

Floating Point

CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael

Connie Chen

Chloe Fong

Chendur Jayavelu

Joshua Tan

Nikolas McNamee

Nahush Shrivatsa

Naama Amiel

Neela Kausik

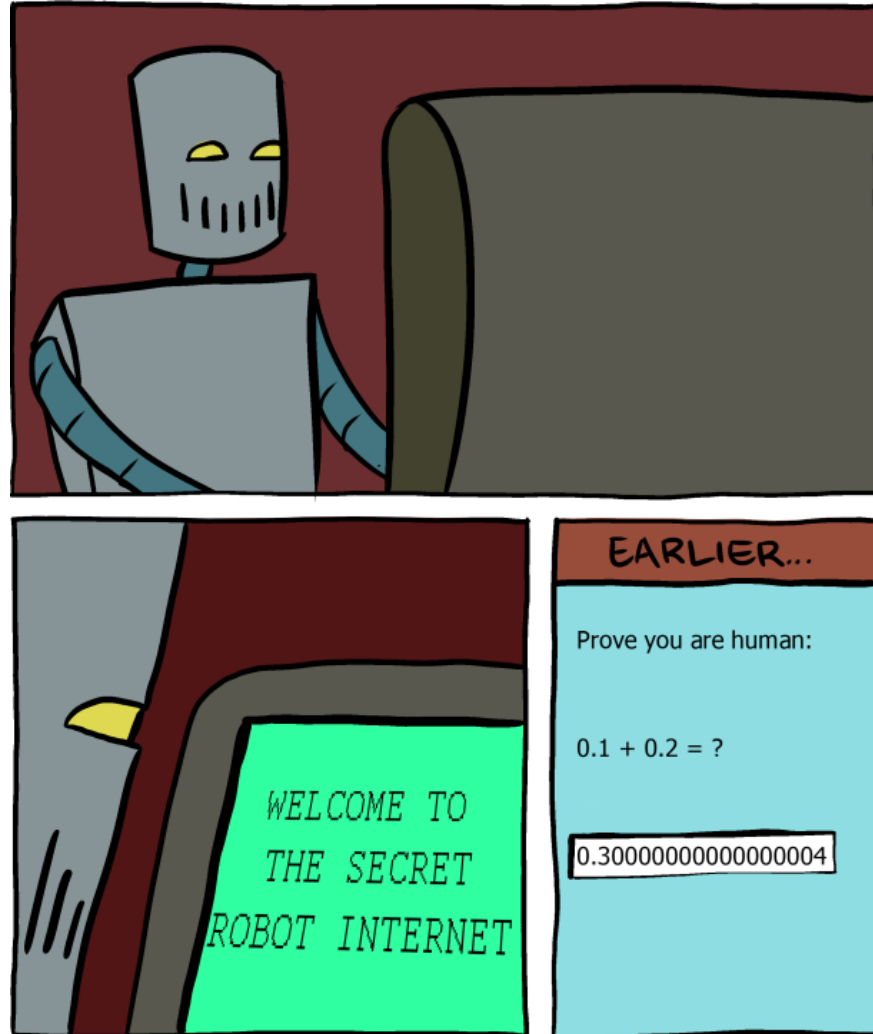
Renee Ruan

Rubee Zhao

Samantha Dreussi

Sean Siddens

Waleed Yagoub



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

Relevant Course Information

- ❖ HW4 due Monday (10/07) @ 11:59 pm
- ❖ Lab 1a due Tuesday (10/08) @ 11:59pm
 - Submit `pointer.c` and `lab1Asynthesis.txt`
 - Make sure you submit *something* to Gradescope before the deadline and that the file names are correct
 - Can use late day tokens to submit up until Thurs 11:59 pm
- ❖ HW5 due Wednesday (10/09) @ 11:59 pm
- ❖ Lab 1b, due Monday (10/14) @ 11:59pm
 - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`

Lab 1b Aside: C Macros

❖ C macros basics:

- 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
- 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 Lecture 4 slides (deck of cards operations) for examples

Reading Review

❖ Terminology:

- normalized scientific binary notation
- trailing zeros
- sign, mantissa, exponent $\leftrightarrow$ bit fields S, M, and E
- `float`, `double`
- biased notation (exponent), implicit leading one (mantissa)
- Special values
- Overflow, underflow, rounding errors

$$\begin{aligned}2^{-1} &= 0.5 \\2^{-2} &= 0.25 \\2^{-3} &= 0.125 \\2^{-4} &= 0.0625\end{aligned}$$

Review Questions

- ❖ Convert 11.375_{10} to normalized binary scientific notation
- ❖ What is the correct value encoded by the following floating point number?

