Today 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** | **Representation**
  - 5 and 3/4 | 101.11₂
  - 2 and 7/8 | 10.11₁₂
  - 63/64 | 0.11111₁₂

**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
  - \( 1/2 + 1/4 + 1/8 + ... + 1/2^i + ... \rightarrow 1.0 \)
  - 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

- **Value**
  - **Representation**
    - $1/3$  $0.01010101[01]_{-2}$
    - $1/5$  $0.001100110011[0011]_{-2}$
    - $1/10$  $0.0001100110011[0011]_{-2}$

---

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 x 10^7; not 0.0000012 but 1.2 x 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:**
  \((-1)^s \times M \times 2^E\)
  - 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 \(s\)
  - \(\text{frac}\) field encodes \(M\) (but is \textit{not equal} to \(M\))
  - \(\text{exp}\) field encodes \(E\) (but is \textit{not equal} to \(E\))
Precisions

- **Single precision: 32 bits**
  
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>23</td>
</tr>
</tbody>
</table>

- **Double precision: 64 bits**
  
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>52</td>
</tr>
</tbody>
</table>

- **Extended precision: 80 bits (Intel only)**
  
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>63 or 64</td>
</tr>
</tbody>
</table>

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

- 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 $\infty$
  - E.g., 1.0/0.0 = $-1.0/-0.0 = +\infty$, 1.0/-0.0 = $-1.0/0.0 = -\infty$

- **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($-1$), $\infty - \infty$, $\infty \times 0$

Normalized Values

- **Condition:** $\text{exp} \neq 000...0$ 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 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:** $M = 1 . \text{xxx...x}_2$
  - $\text{xxx...x}$: bits of $\text{frac}$
  - Minimum when 000...0 ($M = 1.0$)
  - Maximum when 111...1 ($M = 2.0 - \epsilon$)
  - 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**
  - \( M = 1.1000000111001_2 \)
  - \( \text{frac} = 1000000111001_2 \times 0000000000_2 \)

- **Exponent**
  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \text{exp} = 140 = 1001100_2 \)

- **Result**:

\[
\begin{array}{ll}
\text{s} & \text{exp} & \text{frac} \\
0 & \text{1001100} & \text{10000001110011000000000000} \\
\end{array}
\]

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

$(-1)^{s_1} M_1 2^{E_1} * (-1)^{s_2} M_2 2^{E_2}$

Exact Result: $(-1)^s M 2^E$

- Sign $s$: $s_1 \lor s_2$ // xor of $s_1$ and $s_2$
- Significand $M$: $M_1 * M_2$
- 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

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

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

- Not really associative or distributive due to rounding
- Infinites 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 $\rightarrow$ int
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: generally sets to TMin
  - int $\rightarrow$ double
    - Exact conversion, as long as int has $\leq$ 53-bit word size
  - int $\rightarrow$ 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:

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>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)

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...
- \((-1)^F \times M \times 2^E\)
  - 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>

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d7</td>
<td>d6</td>
<td>d5</td>
<td>d4</td>
<td>d3</td>
<td>d2</td>
<td>d1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2</td>
<td>d1</td>
<td>d0</td>
<td>d9</td>
<td>d8</td>
<td>d7</td>
<td>d6</td>
</tr>
<tr>
<td>0100 0000 0000 1001 0001 1110 1011 1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101 0000 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d7</td>
<td>d6</td>
<td>d5</td>
<td>d4</td>
<td>d3</td>
<td>d2</td>
<td>d1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2</td>
<td>d1</td>
<td>d0</td>
<td>d9</td>
<td>d8</td>
<td>d7</td>
<td>d6</td>
</tr>
<tr>
<td>0100 0000 0000 1001 0001 1110 1011 1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101 0000 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></th>
<th>d7</th>
<th>d4</th>
<th>d3</th>
<th>d0</th>
<th>a[1]</th>
<th>a[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0100</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td>0000</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;
}
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:** $\exp = 000...0$

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

- **Significand coded with implied leading 0:** $M = 0 \cdot \text{xxx...x}_2$
  - $\text{xxx...x}$: bits of $\text{frac}$

