

# Floating Point II

CSE 351 Autumn 2021

## Instructor:

Justin Hsia

## Teaching Assistants:

Allie Pflieger

Anirudh Kumar

Assaf Vayner

Atharva Deodhar

Celeste Zeng

Dominick Ta

Francesca Wang

Hamsa Shankar

Isabella Nguyen

Joy Dang

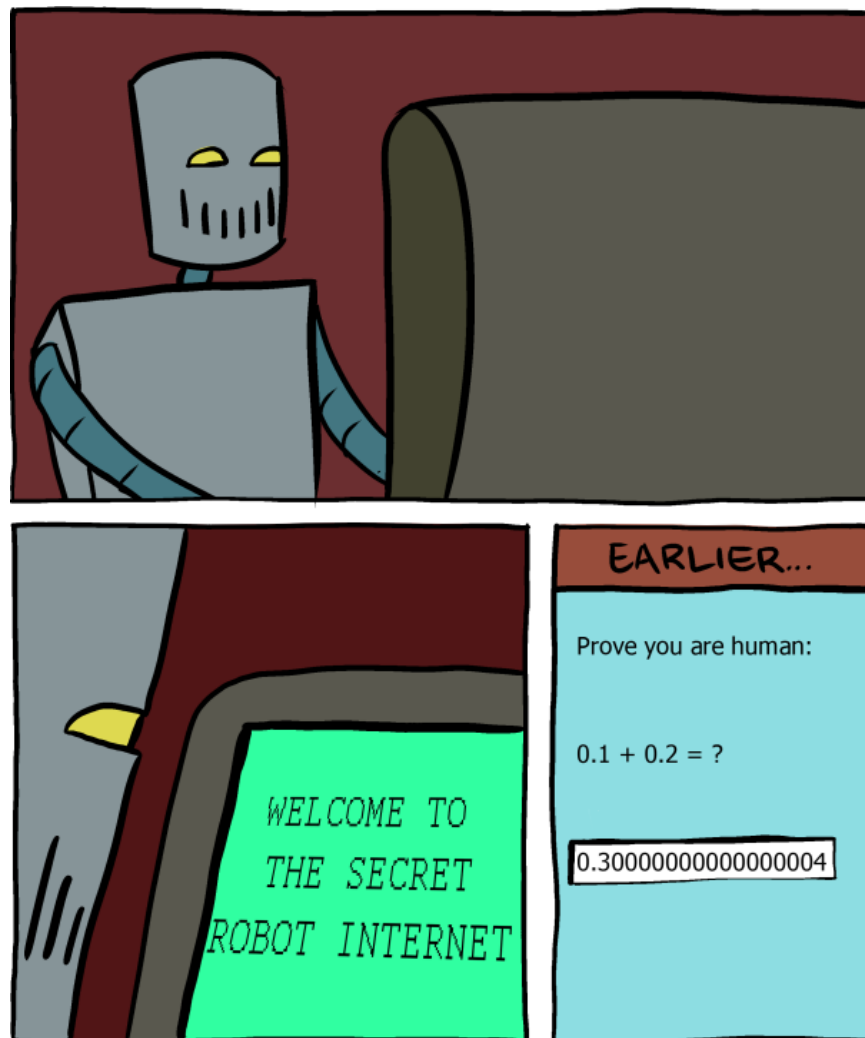
Julia Wang

Maggie Jiang

Monty Nitschke

Morel Fotsing

Sanjana Chintalapati



<http://www.smbc-comics.com/?id=2999>

# Administrivia

- ❖ hw6 due Friday, hw7 due Monday
- ❖ Lab 1a: last chance to submit is tonight @ 11:59 pm
  - One submission per partnership
  - Make sure you check the Gradescope autograder output!
  - Grades hopefully released by end of Sunday (10/17)
- ❖ Lab 1b due Monday (10/18)
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`
- ❖ Section tomorrow on Integers and Floating Point

