### Floating Point Puzzles

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

For each of the following, either:
- argue that it is true for all argument values, or
- explain why it is not true.

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

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

### IEEE Floating Point Standard

**IEEE Standard 754**

- Established in 1985 as a uniform standard for floating point arithmetic
- It is supported by all major CPUs.
- Before 1985 there were many idiosyncratic formats.

**Driven by Numerical Concerns**

- Nice standards for rounding, overflow, underflow
- Hard to make go fast: numerical analysts predominated over hardware types in defining the standard
- Now all (add, subtract, multiply) operations are fast except divide.
The binary number \(b_ib_{i-1}b_2b_1 \ldots b_0.b_{-1}b_{-2}b_{-3} \ldots b_{-j}\) represents a particular (positive) sum. Each digit is multiplied by a power of two according to the following chart:

<table>
<thead>
<tr>
<th>Bit:</th>
<th>(b_i)</th>
<th>(b_{i-1})</th>
<th>(\ldots)</th>
<th>(b_2)</th>
<th>(b_1)</th>
<th>(b_{-1})</th>
<th>(b_{-2})</th>
<th>(b_{-3})</th>
<th>(\ldots)</th>
<th>(b_{-j})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight:</td>
<td>(2^i)</td>
<td>(2^{i-1})</td>
<td>(\ldots)</td>
<td>(4)</td>
<td>(2)</td>
<td>(1/2)</td>
<td>(1/4)</td>
<td>(1/8)</td>
<td>(\ldots)</td>
<td>(2^{-j})</td>
</tr>
</tbody>
</table>

**Representation:**
- Bits to the right of the binary point represent fractional powers of 2.
- This represents the rational number:
  \[\sum_{k=-j}^{i} b_k \times 2^k\]

The sign is treated separately.

### Observations
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of the form 0.11111\ldots2 are just below 1.0
  - \(1/2 + 1/4 + 1/8 + \ldots + 1/2^i \rightarrow 1.0\)
  - We use the notation 1.0 - \(\epsilon\).

### Representable Numbers

**Limitation**
- You can only represent numbers of the form \(y + x/2^i\).
- Other fractions have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.010101010101[01]\ldots2</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]\ldots2</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]\ldots2</td>
</tr>
</tbody>
</table>

### Aside: Converting Decimal Fractions to Binary

If you want to convert a decimal fraction to binary, it's easy if you use a simple iterative procedure.

- Start with the decimal fraction (> 1) and multiply by 2.
- Stop if the result is 0 (terminated binary) or a result you've seen before (repeating binary).
- Record the whole number part of the result.
- Repeat from step 1 with the fractional part of the result.

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.375</td>
<td>(0.375 \times 2 = 0.75)</td>
</tr>
<tr>
<td></td>
<td>(0.75 \times 2 = 1.5)</td>
</tr>
<tr>
<td></td>
<td>(0.5 \times 2 = 1.0)</td>
</tr>
<tr>
<td></td>
<td>(0.0)</td>
</tr>
</tbody>
</table>

The result (following the binary point) is the series of whole numbers components of the answers read from the top, i.e., 0.011.
Aside: Converting Decimal Fractions to Binary (2)

Let’s try another one, 0.1 or 1/10

\[
\begin{align*}
0.1 \times 2 &= 0.2 \\
0.2 \times 2 &= 0.4 \\
0.4 \times 2 &= 0.8 \\
0.8 \times 2 &= 1.6 \\
0.6 \times 2 &= 1.2 \\
0.2 \times 2 &= 0.4
\end{align*}
\]

We could continue, but we see that it’s going to repeat forever (since 0.2 repeats our multiplicand from the second line). Reading the integer parts from the top gives 0[0011], since we’ll repeat the last 4 bits forever.

Floating Point Representation

**Numerical Form**

\[-1^s \times M \times 2^E\]

- Sign bit \(s\) determines whether number is negative or positive.
- Significand \(M\) is normally a fractional value in the range [1.0...2.0).
- Exponent \(E\) weights value by power of two.

**Floats (32-bit floating point numbers)**

Floating Point Precisions

**Encoding**

- The most significant bit is the sign bit.
- The exp field encodes \(E\).
- The frac field encodes \(M\).

