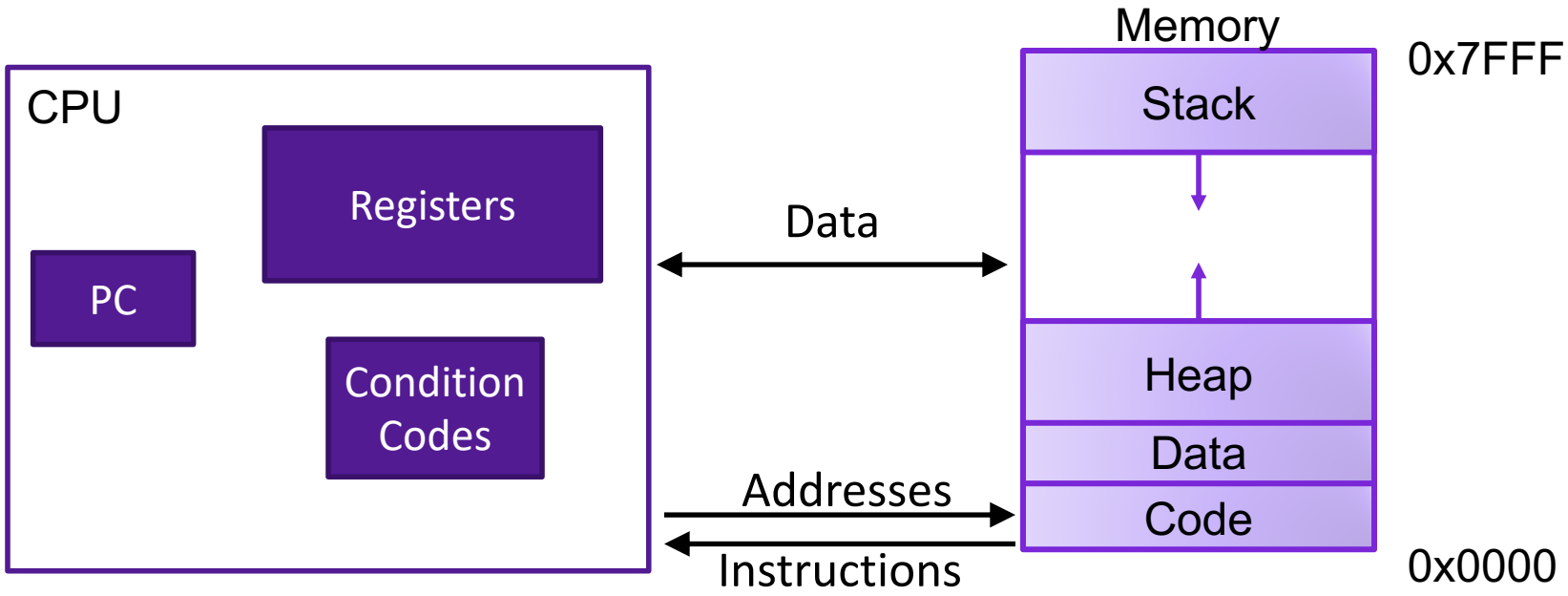
Lecture 7: Procedure Calls in Assembly

CS 105

Fall 2020

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

Assembly Characteristics: 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

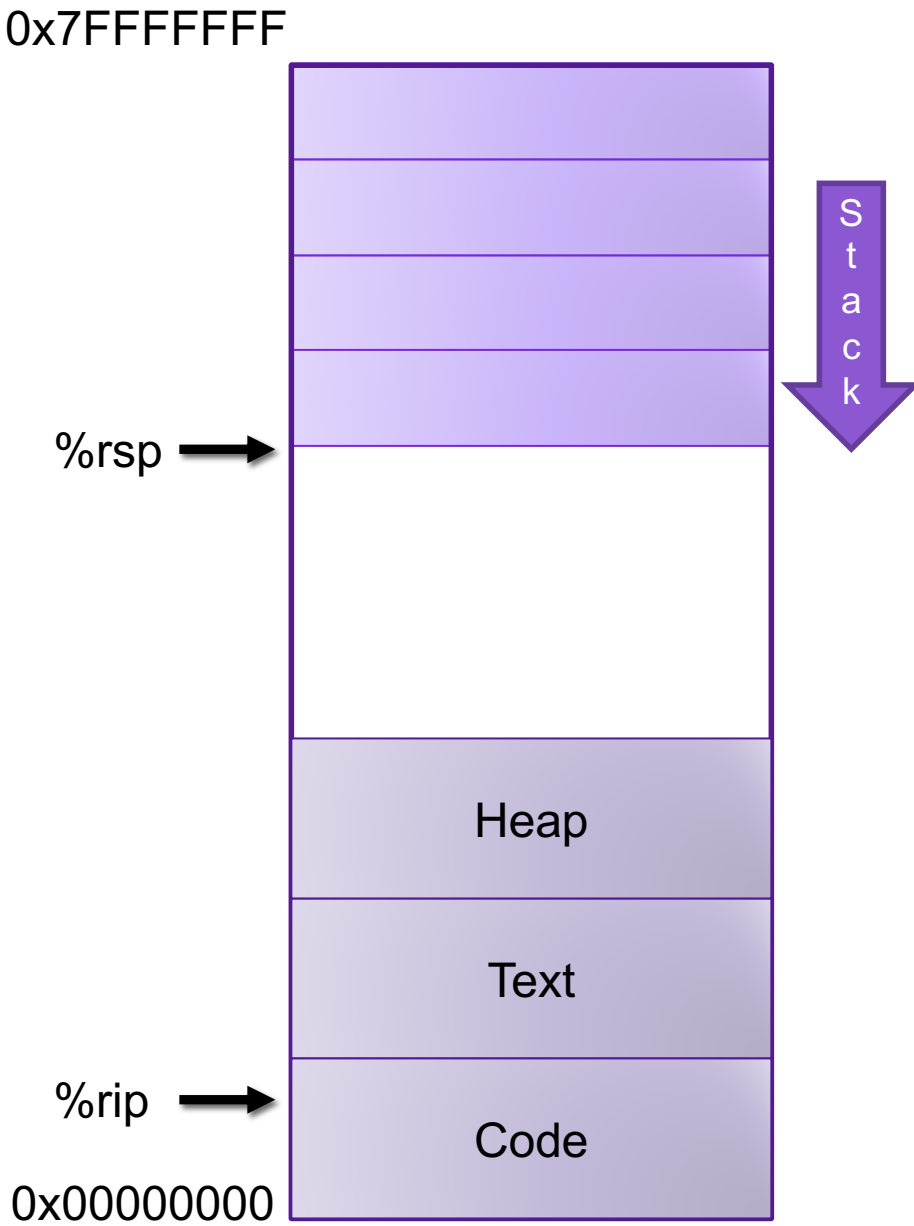
Procedures

- Procedures provide an abstraction that implements some functionality with designated arguments and (optional) return value
  - e.g., functions, methods, subroutines, 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

- `pushq S`:

  `R[%rsp] ← R[%rsp] - 8`

  `M[R[%rsp]] ← S`

- `popq D`:

