Floating-Point Numbers

https://xkcd.com/217/
Floating-point

• Topics
  – Binary fractions
  – Floating-point numbers
  – Issues with floating point
  – Formats and tradeoffs
  – Conversions
Binary fractions

- Extend positional binary integers to store fractions
  - Designate a certain number of bits for the fractional part
  - These bits represent negative powers of two
  - (Just like fractional digits in decimal fractions!)
  - (Also note it’s now a “binary point” not a “decimal point”)

\[
\begin{align*}
101.101 & \quad 4 + 1 + 0.5 + 0.125 = 5.625 \\
& \text{Alternatively: } 5 + \frac{5}{8}
\end{align*}
\]
Another problem

- For scientific applications, we want to be able to store a wide *range* of values
  - From the scale of galaxies down to the scale of atoms
- Doing this with fixed-precision numbers is difficult
  - Even signed 64-bit integers
    - Perhaps allocate half for whole number, half for fraction
    - Range: $\sim 2 \times 10^{-9}$ through $\sim 2 \times 10^9$
Floating-point numbers

- Scientific notation to the rescue!
  - Traditionally, we write large (or small) numbers as $x \cdot 10^e$
  - This is how floating-point representations work
    - Store exponent and fractional parts (the significand) separately
    - The fractional point “floats” on the number line
    - Position of point is based on the exponent

$$0.0123 \times 10^2$$
$$0.123 \times 10^1$$
$$1.23 = 1.23 \times 10^0$$
$$12.3 \times 10^{-1}$$
$$123.0 \times 10^{-2}$$
Floating-point numbers: base-2 scientific notation \( (x \cdot 2^e) \)

- Fixed width field
- Reserve one bit for the sign bit (0 is positive, 1 is negative)
- Reserve \( n \) bits for biased exponent (bias is \( 2^{n-1} - 1 \))
- Use remaining bits for normalized fraction (implicit leading 1)
  - Exception: if the exponent is zero, don’t normalize

\[ 2.5 \rightarrow 0 \ 1000 \ 010 \]

- Sign (+)
- Significand: \((1).01_2 = 1.25\)
- Exponent: \(8 - 7 = 1\)

Value = \((-1)^s \times 1.f \times 2^E = (-1)^0 \times 1.25 \times 2^1 = 2.5\)
Aside: Offset binary

- Alternative to two’s complement
  - Actual value is stored value minus a constant K (in FP: $2^{n-1} - 1$)
  - Also called biased or excess representation
  - Ordering of actual values is more natural

<table>
<thead>
<tr>
<th>Example range (int8_t):</th>
<th>Binary</th>
<th>Unsigned</th>
<th>Two’s C</th>
<th>Offset-127</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000</td>
<td>0</td>
<td>0</td>
<td>-127</td>
</tr>
<tr>
<td></td>
<td>0000 0001</td>
<td>1</td>
<td>1</td>
<td>-126</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>0111 1110</td>
<td>126</td>
<td>126</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>0111 1111</td>
<td>127</td>
<td>127</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1000 0000</td>
<td>128</td>
<td>-128</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1000 0001</td>
<td>129</td>
<td>-127</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>1111 1110</td>
<td>254</td>
<td>-2</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>1111 1111</td>
<td>255</td>
<td>-1</td>
<td>128</td>
</tr>
<tr>
<td>Description</td>
<td>Bit representation</td>
<td>Exponent</td>
<td>Exponent bias</td>
<td>Fraction</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------</td>
<td>----------</td>
<td>---------------</td>
<td>----------</td>
</tr>
<tr>
<td>Zero</td>
<td>0 0000 000</td>
<td>0</td>
<td>-6</td>
<td>$\frac{1}{64}$</td>
</tr>
<tr>
<td>Smallest positive</td>
<td>0 0000 001</td>
<td>0</td>
<td>-6</td>
<td>$\frac{1}{64}$</td>
</tr>
<tr>
<td></td>
<td>0 0000 010</td>
<td>0</td>
<td>-6</td>
<td>$\frac{1}{64}$</td>
</tr>
<tr>
<td></td>
<td>0 0000 011</td>
<td>0</td>
<td>-6</td>
<td>$\frac{1}{64}$</td>
</tr>
<tr>
<td>Largest denormalized</td>
<td>0 0000 111</td>
<td>0</td>
<td>-6</td>
<td>$\frac{1}{64}$</td>
</tr>
<tr>
<td>Smallest normalized</td>
<td>0 0001 000</td>
<td>1</td>
<td>-6</td>
<td>$\frac{1}{64}$</td>
</tr>
<tr>
<td></td>
<td>0 0001 001</td>
<td>1</td>
<td>-6</td>
<td>$\frac{1}{64}$</td>
</tr>
<tr>
<td>values &lt; 1</td>
<td>0 0110 110</td>
<td>6</td>
<td>-1</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td></td>
<td>0 0110 111</td>
<td>6</td>
<td>-1</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td>One</td>
<td>0 0111 000</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0 0111 001</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0 0111 010</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>values &gt; 1</td>
<td>0 1110 110</td>
<td>14</td>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>Largest normalized</td>
<td>0 1110 111</td>
<td>14</td>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>Infinity</td>
<td>0 1111 000</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Figure 2.35  Example nonnegative values for 8-bit floating-point format. There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.
Floating-point issues