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

- $\text{bias} = 2^{w-1} - 1$
- $\text{exponent} = E - \text{bias}$
- $\text{mantissa} = 1.M$

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.*, 6.02×10^{23})
- Very small numbers (*e.g.*, 6.626×10^{-34})
- Special numbers (*e.g.*, ∞ , NaN)



**Floating
Point**

Floating Point Topics

- ❖ Fractional binary numbers
- ❖ IEEE floating-point standard
- ❖ Floating-point operations and rounding
- ❖ Floating-point in C

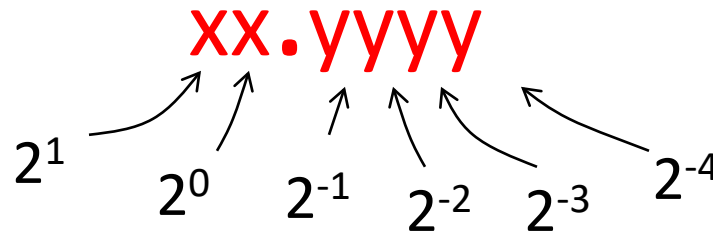


- ❖ There are many more details that we won't cover
 - It's a 58-page standard...

Representation of Fractions

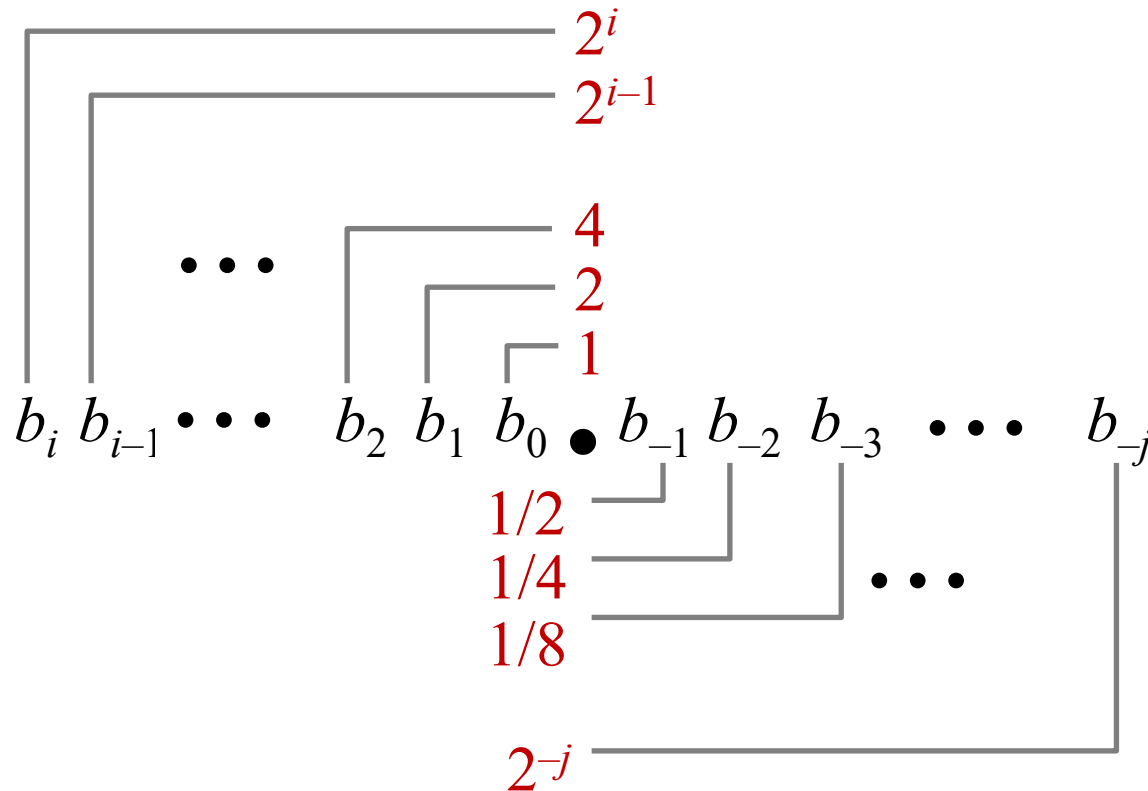
- ❖ **Binary Point**, like decimal point, signifies boundary between integer and fractional parts:

