Inets and Floating Point
CSE 351 Winter 2017

http://xkcd.com/899/
Administrivia

- Lab 1 due Friday
  - How is it going?
- HW 1 out today
  - Numerical representation and executable inspection
Using Shifts and Masks

- Extract the 2\textsuperscript{nd} most significant \textit{byte} of an \texttt{int}:
  - First shift, then mask: \((x\gg 16) \& \ 0\text{xFF}\)

<table>
<thead>
<tr>
<th>(x)</th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x\gg 16)</td>
<td>00000000 00000000 00000001 00000010</td>
</tr>
<tr>
<td>(0\text{xFF})</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
<tr>
<td>((x\gg 16) &amp; \ 0\text{xFF})</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- Extract the sign bit of a signed int:
  - First shift, then mask: \((x \gg 31) \& 0x1\)
    - Assuming arithmetic shift here, but works in either case
    - Need mask to clear 1s possibly shifted in

\[
\begin{array}{|c|c|}
\hline
x & 00000001 00000010 00000011 00000100 \\
\hline
x \gg 31 & 00000000 00000000 00000000 00000000 \\
\hline
0x1 & 00000000 00000000 00000000 00000000 00000001 \\
\hline
(x \gg 31) \& 0x1 & 00000000 00000000 00000000 00000000 00000000 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
x & 10000001 00000010 00000011 00000100 \\
\hline
x \gg 31 & 11111111 11111111 11111111 11111111 \\
\hline
0x1 & 00000000 00000000 00000000 00000000 00000001 \\
\hline
(x \gg 31) \& 0x1 & 00000000 00000000 00000000 00000000 00000001 \\
\hline
\end{array}
\]
Using Shifts and Masks

- **Conditionals as Boolean expressions**
  - For `int x`, what does `(x<<31)>>31` do?

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x=!!123</td>
<td>00000000 00000000 00000000 00000001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x&lt;&lt;31</td>
<td>10000000 00000000 00000000 00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x&lt;&lt;31)&gt;&gt;31</td>
<td>11111111 11111111 11111111 11111111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!x</td>
<td>00000000 00000000 00000000 00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!x&lt;&lt;31</td>
<td>00000000 00000000 00000000 00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(!x&lt;&lt;31)&gt;&gt;31</td>
<td>00000000 00000000 00000000 00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Can use in place of conditional:
  - In C: `if(x) {a=y; } else {a=z; }` equivalent to `a=x?y:z;
  - `a=((x<<31)>>31)&y) | (((!x<<31)>>31)&z);`
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
- Shifting and arithmetic operations
- Multiplication
Multiplication

- What do you get when you multiply $9 \times 9$?

- What about $2^{10} \times 2^{20}$?
Unsigned Multiplication in C

**Operands:**
- **w** bits

**True Product:**
- **2w** bits
- \( u \cdot v \)

**Discard w bits:**
- **w** bits
- \( \text{UMult}_w(u, v) \)

- Standard Multiplication Function
  - Ignores high order **w** bits
- Implements Modular Arithmetic
  - \( \text{UMult}_w(u, v) = u \cdot v \mod 2^w \)
Multiplication with shift and add

- Operation \( u << k \) gives \( u \times 2^k \)
  - Both signed and unsigned

**Operands:** \( w \) bits

**True Product:** \( w + k \) bits

**Discard \( k \) bits:** \( w \) bits

- **Examples:**
  - \( u << 3 \) \( \equiv u \times 8 \)
  - \( u << 5 - u << 3 \) \( \equiv u \times 24 \)
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
Number Representation Revisited

- What can we represent in one word?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses

- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. 6.02×10^{23})
  - Very small numbers (e.g. 6.626×10^{-34})
Fractional Binary Numbers

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
8 & 4 & 2 & 1 \\
\frac{1}{2} & \frac{1}{4} & \frac{1}{8} \\
2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

\[
8 + 2 + 1 + \frac{1}{2} + \frac{1}{8} = \frac{11.625}{2} = 5.625
\]
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

- **Value**
  - 5.75
  - $\frac{21}{16}$ and $\frac{7}{8}$

- **Binary:**
  - $0.111111..._2$
  - $0.10111..._2$
  - $5 + \frac{7}{8} = \frac{41}{8} = \frac{49}{16} = 0.111111..._2$
  - $\frac{5}{8} + \frac{1}{2} + \frac{1}{4} = \frac{101111...}{2^n}$
Fractional Binary Numbers

- **Value**
  - Binary: 101.11₂
    - 5.75
    - 2 and 7/8

- **Observations**
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form 0.111111…₂ are just below 1.0
    - 1/2 + 1/4 + 1/8 + ... + 1/2ⁱ + ... → 1.0
    - Use notation 1.0 − ε
Limits of Representation

