[CSED211] Introduction to Computer Software Systems

Lecture 6: Procedures

Prof. Jisung Park

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

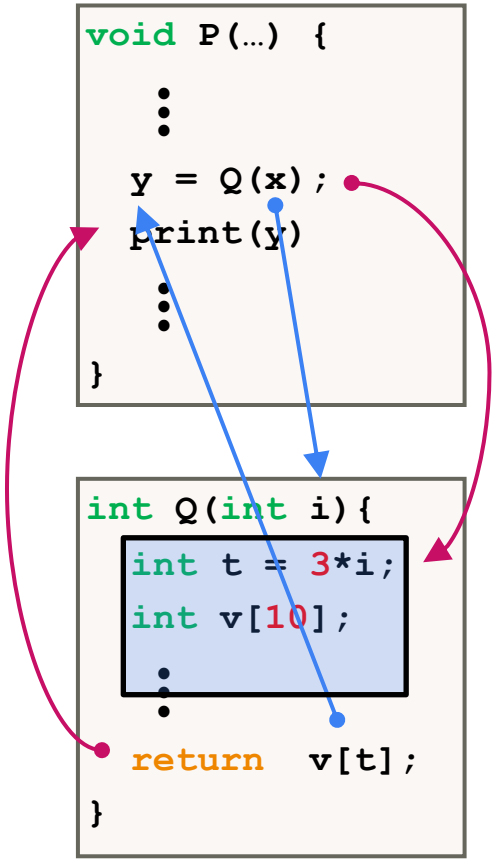
2023.10.04

Lecture Objectives

- Basic functionality of two key pairs: <push, pop> and <call, ret>
- How to identify the different components of a stack
  - Return address, arguments, saved registers, and local variables
- The difference between callee- and caller-saved registers
- How a stack permits functions to be called recursively (re-entrant)

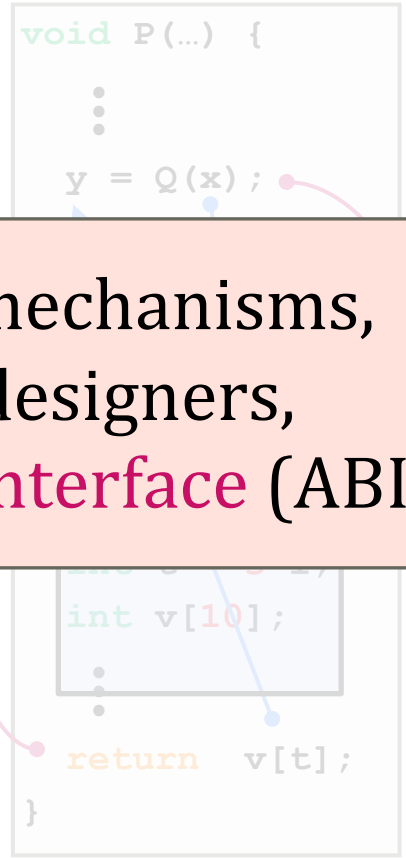
Mechanisms in Procedures

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



Mechanisms in Procedures

- Passing control
  - To the beginning of target procedure code
  - Back to the return point
- Passing data
- Allocating and deallocating memory
- All implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



Machine instructions implement the mechanisms, but the choices are determined by designers, which make up the Application Binary Interface (ABI)

Lecture Agenda: Procedures

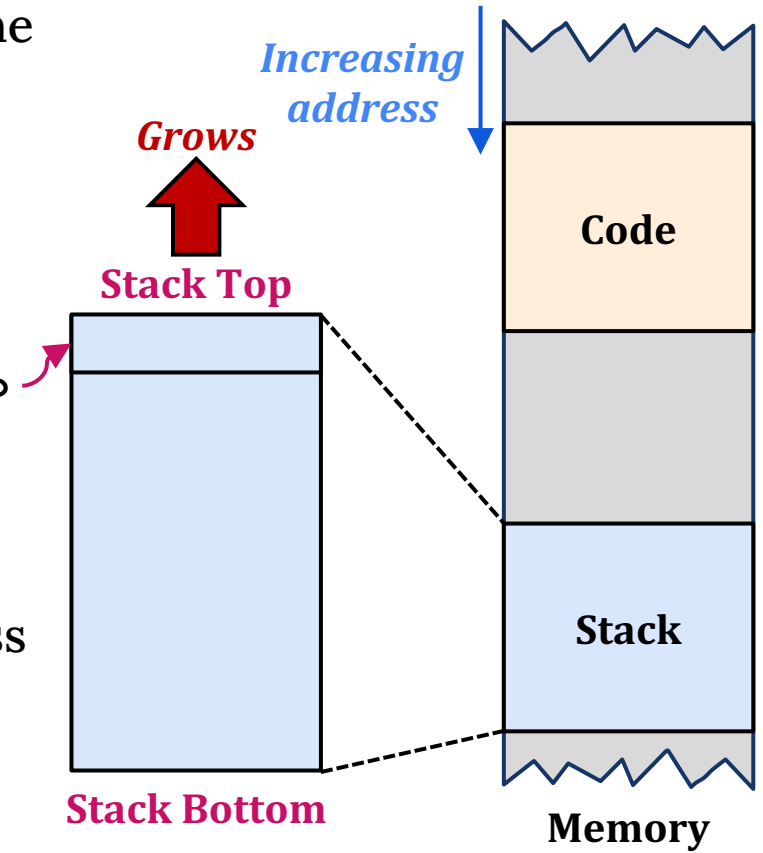
- Stack Structure

- Calling Conventions
  - Passing Control
  - Passing Data
  - Managing Local Data

- Illustrations of Recursion & Pointers

x86-64 Stack

- Memory region managed with stack discipline
  - Last in, first out (LIFO)
  - Memory viewed as array of bytes
  - Different regions have different purposes
    - Like ABI, a policy decision

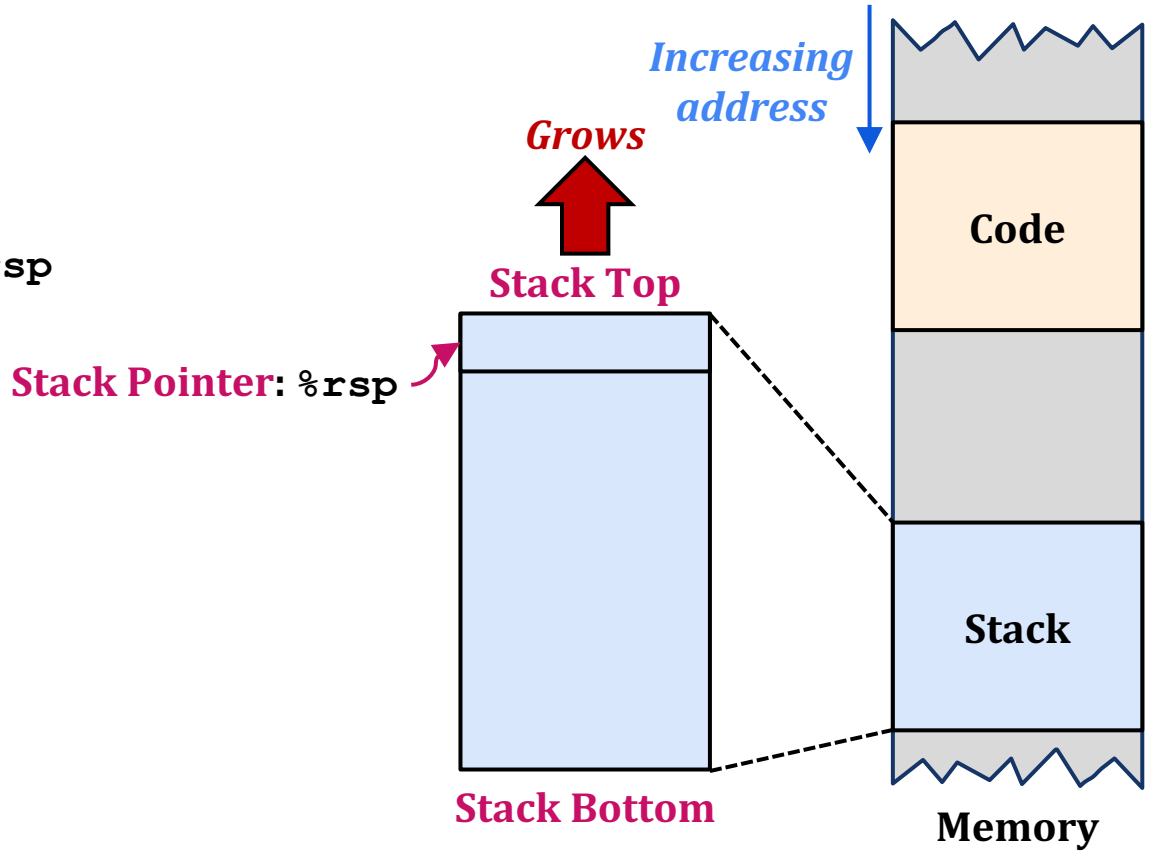


- Grows toward lower (smaller) addresses

- Register `%rsp` stores the lowest stack address
  - i.e., the top element's address

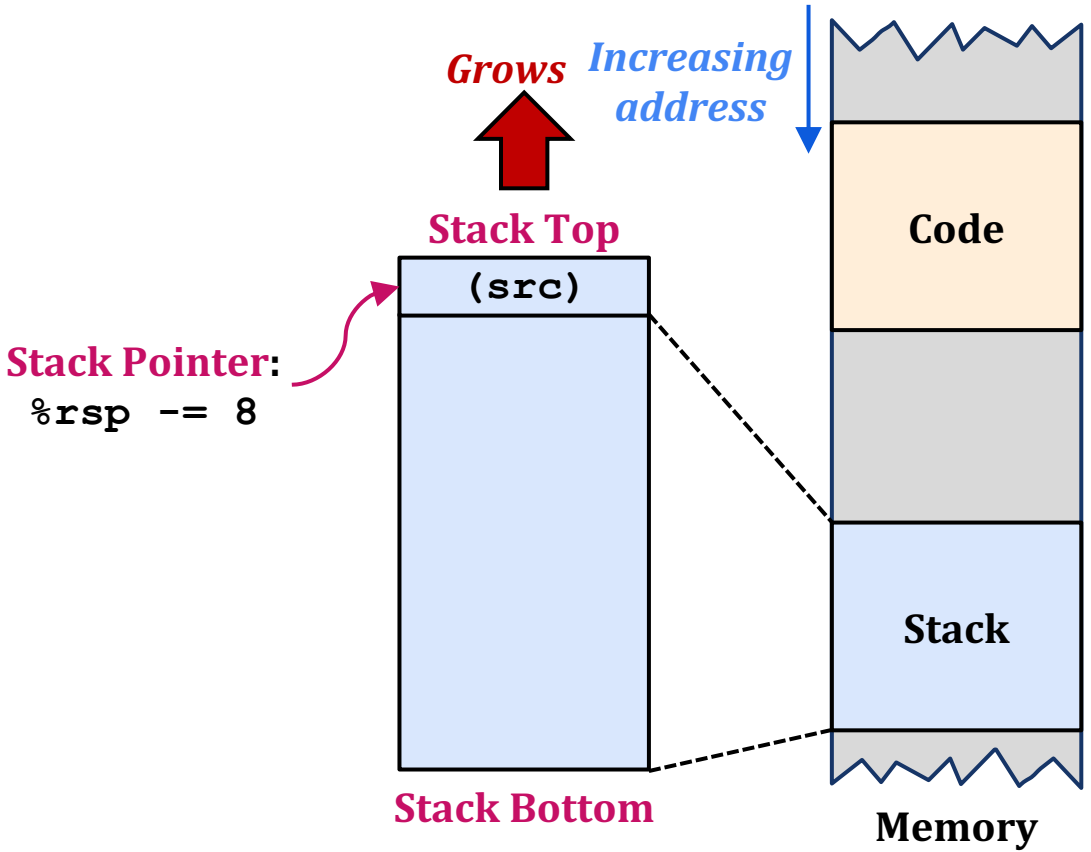
x86-64 Stack: Push & Pop

- **`pushq src`**
  - Fetch the value at **`src`**
  - Decrement **`%rsp`** by 8
  - Write the source value at the address given by **`%rsp`**



