

# Floating Point

CSE 351 Summer, 2021

## Instructor:

Mara Kirdani-Ryan

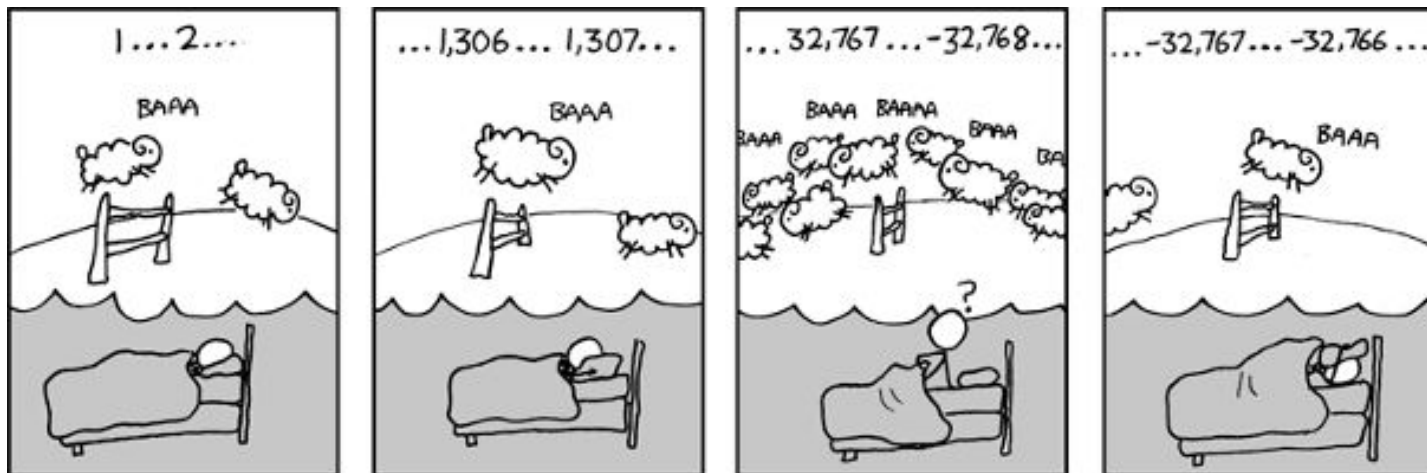
## Teaching Assistants:

Kashish Aggarwal

Nick Durand

Colton Jobes

Tim Mandzyuk



<http://xkcd.com/571/>

# Gentle and Loving Reminders!

- Everything's due at 8pm, unless we've talked!
- hw4, hw5 due tonight
- Lab 1a due tonight!!!
  - Submit `pointer.c` and `lab1Areflect.txt`
- Lab 1b due in a week (7/9), with hw6, hw7
  - Submit `aisle_manager.c`, `store_client.c` and `lab1Breflect.txt`
- US1 due the monday after (7/12)
  - We're here to help! Help each other!
- There's a holiday! Find some joy, not working!

# 1st Floorplan Critique on Wednesday!

- Last lecture of Unit #1!
- Wednesday, July 7, we'll have some in-class time to give each other feedback on unit summaries
  - Come with something to get feedback on!
  - It doesn't need to be polished, early designs are ok!
  - Most designers apologize for sketches, which is silly
  - A few prototypes, a design you're excited about, "what you have so far", all ok!
- Unit summaries will be due Monday, July 12

**How are you feeling today?  
How do you feel about  
learning?**

# Learning Objectives

- At the end of this lecture, you should be able to...
  - Convert between floating point and decimal encoding
  - Give examples of special cases in floating point, along with their binary representations
  - Explain why we shouldn't compare floats for equality
  - Understand the limitations of floating point representations
  - Give a few examples of knowledge-shame manifesting in computing
  - Reflect on this unit to make a floorplan!
    - Though, this one you could do earlier...

# Fixed Point Representation

- Implied binary point. Two example schemes:
  - #1: the binary point is between bits 2 and 3  
 $b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ \text{[.]} \ b_2 \ b_1 \ b_0$
  - #2: the binary point is between bits 4 and 5  
 $b_7 \ b_6 \ b_5 \ \text{[.]} \ b_4 \ b_3 \ b_2 \ b_1 \ b_0$
- Fixed point placement is a tradeoff between *range* and *precision* -- both are fixed!
  - 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  $1.2 \times 10^7$       In C: 1.2e7
    - Not 0.0000012, but  $1.2 \times 10^{-6}$       In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but  $1.1 \times 2^4$
    - Not 0.000101, but  $1.01 \times 2^{-4}$
- We have to divvy up bits (32) between:
  - the sign (1 bit)
  - the mantissa (significand)
  - the exponent

