Floating Point I

CSE 351 Spring 2019

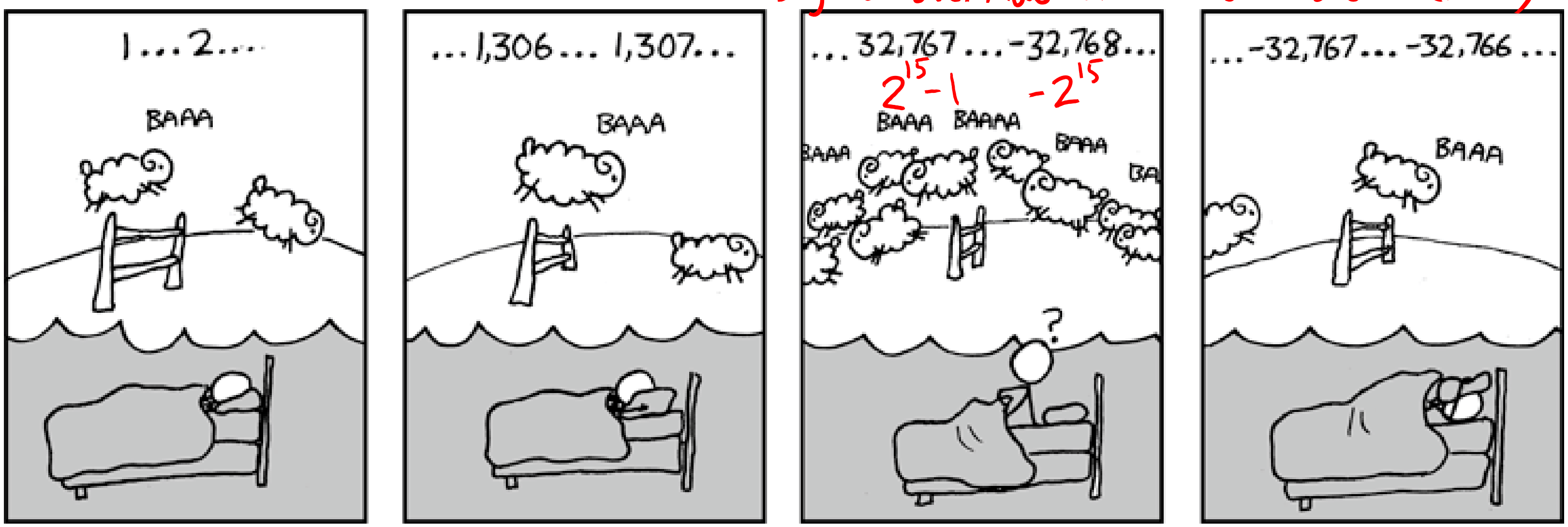
Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai, Jack Eggleston, John Feltrup, Britt Henderson, Richard Jiang, Jack Skalitzky, Sophie Tian, Connie Wang, Sam Wolfson, Casey Xing, Chin Yeoh

signed overflow in 16 bits → short (in C)



