Floating Point
Jan 22, 2004

Topics
• IEEE Floating Point Standard
• Rounding
• Floating Point Operations
• Mathematical properties

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 go fast
• Numerical analysts predominated over hardware types in defining standard

Floating Point Puzzles

For each of the following C expressions, either:
• Argue that it is true for all argument values
• Explain why not true

\[
\begin{align*}
\text{int } x &= \text{(int)(float) } x \\
\text{float } f &= \text{(int)(double) } x \\
\text{float } f &= \text{(float)(double) } f \\
\text{double } d &= \text{(float) } d \\
\text{double } d &= \text{(double) } f \\
\text{d} &= \text{-(-f) ;} \\
\text{2/3} &= \text{2/3.0} \\
\text{d < 0.0 } &\Rightarrow \text{ (d*2) < 0.0 )} \\
\text{d > f } &\Rightarrow \text{-f > -d} \\
\text{d * d >= 0.0 } \\
\text{(d+f)-d == f}
\end{align*}
\]

Assume neither \( d \) nor \( f \) is NaN.

Fractional Binary Numbers

Representation
• Bits to right of “binary point” represent fractional powers of 2
• Represents rational number:
\[
\sum_{k=-j}^{i} b_k \cdot 2^k
\]
### Frac. Binary Number 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>10.111₁₂</td>
</tr>
<tr>
<td>63/64</td>
<td>0.111111₁₂</td>
</tr>
</tbody>
</table>

**Observations**
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form $0.\overline{111111...}_2$ just below 1.0
  - $1/2 + 1/4 + 1/8 + ... + 1/2^i + ... \rightarrow 1.0$
  - Use notation $1.0 - \varepsilon$

### Representable Numbers

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101[01]…₂</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]…₂</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]…₂</td>
</tr>
</tbody>
</table>

### Floating Point Representation

**Numerical Form**
- $-1^s \frac{M}{2^E}$
  - 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 is sign bit
- exp field encodes $E$
- frac field encodes $M$

### Floating Point Precisions

**Encoding**
- MSB is sign bit
- exp field encodes $E$
- frac field encodes $M$

**Sizes**
- Single precision: 8 exp bits, 23 frac bits (32 bits)
- Double precision: 11 exp bits, 52 frac bits (64 bits)
- Extended precision: 15 exp bits, 63 frac bits (80 bits)
  - Only found in Intel-compatible machines
  - Stored in 80 bits
  - 1 bit wasted
FP comes in three flavors

Normalized: When exp isn’t all 0s or 1s
  - Largest range
  - Mantissa has implied leading “1.”

Denormalized: When exp is all 0s
  - evenly spaced close to 0

Special: When exp is all 1s
  - infinities
  - Not a numbers

“Normalized” Numeric Values

Condition
- exp ≠ 000...0 and exp ≠ 111...1

Exponent coded as biased value
- \( E = \exp - \text{Bias} \)
  - \( \exp \): unsigned value denoted by \( \exp \)
  - \( \text{Bias} \): Bias value
    - Single precision: 127 (\( \exp: 1...254 \rightarrow E: -126...127 \))
    - Double precision: 1023 (\( \exp: 1...2046 \rightarrow E: -1022...1023 \))
    - In general: \( \text{Bias} = 2^{e-1} - 1 \), where \( e \) is number of exponent bits

Significand coded with implied leading 1
- \( M = 1.xxx...x_2 \)
  - \( xxx...x \): bits of \( \text{frac} \)
  - Minimum when 000...0 (\( M = 1.0 \))
  - Maximum when 111...1 (\( M = 2.0 - \varepsilon \))
  - Get extra leading bit for “free”

Normalized Encoding Example

Value
- \( \text{float } F = 15213.0; \)
- \( 15213_{10} = 1110110110110112 = 1.1101101101101_2 \cdot 2^{13} \)

Significand
- \( M = 1.1101101101101_2 \)
- \( \text{frac} = 110110110110000000000_2 \)

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

Floating Point Representation (Class 02):
- Hex: 4 6 6 6 D B 4 0 0
- Binary: 0100 0110 0110 1101 1011 1010 0000 0000
- 140: 100 0110 0
- 15213: 2110 1101 1011 01