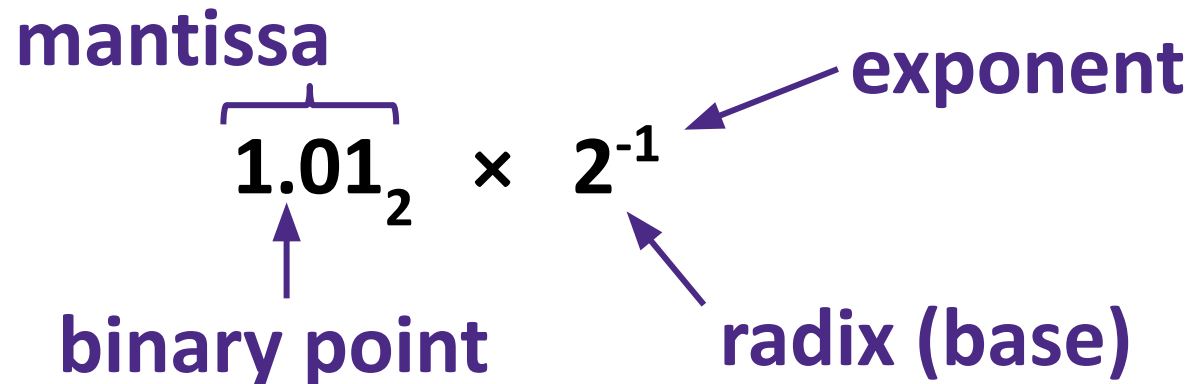
# Scientific Notation (Decimal)

The diagram illustrates the components of the scientific notation  $6.02_{10} \times 10^{23}$ . A bracket above the digits '6.02' is labeled 'mantissa'. An arrow points to the subscript '10' and is labeled 'decimal point'. An arrow points to the base '10' in the power term and is labeled 'radix (base)'. An arrow points to the exponent '23' and is labeled 'exponent'.

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



# Scientific Notation (Binary)



The diagram illustrates the components of the binary scientific notation  $1.01_2 \times 2^{-1}$ . The mantissa is  $1.01_2$ , with a bracket above it labeled "mantissa". The exponent is  $2^{-1}$ , with an arrow pointing to it labeled "exponent". The binary point is the dot in  $1.01_2$ , with an arrow pointing to it labeled "binary point". The radix (base) is  $2$ , with an arrow pointing to it labeled "radix (base)".

- floating point from the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)

# Scientific Notation Translation

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

# Floating Point Topics

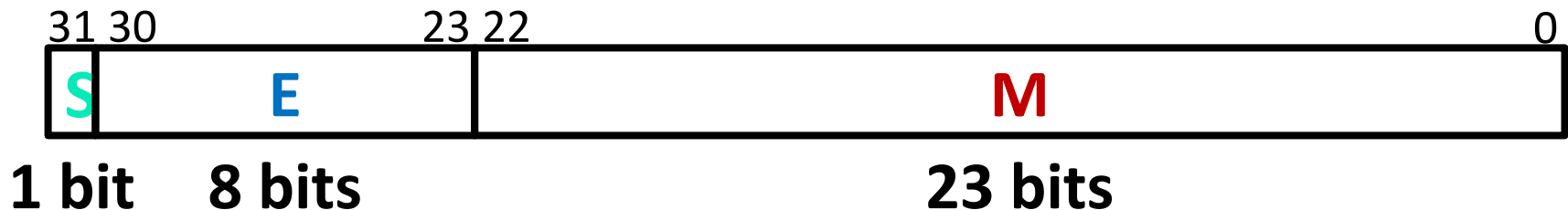
- [illegible]

# IEEE Floating Point

- IEEE 754
  - 1985, uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and results of 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**
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - **Float ops can be an order of magnitude slower than integer ops!**
    - **Also, much more energy intensive, see BTC**

# Floating Point Encoding

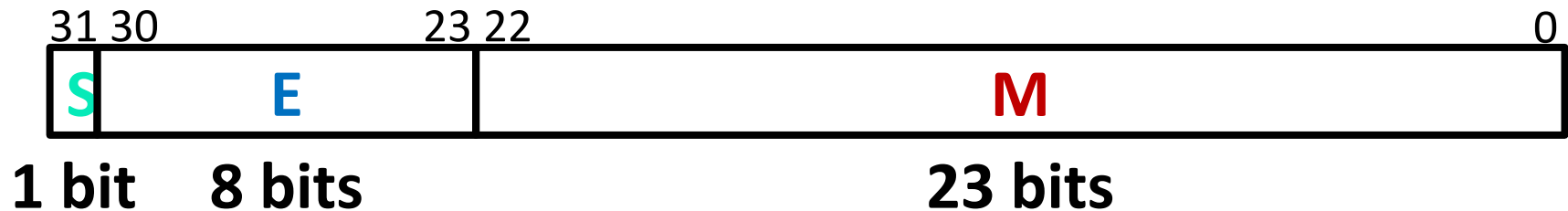
- Use normalized, base 2 scientific notation:
  - Value:  $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
  - Bit Fields:  $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$
- Representation Scheme:
  - **Sign bit** (0 is positive, 1 is negative)
  - **Mantissa** (a.k.a. significand) is the fractional 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**



