WELCOME TO THE SUMMER EDITION

- **Data Lab** due Mon, 3 Jun 2013, 11:59 pm EST
  - 2 grace days per lab; don’t waste them
- **Bomb Lab** out Tues, 4 Jun 2013, 11:59 pm EST
  - After the relevant lecture(s)
- **FAQ on the main site**
  - To be updated...?
ADDITIONAL PROBING

- Questions?
- Progress?
- Autolab?
- Shark?
  - > ssh shark.ics.cs.cmu.edu
BECAUSE EVERYONE NEEDS A GUIDE..

- Getting Help
- Literature
- Bits and Bytes and Good Stuff
- IEEE Floating Point
- Data Lab Hints
- General Lab Information
- Question Time
Meet the TAs

Michael Hansen (mhansen1)  Anita Zhang (anitazha)
I NEED HELP):

- Email us: 15-213-staff@cs.cmu.edu
  - Please attach C files if you have a specific question
- IRC: irc.freenode.net, ##213
  - Anita (anitazha) lurks there daily
- Videos on Blackboard
- Everything else, Autolab: autolab.cs.cmu.edu
- Office hours: Sun-Thurs, 6pm – 9pm, Gates 5205
  - The cluster with the window
  - Both Michael and Anita will be there (mostly)
- Potential Google Hangout to come
Books I Like


There are only 10 kinds of people. Those who understand binary and those who don’t.
**Representation Nutshell**

- **Signed**
  - The most significant bit represents the sign
    - 0 for non-negative, 1 for negative
  - On x86, the 31\(^{st}\) bit (counting from 0)
  - Focus on two’s complement

- **Unsigned**
  - Range from 0 to \(2^k - 1\)
    - Where \(k\) is the number of bits used to represent this value
    - Non-negative values

- **Byte = 8 bits**
**What are “ints”?**

- `int` ≠ `integer`
- Minimum and maximum values are capped by the number of bits
Casting Magic

- What happens when casting between signed and unsigned?
Casting Magic

- Signed ↔ Unsigned
  - Values are "reinterpreted"
  - Bits remain the same
- Mixing signed and unsigned values
  - Values are casted to unsigned first
**What is the Size of....**

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>IA32 (x86)</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10 or 12</td>
<td>10 or 16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
OPERATIONS

- **Bitwise**
  - AND → &
  - OR → |
  - NOT → ~
  - XOR → ^

- **Logical**
  - AND → &&
  - OR → ||
  - NOT → !

- **Values**
  - False → 0
  - True → nonzero
**Pro-Tip**

- Do not get bitwise and/or logical mixed up!!
  - If you are getting weird results, look for this error
Specific Operation Stuff

Shifting

- **Arithmetic**
  - Preserves the sign bit (sometimes sign-extended)
- **Logical**
  - Fills with zeros (in our case)
- Other bits “fall off” (discarded)
- Both will result in the same left shift
- Undefined if negative shift amount (to be discussed)
SHIFTING MATH

- Multiplication/ division by $2^k$
  - Multiply: left shift by $k$
  - Division: right shift by $k$
SPECIFIC OPERATION STUFF

- ...I lied. Kind of.
Specific Operation Stuff

- Division of a negative number by $2^k$
  - Needs a “bias”
  - Division looks like this: $(x + (1 << k) - 1) >> k$
    - $x$ is the value we are dividing
    - $(1 << k) - 1$ is the value we are adding to bias
RANDOM NUMBER STUFF

- Endianness is real
  - How bytes are ordered
  - Representation in memory
- You’ll see it in Bomb Lab (next week)

<table>
<thead>
<tr>
<th>Endian</th>
<th>First byte (lowest address)</th>
<th>Middle bytes</th>
<th>Last byte (highest address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>big</td>
<td><em>Most</em> significant</td>
<td>...</td>
<td><em>Least</em> significant</td>
</tr>
<tr>
<td>little</td>
<td><em>Least</em> significant</td>
<td>...</td>
<td><em>Most</em> significant</td>
</tr>
</tbody>
</table>

