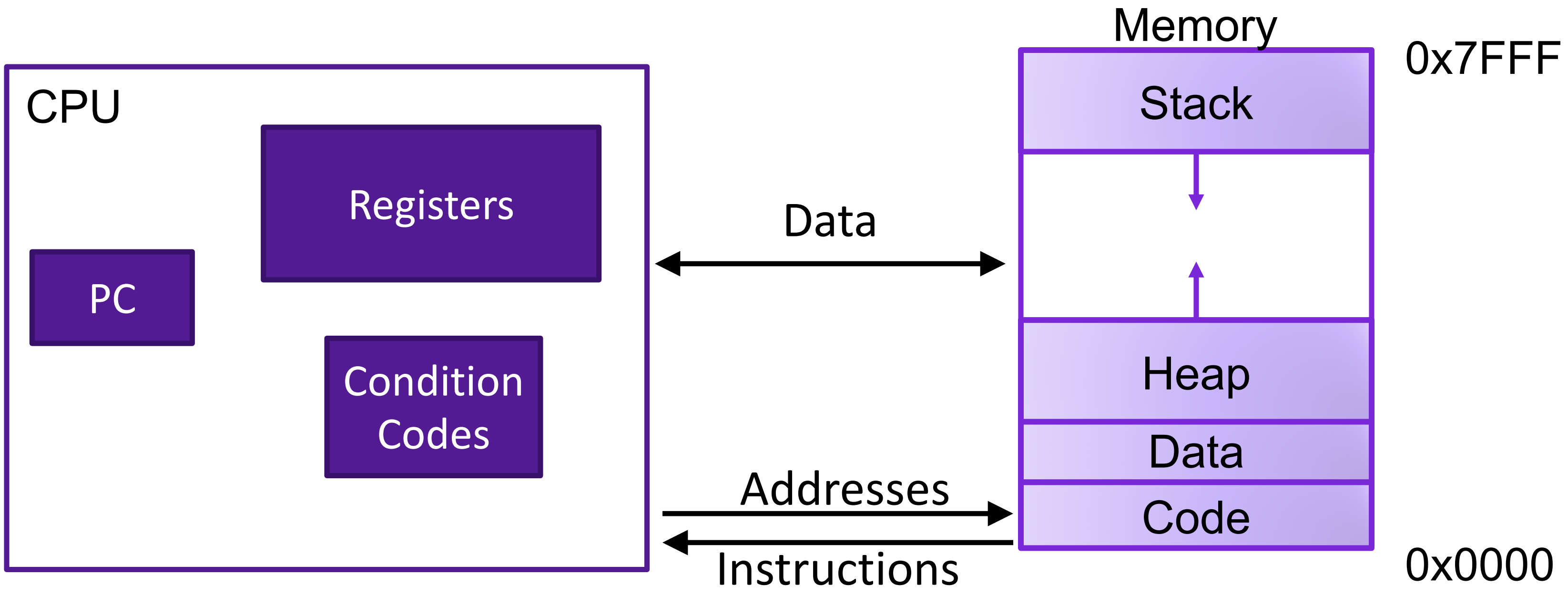
Lecture 6: Procedure Calls in Assembly

CS 105

September 19, 2019

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

