Comp 2100 Week 8

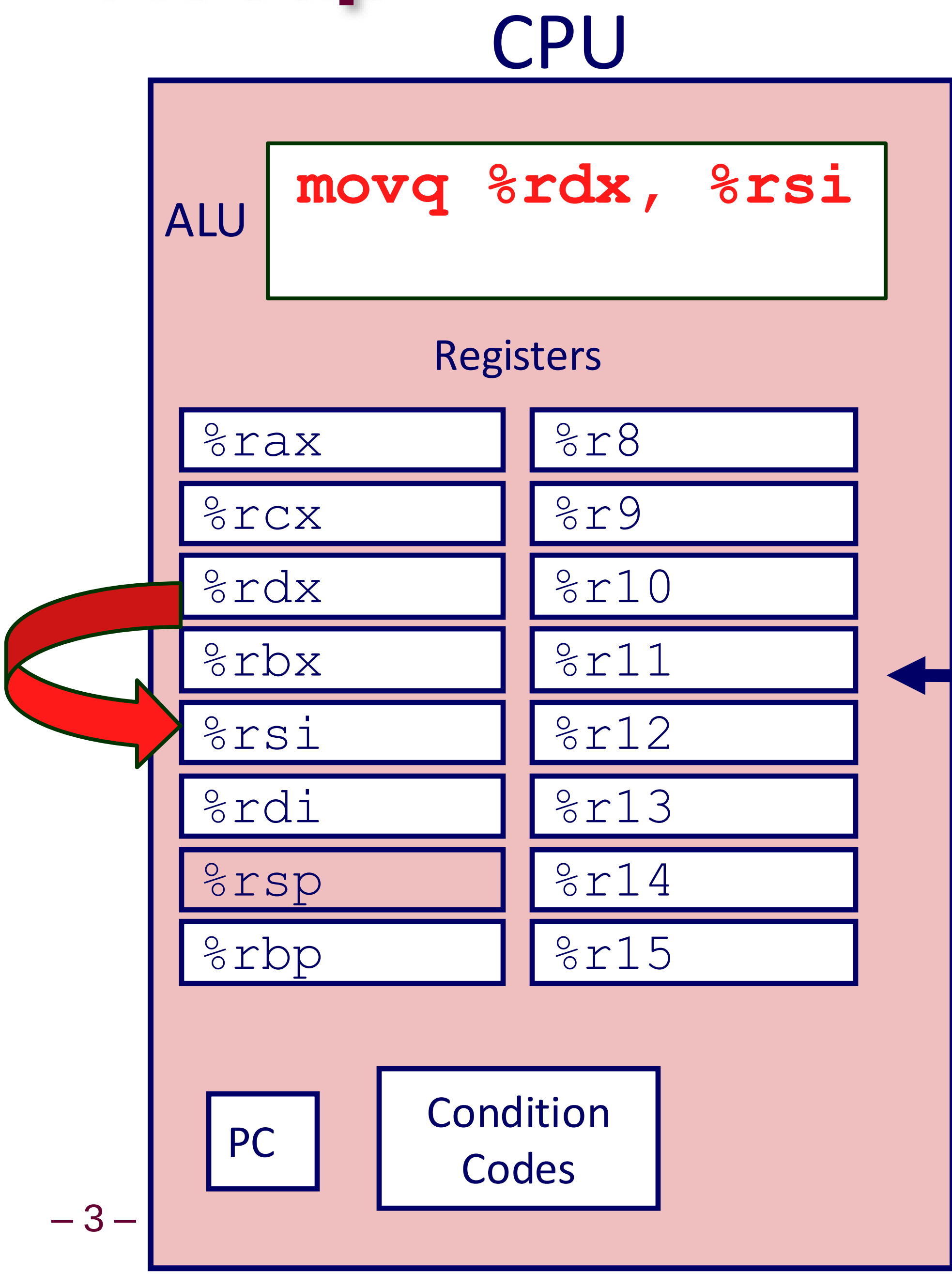
“Control Flow”

Announcements

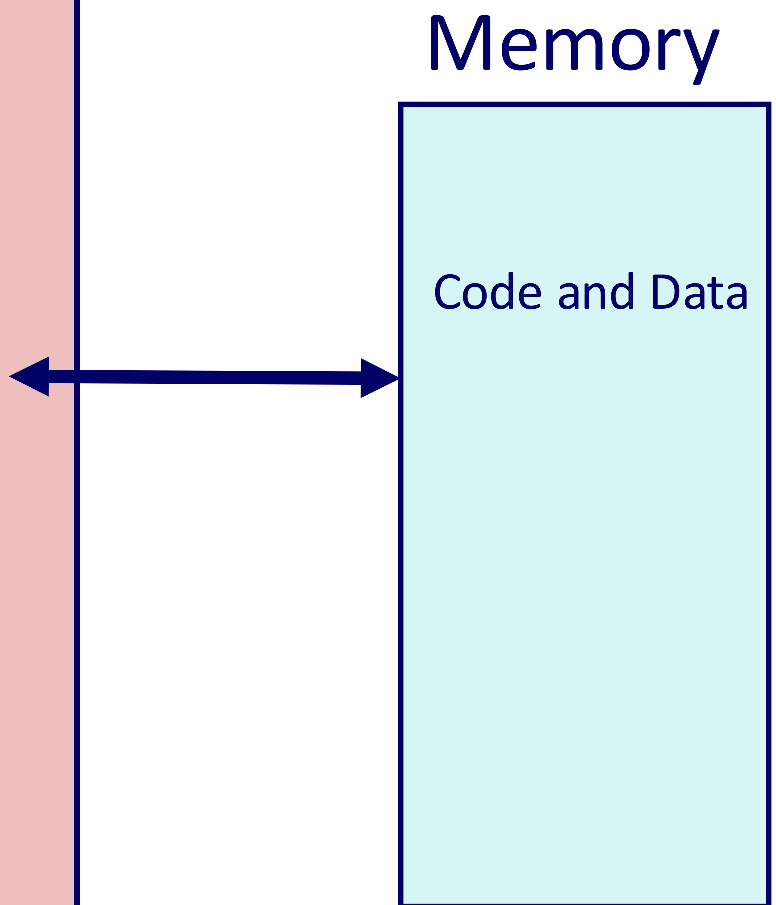
MACQUARIE
University

- **Week 7 and Week 8 Quiz due after break**
- **Bomb Lab #2 on iLearn, 1st phase prog mark due Fri Oct 4**
  - **Tutors will introduce the lab this week in workshops**
  - **Given a personalized binary executable, solve 5 phases**
  - **Learn to read assembly and use the gdb debugger**
  - **Avoid “exploding” the bomb by providing right password at each level,**
  - **set a breakpoint to avoid triggering the bomb message to server**
- **Read Chapter 3.1-3.12 (except 3.11) and do practice problems**

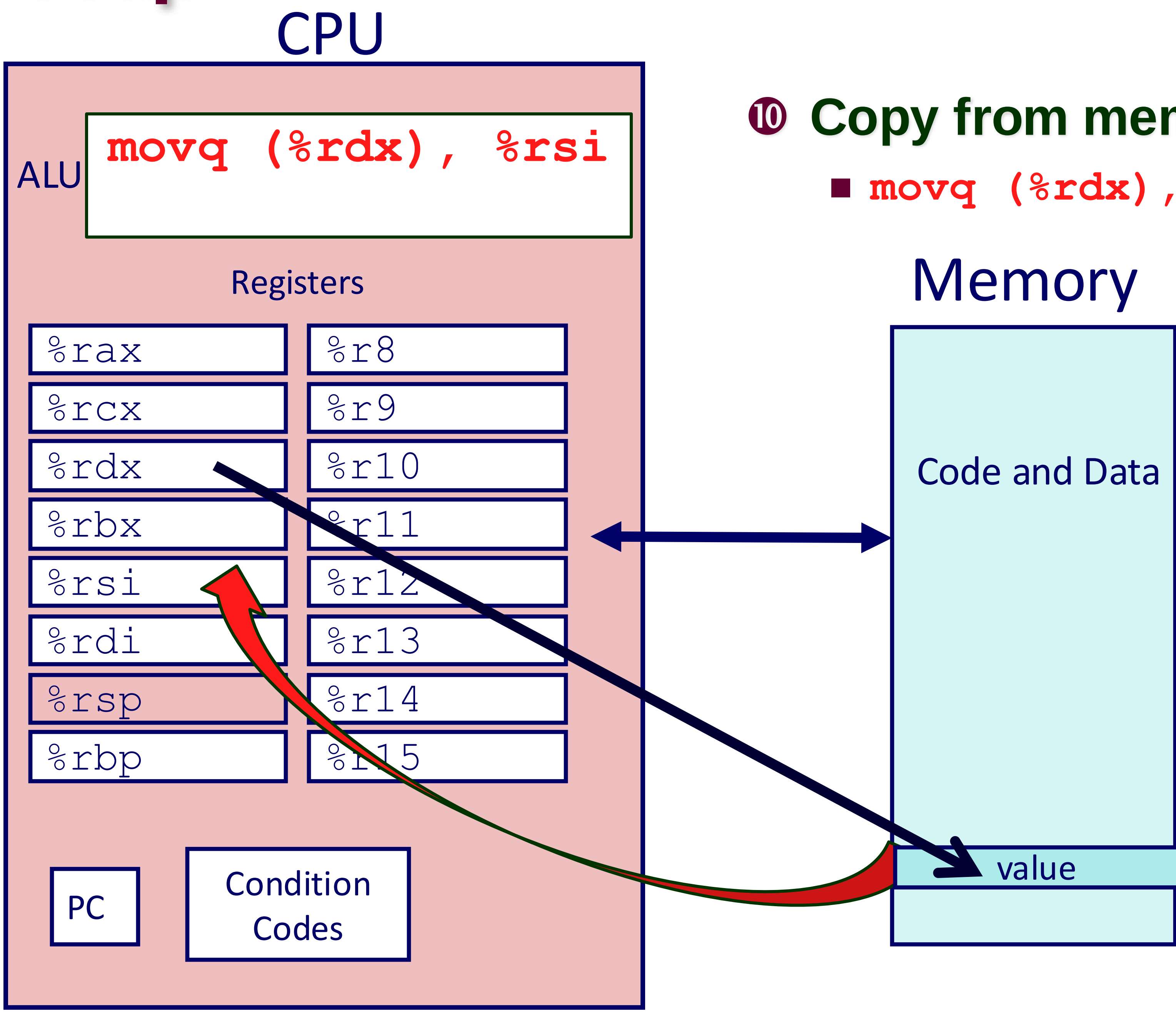
Recap



- ➓ **Introduced move instruction**
  - `movq %rdx, %rsi`



Recap



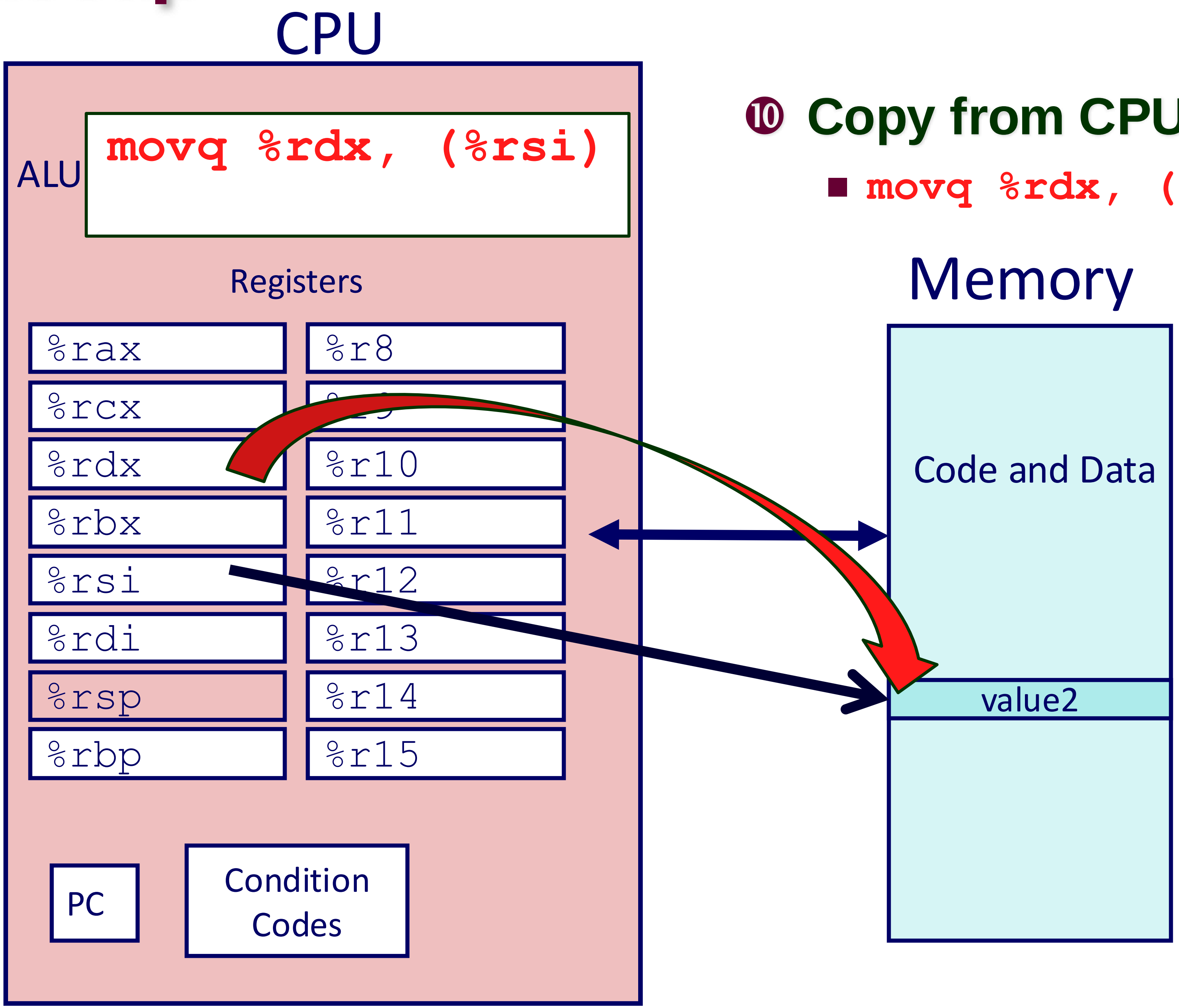
❿ Copy from memory to CPU

- `movq (%rdx), %rsi`

Recap

❿ **Copy from CPU to memory:**

- `movq %rdx, (%rsi)`



Recap

- **Arithmetic instructions: add, sub, and, imul, shr, sal, …**
  - `subq %rdx, %rbx`
- **Conditional branching for if-then-else and for/while loops**
  - `cmpq $5,%rdx` **<<-- generates condition codes ZF, OF, SF, CF**
  - `jge 0xFFD078` **<<-- inspects condition codes**
    - jge compares the 2nd argument (%rdx) of the preceding cmp assembly line to the 1st argument ($5), and will jump if the 2nd argument is greater than or equal to the 1st argument, i.e. %rdx > $5

Conditional Branch

"...Two roads diverged in a wood, and I—

I took the one less traveled by,

And that has made all the difference".

