Floating Point II

CSE 351 Summer 2020

Instructor:

Porter Jones

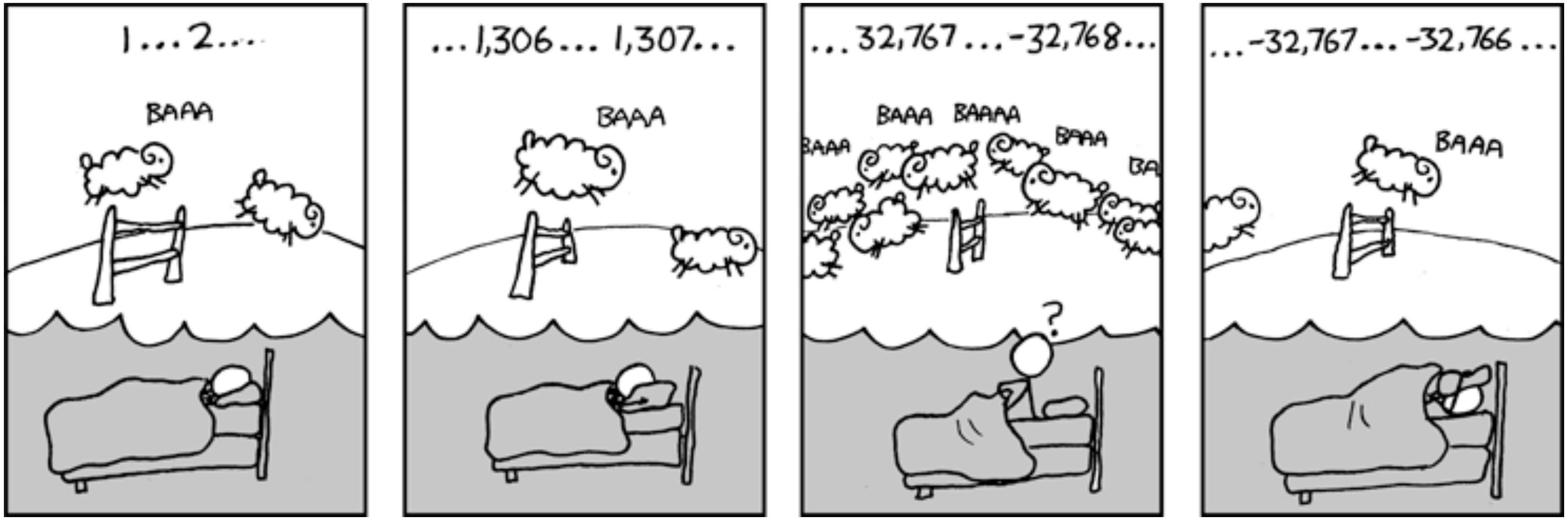
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



http://xkcd.com/571/

Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-6
- hw6 & hw7 due Friday (7/10) – 10:30am
- Lab 1a due tonight at 11:59 pm!!!
  - Submit `pointer.c` and `lab1Areflect.txt`
- Lab 1b due Friday (7/10)
  - Submit `aisle_manager.c`, `store_client.c` and `lab1Breflect.txt`

Fixed Point Representation

- Implied binary point. Two example schemes:

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

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

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

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

- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have
- Fixed point = fixed *range* and fixed *precision*
  - 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 1.2×10^{7} In C: 1.2e7
    - Not 0.0000012, but 1.2×10^{-6} In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but 1.1×2^{4}
    - Not 0.000101, but 1.01×2^{-4}
- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the mantissa (significand)
  - the exponent

Scientific Notation (Decimal)

$$\overbrace{6.02}^{\text{mantissa}}{}_{10} \times 10^{23}$$

exponent

decimal point

radix (base)

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

Scientific Notation (Binary)

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

exponent

binary point

radix (base)

- 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`)