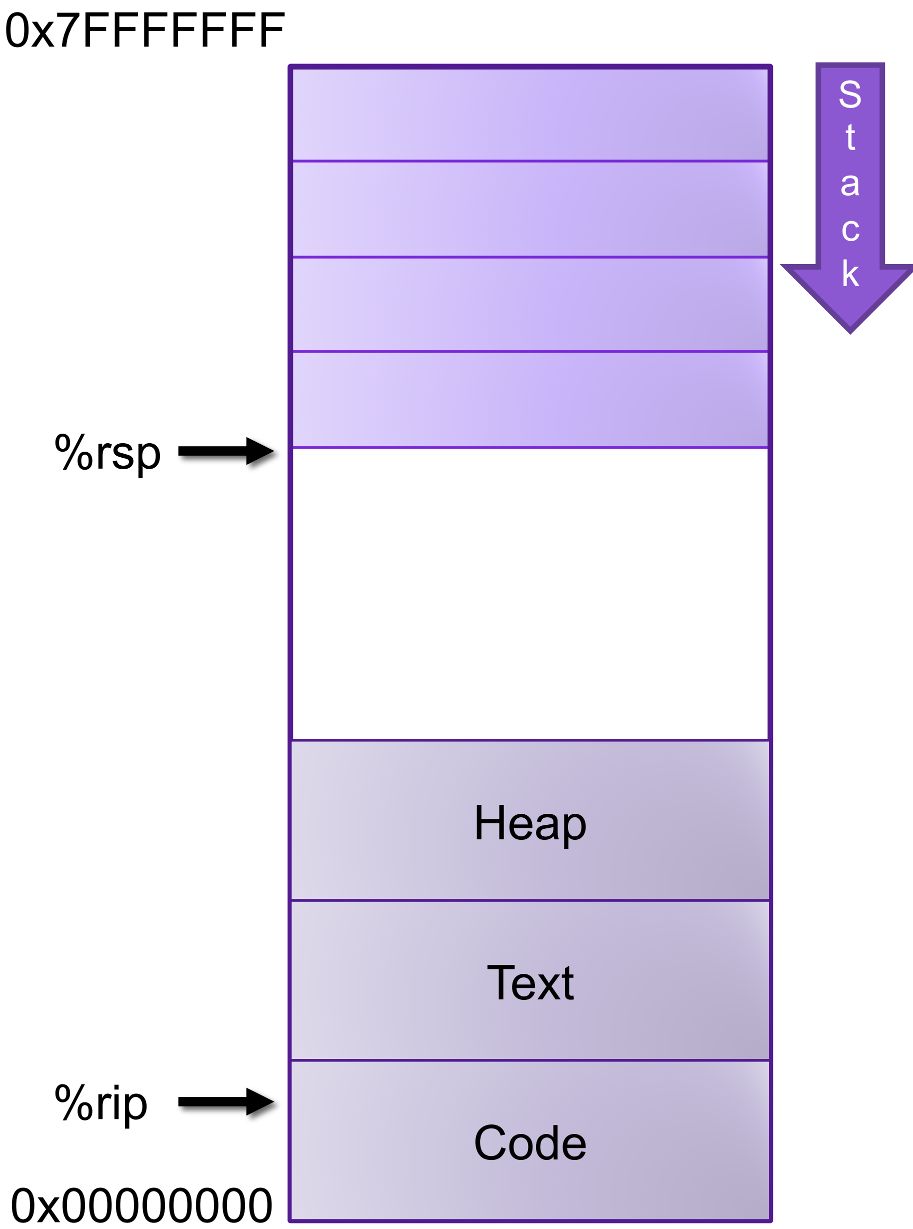
Procedures

- 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:

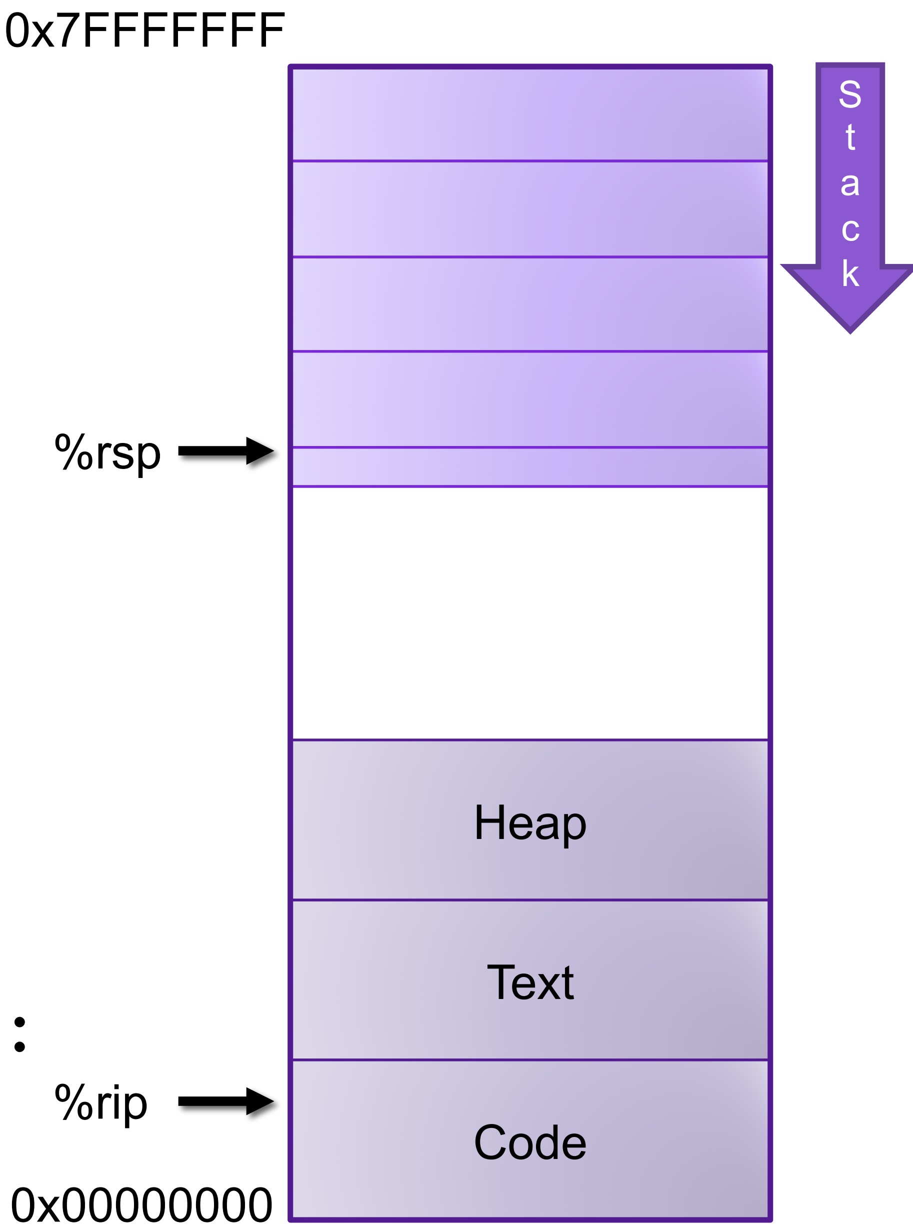
```
D ← M[R[%rsp]]
R[%rsp] ← R[%rsp] + 8
```

- modify %rsp:

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

- modify memory above %rsp:

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



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 in yellow

Procedure Calls, Division of Labor

Caller

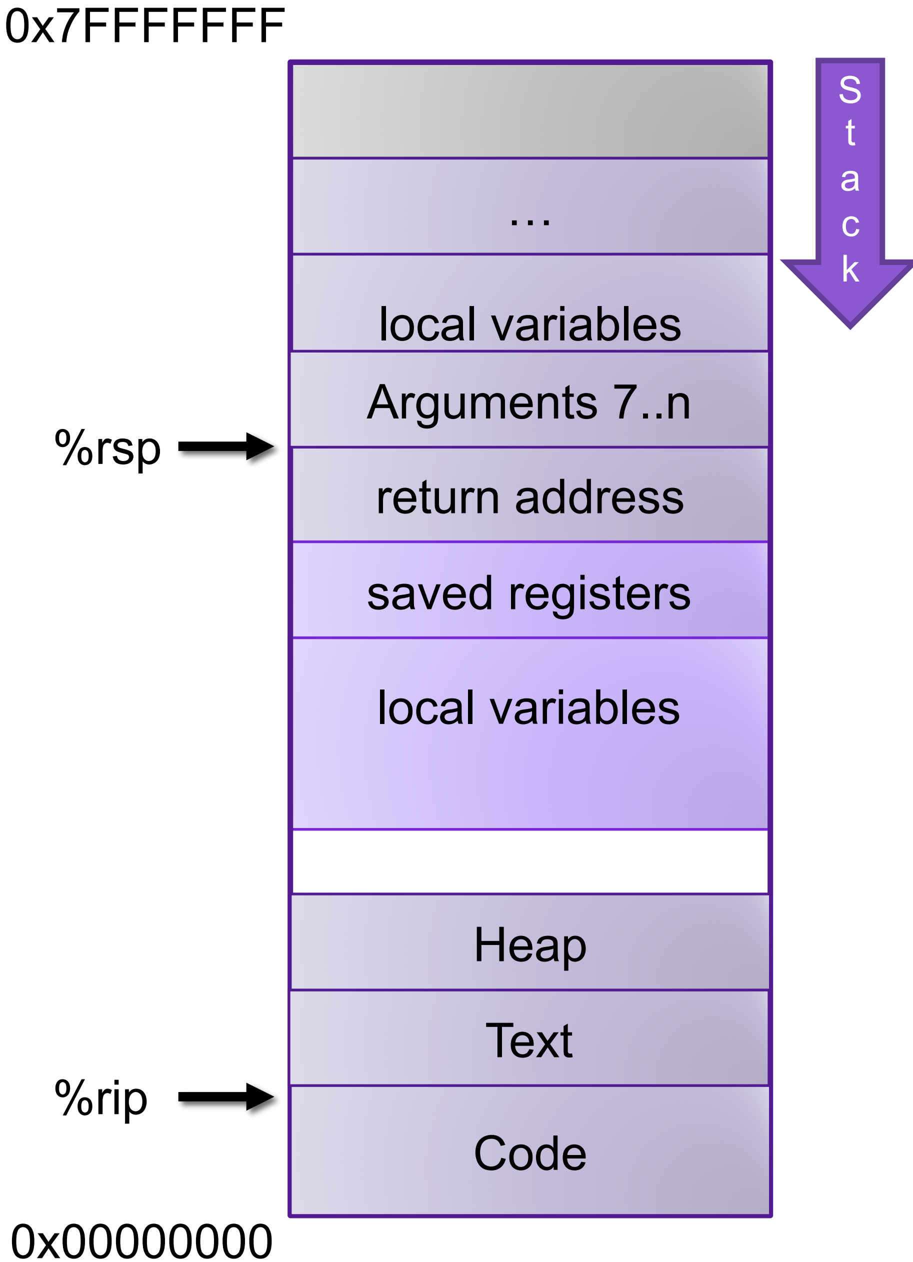
- Before
  - Save registers, if necessary
  - Prepare arguments
  - Make call
- After
  - Restore registers, if necessary
  - Use result

Callee

- Preamble
  - Save registers, if necessary
  - Allocate space on stack
- Exit code
  - Put result in %rax
  - 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: 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  proc
  addq  $16, %rsp
  ret
```

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 + l5;
  return l8;
}

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

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

```
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
```

`enter` and `leave` Instructions

- Complex instructions designed to speed up common operations

- **`enterq size,0`**

```
pushq %rbp
movq %rsp, %rbp
subq size, %rsp
```

Rarely used
The second argument is the nesting level--unimportant in C

- **`leaveq`**

```
movq %rbp, %rsp
popq %rbp
```

Occasionally used,
usually before **`ret`**

Exercise

```
0x400540 <last>:
  400540: 48 89 f8            mov %rdi, %rax      L1
  400543: 48 0f af c6         imul %rsi, %rax     L2
  400547: c3                  ret                 L3

0x400548 <first>:
  400548: 48 8d 77 01         lea 0x1(%rdi),%rsi  F1
  40054c: 48 83 ef 01         sub $0x1, %rdi      F2
  400550: e8 eb ff ff ff      callq 400540 <last> F3
  400555: f3 c2               rep; ret            F4

0x400556 <main>:
  ...
  400560: e8 e3 ff ff         callq 400548 <first> M1
  400565: 48 89 c2            mov %rax, %rdx       M2
  ...
```

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 next week!)
  - 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);
}
```

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

What is in the stack frame?

Preview

```
int proc(int *p);

int example1(int x) {
  int a[5];
  a[3] = 10;
  return proc(a);
}
```

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