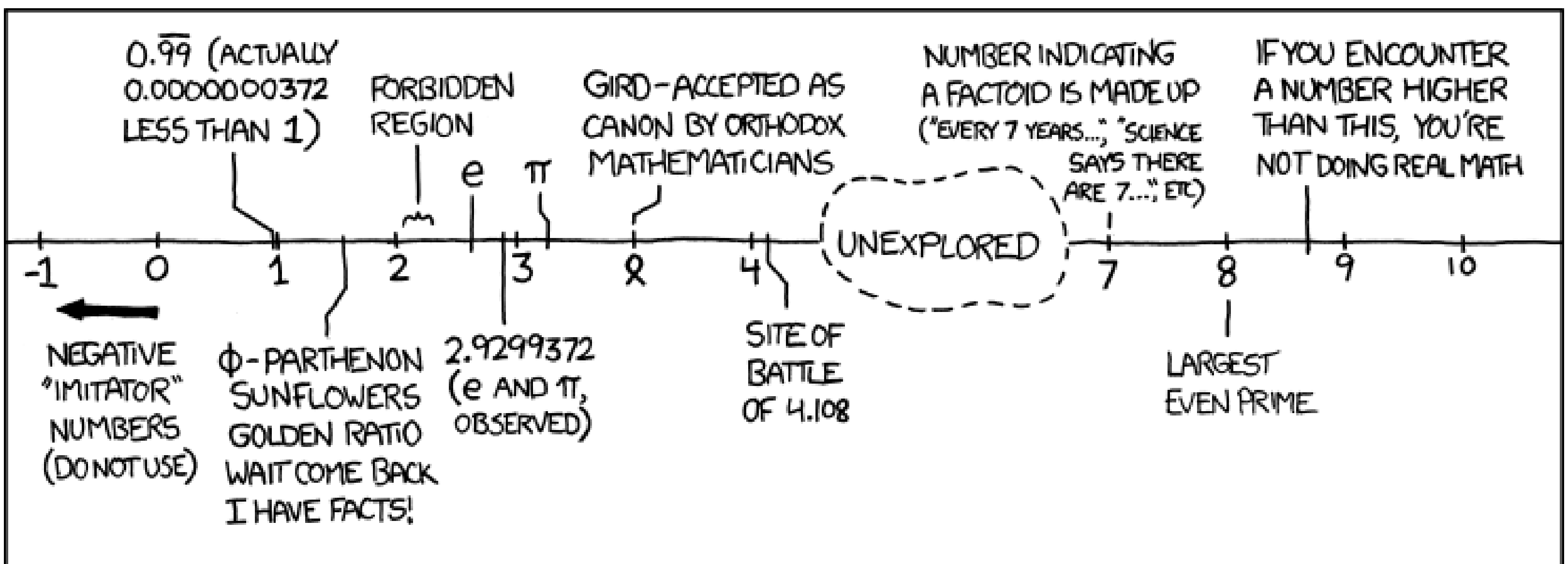
Floating Point

CSE 351 Summer 2022

Instructor:
Kyrie Dowling

Teaching Assistants:
Aakash Srazali Allie Pfleger Ellis Haker



http://xkcd.com/899/

Relevant Course Information

- hw4 due tonight, hw5 due Friday (7/8)

- Lab 1a due tonight @ 11:59 pm
  - Submit `pointer.c` and `lab1Asynthesis.txt`
    - Make sure there are no lingering `printf` statements in your code!
  - 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 Fri 11:59 pm

- Lab 1b due Monday (7/11)
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`

Unit Portfolios

- First unit portfolio is live! Two parts
  - Problem video walk through
  - Reflection summary
- Information can be found on the course website: https://courses.cs.washington.edu/courses/cse351/22su/unit_portfolios/
- Due next Friday, July 15 at 11:59 pm
- Individual, no late days allowed
- Intended to be a low-key reflective assignment
  - ESNU grading aka good-faith effort will get you full credit

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 (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

- Questions from the Reading?

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: xx.yyyy

2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- Example: $10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}$

Representation of Fractions

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

Example 6-bit representation: **xx.yyyy**

2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- In this 6-bit representation:
  - What is the encoding and value of the smallest (most negative) number?
  - What is the encoding and value of the largest (most positive) number?
  - What is the smallest number greater than 2 that we can represent?