  `D ← M[R[%rsp]]`

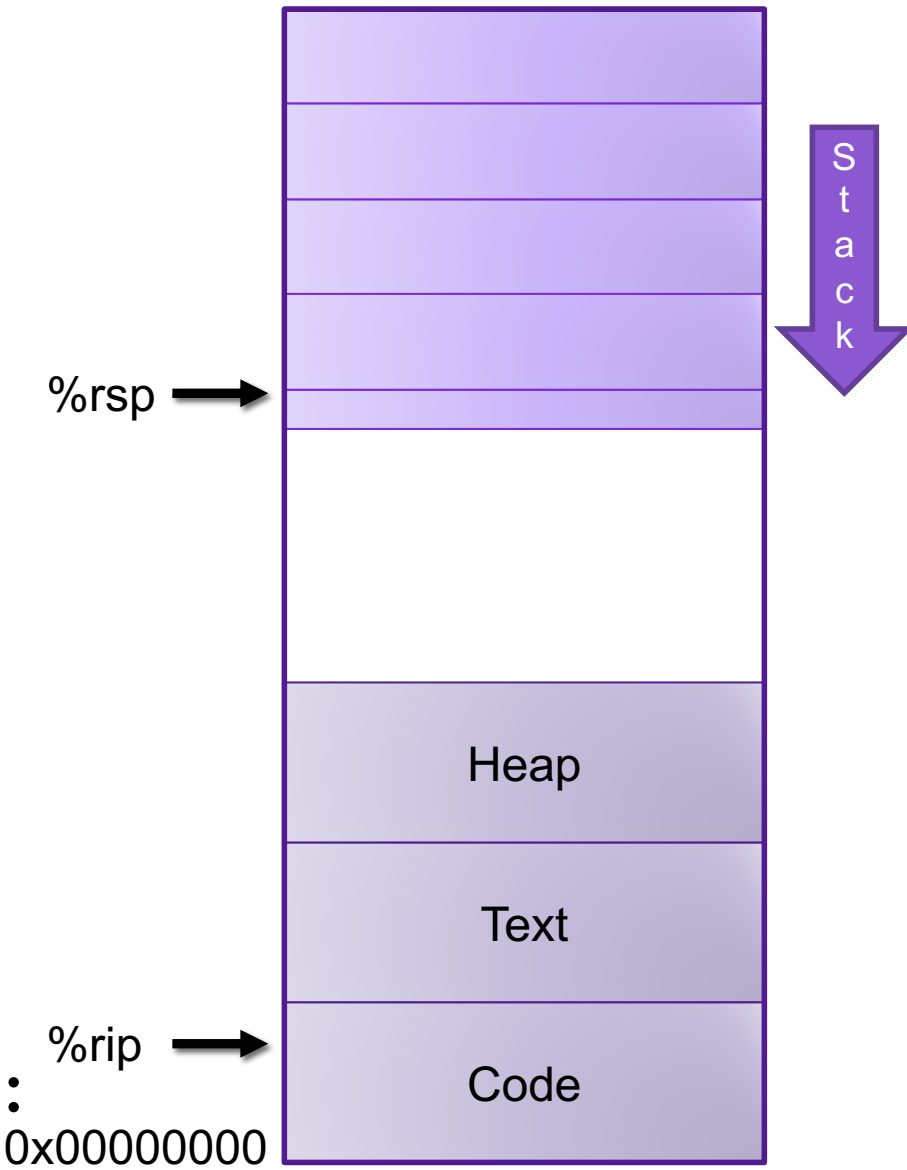
  `R[%rsp] ← R[%rsp] + 8`

- explicitly modify %rsp:

```
subq $4, %rsp
addq $4, %rsp
```

- modify memory above `%rsp`:

```
movl $47, 4(%rsp)
```



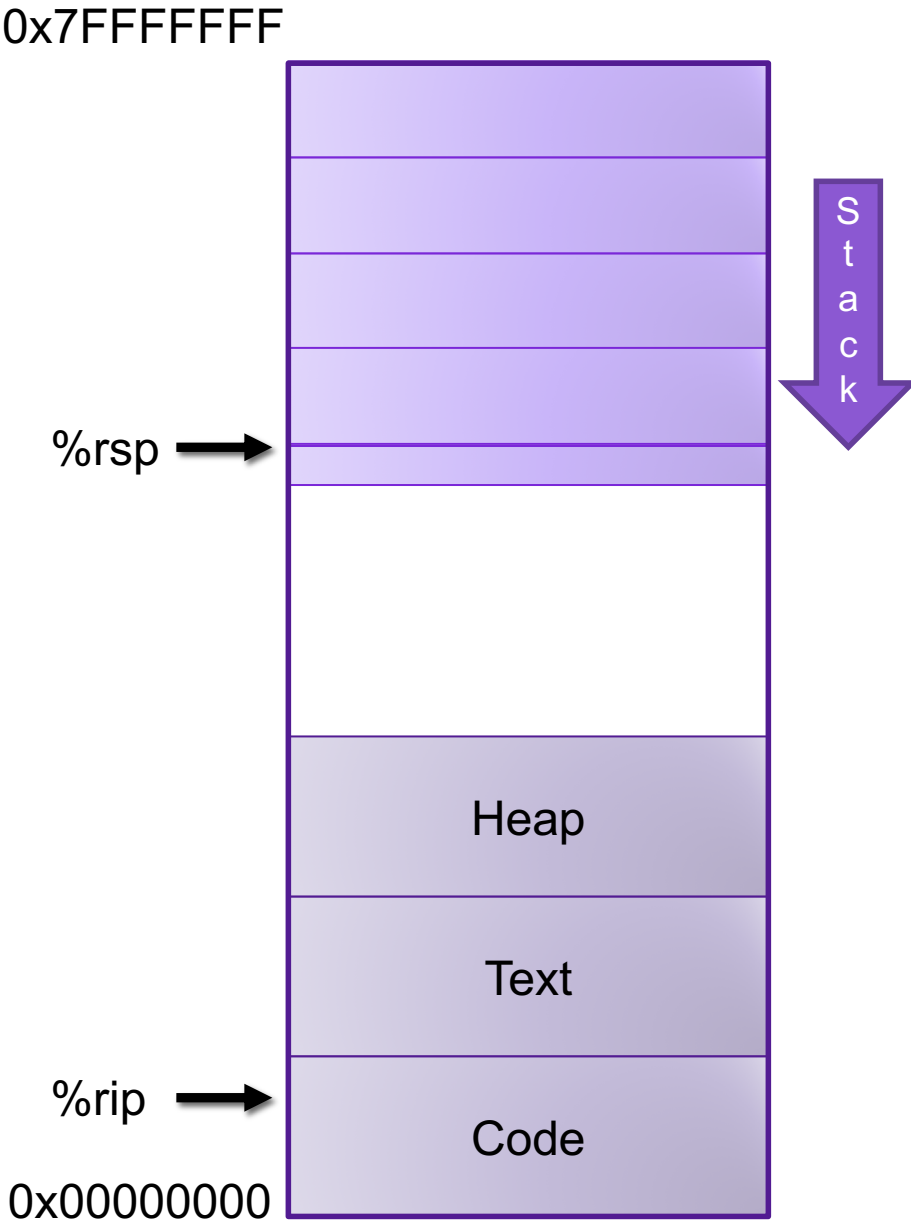
Modifying the Stack

- `call f:`

  ```
  pushq %rip
  movq &f, %rip
  ```

- `ret:`

