Section 2: Integer & Floating Point Numbers

- Representation of integers: unsigned and signed
- Unsigned and signed integers in C
- Arithmetic and shifting
- Sign extension

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

Floating Point in C

- **C offers two levels of precision**

`float`	single precision (32-bit)
`double`	double precision (64-bit)

- **Default rounding mode is round-to-even**
- **`#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, as long as int has ≤ 53-bit word size
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - Not defined when out of range or NaN: generally sets to Tmin

Summary

- **Zero**

0	00000000	00000000000000000000000

- **Normalized values**

s	*1 to 2^k-2*	*significand = 1.M*

- **Infinity**

s	11111111	00000000000000000000000

- **NaN**

s	11111111	*non-zero*

- **Denormalized values**

s	00000000	*significand = 0.M*
s	exp	frac

Summary (cont'd)

- **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!**