Today Topics: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Fractional binary numbers

- What is 1011.101?
Fractional Binary Numbers

■ Representation
  ▪ 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 Numbers: Examples

■ Value                      Representation
  5 and 3/4                   101.11₂
  2 and 7/8                   10.111₂
  63/64                       0.111111₁₂

■ Observations
  ▪ Divide by 2 by shifting right
  ▪ Multiply by 2 by shifting left
  ▪ Numbers of form 0.11111...₂ are just below 1.0
    ▪ 1/2 + 1/4 + 1/8 + ... + 1/2^j + ... → 1.0
    ▪ Use notation 1.0 − ε
Representable Numbers

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101... (2)</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011... (2)</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011... (2)</td>
</tr>
</tbody>
</table>

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 fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard
Floating Point Representation

- **Numerical Form:**
  \((-1)^s M \times 2^E\)
  - Sign bit \(s\) determines whether number is negative or positive
  - Significand (mantissa) \(M\) normally a fractional value in range \([1.0,2.0)\).
  - Exponent \(E\) weights value by power of two

- **Encoding**
  - MSB \(s\) is sign bit \(s\)
  - \(frac\) field encodes \(M\) (but is not equal to \(M\))
  - \(exp\) field encodes \(E\) (but is not equal to \(E\))

```
s exp frac
```

Precisions

- **Single precision: 32 bits**
  ```
s exp frac
  1  8  23
  ```

- **Double precision: 64 bits**
  ```
s exp frac
  1  11  52
  ```

- **Extended precision: 80 bits (Intel only)**
  ```
s exp frac
  1  15  63 or 64
  ```
Normalized Values

- **Condition:** \( \text{exp} \neq 000...0 \text{ and } \text{exp} \neq 111...1 \)

- **Exponent coded as biased value:** \( \text{exp} = E + \text{Bias} \)
  - \( \text{exp} \) is an unsigned value ranging from 1 to \( 2^e - 2 \)
  - Allows negative values for \( E = \text{exp} - \text{Bias} \)
  - \( \text{Bias} = 2^{e-1} - 1 \), where \( e \) is number of exponent bits (bits in \( \text{exp} \))
    - Single precision: 127 (\( \text{exp}: 1...254, E: -126...127 \))
    - Double precision: 1023 (\( \text{exp}: 1...2046, E: -1022...1023 \))

- **Significand coded with implied leading 1:** \( M = 1 . \text{xxx...x}_2 \)
  - \( \text{xxx...x}_2 \): bits of \( \text{frac} \)
  - Minimum when \( 000...0 \) (\( M = 1.0 \))
  - Maximum when \( 111...1 \) (\( M = 2.0 - \varepsilon \))
  - Get extra leading bit for “free”

Normalized Encoding Example

- **Value:** \( \text{Float} \; F = 12345.0; \)
  - \( 12345_{10} = 11000000111001_2 \)
    - \( = 1.1000000111001_2 \times 2^{13} \)

- **Significand**
  - \( M = 1.1000000111001_2 \)
  - \( \text{frac} = 100000011100100000000000_2 \)

- **Exponent**
  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \text{exp} = 140 = 10001100_2 \)

- **Result:**
  - \( \text{Exponent: } [0b10001100] \)
  - \( \text{Significand: } [0b100000011100100000000000000] \)
Denormalized Values

- **Condition:** \( \exp = 000...0 \)

- **Exponent value:** \( E = \exp - \text{Bias} + 1 \) (instead of \( E = \exp - \text{Bias} \))
- **Significand coded with implied leading 0:** \( M = 0 \cdot xxx...x_2 \)
  - \( xxx...x \): bits of \( \text{frac} \)

- **Cases**
  - \( \exp = 000...0, \text{frac} = 000...0 \)
    - Represents value 0
    - Note distinct values: +0 and −0 (why?)
  - \( \exp = 000...0, \text{frac} \neq 000...0 \)
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

Special Values

- **Condition:** \( \exp = 111...1 \)

- **Case:** \( \exp = 111...1, \text{frac} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty \), \( 1.0/-0.0 = -1.0/0.0 = -\infty \)

- **Case:** \( \exp = 111...1, \text{frac} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \text{sqrt}(-1), \infty - \infty, \infty \times 0 \)
Visualization: Floating Point Encodings

