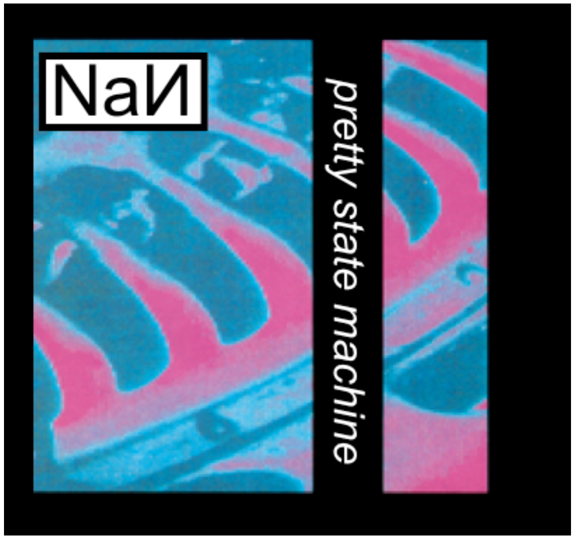
Floating Point Arithmetic



Administrivia

- We will be keeping the lecture online for at least the next 2 weeks
  - Recording a lecture while wearing a mask... 😞
  - However, if there is interest we will have the "not lecturing instructor" in 310 Soda during the lecture...
  - And broadcast the zoom lecture in that space

Outline

- Revisit Number Representation
- Floating-Point Representation and Arithmetic

Back to Number Representation — Working Towards Floating Point

- Reminder, a collection of ***n*** bits can represent one of any 2^n "things"
- Our default is "unsigned integer"
  - 0 to 2^n-1
  - Naturally good for representing addresses
- Also like "signed" as 2s-complement →
  - -2^{n-1} to $2^{n-1}-1$
- For both of these the math is "easy"
  - Addition ***and subtraction*** are the same for both
    - Subtract by just inverting and adding one...

First bit represents +/− (0 ↓ +, 1 ↓ −)
① If +
other bits represents value
② If −
val = −(2ⁿ − value)

Some other cool arithmetic tricks

- Does $x == y$?
  - Easy test: does $x - y == 0$?
- Multiply by 2^n?
  - We left shift (<<) (move the bits to the left) by n
- Can we similarly divide by 2^n?
  - We right shift (>>) by n
  - For unsigned (logical) shift: Left gets 0s
  - For signed (arithmetic) shift: Left gets the sign bit
    - Not quite right for negative numbers:
      you'd say $-1/2 = 0$, but in 2s complement $-1 >> 1 = -1$

But "Any one of 2^n" is whatever we make it to be!

- One alternate representation: Sign/Magnitude
  - Lets have the first bit say the sign (+ or - as 0 or 1)
  - And the rest be unsigned
- Allows us to represent $-2^{n-1}+1$ to $2^{n-1}-1$
- This gives us two zeros (+/- 0)...
- This gives us a cleaner symmetry otherwise
  - Magnitude is consistent for both positive and negative
- But math is more of a pain...
  - So a poor choice if we want to do "simple" math like add and subtract...

Another Alternative Representation: Biased...

- The actual value is the binary value plus a fixed bias
  - So "bias = -127" means the actual number is the binary value with -127 added to it
  - Binary ***00000000*** -> ***-127***
  - Binary ***11111111*** -> ***+128***
- Why do this?
  - Can set our range to be arbitrary
  - No discontinuity around 0
- Disadvantages
  - All bits 0 != 0
  - Math more of a pain: To add A + B...
    - ***A + B - bias*** (To eliminate the extra bias)

Other Numbers

- Numbers with both integer & fractional parts?
  - ex: 1.5
- Very large numbers? (how big is the universe)
  - 860,000,000,000,000,000,000,000,000 m in diameter (give-or-take...)
  - aka 860 yottameters...
- Very small numbers? (Bohr radius of an atom)
  - 0.000000000000000877 m in diameter (give or take ± 0.000000000000000007m)
  - aka 0.877 femtometers...
- Notice the huge range!