- **Rounding error** is the value lost during conversion to a finite significand
  - **Machine epsilon** gives an upper bound on the rounding error
    - (Multiply by value being rounded)
  - Can compound over successive operations
- **Lack of associativity** caused by intermediate rounding
  - Prevents some compiler optimizations
- **Cancellation** is the loss of significant digits during subtraction
  - Can magnify error and impact later operations

```c
double a = 100000000000000000000.0;
double b = -a;
double c = 3.14;
if (((a + b) + c) == (a + (b + c))) {  
  printf("Equal!\n");
} else {
  printf("Not equal!\n");
}
```

<table>
<thead>
<tr>
<th>2.491264 (7)</th>
<th>1.613647 (7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 2.491252 (7)</td>
<td>- 1.613647 (7)</td>
</tr>
<tr>
<td>0.000012 (2)</td>
<td>0.000000 (0)</td>
</tr>
</tbody>
</table>

(5 digits cancelled) (all digits cancelled)
Floating-point numbers

Not evenly spaced! (as integers are)

Adding a least-significant digit adds more value with a higher exponent than with a lower exponent

Floating-point demonstration using Super Mario 64:

https://www.youtube.com/watch?v=9hdFG2GcNuA
**NaNs**

- **NaN** = “Not a Number”
  - Result of $0/0$ and other undefined operations
  - Propagate to later calculations

<table>
<thead>
<tr>
<th>1. Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s \neq 0 &amp; s \neq 255$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. Denormalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s {0 \ldots 0 }$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3a. Infinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s {1 \ldots 1 }$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3b. NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s {1 \ldots 1 }$</td>
</tr>
</tbody>
</table>
Floating-point issues

- Many numbers cannot be represented exactly, regardless of how many bits are used!
  - E.g., \(0.1_{10} \rightarrow 0.0001100110011001100_{2} \ldots\)

- This is no different than in base 10
  - E.g., \(1/3 = 0.333333333 \ldots\)

- If the number can be expressed as a sum of negative powers of the base, it can be represented exactly
  - Assuming enough bits are present
# Floating-point standards

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Exp</th>
<th>Sig</th>
<th>Dec</th>
<th>M_Eps</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfloat16</td>
<td>16</td>
<td>8</td>
<td>7+1</td>
<td>2.408</td>
<td>7.81e-03</td>
</tr>
<tr>
<td>IEEE half</td>
<td>16</td>
<td>5</td>
<td>10+1</td>
<td>3.311</td>
<td>9.77e-04</td>
</tr>
<tr>
<td>IEEE single</td>
<td>32</td>
<td>8</td>
<td>23+1</td>
<td>7.225</td>
<td>1.19e-07</td>
</tr>
<tr>
<td>IEEE double</td>
<td>64</td>
<td>11</td>
<td>52+1</td>
<td>15.955</td>
<td>2.22e-16</td>
</tr>
<tr>
<td>IEEE quad</td>
<td>128</td>
<td>15</td>
<td>112+1</td>
<td>34.016</td>
<td>1.93e-34</td>
</tr>
</tbody>
</table>

**NOTES:**
- Sig is `<explicit>[+<implicit>]` bits
- Dec = \( \log_{10}(2^{\text{Sig}}) \)
- M_Eps (machine epsilon) = \( b^{-(p-1)} = b^{(1-p)} \)
  (upper bound on relative error when rounding to 1)
Floating-point issues

• Single vs. double precision choice
  – Theme: system design involves tradeoffs
  – Single precision arithmetic is faster
    • Especially on GPUs (vectorization & bandwidth)
  – Double precision is more accurate
    • More than twice as accurate!
  – Which do we use?
    • And how do we justify our choice?
    • Does the answer change for different regions of a program?
    • Does the answer change for different periods during execution?
    • This is an open research question (talk to me if you’re interested!)
Question