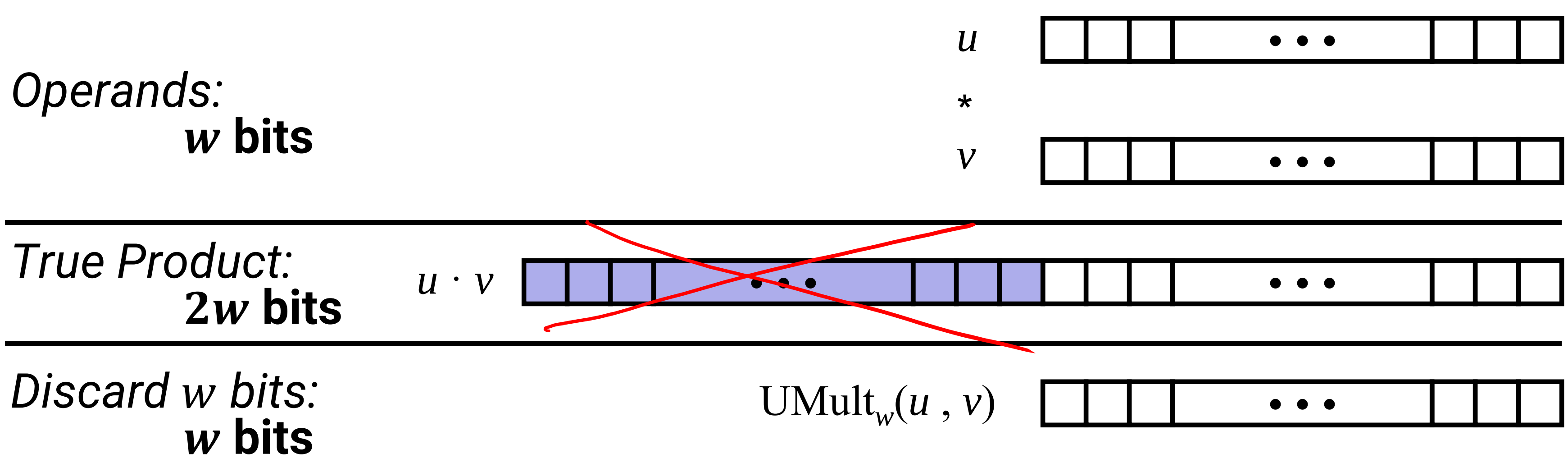
http://xkcd.com/571/

Administrivia

- Lab 1a due Monday 4/15 at 11:59 pm
  - Submit `pointer.c` and `lab1Areflect.txt`
- Lab 1b due Monday (4/22)
  - Submit `bits.c` and `lab1Breflect.txt`

- Homework 2 coming soon, due Wednesday (4/24)
  - On Integers, Floating Point, and x86-64

Unsigned Multiplication in C



- Standard Multiplication Function
  - Ignores high order w bits

When we multiply two unsigned w-bit numbers, it could possibly take 2w bits to store
the result (e.g. if we multiply 2^w-1 by 2^w-1, the max number representable by a 2bit unsigned int, then the product is 2^{2w}--2^{w+1}+ 1 which requires 2^{2w} bits to represent

- Implements Modular Arithmetic
  - $\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

Again, instead of allocating more memory, we just throw away the higher order bits and treat multiplication like its mod 2^w (same as addition)

Multiplication with shift and add

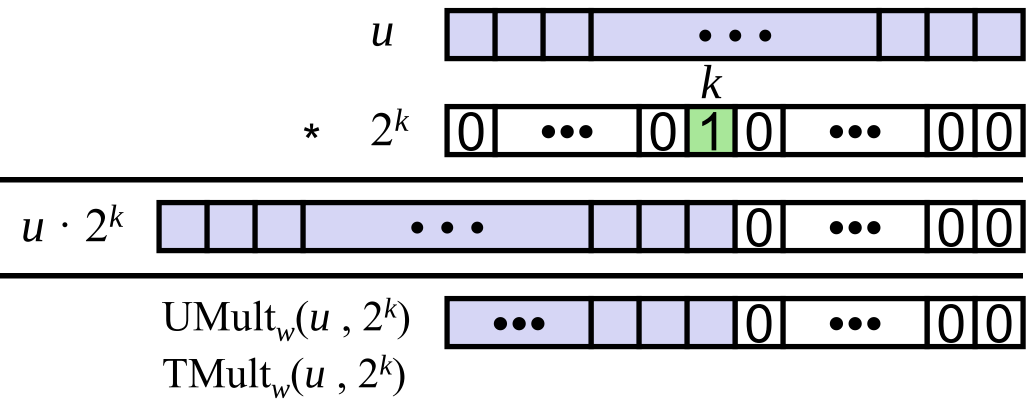
This just appends zeros on the right hand side and throws away higher order bits

- Operation u<<k gives u*2^k (as long as no overflow occurs; operations are still modulo 2^w)
  - Both signed and unsigned

Operands: ***w*** **bits**

True Product: ***w*** **+** ***k*** **bits**

Discard k bits: ***w*** **bits**



- Examples:
  - u<<3 == u * 8
  - u<<5 - u<<3 == u * 24 → 24 = 32 - 8 (2^5 2^3) ; → 24 = 16 + 8
    u<<4 + u<<3
  - Most machines shift and add faster than multiply
    - ***Compiler generates this code automatically***

Number Representation Revisited

- We know how to represent:
  - 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…

Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like `ints`
  - "Gaps" produced in representable numbers means we can lose precision, unlike `ints`

    We can only represent a finite number of numbers with our finite number of bits; so theres no way we can represent any kind of interval of real numbers
    - 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`!

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

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}

0.5 0.25 0.125 0.0625

- In this 6-bit representation:
  - What is the encoding and value of the smallest (most negative) number?

    $00.0000_2 = 0$

  - What is the encoding and value of the largest (most positive) number?