Representation of Fractions

- Look at decimal (base 10) first:
- Decimal "point" signifies boundary between integer and fractional parts:

  Example 6-digit representation:

  xx.yyyy

  10^{1} 10^{0} 10^{-1} 10^{-2} 10^{-3} 10^{-4}

$25.2406_{ten} = 2 \times 10^{1} + 5 \times 10^{0} + 2 \times 10^{-1} + 4 \times 10^{-2} + 6 \times 10^{-4}$

If we assume "fixed decimal point", range of 6-digit representations with this format: 0 to 99.9999. Not much range, but lots of "precision": 6 significant figures

Binary 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}

$$\mathbf{10.1010_{two} = 1x2^{1} + 1x2^{-1} + 1x2^{-3} = 2.625_{ten}}$$

If we assume "fixed binary point", range of 6-bit representations with this format: 0 to 3.9375 (almost 4)

Fractional Powers of 2

i	2^{-i} (base 2)	(base 10)	(fraction)
0	1.0	1.0	1
1	0.01	0.5	1/2
2	0.001	0.25	1/4
3	0.0001	0.125	1/8
4	0.00001	0.0625	1/16
5	0.000001	0.03125	1/32
6	0.0000001	0.015625	1/64
7	0.00000001	0.0078125	1/128
8	0.000000001	0.00390625	1/256
9	0.0000000001	0.001953125	1/512
10	0.00000000001	0.0009765625	1/1024
11	0.000000000001	0.00048828125	1/2048
12	0.0000000000001	0.000244140625	1/4096

Representation of Fractions with Fixed Point
What about addition and multiplication?

Addition is straightforward:

```
  01.100   1.5ten
+ 00.100   0.5ten
---------
  10.000   2.0ten
```

Multiplication a bit more complex:

```
       01.100    1.5ten
       00.100    0.5ten
      -------
       00 000
      000 00
    0 110 0
   00 000
  000 00
 -------------
 0000 110000
 0000.110000
```

Where's the answer, `0.11`? (i.e., 0.5 + 0.25; Need to remember where point is!)

Representation of Fractions

- Our examples used a “fixed” binary point.
  What we really want is to “float” the binary point to make most effective use of limited bits
- With floating-point representation, each numeral carries an exponent field recording the whereabouts of its binary point
- Binary point can be outside the stored bits, so very large and small numbers can be represented

$$\ldots\ 000000.00\underbrace{10101}_{}00000\ldots$$

Store these bits and keep track of the binary point as 2 places to the left of the MSB

Any other solution would lose precision!

Scientific Notation (in Decimal)

mantissa

exponent

$$6.02_{ten} \times 10^{23}$$

decimal point

radix (base)

- Normalized form: no leadings 0s
  (exactly one digit 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}

Other Numbers Redux

- Numbers with both integer & fractional parts?
  - 1.5×10^{0}
    - Also written as 1.5e0
- Very large numbers? (how big is the universe)
  - 8.6×10^{26} m in diameter (give-or-take...)
- Very small numbers? (Bohr radius of an atom)
  - 8.77×10^{-16} m in diameter (give or take $\pm$ 7×10^{-18} m)
- Separate out the notion of "precision" from "range"
  - Can represent a very large range with roughly the same "precision"
    So the universe we can measure relative to the size of the universe...
  - While atoms are measured relative to the size of atoms...

Scientific Notation (in Binary)

mantissa exponent

$$1.01_{\text{two}} \times 2^{-1}$$

binary point radix (base)

- Computer arithmetic that supports it is called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`
    - `double` for double precision

UCB's "Father" of IEEE Floating point

IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989
ACM Turing
Award Winner!