Scientific Notation Translation

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

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$

- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$

- **Practice:** Convert 11.375_{10} to normalized binary scientific notation

1) convert to binary point 11.375

8+2+1 .25+.125

1011 . 011

2) normalize 1.011011 x 2^3

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…

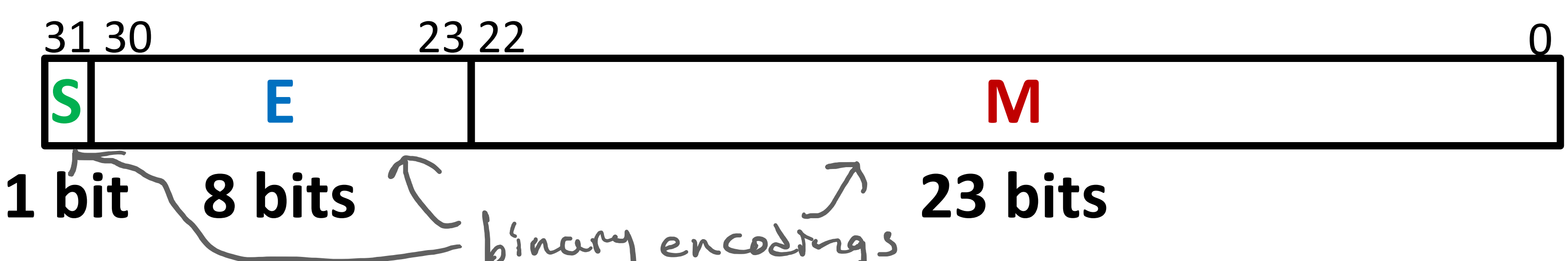
IEEE Floating Point

- IEEE 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating 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**
  - In the end:
    - 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

- 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

$w=8$ $\quad 2^{w-1}-1 = 2^7-1$
$= 128-1$
$= \boxed{127}$

- Use biased notation
  - Read exponent as unsigned, but with *bias* of $2^{w-1}-1 = 127$
  - Representable exponents roughly ½ positive and ½ negative
  - 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
- **Practice:** To encode in biased notation, add the bias then encode in unsigned:
  - Exp = 1 → 128 → E = 0b 1000 0000
  - Exp = 127 → 254 → E = 0b 1111 1110
  - Exp = -63 → 64 → E = 0b 0100 0000
    +127

The Mantissa (Fraction) Field

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 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*

1.1_2

$1 + .5 = 1.5_{10}$

- 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}

Polling Question [FP I – a]

- What is the correct value encoded by the following floating point number?
  - 0b $\overset{S}{\underset{+}{0}}$ $\overset{E}{\underbrace{10000000}_{128}}$ $\overset{M}{11000000000000000000000}$
  - Vote at http://pollev.com/pbjones

Exp = E bits − bias = 128 − 127 = 1

M = 1.1100...00

$+1.11_2 * 2^1$

$= 11.1_2$

$2 + 1 + .5 = 3.5_{10}$

A. + 0.75

B. + 1.5

C. + 2.75

D. + 3.5

E. We're lost…

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.

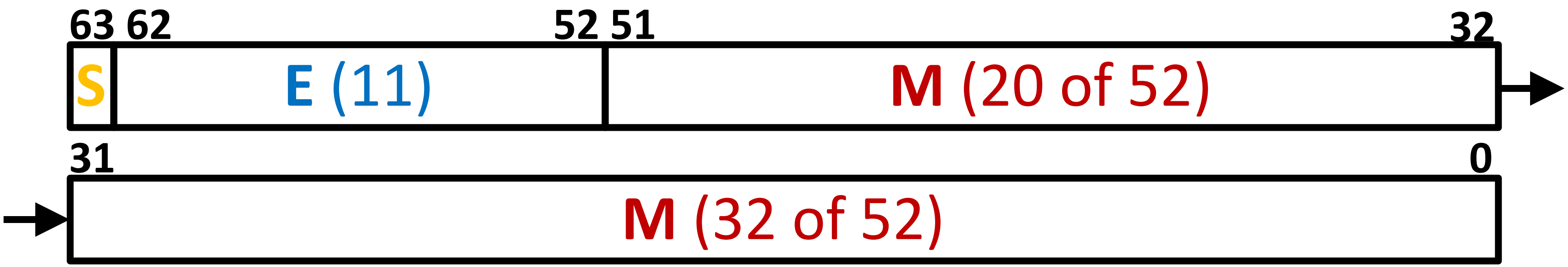
- Decimal → FP
  1. Convert decimal to binary.
  2. Convert binary to normalized scientific notation.
  3. Encode sign as `S` (0/1).
  4. Add the bias to exponent and encode `E` as unsigned.
  5. The first bits after the leading 1 that fit are encoded into `M`.

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?

