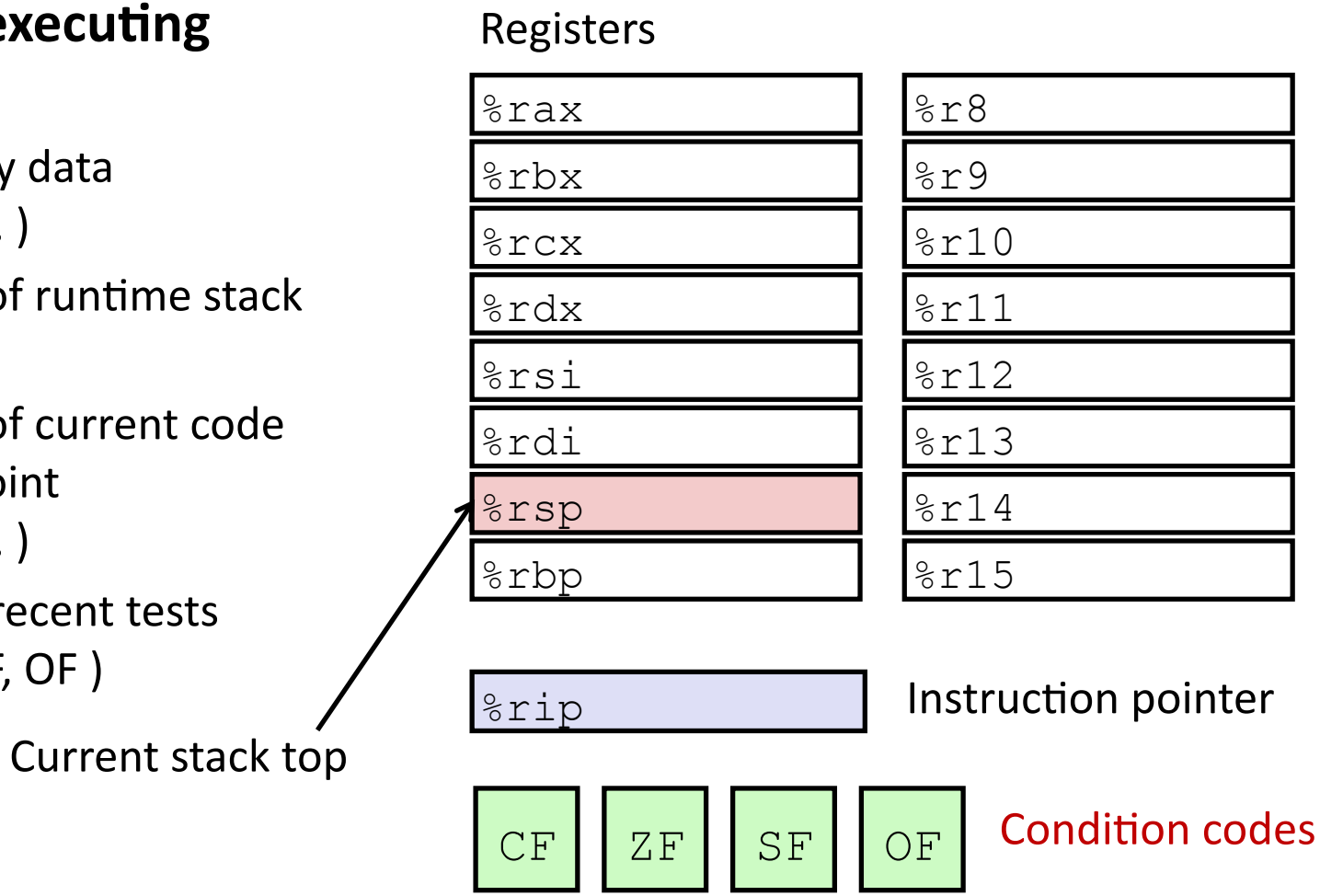
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Today

- Control
  - Condition branches (the x86prime way)
  - **Conditional codes (the x86 way)**
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

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 (Implicit Setting)

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

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

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

  CF set if carry out from most significant bit (unsigned overflow)

  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)`

- **Not set by `leaq` instruction**

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)
```

Condition Codes (Explicit Setting: Test)

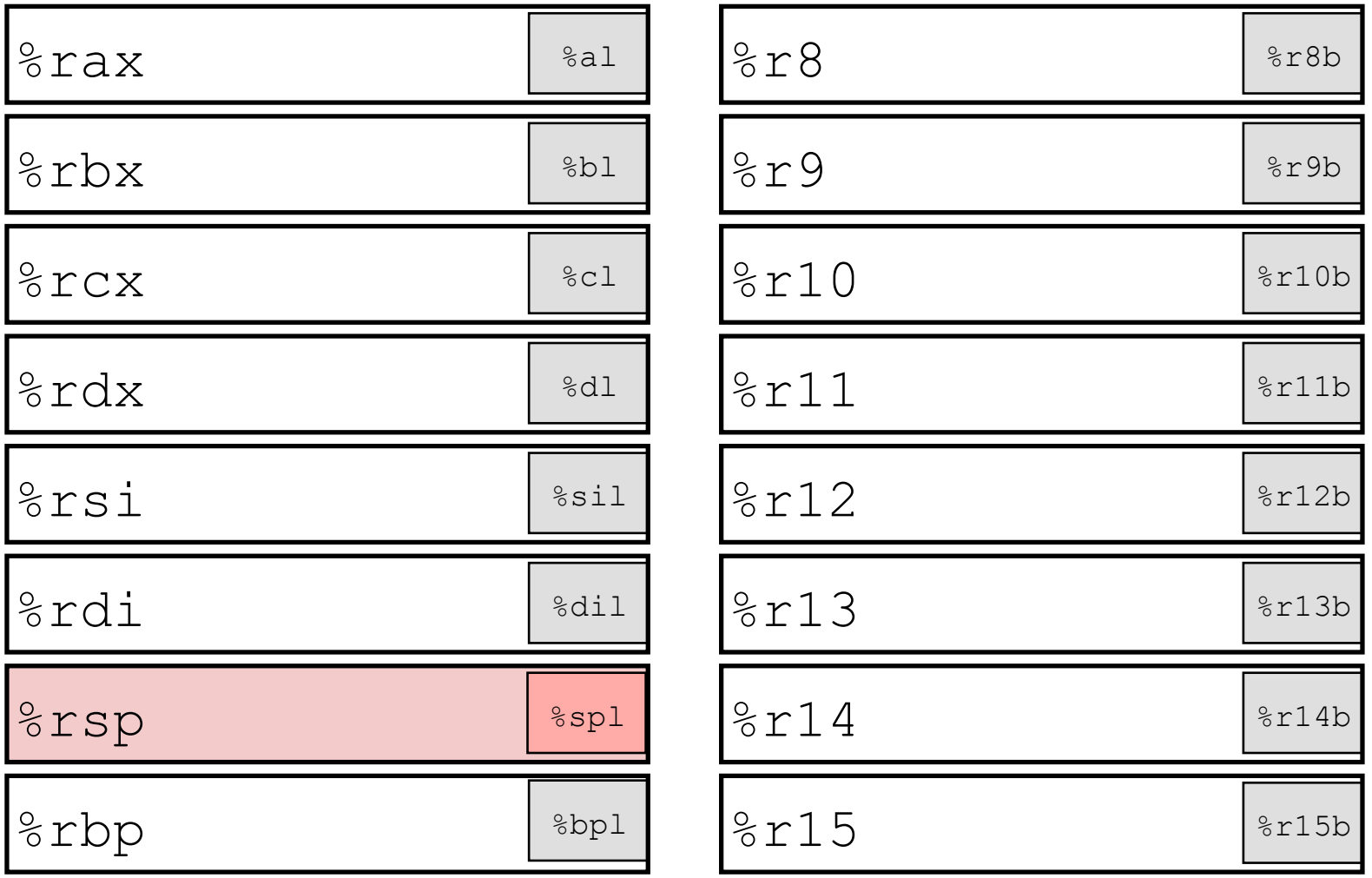
- **Explicit Setting by Test instruction**
  - `testq` Src2, Src1
    - `testq b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of Src1 & Src2
  - Useful to have one of the operands be a mask
  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`

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

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
```

Jumping

- **jX Instructions**
  - 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
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

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   # x:y
    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

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;
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
    cmovle  %rdx, %rax  # if <=, result = eval
    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 branches (the x86prime way)
  - Conditional codes (the x86 way)
  - **Loops**
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

"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, x86prime

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

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

```
#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;                          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 branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - **Switch Statements**
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

```
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

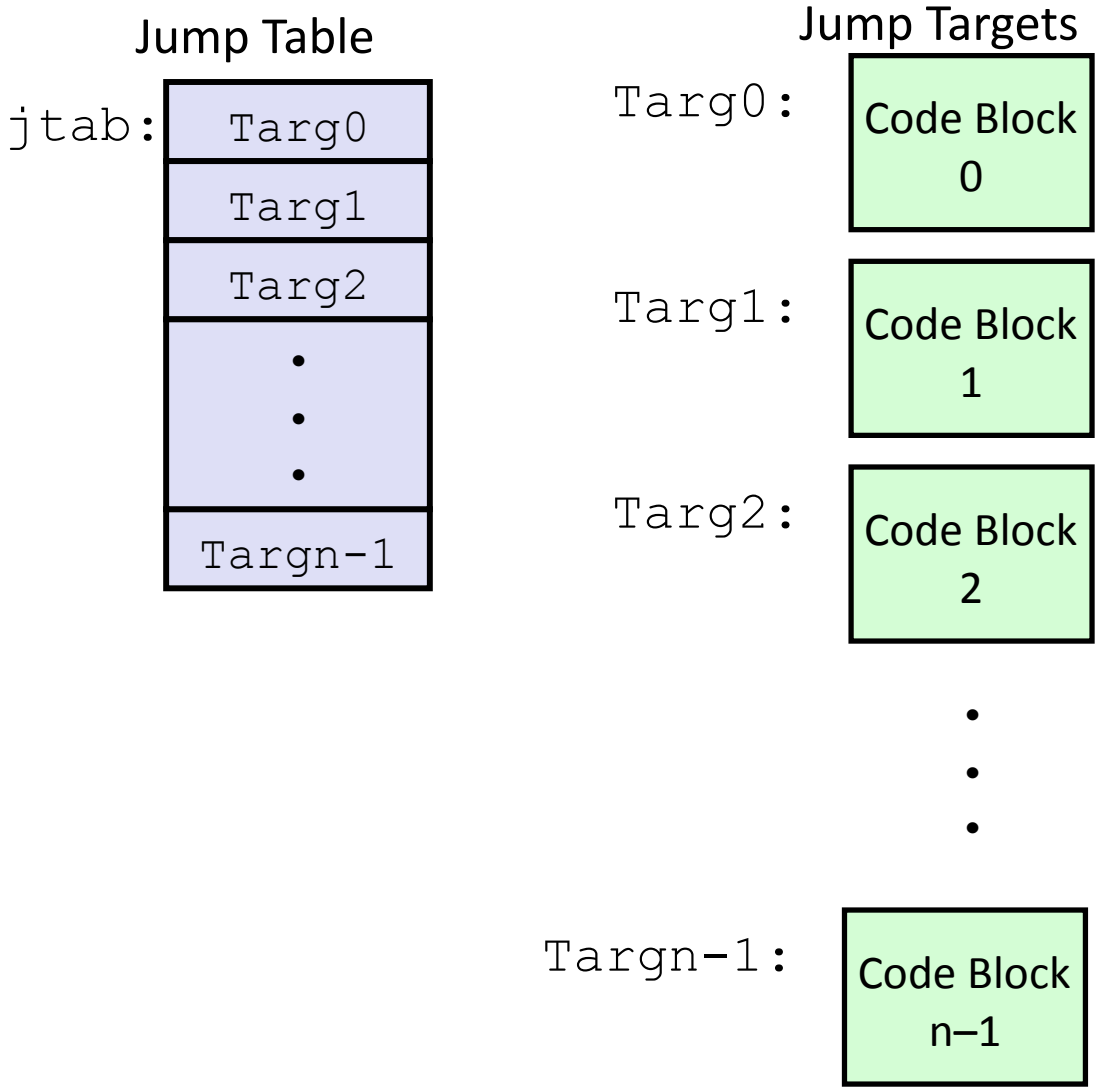
Jump Table Structure

Switch Form

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

Translation (Extended C)

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

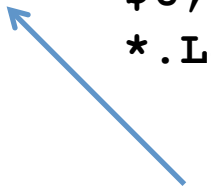


Switch Statement Example, x86prime

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

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cba     $6, %rdi, .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
    cba     $6, %rdi, .L8
    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 $0 \le x \le 6$

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
```

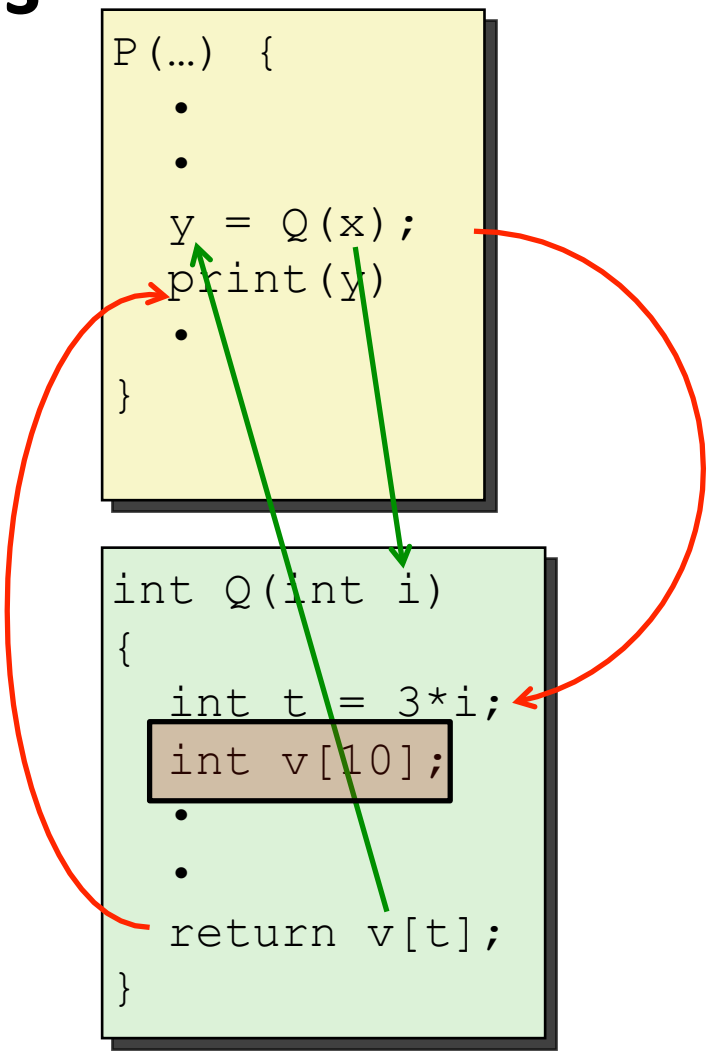
```
    switch(x) {
    case 1:       // .L3
        w = y*z;
        break;
    case 2:       // .L5
        w = y/z;
        /* Fall Through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
     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

