Representation of Information

CS61, Lecture 2
Prof. Stephen Chong
September 6, 2011
Announcements

• Assignment 1 released
  • Posted on http://cs61.seas.harvard.edu/
  • Due one week from today, **Tuesday 13 Sept**
  • First question (survey) due 5:00pm **Thursday 8 Sept**
  • Contains C self assessment
    • If you are not comfortable with all of the questions, may need to spend time getting up to speed with C

• Name tags
  • At back of room
  • Fill in, put in front of you, leave at end of class

• Sections will start next week
  • Section times and signup will be later this week

• Highscore binary
Topics for today

- Representing information
  - Hexadecimal notation
  - Representing integers
  - Storing information
    - Word size, data size
    - Byte ordering
  - Representing strings
  - Representing code
- Basic processor operation
Information

• Computers represent information using **bits**
  • 2-valued signals; binary digits
  • All kinds of information
    • Numbers, memory addresses, instructions, strings, …

• What do the bits `1100 0011` represent?
  • Could be (unsigned) integer 195
  • Could be (signed) integer -61
  • Could be instruction `ret`
  • Depends on context!

• Information is bits plus **context**
  • context = way of interpreting data
Computers

- Computers store bits in memory
- Information stored in memory is both instructions and data
  - But remember, instructions and data are just bits that get interpreted differently!
Computers

• Rather than accessing individual bits, most computers use blocks of 8 bits, called **bytes**
• View memory as a very large array of bytes
  • Memory addresses are another kind of data
Hexadecimal notation

- To make it easier to read bits, we use hexadecimal notation.
- Decimal notation is base 10:
  - Uses digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
  - $xyz$ represents number $x \times 10^2 + y \times 10^1 + z \times 10^0$.
- E.g.,

\[
3874 = 3 \times 10^3 + 8 \times 10^2 + 7 \times 10^1 + 4 \times 10^0
\]

\[
= 3000 + 800 + 70 + 4
\]
Binary notation

- Binary notation is base 2
  - Uses digits 0, 1
  - $xyz$ represents number $x \times 2^2 + y \times 2^1 + z \times 2^0$
  - E.g.

\[
1101
\]

\[
\begin{align*}
1 \times 2^3 &= 8 \\
1 \times 2^2 &= 4 \\
0 \times 2^1 &= 0 \\
1 \times 2^0 &= 1 \\
\end{align*}
\]

\[8 + 4 + 0 + 1 = 13\]
Hexadecimal notation

- Hexadecimal notation is base 16
  - Use digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - \( \text{xyz} \) represents number \( x \times 16^2 + y \times 16^1 + z \times 16^0 \)
  - E.g.

\[
7 \times 16^3 = 7 \times 4096 = 28672
\]
\[
13 \times 16^2 = 13 \times 256 = 3328
\]
\[
11 \times 16^1 = 176
\]
\[
2 \times 16^0 = 2
\]

\[28672 + 3328 + 176 + 2 = 32178\]
Hexadecimal notation

• Why is hexadecimal notation useful for computer programming?

• One hexadecimal digit represents 4 bits

• One byte is two hexadecimal digits
  • E.g., 0x8C = 10001100

• In C (and in this class) we prefix hexadecimal numbers with “0x”
  • E.g., 0x5A = 01011010 = 90 = 5×16 + 10
  • E.g., 0x42 = 01000010 = 66 = 4×16 + 2

• You will get comfortable and familiar with hexadecimal notation.