  ```
  popq %rip
  ```



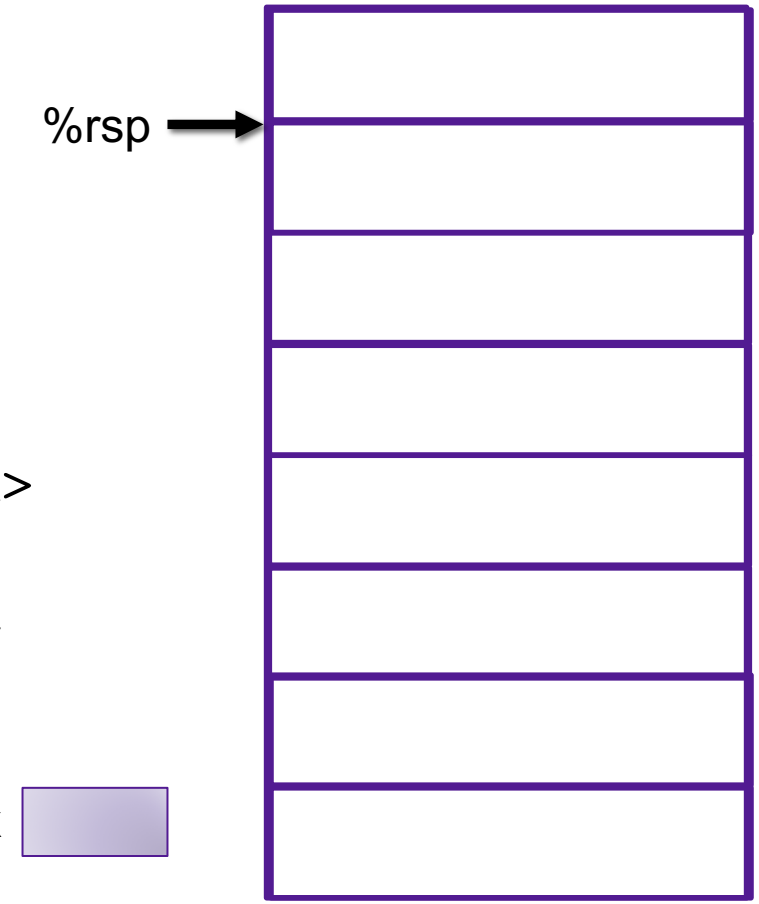
Procedure Call Example: Stack Frame

```
int proc(int *p);

int example1(int x) {
  int a[4];
  a[3] = 10;
  return proc(a);
}
```

```
example1:
  subq   $16, %rsp
  movl   $10, 12(%rsp)
  movq   %rsp, %rdi
  call   0x400546 <proc>
  addq   $16, %rsp
  ret
```

