Floating Point II

CSE 351 Summer 2019

Instructor:

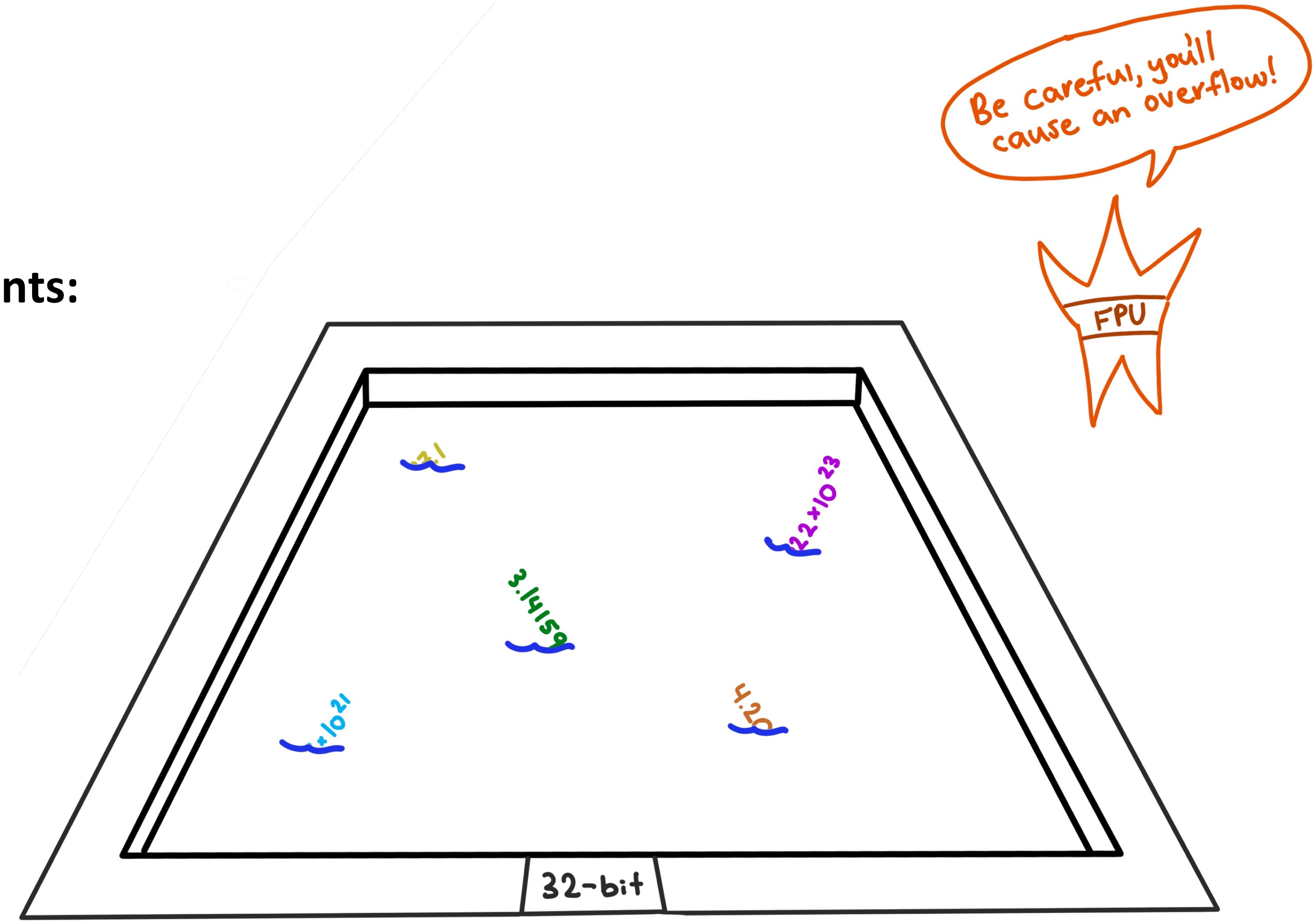
Sam Wolfson

Teaching Assistants:

Rehaan Bhimani

Corbin Modica

Daniel Hsu



FPU = floating point unit

Administrivia

- Lab 1b now due Friday (7/12)
  - Submit `bits.c` and `lab1Breflect.txt` on Gradescope
  - Extra credit must be submitted separately: also submit `bits.c` to "Lab 1b Extra Credit" assignment
- Homework 2 out now, due next Wednesday (7/17)
  - On Integers, Floating Point, and x86-64

Floating Point Topics

- **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…

Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like `int`s
  - "Gaps" produced in representable numbers means we can lose precision, unlike `int`s
    - Some "simple fractions" have no exact representation (*e.g.* 0.2)
    - "Every operation gets a slightly wrong result"
- Floating point arithmetic not associative or distributive
  - *Mathematically* equivalent ways of writing an expression may compute different results
- Never test floating point values for equality!
- Careful when converting between `int`s and `float`s!

