Floating point representation in computer systems

(Slides adapted from CSAPP book)
Announcements

- Due date for the programming assignment is coming! Next class
- A Google group is created for the class (CEN591Fa11), sign up and post your concerns/questions about the lectures and homework
Summary of Previous class

- Data representation in machines and Bit manipulations

- Integers
  - Signed and unsigned integers
  - Integer casting, truncating
  - Integer negation and addition (subtraction)

- This class: Integer multiplication and division, float representation, IEEE float format
Agenda

- Integer multiplication and division
- Fractional Binary numbers
- IEEE floating point representation
- Floating point rounding, addition, and multiplication
- Float in C
- Summary
Integer multiplication and division
Integer Multiplication

- Computing Exact Product of $w$-bit numbers $x, y$
  - Either signed or unsigned

- Ranges
  - Unsigned: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2^w$ bits
  - Two’s complement min: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2^{w-1}$ bits
  - Two’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2^w$ bits

- Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

- Standard Multiplication Function
  - Ignores high order $w$ bits
- Implements Modular Arithmetic
  \[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]
Signed Multiplication in C

Operands: $w$ bits

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same
Three-bit unsigned and Two’s comp multiplication (Overflow examples)

<table>
<thead>
<tr>
<th>Mode</th>
<th>x</th>
<th>y</th>
<th>x.y</th>
<th>Truncated x.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>5</td>
<td>101</td>
<td>011</td>
<td>15</td>
</tr>
<tr>
<td>T</td>
<td>-3</td>
<td>101</td>
<td>011</td>
<td>-9</td>
</tr>
<tr>
<td>U</td>
<td>4</td>
<td>100</td>
<td>111</td>
<td>28</td>
</tr>
<tr>
<td>T</td>
<td>-4</td>
<td>100</td>
<td>111</td>
<td>4</td>
</tr>
<tr>
<td>U</td>
<td>3</td>
<td>011</td>
<td>011</td>
<td>9</td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>011</td>
<td>011</td>
<td>9</td>
</tr>
</tbody>
</table>

U: Unsigned
T: Two’s complement
Power-of-2 Multiply with Shift

Operation
- \( u << k \) gives \( u \times 2^k \)
- Both signed and unsigned

Operands: \( w \) bits

<table>
<thead>
<tr>
<th>True Product: ( w+k ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \times 2^k )</td>
</tr>
</tbody>
</table>

Discard \( k \) bits: \( w \) bits

\[ \text{UMult}_w(u, 2^k) \]
\[ \text{TMult}_w(u, 2^k) \]

Examples
- \( u << 3 \) \( == \) \( u \times 8 \)
- \( u << 5 - u << 3 \) \( == \) \( u \times 24 \)
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
Integer division by powers of 2

- Integer division expectation: result should always be rounded toward zero

<table>
<thead>
<tr>
<th>real</th>
<th>Rounding toward zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14</td>
<td>3</td>
</tr>
<tr>
<td>-3.14</td>
<td>-3</td>
</tr>
</tbody>
</table>

- Operator:
  \[ \lfloor a \rfloor = a' \Rightarrow a' \leq a \leq a'+1 \]
  \[ \lfloor 3.14 \rfloor = 3, \lfloor -3.14 \rfloor = -4 \]

- \( x \) divided by \( y = \lfloor x / y \rfloor \) if \( x \geq 0 \), and \( y > 0 \)
Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - \( u >> k \) gives \( \left\lfloor u / 2^k \right\rfloor \) [Yielding Proper result]
  - Uses logical shift

Operands:

\[
u \quad k
\]

\[
\begin{array}{c|c|c|c|c}
& 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 \\
\hline
u / 2^k & 0 & \cdots & 0 & 0 & & & & \\
\end{array}
\]

Division:

\[
u / 2^k
\]

\[
\begin{array}{c|c|c|c|c}
& 0 & \cdots & 0 & 0 \\
\hline
\left\lfloor u / 2^k \right\rfloor & 0 & \cdots & 0 & 0 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
</tr>
</tbody>
</table>

CEN591 Fall 2011
Signed Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - \( x >> k \) gives \( \lfloor x / 2^k \rfloor \)
  - Uses arithmetic shift
  - Rounds wrong direction when \( u < 0 \)

![Diagram of division with shift](image)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>(-15213)</td>
<td>(-15213)</td>
<td>C4 93</td>
</tr>
<tr>
<td>( y &gt;&gt; 1)</td>
<td>(-7606.5)</td>
<td>(-7607)</td>
<td>E2 49</td>
</tr>
<tr>
<td>( y &gt;&gt; 4)</td>
<td>(-950.8125)</td>
<td>(-951)</td>
<td>FC 49</td>
</tr>
<tr>
<td>( y &gt;&gt; 8)</td>
<td>(-59.4257813)</td>
<td>(-60)</td>
<td>FF C4</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

- Want \( \left\lfloor \frac{x}{2^k} \right\rfloor \) (Round Toward 0) for negative results
  - Compute as \( \left\lfloor \frac{x + 2^k - 1}{2^k} \right\rfloor \)
  - In C: \((x + (1<<k) - 1) >> k\)
Fractional Binary numbers
Fractional Binary Numbers

What is $1011.101_2$?

Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$
## Fractional Binary Numbers: Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3/4</td>
<td>101.11₂</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.111₂</td>
</tr>
<tr>
<td>63/64</td>
<td>1.0111₂</td>
</tr>
</tbody>
</table>

### Observations
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.111111…₂ are just below 1.0
- \( \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^i} + \ldots \rightarrow 1.0 \)
- Use notation 1.0 – ε
Representable Numbers

- **Limitation**
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations

- **Value**
  - **1/3**: $0.0101010101[01]..._2$
  - **1/5**: $0.001100110011[0011]..._2$
  - **1/10**: $0.0001100110011[0011]..._2$
IEEE Floating point
IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard
Floating Point Representation

- **Numerical Form:**
  \[ (-1)^s \, M \, 2^E \]
  - **Sign bit** \( s \) determines whether number is negative or positive
  - **Significand** \( M \) normally a fractional value in range \([1.0, 2.0)\).
  - **Exponent** \( E \) weights value by power of two

- **Encoding**
  - **MSB** \( s \) is sign bit \( s \)
  - **exp** field encodes \( E \) (but is not equal to \( E \))
  - **frac** field encodes \( M \) (but is not equal to \( M \))
Precisions

- Single precision: 32 bits

  - s
  - exp
  - frac
  - 1 8-bits 23-bits

- Double precision: 64 bits

  - s
  - exp
  - frac
  - 1 11-bits 52-bits

- Extended precision: 80 bits (Intel only)

  - s
  - exp
  - frac
  - 1 15-bits 63 or 64-bits
Types of numbers in IEEE floating points

- Normalized values
- Denormalized values
- Special values
Normalized Values

- Condition: exp \(\neq 000\ldots0\) and exp \(\neq 111\ldots1\)

- Exponent coded as **biased** value: \(E = \text{Exp} – \text{Bias (signed exponent)}\)
  - \(\text{Exp}\): unsigned value exp
  - \(\text{Bias} = 2^{k-1} - 1\), where \(k\) is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: \(-126\ldots127\))
    - Double precision: 1023 (Exp: 1…2046, E: \(-1022\ldots1023\))

- Significand coded with implied **leading** 1: \(M = 1.xxx\ldotsx_2\)
  - \(xxx\ldotsx\): bits of frac
  - Minimum when 000\ldots0 (\(M = 1.0\))
  - Maximum when 111\ldots1 (\(M = 2.0 – \varepsilon\))
  - Get extra leading bit for “free”
Normalized Encoding Example

- **Value:** Float $F = 15213.0$;
  - $15213_{10} = 11101101101101_{2}$
  - $= 1.1101101101101_{2} \times 2^{13}$

- **Significand**
  
  $M = \underline{1.1101101101101}_{2}$
  
  $frac = \underline{11011011011010000000000000}_{2}$

- **Exponent**
  
  $E = 13$
  
  $Bias = 127$
  
  $Exp = 140 = 10001100_{2}$

- **Result:**
  
  $0 \ 10001100 \ 11011011011011010100000000000000$
Denormalized Values

- Condition: \( \text{exp} = 000\ldots0 \)

- Exponent value: \( E = -\text{Bias} + 1 \) (instead of \( E = 0 - \text{Bias} \))

- Significand coded with implied leading 0: \( M = 0.\text{xxx}\ldots\text{x}_2 \)
  - \( \text{xxx}\ldots\text{x} \): bits of \( \text{frac} \)

- Cases
  - \( \text{exp} = 000\ldots0, \text{frac} = 000\ldots0 \)
    - Represents zero value
    - Note distinct values: +0 and −0 (are considered different in some ways and the same in others)
  - \( \text{exp} = 000\ldots0, \text{frac} \neq 000\ldots0 \)
    - Numbers very close to 0.0
Special Values

- **Condition:** \( \text{exp} = 111...1 \)

- **Case:** \( \text{exp} = 111...1, \frac{\text{frac}}{} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty \), \( 1.0/-0.0 = -\infty \)

- **Case:** \( \text{exp} = 111...1, \frac{\text{frac}}{} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty - \infty, \infty \times 0 \)
Visualization: Floating Point Encodings

-∞  −Normalized  −Denorm  +Denorm  +Normalized  +∞  NaN  NaN

−0  +0
Tiny Floating Point Example

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the frac

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity
### Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s exp frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0 0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
</tbody>
</table>

**Denormalized numbers**

<table>
<thead>
<tr>
<th>s exp frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0 0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td>0 0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0 0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
</tbody>
</table>

**Normalized numbers**

<table>
<thead>
<tr>
<th>s exp frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0 0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0 0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td>0 0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s exp frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0 1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td>0 1111 000</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>
Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1} - 1 = 3$

- Notice how the distribution gets denser toward zero.
Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3

![Diagram showing distribution of values with a 6-bit format using 3 exponent bits and 2 fraction bits. The diagram includes a number line with markers for denormalized, normalized, and infinity values.]

