CS201 – Lecture 9
IA32 Procedures

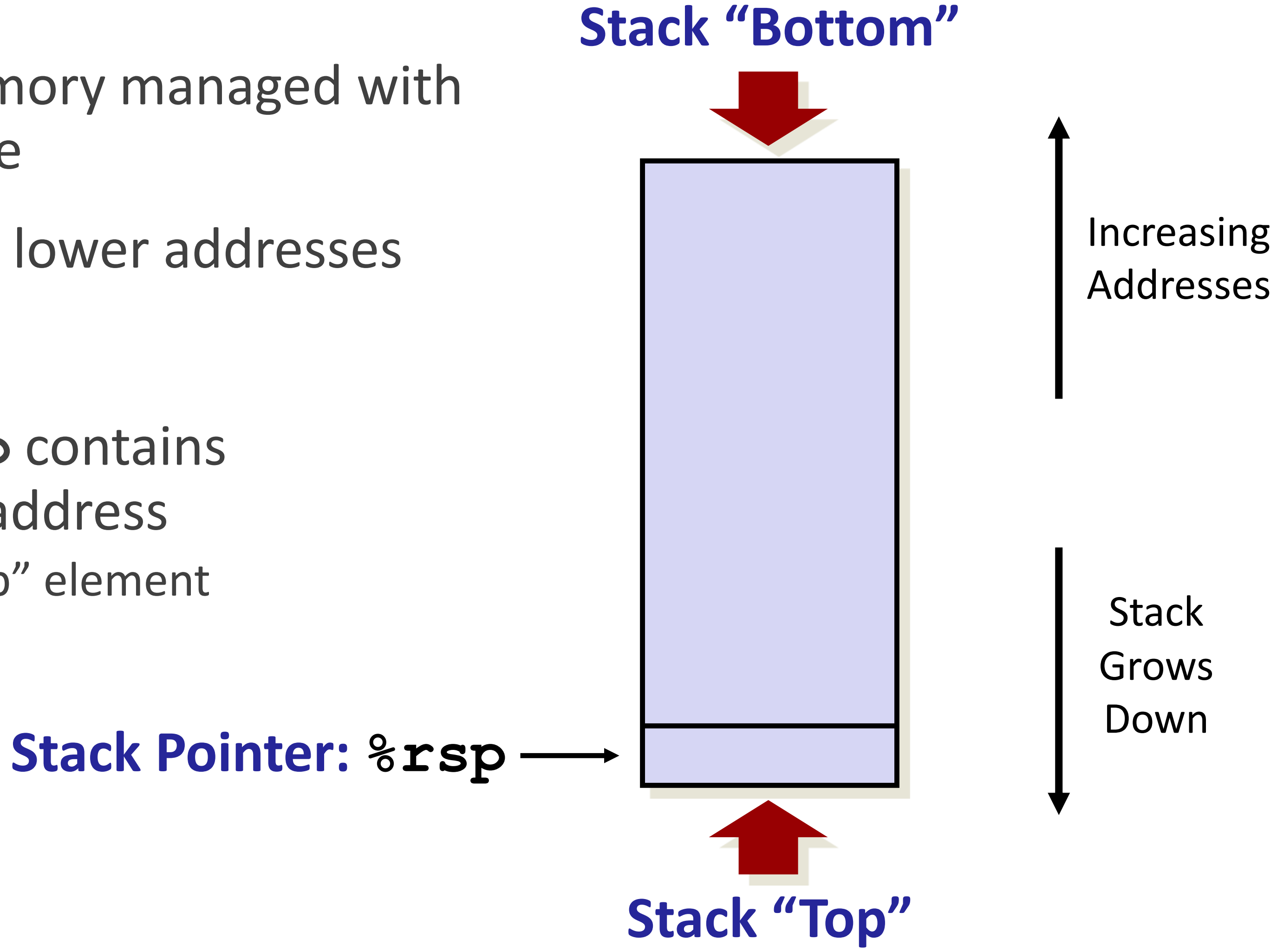
RAOUL RIVAS

PORTLAND STATE UNIVERSITY

Announcements

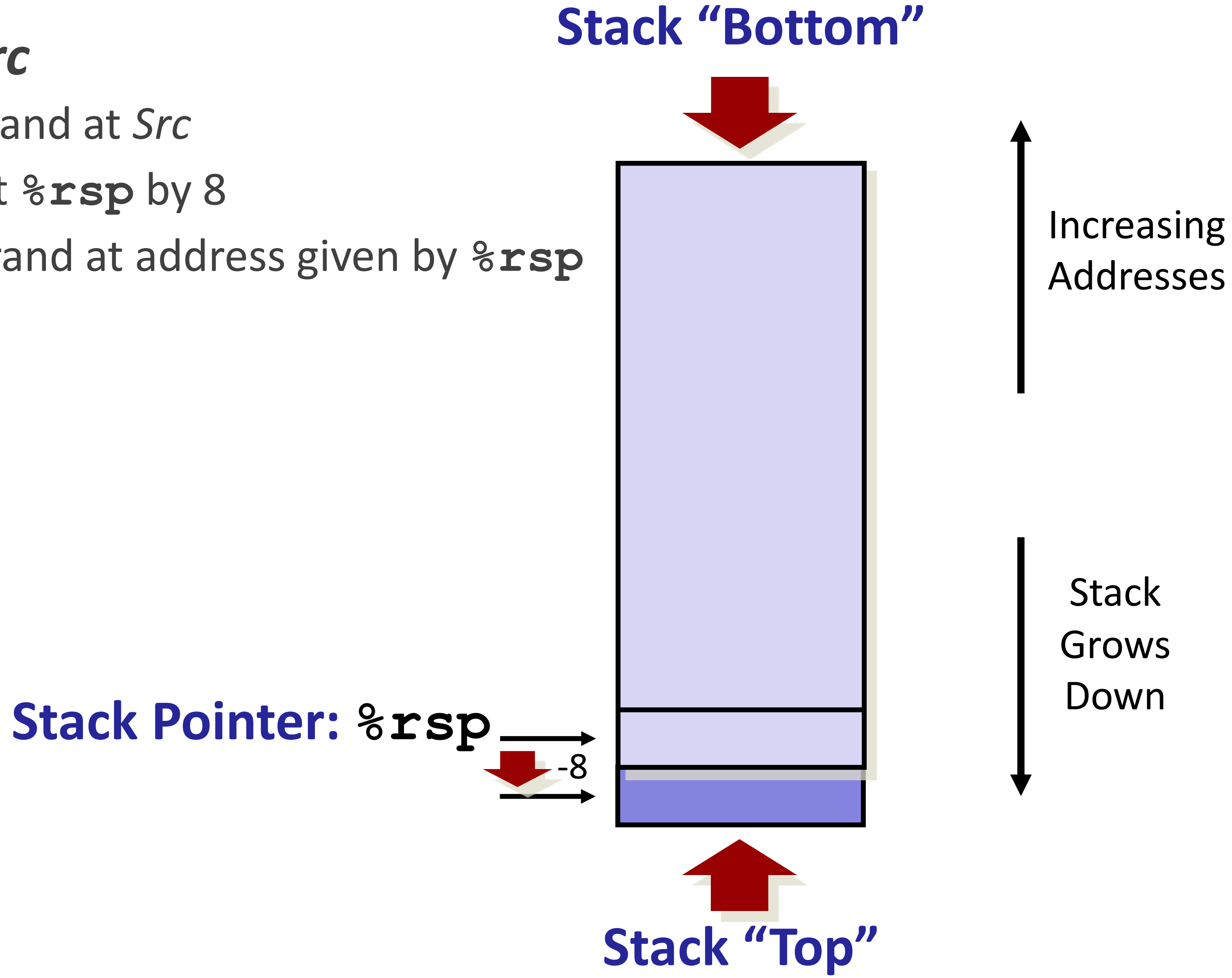
x86-64 Stack

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



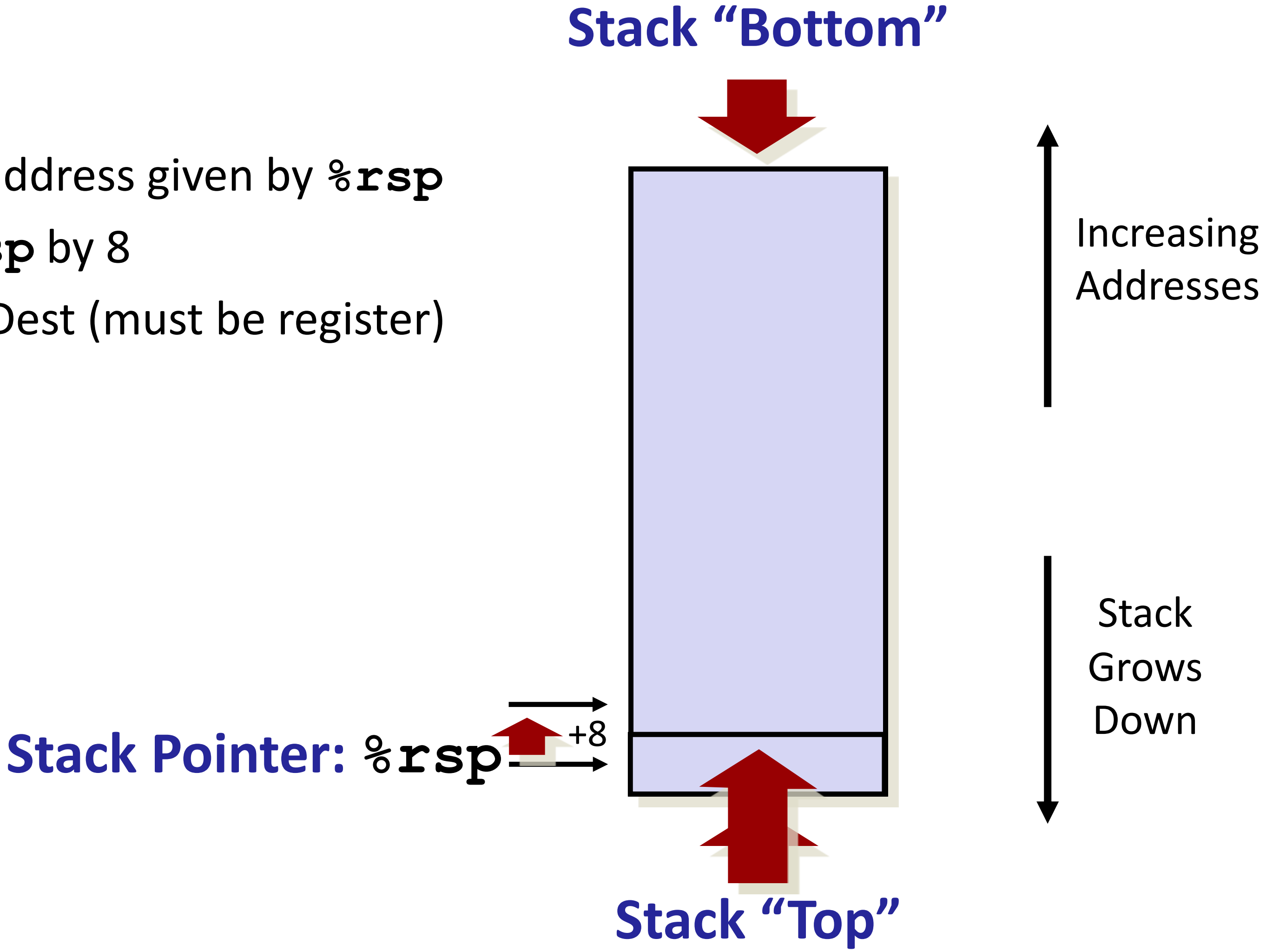