Representation of Fractions

- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

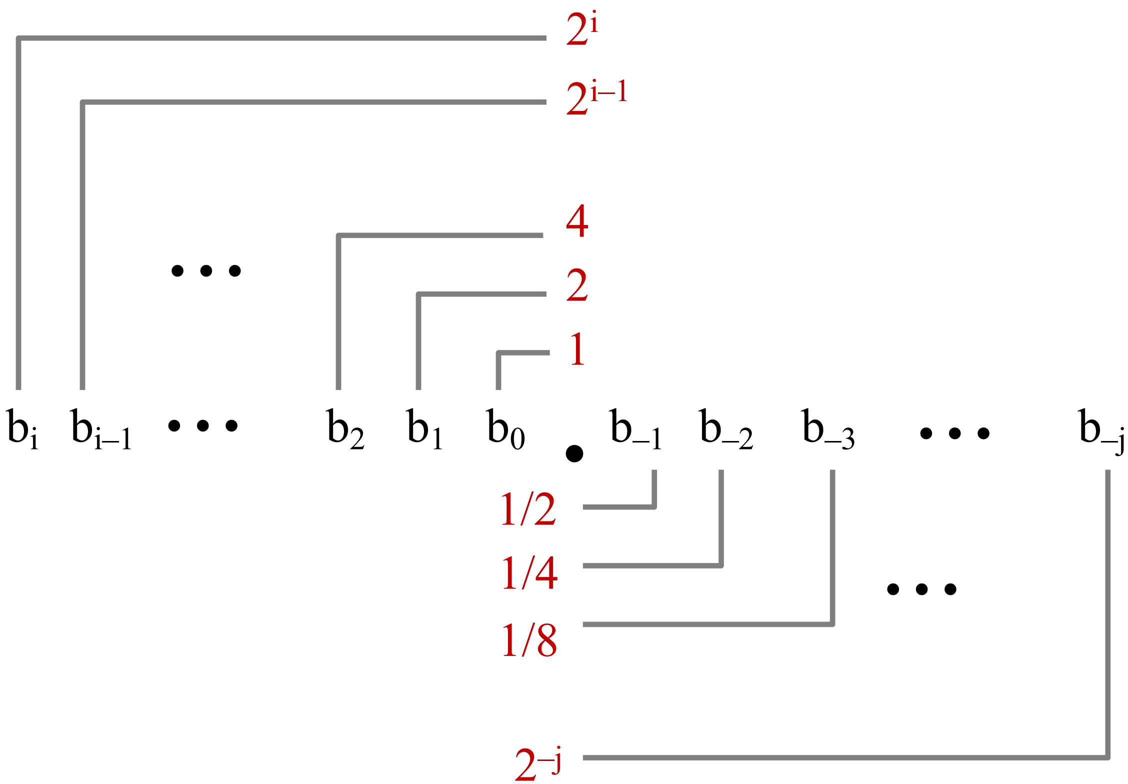
10.3

Example 6-bit representation: xx.yyyy

2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- <u>Example</u>: $10.1010_2 = 1{\times}2^1 + 1{\times}2^{-1} + 1{\times}2^{-3} = 2.625_{10}$

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 Representation
  - 5 and 3/4 $\texttt{101.11}_2$
  - 2 and 7/8 $\texttt{10.111}_2$
  - 47/64 $\texttt{0.101111}_2$
- Observations
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form $\mathtt{0.111111}\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Limits of Representation

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

Value:	Binary Representation:
• 1/3 = $0.333333\ldots_{10}$ =	$0.01010101[01]\ldots_2$
• 1/5 = 0.2	$0.001100110011[0011\ ]\ldots_2$
• 1/10 = 0.1	$0.0001100110011[0011\ ]\ldots_2$

Fixed Point Representation

- Implied binary point. Two example schemes:

  #1: the binary point is between bits 2 and 3

  $b_7\ b_6\ b_5\ b_4\ b_3$ [.] $b_2\ b_1\ b_0$

  #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 we have
- 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!