# Bias Field: Allow negative exponents

- Bias =  $2^{w-1}-1$
- $E = \text{Exponent} + \text{bias}$  (or  $\text{Exponent} = E - \text{Bias}$ )
- IEEE float have  $w=8$ ,  $\text{bias}=127$
- Exponent = 1,  $E = 1 + 127 = 128$
- Exponent = 127;  $E = 127 + 127 = 254$
- Exponent = -63;  $E = -63 + 127 = 64$
- E results from signed arithmetic, represent as unsigned

# The Mantissa (Fraction) Field



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

- Note the implicit 1 in front of the M bit vector
  - Ex: 0b 0011 1111 1100 0000 0000 0000 0000 0000  
is read as  $1.1_2 = 1.5_{10}$ , *not*  $0.1_2 = 0.5_{10}$
  - Gives us an extra bit of *precision*
- Mantissa “limits”
  - Low values near  $M = 0b0\dots0$  are close to  $2^{\text{Exp}}$
  - High values near  $M = 0b1\dots1$  are close to  $2^{\text{Exp}+1}$

# Normalized FP Conversions

## ○ FP $\rightarrow$ Decimal

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

## ○ Decimal $\rightarrow$ 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  $E$  as unsigned.
5. The first bits after the leading 1 that fit are encoded into  $M$ .



# Question!

- What value is encoded by the following float?
  - 0b 0 10000000 11000000000000000000000000000000

💙 + 1.5

💜 + 2.75

❤️ + 0.75

🖤 + 3.5

🧊 Help!

# Question!

- What value is encoded by the following float?
  - 0b 0 10000000 11000000000000000000000000000000

💙 + 1.5

💜 + 2.75

❤️ + 0.75

🖤 + 3.5

🥶 Help!

$$\text{Exp} = E - \text{Bias} = 128 - 127 = 1$$

$$1.11 \times 2^1 = 11.1 = 3.5$$

# Precision and Accuracy

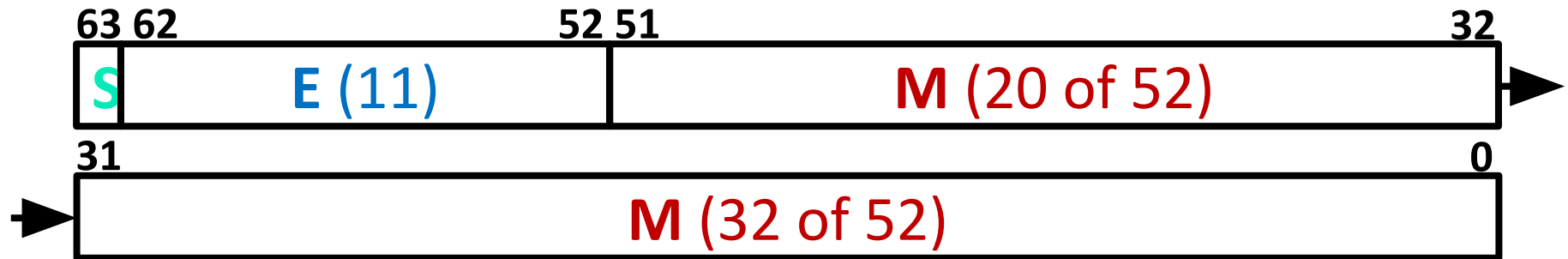
- **Precision** is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy
- **Accuracy** is a measure of the difference between the *actual value of a number* and its computer representation
  - *High precision permits high accuracy but doesn't guarantee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

# Precision and 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

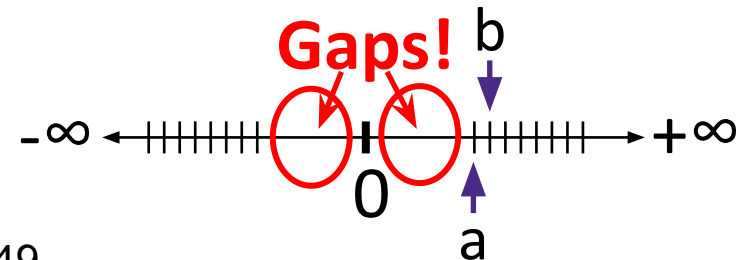
# Representing Very Small Numbers

- But wait... what happened to zero?
  - Using standard encoding  $0x00000000 =$
  - *Special case:* **E** and **M** all zeros  $= 0$ 
    - Two zeros! But at least  $0x00000000 = 0$  like integers

- New numbers closest to 0:


- $a = 1.\textcolor{red}{0}\dots\textcolor{red}{0}_2 \times 2^{-126} = 2^{-126}$
- $b = 1.\textcolor{red}{0}\dots\textcolor{red}{0}\textcolor{red}{1}_2 \times 2^{-126} = 2^{-126} + 2^{-149}$

