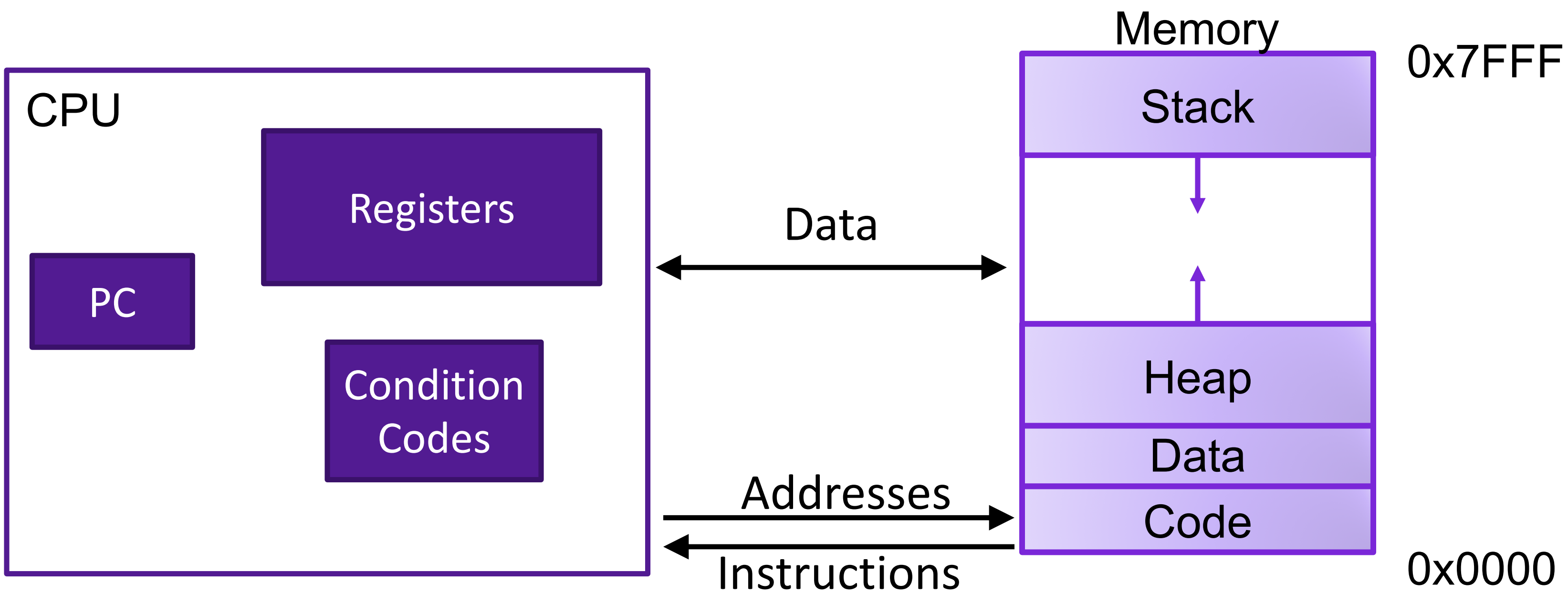
Lecture 6: Procedure Calls in Assembly

CS 105 February 11, 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

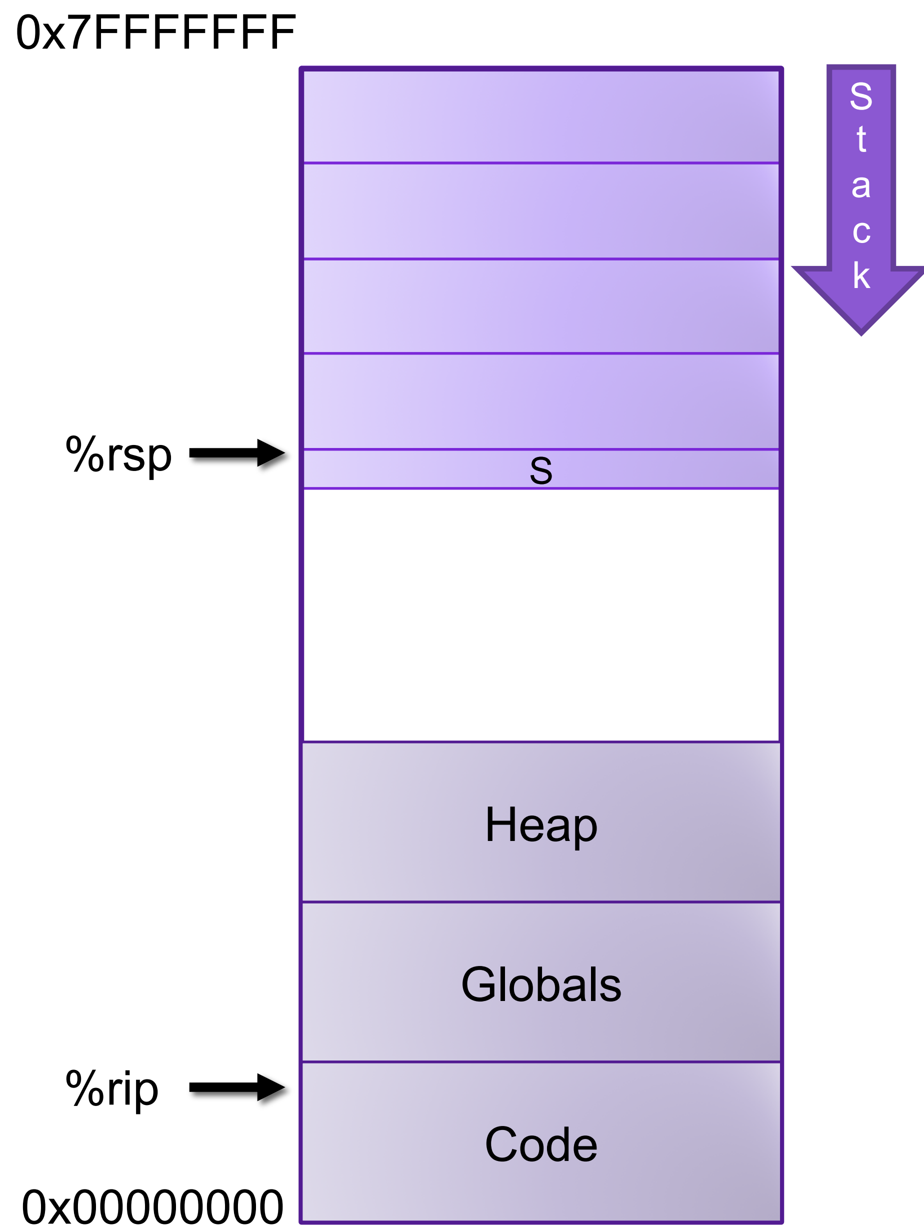
Procedures

- 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 Data:** Must handle parameters and return values
  2) **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
  3) **Allocating memory:** Q must be able to allocate (and deallocate) space for local variables