x86-64 Stack: Push & Pop

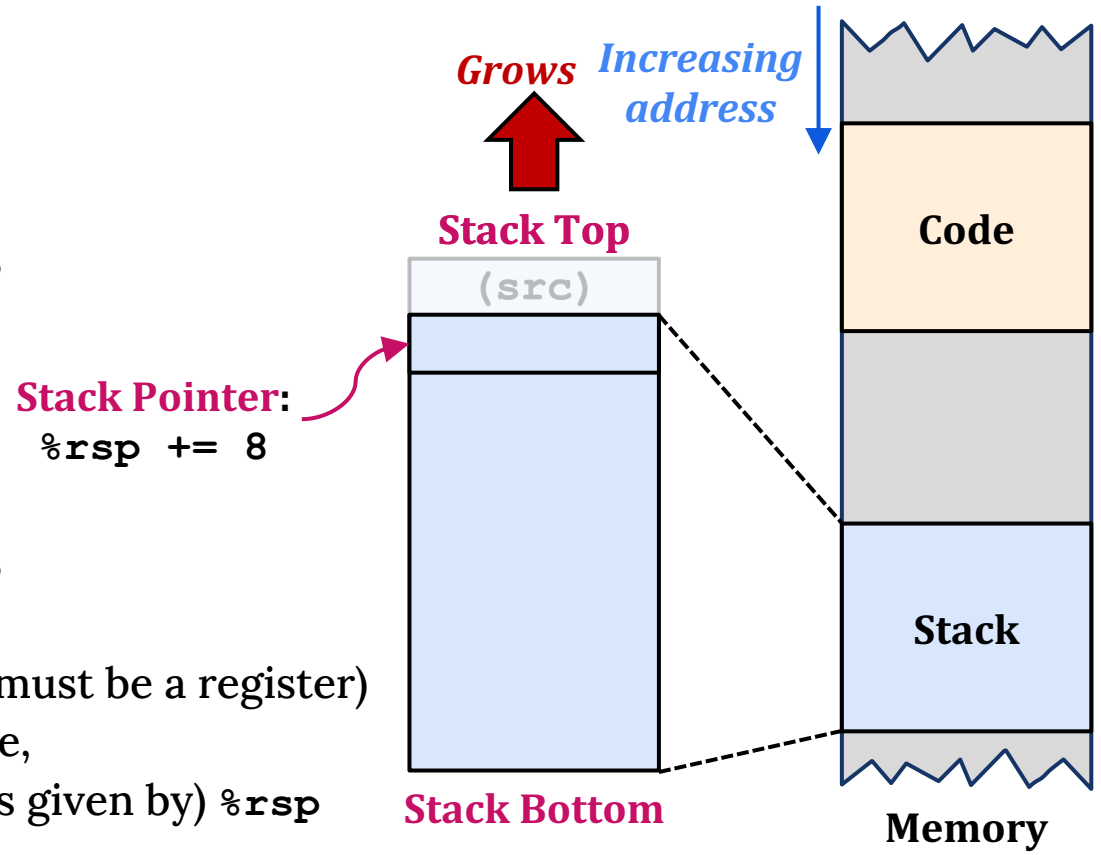
- **`pushq src`**
  - Fetch the value at **`src`**
  - Decrement **`%rsp`** by 8
  - Write the source value at the address given by **`%rsp`**



x86-64 Stack: Push & Pop

- **`pushq src`**
  - Fetch the value at **`src`**
  - Decrement **`%rsp`** by 8
  - Write the source value at the address given by **`%rsp`**

- **`popq dst`**
  - Read the value at the address given by **`%rsp`**
  - Increment **`%rsp`** by 8
  - Store the read value at **`dst`** (must be a register)
  - The memory does not change, but only the value of (address given by) **`%rsp`**



Lecture Agenda: Procedures

- Stack Structure

- Calling Conventions
  - **Passing Control**
  - Passing Data
  - Managing Local Data

- Illustrations of Recursion & Pointers

Code Examples

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

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

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```

Procedure Control Flow

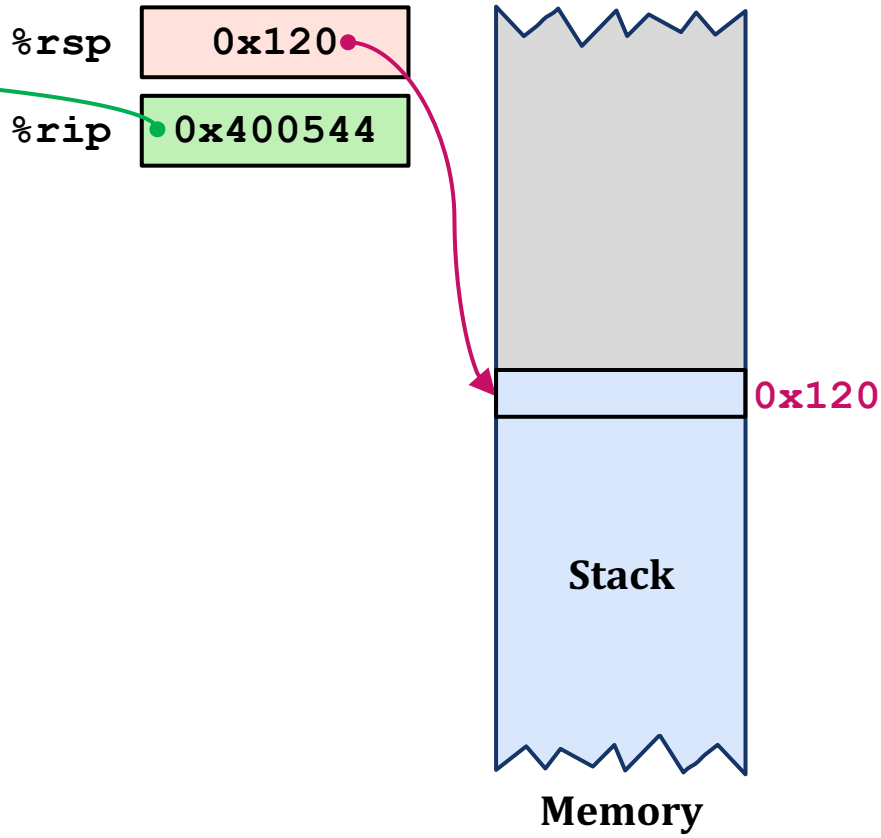
- Use the stack to support procedure call and return

- Procedure call: **`call label`**
  - Push the return address on the stack
    - Return address: the address of the next instruction right after call
  - Jump to **`label`**

- Procedure return: **`ret`**
  - Pop the address from stack
  - Jump to the address

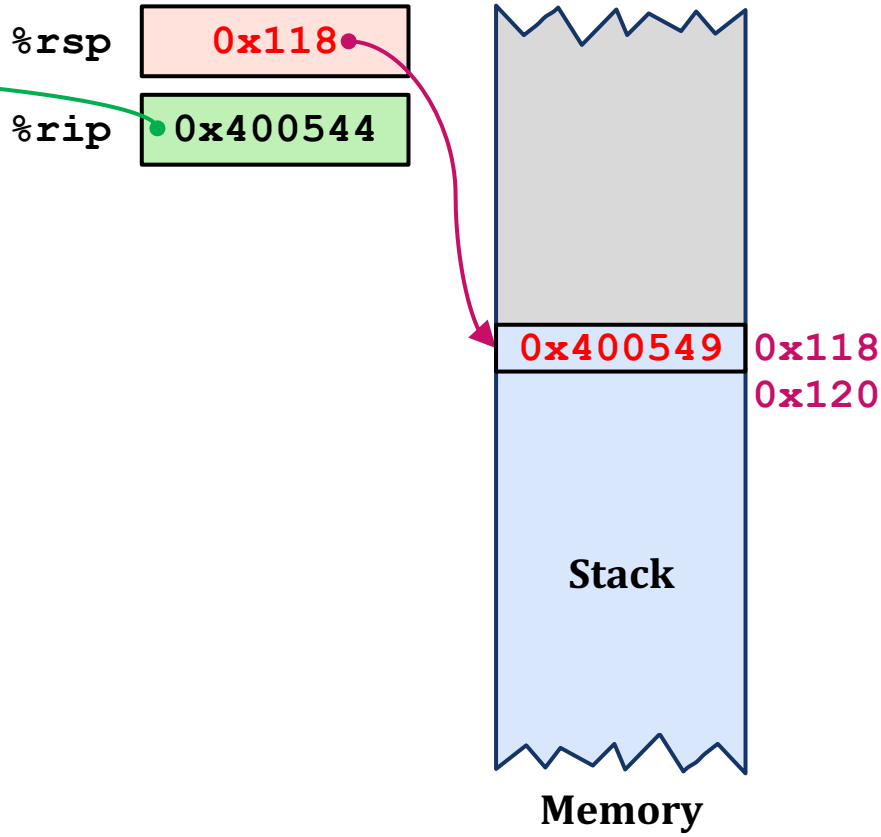
Control Flow

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



Control Flow

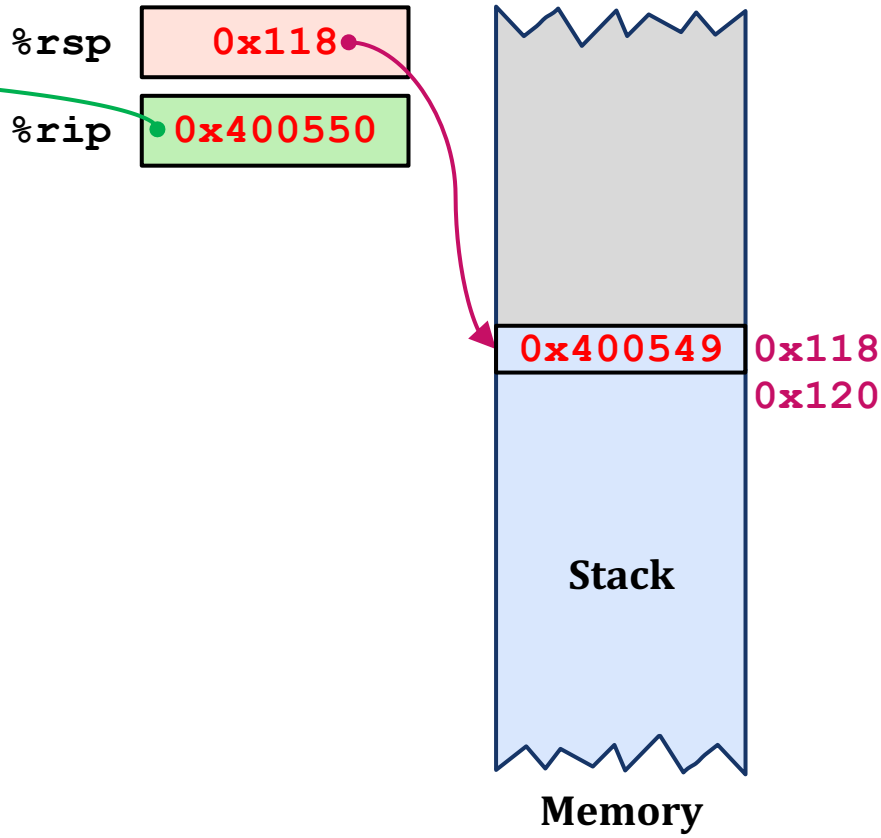
```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



- Procedure call: `call label`
  - Push the return address on the stack
  - Jump to `label`

Control Flow

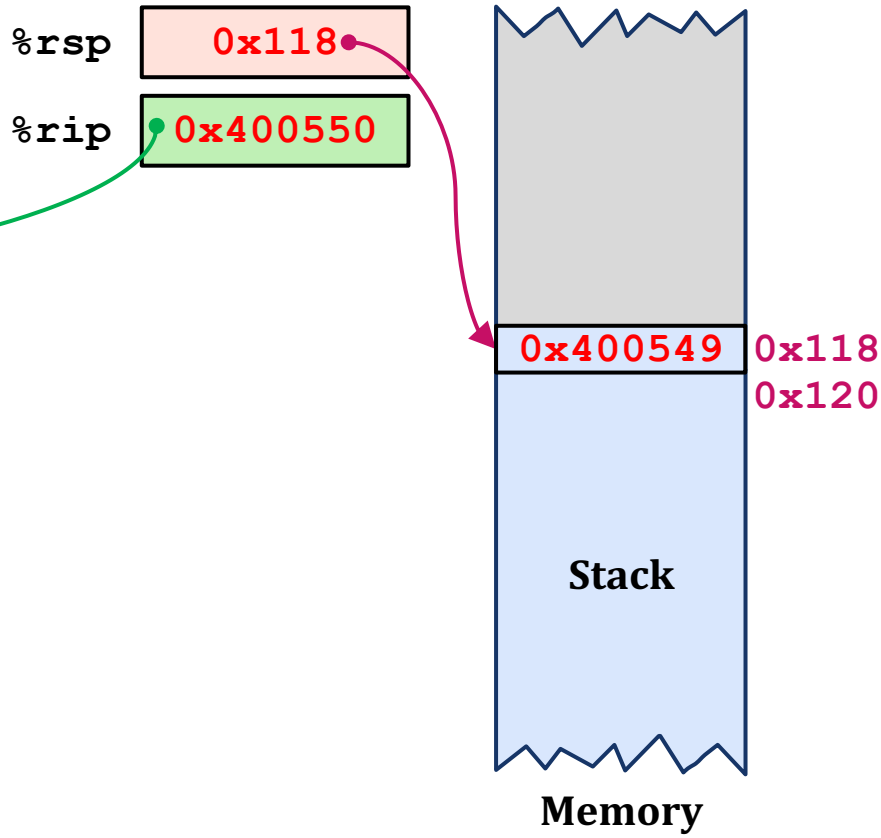
```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



