Announcements

- Lab 1 graded
  - Late days
  - Extra credit
- HW 2 out
- Lab 2 prep in section tomorrow
  - Bring laptops!
- Slides

HTTP://XKCD.COM/1270/
Roadmap

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
110000011111101000011111
```

OS:

Windows 8
Mac
Linux

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly

Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Mechanisms required for 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

- **All implemented with machine instructions**
  - An x86-64 procedure uses only those mechanisms required for that procedure
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

- **Instructions**: Low addresses (0)
- **Literals**: Address range \(2^{N-1}\)
- **Static Data**: Address range \(0\) to \(2^{N-1}\)
- **Dynamic Data (Heap)**: Address range \(2^{N-1}\) to \(2^N - 1\)
- **Stack**: Address range \(2^N - 1\) to \(2^{N+1} - 1\)

- **Local variables**: Procedure context
- **Variables allocated with new or malloc**: Dynamic Data (Heap)
- **Static variables**: (including global variables (C))
- **Large constants**: (e.g., “example”)
Memory Layout

- **Stack**
  - Managed “automatically” (by compiler)
  - Writable; not executable

- **Dynamic Data (Heap)**
  - Managed by programmer
  - Writable; not executable

- **Static Data**
  - Initialized when process starts
  - Writable; not executable

- **Literals**
  - Initialized when process starts
  - Read-only; not executable

- **Instructions**
  - Initialized when process starts
  - Read-only; executable

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-recently-pushed 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`

**Stack "Bottom"**

**Stack "Top"**

**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`

Those bits are still there; we’re just not using them.
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?
Code Examples

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)

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

0000000000400550 <mult2>:
  400550: mov %rdi, %rax
  •
  •
  400557: retq

0x130
0x128
0x120
0x118 0x400549
%rsp 0x118
%rip 0x400557
Procedure Return Example (step 2)

0000000000400540 <multstore>:

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

0x400549
0x120
0x128
0x130

%rip 0x400549

%rsp 0x120

0000000000400550 <mult2>:

- 400550: mov %rdi,%rax
- 
- 400557: retq
Procedures

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

Diagrams show:
- `Diane’s Silk Dress Costs $89`
- Stack levels:
  - `Arg n`
  - `Arg 8`
  - `Arg 7`
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;
}
```

```assembly
muliq %rdi,%rsi # a * b
...
```
Outline

- 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
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
Call Chain Example

yoo(...)
{
  •
  •
  who();
  •
}

who(...)
{
  •
  •
amI();
  •
}

amI(...)
{
  •
  •
  if(...){
    amI()
  }
  •
}

Example Call Chain

Procedure amI is recursive (calls itself)
Example:

call to yoo

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

Stack

```
<table>
<thead>
<tr>
<th>%rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>yoo</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>who</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>amI</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>amI</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>amI</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>amI</th>
</tr>
</thead>
</table>
```
Example:

**call to who**

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

Stack:

```
%rbp
%

who
```

```plaintext
yoo
```
Example:

**call to amI**
Example:

recursive call to amI

Stack

yoo

who

amI

amI

%rbp

%rsp
Example:

(anoth​er) recursive call to `amI`

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

Stack

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

(another) **recursive call to amI**
Return from:

recursive call to amI
Return from:

call to amI
Example:

(Second) call to amI
Return from:

(Second) call to amI
Return from: call to who

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

Stack

```
%rbp
%rsp
```

```c
yoo
who
amI
amI
amI
amI
```
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Extra 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)
April 22
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 (initial state)**

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

**Initial Stack Structure**

- Return addr <main+8> ← %rsp

- **Return address on stack is:**
  - address of instruction immediately following the call to “call_incr” (shown here as main, but could be anything).
Procedure Call Example (step 1)

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

Stack Structure

Allocate space for local vars

- Setup space for local variables
  - Only v1 needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment.

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
Procedure Call Example (step 2)

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

**Stack Structure**

- Return addr `<main+8>`
- 351 ← %rsp+8
- Unused ← %rsp

**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>

**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`
Procedure Call Example (step 3)

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

**Stack Structure**

- Return addr <main+8>
- 351
- Unused
- Return addr <call_incr+?>

**State while inside increment.**

- **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`.