- Normalization and implicit 1 are to blame
- *Special case:* **E** = 0, **M**  $\neq 0$  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:  $\pm 1.\underbrace{0\dots0}_{\text{two}} \times 2^{-126} = \pm 2^{-126}$  
  - Smallest denorm:  $\pm \underbrace{0.\dots01}_{\text{two}} \times 2^{-126} = \pm 2^{-149}$ 
    - There is still a gap between zero and the smallest denormalized number

So much  
closer to 0

# Other Special Cases

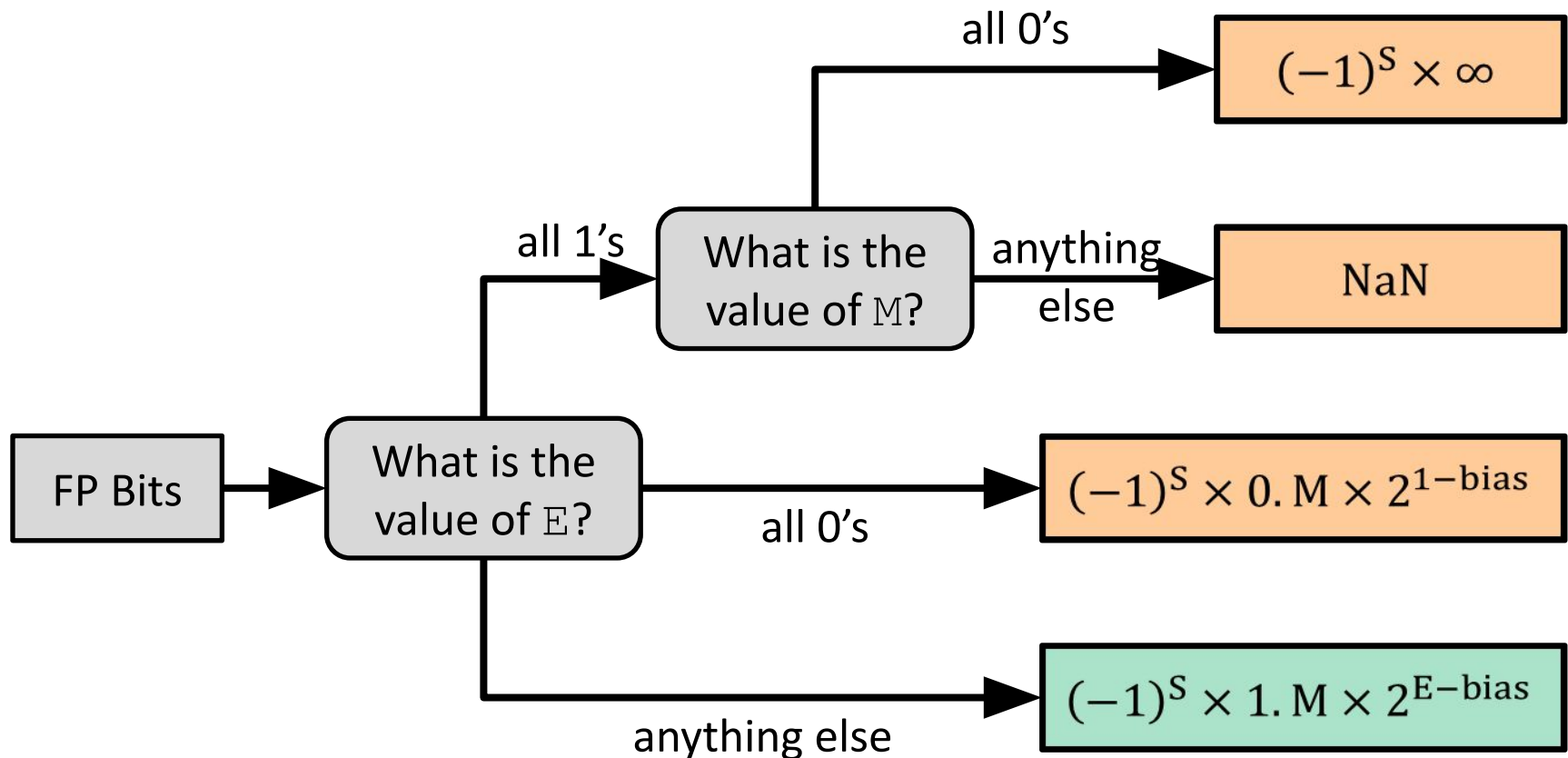
- $E = 0xFF$ ,  $M = 0$ :  $\pm \infty$ 
  - e.g. division by 0
  - Still work in comparisons!
- $E = 0xFF$ ,  $M \neq 0$ : Not a Number (NaN)
  - e.g. square root of negative number,  $0/0$ ,  $\infty - \infty$
  - NaN propagates through computations
  - Value of  $M$  can be useful in debugging
- New largest value (besides  $\infty$ )?
  - $E = 0xFF$  has now been taken!
  - $E = 0xFE$  has largest:  $1.1\dots1_2 \times 2^{127} = 2^{128} - 2^{104}$



