Procedures & The Stack II
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/244/
Administrivia

- Lab 2 due Friday

- **Midterm** on Nov. 2 in lecture
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Historically my exams have averages of 65-70%
  - Check Piazza this week for announcements & practice problems

- **Midterm review session**
  - 5-7pm on Monday, Oct. 31 in EEB 105

- Look for additional staff office hours as well
Example: `increment`

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

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\textsuperscript{st} arg ((p))</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} arg ((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;
}
```

- Return address on stack is the address of instruction immediately following the call to “call_incr”
  - Shown here as `main`, but could be anything

![Initial Stack Structure](image)

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

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

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
Procedure Call Example (step 2)

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

**Stack Structure**

```
long 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
```

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.

**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>
</tbody>
</table>
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+?>` [←%rsp]

- **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></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 Increment**

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

- **State while inside `increment`**
  - *After* code in body has been executed

**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 5)

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

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

**Stack Structure**

```
Return addr <main+8>
%
451
Unused
%
```

- **Update %rax to contain v1+v2**

**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>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**

- Return addr `<main+8>`
- 451
- `Unused`

**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>802</td>
</tr>
</tbody>
</table>

*De-allocate space for local vars*
Procedure Call Example (step 8)

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

- State just before returning from call to `call_incr`

Stack Structure

- Return addr `<main+8>`

<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>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
  - Control has returned to the instruction immediately following the call to call_incr (not shown here)

### 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>802</td>
</tr>
</tbody>
</table>
Procedures

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

- When procedure `yoo` calls `who`:
  - `yoo` is the **caller**
  - `who` is the **callee**

- Can registers be used for temporary storage?

```
yoo:
  ...
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  ...
  ret
```

```
who:
  ...
  subq $18213, %rdx
  ...
  ret
```

- 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

- **“Caller-saved” registers**
  - It is the caller’s responsibility to save any important data in these registers before calling another procedure (i.e. the callee can freely change data in these registers)
  - Caller saves values in its stack frame before calling Callee, then restores values after the call

- **“Callee-saved” registers**
  - It is the callee’s responsibility to save any data in these registers before using the registers (i.e. the caller assumes the data will be the same across the callee procedure call)
  - Callee saves values in its stack frame before using, then restores them before returning to caller
Silly Register Convention Analogy

1) Parents (*caller*) leave for the weekend and give the keys to the house to their child (*callee*)
   - Being suspicious, they put away/hid the valuables (*caller-saved*) before leaving
   - Warn child to leave the bedrooms untouched: “These rooms better look the same when we return!”

2) Child decides to throw a wild party (*computation*), spanning the entire house
   - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed (*callee-saved*) before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms

3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives
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
# x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>Return value - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td><strong>Callee</strong> saved</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack pointer</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td><strong>Callee</strong> saved</td>
</tr>
<tr>
<td><code>%r8</code></td>
<td>Argument #5 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>Argument #6 - <strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td><strong>Caller</strong> saved</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td><strong>Caller</strong> Saved</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td><strong>Callee</strong> Saved</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td><strong>Callee</strong> Saved</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td><strong>Callee</strong> Saved</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td><strong>Callee</strong> 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

![Initial Stack Structure Diagram]

Resulting Stack Structure

![Resulting Stack Structure Diagram]
Callee-Saved Example (step 2)

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

**Stack Structure**

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

**Pre-return Stack Structure**

```
Rtn address %rsp
```

---

**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
```
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.
Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values

- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- 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);
}
```

**Compiler Explorer:**

[https://godbolt.org/g/4ZJbz1](https://godbolt.org/g/4ZJbz1)

- Compiled with `-O1` for brevity instead of `-Og`
- Try `-O2` instead!
Recursive Function: Base Case

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

Trick because some AMD hardware doesn’t like jumping to ret (don’t worry about it)
Recursive Function: **Callee Register Save**

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

Need original value of `x` *after* recursive call to `pcount_r`.

“Save” by putting in `%rbx` *(callee saved)*, but need to save old value of `%rbx` before you change it.

---

**The Stack**

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

### Register Use(s) 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>
</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
%rdi  x (new)  Argument
%rbx  x (old)  Callee saved

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

The Stack

%rsp →
 saved %rbx
rtn <main+?>
...

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

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

### The Stack

- **%rax**: Recursive call return value
- **%rbx**: x (old)

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

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Recursive call return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

**Example:**

if original \( x = 0b1000 \):

- \( \%rbx = 0b0100 \)
- \( \%rbx = 0b0010 \)
- \( \%rbx = 0b0001 \)
- \( \%rbx = 0b1000 \)

**Base case:**

- \( \%rbx = 0b1000 \)
Recursive Function: Result

```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) Type**

<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>
<tr>
<td>%rbx</td>
<td>x&amp;1</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

The Stack

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

%rsp → saved %rbx

...
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) Type

<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>
<tr>
<td>%rbx</td>
<td>Previous</td>
<td>%rbx value</td>
</tr>
</tbody>
</table>

The Stack

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
.L6:
    rep ret
```
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)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)

- Also works for mutual recursion (P calls Q; Q calls P)
x86-64 Stack Frames

- Many x86-64 procedures need no stack frame at all
  - Only return address is pushed onto the stack when calling another procedure

- A procedure *does need* a stack frame when it:
  - Has too many local variables to hold in caller-saved registers
  - Has local variables that are arrays or structs
  - Uses & to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using caller-saved registers and then calls a procedure
  - Modifies/uses callee-saved registers
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

- **Heavy use of registers**
  - Faster than using memory
  - Use limited by data size and conventions

- **Minimize use of the Stack**
One more x86-64 example that shows passing of more than 6 arguments and passing addresses of local variables. The following example, along with a brief recap of x86-64 calling conventions is in the following video:

- https://courses.cs.washington.edu/courses/cse351/videos/05/056.mp4
- Alternate (but similar) version: https://godbolt.org/g/E7UFJ7
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` ← `%rsp`

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

leave empty to increase alignment of x4

pack into two words

← %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 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
    call proc
    . . .

Argument order:
- Diane’s Silk Dress Cost $89
- %rdi, %rsi, %rdx, %rcx, %r8, %r9
  x1, 6x1, x2, 6x2, x3, 6x3

Same instructions as in video, just a different order.
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:

\[
\begin{align*}
\text{call_proc:} & \\
\text{\hspace{1cm} \text{\textbf{. . .}}} & \\
\text{\hspace{1cm} leaq} & 24(\%\text{rsp}),\%\text{rcx} \\
\text{\hspace{1cm} leaq} & 16(\%\text{rsp}),\%\text{rsi} \\
\text{\hspace{1cm} leaq} & 31(\%\text{rsp}),\%\text{rax} \\
\text{\hspace{1cm} movq} & \%\text{rax},8(\%\text{rsp}) \\
\text{\hspace{1cm} movl} & \$4,(\%\text{rsp}) \\
\text{\hspace{1cm} leaq} & 28(\%\text{rsp}),\%\text{r9} \\
\text{\hspace{1cm} movl} & \$3,\%\text{r8d} \\
\text{\hspace{1cm} movl} & \$2,\%\text{edx} \\
\text{\hspace{1cm} movq} & \$1,\%\text{rdi} \\
\text{\hspace{1cm} \textbf{call} proc} & \\
\text{\hspace{1cm} \textbf{. . .}}
\end{align*}
\]

Return address to caller of \textbf{call proc}

\[
\begin{array}{|c|c|c|c|}
\hline
x4 & x3 & x2 & x1 \\
\hline
\text{Arg 8} & \text{Arg 7} & \text{. . .} & \%\text{rsp} \\
\hline
\end{array}
\]
### x86-64 Example (5) – after call to proc

#### long int call_proc()
```c
{
    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:
```assembly
    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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24</th>
<th>16</th>
<th>8</th>
<th>← %rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>Arg 8</td>
<td>Arg 7</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 (6) – de-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:
    .
    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  ; restore stack pointer
    ret
```

Return address to caller of `call_proc` ← %rsp