/unc/comp211

Systems Fundamentals

Toward Floating Points

Fractional Numbers!

Refresher: Base 10 Decimals

What are each of the following values in Base10 decimal representation with a fixed number of possible digits?

1) 1/2

2) 1/3

3) $10\frac{1}{4}$

How do you go from a ratio to a decimal?

- How were you taught how to convert $\frac{1}{2}$ to decimal positional notation 0.5?
  - Think about how non-trivial of a leap this conversion is! Where did that 5-digit come from?!
- Long division! Convert the following rational numbers to decimals: $\frac{1}{8}$, $\frac{9}{11}$

$$\mathbf{8\,\overline{)1}} \qquad\qquad \mathbf{11\,\overline{)9}}$$

- Perform long division of 1/8 and 11/9 up to 4 significant digits.

Fractional/Positional Place Values in Base 10

Digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Negative place values are fractional.

3	2	1	0		-1	-2	-3	-4
				•				
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	10^{-4}
					$\frac{1}{10^1}$	$\frac{1}{10^2}$	$\frac{1}{10^3}$	$\frac{1}{10^4}$
1000	100	10	1	•	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1000}$	$\frac{1}{10000}$

Formalization of Fractional Values in Base-10

Suppose we define a Base10 number d, with $w+f$ place values, (*w whole, f fractional*) as a vector of **decimal digits** indexed from -*f* to *w* with an *implicit decimal between* d_0 and d_{-1}.

$$\vec{d} = [d_{w-1}, \ldots, d_0, d_{-1}, \ldots, d_{-f+1}, d_{-f}]$$

We can determine the value of $\vec{d}$ with the following summation:

$$FractionalValue_{10}(\vec{d}) = \sum_{i=-f}^{w-1} d_i \times 10^i$$

A concrete example:

$$w = 2$$

$$f = 2$$

$$\vec{d} = [1, 0, 2, 5]$$

thus

$$d_1 = 1$$

$$d_0 = 0$$

$$d_{-1} = 2$$

$$d_{-2} = 5$$

$$FV_{10}([1, 0, 2, 5]) = \sum_{i=-f}^{w-1} d_i \times 10^i$$

Fractional/Positional Place Values in Base 2

Digits = { 0, 1 }

Negative place values are fractional.

3	2	1	0		-1	-2	-3	-4
				•				
2^w	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-f}
					$\frac{1}{2^1}$	$\frac{1}{2^2}$	$\frac{1}{2^3}$	$\frac{1}{2^f}$
...	4	2	1	•	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	...

Practice: Convert 1010.1010_2 to rational and positional representations.

3	2	1	0		-1	-2	-3	-4
1	0	1	0	.	1	0	1	0
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
					$\frac{1}{2^1}$	$\frac{1}{2^2}$	$\frac{1}{2^3}$	$\frac{1}{2^f}$
8	4	2	1	•	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

How do you go from base-2 ratio to base-2 decimal? *And* base-10 representation?

- Ratio-to-Decimal: Long division! Work out the long division below.
  - Write out your numbers in base-2 even if you're "thinking" in base-10.

$\frac{\mathbf{1_2}}{\mathbf{100_2}}$

$\frac{\mathbf{1_2}}{\mathbf{11_2}}$

$\mathbf{100_2}\,\overline{)\mathbf{1_2}}$

$\mathbf{11_2}\overline{)\mathbf{1_2}}$

/unc/comp211

Systems Fundamentals

Toward Binary Floating Points

Decimal Floating Points!

Refresher: Base 10 Decimals

What are each of the following values in Base10 decimal representation with a fixed number of possible digits?

1) 1.10×10^2

2) 2.11E-1

How can we best use a fixed number of digit positions to represent a wide range of fractional values?

