Today

- **How’s Lab 1 going?**
  - Some of these are tricky (for us too)
  - Riley has office hours today 12-1pm in CSE 002
  - Section Friday after lecture
  - I’ll be in my office much of the afternoon today, office hours Friday afternoon

- **General feedback has been helpful to me; keep it coming.**

- **Today:**
  - integer review
  - floating point (briefly)
  - machine programming?

- **Textbook shirt!**
4-bit Unsigned vs. Two’s Complement

\[ 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1 \]

\[ -2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1 \]
4-bit Unsigned vs. Two’s Complement

1 0 1 1

\[2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1\]

- \[2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1\]

(math) difference = 16 = 2^4

-5
4-bit Unsigned vs. Two’s Complement

1 0 1 1

\[ 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1 \]

\[ -2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1 \]

(math) difference = 16 = \(2^4\)

-5
8-bit representations

00001001 10000001

11111111 00100111

C: casting between unsigned and signed just reinterprets the same bits.
Sign Extension

0 0 1 0  4-bit  2

0 0 0 0 0 0 1 0  8-bit  2

1 1 0 0  4-bit  -4

?? ?? ? 1 1 0 0  8-bit  -4
Sign Extension

0 0 1 0  4-bit 2
0 0 0 0 0 0 1 0  8-bit 2
1 1 0 0  4-bit -4
0 0 0 0 1 1 0 0  8-bit 12
Sign Extension

\[
\begin{array}{c|c}
4-bit & 8-bit \\
\hline
0010 & 00000010 \\
1100 & 10001100
\end{array}
\]
Sign Extension

0 0 1 0 4-bit 2
0 0 0 0 0 0 1 0 8-bit 2

1 1 0 0 4-bit -4
1 1 1 1 1 1 0 0 8-bit -4
Overflow/Wrapping: Unsigned

addition: drop the carry bit

15
+ 2
\[\text{1111}\]
+ 0010
\[\text{10001}\]

Modular Arithmetic
Overflow/Wrapping: Two’s Complement

addition: drop the carry bit

\[
\begin{array}{c c c c c c}
-1 & 1111 \\
+2 & + 0010 \\
\hline
1 & 10001 \\
\end{array}
\]

\[
\begin{array}{c c c c c c}
6 & 0110 \\
+3 & + 0011 \\
\hline
9 & 1001 \\
\end{array}
\]

Modular Arithmetic
Shifting and Arithmetic

\[
x = 27; \\
y = x \ll 2; \\
y == 108
\]

overflow

\[
x/2^n \\
\text{logical shift right:} \\
\text{shift in zeros from the left}
\]

\[
0 0 0 1 1 0 1 1 \\
\]

\[
x*2^n
\]

\[
0 0 0 1 1 0 1 1 0 0
\]

logical shift left:
shift in zeros from the right

\[
x/2^n \\
\text{unsigned}
\]

\[
im
\]

\[
x = 237; \\
y = x \gg 2; \\
y == 59
\]

rounding (down)
Shifting and Arithmetic

signed
x = -101;
y = x << 2;
y == 108

overflow

x/2^n
arithmetic shift right:
shift in copies of most significant bit from the left

x*2^n

logical shift left:
shift in zeros from the right

y = x >> 2;
y == -5

clarification from Mon.: shifts by n < 0 or n >= word size are undefined
**Roadmap**

### C:

```c
   car *c = malloc(sizeof(car));
   c->miles = 100;
   c->gals = 17;
   float mpg = get_mpg(c);
   free(c);
```

### Java:

```java
   Car c = new Car();
   c.setMiles(100);
   c.setGals(17);
   float mpg = c.getMPG();
```

### Assembly language:

```
get_mpg:
   pushq  %rbp
   movq   %rsp, %rbp
   ...
   popq   %rbp
   ret
```

### Machine code:

```
0111010000011000 100011010000010000000010
1000100111000010
1100000111111101000011111
```

### OS:

- Windows 8
- Mac

### Computer system:

- Intel Core i5
- RAM
- SSD

### Data & addressing
- Integers & floats

### Machine code & C
- x86 assembly programming
- Procedures & stacks

### Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation

### Java vs. C
Floating point topics

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover.
  - It’s a 58-page standard...
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

