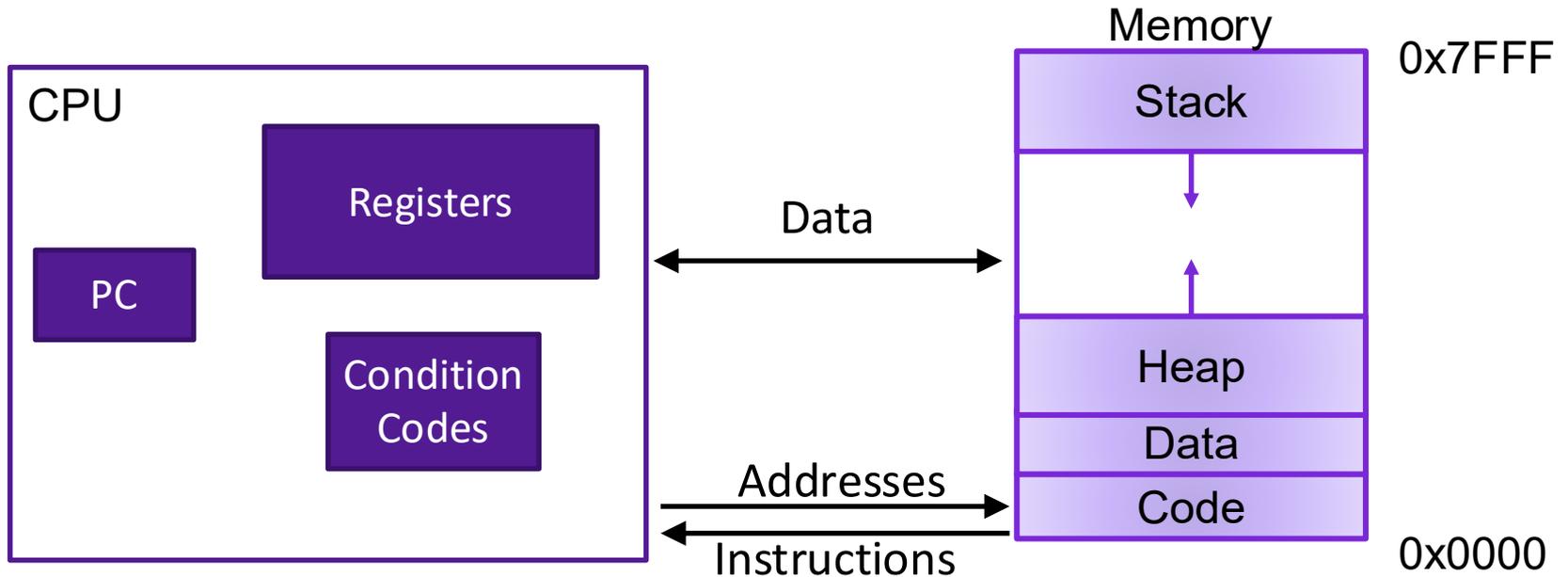


Lecture 7: Procedure Calls in Assembly

CS 105

Spring 2026

Review: Assembly/Machine Code View



Programmer-Visible State

- ▶ PC: Program counter
- ▶ 16 Registers
- ▶ Condition codes

Memory

- ▶ Byte addressable array
- ▶ Code and user data
- ▶ Stack to support procedures

Review: X86-64 Integer Registers

<code>%rax</code> (function result)	<code>%r8</code> (fifth argument)
<code>%rbx</code>	<code>%r9</code> (sixth argument)
<code>%rcx</code> (fourth argument)	<code>%r10</code>
<code>%rdx</code> (third argument)	<code>%r11</code>
<code>%rsi</code> (second argument)	<code>%r12</code>
<code>%rdi</code> (first argument)	<code>%r13</code>
<code>%rsp</code> (stack pointer)	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