Today

- Control
  - Condition branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- **Procedures**
  - **Stack Structure**
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

Mechanisms in Procedures

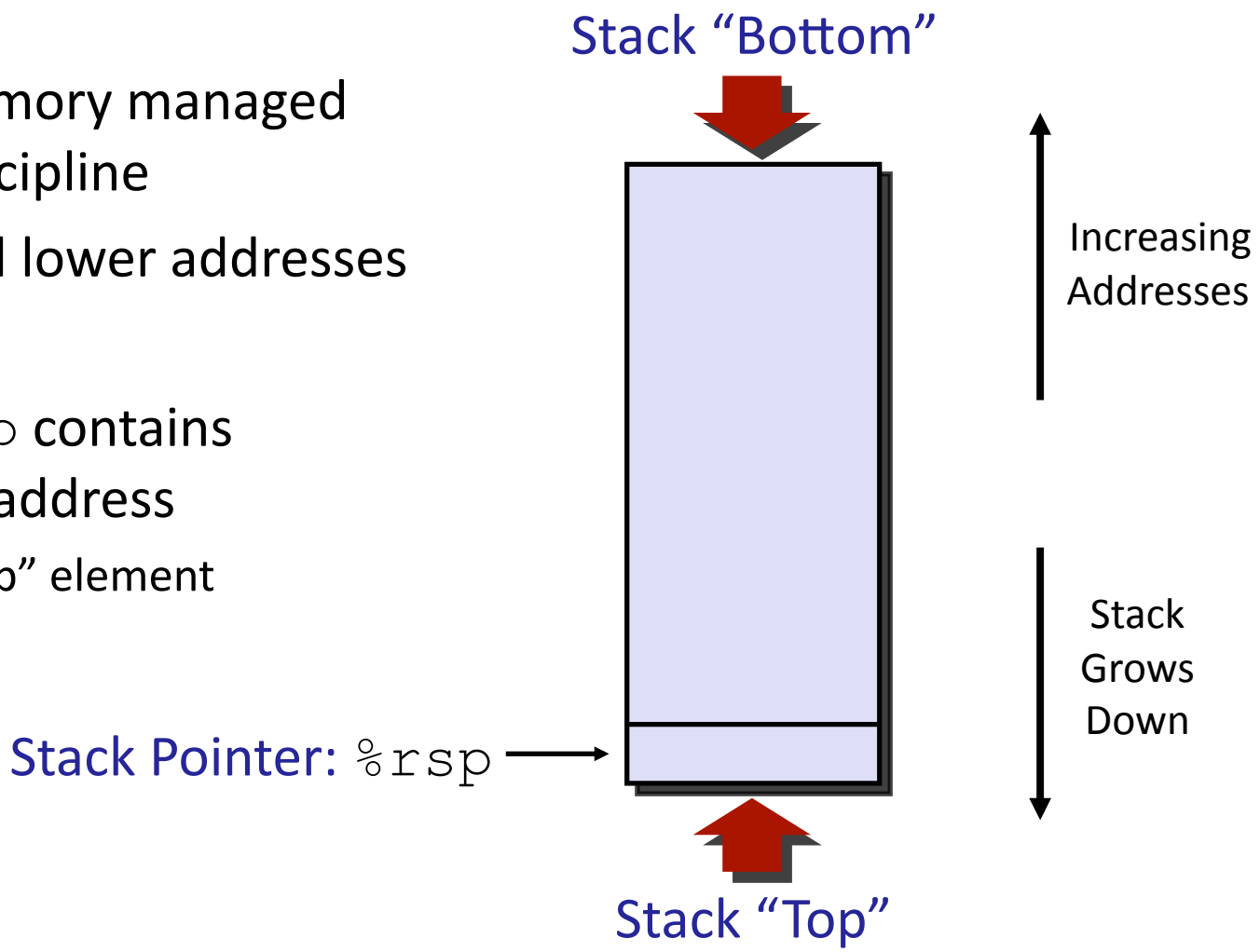
- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of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



x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register `%rsp` contains lowest stack address
  - address of "top" element

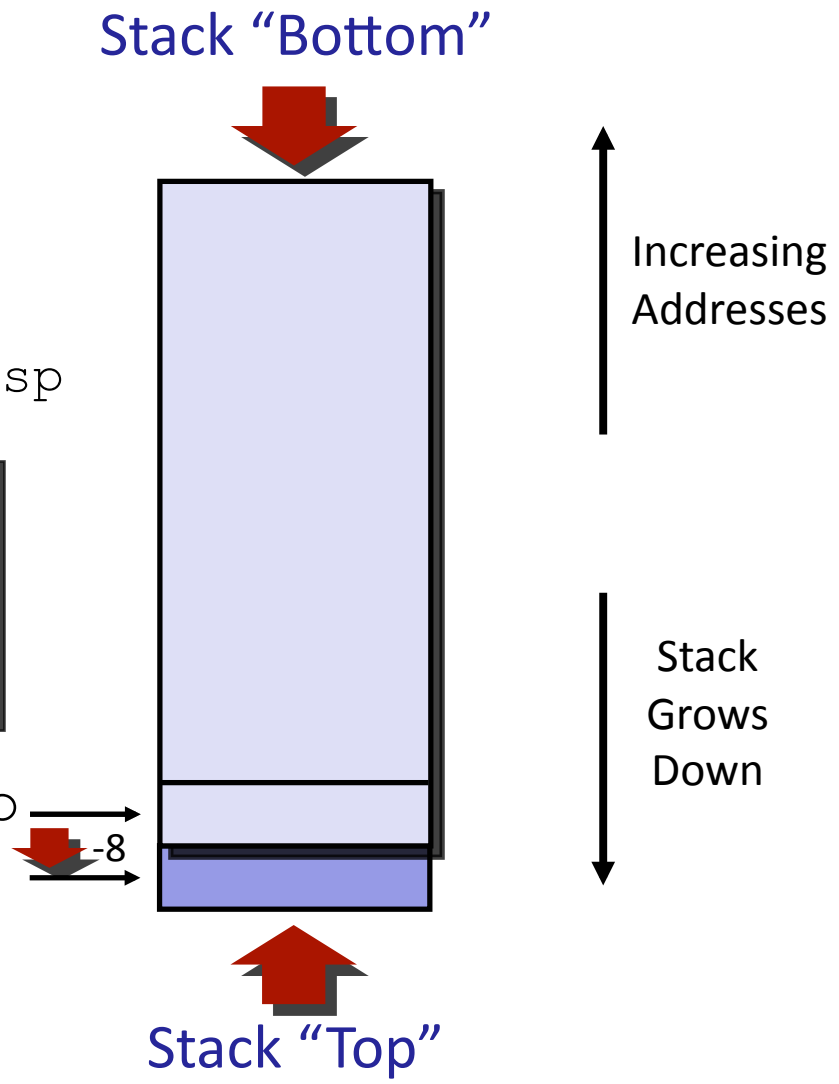


x86-64 Stack: Push

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

pushq is not in x86prime

```
subq $8 %rsp
movq Src (%rsp)
```

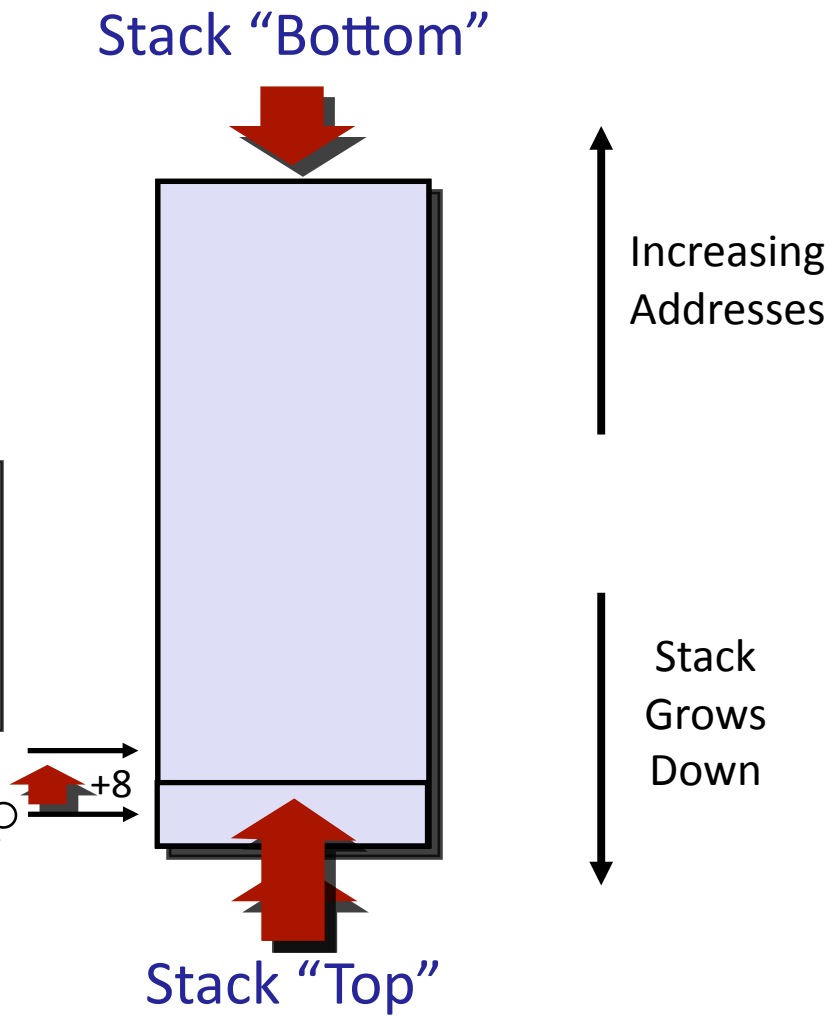


x86-64 Stack: Pop

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

pushq is not in x86prime

```
movq (%rsp) Dest
addq $8 %rsp
```



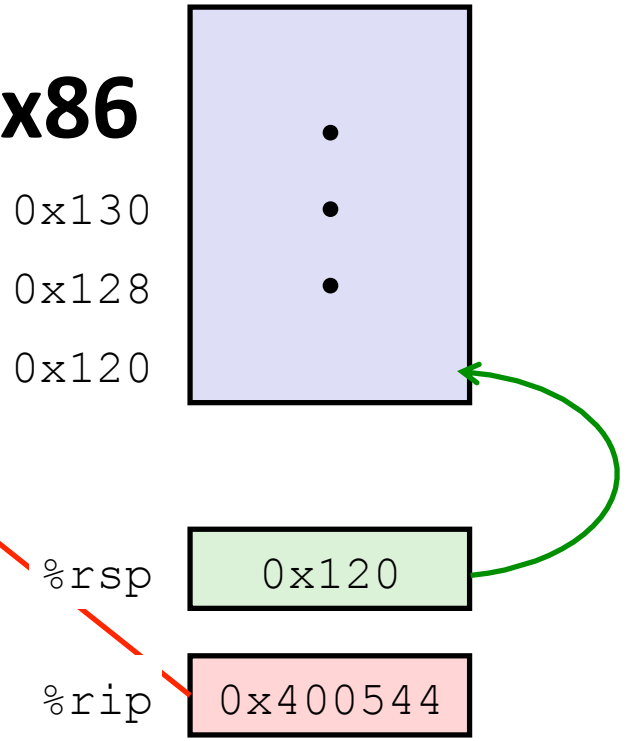
Today

- Control
  - Condition branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- **Procedures**
  - Stack Structure
  - **Calling Conventions**
    - **Passing control**
    - Passing data
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

Control Flow Example #1, x86

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

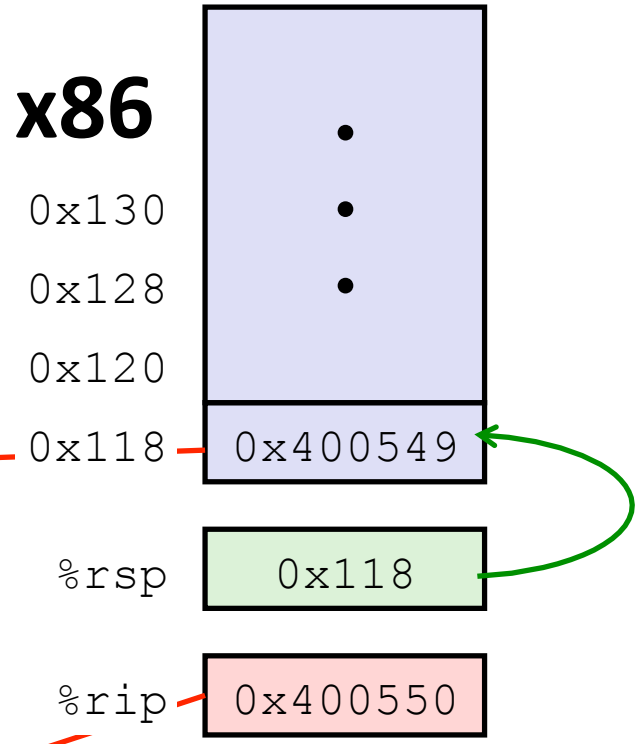
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2, x86

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

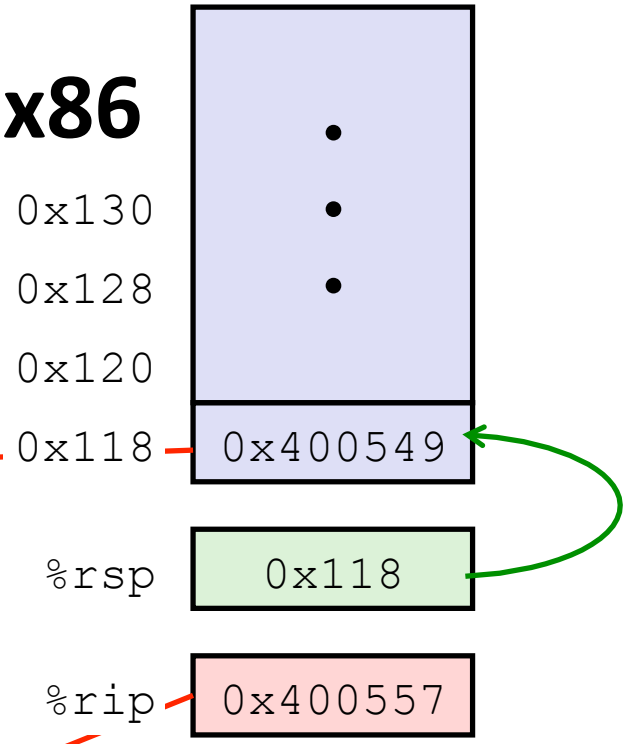
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3, x86