Prof. Kahan

`www.cs.berkeley.edu/~wkahan/`
`.../ieee754status/754story.html`

Goals for IEEE 754 Floating-Point Standard

- Standard arithmetic for reals for all computers
  - Important because computer representation of real numbers is approximate. Want same results on all computers.
- Keep as much precision as possible
- Help programmer with errors in real arithmetic
  - +∞, -∞, Not-A-Number (NaN), exponent overflow, exponent underflow, +/- zero
- Keep encoding that is somewhat compatible with two's complement
  - E.g., +0 in Fl. Pt. is 0 in two's complement
  - Make it possible to sort ***without*** needing to do floating-point comparisons

Floating-Point Representation

- For "single precision", a 32-bit word.
- IEEE 754 single precision Floating-Point Standard:
  - 1 bit for ***sign (s)*** of floating point number
  - 8 bits for ***exponent (E)***
  - 23 bits for ***fraction (F)***
    (get 1 extra bit of precision because leading 1 is implicit: there should always be a 1 so why store it at all?)

$$(-1)^s \times (1 + F) \times 2^E$$

- Can represent approximately numbers in the range of 2.0×10^{-38} to 2.0×10^{38}

Floating-Point Representation

2^{-1} 2^{-2} 2^{-3}

- Normal format: $(-1)^S * 1.xxx\ldots x * 2^{(yyy\ldots y - 127)}$

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- S represents Sign
  - *1 for negative, 0 for positive*
- x's represent Fractional part called Significand
  - *implicit leading 1, signed-magnitude (**not 2's complement**)*
- y's represent Exponent
  - *in* biased notation (bias of ***-127***)
-

Sorting Requirement...

- We can sort the sign field by just +/-...
  - Makes it easy to separate the two.. But what then?
- We need to sort by exponent + mantissa easily
  - Thus biased notation:
    An unsigned comparison between exponents Just Works
    - Bigger is larger
  - And the exponent is more significant, so it just sorts by exponent
  - And when the exponent is the same, the mantissa sorting Just Works
- So we can sort all positive numbers together just like they were integers
- And also an exponent of 0 isn't actually special...
  - The special exponents are MAX and MIN...

Bias Notation (exponent = stored value - 127)

How it is interpreted

How it is encoded

∞, NaN

Getting closer to zero

Zero

Decimal Exponent	signed 2's complement	Biased Notation	Decimal Value of Biased Notation
For infinities		11111111	255
127	01111111	11111110	254
...	...	...	...
2	00000010	10000001	129
1	00000001	10000000	128
0	00000000	01111111	127
-1	11111111	01111110	126
-2	11111110	01111101	125
...	...	...	...
-126	10000010	00000001	1
For Denorms	10000001	00000000	0

64 bits

1 bit	11 bits	52 bits
S	Exponent	Significand

Floating-Point Representation

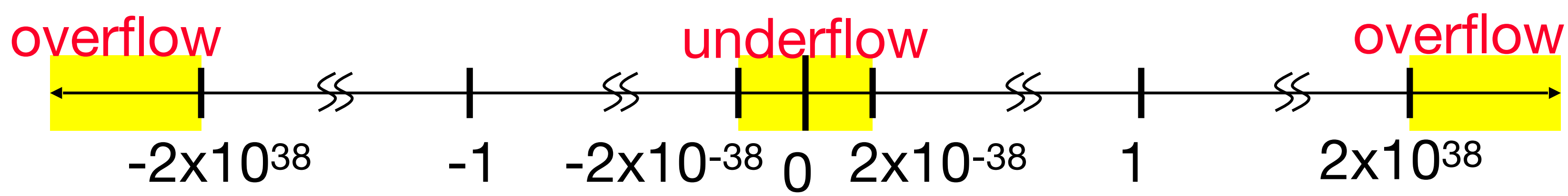
- What about bigger or smaller numbers?
- IEEE 754 Floating-Point ***Double Precision Standard*** (64 bits)
  - 1 bit for ***sign (s)*** of floating-point number
  - 11 bits for ***exponent (E)*** with a bias of -1023
  - 52 bits for ***fraction (F)***
    (get 1 extra bit of precision if leading 1 is implicit)

$(-1)^s \times (1 + F) \times 2^E$

- Can represent from 2.0×10^{-308} to 2.0×10^{308}
- More importantly, 53 bits of precision!
- Recall, 32-bit format called ***Single Precision***
- The FP specifications for bit pattern and biases are printed on the RISC-V green sheet

Floating-Point Representation

- What if result too large?
  (> 2.0×10^{38} , < -2.0×10^{38})
  - Overflow! ⇒ Exponent larger than represented in 8-bit Exponent field
- What if result too small?
  (>0 & < 2.0×10^{-38} , <0 & > -2.0×10^{-38})
  - Underflow! ⇒ Negative exponent larger than represented in 8-bit Exponent field



- What would help reduce chances of overflow and/or underflow?