# Floating Point Encoding Summary

| E           | M        | Meaning          |
|-------------|----------|------------------|
| 0x00        | 0        | $\pm 0$          |
| 0x00        | non-zero | $\pm$ denorm num |
| 0x01 – 0xFE | anything | $\pm$ norm num   |
| 0xFF        | 0        | $\pm \infty$     |
| 0xFF        | non-zero | NaN              |

# Floating Point Interpretation Flow Chart



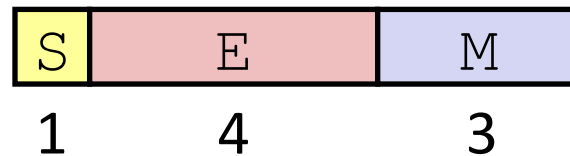
= special case

# Floating point topics

- [illegible]

# Tiny Floating Point Representation

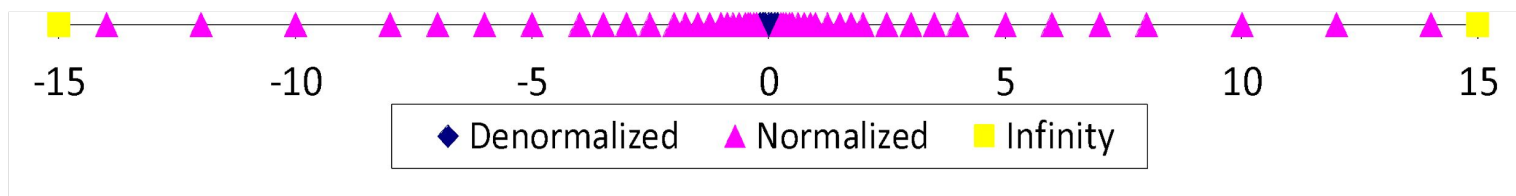
- 8 bit floating point to talk about some key points



- Assume same properties of IEEE floats
  - Bias =  $2^{w-1} - 1 = 7$
  - Zero encoding = 1000 0000 or 0000 0000
  - +Inf = 0 1111 000
  - Largest Normalized number = 0 1110 111
  - Smallest Normalized number = 0 0001 000

# 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?
  - What is this “step” when **Exp** = 0?
  - What is this “step” when **Exp** = 100?
- Distribution of values is denser toward zero



# Floating Point Rounding

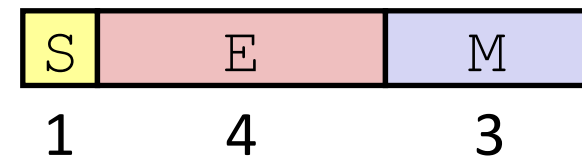
This is extra  
(non-testable)  
material

❖ 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)

❖ In our tiny example:

- Man = 1.001 01 rounded to M = 0b001
- Man = 1.001 11 rounded to M = 0b010
- Man = 1.001 10 rounded to M = 0b010



# Floating Point Ops: Basic Idea

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



- $x \oplus_f y = \text{Round}(x \oplus y)$
- $x \otimes_f y = \text{Round}(x \otimes y)$
- 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

# Spooky things in FP

- Overflow yields  $\pm\infty$  and underflow yields 0
- Can still use  $\pm\infty$  and NaN in operations!
  - Result usually  $\pm\infty$  or NaN, but not intuitive
- FP Ops don't work like real math! **Rounding!**
  - Not associative!
 
$$(3.14 + 1e100) - 1e100 \neq 3.14 + (1e100 - 1e100)$$

0
3.14
  - Not distributive!
 
$$100 * (0.1 + 0.2) \neq 100 * 0.1 + 100 * 0.2$$

30.0000000000000003553
30
  - Not cumulative!
    - Repeatedly adding a small number to a large one might do nothing!