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

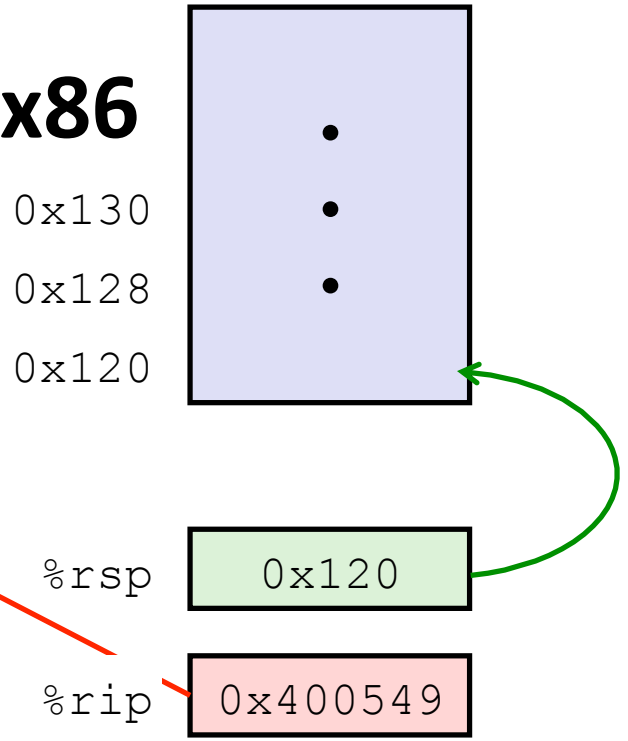
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #4, x86

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

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



Control flow x86prime

- call and return only affects the PC
  - No interaction with call stack

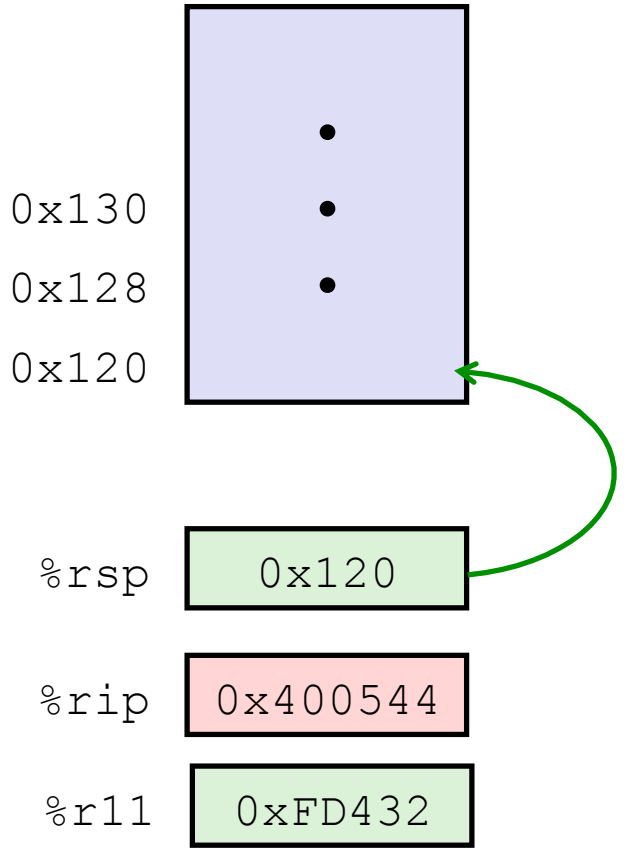
- Call stores return address in register
  - `call Label, Dst`
- Return reads return address in register
  - `Ret  Src`
  - NB! Src is not the return value

Control flow x86prime

- call and return only affects the PC
  - No interaction with call stack

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550, %r11 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret %r11
```

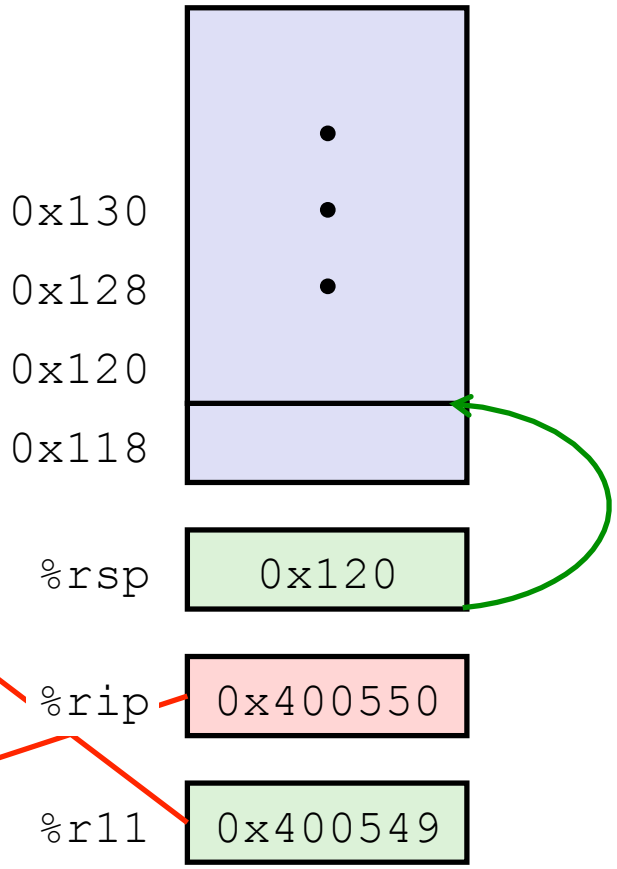


Control flow x86prime

- call and return only affects the PC
  - No interaction with call stack

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550, %r11 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret %r11
```

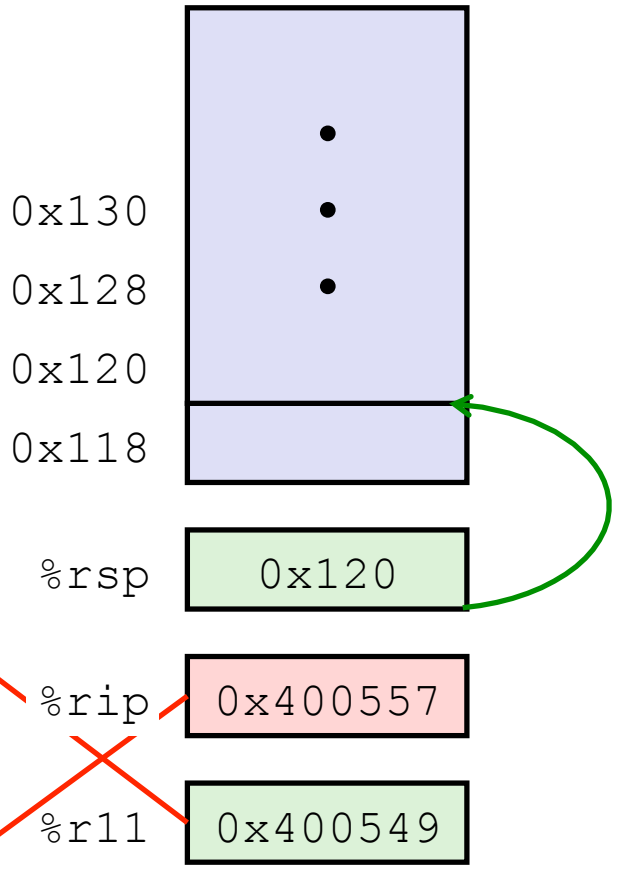


Control flow x86prime

- call and return only affects the PC
  - No interaction with call stack

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550, %r11 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret %r11
```

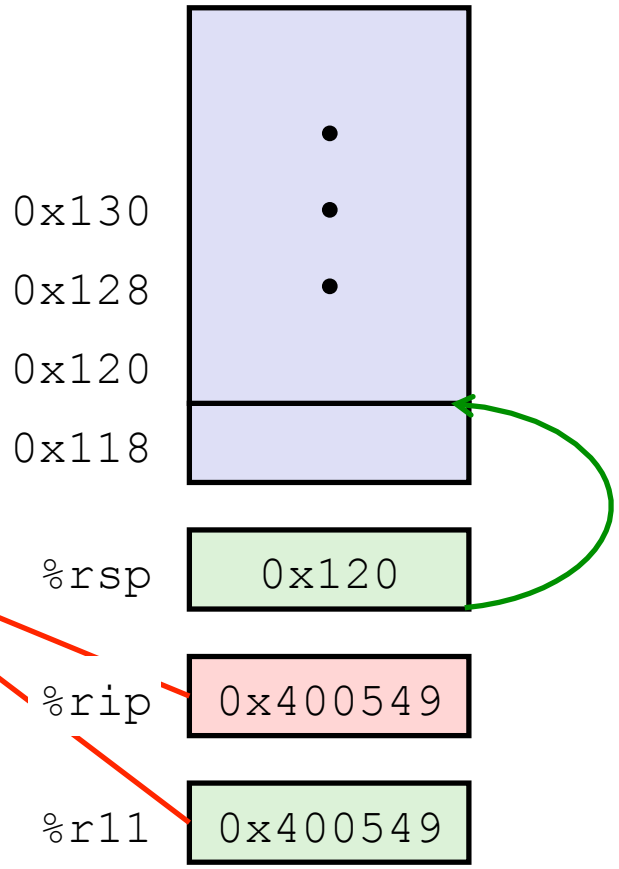


Control flow x86prime

- call and return only affects the PC
  - No interaction with call stack

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550, %r11 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret %r11
```

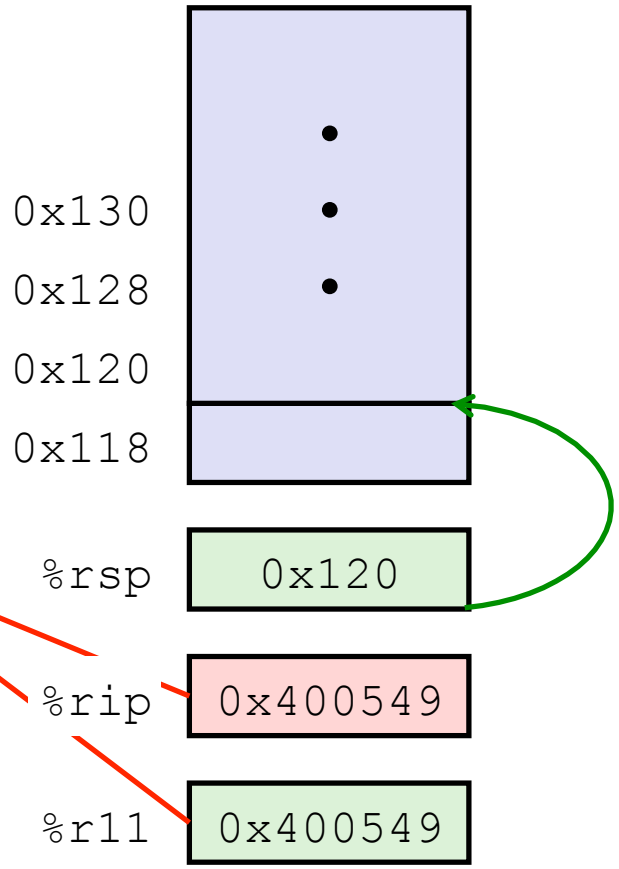


Control flow x86prime

- call and return only affects the PC
  - No interaction with call stack

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550, %r11 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret %r11
```



Reuse %r11?

Push/pop (spill) to stack

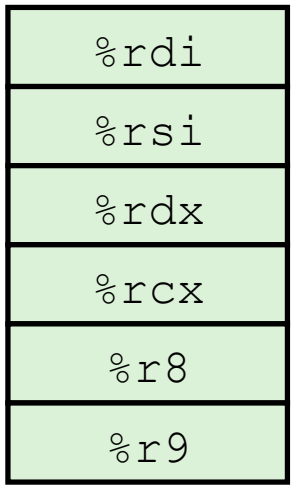
Today

- Control
  - Condition branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- **Procedures**
  - Stack Structure
  - **Calling Conventions**
    - Passing control
    - **Passing data**
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

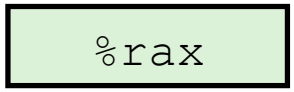
Procedure Data Flow

Registers

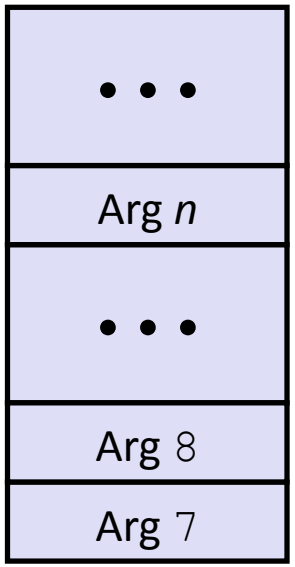
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed

Data Flow Examples, x86

```
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

- Control
  - Condition branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- **Procedures**
  - Stack Structure
  - **Calling Conventions**
    - Passing control
    - Passing data
    - **Managing local data**
  - Illustration of Recursion
- Arrays and Structures