Review: Assembly Operations

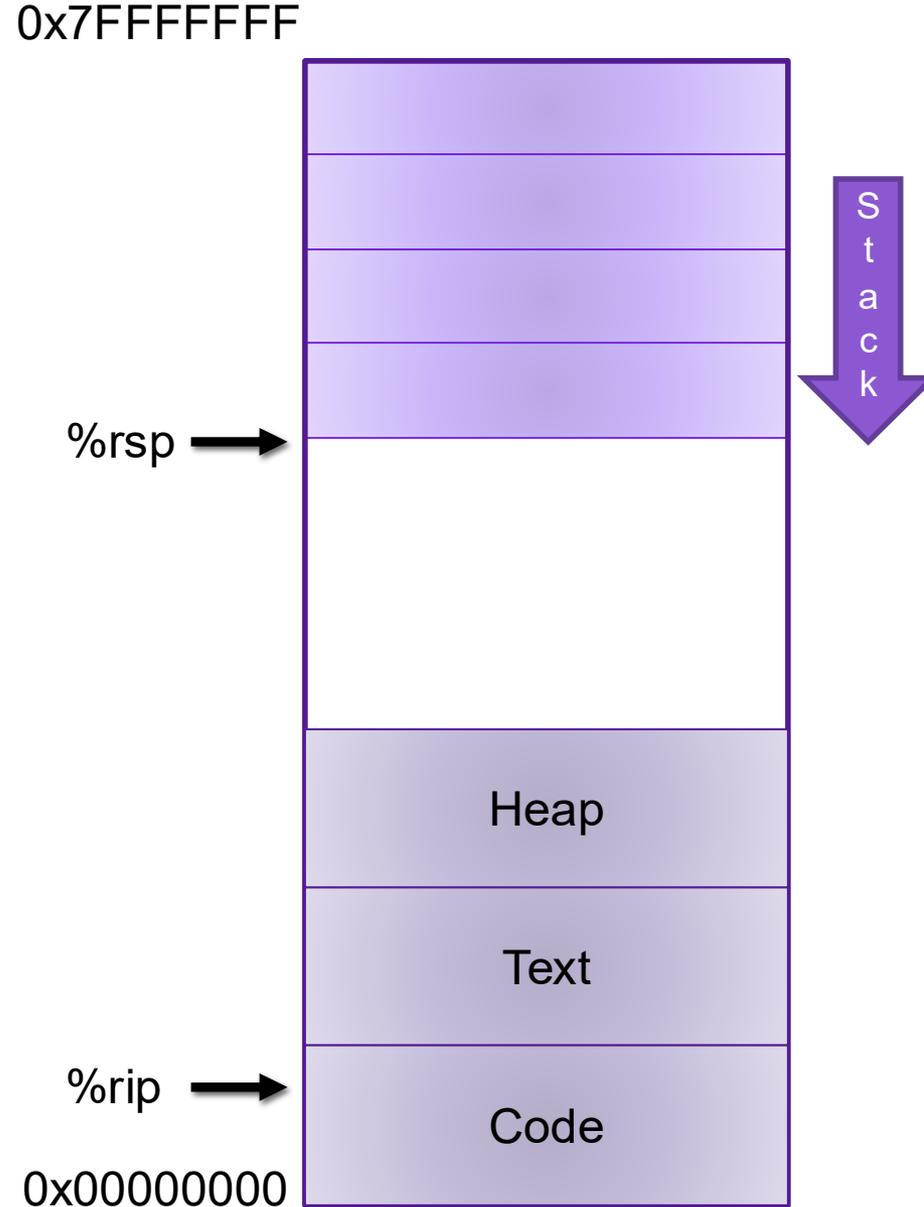
- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
 - Conditional branches
 - Jumps to/from procedures

Procedures

- Procedures provide an abstraction that implements some functionality with designated arguments and (optional) return value
 - e.g., functions, methods, event handlers
- To support procedures at the machine level, we need mechanisms for:
 - 1) **Passing Control:** When procedure P calls procedure Q, program counter must be set to address of Q, when Q returns, program counter must be reset to instruction in P following procedure call
 - 2) **Passing Data:** Must handle parameters and return values
 - 3) **Allocating memory:** Q must be able to allocate (and deallocate) space for local variables

The Stack

- the stack is a region of memory (traditionally the "top" of memory)
- grows "down"
- provides storage for functions (i.e., space for allocating local variables)
- `%rsp` holds address of top element of stack