Example 6-bit
representation:



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

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 3/4 101.11_2
 - 2 and 7/8 10.111_2
 - 47/64 0.101111_2
- ❖ Observations
 - Shift left = multiply by power of 2
 - Shift right = divide by power of 2

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:

- $1/3 = 0.333333..._{10} = 0.01010101[01]..._2$
- $1/5 = 0.001100110011[0011]..._2$
- $1/10 = 0.0001100110011[0011]..._2$

Fixed Point Representation

- ❖ Implied binary point. Two example schemes:

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

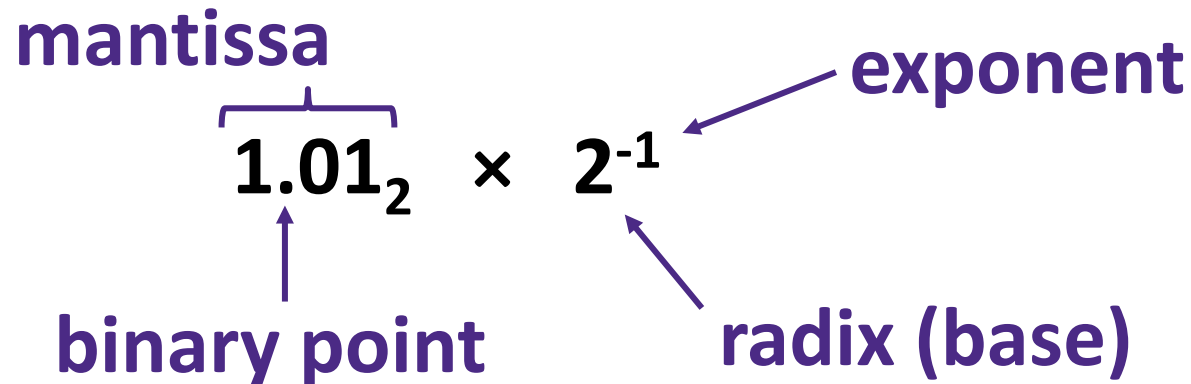
$b_7 b_6 b_5 b_4 b_3 \text{ [.] } b_2 b_1 b_0$

#2: the binary point is between bits 4 and 5

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

- ❖ Which scheme is best?

Binary Scientific Notation (Review)



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)".

- ❖ *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**
 - 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 (Review)

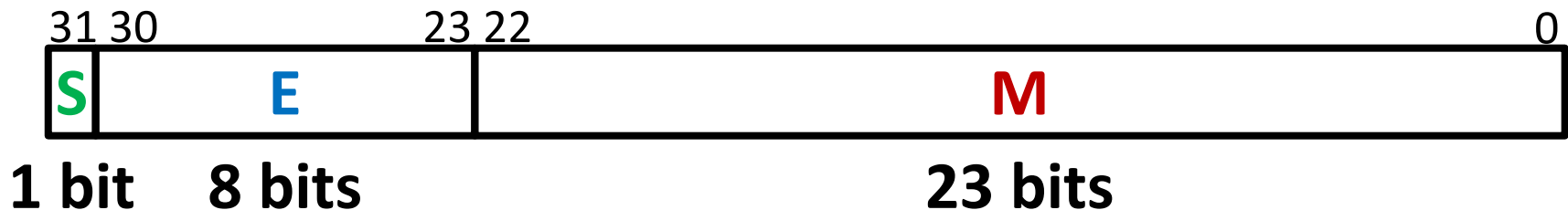
❖ Use normalized, base 2 scientific notation:

■ Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

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

❖ 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**

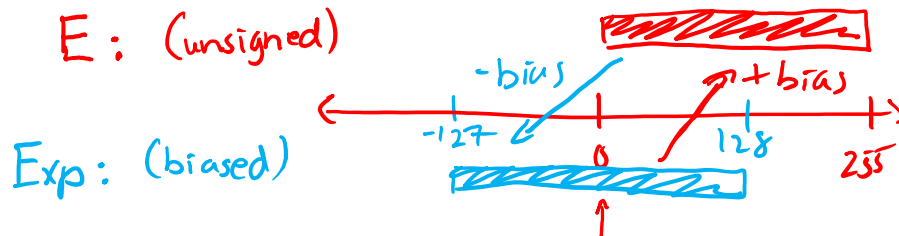


The Exponent Field (Review)

❖ Use **biased notation**

$w=8$, can encode $2^8=256$ exponents

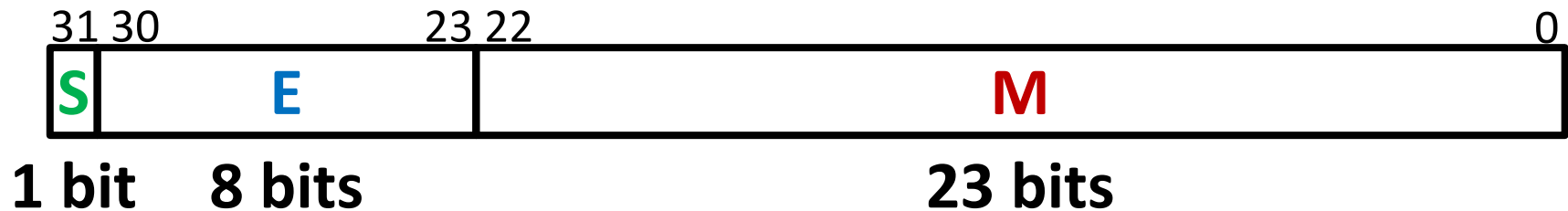
- Read exponent as unsigned, but with **bias** of $2^{w-1}-1 = 127$
- Representable exponents roughly $\frac{1}{2}$ positive and $\frac{1}{2}$ negative
- $\text{Exp} = E - \text{bias} \leftrightarrow E = \text{Exp} + \text{bias}$
 - Exponent 0 ($\text{Exp} = 0$) is represented as $E = 0b\ 0111\ 1111 = 2^7 - 1$



❖ Why biased?

- Makes floating point arithmetic easier
- Makes somewhat compatible with two's complement hardware

The Mantissa (Fraction) Field (Review)



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

- ❖ 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* $0.1_2 = 0.5_{10}$
 - Gives us an extra bit of *precision*
- ❖ Mantissa “limits”
 - Low values near $M = 0b0\dots0$ are close to 2^{Exp}
 - High values near $M = 0b1\dots1$ are close to $2^{\text{Exp}+1}$

Normalized Floating Point Conversions

❖ FP → 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 → 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.

Practice Question

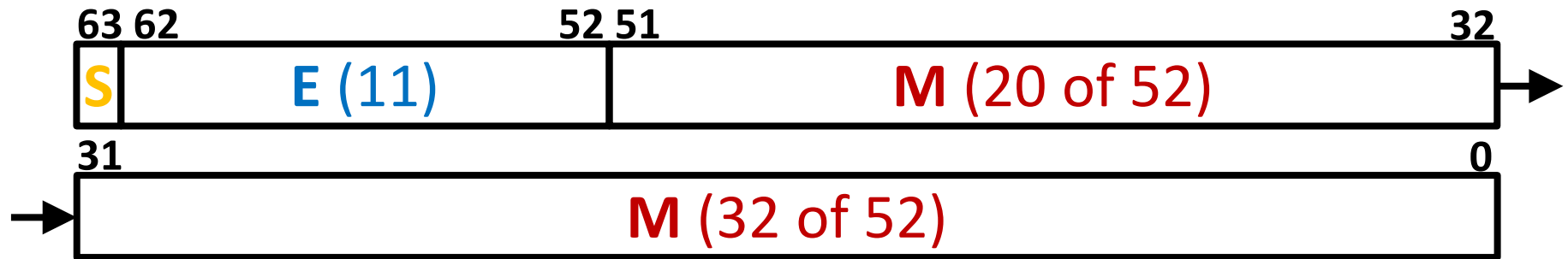
- ❖ Convert the decimal number
 $-11.375 = -1.011011 * 2^3$
into floating point representation

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

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
- ❖ $E = 0xFF$, $M = 0$: $\pm \infty$
 - *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

Floating Point Encoding Summary

	E	M	Meaning
smallest E (all 0's)	0x00	0	± 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

New Representation Limits

❖ New largest value (besides ∞)?