**Sizes**

- Single precision: 8 exp bits, 23 frac bits, for 32 bits total
- Double precision: 11 exp bits, 52 frac bits, for 64 bits total
- Extended precision: 15 exp bits, 63 frac bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits: an explicit “1” bit appears in the format, except when exp is 0.
**Normalized Numeric Values**

- **Condition:** \( \text{exp} \neq 000\ldots0 \) and \( \text{exp} \neq 111\ldots1 \)

- **Exponent is coded as a biased value**

  \[ E = \text{Exp} - \text{Bias} \]

  - **\( \text{Exp} \):** unsigned value denoted by \( \text{exp} \).
  - **\( \text{Bias} \):** Bias value
    - Single precision: 127 (\( \text{Exp} = 1\ldots254, E = -126\ldots127 \))
    - Double precision: 1023 (\( \text{Exp} = 1\ldots2046, E = -1022\ldots1023 \))
    - In general: \( \text{Bias} = 2^e - 1 \), where \( e \) is the number of exponent bits

- **Significand coded with implied leading 1**

  \( M = 1.xxx\ldots x_2 \)

  - \( xxx\ldots x \): bits of \( \text{frac} \)
  - Minimum when 000\ldots0 \( (M = 1.0) \)
  - Maximum when 111\ldots1 \( (M = 2.0 - \epsilon) \)
  - We get the extra leading bit “for free.”

**Normalized Encoding Example**

- **Value:**
  
  ```
  float F = 15213.0;
  ```

  \( 15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13} \)

- **Significand**

  - \( M = 1.1101101101101_2 \)
  - \( \text{frac} = 11011011011010000000000000 \)

- **Exponent**

  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \text{Exp} = 140 = 10001100 \)

---

**Normalized Encoding Example**

- **Floating Point Representation**

  Hex: 466DB400

  Binary: 0100 0110 0110 1101 1011 0100 0000 0000

  140: 100 0110 0

  15213: 1110 1101 1011 01
Normalized Example

Given the bit string 0x40500000, what floating point number does it represent?

Writing this as a bit string gives us:

0 10000000 10100000000000000000000

We see that this is a positive, normalized number.

\[ \exp = 128 - 127 = 1 \]

So, this number is:

\[ 1.101_2 \times 2^1 = 11.01_2 = 3.25_{10} \]

Denormalized Values

**Condition:** \( \exp = 000\ldots0 \)

**Value**

- Exponent values: \( E = -\text{Bias} + 1 \) Why this value?
- Significand value: \( M = 0.xxx\ldots x_2 \), where \( xxx\ldots x \) are the bits of frac.

**Cases**

- \( \exp = 000\ldots0 \) and \( \text{frac} = 000\ldots0 \)
  - represents values of 0
  - notice that we have distinct +0 and -0
- \( \exp = 000\ldots0 \) and \( \text{frac} \neq 000\ldots0 \)
  - These are numbers very close to 0.0
  - Lose precision as they get smaller
  - Experience “gradual underflow”

Denormalized Example

Given the bit string 0x80600000, what floating point number does it represent?

Writing this as a bit string gives us:

1 00000000 11000000000000000000000

We see that this is a negative, denormalized number with value:

\[ -0.112 \times 2^{-126} = -1.12 \times 2^{-127} \]
Why That Exponent

The exponent (it’s not a bias) for denormalized floats is −126. Why that number?

The smallest positive normalized float is \(1.0_2 \times 2^{-126}\). Where did I get that number? All positive normalized floats are bigger.

The largest positive denormalized float is \(0.1111111111111111111_2 \times 2^{-126}\). Why? All positive denorms are between this number and 0.

Note that the smallest norm and the largest denorm are incredibly close together. Thus, the normalized range flows naturally into the denormalized range because of this choice of exponent for denorms.

Condition: \(\text{exp} = 111\ldots1\)

Cases

- \(\text{exp} = 111\ldots1\) and \(\text{frac} = 000\ldots0\)
  - Represents value of infinity (\(\infty\))
  - Result returned for operations that overflow
  - Sign indicates positive or negative
  - E.g., \(1.0/0.0 = -1.0/0.0 = +\infty, 1.0/(-0.0) = -\infty\)

- \(\text{exp} = 111\ldots1\) and \(\text{frac} \neq 000\ldots0\)
  - Not-a-Number (NaN)
  - Represents the case when no numeric value can be determined
  - E.g., \(\sqrt{-1}, \infty - \infty\)

Not a Number

How many 32-bit NaN’s are there?

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.

This has the general form of the IEEE Format

- Has both normalized and denormalized values.
- Has representations of 0, NaN, infinity.