**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>100</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 4)

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

Stack Structure

- Return addr <main+8>
- 451
- Unused
- Return addr <call_incr+?>

- Call while inside `increment`.
  - After code in body 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>

Increment:

```asm
movq (%rdi), %rax # x = *p
addq %rax, %rsi # y = x+100
movq %rsi, (%rdi) # *p = y
ret
```

State while inside `increment`.

- After code in body has been executed.
Procedure Call Example (step 5)

```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`<br>
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

**Stack Structure**

```
...<br>
Return addr <main+8><br>
451<br>Unused<br>%rsp+8<br>%rsp  
```

**After returning from call to increment.**
- Registers and memory have been modified and return address has been popped off stack.

<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;
}
```

**Stack Structure**

- Return addr `<main+8>` ←%rsp+8
- 451 ←%rsp
- Unused

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>451+351</td>
</tr>
</tbody>
</table>

Update `%rax` to contain `v1+v2`
Procedure Call Example (step 7)

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

Stack Structure

<table>
<thead>
<tr>
<th>Return addr &lt;main+8&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>451</td>
</tr>
<tr>
<td>Unused</td>
</tr>
</tbody>
</table>

De-allocate space for local vars

Register | Use(s) |
----------|--------|
%rax      | 802    |
Procedure Call Example (step 8)

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

Stack Structure

Stack

Return addr <main+8> ← %rsp

State just before returning from call to call_incr.

---

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>
Procedure Call Example (step 9)

```c
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
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here).

---

```
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?
  - No! 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 stack frame before calling
    - Caller restores values after the call
  - “Callee Saved”
    - Callee saves temporary values in its stack frame before using
    - Callee restores them before returning to caller
x86-64 Linux Register Usage, part 1

- **%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
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

**Callee**-saved Temporaries
- %rbx
- %r12
- %r13
- %r14

Special
- %rbp
- %rsp
## x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r10</td>
<td></td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r11</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r12</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td></td>
<td>Callee saved</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

```
<table>
<thead>
<tr>
<th>Rtn address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>
```

Resulting Stack Structure

```
<table>
<thead>
<tr>
<th>Rtn address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp+8</td>
</tr>
<tr>
<td>Saved %rbx</td>
</tr>
<tr>
<td>351</td>
</tr>
<tr>
<td>Unused</td>
</tr>
</tbody>
</table>
```
Callee-Saved Example (step 2)

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

Stack Structure

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

<table>
<thead>
<tr>
<th>Saved%rbx</th>
</tr>
</thead>
<tbody>
<tr>
<td>351</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>

Pre-return Stack Structure

<table>
<thead>
<tr>
<th>Rtn address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>

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

```
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);
}
```

**Register Use(s) and Type**

<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>

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

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

Trick because some HW doesn’t like jumping to ret
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);
}
```

<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
je .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) Type
%rdi x Argument

<main+15>
Saved %rbx ← %rsp
Recursive Function: Call Setup

```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 &gt;&gt; 1</td>
<td>Recursive arg</td>
</tr>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
</tbody>
</table>

Pseudocode for `pcount_r`:

```assembly
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
```

```assembly
.L6: rep; ret
```

Stack Frame:

- `rtn <main+?>`
- `Saved %rbx ← %rsp`
Recursive Function: Call

/* 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>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Recursive call return value</td>
<td>–</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            Return value

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

rtn <main+?>
Saved %rbx ← %rsp
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);
}
```

<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>

```
ptcount_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 ← %rdi

Saved %rbx

rtn <main+ ?>
```
Observations About Recursion

- **Works without any 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 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
• • •
```

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

```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

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

←%rsp
x86-64 Example (3) – setup params to 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 # %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
```

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

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>Arguments passed in</th>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Arg 8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Arg 7              | ←%rsp

Same instructions as in video, just a different order.
x86-64 Example (4) – setup params to 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>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>←%rsp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Note sizes
x86-64 Example (5) – call 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
- 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>x1</td>
<td></td>
</tr>
<tr>
<td>Arg 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 7</td>
<td></td>
<td></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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>←%rsp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

movs__:
move and sign extend

cltq:
sign extend %eax into %rax
(special-case to save space)
x86-64 Example (7) – de-allocate local vars

```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
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