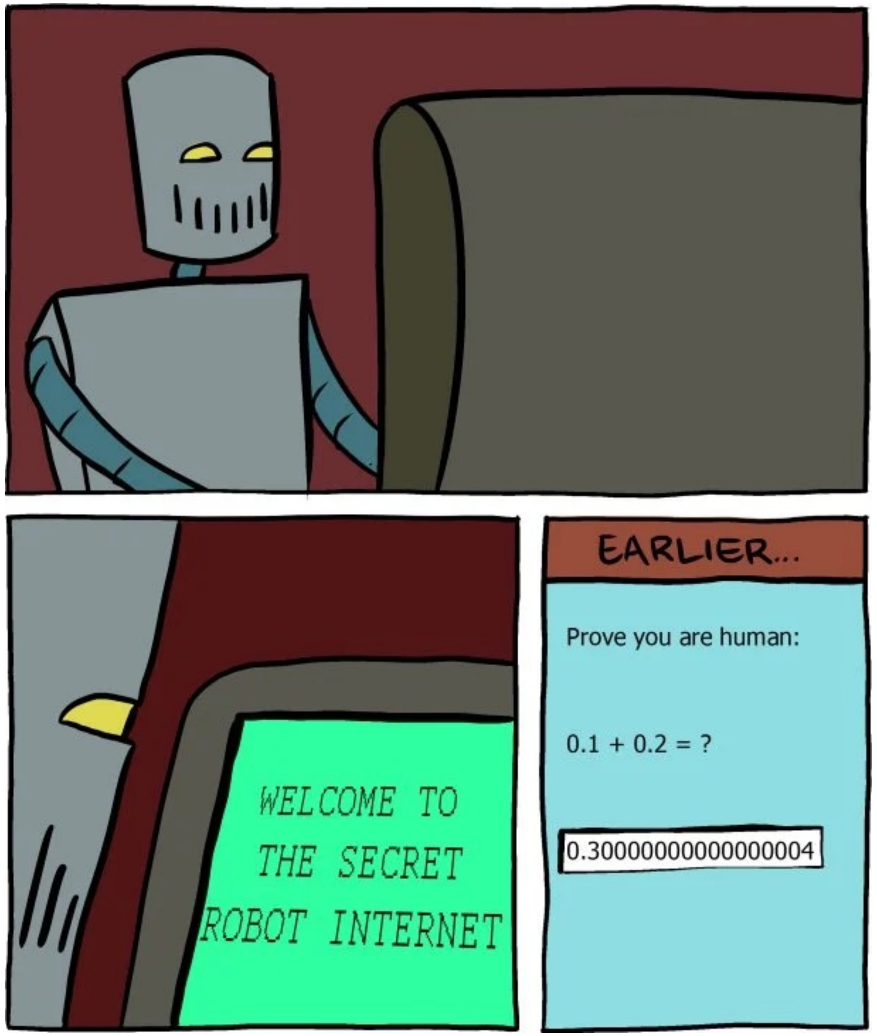
# Floating Point
## CSE 351 Spring 2024

**Instructor:**
Elba Garza

**Teaching Assistants:**

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson




https://0.30000000000000004.com

# Announcements, Reminders

- HW4 due tonight, HW5 due Monday (8 Apr)
  - Getting ahead a bit: no RD due for Friday due to combined RD9/10!
- Lab 1a due Monday (8 Apr)
  - Submit `pointer.c` and `lab1Asynthesis.txt` on Gradescope by deadline!
    - Make sure there are no lingering `printf` statements in your code!
  - Can use (up to two) late day tokens to submit up until Wednesday 10 Apr at 11:59 PM
  - If you are submitting with a partner, ensure that <u>you add them to the submission</u>
- Lab 1b due Monday (15 Apr)
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`