Design Challenge: Representing Fractional Values w/ a Finite # of Digits

- Consider the following formula: $\boldsymbol{x = 10^{E} \times C}$
  - The digits of E make up the ***exponent*** (aka order of magnitude)
  - The digits of C make up the ***significand*** *(aka coefficient or mantissa)*
    - Assume there is an **implicit decimal point *after* the first digit of C!**

- You have 3 positions to store base-10 digits 0-9 in order to represent ***any*** $\boldsymbol{x}$
  1. How many of the 3 positions would you allocate for E? To C?
     - What is the largest value you can represent with your decision? The smallest?
     - What are the fundamental trade-offs in allocating positions to E vs. C?
  2. How would you represent negative Es *without* a negative symbol?
     - With negative Es you can represent fractional values $\boldsymbol{0 < x < 1}$

Design Proposal:
0 Exponent Digits, 3 Significand Digits

Smallest Non-zero Value				Largest Value		
0	0	1		9	9	9

0.01_{10} 9.99_{10}

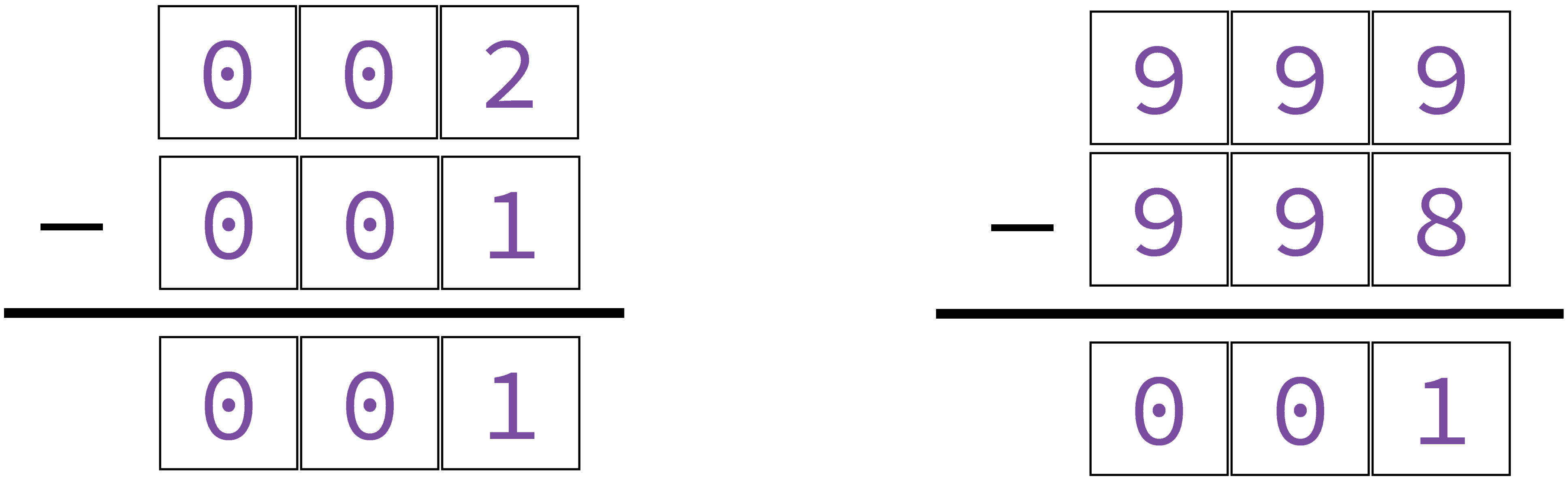
0.01_{10} 9.99_{10}

- Under this design, we can represent values between 0.01 and 9.99.

Design Evaluation - Precision

- Consider ***precision*** **in terms of the "next closest" value** you can represent to any number.
  - From the *smallest non-zero value* it would be the *next largest value*
  - From the *largest value* it would be the *next smallest value*

- Notice in this design we have a consistent precision throughout the entire range of values we can represent. The next closest value to any value in our range is always 0.01 away.



0.01_{10} 0.01_{10}

Design Proposal:
1 Exponent Digit, 2 Significand Digits

Smallest Non-zero Value

0	0	1

$$10_{10}^{0} \times 0.1_{10}$$

$$0.1_{10}$$

Largest Value

9	9	9

$$10_{10}^{9} \times 9.9_{10}$$

$$9,900,000,000_{10}$$

- Under this design, values range between 0.1 and 9.9 billion!