– Robert Frost, from "The Road Not Taken"

Jumping

- jX Instructions

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)

Jumping - Unconditional

- `jmp` **Label**
  - will encode the Label (address to jump to) as part of the jump instruction at compile time
- `jmp` ***%rax**
  - uses the value in register %rax at run time as the jump target
- `jmp` ***(%rax)**
  - reads the jump target from memory, using the value in %rax as the read address.
  - Used to implement switch statements – see later slides
- **There is no jmp %rax or jmp (%rax) – compiler won't generate this syntax.**

Conditional Branch Example (Jump Style)

MACQUARIE University

```
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:                  # x: rdi   y: rsi
   cmpq   %rsi, %rdi # flags = x ?? y
   jle    .L4             # if flags.le goto L4
                          # if x <= y goto L4
   movq   %rdi, %rax # rax = x
   subq   %rsi, %rax # rax -= y
   ret                    # return x - y
.L4:                      #L4: x <= y
   movq   %rsi, %rax # rax = y
   subq   %rdi, %rax # rax -= x
   ret                    # return y - x
```

code was produced with

`$gcc -O1 -S -fno-if-conversion absdiff.c`

is_ok example

```c
long is_ok(char *line)
{
  if (line[0] == 'o' && line[1] == 'k')
    return 1;
  return 0;
}
```

```
…<is_ok>:
 0:…  mov     $0x0,%eax           # eax = 0 {rax = 0}
 5:…  cmpb    $0x6f,(%rdi)        # *rdi ? 0x6f 'o'
 8:…  jne     14 <is_ok+0x14>     # jump !=
 a:…  cmpb    $0x6b,0x1(%rdi)     # rdi[1] ? 0x6b 'k'
 e:…  sete    %al                 # al = rdi[1]==0x6b
11:…  movzbl  %al,%eax            # eax=(unsigned)al
14:…  repz retq                   # return
```

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 negtest = x <= y;
    if (negtest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Conditional Branch Example (Jump Style)

$gcc -O1 -S -fno-if-conversion absdiff.c

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

```
absdiff:                    # x: rdi   y: rsi
   cmpq   %rsi, %rdi  # flags = x ?? y
   jle    .L4         # if flags.le goto L4
                      # if x <= y goto L4
   movq   %rdi, %rax  # rax = x
   subq   %rsi, %rax  # rax -= y
   ret                # return x - y
.L4:                  #L4: x <= y
   movq   %rsi, %rax  # rax = y
   subq   %rdi, %rax  # rax -= x
   ret                # return y - x
```

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

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

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

Conditional Move

- ❿ **cmovX src, dest**
  - *Set dest=src only if condition X holds*
  - **More efficient than conditional branching for highly pipelined processors – easier to guess the next instruction to execute**
  - **But overhead: both branches are evaluated**

cmovX	Condition	Description
`cmove`	`ZF`	Equal / Zero
`cmovne`	`~ZF`	Not Equal / Not Zero
`cmovs`	`SF`	Negative
`cmovns`	`~SF`	Nonnegative
`cmovg`	`~(SF^OF)&~ZF`	Greater (Signed)
`cmovge`	`~(SF^OF)`	Greater or Equal (Signed)
`cmovl`	`(SF^OF)`	Less (Signed)
`cmovle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`cmova`	`~CF&~ZF`	Above (unsigned)
`cmovb`	`CF`	Below (unsigned)

Using Conditional Moves

MACQUARIE University

- 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;
elseval = Else_Expr;
negtest = !Test;
if (negtest) result = elseval;
return result;
```

Conditional Move Example

MACQUARIE University

```
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  # rax = x
   subq    %rsi, %rax  # rax = x-y
   movq    %rsi, %rdx  # rdx = y
   subq    %rdi, %rdx  # rdx = y-x
   cmpq    %rsi, %rdi  # flags = x ?? y
   cmovle  %rdx, %rax  # if flags.le, rax = rdx
   ret                 # return rax
```

Bad Cases for Conditional Move

Expensive Computations

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

- Both values would get computed
- Expensive computation

Risky Computations

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

- Both values would get computed
- May have undesirable effects

Computations with side effects

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

- Both values would get computed
- Side effects violate C semantics so must be side-effect free

Loops

Do while loop
While loop

For loop

Loops in C

```
do {
    body-statement
} while (test-expr);
```

- **Executes body-statement**
- **Then tests expression**
  - **If true, loops back to 'do'**
  - **Else exit**

```
while (test-expr)
{
    body-statement
}
```

- **Tests expression first**
  - **If true, executes body & loops back to 'while'**
  - **Else exit**

```
for(init; test; update) {
    body-statement
}
```

- **Initializes first**
- **If test is true**
  - **Execute body statement**
  - **Execute update & loop back to 'for'**
- **Else exit**

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

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

“Do-While” Loop Compilation

Goto Version

```
long pcount_go(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, %eax      #  result = 0
.L2:                      #loop:
   movq     %rdi, %rdx    #  rdx = x
   andl     $1, %edx      #  rdx &= 0x1
   addq     %rdx, %rax    #  result += rdx
   shrq     %rdi          #  x >>= 1                  // set flags
   jne      .L2           #  if (x != 0) goto loop // flags.ne
   rep; ret               #  return result
```

General "Do-While" Translation

C Code

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

Goto Version

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

Body:

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

"While" Loop Example

C Code

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

Goto Version

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

- Is this code equivalent to the do-while version?
  - Must jump over the loop if test initially fails

General “While” Translation

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:
```

"For" Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
#define WSIZE 8*sizeof(long)
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;
}
```

MACQUARIE University

"For" Loop → While Loop

For Version

```
for (Init; Test; Update )

      Body
```

While Version

```
Init;

while (Test ) {

      Body

      Update;

}
```

"For" Loop to 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 to 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))
    goto done;                            ! Test
loop:
  {                                       Body
    unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
  i++;                                    Update
  if (i < WSIZE)                          Test
    goto loop;
done:
  return result;
}
```

Switch Statement

```
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 Statements

MACQUARIE University

- ❿ **Compact replacement for a long series of if – else if – else if - …**
  - Many if … else if … else if's … are slow to evaluate due to many compares and conditional jumps
  - Instead use a switch statement – good if many contiguous cases and switch argument is simple
    - Contiguous needed to avoid sparse jump table

Switch Statements

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

➓ Special cases

- Multiple case labels: 5 & 6
- Fall through cases: 2
- Missing cases: 4 goes to default

```c
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 Statements

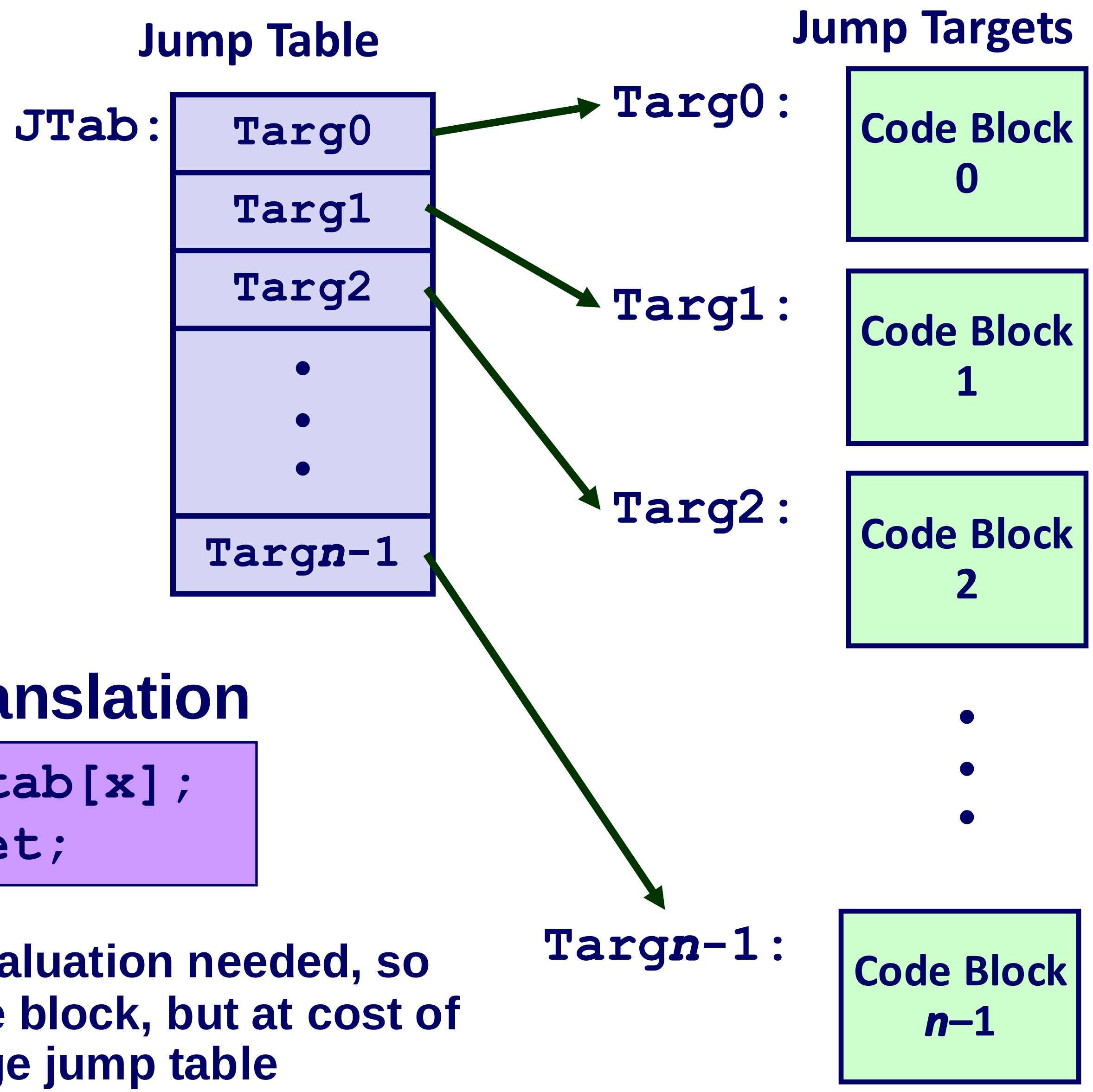
MACQUARIE University

- Implemented as a Jump Table (array of addresses)
  - Index into array and jump to branch target
  - Avoids conditionals
  - Good when cases are small integer constants
- GCC picks one based on case structure

Jump Table Structure

Switch Form

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



Approx. Translation

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

Note: no conditional evaluation needed, so very fast to jump to code block, but at cost of a potentially large jump table

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)
```

Indexes into jump table and jumps to correct code block

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

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             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump

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
```

Assembly Setup Explanation

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

- ❿ **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 $0 \leq x \leq 6$

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
```

MACQUARIE University

Jump Table

Jump table

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

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

Code Blocks (x == 1)

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

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

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

Handling Fall-Through

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

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

```
case 3:
        w = 1;

merge:
        w += z;
```

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

MACQUARIE University

```
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

Sparse Switch Example

```
/* Return x/1111 if x is multiple
   && <= 9999.  -1 otherwise */
int div1111(int x)
{
  switch(x) {
  case    0: return 0;
  case 1111: return 1;
  case 2222: return 2;
  case 3333: return 3;
  case 4444: return 4;
  case 5555: return 5;
  case 6666: return 6;
  case 7777: return 7;
  case 8888: return 8;
  case 9999: return 9;
  default: return -1;
  }
}
```

- Not practical to use jump table
  - Would require 10000 entries!
  - Most of entries in jump table are the same default
- Obvious translation into if-then-else would have maximum of 9 tests in the worst case
  - Can we do better?
  - Yes, use a tree-based approach – logarithmic in # tests (e.g. 4)

Summarizing Control Flow

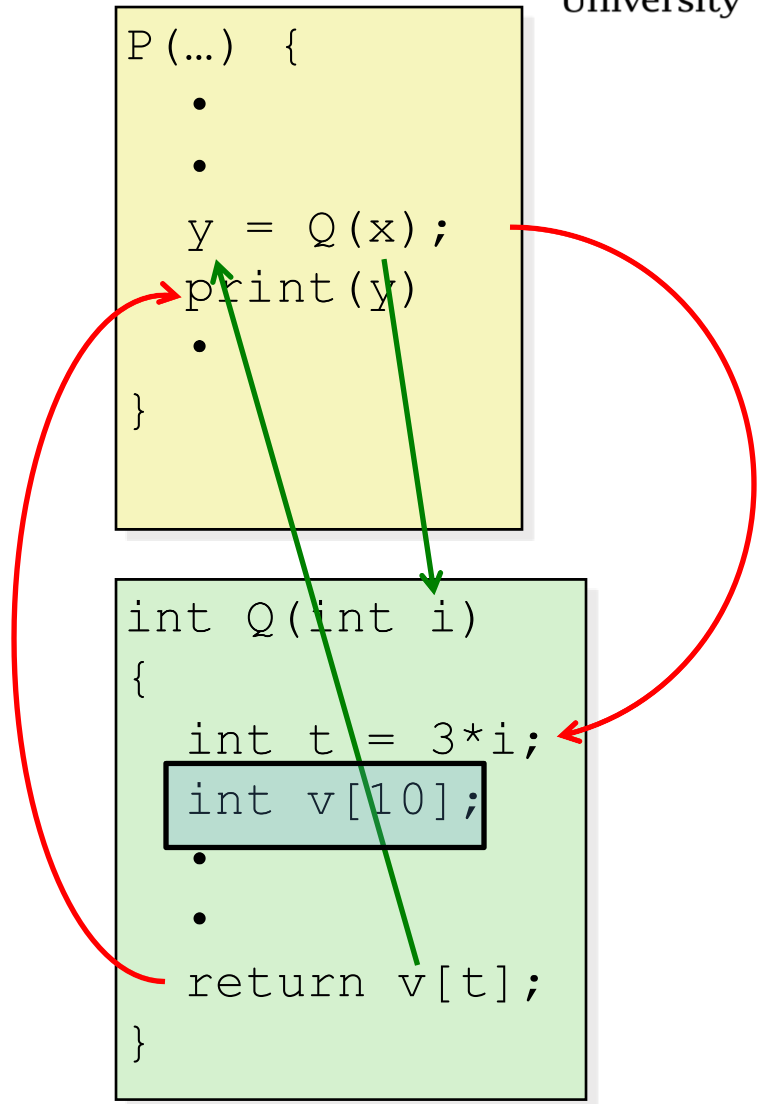
- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code for more complex control
- Standard Techniques
  - Loops converted to do-while form
  - Large switch statements use jump tables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

Procedures

Changes control flow as well

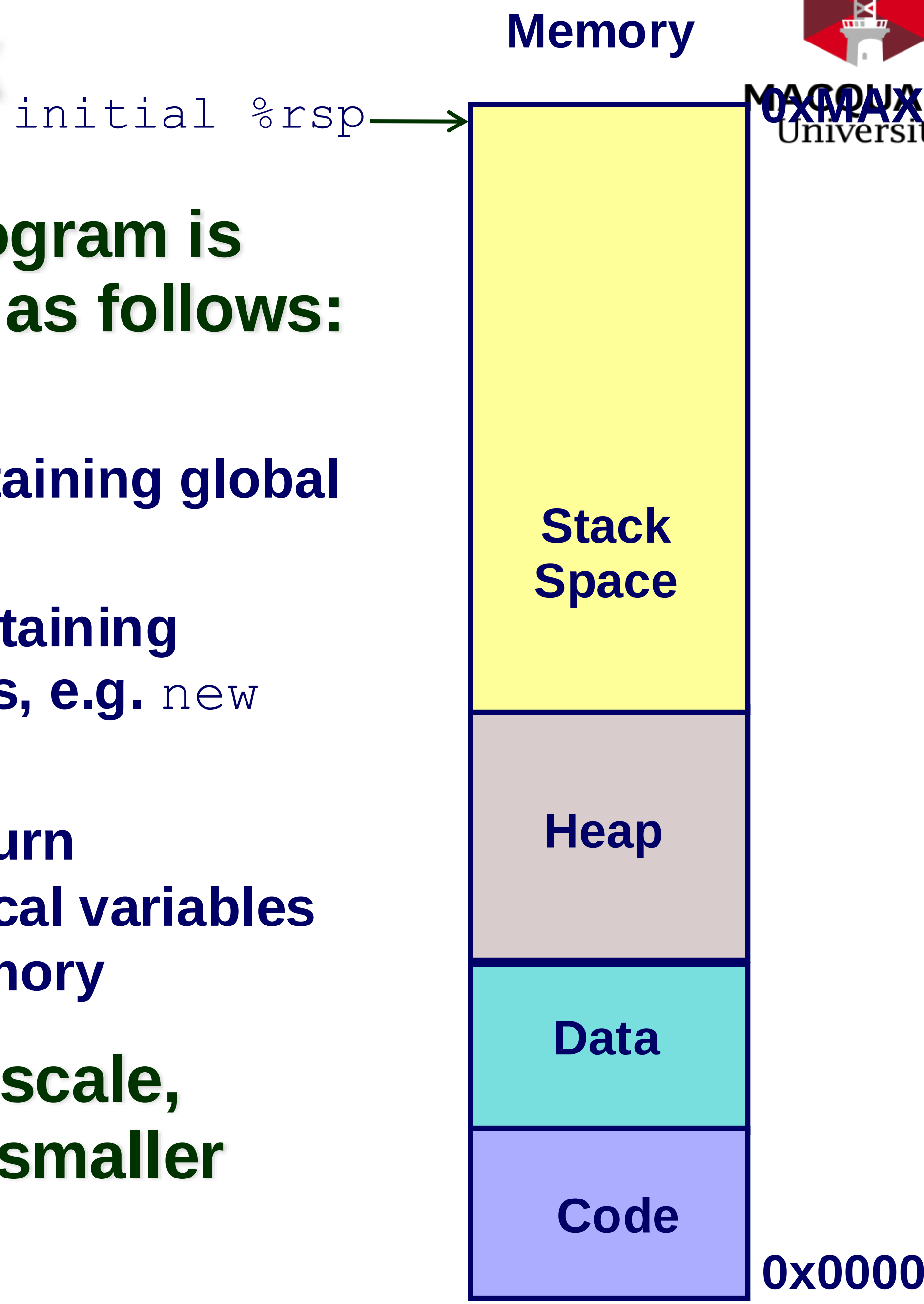
Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Efficient 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



