Floating Point I

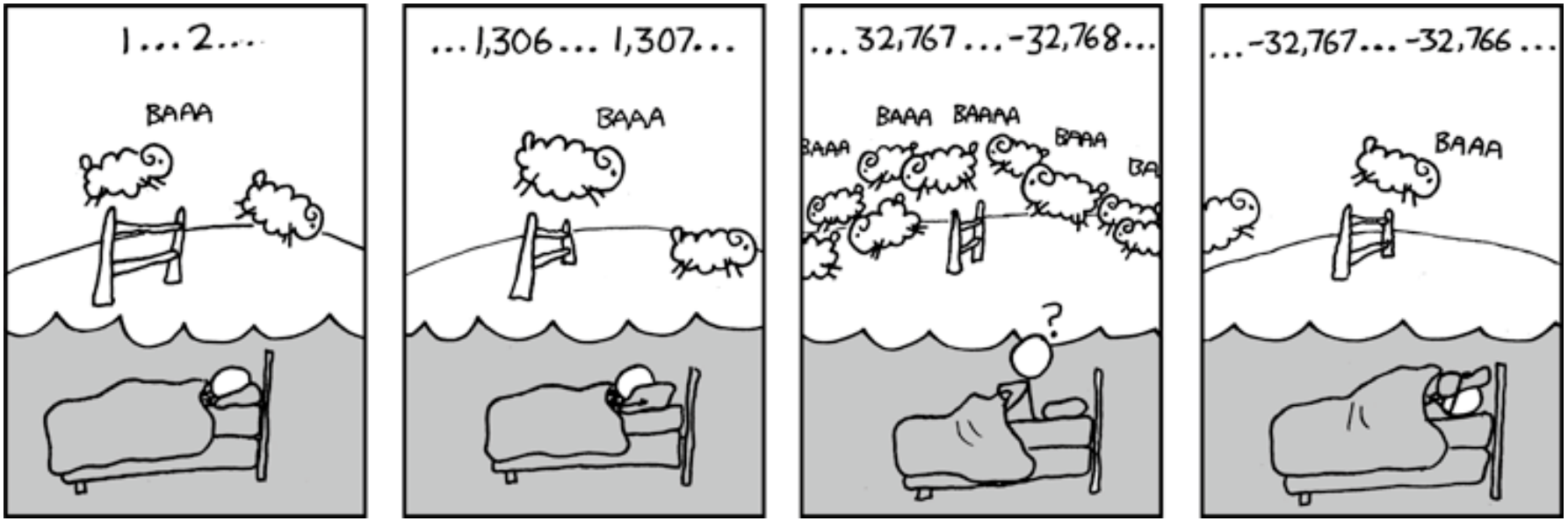
CSE 351 Autumn 2019

Instructor:

Justin Hsia

Teaching Assistants:

Andrew Hu
Antonio Castelli
Cosmo Wang
Diya Joy
Ivy Yu
Kaelin Laundry
Maurice Montag
Melissa Birchfield
Millicent Li
Suraj Jagadeesh