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

Special Values

- **Condition:** $\exp = 111...1$

- **Case:** $\exp = 111...1, \frac{}{} = 000...0$
  - 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:** $\exp = 111...1, \frac{}{} \neq 000...0$
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., $\sqrt{-1}, \infty - \infty, \infty \times 0$
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 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></td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 001</td>
<td></td>
<td>-6</td>
<td>$\frac{1}{8}\times\frac{1}{64} = \frac{1}{512}$ lowest norm</td>
</tr>
<tr>
<td>0 0000 010</td>
<td></td>
<td>-6</td>
<td>$\frac{2}{8}\times\frac{1}{64} = \frac{2}{512}$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0000 110</td>
<td></td>
<td>-6</td>
<td>$\frac{6}{8}\times\frac{1}{64} = \frac{6}{512}$ largest denorm</td>
</tr>
<tr>
<td>0 0000 111</td>
<td></td>
<td>-6</td>
<td>$\frac{7}{8}\times\frac{1}{64} = \frac{7}{512}$</td>
</tr>
<tr>
<td>0 0001 000</td>
<td></td>
<td>-6</td>
<td>$\frac{8}{8}\times\frac{1}{64} = \frac{8}{512}$ smallest norm</td>
</tr>
<tr>
<td>0 0001 001</td>
<td></td>
<td>-6</td>
<td>$\frac{9}{8}\times\frac{1}{64} = \frac{9}{512}$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0110 110</td>
<td>1</td>
<td>-1</td>
<td>$\frac{14}{8}\times\frac{1}{2} = \frac{14}{16}$ closest to 1 below</td>
</tr>
<tr>
<td>0 0110 111</td>
<td>1</td>
<td>-1</td>
<td>$\frac{15}{8}\times\frac{1}{2} = \frac{15}{16}$</td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0</td>
<td></td>
<td>$\frac{8}{8}\times1 = 1$</td>
</tr>
<tr>
<td>0 0111 001</td>
<td>0</td>
<td></td>
<td>$\frac{9}{8}\times1 = \frac{9}{8}$ closest to 1 above</td>
</tr>
<tr>
<td>0 0111 010</td>
<td>0</td>
<td></td>
<td>$\frac{10}{8}\times1 = \frac{10}{8}$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1110 110</td>
<td>7</td>
<td></td>
<td>$\frac{14}{8}\times128 = 224$ largest norm</td>
</tr>
<tr>
<td>0 1110 111</td>
<td>7</td>
<td></td>
<td>$\frac{15}{8}\times128 = 240$</td>
</tr>
<tr>
<td>0 1111 000</td>
<td>n/a</td>
<td></td>
<td>inf</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

![](image)

---

**Interesting Numbers**

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>{single, double}</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>00...01</td>
<td></td>
<td>$2^{-23} \cdot 2^{-126,1022}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Single</td>
<td></td>
<td></td>
<td>$1.4 \times 10^{-45}$</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td></td>
<td></td>
<td>$4.9 \times 10^{-324}$</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td></td>
<td>$(1.0 - \epsilon) \cdot 2^{-126,1022}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Single</td>
<td></td>
<td></td>
<td>$1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td></td>
<td></td>
<td>$2.2 \times 10^{-308}$</td>
</tr>
<tr>
<td>Smallest Pos. Norm.</td>
<td>00...01</td>
<td>00...00</td>
<td></td>
<td>$1.0 \cdot 2^{-126,1022}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Just larger than largest denormalized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td>01...11</td>
<td>00...00</td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td></td>
<td>$(2.0 - \epsilon) \cdot 2^{127,1023}$</td>
</tr>
</tbody>
</table>
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></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)
    | Value | Binary | Rounded | Action     | Rounded Value |
    |-------|--------|---------|------------|---------------|
    | 2 3/32| 10.00 111₂ | 10.00₂  | (<1/2—down) | 2             |
    | 2 3/16| 10.00 110₂ | 10.01₂  | (>1/2—up)   | 2 1/4         |
    | 2 7/8 | 10.11 100₂ | 11.00₂  | (1/2—up)    | 3             |
    | 2 5/8 | 10.10 100₂ | 10.10₂  | (1/2—down)  | 2 1/2         |