CMSC 216
Introduction to Computer Systems
Lecture 15
Assembly Language, cont.
Larry Herman & Alan Sussman

Notes

- Project 4 due Monday
  - project 3 secret tests posted – fix your problems with project 3!
  - public tests posted very soon
- Read Bryant and O’Hallaron Section 4.1 (Y86 subset) and Chapter 3, for more info on IA-32 instruction set architecture

A full assembly

- This is the result of running the Y86 assembler on the example assembler source code:

```
0x000: 308000000000 | irmovl $0,%eax    # sum = 0
0x006: 308101000000 | irmovl $1,%ecx    # num = 1
0x00c: 6010         | Loop: addl %ecx,%eax # sum += num
0x00e: 308201000000 | irmovl $1,%edx    # tmp = 1
0x014: 6021         | addl %edx,%ecx # num++
0x016: 3082e8030000 | irmovl $1000,%edx # lim = 1000
0x01c: 6112         | subl %ecx,%edx # if lim - num >= 0
0x01e: 750c000000   | jge Loop     # loop again
0x023: f308         | wrint %eax     # printf("%d", sum)
0x025: 30820a000000 | irmovl $10,%edx # ch = \n'
0x02b: f128         | wrch %edx      # printf("%c", ch)
0x02d: 10           | halt
```
Y86 program state

- $2^{12}$ bytes of memory
- You can set the stack to start somewhere other than 0x1000, but you have to explicitly set it

![](image)

Working with Y86

- Source code is usually stored in *.ys files
- On the Grace systems, there will be two programs available in ~/216public/bin for working with Y86 programs
  - **yas** is the Y86 assembler, which creates *.yo files
    - Run like this: `yas prog.ys`
  - **yis** is the Y86 simulator, which operates on *.yo files
    - Run like this: `yis prog.yo`

Y86 data movement instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>irmovl V,R</td>
<td>Reg[R] ← V</td>
<td>Immediate-to-register move</td>
</tr>
<tr>
<td>rrmovl rA,rB</td>
<td>Reg[rB] ← Reg[rA]</td>
<td>Register-to-register move</td>
</tr>
</tbody>
</table>

- **irmovl** is used to place known numeric values (labels or numeric literals) into registers
- **rrmovl** copies a value between registers
- **rmmovl** stores a word in memory
- **mrmovl** loads a word from memory
- **rmovl** and **mrmovl** are the only instructions that access memory - Y86 is a load/store architecture

Examples of data movement

```
irmovl $55,%edx     # d = 55
rrmovl %edx,%ebx    # b = d
irmovl Array,%eax  # a = Array
rmmovl %ebx,4(%eax) # a[1] = 55
mrmovl 0(%eax),%ecx # c = a[0]
halt
 .align 4
Array:
   .long 0x6f
   .long 0x84
```
Data movement example, cont.

- **Assembler output:**
  
  ```
  0x000: 308237000000 | irmovl $55,%edx  # d = 55
  0x006: 2023         | rrmovl %edx,%ebx  # b = d
  0x008: 30801c000000 | irmovl Array,%eax # a = Array
  0x00e: 403004000000 | rmmovl %ebx,4(%eax) # a[1] = 55
  0x014: 501000000000 | mrmovl 0(%eax),%ecx # c = a[0]
  0x01a: 10           | halt
  ```

- **Simulator output:**

  Stopped in 6 steps at PC = 0x1b. Exception 'HLT', CC Z=1 S=0 O=0
  Changes to registers:
  
  ```
  %eax: 0x00000000 0x0000001c
  %ecx: 0x00000000 0x0000000a
  %edx: 0x00000000 0x00000037
  %ebx: 0x00000000 0x00000037
  ```

  Changes to memory:
  
  ```
  0x0020: 0x00000084 0x00000037
  ```

Y86 input/output instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdch R</td>
<td>scanf(&quot;%c&quot;, &amp;Reg[R])</td>
<td>Read character</td>
</tr>
<tr>
<td>rdint R</td>
<td>scanf(&quot;%d&quot;, &amp;Reg[R])</td>
<td>Read integer</td>
</tr>
<tr>
<td>wrch R</td>
<td>printf(&quot;%c&quot;, Reg[R])</td>
<td>Write character</td>
</tr>
<tr>
<td>writ R</td>
<td>printf(&quot;%d&quot;, Reg[R])</td>
<td>Write integer</td>
</tr>
</tbody>
</table>

- All these instructions are extensions to Y86 we've added to the ones in the book
- These are what allow you to interact with the simulator and write more interesting programs