Binary Scientific Notation (Review)

mantissa

exponent

$$1.01_2 \times 2^{-1}$$

binary point

radix (base)

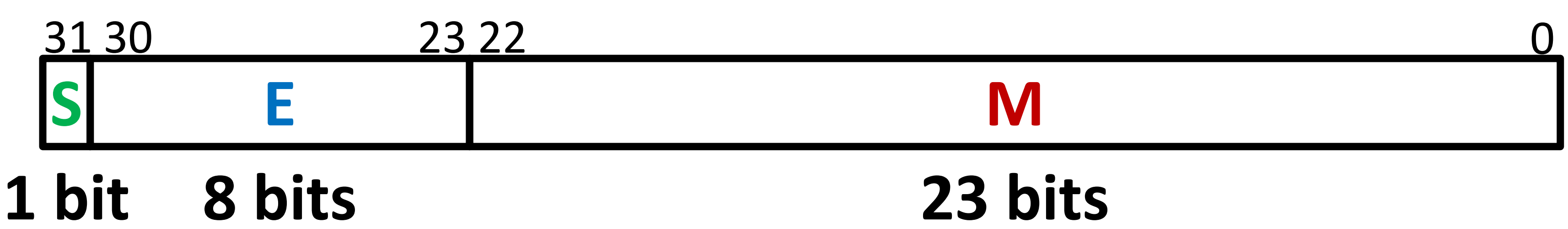
- *Normalized form*: exactly one digit (non-zero) to left of binary point
- Computer arithmetic that supports this called floating point due to the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)

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

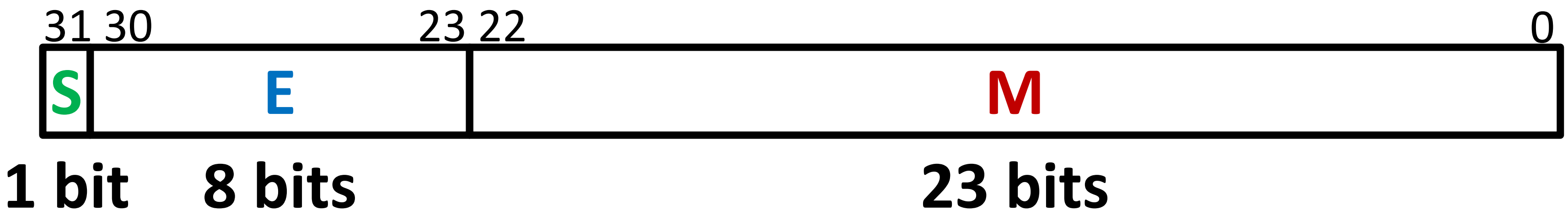


The Exponent Field (Review)

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

- 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 leading 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…0 are close to 2^{Exp}
  - High values near M = 0b1…1 are close to 2^{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 $2^{E - bias}$.
  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.

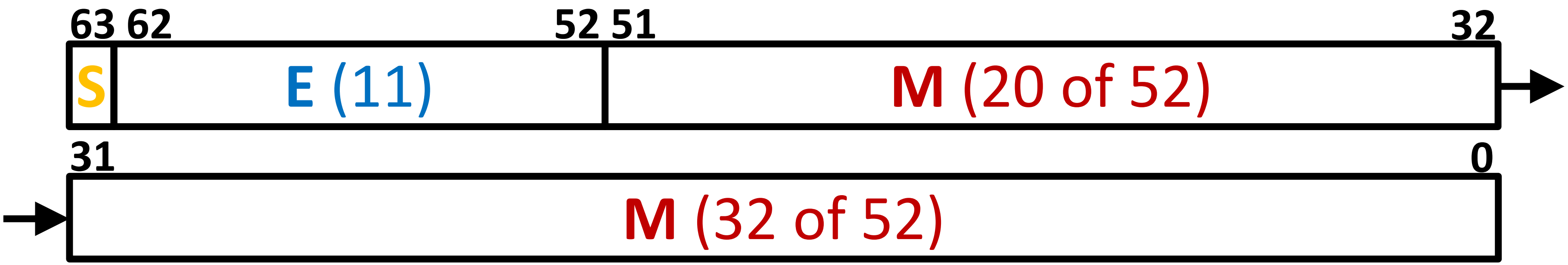
Precision and Accuracy

- 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

Floating Point Topics

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

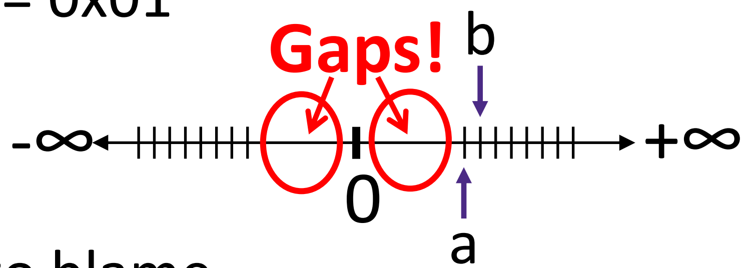
- There are many more details that we won't cover
  - It's a 58-page standard…