- $E = 0xFF$ has now been taken!
- $E = 0xFE$ has largest: $1.1...1_2 \times 2^{127} = 2^{128} - 2^{104}$

❖ New numbers closest to 0:

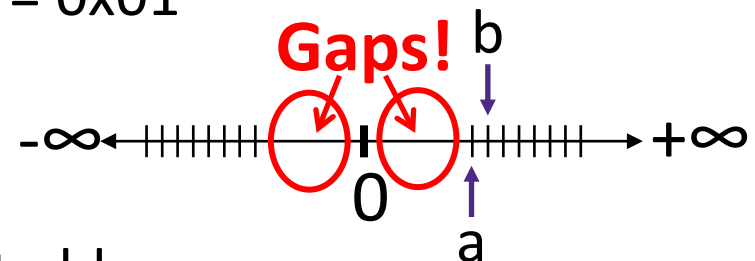
- $E = 0x00$ taken; next smallest is $E = 0x01$

- $a = 1.0...00_2 \times 2^{-126} = 2^{-126}$

- $b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$

- Normalization and implicit 1 are to blame

- *Special case:* $E = 0$, $M \neq 0$ 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: $\pm 1.0\dots00_{\text{two}} \times 2^{-126} = \pm 2^{-126}$

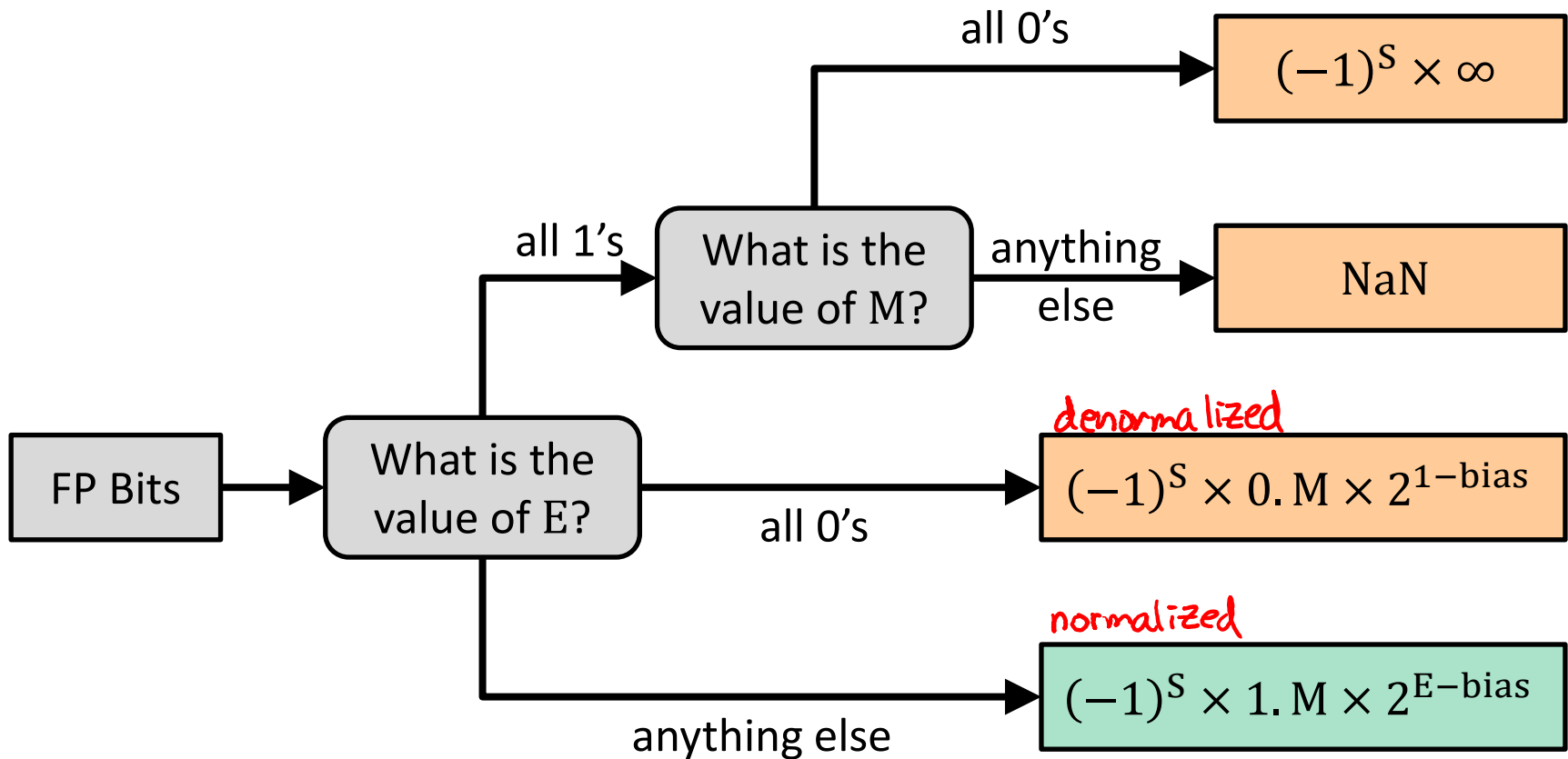
- Smallest denorm: $\pm 0.0\dots01_{\text{two}} \times 2^{-126} = \pm 2^{-149}$

