C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq  %rbp
    movq   %rsp, %rbp
    ...
    popq   %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111111010000011111
```

Computer system:

- OS:
  - Windows 8
  - macOS

- Memory & data
- Integers & floats
- Machine code & C
- x86 assembly
- Procedures & stacks
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C
Mechanisms in Procedures

- **Passing control**
  - To beginning of procedure code
  - Back to return point

- **Passing data**
  - Procedure arguments
  - Return value

- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return

- **Mechanisms all implemented with machine instructions**

- **x86-64 implementation of a procedure uses only those mechanisms required**
Questions to answer about procedures

- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

- To answer some of these questions, we need a *call stack* ...
Outline

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion
Memory Layout

- **High Addresses**
  - Stack
    - Local variables; procedure context
  - Dynamic Data (Heap)
    - Variables allocated with `new` or `malloc`
  - Static Data
    - Static variables (including global variables (C))
  - Literals
    - Literals (e.g., “example”)
- **Low Addresses**
  - Instructions
  - 2^N-1

Winter 2016
Memory Layout

- **Stack**: Writable; not executable. Managed “automatically” (by compiler).
- **Dynamic Data (Heap)**: Writable; not executable. Managed by programmer.
- **Static Data**: Writable; not executable. Initialized when process starts.
- **Literals**: Read-only; not executable. Initialized when process starts.
- **Instructions**: Read-only; executable. Initialized when process starts.

**segmentation faults?**
x86-64 Stack

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register `%rsp` contains lowest stack address
  - `%rsp = address of top element, the most-recentlypushed item that is not-yet-popped`

Stack Pointer: `%rsp`
x86-64 Stack: Push

- **pushq** `Src`
  - Fetch operand at `Src`
    - `Src` can be reg, memory, immediate
  - **Decrement** `%rsp` by 8
  - Store value at address given by `%rsp`

- **Example:**
  - **pushq** `%rcx`
    - Adjust `%rsp` and store contents of `%rcx` on the stack

```
Stack Pointer: `%rsp`  -8
Stack “Top”
```

```
Stack “Bottom”
```

```
High Addresses
Increasing Addresses
Stack Grows Down
Low Addresses 0x00...00
```
x86-64 Stack: Pop

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

Example:

- `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Stack Pointer: `%rsp`
x86-64 Stack: Pop

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

- **Example:**
  - `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Those bits are still there; we’re just not using them.
Today

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion
Procedure Call Overview

- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (e.g. if no arguments or no return value)
The **convention** of where to leave/find things is called the calling convention (or procedure call linkage).

- Details vary between systems
- We will see the convention for x86-64/Linux in detail
- What could happen if our program didn’t follow these conventions?
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}

long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  1. Push return address on stack (*why?, and which exact address?*)
  2. Jump to `label`
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call**: `call label`
  1. Push return address on stack
  2. Jump to `label`
- **Return address**:
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:
    
    ```
    400544: callq 400550 <mult2>
    400549: movq %rax,(%rbx)
    Return address = 0x400549
    ```
- **Procedure return**: `ret`
  1. Pop return address from stack
  2. Jump to address

next instruction happens to be a move, but could be anything
Procedure Call Example (step 1)

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

0000000000400550 <mult2>:
  400550: mov %rdi,%rax
  •
  •
  400557: retq
Procedure Call Example (step 2)

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

0000000000400550 <mult2>:
  400550: mov %rdi,%rax
  
  400557: retq
Procedure Return Example (step 1)

00000000000400540 <multstore>:
  
  400544: callq 400550 <mult2>
  400549: mov %rax,(%rbx)
  
00000000000400550 <mult2>:
  
  400550: mov %rdi,%rax
  
  400557: retq
Procedure Return Example (step 2)

00000000000400540 <multstore>:
  
  400544: callq 400550 <mult2>
  400549: mov %rax,(%rbx)
  

00000000000400550 <mult2>:
  400550: mov %rdi,%rax
  
  400557: retq
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustrations of Recursion & Pointers
Procedure Data Flow

Registers – NOT in Memory!

- First 6 arguments
  - `%rdi`
  - `%rsi`
  - `%rdx`
  - `%rcx`
  - `%r8`
  - `%r9`

- Return value
  - `%rax`

Stack – in Memory!

- Only allocate stack space when needed

High Addresses

0x00...00

Low Addresses
X86-64 Return Values

- By convention, values returned by procedures are placed in the `%rax` register
  - Choice of `%rax` is arbitrary, could have easily been a different register
- **Caller** must make sure to save the contents of `%rax` before calling a **callee** that returns a value
  - Part of register-saving convention
- **Callee** places return value into the `%rax` register
  - Any type that can fit in 8 bytes – integer, float, pointer, etc.
  - For return values greater than 8 bytes, best to return a pointer to them
- Upon return, **caller** finds the return value in the `%rax` register
Data Flow
Examples

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

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: movq %rdx,%rbx    # Save dest
    400544: callq 400550 <mult2> # mult2(x,y)
    # t in %rax
    400549: movq %rax,(%rbx)   # Save at dest
    ...
```

