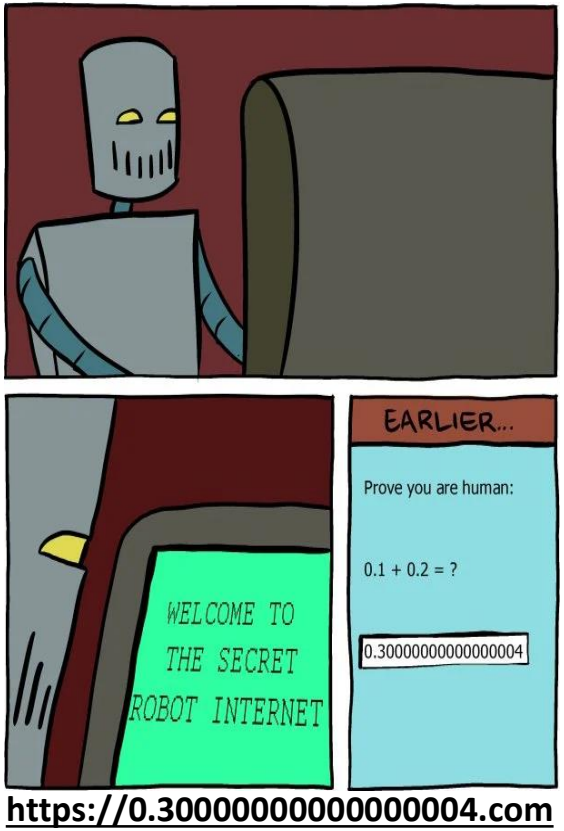
Floating Point

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

- HW4 due today (11:59pm)
- Due Wednesday, 7/3
  - RD7 (1pm)
  - HW5 (11:59pm)
  - **Lab1a (11:59pm)**
    - Late due date Friday, 7/5
- No class on Thursday
  - Video section on Panopto
- Quiz 1 released this Friday!

Lab1a Reminders

- Should compile without warnings, pass all tests *and* dlc.py
- Submit `pointer.c` and `lab1Asynthesis.txt` to Gradescope
- 1 submission per group
  - If you're submitting with a partner, please add them to your submission
- Please wait for the autograder to finish so you can confirm it worked!
  - Don't wait until the last minute to submit!
- Please remove debug print statements before submitting!
- Remember: 1-on-1 request form

Quiz 1

- Out on Friday, 7/5, due Friday, 7/12 on Gradescope
- Covers everything up through floating point
- Open-note, open-book
- See the Exams page on the course website for practice materials
- You can discuss with classmates, but all work must be your own!
  - “Abbot Elementary Rule”: talk with other students, then do something else for a while *before* writing your submission
  - i.e. brainstorming with others is fine, but you should **not** be writing your answers together
- Staff can answer clarifying questions, but not content questions
  - Please make all quiz-related Ed questions private!

Review Questions

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

1. Convert 11.375_{10} to normalized binary scientific notation

8	4	2	1	0.5	0.25	0.125
1	0	1	1 .	0	1	1

→ $1.011011_2 \cdot 2^3$

shift right by 3 to get 1.011011, then multiply by 2^3 to get back the original value

1. What is the value (in decimal) encoded by the following floating-point number?

bias $= 2^7 - 1 = 127$

implicit 1.—

S E M

0b 0 1000 0000 110 0000 0000 0000 0000 0000

positive

128

$+ 1.11_2 \cdot 2^{128-127} = 1.11_2 \cdot 2 = 11.1_2 = 3.5$

multiply by 2 = shift left by 1

Number Representation Revisited

- What can we represent in C so far?
  - Signed and unsigned integers
  - Characters
  - Addresses

- How do we encode the following?
  - Real numbers (ex: 3.14159)
  - Very large numbers (ex: $6.02*10^{23}$)
  - Very small numbers (ex: $6.26*10^{-34}$)
  - Special cases (ex: ∞, NaN)