# Getting Help with 351

- ❖ Lecture recordings, readings, inked slides
- ❖ Form a study group!
  - Good for everything but labs, which should be done in pairs
  - Communicate regularly, use the class terminology, ask and answer each others' questions, show up to OH together
- ❖ Attend office hours
  - Use the OH queue, but can also chat with other students there – help each other learn!
- ❖ Post on Ed Discussion
- ❖ Request a 1-on-1 meeting
  - Available on a limited basis for special circumstances

# Reading Review

- ❖ Terminology:
  - Special cases
    - Denormalized numbers
    - $\pm\infty$
    - Not-a-Number (NaN)
  - Limits of representation
    - Overflow
    - Underflow
    - Rounding
  
- ❖ Questions from the Reading?

# Review Questions

❖ What is the value of the following floats?

▪  $0x00000000 \Rightarrow S=0, E=0, M=0 \Rightarrow \boxed{+0}$

▪  $0xFF800000 \Rightarrow S=1, E=\text{all 1's}, M=0 \Rightarrow \boxed{-\infty}$



❖ For the following code, what is the smallest value of  $n$  that will encounter a limit of representation?

```
float f = 1.0; // 2^0
for (int i = 0; i < n; ++i)
    f *= 1024; // 1024 = 2^10
printf("f = %f\n", f);
E_max = 0xFE, Exp_max = 254 - 127 = 127
```

$n$	$f$
1	$2^{10}$
2	$2^{20}$
3	$2^{30}$
4	$2^{40}$
⋮	⋮

always  $M=0$

for  $\boxed{n=13}$ , we hit  $2^{130}$ , which causes  $\boxed{\text{overflow}}$

# Floating Point Encoding Summary (Review)

	E	M	Interpretation
smallest E (all 0's)	0x00	0	$\pm 0$
	0x00	non-zero	$\pm$ denorm num
everything else	0x01 – 0xFE	anything	$\pm$ norm num
largest E (all 1's)	0xFF	0	$\pm \infty$
	0xFF	non-zero	NaN

# Special Cases

❖ But wait... what happened to zero?

- *Special case:* **E** and **M** all zeros = 0
- Two zeros! But at least  $0x00000000 = 0$  like integers  
 $0x80000000 = -0$

❖ **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 *(tells you cause of NaN)*

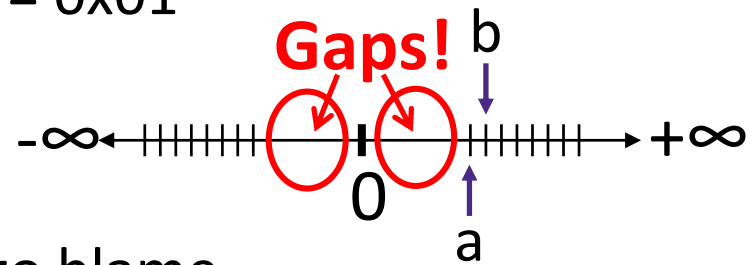
# New Representation Limits

## ❖ New largest value (besides $\infty$ )?

- $E = 0xFF$  has now been taken!
- $E = 0xFE$  has largest:  $1.\overset{23 \text{ ones}}{1} \times 2^{127} = 2^{128} - 2^{104}$   
↳ 254-bias

## ❖ New numbers closest to 0:

- $E = 0x00$  taken; next smallest is  $E = 0x01$   $Exp = -126$
- $a = 1.0...0_2 \times 2^{-126} = 2^{-126}$
- $b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$  ↳ 23



- Normalization and implicit 1 are to blame
- *Special case:*  $E = 0, M \neq 0$  are **denormalized numbers** (0.M)  
normalized: 1.M