Solution: a Call Stack

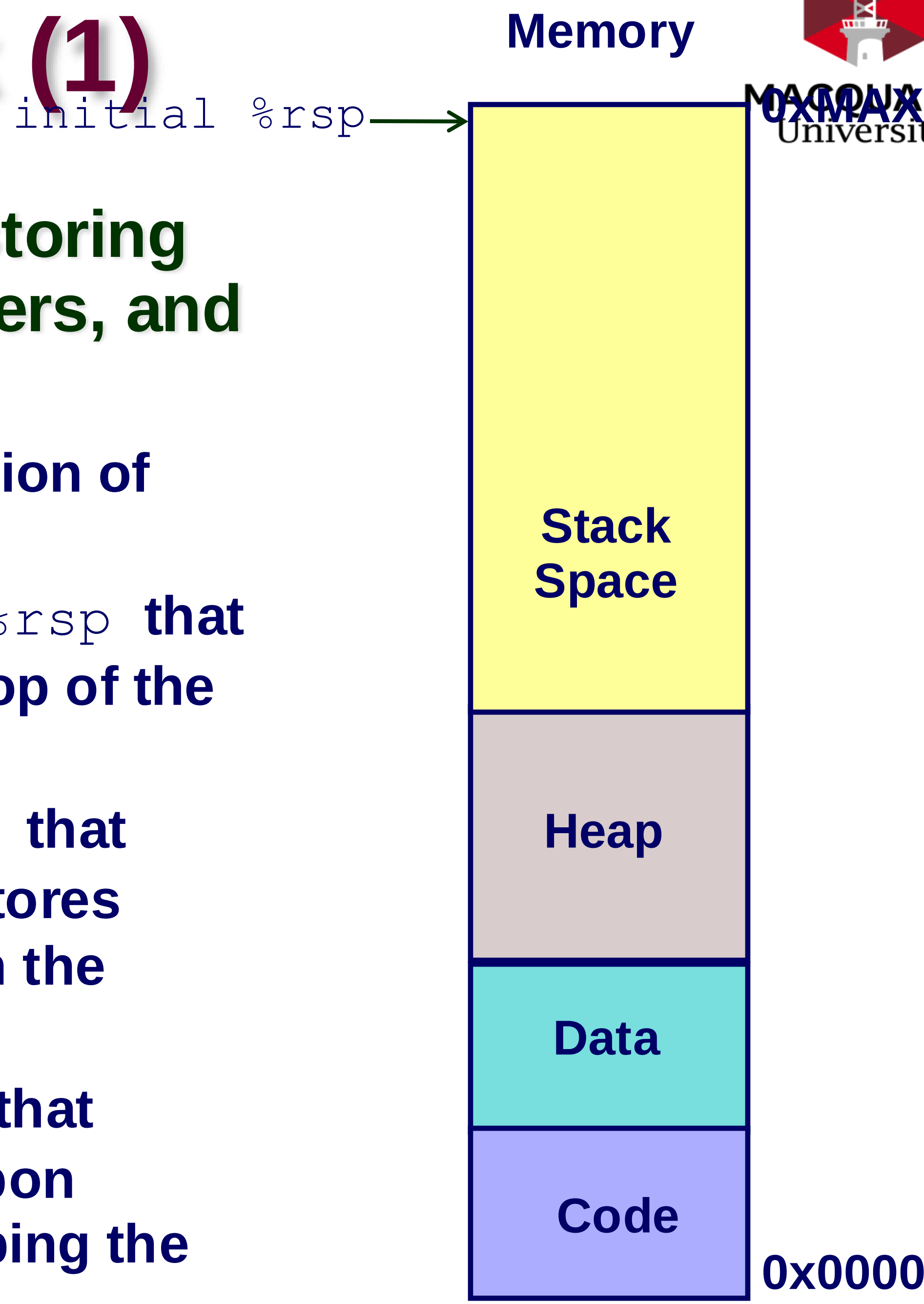
- Your executing software program is typically laid out in memory as follows:
  - Code in low memory
  - A Data section above that containing global variables
  - A Heap section above that containing dynamically allocated variables, e.g. `new` object in Java or C++
  - A Stack section containing return addresses, parameters, and local variables inside functions is in high memory
- Note sections not shown to scale, some could be much larger/smaller than shown



MACQUARIE University

Solution: a Call Stack (1)

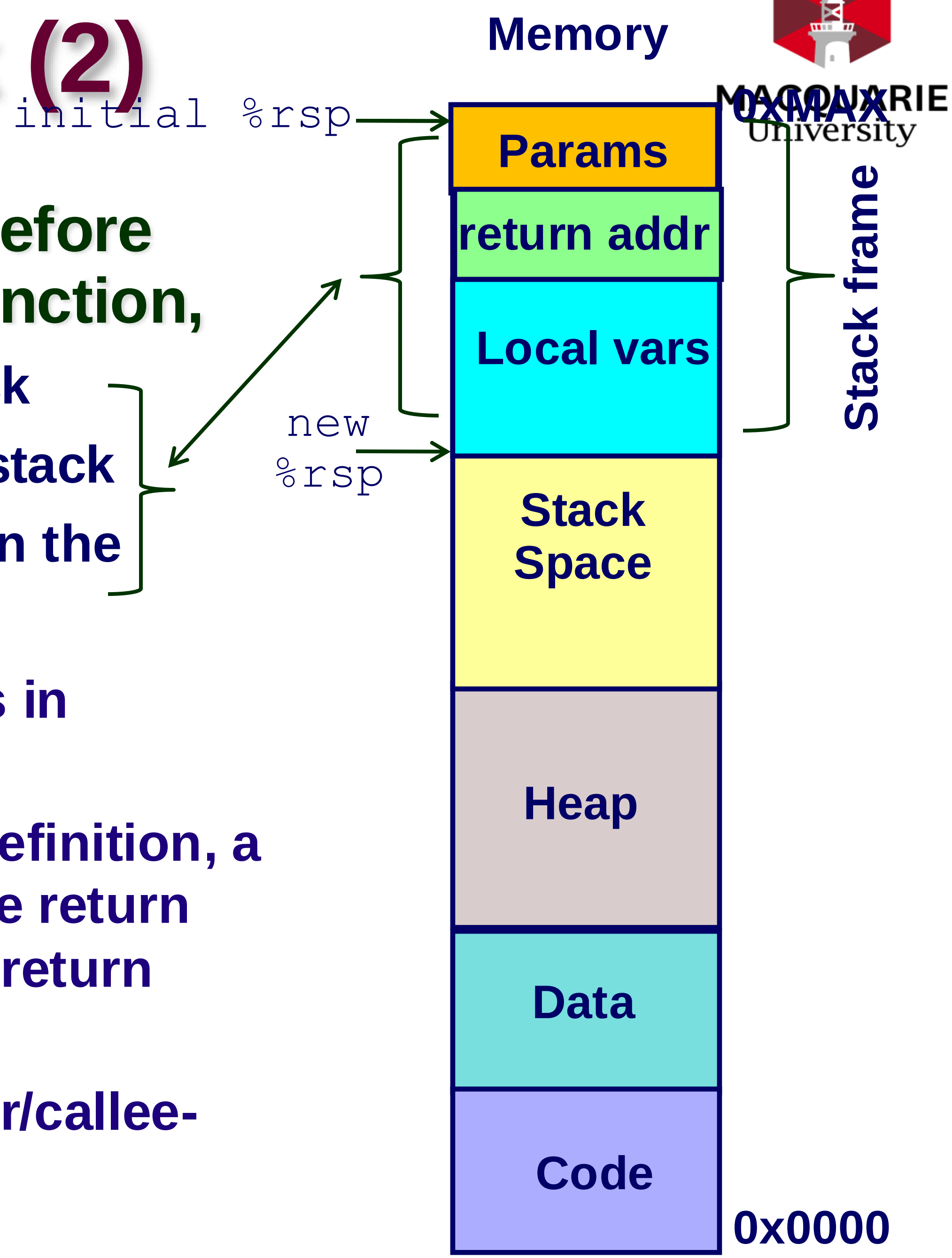
- **Supports function calls by storing the return address, parameters, and local variables on a *stack***
  - **Allocate a special reusable region of memory for the stack**
  - **Create a special CPU register `%rsp` that the stores the location of the top of the stack, i.e. the *stack pointer***
  - **Create a CPU instruction `call` that before jumping to a function stores (pushes) the return address on the stack**
  - **Create a CPU instruction `ret` that jumps to the return address upon exiting from the function, popping the address from the stack**



Solution: a Call Stack (2)

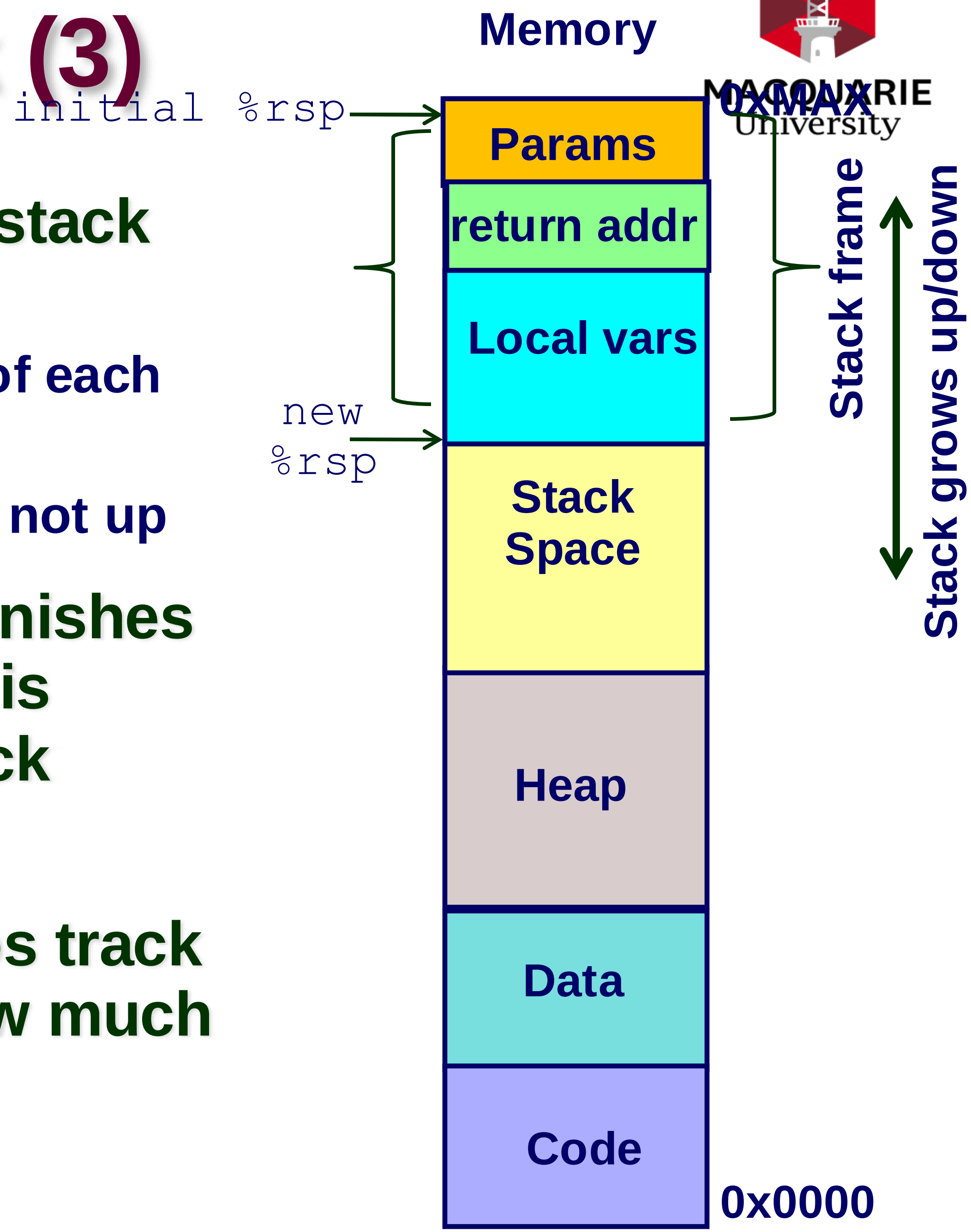
⑩ **On each function call, and before executing the body of the function,**

- **Push parameters onto the stack**
- **Push return address onto the stack**
- **Push/allocate local variables on the stack**
- **Caveat #1: passing parameters in registers is more efficient**
- **Caveat #2: depending on the definition, a stack frame could start with the return address and run until the next return address**
- **Caveat #3: not shown are caller/callee-save registers saved on stack**



Solution: a Call Stack (3)

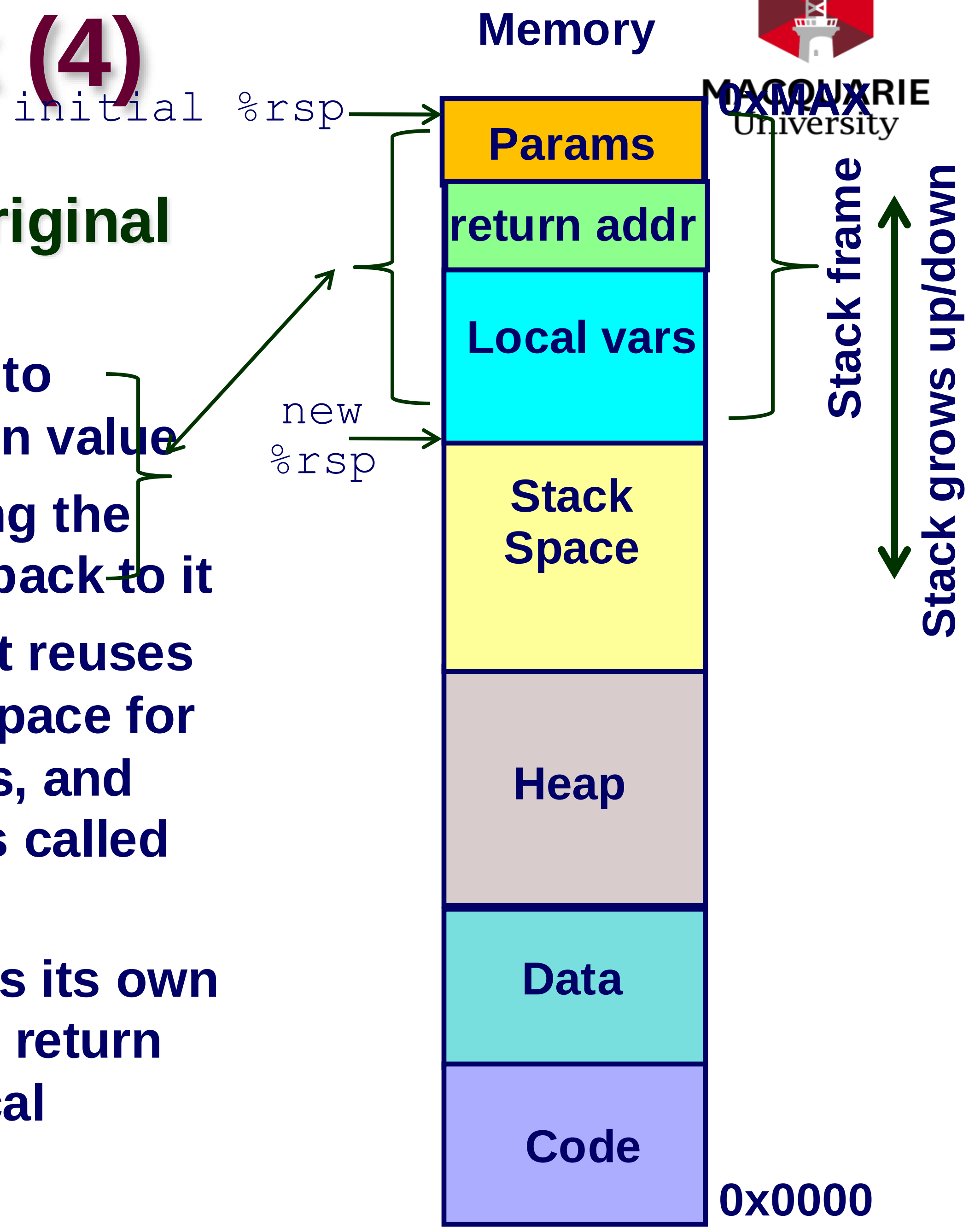
- With each function call, the stack grows further down
  - Stack frames are piled on top of each other
  - Counterintuitive to grow down not up
- When the current function finishes and returns, its stack frame is peeled off the top of the stack
  - Deallocate stack frame
- x86-64 stack discipline keeps track of stack frame sizes and how much to move %rsp up/down
  - Embedded in the assembly,
    - e.g. subq $16, %rsp