x86-64 Stack: Push

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



x86-64 Stack: Pop

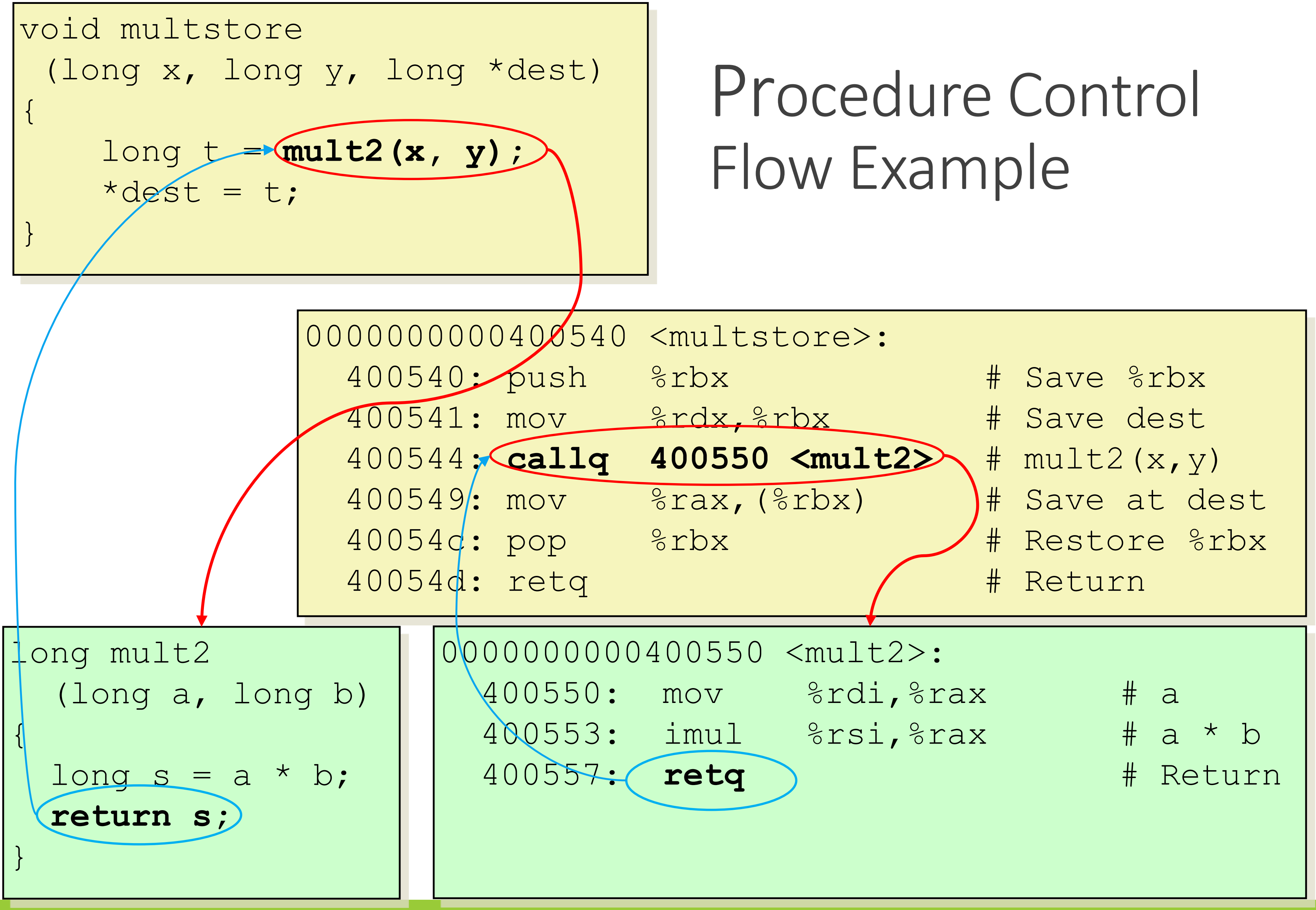
- **`popq`** ***Dest***
  - Read value at address given by **`%rsp`**
  - Increment **`%rsp`** by 8
  - Store value at Dest (must be register)