# Denorm Numbers

This is extra  
(non-testable)  
material

## ❖ Denormalized numbers

- No leading 1
- Uses implicit exponent of  $-126$  even though  $E = 0x00$

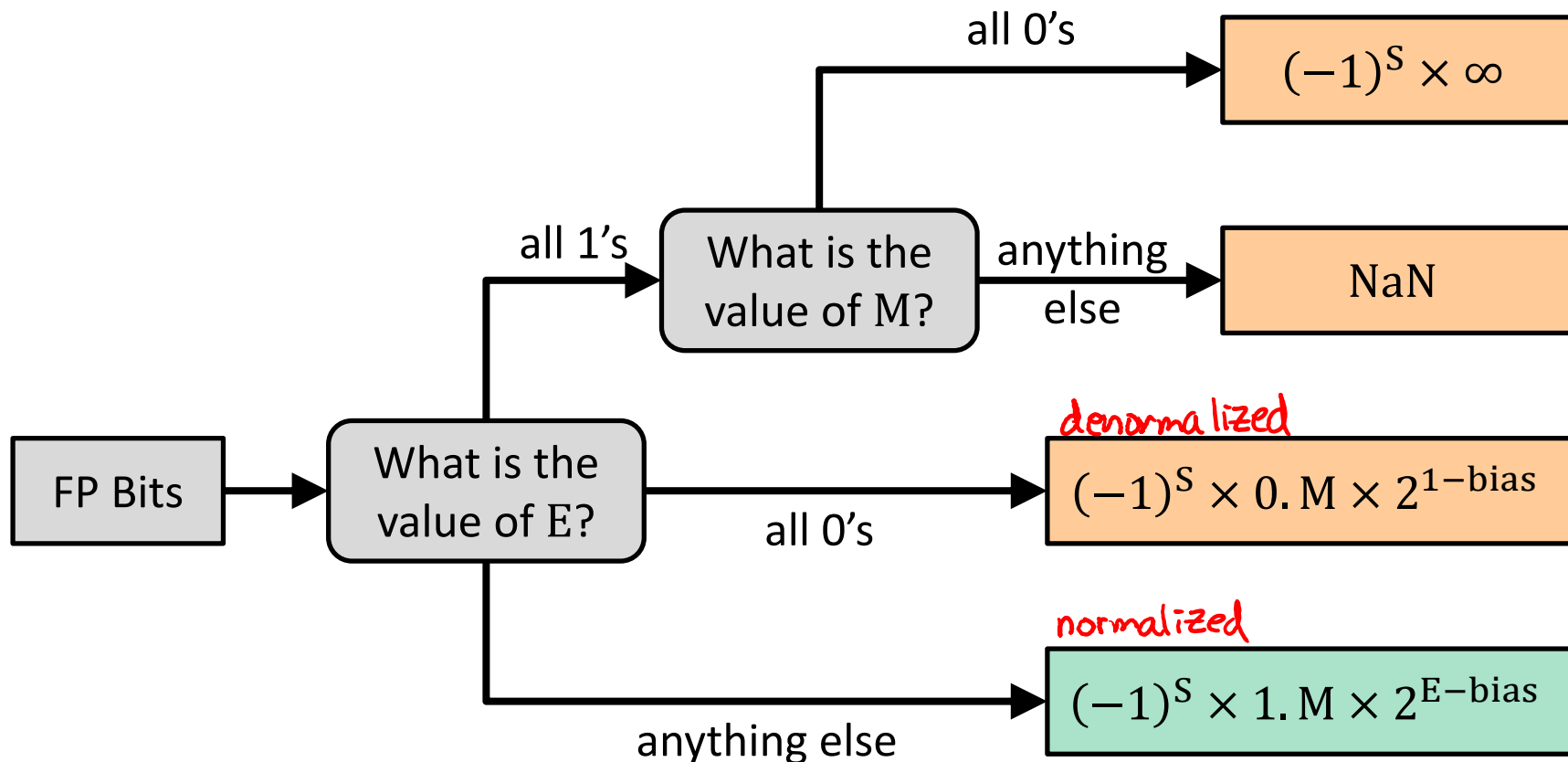
## ❖ Denormalized numbers close the gap between zero and the smallest normalized number