- Random example: 0x59645322
  - Big: (lower) 59 64 53 22 (higher)
  - Little: (lower) 22 53 64 59 (higher)
Fractional Binary

\[
\begin{array}{ccccccccc}
   b_i & b_{i-1} & \cdots & b_2 & b_1 & b_0 & b_{-1} & b_{-2} & b_{-3} & \cdots & b_{-j} \\
\end{array}
\]

\[
\begin{array}{cccccc}
   2^i & 2^{i-1} & 4 & 2 & 1 & \bullet & \cdots & \bullet \\
\end{array}
\]

\[
\begin{array}{cccc}
   1/2 & 1/4 & 1/8 & \cdots \\
   2^{-j} \\
\end{array}
\]
(Quick and Dirty) Floating Point

What is this floating point stuff?
- Another type of data representation
- Enables support for a wide range of numbers
- Symmetric on its axis (has ±0)
(Quick and Dirty) Floating Point

- Consists of 3 parts
  - Sign bit
  - Exponent bits
  - Fraction bits (the “mantissa”)

Getting the floating point
- Value $\rightarrow (-1)^s \times M \times 2^E$
  - $S \rightarrow$ sign
  - $M \rightarrow$ mantissa
  - $E \rightarrow$ shift amount (exponent bits uses ‘e’ or ‘exp’)
- Bias $\rightarrow 2^{k-1} - 1$
  - Used in the math to convert between actual values and floating point values
(Quick and Dirty) Floating Point

For single precision (32 bit) floating point:
- Fraction (frac): 23 bits
- Exponent (exp): 8 bits
- Sign (s): 1 bit
- Bias = 127

The diagram shows the layout of the floating point representation for 32 bits, with the sign bit at the leftmost position, followed by the exponent and fraction bits.
### (Quick and Dirty) Floating Point

<table>
<thead>
<tr>
<th>Normalized</th>
<th>Denormalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (\text{exp} \neq 00\ldots0)</td>
<td>- (\text{exp} = 00\ldots0)</td>
</tr>
<tr>
<td>- (\text{exp} \neq 11\ldots1)</td>
<td>- (\text{E} = 1 - \text{bias})</td>
</tr>
<tr>
<td>- (\text{E} = \exp - \text{bias})</td>
<td>- (\text{E} = 1 - \text{bias})</td>
</tr>
<tr>
<td>- (\text{M} = 1.xxxxxx)</td>
<td>- (\text{M} = 0.xxxxxx)</td>
</tr>
<tr>
<td>- (\text{xxxxxx}) is the frac</td>
<td>- (\text{xxxxxx}) is the frac</td>
</tr>
<tr>
<td>- Implied leading 1</td>
<td>- Leading 0</td>
</tr>
<tr>
<td></td>
<td>- (\text{frac} = 0) means (\pm 0)</td>
</tr>
</tbody>
</table>
# Special Cases

## Infinity
- exp = 11....1
- frac = 00...0
  - Division by 0, ±∞

## Not a Number
- exp = 11....1
- frac ≠ 00...0
  - sqrt(-1), ∞ - ∞, ∞ x 0
Special cases

- BTW, infinity and NaN are not the same
  - Infinity is “overflow”
  - NaN is not a number
    - “Mathematically undefined” in my book
  - Be aware of this for float_abs()
**LEGIT FLOATING POINT RULES**

- **Rounding**
  - Rounds to even
  - Used to avoid statistical bias
  - $1.1011 \rightarrow 1.11 (>1/2, \text{up})$
  - $1.1010 \rightarrow 1.10 (1/2, \text{down})$
  - $1.0101 \rightarrow 1.01 (>1/2, \text{down})$
  - $1.0110 \rightarrow 1.10 (1/2, \text{up})$

- **Addition and Multiplication...**
  - Are lies
  - Associativity/ distributivity may not hold
  - $3.14 + (1\text{e}20 - 1\text{e}20) \text{ vs. } (3.14 + 1\text{e}20) - 1\text{e}20$
Floating Point on Exams

Let’s pretend we have a 5-bit floating point representation with no sign bit... (sadness)

- $k = 3$ exponent bits (bias = 3)
- $n = 2$ fraction bits

<table>
<thead>
<tr>
<th>Value</th>
<th>Floating Point Bits</th>
<th>(Rounded) Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/32</td>
<td>001 00</td>
<td>1/4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FLOATING POINT ON EXAMS

Let’s pretend we have a 5-bit floating point representation with no sign bit... (sadness)

- k = 3 exponent bits (bias = 3)
- n = 2 fraction bits

<table>
<thead>
<tr>
<th>Value</th>
<th>Floating Point Bits</th>
<th>(Rounded) Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/32</td>
<td>001 00</td>
<td>1/4</td>
</tr>
<tr>
<td>3</td>
<td>100 10</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>110 00</td>
<td>8</td>
</tr>
<tr>
<td>3/16</td>
<td>000 11</td>
<td>3/16</td>
</tr>
<tr>
<td>15/2</td>
<td>110 00</td>
<td>8</td>
</tr>
</tbody>
</table>
FLOATING POINT ON EXAMS

Consider two 7 bit floating point representations based on the IEEE format. Neither has a sign bit.