Floating Point Topics

- **Fractional binary numbers (fixed point)**
- **Floating point**
  - **IEEE standard**
- Floating point operations and rounding
- Floating point in C

There are many more details we won't cover (it's a 58-page standard…)

  - Bonus slides at the end :)

Binary Representation of Fractions

- Let's start by looking at base 10:
  - Each place represents a power of 10. Power decreases as you read left->right
  - Decimal point marks when the negative powers start

Ex:

10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
1	2	3	.	4	5	6

- Base 2 is similar:
  - Every place to the right of the **binary point** represents a negative power of 2

Ex:

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

$= 1*2^2 + 0*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3} = 5.375_{10}$

Limits of Representation

- Even with an arbitrary number of bits, can only represent numbers of the form $x*2^y$
- Other rational numbers have infinite bit representations

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

Fixed Point Encoding

- Implied binary point between two bits in a number
- Two examples
  1. 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. Binary point is between bits 4 and 5
     $b_7 b_6 b_5$ (.) $b_4 b_3 b_2 b_1 b_0$
- Which scheme is better?

Real-world example: money!

$10.95

only count 2 places after the decimal point

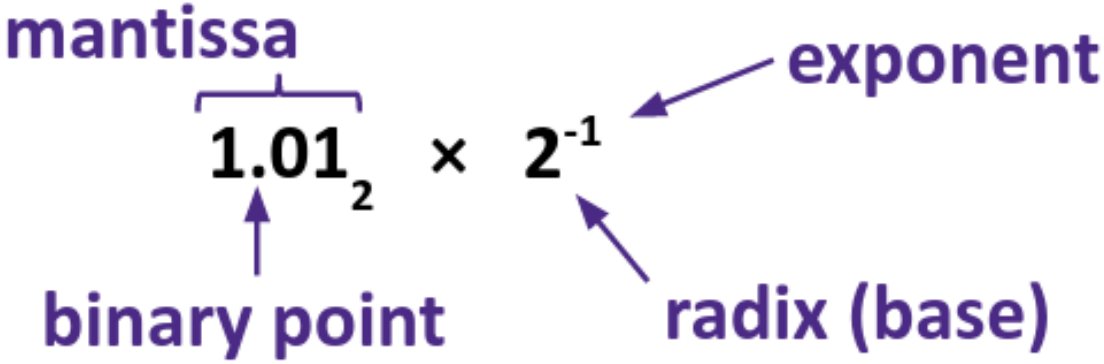
Depends! 1 can encode higher values, but w/ less precision.
2 has more precision, but can only encode whole #s up to 7.
No good solution for all cases, so we don't use it.

Floating Point Representation (Review)

Reminder: shift right by n places in base b = divide by b^n

- Based on **scientific notation**
  - In decimal:
    - 12000000 -> 1.2×10^7 → shift right by 7, so multiply by 10^7 to get back to the same value
    - 0.0000012 -> 1.2×10^{-6}
  - In binary:
    - 11000.000 -> 1.1×2^4 → shift right by 4, so multiply by 2^4
    - 0.00011 -> 1.1×2^{-4} → shift left by 4, so multiply by 2^{-4}
- Divvy up the bits in our encoding
  - Sign (+/-)
  - Exponent
  - Mantissa (everything after the binary point)

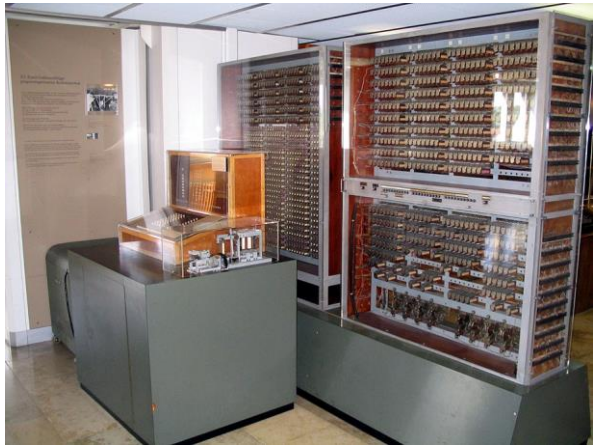
Binary Scientific Notation (Review)