Modifying the Stack ^{0x7FFFFFFF}

- `pushq S:`

$R[\%rsp] \leftarrow R[\%rsp] - 8$

$M[R[\%rsp]] \leftarrow S$

- `popq D:`

$D \leftarrow M[R[\%rsp]]$

$R[\%rsp] \leftarrow R[\%rsp] + 8$

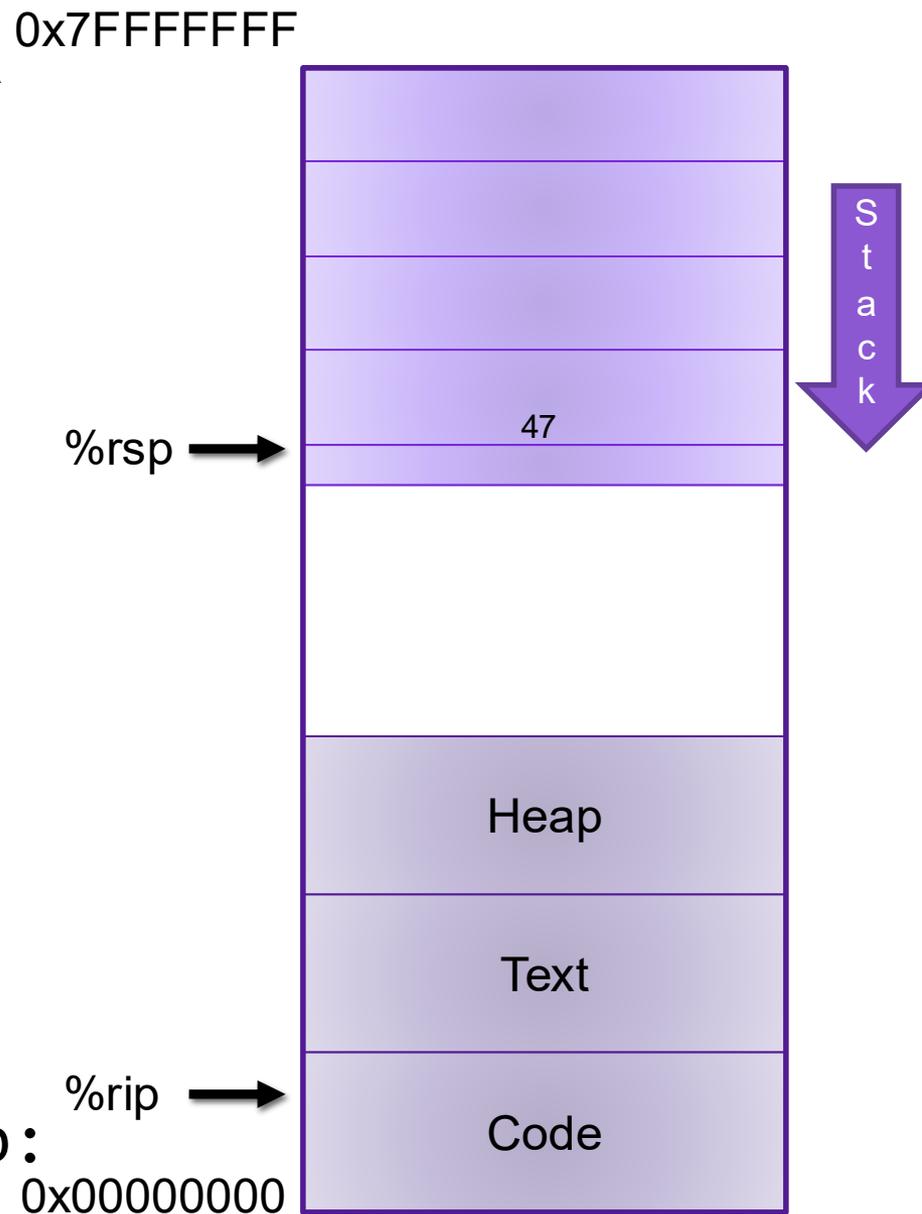
- explicitly modify `%rsp:`

`subq $4, %rsp`

`addq $4, %rsp`

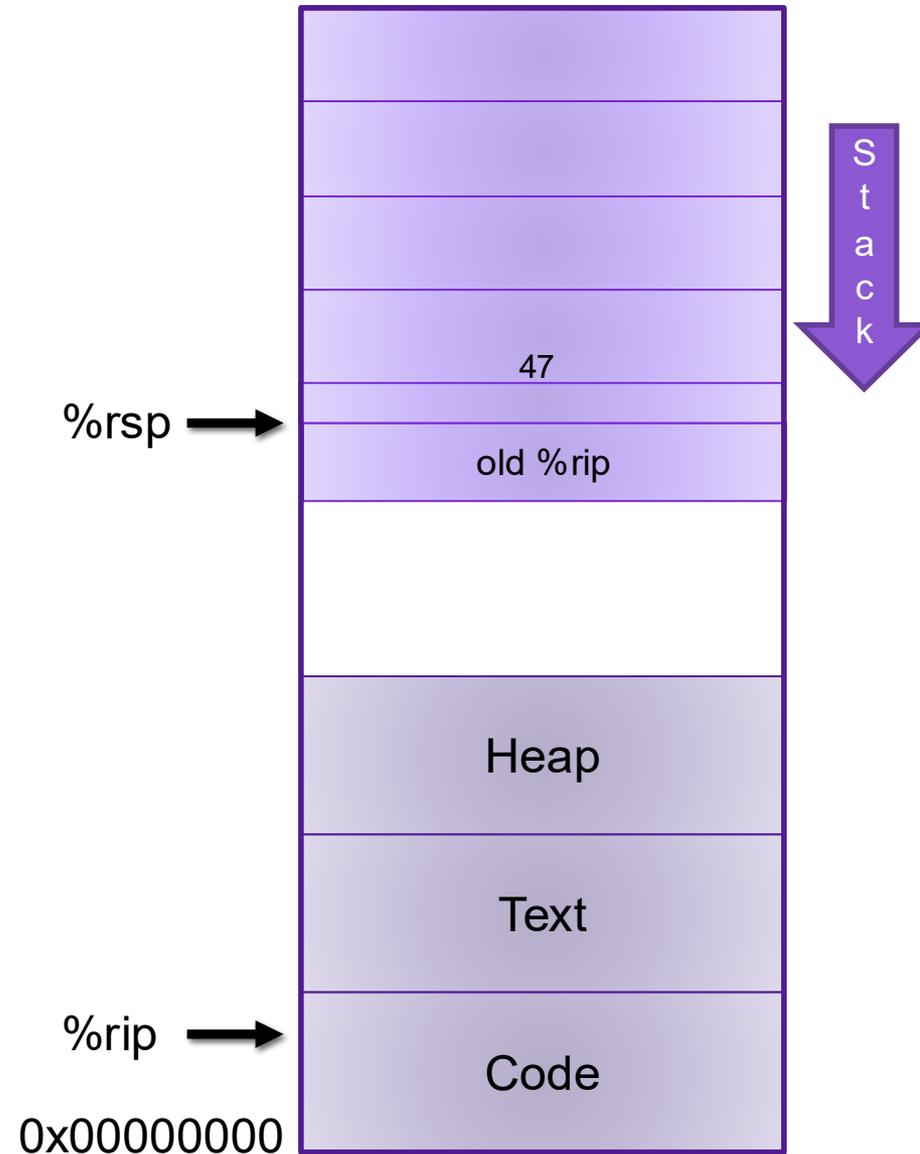
- modify memory above `%rsp:`

`movl $47, 4(%rsp)`



Modifying the Stack ^{0x7FFFFFFF}

- `call f:`
 `pushq %rip`
 `movq &f, %rip`
- `ret:`
 `popq %rip`



Example: Modifying the Stack

```
int proc(int* p){
    return p[3];
}

int example1(int x) {
    int a[4];
    a[0] = 13;
    a[1] = 47;
    a[2] = 105;
    a[3] = 41;
    return proc(a) + 1;
}
```

```
proc:
    movl    12(%rdi), %eax
    ret
```

```
example1:
    subq    $16, %rsp
    movl    $13, (%rsp)
    movl    $47, 4(%rsp)
    movl    $105, 8(%rsp)
    movl    $41, 12(%rsp)
    movq    %rsp, %rdi
    call    0x400596 <proc>
    addl    $1, %rax
    addq    $16, %rsp
    ret
```

Exercise 1: Modifying the Stack

0x400557 <fun>:

400557: movq \$13, 16(%rsp) %rsp →

40055a: ret

0x40055b <main>:

%rip → 40055b: sub \$8, %rsp

40055f: pushq \$47

400560: callq 400557 <fun>

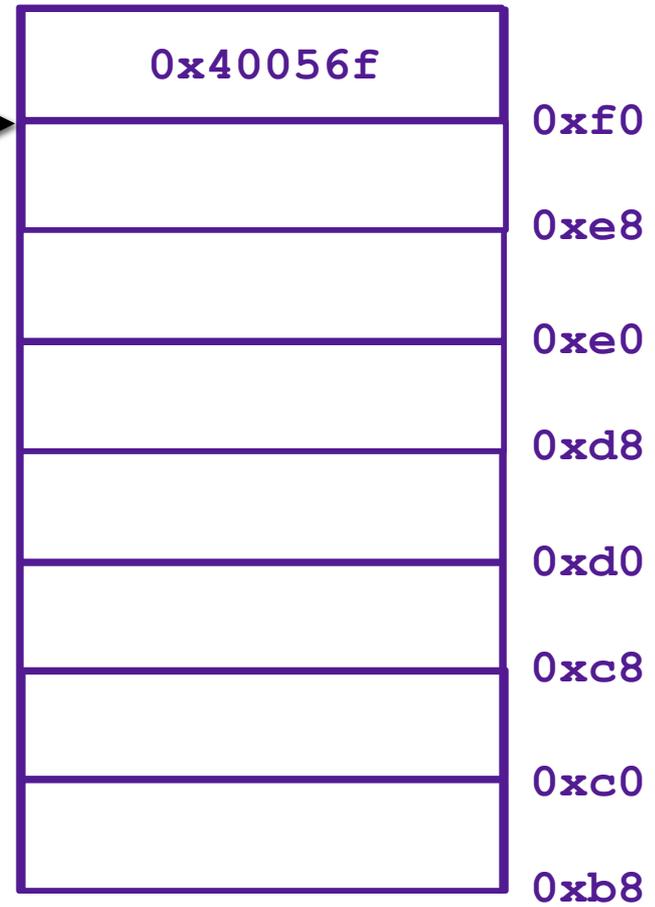
400565: popq %rax

400566: addq (%rsp), %rax

40056a: addq \$8, %rsp

40056e: ret

%rax



What's the value in %rax immediately before the instruction at 0x40056e is executed?
What's the value in %rsp immediately before the instruction at 0x40056e is executed?

Procedure Calls (simplified)

Caller

- Before
 - Put arguments in place (if there are parameter vars)
 - Make call
- After
 - Use result (if non-void)

Callee

- Preamble
 - Allocate space on stack (if needed)
- Exit code
 - Put return value in place (if non-void function)
 - Deallocate space on stack (if allocated)
 - Return

Example: Modifying the Stack

```
int example1(int x) {  
    int a[4];  
    a[0] = 13;  
    a[1] = 47;  
    a[2] = 105;  
    a[3] = 41;  
    return proc(a) + 1;  
}
```

example1:

subq \$16, %rsp	allocate
movl \$13, (%rsp)	
movl \$47, 4(%rsp)	
movl \$105, 8(%rsp)	
movl \$41, 12(%rsp)	
movq %rsp, %rdi	args
call 0x400596 <proc>	call
addl \$1, %rax	ret. val
addq \$16, %rsp	dealloc.
ret	return

Maintaining Variable state

```
int function() {  
    int x = 47;  
    int y = 13;  
    mystery(y);  
  
    // what is x?  
    // what is y?  
}
```

```
function:  
    movl    $47, %rbx  
    movl    $13, %rdi  
    call   0x40042a <mystery>  
  
# what is in %rbx?  
# what is in %rdi?  
ret
```

X86-64 Register Usage Conventions

<code>%rax</code> (function result)	<code>%r8</code> (fifth argument)
<code>%rbx</code>	<code>%r9</code> (sixth argument)
<code>%rcx</code> (fourth argument)	<code>%r10</code>
<code>%rdx</code> (third argument)	<code>%r11</code>
<code>%rsi</code> (second argument)	<code>%r12</code>
<code>%rdi</code> (first argument)	<code>%r13</code>
<code>%rsp</code> (stack pointer)	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

