ANITA’S SUPER AWESOME RECITATION SLIDES
15/18-213: Introduction to Computer Systems
Bit Logic and Floating Point, 27 January 2014
Anita Zhang
WELCOME TO THE SPRING EDITION

- Data Lab due Thursday, 30 Jan 2014, 11:59 PM
  - 2 grace days per lab, 5 per semester
  - Don’t waste your late days
- Bomb Lab out Thursday, 30 Jan 2014, 12:00 AM
  - After the relevant lecture(s)
- FAQ on the main site (cs.cmu.edu/~213)
  - ... Is actually outdated
  - “Permission denied....”
  - “How to untar...”
ADDITIONAL PROBING

- Quick 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
- Floating Point Recap
- Question Time
I NEED HELP):

- Email us: 15-213-staff@cs.cmu.edu
  - Please attach code if you have a specific question
    - Attempt to (+ show evidence of!) debugging before asking us
  - Goes to TAs and Professors
- Autolab: autolab.cs.cmu.edu
  - No Blackboard (woohoo!)
- Office Hours: Wean 5207, Sun – Thurs, 6 – 8 PM
  - The only Linux cluster in Wean
  - 2-3 TAs at your service, each session
    - Depending on the day there may be a ton of students in line
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 31st 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**
  - Only here because people forget
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?
  - Signed ↔ Unsigned
    - Values are “reinterpreted”
    - Bits remain the same

- Mixing signed and unsigned values
  - Values are casted to unsigned first (most of the time)
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 (on these machines)
- 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 $k$

- Shifting rounds towards negative infinity
  - Math-performing humans round towards 0
  - How do we round negative values toward 0?
DIVISION BY SHIFTING (NEGATIVES)

- Division of a negative number by $2^k$
  - Needs a bias
    - Bias will push the number up so it rounds towards 0
  - 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
    - Remember, only applies to negative values of $x$
FOR THE VISUAL/MATH INCLINED

Dividend:

\[ x = \underbrace{1\cdot\cdot\cdot}_{k} \cdot \underbrace{\cdot\cdot\cdot}_{\text{remainder}} \]

\[ +2^k - 1 = \underbrace{0\cdot\cdot\cdot}_{k} | \underbrace{0\cdot\cdot\cdot1\cdot\cdot\cdot}_{\text{remainder}} \cdot \underbrace{1\cdot\cdot\cdot}_{\text{remainder}} \]

If this contains a 1...

Notice: dividend is negative

Divisor:

\[ /2^k \]

\[ \left\lfloor \frac{x}{2^k} \right\rfloor = \underbrace{1\cdot\cdot\cdot}_{\text{quotient}} | \underbrace{1\cdot\cdot\cdot}_{\text{remainder}} \cdot \underbrace{\cdot\cdot\cdot}_{\text{remainder}} \]

Binary Point

Incremented by 1

Incremented by 1

Biasing adds 1 to final result
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>Last byte (Highest address)</th>
<th>Middle bytes</th>
<th>First byte (lowest address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>big</td>
<td>Least significant</td>
<td>...</td>
<td>Most significant</td>
</tr>
<tr>
<td>little</td>
<td>Most significant</td>
<td>...</td>
<td>Least significant</td>
</tr>
</tbody>
</table>

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

- $2^i$
- $2^{i-1}$
- 4
- 2
- 1
- $2^{-j}$
- $\frac{1}{2}$
- $\frac{1}{4}$
- $\frac{1}{8}$
- $\ldots$
(Quick and Dirty) Floating Point

- What is this floating point stuff?
  - Another type of data representation
  - Enables support for a wide ranges 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

\[ \frac{0.15625}{2^{23}} = 0.15625 \]
(Quick and Dirty) Floating Point

<table>
<thead>
<tr>
<th>Normalized</th>
<th>Denormalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp ≠ 00...0</td>
<td>exp = 00..0</td>
</tr>
<tr>
<td>exp ≠ 11...1</td>
<td>E = 1 – bias</td>
</tr>
<tr>
<td>E = exp – bias</td>
<td>M = 0.xxxxxx</td>
</tr>
<tr>
<td>M = 1.xxxxxx</td>
<td></td>
</tr>
</tbody>
</table>
  • xxxxxxx is the frac                                  |
  • Implied leading 1                               |
  • Leading 0                                       |
  • frac = 0 means ±0                                |
# Special Cases

## Infinity
- $\exp = 11....1$
- $\text{frac} = 00...0$
  - Division by 0, $\pm \infty$

## Not a Number (NaN)
- $\exp = 11....1$
- $\text{frac} \neq 00...0$
  - $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$
By the way, infinity and NaN are not the same

- Infinity is “overflow”
- NaN is not a number
  - “Mathematically undefined” is how I see it...
Legit Floating Point Rules

- Rounding
  - Rounds to even
    - Used to avoid statistical bias
    - $1.1011 \rightarrow 1.11$ (greater than $1/2$, up)
    - $1.1010 \rightarrow 1.10$ (equal to $1/2$, down)
    - $1.0101 \rightarrow 1.01$ (less than $1/2$, down)
    - $1.0110 \rightarrow 1.10$ (equal to $1/2$, up)

- Addition and Multiplication...
  - Are lies
    - Associative/ distributive properties may not hold
    - $3.14 + (1e20 - 1e20)$ vs. $(3.14 + 1e20) - 1e20$
    - Don’t need to do this in this class
INSIGHT INTO ROUNDING

- Round to even
  - How does it avoid statistical bias of rounding up or down on half?