<table>
<thead>
<tr>
<th>Binary value</th>
<th>Dec. value</th>
<th>Hex. digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>
Other bases

• Why hexadecimal?
  • Base 16, so corresponds to 4 bits
  • More compact than binary, but 16 possible values small enough to be understandable

• Octal notation also common
  • Corresponds to 3 bits

• C notation
  • Leading 0 (zero) on integer constant means octal
  • Leading 0x on integer constant means hexadecimal
  • E.g., \(31 = 037 = 0x1f\)
Base 64

- An ASCII representation of binary data

<table>
<thead>
<tr>
<th>Value</th>
<th>Char</th>
<th>Value</th>
<th>Char</th>
<th>Value</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>16</td>
<td>Q</td>
<td>32</td>
<td>g</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>17</td>
<td>R</td>
<td>33</td>
<td>h</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>18</td>
<td>S</td>
<td>34</td>
<td>i</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>19</td>
<td>T</td>
<td>35</td>
<td>j</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>20</td>
<td>U</td>
<td>36</td>
<td>k</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>21</td>
<td>V</td>
<td>37</td>
<td>l</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>22</td>
<td>W</td>
<td>38</td>
<td>m</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
<td>23</td>
<td>X</td>
<td>39</td>
<td>n</td>
</tr>
<tr>
<td>8</td>
<td>I</td>
<td>24</td>
<td>Y</td>
<td>40</td>
<td>o</td>
</tr>
<tr>
<td>9</td>
<td>J</td>
<td>25</td>
<td>Z</td>
<td>41</td>
<td>p</td>
</tr>
<tr>
<td>10</td>
<td>K</td>
<td>26</td>
<td>a</td>
<td>42</td>
<td>q</td>
</tr>
<tr>
<td>11</td>
<td>L</td>
<td>27</td>
<td>b</td>
<td>43</td>
<td>r</td>
</tr>
<tr>
<td>12</td>
<td>M</td>
<td>28</td>
<td>c</td>
<td>44</td>
<td>s</td>
</tr>
<tr>
<td>13</td>
<td>N</td>
<td>29</td>
<td>d</td>
<td>45</td>
<td>t</td>
</tr>
<tr>
<td>14</td>
<td>O</td>
<td>30</td>
<td>e</td>
<td>46</td>
<td>u</td>
</tr>
<tr>
<td>15</td>
<td>P</td>
<td>31</td>
<td>f</td>
<td>47</td>
<td>v</td>
</tr>
<tr>
<td>16</td>
<td>R</td>
<td>32</td>
<td>s</td>
<td>48</td>
<td>w</td>
</tr>
<tr>
<td>17</td>
<td>S</td>
<td>33</td>
<td>h</td>
<td>49</td>
<td>x</td>
</tr>
<tr>
<td>18</td>
<td>T</td>
<td>34</td>
<td>i</td>
<td>50</td>
<td>y</td>
</tr>
<tr>
<td>19</td>
<td>U</td>
<td>35</td>
<td>j</td>
<td>51</td>
<td>z</td>
</tr>
<tr>
<td>20</td>
<td>V</td>
<td>36</td>
<td>k</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>W</td>
<td>37</td>
<td>l</td>
<td>53</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>X</td>
<td>38</td>
<td>m</td>
<td>54</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>Y</td>
<td>39</td>
<td>n</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>Z</td>
<td>40</td>
<td>o</td>
<td>56</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>a</td>
<td>41</td>
<td>p</td>
<td>57</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>b</td>
<td>42</td>
<td>q</td>
<td>58</td>
<td>6</td>
</tr>
<tr>
<td>27</td>
<td>c</td>
<td>43</td>
<td>r</td>
<td>59</td>
<td>7</td>
</tr>
<tr>
<td>28</td>
<td>d</td>
<td>44</td>
<td>s</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>29</td>
<td>e</td>
<td>45</td>
<td>t</td>
<td>61</td>
<td>9</td>
</tr>
<tr>
<td>30</td>
<td>f</td>
<td>46</td>
<td>u</td>
<td>62</td>
<td>+</td>
</tr>
<tr>
<td>31</td>
<td>g</td>
<td>47</td>
<td>v</td>
<td>63</td>
<td>/</td>
</tr>
</tbody>
</table>

Message-ID: <4E614710.8010700@seas.harvard.edu>
Date: Fri, 2 Sep 2011 17:13:52 -0400
From: John Harvard <jharvard@college.harvard.edu>
To: Stephen Chong <chong@seas.harvard.edu>
Subject: Homework
Content-Type: multipart/mixed; boundary="--------07060310202000801030000"
Return-Path: jharvard@college.harvard.edu
X-Originating-IP: [10.243.39.38]
MIME-Version: 1.0

----------07060310202000801030000
Content-Type: text/plain; charset="ISO-8859-1"; format=flowed
Content-Transfer-Encoding: 7bit

Hi Prof. Chong,
Attached is my homework. Sorry it's late.