- **Normalized form**: exactly one (non-zero) bit to the left of the binary point
- Called “floating point” because the binary point “floats” to different parts of the number (as opposed to fixed)

Floating Point History

- 1914: first design by Leonardo Torres y Quevedo
- 1940: implementations by Konrad Zuse, but not exactly the same as the modern standard
- 1985: IEEE 754 standard
  - Primary architect was William Kahan, who won a Turing Award for this work
  - Standardized bit encoding, well-defined behavior for *all* operations
  - Still what we use today!

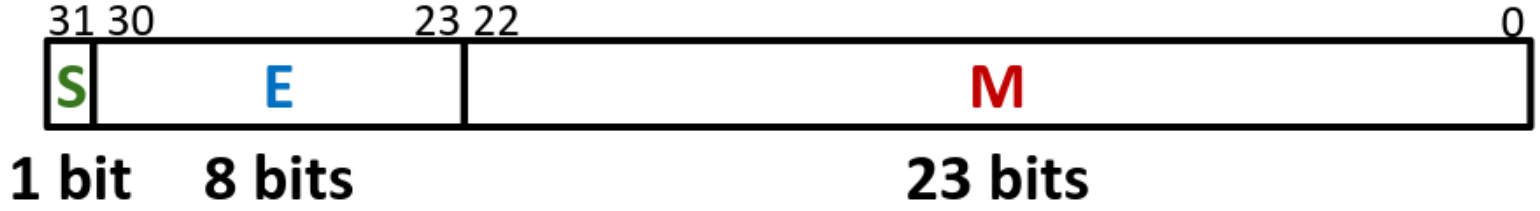


IEEE Floating Point

- IEEE 754 (established 1985)
  - Developed to make numerically-sensitive programs portable
  - Specifies two things: a *representation scheme* and the *result of operations*
  - Supported by all major CPUs
- Two opposing concerns:
  - **Scientists** numerical analysts want them to be as *real* as possible
  - **Engineers** want them to be *easy to implement* and *fast*
  - Who won? Mostly scientists
    - Nice standards for rounding, overflow, underflow, but complex for hardware
    - **Float operations can be an order of magnitude slower than integer ops!**
      - CPU speed commonly measured in **FLOPS** (float ops per second)

Floating Point Encoding (Review)

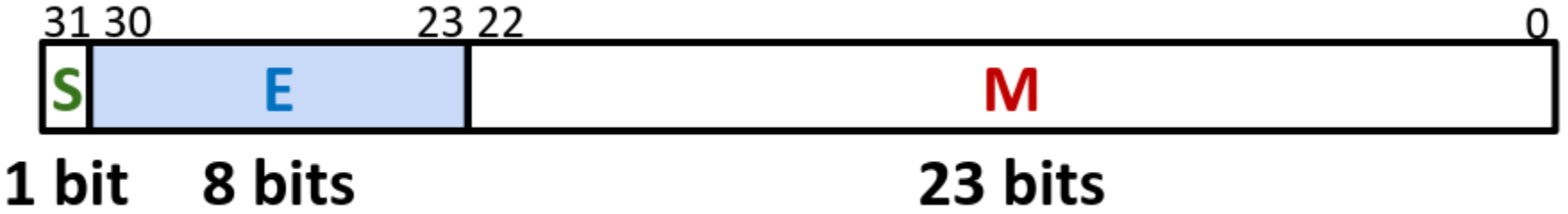
- Value = ±1.Mantissa * $2^{Exponent}$
- Bit fields: $(-1)^{S}$ * 1.**M** * 2^{E}
- Representation scheme:
  - Sign bit (**S**): 0 is positive, 1 is negative
  - Mantissa (a.k.a. significand): the fractional part of the number in normalized form, encoded in the bit vector **M**
  - Exponent: weighs the number by a (possible negative) power of 2, encoded in the bit vector **E**