Special Cases

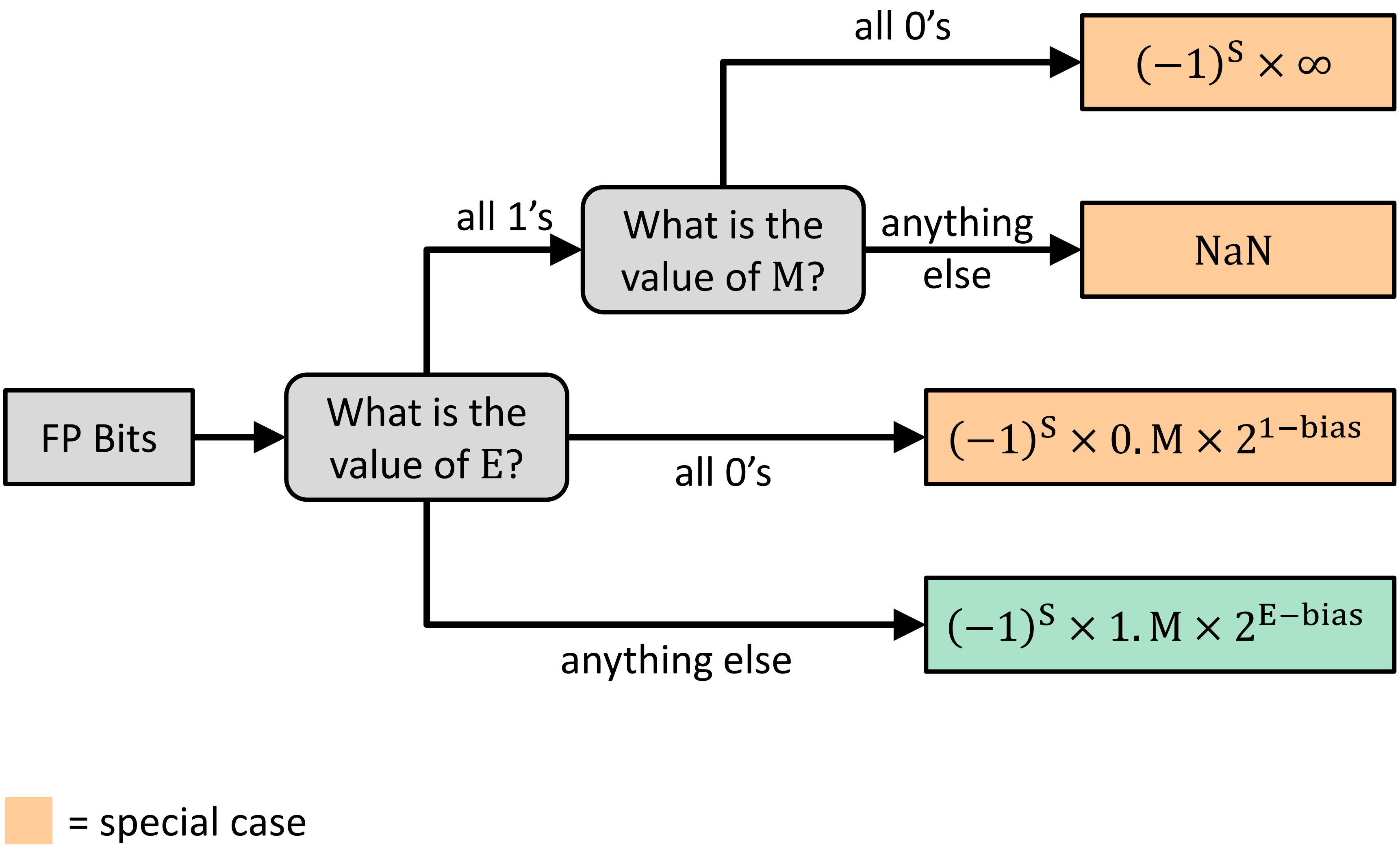
- But wait… what happened to zero?
  - *Special case:* E and M all zeros = 0
  - Two zeros! But at least 0x00000000 = 0 like integers
- E = 0xFF, M = 0: ± ∞
  - *e.g.*, division by 0
  - Still work in comparisons!
- E = 0xFF, M ≠ 0: Not a Number (NaN)
  - *e.g.*, square root of negative number, 0/0, ∞–∞
  - NaN propagates through computations
  - Value of M can be useful in debugging