Solution: a Call Stack (4)

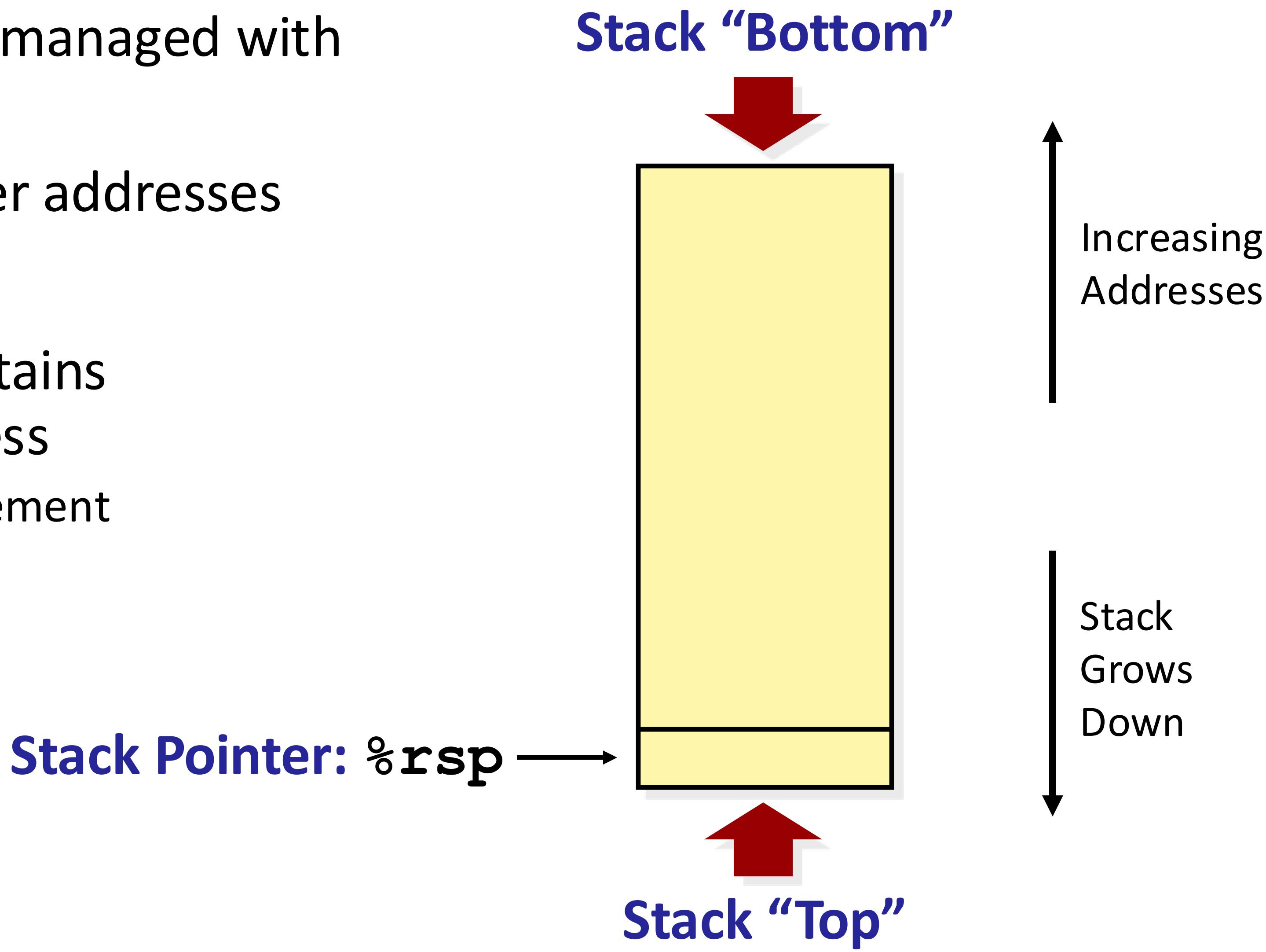
❿ **In this way, we satisfy the original design goals:**

1. Enables passing parameters to functions and returning return value
2. Automated support for storing the return address and jumping back to it
3. Memory-efficient storage that reuses memory and only allocates space for parameters, return addresses, and local variables if a function is called
4. Supports recursion – each instance/call of a function has its own stack frames to store its own return address, parameters, and local variables



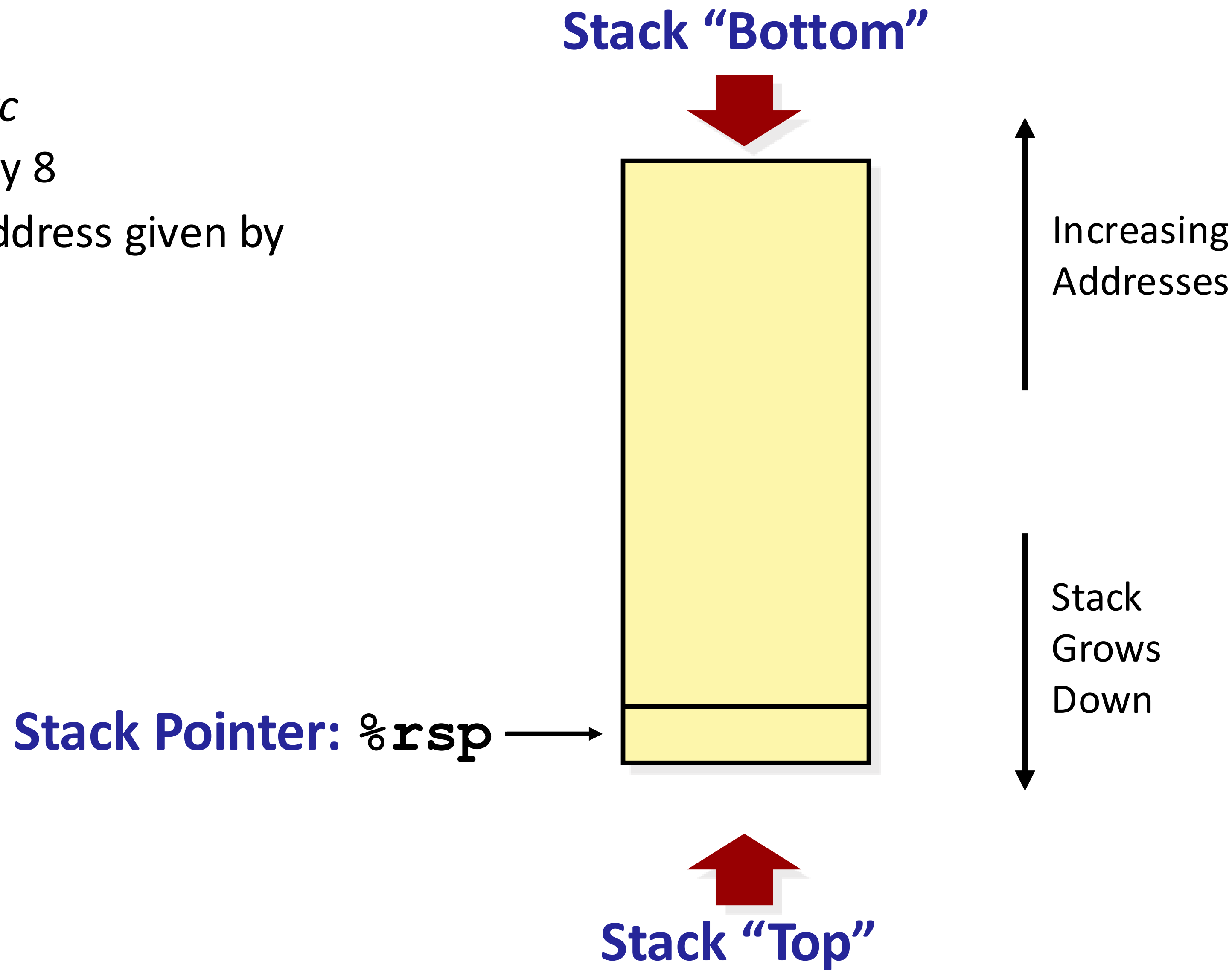
x86-64 Stack

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



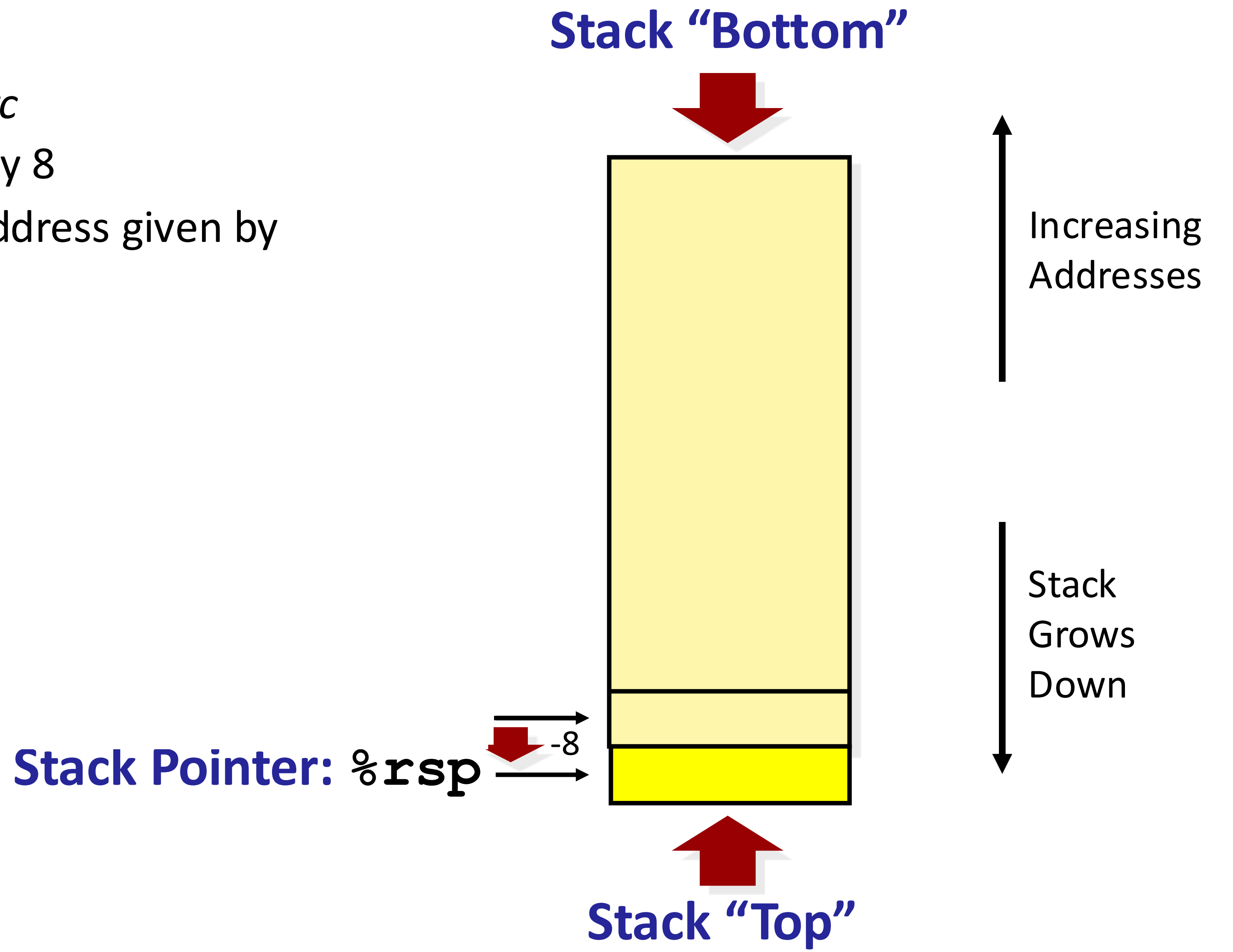
x86-64 Stack: Push

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



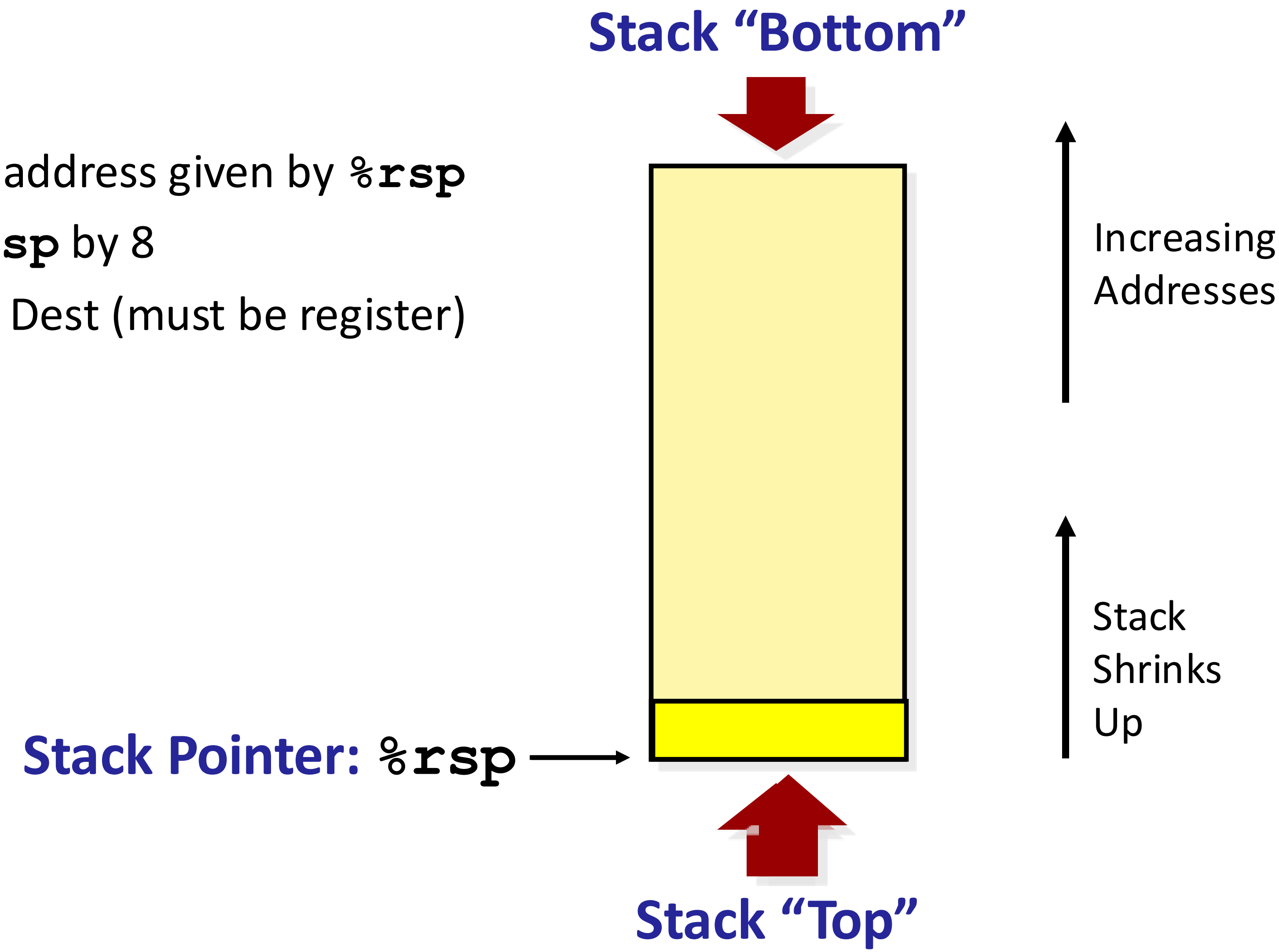
x86-64 Stack: Push (afterwards)