- Smallest norm:  $\pm 1.0\dots0_{\text{two}} \times 2^{-126} = \pm 2^{-126}$
- Smallest denorm:  $\pm 0.0\dots01_{\text{two}} \times 2^{-126} = \pm 2^{-149}$

- There is still a gap between zero and the smallest denormalized number

So much  
closer to 0

# Floating Point Decoding Flow Chart

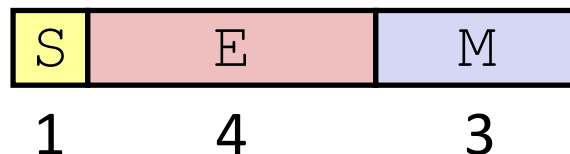


= special case



# Tiny Floating Point Representation

- ❖ We will use the following **8-bit** floating point representation to illustrate some key points:



- ❖ Assume that it has the same properties as IEEE floating point:

- bias =  $2^{w-1} - 1 = 2^{4-1} - 1 = 7$
  - encoding of  $-0 = 0b\ 1\ 0000\ 000 = 0x\ 80$
  - encoding of  $+\infty = 0b\ 0\ 1111\ 000 = 0x\ 78$
  - encoding of the largest (+) normalized # =  $0b\ 0\ 1110\ 111 = 0x\ 77$
  - encoding of the smallest (+) normalized # =  $0b\ 0\ 0001\ 000 = 0x\ 08$
- $\rightarrow 1.111_2 \times 2^{14-7}$   
 $\hookrightarrow 1.0_2 \times 2^{1-7}$

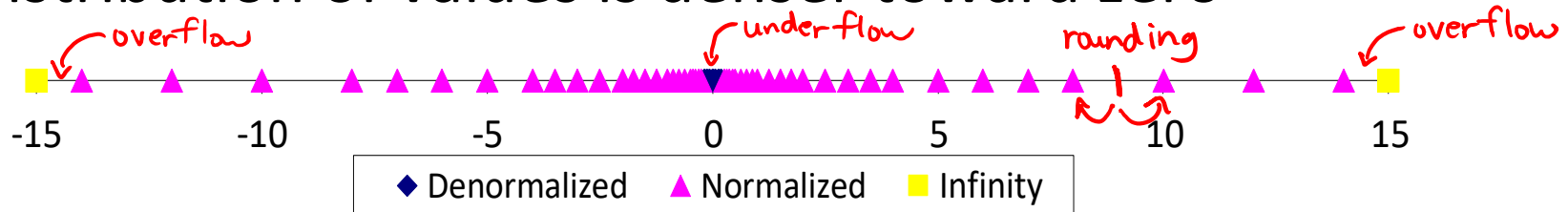
# Distribution of Values (Review)

- ❖ 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?       $2^{-23}$
  - What is this "step" when **Exp** = 100?       $2^{77}$

if  $M = 0b\ 0\dots 00$ , then  $2^{\text{Exp}} \times 1.0$   
 if  $M = 0b\ 0\dots 01$ , then  $2^{\text{Exp}} \times (1 + 2^{-23})$   


---

 diff =  $2^{\text{Exp}-23}$



# Floating Point Rounding

This is extra  
(non-testable)  
material

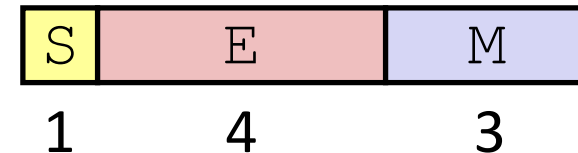
- ❖ The IEEE 754 standard actually specifies different rounding modes:

★ 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 (down) *< half*
- Man = 1.001/11 rounded to M = 0b010 (up) *> half*
- Man = 1.001/10 rounded to M = 0b010 (up) *== half*
- Man = 1.000/10 rounded to M = 0b000 (down) *even digit*



# Floating Point Operations: Basic Idea

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



- ❖  $x +_f y = \text{Round}(x + y)$
- ❖  $x *_f y = \text{Round}(x * y)$
  
- ❖ 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  $\pm\infty$  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:  $(3.14 + 1e100) - 1e100 \neq 3.14 + (1e100 - 1e100)$ 

$10^{100}$   
 $0$

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

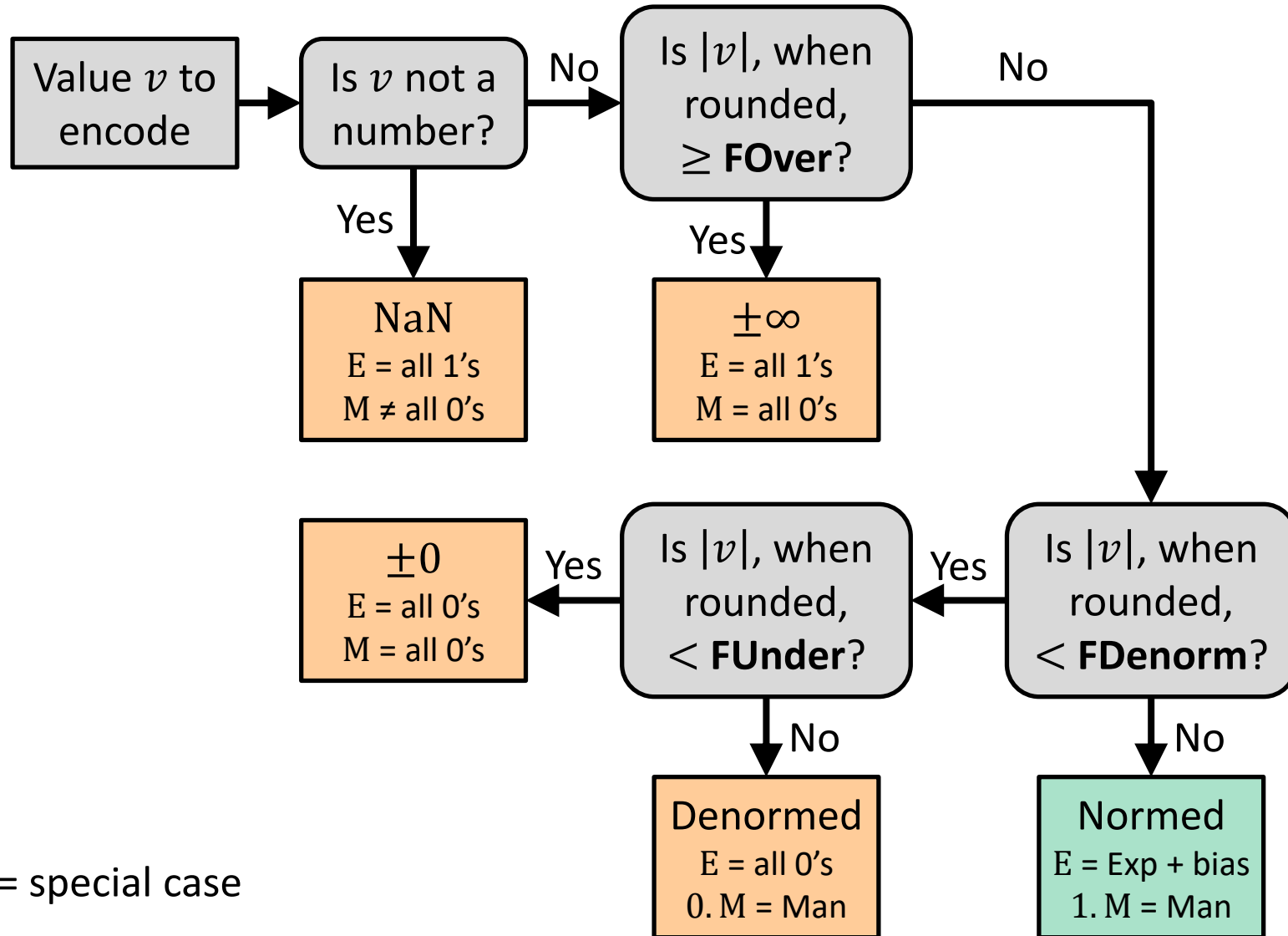
$30.0000000000000003553$

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



# Floating Point Encoding Flow Chart

This is extra (non-testable) material



= special case

# 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 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 use  $\text{abs}(f1 - f2) < 2^{-20}$   
↑ 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 ints 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)

# Exploration Question

- ❖ We execute the following code in C. How many bytes are the same (value and position) between `i` and `f`?

```
int i = 384; // 2^8 + 2^7
float f = (float) i;
```

- A. 0 bytes**
- B. 1 byte
- C. 2 bytes
- D. 3 bytes
- E. We're lost...

$$= 0b \overset{8}{1} \overset{7}{1} \overset{6}{0} \overset{5}{0} \overset{4}{0} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{0}$$


---


$$= 1.1_2 \times 2^8$$

$$S = 0$$

$$E = 8 + 127 = 135$$

$$= 0b 1000 0111$$

$$M = 0b 10 \dots 0$$


---


$$0b 0 1000 0111 100 \dots 0$$

`i` stored as `0x 00 00 01 80`  
`f` stored as `0x 43 C0 00 00`

# Discussion Questions

- ❖ How do you feel about floating point?