# Exams

- Midterm and final exams will be taken on Gradescope
- Open for 48 hours and 72 hours respectively, no time limit
  - Designed to take 1-3 hours
  - Midterm open May 6$^{th}$ at 00:00, due May 7$^{th}$ at 23:59
  - Final open June 3$^{rd}$ at 00:00, due June 5$^{th}$ at 23:59
- Open book, open notes, open (.*)
  - But <u>not</u> group work—taken individually
  - High-level discussion with classmates OK, but you must write answers on your own (like labs, but without a partner)
- Mixture of problem-solving, design, and personal reflection questions (short answer & open ended)

# Lab 1b Aside: C Macros

from LC4

- C macros basics: e.g. #define SUIT_MASK 0x30
  - Basic syntax is of the form: `#define NAME expression`
  - Allows you to use `NAME` instead of `expression` in code
    - Does naïve copy and replace before compilation – everywhere the characters `NAME` appear in the code, the characters `expression` will now appear instead
    - **Not** the same as a Java constant, but used in a similar way
  - Useful to help with readability/factoring in code

- You'll use C macros in Lab 1b for defining bit masks
  - See Lab 1b starter code and LC4 slides (card operations) for examples

# Reading Review

- Terminology:
  - normalized scientific binary notation
  - trailing zeros
  - sign, mantissa, exponent ↔ bit fields S, M, and E
  - `float`, `double`
  - biased notation (exponent), implicit leading one (mantissa)
  - rounding errors

# Review Questions

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

$\overbrace{11}^{8+2+1}.\overbrace{375}^{0.25+0.125}$

- Convert $11.375_{10}$ to normalized binary scientific notation

$1\ 0\ 1\ 1\ .\ 0\ 1\ 1 \xrightarrow{\text{normalize}} 1.011011 \times 2^3$

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

S E M

`0b 0 | 1000 0000 | 110 0000 0000 0000 0000 0000`

Sign = +   exponent = 128 − 127 = 1   Mantissa = 1.11 (implicit 1)

$\Longrightarrow +1.11 \times 2^1$

exponent (E)   Sign (S)   mantissa (M)

- exponent = E – bias, where bias = $2^{w-1}$-1 $= 2^{8-1} - 1 = 127$
- mantissa = 1.M

not normalized $= 11.1 = 3.5_{10}$

# Number Representation Revisited

- What can we represent in one word?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses
- How do we encode the following:
  - Real numbers (e.g., 3.14159)
  - Very large numbers (e.g., $6.02 \times 10^{23}$)
  - Very small numbers (e.g., $6.626 \times 10^{-34}$)
  - Special numbers (e.g., ∞, NaN)

**Floating Point**

**Cram it all into one encoding?!**

# 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…

# Representation of Fractions

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

Example 6-bit representation:

$$\underset{2^1\ 2^0}{xx}.\underset{2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}}{yyyy}$$

What if this binary point wasn't fixed?

We could represent a larger range of values! Let's make it float!

- Example:

$$10.1010_2 = 1\times 2^1 + 1\times 2^{-1} + 1\times 2^{-3} = 2.625_{10}$$

# Binary Scientific Notation (Review)

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

exponent

binary point

radix (base)

- **Normalized form**: exactly one digit (non-zero) to left of binary point
- 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`)

# IEEE Floating Point

- IEEE 754 (established in 1985)
  - Standard to make numerically-sensitive programs portable
  - Specifies two things: *representation scheme* and result of *floating point operations*
  - 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**.
  - Who won?

    Scientists mostly won out:
    - Nice standards for rounding, overflow, underflow, but... complex for hardware
    - **Float operations can be an order of magnitude slower than integer ops ➔ so slow, it's used as a performance gauge! (e.g. FLOPS/s)**

# Floating Point Encoding (Review)

- 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**

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

# The Exponent Field (Review)

- Use biased notation
  - Read exponent as unsigned, but with a **bias** of $2^{w-1}-1$ (bias = 127, for E of 8 bits)
  - Representable exponents roughly ½ positive and ½ negative
  - Exp = E – bias ↔ E = Exp + bias
    - Exponent value of 0 (Exp = 0) is thus represented as E = 0b 0111 1111

- Why biased?
  - Makes floating point arithmetic easier—somewhat compatible with two's complement hardware. Very hand wave-y, I know.
  - Now it's a sign-and-magnitude representation!

