Floating Point

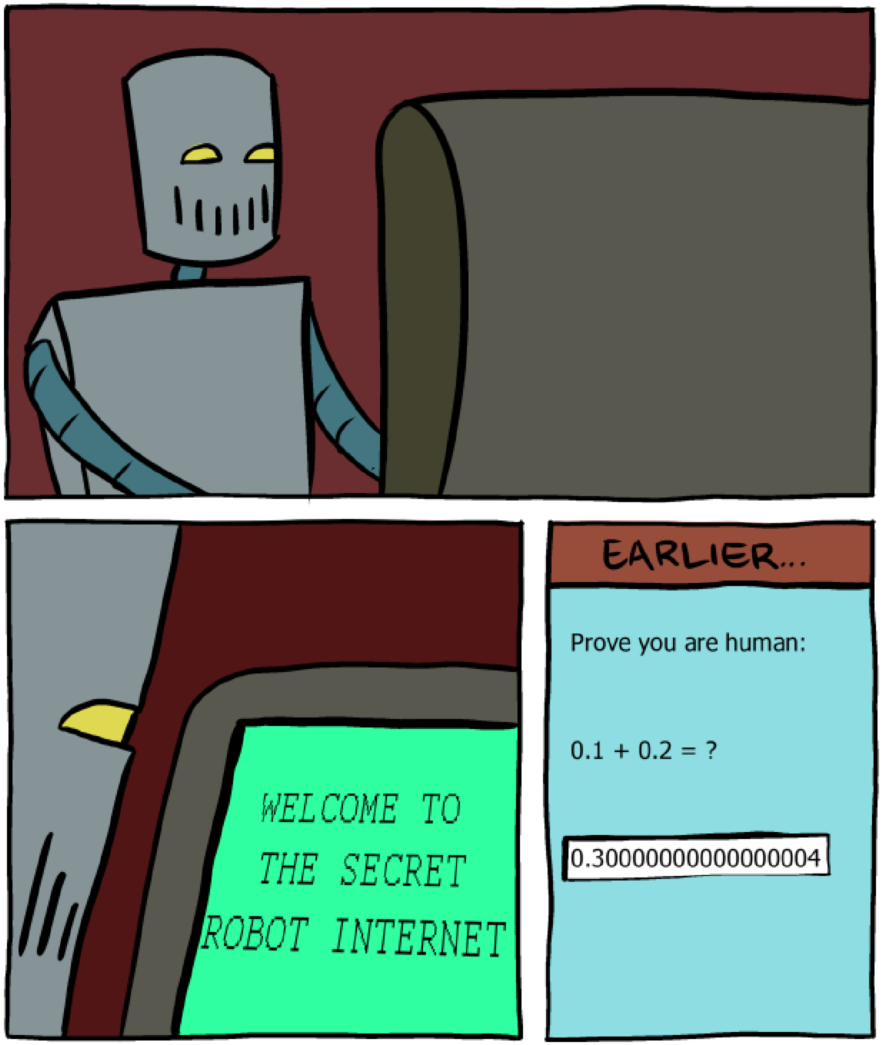
CSE 351 Autumn 2024

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 ↔ bit fields S, M, and E
  - `float`, `double`
  - biased notation (exponent), implicit leading one (mantissa)
  - Special values
  - Overflow, underflow, rounding errors

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

Review Questions

- Convert $\underbrace{11}_{8+2+1}.\underbrace{375}_{0.25+0.125}{}_{10}$ to normalized binary scientific notation

  $2^3 + 2^1 + 2^0 + 2^{-2} + 2^{-3}$

  $1011.011_2 \Rightarrow 1.011011 * 2^3$

- What is the correct value encoded by the following floating point number?

