Today

- Quiz 5
- Addressing Modes
- Branching (basics)

Assignments

- HW6 out (later today)
Memory References in Assembly

- Recall that memory is treated like a **large array of bytes**
- An address computed from an operand is called the “**effective address**”

**x86 “Addressing Modes”** (where \textit{Imm} means “immediate” value)

(\%rax) \hspace{1cm} \text{indirect address (“go to” mem location in \%rax)}

(\%rax,\%rdx) \hspace{1cm} \text{base + index … (\%rax + \%rdx)}

- useful for indexing arrays of bytes (e.g., strings)
- e.g., for array[4] where array address in \%rax and index in \%rdx

(\%rax,\%rdx,s) \hspace{1cm} \text{base + index \times s … (\%rax + \%rdx*s)}

- where s is 1, 2, 4 or 8
- for indexing arrays of items larger than bytes (e.g., array of 4-byte ints)

\textit{Imm}(…) \hspace{1cm} \text{adds an offset to any of the above forms}

- 1(\%rax) computes effective address (1 + \%rax)
- 8(\%rax, \%rdx) computes effective address (8 + \%rax + \%rdx)
- etc
Some “limits” of addressing in x86-64

- With full 64-bit addresses, 18,446,744,073,709,551,615 addresses
- Current machines only allow 48-bit memory addresses
- Even with 48-bit addresses, this is 281,474,976,710,656 addresses
- Which is \( \approx 281 \) terabytes, which is \( \approx 281,000 \) gigabytes
Pointers

- addressing modes support the notion of “pointers” (like in C/C++)
- also support arrays and array access (which are really just pointers)

Basic C-style pointer example: (* Notional (not actual compiled code)

<table>
<thead>
<tr>
<th>C code snippet</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 42;</td>
<td></td>
</tr>
<tr>
<td>int* xptr = &amp;x;</td>
<td></td>
</tr>
<tr>
<td>*xptr = 27;</td>
<td></td>
</tr>
<tr>
<td>int* yptr = xptr;</td>
<td></td>
</tr>
<tr>
<td>*yptr = 42;</td>
<td></td>
</tr>
</tbody>
</table>

... (*) Notional (not actual compiled code)

Basic C-style array example: (* Notional (not actual compiled code)

<table>
<thead>
<tr>
<th>C code snippet</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>int array[] = {20,30,10,40,50};</td>
<td></td>
</tr>
<tr>
<td>int v2 = array[1];</td>
<td></td>
</tr>
<tr>
<td>int v3 = *(array + 2);</td>
<td></td>
</tr>
<tr>
<td>int* ptr = array;</td>
<td></td>
</tr>
<tr>
<td>++ptr;</td>
<td></td>
</tr>
<tr>
<td>*ptr = 60;</td>
<td></td>
</tr>
</tbody>
</table>

... (*) Notional (not actual compiled code)
The `lea` instruction (load effective address)

```
lea src, dst  \  ... dst = &src

- src is an effective address / memory reference (e.g., 4(%rax)
- dst contains the computed address (not the value at that address!)

• example:

  leaq 4(%rax), %rbx  # rbx = 4 bytes beyond address in %rax
  addq $32, (%rbx)    # add 32 to value "pointed to" by %rbx
```
Conditional Statements

Recall condition codes (eflags/rflags register):

- **ZF**: Zero Flag – most recent a/l (arithmetic/logic) instruction yielded zero
- **SF**: Sign Flag – most recent a/l instruction yielded negative value
- **OF**: Overflow Flag – most recent a/l instruction caused 2’s-comp overflow
- **CF**: Carry Flag – most recent a/l instruction had a carry out or borrow

Note that CF is only useful for unsigned values

**cmp** instruction

```
  cmps src_1, src_2 ...
```

- after subtracting, just sets the above condition codes
- i.e., same as **sub**, but nothing stored (only condition code changes)
- example:

```
  movl $42, %eax
  movl $27, %ebx
  cmp  %eax, %ebx  # 27 - 42 = -15 ... eg, sets SF (negative)
```
Jump instructions

jmp Label
• simplest form of the jmp instruction
• unconditionally jumps to location at label (sets IP to label address)
• Example:

```
xorl %eax, %eax
L1: incl %eax
    jmp L1           # infinite loop!
```

je Label
• Jump to address at label if zero flag set
• Example:

```
cmpl %eax, %ebx    # set flags
je  L1             # jump if zero flag (ZF)
jmp  L2            # unconditional jump
```

L1: do something ...
L2: do something else ...
Other variants:

- `jne Label` ... not equal (not ZF flag set)
- `js Label` ... < 0 (SF flag set)
- `jns Label` ... > 0 (SF flag not set)
- `jl Label` ... a < b via cmp (signed)
- `jle Label` ... a <= b via cmp (signed)
- `jg Label` ... a > b via cmp (signed)
- `jge Label` ... a >= b via cmp (signed)
- `jb Label` ... a > b via cmp (unsigned)
- `jbe Label` ... a <= b via cmp (unsigned)
- `ja Label` ... a > b via cmp (unsigned)
- `jae Label` ... a >= b via cmp (unsigned)