<table>
<thead>
<tr>
<th>Number</th>
<th>Action</th>
<th>Rounded Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.01002</td>
<td>truncate</td>
<td>1.012</td>
</tr>
<tr>
<td>1.01012</td>
<td>below half; round down</td>
<td>1.012</td>
</tr>
<tr>
<td>1.01102</td>
<td>interesting case; round to even</td>
<td>1.102</td>
</tr>
<tr>
<td>1.01112</td>
<td>above half; round up</td>
<td>1.102</td>
</tr>
<tr>
<td>1.10002</td>
<td>truncate</td>
<td>1.102</td>
</tr>
<tr>
<td>1.10012</td>
<td>below half; round down</td>
<td>1.102</td>
</tr>
<tr>
<td>1.10102</td>
<td>Interesting case; round to even</td>
<td>1.102</td>
</tr>
<tr>
<td>1.10112</td>
<td>above half; round up</td>
<td>1.112</td>
</tr>
<tr>
<td>1.11002</td>
<td>truncate</td>
<td>1.112</td>
</tr>
</tbody>
</table>
When converting from `float` to `int`, assume normalized first

- It will be normalized most of the time
  - Easier to convert too
- If it is denormalized, you will be able to tell quickly when doing the normalized math
  - The exponent won’t make sense, for example
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>
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
    - Don’t have whitespace before a closing curly brace
**Data Lab Tools**

- **Extra tools**
  - `.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
  - `.ishow value`
    - Where value is a hex or decimal number
    - Outputs value in hex, signed, and unsigned
    - 32-bits
Starting Data Lab (Newb Edition)

- Untar the lab handout
  - `> tar xvf labhandout.tar`
- Solve puzzles provided in `bits.c`
  - Only file to get turned in
- Test using provided tools
  - You should not be using Autolab to check your work!
  - Everything should be tested by the time you submit
    - `driver.pl` assigns your final grade, NOT `btest`
Datalab Other Stuff

- Operator precedence
  - There are charts. Google them.
  - Alternatively use parenthesis and never worry again.
- Hint: !, 0, and $T_{\text{min}}$ are cool and useful
- No bonus points for having smallest op count
- Other hints in no particular order or reference:
  - Divide and conquer
  - Round to even with floating points
- Undefined behavior
  - Shifting by 32 and why you get strange results
    - My small rant to follow
“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...
Labs, Warnings

- Permission denied
  - Are you working on a Shark machine?
  - Did you untar on a Linux machine?
  - Learning basic commands can fix this
    - `chmod +x executable`
      - Sets executable bits for executable
**Walk-Through (Number to Float)**

- Convert: 27 to 8-bit float (s eeee fff)
- Positive so we know $S = 0$
- Turn 27 to bits:
  - $27_{10} = 11011_2$
- Normalized value so let's fit in the leading 1
  - $1.1011_2 \times 2^4$
- But we only have 3 fraction bits so we must round. Digits after rounding equal half, last rounding digit is 1 so we round up.
  - $1.1011_2 = 1.110_2$
- Frac (F) = $110_2$
Walk-Through (Number to Float)

- Calculate the exponent:
  - $1.1100_2 \times 2^4$

- Calculate the exponent bits:
  - $E = Exponent - Bias = Exponent - 7 = 4$
  - $Exponent = 4 + 7 = 11_{10}$

- So the exponent is 11, in bits:
  - $Exponent = 1011_2$

- Answer: $0 \ 1011 \ 110_2$
Walk-Through (Float to Number)

- Convert: 0 0000 1102 to floating point value
- Sign bit tells us its positive
- It is denormalized because of the 0 exponent so let's figure out $E$:
  - $E = -Bias + 1 = -7 + 1 = -6$
- Now the fraction (remember the leading 0):
  - $0.110 \times 2^{-6}$
- Put it all together:
  - $0.110_2 \times 2^{-6} = 0.000000110_2$
FLOATING POINT QUICK REFERENCE

- **Value** = \((-1)^s \times M \times 2^E\)
- **Bias** = \(2^{k-1} - 1\), \(k\) is the number of \(exp\) bits

**Normalized**
- \(exp \neq 00...0\) and \(exp \neq 11...1\)
- \(E = exp - bias\)
- \(M = 1.frac\)

**Denormalized**
- \(exp = 00..0\)
- \(E = 1 - bias\)
- \(M = 0.frac\)

**Special Cases**
- +/- Infinity: \(exp = 111...1\) and \(frac = 000...0\)
- +/- NaN: \(exp = 111...1\) and \(frac \neq 000...0\)
- +/- 0: \(s = 0\), \(exp = 000...0\) and \(frac = 000...0\)
Questions & Credits Page

- http://www.superiorssilkscreen.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