- Procedure call: `call label`
  - Push the return address on the stack
  - Jump to `label`

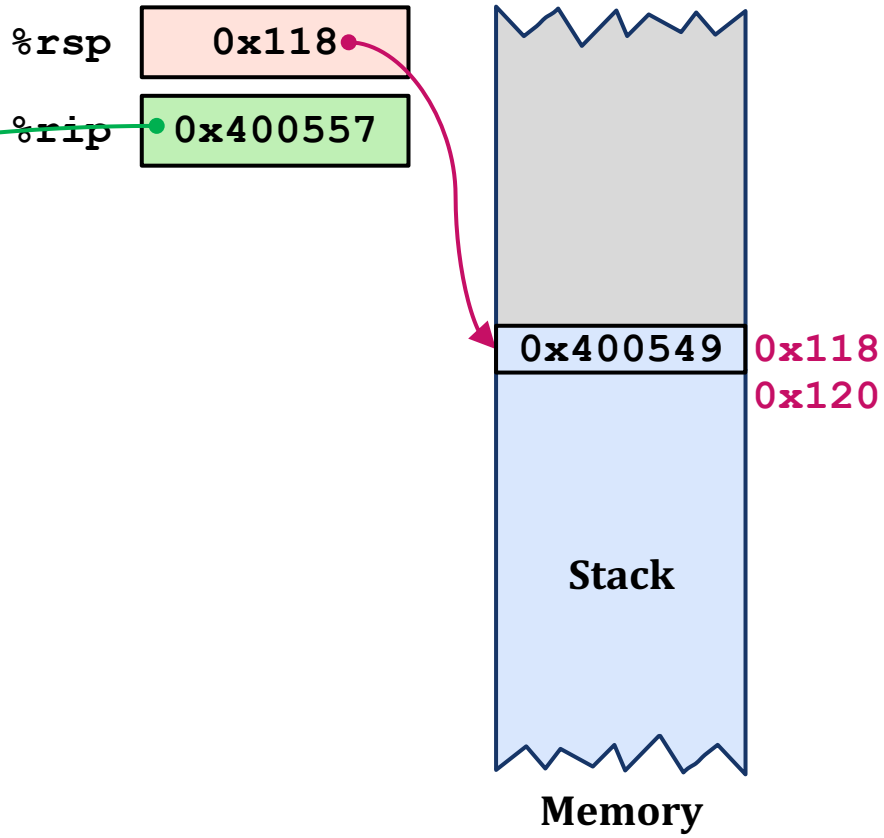
Control Flow

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



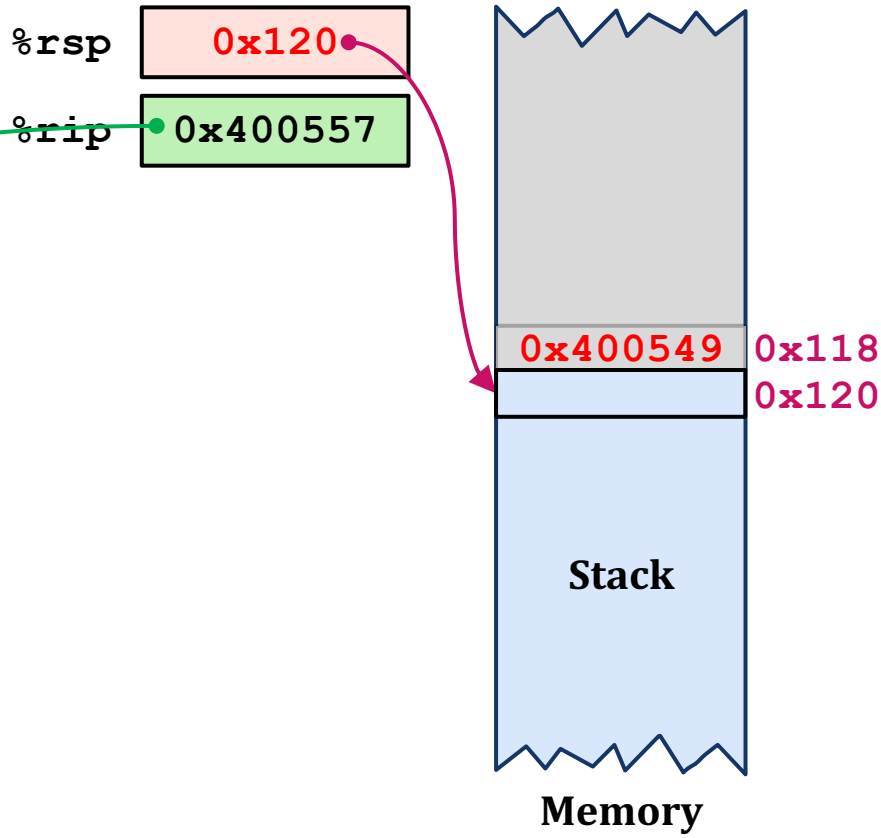
Control Flow

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



Control Flow

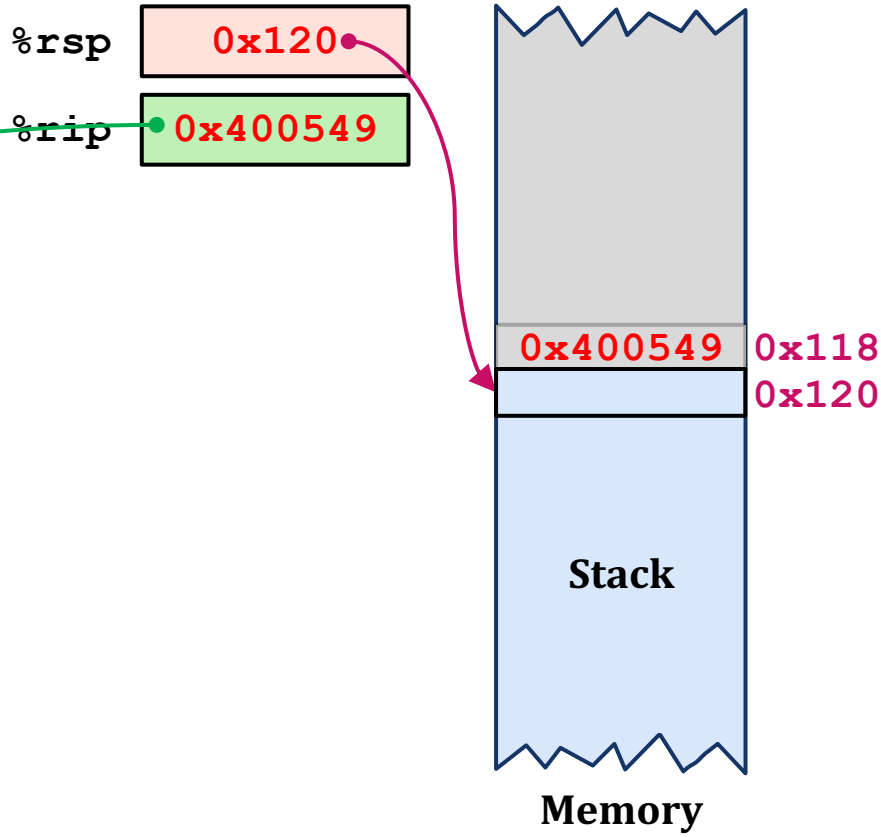
```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



- Procedure return: **`ret`**
  - Pop the address from stack
  - Jump to address

Control Flow

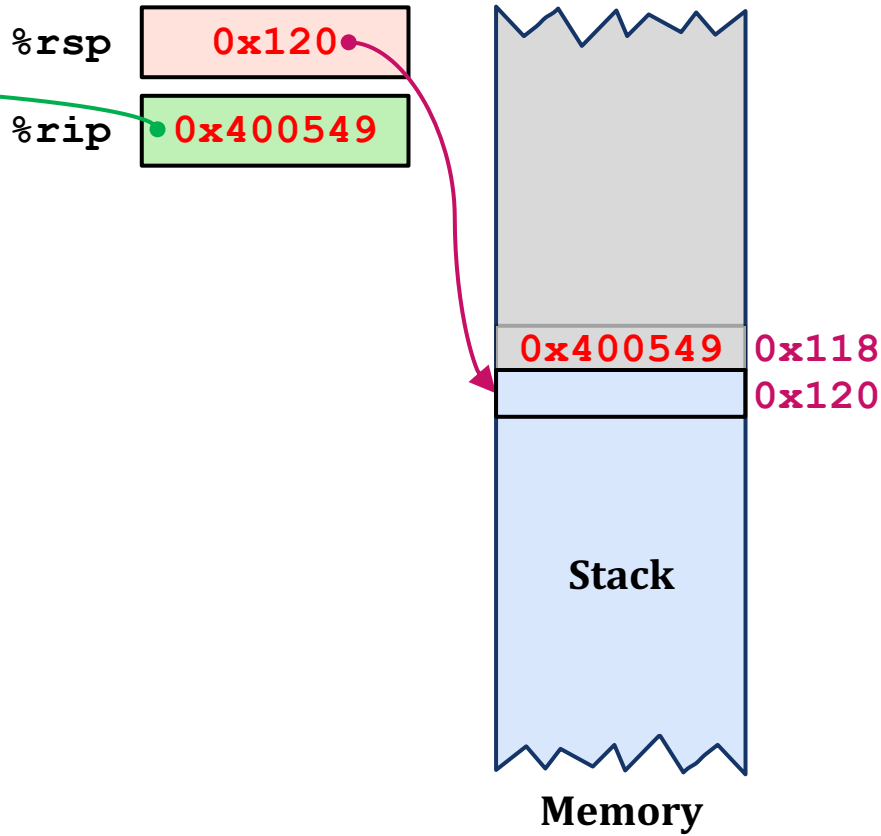
```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



- Procedure return: **`ret`**
  - Pop the address from stack
  - Jump to address

Control Flow

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b
  400557: retq                     # Return
```



Lecture Agenda: Procedures

- Stack Structure

- Calling Conventions
  - Passing Control
  - **Passing Data**
  - Managing Local Data

- Illustrations of Recursion & Pointers

Procedure Data Flow

- Passing arguments
  - First six arguments: registers (`%rdi` → `%rsi` → `%rdx` → `%rcx` → `%r8` → `%r9`)
  - From seventh argument: stack (only when needed)

- Passing the return value: `%rax` register

Code Examples

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

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

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y) → %rax
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
    ⋮
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550: mov    %rdi,%rax         # a
  400553: imul   %rsi,%rax         # a * b → %rax
  400557: retq                     # Return
```