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 and 3/4</td>
<td>101.11&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>2 and 7/8</td>
<td>10.111&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>47/64</td>
<td>0.101111&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

## Observations

- Shift left = multiply by power of 2
- Shift right = divide by power of 2
- Numbers of the form 0.11111...<sub>2</sub> are just below 1.0

## Limitations:

- Exact representation possible only for numbers of the form \( x \times 2^y \)
- Other rational numbers have repeating bit representations
  - \( 1/3 = 0.33333..._{10} = 0.01010101[01]..._{2} \)
Fixed Point Representation

- **Implied binary point.** Examples:
  
  #1: the binary point is between bits 2 and 3
  \[ b_7 b_6 b_5 b_4 b_3 \cdots b_2 b_1 b_0 \]
  #2: the binary point is between bits 4 and 5
  \[ b_7 b_6 b_5 \cdots b_4 b_3 b_2 b_1 b_0 \]

- **Same hardware as for integer arithmetic.**
  
  #3: integers! the binary point is after bit 0
  \[ b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \cdots \]

- **Fixed point = fixed range and fixed precision**
  
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers
IEEE Floating Point

- Analogous to scientific notation
  - 12000000 \(1.2 \times 10^7\) C: 1.2e7
  - 0.0000012 \(1.2 \times 10^{-6}\) C: 1.2e-6

- IEEE Standard 754 used by all major CPUs today

- Driven by numerical concerns
  - Rounding, overflow, underflow
  - Numerically well-behaved, but hard to make fast in hardware
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 \( s \)
  - \( \text{exp} \) field encodes \( E \) (but is \textit{not equal} to \( E \))
  - \( \text{frac} \) field encodes \( M \) (but is \textit{not equal} to \( M \))

\[ \begin{array}{ccc}
\text{s} & \text{exp} & \text{frac} \\
\end{array} \]
Precisions

- Single precision: 32 bits
  
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>8 bits</td>
<td>23 bits</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 bit</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

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 / 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
  ▪ $0.011 \times 2^5$ 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.

■ Special values:
  ▪ zero: $s = 0$  $\exp = 00...0$  $\frac{a}{} = 00...0$
  ▪ $+\infty, -\infty$: $\exp = 11...1$  $\frac{a}{} = 00...0$

\[
1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -1.0/0.0 = -\infty
\]

▪ NaN (“Not a Number”): $\exp = 11...1$  $\frac{a}{} != 00...0$
  Results from operations with undefined result: $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$, etc.

▪ note: $\exp=11...1$ and $\exp=00...0$ are reserved, limiting $\exp$ range...
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 significand to fit into frac
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 operations are not always associative or distributive, due to rounding!
  - $(3.14 + 1e10) - 1e10 \neq 3.14 + (1e10 - 1e10)$
  - $1e20 * (1e20 - 1e20) \neq (1e20 * 1e20) - (1e20 * 1e20)$
Floating Point in C

- C offers two levels of precision
  
  float    single precision (32-bit)
  double   double precision (64-bit)

- `#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; overflow not possible
  - int → double or float → double
    - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
  - long int → double
    - Rounded or exact, depending on word size
  - double or float → int
    - Truncates fractional part (rounded toward zero)
    - Not defined when out of range or NaN: generally sets to Tmin
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 $T_{\text{Min}}$ in 2038

- **other related bugs**
  - 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)
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!
Many more details for the curious...

- Exponent bias
- Denormalized values – to get finer precision near zero
- Distribution of representable values
- Floating point multiplication & addition algorithms
- Rounding strategies

- We won’t be using or testing you on any of these extras in 351.
HW 1 will be posted this afternoon

- due Wednesday, July 10.
- Integers
- Floats (simple stuff)
- Disassembly
Normalized Values

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

- **Condition:** \( \text{exp} \neq 000...0 \) and \( \text{exp} \neq 111...1 \)
- **Exponent coded as biased value:** \( E = \text{exp} - \text{Bias} \)
  - \( \text{exp} \) is an *unsigned* value ranging from 1 to \( 2^{k-2} \) (\( k = \# \text{ bits in exp} \))
  - \( \text{Bias} = 2^{k-1} - 1 \)
    - Single precision: 127 (so \( \text{exp}: 1...254, E: -126...127 \))
    - Double precision: 1023 (so \( \text{exp}: 1...2046, E: -1022...1023 \))
    - These enable negative values for \( E \), for representing very small values