Floating Point Representation

- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but 1.2×10^7 In C: 1.2e7
    - Not 0.0000012, but 1.2×10^{-6} In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but 1.1×2^4
    - Not 0.000101, but 1.01×2^{-4}
- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the mantissa (significand)
  - the exponent

Scientific Notation (Decimal)

$$\overbrace{6.02}^{\text{mantissa}}{}_{10} \times 10^{23}$$

exponent

decimal point

radix (base)

- *Normalized form*: exactly one digit (non-zero) to left of decimal point
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (Binary)

exponent

$$\overbrace{1.01}^{\text{mantissa}}{}_2 \times 2^{-1}$$

binary point

radix (base)

- Computer arithmetic that supports this called floating point due to the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)

Scientific Notation Translation

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$
$2^{-4} = 0.0625$

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$
- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$
- **Practice:** Convert 11.375_{10} to binary scientific notation

$1011.011 \Rightarrow 1.011011 \times 2^3$

Floating Point Topics

- 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…

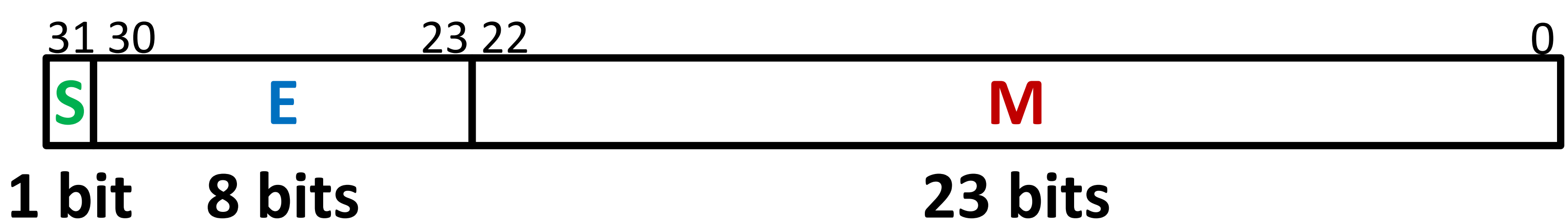
IEEE Floating Point

- 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 Encoding

- Use normalized, base 2 scientific notation:
  - Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
  - Bit Fields: $(-1)^{S} \times 1.M \times 2^{(E-\text{bias})}$
- Representation Scheme:
  - Sign bit (0 is positive, 1 is negative)
  - Mantissa (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector **M**
  - Exponent weights the value by a (possibly negative) power of 2 and encoded in the bit vector **E**



The Exponent Field

- Use biased notation
  - Read exponent as unsigned, but with *bias* of $2^{w-1}-1$ = 127
  - Representable exponents roughly ½ positive and ½ negative
  - Exponent 0 (Exp = 0) is represented as E = 0b 0111 1111
- Why biased?
  - Makes floating point arithmetic easier
  - Makes somewhat compatible with two's complement
- **Practice:** To encode in biased notation, add the bias then encode in unsigned:
  - Exp = 1 → 128 → E = 0b 1000 0000
  - Exp = 127 → 254 → E = 0b 1111 1110
  - Exp = -63 → 64 → E = 0b 0100 0000

The Mantissa (Fraction) Field

31	30 … 23	22 … 0
S	E	M
1 bit	8 bits	23 bits

$$(-1)^S \times (1 . M) \times 2^{(E-\text{bias})}$$

- Note the implicit 1 in front of the M bit vector
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$

    1.1×2^0

    $127-127$
  - Gives us an extra bit of *precision*
- Mantissa "limits"
  - Low values near M = 0b0…0 are close to 2^{Exp}
  - High values near M = 0b1…1 are close to 2^{Exp+1}

Peer Instruction Question

- What is the correct value encoded by the following floating point number?
  - 0b 0 $\underbrace{\overset{128}{10000000}}_{\text{exp}}$ $\underbrace{11000000000000000000000}_{\text{mantissa}}$
  - Vote at http://pollev.com/wolfson

