CMPT 295
Machine-Level Programming
Lecture 17 – Function Call + Stack - Managing local data
Last Lecture

- Passing data mechanism – using stack
  - x86-64 function call convention (ABI):
    - Before calling a function, **caller** must copy the first 6 (integral) arguments into registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, respectively.
    - Then it must push the rest of the arguments on the stack in reverse order.
    - When passing data that is a memory address (i.e., a pointer) during function call, **caller** must make use of the stack in order to create such a memory address.
    - Before returning to **caller**, **callee** must copy returned value into `%rax`.

- Stack frame content
Today’s Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, move operation
  - Memory addressing modes
- Operation leaq and Arithmetic & logical operations
- Conditional Statement – Condition Code + cmovX
- Loops
  - Function call – Stack – Recursion
    - Overview of Function Call
    - Memory Layout and Stack - x86-64 instructions and registers
    - Passing control
    - Passing data – Calling Conventions
    - Managing local data
    - Recursion
- Array
- Floating-point operations
Managing local data

- When we need local variables and/or more temporary space, can we use registers?
  - Yes! First choice as they are the fastest storage location on a computer

- But, since registers are shared by all functions, we need to follow some convention, otherwise:

```assembly
who:
  . . .
  movq $15213, %rbx
  call amI
  addq %rbx, %rax
  . . .
  ret

amI:
  . . .
  subq $18213, %rbx
  . . .
  ret
```
x86-64 “register saving” convention

- When we need local variables and/or more temporary space, we can use:

1. Registers
   - A function can utilise unused registers (only when needed)
   - Some registers are referred to as **callee saved** registers:
     - %rbx, %rsp, %rbp, %r12 to %r15
   - **Callee** must preserve the values of these registers before using them
   - **How?**
     - **Example:** `pushq %r13` at start of function, then **callee** can use `%r13`, then `popq %r13` before returning to the **caller**
x86-64 “register saving” convention

1. Register (cont’d)
   - Some registers are referred to as **caller saved** registers:
     - %=r10, %=r11, %=rax and all 6 registers used for arguments (data passing)
     - **Caller** must preserve the values of these registers before using them
   - How?
     - Example: `pushq %r10` before calling a function, then `popq %r10` after the **callee** has returned

   If `pushq` more than 1 register, then `popq` them in reverse order.
x86-64 “register saving” convention

- **Solution 1:**

  who:
  ```
  movq $15213, %rbx 
call amI 
  addq %rbx, %rax 
  ret 
  ```

  amI:
  ```
  subq $18213, %rbx 
  ret 
  ```

- **Solution 2:**

  who:
  ```
  movq $15213, %r10 
  call amI 
  addq %r10, %rax 
  ret 
  ```

  amI:
  ```
  subq $18213, %r10 
  ret 
  ```
x86-64 “register saving” convention

**callee saved** registers:
- **Callee** must save & restore

<table>
<thead>
<tr>
<th>%rbx</th>
<th>%r12</th>
<th>%r13</th>
<th>%r14</th>
<th>%r15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callee saved Temporaries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>%rsp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**caller saved** registers:
- Can be modified by **callee**

<table>
<thead>
<tr>
<th>%rax</th>
<th>%rdi</th>
<th>%rsi</th>
<th>%rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%rcx</th>
<th>%r8</th>
<th>%r9</th>
<th>%r10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters/arguments</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller saved temporaries</td>
</tr>
</tbody>
</table>
Managing local data

- When we need local variables and/or more temporary space, we can use:

2. Stack

- A function can use the stack to store the values of its local variables and temporary space

- Set-up and Clean-up code:
  - Example: `subq $16, %rsp` and `addq $72, %rsp`

- During function execution:
  - Example: `leaq, mov*`

If we run out of registers!

Must remember to clean-up the stack before returning to the caller!
Local variables on Stack – Example

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

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $15213, 8(%rsp)
    movl $3000, %esi
    leaq 8(%rsp), %rdi
    call incr
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```
Callee saved registers – Example

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

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq  %rbx
    subq   $16, %rsp
    movq   %rdi, %rbx
    movq   $15213, 8(%rsp)
    movl   $3000, %esi
    leaq   8(%rsp), %rdi
    call   incr
    addq   %rbx, %rax
    addq   $16, %rsp
    popq   %rbx
    ret
```
Managing local data in registers and on stack

x86-64 “register saving” convention:

- **callee saved** registers: %rbx, %rsp, %rbp, %r12 to %r15
- **caller saved** registers: %r10, %r11, %rax and all 6 registers used for arguments (data passing)
Next lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, move operation
  - Memory addressing modes
- Operation leaq and Arithmetic & logical operations
- Conditional Statement – Condition Code + cmovX
- Loops
- Function call – Stack – Recursion
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Floating-point operations