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

So much
closer to 0



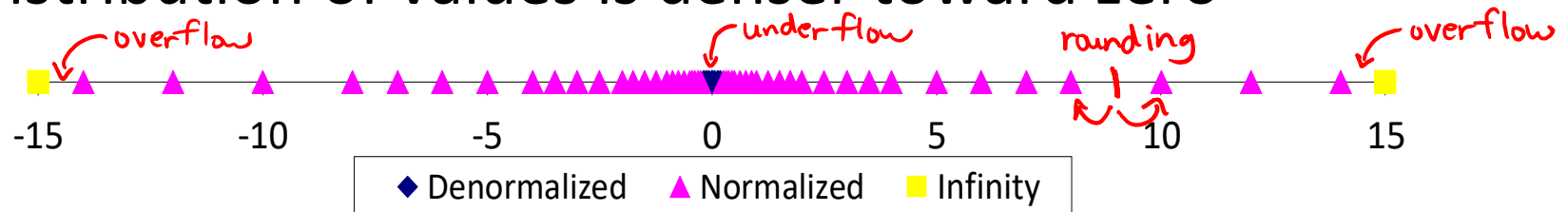
Floating Point Interpretation Flow Chart



= special case

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 Topics

- ❖ Fractional binary numbers
- ❖ IEEE floating-point standard
- ❖ Floating-point operations and rounding
- ❖ Floating-point in C



Floating Point Operations: Basic Idea

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



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

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 Conversions in C

❖ Casting between `int`, `float`, and `double` **changes** the bit representation

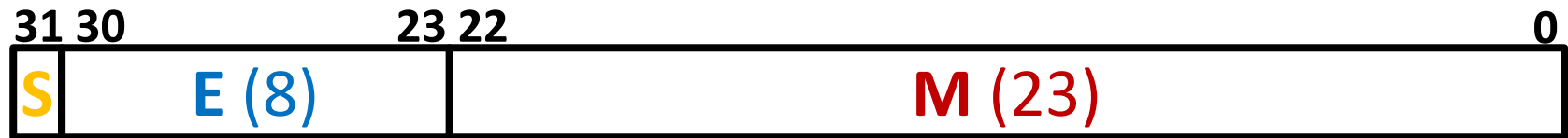
- `int` $\rightarrow$ `float`
 - May be rounded (not enough bits in mantissa: 23)
 - Overflow impossible
- `int` or `float` $\rightarrow$ `double`
 - Exact conversion (all 32-bit `ints` representable)
- `long` $\rightarrow$ `double`
 - Depends on word size (32-bit is exact, 64-bit may be rounded)
- `double` or `float` $\rightarrow$ `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)

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)

Floating Point Representation Summary

❖ Floating point approximates real numbers:



- Handles large numbers, small numbers, special numbers
- Exponent in biased notation ($\text{bias} = 2^{w-1} - 1$)
 - 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*

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`!

Summary

E	M	Meaning
0x00	0	± 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