  $\textbf{0b } \underbrace{\textbf{0}}_{S} \mid \underbrace{\textbf{1000 0000}}_{E} \mid \underbrace{\textbf{110 0000 0000 0000 0000 0000}}_{M}$

  - bias = $2^{w-1}-1$
  - exponent = E – bias $= 2^7 - 127 = 128 - 127 = 1$ ← exponent
  - mantissa = 1.M

positive #

$(-1)^{0} * 1.1100_2 * 2^1 \Rightarrow \underbrace{11}_{3_{10}}.\underbrace{1}_{0.5_{10}}{}_2 \Rightarrow \boxed{3.5_{10}}$

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

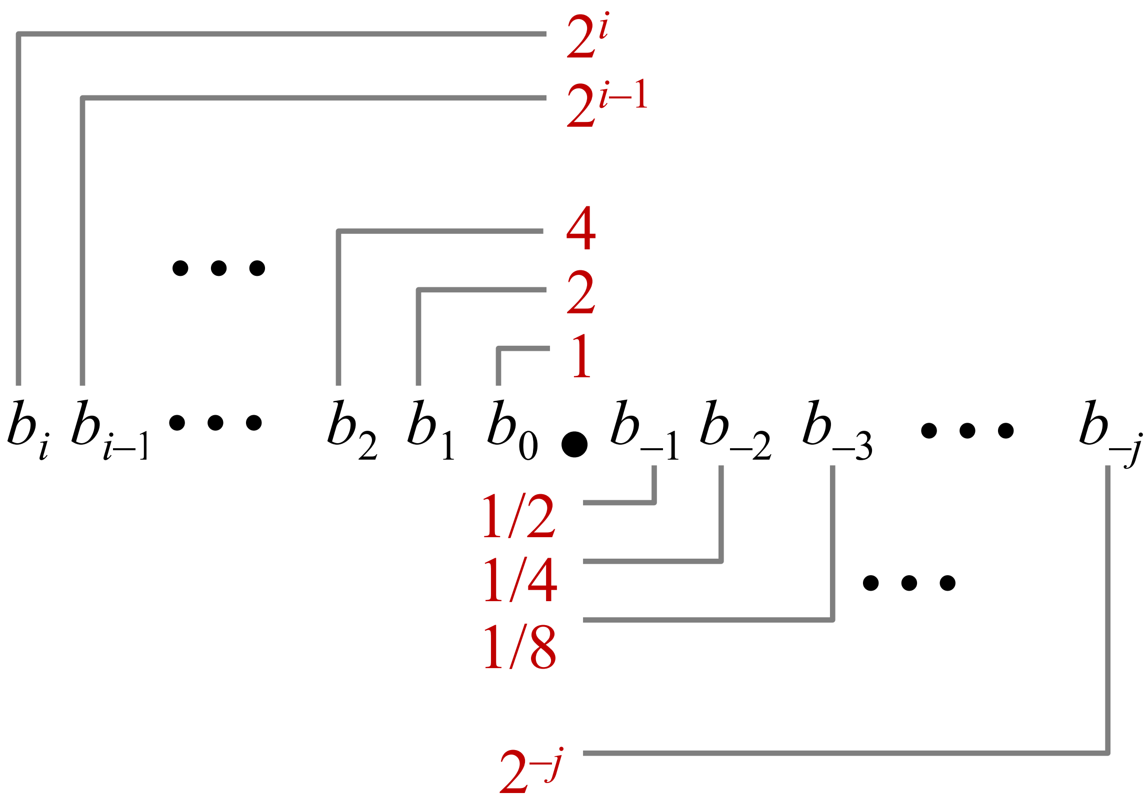
- **Binary Point**, like decimal point, signifies boundary between integer and fractional parts:

  Example 6-bit representation: `xx.yyyy`

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

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

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

Decimal		**Value:**	**Binary Representation:**
	•	1/3 = $0.333333\ldots_{10}$ =	$0.01010101[01]\ldots_2$
0.2	•	1/5 =	$0.001100110011[0011\ ]\ldots_2$
0.1	•	1/10 =	$0.0001100110011[0011\ ]\ldots_2$

Fixed Point Representation

- Implied binary point. Two example schemes:
  - #1: the binary point is between bits 2 and 3
    - $b_7\ b_6\ b_5\ b_4\ b_3$ [.] $b_2\ b_1\ b_0$
  - #2: the binary point is between bits 4 and 5
    - $b_7\ b_6\ b_5$ [.] $b_4\ b_3\ b_2\ b_1\ b_0$
- Which scheme is best?