----------07060310202000801030000
Content-Type: application/octet-stream; name="hw1.pdf"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="hw1.pdf"

JVBERi0xLjMKJcTl8uXrp/Og0MTGcjgMCMVymOKPdwgLxlbmd0aCA1IDAgUlAvRmlsdGVyIG9G
bG9IcXJldGlvZS9vZ3NzcyBhbmQgZm9yIHRoZSBzdHJlYW0gY29kZSByZXRl
MjBjYXQgYnkgc2ZvcmQgY29kZSB3aXRoIG1lZGltcw==
Representing integers

- Given \( n \) bits to store an integer, we can represent \( 2^n \) different values.
- If we just care about non-negative (aka unsigned) integers, we can easily store the values
  \( 0, 1, 2, \ldots, 2^n - 1 \).
  - E.g., for 4 bits
  - \( 0x2 = 2 \)
  - \( 0xB = 11 \)
  - \( 0xF = 15 = 2^4 - 1 \)
Integer overflow

- With $n$ bits, we can represent values $0, 1, 2, \ldots, 2^n-1$
- **Overflow** occurs when we have a result that doesn’t fit in the $n$ bits
  - E.g., using 4 bits: $0xF + 0x1$

\[
\begin{align*}
0xF & = \begin{array}{c|c|c|c|c}
1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 \\
\hline
1 & 0 & 0 & 0 & 0
\end{array} \\
0x1 & = \begin{array}{c|c|c|c|c}
0 & 0 & 0 & 0 & 0
\end{array}
\end{align*}
\]

\[
0xF + 0x1 = 0x0 \quad \text{Overflow}!!
\]
Integer overflow

$Add_4(u, v)$
Integer overflow

$U\text{Add}_4(u, v)$
Representing negative integers

- Have seen how to represent **unsigned integers** (i.e., non-negative integers)
- How do we represent negative integers?
- Three common encodings:
  - Sign and magnitude
  - Ones’ complement
  - Two’s complement
Sign and magnitude

- Use one bit to represent sign, remaining bits represent magnitude
- With $n$ bits, have $n-1$ bits for magnitude
  - E.g., with 4 bits, can represent integers
    - $-7, -6, \ldots, -1, 0, 1, \ldots, 6, 7$

1011 represents -3
- sign: -ve
- magnitude: 3
Properties of sign and magnitude

• Straight-forward and intuitive
• Two different representations of zero!
  • E.g., using 4 bits, 1000 and 0000 both represent zero!
• Arithmetic operations need different implementation than for unsigned
  • E.g., addition, using 4 bits
    • unsigned: 0001 + 1001 = 1 + 9 = 10 = 1010
    • sign and magnitude: 0001 + 1001 = 1 + -1 = 0 = 0000
Ones’ complement

• If integer \( k \) is represented by bits \( b_1...b_n \), then \(-k\) is represented by \( \overline{11...11} - b_1...b_n \) (where \(|\overline{11...11}|=n\))
  • Equivalent to flipping every bit of \( b \)
  • E.g., using \( n=4 \) bits:
    • \( 6 = 0110 \)
    • \(-6 = \overline{1111} - 0110 = 1001 \)
• Using \( n \) bits, can represent numbers \( 2^n - 1 \) values
  • E.g., using 4 bits, can represent integers
    -7, -6, ..., -1, 0, 1, ..., 6, 7
  • Like sign and magnitude, first bit indicates whether number is negative
Properties of ones’ complement