Denormalized Values

Condition
- \( \exp = 000...0 \)

Value
- Exponent value \( \exp = -\text{Bias} + 1 \)
- Significand value \( M = 0.xxx...x_2 \)
  - \( xxx...x \): bits of \( \text{frac} \)

Cases
- \( \exp = 000...0, \text{frac} = 000...0 \)
  - Represents value 0
  - Note that have distinct values +0 and -0
- \( \exp = 000...0, \text{frac} ≠ 000...0 \)
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - “Gradual underflow”
Special Values

Condition
- \( \exp = 111...1 \)

Cases
- \( \exp = 111...1, \frac{\text{frac}}{0} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/0.0 = +\infty, \ 1.0/-0.0 = -\infty \)
- \( \exp = 111...1, \frac{\text{frac}}{0} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \ \infty - \infty \)

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 \( \frac{\text{frac}}{0} \)

Same General Form as IEEE Format
- Normalized, denormalized
- Representation of 0, NaN, infinity

Values Related to the Exponent

<table>
<thead>
<tr>
<th>Exp</th>
<th>Exp</th>
<th>E</th>
<th>(2^E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>-6</td>
<td>1/64</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>-6</td>
<td>1/64</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>-5</td>
<td>1/32</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>-4</td>
<td>1/16</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-3</td>
<td>1/8</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-2</td>
<td>1/4</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-1</td>
<td>1/2</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>+1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>+2</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>+3</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>+4</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>+5</td>
<td>32</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>+6</td>
<td>64</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>+7</td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>n/a</td>
<td>inf, NaN</td>
</tr>
</tbody>
</table>
### Dynamic Range

<table>
<thead>
<tr>
<th>Exp</th>
<th>Fraction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-6</td>
<td>1/8 * 1/64</td>
<td>1/512</td>
</tr>
<tr>
<td>-6</td>
<td>2/8 * 1/64</td>
<td>2/512</td>
</tr>
<tr>
<td>-6</td>
<td>6/8 * 1/64</td>
<td>6/512</td>
</tr>
<tr>
<td>-6</td>
<td>7/8 * 1/64</td>
<td>7/512</td>
</tr>
<tr>
<td>-6</td>
<td>8/8 * 1/64</td>
<td>8/512</td>
</tr>
<tr>
<td>-6</td>
<td>9/8 * 1/64</td>
<td>9/512</td>
</tr>
</tbody>
</table>

### Denormalized numbers

<table>
<thead>
<tr>
<th>Exp</th>
<th>Fraction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-6</td>
<td>1/8 * 1/64</td>
<td>1/512</td>
</tr>
<tr>
<td>-6</td>
<td>2/8 * 1/64</td>
<td>2/512</td>
</tr>
<tr>
<td>-6</td>
<td>6/8 * 1/64</td>
<td>6/512</td>
</tr>
<tr>
<td>-6</td>
<td>7/8 * 1/64</td>
<td>7/512</td>
</tr>
<tr>
<td>-6</td>
<td>8/8 * 1/64</td>
<td>8/512</td>
</tr>
<tr>
<td>-6</td>
<td>9/8 * 1/64</td>
<td>9/512</td>
</tr>
</tbody>
</table>

### Normalized numbers

<table>
<thead>
<tr>
<th>Exp</th>
<th>Fraction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>14/8 * 1/2</td>
<td>14/16</td>
</tr>
<tr>
<td>-1</td>
<td>15/8 * 1/2</td>
<td>15/16</td>
</tr>
<tr>
<td>7</td>
<td>14/8 * 128</td>
<td>224</td>
</tr>
</tbody>
</table>

### Distribution of Values

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

Notice: the distribution gets denser towards 0.