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



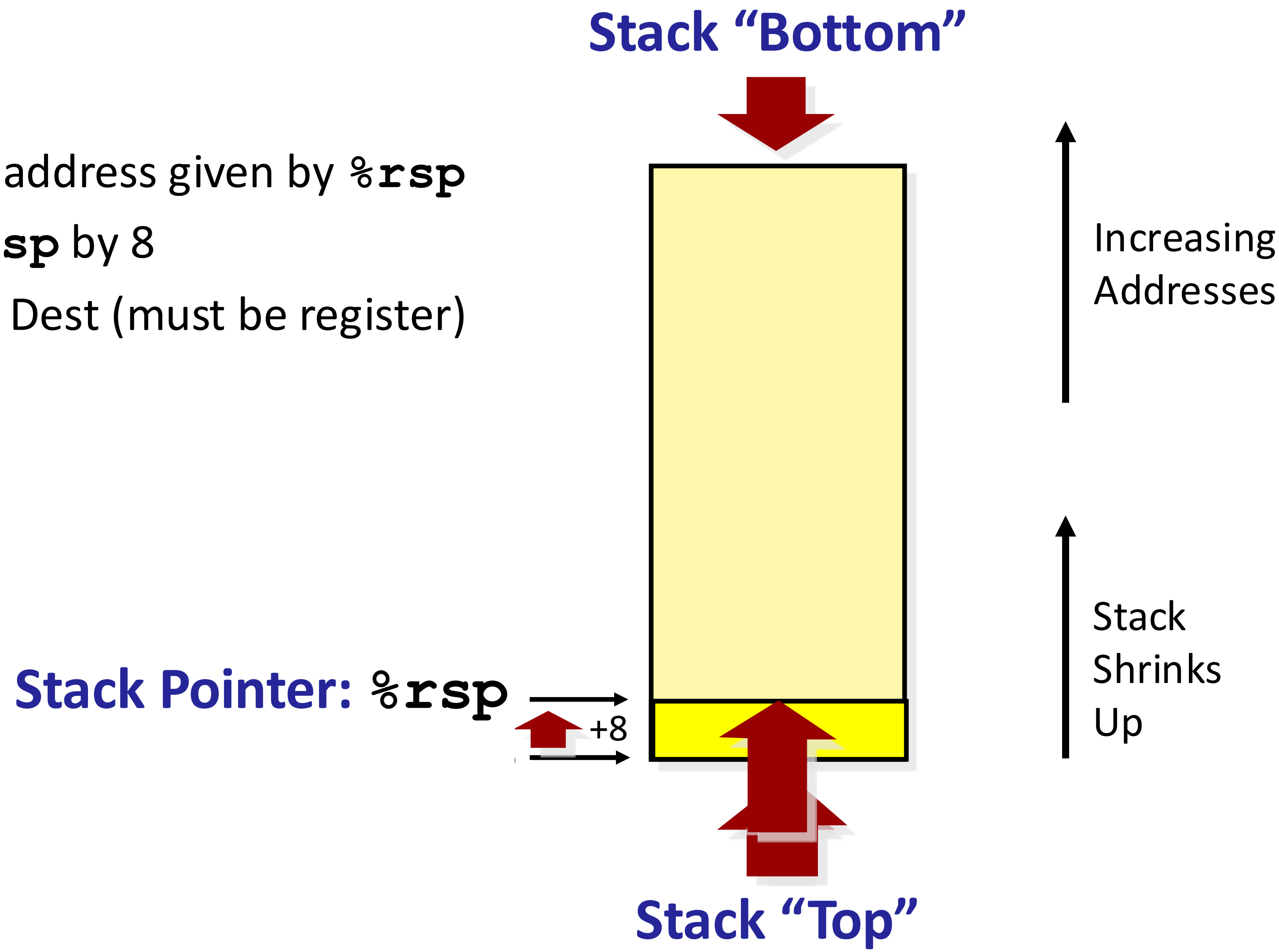
x86-64 Stack: Pop

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



x86-64 Stack: Pop (afterwards)

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



Passing parameters via CPU registers

By default, up to 6 arguments/parameters can be passed into a function.

- **`%rdi`**, …, **`%r9`**
  - If you have more than 6 parameters, then either put these into a data structure and pass one pointer to the data structure as an argument, or the compiler will just push overflow arguments onto the stack

Argument #1	`%rdi`
Argument #2	`%rsi`
Argument #3	`%rdx`
Argument #4	`%rcx`
Argument #5	`%r8`
Argument #6	`%r9`
Return value	`%rax`

By default, the return value is stored in %rax

- **`%rax`**
  - Return value

Procedure Control Flow

Passing Control

Code Examples

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

```
0000000000400540 <multstore>:
  400540: push   %rbx            # Save %rbx
  400541: mov    %rdx,%rbx       # Save dest       dest = rbx = rdx
  400544: callq  400550 <mult2>  #                 t = mult2(x,y)
  400549: mov    %rax,(%rbx)     # Save at dest    *dest = t
  40054c: pop    %rbx            # Restore %rbx
  40054d: retq                   # Return          return
```

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

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

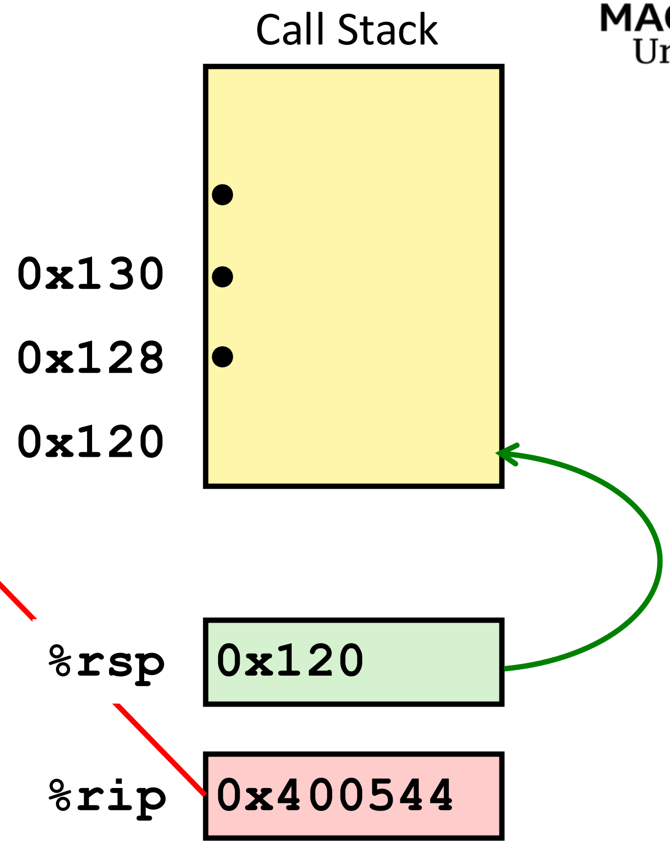
Procedure Control Flow

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

Control Flow Example

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

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



Note: in this example, the stack memory addresses are artificially low in value to make it easy to follow. Real stack addresses will be high memory values

Control Flow Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq 0x400550 ult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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

Call Stack

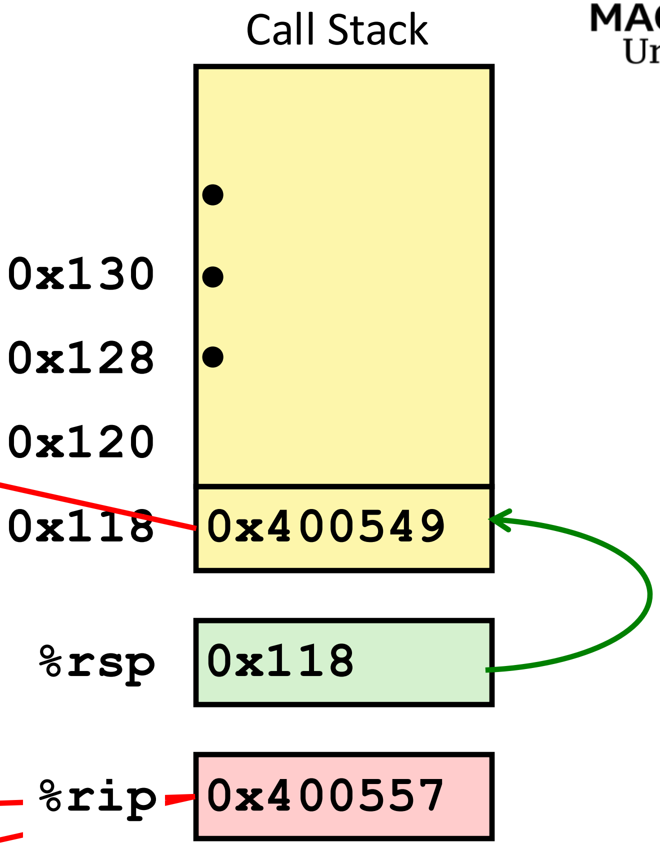
	•
0x130	•
0x128	•
0x120	
0x118	0x400549

%rsp	0x118
%rip	0x400549

Control Flow Example

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

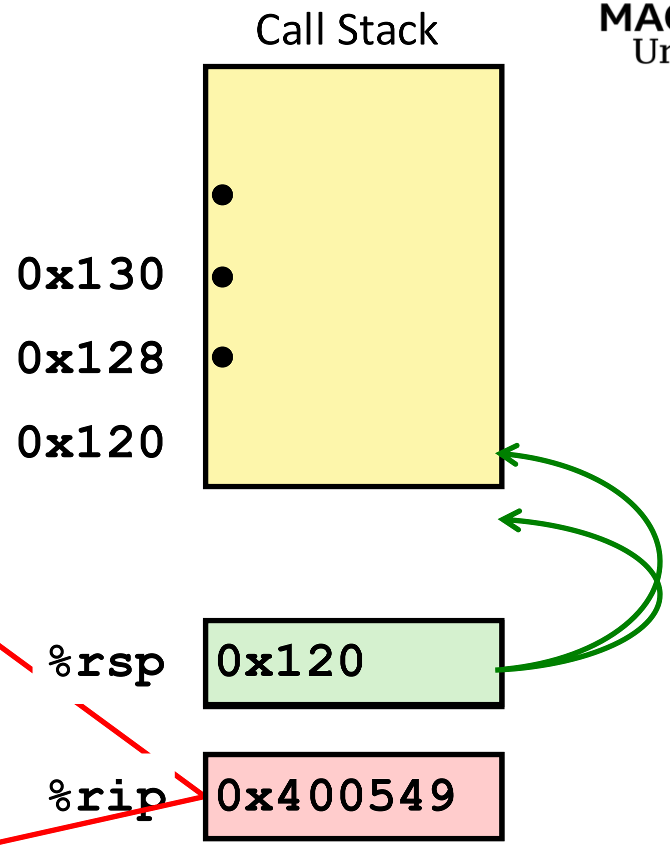
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example

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

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



Procedure Call Chain and Recursion

Stack-Based Languages

- Stack allocated in ***Frames***
  - Frame = state for single procedure instantiation
- Stack discipline
  - Callee returns before caller does
  - Procedure state is needed: from call to return
- 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

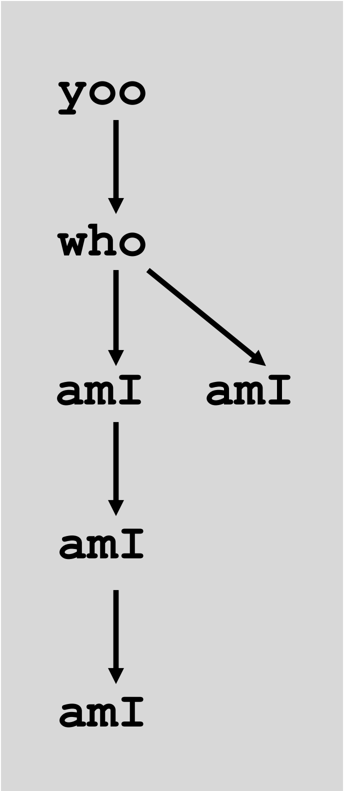
Call Chain Example

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

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

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

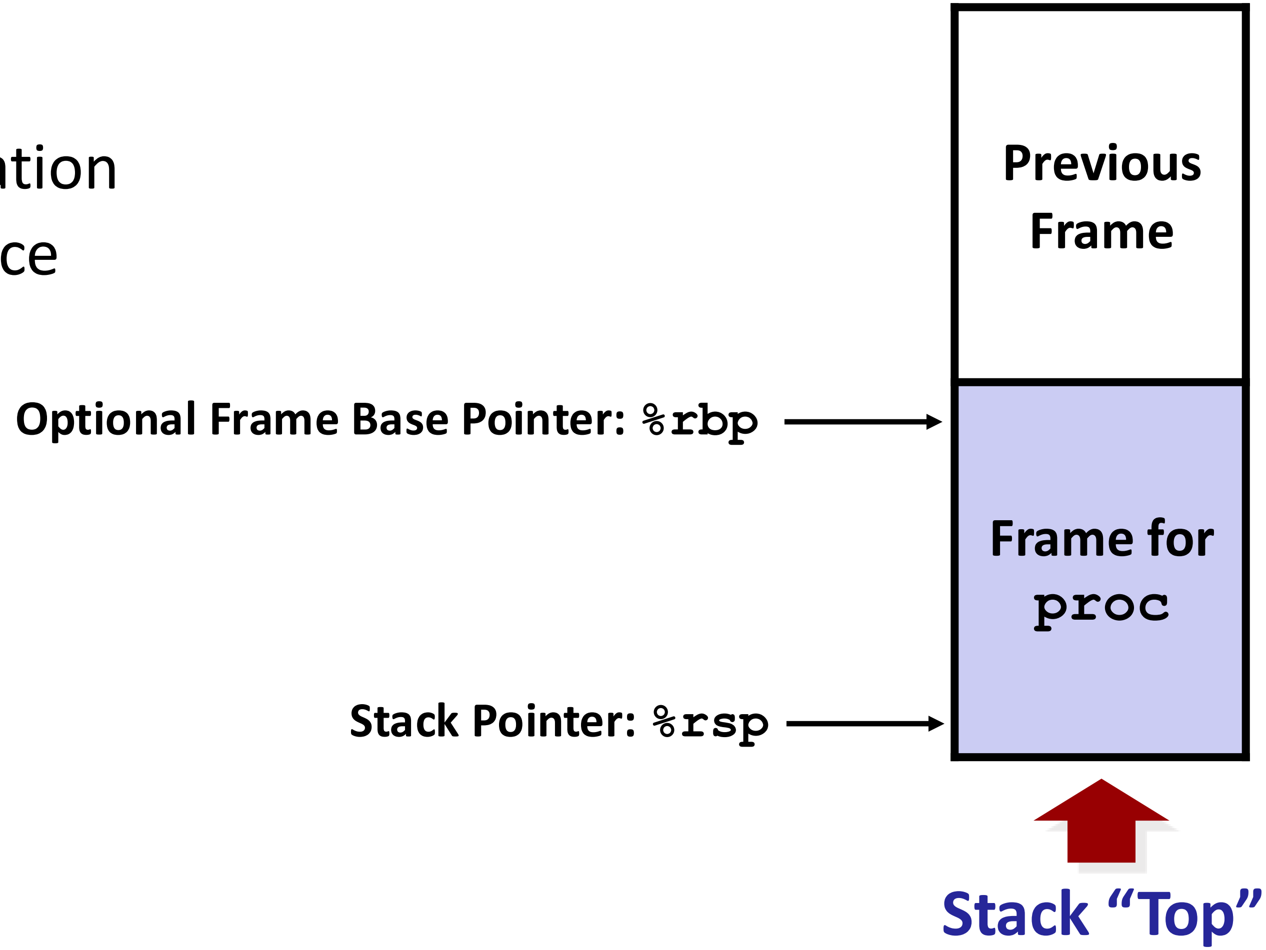
Example Call Chain



Procedure `amI()` is recursive

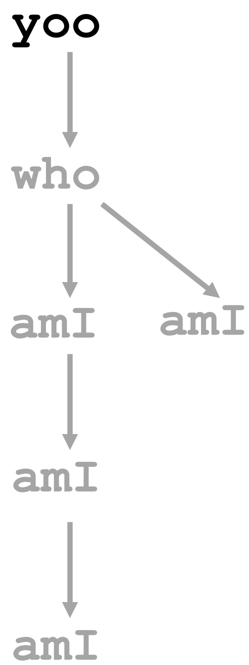
Stack Frames

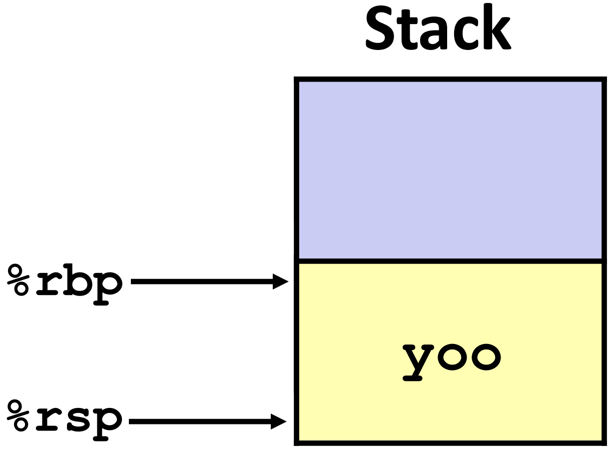
- Contents
  - Local variables
  - Return information
  - Temporary space