# The Mantissa (Fraction) Field (Review)

| 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 leading 1** in front of M bit vector (Normalized form)
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000
    Read as $1.1_2 = 1.5_{10}$, not $0.1_2 = 0.5_{10}$, because of implicit leading 1
  - A "free" extra bit of precision!
- Mantissa "limits"
  - Low values near M = 0b000…000 are close to $2^{\text{Exp}}$
  - High values near M = 0b111…111 are close to $2^{\text{Exp}+1}$

$1.0000\ldots \times 2^{Exp}$

$1.11111\ldots \times 2^{Exp}$

Difference b/w $2^{Exp+1}$ & $1.1111\ldots \times 2^{Exp}$ is small:

$2^{Exp+1} - \underbrace{2^{Exp-23}}_{\text{Small but finite!}}$

# Normalized Floating Point Conversions

- FP → Decimal
  1. Append the bits of $\mathrm{M}$ to implicit leading 1 to form the mantissa.
  2. Multiply the mantissa by $2^{\mathrm{E} - \text{bias}}$.
  3. Multiply the sign $(-1)^{S}$.
  4. Multiply out the exponent by shifting the binary point.
  5. Convert from binary to decimal.

- Decimal → FP
  1. Convert decimal to binary.
  2. Convert binary to normalized scientific notation.
  3. Encode sign as $S$ (0/1).
  4. Add the bias to exponent and encode $\mathrm{E}$ as unsigned.
  5. The first bits after the leading 1 that fit are encoded into $\mathrm{M}$.

# Example & Practice Question

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

- Convert the decimal number **-11.375** into floating point representation

$S = 1$ (Since -11.375 is negative)

$-1011.011 \xrightarrow{\text{normalize}} -1.\underbrace{011011}_{\text{Mantissa}} \times 2^{3}$ ← Exponent

$E = 3_{10} + 127_{10} = 130_{10} \rightarrow$ 0b10000010

confused? See Slide 13!

$M = 011011$

∴ Floating Point ⟹ [0b 1100 | 0001 | 0011 | 0110 | 0000 | 0... ]
(C 1 3 6)

In hex: 0xC136

Exponent = E – bias ↔ E = Exponent + bias

Mantissa = 1.M

# Precision and Accuracy

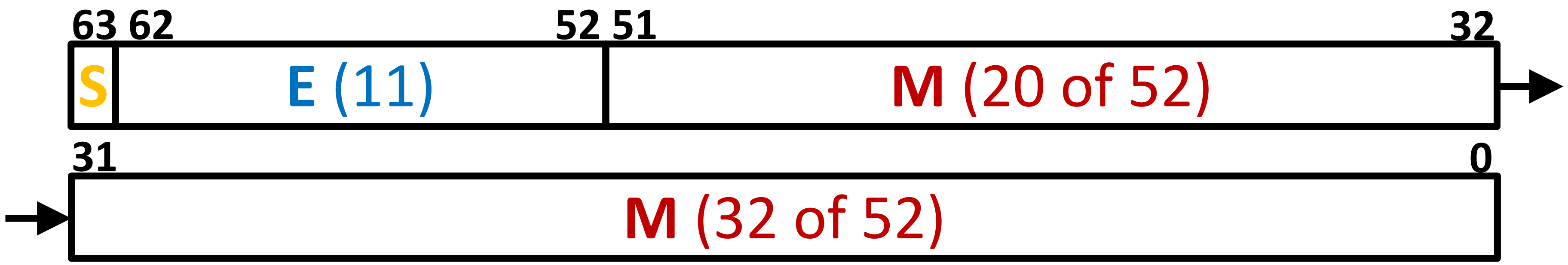
- **Precision** is a count of the number of bits in a computer word used to represent a value, i.e. **capacity** for accuracy
- **Accuracy** is a measure of the difference between the **actual value of a number** and its computer representation

**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

# 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…

# Special Cases & Encodings

But wait, how to represent zero & other fun stuff...?