Lets consider two exponents "special"

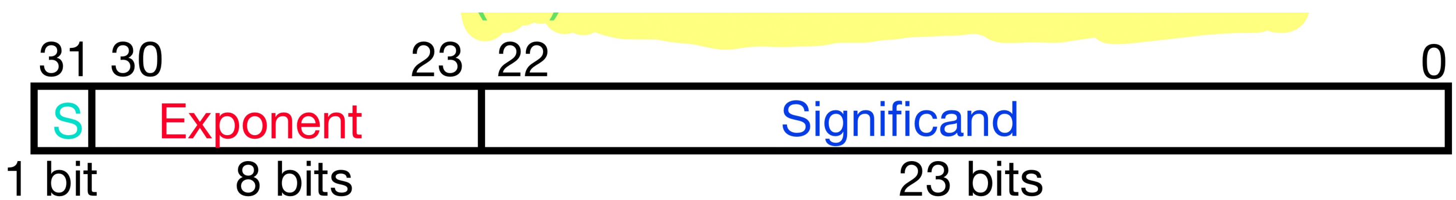
- Exponent all-zeros
  - Very small numbers
- Exponent all-ones
  - Infinity/NaN...
- What these do we will get to in a bit...

Example

- What's the base 10 value of this single precision Floating Point number?

$$\underbrace{1}_{-1}\ \underbrace{1000\ 0000}_{E = 2^7 - 127 = 1}\ \ \underbrace{1000\ 0000\ 0000\ 0000\ 0000\ 000}_{0}$$

$$(-1) \times (1.1)_2 \times 2^1 = -1 \times 2 \times (1.1)_2$$



Example

- What's the base-10 value of this single precision Floating Point number?
- 1 1000 0000 1000 0000 0000 0000 0000 000
- $-1 * 2^{128-127} * 1.1_2$
- -1.5 * 2
- -3

More Floating Point: Preview

- What about 0?
  - Bit pattern all 0s means 0 (so no implicit leading 1 in this case)
- What if divide 1 by 0?
  - Can get infinity symbols +∞, -∞
  - Sign bit 0 or 1, largest exponent (all 1s), 0 in fraction
- What if do something stupid? (∞ – ∞, 0 ÷ 0)
  - Can get special symbols NaN for "Not-a-Number"
  - Sign bit 0 or 1, largest exponent (all 1s), not ***zero in fraction***
- What if result is too big?
  - Get ***overflow*** in exponent, alert programmer!
- What if result is too small?
  - Get ***underflow*** in exponent, alert programmer!

Representation for 0

- Represent 0?
  - Exponent all zeroes
  - Significand all zeroes
  - What about sign? Both cases valid!

```
+0: 0 00000000 00000000000000000000000
-0: 1 00000000 00000000000000000000000
```

Because it isn't ***really zero!***

- +0 is really "This number is too small to represent and either zero or somewhere between 0 and our smallest number"
- -0 is really "This number is too small to represent, and either zero or somewhere between 0 and our smallest negative number"

Representation for $\pm\infty$

- In FP, divide by 0 should produce $\pm\infty$, not overflow
- Why?
  - OK to do further computations with ∞
    E.g., $X/0 > Y$ may be a valid comparison
- IEEE 754 represents $\pm\infty$
  - Most positive exponent reserved for ∞
  - Significand ***all zeroes***

Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	Normal Floating Point
255	0	Infinity
255	Nonzero	???

Representation for Not-a-Number

- What do I get if I calculate sqrt(-4.0)or 0/0?
  - If ∞ not an error, these shouldn't be either
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN, X) = NaN
  - Can use the significand to identify which! (e.g., quiet NaNs and signaling NaNs)
- Watch out for NaN in comparisons!