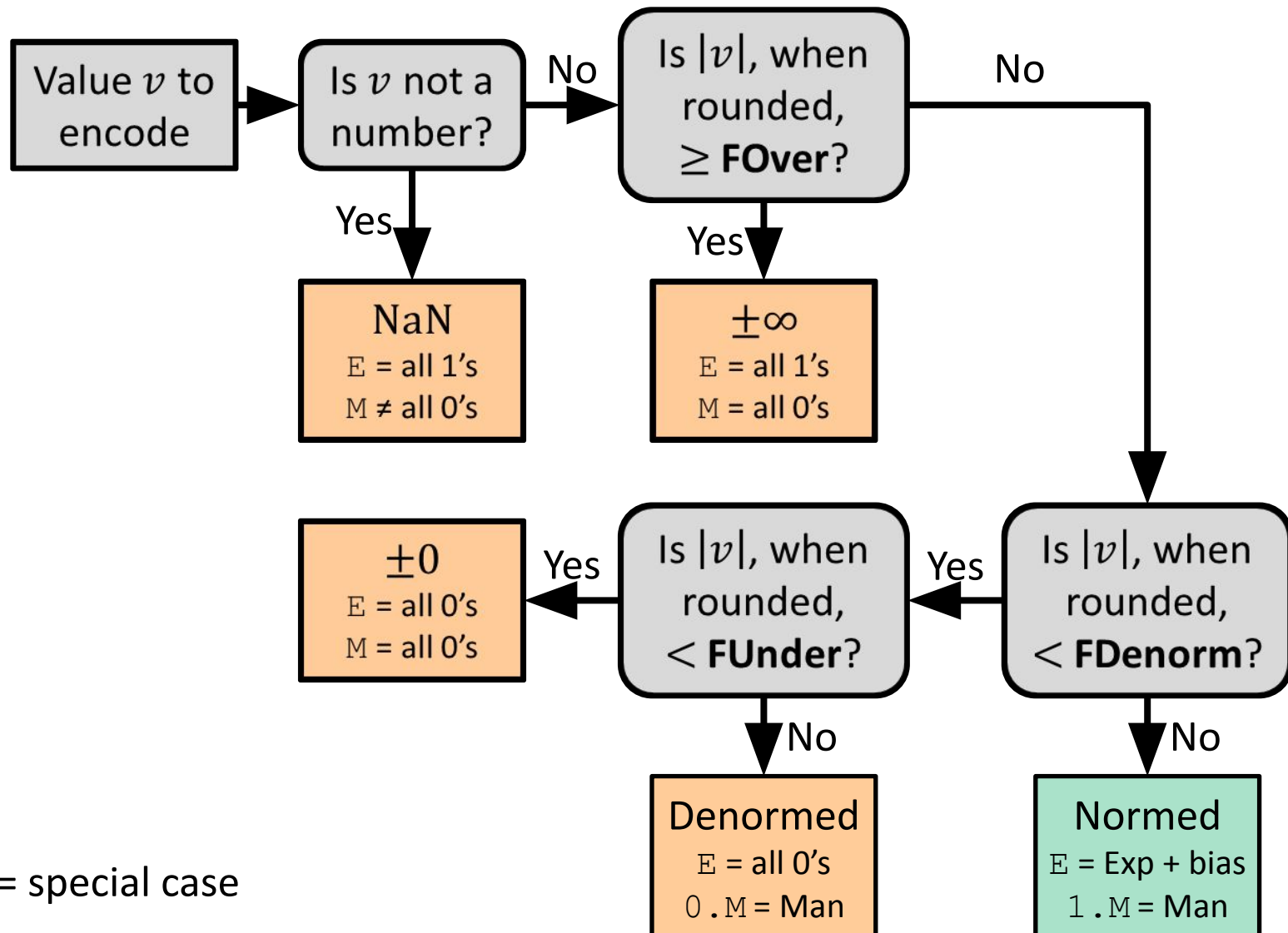
# Aside: Limits of Interest

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

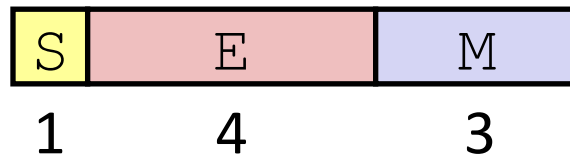
# Floating Point Encoding Flow Chart



**No one understands  
everything about FP!  
Not even researchers!**

# Question!

- Using our **8-bit** representation, what value get stored when we try to encode **384** =  $2^8 * 2^7$



+ 256



+ 384



+ Infinity



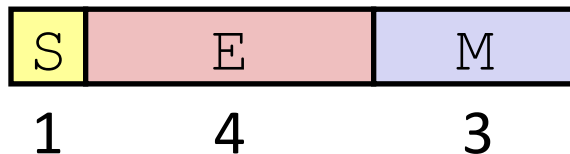
NaN



Help!!

# Question!

- Using our **8-bit** representation, what value gets stored when we try to encode **2.625** =  $2^1 + 2^{-1} + 2^{-3}$ ?



💙 + 2.5

💜 + 2.625

❤️ + 2.75

🖤 + 3.25

🧊 Help!!



# 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
- 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/double changes the bit representation!**
  - `int → float`; might round, won't overflow
  - `int → double`; works fine, lots of space
  - `long → double`; might be fine, depends on long size
  - `double/float → int`;
    - Undefined when NaN or Inf
    - Truncates fractional part -- rounds to zero

# 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 `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 compare floating point values for equality!
- Careful when converting between `ints` and `floats`!