- **Case 1:** E and M all zeros → **0**
  - Wait, what about the S bit? Two zeros!

    But at least 0x00000000 = 0 like integers

- **Case 2:** E = 0xFF, M = 0 → **± ∞**
  - *e.g.*, division by 0
  - Still work in comparisons!

- **Case 3:** E = 0xFF, M ≠ 0 → Not a Number (**NaN**)
  - *e.g.*, square root of negative number, 0/0, ∞–∞
  - NaN propagates through computations
  - Value of M can be useful in debugging

We need a specific encoding for 0 because otherwise:

$1.0 \times 2^0 = \boxed{1_{10}}$

Mantissa of 0...

exponent of 0...

Implicit 1 ruins it!

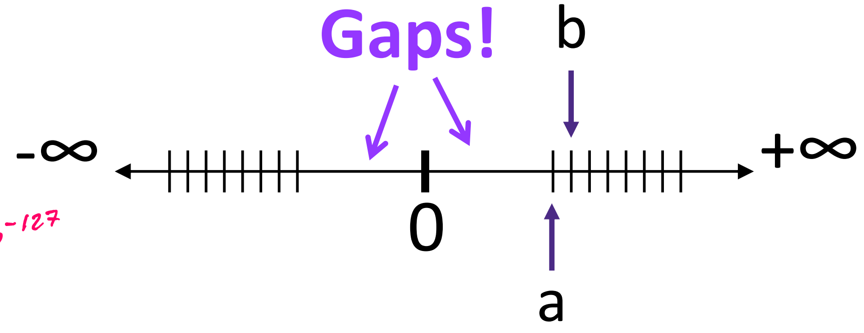
# New Representation Limits due to Special Cases

- What's now the largest value (besides ∞)?
  - E = 0xFF has now been taken by **Case 2** & **Case 3**!
  - E = 0xFE is now largest: $\boxed{1.1\ldots1_2 \times 2^{127}} = \boxed{2^{128} - 2^{104}}$

