CMSC 216
Introduction to Computer Systems
Lecture 14
Assembly Language, cont.
Administrivia

• Continue reading Bryant and O’Hallaron Section 4.1 (Y86 subset) and Chapter 3, for more info on IA-32 instruction set architecture
Section 3.7, Bryant and O'Hallaron

PROCEDURE IMPLEMENTATION
Reminder: Y86 stack instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl R</td>
<td>Reg[%esp] ← Reg[%esp] - 4; Mem[Reg[%esp]] ← Reg[R]</td>
<td>Push on to stack</td>
</tr>
<tr>
<td>popl R</td>
<td>Reg[R] ← Mem[Reg[%esp]]; Reg[%esp] ← Reg[%esp] + 4</td>
<td>Pop off of stack</td>
</tr>
</tbody>
</table>

- **pushl** and **popl** provide quick ways to work with the program stack
- Without **pushl**, a push operation would look like this (assuming the R above is %eax):
  - `irmovl $4, %esi`
  - `subl %esi, %esp`
  - `rmmovl %eax, (%esp)`
### 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</td>
<td>Call function</td>
</tr>
<tr>
<td></td>
<td>PC ← Label</td>
<td></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 by getting the value that was pushed in the call
- functions are responsible for ensuring the stack is properly adjusted before returning (or you won’t get the same value back during the return that you pushed on in the call)
Quick example

• What is the value of \%eax after this sequence of instructions?

  call next

  next: popl \%eax

• Whatever the address of next is
  (the memory address where popl instruction is)

• Note: this is NOT the PC, but it is related to it
  – PC is the address of the next instruction to be executed; \%eax now has address of most recently executed instruction (the popl)
Implementing functions

• When calling a function, the computer needs to know where execution will resume once the function returns (the *return address*)

• The *call* instruction stores this address on the stack; when the called function (the *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

- `%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:
  
  ```
  irmovl $0x1000, %esp
  ```

- Then, it can push and pop as necessary
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--);
}

main:           irmovl $0x1000, %esp
                call   f
                irmovl $1, %ebx
                rmmovl %ebx, n
                call   g
                halt
                f:           call   g
                ret
                g:           mrmovl n, %ecx
                      Loop:  irmovl $0, %eax
                              addl   %eax, %ecx
                              jle  EndLoop
                              wrint  %ecx
                              irmovl $1, %edx
                              subl   %edx, %ecx
                              rmmovl %ecx, n
                              jmp   Loop
                EndLoop:  ret
                      .align 4
                n:       .long 2
Simple function call example

```
0x000: 308400100000 | main:    irmovl $0x1000, %esp
0x006: 801d000000   | call    f
0x00b: 308301000000 | irmovl $1, %ebx
0x011: 40384c000000 | rmmovl %ebx, n
0x017: 8023000000   | call    g
0x01c: 10           | halt
0x01d: 8023000000   | f:      call    g
0x022: 90           | ret
0x023: 50184c000000 | g:      mrmovl n, %ecx
0x029: 308000000000 | Loop:   irmovl $0, %eax
0x02f: 6001         | addl    %eax, %ecx
0x031: 714b000000   | jle     EndLoop
0x036: f318         | wrint   %ecx
0x038: 308201000000 | irmovl $1, %edx
0x03e: 6121         | subl    %edx, %ecx
0x040: 40184c000000 | rmmovl %ecx, n
0x046: 7029000000   | jmp     Loop
0x04b: 90           | EndLoop: ret
0x04c:              | .align 4
0x04c: 02000000     | n:      .long 2
```

CMSC 216 - Wood, Sussman, Herman, Plane
Simple recursion example

```c
int n = 3;

void f();

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

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

```asm
main:  irmovl $0x1000, %esp
       call    f
       halt

f:     mrmovl n, %ecx
       irmovl $0, %eax
       addl   %eax, %ecx
       jle    EndIf
       wrint  %ecx
       irmovl $1, %eax
       subl   %eax, %ecx
       rmmovl %ecx, n
       call    f

EndIf:  ret

.align 4

n:      .long 3
```
# Simple recursion example

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Value/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000: 308400100000</td>
<td>irmovl</td>
<td>$0x1000, %esp</td>
<td>main: call f</td>
</tr>
<tr>
<td>0x006: 800c000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00b: 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00c: 501838000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x012: 308000000000</td>
<td>irmovl</td>
<td>$0, %eax</td>
<td>f:</td>
</tr>
<tr>
<td>0x018: 6001</td>
<td></td>
<td></td>
<td>addl %eax, %ecx</td>
</tr>
<tr>
<td>0x01a: 7134000000</td>
<td></td>
<td>jle EndIf</td>
<td></td>
</tr>
<tr>
<td>0x01f: f318</td>
<td></td>
<td>wrint %ecx</td>
<td></td>
</tr>
<tr>
<td>0x021: 308001000000</td>
<td>irmovl</td>
<td>$1, %eax</td>
<td></td>
</tr>
<tr>
<td>0x027: 6101</td>
<td></td>
<td>subl %eax, %ecx</td>
<td></td>
</tr>
<tr>
<td>0x029: 401838000000</td>
<td>rmmovl</td>
<td>%ecx, n</td>
<td></td>
</tr>
<tr>
<td>0x02f: 800c000000</td>
<td></td>
<td></td>
<td>call f</td>
</tr>
<tr>
<td>0x034: 90</td>
<td></td>
<td></td>
<td>EndIf: ret</td>
</tr>
<tr>
<td>0x038:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x038: 0300000000</td>
<td></td>
<td>n:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.align 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.long 3</td>
<td></td>
</tr>
</tbody>
</table>
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

- Caller's stack frame
- Current stack frame

- Earlier frames
  - Caller's local variables and other stack data
  - Return address
  - Saved %ebp
  - Current frame's registers, local vars, and other stack data

%ebp
%esp
 Leaving a function

• x86 has a **leave** instruction, to make things easier for assembly programmers just before a return.

**Y86 does not have the leave instruction.**

• 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() {
    f();
    f();
    return 0;
}

void f() {
    int a, b, c, d;
    a = 30;
    b = 25;
    c = a + b;
    printf("%d", c);
    d = 300;
}
```
Passing parameters