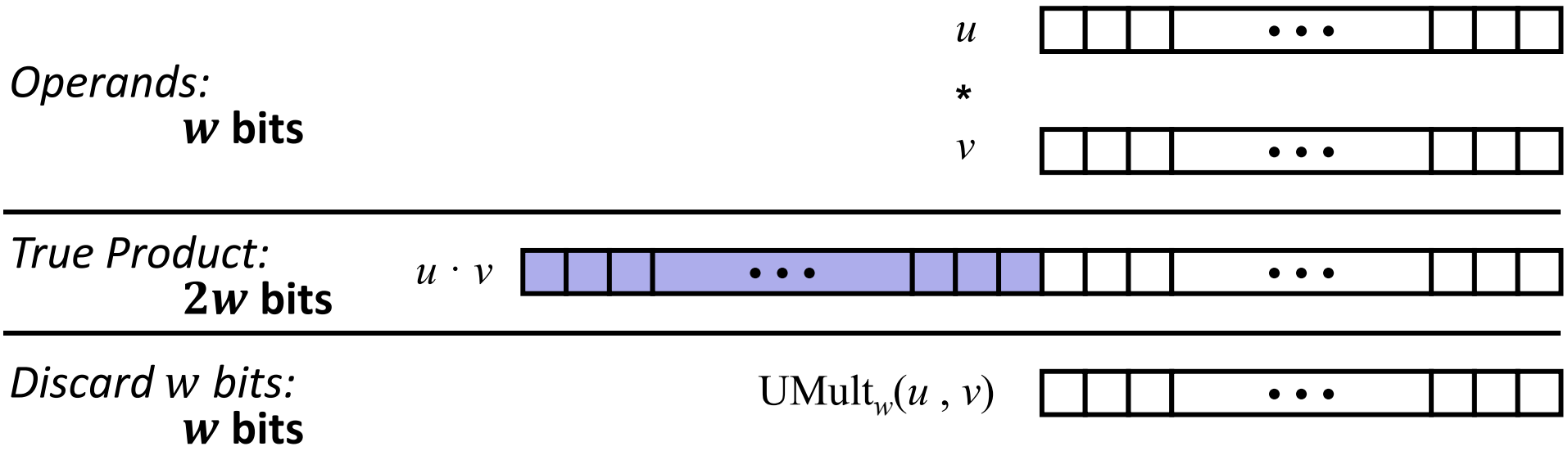


http://xkcd.com/571/

Administrivia

- Lab 1a due tonight at 11:59 pm
  - Submit `pointer.c` and `lab1Areflect.txt`
  - Make sure you submit *something* before the deadline and that the file names are correct
- hw5 due Wednesday, hw6 due Friday
- Lab 1b due next Monday (10/14)
  - Submit `bits.c` and `lab1Breflect.txt`

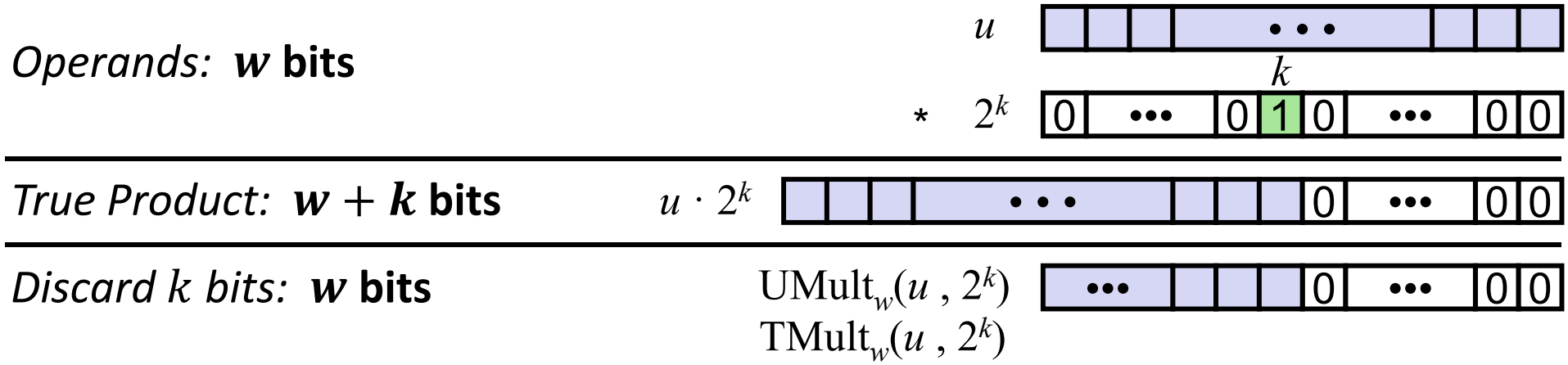
Unsigned Multiplication in C



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

Multiplication with shift and add

- Operation `u<<k` gives `u*`$\mathbf{2^k}$
  - Both signed and unsigned



- Examples:
  - `u<<3 == u * 8`
  - `u<<5 - u<<3 == u * 24`
  - Most machines shift and add faster than multiply
    - ***Compiler generates this code automatically***