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

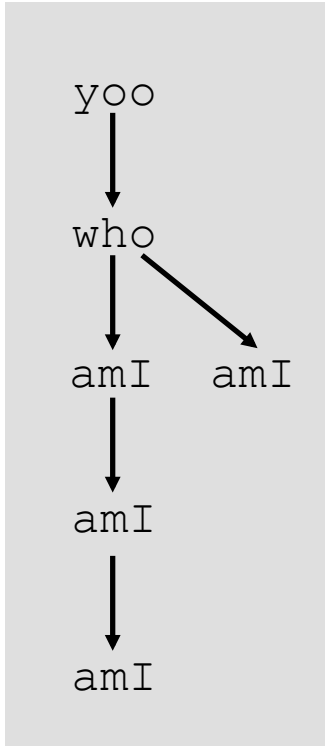
Call Chain Example

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

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

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

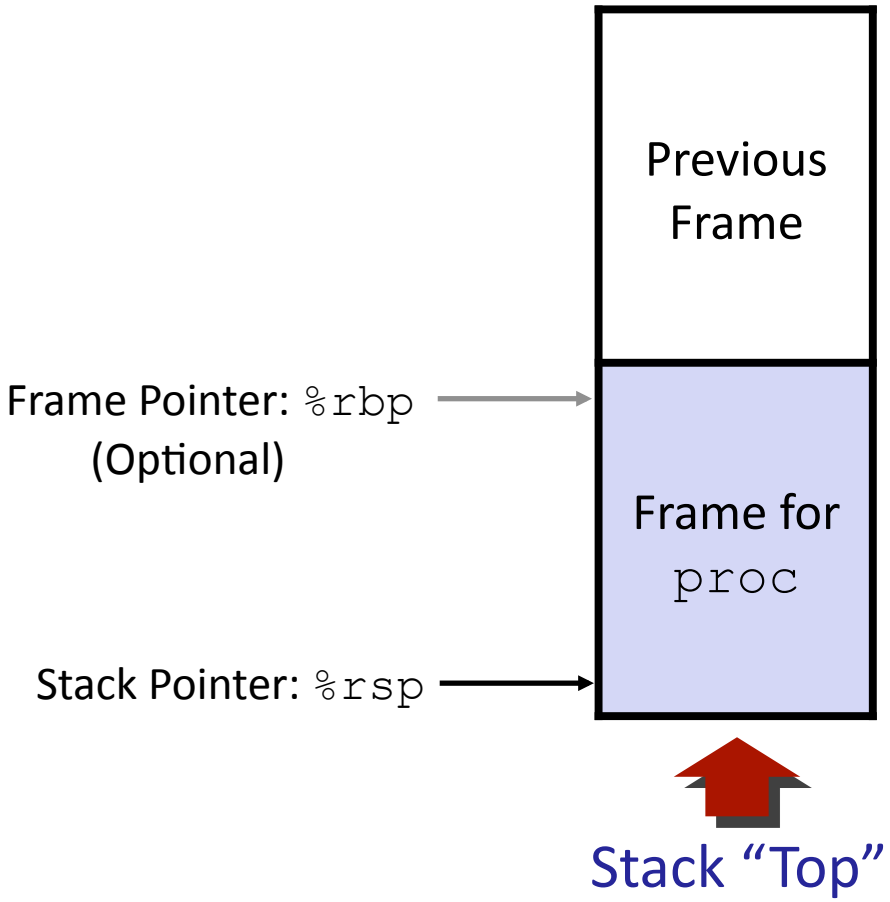
Example
Call Chain



Procedure `amI()` is recursive

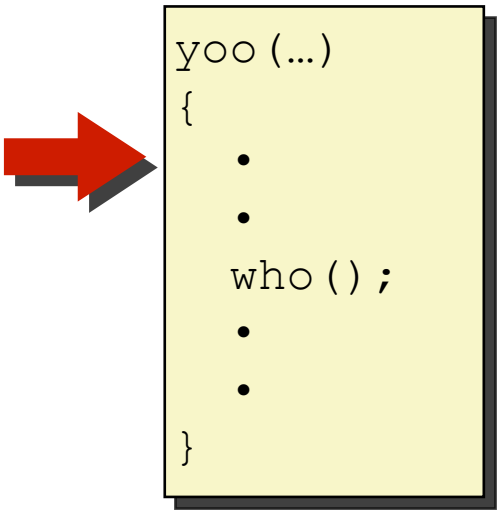
Stack Frames

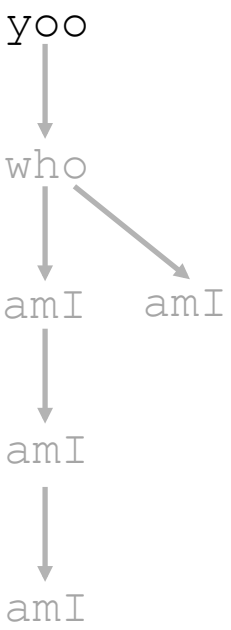
- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