Lecture Agenda: Procedures

- Stack Structure

- Calling Conventions
  - Passing Control
  - Passing Data
  - **Managing Local Data**

- Illustrations of Recursion & Pointers

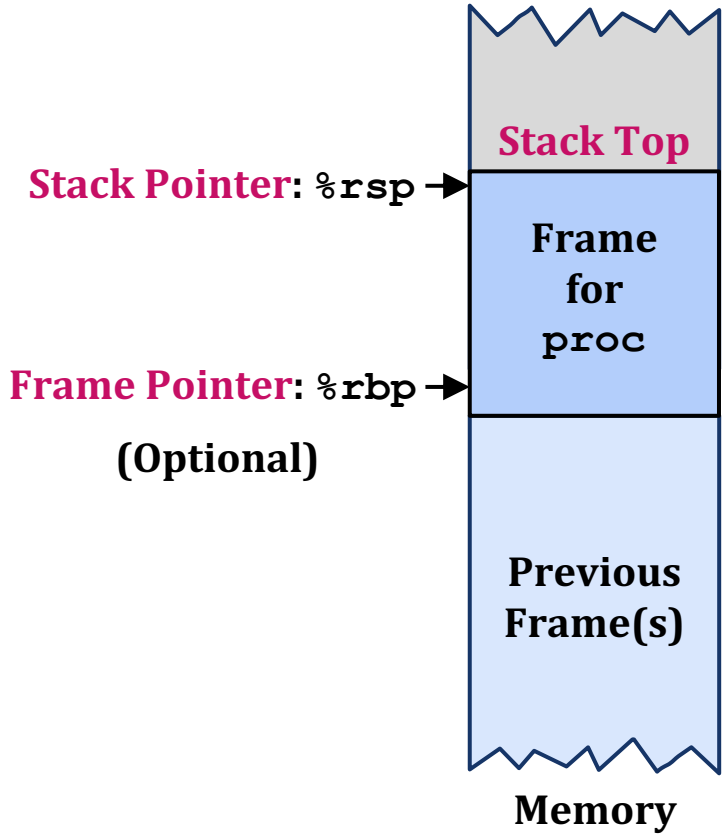
Stack-Based Languages

- Languages that support recursion
  - e.g., C, Pascal, and Java
  - Code must be re-entrant
    - Multiple simultaneous instantiations of a single procedure
  - Need some place to store each instantiation's state
    - Arguments, local variables, and return pointer
- Stack discipline
  - State for the given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in frames
  - State for a single procedure instantiation

Stack Frames

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

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



Call Chain Example

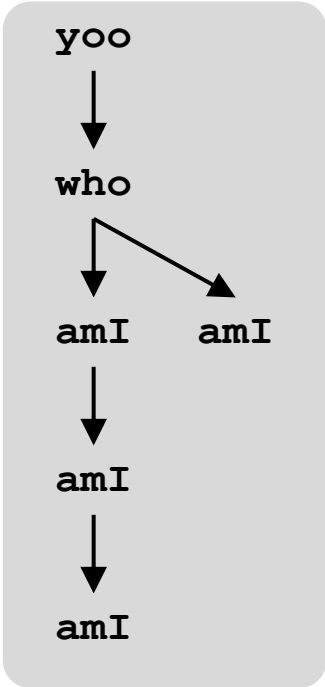
```
yoo(...) {
    ⋮
  who();
    ⋮
}
```

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

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

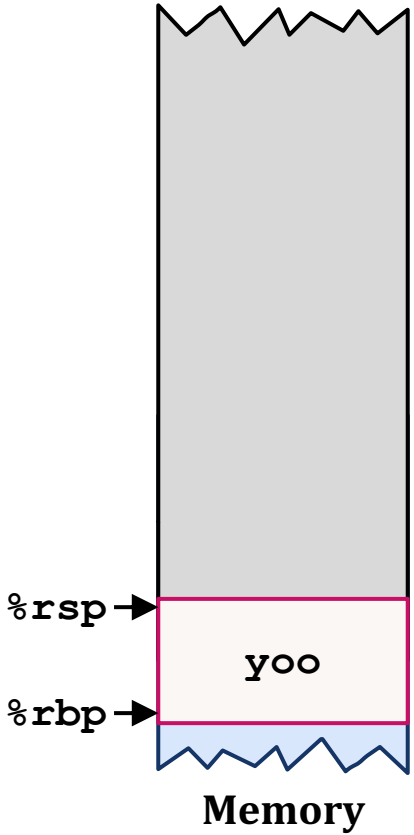
Procedure `amI()` is recursive

Example Call Chain

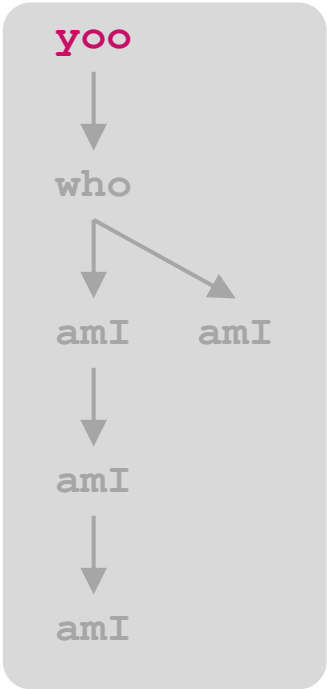


Call Chain Example

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

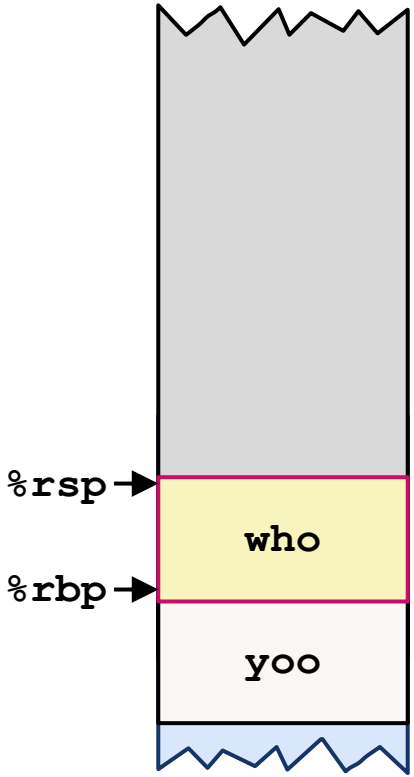


Example Call Chain

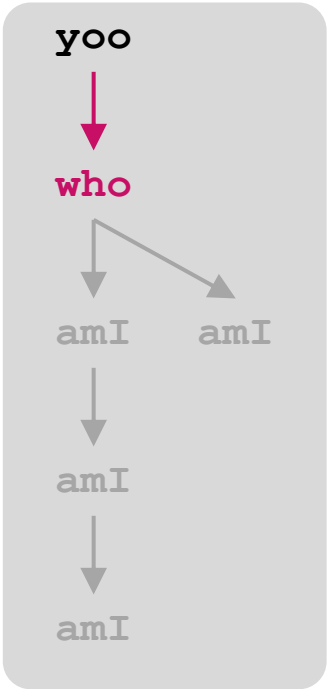


Call Chain Example

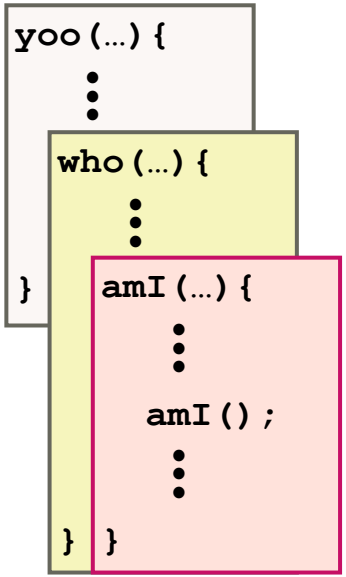
```
yoo(...) {
    ⋮
    who(...) {
        ⋮
    amI();
        ⋮
    amI();
        ⋮
    }
}
```