    $11.111\underbrace{1}_{2^{-4}} = 4 - 2^{-4}$

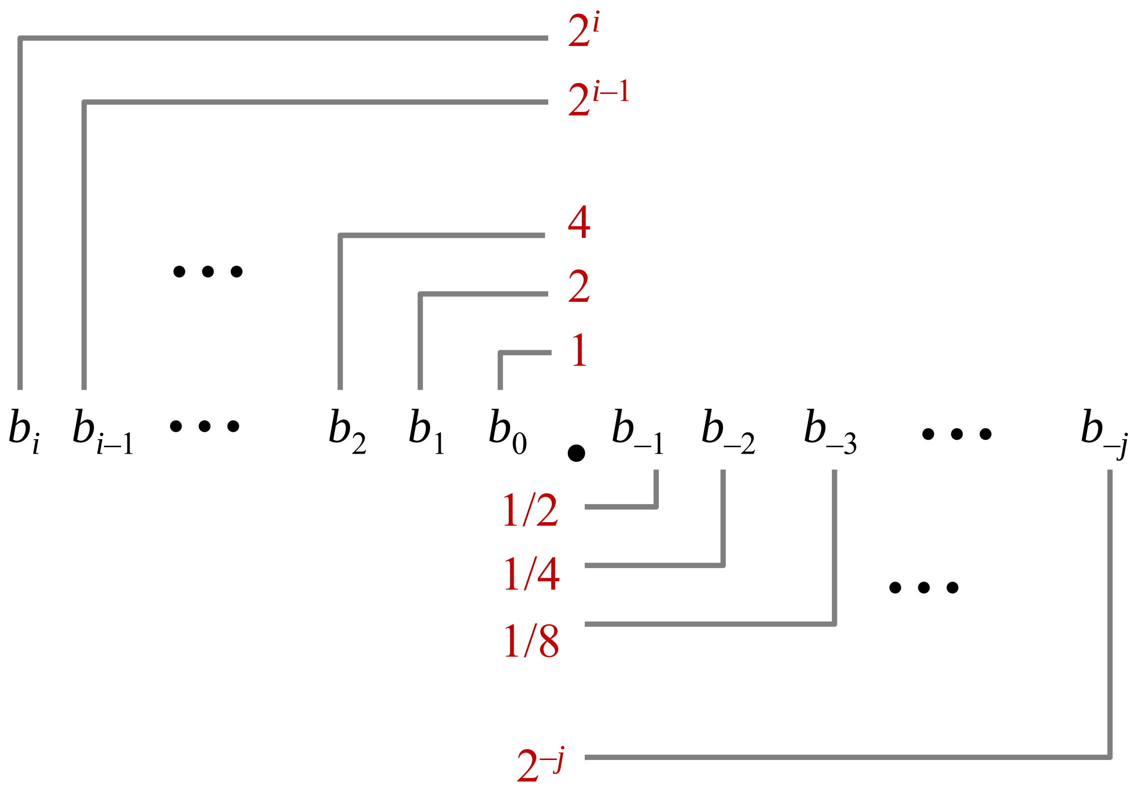
  - What is the smallest number greater than 2 that we can represent?

    $2 = 10.0000_2$

    $10.0001 = 2 + 2^{-4}$

    can't represent anything in-between! ☹

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
  - Numbers of the form $\mathbf{0.111111\ldots_2}$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

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)
    where x is an integer
  - Other rational numbers have repeating bit representations

Value: **Binary Representation:**

- $1/3 = 0.333333..._{10} =$ $0.01010101[01]..._2$
- $1/5 =$ 0.2_{10} $0.001100110011[0011]..._2$
- $1/10 =$ 0.1_{10} $0.0001100110011[0011]..._2$

<u>Fixed</u> Point Representation

e.g how many bits do we allocate to the decimal part and how many bits do we allocate to the integer part

- 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

As opposed to fractional binary numbers, we now try to represent numbers of the form decimal * 2^exponent
So we've given more flexibility to the first term essentially

- 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 $\boxed{1.1 \times 2^4}$
    - 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) this is the number being multiplied by 2^k
  - the exponent The power on 2^k

Scientific Notation (Decimal)

(significand)
mantissa

exponent

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

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: 0.1×10^{-8}, 10.0×10^{-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`)

The question still is what is the best way to encode the "mantissa" and the exponent, and how many bits should we allocate to represent each.

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 binary scientific notation

$8+2+1+0.25+0.125$

$2^3+2^1+2^0+2^{-2}+2^{-3} = 1011.011_2 = \boxed{1.011011 \times 2^3}$

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**

  competing goals
  - 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: (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**

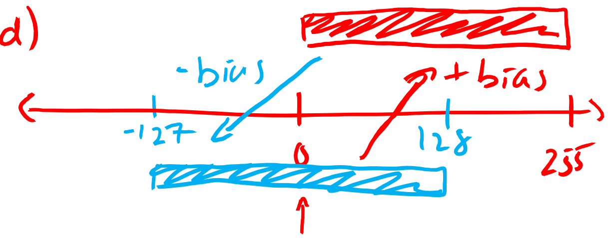
binary encodings

single precision float

The Exponent Field

E: (unsigned)

Exp: (biased)



- 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
  - Exponent 0 (Exp = 0) is represented as E = 0b 0111 1111 $= 2^7 - 1$

