Floating-Point Numbers
Floating-point

• Topics
  - Binary fractions
  - Floating-point representation
  - Conversions and rounding error
Binary fractions

- Now we can store integers
  - But what about general real numbers?
- 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!)

```
101.101
```

```
4  2  1  1/2  1/4  1/8
```

```
4 + 1 + 0.5 + 0.125 = 5.625  (alternatively: 5 + 5/8)
```
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: \(~2 \times 10^{-9}\) through \(~2 \times 10^{9}\)

Floating-point demonstration using Super Mario 64:
https://www.youtube.com/watch?v=9hdFG2GcNuA
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 decimal 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

- However, computers use binary
  - So floating-point numbers use 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 \(2^{n-1} - 1\))
    - Avoids having to use two’s complement
  - Use remaining bits for normalized fraction (implicit leading 1)
    - Exception: if the exponent is zero, don’t normalize

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

Value = \((-1)^s \times 1.f \times 2^E\)
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

### Example range (int8_t):

<table>
<thead>
<tr>
<th>Binary</th>
<th>Unsigned</th>
<th>Two’s C</th>
<th>Offset-127</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0</td>
<td>0</td>
<td>-127</td>
</tr>
<tr>
<td>0000 0001</td>
<td>1</td>
<td>1</td>
<td>-126</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1110</td>
<td>126</td>
<td>126</td>
<td>-1</td>
</tr>
<tr>
<td>0111 1111</td>
<td>127</td>
<td>127</td>
<td>0</td>
</tr>
<tr>
<td>1000 0000</td>
<td>128</td>
<td>-128</td>
<td>1</td>
</tr>
<tr>
<td>1000 0001</td>
<td>129</td>
<td>-127</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 1110</td>
<td>254</td>
<td>-2</td>
<td>127</td>
</tr>
<tr>
<td>1111 1111</td>
<td>255</td>
<td>-1</td>
<td>128</td>
</tr>
</tbody>
</table>
Floating-point numbers

Not evenly spaced! (as integers are)

Consider these examples:

1.00000 \times 2^0 \rightarrow 1.00001 \times 2^0
1.00000 \times 2^{100} \rightarrow 1.00001 \times 2^{100}

