Data Representation: Floating Point Numbers

Csci 2021 - Machine Architecture and Organization
Professor Pen-Chung Yew

With sides from Randy Bryant and Dave O’Hallaron
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 $1011.101_2$?
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

- **Value**            **Representation**
  - 5 3/4              101.11₂
  - 2 7/8              10.111₂
  - 1 7/16             1.0111₂

- **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 + ... + 1/2ⁱ + ... → 1.0
  - Use notation 1.0 − ε
Representable Numbers

■ Limitation #1
  ▪ Can only exactly represent numbers of the form $x/2^k$
    ▪ Other rational numbers may have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101[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>

■ Limitation #2
  ▪ Just one setting of binary point within the $w$ bits
    ▪ Limited range of numbers (very small values? very large?)
Floating Point Numbers

- 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 CPU vendors

- **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
Announcement 2/3/2016

- Prof. Yew’s office hour on Friday 2/5/2016 has been extended by ½ hour, i.e. from 10am-11:30am
- Data Lab due 11:55pm, next Monday 2/8/2016
  - Check Data Lab Forum for common Q&As, or post your problems there.
- Homework Assignment #1 has been issued on Monday 2/1/2016
  - Download from Moodle class web page
  - Due date before class Wednesday 2/10/2016 (check class schedule)
Review: Representable Numbers

- **Limitation #1**
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers may have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1/3$</td>
<td>$0.0101010101[01]..._2$</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>

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

- **Numerical Form:**
  \[ (-1)^s \times M \times 2^E \]
  - 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**
  - Most significant bit (MSB) is sign bit \( s \)
  - exp field encodes \( E \) (but is not equal to \( E \))
  - frac field encodes \( M \) (but is not equal to \( M \))
  - Three different encoding schemes: **(1) Normalized** (\( E \neq 000\ldots000, E \neq 111\ldots1111 \)), **(2) Denormalized** (\( E = 000\ldots000 \)), **(3) Not-a-Number (NaN)** (\( E = 1111\ldots111 \))
Precision options

- Single precision: 32 bits
  - Sign (s): 1 bit
  - Exponent (exp): 8 bits
  - Fraction (frac): 23 bits

- Double precision: 64 bits
  - Sign (s): 1 bit
  - Exponent (exp): 11 bits
  - Fraction (frac): 52 bits

- Extended precision: 80 bits (Intel only, internal to hardware)
  - Sign (s): 1 bit
  - Exponent (exp): 15 bits
  - Fraction (frac): 63 or 64 bits
(1) “Normalized” Values

- **When**: exp ≠ 000...0 and exp ≠ 111...1

- **Exponent coded as a biased value**: Exp = E + Bias
  - **Exp**: unsigned value of exp field
  - **Bias** = $2^{k-1} - 1$, where $k$ is the 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 − ε)
  - Get extra leading bit for “free”

$v = (-1)^s \ M \ 2^E$
Normalized Encoding Example

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

- **Significand**
  - $M = 1.1101101101101_{2}$
  - $frac = 11011011011010000000000000_{2}$

- **Exponent**
  - $E = 13$
  - $Bias = 127$
  - $Exp = 140 = 10001100_{2}$

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

```
0 10001100 11011011011010000000000000
```
(2) Denormalized Values

- Condition: \( \text{exp} = 000\ldots0 \)

- Exponent value: \( \text{E} = 1 - \text{Bias} \) (instead of \( \text{E} = 0 - \text{Bias} \))

- Significand coded with implied leading 0: \( \text{M} = 0.xxx\ldotsx_2 \)
  - \( xxx\ldotsx \): bits of \( \text{frac} \)
  - Allows \textit{gradual underflow}

- Cases
  - \( \text{exp} = 000\ldots0, \text{frac} = 000\ldots0 \)
    - Represents \textit{zero} value
    - Note distinct values: \( +0 \) and \( -0 \) (why?)
  - \( \text{exp} = 000\ldots0, \text{frac} \neq 000\ldots0 \)
    - Numbers closest to 0.0
    - Equi-spaced

\[
v = (-1)^s \text{M} 2^E
\]
\[
E = 1 - \text{Bias}
\]
(3) 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 (\( \text{NaN} \))
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \ \infty - \infty, \ \infty \times 0 \)
Visualization: Floating Point Encodings

