Floating Point

15-213/18-213/15-513: Introduction to Computer Systems
4th Lecture, Sept. 7, 2017

Today’s Instructor:
Phil Gibbons
Today: 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 $1011.101_2$?
**Fractional Binary Numbers**

- **Representation**
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number:
    \[ \sum_{k=-j}^{i} b_k \times 2^k \]
## Fractional Binary Numbers: Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3/4</td>
<td>101.11₂</td>
<td>$5 \frac{3}{4} = \frac{23}{4} = 4 + \frac{1}{2} + \frac{1}{4}$</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.111₂</td>
<td>$2 \frac{7}{8} = \frac{23}{8} = 2 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}$</td>
</tr>
<tr>
<td>1 7/16</td>
<td>1.0111₂</td>
<td>$1 \frac{7}{16} = \frac{23}{16} = 1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}$</td>
</tr>
</tbody>
</table>

### Observations

- Divide by 2 by shifting right (unsigned)
- Multiply by 2 by shifting left
- Numbers of form $0.111111...₂$ are just below 1.0
  - $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^i} + ... \rightarrow 1.0$
- Use notation $1.0 - \epsilon$
Representable Numbers

- **Limitation #1**
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - **Value** | **Representation**
    - $1/3$ | $0.0101010101\ldots_2$
    - $1/5$ | $0.001100110011\ldots_2$
    - $1/10$ | $0.0001100110011\ldots_2$

- **Limitation #2**
  - Just one setting of binary point within the $w$ bits
    - Limited range of numbers (very small values? very large?)
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
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
  - Some CPUs don’t implement IEEE 754 in full
    - e.g., early GPUs, Cell BE processor

- **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 \( M \) normally a fractional value in range \([1.0, 2.0)\).
  - Exponent \( E \) weights value by power of two

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

Example:
\[ 15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13} \]
Precision options

- **Single precision:** 32 bits
  \(\approx 7\) decimal digits, \(10^{\pm38}\)

- **Double precision:** 64 bits
  \(\approx 16\) decimal digits, \(10^{\pm308}\)

- **Other formats:** half precision, quad precision
Three “kinds” of floating point numbers

00...00

denormalized

exp ≠ 0 and exp ≠ 11...11

normalized

11...11

special
“Normalized” Values

- When: \(\exp \neq 000...0\) and \(\exp \neq 111...1\)

- Exponent coded as a biased value: \(E = \exp - \text{Bias}\)
  - \(\exp\): unsigned value of exp field
  - \(\text{Bias} = 2^{k-1} - 1\), where \(k\) is number of exponent bits
    - Single precision: 127 (\(\exp\): 1...254, E: -126...127)
    - Double precision: 1023 (\(\exp\): 1...2046, E: -1022...1023)

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

- **Value:** float $F = 15213.0$;
  - $15213_{10} = 1110110110110112$
  - $= 1.11011011011012 \times 2^{13}$

- **Significand**
  - $M = 1.11011011011012$
  - $frac = 11011011011010000000000002$

- **Exponent**
  - $E = 13$
  - $Bias = 127$
  - $exp = 140 = 100011002$

- **Result:**
  - $v = (-1)^s \ M \ 2^E$
  - $E = exp - Bias$

```
\begin{array}{c|c|c}
\hline
s & \text{exp} & \text{frac} \\
\hline
0 & 10001100 & 110110110110100000000000000 \\
\hline
\end{array}
```
Denormalized Values

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

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

- **Significand coded with implied leading 0:** $M = 0.xxx...x_2$
  - $xxx...x$: bits of $\text{frac}$

- **Cases**
  - $\text{exp} = 000...0, \text{frac} = 000...0$
    - Represents zero value
    - Note distinct values: +0 and −0 (why?)
  - $\text{exp} = 000...0, \text{frac} \neq 000...0$
    - Numbers closest to 0.0
    - Equispaced

\[ v = (-1)^s M \cdot 2^E \]
\[ E = 1 - \text{Bias} \]
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 = $−\infty$

- **Case:** exp = 111...1, frac ≠ 000...0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(−1), $\infty - \infty$, $\infty \times 0$
C float Decoding Example

float: 0xC0A00000

binary: ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___

1 8-bits 23-bits

E =

S =

M =

\( \nu = (-1)^s \ M \ 2^E \) =

\( E = \exp - Bias \)