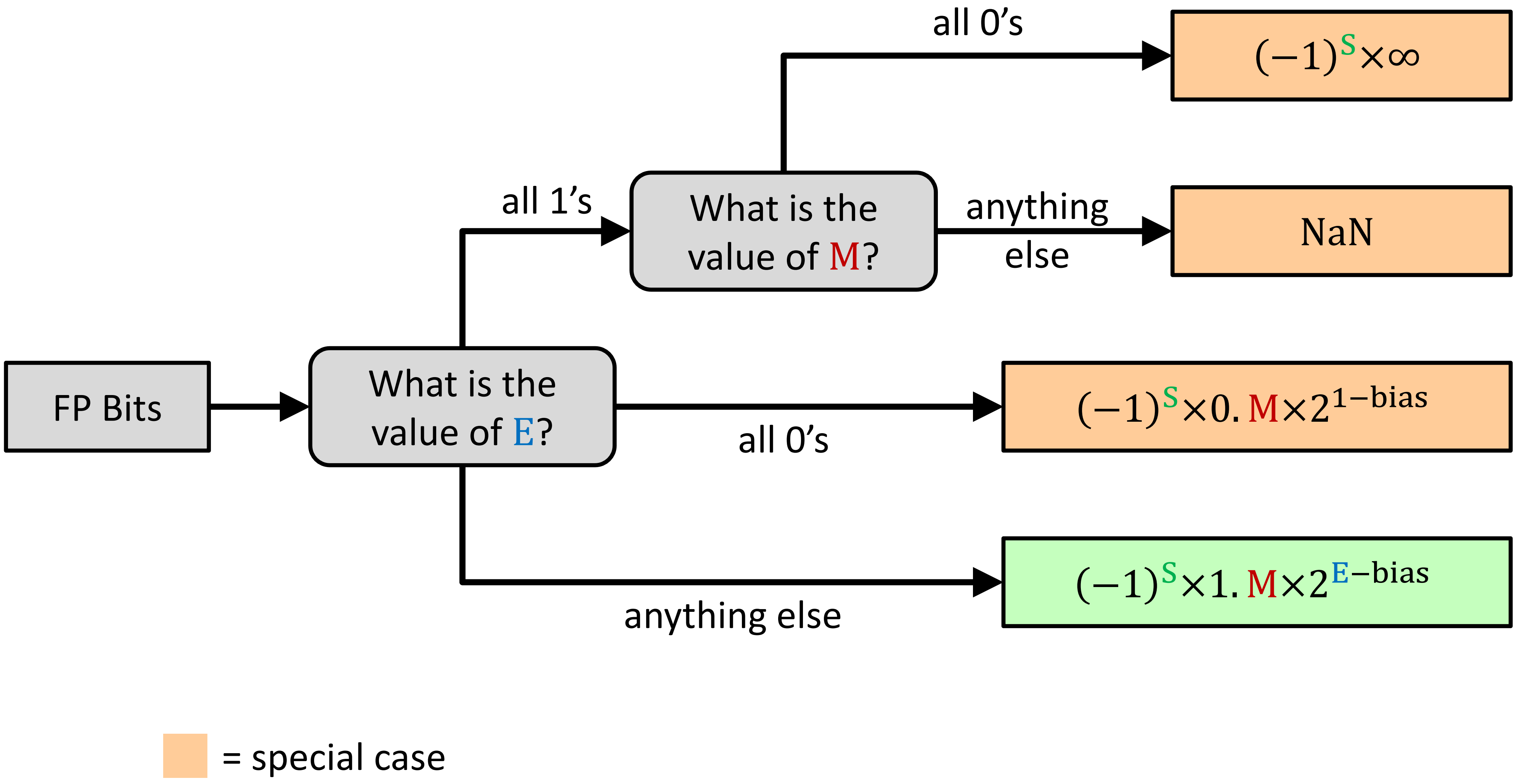
    Used to be able to be $2^{128}$

- What are now the numbers closest to 0? (i.e. M = 0)
  - E = 0x00 taken by **Case 1**; so next smallest is E = 0x01

    $2^{-126}$, used to be able to be $2^{-127}$
  - $a = 1.0\ldots00_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0\ldots01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit leading 1 are to blame
  - Leads to another Special case: E = 0, M ≠ 0 are **denormalized numbers**
    - Mantissa has implicit leading 0 instead of implicit leading 1
    - Store much smaller numbers



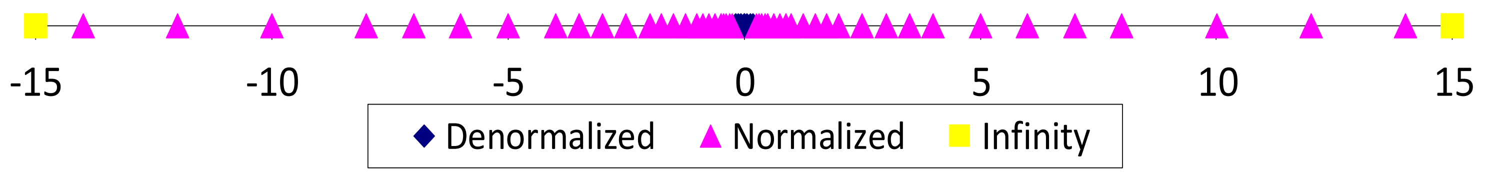
# Floating Point Decoding Flow Chart



# Distribution of Values (Review)

- 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 next largest representable number?
  - What is this "step" when Exp = 0? $\mathbf{2^{-23}}$
  - What is this "step" when Exp = 100? $\mathbf{2^{77}}$
- Distribution of values is denser toward zero:

**You can represent really large numbers, or really precise numbers, but not both!**



# Floating Point Operations: Basic Idea

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

| S | E | M |
|---|---|---|

- `x +f y = Round(x + y)`
- `x *f y = 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 $\pm\infty$ and underflow yields 0
- Floats with value $\pm\infty$ and NaN can be used in operations
  - Result usually still $\pm\infty$ 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)`
    - `0` vs. `3.14`
  - Not distributive: `100*(0.1+0.2) != 100*0.1+100*0.2`
    - `30.000000000000003553` vs. `30`
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