Example

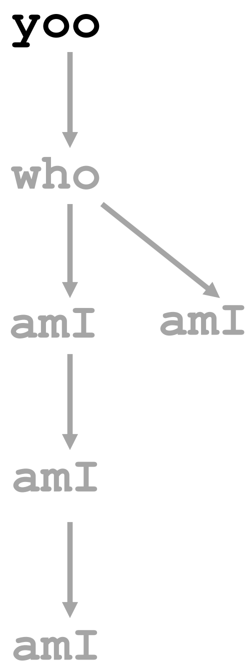
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

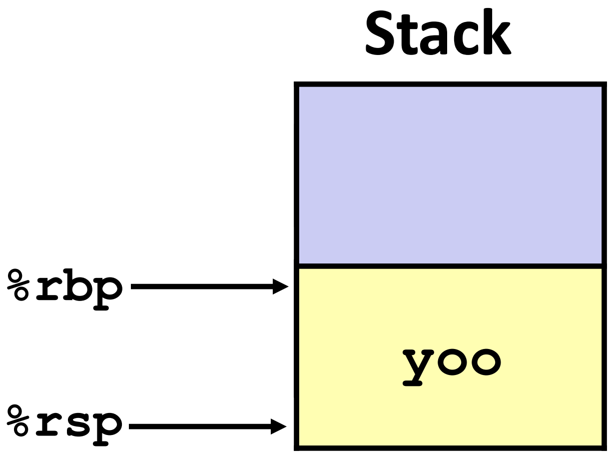




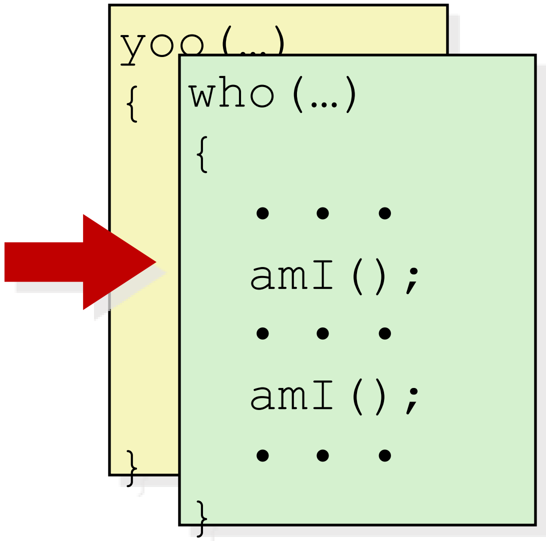
Example

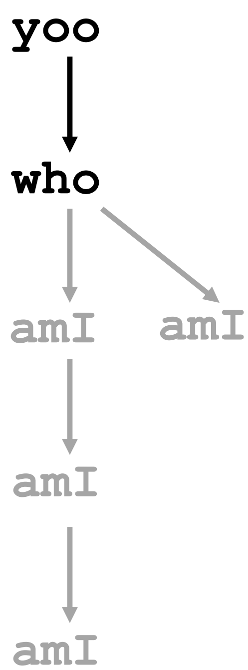
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

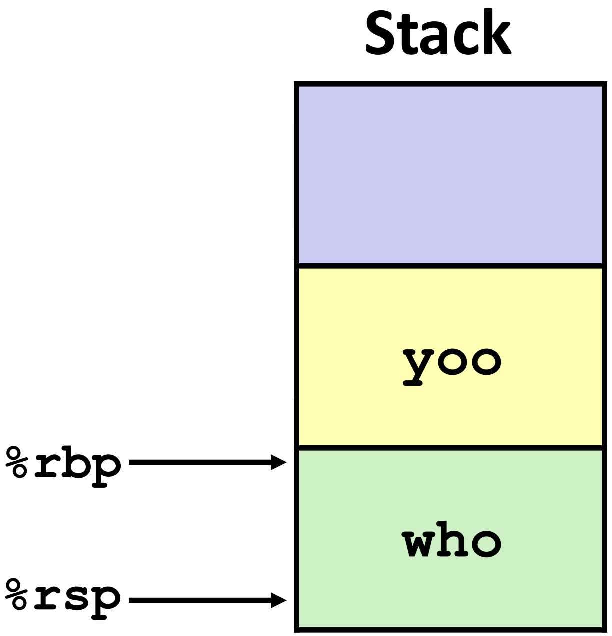




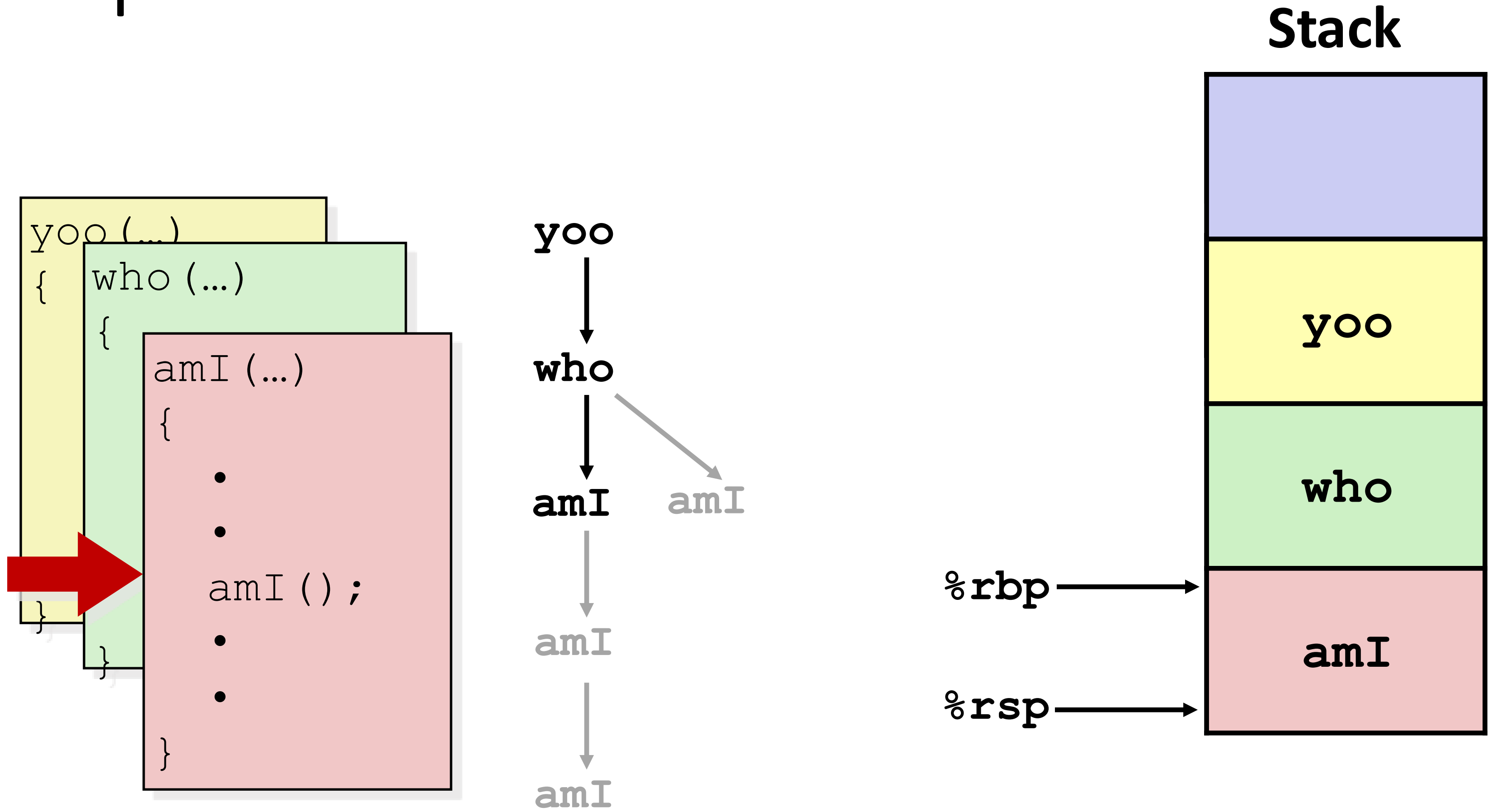
Example





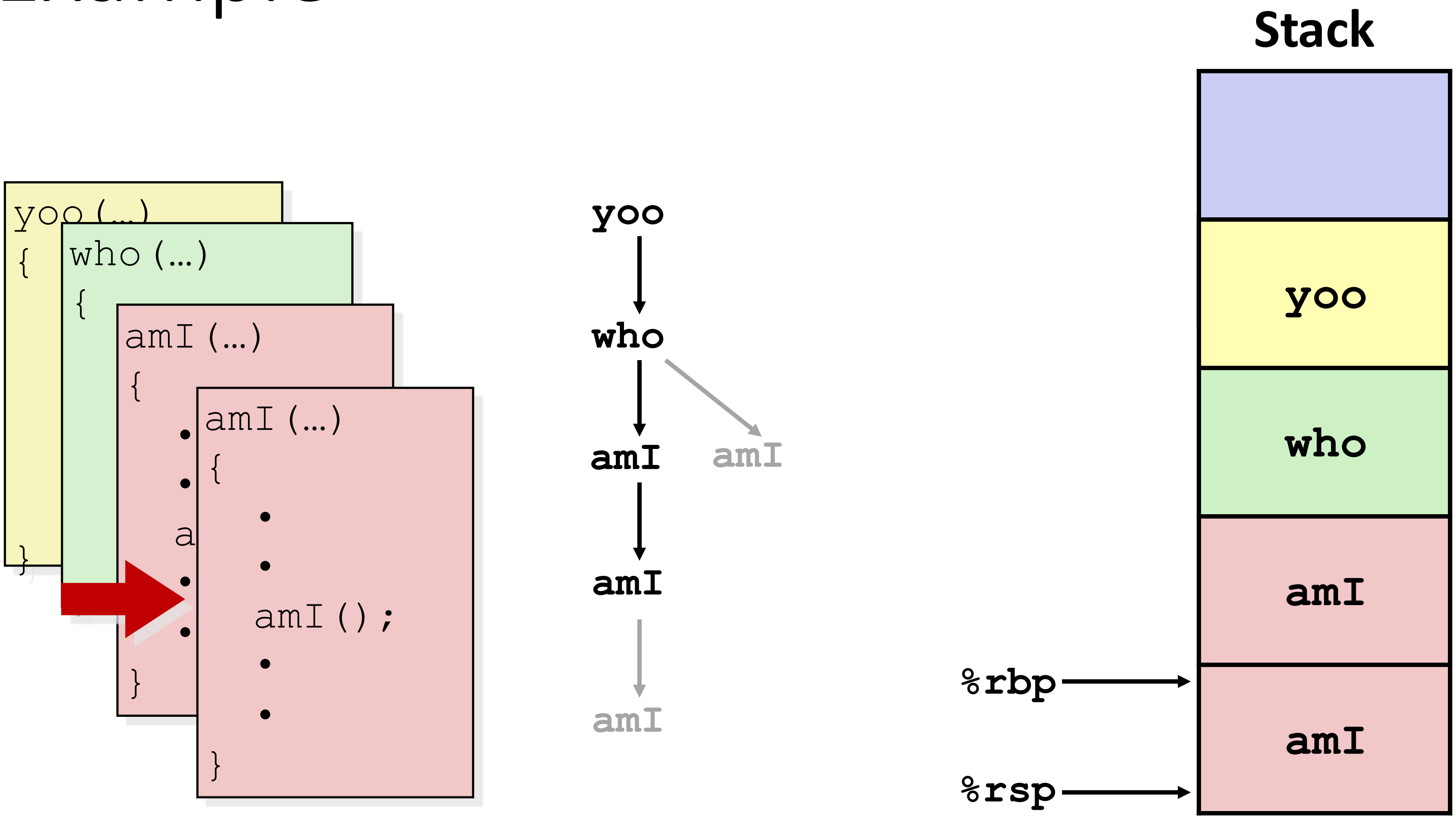


Example

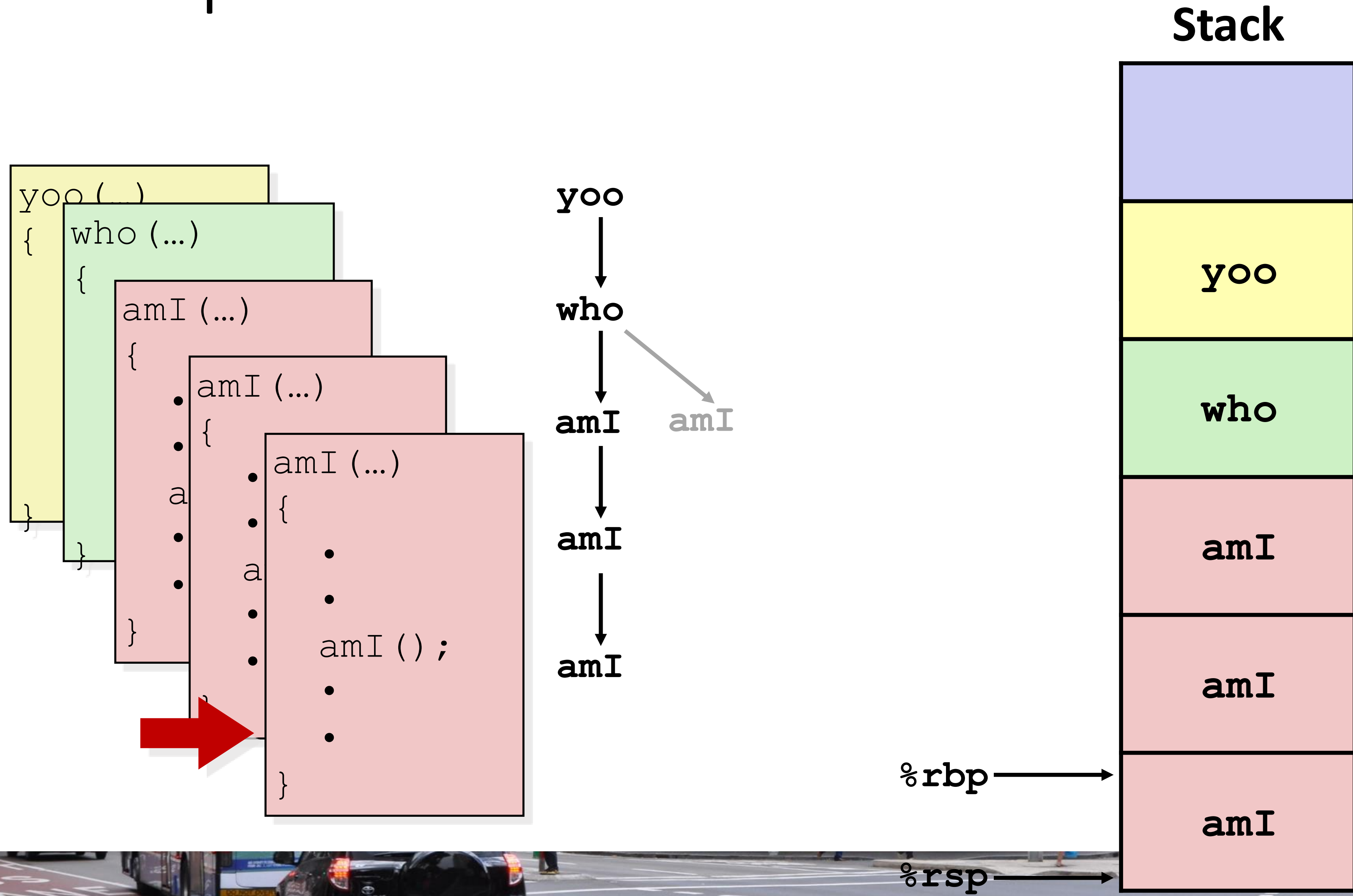


yoo (…)
{
}
who (…)
{
}
amI (…)
{
amI ();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp

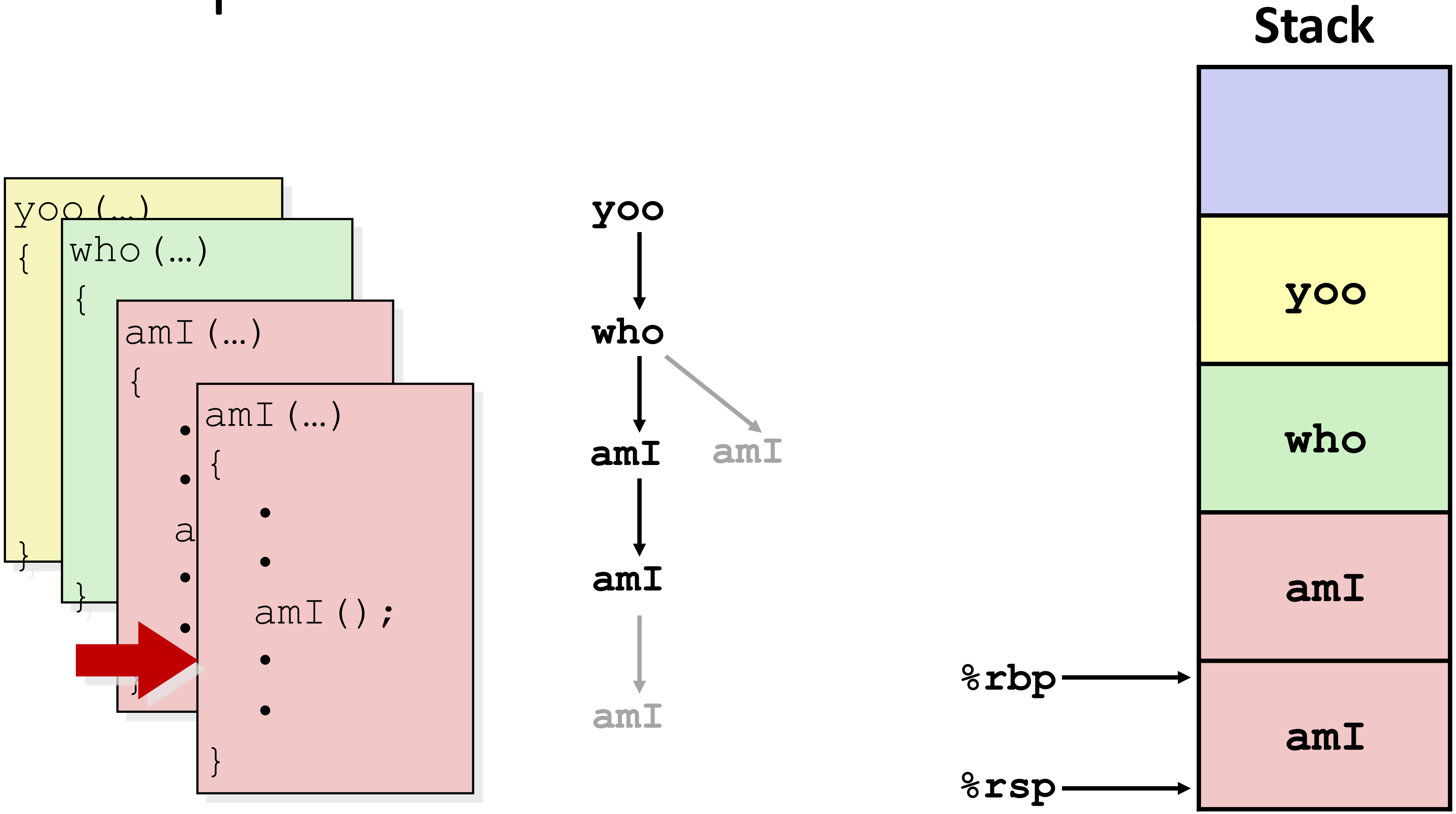
Example


yoo (…)
{
}
who (…)
{
amI (…)
{
a
}
amI (…)
{
amI ();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%rbp
%rsp

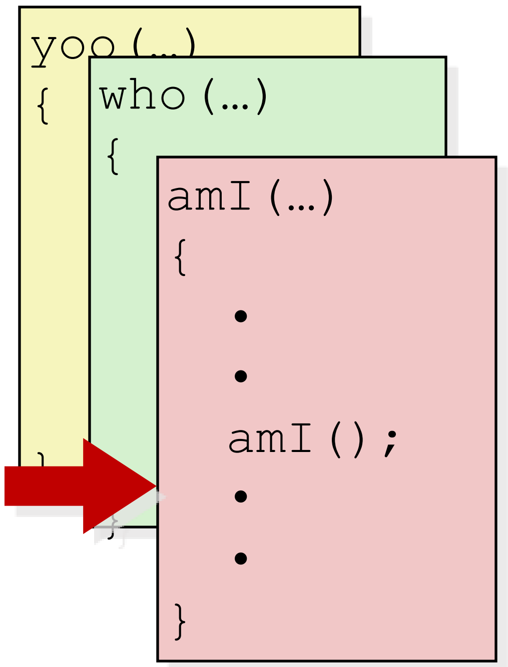
Example

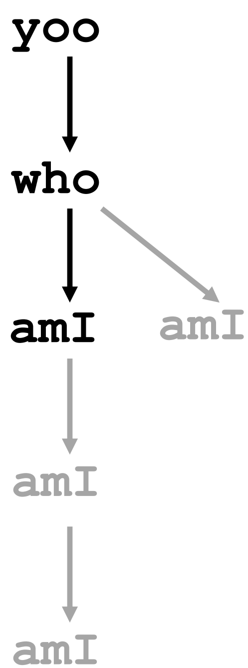

yoo (…)
{
}
who (…)
{
}
amI (…)
{
}
amI (…)
{
}
amI (…)
{
amI();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
amI
%rbp
%rsp

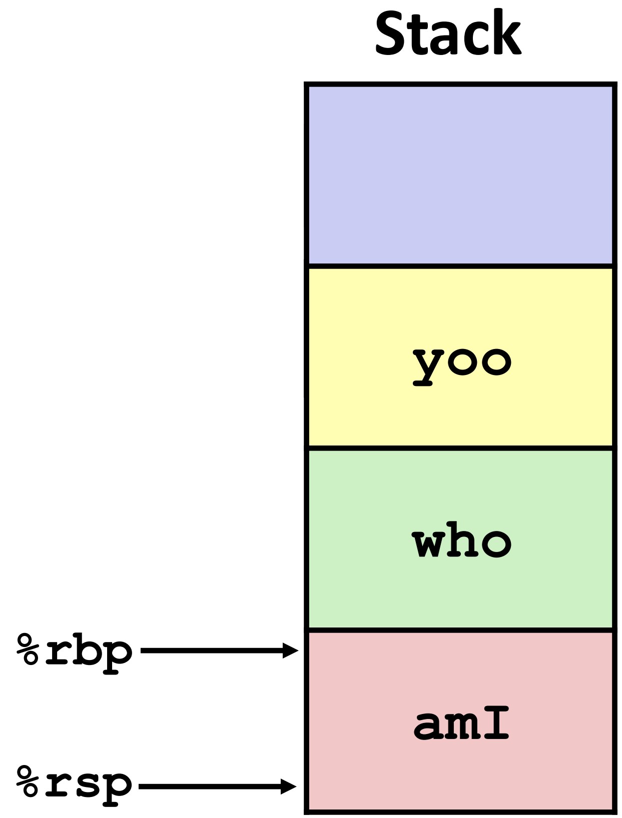
Example


yoo (…)
{
}
who (…)
{
}
amI (…)
{
a
}
amI (…)
{
amI ();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%rbp
%rsp

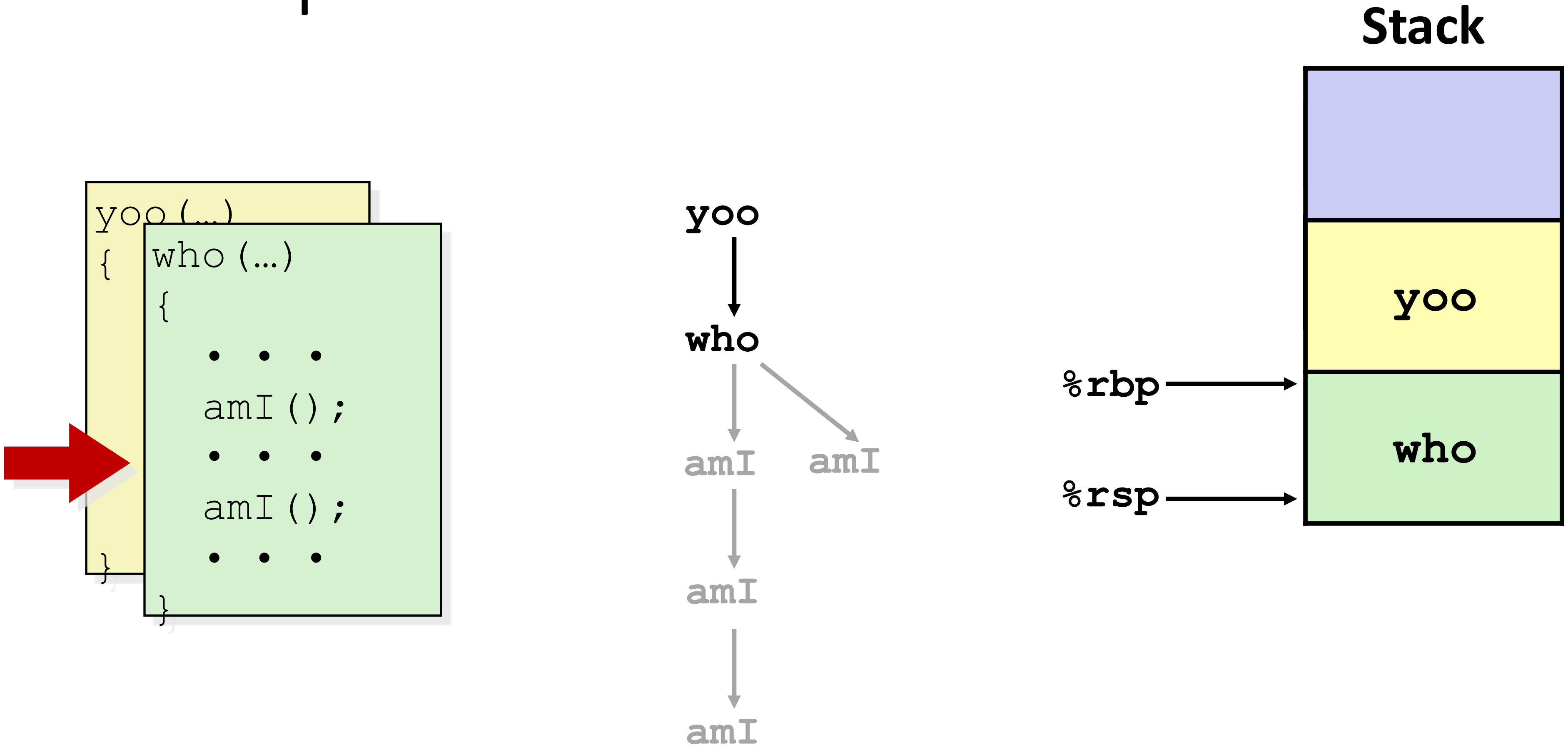
Example


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

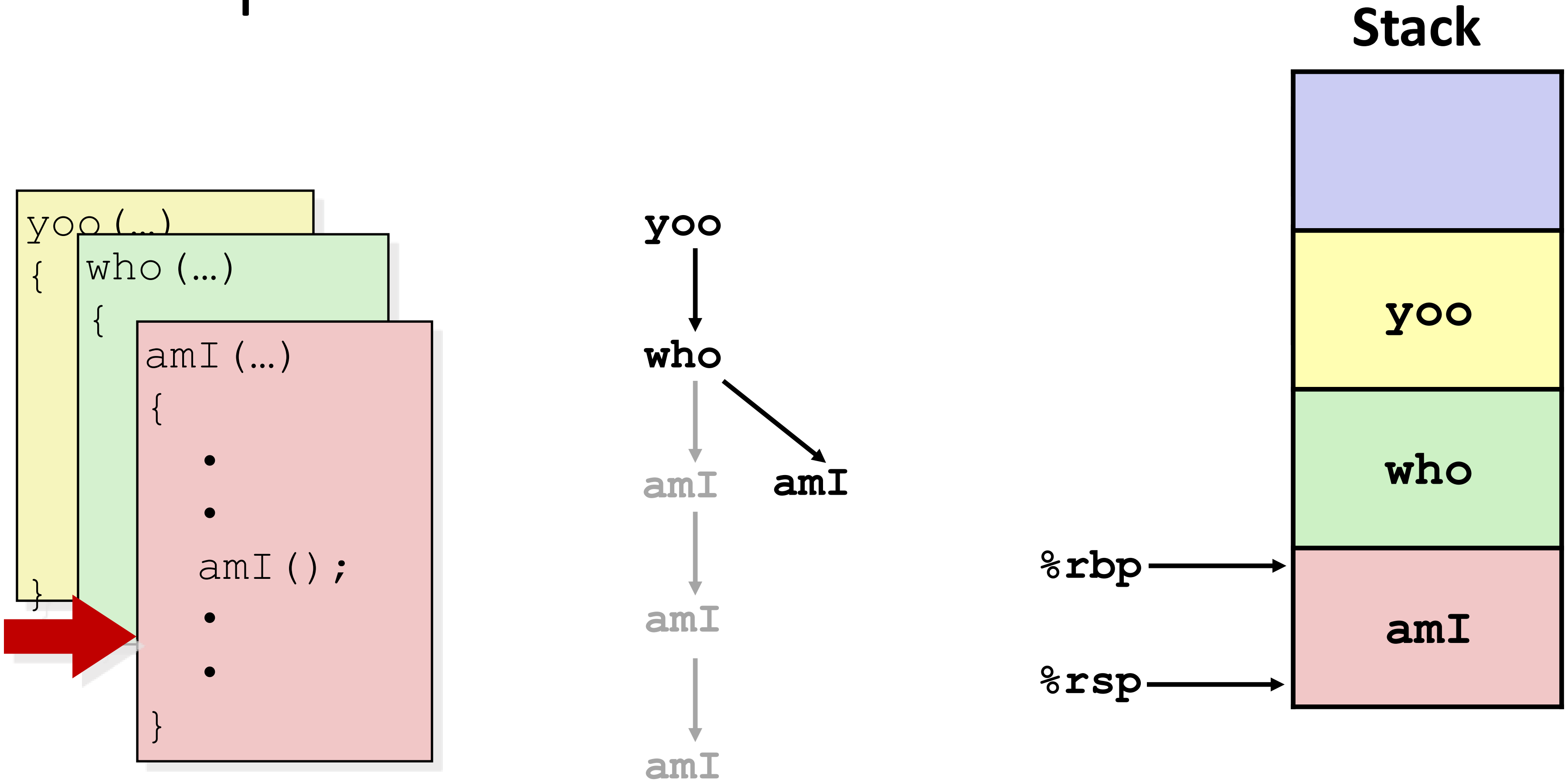

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

Example

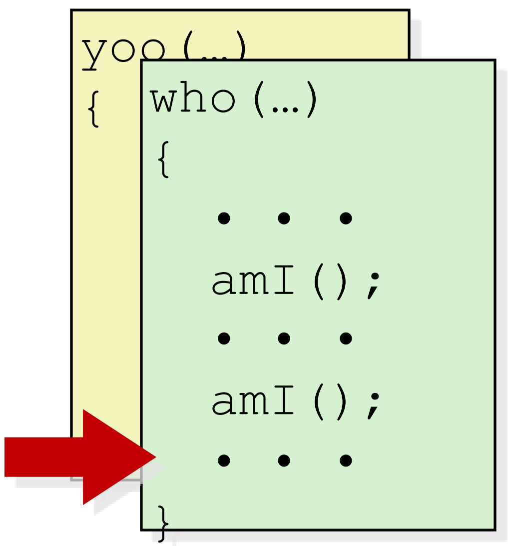


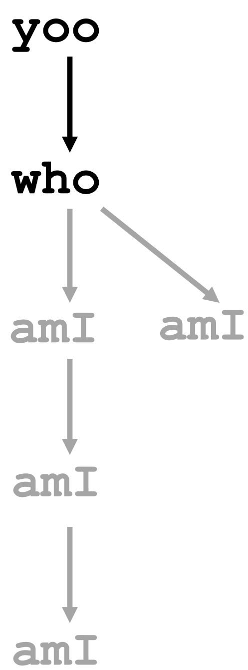
Example

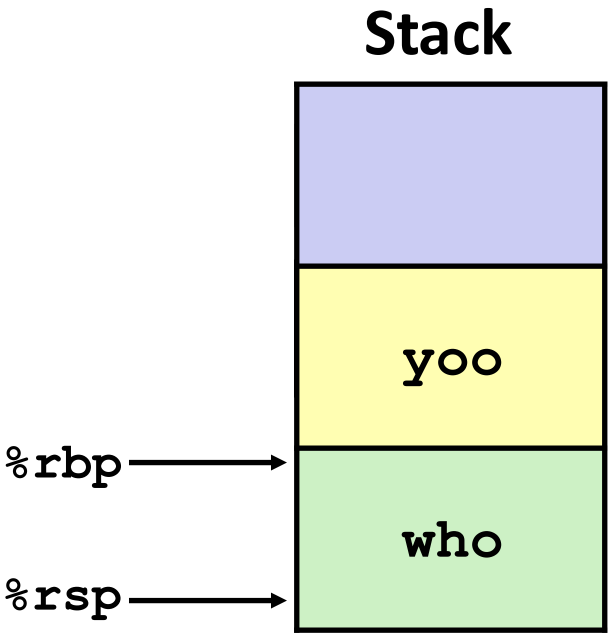


yoo (…)
{
}
who (…)
{
amI (…)
{
•
•
amI ();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%rbp
%rsp

