CMSC 313
Introduction to Computer Systems
Lecture 26
Procedure Implementation

Alan Sussman
als@cs.umd.edu

Y86 procedure-related instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Label</td>
<td>push PC on stack PC ← Label</td>
<td>Call function</td>
</tr>
<tr>
<td>ret</td>
<td>PC ← pop value from stack</td>
<td>Return from function</td>
</tr>
</tbody>
</table>

- These two instructions are used to handle function calls in Y86
- `call` instruction pushes the return address on the stack, then jumps to the destination address
- `ret` instruction returns from a call
- functions are responsible for ensuring the stack is properly adjusted before returning
Implementing functions

- When calling a function, the computer needs to know where execution will resume once the function returns (the \textit{return address}).
- The \texttt{call} instruction stores this address on the stack; when the called function (the \textit{callee}) is done, we'll need to undo any changes made to the stack so that we can get back to the caller.

Call stack

- \texttt{%esp} contains the address of the top of the stack (the address of the element most recently pushed).
- The stack grows downward, from a maximum address of 0x1000.
- To use it, an assembly program must first initialize the stack pointer:
  \begin{verbatim}
  irmovl $0x1000, %esp
  \end{verbatim}
- Then, it can push and pop as necessary.

Simple function call example

```assembly
main:    irmovl $0x1000, %esp
          call   f
          rmmovl %ebx, n
          call   g
          halt
f:       call   g
n = 1;
g:       mrmovl n, %ecx
          ret
Loop:    irmovl $0, %eax
          addl %eax, %ecx
          jle EndLoop
          writeln %ecx
          irmovl $1, %edx
          subl %edx, %ecx
          rmmovl %ecx, n
          jmp   Loop
EndLoop: ret
n:       .long 2
```

Simple function call example

```assembly
int n = 2;
void f();
void g();

int main() {
  f();
  n = 1;
  g();
  return 0;
}
void f() {
  g();
}
void g() {
  while (n > 0)
    printf("%d", n--);
}
```

Simple function call example

```assembly
0x000: 308400100000 | main:    irmovl $0x1000, %esp
          call   f
          rmmovl %ebx
          call   g
          halt
0x011: 90           |          ret
0x012: 50184c000000 | g:       mrmovl n, %ecx
          jle EndLoop
          writeln %ecx
          irmovl $1, %edx
          subl %edx, %ecx
          rmmovl %ecx, n
          jmp   Loop
EndLoop: ret
n:       .long 2
```

Simple function call example

```assembly
0x000: 308400100000 | main:    irmovl $0x1000, %esp
          call   f
          rmmovl %ebx
          call   g
          halt
0x011: 90           |          ret
0x012: 50184c000000 | g:       mrmovl n, %ecx
          jle EndLoop
          writeln %ecx
          irmovl $1, %edx
          subl %edx, %ecx
          rmmovl %ecx, n
          jmp   Loop
EndLoop: ret
n:       .long 2
```
Simple recursion example

```c
int n = 3;
void f();

int main() {
    f();
    return 0;
}

void f() {
    if (n > 0) {
        printf("%d", n--);
        f();
    }
}
```

Local automatic variables

- These are also kept on the stack
- Multiple versions of the "same" variable can be kept; think of local variables in a recursive function
- We organize the stack into frames - one frame exists for each active (not yet returned) function call
  - We use the register %ebp to point to the first element in the current frame (the frame pointer)

Stack frames

```
<table>
<thead>
<tr>
<th>caller's stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>current stack frame</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>saved %ebp</td>
</tr>
<tr>
<td>current frame's registers, local vars, and other stack data</td>
</tr>
<tr>
<td>earlier frames</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>main:</th>
<th>irmovl $0x1000, %esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>f:</td>
<td>mrmovl n, %ecx</td>
</tr>
<tr>
<td></td>
<td>irmovl $0, %eax</td>
</tr>
<tr>
<td></td>
<td>addl %eax, %ecx</td>
</tr>
<tr>
<td>jle</td>
<td>EndIf</td>
</tr>
<tr>
<td>wrnt</td>
<td>%ecx</td>
</tr>
<tr>
<td>irmovl</td>
<td>$1, %eax</td>
</tr>
<tr>
<td>subl</td>
<td>%eax, %ecx</td>
</tr>
<tr>
<td>rmmovl</td>
<td>%ecx, n</td>
</tr>
<tr>
<td>call</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>EndIf:</td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td>.align 4</td>
</tr>
<tr>
<td>n:</td>
<td>.long 3</td>
</tr>
</tbody>
</table>
```
Leaving a function