# 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!**

Instead do something like

$if((x-y) < \varepsilon)\ldots$

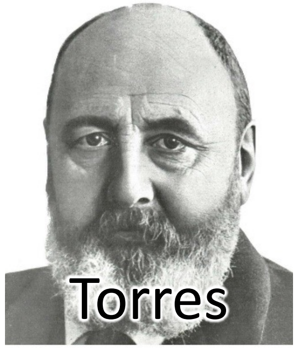
for some small epsilon value...

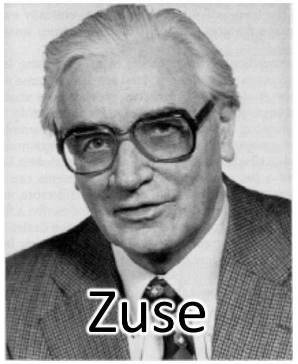
# Floating Point Conversions in C

- 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 are 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)

# More on Floating Point History

- Early days
  - First design with floating-point arithmetic in 1914 by Leonardo Torres y Quevedo
  - Implementations started in 1940 by Konrad Zuse, but with differing field lengths (usually not summing to 32 bits) and different subsets of the special cases
- IEEE 754 standard created in **1985**
  - Primary architect was William Kahan, who won a Turing Award for this work
  - Standardized bit encoding, well-defined behavior for *all* arithmetic operations




# 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)

# 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 = $2^{w-1} – 1$)
    - Size of exponent field determines our *representable range*
    - Outside of representable exponents is **overflow** and **underflow**
  - Mantissa approximates fractional portion of binary point
    - Size of mantissa field determines our representable *precision*
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes *rounding*

# 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 `ints`
    - 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 `ints` and `floats`!

# Summary

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

- Floating point encoding has many limitations
  - Overflow, underflow, rounding
  - Rounding is a HUGE issue due to limited mantissa bits and gaps that are scaled by the value of the exponent
  - Floating point arithmetic is NOT associative or distributive
- Converting between integral and floating point data types *does* change the bits