New Representation Limits

- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: $1.1\ldots1_2 \times 2^{127} = 2^{128} - 2^{104}$
- New numbers closest to 0:
  - E = 0x00 taken; next smallest is E = 0x01
  - $a = 1.0\ldots00_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0\ldots01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - *Special case:* E = 0, M ≠ 0 are denormalized numbers
    - Mantissa has implicit 0 instead of implicit 1
    - Store much smaller numbers

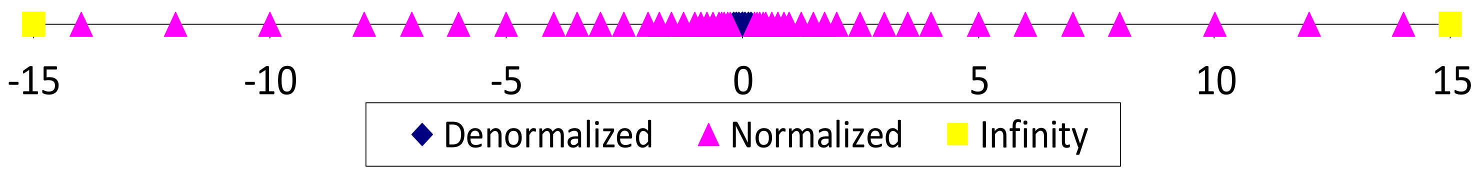


Floating Point Decoding Flow Chart



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?
  - What is this "step" when Exp = 100?
- Distribution of values is denser toward zero



Floating Point Operations: Basic Idea

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

S	E	M

- $x +_f y = \text{Round}(x + y)$
- $x *_f y = \text{Round}(x * y)$

- 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 != 3.14+(1e100-1e100)`
    - **0** / **3.14**
  - Not distributive: `100*(0.1+0.2) != 100*0.1+100*0.2`
    - **30.000000000000003553** / **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!

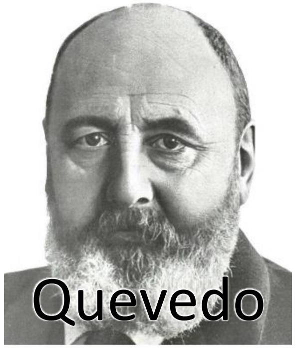
Floating Point Conversions in C

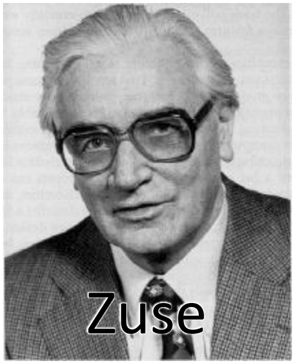
!!!

- 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



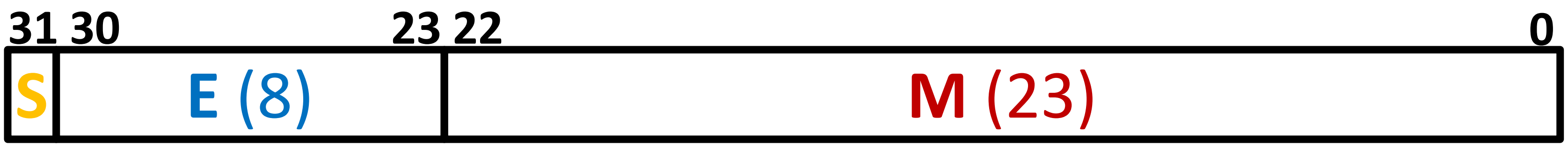


Number Representation Really Matters

- **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:



  - 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 `int`s
    - 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 `int`s and `float`s!