-1 -0.5 0 0.5 1

- Denormalized • Normalized ▲ Infinity
# Interesting Numbers

**Description**

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>00...01</td>
<td>$2^{-{23,52}} \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Single $\approx 1.4 \times 10^{-45}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Double $\approx 4.9 \times 10^{-324}$</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Single $\approx 1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Double $\approx 2.2 \times 10^{-308}$</td>
</tr>
<tr>
<td>Smallest Pos. Normalized</td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Just larger than largest denormalized</td>
</tr>
<tr>
<td>One</td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>$(2.0 - \varepsilon) \times 2^{{127,1023}}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Single $\approx 3.4 \times 10^{38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Double $\approx 1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>
Special Properties of Encoding

- FP Zero Same as Integer Zero
  - All bits = 0

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider $-0 = 0$
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity
Rounding, addition, multiplication
Floating Point Operations: Basic Idea

- $x +_f y = \text{Round}(x + y)$
- $x \times_f y = \text{Round}(x \times y)$

Basic idea
- First compute exact result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into frac
## Rounding modes

- **Rounding Modes** (illustrate with $ rounding)

<table>
<thead>
<tr>
<th></th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>$-1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Towards zero</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>$-1</td>
</tr>
<tr>
<td>Round down ((-∞))</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>$-2</td>
</tr>
<tr>
<td>Round up ((+∞))</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$3</td>
<td>$-1</td>
</tr>
<tr>
<td>Nearest Even (default)</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$-2</td>
</tr>
</tbody>
</table>
Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under- estimated

- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth
    - 1.2349999 1.23 (Less than half way)
    - 1.2350001 1.24 (Greater than half way)
    - 1.2350000 1.24 (Half way—round up)
    - 1.2450000 1.24 (Half way—round down)
Rounding Binary Numbers

- **Binary Fractional Numbers**
  - “Even” when least significant bit is 0
  - “Half way” when bits to right of rounding position = 100…2

- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3/32</td>
<td>10.00011₂</td>
<td>10.00₂</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110₂</td>
<td>10.01₂</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100₂</td>
<td>11.00₂</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100₂</td>
<td>10.10₂</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>
FP Multiplication

\[ (-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2} \]

- Exact Result: \((-1)^s M 2^E\)
  - Sign \(s\): \(s_1 \wedge s_2\)
  - Significand \(M\): \(M_1 \times M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

- Fixing
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - If \(E\) out of range, overflow
  - Round \(M\) to fit \textbf{frac} precision

- Implementation
  - Biggest chore is multiplying significands
Floating Point Addition

\[ (-1)^{s_1} M_1 \times 2^{E_1} + (-1)^{s_2} M_2 \times 2^{E_2} \]

- Assume \( E_1 > E_2 \)

Exact Result: \((-1)^s M \times 2^E\)
- Sign \( s \), significand \( M \):
  - Result of signed align & add
- Exponent \( E \): \( E_1 \)

Fixing
- If \( M \geq 2 \), shift \( M \) right, increment \( E \)
- If \( M < 1 \), shift \( M \) left \( k \) positions, decrement \( E \) by \( k \)
- Overflow if \( E \) out of range
- Round \( M \) to fit \textit{frac} precision
Mathematical Properties of FP Add

- Compare to those of Abelian Group
  - Closed under addition?
    - But may generate infinity or NaN
  - Commutative?
  - Associative?
    - Overflow and inexactness of rounding
  - 0 is additive identity?
  - Every element has additive inverse
    - Except for infinities & NaNs

- Monotonicity
  - \( a \geq b \Rightarrow a+c \geq b+c ? \)
    - Except for infinities & NaNs
Mathematical Properties of FP Mult

- Compare to Commutative Ring
  - Closed under multiplication?
    - But may generate infinity or NaN
  - Multiplication Commutative?
  - Multiplication is Associative?
    - Possibility of overflow, inexactness of rounding
  - 1 is multiplicative identity?
  - Multiplication distributes over addition?
    - Possibility of overflow, inexactness of rounding

- Monotonicity
  - $a \geq b \land c \geq 0 \Rightarrow a \times c \geq b \times c$?
    - Except for infinities & NaNs
Floating point in C
Floating Point in C