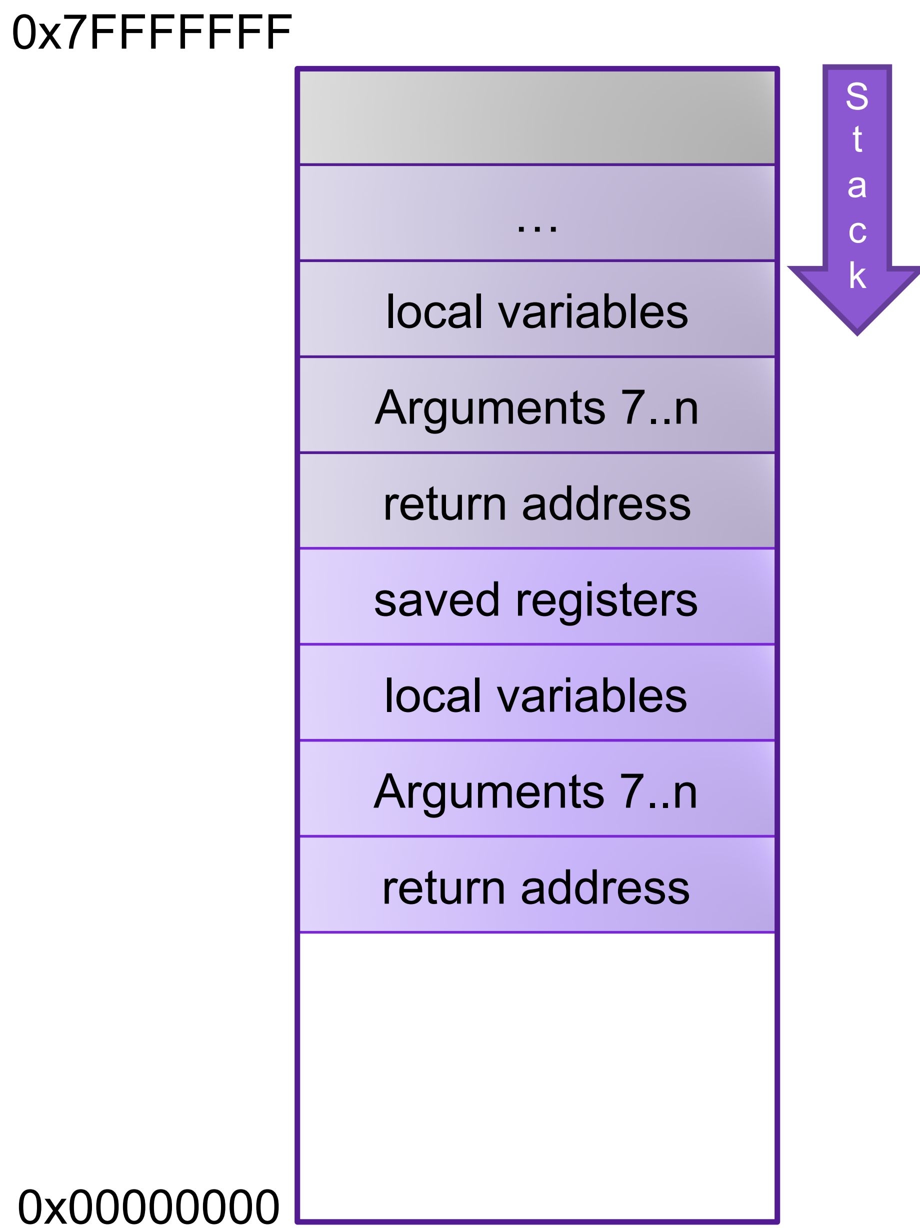
The Stack

- the stack is a region of memory (traditionally the "top" of memory
- grows "down"
- provides storage for functions
- %rsp holds address of top element of stack
- pushq S:
  R[%rsp] ← R[%rsp] – 8;
  M[R[%rsp]] ← S
- popq D:
  D ← M[R[%rsp]]
  R[%rsp] ← R[%rsp] + 8;



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 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 Calls, Division of Labor

Caller

- Before
  - Save registers, if necessary
  - Prepare arguments
  - Make call
    - Implicitly push return address
- After
  - Restore registers
  - Use result

Callee

- Preamble
  - Save registers
  - Allocate space on stack
- Exit code
  - Put result in %rax
  - Restore registers
  - Deallocate space on stack
  - Return

X86-64 Register Usage Conventions

`%rax`, function result	`%r8`
`%rbx`	`%r9`
`%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

Subprogram Example

```
int proc (int *p);

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

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

Listing produced with **`gcc -S -Og source.c`**

`Overflow` preserves **`%rsp`**

Where, relative to **`p`**, is the return address for **`proc`**?

Where, relative to **`p`**, is the return address for **`overflow`**?

Subprogram example

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

gcc –Og –S sum.c

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

gcc –S sum.c

```
sumstore:
   pushq    %rbp
   movq     %rsp, %rbp
   subq     $48, %rsp
   movq     %rdi, -24(%rbp)
   movq     %rsi, -32(%rbp)
   movq     %rdx, -40(%rbp)
   movq     -32(%rbp), %rdx
   movq     -24(%rbp), %rax
   movq     %rdx, %rsi
   movq     %rax, %rdi
   call     plus
   movq     %rax, -8(%rbp)
   movq     -40(%rbp), %rax
   movq     -8(%rbp), %rdx
   movq     %rdx, (%rax)
   leave
   ret
```

`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

```
P:
  pushq %r15
  pushq %r14
  pushq %r13
  pushq %r12
  pushq %rbp
  pushq %rbx
  subq $24, %rsp
  movq %rdi, %rbx
  leaq 1(%rdi), %r15
  leaq 2(%rdi), %r14
  leaq 3(%rdi), %r13
  leaq 4(%rdi), %r12
  leaq 5(%rdi), %rbp
  leaq 6(%rdi), %rax
  movq %rax, (%rsp)
  leaq 7(%rdi), %rdx
  movq %rdx, 8(%rsp)
  movl $0, %eax
  call Q
```

1. Identify which local values get stored in callee-saved registers
2. Identify which local values get stored on the stack
3. Why couldn't the program store all of the local values in callee-saved registers?

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);
}
```

What is in the stack frame?

```
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
```