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

Winter 2016
Today

■ Procedures
  ▪ Stack Structure
  ▪ Calling Conventions
    ▪ Passing control
    ▪ Passing data
    ▪ Managing local data
  ▪ Illustration of Recursion
Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java, most modern languages
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

- Stack allocated in *frames*
  - State for a single procedure instantiation
Call Chain Example

Procedure `amI` is recursive (calls itself)
Stack Frames

Contents
- Return address
- If Needed:
  - Local variables
  - Temporary space

Management
- Space allocated when procedure is entered
  - “Set-up” code
- Space deallocated upon return
  - “Finish” code
Example:

call to yoo

```
yoo (...) {
    ....
    who ();
    ....
}
```
Example:

call to who

```c
yoo() {
  who(...) {
    • amI();
    • amI();
    •
  }
}
```

Stack

```
%rbp

yoo

amI

who

%rsp
```

Winter 2016
Example:

call to amI

Stack

Winter 2016
Procedures and Stacks
Example:

recursive call to \texttt{amI}
Example:

(Another) recursive call to amI
Return from:

(another) recursive call to amI

Stack

yoo
who
amI
amI
amI
%rbp
%rsp
amI
Return from:

recursive call to amI

Stack

```
yoo
{    who(...)
    {
        amI(...)
        {
            •
            if(){
                amI()
            }
            •
        }
    }
}
```

```
yoo
{    who
    {
        amI
        {
            •
            if(){
                amI()
            }
            •
        }
    }
}
```

```%rbp
%rsp
```
Return from:

call to amI

```c
yoo()
{
    who(...)
    {
        •
        amI();
        •
        amI();
        •
    }
}
```
Example:

(second) call to amI

```
yoo() {
  who(...) {
    amI(...) {
      •
      if() {
        amI()
      }
    }
    •
  }
}
```
Return from:

(second) call to amI

Stack

{ %rbp

} %rsp

amI

amI

amI

amI

amI

who

who

who {...)

who {...)

who {...)

who {...)

yoo

yoo

yoo

yoo

amI();

amI();

amI();

amI();
Return from: call to who

```c
yoo (...) {
  •
  •
  who ();
  •
}
```

Stack

```
%rbp
%rsp
yoo
who
ami ami
ami
ami
ami
```
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Arguments (if > 6 args) for this call
  - Return address
    - Pushed by `call` instruction

- **Current/ Callee Stack Frame**
  - Old frame pointer (optional)
  - Saved register context
    (when reusing registers)
  - Local variables
    (If can’t be kept in registers)
  - “Argument build” area
    (If callee needs to call another function - parameters for function about to call, if needed)
Example: increment

```c
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

increment:
```
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument (val, y)</td>
</tr>
<tr>
<td>%rax</td>
<td>(x), Return value</td>
</tr>
</tbody>
</table>
Procedure Call Example

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Initial Stack Structure

```
        ...            %rsp
        |              |
```