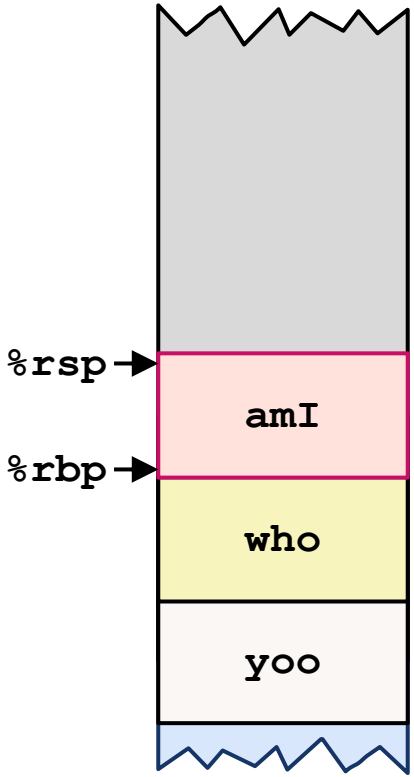


Example Call Chain



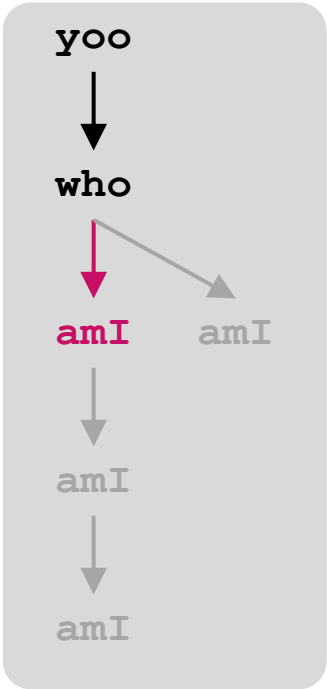
Call Chain Example



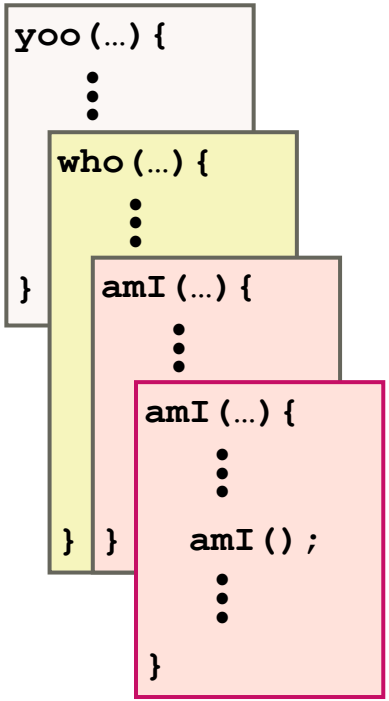


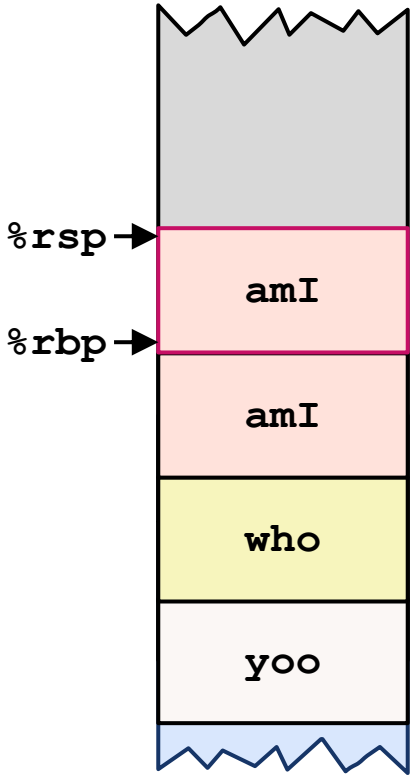
Memory

Example Call Chain



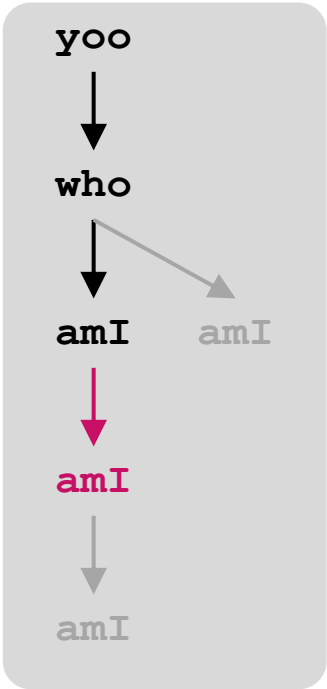
Call Chain Example



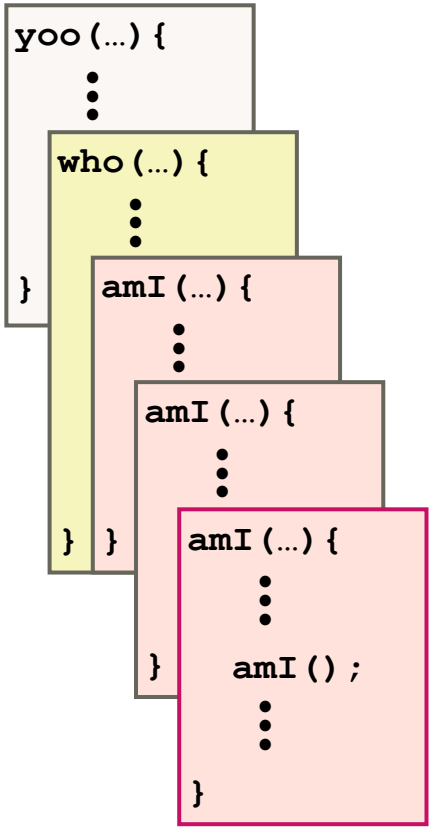


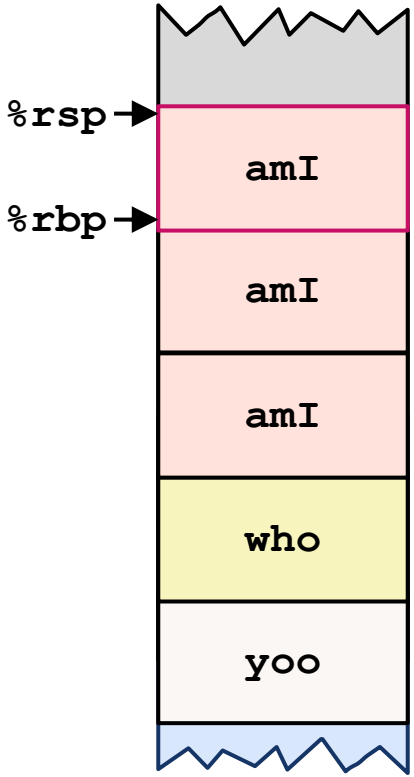
Memory

Example Call Chain



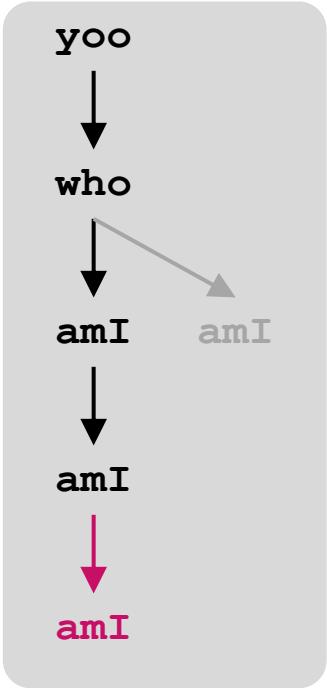
Call Chain Example



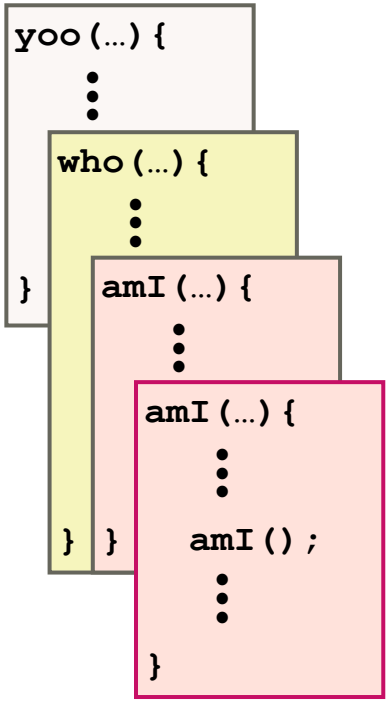


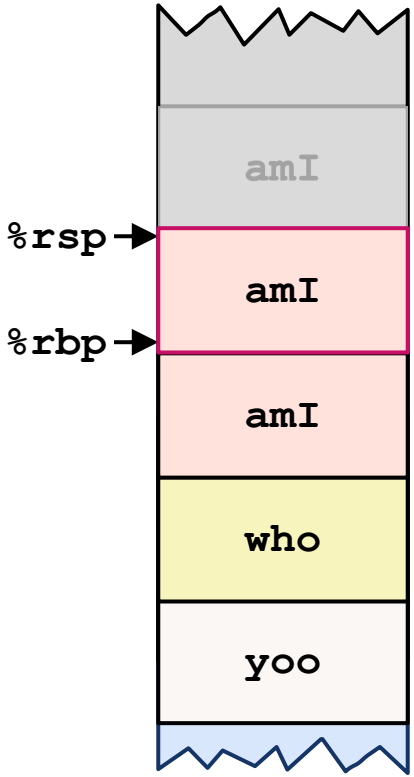
Memory

Example Call Chain



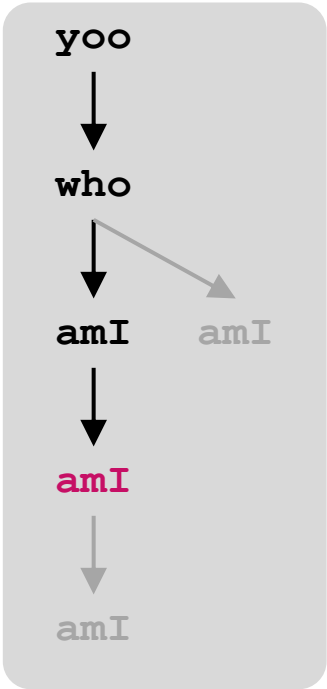
Call Chain Example



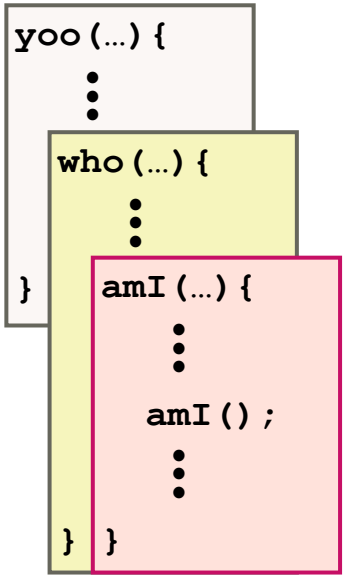


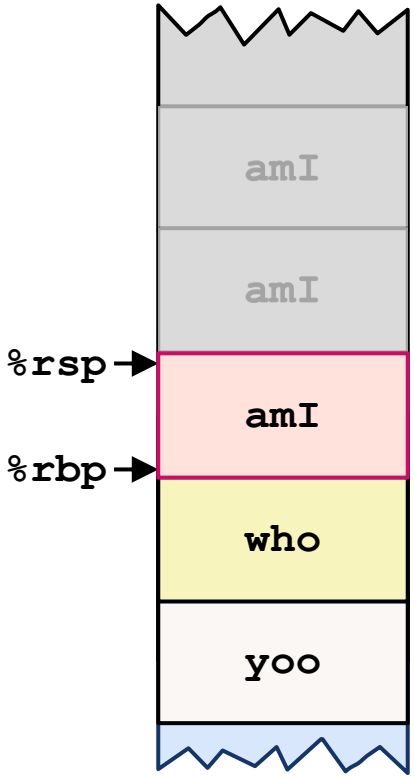
Memory

Example Call Chain



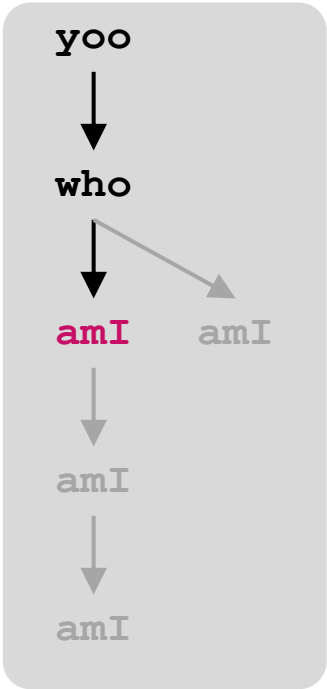
Call Chain Example



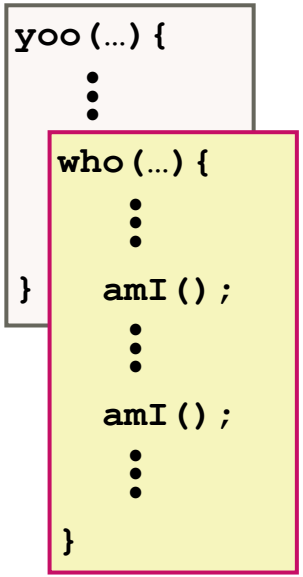


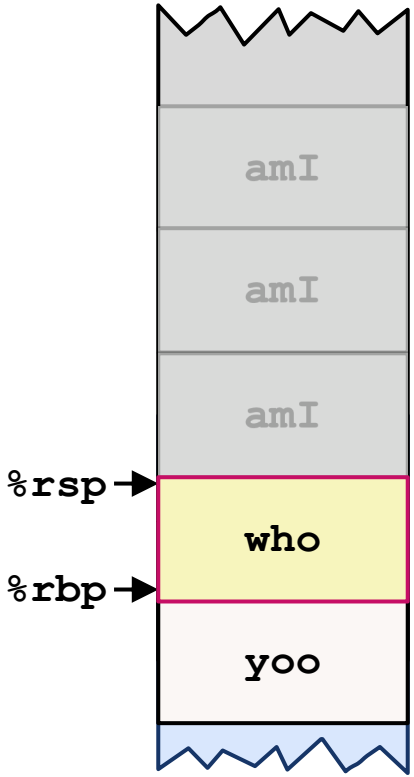
Memory

Example Call Chain



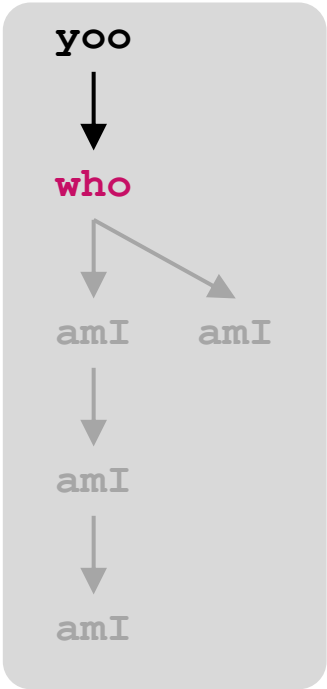
Call Chain Example



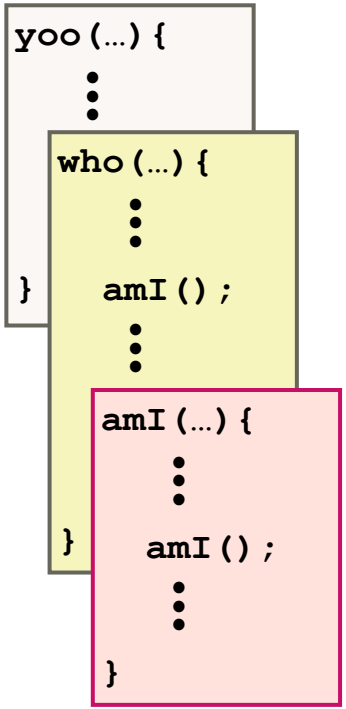


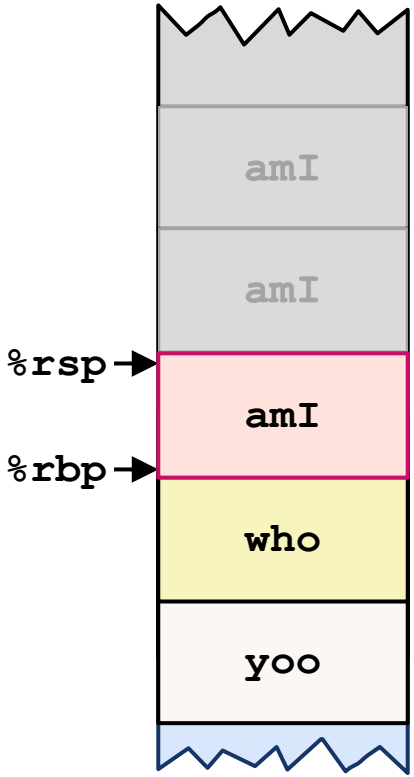
Memory

Example Call Chain



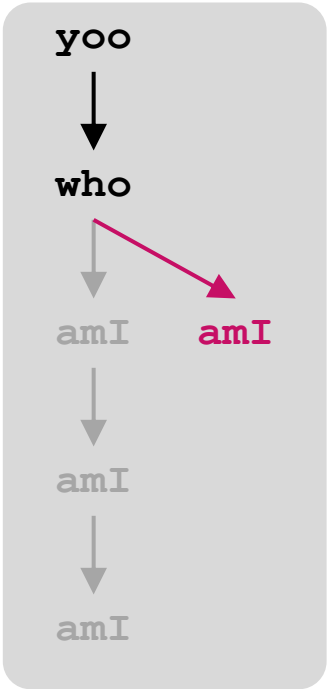
Call Chain Example





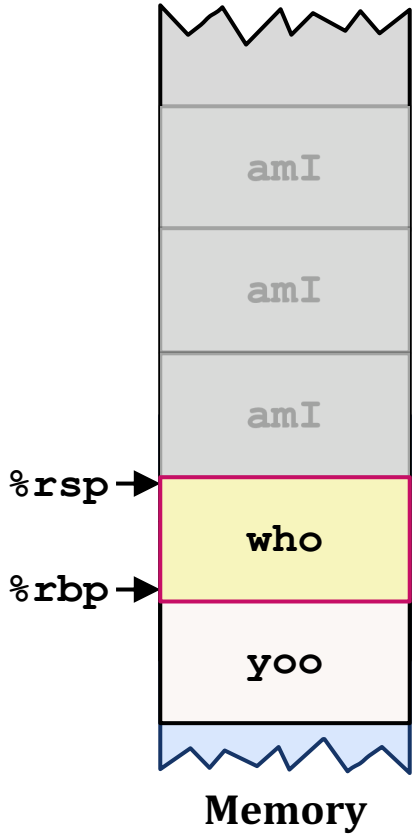
Memory

Example Call Chain



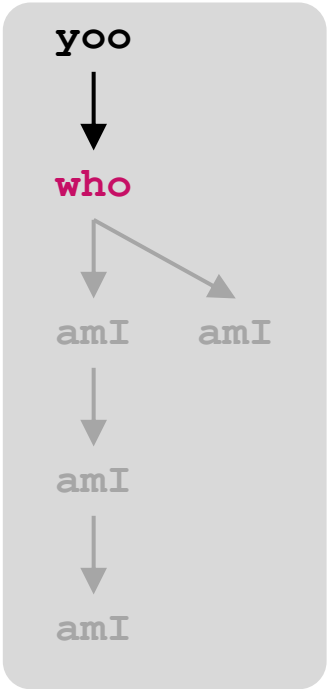
Call Chain Example

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



Memory

Example Call Chain

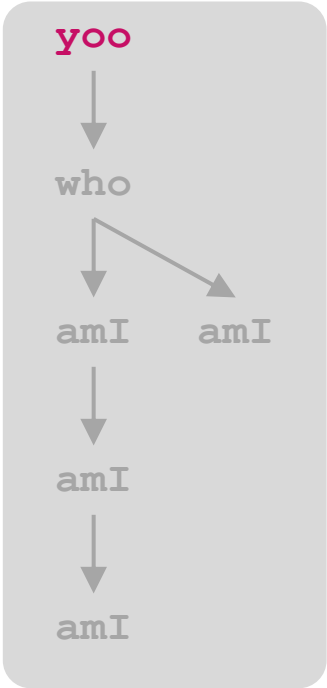


Call Chain Example

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

Memory

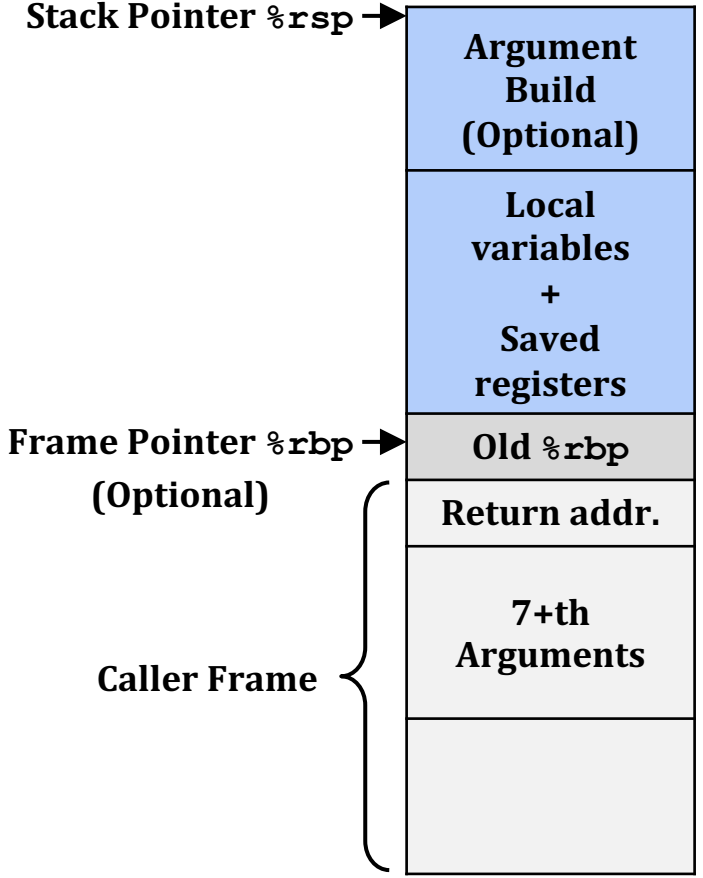
Example Call Chain



x86-64/Linux Stack Frame

- Current (callee) stack frame (top to bottom)
  - Argument build (optional):
    parameters for function about to call