exp: $128-\underbrace{127}_{\text{bias}}=1$

mant: 1.11×2^1

$= 11.1$

$= 2 + 1 + \frac{1}{2} = 3.5$

A. + 0.75

B. + 1.5

C. + 2.75

D. + 3.5

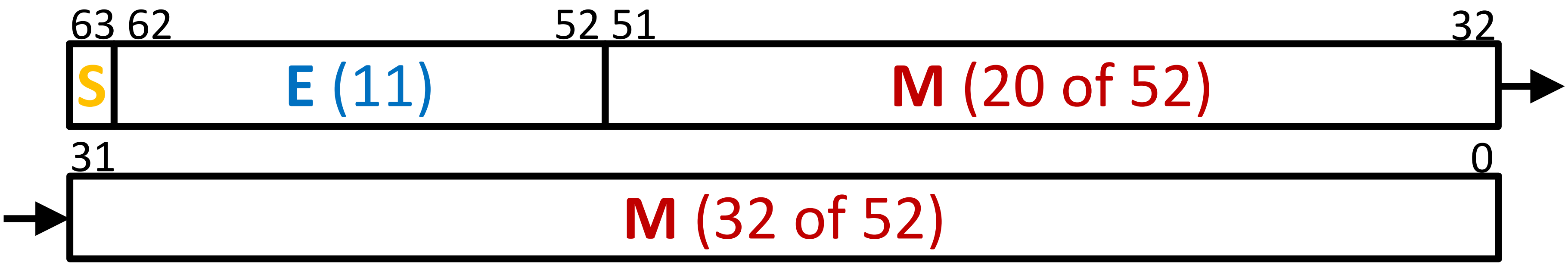
E. We're lost…

Precision and Accuracy

- Precision is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy
- Accuracy is a measure of the difference between the *actual value of a number* and its computer representation
  - *High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.*
  - **Example:** `float pi = 3.14;`
    - `pi` will be represented using all 24 bits of the mantissa (highly precise), but is only an approximation (not accurate)

Need Greater Precision?

- Double Precision (vs. Single Precision) in 64 bits

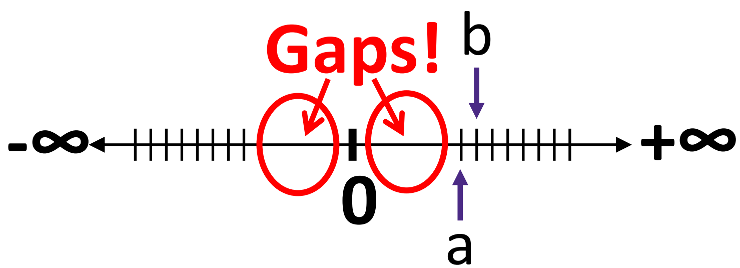


  - C variable declared as `double`
  - Exponent bias is now $2^{10}–1 = 1023$
  - **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
  - **Disadvantages:** more bits used, slower to manipulate

Representing Very Small Numbers

- But wait… what happened to zero?
  - Using standard encoding 0x00000000 = 1.0×2^{-127}
  - *Special case:* E and M all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers

- New numbers closest to 0:
  - $a = 1.0…0_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0…01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - *Special case:* E = 0, M ≠ 0 are denormalized numbers



Denorm Numbers

This is extra (non-testable) material

- Denormalized numbers
  - No leading 1
  - Uses implicit exponent of –126 even though E = 0x00

- Denormalized numbers close the gap between zero and the smallest normalized number
  - Smallest norm: $\pm\ 1.0\ldots0_{two}\times2^{-126} = \pm\ 2^{-126}$
  - Smallest denorm: $\pm\ 0.0\ldots01_{two}\times2^{-126} = \pm\ 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number

So much closer to 0

Other Special Cases

- E = 0xFF, M = 0: $\pm\infty$
  - *e.g.* division by 0
  - Still work in comparisons!
- E = 0xFF, M ≠ 0: Not a Number (NaN)
  - *e.g.* square root of negative number, 0/0, $\infty-\infty$
  - NaN propagates through computations
  - Value of M can be useful in debugging
- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: $1.1\ldots1_2\times2^{127} = 2^{128} - 2^{104}$

Floating Point Encoding Summary

	E	M	Meaning
smallest E (all zeroes)	0x00	0	± 0
	0x00	non-zero	± denorm num
everything else	0x01 – 0xFE	anything	± norm num
largest E (all ones)	0xFF	0	± ∞
	0xFF	non-zero	NaN

Summary

- Floating point approximates real numbers:

31	30 … 23	22 … 0
S	E (8)	M (23)

  - Handles large numbers, small numbers, special numbers
  - Exponent in biased notation (bias = 2^{w-1}–1) (if E=8, bias is 127)
    - Outside of representable exponents is *overflow* and *underflow*
  - Mantissa approximates fractional portion of binary point
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes *rounding*

E	M	Meaning
0x00	0	± 0
0x00	non-zero	± denorm num
0x01 – 0xFE	anything	± norm num
0xFF	0	± ∞
0xFF	non-zero	NaN