- x86 has a `leave` instruction, to make things easier for assembly programmers just before a return
- The equivalent Y86 code is:
  - `rrmovl %ebp, %esp # give up current frame`
  - `popl %ebp # restore caller's frame pointer`
- This puts the stack pointer in the correct position so that a subsequent `ret` instruction will return to the correct location

Storing local variables

- These are stored at the top of the stack (nearest to `%esp`)
- Inside the current frame:
  ```c
  void f() {
      int var1, var2, var3;
      ...
  }
  ```
- Accessed by offsets from `%esp`

Local variable example

```c
void f();
int main() {     
  main:      
    irmovl $0x1000, %esp # init stack ptr
    call f
    call f
    halt
    pushl %ebp # save old frame ptr
    rrmovl %esp, %ebp # set new frame ptr
    irmovl $16, %eax # sub 16 (4 ints)
    subl %eax, %esp # from stack ptr
    irmovl $30, %eax # a = 30
    rmovl %eax, 12(%esp) # store a in stack
    irmovl $25, %eax # b = 25
    rmovl %eax, 8(%esp) # store b in stack
    rmovl 12(%esp), %eax # load a
    rmovl 8(%esp), %ecx # load b
    addl %eax, %ecx # c = a + b
    rmovl %ecx, 4(%esp) # store c in stack
    rmovl 4(%esp), %eax # load c
    writeln %eax # print c
    irmovl $300, %eax # d = 300
    rmovl %eax, 0(%esp) # store d in stack
    rmovl %ebp, %esp # reset stack ptr
    popl %ebp # restore old frame
    ret
  }
}
```

Passing parameters

- Similar to local variables, but are pushed on the stack in the calling frame
- Accessed by offsets of `%ebp`; a function has to reach into the calling frame to access these
- If you're just passing one or two parameters, you may be able to just store the values in registers (faster than stack storage), but we'll just cover stack usage here
<table>
<thead>
<tr>
<th>Parameter example</th>
</tr>
</thead>
<tbody>
<tr>
<td>void f(int);</td>
</tr>
<tr>
<td>int main() {</td>
</tr>
<tr>
<td>f(72);</td>
</tr>
<tr>
<td>printf(&quot;\n&quot;);</td>
</tr>
<tr>
<td>return 0;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void f(int i) {</td>
</tr>
<tr>
<td>printf(&quot;%d&quot;, i);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>main:</td>
</tr>
<tr>
<td>irmovl $0x1000, %esp # init stack ptr</td>
</tr>
<tr>
<td>irmovl $72, %eax  # load arg to reg</td>
</tr>
<tr>
<td>pushl %eax       # push arg on stack</td>
</tr>
<tr>
<td>call f           # printf(&quot;\n&quot;);</td>
</tr>
<tr>
<td>irmovl $10, %eax # printf(&quot;\n&quot;);</td>
</tr>
<tr>
<td>wrch %eax        # return 0;</td>
</tr>
<tr>
<td>halt             # return 0;</td>
</tr>
<tr>
<td>f:</td>
</tr>
<tr>
<td>pushl %ebp       # save old frame ptr</td>
</tr>
<tr>
<td>rrmovl %ebp, %ebp # set new frame ptr</td>
</tr>
<tr>
<td>rmovl 8(%ebp), %eax # load i param</td>
</tr>
<tr>
<td>wrint %eax       # print i</td>
</tr>
<tr>
<td>rrmovl %ebp, %esp # reset stack ptr</td>
</tr>
<tr>
<td>popl %ebp        # reset stack ptr</td>
</tr>
<tr>
<td>ret              # restore frame ptr</td>
</tr>
</tbody>
</table>