- Limitations:
  - Even given an arbitrary number of bits, can only **exactly** represent numbers of the form $x \times 2^y$ (y can be negative)
  - Other rational numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1/3 = 0.333333\ldots_{10}$</td>
<td>$0.01010101[01]\ldots_2$</td>
</tr>
<tr>
<td>$1/5 = 0.001100110011\ldots$</td>
<td>$0.001100110011[0011]\ldots_2$</td>
</tr>
<tr>
<td>$1/10 = 0.0001100110011\ldots$</td>
<td>$0.0001100110011[0011]\ldots_2$</td>
</tr>
</tbody>
</table>
Fixed Point Representation

- Binary point has a fixed position
  - Position = number of binary digits before and after
- Implied binary point. Two example schemes:
  - #1: the binary point is between bits 2 and 3
    \[b_7 b_6 b_5 b_4 \; . \; b_2 b_1 b_0\]
  - #2: the binary point is between bits 4 and 5
    \[b_7 b_6 b_5 \; . \; b_4 b_3 b_2 b_1 b_0\]
- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision
- Fixed point = fixed range and fixed precision
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers
- Hard to pick how much you need of each!
- How do we fix this?

“Rarely” used in practice. Not built-in.
Scientific Notation (Decimal)

- **Normalized form**: exactly one digit (non-zero) to left of decimal point

- Alternatives to representing $1/1,000,000,000$
  - Normalized: $1.0 \times 10^{-9}$
  - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$
Scientific Notation (Binary)

- Computer arithmetic that supports this called **floating point** due to the “floating” of the binary point
  - Declare such variable in C as **float**
IEEE Floating Point

- IEEE 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - Scientists/numerical analysts want them to be as real as possible
  - Engineers want them to be easy to implement and fast
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - Float operations can be an order of magnitude slower than integer ops
Floating Point Representation

- Numerical form:

\[ V_{10} = (-1)^s \times M \times 2^E \]

- 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 a (possibly negative) power of two
Floating Point Representation

- **Numerical form:**
  \[ V_{10} = (-1)^s \times M \times 2^E \]
  - 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 a (possibly negative) power of two

- **Representation in memory:**
  - 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 \( M \))
Precisions

- **Single precision:** 32 bits

  - `s exp frac`
  - 1 bit 8 bits 23 bits

- **Double precision:** 64 bits

  - `s exp frac`
  - 1 bit 11 bits 52 bits

- Finite representation means not all values can be represented exactly. Some will be approximated.
Normalization and Special Values

\[ V = (-1)^S \times M \times 2^E \]

- "Normalized" = \( M \) has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 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, we don't bother to store it

- How do we represent 0.0?
  Or special or undefined values like 1.0/0.0?
Normalization and Special Values

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

- “Normalized” = \( M \) has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 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, we don't bother to store it.

- Special values (“denormalized”):
  - **Zero (0):** \( \text{exp} = 00...0, \text{frac} = 00...0 \)
  - **\( +\infty, -\infty \):** \( \text{exp} = 11...1, \text{frac} = 00...0 \)
    - \( 1.0/0.0 = -1.0/-0.0 = +\infty \)
    - \( 1.0/-0.0 = -1.0/0.0 = -\infty \)
  - **NaN (“Not a Number”):** \( \text{exp} = 11...1, \text{frac} \neq 00...0 \)
    - Results from operations with undefined result:
      - \( \sqrt{-1}, \infty-\infty, \infty\times0, \ldots \)
  - **Note:** exp=11...1 and exp=00...0 are reserved, limiting exp range...
Normalized Values

\[ V = (-1)^S \times M \times 2^E \]

- **Condition:** \( exp \neq 000\ldots0 \) and \( exp \neq 111\ldots1 \)
- **Exponent coded as biased value:** \( E = exp - Bias \)
  - \( exp \) is an *unsigned* value ranging from 1 to \( 2^{k-2} \) (\( k = \) # bits in \( exp \))
  - \( Bias = 2^{k-1} - 1 \)
    - Single precision: \( 127 \) (so \( exp: 1\ldots254, E: -126\ldots127 \))
    - Double precision: \( 1023 \) (so \( exp: 1\ldots2046, E: -1022\ldots1023 \))
  - These enable negative values for \( E \), for representing very small values
Normalized Values

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

- **Condition:** \( \text{exp} \neq 000\ldots0 \) and \( \text{exp} \neq 111\ldots1 \)
- **Exponent coded as biased value:** \( E = \text{exp} - \text{Bias} \)
  - \( \text{exp} \) is an *unsigned* value ranging from 1 to \( 2^{k-2} \) (\( k \) == # bits in \( \text{exp} \))
  - \( \text{Bias} = 2^{k-1} - 1 \)
    - Single precision: 127 (so \( \text{exp}: 1\ldots254, \ E: -126\ldots127 \))
    - Double precision: 1023 (so \( \text{exp}: 1\ldots2046, \ E: -1022\ldots1023 \))
  - These enable negative values for \( E \), for representing very small values
    - Could have encoded with 2’s complement or sign-and-magnitude
    - This just made it easier for HW to do float-exponent operations
- **Mantissa coded with implied leading 1:** \( M = 1.xxx\ldots x_2 \)
  - \( xxx\ldots x \): the \( n \) bits of frac
  - Minimum when 000\ldots0 \( (M = 1.0) \)
  - Maximum when 111\ldots1 \( (M = 2.0 - \varepsilon) \)
  - Get extra leading bit for “free”
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.

