Floating Point

15-213/18-213/15-513: Introduction to Computer Systems
4th Lecture, May 22, 2020
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
Fractional binary numbers

- What is $1011.101_2$?
Fractional Binary Numbers

- 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

- **Value**
  - 5 3/4 = 23/4
  - 2 7/8 = 23/8
  - 1 7/16 = 23/16

- **Representation**
  - 101.11₂ = 4 + 1/2 + 1/4
  - 10.111₂ = 2 + 1/2 + 1/4 + 1/8
  - 1.011₁₁₂ = 1 + 1/4 + 1/8 + 1/16

- **Observations**
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form 0.111111...₂ are just below 1.0
    - 1/2 + 1/4 + 1/8 + ... + 1/2ⁱ + ... → 1.0
  - Use notation 1.0 − ε
Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - Value | Representation
  - 1/3   | 0.0101010101[01]...2
  - 1/5   | 0.001100110011[0011]...2
  - 1/10  | 0.0001100110011[0011]...2

- Limitation #2
  - Just one setting of binary point within the $w$ bits
    - Limited range of numbers (very small values? very large?)
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
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
  - Some CPUs don’t implement IEEE 754 in full
    e.g., early GPUs, Cell BE processor

- **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
This is important!

- Ariane 5 explodes on maiden voyage: $500 MILLION dollars lost
  - 64-bit floating point number assigned to 16-bit integer
  - Causes rocket to get incorrect value of horizontal velocity and crash

- Patriot Missile defense system misses scud – 28 people die
  - System tracks time in tenths of second
  - Converted from integer to floating point number.
  - Accumulated rounding error causes drift. 20% drift over 8 hours.
  - Eventually (on 2/25/1991 system was on for 100 hours) causes range mis-estimation sufficiently large to miss incoming missiles.
Floating Point Representation

Numerical Form:

\[ (-1)^s \times M \times 2^E \]

- **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

Example:

\[ 15213_{10} = (-1)^0 \times 1.110110111011012 \times 2^{13} \]
Precision options

- Single precision: 32 bits
  \( \approx 7 \text{ decimal digits, } 10^{\pm38} \)

- Double precision: 64 bits
  \( \approx 16 \text{ decimal digits, } 10^{\pm308} \)

- Other formats: half precision, quad precision
Three “kinds” of floating point numbers

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e-bits</td>
<td>f-bits</td>
</tr>
</tbody>
</table>

- 00...00: denormalized
- exp ≠ 0 and exp ≠ 11...11: normalized
- 11...11: special
“Normalized” Values

- When: $\text{exp} \neq 000\ldots0$ and $\text{exp} \neq 111\ldots1$

- Exponent coded as a \textit{biased} value: $E = \text{exp} - \text{Bias}$
  - $\text{exp}$: unsigned value of exp field
  - $\text{Bias} = 2^{k-1} - 1$, where $k$ is number of exponent bits
    - Single precision: 127 ($\text{exp}: 1\ldots254$, $E: -126\ldots127$)
    - Double precision: 1023 ($\text{exp}: 1\ldots2046$, $E: -1022\ldots1023$)

- Significand coded with implied leading 1: $M = 1.\text{xxx}\ldots\text{x}_2$
  - xxx\ldotsx: bits of frac field
  - Minimum when $\text{frac}=000\ldots0$ ($M = 1.0$)
  - Maximum when $\text{frac}=111\ldots1$ ($M = 2.0 - \epsilon$)
  - Get extra leading bit for “free”

$v = (-1)^s \ M \ 2^E$
Normalized Encoding Example

- **Value:** \( \text{float } F = 15213.0; \)
  - \( 15213_{10} = 11101101101101_2 \)
    - \( = 1.1101101101101_2 \times 2^{13} \)

- **Significand**
  - \( M = \underline{1.1101101101101} \)
  - \( \text{frac} = \underline{1101101101101000000000000_2} \)

- **Exponent**
  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \text{exp} = 140 = 10001100_2 \)

- **Result:**
  - \( \text{sign} \quad \text{exp} \quad \text{frac} \)
  - \( 0 \quad 10001100 \quad 11011011011010000000000000 \)

\[ v = (-1)^s \cdot M \cdot 2^E \]
\[ E = \text{exp} - \text{Bias} \]
Denormalized Values

- **Condition**: $\exp = 000...0$

- **Exponent value**: $E = 1 - \text{Bias}$ (instead of $\exp - \text{Bias}$) (why?)