NaN ≥ x	NaN ≤ x	NaN > x
Always **False**	Always **False**	Always **False**

NaN < x	NaN = x	NaN ≠ x
Always **False**	Always **False**	Always **True**

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable positive number:

    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable positive number:

    $b = 1.000\ldots\ldots1_2 * 2^{-126}$

    $= (1 + 0.00\ldots1_2) * 2^{-126}$

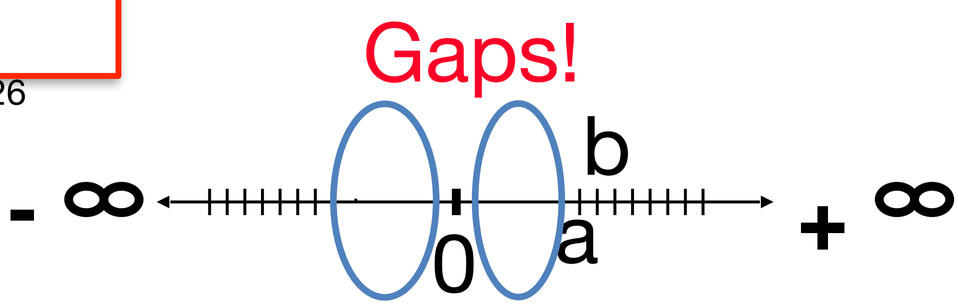
    $= (1 + 2^{-23}) * 2^{-126}$

    $= \boxed{2^{-126} + 2^{-149}}$

Normalization and implicit 1 are to blame!

$a - 0 = 2^{-126}$

$b - a = 2^{-149}$

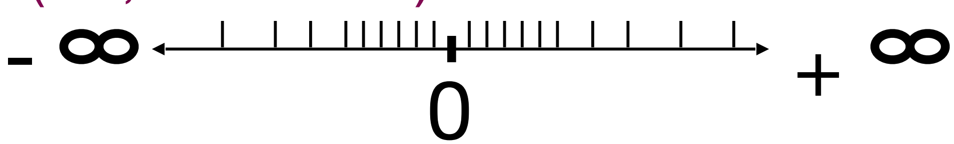


Representation for Denorms (2/2)

- Solution:
  - We still haven't used Exponent = 0, Significand nonzero
  - Denormalized number: no (implied) leading 1, *implicit exponent* **= -126**
  - Smallest representable positive number:

    a = 2^{-149} (i.e., 2^{-126} x 2^{-23})
  - Second-smallest representable positive number:

    b = 2^{-148} (i.e., 2^{-126} x 2^{-22})



Special Numbers Summary

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	aynthing	Normal Floating Point
255	0	Infinity
255	Nonzero	NaN

Saving Bits

- Many applications in machine learning, graphics, signal processing can make do with lower precision
- IEEE “half-precision” or “FP16” uses 16 bits of storage
  - 1 sign bit
  - 5 exponent bits (exponent bias of 15)
  - 10 significand bits
- It isn't just saving "bits", it is increasing parallelism...
  - The space of 4 16b floating point ALUs is ~= a single 64b ALU
  - Often tricks to make a single 64b ALU divisible into 4 16b ALUs!

So In Review...

- Floating point: we interpret sequence of bits differently than for signed/unsigned integers
  - A single sign bit (0 == positive, 1 == negative)
  - An exponent in biased form (bias = $-2^{n-1}+1$)
  - A mantissa with an implicit leading 1
- Complications occur at the edges
  - Maximum exponent {1's} -> Either ∞ or NaN
  - Minimum exponent {0's} -> Either 0 or a ***denormalization*** F=0 cannot represent
    - Fixed exponent, no more implicit leading 1

So Real Choice is Precision v Performance

- Half Precision: 16b
  - 1b signed
  - 5b exponent, bias -15
  - 10b significand
- Single precision: 32b
  - 1b signed
  - 8b exponent, bias -127
  - 24b significand
  - **`float`**
- Double precision: 64b
  - 1b signed
  - 11b exponent, bias -1023
  - 53b significand
  - **`double`**
- Quad precision: 128b
  - 1b signed
  - 15b exponent, bias -16383
  - 113b significand