  - Local variables (if unable to keep in registers)
  - Saved register context
  - Old frame pointer (optional)

- Caller stack frame
  - Return address
    - Pushed by `call` instruction
  - 7th~ arguments for this call (if exist)



Example: `incr`

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

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

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

Example: `incr`

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

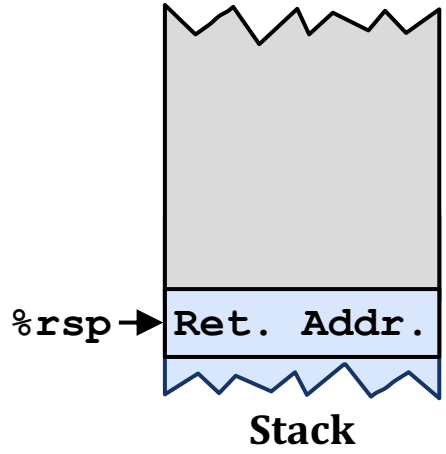
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

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

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

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

Register	Use(s)
%rdi	XX
%rsi	XX
%rax	XX



Example: `incr`

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

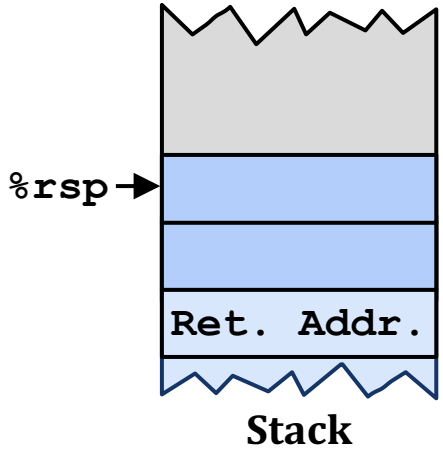
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

Register	Use(s)
%rdi	XX
%rsi	XX
%rax	XX



Example: `incr`

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

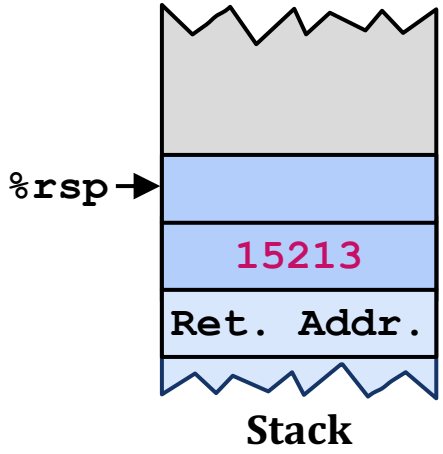
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

Register	Use(s)
%rdi	XX
%rsi	XX
%rax	XX



Example: `incr`

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

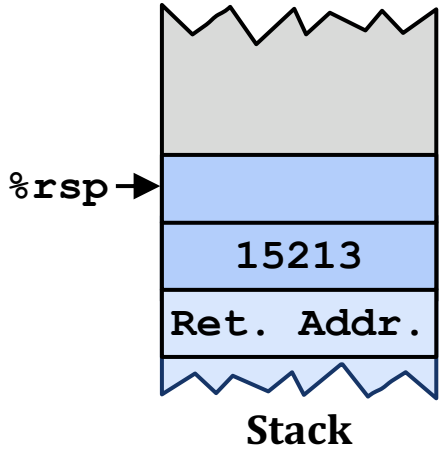
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

Register	Use(s)
%rdi	XX
%rsi	3000
%rax	XX



Example: `incr`

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

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

Register	Use(s)
%rdi	XX
%rsi	3000

Aside: **`movl $3000, %esi`**

- **`movl`** instruction with **`%exx`** (dest) zeros out high order 32 bits.
- Why use **`movl`** instead of **`movq`**? 2-byte shorter.