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

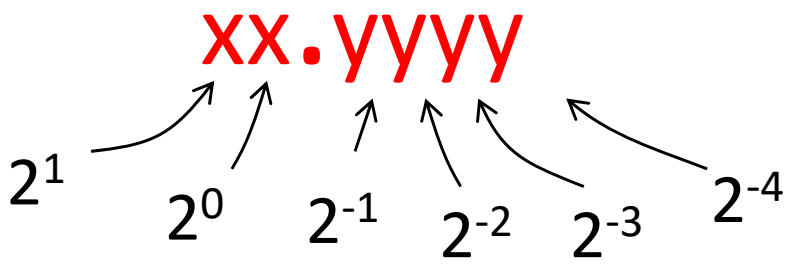
- **Fractional binary numbers**
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- 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:

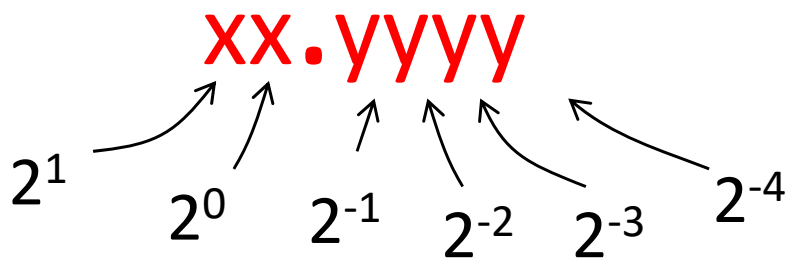


- Example: $10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}$

Representation of Fractions

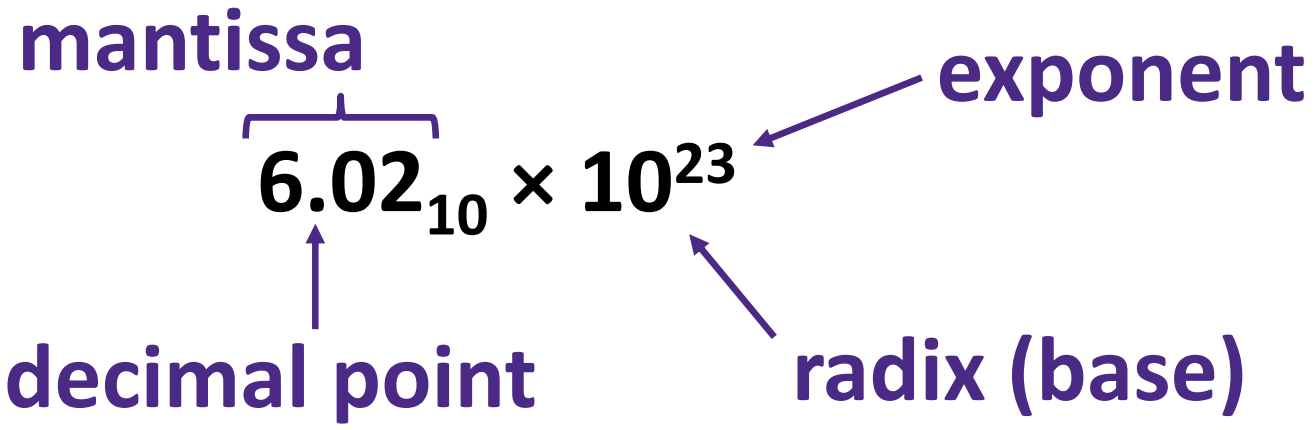
- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

Example 6-bit representation:



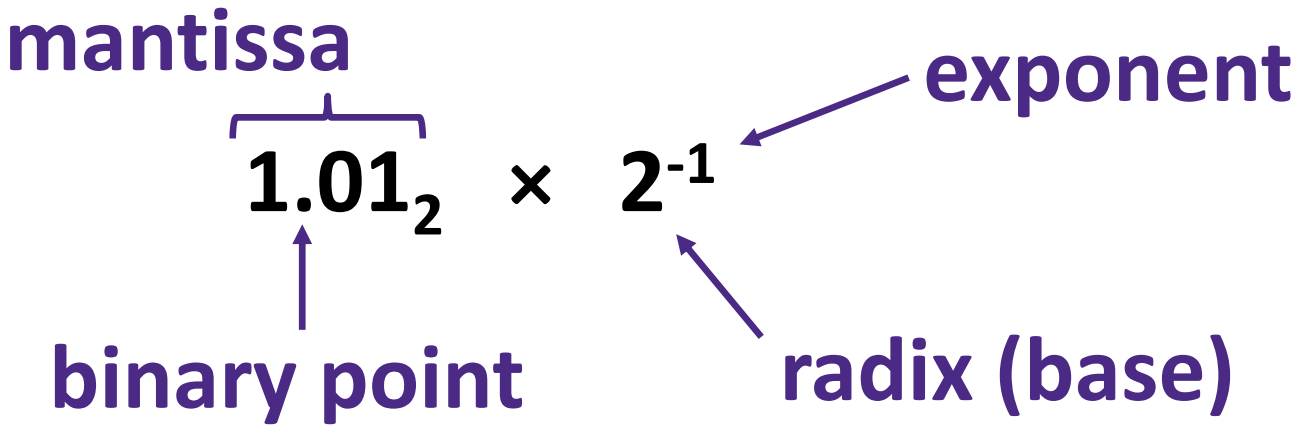
- In this 6-bit representation:
  - What is the encoding and value of the smallest (most negative) number?
  - What is the encoding and value of the largest (most positive) number?
  - What is the smallest number greater than 2 that we can represent?

Scientific Notation (Decimal)



- *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: 0.1×10^{-8},10.0×10^{-10}

Scientific Notation (Binary)



- 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

- 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

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…

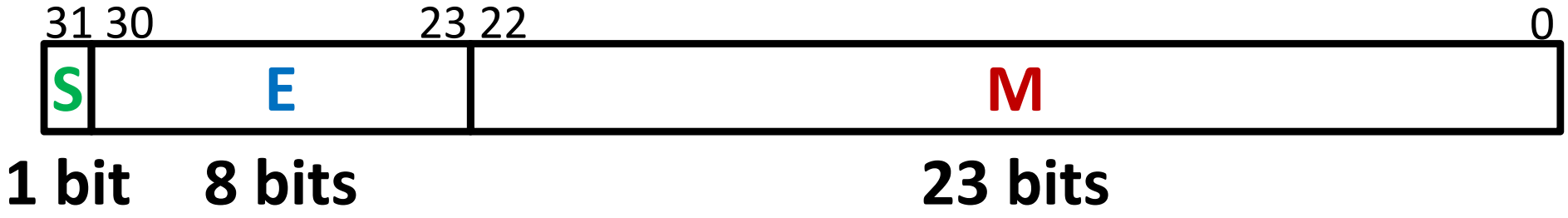
IEEE Floating Point

- 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

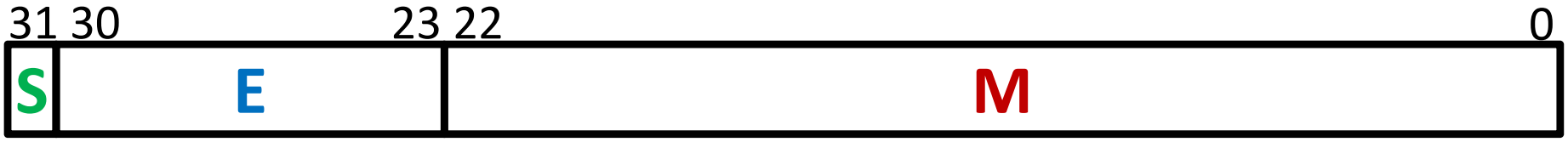
- Use normalized, base 2 scientific notation:
  - Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
  - Bit Fields: $(-1)^{S} \times 1.M \times 2^{(E-\text{bias})}$
- Representation Scheme:
  - Sign bit (0 is positive, 1 is negative)
  - Mantissa (a.k.a. significand) is the fractional 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

- 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 → → E = 0b
  - Exp = 127 → → E = 0b
  - Exp = -63 → → E = 0b

The Mantissa (Fraction) Field



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

- 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^{Exp+1}

Polling Question

- What is the correct value encoded by the following floating point number?
  - 0b 0 10000000 11000000000000000000000
  - Vote at http://PollEv.com/justinh

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 $2^{E - bias}$.
  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.