- Same implementation of arithmetic operations as for unsigned
  - E.g., addition, using 4 bits
    - unsigned: 0001 + 1001 = 1 + 9 = 10 = 1010
    - ones’ complement: 0001 + 1001 = 1 + -6 = -5 = 1010

- Two different representations of zero!
  - E.g., using 4 bits, 1111 and 0000 both represent zero!
Two’s complement

• If integer $k$ is represented by bits $b_1...b_n$, then $-k$ is represented by $100...00 - b_1...b_n$ (where $|100...00| = n+1$)
  • Equivalent to taking ones’ complement and adding 1
  • E.g., using 4 bits:
    • $6 = 0110$
    • $-6 = 10000 - 0110 = 1010 = (1111 - 0110) + 1$
• Using $n$ bits, can represent numbers $2^n$ values
  • E.g., using 4 bits, can represent integers
    - $-8, -7, \ldots, -1, 0, 1, \ldots, 6, 7$
  • Like sign and magnitude and ones’ complement, first bit indicates whether number is negative
Properties of two’s complement

• Same implementation of arithmetic operations as for unsigned
  • E.g., addition, using 4 bits
    • unsigned: $0001 + 1001 = 1 + 9 = 10 = 1010$
    • two’s complement: $0001 + 1001 = 1 + -7 = -6 = 1010$
• Only one representation of zero!
  • Simpler to implement operations
• Not symmetric around zero
  • Can represent more negative numbers than positive numbers
• Most common representation of negative integers
Converting to and from two’s complement

• To encode a negative number in two’s complement in $n$ bits:
  • Compute out the binary notation for the absolute value using $n$ bits
  • Invert the bits
  • Add 1
• E.g., to encode -5 using 8 bits
  • $5 = 00000101$ using 8 bits
  • Invert the bits: $11111010$
  • Add one: $11111010 + 1 = 11111011$
• -5 encoded in two’s complement using 8 bits is $11111011$
Converting to and from two’s complement

• To decode two’s complement:
  • If the first bit is 0 then number is positive
  • If the first bit is 1, then number is negative:
    • subtract 1
    • invert bits

• E.g., 110010
  • Subtract one: 110010 - 1 = 110001
  • Invert the bits: 001110 = 14
  • 110010 encodes -14
Integer overflow

• Overflow can also occur with negative integers
• With 32 bits, maximum integer expressible in 2’s complement is $2^{31}-1 = \text{0x7fffffff}$
  
  $\text{0x7fffffff} + \text{0x1} = \text{0x80000000} = -2^{31}$
  
  • Minimum integer expressible in 32-bit 2’s complement
• $\text{0x80000000} + \text{0x80000000} = \text{0x0}$
Integer overflow

\[ \text{UAdd}_4(u, v) \]

Overflow
Integer overflow

$TAdd_4(u, v)$

NegOver

PosOver
Topics for today

• Representing information
  • Hexadecimal notation
  • Representing integers
• Storing information
  • Word size, data size
  • Byte ordering
• Representing strings
• Representing code
• Basic processor operation
Word size

- Every computer has a word size
  - Indicates number of bits that can be used to store integers, memory addresses
- We are in transition between 32-bit machines and 64-bit machines
  - 32-bit machines can name $2^{32}$ different memory locations
    - 1 byte per memory location = 4 gigabytes (= $4 \times 2^{30}$ bytes)
  - 64-bit machines can name $2^{64}$ different memory locations
    - 1 byte per memory location = 16 exabytes (= $16 \times 2^{60}$ bytes)
Data sizes

- C language has multiple data formats for integer and floating-point data

<table>
<thead>
<tr>
<th>C declaration</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
## Typical ranges