**call_incr:**

- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

**Register Use(s):**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
</tbody>
</table>
Procedure Call Example  (initial state)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Initial Stack Structure**

```
...  
Rtn address  %rsp
```

**Return address on stack is address of instruction immediately following the call to “call_incr” (not shown here).**
Procedure Call Example (step 1)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

State after setup of space for local variables, in this case only v1 needs space on the stack. The compiler allocated extra space and often does this for a variety of reasons including alignment.
Procedure Call Example (step 2)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Aside: movl is used because 100 is a small positive value that fits in 32 bits. High order bits of rsi get set to zero automatically. It takes one less byte to encode a movl than a movq.

Set up parameters for call to increment

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
</tbody>
</table>

Stack Structure

- Rtn address
- 351
- Unused
Procedure Call Example (step 3)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

call_incr:
- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

increment:
- `movq (%rdi), %rax`
- `addq %rax, %rsi`
- `movq %rsi, (%rdi)`
- `ret`

Stack Structure

<table>
<thead>
<tr>
<th>Register Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

State while inside the function `increment`. Return address on top of stack is address of the `addq` instruction immediately following call to `increment`.
Procedure Call Example (step 4)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

call_incr:
- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

Increment:
- `movq (%rdi), %rax`
- `addq %rax, %rsi`
- `movq %rsi, (%rdi)`
- `ret`

Stack Structure:
- ... 
- Rtn address
  - 451
- Unused
- Rtn address

State while inside the function `increment`. After code in body of `increment` has been executed.

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

State after returning from call to `increment`. Registers and memory have been modified and return address has been popped off stack.

### Stack Structure

<table>
<thead>
<tr>
<th>Rtn address</th>
<th>%rsp+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>451</td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td>%rsp</td>
</tr>
</tbody>
</table>

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 6)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

call_incr:

```
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```

Stack Structure:

- Rtn address: 451
- Unused

Update %rax to contain v1 + v2

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>451 + 351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Stack Structure

- Rtn address
- 451
- Unused

De-allocate space for local vars

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Stack Structure**

State just before returning from call to `call_incr`.

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 9)

long call_incr() {
  long v1 = 351;
  long v2 = increment(&v1, 100);
  return v1+v2;
}

State immediately AFTER returning from call to call_incr. Return addr has been popped off stack and control has returned to the instruction immediately following the call to call_incr (not shown here).

Final Stack Structure

```
call_incr:
  subq  $16, %rsp
  movq  $351, 8(%rsp)
  movl  $100, %esi
  leaq  8(%rsp), %rdi
  call  increment
  addq  8(%rsp), %rax
  addq  $16, %rsp
  ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Register Saving Conventions

- 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! Either:
    - `caller` should save `%rdx` before the call (and restore it after the call)
    - `callee` should save `%rdx` before using it (and restore it before returning)
Register Saving Conventions

■ When procedure yoo calls who:
  ▪ yoo is the caller
  ▪ who is the callee

■ Can a register be used for temporary storage?

■ Conventions
  ▪ “Caller Saved”
    ▪ Caller saves temporary values in its frame before calling
    ▪ Caller restores values after the call
  ▪ “Callee Saved”
    ▪ Callee saves temporary values in its frame before using
    ▪ Callee restores them before returning to caller
x86-64 Linux Register Usage, part1

- **%rax**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure

- **%rdi, ..., %r9**
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure

- **%r10, %r11**
  - **Caller**-saved & restored
  - Can be modified by procedure

**Return value**

- %rax
- %rdi
- %rsi
- %rdx
- %rcx
- %r8
- %r9

**Arguments**

- %r10
- %r11
x86-64 Linux Register Usage, part 2

- `%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
# x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Saved by</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Caller</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
<td>Callee</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>Caller</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>Caller</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>Caller</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>Caller</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
<td>Callee</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5</td>
<td>Caller</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
<td>Caller</td>
</tr>
<tr>
<td>%r10</td>
<td></td>
<td>Caller</td>
</tr>
<tr>
<td>%r11</td>
<td></td>
<td>Caller</td>
</tr>
<tr>
<td>%r12</td>
<td></td>
<td>Callee</td>
</tr>
<tr>
<td>%r13</td>
<td></td>
<td>Callee</td>
</tr>
<tr>
<td>%r14</td>
<td></td>
<td>Callee</td>
</tr>
<tr>
<td>%r15</td>
<td></td>
<td>Callee</td>
</tr>
</tbody>
</table>
Callee-Saved Example, (step 1)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