- **Significand coded with implied leading 1:** \( M = 1 . \text{xxx}...\text{x} \)
  - \( \text{xxx}...\text{x} \): the \( n \) 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

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

- **Value:** \( \text{float } f = 12345.0; \)
  - \( 12345_{10} = 11000000111001_2 \)
    - \( = 1.1000000111001_2 \times 2^{13} \) (normalized form)

- **Significand:**
  - \( M = 1.1000000111001_2 \)
  - \( \text{frac} = 10000001110010000000000000_2 \)

- **Exponent:** \( E = \text{exp} - \text{Bias} \), so \( \text{exp} = E + \text{Bias} \)
  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \text{exp} = 140 = 10001100_2 \)

- **Result:**
  - \( 0 \text{0001100 10000001110010000000000000} \)
Denormalized Values

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

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

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

- Condition: \( \text{exp} = 111...1 \)

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

- Case: \( \text{exp} = 111...1, \ \text{frac} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty - \infty, \infty \times 0 \)
Visualization: Floating Point Encodings
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 \texttt{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</td>
<td>000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>$1/8 \times 1/64 = 1/512$</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>$2/8 \times 1/64 = 2/512$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>$6/8 \times 1/64 = 6/512$</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>$7/8 \times 1/64 = 7/512$</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>$8/8 \times 1/64 = 8/512$</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>$9/8 \times 1/64 = 9/512$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>110</td>
<td>-1</td>
<td>$14/8 \times 1/2 = 14/16$</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>$15/8 \times 1/2 = 15/16$</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>$8/8 \times 1 = 1$</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>$9/8 \times 1 = 9/8$</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>$10/8 \times 1 = 10/8$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>$14/8 \times 128 = 224$</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>$15/8 \times 128 = 240$</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

- **Denormalized numbers**: Closest to zero.
- **Normalized numbers**: Closest to 1 below and above.
- **Denormalized numbers**: Largest denorm, smallest norm.
- **Normalized numbers**: Largest norm.
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

- Denormalized
- Normalized
- Infinity
# 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>$2^{-{23,52}} \ast 2^{-{126,1022}}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \ast 2^{-{126,1022}}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Norm.</td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \ast 2^{-{126,1022}}$</td>
</tr>
<tr>
<td></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>$(2.0 - \varepsilon) \ast 2^{127,1023}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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
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: \(s_1 \ xor \ s_2\)  // xor of s1 and s2
  - 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
Closer Look at Round-To-Even

- 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)

<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>10.00011_2</td>
<td>10.00_2</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110_2</td>
<td>10.01_2</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100_2</td>
<td>11.00_2</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100_2</td>
<td>10.10_2</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>
#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\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 = 1.0000000000
f2 = 1.000000119
f1 == f3? yes
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:

<table>
<thead>
<tr>
<th>Saved State</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
Representing 3.14 as a Double FP Number

- $1073741824 = 0100 0000 0000 0000 0000 0000 0000 0000$
- $3.14 = 11.0010 0011 1101 0111 0000 1010 000...$
- $(-1)^s M 2^E$
  - $S = 0$ 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)

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  

\[
\begin{array}{cccccccccc}
& & & & & & & & & \\
& & & & & & & & & \\
& & & & & & & & & \\
& & & & & & & & & \\
\text{d7} & \ldots & \text{d4} & & & & & & & \\
0100 & 0000 & 0000 & 1001 & 0001 & 1110 & 1011 & 1000 & \\
3 & & & & & & & & \\
\text{d3} & \ldots & \text{d0} & & & & & & & \\
0101 & 0000 & & & & & & & \\
2 & & & & & & & & \\
\text{a[1]} & & & & & & & & & \\
& & & & & & & & & \\
\text{a[0]} & & & & & & & & & \\
& & & & & & & & & \\
0 & & & & & & & & & \\
\end{array}
\]  

Location accessed by fun(i)
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>d6</th>
<th>d5</th>
<th>d4</th>
<th>d3</th>
<th>d2</th>
<th>d1</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0100</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>1001</td>
<td>0001</td>
<td>1110</td>
<td>1011</td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
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

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<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>
<td>d0</td>
<td>a[1]</td>
<td>a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0101</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location accessed by fun(i)