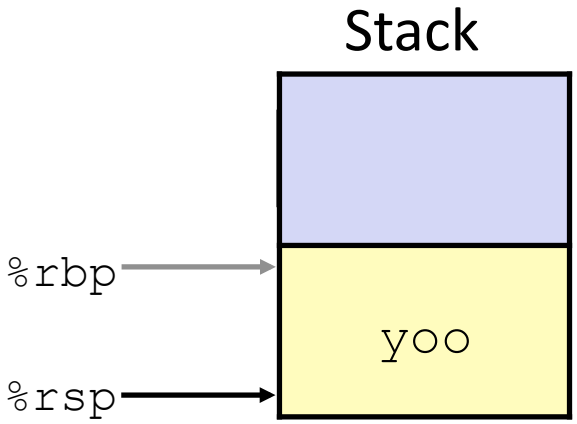


- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction

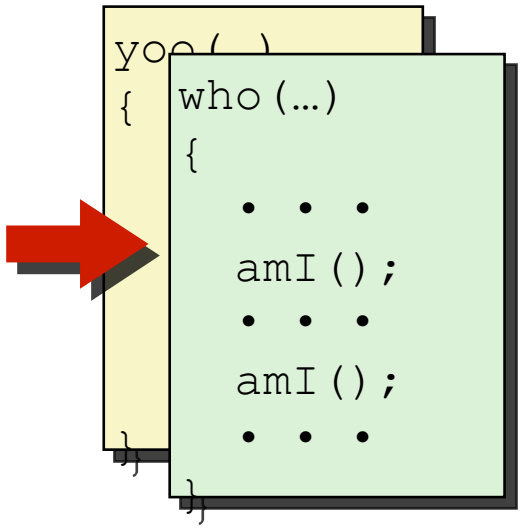
Example

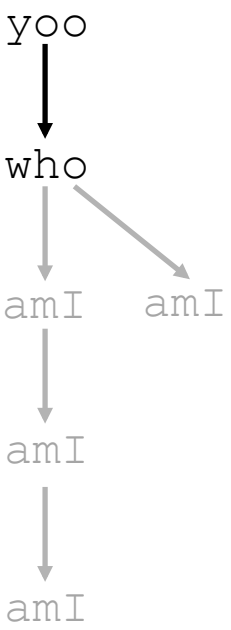


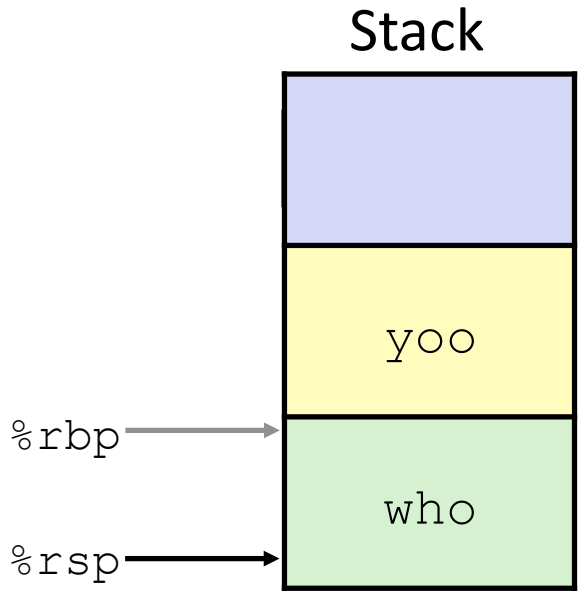




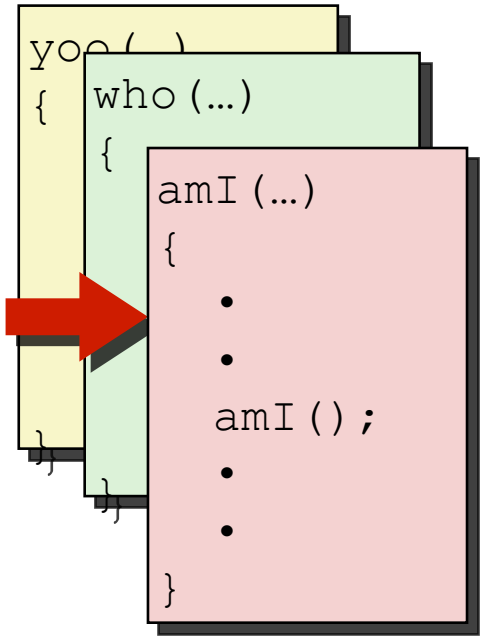
Example

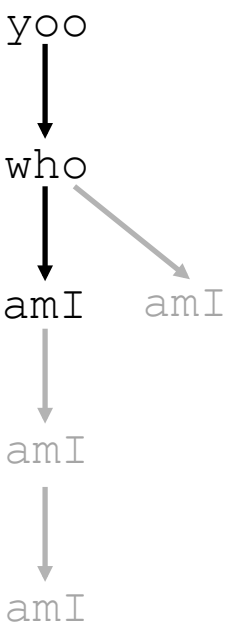


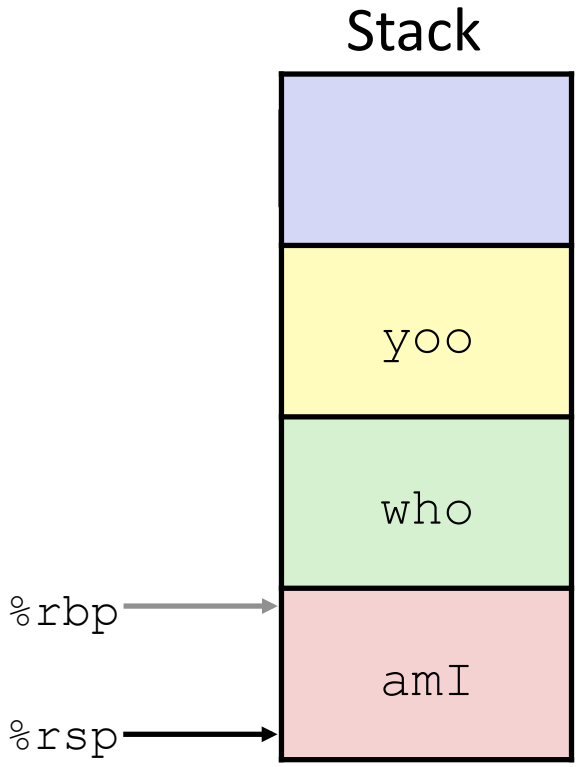




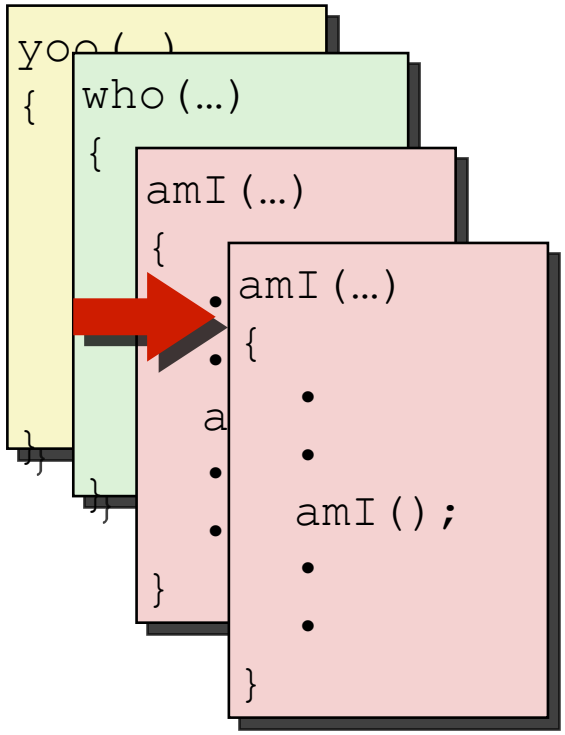
Example

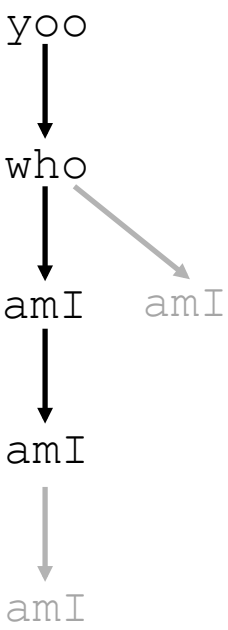

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

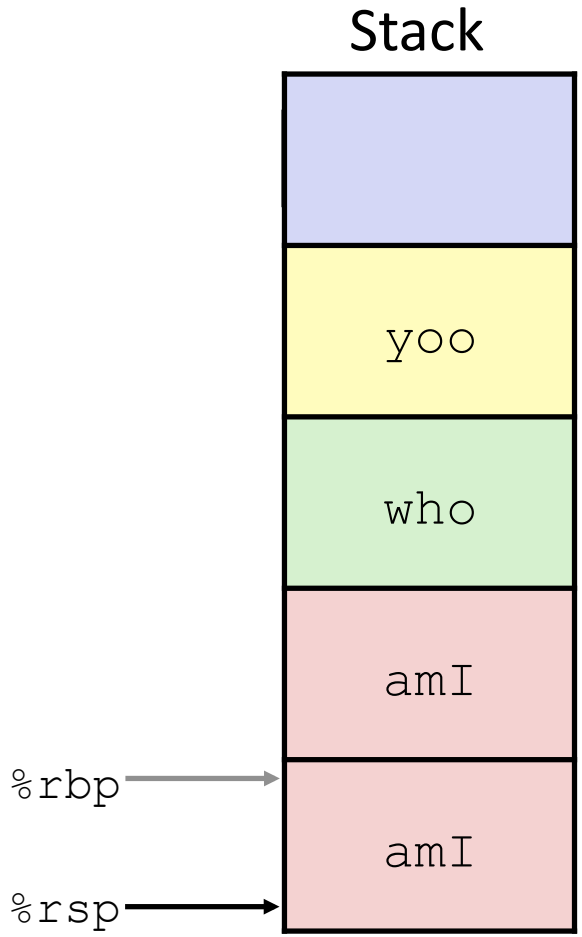

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

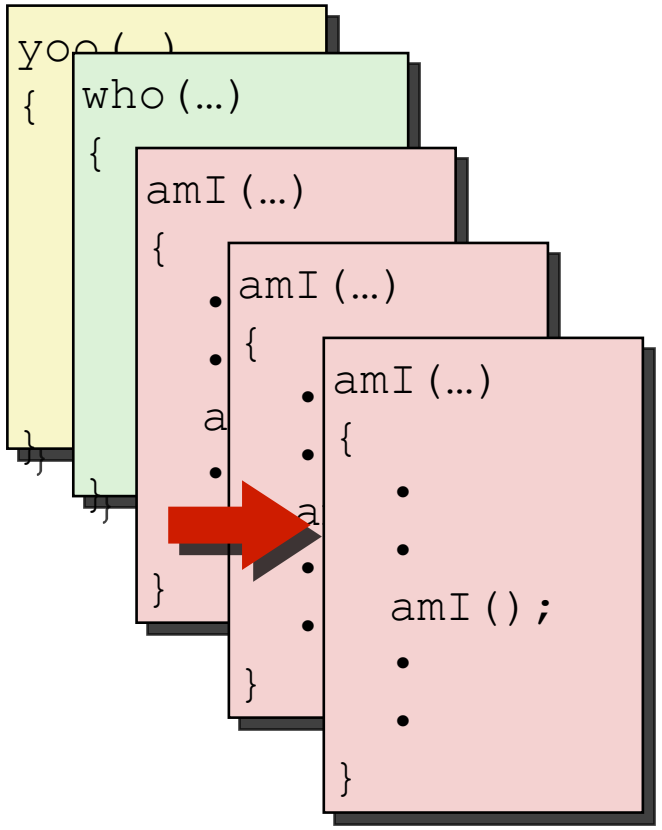
Example

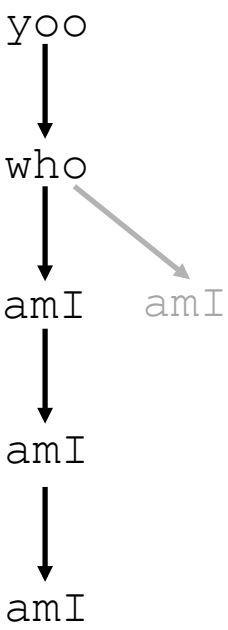

who(…)
{
amI(…)
{
amI(…)
{
amI();
}

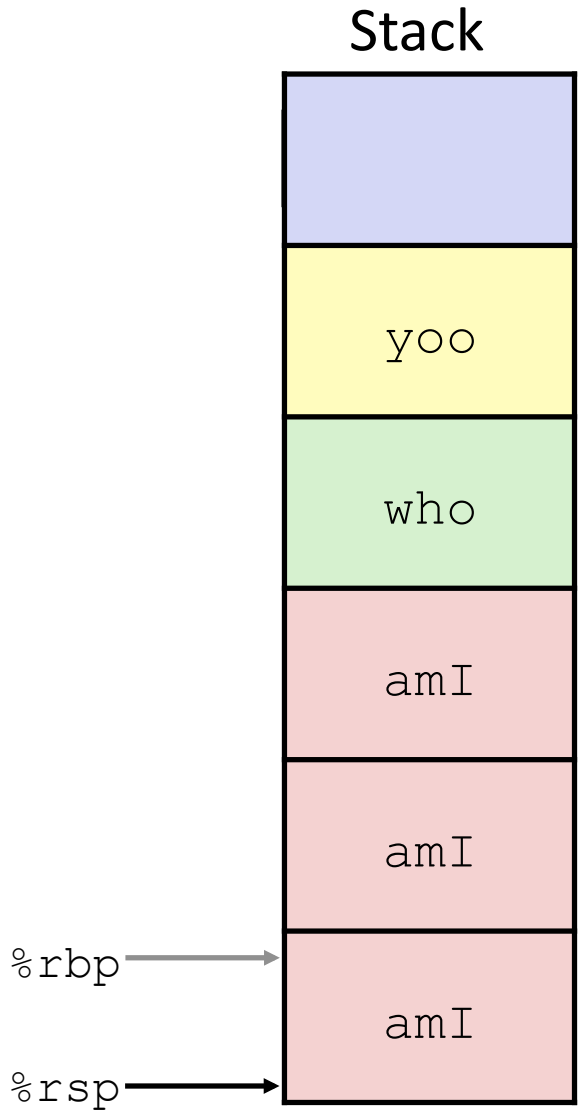

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

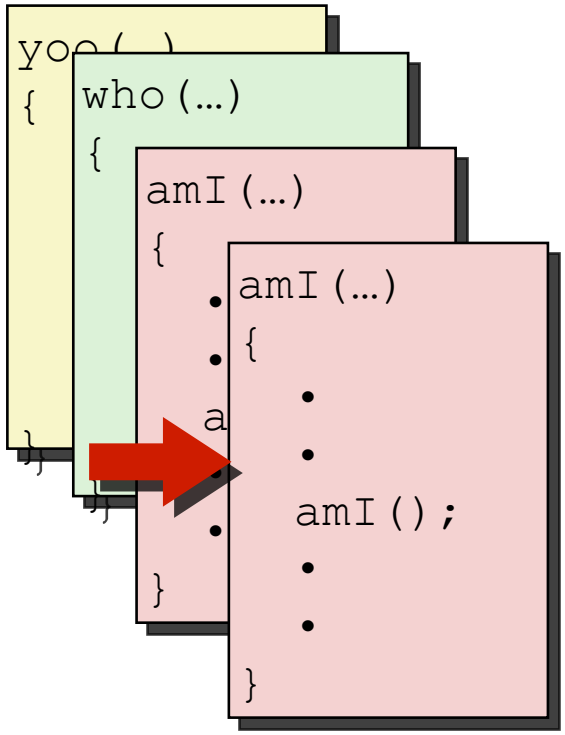
Example

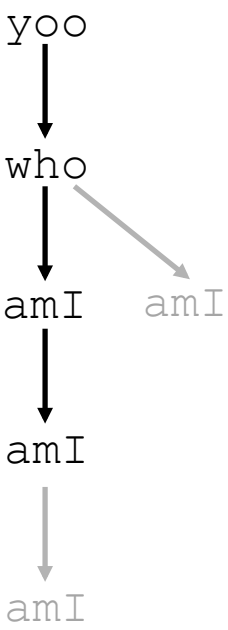

who (…)
amI (…)
amI (…)
amI (…)
amI();