Floating point topics

- 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…

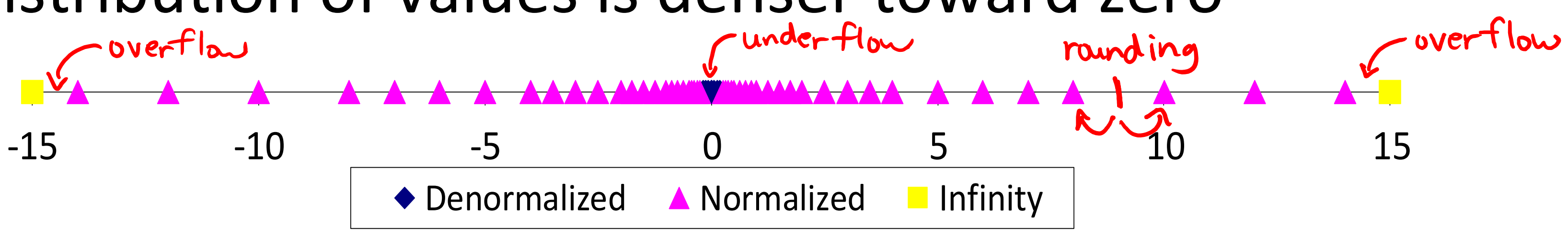
Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity Overflow (Exp too large)
  - Between zero and smallest denorm Underflow (Exp too small)
  - Between norm numbers? Rounding
- Given a FP number, what's the bit pattern of the next largest representable number?

  if M=0b0...00, then $2^{Exp} \times 1.0$

  if M=0b0...01, then $2^{Exp} \times (1+2^{-23})$

  diff = 2^{Exp-23}
  - What is this "step" when Exp = 0? 2^{-23}
  - What is this "step" when Exp = 100? 2^{77}
- Distribution of values is denser toward zero



Floating Point Rounding

This is extra (non-testable) material

- The IEEE 754 standard actually specifies different rounding modes:
  - Round to nearest, ties to nearest even digit
  - Round toward $+\infty$ (round up)
  - Round toward $-\infty$ (round down)
  - Round toward 0 (truncation)

- Tiny 8-bit example:
  - $< 1/2$
    Man = 1.001 01 rounded to M = 0b001
  - $> 1/2$
    Man = 1.001 11 rounded to M = 0b010
  - $== 1/2$, round to even
    Man = 1.001 10 rounded to M = 0b010

S	E	M
1	4	3

Floating Point Operations: Basic Idea

Value = $(-1)^{S} \times \text{Mantissa} \times 2^{\text{Exponent}}$

S	E	M

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

- Basic idea for floating point operations:
  - First, compute the exact result
  - Then *round* the result to make it fit into the specified precision (width of M)
    - Possibly over/underflow if exponent outside of range

Mathematical Properties of FP Operations

- Overflow yields ±∞ and underflow yields 0
- Floats with value ±∞ and NaN can be used in operations
  - Result usually still ±∞ or NaN, but not always intuitive
- Floating point operations do not work like real math, due to rounding
  - Not associative: `(3.14+1e100)-1e100 != 3.14+(1e100-1e100)`

    rounded off

    0 cannot be represented exactly 3.14
  - Not distributive: `100*(0.1+0.2) != 100*0.1+100*0.2`

    30.000000000000003553 30
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

Floating point topics

- 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…

Floating Point in C

!!!

- Two common levels of precision:

`float`	`1.0f`	single precision (32-bit)
`double`	`1.0`	double precision (64-bit)

- `#include <math.h>` to get `INFINITY` and `NAN` constants
  <float.h> for additional constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!
  - Instead, use: `abs(f1-f2)` $< 2^{-20}$

Some arbitrary threshold

Floating Point Conversions in C

!!!

- **Casting between `int`, `float`, and `double` changes the bit representation**
  - `int` → `float`
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - `int` or `float` → `double`
    - Exact conversion (all 32-bit `int`s representable)
  - `long` → `double`
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - "Not defined" when out of range or NaN: generally sets to `Tmin` (even if the value is a very big positive)

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 = %10.9f\n", f1);
  printf("f2 = %10.9f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - "Gaps" produced in representable numbers means we can lose precision, unlike `int`s
    - Some "simple fractions" have no exact representation (*e.g.* 0.2)
    - "Every operation gets a slightly wrong result"
- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results
- Never test floating point values for equality!
- Careful when converting between `int`s and `float`s!

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)