• Which of the following conversions are “safe” (i.e., the value can always be preserved)?
  - A) 32-bit signed int → 32-bit floating-point
  - B) 32-bit signed int → 64-bit floating-point
  - C) 32-bit floating-point → 32-bit signed int
  - D) 32-bit floating-point → 64-bit signed int
  - E) 32-bit floating-point → 64-bit floating-point
  - F) 64-bit floating-point → 32-bit floating-point
### Conversion and rounding

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32</td>
<td>Int32</td>
</tr>
<tr>
<td>Int32</td>
<td>-</td>
</tr>
<tr>
<td>Int64</td>
<td>O</td>
</tr>
<tr>
<td>Float</td>
<td>OR</td>
</tr>
<tr>
<td>Double</td>
<td>OR</td>
</tr>
</tbody>
</table>

- **O** = overflow possible
- **R** = rounding possible

- "-" is safe
Round-to-even: round to nearest, on ties favor even numbers to avoid statistical biases

In binary, to round to bit $i$, examine bit $i+1$:
- If 0, round down
- If 1 and any of the bits following are 1, round up
- Otherwise, round up if bit $i$ is 1 and down if bit $i$ is 0

<table>
<thead>
<tr>
<th>Mode</th>
<th>$.140</th>
<th>$.160</th>
<th>$.150</th>
<th>$.250</th>
<th>$.150</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round-to-even</td>
<td>$1$</td>
<td>$2$</td>
<td>$2$</td>
<td>$2$</td>
<td>$-2$</td>
</tr>
<tr>
<td>Round-toward-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>
</tbody>
</table>

**Figure 2.37 Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.
Manual conversions

- To fully understand how floating-point works, it helps to do some conversions manually
  - This is unfortunately a bit tedious and very error-prone
  - There are some general guidelines that can help it go faster
  - You will get better and faster with practice
  - Use the fp.c utility (github.com/lam2mo/fp) to generate practice problems and test yourself!
    - Compile: gcc -o fp fp.c -lm
    - Run: ./fp <exp_len> <sig_len>
    - It will generate all positive floating-point numbers using that representation
    - Choose one and convert the binary to decimal or vice versa

...
Textbook's technique

\( e \): The value represented by considering the exponent field to be an unsigned integer

\( E \): The value of the exponent after biasing

\( 2^E \): The numeric weight of the exponent

\( f \): The value of the fraction

\( M \): The value of the significand

\( 2^E \times M \): The (unreduced) fractional value of the number

\( V \): The reduced fractional value of the number

Decimal: The decimal representation of the number

If this technique works for you, great!
If not, here's another perspective...
Converting floating-point numbers

- **Floating-point → decimal:**
  - 1) Sign bit (s):
    - Value is negative iff set
  - 2) Exponent (exp):
    - All zeroes: denormalized ($E = 1$-bias)
    - All ones: NaN unless $f$ is zero (which is infinity) – **DONE!**
    - Otherwise: normalized ($E = \text{exp}$-bias)
  - 3) Significand ($f$):
    - If normalized: $M = 1 + f / 2^m$ (where $m$ is the # of fraction bits)
    - If denormalized: $M = f / 2^m$ (where $m$ is the # of fraction bits)
  - 4) Value = $(-1)^s \times M \times 2^E$

Note: $\text{bias} = 2^{n-1} - 1$ (where $n$ is the # of exp bits)
Converting floating-point numbers

- Decimal → floating-point (normalized only)
  - 1) Convert to unsigned fractional binary format
    - Set sign bit
  - 2) Normalize to 1.xxxxxx
    - Keep track of how many places you shift left (negative for shift right)
    - The “xxxxxx” bit string is the significand (pad with zeros on the right)
    - If there aren’t enough bits to store the entire fraction, the value is rounded
  - 3) Encode resulting binary/shift offset (E) using bias representation
    - Add bias and convert to unsigned binary
    - If the exponent cannot be represented, result is zero or infinity

Example (4-bit exp, 3-bit frac):

2.75 (dec) → 10.11 (bin) → 1.011 × 2\(^1\) (bin) → 0 1000 011

Bias = 2\(^{4-1}\) – 1 = 7
Exp: 1 + 7 = 8

Note:
bias = 2\(^{n-1}\) - 1
(where \(n\) is the # of exp bits)