Procedure Control Flow

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

Procedure Control Flow Example



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

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

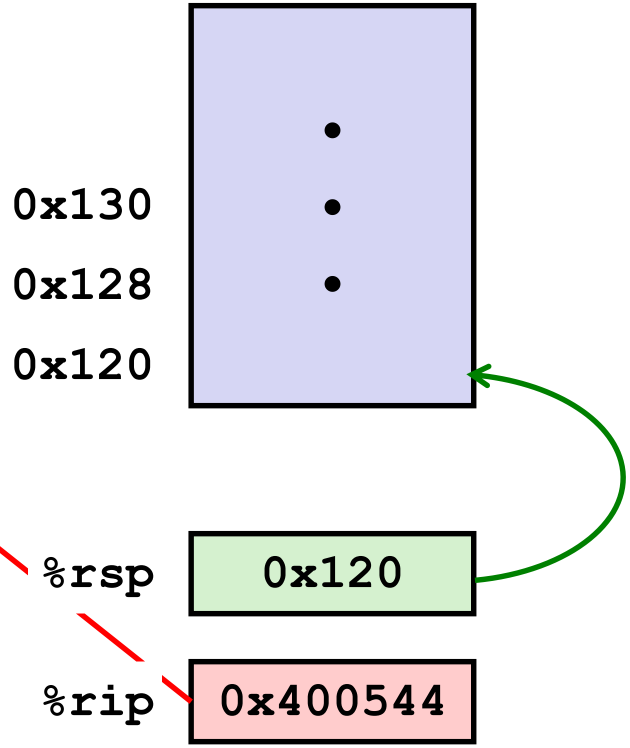
```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

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

Call Flow

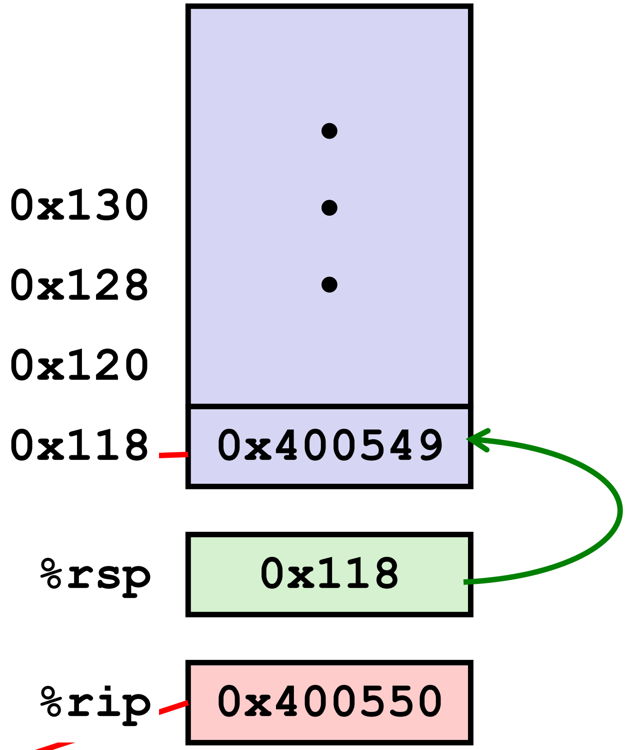
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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



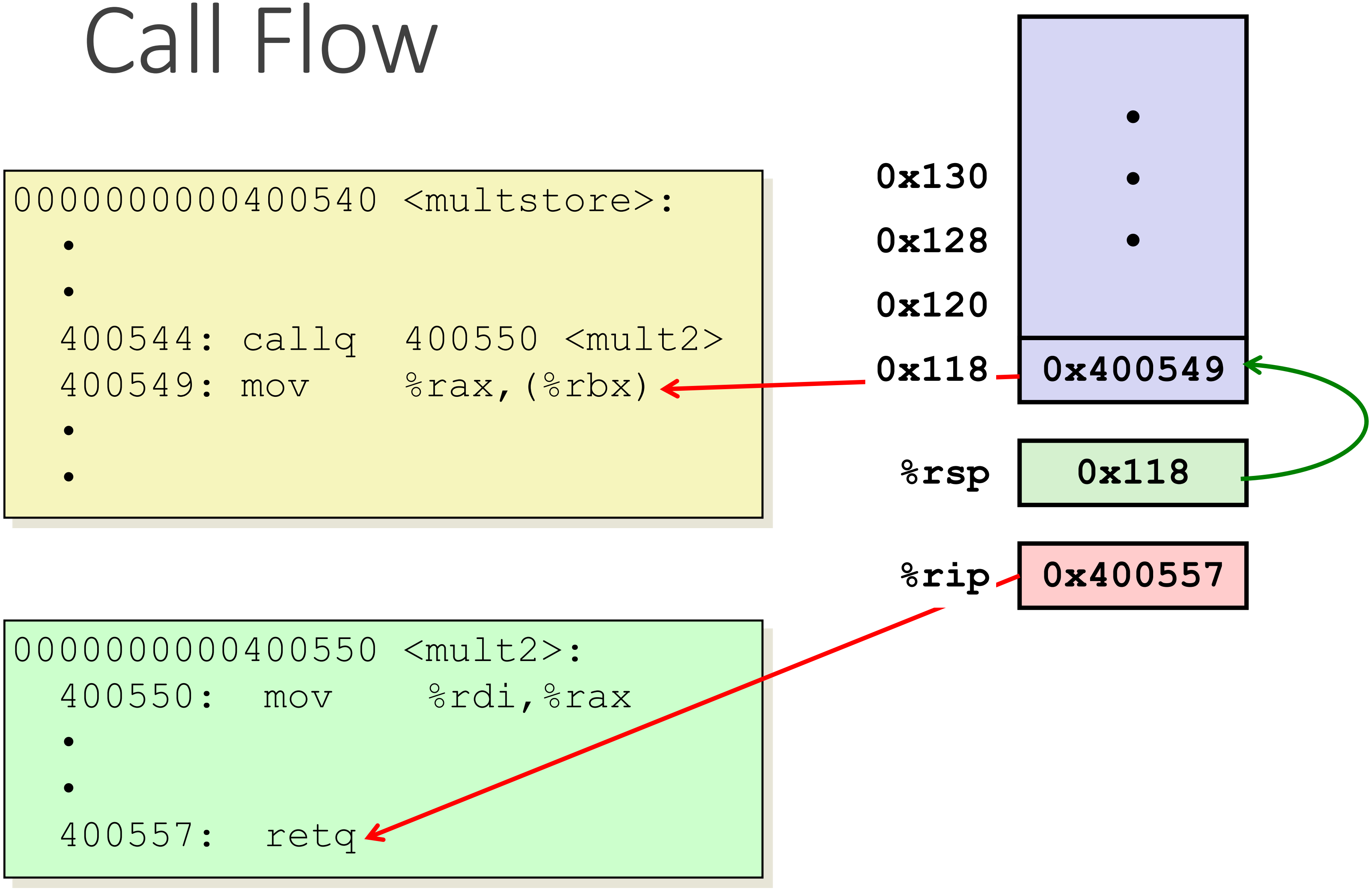
Call Flow

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



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

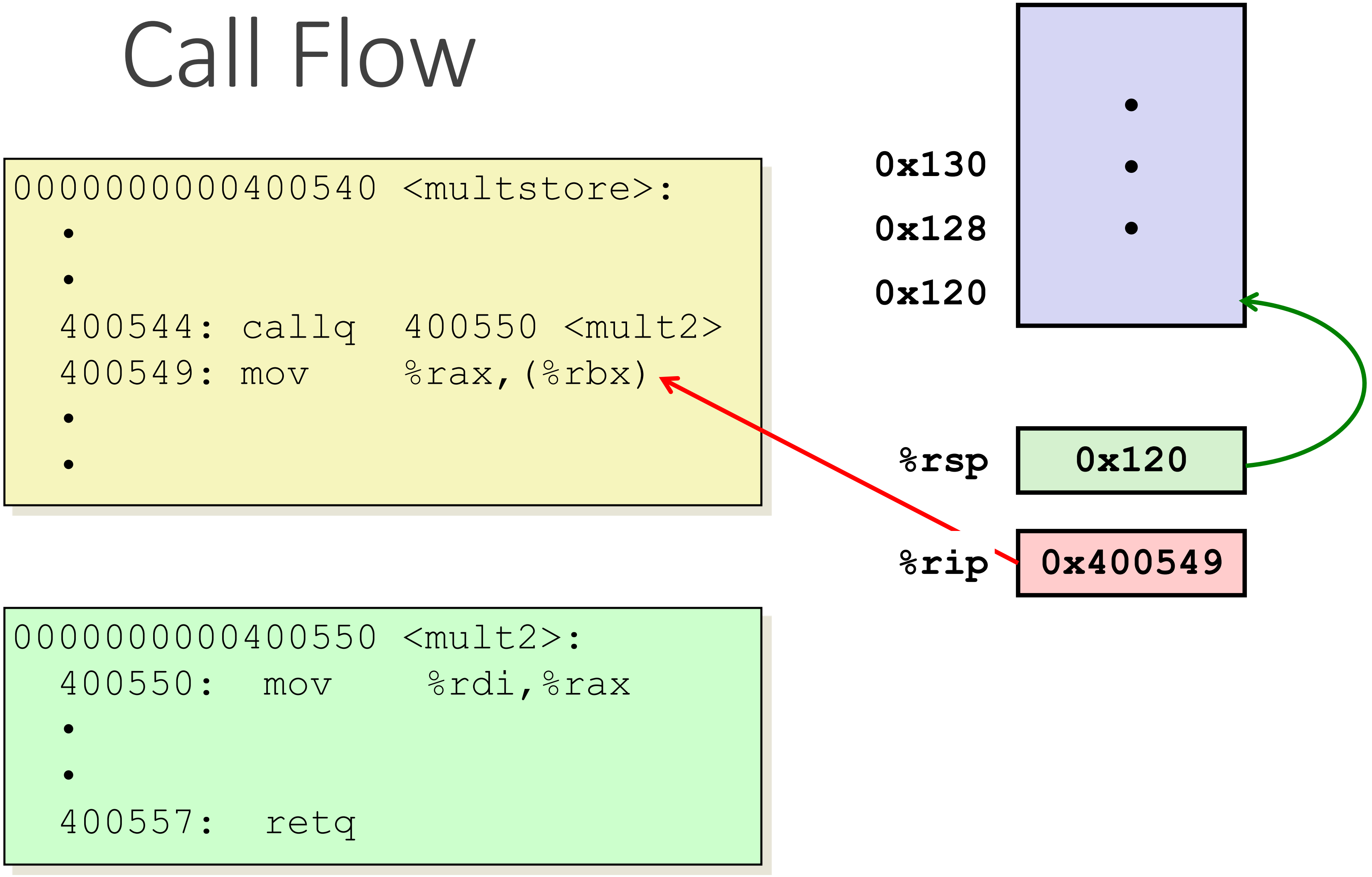
Call Flow



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

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

Call Flow



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

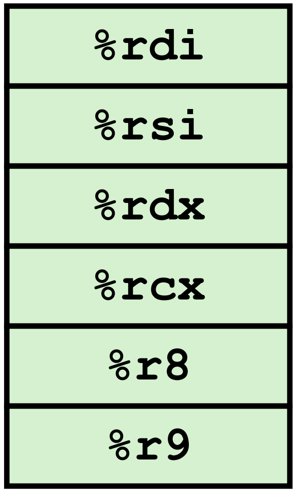
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Calling Convention