Callee-saved registers are shaded

Procedure Calls, Division of Labor

Caller

- Before
 - Save caller-saved registers to stack (if used after call)
 - Put arguments in place (if there are parameters)
 - Make call
- After
 - Restore caller-saved register (if used after call)
 - Use result (if non-void)

Callee

- Preamble
 - Save callee-saved registers (if will use)
 - Allocate space on stack (if needed)
- Exit code
 - Put return value in place (if non-void function)
 - Restore callee-saved registers (if used)
 - Deallocate space on stack (if allocated)
 - Return

Exercise 2: Value Passing

0x400540 <last>:

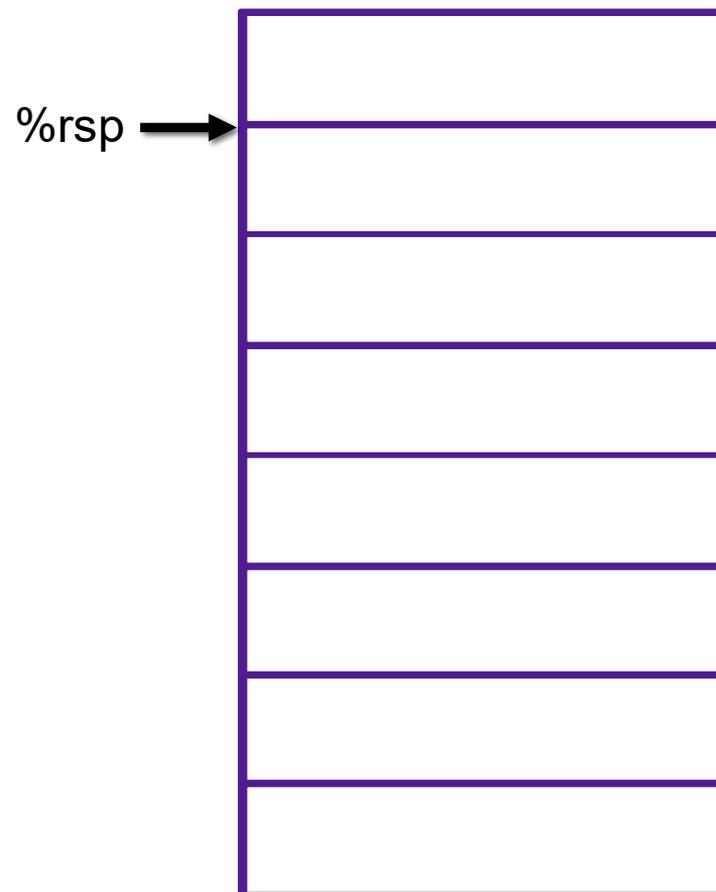
```
400540: mov %rdi, %rax
400543: imul %rsi, %rax
400547: ret
```

0x400548 <first>:

```
400548: lea 0x1(%rdi),%rsi
40054c: sub $0x1, %rdi
400550: callq 400540 <last>
400555: rep; ret
```

0x400556 <main>:

```
400560: mov $4, %rdi
400563: callq 400548 <first>
400568: addq $0x13, %rax
40056c: ret
```



%rdi

%rsi

%rax

%rip

What value gets returned by main?

Handling Extra Parameters

- Conventions define 6 registers for storing arguments
- If function has more than 6 parameters, additional arguments go on the stack