Example

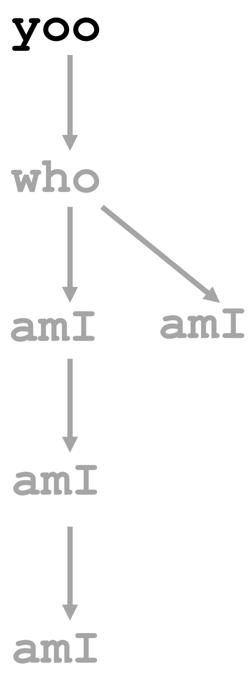


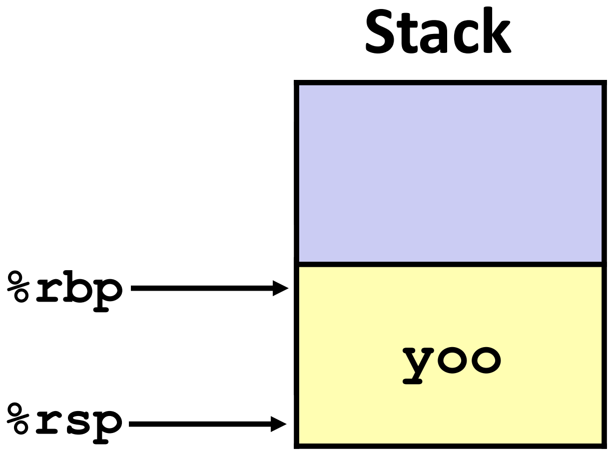




Example

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

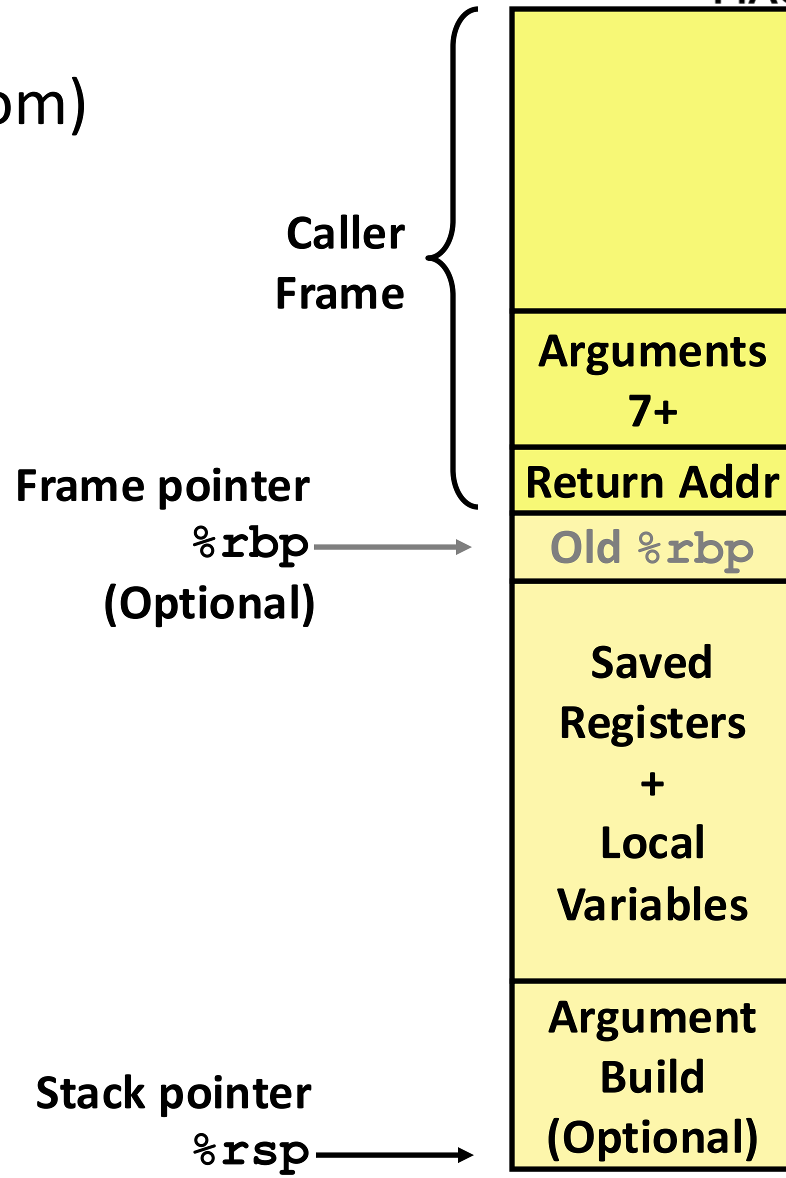




x64/Linux Stack Frame

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

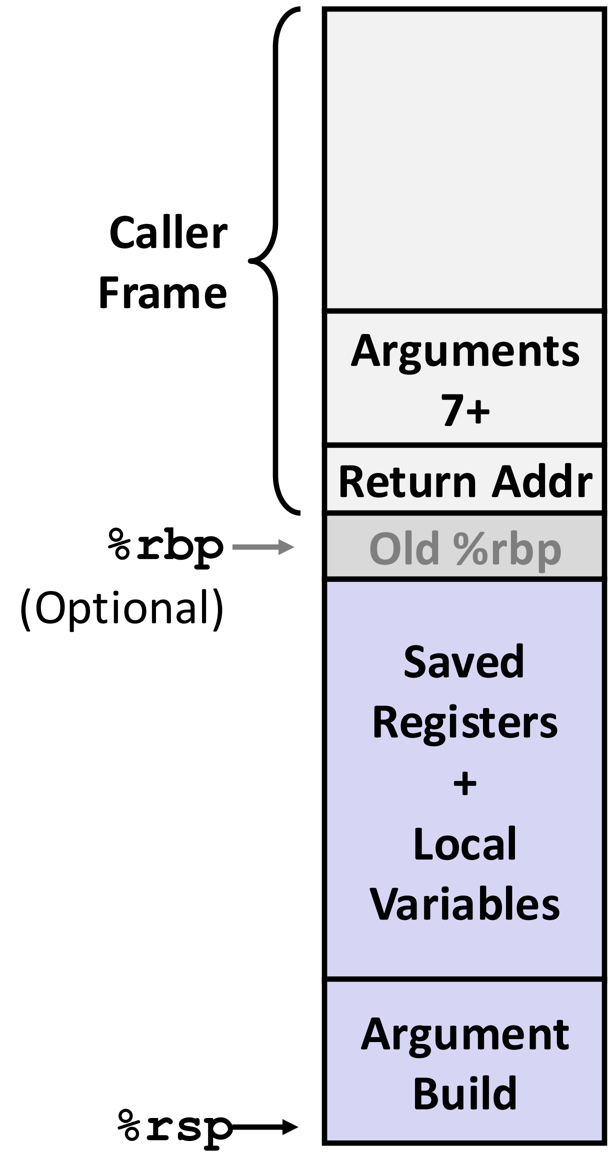
- Used from Caller Stack Frame
  - Return address
    - Pushed by **`call`** instruction
  - Arguments for this call



Backup

x64 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 calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments in registers and at top of stack
  - Result return in `%rax`
- Pointers are addresses of values
  - On stack or global



```
long do_op
  (long op, long x, long y)
{
  long r;
  switch (op) {
    case 6: // r = x + 2 * y
      y = y * 2;
      // Note fall through
    case 1:
      r = x + y;
      break;
    case 2:
      r = x - y;
      break;
    case 3:
    case 4:
      r = x * y;
      break;
    default:
      r = 0;
  }
  return r;
}
```

Switch Statement Example

- Multiple case labels
  - Here: 3 and 4
- Fall through cases
  - Here: 6
- Missing cases
  - Here: 5

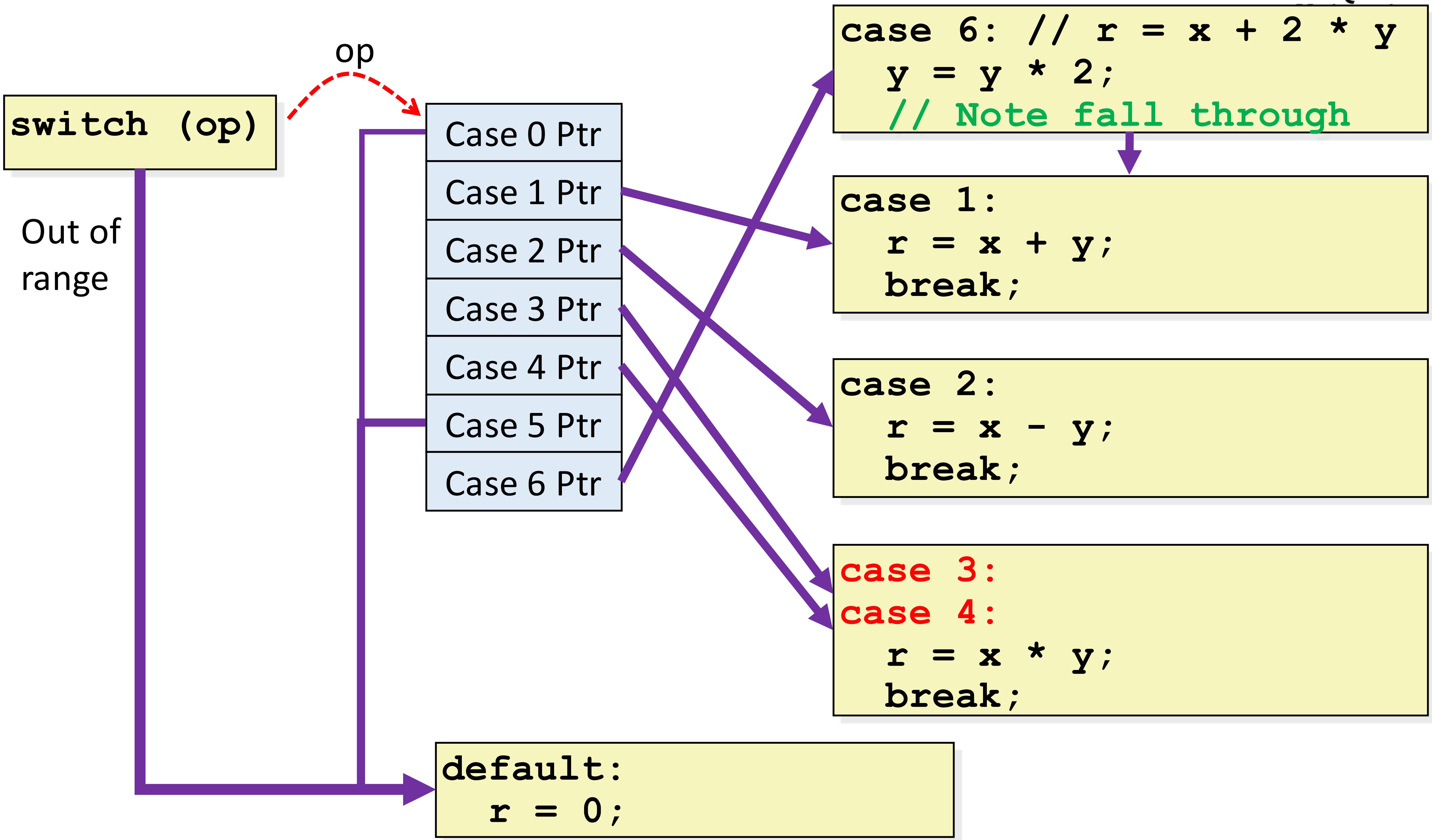
Switch statement – entire code

MACQUARIE University

```
do_op:
        cmpq    $6, %rdi
        ja      .L8
        leaq    .L4(%rip), %r8
        movslq  (%r8,%rdi,4),%rcx
        addq    %r8, %rcx
        jmp     *%rcx
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L6-.L4
        .long   .L6-.L4
        .long   .L8-.L4
        .long   .L7-.L4
```

```
        .text
.L7:
        addq    %rdx, %rdx
.L3:
        leaq    (%rdx,%rsi), %rax
        ret
.L5:
        movq    %rsi, %rax
        subq    %rdx, %rax
        ret
.L6:
        movq    %rdx, %rax
        imulq   %rsi, %rax
        ret
.L8:
        movl    $0, %eax
        ret
```

Switch statement – the basic idea



Switch statement – the cases

Registers: op → rdi   x → rsi   y → rdx

```
Case 6    .L7:
              addq    %rdx, %rdx            # rdx += rdx;  rdx *= 2
Case 1    .L3:                              # Note fall through
              leaq    (%rdx,%rsi), %rax     # rax = rdx + rsi
              ret                           # return rax
          .L5:
Case 2        movq    %rsi, %rax            # rax = rsi
              subq    %rdx, %rax            # rax -= rdx
              ret                           # return rax
          .L6:
Case 3        movq    %rdx, %rax            # rax = rdx
Case 4        imulq   %rsi, %rax            # rax *= rsi
              ret                           # return rax
          .L8:
              movl    $0, %eax              # rax = 0
Default       ret                           # return rax
```

Switch statement – the jump table

```
    leaq    .L4(%rip), %r8          # r8 = .L4 (address)
    movslq  (%r8,%rdi,4), %rcx      # rcx = Mem[r8 + op*4]
    addq    %r8, %rcx               # rcx += r8; rcx += .L4
    jmp     *%rcx                   # goto *rcx; // Not C
```

```
    .section   .rodata              # Read only data
    .align 4
.L4:                                # The jump table
    .long   .L8-.L4                 # Case 0: Default  .L8
    .long   .L3-.L4                 # Case 1:          .L3
    .long   .L5-.L4                 # Case 2:          .L5
    .long   .L6-.L4                 # Case 3:          .L6
    .long   .L6-.L4                 # Case 4:          .L6
    .long   .L8-.L4                 # Case 5: Default  .L8
    .long   .L7-.L4                 # Case 6:          .L7
```

Switch statement – the range check

```
do_op:
      cmpq    $6, %rdi       # flags = rdi ?? 6
      ja      .L8            # if (flags.a) goto .L8
                             # if ((unsigned) rdi > 6) goto .L8
```

Unsigned comparison of rdi with 6 (jump above)

- If signed op > 6, table lookup would overflow.
- If signed op < 0, table lookup would underflow.
- Signed op < 0 ---> unsigned op is huge!
- Equivalent to:

  if (rdi < 0 || rdi > 6) goto .L8

Switch statement – entire code

```
do_op:
        cmpq    $6, %rdi
        ja      .L8
        leaq    .L4(%rip), %r8
        movslq  (%r8,%rdi,4),%rcx
        addq    %r8, %rcx
        jmp     *%rcx
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L6-.L4
        .long   .L6-.L4
        .long   .L8-.L4
        .long   .L7-.L4
```

```
        .text
.L7:
        addq    %rdx, %rdx
.L3:
        leaq    (%rdx,%rsi), %rax
        ret
.L5:
        movq    %rsi, %rax
        subq    %rdx, %rax
        ret
.L6:
        movq    %rdx, %rax
        imulq   %rsi, %rax
        ret
.L8:
        movl    $0, %eax
        ret
```