• Similar to local variables, but are pushed on the stack in the caller’s 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
Parameter example

```c
void f(int);

int main() {
    f(72);
    printf("\n");
    return 0;
}

void f(int i) {
    printf("%d", i);
}
```

```asm
main:  irmovl $0x1000, %esp  # init stack ptr
       irmovl $72, %eax   # load arg to reg
       pushl %eax        # push arg on stack
       call f
       irmovl $10, %eax  # printf("\n");
       wrch %eax
       halt              # return 0;

f:    pushl %ebp        # save old frame ptr
       rrmovl %esp, %ebp # set new frame ptr
       mrmovl 8(%ebp), %eax # load i param
       wrint %eax         # print i
       rrmovl %ebp, %esp  # reset stack ptr
       popl %ebp          # restore frame ptr
       ret
```
Recursive parameter example

```c
void f(int);

int main() {
    f(5);
    printf("\n");
    return 0;
}

void f(int i) {
    if (i > 0)
        f(i-1);
    printf("%d", i);
}
```
Register usage conventions

• Notice in the last example we had to reload the value of the parameter \( i \) after the recursive function call, because recursive calls to \( f \) would destroy the value in \( %eax \)

• A convention has developed as to which registers can be used by a procedure without destroying data from the caller

• When P calls Q...
  – Q can do anything it wants with the caller save registers; P is required to store these before the call if there's important data there
  – Q must save the values of the callee save registers, and restore them before returning (if Q uses these registers); P can use these to maintain data across function calls

• For reference (i.e., don't memorize this), and by convention, \( %eax, %edx, \) and \( %ecx \) are caller save, while \( %ebx, %esi, \) and \( %edi \) are callee save
Using callee save registers

```
f:    pushl  %ebp
     rrmovl  %esp, %ebp

     mrmovl  8(%ebp), %eax
     irmovl $0, %ecx
     addl  %ecx, %eax
     jle  EndIf
     irmovl $-1, %ecx
     addl  %eax, %ecx
     pushl  %ecx
     call  f

EndIf: mrmovl  8(%ebp), %eax
     writ %eax
     rrmovl  %ebp, %esp
     popl  %ebp
     ret

f:    pushl  %ebp
     rrmovl  %esp, %ebp
     pushl  %ebx

     mrmovl  8(%ebp), %ebx
     irmovl $0, %ecx
     addl  %ecx, %ebx
     jle  EndIf
     irmovl $-1, %ecx
     addl  %ebx, %ecx
     pushl  %ecx
     call  f
     popl  %ecx

EndIf: writ %ebx
     popl  %ebx
     rrmovl  %ebp, %esp
     popl  %ebp
     ret
```

# store callee save
# holds value of i
# pop arg off stack
# no need for reload
# restore val of reg
Returning values

• In Y86 (and x86), the convention is to store return values in $eax, and have the caller read this value after the call returns

• This works well enough for integers, but what about something like structs?
  – If the struct has $\leq 3$ values, you could use all the caller save registers to pass the value back
  – The struct could be placed in the stack and accessed immediately after the function call
Example of returning values

```c
int f(int);

int main() {
    static int n;
    n = f(5);
    printf("%d", n);
    printf("\n");
    return 0;
}

int f(int i) {
    return (i+1) * 3;
}
```

```asm
main:  irmovl $0x1000, %esp     # init stack ptr
       irmovl $5, %eax         # load arg to reg
       pushl %eax              # push arg on stack
       call f                  # store return value
       rmmovl %eax, n          # load value of n
       mrmovl n, %eax          # load value of n
       wrint %eax              # printf("\n");
       irmovl $10, %eax        # printf("\n");
       wrch %eax               # return 0;
       halt                    # return 0;

f:     pushl %ebp             # save old frame ptr
       rrmovl %esp, %ebp       # set new frame ptr
       mrmovl 8(%ebp), %eax   # load param i
       irmovl $1, %ecx        # m + 1
       addl %ecx, %eax        # add 1
       irmovl $3, %ecx        # for * 3
       multl %ecx, %eax       # multiply by 3
       rrmovl %ebp, %esp      # reset stack ptr
       popl %ebp               # restore frame ptr
       ret

.nalign 4
n:      .long 0
```