Exercise 1: Modifying the Stack

```
   0x400557 <fun>:
     400557: mov $13, 16(%rsp)
     40055a: ret

   0x40055b <main>:
%rip ➔ 40055b: sub $8, %rsp
     40055f: push $47
     400560: callq 400557 <fun>
     400565: popq %rax
     400566: addq (%rsp), %rax
     40056a: addq $8, %rsp
     40056e: ret
```



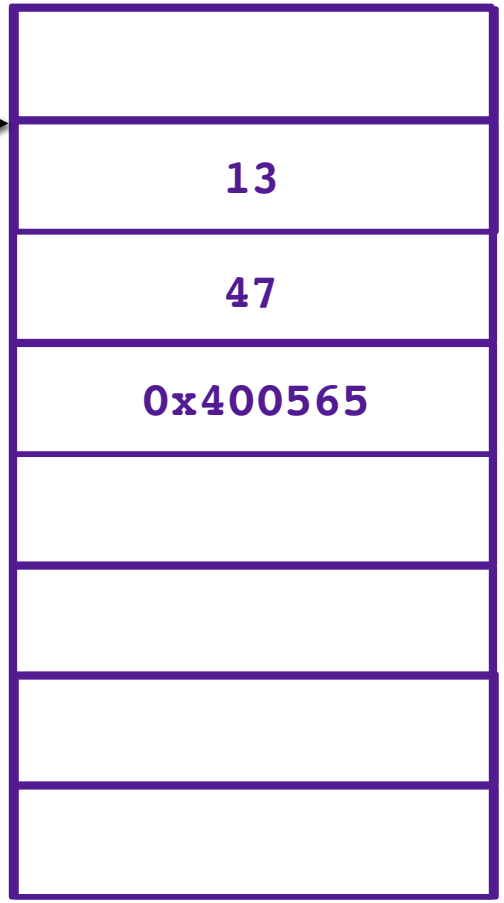
What is the value in %rax immediately before main returns?
What is the value in %rsp immediately before main returns?

Exercise 1: Modifying the Stack

```
0x400557 <fun>:
  400557: mov $13, 16(%rsp)
  40055a: ret

0x40055b <main>:
%rip → 40055b: sub $8, %rsp
  40055f: push $47
  400560: callq 400557 <fun>
  400565: popq %rax
  400566: addq (%rsp), %rax
  40056a: addq $8, %rsp
  40056e: ret
```

%rax **60**

%rsp →



What is the value in %rax immediately before main returns?
What is the value in %rsp immediately before main returns?