The Exponent Field (Review)

- Use **biased notation**
  - Read as unsigned, but with a **bias** of $2^{w-1}-1$ (127, for an 8-bit **E** field)
  - Exponent = **E** - bias ↔ **E** = Exponent + bias
- Why?
  - Makes floating point arithmetic easier
  - Somewhat compatible with two's complement hardware

why not use 2's comp?
makes float comparison easier



The Mantissa Field (Review)

- **Implicit leading 1** before the binary point
  - There's always a 1 there in normalized form, so we don't need to encode it!
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000
    - Read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$
- Mantissa "limits"
  - Low values (near **M** = 0b00…00) are close to 2^{Exp}
  - High values (near **M** = 0b11…11) are close to 2^{Exp+1}

Normalized Floating Point Conversions

FP -> Decimal

1. Append bits of **M** to leading 1
2. Multiply by $2^{E\text{-bias}}$
3. "Multiply out" exponent by shifting
   a. If exp < 0, shift *right* (logical) by -exp
   a. If exp > 0, shift *left* by exp
4. Multiply by sign (-1^{S})
5. Convert from binary to decimal

Decimal -> FP

1. Convert from decimal to binary
2. Convert to normalized form
   a. Shift left or right (logical) until there's a single 1 before the binary point
   b. Multiply by 2^{exp}, where exp = number of places shifted (negative for left shift, positive for right)
3. **S** = 0 if positive, 1 if negative
4. **E** = exp + bias
5. **M** = bits after the binary point

Practice Question

bias = $2^7 - 1 = 127$

Convert the decimal number -7.375 into floating point representation.

4	2	1	0.5	0.25	0.125
1	1	1.	0	1	1

$\rightarrow -1.11011_2 \cdot 2^2$

S = 1 (negative)

E = 2 + 127 = 129 = 0b10000001

M = 11011000...00 (23b total)

0b 1 10000001 11011000...00

Challenge Question:

Find the value of the following sum in normalized binary scientific notation:

$1.01_2 * 2^0 + 1.11_2 * 2^2$

$= 1.01_2$ $= 111_2$

$1.01_2 + 111 = 1000.01_2 = 1.00001_2 \cdot 2^3$

Special Cases

- But wait, what happened to zero?
  - **Special case**: **E** and **M** are all 0s
    - Two zeroes… but at least 0x00..00 is 0, like integers
- Other special cases: **E** = 0xFF
  - **M** = 0: **+/- ∞**
    - Ex: division by 0
    - Still works for comparisons!
  - **M** ≠ 0: **NaN**
    - Ex: 0/0, ∞-∞, imaginary numbers
    - Value of **M** tells you what the error was
    - Propagates through computations

Floating Point Topics

- Fractional binary numbers (fixed point)
- Floating point
  - IEEE standard
- **Floating point operations and rounding**
- Floating point in C

Precision and Accuracy

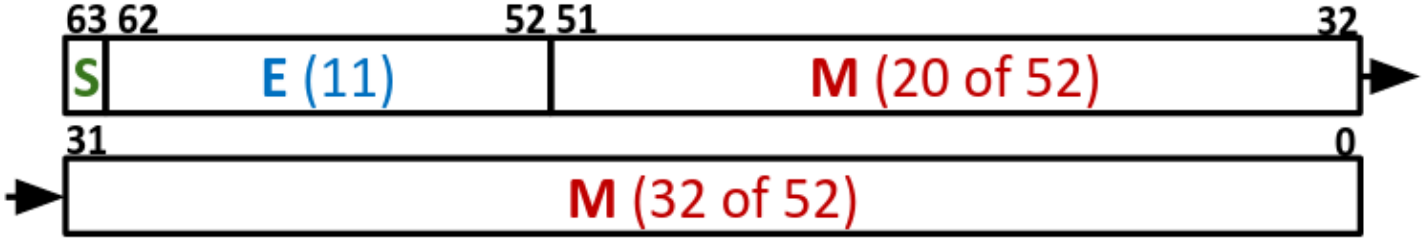
- **Accuracy** is a measure of the difference between the actual value of a number and its computer representation
- **Precision** is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy
- **High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.**