![Floating Point Diagram]

Tiny Floating Point Example

```
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
```

- **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

- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity
### Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s_exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0 0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>6/8*1/64 = 6/512</td>
<td>largest denorm</td>
</tr>
<tr>
<td>0 0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td>smallest norm</td>
</tr>
<tr>
<td>0 0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>14/8*1/2 = 14/16</td>
<td>closest to 1 below</td>
</tr>
<tr>
<td>0 0110 110</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td>0 0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td>0 0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
</tr>
<tr>
<td>0 1110 110</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0 1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
<tr>
<td>0 1111 111</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Distribution of Values

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1-1} = 3$

- Notice how the distribution gets denser toward zero.
Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - Bias is 3

![Diagram showing distribution of values]

**Interesting Numbers**

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Zero</strong></td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td><strong>Smallest Pos. Denorm.</strong></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 $\times 10^{-45}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>4.9 $\times 10^{-324}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Largest Denormalized</strong></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 $\times 10^{-38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>2.2 $\times 10^{-308}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Smallest Pos. Norm.</strong></td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \times 2^{-126,1022}$</td>
</tr>
<tr>
<td>Just larger than largest denormalized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>One</strong></td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Largest Normalized</strong></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 $\times 10^{38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>1.8 $\times 10^{308}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Special Properties of Encoding

- **Floating point zero** (0⁺) exactly the same bits as integer zero
  - All bits = 0

- **Can (Almost) UseUnsigned Integer Comparison**
  - Must first compare sign bits
  - Must consider 0⁻ = 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: Basic Idea

- \( x +_f y = \text{Round}(x + y) \)

- \( x *_f y = \text{Round}(x * y) \)

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into frac
Rounding

- **Rounding Modes (illustrate 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>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 (default)</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$-2</td>
</tr>
</tbody>
</table>

- **What are the advantages of the modes?**

Closer Look at Round-To-Nearest

- **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 under-estimated

- **Applying to Other Decimal Places / Bit Positions**
  - 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**
  - “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>10.00₂</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00₁₁₀₂</td>
<td>10.01₂</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.₁₁₀₀₀₂</td>
<td>11.₀₀₂</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.₁₀₁₀₀₂</td>
<td>10.₁₀₂</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>

Floating Point Multiplication

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

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

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

- **Implementation**
  - Biggest chore is multiplying significands
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 $s$, significand $M$:
    - Result of signed align & 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 $k$
  - Overflow if $E$ out of range
  - Round $M$ to fit $\frac{\text{frac}}{}$ precision

---

Mathematical Properties of FP Operations

- Not really associative or distributive due to rounding
- Infinities and NaNs cause issues (e.g., no additive inverse)
- Overflow and infinity
Floating Point in C

- C Guarantees Two Levels
  - float single precision
  - double double precision

- Conversions/Casting
  - Casting 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 long as int has ≤ 53 bit word size
  - int -> float
    - Will round according to rounding mode

Memory Referencing Bug (Revisited)

def fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0)  ->  3.14
fun(1)  ->  3.14
fun(2)  ->  3.1399998664856
fun(3)  ->  2.00000061035156
fun(4)  ->  3.14, then segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Saved State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>
Representing 3.14 as a Double FP Number

- 1073741824 = 0100 0000 0000 0000 0000 0000 0000 0000
- 3.14 = 11.0010 0011 1101 0111 0000 1010 000...
- \((-1)^s M 2^e\)
  - S = 0 encoded as 0
  - M = 1.1001 0001 1110 1011 1000 0101 000.... (leading 1 left out)
  - E = 1 encoded as 1024 (with bias)

<table>
<thead>
<tr>
<th>exp (11)</th>
<th>frac (first 20 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100 0000 0000 1001 0001 1110 1011 1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frac (another 32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101 0000 ...</td>
</tr>
</tbody>
</table>

Memory Referencing Bug (Revisited)

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0)  -> 3.14
fun(1)  -> 3.14
fun(2)  -> 3.1399998664856
fun(3)  -> 2.00000061035156
fun(4)  -> 3.14, then segmentation fault

Saved State

<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
| d7 ... d4 | d3 ... d0 | a[1] | a[0] | Location accessed by fun(i)
Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^e$
- 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