Procedure Call Example: Arguments

```
int func1(int x1, int x2, int x3,
          int x4, int x5, int x6,
          int x7, int x8){
    int y1 = x1+x2;
    int y2 = x3+x4;
    int y3 = x5+x6;
    int y4 = x7+x8;
    int y5 = 4;
    int y6 = 13;
    int y7 = 47;
    int y8 = y1 + y2 + y3 + y4 + y5
            + y6 + y7;
    return y8;
}

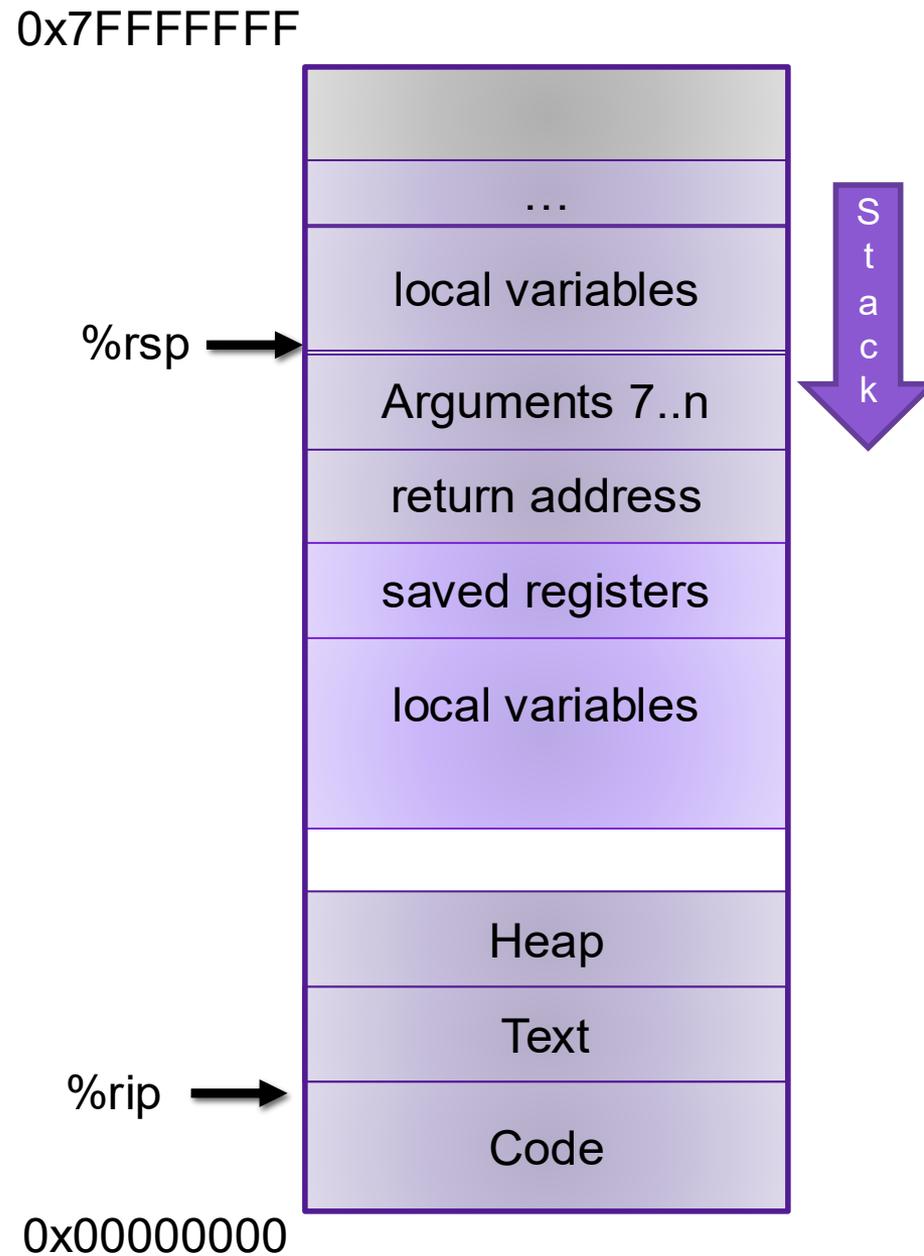
int main(int argc, char *argv[]){
    int x = func1(1,2,3,4,5,6,7,8);
    return x;
}
```

```
func1:
    addl    %edi, %esi
    addl    %ecx, %edx
    addl    %r9d, %r8d
    movl    16(%rsp), %eax
    addl    8(%rsp), %eax

main:
    movl    $1, %edi
    movl    $2, %esi
    movl    $3, %edx
    movl    $4, %ecx
    movl    $5, %r8d
    movl    $6, %r9d
    pushq   $8
    pushq   $7
    callq   _function1
    addq    $16, %rsp
    retq
```