Example: `float pi = 3.14;`

  - `pi` will be represented with all 24 bits of mantissa (highly precise), but still an approximation

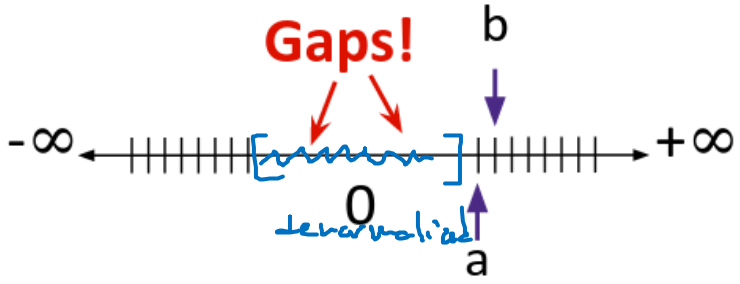
Need Greater Precision?

- 64 bits = **double precision**
- Exponent bias is now $2^{10}–1 = 1023$
- Advantages
  - Greater precision (larger mantissa), greater range (larger exponent field)
- Disadvantages
  - More space used, slower to manipulate



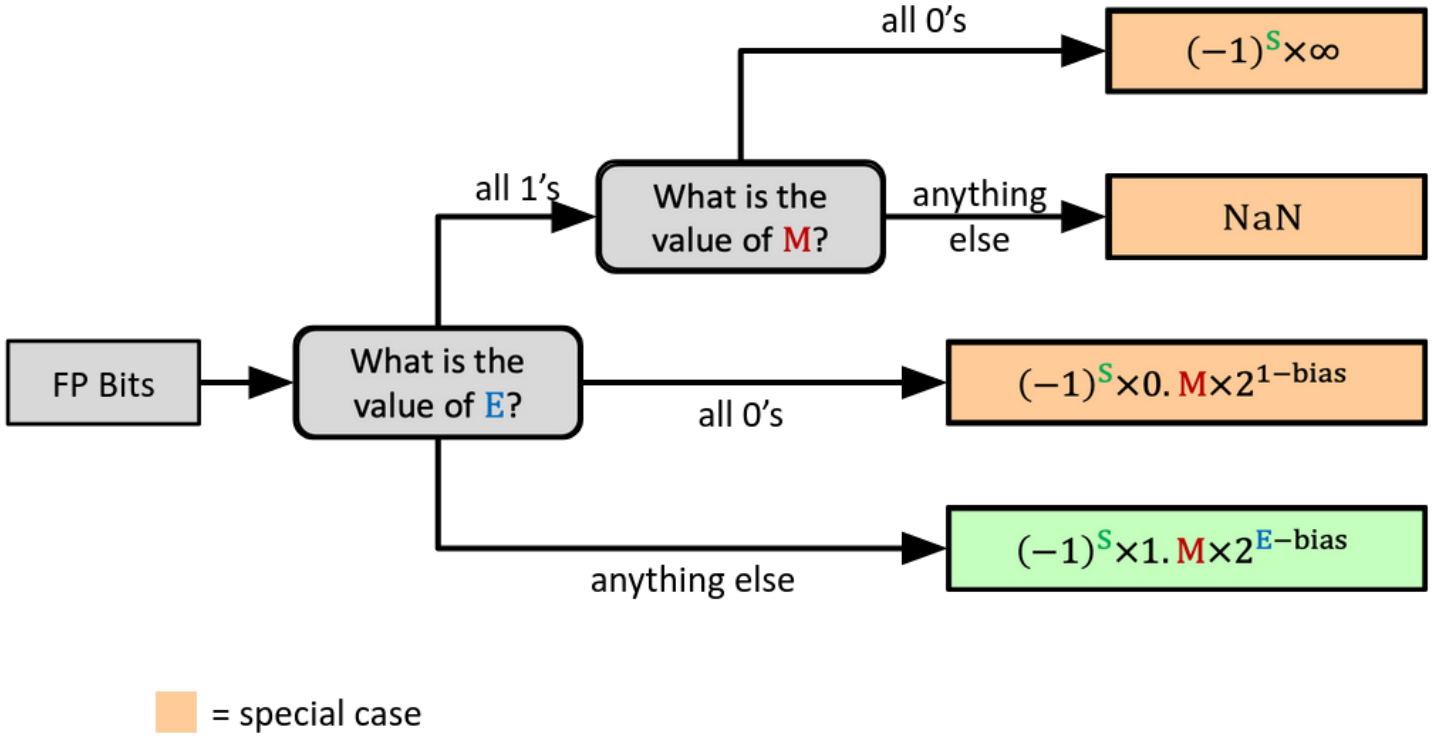
Representation Limits

- Largest value (besides ∞)?
  - **E** = 0xFF has been taken!
  - **E** = 0xFE, **M** = 0b11…11 -> $1.1\ldots1_2 * 2^{127} = 2^{128} - 2^{104}$
    - Much bigger than the max int!
- Smallest (non-zero) value?
  - **E** = 0x00 has been taken!
  - **E** = 0x01, **M** = 0b00…00 -> 2^{-126}
    - Normalization and implicit 1 are to blame
  - *Another special case*: **denormalized numbers**
    - **E** = 0, **M** ≠ 0 are
    - Replace implicit leading 1 with a 0



E still treated as 0, so value is $\pm 0.M \cdot 2^{-127}$

Floating Point Decoding Flow Chart