Design Evaluation - Precision

Exponent Digits Buy You Range at the Cost of Precision

- Notice that you have **dramatically lower precision** as the exponent increases!
- There's also a significant *imbalance* between being able to represent 0 < x < 1 and 1 < x < max. If we could represent *negative exponents* it would help!

	0	0	2
−	0	0	1
	0	0	1

$10^{0}_{10} \times 0.1_{10}$

0.1_{10}

The next closest representable value is 10 million away!?!

	9	9	9
−	9	9	8
	9	0	1

$10^{9}_{10} \times 0.1_{10}$

$10{,}000{,}000_{10}$

Modified Design Proposal:

1 Biased Exponent Digit, 2 Significand Digits

- Let's **bias** our exponent by **-4**
  - Take our **exponent** and **subtract 4** from it so that our biased **E** range is **-4** to **5**.
  - Note: The -4 bias is itself a design decision with trade-offs. It could have been *any* number.

Smallest Value

0	0	1

$$10_{10}^{0-4} \times 0.1_{10}$$

$$10_{10}^{-4} \times 0.1_{10}$$

$$0.00001_{10}$$

Largest Value

9	9	9

$$10_{10}^{9-4} \times 9.9_{10}$$

$$10_{10}^{5} \times 9.9_{10}$$

$$990,000.0_{10}$$

- PollEv.com/compunc - what is the value of **[2, 1, 1]** in this proposal?

"Floating Point"

There's a point floating around down there!

The significand's position gives a sense of precision.

0	1	2
1	1	2
2	1	2
3	1	2
4	1	2
5	1	2
6	1	2
7	1	2
8	1	2
9	1	2

$10_{10}^{0-4} \times 1.2_{10}$ $\quad 0.000120000_{10}$

$10_{10}^{1-4} \times 1.2_{10}$ $\quad 00.00120000_{10}$

$10_{10}^{2-4} \times 1.2_{10}$ $\quad 000.0120000_{10}$

$10_{10}^{3-4} \times 1.2_{10}$ $\quad 0000.120000_{10}$

$10_{10}^{4-4} \times 1.2_{10}$ $\quad 00001.20000_{10}$

$10_{10}^{5-4} \times 1.2_{10}$ $\quad 000012.0000_{10}$

$10_{10}^{6-4} \times 1.2_{10}$ $\quad 0000120.000_{10}$

$10_{10}^{7-4} \times 1.2_{10}$ $\quad 00001200.00_{10}$

$10_{10}^{8-4} \times 1.2_{10}$ $\quad 000012000.0_{10}$

$10_{10}^{9-4} \times 1.2_{10}$ $\quad 000012000\text{0}._{10}$

Extended Design Proposal:
1 Sign, 1 Biased Exponent Digit, 2 Significand Digits

- With a position to store a sign, we can represent positive and negative values across a wide range, with a loss

Smallest Negative Value

−	9	9	9

$-990,000_{10}$

Smallest Positive Value

+	0	0	1

$+0.00001_{10}$

Largest Negative Value

−	0	0	1

-0.00001_{10}

Largest Positive Value

+	9	9	9

$+990,000_{10}$

- Under this design, values range between 0.00001 and 990,000!

/unc/comp211

Systems Fundamentals

Binary Floating-Point Representations!

