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**

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 and 3/4</td>
<td>101.11₂</td>
</tr>
<tr>
<td>2 and 7/8</td>
<td>10.111₁₂</td>
</tr>
<tr>
<td>63/64</td>
<td>0.11111₁₂</td>
</tr>
</tbody>
</table>

**Observations**

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.11111₁₂... are just below 1.0
  - \( 1/2 + 1/4 + 1/8 + ... + 1/2^i + ... \rightarrow 1.0 \)
  - Use notation 1.0 − \( \varepsilon \)
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>$0.01010101[01]..._2$</td>
</tr>
<tr>
<td>$1/5$</td>
<td>$0.001100110011[0011]..._2$</td>
</tr>
<tr>
<td>$1/10$</td>
<td>$0.0001100110011[0011]..._2$</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:**
  \((-1)^s M 2^E\)
  - 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
1    8    23
```

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:** \( \exp \neq 000\ldots 0 \) and \( \exp \neq 111\ldots 1 \)

- **Exponent coded as biased value:** \( \exp = E + \text{Bias} \)
  - \( \exp \) is an unsigned value ranging from 1 to \( 2^{e-2} \)
    - Allows negative values for \( E ( = \exp - \text{Bias}) \)
  - \( \text{Bias} = 2^{e-1} - 1 \), where \( e \) is number of exponent bits (bits in \( \exp \))
  - Single precision: 127 (\( \exp: 1\ldots254, E: -126\ldots127 \))
  - Double precision: 1023 (\( \exp: 1\ldots2046, E: -1022\ldots1023 \))

- **Significand coded with implied leading 1:** \( M = 1.\text{xxx}\ldots x_2 \)
  - \( \text{xxx}\ldots x \): bits of \( \text{frac} \)
  - Minimum when 000\ldots0 \( (M = 1.0) \)
  - Maximum when 111\ldots1 \( (M = 2.0 - \varepsilon) \)
  - Get extra leading bit for “free”

Normalized Encoding Example

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

- **Significand**
  - \( M = 1.1000000111001 \)
  - \( \text{frac} = 100000011100100000000000_{2} \)

- **Exponent**
  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \text{Exp} = 140 = 10001100_{2} \)

- **Result:**
  - \( 0 \text{ 10001100 100000011100100000000000} \)
    - \( s \text{  exp frac} \)
Denormalized Values

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

- Exponent value: \( E = \text{exp} - \text{Bias} + 1 \) (instead of \( E = \text{exp} - \text{Bias} \))

- Significand coded with implied leading 0: \( M = 0 . \ xxx...x2 \)
  - \( xxx...x \): bits of \( \text{frac} \)

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

Special Values

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

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

-∞ - Normalized  - Denorm  + Denorm  + Normalized  + ∞  NaN  -0  +0

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

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

<table>
<thead>
<tr>
<th>s exp frac E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000 -6</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 001 -6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0 0000 010 -6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0 0000 110 -6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0 0000 111 -6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td>0 0001 000 -6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0 0001 001 -6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0 0110 110 -1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0 0110 111 -1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td>0 0111 000 0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0 0111 001 0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td>0 0111 010 0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0 1110 110 7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0 1110 111 7</td>
<td>15/8*128 = 240</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

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>Exp</th>
<th>Frac</th>
<th>Numeric Value</th>
</tr>
</thead>
</table>
| **Zero**             | 00...00 | 00...00 | 0.0  
| **Smallest Pos. Denorm.** | 00...00 | 00...01 | $2^{- (23,52)} \times 2^{- 126,1022}$  
|                      |      |      | 1.4 * 10^{-45} \text{ Single}  
|                      |      |      | $4.9 \times 10^{-324}$ \text{ Double}  
| **Largest Denormalized** | 00...00 | 11...11 | $(1.0 - \epsilon) \times 2^{- 126,1022}$  
|                      |      |      | $1.18 \times 10^{-38}$ \text{ Single}  
|                      |      |      | $2.2 \times 10^{-308}$ \text{ Double}  
| **Smallest Pos. Norm.**   | 00...01 | 00...00 | 1.0 * 2^{- 126,1022}  
|                     |      |      | Just larger than largest denormalized  
| **One**              | 01...11 | 00...00 | 1.0  
| **Largest Normalized** | 11...10 | 11...11 | $(2.0 - \epsilon) \times 2^{127,1023}$  
|                      |      |      | $3.4 \times 10^{38}$ \text{ Single}  
|                      |      |      | $1.8 \times 10^{108}$ \text{ Double}  

[Diagram showing distribution of values with denormalized, normalized, and infinity markers.]
Special Properties of Encoding

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

- Can (Almost) Use Unsigned 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

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

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

- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
  - Possibly round to fit into $\text{frac}$
Rounding

- **Rounding Modes (illustrate with $ rounding)**

<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 (-(\infty))</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$2</td>
</tr>
<tr>
<td>Round up ((+\infty))</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 = \(100\ldots\)

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>10.00011(_2)</td>
<td>10.00(_2)</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110(_2)</td>
<td>10.01(_2)</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100(_2)</td>
<td>11.00(_2)</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.101000(_2)</td>
<td>10.10(_2)</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>

Floating Point Multiplication

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

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

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

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>s</th>
<th>exp (11)</th>
<th>frac (first 20 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100 0000 0000</td>
<td>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

<table>
<thead>
<tr>
<th>Saved State</th>
<th>4</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>

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