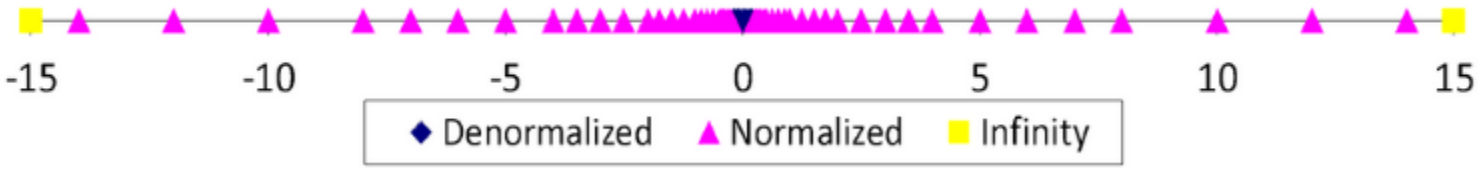


Distribution of Values (Review)

- Like integers, floats cannot represent every possible value
  - **Overflow**: the number we want to store is too large to be represented

- Floats have other problems too, due to finite *precision*:
  - **Underflow**: value is too small to represent
    - Between 0 and the smallest denormalized number
  - **Rounding**: we don't have enough bits
    - Between two representable values

Floats can represent very large *or* very precise numbers, but not both!



Representational Errors

- **Overflow** yields ±∞, **underflow** yields 0
- ±∞ and NaN can be used in operations
  - Result is usually still the same, but not always intuitive
- **Floating point operations do not work like real math, due to rounding**
  - Not associative
    - Ex: $(3.14 + 10^{100}) - 10^{100} != 3.14 + (10^{100} - 10^{100})$
  - Not distributive
    - Ex: $100*(0.1 + 0.2) != 100*0.1 + 100*0.2$
  - Not cumulative
    - Repeatedly adding a very small number to a very large one may do nothing

Why does this matter?