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

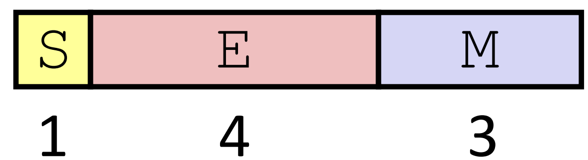
BONUS SLIDES

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

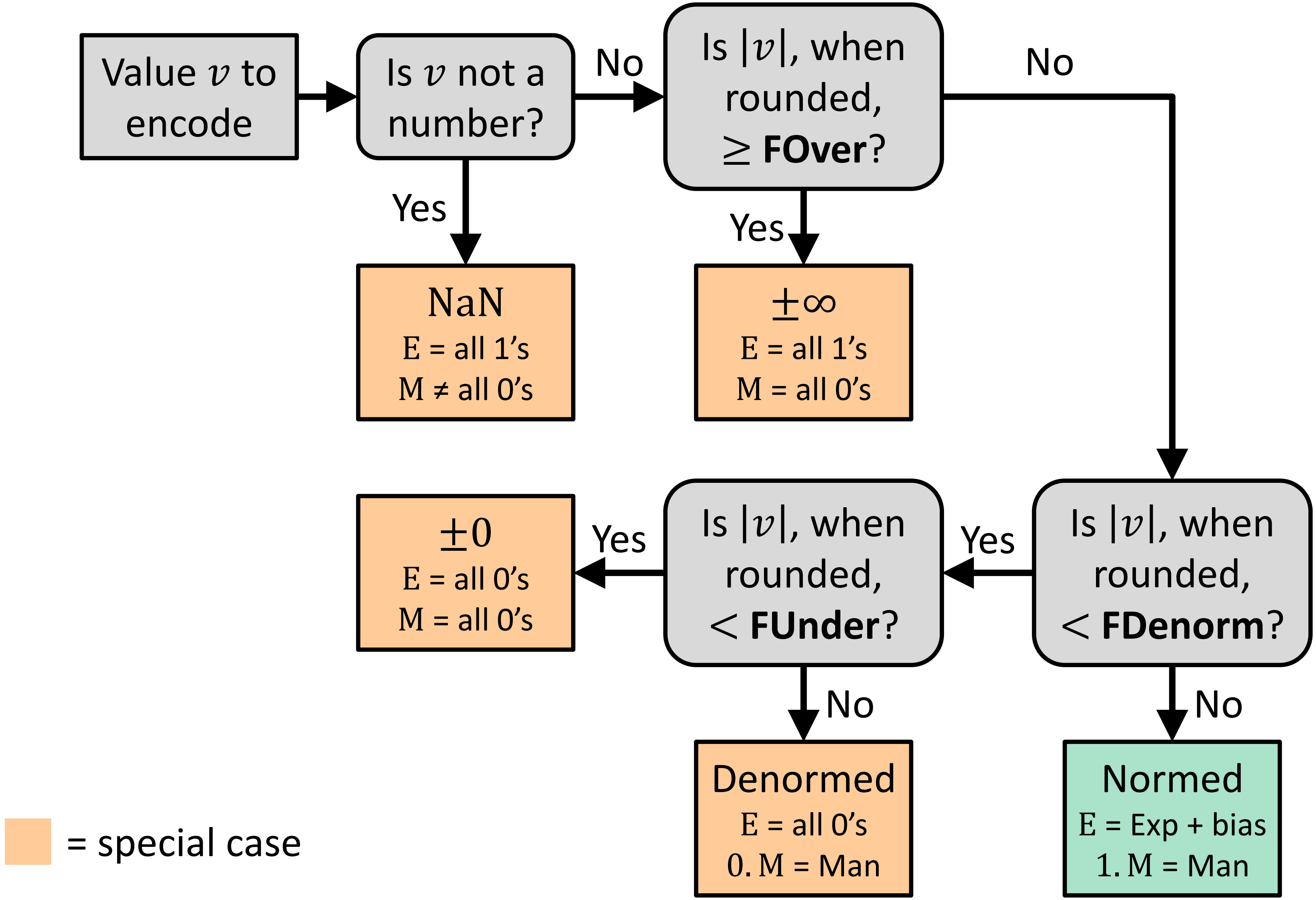
- The IEEE 754 standard actually specifies different rounding modes:
  - Round to nearest, ties to nearest even digit
  - Round toward $+\infty$ (round up)
  - Round toward $-\infty$ (round down)
  - Round toward 0 (truncation)



- In our tiny example:
  - Man = 1.001 01 rounded to M = 0b001
  - Man = 1.001 11 rounded to M = 0b010
  - Man = 1.001 10 rounded to M = 0b010
  - Man = 1.000 10 rounded to M = 0b000

Floating Point Encoding Flow Chart

This is extra (non-testable) material



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

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: $\pm\, 1.0\ldots0_{two}\times2^{-126} = \pm\, 2^{-126}$
  - Smallest denorm: $\pm\, 0.0\ldots01_{two}\times2^{-126} = \pm\, 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number

So much closer to 0

Floating Point in the "Wild"

- 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