- Specification detailing how a particular language and platform implement function calls
  - **Argument Passing**: How we pass arguments?
  - **Return Value**: How we return values?
  - **Register Saving Convention**: Which registers are preserved?
  - **Stack Frame Format and Management**: How stack is managed?
- Many Calling Conventions
  - CDECL – Used by C in x86 platforms
  - STDCALL – Windows API calls
  - System V AMD64 – GCC/Linux in x64 platforms

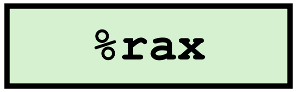
X64 Argument Passing

- Registers
  - First 6 arguments



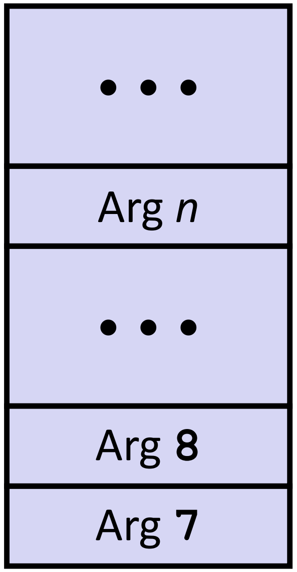
`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

  - Return value



`%rax`

- Stack



• • •
Arg *n*
• • •
Arg 8
Arg 7

- Only allocate stack space when needed

X64 Argument Passing

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

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx       # Save dest
  400544: callq  400550 <mult2>  # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)     # Save at dest
  • • •
```

