Floating Point Puzzles

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

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

- \( x == (\text{int})(\text{float}) \ x \)
- \( x == (\text{int})(\text{double}) \ x \)
- \( f == (\text{float})(\text{double}) \ f \)
- \( d == (\text{float}) \ d \)
- \( f == -(\text{-f}) \)
- \( 2/3 == 2/3.0 \)
- \( d < 0.0 \implies ((d*2) < 0.0) \)
- \( d > f \implies -f < -d \)
- \( d * d >= 0.0 \)
- \( (d+f)-d == f \)

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

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

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

\[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
Fractional Binary Number Examples

Value | Representation
--- | ---
5\(\frac{3}{4}\) | 10.11\(_2\)
2\(\frac{7}{8}\) | 10.111\(_2\)
63/64 | 0.111111\(_2\)

Observation
- Divide by 2 by shifting right
- Numbers of form 0.111111\(_2\) just below 1.0
  - Use notation 1.0 – \(\varepsilon\)

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

Value | Representation
--- | ---
1/3 | 0.0101010101\([01]\)\(_2\)
1/5 | 0.001100110011\([0011]\)\(_2\)
1/10 | 0.0001100110011\([0011]\)\(_2\)

Fractional Binary Number Examples

Value | Representation
--- | ---
5\(\frac{3}{4}\) | 10.11\(_2\)
2\(\frac{7}{8}\) | 10.111\(_2\)
63/64 | 0.111111\(_2\)

Observation
- Divide by 2 by shifting right
- Numbers of form 0.111111\(_2\) just below 1.0
  - Use notation 1.0 – \(\varepsilon\)

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

Value | Representation
--- | ---
1/3 | 0.0101010101\([01]\)\(_2\)
1/5 | 0.001100110011\([0011]\)\(_2\)
1/10 | 0.0001100110011\([0011]\)\(_2\)

Normalized Encoding Example

Value
- Float F = 15213.0;
- 15213\(_{10}\) = 11101101101101\(_2\)

Significand
- \(M = 1.1101101101101_2 = 1.1101101101101 \times 2^{13}\)

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

Floating Point Representation (Class 02):
- Hex: 4 6 6 D B 4 0 0
- Binary: 0100 0110 0110 1101 1011 0100 0000 0000
- 140: 100 0110 0
- 15213: 1110 1101 1011 01
Denormalized Values

Condition
• \( \text{exp} = 000...0 \)

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

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

Special Values

Condition
• \( \text{exp} = 111...1 \)

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

Summary of Floating Point
Real Number Encodings

-\( \infty \) - Normalized - Denorm - Normalized +\( \infty \)
-0 - +0
\( +\infty \) - NaN - 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 \( \text{frac} \)

• 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>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-6</td>
<td>1/64</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-6</td>
<td>1/64</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-5</td>
<td>1/32</td>
</tr>
<tr>
<td>3</td>
<td>0011</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>s exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>001</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>010</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>111</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Distribution of Representable Values

6-bit IEEE-like format
- K = 3 exponent bits
- n = 2 significand 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>Single</td>
<td>≈ 1.4 X 10^{-45}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>≈ 4.9 X 10^{-324}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \epsilon) \times 2^{-126,1022}$</td>
</tr>
<tr>
<td>Single</td>
<td>≈ 1.18 X 10^{-38}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>≈ 2.2 X 10^{-308}</td>
<td></td>
<td></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>$(2.0 - \epsilon) \times 2^{127,1023}$</td>
</tr>
<tr>
<td>Single</td>
<td>≈ 3.4 X 10^{38}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>≈ 1.8 X 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>Mode</th>
<th>Zero</th>
<th>Round down (\pm)</th>
<th>Round up (\pm)</th>
<th>Nearest Even (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>$1.00</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$1.00</td>
</tr>
<tr>
<td>$1.40</td>
<td>$1.60</td>
<td>$1.50</td>
<td>$2.50</td>
<td>$1.50</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.

A 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
- 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 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3/32</td>
<td>10.000112</td>
<td>10.00</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.001102</td>
<td>10.01</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.111002</td>
<td>11.00</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.101002</td>
<td>10.10</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>

FP Multiplication

Operands
(-1)^s1 M1 2^{E1}
(-1)^s2 M2 2^{E2}

Exact Result
(-1)^s M 2^E
- Sign s: s1 ^ s2
- Significand M: M1 * M2
- Exponent E: E1 + E2

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)^s1 M1 2^{E1}
(-1)^s2 M2 2^{E2}

Assume E1 > E2

Exact Result
(-1)^s M 2^E
- Sign s, significand M:
  - Result of signed align & add
- Exponent E: E1

Fixing
- If M ≥ 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 frac 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 ≥ b ⇒ a+c ≥ b+c? ALMOST
  - Except for infinities & NaNs
Algebraic 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

- \( x == (int)(float) x \)
- \( x == (int)(double) x \)
- \( f == (float)(double) f \)
- \( d == (float) d \)
- \( f == -(f) \);
- \( 2/3 == 2/3.0 \)
- \( d < 0.0 \Rightarrow ((d*2) < 0.0) \)
- \( d > f \Rightarrow -f < -d \)
- \( d \cdot d >= 0.0 \)
- \( (d+f)-d == f \)

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