Stack Frames

- Each function called gets a stack frame
- Passing data:
 - calling procedure P uses registers (and stack) to provide parameters to Q.
 - Q uses register `%rax` for return value
- Passing control:
 - **call <proc>**
 - Pushes return address (current `%rip`) onto stack
 - Sets `%rip` to first instruction of proc
 - **ret**
 - Pops return address from stack and places it in `%rip`
- Local storage:
 - allocate space on the stack by decrementing stack pointer, deallocate by incrementing



Recursion

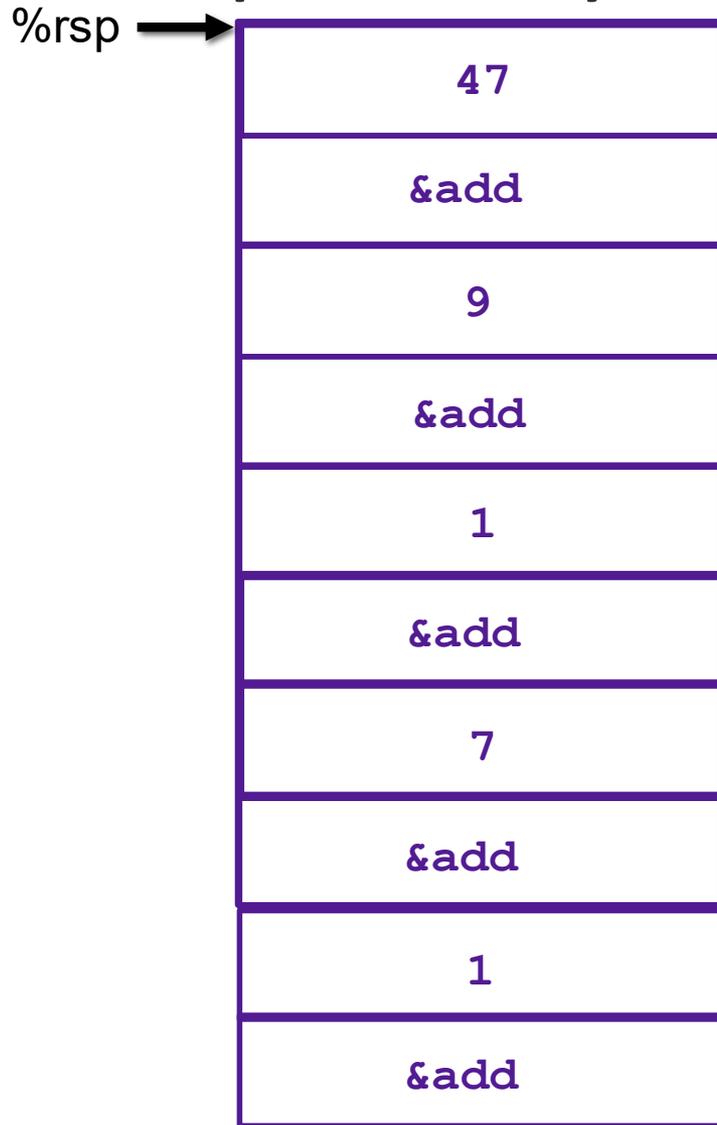
- Handled Without Special Consideration
 - Stack frames mean that each function call has private storage
 - Saved registers & local variables
 - Saved return pointer
 - Register saving conventions prevent one function call from corrupting another's data
 - Unless the C code explicitly does so (more later!)
 - 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

Array Recursion

```
int sum_digits_r(int* z, int i){  
  
    if(i >= 5){  
        return 0;  
    }  
  
    int val = z[i];  
  
    int sum_r = sum_digits_r(z,i+1);  
  
    return sum + val;  
}
```

```
sum_digits_r:  
    cmp     $4, %rsi  
    jle    L2  
    mov    $0, %rax  
    ret  
L2:  
    push   %rbx  
    mov    (%rdi,%rsi,4), %ebx  
    incr   $1, %rsi  
    call   sum_digits_r  
    add    %ebx, %eax  
    pop    %rbx  
    ret
```

Example: Array Recursion



```
sum_digits_r:
    cmp     $4, %rsi
    jle    L2
    mov     $0, %rax
    ret
L2:
    push   %rbx
    mov    (%rdi,%rsi,4), %ebx
    incr  $1, %rsi
    call  sum_digits_r
    add   %ebx, %eax
    pop   %rbx
    ret
```