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

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

Register Saving Convention

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

- Can register be used for temporary storage?

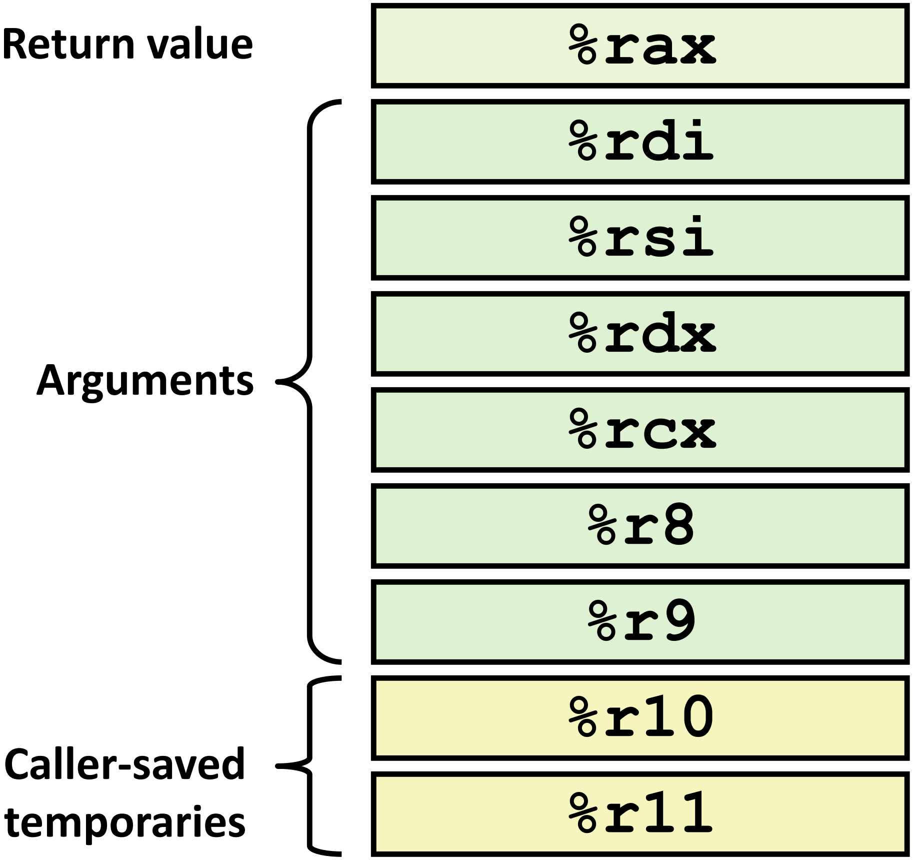
```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

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

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

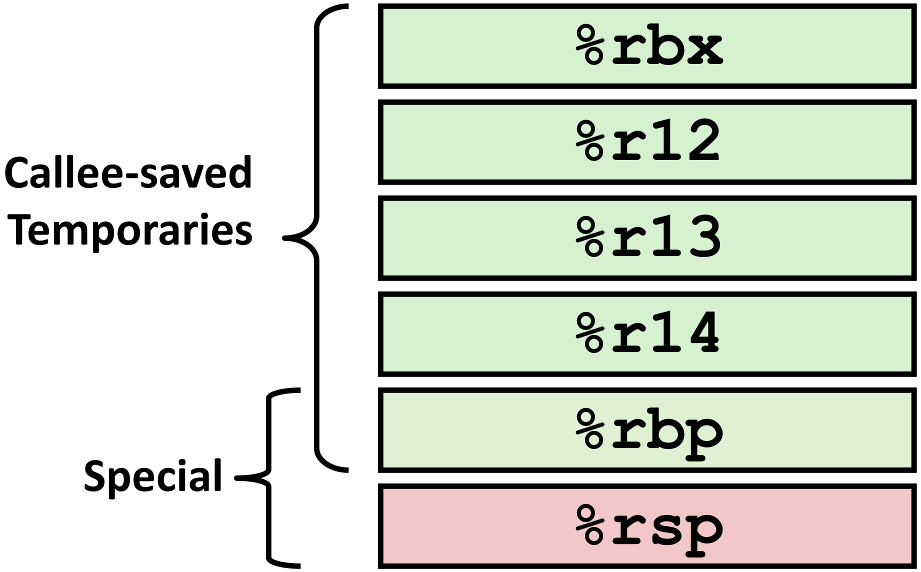
X64 Linux Register Saving

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



X64 Linux Register Saving

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

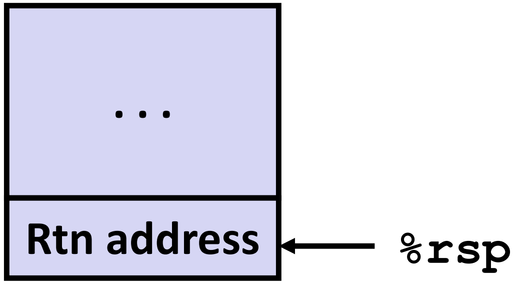


Register Saving Example

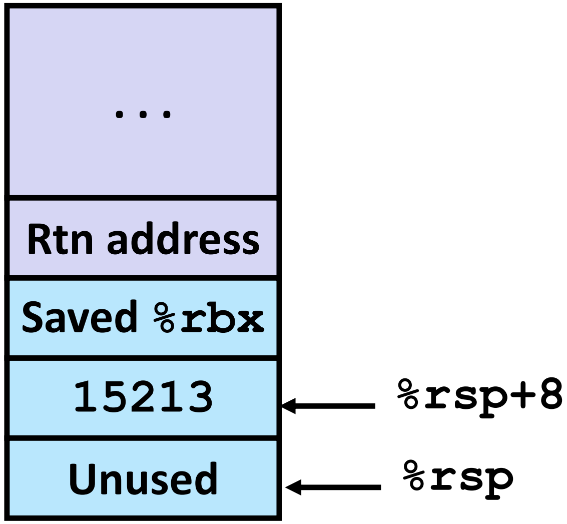
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure

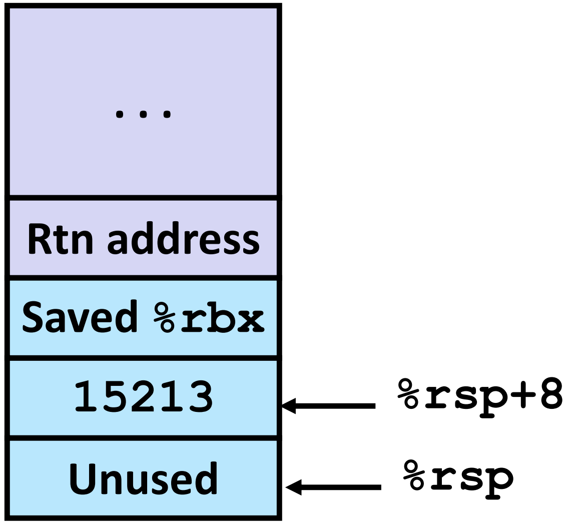


Register Saving Example

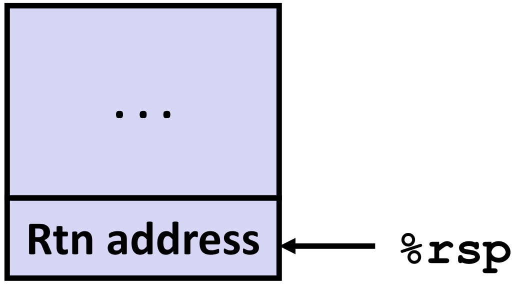
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

Resulting Stack Structure



Pre-return Stack Structure