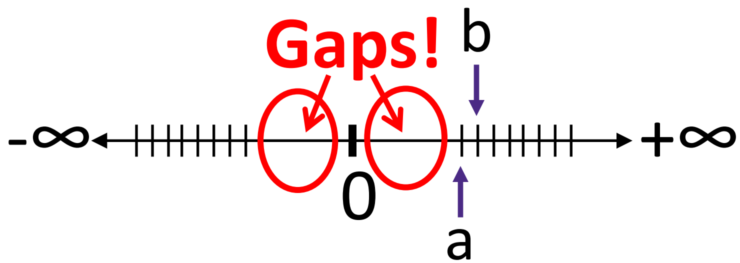
- Double Precision (vs. Single Precision) in 64 bits



  - C variable declared as `double`
  - Exponent bias is now $2^{10}-1 = 1023$ $2^{w-1}-1$
  - **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? Exp = −127 Man = 1.00…0
  - S=0 E=0 M=0 ⟹
  - Using standard encoding 0x00000000 = $1.0 \times 2^{-127} \neq 0$
  - *Special case:* E and M all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers
    - 0x80000000 = −0
- New numbers closest to 0:
  - a = $1.0…0_2 \times 2^{-126} = 2^{-126}$ 23
  - b = $1.0…01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - *Special case:* E = 0, M ≠ 0 are denormalized numbers 0.M
    - no implicit 1



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

Other Special Cases

all ones

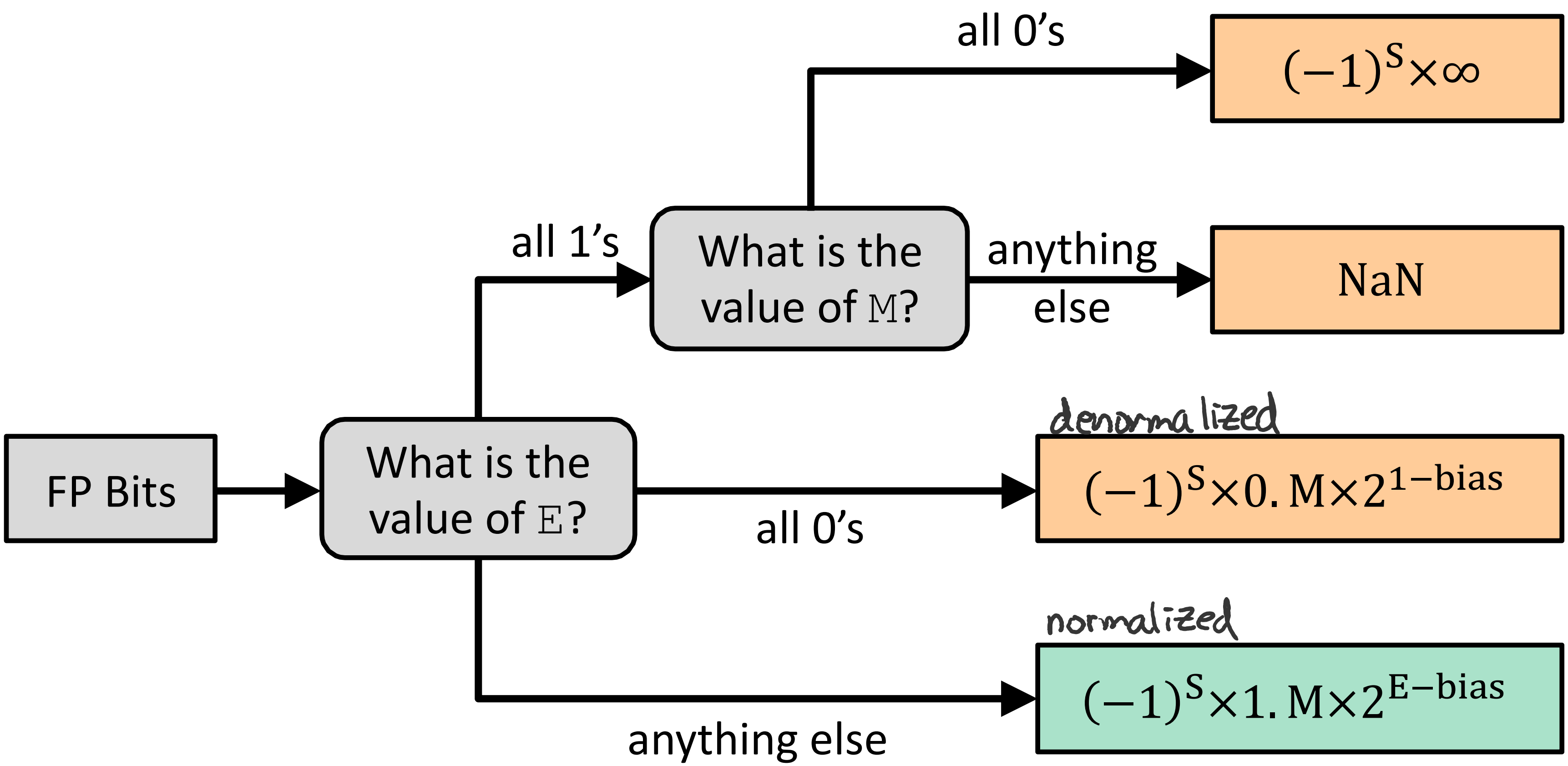
- E = 0xFF, M = 0: $\pm\infty$
  - *e.g.* division by 0
  - Still work in comparisons!
- E = 0xFF, M ≠ 0: Not a Number (NaN)
  - *e.g.* square root of negative number, 0/0, $\infty - \infty$
  - NaN propagates through computations
  - Value of M can be useful in debugging (tells you cause of NaN)
- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: $1.\underbrace{1\ldots1}^{23\text{ ones}}{}_2 \times 2^{127} = 2^{128} - 2^{104}$