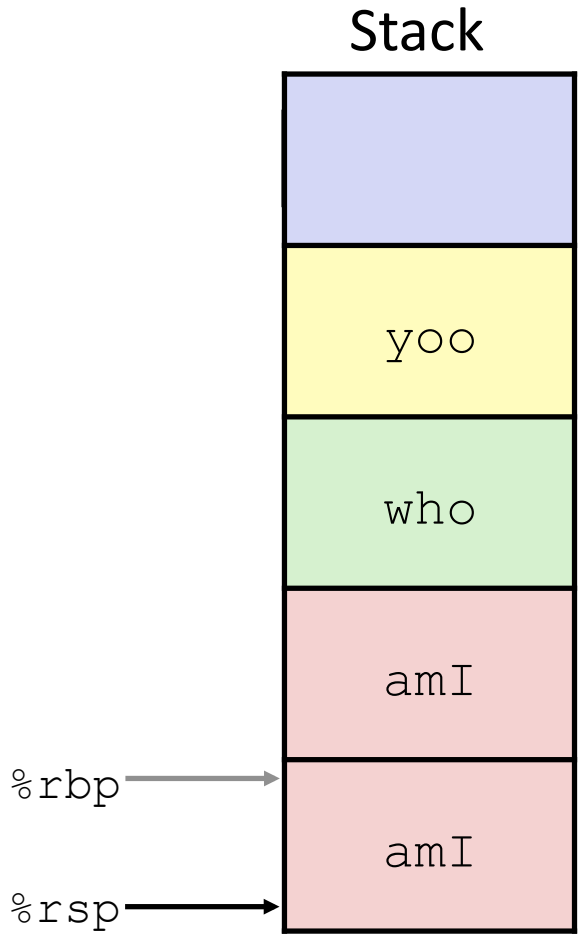

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

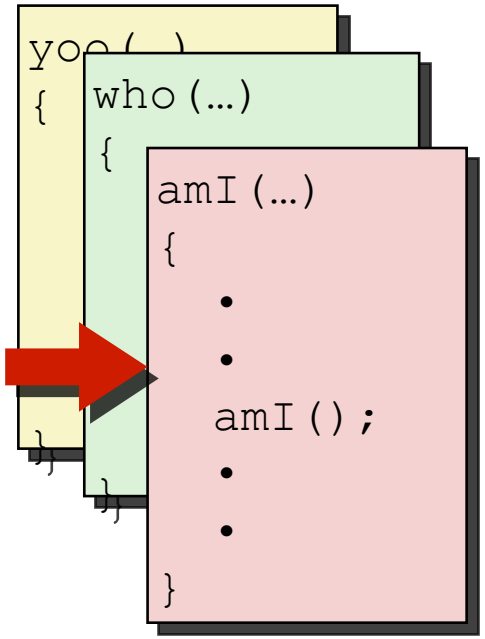
Example

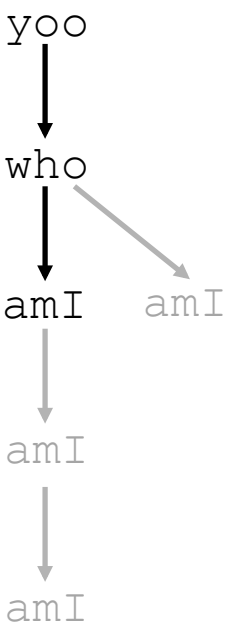

who(…)
{
amI(…)
{
amI(…)
{
amI();
}

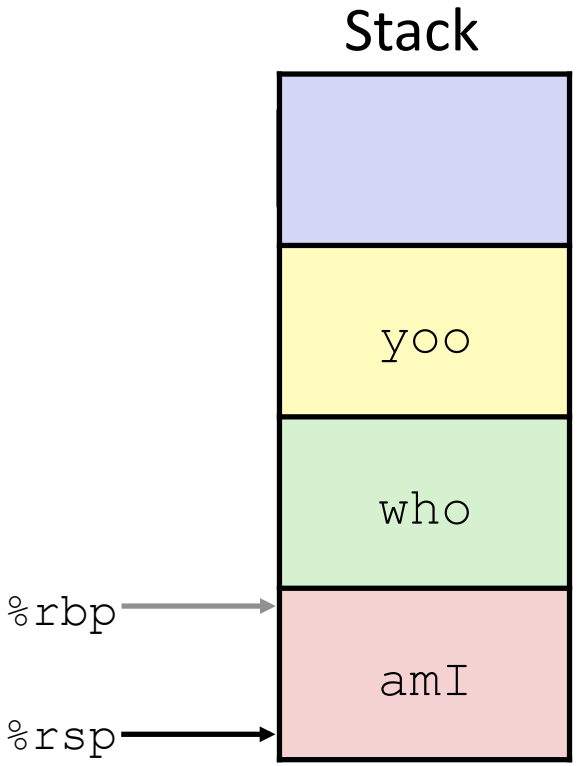

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

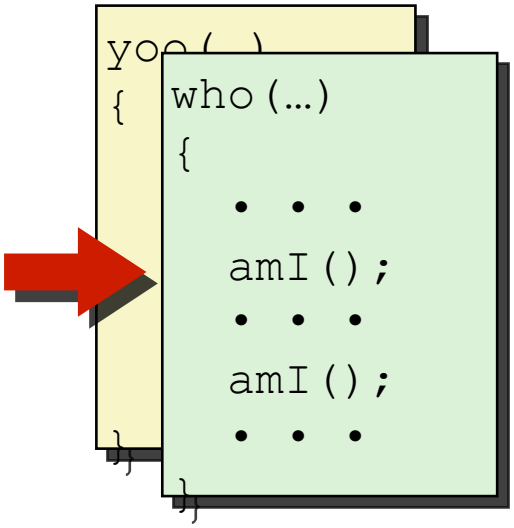
Example

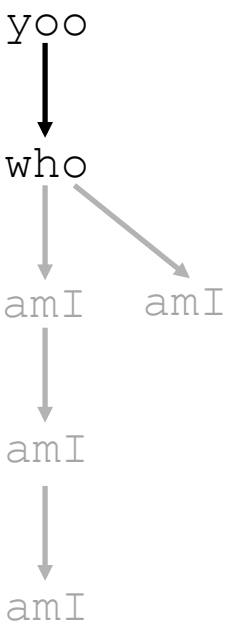

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

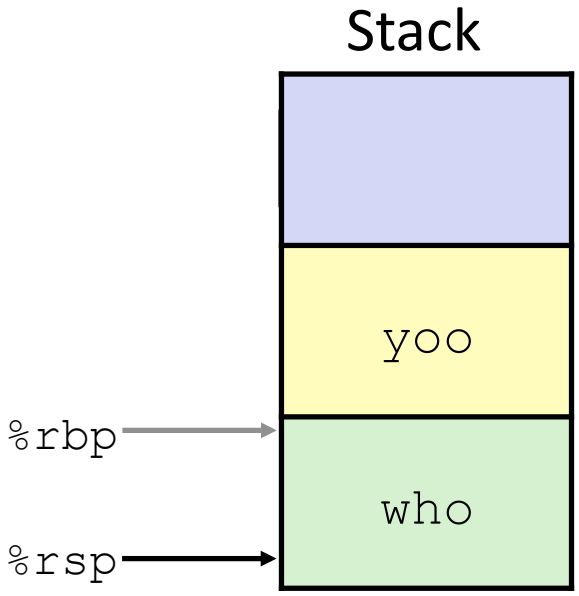

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

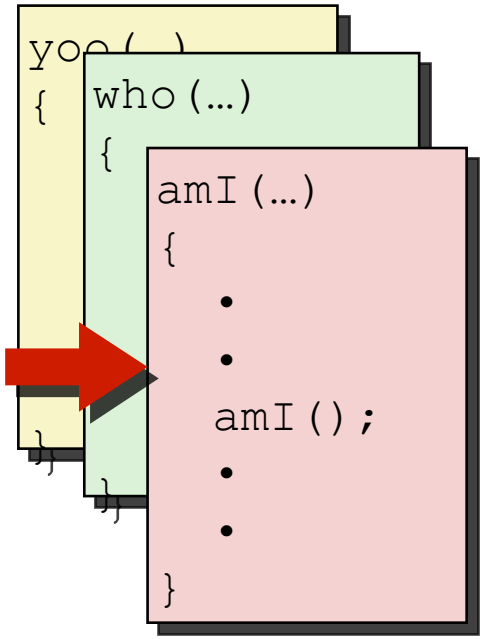
Example

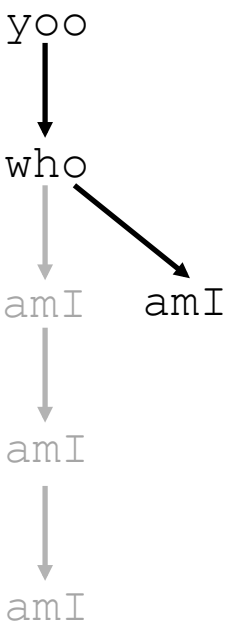


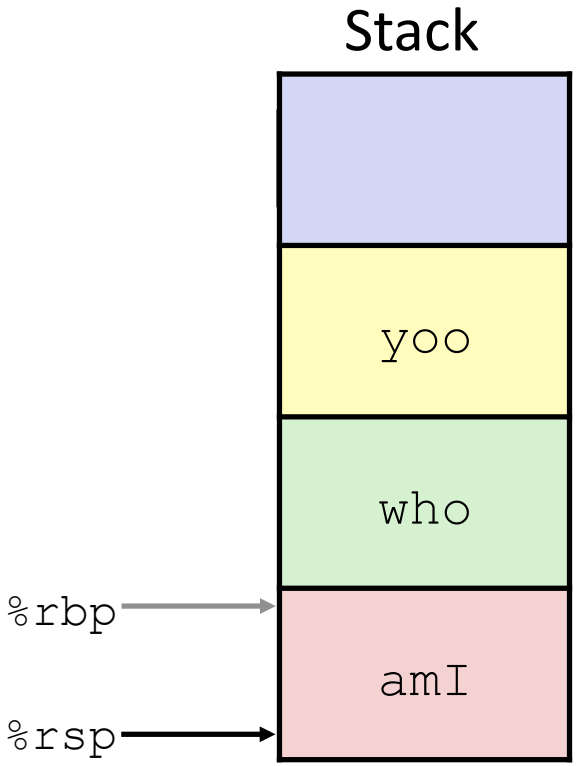




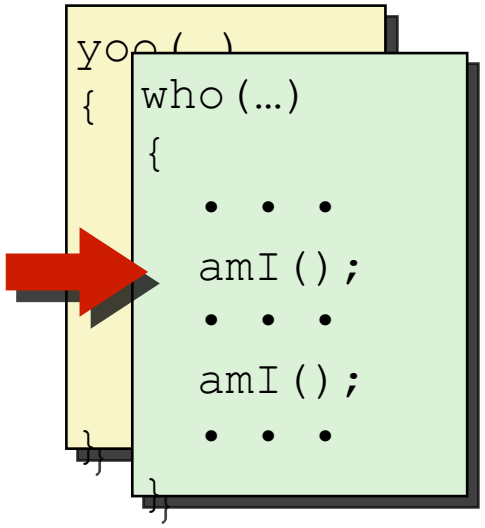
Example

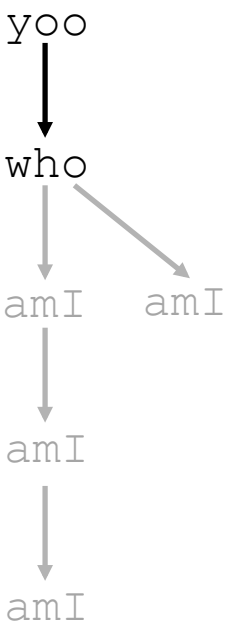

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

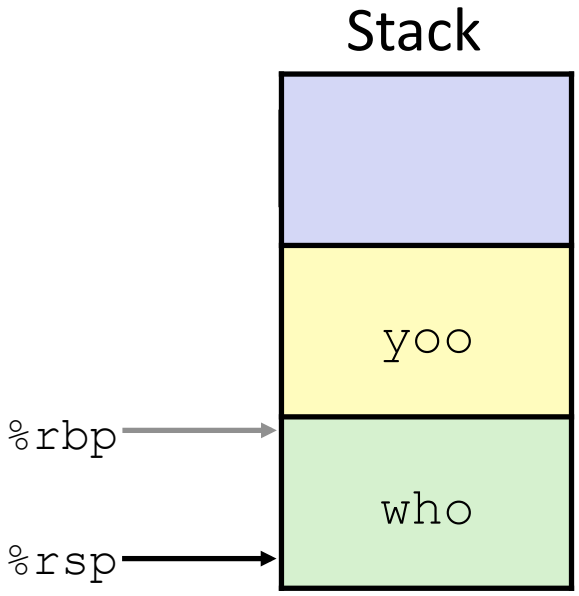

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

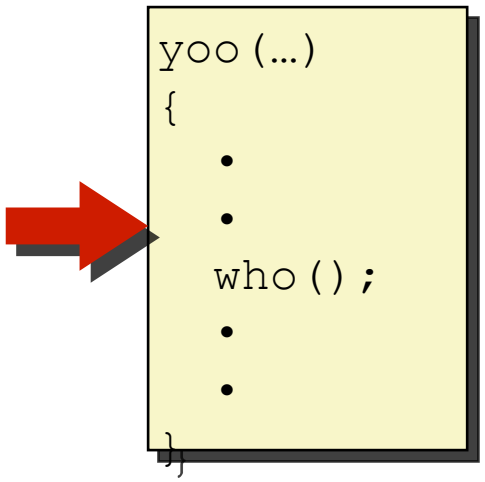
Example

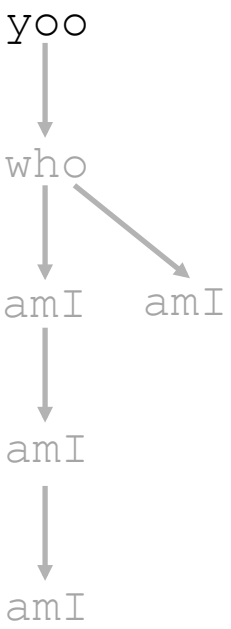


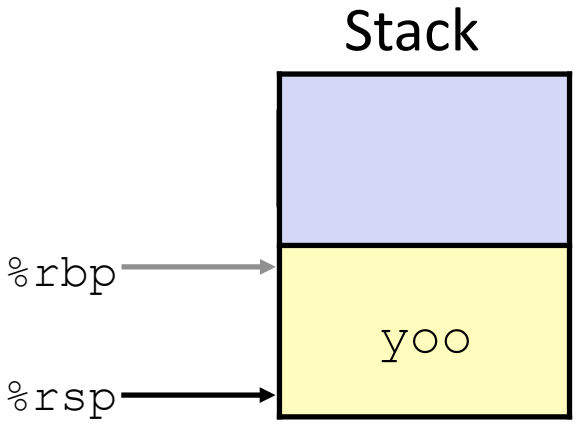




Example



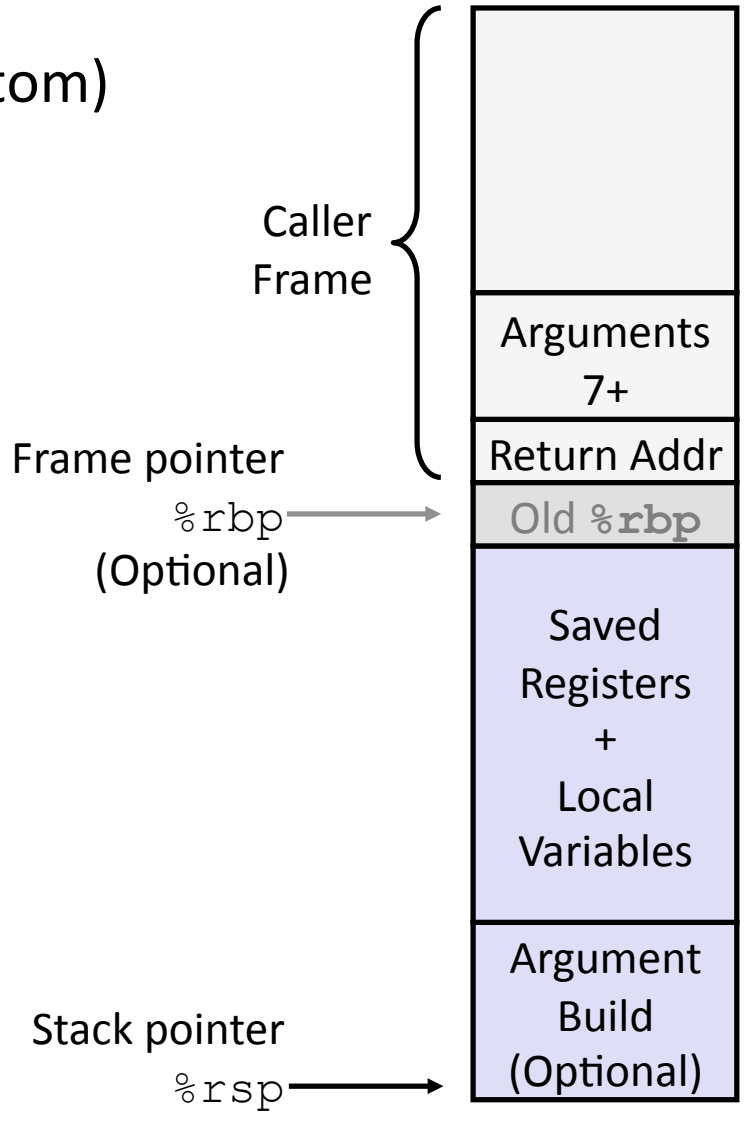




x86-64/Linux Stack Frame

- 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)

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



Example: `incr, x86`

```
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
```

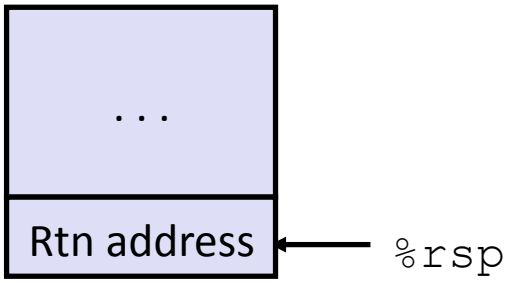
Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

Example: Calling `incr` #1

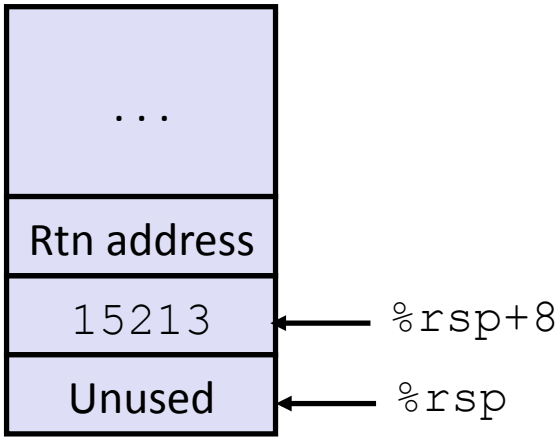
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

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

Initial Stack Structure



Resulting Stack Structure

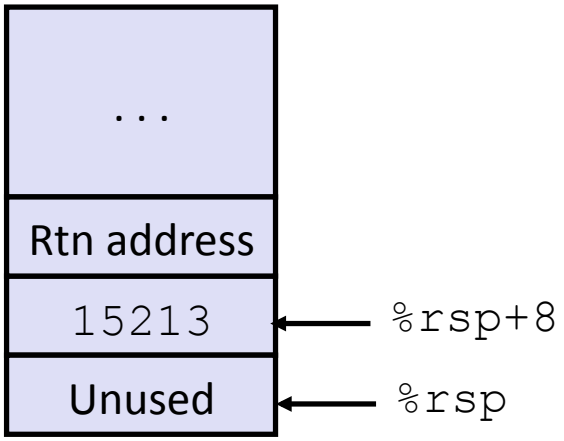


Example: Calling `incr` #2

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

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

Stack Structure



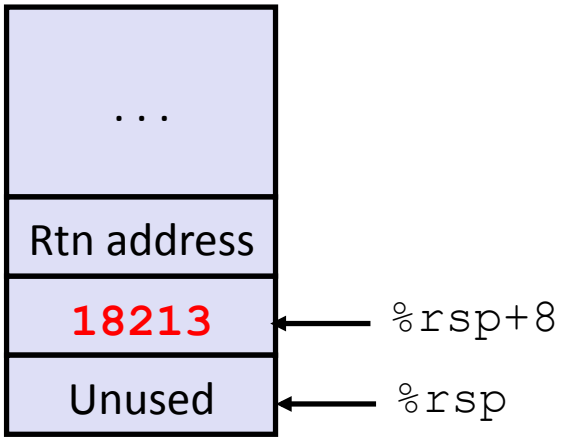
Register	Use(s)
%rdi	&v1
%rsi	3000

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



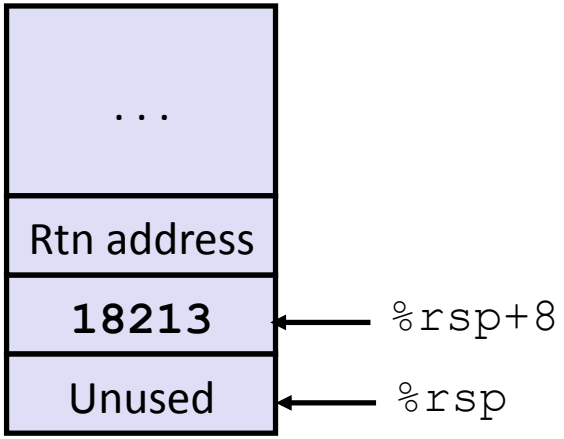
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

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

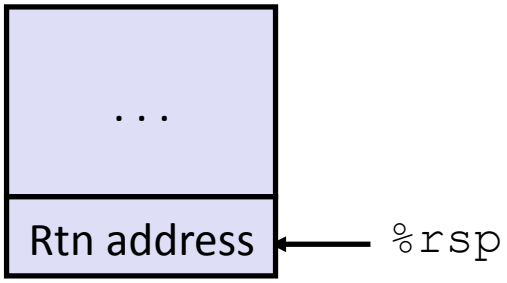
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

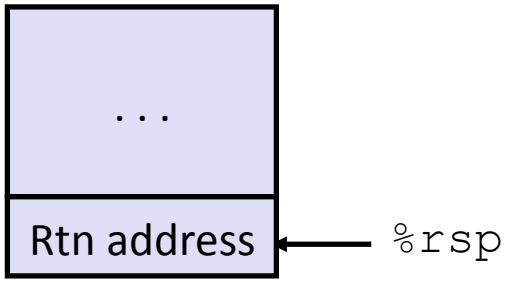
Updated Stack Structure



Example: Calling `incr` #5

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

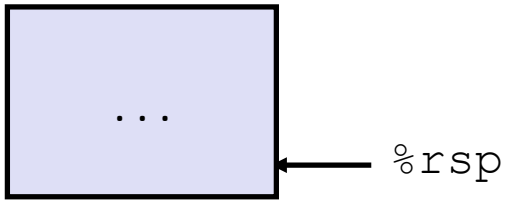
Updated Stack Structure




```
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, x86

- 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
```