Why the Stack?

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

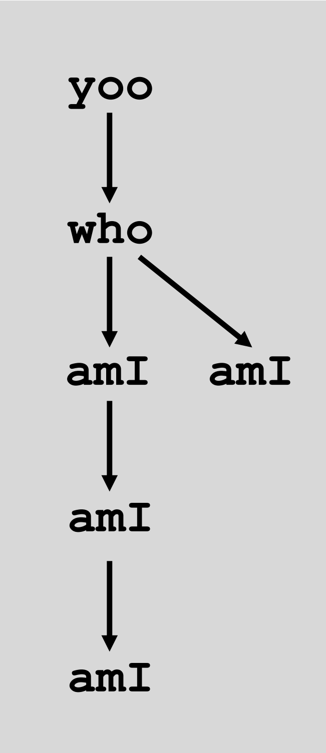
Call Chain

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

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

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

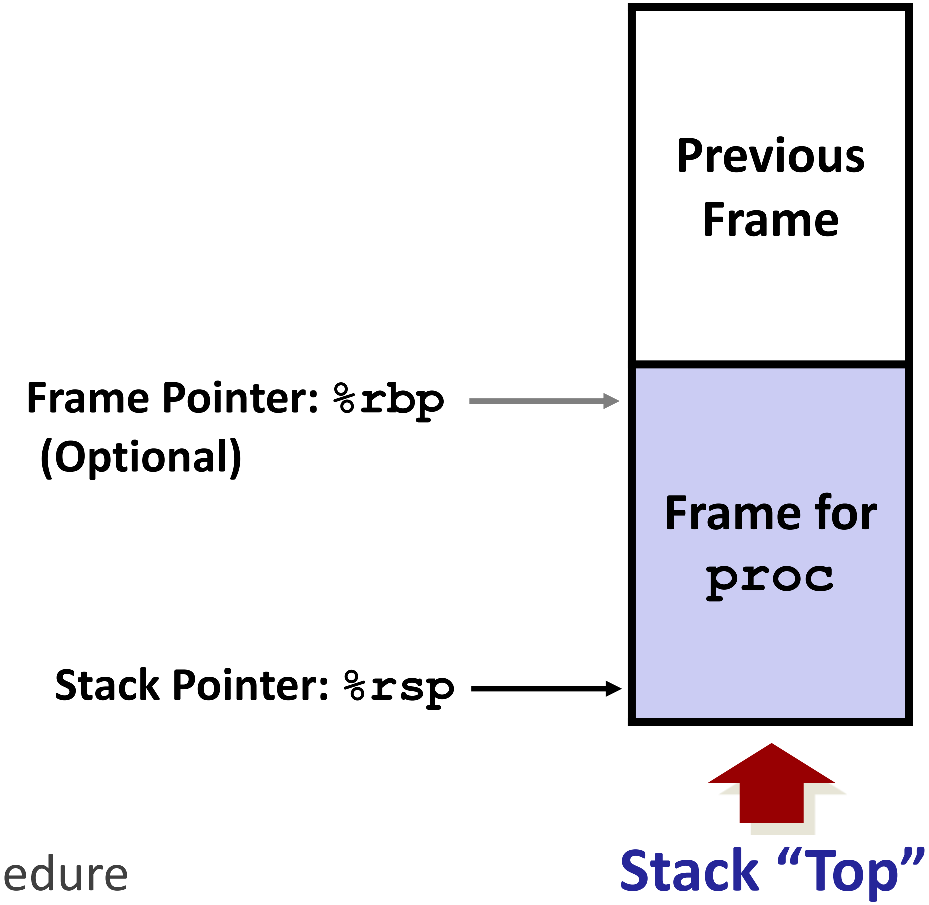
Example
Call Chain



Procedure `amI()` is recursive

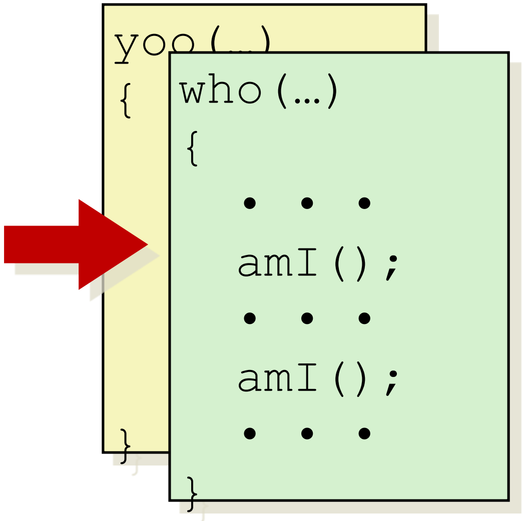
Stack Frame

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

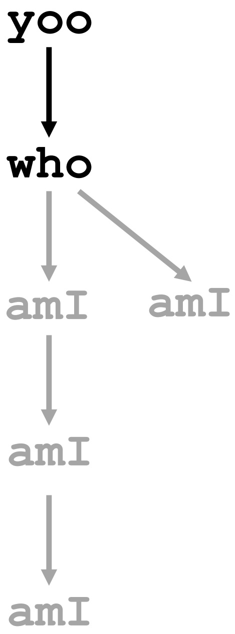


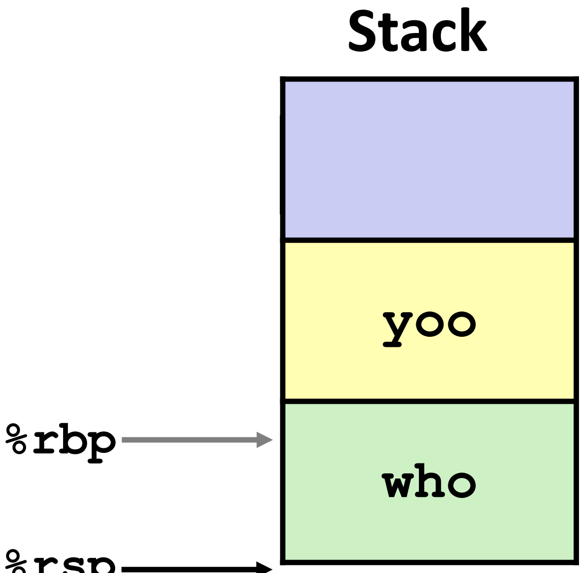
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by `call` instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by `ret` instruction

Call Chain

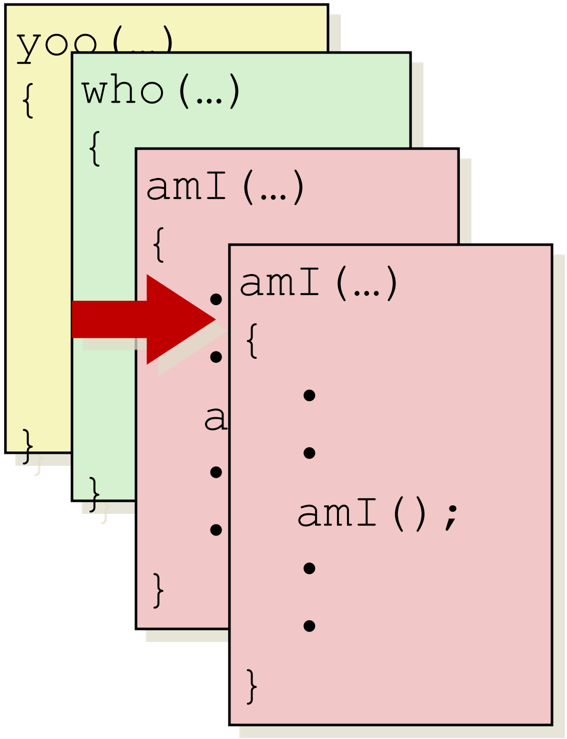


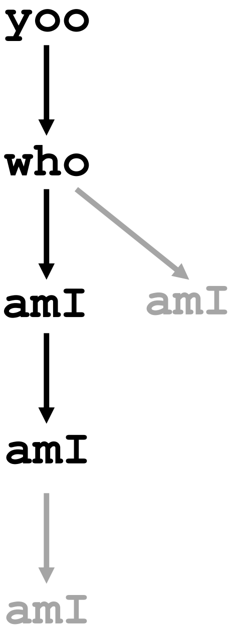
```
yoo(…)
{
  who(…)
  {
    • • •
    amI();
    • • •
    amI();
    • • •
  }
}
```



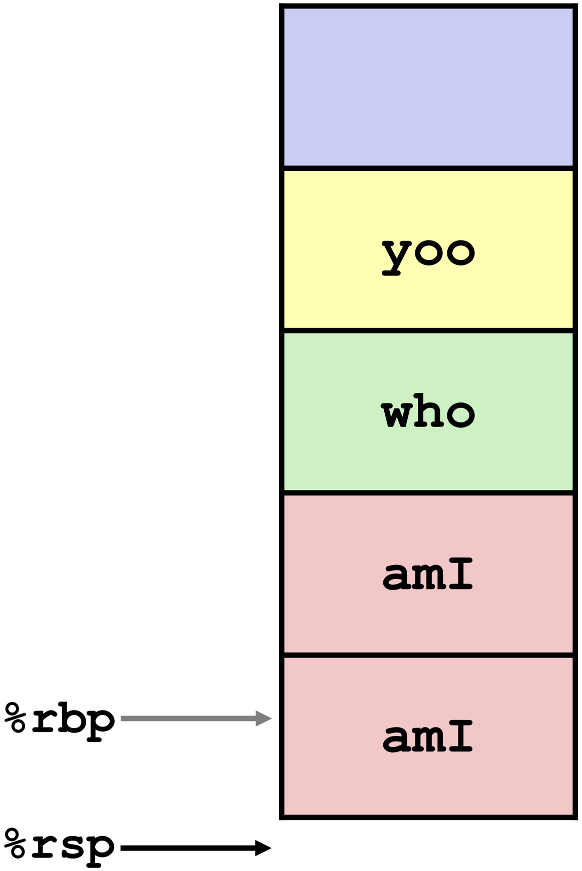


Call Chain





Stack



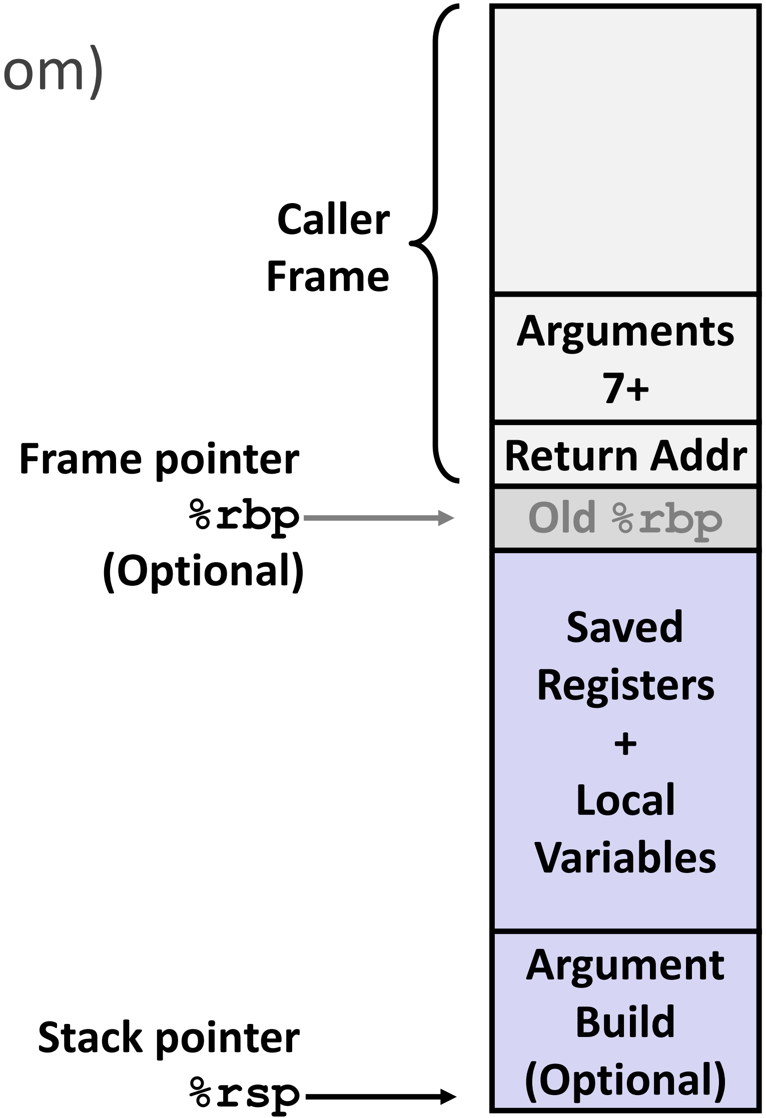
Recursion