Proposal: Binary Floating Point with a Minifloat
1 Sign, 3 Biased Exponent Digits, 4 Significand Digits

- Sign: 0 for positive, 1 for negative
- Bias: -3_{10}
- Implicit binary point (.) following first significand digit

$$2^{E-B} \times C$$

1)

0	0	1	1	1	1	0	0

2)

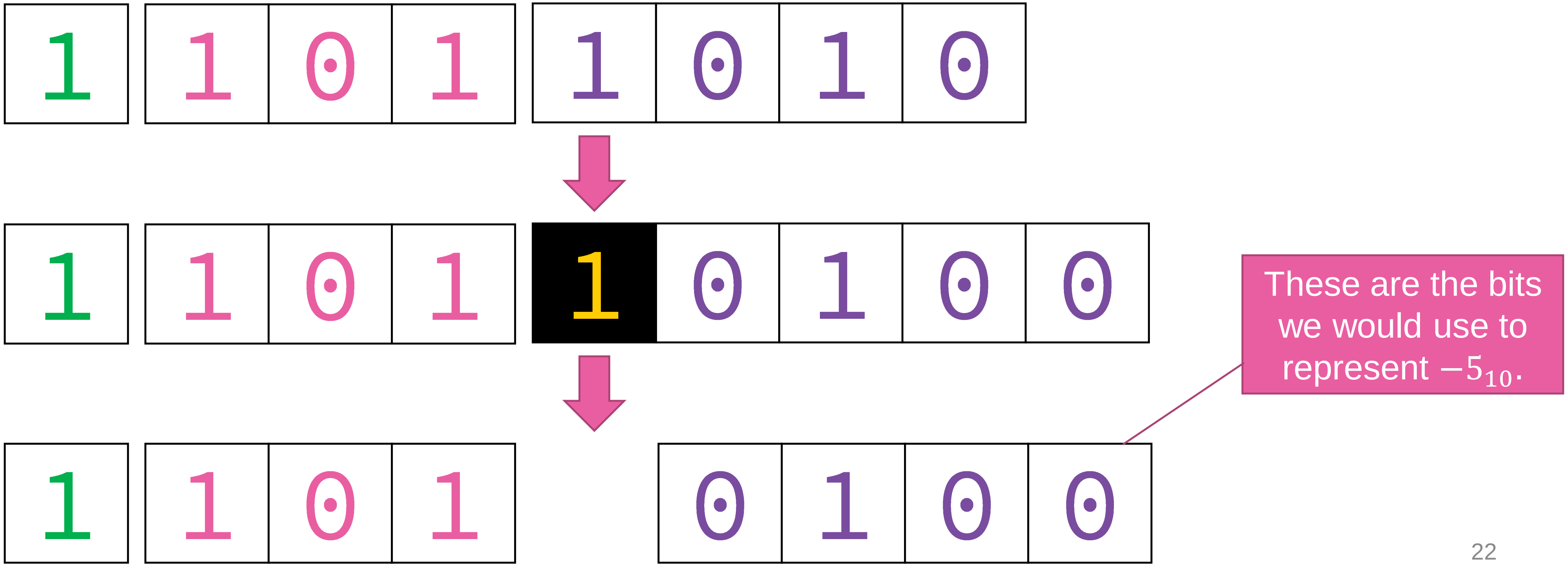
1	1	0	1	1	0	1	0

- Convert the second number to decimal.

Modified Proposal: Implicit Leading 1. in Significand

1 Sign, 3 Biased Exponent Digits, 4 Significand Digits

- What is the most significant digit in a binary floating point always going to be? ...
- **1**! So let's assume there's a leading 1 and get a ***free bit*** of precision!



Practice with the Modified Minifloat Proposal

$$2_{10}^{E-3} \times C_2$$

Implicit Leading 1. in Significand

1 Sign, 3x -3 Biased Exponent Digits, 4 Significand Digits

- What is the following minifloat bit pattern in decimal?

0	1	0	0	1	0	0	0

- How would you represent -4.25_{10} in minifloat?

But wait, how do you represent 0 in this proposal?!?

Implicit Leading 1. in Significand

1 Sign, 3x -3 Biased Exponent Digits, 4 Significand Digits

0	?	?	?	1	0	0	0	0

$$2_{10}^{E-3} \times C$$

You can't represent 0!?! Is there really no such thing as a free bit?

Back to the drawing boards...

What if we give up our *largest* and *smallest exponents* to be able to encode a few *special cases*?