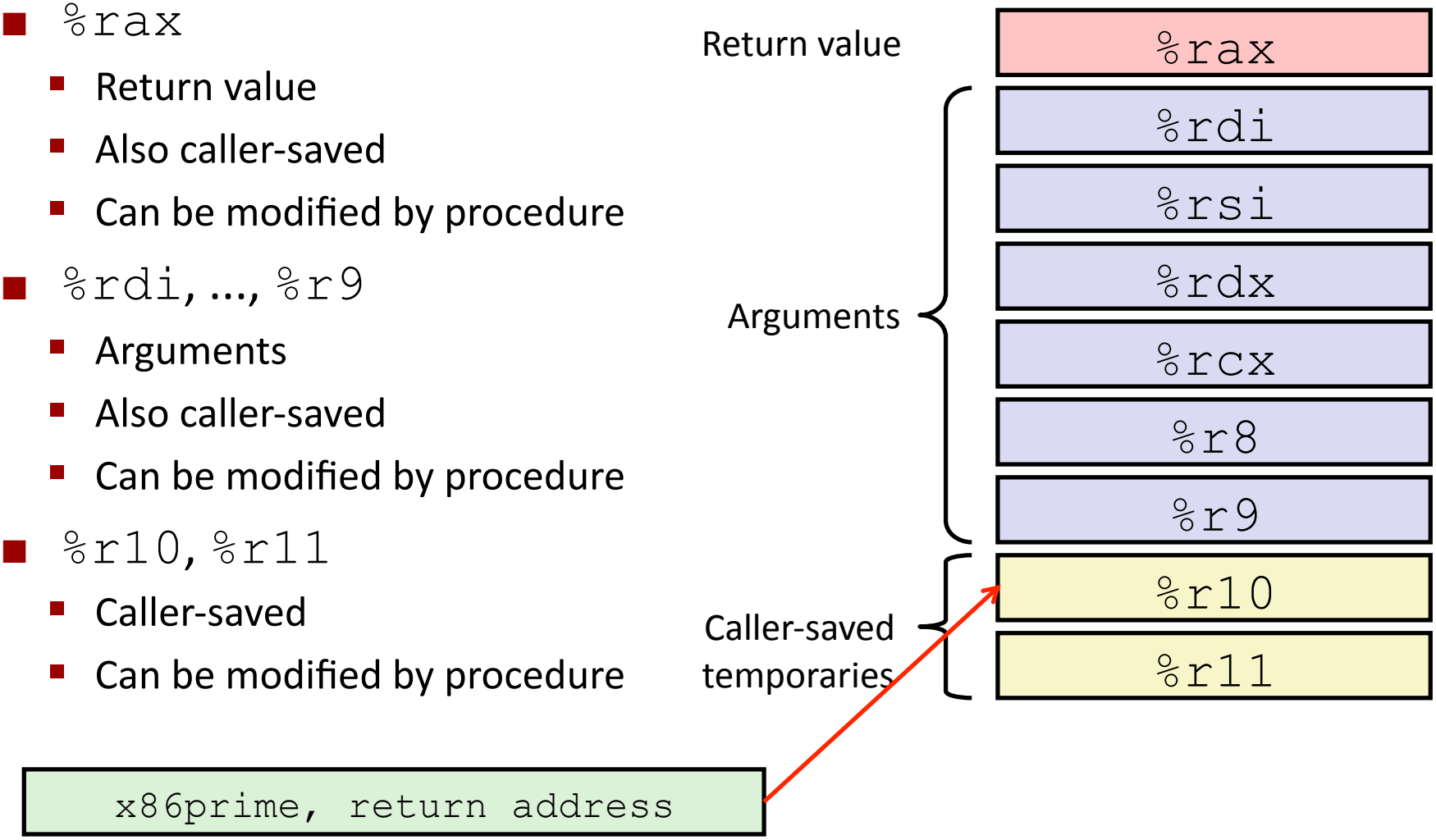
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble ➙ something should be done!
    - Need some coordination

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

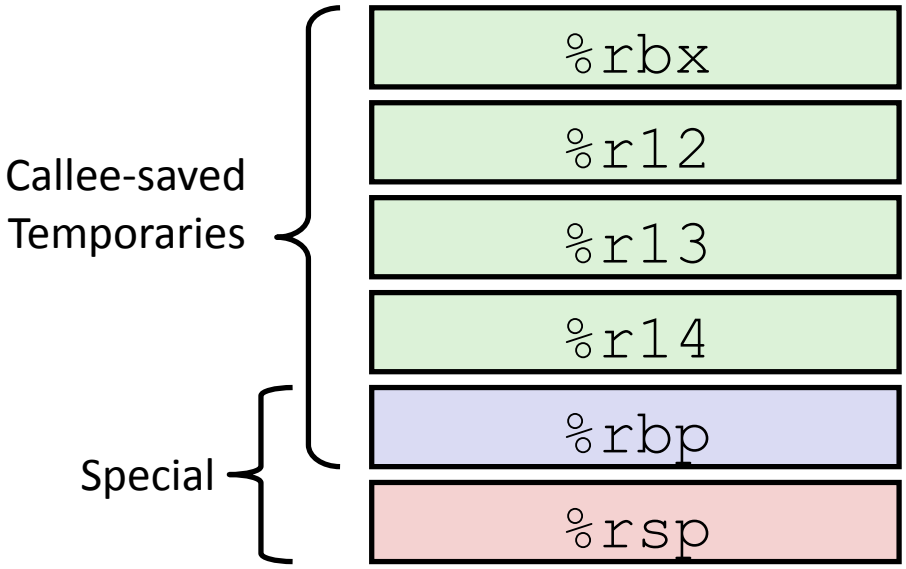
x86-64 Linux Register Usage #1

- %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
  - 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

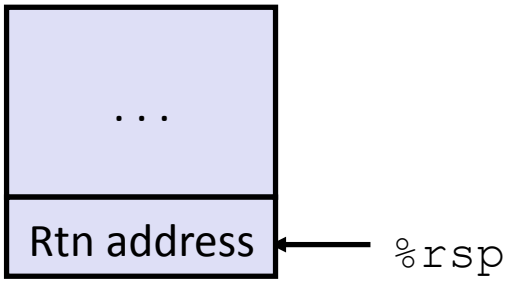


Callee-Saved Example #1, x86

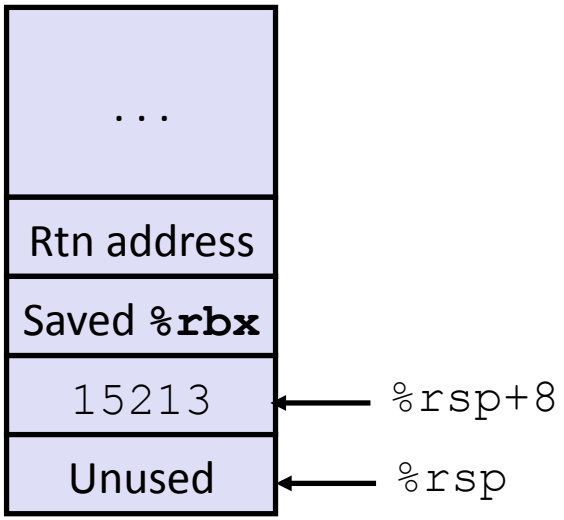
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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

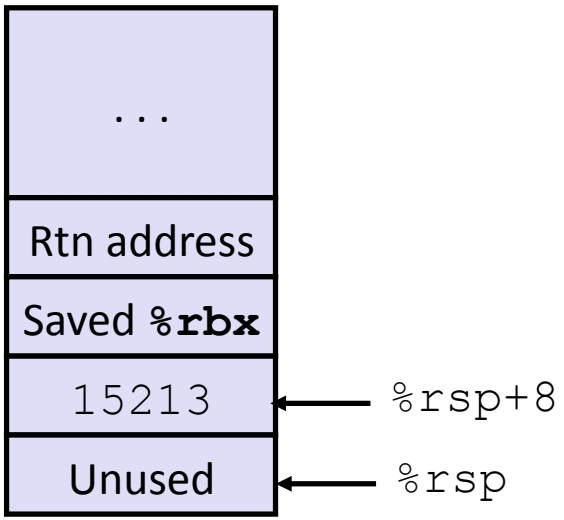


Callee-Saved Example #2

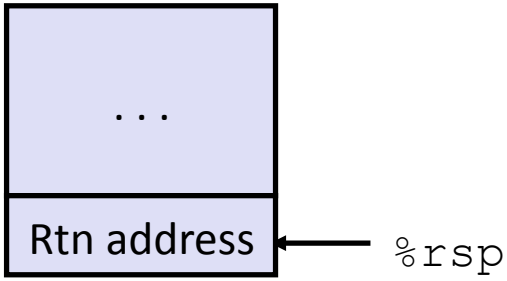
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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



Today

- Control
  - Condition branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - **Illustration of Recursion**
- Arrays and Structures

Recursive Function, x86

```
/* 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
```

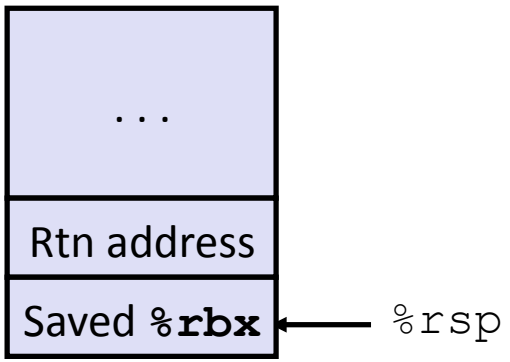
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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
```

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
```

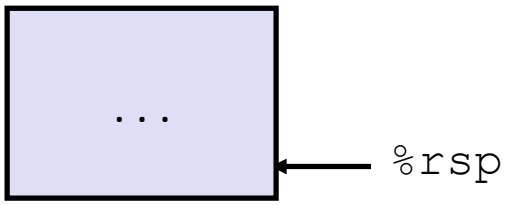
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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
.L6:
   rep; ret
```

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



Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P

Today

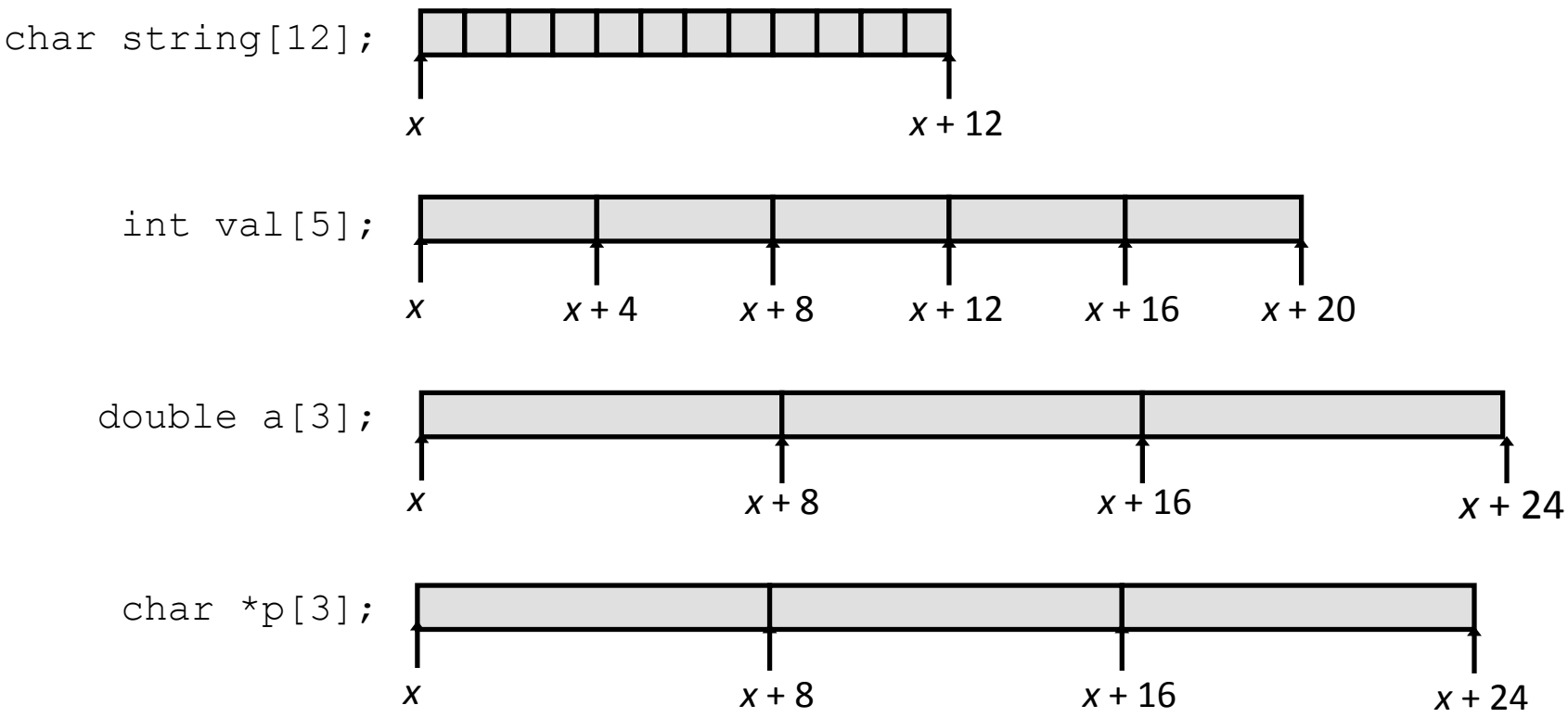
- Control
  - Condition branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
- **Arrays and Structures**

Array Allocation

- Basic Principle

  T **`A[`***L***`];`**

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory

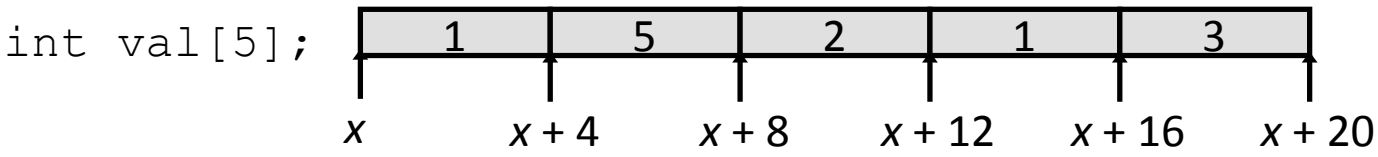


Array Access

- Basic Principle

  T **A[***L***];**

  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

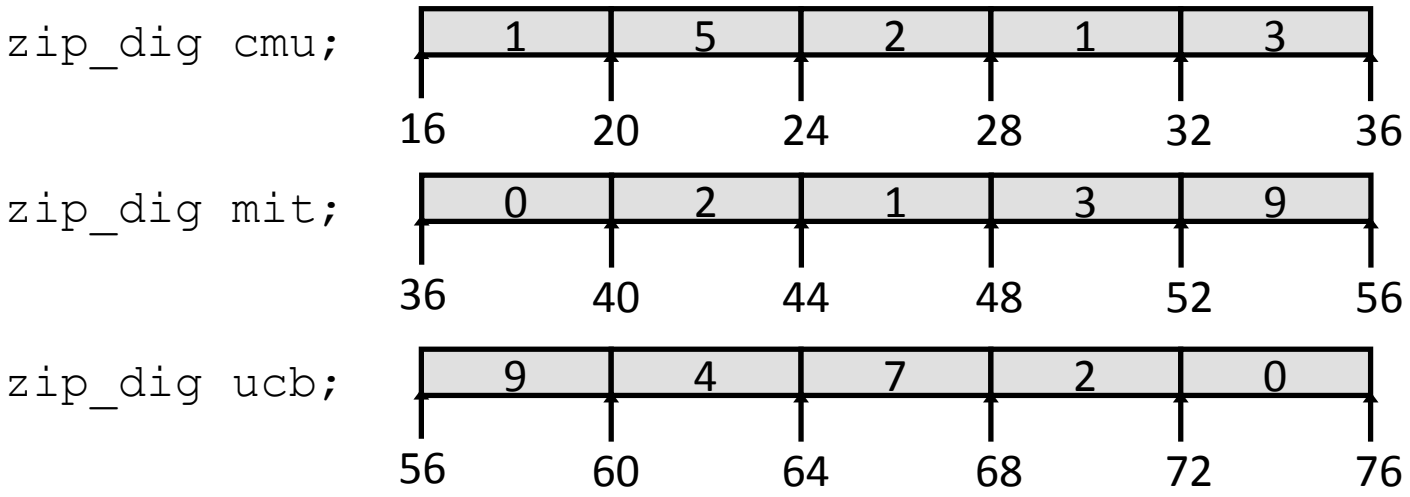


- Reference Type Value

Reference	Type	Value
val[4]	**int**	3
val	**int ***	*x*
val+1	**int ***	*x* + 4
&val[2]	**int ***	*x* + 8
val[5]	**int**	??
***(val+1)**	**int**	5
val + *i*	**int ***	*x* + 4 *i*

Array Example

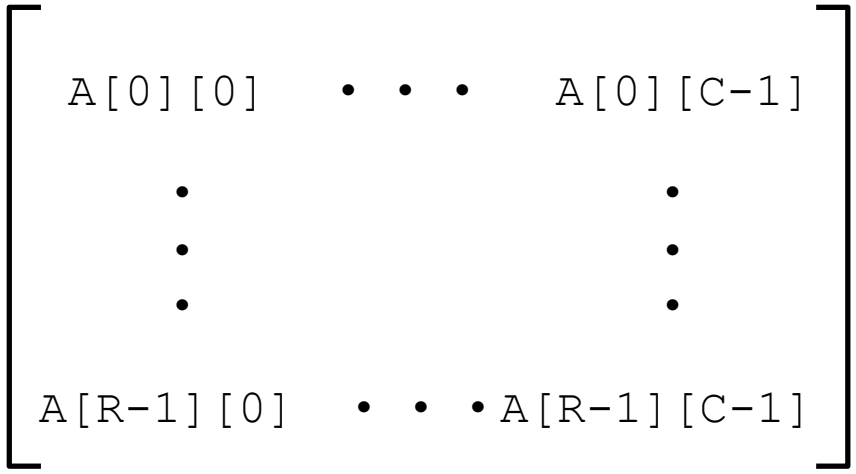
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



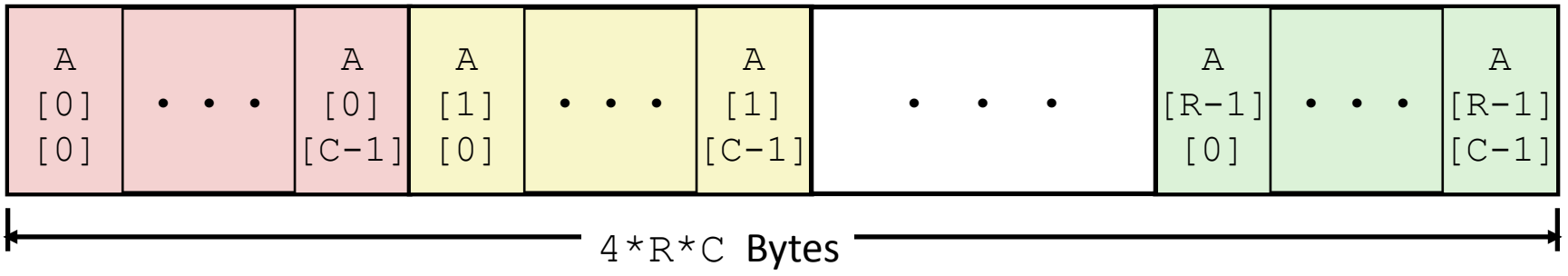
- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Multidimensional (Nested) Arrays

- Declaration
  - *T* **A**[*R*][*C*];
    - 2D array of data type *T*
    - *R* rows, *C* columns
    - Type *T* element requires *K* bytes
- Array Size
  - *R* * *C* * *K* bytes
- Arrangement
  - Row-Major Ordering



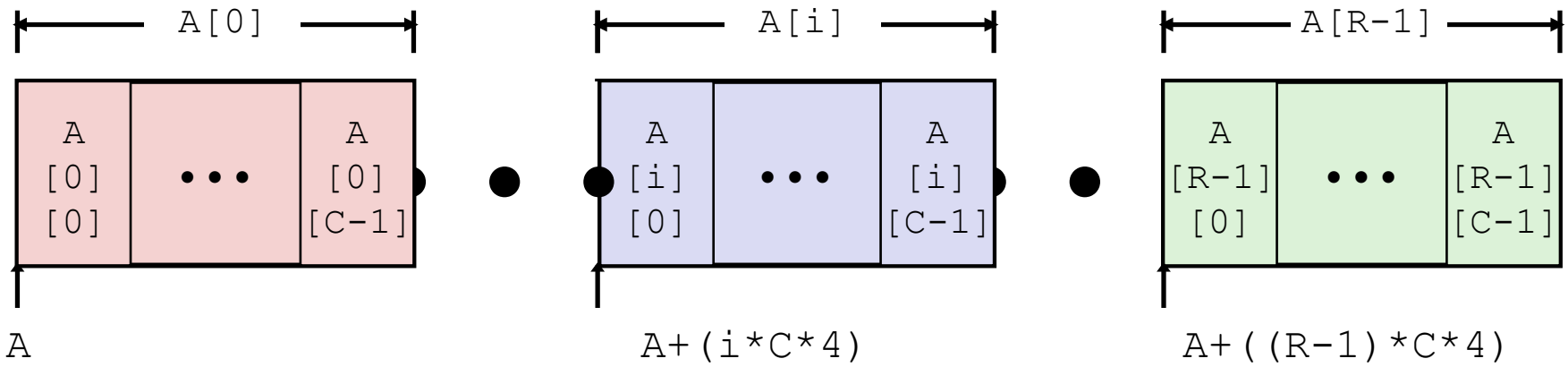
`int A[R][C];`



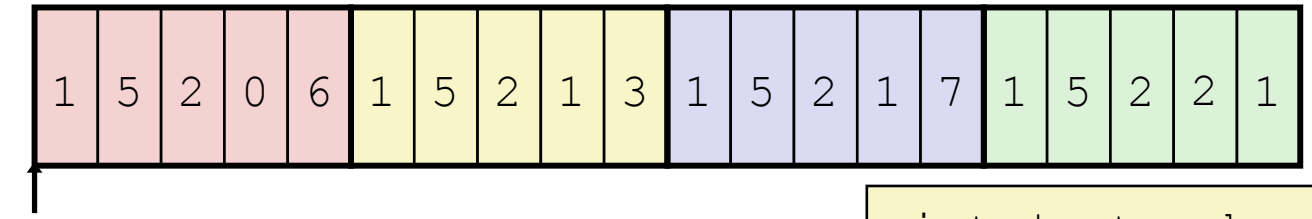
Nested Array Row Access

- Row Vectors
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **`A`** **`+`** *i* * (*C* * *K*)

