**Floating Point Puzzles**

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

\[
\begin{align*}
\text{int } x &= \ldots; \\
\text{float } f &= \ldots; \\
\text{double } d &= \ldots;
\end{align*}
\]

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

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

**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:
  \[
  \sum_{k=-j}^{i} b_k \cdot 2^k
  \]
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>10.111₂</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 0.111111₁₂ just below 1.0
  - 1/2 + 1/4 + 1/8 + ... + 1/2ⁱ + ... → 1.0
- Use notation 1.0 – ε

Representable Numbers

<table>
<thead>
<tr>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Can only exactly represent numbers of the form x/2ᵏ</td>
</tr>
<tr>
<td>- Other numbers have repeating bit representations</td>
</tr>
</tbody>
</table>

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

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 total
- Double precision: 11 exp bits, 52 frac bits
  - 64 bits total
- Extended precision: 15 exp bits, 63 frac bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
    - 1 bit wasted
**Normalized Encoding Example**

**Value**

Float $F = 15213.0$;

$15213_{10} = 11101101101101_{2} = 1.1101101101101_{2} \times 2^{13}$

**Significand**

$M = \frac{1.1101101101101_{2}}{110110110110100000000000_{2}}$

**Exponent**

$E = 13$

$Bias = 127$

$Exp = 140 = 10001100_{2}$

---

**Floating Point Representation (Class 02):**

Hex: 466DB40

Binary: 01000110010110111110100110100000000000000

140: 10001110

15213: J110 1101 1011 01

---

**Special Values**

**Condition**

- $exp = 111...1$

**Cases**

- $exp = 111...1, frac = 000...0$
  - Represents value $\infty$ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $-1.0/-0.0 = -\infty$
- $exp = 111...1, frac \neq 000...0$
  - 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

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

NaN −0 +0 NaN

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

<table>
<thead>
<tr>
<th>Exp</th>
<th>exp</th>
<th>E</th>
<th>$2^E$</th>
</tr>
</thead>
</table>
| 0   | 0000| 06| 1/64 | (denorms)
| 1   | 0001| 06| 1/64 |
| 2   | 0010| 05| 1/32 |
| 3   | 0011| 04| 1/16 |
| 4   | 0100| 03| 1/8  |
| 5   | 0101| 02| 1/4  |
| 6   | 0110| 01| 1/2  |
| 7   | 0111| 01| 1    |
| 8   | 1000| +2| 2    |
| 9   | 1001| +3| 4    |
| 10  | 1010| +4| 8    |
| 11  | 1011| +5| 16   |
| 12  | 1100| +6| 32   |
| 13  | 1101| +7| 64   |
| 14  | 1110| +7| 128  |
| 15  | 1111| n/a|     |

Dynamic Range

<table>
<thead>
<tr>
<th>s exp frac E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000 -6</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 001 -6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0 0000 010 -6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>0 0000 110 -6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0 0001 000 -6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0 0001 001 -6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td>0 0110 110 -1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0 0110 111 -1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td>0 0111 000 0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0 0111 001 0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td>0 0111 010 0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td>0 1110 110 7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0 1110 111 7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td>0 1111 000 n/a inf</td>
<td></td>
</tr>
</tbody>
</table>
**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.

---

**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>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-(126,1022)}$</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>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>
</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 $\frac{1}{2}$

Rounding Modes (illustrate with $\$\$ rounding)

<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>10.00011_2</td>
<td>10.001_2</td>
<td>&lt;1/2—down</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110_2</td>
<td>10.010_2</td>
<td>&gt;1/2—up</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100_2</td>
<td>11.000_2</td>
<td>1/2—up</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100_2</td>
<td>10.100_2</td>
<td>1/2—down</td>
<td>2 1/2</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
  - 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...

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>10.00011_2</td>
<td>10.001_2</td>
<td>&lt;1/2—down</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110_2</td>
<td>10.010_2</td>
<td>&gt;1/2—up</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100_2</td>
<td>11.000_2</td>
<td>1/2—up</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100_2</td>
<td>10.100_2</td>
<td>1/2—down</td>
<td>2 1/2</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 $s$: $s_1 \wedge s_2$
- 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 $\frac{1}{2}$ precision

Implementation
- Biggest chore is multiplying significands
**FP Addition**

**Operands**

\[-\frac{1}{2^{E_1}} \times M_1 \quad 2^{E_1}\]

\[-\frac{1}{2^{E_2}} \times M_2 \quad 2^{E_2}\]

- Assume \( E_1 > E_2 \)

**Exact Result**

\[-\frac{1}{2^{E}} \times M \]

- 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 float 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**

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

---

**Floating Point in C**

**C Guarantees Two Levels**

- *float* single precision
- *double* double precision

**Conversions**

- Casting between int, float, 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

Assume neither d nor f is NAN

- \( x = \text{int}(\text{float}) \times \) No: 24 bit significand
- \( x = \text{int}(\text{double}) \times \) Yes: 53 bit significand
- \( f = \text{float}(\text{double}) \times \) Yes: increases precision
- \( d = \text{float} \) No: loses precision
- \( f = -(-f) \) Yes: Just change sign bit
- \( 2/3 = 2/3.0 \) No: \( 2/3 \approx 0 \)
- \( d < 0.0 \Rightarrow (d*2 < 0.0) \) Yes!
- \( d > f \Rightarrow -f > -d \) 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