- Old proposal: Exponent ranges from 2^{-3} to 2^{4}
- New proposal: Exponent ranges from 2^{-2} to 2^{3}
  - This frees up exponent bit patterns **000** and **111** for special cases!

- First special case: **000** in Exponent Field makes a **Denormalized Value**
  - This is a ***denormalized value*** and ***has an implicit leading 0.*** in its **significand**
  - Now we can represent 0! With: [`0 000 0000`]
  - The value of the exponent will be 1-bias (in our minifloat: 1-3: **-2**)
    - With the same minimum exponent but without the leading 1, we can represent values even closer to 0 than in normalized form.

Normalized vs. Denormalized Comparison

Normalized form:

0	0	0	1	0	0	0	1

You can tell this is **normalized** because not all 0's and not all 1's in **exponent** bits.

$$2_{10}^{1-3} \times 1.0001_2$$

$$0.010001_2$$

Notice denormalized has the same exponent as the smallest possible normalized exponent.

Denormalized form:

0	0	0	0	0	0	0	1

You can tell this is **denormalized** because all 0's in **exponent** bits.

$$2_{10}^{-2} \times 0.0001_2$$

$$0.000001_2$$

No leading 1 enables us to represent values closer to 0!

Other Special Cases

- Three special cases can be encoded when the exponent is all **1**'s:

```
1. +Infinity: [0 111 0000]
    • 0 sign bit, all 1s in exponent, all 0s in significand.

2. -Infinity: [1 111 0000]
    • 1 sign bit, all 1s in exponent, all 0s in significand.

3. NaN:        [0 111 0001] ... [1 111 1111]
```

- **Not a Number: When *any* bit in the significand is not a 0.**

IEEE 754 Floating Point

- **Floating point** standard established in 1985
  - Effectively all modern floating-point implementations use the standard!
- Floating point patterns are made of:

  1. **leading sign-bit**, followed by
  2. **biased exponent bits** followed by
  3. **significand bits** with an implicit leading **1.**

  - **Special cases** when **exponent** field is all
    - **0's - Denormalized** - implicit leading 0 in significand
    - **1's - Special values** - +/- infinity, NaN

- **Single-precision** Floating Point: **32-bits**
  - This is a **`float`** in C, Java, and so on

- **Double-precision** Floating Point: **64-bits**
  - This is a **`double`** in C, Java, and so on

32-bit float	
Exponent Bits	8
Bias	-127
Significand Bits	23

64-bit double	
Exponent Bits	11
Bias	-1023
Significand Bits	53

$$2^{E-B} \times C$$

Tools for Tinkering and Checking Understanding

- Best I've found: https://float.exposed/
- I would encourage challenging yourself to make conversions to and from *half-width* floating point precision:
- **1 Sign Bit**
- **5 Exponent Bits**, **-15 Bias**
- **10 Significand Bits**

What do you *need to know* about floating point?

- **You lose precision as your numbers grow away from 0**
  - **double**'s maximum value is 1.8×10^{308} - *next value is is* 2.0×10^{292} *away!*
  - **Takeaway**: If you are working with large numbers and precision matters:
    - Spend a lot more time on the numerical analysis of floating point appropriateness, or
    - Use an arbitrary precision arithmetic library (no loss in precision for loss in performance)

- **Many values cannot be represented without some round-off error:**
  - Examples: $^1/_3$, 0.1_{10}
  - This leads to surprising outcomes: 0.1 + 0.2 != 0.3
  - Takeaway: If you are using relational operators (==, !=, >, <) with floating point values you should use a method for determining if they're ***nearly equal****.*
    - Naive intuition: abs(a - b) < epsilon -- where epsilon is 0.0001 *this fails in many edge cases*!
    - If you're doing this, consult the internet and public documentation on best practices

- **Floating point arithmetic is not associative:**
  - (a + b) + c *does not always equal* a + (b + c)
  - When testing for *exact equality* this is nearly always a concern due to round-off
  - When testing for *near equality* this can be a concern if the exponents of a, b, and c are very different from one another