- **Significand coded with implied leading 0**: $M = 0.xxx...x$
  - $xxx...x$: bits of $\text{frac}$

- **Cases**
  - $\exp = 000...0$, $\text{frac} = 000...0$
    - Represents zero value
    - Note distinct values: +0 and −0 (why?)
  - $\exp = 000...0$, $\text{frac} \neq 000...0$
    - Numbers closest to 0.0
    - Equispaced

\[ v = (-1)^s M 2^E \]
\[ E = 1 - \text{Bias} \]
Special Values

- **Condition:** $\exp = 111\ldots 1$

- **Case:** $\exp = 111\ldots 1$, $\frac{c}{c} = 000\ldots 0$
  - 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:** $\exp = 111\ldots 1$, $\frac{c}{c} \neq 000\ldots 0$
  - **Not-a-Number (NaN)**
  - Represents case when no numeric value can be determined
  - E.g., $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$
**C float Decoding Example**

**float: 0xC0A00000**

**binary: ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ **

**1 8-bits 23-bits**

- **E =**
- **S =**
- **M =**

**v = \((-1)^s M 2^E = \)**

**Bias = 2^{k-1} - 1 = 127**

**v = \((-1)^s M 2^E = \)**
C float Decoding Example #1

float: 0xC0A00000

binary: \[1100\ 0000\ 1010\ 0000\ 0000\ 0000\ 0000\ 0000\]

\[E = 1010\ 0000\ 0000\ 0000\ 0000\ 0000\]

\[S = 1\]

\[M = 1.010\ 0000\ 0000\ 0000\ 0000\ 0000\]

\[v = (-1)^s \ M \ 2^E = \]

\[= 1.25\]

\[v = \exp(-\text{Bias})\]
C float Decoding Example #1

float: 0xC0A00000

decimal: 1.25

binary: 1100 0000 1010 0000 0000 0000 0000 0000

1 1000 0001 010 0000 0000 0000 0000 0000

1 8-bits 23-bits

\[ E = \exp - \text{Bias} = 129 - 127 = 2 \text{ (decimal)} \]

\[ S = 1 \rightarrow \text{negative number} \]

\[ M = 1.010 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \]

\[ = 1 + \frac{1}{4} = 1.25 \]

\[ v = (-1)^S \ M \ 2^E = (-1)^1 \times 1.25 \times 2^2 = -5 \]
C float Decoding Example #2

float: 0x001C0000

binary: 0000 0000 0001 1100 0000 0000 0000 0000

\( E = 1 \) – Bias

\( S = 1 \) negative number

\( M = 0.01000000000000000000000000000000 \)

\( v = (-1)^S M 2^E = \)

\( v = 1.25 \)

float: 0x001C0000

binary: 0000 0000 001 1100 0000 0000 0000 0000 0000 0000

Hex Decimal Binary

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
C float Decoding Example #2

float: 0x001C0000

binary:

0000 0000 0001 1100 0000 0000 0000 0000

\[
\begin{array}{cc}
1 & 8\text{-bits} \\
0 & 23\text{-bits}
\end{array}
\]

\[E = 1 - \text{Bias} = 1 - 127 = -126 \text{ (decimal)}\]

\[S = 0 \rightarrow \text{positive number}\]

\[M = 0.0011100000000000000000 = 1/8 + 1/16 + 1/32 = 7/32 = 7 \times 2^{-5}\]

\[v = (-1)^s M 2^E = (-1)^0 \times 7 \times 2^{-5} \times 2^{-126} = 7 \times 2^{-131}\]

\[\approx 2.571393892 \times 10^{-39}\]
Visualization: Floating Point Encodings

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

NaN  −0  +0

NaN
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
Tiny Floating Point Example

- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit
  - the next four bits are the `exp`, 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 (s=0 only)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td>(-1)^0 (0+1/4) * 2^-6</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td>largest denorm</td>
</tr>
<tr>
<td>0</td>
<td>0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td>smallest norm</td>
</tr>
<tr>
<td>0</td>
<td>0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td>(-1)^0 (1+1/8) * 2^-6</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td>closest to 1 below</td>
</tr>
<tr>
<td>0</td>
<td>0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 000</td>
<td>0</td>
<td>8/8*1  = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 001</td>
<td>0</td>
<td>9/8*1  = 9/8</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td>0</td>
<td>0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
</tr>
<tr>
<td>0</td>
<td>1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
</tbody>
</table>

\[ v = (-1)^s \, M \, 2^E \]

**norm:** \( E = \exp - \text{Bias} \)

**denorm:** \( E = 1 - \text{Bias} \)
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.**

![Diagram showing the distribution of values with 6-bit exponent and 2-bit fraction bits, with 8 values distributed across the range from -15 to 15.](image)
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 denormalized, normalized, and infinity markers.]
Special Properties of the IEEE 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? The answer is complicated.
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/16836
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
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 $\text{frac}$
## Rounding

- **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>

*Round to nearest, but if half-way in-between then round to nearest even*
Closer Look at Round-To-Even