    254 - bias

Floating Point Encoding Summary

	E	M	Meaning
smallest E (all 0's)	0x00	0	± 0
	0x00	non-zero	± denorm num
everything else	0x01 – 0xFE	anything	± norm num
largest E (all 1's)	0xFF	0	± ∞
	0xFF	non-zero	NaN

Floating Point Interpretation Flow Chart



= special case

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…

Tiny Floating Point Representation

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

S	E	M
1	4	3

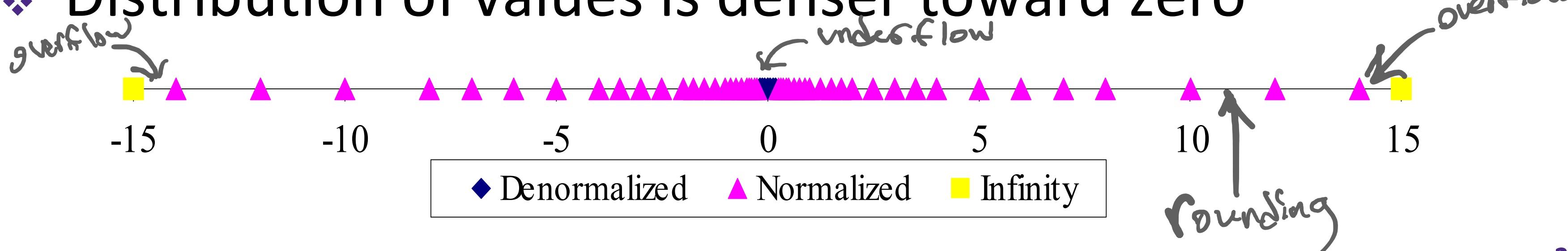
- Assume that it has the same properties as IEEE floating point:
  - bias = $2^{w-1}-1 = 2^{4-1}-1 = 8-1 = 7$
  - encoding of -0 = 0b 1 0000 000 = 0x80
  - encoding of $+\infty$ = 0b 0 1111 000 = 0x78
  - encoding of the largest (+) normalized # = 0b 0 1110 111 = 0x77
  - encoding of the smallest (+) normalized # = 0b 0 0001 000 = 0x08

can't use 1111 or 0000 because of special cases

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?

  if M=0b0...00 then $2^{Exp} * 1.0$
  if M=0b0...01 then $2^{Exp} * (1+2^{-23})$
  diff = 2^{Exp-23}
  - What is this "step" when Exp = 0? 2^{-23}
  - What is this "step" when Exp = 100? 2^{77}
- 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:

S	E	M
1	4	3

  - Man = 1.001 01 rounded to M = 0b001 down
    - $<.5$
  - Man = 1.001 11 rounded to M = 0b010 up
    - $>.5$
  - Man = 1.001 10 rounded to M = 0b010 up
    - $==.5$

Man = 1.000 10 rounded to M = 0b000 down

$==.5$

nearest even digit

Floating Point Operations: Basic Idea

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

S	E	M

- $\texttt{x} +_{\texttt{f}} \texttt{y} = \texttt{Round(x + y)}$
- $\texttt{x} *_{\texttt{f}} \texttt{y} = \texttt{Round(x * y)}$

- 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 ±∞ and underflow yields 0
- Floats with value ±∞ and NaN can be used in operations
  - Result usually still ±∞ 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)`