  - Do you feel like the limitations are acceptable?
  
  - Does this affect the way you'll think about non-integer arithmetic in the future?
  
  - Are there any changes or different encoding schemes that you think would be an improvement?

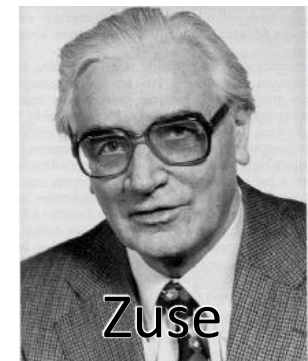
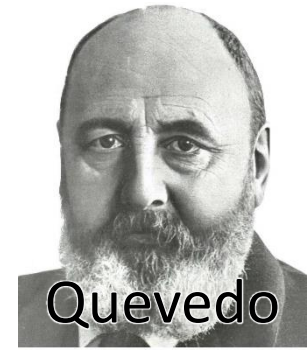
# More on Floating Point History

## ❖ 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



# Floating Point in the “Wild”

- ❖ 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



# 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** test floating point values for equality!
- ❖ **Careful** when converting between ints and floats!

# 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

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

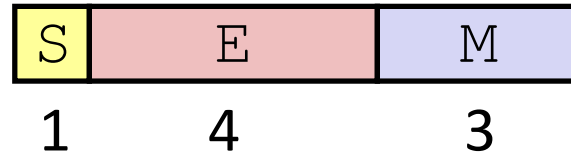
- ❖ Floating point encoding has many limitations
  - Overflow, underflow, rounding
  - Rounding is a HUGE issue due to limited mantissa bits and gaps that are scaled by the value of the exponent
  - Floating point arithmetic is NOT associative or distributive
- ❖ Converting between integral and floating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

# BONUS SLIDES

An example that applies the IEEE Floating Point concepts to a smaller (8-bit) representation scheme.

These slides expand on material covered today, so while you don't need to read these, the information is "fair game."

# 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, denormalized numbers, NaN,  $\infty$

# Dynamic Range (Positive Only)

	S	E	M	Exp	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	closest to zero
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denorm
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
0	1110	111	7	$15/8 * 128 = 240$	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