```
int A[R][C];
```



Nested Array Row Access Code




```
int *get_pgh_zip(int index)
                 {
     return pgh[index];
                 }
```

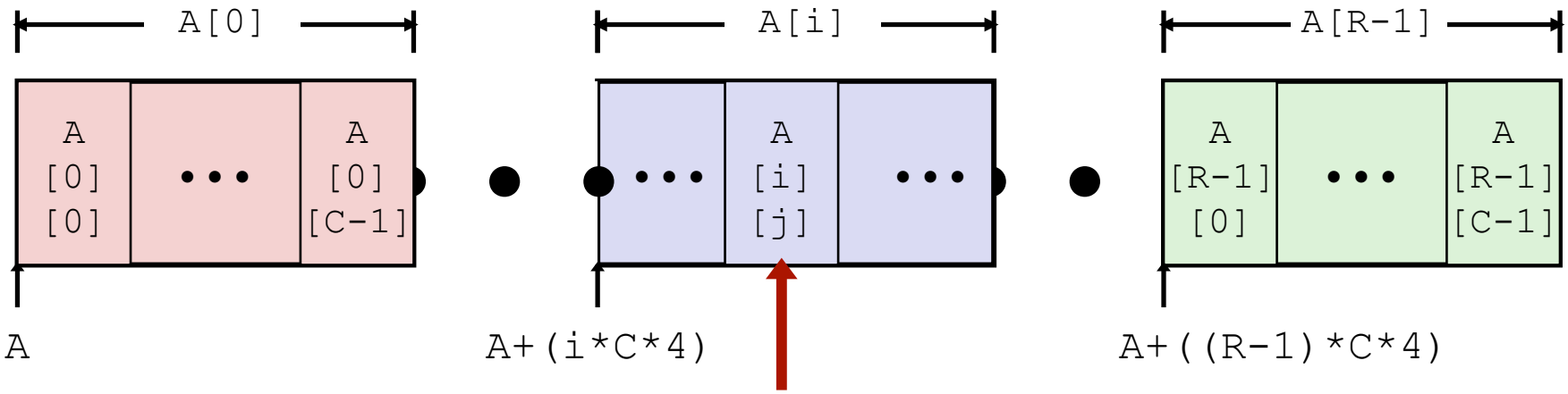
```
                  # %rdi = index
           leaq (%rdi,%rdi,4),%rax     # 5 * index
  leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```

- Row Vector
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- Machine Code
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

Nested Array Element Access

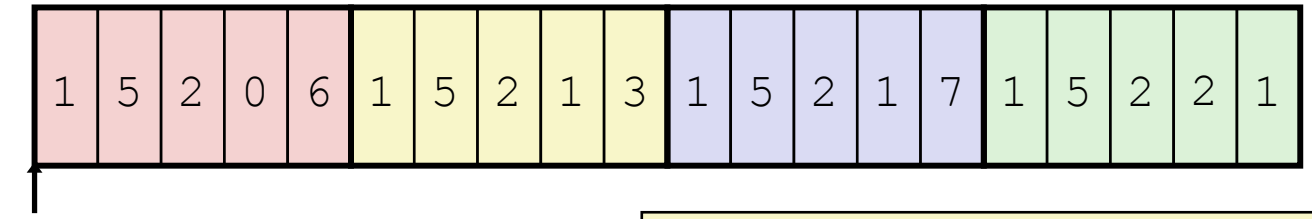
- Array Elements
  - **`A[i][j]`** is element of type *T,* which requires *K* bytes
  - Address **`A`** **`+`** $i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```



A+(i*C*4)+

Nested Array Element Access Code



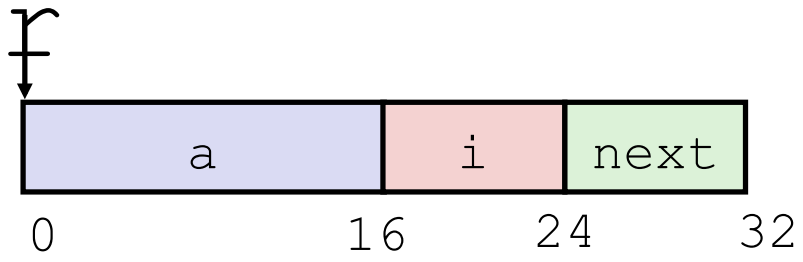

```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

```
                    leaq (%rdi,%rdi,4), %rax # 5*index
                     addl %rax, %rsi   # 5*index+dig
movl  pgh(,%rsi,4), %eax       # M[pgh + 4*(5*index+dig)]
```

- Array Elements
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
    - = **`pgh + 4*(5*index + dig)`**

Structure Representation

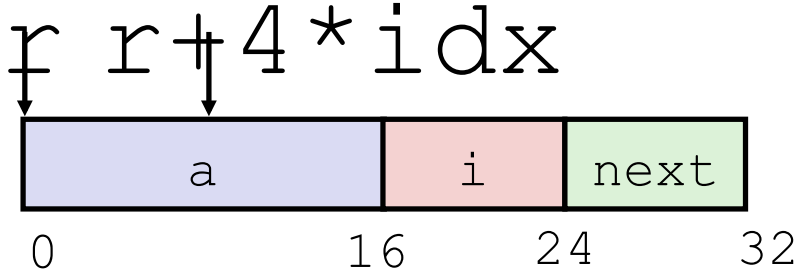
```
struct rec {
    int a[4];
    size_t i;
struct rec *next;
};
```



- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
struct rec *next;
    };
```



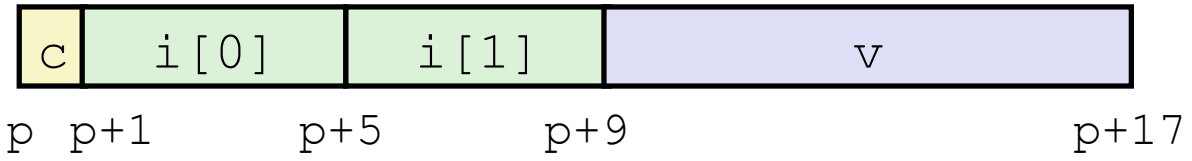
- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

```
int *get_ap
(struct rec *r, size_t idx)
{
return &r->a[idx];
}
```

```
# r in %rdi, idx in %rsi
 leaq  (%rdi,%rsi,4), %rax
 ret
```

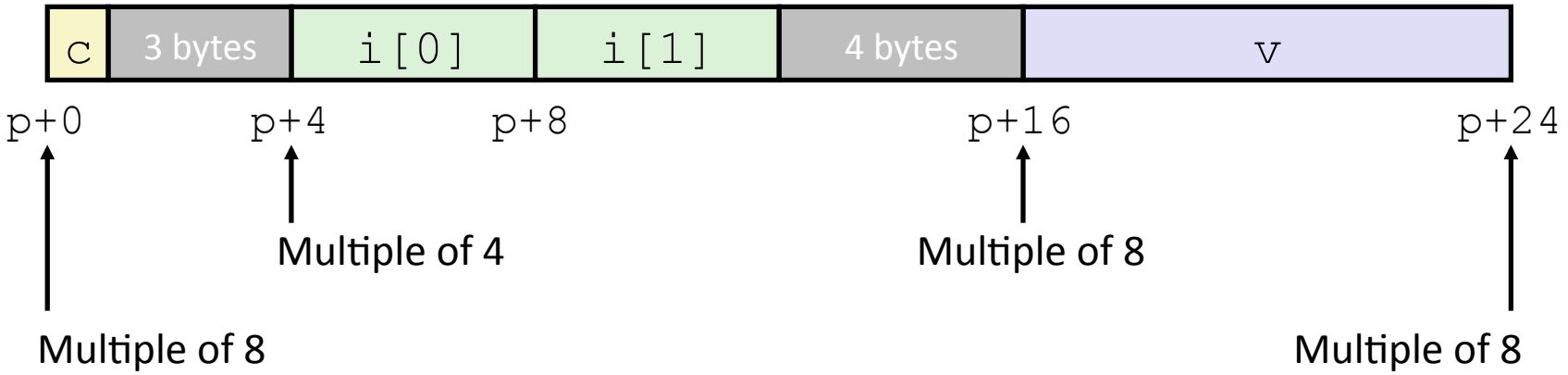
Structures & Alignment

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



Alignment Principles

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (x86-64)

- 1 byte: `char`, …
  - no restrictions on address
- 2 bytes: `short`, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `long`, `char *`, …
  - lowest 3 bits of address must be 000_2
- 16 bytes: `long double` (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Summarizing Control

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

Today

- Control
  - Condition branches (the x86prime way)
  - Conditional codes (the x86 way)
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
- Arrays and Structures

x86-64 Procedure Summary

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