    E - bias = 0 = Exp ✓
- 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 $\xrightarrow{+\text{bias}}$ 128 $\xrightarrow{\text{encode}}$ E = 0b 1000 0000
  - Exp = 127 → 254 → E = 0b 1111 1110 $(254 = 255 - 1 = (2^8 - 1) - 1)$ 8 ones in a row
  - Exp = -63 → 64 → E = 0b 0100 0000

conflicting notation with what is used in the textbook. The textbook calls E = exp and Exp = E

The Mantissa (Fraction) Field

31	30 ... 23	22 ... 0
S	exp E	frac M
1 bit	8 bits	23 bits

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

$1.1 * 2^0$

- 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* $0.1_2 = 0.5_{10}$

    exponent = 0

    $2^0 \times 1.10\ldots0$

    1
  - Gives us an extra bit of *precision*
- Mantissa "limits"
  - Low values near M = 0b0…0 are close to 2^{Exp}

    $2^{Exp} \times 1.0\ldots0 = 2^{Exp}$
  - High values near M = 0b1…1 are close to $2^{\text{Exp}+1}$

    $2^{Exp} \times 1.1\ldots1 = 2^{Exp}(2 - 2^{-23}) = 2^{Exp+1} - 2^{Exp-23}$

Peer Instruction Question

Bias = 127

- What is the correct value encoded by the following floating point number?
  - 0b 0 10000000 11000000000000000000000

    S E M

    128 - 127

    + Exp = 1 Man = 1.110…0

    implicit
  - Vote at http://pollev.com/rea

A. + 0.75

B. + 1.5

C. + 2.75

D. + 3.5

E. We're lost…

$+1.11_2 \times 2^1$

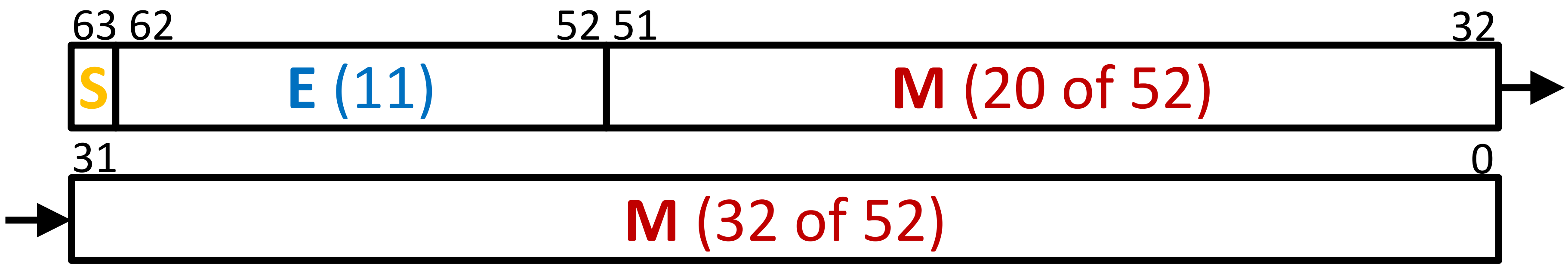
$11.1_2 = 2^1 + 2^0 + 2^{-1} = 3.5$

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

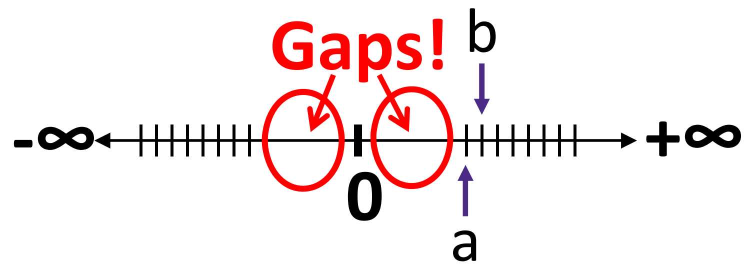


  - C variable declared as `double` vs float which is only single precision
  - Exponent bias is now $2^{10}–1 = 1023$, bias $= 2^{w-1}-1$
  - **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
  - **Disadvantages:** more bits used, slower to manipulate

Representing Very Small Numbers

- But wait… what happened to zero?
  S=0, E=0, M=0 ⇒ Exp = −127, Man = 1.0…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:
  (E = 0x01, Exp = −126)
  - a = $1.0…0_2 \times 2^{-126} = 2^{-126}$
  - b = $1.0…01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$ (23)
  - 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

Because we have taken out the implicit 1 at the front of M
Namely we use 0.M x 2^exp instead of 1.M x 2^exp

- 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

So much closer to 0

Other Special Cases

- 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
- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: $1.1\ldots1_2 \times 2^{127} = 2^{128} - 2^{104}$

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

Summary

- Floating point approximates real numbers:

31	30 … 23	22 … 0
S	E (8)	M (23)

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

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

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	
	…				
Exp = 1 - Bias frac = M	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
Exp = E - Bias frac = 1 + M	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