■ Default Rounding Mode
  ▪ Hard to get any other kind without dropping into assembly
    ▪ C99 has support for rounding mode management
  ▪ 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
    7.8949999  7.89  (Less than half way)
    7.8950001  7.90  (Greater than half way)
    7.8950000  7.90  (Half way—round up)
    7.8850000  7.88  (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 Value</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>
Rounding

Guard bit: LSB of result

Round bit: 1st bit removed

Round up conditions

- Round = 1, Sticky = 1 → > 0.5
- Guard = 1, Round = 1, Sticky = 0 → Round to even

<table>
<thead>
<tr>
<th>Fraction</th>
<th>GRS</th>
<th>Increment</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>1.1010000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>1.0001000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>1.0011000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>1.0001010</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>1.1111100</td>
<td>111</td>
<td>Y</td>
<td>10.000</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: \(s_1 \land s_2\)
- 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 \(\text{frac}\) precision

Implementation
- Biggest chore is multiplying significands

4 bit significand: \(1.010_2 \times 1.110_2 = 10.0011_2\)
\[
= 1.00011_2 \times 2^6 = 1.0011_2 \times 2^6
\]
Floating Point Addition

- \((-1)^{s_1} M_1 \ 2^{E_1} + (-1)^{s_2} M_2 \ 2^{E_2}\)
  - Assume \(E_1 > E_2\)

- **Exact Result:** \((-1)^s \ M \ 2^E\)
  - 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 \(\text{frac}\) precision

\[
1.010\times2^2 + 1.110\times2^3 = (0.1010 + 1.1100)\times2^3 = 10.0110 \times 2^3 = 1.00110 \times 2^4 = 1.010 \times 2^4
\]
# Mathematical Properties of FP Add

**Compare to those of Abelian Group**

- Closed under addition? **Yes**
  - But may generate infinity or NaN
- Commutative? **Yes**
- Associative? **No**
  - Overflow and inexactness of rounding
  - \((3.14+1e10)-1e10 = 0,\ 3.14+(1e10-1e10) = 3.14\)
- 0 is additive identity? **Yes**
- Every element has additive inverse? **Almost**
  - Yes, except for infinities & NaNs

**Monotonicity**

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

- **Compare to Commutative Ring**
  - Closed under multiplication?  **Yes**
    - But may generate infinity or NaN
  - Multiplication Commutative?  **Yes**
  - Multiplication is Associative?  **No**
    - Possibility of overflow, inexactness of rounding
      - Ex: $(1e20*1e20)*1e-20=\text{inf}, 1e20*(1e20*1e-20)=1e20$
  - 1 is multiplicative identity?  **Yes**
  - Multiplication distributes over addition?  **No**
    - Possibility of overflow, inexactness of rounding
      - $1e20*(1e20-1e20)=0.0, 1e20*1e20-1e20*1e20=\text{NaN}$

- **Monotonicity**
  - $a \geq b \& c \geq 0 \Rightarrow a*c \geq b*c$?  **Almost**
    - Except for infinities & NaNs
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
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` | ✗
- `x == (int)(double) x` | ✓
- `f == (float)(double) f` | ✓
- `d == (double)(float) d` | ✗
- `f == -(-f)` | ✓
- `2/3 == 2/3.0` | ✗
- `d < 0.0 ⇒ ((d*2) < 0.0)` | ✓
- `d > f ⇒ -f > -d` | ✓
- `d * d >= 0.0` | ✓
- `(d+f)−d == f` | ✗
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
Additional Slides
Creating Floating Point Number

Steps

- Normalize to have leading 1
- Round to fit within fraction
- Postnormalize to deal with effects of rounding

Case Study

- Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

<table>
<thead>
<tr>
<th>Integer</th>
<th>Binary 8-bit Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
</tr>
<tr>
<td>15</td>
<td>00001101</td>
</tr>
<tr>
<td>33</td>
<td>00010001</td>
</tr>
<tr>
<td>35</td>
<td>00010011</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
</tr>
</tbody>
</table>
Normalize

**Requirement**

- Set binary point so that numbers of form 1.xxxxx
- Adjust all to have leading one
  - Decrement exponent as shift left

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Fraction</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
<td>1.0000000</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>00001101</td>
<td>1.1010000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.0001000</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.0011000</td>
<td>4</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.0001010</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.1111100</td>
<td>5</td>
</tr>
</tbody>
</table>
Postnormalize

**Issue**

- Rounding may have caused overflow
- Handle by shifting right once & incrementing exponent

<table>
<thead>
<tr>
<th>Value</th>
<th>Rounded</th>
<th>Exp</th>
<th>Adjusted</th>
<th>Numeric Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.000</td>
<td>7</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1.101</td>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1.000</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1.010</td>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>1.001</td>
<td>7</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>10.000</td>
<td>5</td>
<td>1.000/6</td>
<td>64</td>
</tr>
</tbody>
</table>
## Interesting Numbers

<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>