<table>
<thead>
<tr>
<th>C declaration</th>
<th>32-bit machine</th>
<th>64-bit machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min.</td>
<td>max.</td>
</tr>
<tr>
<td>char</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long long</td>
<td>-2^63</td>
<td>2^63-1</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>0</td>
<td>2^64-1</td>
</tr>
</tbody>
</table>
Byte ordering

• Memory is big array of bytes
  • Address of location is integer index into array

• When we have data that is more than one byte long, which order do we store the bytes?
  • **Big-endian**: most significant bytes first in memory
  • **Little-endian**: least significant bytes first in memory
    • Most Intel-compatible machines are little-endian

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
</tr>
<tr>
<td>1:</td>
</tr>
<tr>
<td>2:</td>
</tr>
<tr>
<td>3:</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Byte ordering example

• Consider 32-bit (4 byte) integer 0xFF5FE26D
• Suppose stored at memory address 0x100
  • i.e., occupies locations 0x100, 0x101, 0x102, 0x103
• Big endian: most significant bits first

<table>
<thead>
<tr>
<th></th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>0xFF</td>
<td>0x5F</td>
<td>0xE2</td>
</tr>
</tbody>
</table>

• Little endian: least significant bits first

<table>
<thead>
<tr>
<th></th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>0x6D</td>
<td>0xE2</td>
<td>0x5F</td>
</tr>
</tbody>
</table>
Representing strings

• A string in C is an array of characters terminated by the null character (the character having encoding 0)

• Each character is encoded
  
  • Most common encoding is ASCII
    
    • Each character encoded in a single byte
    
    • Good for English-language documents, but not so good for special characters, e.g., é, ç, Φ, ش, Ñ, …
    
    • Run man ascii to see the encoding

• E.g., Encoding the string “CS61”
  
  • Encoding of ‘C’ is 0x43, ‘S’ is 0x53, ‘6’ is 0x36, ‘1’ is 0x31

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
<th>0x104</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x43</td>
<td>0x53</td>
<td>0x36</td>
<td>0x31</td>
</tr>
</tbody>
</table>

... C S 6 1 null...
Representing code

• A computer program can also be encoded in bytes
  • Bytes represent instructions for the computer to perform
• Different types of machines use different (and incompatible) instructions and encodings
  • E.g., given C function
    ```c
    int sum(int x, int y) { return x + y; }
    ```
    the following machine code is produced
    • Linux 32: 55 89 E5 8B 45 0C 03 45 08 C9 C3
    • Windows: 55 89 E5 8B 45 0C 03 45 08 C9 C3
    • Sun: 81 C3 E0 08 90 02 00 09
    • Linux 64: 55 48 89 E5 89 7D FC 89 75 F8 03 45 FC C9 C3
Topics for today

• Representing information
  • Hexadecimal notation
  • Representing integers
  • Storing information
    • Word size, data size
    • Byte ordering
  • Representing strings
  • Representing code
• Basic processor operation
• **An instruction** is a single operation that the CPU can perform
  • add, subtract, copy, call, ...

• **Machine code** is bit-level representation of instructions
  • E.g., in x86: \texttt{0x83 0xEC 0x10} represents “subtract 0x10 from value in the \%esp register”
  • Different instructions may take different number of bits to represent
  • Hard for humans to read

• **Assembly** is human-readable form of machine code
  • E.g., `sub 0x10, \%esp`
Instruction Set Architecture (ISA)

- Definition of machine instructions and format used internally by CPU
  - What instructions the processor can perform, how they are represented, what data types they operate on, etc.
- Specific to the kind of chip and manufacturer
  - Many ISAs, e.g., Alpha, ARM, MIPS, PowerPC, SPARC
- In this course we study the **Intel IA-32 ISA** (aka **x86**)
  - Originated by Intel
  - For 32 bit architectures
  - Evolved (and backward compatible) from earlier ISAs
- Will see some of **x86-64**
  - 64 bit extension of x86
Processor architecture

- CPU executes a series of instructions
  - Each instruction is a simple operation: add, load, store, jump, etc.
  - Instructions stored in memory
  - Program Counter (PC) holds memory address of next instruction