I/O example

- **Assembler output:**

  ```
  0x000: f208 | rdint %eax  # a = 65 (via scanf())
  0x002: f038 | rdch %ebx  # b = 'B' (via scanf())
  0x004: f308 | wrint %eax  # printf("%d", a)
  0x006: f108 | wrch %eax  # printf("%c", a)
  0x008: f338 | wrint %ebx  # printf("%d", b)
  0x00a: f138 | wrch %ebx  # printf("%c", b)
  0x00c: 30810a000000 | irmovl $10,%ecx  # c = 10
  0x010: f118 | wrch %ecx  # printf("%c", c)
  0x014: 10 | halt
  ```

- **Simulator run:**

  ```
  echo 65B | yis io.yo
  65A66B
  ```

  Stopped in 9 steps at PC = 0x15. Exception 'HLT', CC Z=1 S=0 O=0
  Changes to registers:
  
  ```
  %eax: 0x00000000 0x0000006f
  %ecx: 0x00000000 0x0000000a
  %ebx: 0x00000000 0x00000037
  ```

  Changes to memory:
  
  ```
  0x0020: 0x00000084 0x00000037
  ```

Y86 integer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subl S,D</td>
<td>Reg[D] ← Reg[D] - Reg[S]</td>
<td>Subtract</td>
</tr>
<tr>
<td>divl S,D</td>
<td>Reg[D] ← Reg[D] / Reg[S]</td>
<td>Integer division*</td>
</tr>
<tr>
<td>modl S,D</td>
<td>Reg[D] ← Reg[D] % Reg[S]</td>
<td>Remainder*</td>
</tr>
</tbody>
</table>

- All these instructions operate on two integers, and set the condition code flags appropriately
- Instructions marked with an asterisk (*) are extensions to Y86 we've added to the ones in the book
**Integer instruction example**

- **Assembler output:**
  - 0x000: 308003000000 | irmovl $3,%eax  # a = 3
  - 0x006: 308305000000 | irmovl $5,%ebx  # b = 5
  - 0x00c: 6003         | addl %eax,%ebx  # b = a + b
  - 0x00e: f308         | addl %eax,%ebx
  - 0x010: 308620000000 | irmovl $32,%esi  # 32 == ' ' 
  - 0x016: f168         | wrch %esi
  - 0x018: f338         | wrch %ebx
  - 0x01a: 30860a000000 | irmovl $10,%esi  # 10 == '
'
  - 0x020: f168         | wrch %esi
  - 0x022: 10           | halt

- **Simulator run:**
  - 3 8
  ... 

- Notice these instructions are destructive; they overwrite the second operand
  - Need to make copies if you need old values

**Condition codes**

- Performing integer operations causes various flags to be set, describing the attributes of the result of the operation
- These are used by other, subsequent instructions to perform conditional branching
- The three we are concerned with are:
  - OF: overflow flag; did the operation overflow?
  - SF: sign flag; is the result negative?
  - ZF: zero flag; is the result zero?

**Branch instructions**

- These are used to perform the effect of if statements, loops, and switches
- When encountered, if a certain condition is true, control flow will then go to the address specified, rather than advancing to the next instruction
  - The address of the next instruction to be executed is held in the program counter; in many architectures, this is held in an accessible register (not so with Y86).

**Y86 branch instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Branch if...</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp Label</td>
<td>1</td>
<td>Unconditional jump</td>
</tr>
<tr>
<td>jle Label</td>
<td>(SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>jl Label</td>
<td>SF ^ OF</td>
<td>Jump if less</td>
</tr>
<tr>
<td>je Label</td>
<td>ZF</td>
<td>Jump if equal</td>
</tr>
<tr>
<td>jne Label</td>
<td>~ZF</td>
<td>Jump if not equal</td>
</tr>
<tr>
<td>jge Label</td>
<td>~(SF ^ OF)</td>
<td>Jump if greater or equal</td>
</tr>
<tr>
<td>jg Label</td>
<td>~(SF ^ OF) &amp; ~ZF</td>
<td>Jump if greater</td>
</tr>
</tbody>
</table>

- Each instruction relies on the condition codes set by the most recent integer instruction
Branch example 1

- Assembler output:

0x000: f208 | rdint %eax
0x002: 308700000000 | irmovl $0,%edi # consistent zero
0x008: 308600000000 | irmovl $0,%esi # sum = 0
0x00e: 6070 | addl %edi,%eax
0x010: 732000000000 | je EndLoop
0x015: 6006 | Loop: addl %eax,%esi # sum += n
0x017: f208 | rdint %eax
0x019: 6070 | addl %edi,%eax
0x01b: 741500000000 | jne Loop
0x020: f368 | EndLoop: wrint %esi
0x022: 30830a000000 | irmovl $10,%ebx
0x028: f13b | wrch %ebx
0x02a: 10 | halt

- Simulator output:

$ echo 1 4 9 16 25 0 | yis io.yo
55
Stopped in 29 steps at PC = 0x2b. Exception 'HLT', CC Z=1 S=0 O=0
...

CMSC 216 - Wood, Sussman, Herman