### 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>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>0</td>
<td>0</td>
<td>2^-126, 10^-32</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>0</td>
<td>1</td>
<td>(10-ε) X 2^-126, 10^-38</td>
</tr>
<tr>
<td>Smallest Pos. Normalized</td>
<td>0</td>
<td>0</td>
<td>1.0 X 2^-126, 10^-38</td>
</tr>
<tr>
<td>One</td>
<td>1</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>1</td>
<td>1</td>
<td>(20-ε) X 2^127, 10^32</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

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 underestimated

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 |

Floating Point Operations

Conceptual View
- First compute exact result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into frac

Rounding Modes (illustrate with $ rounding)

<table>
<thead>
<tr>
<th>Value</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1.40</td>
<td>$1.60</td>
</tr>
</tbody>
</table>

- Zero
- Round down (~)
- Round up (+)
- Nearest Even (default)

Note:
1. Round down: rounded result is close to but no greater than true result.
2. Round up: rounded result is close to but no less than true result.

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>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/3/2</td>
<td>10.00111 10.002  (1/2—down)</td>
</tr>
<tr>
<td>2/3/16</td>
<td>10.0011110 10.0110  (1/2—up)</td>
</tr>
<tr>
<td>2/7/8</td>
<td>10.111001 11.00002 (1/2—up)</td>
</tr>
<tr>
<td>2/5/8</td>
<td>10.101001 10.1000 (1/2—down)</td>
</tr>
</tbody>
</table>
FP Multiplication

Operands

\((-1)^{s_1} M_1 2^{E_1}\) • \((-1)^{s_2} M_2 2^{E_2}\)

Exact Result

\((-1)^s M 2^E\)

- Sign s: \(s_1 \oplus s_2\)
- Significand M: \(M_1 \ast M_2\)
- 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 \(\varepsilon_{\text{mac}}\) precision

Implementation

- Biggest chore is multiplying significands

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
- 0 is additive identity? YES
- Every element has additive inverse ALMOST
  - Except for infinities & NaNs

Monotonicity

- \(a \geq b \Rightarrow a + c \geq b + c?\) ALMOST
  - Except for infinities & NaNs

FP Addition

Operands

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

- Assume \(E_1 \geq E_2\)

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 \(\varepsilon_{\text{mac}}\) precision

Math. 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
- 1 is multiplicative identity? YES
- Multiplication distributes over addition? NO
  - Possibility of overflow, inexactness of rounding

Monotonicity

- \(a \geq b \& c \geq 0 \Rightarrow a \times c \geq b \times c?\) ALMOST
  - Except for infinities & NaNs
Floating Point in C

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

Conversions
- Casting between `int`, `float`, & `double` changes numeric values
- Double or float to int
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    - Generally saturates to TMin or TMax
- `int` to `double`
  - Exact conversion, as long as int has ≤ 53 bit word size
- `int` to `float`
  - Will round according to rounding mode

Answers to Floating Point Puzzles

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x == (int)(float) x</code></td>
<td>No: 24 bit significand</td>
</tr>
<tr>
<td><code>x == (int)(double) x</code></td>
<td>Yes: 53 bit significand</td>
</tr>
<tr>
<td><code>f == (float)(double) f</code></td>
<td>Yes: increases precision</td>
</tr>
<tr>
<td><code>d == (float) d</code></td>
<td>No: loses precision</td>
</tr>
<tr>
<td><code>f == -(-f)</code></td>
<td>Yes: Just change sign bit</td>
</tr>
<tr>
<td><code>2/3 == 2/3.0</code></td>
<td>No: 2/3 == 0</td>
</tr>
<tr>
<td><code>d &lt; 0.0 &lt;=&gt; ((d+2) &lt; 0.0)</code></td>
<td>Yes!</td>
</tr>
<tr>
<td><code>d &gt; f &lt;=&gt; -f &gt; -d</code></td>
<td>Yes!</td>
</tr>
<tr>
<td><code>d * d &gt;= 0.0</code></td>
<td>Yes!</td>
</tr>
<tr>
<td><code>(d+f)-d == f</code></td>
<td>No: Not associative</td>
</tr>
</tbody>
</table>

Ariane 5

- Exploded 37 seconds after liftoff
- Cargo worth $500 million

Why
- Computed horizontal velocity as floating point number
- Converted to 16-bit integer
- Worked OK for Ariane 4
- Overflowed for Ariane 5
  - Used same software

Summary

IEEE Floating Point Has Clear Mathematical Properties
- Represents numbers of form $M \times 2^E$
- 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