Floating Point
Feb 13, 2002

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

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

1. `x == (int)(float) x`
2. `x == (int)(double) x`
3. `f == (float)(double) f`
4. `d == (float) d`
5. `f == -(~f)`
6. `2/3 == 2/3.0`
7. `d < 0.0    \Rightarrow (d*2) < 0.0)`
8. `d > f    \Rightarrow -f > -d`
9. `d * d >= 0.0`
10. `(d+f)-d == f`

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

Fractional Binary Numbers

Representation
- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:
  \[
  \sum_{i=-j}^{j} b_i \cdot 2^i
  \]
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>10.111₁₁₂</td>
</tr>
<tr>
<td>63/64</td>
<td>0.111111₁₁₂</td>
</tr>
</tbody>
</table>

Observations
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.11111₁₁₂ just below 1.0
  - \(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^n} \to 1.0\)
  - Use notation 1-\(\epsilon\): 2

Representable Numbers

Limitation
- Can only exactly represent numbers of the form \(x/2^n\)
- Other numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101₁₁₂₁₁₂ (\to)</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011₁₁₂₁₁₂ (\to)</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011₁₁₂₁₁₂ (\to)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>(s)</th>
<th>(\text{exp})</th>
<th>(\text{frac})</th>
</tr>
</thead>
</table>
- MSB is sign bit
- \(\text{exp}\) field encodes \(E\)
- \(\text{frac}\) field encodes \(M\)

Floating Point Precisions

Encoding

<table>
<thead>
<tr>
<th>(s)</th>
<th>(\text{exp})</th>
<th>(\text{frac})</th>
</tr>
</thead>
</table>
- MSB is sign bit
- \(\text{exp}\) field encodes \(E\)
- \(\text{frac}\) field encodes \(M\)

Sizes
- Single precision: 8 \(\text{exp}\) bits, 23 \(\text{frac}\) bits
  - 32 bits total
- Double precision: 11 \(\text{exp}\) bits, 52 \(\text{frac}\) bits
  - 64 bits total
- Extended precision: 15 \(\text{exp}\) bits, 63 \(\text{frac}\) bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
  - 1 bit wasted
“Normalized” Numeric Values

Condition
- \( \exp \neq 000...0 \) and \( \exp \neq 111...1 \)

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

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

Normalized Encoding Example

Value
- \( \text{Float} F = 15213.0; \)
- \( 15213_{10} = 1110101101101_2 = 1.1101101101101 \times 2^{13} \)

Significant
- \( M = 1.1101101101101 \)
- \( \text{frac} = 11011011011010000000000000_2 \)

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

Floating Point Representation (Class 02):
- Hex: 4 6 6 6 6 0 0 0 0 0 0 0 0 0 0
- Binary: 0100 0110 0110 1101 1011 0100 0000 0000
- 140: 100 0110 0
- 15213: J110 1101 1011 01

Denormalized Values

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

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

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

Special Values

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

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

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

Values Related to the Exponent

Dynamic Range
Distribution of Values

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

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>$2^{-23.521} \times 2^{-128}$</td>
</tr>
<tr>
<td>Single</td>
<td>$1.19 \times 10^{-38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$2.2 \times 10^{-308}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...01</td>
<td>11...11</td>
<td>$(1.0 - \epsilon) \times 2^{-128}$</td>
</tr>
<tr>
<td>Just larger than largest denormalized</td>
<td>$1.0 \times 2^{-128}$</td>
<td></td>
<td></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>$(2.0 - \epsilon) \times 2^{127.1023}$</td>
</tr>
<tr>
<td>Single</td>
<td>$3.4 \times 10^{38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$1.8 \times 10^{308}$</td>
<td></td>
<td></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
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>Binary</th>
<th>Rounded Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3/32</td>
<td>10.00011, 10.00, (1/2—down)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2.3/16</td>
<td>10.00110, 10.01, (1/2—up)</td>
<td>2/1/4</td>
<td></td>
</tr>
<tr>
<td>2.78</td>
<td>10.11100, 11.00, (1/2—up)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2.58</td>
<td>10.10100, 10.10, (1/2—down)</td>
<td>2 1/2</td>
<td></td>
</tr>
</tbody>
</table>

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.

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

Examples:
- 1.2349999 → 1.24 (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...

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3/32</td>
<td>10.00011, 10.00, (1/2—down)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2.3/16</td>
<td>10.00110, 10.01, (1/2—up)</td>
<td>2 1/4</td>
<td></td>
</tr>
<tr>
<td>2.78</td>
<td>10.11100, 11.00, (1/2—up)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2.58</td>
<td>10.10100, 10.10, (1/2—down)</td>
<td>2 1/2</td>
<td></td>
</tr>
</tbody>
</table>

FP Multiplication

Operands

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

Exact Result

\((-1)^s M 2^E\)
- Sign s = s_1 \lor s_2
- Significand M = M_1 \times M_2
- Exponent E = E_1 + E_2

Fixing
- If M ≥ 2, shift M right, increment E
- If E out of range, overflow
- Round M to fit frac precision

Implementation
- Biggest chore is multiplying significands
FP Addition

Operands

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

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

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
- \( a \geq b \) & \( c \geq 0 \) \( \Rightarrow a \cdot c \geq b \cdot c \) ALMOST
  - Except for infinities & NaNs

Floating Point in C

C Guarantees Two Levels

- float single precision
- double double precision

Conversions
- Casting between int, float, and 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 \( \leq 53 \) bit word size
  - int to float
    - Will round according to rounding mode
Answers to Floating Point Puzzles

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

Assume neither `d` nor `f` is NaN

- `x == (int)(float) x` : No: 24 bit significand
- `x == (int)(double) x` : Yes: 53 bit significand
- `f == (float)(double) f` : Yes: increases precision
- `d == (float) d` : No: loses precision
- `f == -(-f)` : Yes: Just change sign bit
- `2/3 == 2/3.0` : No: 2/3 = 0
- `d < 0.0 => (d*2) < 0.0)` : Yes!
- `d > f => -f > -d` : Yes!
- `d * d >= 0.0` : Yes!
- `(d+f)-d == f` : No: Not associative

---

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

---

Page 8