# Number Representation Matters!

*Especially if you care about money!*

- **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 division) HW bug (\$475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch (\$193 million)

# That's the end of U1!

# Unit 1, Technical Summary

- Binary, Hex, ASCII (L1)
- Representation of memory in hardware
  - Alignment, endianness (L2)
  - Variable assignment, arrays, pointers (L3)
  - Address manipulation with pointer arithmetic (L3)
- Data & Operations
  - Boolean logic; Logical & bitwise ops (L4)
  - Signed vs. unsigned integers, and their limits (L5)
    - Two's complement, motivated by sign-magnitude
  - Floating point representations and limits (L6)
    - IEEE standard, motivated by fixed point

# Unit 1, Socio-technical summary

- Standards encoding priorities of creators
- Values in first modern computers
  - What is “robot” work? Who’s done it?
- Values in C (C == Camping; *explore the frontier*)
- ~~Insulation~~
- Values in the original first computers
- Shame (this lecture)

# Why'd we learn this?

# Discussion Norms

- Everyone's experience is valid!
- Feelings are valid too!
- Everyone should have space to share!
  - Though, no one's required to share. A “yes” or “no” without explanation is more than enough.
  - Saying “I don't feel like sharing” is good too!
- **I've got a whole heap of hurt here!**
  - Some of y'all might as well
  - Try to be compassionate to everyone!
- A brief grounding...

**Breakouts: Shame!**  
**Come up with a**  
**working definition!**

**We'll share out in chat!**

**“I want to make sure  
that we emphasize,  
don’t compare floats  
for equality”**

*Why?*



# The “Embarrassed” list

- **Problem:** What to teach CS undergrads?
  - We have some ideas, but it's good to revisit
- ***The Embarrassed List***
  - Go around to faculty, ask “What would you be embarrassed if students graduated not knowing?”
  - Create a list from these responses
  - Realize that the list is way too long to teach everything
  - *Start compromising!*
- This is more about compromise than shame
  - *Let's dig deeper into assumptions!*

# A Caveat:

- My opinions, my view of the space that we're in
- Not vilifying anyone, curricular development is really challenging, and this is pretty good
- More based on my experiences:
  - undergrad
  - internships
  - other CS spaces
  - Grad school in computer architecture (CEd's lovely)
- My experience of CS culture!
  - Maybe relevant to some of you!

# Embarrassed list; what's going on?

- **Problem:** What to teach CS undergrads?
  - **Value:** Students should *represent this institution well!*
- **We don't want anyone to embarrass us!**

# Embarrassed list; what's going on?

- **Problem:** What to teach CS undergrads?
  - **Value:** Students should *represent this institution well!*
- **We don't want anyone to embarrass us!**
  - We don't want to be **ashamed** of our students
- Let's make sure they know....
  - Vim, it might come up in an interview
    - (in an interview, for a 21sp TA)
  - How to interact with a terminal
  - Not to compare floats for equality
  - ...
  - ...

# Embarrassed list; what's going on?

- **Problem:** What to teach CS undergrads?
  - **Value:** Students should *represent this institution well!*
- **We don't want anyone to embarrass us!**
  - We don't want to be **ashamed** of our students
- Let's make sure they know....
  - Vim, it might come up in an interview
    - (in an interview, for a 21sp TA)
  - How to interact with a terminal
  - Not to compare floats for equality
  - ...
  - ...
  - "how to be a computer scientist"

# Shame & Legitimacy

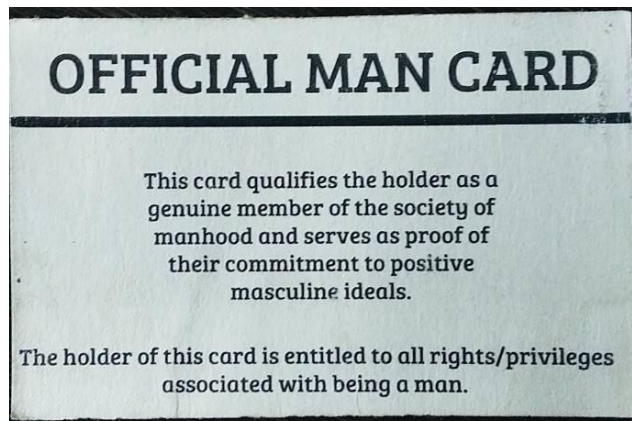
- “We need to make sure that our graduates are **seen as legitimate** computer scientists by employers”
  - “Walk the walk, talk the talk, be seen doing it”
  - Illegitimate students might embarrass us!
    - Not really, but it might change the *reputation* of the Allen School