Binary Scientific Notation (Review)

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

exponent

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**

FLOPs

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: (3 separate fields within 32 bits)
  - 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**

values

31	30 … 23	22 … 0
S	**E**	**M**
1 bit	8 bits	23 bits

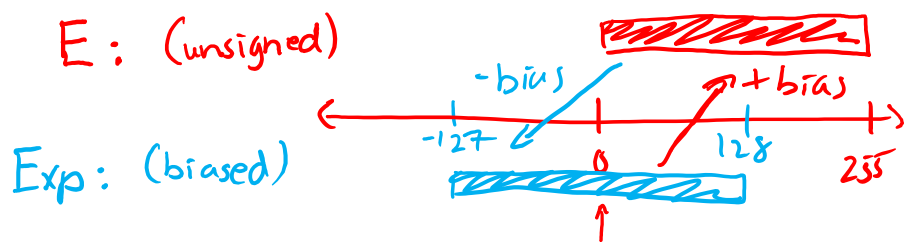
binary encodings

The Exponent Field (Review)

- Use biased notation

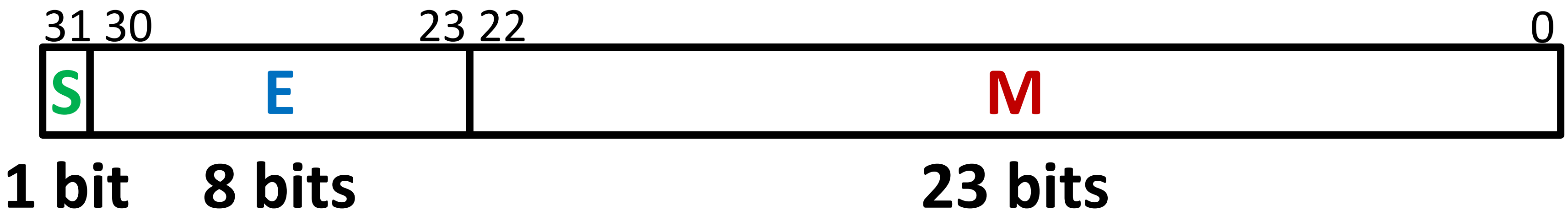
w=8, can encode $2^8=256$ exponents

  - Read exponent as unsigned, but with *bias* of $2^{w-1}-1$ = 127
  - Representable exponents roughly ½ positive and ½ negative
  - Exp = E – bias ↔ E = Exp + bias
    - Exponent 0 (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…0 are close to 2^{Exp}
  - High values near M = 0b1…1 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

$S = 1$

$E = 3_{10} + \overset{\text{bias}}{127} = 130_{10} \rightarrow 0b10000010$

$M = \underbrace{0110110\ldots0}_{\text{23 bits}}$

$0b\,\overset{S}{1}\,|\,\overset{E}{10000010}\,|\,\underbrace{\overset{M}{0110110\ldots0}}_{\text{23 bits}}$

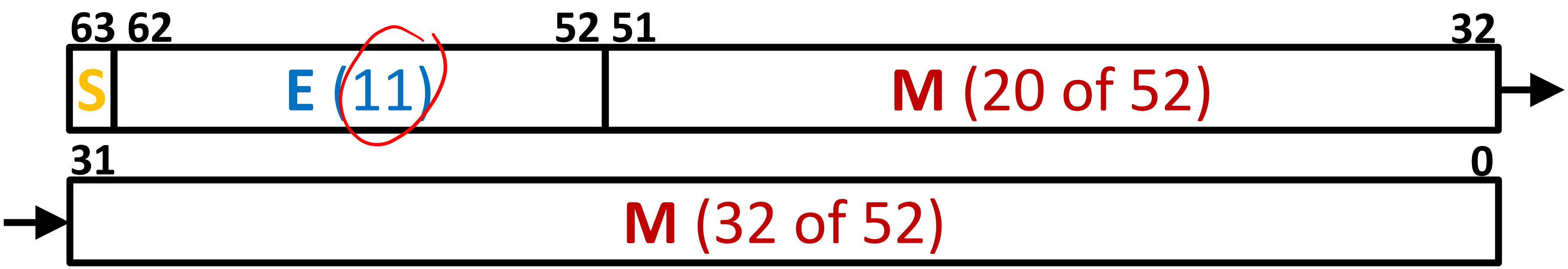
Precision and Accuracy

- 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

1111 1111 0x80000000 = -0

- E = 0xFF, M = 0: ± ∞
  - *e.g.*, division by 0
  - Still work in comparisons!