**Initial Stack Structure**

```
... %rsp
Rtn address
```

**Resulting Stack Structure**

```
... %rsp+8
Rtn address
Saved %rbx
351
Unused %rsp
```
Callee-Saved Example (step 2)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

call_incr2:

- `pushq %rbx`
- `subq $16, %rsp`
- `movq %rdi, %rbx`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq %rbx, %rax`
- `addq $16, %rsp`
- `popq %rbx`
- `ret`

Resulting Stack Structure:

```
...  
Rtn address
Saved %rbx
351  %rsp+8
Unused  %rsp
```

Pre-return Stack Structure:

```
...  
Rtn address  %rsp
```
Why Caller and Callee Saved?

- We want *one* calling convention to simply separate implementation details between caller and callee.

- In general, neither caller-save nor callee-save is “best”:
  - If caller isn’t using a register, caller-save is better.
  - If callee doesn’t need a register, callee-save is better.
  - If “do need to save”, callee-save generally makes smaller programs.
    - Functions are called from multiple places.

- So… “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring.
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
Recursive Function

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

```assembly
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```
Recursive Function: Base Case

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

### pcount_r:
- `movl $0, %eax`
- `testq %rdi, %rdi`
- `je .L6`
- `pushq %rbx`
- `movq %rdi, %rbx`
- `andl $1, %ebx`
- `shrq %rdi`
- `call pcount_r`
- `addq %rbx, %rax`
- `popq %rbx`

**.L6:**
- `rep; ret`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Recursive Function: Register Save

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

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
</tbody>
</table>

**pcount_r:**

```
movl $0, %eax
testq %rdi, %rdi
ej .L6
pushq %rbx
movq %rdi, %rbx
andl $1, %ebx
shrq %rdi
call pcount_r
addq %rbx, %rax
popq %rbx

.L6:
  rep; ret
```

**Register Use(s):**

- %rdi: x
- %rbx: Saved
- %rsp: Rtn address

Winter 2016

Procedures and Stacks
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

Recursive Function: Call Setup

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x &gt;&gt; 1</td>
<td>Rec. argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
</tbody>
</table>
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}

Register | Use(s)            | Type           
----------|-------------------|----------------
%rbx      | x & 1             | Callee-saved   
%rax      | Recursive call return value |
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

Register | Use(s)     | Type        
----------|------------|-------------
%rbx      | x & 1      | Callee-saved
%rax      | Return value |
Recursive Function: Completion

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

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

... %rsp
Observations About 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 - described in future lecture)
  - 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
x86-64 Stack Frames

- Often (ideally), x86-64 functions need no stack frame at all
  - Just a return address is pushed onto the stack when a function call is made

- A function *does* need a stack frame when it:
  - Has too many local variables to hold in registers
  - Has local variables that are arrays or structs
  - Uses the address-of operator (&) to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Needs to save the state of caller-save registers before calling a procedure
  - Needs to save the state of callee-save registers before modifying them
x86-64 Procedure Summary

- **Important Points**
  - Procedures are a combination of instructions and conventions
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
  - **Recursion handled by normal calling conventions**
    - Caller can store values in local stack frame and in callee-saved registers
    - Put function arguments at top of stack
    - Result return in %rax
One more x86-64 example

- Example of passing more than 6 parameters and passing addresses of local variables
- The following example, along with a brief re-cap of x86-64 calling conventions is in this video:

5. Procedures and Stacks
   - ...
   - 6. x86-64 Calling Conventions

https://courses.cs.washington.edu/courses/cse351/videos/05/056.mp4
x86-64 Example (1)

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
```
subq $32,%rsp
movq $1,16(%rsp)  # x1
movl $2,24(%rsp)  # x2
movw $3,28(%rsp)  # x3
movb $4,31(%rsp)  # x4
• • •
```

Return address to caller of call_proc

NB: Details may vary depending on compiler.
x86-64 Example (2) – Allocate local vars

long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1 + x2) * (x3 - x4);
}

call_proc:
    subq $32,%rsp
    movq $1,16(%rsp) # x1
    movl $2,24(%rsp) # x2
    movw $3,28(%rsp) # x3
    movb $4,31(%rsp) # x4
    ...

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
</table>
|    |    |    | 24
| x1 |    |    | 16
|    |    |    | 8
|    |    |    | %rsp
x86-64 Example (3) – setup params to proc

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:

• • •

