Floating Point

15-213: Introduction to Computer Systems
4th Lecture, Sept. 8, 2016

Today’s Instructor:
Randy Bryant
Correction from last time
Unsigned Multiplication in C

Operands: \( w \) bits

True Product: \( 2^w \) bits

Discard \( w \) bits: \( w \) bits

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits

- **Implements Modular Arithmetic**
  \[
  \text{UMult}_w(u, v) = u \cdot v \mod 2^w
  \]

\[
\begin{array}{cccc}
1110 & 1001 & E9 & 223 \\
* & 1101 & 0101 & * D5 * 213 \\
--- & 1100 & 0001 & C1DD 47499 \\
1101 & 1101 & DD & 221 \\
\end{array}
\]
Signed Multiplication in C

Operands: $w$ bits

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits
  - *Some of which are different for signed vs. unsigned multiplication*
  - Lower bits are the same

\[
\begin{array}{c}
\text{1111 1111 1110 1001} \\
\times \text{1111 1111 1101 0101} \\
\text{0000 0011 1101 1101}
\end{array}
\]

\[
\begin{array}{c}
\text{E9} \\
\times \text{D5} \\
\text{03DD}
\end{array}
\]

\[
\begin{array}{c}
\text{1101 1101} \\
\text{DD}
\end{array}
\]

\[
\begin{array}{c}
\text{989} \\
\text{989}
\end{array}
\]

\[
\begin{array}{c}
\text{1101 1101} \\
\text{DD}
\end{array}
\]

\[
\begin{array}{c}
\text{4} \\
\text{4}
\end{array}
\]

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**

- 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

- **Value**
  - $5 \frac{3}{4} = 23/4$
  - $2 \frac{7}{8} = 23/8$
  - $1 \frac{7}{16} = 23/16$

- **Representation**
  - $101.11_2 = 4 + 1 + 1/2 + 1/4$
  - $10.111_2 = 2 + 1/2 + 1/4 + 1/8$
  - $1.0111_2 = 1 + 1/4 + 1/8 + 1/16$

**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
  - $1/2 + 1/4 + 1/8 + ... + 1/2^i + ... \Rightarrow 1.0$
- Use notation $1.0 - \varepsilon$
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[01]...$_2$
  - 1/5 | 0.001100110011[0011]...$_2$
  - 1/10 | 0.0001100110011[0011]...$_2$

- 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

- 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 \( s \)
  - \( \text{exp} \) field encodes \( E \) (but is not equal to \( E \))
  - \( \text{frac} \) field encodes \( M \) (but is not equal to \( M \))

Example:
\[ 15213_{10} = (-1)^0 \times 1.11011011011012 \times 2^{13} \]
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
“Normalized” Values

■ When: exp ≠ 000...0 and exp ≠ 111...1

■ Exponent coded as a biased value: \( E = \text{Exp} - \text{Bias} \)
  - Exp: unsigned value of exp field
  - 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: \( M = 1.xxx...x \)
  - xxx...x: bits of frac field
  - Minimum when frac=000...0 (M = 1.0)
  - Maximum when frac=111...1 (M = 2.0 - \( \varepsilon \))
  - Get extra leading bit for “free”
Normalized Encoding Example

- **Value**: \( \text{float } F = 15213.0; \)
  - \( 15213_{10} = 11101101101101_2 \)
  - \( = 1.1101101101101_2 \times 2^{13} \)

- **Significand**
  \( M = \quad 1.1101101101101_2 \)
  \( frac = \quad 11011011011010000000000000_2 \)

- **Exponent**
  \( E = \quad 13 \)
  \( Bias = \quad 127 \)
  \( Exp = \quad 140 = \quad 10001100_2 \)

- **Result**:
  \[
  \begin{array}{cccccc}
  & s & \text{exp} & \text{frac} \\
 0 & 10001100 & 1101101101101000000000000000 \\
  \end{array}
  \]

\[ v = (-1)^s M \times 2^E \]
\[ E = \text{Exp} - \text{Bias} \]
Denormalized Values

- Condition: exp = 000...0

- Exponent value: E = 1 – Bias (instead of 0 – Bias)

- Significand coded with implied leading 0: M = 0.xxx...x2
  - xxx...x: bits of frac

- Cases
  - exp = 000...0, frac = 000...0
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - exp = 000...0, frac ≠ 000...0
    - Numbers closest to 0.0
    - Equispaced

\[ v = (-1)^s \, M \, 2^E \]
\[ E = 1 - \text{Bias} \]
Special Values

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

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

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

float: 0xC0A00000

binary: ______ ______ ______ ______ ______ ______ ______ ______

1 8-bits 23-bits

E = -> Exp = (decimal)