X86-64 Register Usage Conventions

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

Callee-saved registers are shaded

Procedure Calls, Division of Labor

Caller

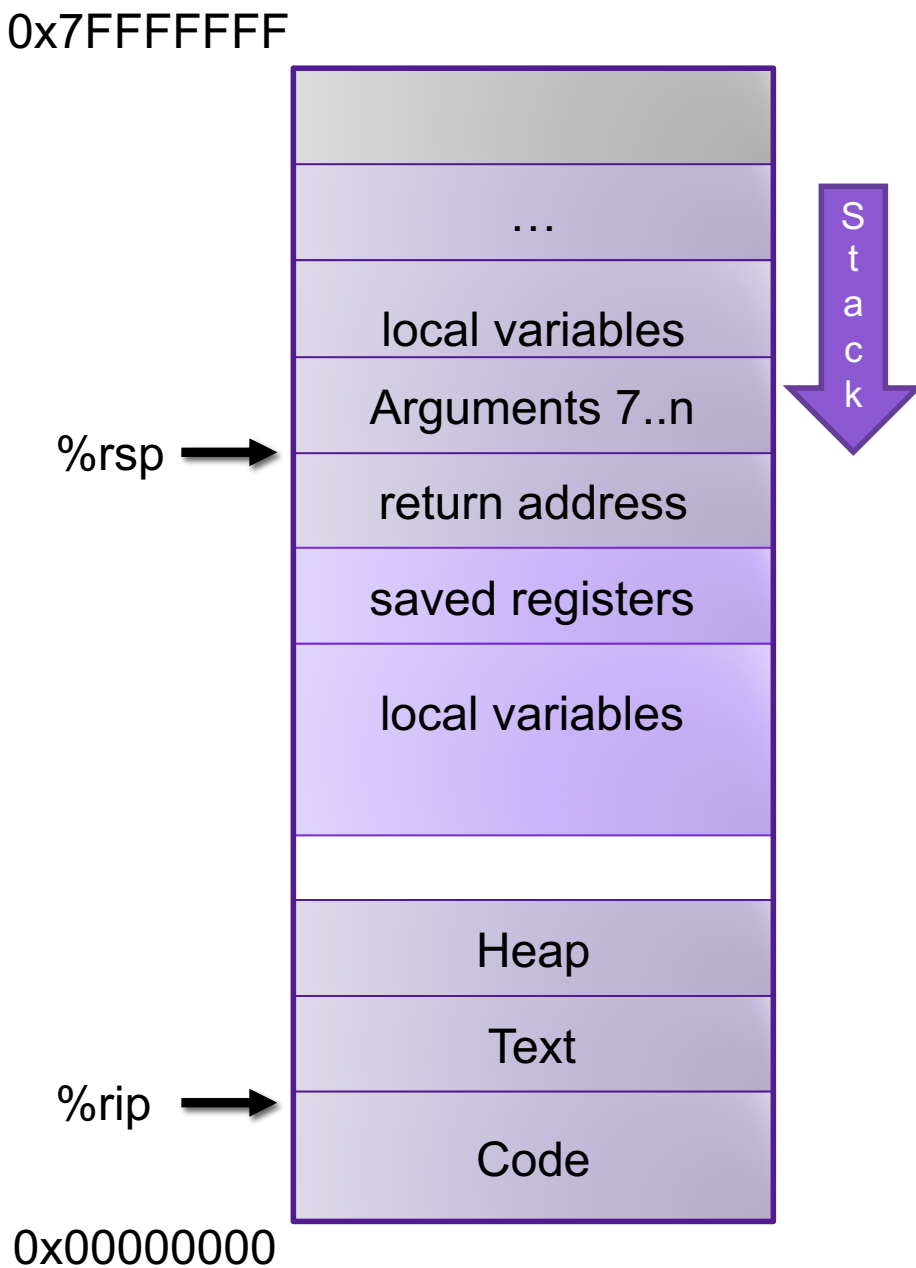
- Before
  - Save registers, if necessary
  - Put arguments in place
  - Make call
- After
  - Restore registers, if necessary
  - Use result

Callee

- Preamble
  - Save registers, if necessary
  - Allocate space on stack
- Exit code
  - Put return value in place
  - Restore registers, if necessary
  - Deallocate space on stack
  - Return

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



Procedure Call Example: Arguments

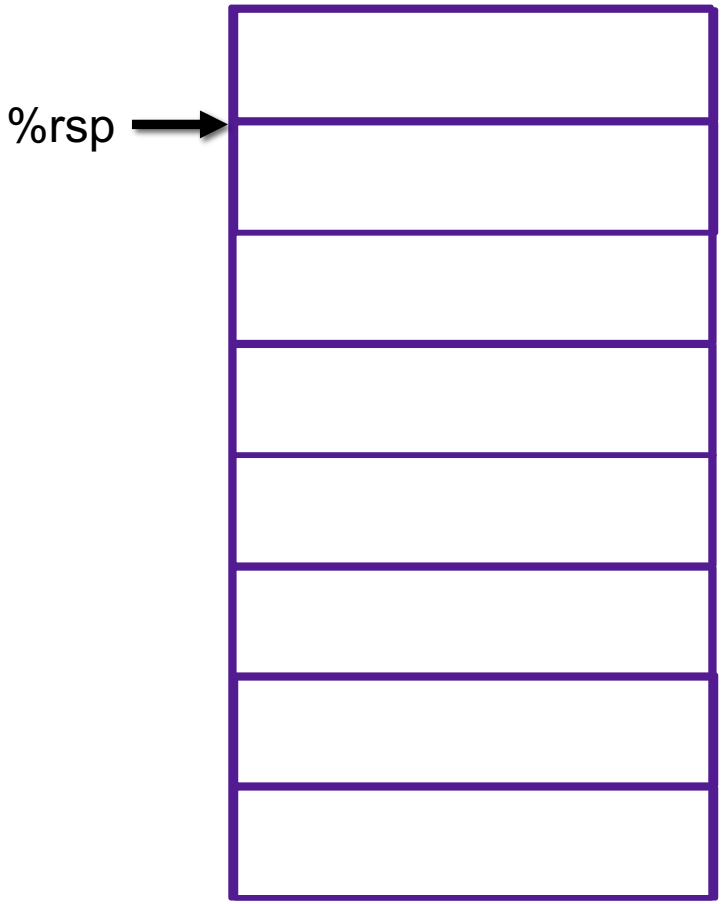
```
int func1(int x1, int x2, int x3,
          int x4, int x5, int x6,
          int x7, int x8){
  int l1 = x1+x2;
  int l2 = x3+x4;
  int l3 = x5+x6;
  int l4 = x7+x8;
  int l5 = 4;
  int l6 = 13;
  int l7 = 47;
  int l8 = l1 + l2 + l3 + l4 + l5
             + l6 + l7;
  return l8;
}

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
```