```
leaq 24(%rsp),%rcx  # %rcx=&x2
leaq 16(%rsp),%rsi  # %rsi=&x1
leaq 31(%rsp),%rax  # %rax=&x4
movq %rax,8(%rsp)   # arg8=&4
movl $4,(%rsp)      # arg7=4
leaq 28(%rsp),%r9   # %r9=&x3
movl $3,%r8d        # %r8 = 3
movl $2,%edx        # %rdx = 2
movq $1,%rdi        # %rdi = 1
```

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

Same instructions as in video, just a different order.
### x86-64 Example (4) – setup params to proc

```c
long int call_proc() {
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

**call_proc:**

- \( \text{leaq} \ 24(\%\text{rsp}),\%\text{rcx} \)
- \( \text{leaq} \ 16(\%\text{rsp}),\%\text{rsi} \)
- \( \text{leaq} \ 31(\%\text{rsp}),\%\text{rax} \)
- \( \text{movq} \ %\text{rax},8(\%\text{rsp}) \)
- \( \text{movl} \ $4,(\%\text{rsp}) \)
- \( \text{leaq} \ 28(\%\text{rsp}),%r9 \)
- \( \text{movl} \ $3,%r8d \)
- \( \text{movl} \ $2,%edx \)
- \( \text{movq} \ $1,%rdi \)
- \( \text{call} \ \text{proc} \)

**Arguments passed in (in order):**

- rdi, rsi, rdx, rcx, r8, r9

**Note sizes**

- Arg 8 (8 bytes)
- Arg 7 (8 bytes)

**Return address to caller of call_proc**

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

**%rsp**
x86-64 Example (5) – call proc

long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

call_proc:
    • • •
    leaq 24(%rsp),%rcx
    leaq 16(%rsp),%rsi
    leaq 31(%rsp),%rax
    movq  %rax,8(%rsp)
    movl $4,(%rsp)
    leaq 28(%rsp),%r9
    movl $3,%r8d
    movl $2,%edx
    movq $1,%rdi
    call proc
    • • •

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arg 8</td>
</tr>
<tr>
<td>Arg 7</td>
</tr>
</tbody>
</table>

Return address to line after call to proc

%rsp
x86-64 Example (6) – after call to proc

```c
long int call_proc() {
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1 + x2) * (x3 - x4);
}
```

call_proc:
```
movswl 28(%rsp),%eax  # %eax=x3
movsbl 31(%rsp),%edx  # %edx=x4
subl %edx,%eax       # %eax=x3-x4
cltq
movslq 24(%rsp),%rdx  # %rdx=x2
addq 16(%rsp),%rdx    # %rdx=x1+x2
imulq %rdx,%rax      # %rax=rax*rdx
addq $32,%rsp
ret
```

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Arg 8</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Arg 7</td>
<td>%rsp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

movs – move and sign extend

cltq - sign extend
%eax into %rax
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

call_proc:
    ...
    movswl 28(%rsp),%eax
    movsbl 31(%rsp),%edx
    subl %edx,%eax
    cltq
    movslq 24(%rsp),%rdx
    addq 16(%rsp),%rdx
    imulq %rdx,%rax
    addq $32,%rsp
    ret

Return address to caller of call_proc
%rsp