Adding a least-significant digit adds more value with a higher exponent than with a lower exponent.
Representable values for 6-bit floating-point format. There are $k = 3$ exponent bits and $n = 2$ fraction bits. The bias is 3.
```
<table>
<thead>
<tr>
<th>Description</th>
<th>Bit representation</th>
<th>Exponent</th>
<th>Fraction</th>
<th>Value</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0 0000 000</td>
<td>0 -6</td>
<td>1/64</td>
<td>0 512</td>
<td>0</td>
</tr>
<tr>
<td>Smallest positive</td>
<td>0 0000 001</td>
<td>0 -6</td>
<td>1/64</td>
<td>0 512</td>
<td>1/512</td>
</tr>
<tr>
<td></td>
<td>0 0000 010</td>
<td>0 -6</td>
<td>1/64</td>
<td>0 512</td>
<td>2/256</td>
</tr>
<tr>
<td></td>
<td>0 0000 011</td>
<td>0 -6</td>
<td>1/64</td>
<td>0 512</td>
<td>3/512</td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest denormalized</td>
<td>0 0000 111</td>
<td>0 -6</td>
<td>7/8</td>
<td>7 512</td>
<td>7 512</td>
</tr>
<tr>
<td>Smallest normalized</td>
<td>0 0001 000</td>
<td>1 -6</td>
<td>1/64</td>
<td>8 512</td>
<td>1/64</td>
</tr>
<tr>
<td></td>
<td>0 0001 001</td>
<td>1 -6</td>
<td>1/64</td>
<td>9 512</td>
<td>9 512</td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>values &lt; 1</td>
<td>0 0110 110</td>
<td>6 -1</td>
<td>1/2</td>
<td>6 16</td>
<td>14 16</td>
</tr>
<tr>
<td></td>
<td>0 0110 111</td>
<td>6 -1</td>
<td>1/2</td>
<td>7 16</td>
<td>15 16</td>
</tr>
<tr>
<td>One</td>
<td>0 0111 000</td>
<td>7 0</td>
<td>1/8</td>
<td>8 8</td>
<td>8 8</td>
</tr>
<tr>
<td></td>
<td>0 0111 001</td>
<td>7 0</td>
<td>1/8</td>
<td>9 8</td>
<td>9 8</td>
</tr>
<tr>
<td></td>
<td>0 0111 010</td>
<td>7 0</td>
<td>2/8</td>
<td>10 8</td>
<td>10 8</td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>values &gt; 1</td>
<td>0 1110 110</td>
<td>14 7</td>
<td>128</td>
<td>8 16</td>
<td>1792 8</td>
</tr>
<tr>
<td>Largest normalized</td>
<td>0 1110 111</td>
<td>14 7</td>
<td>128</td>
<td>8 16</td>
<td>1920 8</td>
</tr>
<tr>
<td>Infinity</td>
<td>0 1111 000</td>
<td>—</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.

what about values higher than this one?
## Floating-point numbers

1. **Normalized**

<table>
<thead>
<tr>
<th>s</th>
<th>( \neq 0 ) &amp; ( \neq 255 )</th>
<th>f</th>
</tr>
</thead>
</table>

2. **Denormalized**

   | s | 0 0 0 0 0 0 0 0 | f |

3a. **Infinity**

   | s | 1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

3b. **NaN**

   | s | 1 1 1 1 1 1 1 1 | \( \neq 0 \) |
NaNs

- NaN = “Not a Number”
  - Result of 0/0 and other undefined operations
  - Propagate to later calculations
  - Quiet and signaling variants (qNaN and sNaN)
  - Allowed a neat trick during my dissertation research:
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

- **Cancelation** 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");
}
```

```
  2.491264  (7)   1.613647  (7)
- 2.491252  (7)  - 1.613647  (7)
  0.000012  (2)   0.000000  (0)

(5 digits cancelled)   (all digits cancelled)
```
Floating-point issues

- Many numbers cannot be represented exactly, regardless of how many bits are used!
  - E.g., $0.1_{10} \rightarrow 0.00011001100110011001100_{2} \ldots$

- 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
<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>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 standards

IEEE Floating-Point Numbers

Value is: \((-1)^{\text{sign}} \times 1.\text{frac} \times 2^{\text{exp}}\)
Conversion and rounding

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

To:

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>$1.40$</th>
<th>$1.60$</th>
<th>$1.50$</th>
<th>$2.50$</th>
<th>$-1.50$</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.
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!)
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 also get faster with practice
- Use the fp.c utility (posted on the resources page) to generate practice problems and test yourself!
  - Compile: gcc -o fp fp.c
  - 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 ($E = exp$-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 x 2^1 (bin) → 0 1000 011

Bias = 2^(n-1) – 1 = 7

Exp: 1 + 7 = 8

Note:

bias = 2^(n-1) - 1

(where n is the # of exp bits)
Example (textbook pg. 119)

\[ 12345_{10} \rightarrow 11000000111001_2 \]
\[ \rightarrow 1.1000000111001_2 \times 2^{13} \]
\[ \text{exp} = 13 + 127 \text{ (bias)} = 140 = 10001100_2 \]
\[ \rightarrow 0 \ 10001100 \ 1000000111001000000000000 \]

(note the shared bits that appear in all three representations)
Exercises

• What are the values of the following numbers, interpreted as floating-point numbers with a 3-bit exponent and 2-bit significand?
  
  – What about a 2-bit exponent and a 3-bit significand?

  $\text{001100} \quad \text{011001}$

• Convert the following values to a floating-point value with a 4-bit exponent and a 3-bit significand. Write your answers in hex.

  $-3 \quad 0.125 \quad 120 \quad \infty$