Example: `incr`

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

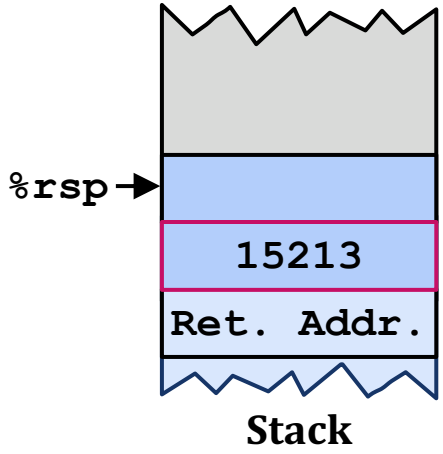
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

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



Example: `incr`

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

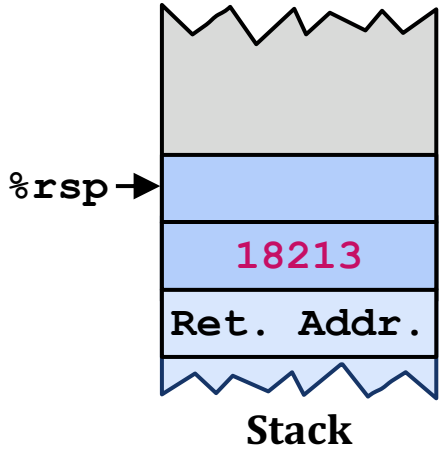
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

Register	Use(s)
%rdi	&v1
%rsi	18213
%rax	15213



Example: `incr`

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

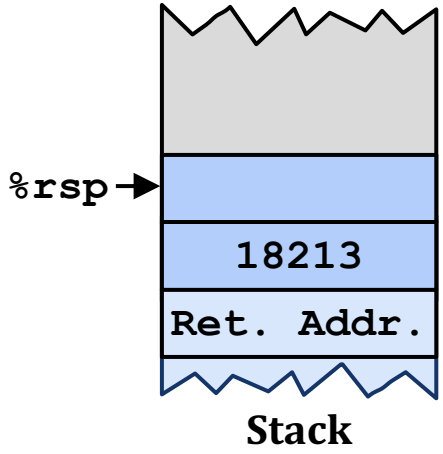
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

Register	Use(s)
%rdi	&v1
%rsi	18213
%rax	33426



Example: `incr`

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

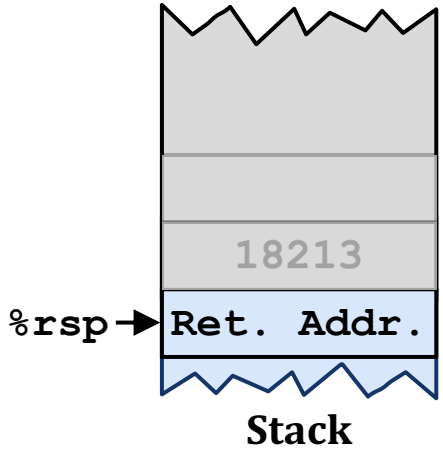
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

Register	Use(s)
%rdi	&v1
%rsi	18213
%rax	33426



Example: `incr`

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

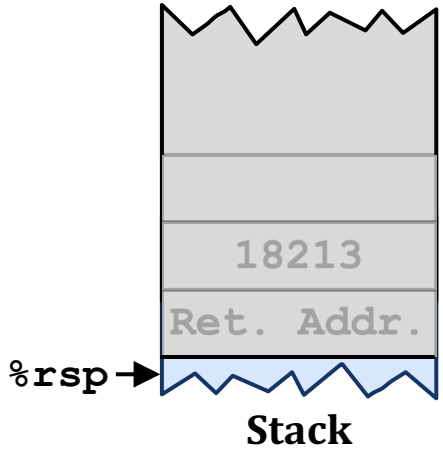
```
incr:
   movq     (%rdi), %rax
   addq     %rax, %rsi
   movq     %rsi, (%rdi)
   ret
```

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

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

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

Register	Use(s)
%rdi	&v1
%rsi	18213
%rax	33426



Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the caller
  - `who` is the callee
- Can register be used for temporary storage?
  - Register `%rdx` might be overwritten by `who`
  - This could be trouble: something should be done!
    - Need some coordination
- Conventions
  - Caller-saved: caller saves temporary values in its frame before the call
  - Callee-saved: callee saves temporary values in its frame before using and restores them before returning to caller

```
yoo:
      ⋮
    movq $15213, %rdx
    call who
    addq %rdx, %rax
      ⋮
    ret
```

```
who:
      ⋮
    subq $18213, %rdx
      ⋮
    ret
```

x86-64 Linux Register Usage#1

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure

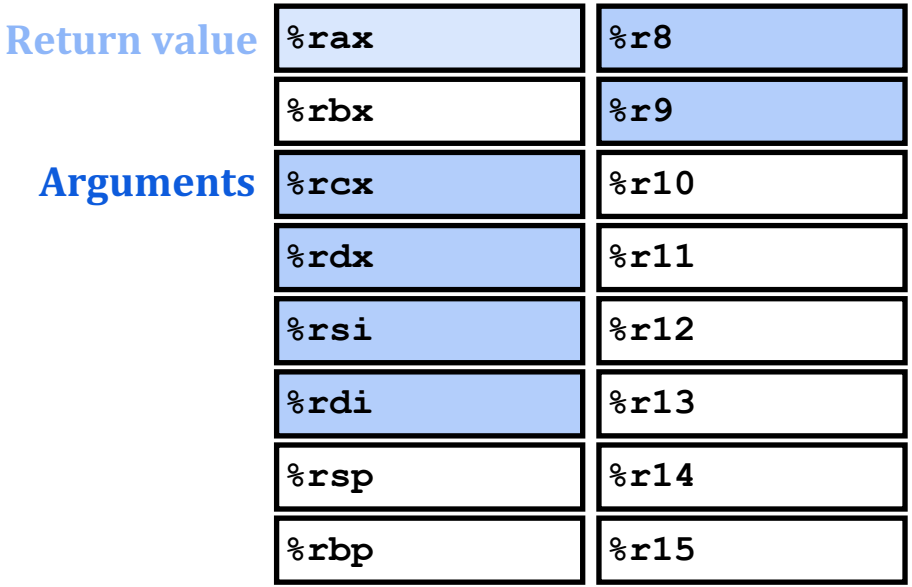
Return value

`%rax`	`%r8`
`%rbx`	`%r9`
`%rcx`	`%r10`
`%rdx`	`%r11`
`%rsi`	`%r12`
`%rdi`	`%r13`
`%rsp`	`%r14`
`%rbp`	`%r15`

x86-64 Linux Register Usage#1

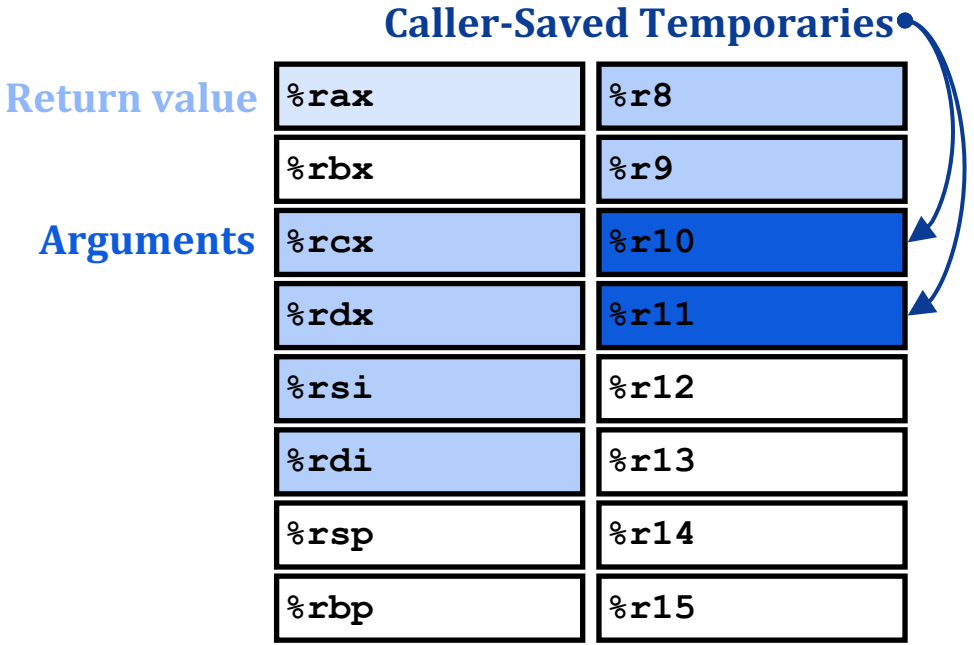
- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure

- `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure



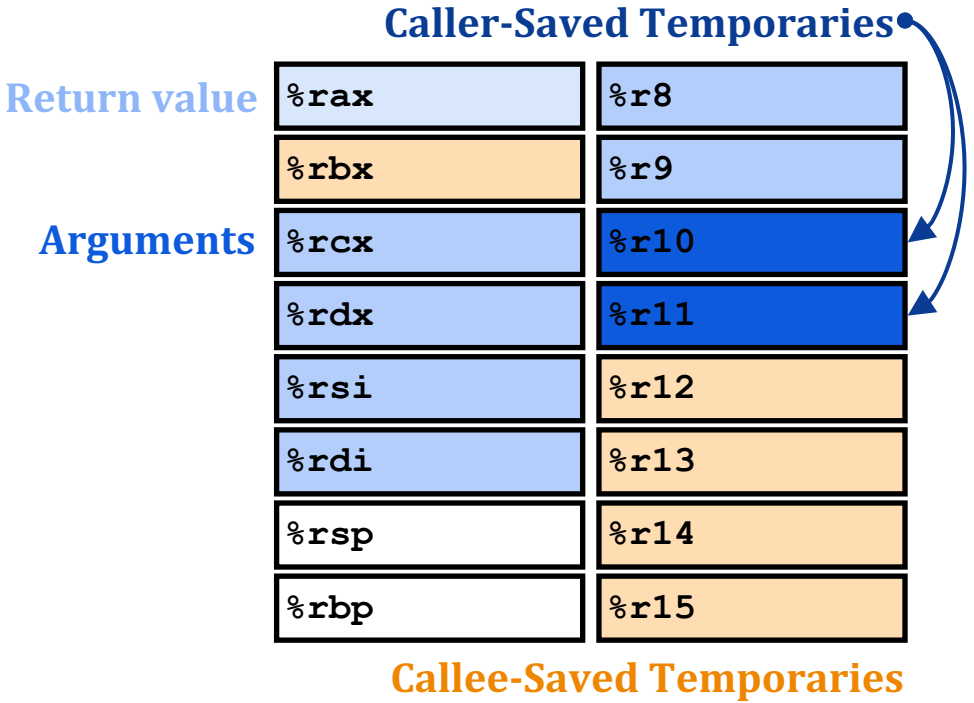
x86-64 Linux Register Usage#1

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved
  - Can be modified by procedure



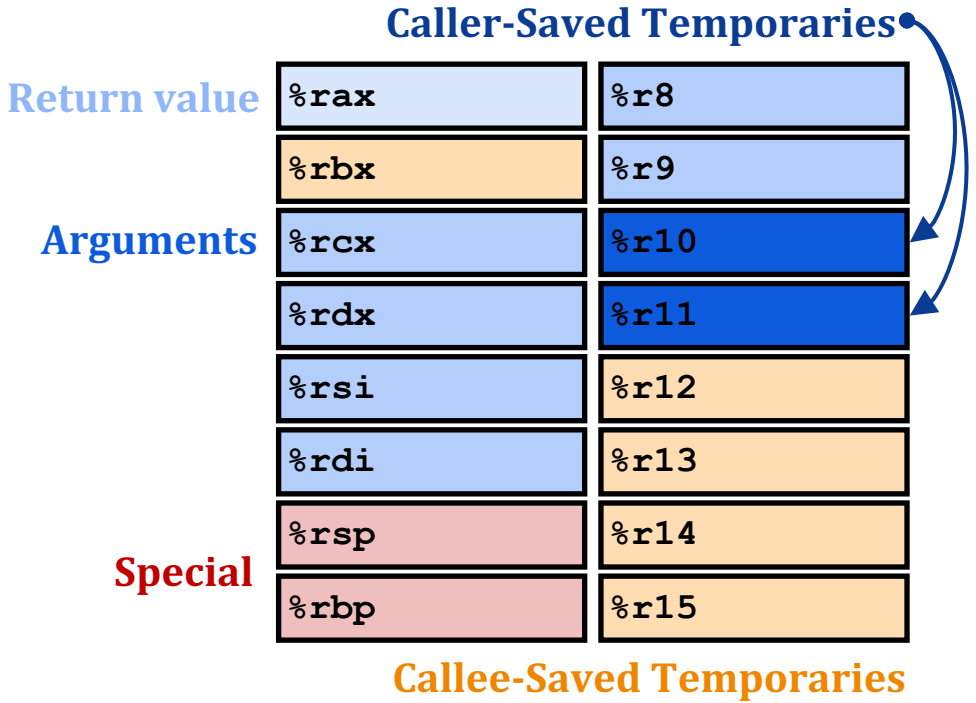
x86-64 Linux Register Usage#2

- `%rbx,%r12,%r13,%r14,%r15`
  - Callee-saved
  - Callee must save & restore



x86-64 Linux Register Usage#2

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



Callee-Saved Example

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

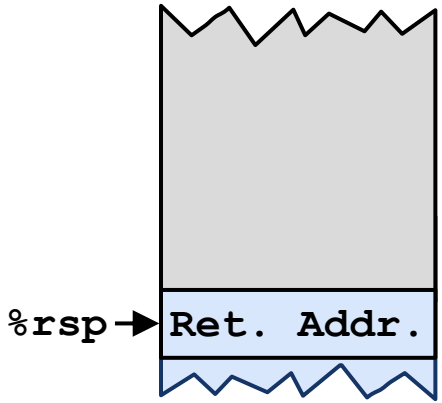
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

- Argument **`x`** comes in register **`%rdi`**
  - Need to keep the value until the end of the procedure
- Need **`%rdi`** for calling the **`incr`** procedure.
- Where should we put **`x`** to use it after the **`incr`** procedure?

