Today’s Topics

- Floating Point Numbers
- 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**
  - 5 and 3/4: \( 101.11_2 \)
  - 2 and 7/8: \( 10.111_2 \)
  - 63/64: \( 0.111111_2 \)

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of the form \( 0.11111..._2 \) are just below 1.0
    - \( 1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0 \)
    - 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

- **Value**
  - **Representation**
  - $1/3 \quad 0.0101010101[01]..._2$
  - $1/5 \quad 0.001100110011[0011]..._2$
  - $1/10 \quad 0.0001100110011[0011]..._2$

---

Fixed Point Representation

- float $\rightarrow$ 32 bits; double $\rightarrow$ 64 bits
- We might try representing fractional binary numbers by picking a fixed place for an implied binary point
  - “fixed point binary numbers”
- Let’s do that, using 8 bit floating point numbers as an example
  - #1: the binary point is between bits 2 and 3
    - $b_7 b_6 b_5 b_4 b_3 [.] b_2 b_1 b_0$
  - #2: the binary point is between bits 4 and 5
    - $b_7 b_6 b_5 [.] b_4 b_3 b_2 b_1 b_0$
  - The position of the binary point affects the range and precision
    - range: difference between largest and smallest numbers possible
    - precision: smallest possible difference between any two numbers
Fixed Point Pros and Cons

- **Pros**
  - It's simple. The same hardware that does integer arithmetic can do fixed point arithmetic
    - In fact, the programmer can use ints with an implicit fixed point
      - E.g., int balance; // number of pennies in the account
    - Ints are just fixed point numbers with the binary point to the right of \( b_0 \)

- **Cons**
  - There is no good way to pick where the fixed point should be
    - Sometimes you need range, sometimes you need precision
    - The more you have of one, the less of the other

What else could we do?
IEEE Floating Point

- Fixing fixed point: analogous to scientific notation
  - Not 12000000 but $1.2 \times 10^7$; not 0.0000012 but $1.2 \times 10^{-6}$
- 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:
  \[ (-1)^s \cdot M \cdot 2^E \]
  - 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 $s$
  - $\text{frac}$ field encodes $M$ (but is not equal to $M$)
  - $\text{exp}$ field encodes $E$ (but is not equal to $E$)
Precisions

- Single precision: 32 bits
  \[
  \begin{array}{c|c|c}
  s & \text{exp} & \text{frac} \\
  1 & 8 & 23 \\
  \end{array}
  \]

- Double precision: 64 bits
  \[
  \begin{array}{c|c|c}
  s & \text{exp} & \text{frac} \\
  1 & 11 & 52 \\
  \end{array}
  \]

- Extended precision: 80 bits (Intel only)
  \[
  \begin{array}{c|c|c}
  s & \text{exp} & \text{frac} \\
  1 & 15 & 63 \text{ or } 64 \\
  \end{array}
  \]

Normalization and Special Values

- “Normalized” means mantissa has form 1.xxxxx
  - 0.011 \times 2^5 and 1.1 \times 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, don't bother to store it

- How do we represent 0.0? How about 1.0/0.0?
Normalization and Special Values

- “Normalized” means mantissa has form 1.xxxxx
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, don’t bother to store it

- Special values:
  - The float value 00...0 represents zero
  - If the exp == 11...1 and the mantissa == 00...0, it represents ±∞
    - E.g., 1.0/0.0 = -1.0/-0.0 = +∞, 1.0/-0.0 = -1.0/0.0 = -∞
  - If the exp == 11...1 and the mantissa != 00...0, it represents NaN
    - “Not a Number”
    - Results from operations with undefined result
      - E.g., sqrt(-1), ∞ - ∞, ∞ * 0

Normalized Values

- Condition: exp ≠ 000...0 and exp ≠ 111...1

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

- Significand coded with implied leading 1: M = 1.xxx...x
  - xxx...x: bits of frac
  - Minimum when 000...0 (M = 1.0)
  - Maximum when 111...1 (M = 2.0 – ε)
  - Get extra leading bit for “free”
Normalized Encoding Example

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

- **Significand**
  - $M = 1.1000000111001$
  - $\frac{frac}{=} = 10000001110010000000000_2$

- **Exponent**
  - $E = 13$
  - $Bias = 127$
  - $exp = 140 = 10001100_2$

- **Result**:

  $\begin{bmatrix}
  \text{s} & \text{exp} & \text{frac} \\
  0 & 10001100 & 100000011100100000000000
  \end{bmatrix}$

How do we do operations?

- **Is representation exact?**
- **How are the operations carried out?**
Floating Point Operations: Basic Idea

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

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

Floating Point Multiplication

\((-1)^{s_1} M_1 \ 2^{E_1} \times (-1)^{s_2} M_2 \ 2^{E_2}\)

Exact Result: \((-1)^s M \ 2^E\)
- Sign \(s\): \(s_1 \oplus s_2\) \(\text{// xor of } s_1 \text{ and } s_2\)
- Significand \(M\): \(M_1 \times M_2\)
- Exponent \(E\): \(E_1 + E_2\)

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

\[ (-1)^{s_1} M_1 \cdot 2^{E_1} + (-1)^{s_2} M_2 \cdot 2^{E_2} \]
Assume \( E_1 > E_2 \)

- **Exact Result:** \((-1)^s M \cdot 2^E\)
  - 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 precision