-∞ −∞ +∞
−Normalized −Denorm +Denorm +Normalized
NaN −0 +0 NaN
Floating Point Numbers

- 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
- Sign bit is in the most significant bit
- Next four bits are the exponent, with a bias of 7
- Last three bits are the $\text{frac}$

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

### Denormalized numbers

<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></td>
<td>0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
</tbody>
</table>

### Normalized numbers

<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>0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0111 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
</tbody>
</table>

### Denormalized numbers

<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>1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
n: E = \text{Exp} - \text{Bias} = \text{Exp} - 7
\]

\[
d: E = 1 - \text{Bias} = 1 - 7 = -6
\]
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.

```
s      exp      frac
1      3-bits  2-bits
```

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

Diagram showing distribution of values with denormalized, normalized, and infinity symbols.
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?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity
    - Negative is smaller than positive (not so in 2’s complement)
Floating Point Numbers

- 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>$1.4$</th>
<th>$1.6$</th>
<th>$1.5$</th>
<th>$2.5$</th>
<th>$-1.5$</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 Even (${\text{default}}$)</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 – Round to Even
  - Need to use assembly programming to get other rounding modes
  - 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

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

- **Exact Result:** \((-1)^s M \times 2^E\)
  - Sign s: \(s_1 \times s_2\)
  - 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 \(\text{frac}\) 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 \(E\): \(E_1\) (the larger of \(E_1\) and \(E_2\))

- **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

Get binary points lined up
Review: Floating Point Encodings

Diagram of floating point encodings:

- $\infty$
- $\sim\infty$
- Normalized
- Denorm
- +Denorm
- +Normalized

NaN

-0
+0
Review: 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
Review: A Close-up View

- **6-bit IEEE-like format**
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - Bias is $3$

```
1 3-bits
2-bits
```

- Denormalized
- Normalized
- Infinity
### Denormalized numbers

<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></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></td>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 011</td>
<td>-6</td>
<td>3/8*1/64 = 3/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 100</td>
<td>-5</td>
<td>4/8*1/64 = 4/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 101</td>
<td>-4</td>
<td>5/8*1/64 = 5/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 110</td>
<td>-3</td>
<td>6/8*1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000 111</td>
<td>-2</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
</tbody>
</table>

**Closest to zero**

### Normalized numbers

<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>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>0011 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
</tbody>
</table>

**Closest to 1 below**

**Closest to 1 above**

**Largest norm**
Review: 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 \(E\): \(E_1\) (the larger of \(E_1\) and \(E_2\))

- **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

Get binary points lined up

\[E_1 - E_2\]
Mathematical Properties of FP Add

- **Mathematical Properties**
  - 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? Yes, except for infinities & NaNs
    - Almost

- **Monotonicity**
  - $a \geq b \Rightarrow a + c \geq b + c$? Almost
    - Except for infinities & NaNs
Mathematical Properties of FP Mult

- **Mathematical Properties**
  - Multiplication Commutative? Yes
  - Multiplication is Associative? No
    - Possibility of overflow, inexactness of rounding
      - Ex: \((1e20 \times 1e20) \times 1e-20 = \inf, 1e20 \times (1e20 \times 1e-20) = 1e20\)
  - 1 is multiplicative identity? Yes
  - Multiplication distributes over addition? No
    - Possibility of overflow, inexactness of rounding
      - \(1e20 \times (1e20 - 1e20) = 0.0, 1e20 \times 1e20 - 1e20 \times 1e20 = \text{NaN}\)

- **Monotonicity**
  - \(a \geq b \ & c \geq 0 \Rightarrow a \times c \geq b \times c?\) Almost
    - Except for infinities & NaNs
Floating Point Numbers

- 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**
  - **Type 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
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
### 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 $\approx 1.4 \times 10^{-45}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Double $\approx 4.9 \times 10^{-324}$</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 $\approx 1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Double $\approx 2.2 \times 10^{-308}$</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 $\approx 3.4 \times 10^{38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Double $\approx 1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>

![Graph of Floating Point Numbers](chart.png)
Machine-Level Data Representation (Done) to Machine-Level Program Representation (Next)