IEEE 754 Rules & Properties
Analysis of IEEE 754

- As we saw last time, IEEE 754...
  - can represent numbers at wildly different magnitudes (limited by the length of the exponent)
  - provides the same relative accuracy at all magnitudes (limited by the length of the mantissa)
- There are some other nice properties as well related to rounding and arithmetic operations as we’ll see today.
- Turns out there are some drawbacks as well.
Distribution of values

- Remember our 6-bit version of IEEE 754?

- The below graph plots values along a number line between negative and positive infinity.

- Notice how we lose precision as the whole numbers get larger.

- Why is that?
Remember this?

- We saw this on day one...

- **Example**: Is \( (x + y) + z = x + (y + z) \)?
  - for integral types? yes.
  - for floating point types?
    - \((-1e20 + 1e20) + 3.14 == 3.14\)
    - \(-1e20 + (1e20 + 3.14) == 0.0\)

- Do you have any intuition as to why yet?
Special properties of IEEE encoding

- Floating point zero is all zeroes at the bit level.
  - This means zero is all 0’s.

- Can use unsigned integer comparison at the bit level, with a couple notable exceptions…
  - Must consider sign bit
  - Must consider positive and negative 0

- NaN’s
  - Using unsigned comparison a NaN be greater than any other value.
  - Bit-identical NaN values must not be considered equal.

- Otherwise proper ordering, even across types (ex. norm vs denorm)
Interpreting as unsigned bit patterns

- Lets convince ourselves..
- Pick two and test.
- Denorm 000011 and Norm 000101
  - What are their decimal values with unsigned int interpretation?
- Denorm 100001 and Norm 000111
  - What are their decimal values with unsigned int interpretation?
- Special case, the sign bit.
Rounding

- When you do an operation on two floating point numbers such as multiplication or addition, no assurance that there are enough bits to hold result.

- We need a rounding strategy
  - \( x +_f y = \text{Round}(x + y) \)
  - \( x *_f y = \text{Round}(x * y) \)

- Basic idea
  - Compute exact result, make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into \textit{frac}
Rounding modes

- IEEE 754 rounding modes
  
<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>

- IEEE 754 does Nearest Even rounding by default,
  
  - Note that the decision to go to the ‘nearest even’ only arises when you are exactly half-way between two possible rounded values.
  - All others rounding modes are statistically biased.
  - You can change mode, but you have to drop to assembly to do so.
Round to ‘nearest-even’ in decimal

- Applying to other decimal places / bit positions
- When exactly half-way between two possible values, *round so that least significant digit is even*
- E.g., round to nearest hundredth (2 digits right of decimal point)

<table>
<thead>
<tr>
<th>7.8949999</th>
<th>7.89</th>
<th>(Less than half way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.8950001</td>
<td>7.90</td>
<td>(Greater than half way)</td>
</tr>
<tr>
<td>7.8950000</td>
<td>7.90</td>
<td>(Half-way - round up so that the LSD is even)</td>
</tr>
<tr>
<td>7.8850000</td>
<td>7.88</td>
<td>(Half-way - round down so that the LSD is even)</td>
</tr>
</tbody>
</table>
## Round to ‘nearest-even’ in binary

- **Binary fractional numbers**
  - “Half-way” when bits to right of rounding position = 100…0₂
  - “Even” when least significant bit is 0

- E.g., round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value₁₀</th>
<th>Value₂</th>
<th>Rounded₂</th>
<th>Action</th>
<th>Rounded₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3/32</td>
<td>10.00011</td>
<td>10.00</td>
<td>(less than 1/2)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110</td>
<td>10.01</td>
<td>(greater than 1/2)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11000</td>
<td>11.00</td>
<td>(1/2 round-up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100</td>
<td>10.10</td>
<td>(1/2 round-down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>
Floating point multiplication

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

- Exact Result: \((-1)^s M \times 2^E\)
  - Sign: \(s_1 \oplus s_2\)
  - Matissa: \(M_1 \times M_2\)
  - Exponent: \(E_1 + E_2\)

- Fixing
  - If \(M \geq 2\), shift \(M\) left, increment \(E\)
  - If \(E\) out of range, overflow out of range.
  - Round \(M\) to fit \(\text{frac}\) precision
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 $s$, mantissa $M$ are result of signed align & add.
  - Exponent $E$ is just $E_1$

- Fixing
  - If $M \geq 2$, shift $M$ left, increment $E$
  - if $M < 1$, shift $M$ left $k$ positions, decrement $E$ by $k$
  - If $E$ out of range, overflow out of range
  - Round $M$ to fit $\text{frac}$ precision
Properties of floating point addition

- Closed under addition? **Yes**
  - But may generate infinity or NaN

- Commutative? **Yes**

- Associative? **No**
  - Due to overflow and inexactness of rounding
  - \((-1e20 + 1e20) + 3.14 == 3.14\)
  - \(-1e20 + (1e20 + 3.14) == 0.0\)

- 0 is additive identity? **Yes**

- Every element has additive inverse? **Almost**
  - Yes, except for infinities & NaNs

- Monotonicity **Almost**
  - \(a \geq b \Rightarrow a + c \geq b + c?\)
  - Except for infinities & NaNs
Properties of floating point multiplication

- Closed under multiplication? **Yes**
  - But may generate infinity or NaN

- Commutative? **Yes**

- Associative? **No**
  - Due to overflow and inexactness of rounding
  - \((1e20 \times 1e20) \times 1e-20 = inf\), \(1e20 \times (1e20 \times 1e-20) = 1e20\)

- 1 is multiplicative identity? **Yes**

- Multiplication distributes over addition? **No**
  - Due to overflow and inexactness of rounding
  - \(1e20 \times (1e20 - 1e20) = 0.0\), \(1e20 \times 1e20 - 1e20 \times 1e20 = NaN\)

- Monotonicity **Almost**
  - \(a \geq b \land c \geq 0 \implies a \times c \geq b \times c?\)
  - Except for infinities & NaNs
Floating point in C

- Coercion and casting
  - Coercion 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 int has <= 53 bits

  - int → float
    - Will round according to rounding mode, as int has >= 23 bits

- See coercion_casting.c
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 = ...;
```

Assume neither d nor f is NaN

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

See float_puzzles.c
Summary

- IEEE Floating Point has clear mathematical properties
- 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