- Denormalized  ▲ Normalized  ▼ Infinity
Floating Point Operations

- Unlike the representation for integers, the representation for floating-point numbers is not exact.
- We have to know how to round from the real value.
Floating Point Operations: Basic Idea

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

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

**Basic idea for floating point operations:**

- First, **compute the exact result**
- Then, **round** the result to make it fit into desired precision:
  - Possibly overflow if exponent too large
  - Possibly drop least-significant bits of mantissa to fit into frac
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 \( s \), mantissa \( M \):
    - 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
Floating Point Multiplication

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

- **Exact Result:** \((-1)^s \times M \times 2^E\)
  - **Sign** \(s\): \(s_1 \oplus s_2\)
  - **Mantissa** \(M\): \(M_1 \times M_2\)
  - **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
Rounding modes

Possible rounding modes (illustrated with dollar 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>Round-toward-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>Round-to-nearest</td>
<td>$1</td>
<td>$2</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>Round-to-even</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$-2</td>
</tr>
</tbody>
</table>

Round-to-even avoids statistical bias in repeated rounding.
- Rounds up about half the time, down about half the time.
- Default rounding mode for IEEE floating-point
Mathematical Properties of FP Operations

- Exponent overflow yields $+\infty$ or $-\infty$

- Floats with value $+\infty$, $-\infty$, and NaN can be used in operations
  - Result usually still $+\infty$, $-\infty$, or NaN; sometimes intuitive, sometimes not

- Floating point ops do not work like real math, due to rounding!
  - Not associative: $(3.14 + 1e100) - 1e100 \neq 3.14 + (1e100 - 1e100)$
  - Not distributive: $100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2$
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing
Floating Point in C

- C offers two (well, 3) levels of precision
  
  float 1.0f single precision (32-bit)
  double 1.0 double precision (64-bit)
  long double 1.0L (double double, quadruple, or "extended") precision (64-128 bits)

- #include <math.h> to get INFINITY and NAN constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  - Just avoid them!
Floating Point in C

- Conversions between data types:
  - Casting between int, float, and double changes the bit representation.
  - int → float
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - int → double or float → double
    - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
  - long → double
    - Rounded or exact, depending on word size (64-bit → 52 bit mantissa ⇒ round)
  - double or float → int
    - Truncates fractional part (rounded toward zero)
      - E.g. 1.999 → 1, -1.99 → -1
    - “Not defined” when out of range or NaN: generally sets to Tmin (even if the value is a very big positive)
Number Representation Really Matters

- 1991: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- 1996: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- 2000: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- 2038: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- other related bugs
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
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 = %.10f\n", f1);
    printf("f2 = %.10f\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 0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes
Q&A: THE PENTIUM FDIV BUG  
(floating point division)

Q: What do you get when you cross a Pentium PC with a research grant?
A: A mad scientist.

Q: Complete the following word analogy:
   Add is to Subtract as Multiply is to:
   1) Divide
   2) ROUND
   3) RANDOM
   4) On a Pentium, all of the above
A: Number 4.

Q: What algorithm did Intel use in the Pentium's floating point divider?
A: "Life is like a box of chocolates."
(Source: F. Gump of Intel)

Q: According to Intel, the Pentium conforms to the IEEE standards 754 and 854 for floating point arithmetic. If you fly in aircraft designed using a Pentium, what is the correct pronunciation of "IEEE"?
A: Aaaaaaiiiiiiiiiieeeeeeee!
(Source: http://www.columbia.edu/~sss31/rainbow/pentium.jokes.html)

http://www.smbc-comics.com/?id=2999
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., 0.2)
  - 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!
- Careful when converting between ints and floats!
More details for the curious. These slides expand on material covered today

- Tiny Floating Point Example
- Distribution of Values
Visualization: Floating Point Encodings

-∞ - ∞

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

-0 +0 NaN NaN
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 exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 000</td>
<td>0 0000 001</td>
<td>0 0000 010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0 0000 110</td>
<td>0 0000 111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0 0001 000</td>
<td>0 0001 001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0 0110 110</td>
<td>0 0110 111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0 0111 001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0 0111 010</td>
<td>0 0111 010</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0 1110 110</td>
<td>0 1110 111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

- Denormalized numbers
- Normalized numbers
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

![Diagram showing distribution of values with a 6-bit format, including denormalized, normalized, and infinity values.](image-url)
## 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 (\approx 1.4 \times 10^{-45})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Double (\approx 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 (\approx 1.18 \times 10^{-38})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Double (\approx 2.2 \times 10^{-308})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Norm.</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>
</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