- CPU can read or write memory over the memory bus
  - Can generally read or write a single byte or word at a time

### Memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>0x6D</td>
</tr>
<tr>
<td>1:</td>
<td>0xE2</td>
</tr>
<tr>
<td>2:</td>
<td>0x5F</td>
</tr>
<tr>
<td>3:</td>
<td>0xFF</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Processor operation

- Basic processor operation:
  - 1) Fetch instruction from memory address pointed to by program counter $PC$
  - 2) Execute instruction
  - 3) Set $PC$ to address of next instruction

- Where is the next instruction?
  - Not just “$PC + 1$” – each instruction can be a different size!
  - “Jump” instruction also sets $PC$ to new value
• Registers are used to store “temporary” data on the CPU itself
  • Extremely fast to access a register: 1 clock cycle (0.4 ns on a 2.4 GHz processor)
  • But reading or writing memory can ~40 ns (depends on a lot of factors)
    • Nearly 100x “slowdown” to go to memory!
• The Intel x86 has eight 32-bit registers.
  • Named `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, `%ebp`
  • There are conventions on how certain registers are used
Condition Flags (CF) hold information on state of last instruction

- Each flag is one bit.
- Often used by other instructions to decide what to do.
- e.g., **Overflow flag** is set to 1 if you add two registers, and the value overflows a word.
- **Zero flag** set to 1 if result of an operation is zero

```plaintext
subl $0x42, %eax  # Subtract 0x42 from value in %eax
jz   $0x80495BC   # If zero flag set, jump to instruction
           #   at 0x80495BC
```
"Move" instructions are used to read/write registers and memory locations

movl $0x00001000, %eax  # Set %eax register to value 0x1000

$ prefix indicates constant.
e.g., $0x5395 is value 0x5395
"Move" instructions are used to read/write registers and memory locations

movl $0x00001000, %eax  # Set %eax register to value 0x00001000
movl 0x0000ea5f, %eax   # Set %eax register to contents of memory address 0x0000ea5f
"Move" instructions are used to read/write registers and memory.

- `movl $0x00001000, %eax`  # Set %eax register to value 0x00001000
- `movl 0x0000ea5f, %eax`  # Set %eax register to contents of memory address 0x0000ea5f
- `movl (%eax), %ebx`       # Set %ebx register to contents of memory address stored in %eax

(%eax) means access memory at the address contained in %eax. Just like dereferencing a pointer!

Memory:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x6D</td>
</tr>
<tr>
<td>1</td>
<td>0xE2</td>
</tr>
<tr>
<td>2</td>
<td>0x5F</td>
</tr>
<tr>
<td>3</td>
<td>0xFF</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Register to value 0x00001000
Register to contents of memory address 0x0000ea5f

# Set %ebx register to contents of memory address stored in %eax
“Move” instructions are used to read/write registers and memory locations

movl $0x00001000, %eax  # Set %eax register to value 0x00001000
movl 0x0000ea5f, %eax   # Set %eax register to contents of memory address 0x0000ea5f
movl (%eax), %ebx       # Set %ebx register to contents of memory address stored in %eax
More on memory

- View memory as large array of bytes
- Some conventions on how array is used
- Stack
  - Used to implement function calls, local storage
  - Every time function called, stack grows
  - Every time function returns, stack shrinks
- Heap
  - Dynamically allocated storage for program
  - Expands and contracts as result of calls to malloc and free

<table>
<thead>
<tr>
<th>Kernel virtual memory</th>
<th>User stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared libraries</td>
<td></td>
</tr>
<tr>
<td>Runtime heap</td>
<td></td>
</tr>
<tr>
<td>Read/Write data</td>
<td></td>
</tr>
<tr>
<td>Read-only code and data</td>
<td></td>
</tr>
</tbody>
</table>

(32) 0x08048000
(64) 0x00400000
0x00000000