- C Guarantees Two Levels
  - `float` single precision
  - `double` double precision

- Conversions/Casting
  - Casting between `int`, `float`, and `double` changes bit representation
  - `double/float → int`
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to Tmin
  - `int → double`
    - Exact conversion, as long as `int` has ≤ 53 bit word size
  - `int → float`
    - Will round according to rounding mode
Floating Point Puzzles

For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```c
int x = ...;
float f = ...;
double d = ...;
```

Assume neither `d` nor `f` is NaN

- `x == (int)(float) x`  
  No, ex: `TMax`
- `x == (int)(double) x`  
  Yes, double precision > int precision
- `f == (float)(double) f`  
  Yes, double precision > float precision
- `d == (float) d`  
  No, double precision > float precision
- `f == -(f);`  
  Yes, Converted to a float point
- `2/3 == 2/3.0`  
- `d < 0.0`  
  Yes, even under overflow
- `d > f`  
  -f > -d
- `d * d >= 0.0`  
  Yes, even under overflow
- `(d+f)-d == f`  
  No, due to rounding
Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers
Some hints to homework 1
Quiz Solutions

Q1: For a given integer number, how can you mask its $n$ significant bits to 0?

Sample number: $xxxxxxxxxxxxxxxxxxxxxxxxxxxx$  
$n$ bits  
$32-n$ bits

$$\text{mask} = \sim 0 << (32-n) : 11111100000000000000000000000000$$

$$\text{Result} = x \& \sim \text{mask}$$

Q2: How can you convert an integer $x$ to the integer $y$ such that each group of 4 bits at $y$ represents number of 1 bits at the corresponding 4 bits of $x$. (ex: $x=0x33333333$, then $y=0x22222222, 0x40404040, y=0x10101010$)

$$\text{mask} = 0x11111111$$

$$y = (x \& \text{mask}) + (x >>1 \& \text{mask}) + (x >>2 \& \text{mask}) + (x >> 3 \& \text{mask})$$

Q3: Given an integer $x$, add $y$ to $x$ if $x$ is positive. ($x = x+y \text{ if } x \geq 0$)

$$\text{mask} = \sim (x >> 31)$$

$$s = y \& \text{mask} \ (s=0 \text{ if } x \leq 0, \text{ and } s=y \text{ if } x \geq 0)$$

$$x = x + s$$
<table>
<thead>
<tr>
<th>Function</th>
<th>Sample input output</th>
<th>Hints</th>
</tr>
</thead>
</table>
| logicalShift(x,n) | 0 <= n <= 31        | 1- C does arithmetic shift, you have to mask off upper 1’s
|                   | In: 0x87654321,4    | 2- 32-n?
|                   | ) → out: 0x08765432  | ~0 << (32-n)??                                                                                                                     |
| bitCount(int x)   | In:(5) out: 2       | 1- calculate number of 1 bits at every 4 bits, and then merge them
|                   | In:(7) → out: 3     | 2- Assume a Mask such as:
|                   |                     | 0x11111111:(10001000…)2
|                   |                     | 3- What is the value of s?
|                   |                     | S=(x&mask)+(x >>1 & mask) +(x >>2 & mask)+ (x >> 3 & amsk)                                                                         |
| divpwr2(x,n)      | (x/2^n)             | 1- Remember to round toward zero
|                   | In:(15,1) → out: 7  | 2- if x<0 → add bias (i.e., 2^n-1)
|                   | In:(-33,4) → out: -2| 3- Distinguish the sign of numbers using the MSB
|                   |                     | x>>31=0000… (x>=0)
|                   |                     | x>>31=1111.. (x<0)
Extra Slides
Compiled Unsigned Division Code

C Function

```c
unsigned udiv8(unsigned x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
shrl $3, %eax
```

Explanation

```
# Logical shift
return x >> 3;
```
Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want $\left\lfloor \frac{x}{2^k} \right\rfloor$ (Round Toward 0)
  - Compute as $\left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor$
    - In C: $(x + (1<<k) - 1) >> k$
    - Biases dividend toward 0

Case 1: No rounding

Dividend:

<table>
<thead>
<tr>
<th>$u$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>+2$^k$-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Divisor:

<table>
<thead>
<tr>
<th>$u/2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

$[u/2^k]$ =

<table>
<thead>
<tr>
<th>$u/2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Biasing has no effect
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend: $x + 2^k - 1$

Divisor: $\lfloor x / 2^k \rfloor$

Biasing adds 1 to final result
Compiled Signed Division Code

C Function

```c
int idiv8(int x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```assembly
testl %eax, %eax
js    L4
L3:
    sarl $3, %eax
    ret
L4:
    addl $7, %eax
    jmp  L3
```

Explanation

```plaintext
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```