<table>
<thead>
<tr>
<th>Exp</th>
<th>exp</th>
<th>E</th>
<th>(2^E)</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>-6</td>
<td>1/64</td>
<td>(denorms)</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-6</td>
<td>1/64</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-5</td>
<td>1/32</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-4</td>
<td>1/16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-3</td>
<td>1/8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-2</td>
<td>1/4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-1</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>+1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>+2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>+3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>+4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>+5</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>+6</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>+7</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>n/a</td>
<td>(inf, NaN)</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Range

<table>
<thead>
<tr>
<th>s</th>
<th>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>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>$1/8 \times 1/64 = 1/512$</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>$2/8 \times 1/64 = 2/512$</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>$6/8 \times 1/64 = 6/512$</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>$7/8 \times 1/64 = 7/512$</td>
</tr>
</tbody>
</table>

Denormalized numbers

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>$8/8 \times 1/64 = 8/512$</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>$9/8 \times 1/64 = 9/512$</td>
</tr>
</tbody>
</table>

Normalized numbers

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0110</td>
<td>110</td>
<td>-1</td>
<td>$14/8 \times 1/2 = 14/16$</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>$15/8 \times 1/2 = 15/16$</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>$8/8 \times 1 = 1$</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>$9/8 \times 1 = 9/8$</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>$10/8 \times 1 = 10/8$</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>$14/8 \times 128 = 224$</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>$15/8 \times 128 = 240$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1111</td>
<td>n/a</td>
<td>∞</td>
<td>$2^{127.1023}$</td>
</tr>
</tbody>
</table>

Notice that the floating point numbers are not distributed evenly on the number line.

Special Properties of Encoding

**FP Zero is the Same as Integer Zero:** All bits are 0.

**Can (Almost) Use Unsigned Integer Comparison**

- Must first compare sign bits.
- Must consider $-0 = 0$.
- NaNs are problematic:
  - Will be greater than any other values.
  - What should the comparison yield?
- Otherwise, it's OK.

**Denorm vs. normalized works.**

**Normalized vs. infinity works.**
Floating Point Operations

Conceptual View
- First compute the exact result.
- Make it fit into the desired precision.
  - Possibly overflows if exponent is too large.
  - Possibly round to fit into frac.

Rounding Modes (illustrated with $ rounding)

<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>Toward Zero</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$1</td>
</tr>
<tr>
<td>Round down ($-\infty$)</td>
<td>$1 $1 $1 $2 $-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Round up ($+\infty$)</td>
<td>$2 $2 $2 $3 $-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nearest even (default)</td>
<td>$1 $2 $2 $2 $-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Round down: rounded result is close to but no greater than true result.
- Round up: rounded result is close to but no less than true result.

Default Rounding Mode
- Hard to get any other kind without dropping into assembly.
- All others are statistically biased; the sum of a set of integers will consistently be under- or over-estimated.

Applying to Other Decimal Places / Bit Positions
When exactly halfway between two possible values, round so that the least significant digit is even.

E.g., round to the nearest hundredth:

- 1.2349999 → 1.23
- 1.2350001 → 1.23
- 1.2350000 → 1.24
- 1.2450000 → 1.24

Binary Fractional Numbers
- “Even” when least significant bit is 0.
- Half way when bits to the right of rounding position = 100...2.

Examples
E.g., Round to nearest 1/4 (2 bits to 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>2/32</td>
<td>10.00011₂</td>
<td>10.00</td>
<td>(&lt; 1/2: down)</td>
<td>2</td>
</tr>
<tr>
<td>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>7/8</td>
<td>10.111₀₀₂</td>
<td>11.00</td>
<td>(1/2: up)</td>
<td>3</td>
</tr>
<tr>
<td>5/8</td>
<td>10.101₀₀₂</td>
<td>10.10</td>
<td>(1/2: down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>

When rounding to even, consider the two possible choices and choose the one with a 0 in the final position.

Example: round to even at the 1/4 position:

- 1.10100000 → 1.10100000 (1 5/8)
- 1.11 → 1.11 (1 3/4)
- 1.11 → 1.11 (1 3/4)
- 1.11 → 1.11 (1 5/8)
- 10.00 → 1.11000000 (1 7/8)
- 10.00 → 1.11000000 (1 7/8)
- 2.0
**FP Multiplication**

**Operands:** \((-1)^{S_1} \times M_1 \times 2^{E_1}, (-1)^{S_2} \times M_2 \times 2^{E_2}\)

**Exact Result:** \((-1)^S \times M \times 2^E\)
- Sign S: \(S_1\) xor \(S_2\)
- Significand M: \(M_1 \times M_2\)
- Exponent E: \(E_1 + E_2\)

**Fixing**
- If \(M \geq 2\), shift M right, increment E
- E is out of range, overflow
- Round M to fit frac precision

**Implementation**

Biggest chore is multiplying significands.