Callee-Saved Example

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

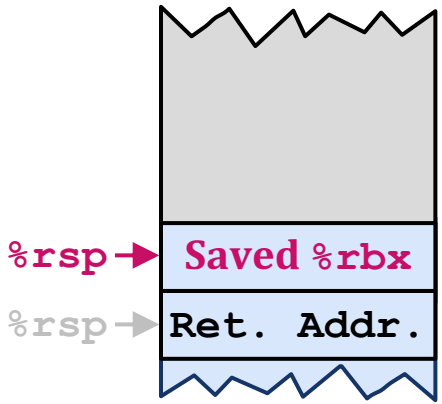
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

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

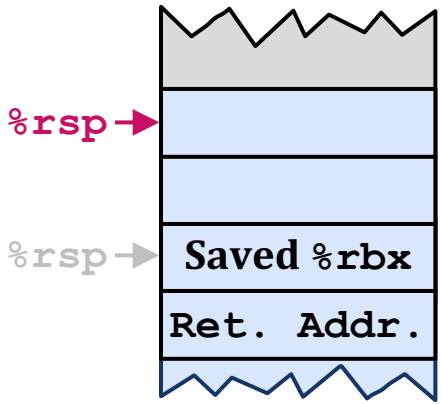
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

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

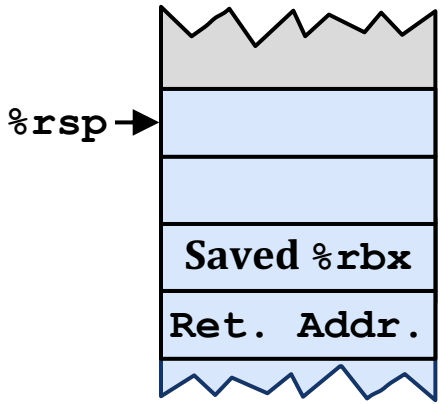
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

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

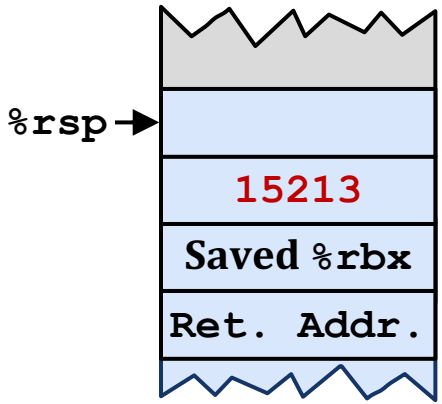
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx # x → %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

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

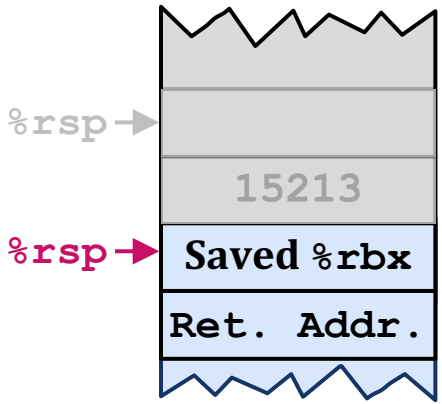
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx # x → %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

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

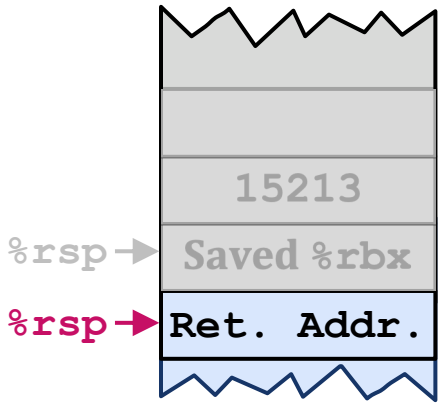
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx # x → %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
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 # x → %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx # Saved → %rbx
  ret
```



Lecture Agenda: Procedures

- Stack Structure
- Calling Conventions
  - Passing Control
  - Passing Data
  - Managing Local Data
- Illustrations of Recursion & Pointers

Recursive Function Example

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

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

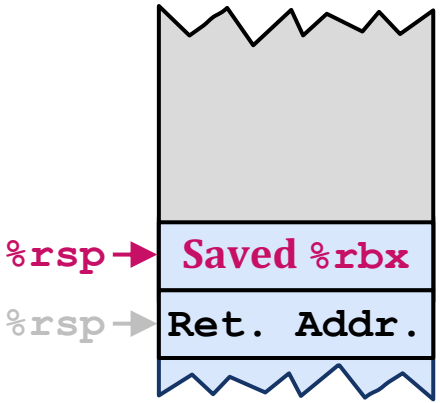
Recursive Function Example: Terminal Case

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

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

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

Recursive Function Example: Register Save

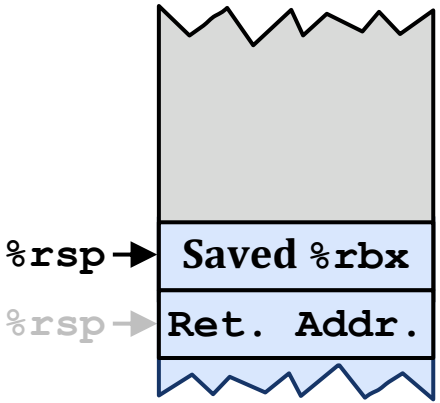



```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return   (x & 1)
           + pcount_r(x >> 1);
}
```

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

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

Recursive Function Example: Call Setup

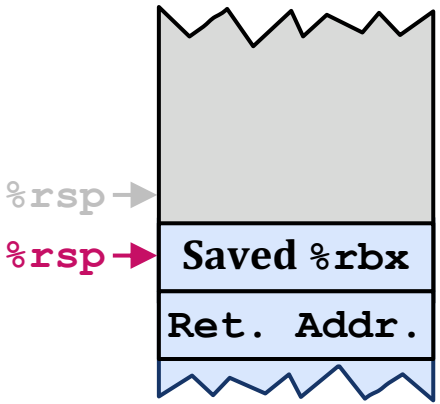



```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return   (x & 1)
           + pcount_r(x >> 1);
}
```

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

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

Recursive Function Example: Recursive Call

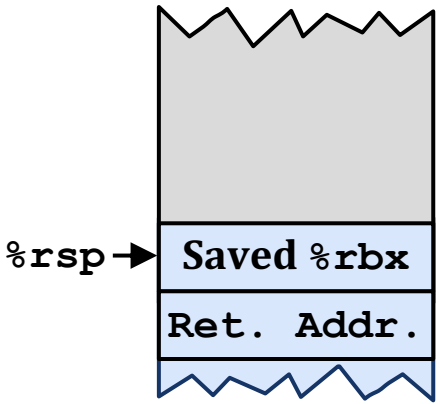



```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return   (x & 1)
           + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	0	Rec. argument
%rax	Ret. value	Rec. Ret. value
%rbx	x & 1	Callee-saved

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

Recursive Function Example: Result

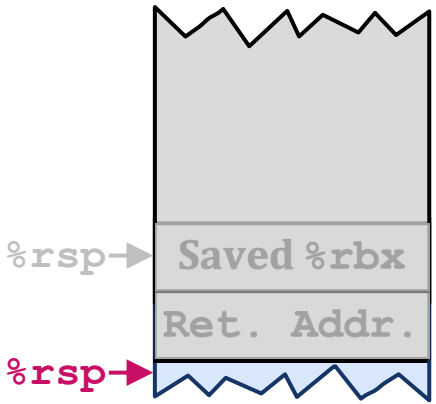



```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return   (x & 1)
           + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	0	Rec. argument
%rax	Ret. value	Ret. value
%rbx	x & 1	Callee-saved

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

Recursive Function Example: Completion




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

Register	Use(s)	Type
%rdi	0	Rec. argument
%rax	Ret. value	Ret. value
%rbx	Saved %rbx	Callee-saved

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

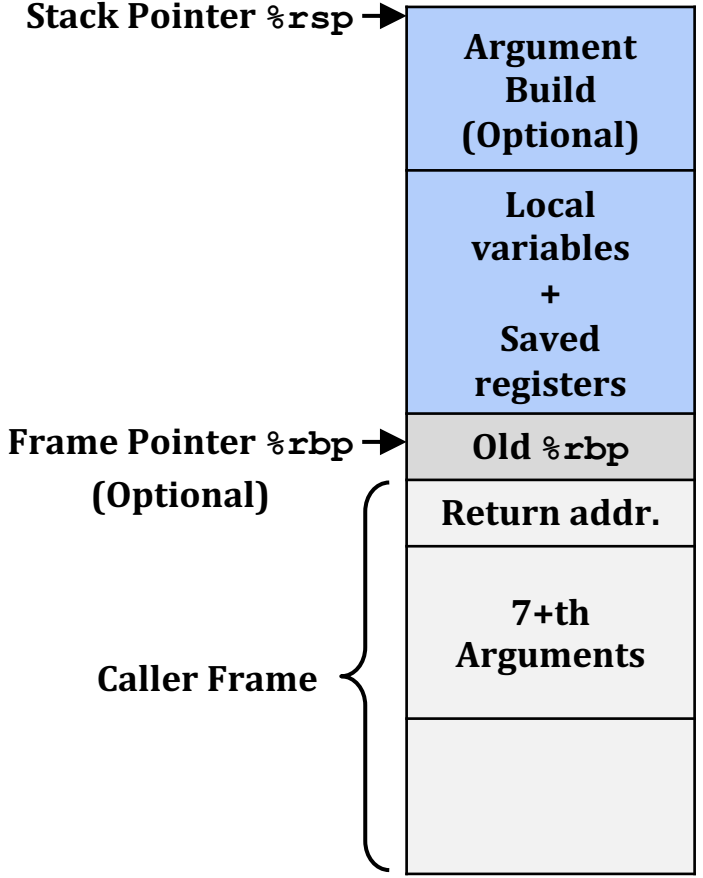
Observations

- Handled w/o special consideration
  - Stack frames enable each function call to have private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent a function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - Stack discipline follows call/return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out

- Also works for mutual recursion
  - P calls Q; Q calls P

x86-64 Procedure Summary

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



Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0 + b1 + b2 + b3 + b4;
}

long add10(long a0, long a1, long a2, long a3, long a4,
           long a5, long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4) + add5(a5, a6, a7, a8, a9);
}
```

- Where are **a0**, ..., **a9** passed? **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8**, **%r9**, and stack
- Where are **b0**, ..., **b4** passed? **%rdi**, **%rsi**, **%rdx**, **%rcx**, and **%r8**
- Which registers do we need to save?
  Ill-posed question; needs assembly
  → **%rbx**, **$rbp**, and **%r9** (during the first add5)

Small Exercise

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0 + b1 + b2 + b3 + b4;
}

long add10(long a0, long a1, long a2, long a3, long a4,
           long a5, long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4) + add5(a5, a6, a7, a8, a9);
}
```

```
add5:
    addq    %rsi, %rdi
    addq    %rdi, %rdx
    addq    %rdx, %rcx
    leaq    (%rcx,%r8), %rax
    ret
```

```
add10:
    pushq   %rbp
    pushq   %rbx
    movq    %r9, %rbp
    call    add5
    movq    %rax, %rbx
    movq    48(%rsp), %r8
    movq    40(%rsp), %rcx
    movq    32(%rsp), %rdx
    movq    24(%rsp), %rsi
    movq    %rbp, %rdi
    call    add5
    addq    %rbx, %rax
    popq    %rbx
    popq    %rbp
    ret
```

[CSED211] Introduction to Computer Software Systems

Lecture 6: Procedures

Prof. Jisung Park

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.10.04