Hmm... if we round at every operation...
Mathematical Properties of FP Operations

- Not really associative or distributive due to rounding
- Infinities and NaNs cause issues
- 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

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:

Replicating 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...
- \((-1)^s \times M \times 2^E\)
  - $s = 0$ encoded as 0
  - $M = 1.1001 0001 1110 1011 1000 \ldots$ (leading 1 left out)
  - $E = 1$ encoded as 1024 (with bias)

\[
\begin{array}{c|c|c}
  s & \text{exp (11)} & \text{frac (first 20 bits)} \\
  0 & 100 0000 0000 & 1001 0001 1110 1011 1000 \\
\end{array}
\]

\[
\text{frac (the other 32 bits)}
\]

0101 0000 ...
Memory Referencing Bug (Revisited)

```c
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>d7 ... d4</th>
<th>d3 ... d0</th>
<th>a[1]</th>
<th>a[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 0000 0000 1001 0001 1110 1011 1000</td>
<td>0100 0000 ...</td>
<td>0101 0000 ...</td>
<td>0100 0000</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
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

| d7 ... d4 | 0100 0000 0000 0000 0000 0000 0000 | 4 |
| d3 ... d0 | 0101 0000 ... | 2 |
| a[1]      | 0100 0000 0000 0000 0000 0000 0000 | 1 |
| a[0]      | 0100 0000 0000 0000 0000 0000 0000 | 0 |

Location accessed by fun(i)

Floating Point and the Programmer

#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for (i=0; i<10; i++) {
        f2 += 1.0/10.0;
    }
    printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.8f\n", f1);
    printf("f2 = %10.8f\n", f2);
    f1 = 1e30;
    f2 = 1e-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");
    return 0;
}

$ ./a.out
0x3f800000  0x3f800000
f1 = 1.00000000
f2 = 1.000000119
f1 == f3? yes
Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation
    - E.g., 0.1
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”

- Mathematically equivalent ways of writing an expression may compute different results
  - Violates associativity/distributivity

- NEVER test floating point values for equality!

Additional details

- Denormalized values – to get finer precision near zero
- Tiny floating point example
- Distribution of representable values
- Rounding
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:** \( M = 0 \). \( \text{xxx...x} \times 2^E \)
  - \( \text{xxx...x} \): bits of \( \text{frac} \)

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

Special Values

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

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

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 \( \text{frac} \)

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

```
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```

![Distribution of Values](image)

### Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Zero</strong></td>
<td>00...00</td>
<td>00...00</td>
</tr>
<tr>
<td><strong>Smallest Pos. Denorm.</strong></td>
<td>00...00</td>
<td>00...01</td>
</tr>
<tr>
<td>Single</td>
<td>$1.4 \times 10^{-45}$</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$4.9 \times 10^{-324}$</td>
<td></td>
</tr>
<tr>
<td><strong>Largest Denormalized</strong></td>
<td>00...00</td>
<td>11...11</td>
</tr>
<tr>
<td>Single</td>
<td>$1.18 \times 10^{-38}$</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$2.2 \times 10^{-308}$</td>
<td></td>
</tr>
<tr>
<td><strong>Smallest Pos. Norm.</strong></td>
<td>00...01</td>
<td>00...00</td>
</tr>
<tr>
<td></td>
<td>Just larger than largest denormalized</td>
<td></td>
</tr>
<tr>
<td><strong>One</strong></td>
<td>01...11</td>
<td>00...00</td>
</tr>
<tr>
<td><strong>Largest Normalized</strong></td>
<td>11...10</td>
<td>11...11</td>
</tr>
<tr>
<td>Single</td>
<td>$3.4 \times 10^{38}$</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$1.8 \times 10^{308}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Numeric Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.0$</td>
<td><strong>Zero</strong></td>
</tr>
<tr>
<td>$2^{-23,52} \times 2^{-126,1022}$</td>
<td><strong>Single</strong>, <strong>Double</strong></td>
</tr>
<tr>
<td>$(1.0 - \epsilon) \times 2^{-126,1022}$</td>
<td><strong>Largest Denormalized</strong></td>
</tr>
<tr>
<td>$1.0 \times 2^{-126,1022}$</td>
<td><strong>Smallest Pos. Norm.</strong></td>
</tr>
<tr>
<td>$1.0$</td>
<td><strong>One</strong></td>
</tr>
<tr>
<td>$(2.0 - \epsilon) \times 2^{127,1023}$</td>
<td><strong>Largest Normalized</strong></td>
</tr>
</tbody>
</table>

*Autumn 2012*

*Three Floating Point Numbers*
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

Rounding

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

<table>
<thead>
<tr>
<th>Value</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 = \(100\ldots2\)

- Examples
  - Round to nearest 1/4 (2 bits right of binary point)
    | Value | Binary   | Rounded   | Action          | Rounded Value |
    |-------|----------|-----------|-----------------|---------------|
    | 2 3/32 | 10.000112 | 10.002    | (<1/2—down)    | 2             |
    | 2 3/16 | 10.001102 | 10.012    | (>1/2—up)      | 2 1/4         |
    | 2 7/8  | 10.111002 | 11.002    | (1/2—up)       | 3             |
    | 2 5/8  | 10.101002 | 10.102    | (1/2—down)     | 2 1/2         |