all ones

- 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 (tells you cause of NaN)

Floating Point Encoding Summary

E	M	Meaning
0x00	0	± 0
0x00	non-zero	± denorm num
0x01 – 0xFE	anything	± norm num
0xFF	0	± ∞
0xFF	non-zero	NaN

smallest E (all 0's)

everything else

largest E (all 1's)

zero

Patterns: 0000 0001 → 1111 1110 ← Interpret as Unsigned

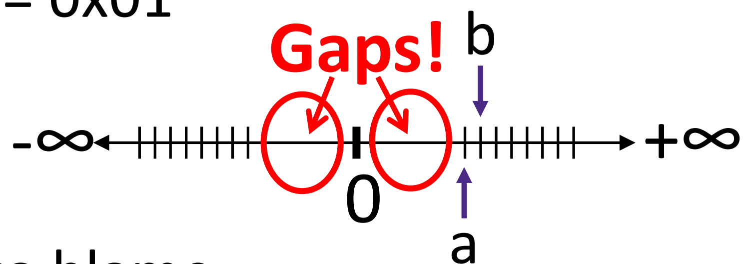
1 → 254

1-127 254-127

-126 → 127 } ← Represented Range of exponents

New Representation Limits

- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: $1.\underbrace{1\ldots1}_{\text{23 ones}}{}_2\times 2^{127} = 2^{128} - 2^{104}$
    254 - bias
- New numbers closest to 0:
  - E = 0x00 taken; next smallest is E = 0x01 Exp = -126
  - $a = 1.0\ldots00_2\times\boxed{2^{-126}} = 2^{-126}$ 23
  - $b = 1.0\ldots01_2\times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - *Special case:* E = 0, M ≠ 0 are denormalized numbers (0.M)
    normalized: 1.M



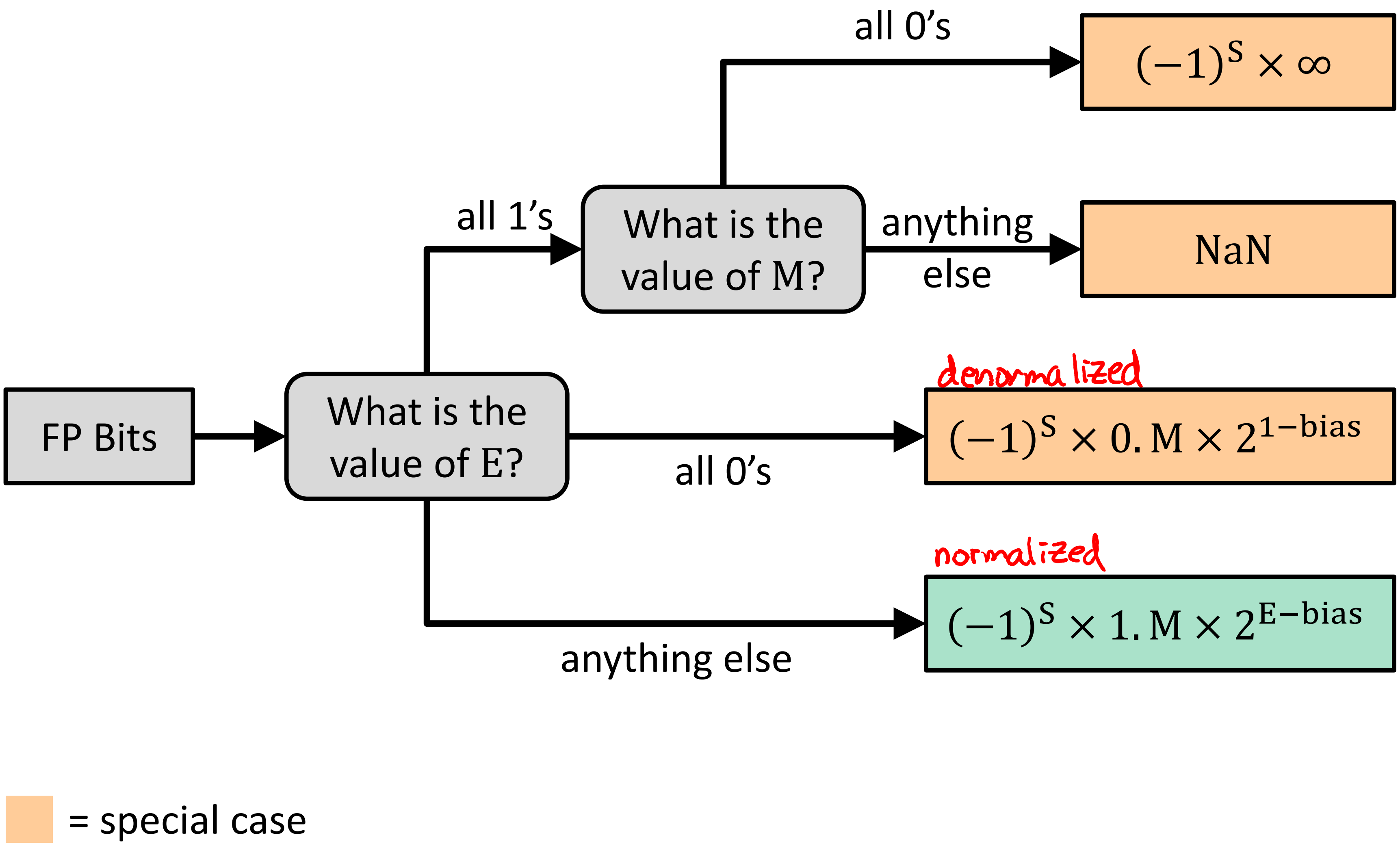
Denorm Numbers

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: $\pm\, 1.0\ldots00_{\text{two}} \times 2^{-126} = \pm\, 2^{-126}$
  - Smallest denorm: $\pm\, 0.0\ldots01_{\text{two}} \times 2^{-126} = \pm\, 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number

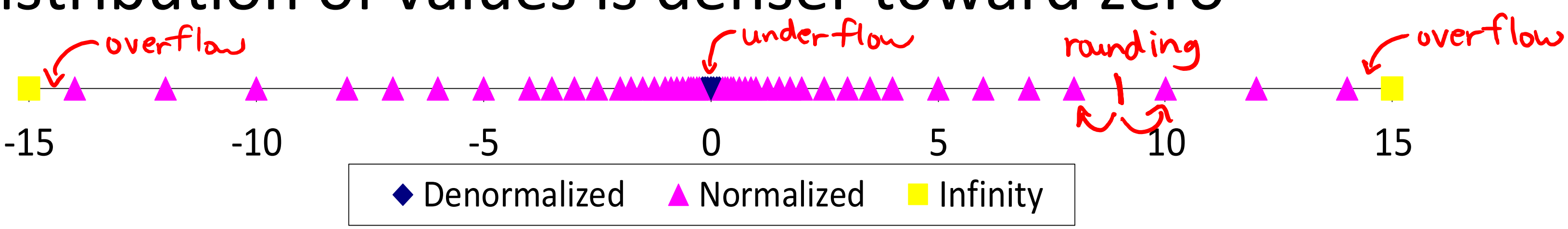
So much closer to 0

Floating Point Interpretation Flow Chart



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

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

Floating Point Operations: Basic Idea

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

S	E	M

- `x +f y = Round(x + y)`
- `x *f y = 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: $\underbrace{\overbrace{(\texttt{3.14+}\overset{10^{100}}{\texttt{1e100}})}^{10^{100}}\texttt{-}\overset{10^{100}}{\texttt{1e100}}}_{\texttt{0}}$ `!=` $\underbrace{\texttt{3.14+}\overbrace{(\overset{10^{100}}{\texttt{1e100}}\texttt{-}\overset{10^{100}}{\texttt{1e100}})}^{0}}_{\texttt{3.14}}$
  - Not distributive: $\underbrace{\texttt{100*(0.1+0.2)}}_{\texttt{30.000000000000003553}}$ `!=` $\underbrace{\texttt{100*0.1+100*0.2}}_{\texttt{30}}$
  - 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!

instead use $abs(f1 - f2) < 2^{-20}$

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 `int`s 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)

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 (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 `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