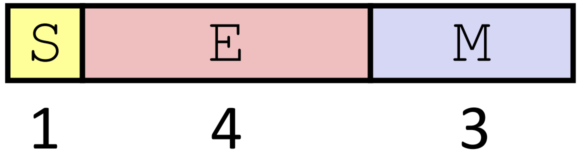
# BONUS SLIDES

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

# Floating Point Rounding

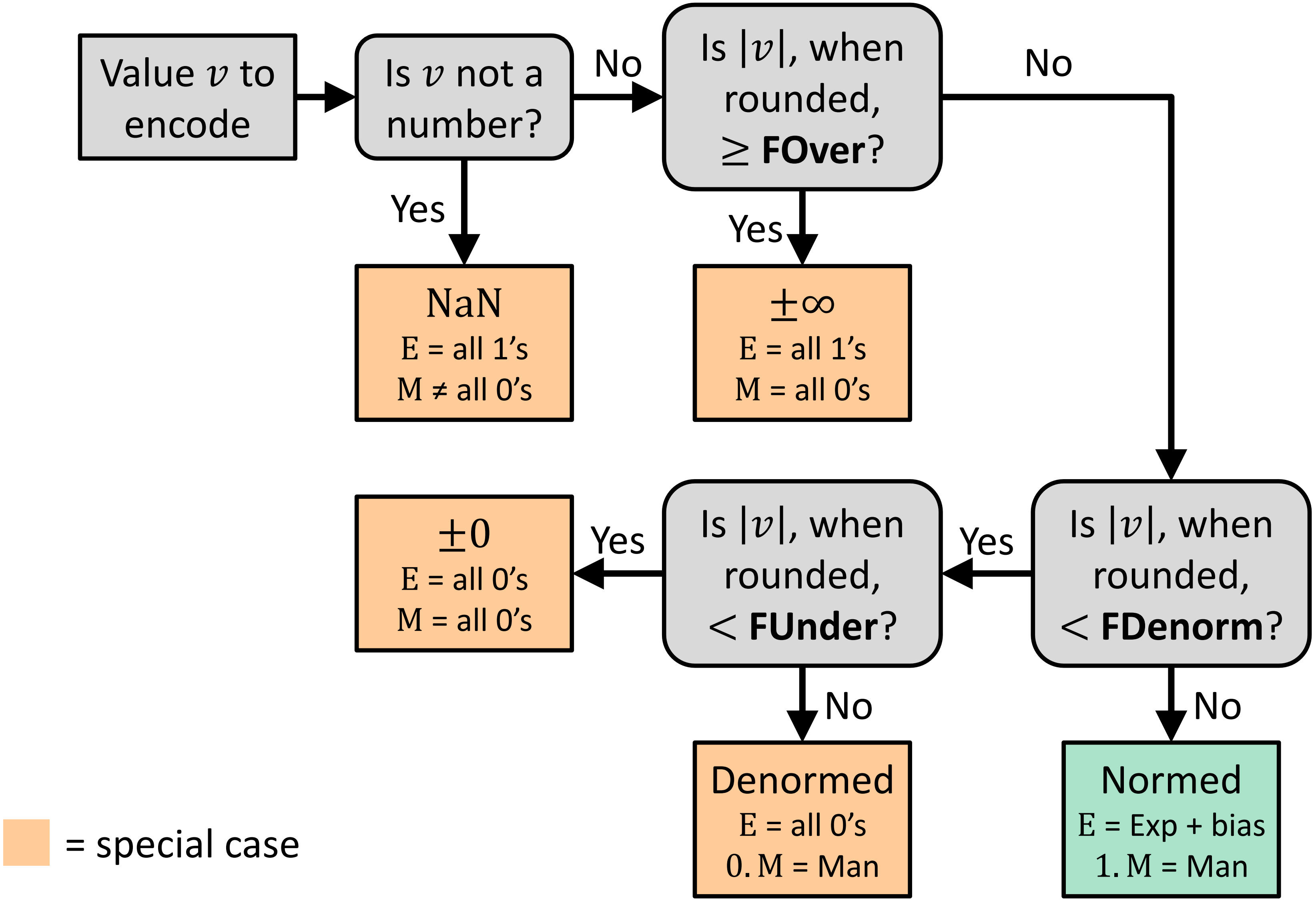
This is extra (non-testable) material

- The IEEE 754 standard actually specifies different rounding modes:
  - Round to nearest, ties to nearest even digit
  - Round toward $+\infty$ (round up)
  - Round toward $-\infty$ (round down)
  - Round toward 0 (truncation)
- In our tiny example:
  - Mantissa = 1.001 01 rounded to M = 0b001
  - Mantissa = 1.001 11 rounded to M = 0b010
  - Mantissa = 1.001 10 rounded to M = 0b010
  - Mantissa = 1.000 10 rounded to M = 0b000



This is extra (non-testable) material

# Floating Point Encoding Flow Chart



# Limits of Interest

This is extra (non-testable) material

- The following thresholds will help give you a sense of when certain outcomes come into play, but don't worry about the specifics:
  - **FOver** = $2^{\text{bias}+1}$ = $2^8$
    - This is just larger than the largest representable normalized number
  - **FDenorm** = $2^{1-\text{bias}}$ = $2^{-6}$
    - This is the smallest representable normalized number
  - **FUnder** = $2^{1-\text{bias}-m}$ = $2^{-9}$
    - $m$ is the width of the mantissa field
    - This is the smallest representable denormalized number

# Denormalized 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: $\pm\ 1.0...0_{two} \times 2^{-126} = \pm\ 2^{-126}$
  - Smallest denorm: $\pm\ 0.0...01_{two} \times 2^{-126} = \pm\ 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number

**So much closer to 0**

# 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 |