Format A
- \( k = 3 \) exponent bits (bias = 3)
- \( n = 4 \) fraction bits

Format B
- \( k = 4 \) exponent bits (bias = 7)
- \( n = 3 \) fraction bits

<table>
<thead>
<tr>
<th>Format A</th>
<th>Format B</th>
</tr>
</thead>
<tbody>
<tr>
<td>011 0000</td>
<td>0111 000</td>
</tr>
<tr>
<td>101 1110</td>
<td></td>
</tr>
<tr>
<td>010 1001</td>
<td></td>
</tr>
<tr>
<td>110 1111</td>
<td></td>
</tr>
<tr>
<td>000 0001</td>
<td></td>
</tr>
</tbody>
</table>
Floating Point on Exams

- Consider two 7 bit floating point representations based on the IEEE format. Neither has a sign bit.
- Format A
  - $k = 3$ exponent bits (bias = 3)
  - $n = 4$ fraction bits
- Format B
  - $k = 4$ exponent bits (bias = 7)
  - $n = 3$ fraction bits

<table>
<thead>
<tr>
<th>Format A</th>
<th>Format B</th>
</tr>
</thead>
<tbody>
<tr>
<td>011 0000</td>
<td>0111 000</td>
</tr>
<tr>
<td>101 1110</td>
<td>1001 111</td>
</tr>
<tr>
<td>010 1001</td>
<td>0110 100</td>
</tr>
<tr>
<td>110 1111</td>
<td>1011 000</td>
</tr>
<tr>
<td>000 0001</td>
<td>0001 000</td>
</tr>
</tbody>
</table>
DATA LAB OTHER STUFF

- Use the tools
  - `./driver.pl`
    - Exhaustive autograder (uses provided tools)
  - `./bddcheck/check.pl`
    - Exhaustive
  - `./btest`
    - Not exhaustive
  - `./dlc`
    - This one will hate you if you’re not writing C like it’s 1989
    - Declare all your variables at the beginning of the function
DATA LAB TOOLS

Extra tools

• \texttt{.fshow value}
  - Where value is a hex or decimal number for a floating point
  - Shows the hex for value and breaks it down into the floating point parts (sign, exponent, fraction)
  - Single precision floating point

• \texttt{.ishow value}
  - Where value is a hex or decimal number
  - Outputs value in hex, signed, and unsigned
  - 32-bits
Datalab Other Stuff

- Operator precedence
  - There are charts. Google them.

- bitCount
  - Divide and conquer

- isPower2
  - Actually do and write down operations on paper

- float_i2f
  - You will need to round

- Undefined behavior
  - Shifting by 32
    - And why you get strange results
**UNDEFINED BEHAVIOR (ADV. TOPIC)**

“These instructions shift the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). **Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded.** At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. **The count is masked to five bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1.**”
Labs, in General

- Aim to do all your work on our Shark machines
  - Obtain a terminal/SSH client of sorts
  - Use the following command
    - `ssh andrewID@shark.ics.cs.cmu.edu`
      - `andrewID` is your Andrew ID
      - `shark` can be replaced with a specific shark hostname
        - If left as `shark`, you will be assigned a random shark
    - `tar xvf labhandout.tar`
      - Untarring on the Unix machines may prevent headaches
      - Work out of your private directory
  - Use a text editor straight from the Shark machine
    - Vim, emacs, gedit, nano, pico...
QUESTIONS & CREDITS PAGE

- http://www.superiorsilkscreen.com
- http://www.wikipedia.org/
- http://www.cs.cmu.edu/~213/
- http://jasss.soc.surrey.ac.uk/9/4/4/fig1.jpg
- Intel x86 Instruction Set Reference