S =

M = 1.25

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

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

float: \texttt{0xC0A00000}

binary: \texttt{1100 0000 1010 0000 0000 0000 0000 0000}

\begin{array}{c|c|c}
1 & 1000 0001 & 010 0000 0000 0000 0000 0000 0000 \\
\hline
1 & 8-bits & 23-bits
\end{array}

E = \rightarrow \text{Exp} = \quad \text{(decimal)}

S =

M = 1.

v = \left( -1 \right)^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 = 129 -> Exp = 129 – 127 = 2 (decimal)

S = 1 -> negative number

M = 1.010 0000 0000 0000 0000 0000 0000

= 1 + 1/4 = 1.25

v = (−1)^s M 2^E = (−1)^1 * 1.25 * 2^2 = -5

Bias = 2^{k-1} – 1 = 127
Visualization: Floating Point Encodings

\[ \begin{align*}
-\infty & \quad -\text{Normalized} & \quad -\text{Denorm} & \quad +\text{Denorm} & \quad +\text{Normalized} & \quad +\infty \\
\text{NaN} & \quad -0 & \quad +0 & \quad \text{NaN}
\end{align*} \]
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 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</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest denorm</td>
</tr>
<tr>
<td>0</td>
<td>0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>smallest norm</td>
</tr>
<tr>
<td>0</td>
<td>0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 below</td>
</tr>
<tr>
<td>0</td>
<td>0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>closest to 1 above</td>
</tr>
<tr>
<td>0</td>
<td>0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>largest norm</td>
</tr>
<tr>
<td>0</td>
<td>1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
</tbody>
</table>

\[ v = (-1)^s M 2^E \]
\[ n: E = \text{Exp} - \text{Bias} \]
\[ d: E = 1 - \text{Bias} \]
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>

-1 -0.5 0 0.5 1

- 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
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>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 Even (default)</td>
<td>$1 ↓</td>
<td>$2 ↑</td>
<td>$2 ↑</td>
<td>$2 ↓</td>
<td>−$2 ↓</td>
</tr>
</tbody>
</table>
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>
**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 `frac` precision

**Implementation**
- Biggest chore is multiplying significands

```
4 bit mantissa: 1.010*2^2 \times 1.110*2^3 = 10.0011*2^5
             = 1.00011*2^6 = 1.001*2^6
```
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

\[
\begin{align*}
(–1)^{s_1} M_1 & \quad + \quad (–1)^{s_2} M_2 \\
& \quad + \quad (–1)^s M
\end{align*}
\]

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? Yes
    - But may generate infinity or NaN
  - Commutative? Yes
  - Associative? No
    - Overflow and inexactness of rounding
  - $3.14 + 1e10 - 1e10 = 0$, $3.14 + (1e10 - 1e10) = 3.14$
  - 0 is additive identity? Yes
  - Every element has additive inverse? Almost
    - Yes, except for infinities & NaNs

- Monotonicity
  - $a \geq b \Rightarrow a+c \geq b+c$? 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**
  - \( a \geq b \text{ and } c \geq 0 \Rightarrow a * c \geq b * c? \)
    - Except for infinities & NaNs
    - Almost
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 == (int)(float) x` ✗
- `x == (int)(double) x` ✓
- `f == (float)(double) f` ✓
- `d == (double)(float) d` ✗
- `f == -(-f);` ✓
- `2/3 == 2/3.0` ✗
- `d < 0.0 ⇒ ((d*2) < 0.0)` ✓
- `d > f ⇒ -f > -d` ✓
- `d * 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
HOW ROUNding WORKS
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>Number</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>1.00000000</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>00001101</td>
<td>1.10100000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.00010000</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.00110000</td>
<td>4</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.00010100</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.11111000</td>
<td>5</td>
</tr>
</tbody>
</table>
Rounding

Guard bit: LSB of result
Round bit: 1st bit removed

Sticky bit: OR of remaining bits

Round up conditions
- Round = 1, Sticky = 1 → > 0.5
- Guard = 1, Round = 1, Sticky = 0 → 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>
Postnormalize

**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>128</td>
<td></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>
Additional Slides
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></td>
<td></td>
<td></td>
<td>Single ≈ $1.4 \times 10^{-45}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Double ≈ $4.9 \times 10^{-324}$</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></td>
<td></td>
<td></td>
<td>Single ≈ $1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Double ≈ $2.2 \times 10^{-308}$</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></td>
<td></td>
<td></td>
<td>Just larger than largest denormalized</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></td>
<td></td>
<td></td>
<td>Single ≈ $3.4 \times 10^{38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Double ≈ $1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>