\( Bias = 2^{k-1} - 1 = 127 \)
C float Decoding Example

float: 0xC0A00000

binary: \[ \begin{array}{cccccccccccc}
1 & 1000 & 0000 & 1 & 010 & 0000 & 0000 & 0000 & 0000 & 0000 \\
\end{array} \]

\[ \begin{array}{c}
1 \\
\end{array} \quad \begin{array}{c}
8\text{-bits} \\
\end{array} \quad \begin{array}{c}
23\text{-bits} \\
\end{array} \]

\[ E = \]
\[ S = \]
\[ M = 1. \]

\[ v = (-1)^s M 2^E = \]
C float Decoding Example

float: 0xC0A00000

binary: 1100 0000 1010 0000 0000 0000 0000 0000

1 1000 0001 010 0000 0000 0000 0000 0000

1 8-bits 23-bits

\[ E = \exp - \Bias = 129 - 127 = 2 \text{ (decimal)} \]

\[ S = 1 \rightarrow \text{negative number} \]

\[ M = 1.010 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \]

\[ = 1 + \frac{1}{4} = 1.25 \]

\[ v = (-1)^S \ M \ 2^E = (-1)^1 \ 1.25 \cdot 2^2 = -5 \]
Visualization: Floating Point Encodings

-∞  −Normalized  −Denorm  +Denorm  +Normalized  +∞

NaN  −0  +0  NaN

NaN
Today: Floating Point

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

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the $\text{exp}$, 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 (s=0 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>(-1)^0 (0+1/4) *2^-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0000 011</td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0000 110</td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0000 111</td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0001 000</td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0001 001</td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0110 110</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0110 111</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0111 000</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0111 001</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0111 010</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1110 110</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1110 111</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1111 000</td>
<td>n/a</td>
</tr>
</tbody>
</table>

\[
v = (-1)^s M \cdot 2^E
\]