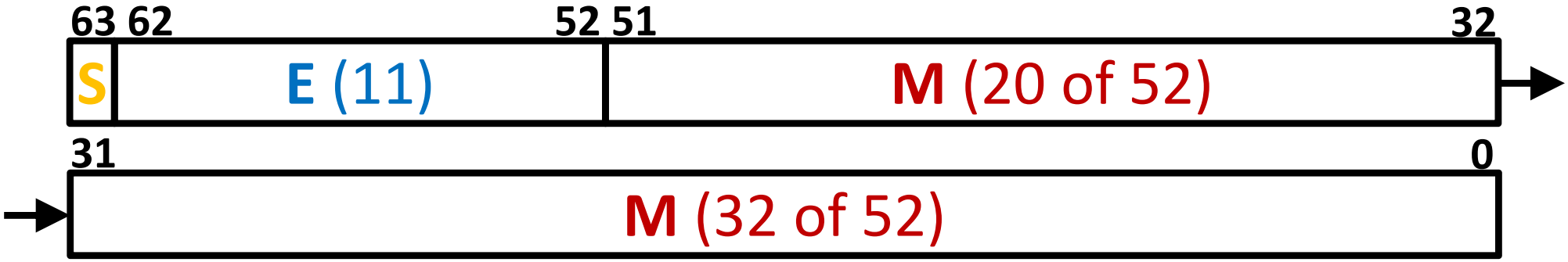
Precision and Accuracy

- 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

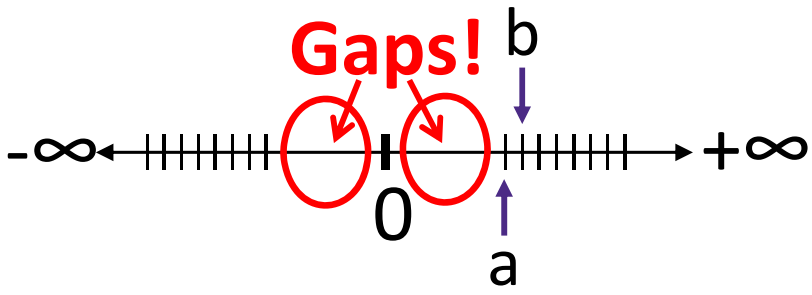


  - C variable declared as `double`
  - Exponent bias is now $2^{10}-1 = 1023$
  - **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
  - **Disadvantages:** more bits used, slower to manipulate

Representing Very Small Numbers

- But wait… what happened to zero?
  - Using standard encoding 0x00000000 =
  - *Special case:* E and M all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers

- New numbers closest to 0:
  - a = $1.0\ldots0_2 \times 2^{-126} = 2^{-126}$
  - 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



Denorm Numbers

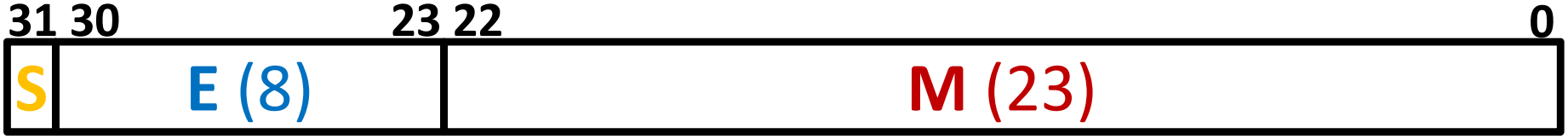
This is extra (non-testable) material

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

Summary

- Floating point approximates real numbers:



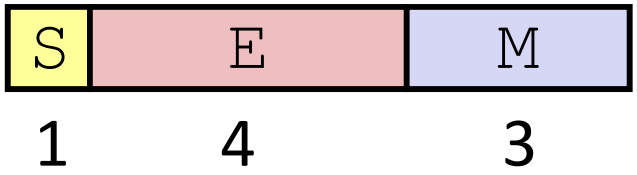
  - Handles large numbers, small numbers, special numbers
  - Exponent in biased notation (bias = $2^{w-1}-1$)
    - Size of exponent field determines our representable *range*
    - Outside of representable exponents is *overflow* and *underflow*
  - Mantissa approximates fractional portion of binary point
    - Size of mantissa field determines our representable *precision*
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes *rounding*

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, ∞

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