Exercise 2: Value Passing

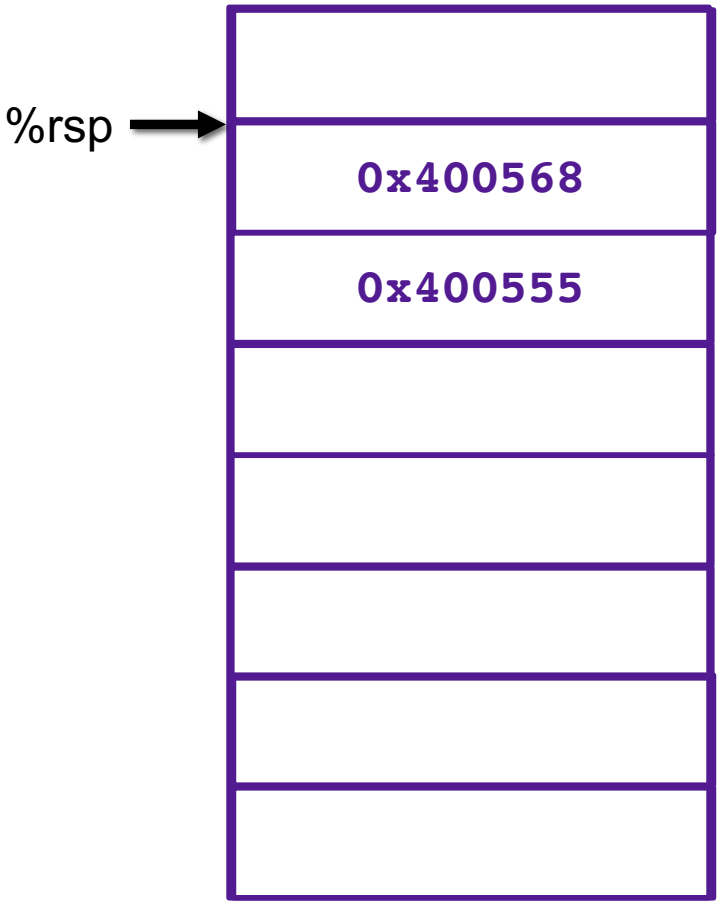
```
0x400540 <last>:
  400540: mov %rdi, %rax
  400543: imul %rsi, %rax
  400547: ret

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
```



What value gets returned by main?

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
```



What value gets returned by main?

%rdi	%rsi	%rax	%rip
3	5	34	0x40056c

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

Recursive Function

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

What is in the stack frame?

```
bitcount_r:
  testq   %rdi, %rdi
  je      .L3
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    bitcount_r
  addq    %rbx, %rax
  popq    %rbx
  ret
.L3: # Base Case
  movl    $0, %eax
  ret
```

Exercise 3: Feedback

1. Rate how well you think this recorded lecture worked
    1. Better than an in-person class
    2. About as well as an in-person class
    3. Less well than an in-person class, but you still learned something
    4. Total waste of time, you didn't learn anything
2. How much time did you spend on this video lecture (including time spent on exercises)?
3. Do you have any questions that you would like me to address in this week's problem session?
4. Do you have any other comments or feedback?