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

Linux Stack Frame

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

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


Caller Frame
Arguments 7+
Return Addr
Frame pointer %rbp (Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build (Optional)
Stack pointer %rsp

GNU AS Assembler Directives

- Assembler directives are commands used as part of the assembler syntax.
- However, they are not part of the Intel x86-64 ISA.
  - Think of them as helper commands
- They start with the period character
- Some important Assembler directives are:
  - **`.text`**: Assemble the following statements onto the text section
  - **`.data`**: Assemble the following statements onto the data section
  - **`.byte`** *`value`*: Assemble a byte with the specified value
  - **`.quad`** *`value`*: Assemble a quadword with the specified value
  - **`.asciz`** *`”string”`*: Assemble a NULL terminated string
  - **`.global`** *`symbol`*: Makes the symbol visible to the linker (Global Symbol)
  - **`.rept`** *`count`*`/`**`.endr`**: Repeat the sequence of lines between the **.rept** directive and the next **.endr** directive count times
- More info: ftp://ftp.gnu.org/old-gnu/Manuals/gas/html_chapter/as_7.html

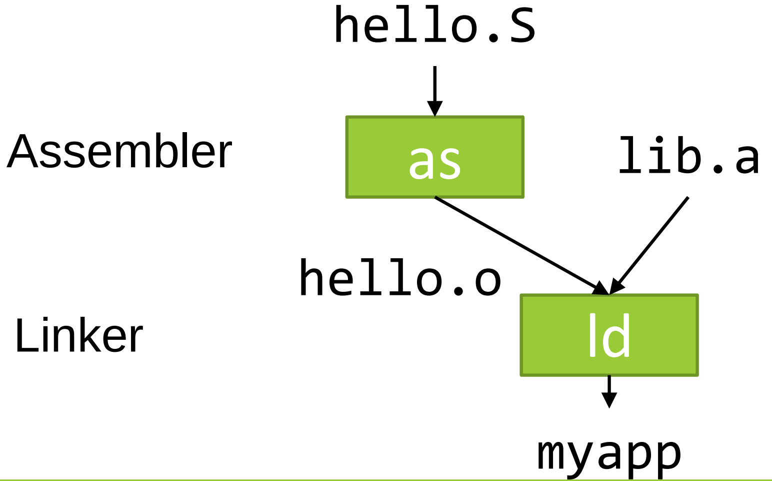
Assembly + libC “Hello World’

hello.S

```
    .global main

    .text
main:
    mov  $message, %rdi     # First parameter in %rdi
    call    puts            # puts(message)
    ret                     # Return to C library code

message:
    .asciz "Hello, world"   # asciz puts a 0 byte at the end
```



```
as --64 hello.S -o hello.o
gcc -o myapp hello.o
```

Summary

- Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P: **Last-In First-Out (LIFO)**
- PUSH and POP instructions are used to control the stack
- CALL and RET are used to implement procedure calls
- Calling conventions are specifications about how a particular language and platform implement procedure calls
  - Argument Passing, Registers Saving Conventions, Stack Frame
- Recursion does not require any special handling