- **1982:** Vancouver Stock Exchange 10% error in less than 2 years
- **1991:** Patriot missile targeting error
  - Clock skew due to conversion from int to float
- **1994:** Intel Pentium FDIV (float division) hardware bug ($475 million)
- **1996:** Ariane 5 rocket exploded ($1 billion)
  - Overflow converting 64-bit float to 16-bit int
- **1997:** USS Yorktown “smart” warship stranded
  - Divide by zero

Floating Point Topics

- Fractional binary numbers (fixed point)
- Floating point
  - IEEE standard
- Floating point operations and rounding
- **Floating point in C**

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
- `#include <float.h>` for additional constants
- **Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!**

Floating Point Conversions in C

- **Casting between int, float, and double changes the bit representation!**
- `int -> float`
  - May be rounded (not enough mantissa bits)
  - Overflow impossible
- `int or float -> double`
  - No rounding or overflow possible
- `long -> double`
  - Depends on word size (32-bit is exact, 64-bit may cause rounding)
- `double or float -> int`
  - Truncates fractional part (rounded towards zero)
  - **Undefined** when out of range or NaN, typically sets to TMin

Summary

- Floating point approximates real numbers using **binary scientific notation**
  - Exponent in **biased notation**
- Standard encoding is **IEEE 754**
  - Defines standard bit width for fields, behavior in operations, and special cases
- Floats also suffer from having a fixed bit width
  - Can get **overflow**, but also **underflow** and **rounding**
- **Floating point arithmetic can have unexpected results!**
  - ***Never*** test floats for equality
- Conversion between float and other data types can cause errors
  - Be especially careful when converting between `int` and `float`!

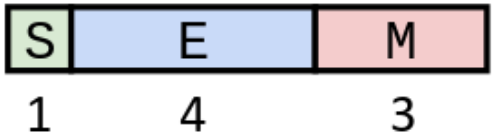
BONUS SLIDES

Some additional information about floating point numbers. We won't test you on this, but you may find it interesting :)

Floating Point Rounding

This is extra (non-testable) material

- The IEEE 754 standard actually specifies different rounding modes:
  1. **Round to nearest, ties to nearest even number** (standard)
  2. Round toward +∞ (round up)
  3. Round toward -∞ (round down)
  4. Round toward 0 (truncate)
- In our tiny example (**E** = 4 bits, **M** = 3 bits) with standard rounding
  - Mantissa = 1.001 01 rounds to M = 0b001
  - Mantissa = 1.001 11 rounds to M = 0b010
  - Mantissa = 1.001 10 rounds to M = 0b010
  - Mantissa = 1.000 10 rounds to M = 0b000



Limits of Interest

This is extra (non-testable) material

- The following thresholds can help you get a sense of when certain outcomes come into play, but don't worry about the specifics
  - **FOver** = 2^{bias+1}
    - Largest representable normalized number
  - **FDenorm** = 21^{1-bias}
    - Smallest representable normalized number
  - **FUnder** = $2^{1-bias-m}$
    - *m* is the width of the mantissa field
    - Smallest representable denormalized number

Denormalized Numbers

This is extra (non-testable) material

- 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: $\pm\ 1.0\ldots0_2 \times 2^{-126} = \pm\ 2^{-126}$
  - Smallest denorm: $\pm\ 0.0\ldots01_2 \times 2^{-126} = \pm\ 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number

Much closer to zero!

Floating Point in the “Wild”

This is extra (non-testable) material

- 3 formats from IEEE 754 standard widely used in computer hardware and languages
  - In C, called `float`, `double`, `long double`
- Common applications:
  - 3D graphics: textures, rendering, rotation, translation
  - “Big Data”: scientific computing at scale, machine learning
- Non-standard formats in domain-specific areas:
  - **Bfloat16:** training ML models; range more valuable than precision
  - **TensorFloat-32:** Nvidia-specific hardware for Tensor Core GPUs

Type	S bits	E bits	M bits	Total bits
Half-precision	1	5	10	16
Bfloat16	1	8	7	16
TensorFloat-32	1	8	10	19
Single-precision	1	8	23	32