**Norm:**
\[
E = \exp - \text{Bias}
\]

**Denorm:**
\[
E = 1 - \text{Bias}
\]
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.

- 8 values

- Denormalized ▲ Normalized ■ Infinity
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-bits</td>
<td>2-bits</td>
</tr>
</tbody>
</table>
```

- Denormalized
- Normalized
- Infinity
Special Properties of the IEEE Encoding

- **FP Zero Same as Integer Zero**
  - All bits = 0

- **Can (Almost) Use Unsigned Integer Comparison**
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield? The answer is complicated.
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/1221
Today: Floating Point

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

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

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>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↑</td>
</tr>
<tr>
<td>Round down (−∞)</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>Round up (+∞)</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>Nearest Even* (default)</td>
<td>↓</td>
<td>↑</td>
<td>↑</td>
<td>↓</td>
<td>↓</td>
</tr>
</tbody>
</table>

*Round to nearest, but if half-way in-between then round to nearest even*
Closer Look at Round-To-Even

■ Default Rounding Mode
  ▪ Hard to get any other kind without dropping into assembly
  ▪ C99 has support for rounding mode management
  ▪ 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
    7.8949999  7.89  (Less than half way)
    7.8950001  7.90  (Greater than half way)
    7.8950000  7.90  (Half way—round up)
    7.8850000  7.88  (Half way—round down)
Rounding Binary Numbers

- **Binary Fractional Numbers**
  - “Even” when least significant bit is 0
  - “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>10.111002</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>
Rounding

1. BBGRXXX

Guard bit: LSB of result

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

Round up conditions

- Round = 1, Sticky = 1 $\rightarrow$ > 0.5
- Guard = 1, Round = 1, Sticky = 0 $\rightarrow$ Round to even

<table>
<thead>
<tr>
<th>Value</th>
<th>Fraction</th>
<th>GRS</th>
<th>Incr?</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.0000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>15</td>
<td>1.1010000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>17</td>
<td>1.0001000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>19</td>
<td>1.0011000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>138</td>
<td>1.0001010</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.1111100</td>
<td>111</td>
<td>Y</td>
<td>10.000</td>
</tr>
</tbody>
</table>
FP 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 \(s\): \(s_1 \wedge s_2\)
  - 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 \texttt{frac} precision

- **Implementation**
  - Biggest chore is multiplying significands

\[
\begin{align*}
\text{4 bit mantissa: } & 1.010 \times 2^2 \times 1.110 \times 2^3 = 10.0011 \times 2^5 \\
& = 1.00011 \times 2^6 = 1.001 \times 2^6
\end{align*}
\]
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 \(\frac{\text{fraction}}{\text{precision}}\)

Get binary points lined up

\[
\begin{align*}
(-1)^{s_1} M_1 + (-1)^{s_2} M_2 & \quad \text{Exponent} \quad E_1 - E_2 \\
\end{align*}
\]

\[
\begin{align*}
(-1)^{s_1} M_1 & \quad + \\
\end{align*}
\]

\[
\begin{align*}
(-1)^{s_2} M_2 & \quad \text{Result of signed align & add} \\
\end{align*}
\]

\[
\begin{align*}
(-1)^s M & \quad \text{Sign, significand} \\
\end{align*}
\]

\[
\begin{align*}
\text{Overflow if } E & \text{ out of range} \\
\text{Round } M & \text{ to fit } \frac{\text{fraction}}{\text{precision}}
\end{align*}
\]

1.010*2^2 + 1.110*2^3 = (0.1010 + 1.1100)*2^3
= 10.0110 * 2^3 = 1.00110 * 2^4 = 1.010 * 2^4
Mathematical Properties of FP Add

- **Compare to those of Abelian Group**
  - Closed under addition? \(\text{Yes}\)
    - But may generate infinity or NaN
  - Commutative? \(\text{Yes}\)
  - Associative? \(\text{No}\)
    - Overflow and inexactness of rounding
    - \((3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14\)
  - 0 is additive identity? \(\text{Yes}\)
  - Every element has additive inverse? \(\text{Almost}\)
    - Yes, except for infinities & NaNs

- **Monotonicity**
  - \(a \geq b \Rightarrow a+c \geq b+c?\) \(\text{Almost}\)
    - Except for infinities & NaNs
Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication? Yes
    - But may generate infinity or NaN
  - Multiplication Commutative? Yes
  - Multiplication is Associative? No
    - Possibility of overflow, inexactness of rounding
    - Ex: \((1e20*1e20)*1e-20=\text{inf}, 1e20*(1e20*1e-20)=1e20\)
  - 1 is multiplicative identity? Yes
  - Multiplication distributes over addition? No
    - Possibility of overflow, inexactness of rounding
    - \(1e20*(1e20-1e20)=0.0, 1e20*1e20 - 1e20*1e20 = \text{NaN}\)

- **Monotonicity**
  - \(a \geq b \land c \geq 0 \Rightarrow a \cdot c \geq b \cdot c\)? Almost
    - Except for infinities & NaNs
Today: Floating Point

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

For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

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

Assume neither d nor f is NaN

- \( x == (\text{int})(\text{float}) x \)
- \( x == (\text{int})(\text{double}) x \)
- \( f == (\text{float})(\text{double}) f \)
- \( d == (\text{double})(\text{float}) d \)
- \( f == -(\text{negate\ f}); \)
- \( 2/3 == 2/3.0 \)
- \( d < 0.0 \Rightarrow ((d*2) < 0.0) \)
- \( d > f \Rightarrow -f > -d \)
- \( d \times d >= 0.0 \)
- \( (d+f)-d == f \)
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
Additional Slides
Creating Floating Point Number

- **Steps**
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding

- **Case Study**
  - Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
</tr>
<tr>
<td>15</td>
<td>00001101</td>
</tr>
<tr>
<td>33</td>
<td>00010001</td>
</tr>
<tr>
<td>35</td>
<td>00010011</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
</tr>
</tbody>
</table>
Normalize

■ Requirement

- Set binary point so that numbers of form 1.xxxxx
- Adjust all to have leading one
  - Decrement exponent as shift left

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Fraction</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
<td>1.0000000</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>00001101</td>
<td>1.1010000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.0001000</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.0011000</td>
<td>4</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.0001010</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.1111100</td>
<td>5</td>
</tr>
</tbody>
</table>
### Issue
- Rounding may have caused overflow
- Handle by shifting right once & incrementing exponent

<table>
<thead>
<tr>
<th>Value</th>
<th>Rounded</th>
<th>Exp</th>
<th>Adjusted</th>
<th>Numeric Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.000</td>
<td>7</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>1.101</td>
<td>3</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>1.000</td>
<td>4</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>1.010</td>
<td>4</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>138</td>
<td>1.001</td>
<td>7</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td>63</td>
<td>10.000</td>
<td>5</td>
<td>1.000/6</td>
<td>64</td>
</tr>
</tbody>
</table>
### Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>00...01</td>
<td>$2^{-{23,52}} \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>- Single ≈ $1.4 \times 10^{-45}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Double ≈ $4.9 \times 10^{-324}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>- Single ≈ $1.18 \times 10^{-38}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Double ≈ $2.2 \times 10^{-308}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Normalized</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>One</td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>$(2.0 - \varepsilon) \times 2^{127,1023}$</td>
</tr>
<tr>
<td>- Single ≈ $3.4 \times 10^{38}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Double ≈ $1.8 \times 10^{308}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>