    rounded away 10^{100} 10^{100} 10^{100} 10^{100}

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

    can't represent exactly 10 20

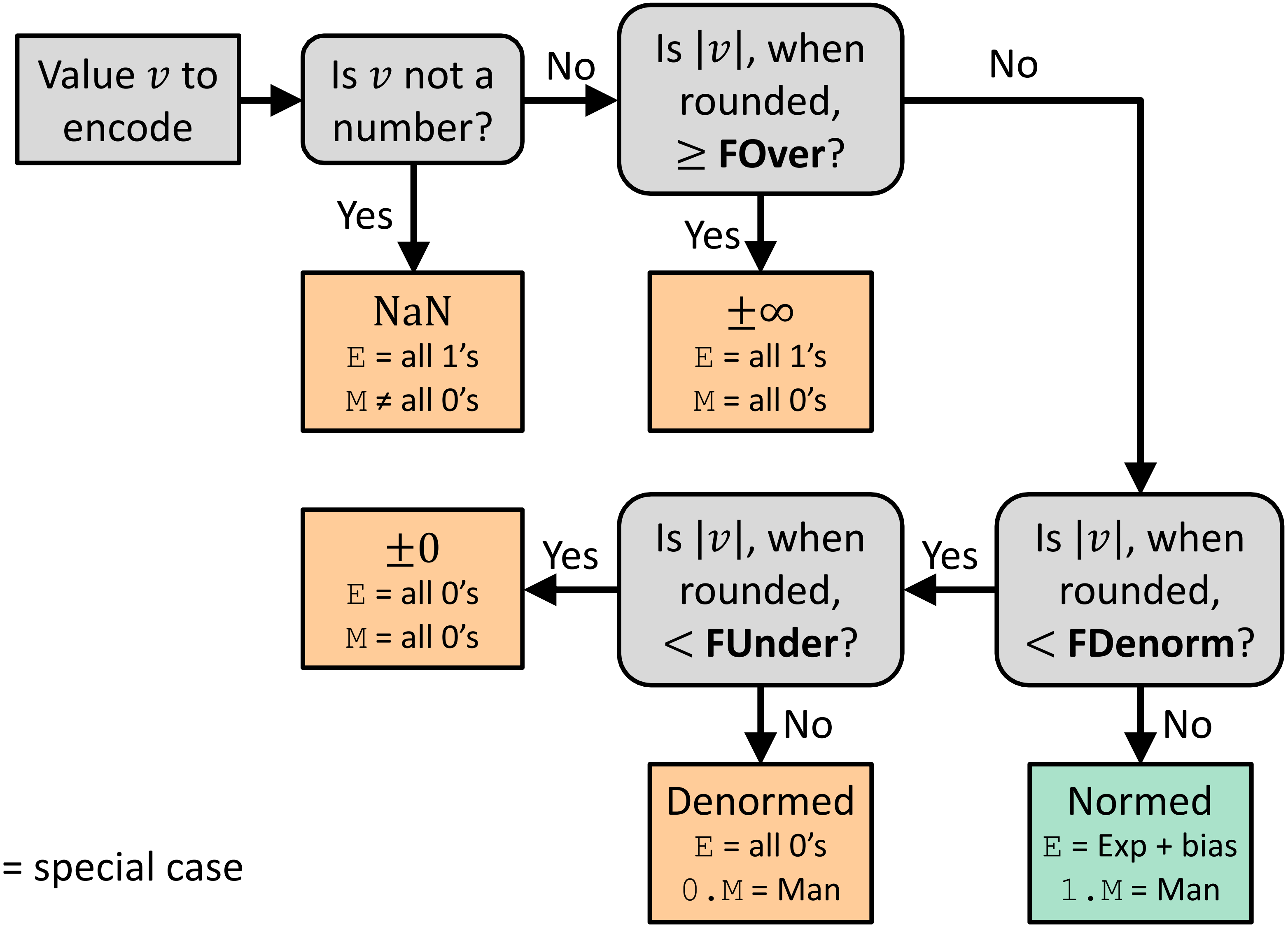
    30.000000000000003553 **30**
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

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



= special case

Example Question [FP II - a]

- Using our **8-bit** representation, what value gets stored when we try to encode **384** = $2^8 + 2^7$? $= 2^8(1+2^{-1})$

$= 2^8 * 1.1_2$

S	E	M
1	4	3

$S = 0$

$E = \text{Exp} + \text{bias}$

$= 8 + 7 = 15$

$= 0b1111$

falls outside normalized exponent range

- No voting

A. + 256

B. + 384

C. $\boxed{+\ \infty}$

D. NaN

E. We're lost…

This number is too large so we store

$+\infty \Rightarrow 0b0\ 1111\ 000$

Polling Question [FP II - b]

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

S	E	M
1	4	3

  - Vote at http://pollev.com/pbjones

A. + 2.5

B. + 2.625

C. + 2.75

D. + 3.25

E. We're lost…

$2^1(1+2^{-2}+2^{-4})$

$= 2^1 * 1.0101$

$S = 0$

$E = Exp + bias$

$= 1 + 7 = 8$

$= 0b1000$

$M = 0b\,010\cancel{1}$

can only store 3 bits

Stored as 0b0 1000 010 = 2.5

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

S	E	M
1	4	3

- 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, ∞

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