---

**Multiplication Examples**

**Decimal Example**

\((-3.4 \times 10^3)(5.2 \times 10^4)\)

\[= -(3.4 \times 5.2)(10^2 \times 10^4)\]

\[= -17.68 \times 10^6\]

\[= -1.768 \times 10^7\]

Adjust exponent

**Binary Example**

\((-1.01 \times 2^2)(1.1 \times 2^4)\)

\[= -(1.01 \times 1.1)(2^2 \times 2^4)\]

\[= -1.111 \times 2^6\]

\[= -10.0 \times 2^6\]

Round to even

\[= -1.0 \times 2^7\]

Adjust exponent

---

**FP Addition**

**Operands:** \((-1)^{S_1} \times M_1 \times 2^{E_1}, (-1)^{S_2} \times M_2 \times 2^{E_2}\)

Assume \(E_1 > E_2\)

**Exact Result:** \((-1)^S \times M \times 2^E\)
- Sign S, Significand M; result of signed align and 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 \(k\)
- If E is out of range, overflow
- Round M to fit frac precision

Be careful if you try to do this in the floating point format, rather than in scientific notation. Since the exponents are biased in FP format, adding them would give you:

\[(2 + \text{bias}) + (4 + \text{bias}) = 6 + 2*\text{bias}\]

To adjust you have to subtract the bias.

If you try to do this in the FP form, recall that both exponents are biased.
**Decimal Example**

\[-3.4 \times 10^2 + (5.2 \times 10^4) = (-3.4 + 520.0) \times 10^2 = 516.6 \times 10^2 \approx 5.166 \times 10^4 = 5.17 \times 10^4\]

**Binary Example**

\[-1.01 \times 2^2 + (1.1 \times 2^4) = (-1.01 + 110.0) \times 2^2 = 100.11 \times 2^2 \approx 1.0011 \times 2^4 = 1.01 \times 2^4\]

**Mathematical Properties of FP Add**

- Closed under addition? Yes, but may generate infinity or NaN.
- Commutative? Yes.
- Associative? No, because of overflow and inexactness of rounding.
- 0 is additive identity? Yes.
- Every element has additive inverse? Almost, except for infinities and NaNs.

**Monotonicity**

\[a \geq b \implies a + c \geq b + c? \text{ Almost, except for infinities and NaNs.}\]

**Mathematical Properties of FP Mult**

- Closed under multiplication? Yes, but may generate infinity or NaN.
- Multiplication Commutative? Yes.
- Multiplication is Associative? No, because of possible overflow and inexactness of rounding.
- 1 is multiplicative identity? Yes.
- Multiplication distributes over addition? No, because of possible overflow and inexactness of rounding.

**Monotonicity**

\[a \geq b & \land c \geq 0 \implies a \times c \geq b \times c? \text{ Almost, except for infinities and NaNs.}\]

**Floating Point in C**

**C guarantees two levels**

- float: single precision
- double: double precision

**Conversions**

- Casting among 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 ≤ 53-bit word size
  - int to float: will round according to rounding mode.
Assume neither d nor f is NaN.

\[
\begin{align*}
\text{int } x &= \ldots; \\
\text{float } f &= \ldots; \quad \text{No: 24 bit significand} \\
\text{double } d &= \ldots; \\
\text{Yes: 53 bit significand} \\
x &= (\text{int})(\text{float}) x \\
\text{Yes: increases precision} \\
x &= (\text{int})(\text{double}) x \\
\text{No: loses precision} \\
f &= (\text{float})(\text{double}) f \\
\text{Yes: just change sign bit} \\
d &= (\text{float}) d \\
\text{No: not associative} \\
f &= -(-f) \\
\text{Yes} \\
2/3 &= 2/3.0 \\
\text{Yes} \\
d < 0.0 & \rightarrow ((d*2) < 0.0) \\
\text{Yes} \\
d > f & \rightarrow -f > -d \\
\text{Yes} \\
d*d >= 0.0 \\
\text{Yes} \\
(d+f)-d &= f \\
\text{No: not associative}
\end{align*}
\]

The cause of the failure was a software error in the inertial reference system.

Specifically, a 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer.

The number was larger than 32,767, the largest integer storeable in a 16-bit signed integer, and thus the conversion failed.

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion.

The destroyed rocket and its cargo were valued at $500 million.

**IEEE Floating Point has Clear Mathematical Properties**

- Represents numbers of the 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 and distributivity.
  - Makes life difficult for compilers and serious numerical application programmers.