Today: Floats!

Fractional binary numbers
- What is 1011/101?

Fractional Binary Numbers
- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: $\frac{\sum b_i 2^{-i}}{2^n}$

Fractional Binary Numbers: Examples
- Value
  - $\frac{1}{2}$ and $\frac{3}{4}$
  - $\frac{1}{4}$ and $\frac{7}{8}$
  - $\frac{3}{8}$ and $\frac{5}{8}$

- Representation
  - $0.1011101_2$
  - $0.1111101_2$

- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.11111\ldots$ are just below 1.0
  - $1/2 = 1/4 + 1/8 + \ldots + 1/2^n + \ldots \rightarrow 1.0$
  - Use notation $1.0 - \epsilon$

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

Representable Numbers
- Limitation
  - Can only exactly represent numbers of the form $x/2^n$
  - Other rational numbers have repeating bit representations

- Value
  - $1/3$
  - $1/5$
  - $1/10$

- Representation
  - $0.010101010101011\ldots$
  - $0.00110011001100111\ldots$
  - $0.00011001100110011111\ldots$
Fixed Point Representation

- float $\rightarrow$ 32 bits; double $\rightarrow$ 64 bits
- We might try representing fractional binary numbers by picking a fixed place for an implied binary point
  
  - “fixed point binary numbers”
  
  - Let’s do that, using 8 bit floating point numbers as an example
  
  1. The binary point is between bits 2 and 3.
  
  \[ b_1, b_2, b_3, b_4 \] \[ \{ b_5, b_6, b_7 \} \]
  
  2. The binary point is between bits 4 and 5.
  
  \[ b_1, b_2, b_3, b_4 \] \[ \{ b_5, b_6, b_7, b_8 \} \]
  
  - The position of the binary point affects the range and precision:
    - Range: difference between the largest and smallest representable numbers.
    - Precision: smallest possible difference between any two numbers.

What else could we do?

IEEE Floating Point

- Fixing fixed point: analogous to scientific notation
  - $\times 10^n$:
    - $1.2 \times 10^7$; $0.0000032$; $1.2 \times 10^{-6}$

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
    - Supported by all major CPUs

- 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:
  
  \[ (-1)^s \times M \times 2^E \]
  
  - Sign bit $s$ determines whether number is negative or positive.
  - Significant (mantissa) $M$ normally a fractional value in range [1.0,2.0).
  - Exponent $E$ weights value by power of two

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

Precisions

- Single precision: 32 bits
  
  - $s$ exp frac
  
  \[ 1 \quad 8 \quad 24 \]

- Double precision: 64 bits
  
  - $s$ exp frac
  
  \[
  1 \quad 11 \quad 52
  \]

- Extended precision: 80 bits (Intel only)
  
  - $s$ exp frac
  
  \[
  1 \quad 15 \quad 63 \text{ or } 64
  \]
Normalization and Special Values

- "Normalized" means mantissa has form $1.xxx$.
  - $0.011 \times 2^5$ 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 $1$, don't bother to store it.

- How do we do $0/0$? How about $1/0$?

Special values:

- The float value $00...0$ represents zero.
- If the exp $= 11...1$ and the mantissa $= 00...0$, it represents $\pm \infty$.
  - E.g., $10.0/0.0 \rightarrow \infty$ if the exp $= 11...1$ and the mantissa $= 00...0$, it represents NaN.
  - "Not a Number".
  - Results from operations with undefined result.
    - E.g., $0/0$.

How do we do operations?

- Is representation exact?
- How are the operations carried out?

Floating Point Operations: Basic Idea

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

Basic idea:

- First, compute exact result.
- Make it fit into desired precision.
  - Possibly overflow if exponent too large
  - Possibly round to fit $\pm \text{Frac}$

Floating Point Multiplication

$\text{(1)}^2 \text{ M}_1 \text{ z}^2 \times \text{(1)}^2 \text{ M}_2 \text{ z}^2$

- Exact Result: $\text{(1)}^2 \text{ M}_1 \text{ z}^2$
  - Sign $s$:
  - Significant M: $\text{M}_1 \times \text{M}_2$.
  - Exponent $E$: $E_1 + E_2$

Fixing:

- If $M > 2$, shift M right, increment $E$.
- If $E$ out of range, overflow.
- Round $M$ to fit $\text{Frac}$ precision.

Implementation:

- What is hardest?

Floating Point Addition

$\text{(1)}^1 \text{ M}_1 \text{ z}^1 + \text{(1)}^2 \text{ M}_2 \text{ z}^2$

Assume $E_1 > E_2$

- Exact Result: $\text{(1)}^1 \text{ M}_1 \text{ z}^1$
  - Sign $s$, significant M:
  - Result of signed align & add $E$.
  - Exponent $E_1$.

Fixing:

- If $M = 2$, shift M right, decrement $E$.
- 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.
Hmm... if we round at every operation...

\[(a + b) + c\]

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 \rightarrow int
    - truncates fractional part
    - like rounding toward zero
    - not defined when out of range or NaN: generally sets to TMin
  - int \rightarrow float
    - exact conversion, why?
    - int \rightarrow float
      - will round according to rounding mode

Mathematical Properties of FP Operations

- Not really associative or distributive due to rounding
- Infinities and NaNs cause issues
- Overflow and infinity

Memory Referencing Bug (Revisited)

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 3.14;
    float f2 = 0.89;
    int i;
    f2 = 1.1/0.89;  //  
    printf("Value of f2 after division: \%f\n", f2);
    if (f2 == 1.23456789) {
        printf("Value of f2 matches expected value: true\n", f2);
    } else {
        printf("Value of f2 does not match expected value: false\n", f2);
    }
    return 0;
}
```

Explanation:

- Line 7: `f2 = 1.1/0.89;` 
- Location accessed by `main()`

Floating Point and the Programmer

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 3.14;
    float f2 = 0.89;
    int i;
    f2 = 1.1/0.89;  //  
    printf("Value of f2 after division: \%f\n", f2);
    if (f2 == 1.23456789) {
        printf("Value of f2 matches expected value: true\n", f2);
    } else {
        printf("Value of f2 does not match expected value: false\n", f2);
    }
    return 0;
}
```
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., 0.1
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”

- Mathematically equivalent ways of writing an expression may compute differing results

- NEVER test floating point values for equality!