# Shame & Legitimacy

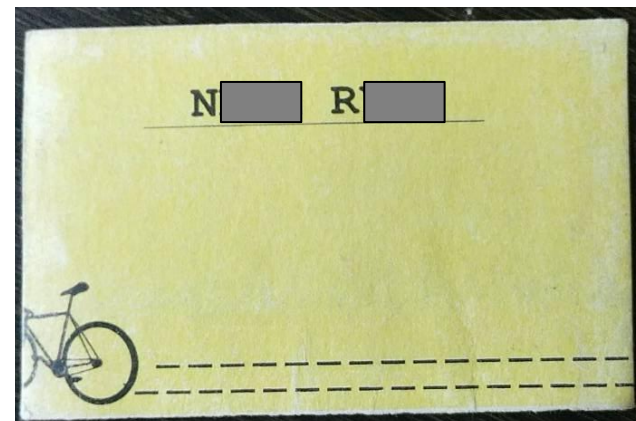
- “We need to make sure that our graduates are **seen as legitimate** computer scientists by employers”
  - “Walk the walk, talk the talk, be seen doing it”
  - Illegitimate students might embarrass us!
    - Not really, but it might change the *reputation* of the Allen School
- There’s so much tied up with legitimacy in CS...
- It’s like a **man card**

# “man card”

*n. Requirement to be “accepted” as a “respectable” member of the “male” community. Can be revoked by other “respectable males” for doing “not-respectable male” things. (scare quotes mine)*



*Front*



*Back*



**What would be  
required of a  
“Computer Scientist”  
card?**

# Don't compare floats for equality!

# Allen School & Shame (EE, Microsoft)

- I haven't felt policed in the same way here!
  - It's been really lovely, honestly
- Other spaces might be less kind
- You're here, regardless of whether you believe...
  - That you need to learn Vim
  - That you need to understand FP special cases
  - That you need to understand how a computer works
- Hopefully, wherever you end up, you can find people that don't weaponize their insecurity and shame for you not knowing something

# What I remember about FP

- **Don't compare floats for equality!**

...

- Something about a mantissa?
- And there's an exponent that does something?
- Let me check my notes...
- ...oh yeah, there's this formula!
- And these special cases to note!
- Got it, got it.

**I'm not saying that this isn't  
important...**

**...but loving yourself wholeheartedly  
is probably much more important**

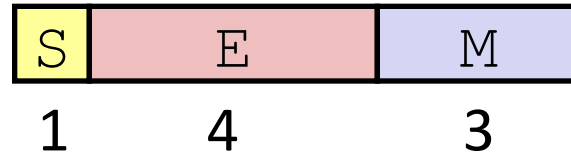
**If someone shames you for not knowing something, and they try to take away your “CS card”...**

**... Ask them, “Who hurt you?”  
You’ll maintain the moral high ground!**

# BONUS SLIDES

An example that applies the IEEE Floating Point concepts to a smaller (8-bit) representation scheme. [More information for the curious and interested!](#)

# Tiny Floating Point Example



- 8-bit Floating Point Representation
  - The sign bit is in the most significant bit (MSB)
  - The next four bits are the exponent, with a bias of  $2^{4-1}-1 = 7$
  - The last three bits are the mantissa
- Same general form as IEEE Format
  - Normalized binary scientific point notation
  - Similar special cases for 0, denorm numbers, NaN,  $\infty$



# Dynamic Range (Positive Only)

S E M Exp Value

Denormalized  
numbers

|     |      |     |    |                      |
|-----|------|-----|----|----------------------|
| 0   | 0000 | 000 | -6 | 0                    |
| 0   | 0000 | 001 | -6 | $1/8 * 1/64 = 1/512$ |
| 0   | 0000 | 010 | -6 | $2/8 * 1/64 = 2/512$ |
| ... |      |     |    |                      |
| 0   | 0000 | 110 | -6 | $6/8 * 1/64 = 6/512$ |
| 0   | 0000 | 111 | -6 | $7/8 * 1/64 = 7/512$ |

closest to zero

largest denorm

Normalized  
numbers

|     |      |     |    |                      |
|-----|------|-----|----|----------------------|
| 0   | 0001 | 000 | -6 | $8/8 * 1/64 = 8/512$ |
| 0   | 0001 | 001 | -6 | $9/8 * 1/64 = 9/512$ |
| ... |      |     |    |                      |
| 0   | 0110 | 110 | -1 | $14/8 * 1/2 = 14/16$ |
| 0   | 0110 | 111 | -1 | $15/8 * 1/2 = 15/16$ |
| 0   | 0111 | 000 | 0  | $8/8 * 1 = 1$        |
| 0   | 0111 | 001 | 0  | $9/8 * 1 = 9/8$      |
| 0   | 0111 | 010 | 0  | $10/8 * 1 = 10/8$    |
| ... |      |     |    |                      |
| 0   | 1110 | 110 | 7  | $14/8 * 128 = 224$   |
| 0   | 1110 | 111 | 7  | $15/8 * 128 = 240$   |

smallest norm

closest to 1 below

closest to 1 above

largest norm

|   |      |     |         |  |
|---|------|-----|---------|--|
| 0 | 1111 | 000 | n/a inf |  |
|---|------|-----|---------|--|

# Special Properties of Encoding

- Floating point zero ( $0^+$ ) exactly the same bits as integer zero
  - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider  $0^- = 0^+ = 0$
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity