Posit Arithmetic
Lecture 1

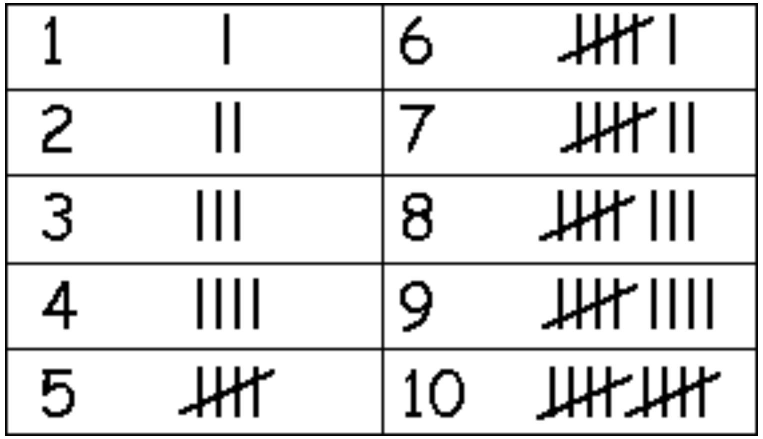
Prof. John L. Gustafson

A*STAR and National University of Singapore

26 March 2019

The Most Ancient of All Formats: *Unary*

Repeated symbols for counting numbers 1, 2, 3,…



Regime Notation: Signed Unary, by Using *Bits*

000001	**–5**
00001	**–4**
0001	**–3**
001	**–2**
01	**–1**
10	**0**
110	**1**
1110	**2**
11110	**3**
111110	**4**

A run of *runlength* identical *r* bits is terminated by the opposite bit.

Represents an integer *k* which we will need later, so please remember this part!

$k = -\mathit{runlength}$ if r is 0,

$k = \mathit{runlength} - 1$ if r is 1.

Numbers near 0 require very few bits.

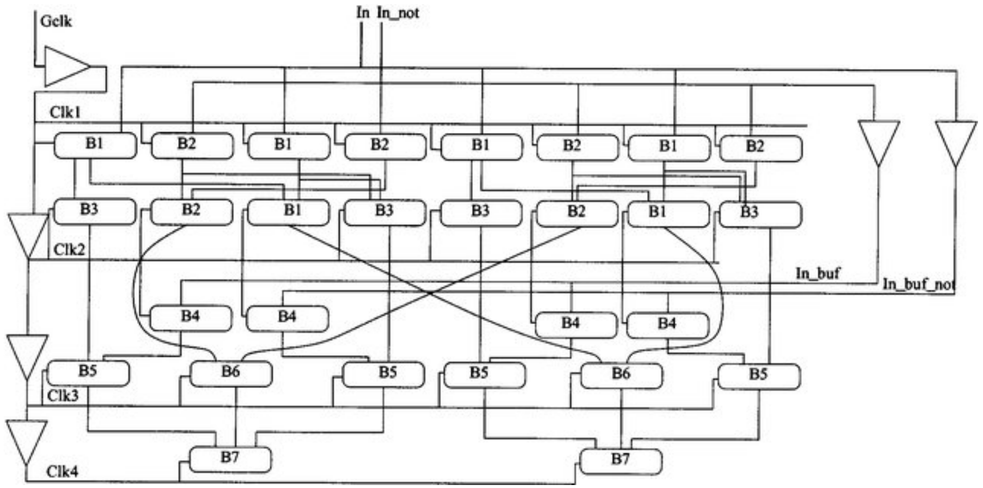
Circuits exist for decoding regime bits

000001	−5
00001	−4
0001	−3
001	−2
01	−1
10	0
110	1
1110	2
11110	3
111110	4

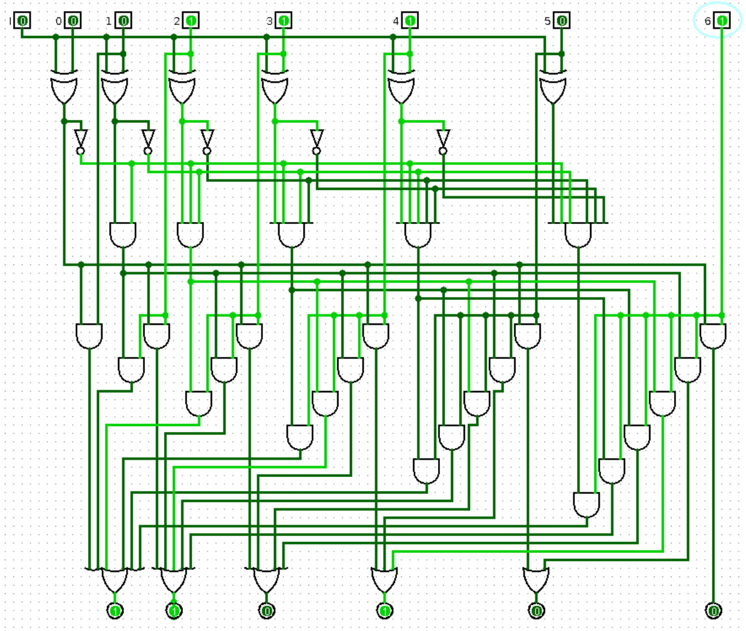
$$\begin{array}{r} 1.0005 \\ \underline{-1.0003} \\ 0.0002 \end{array}$$

$\Rightarrow 2\times10^{-4}$

The CLZ (count leading zeros) instruction is already part of standard floating-point circuits.

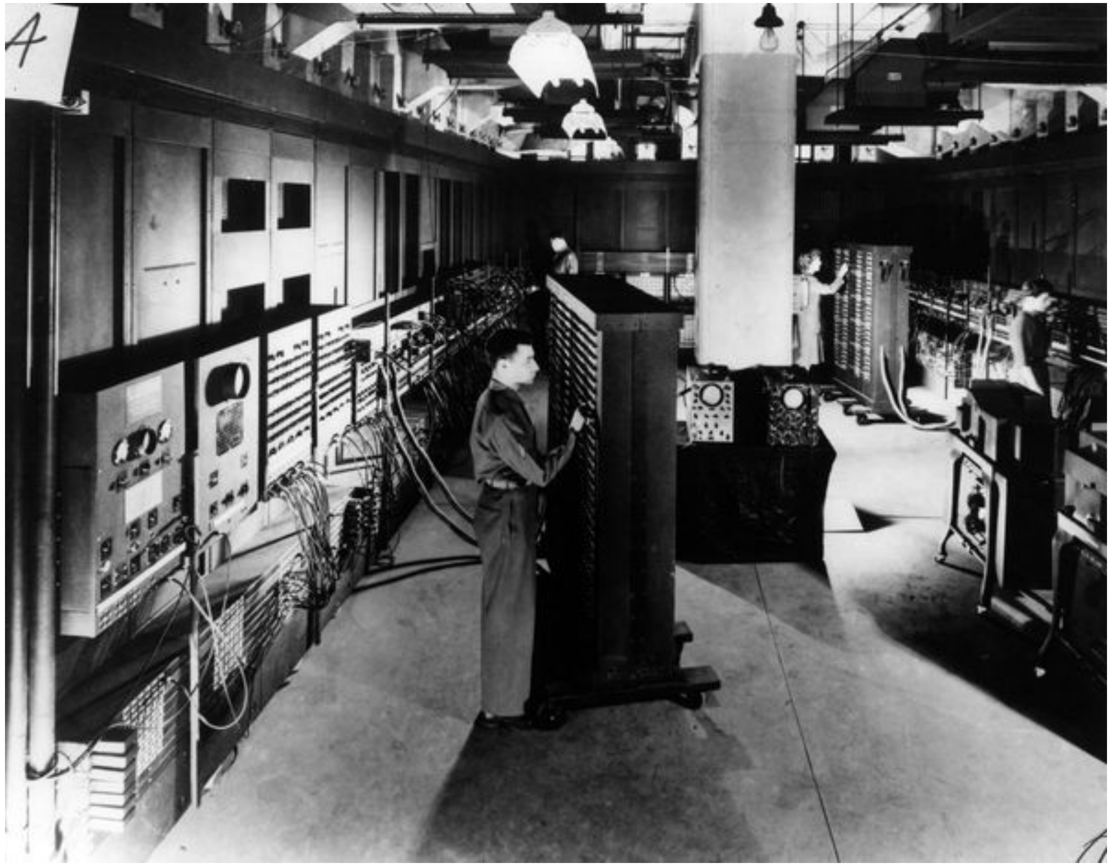


CLZ logic (Wikipedia)



Regime shifter (I. Yonemoto)

Early computers used *decimal* internally!



ENIAC, 1946

This is an example of the mistake of imposing human tastes on hardware design.

144

0001 0100 0100

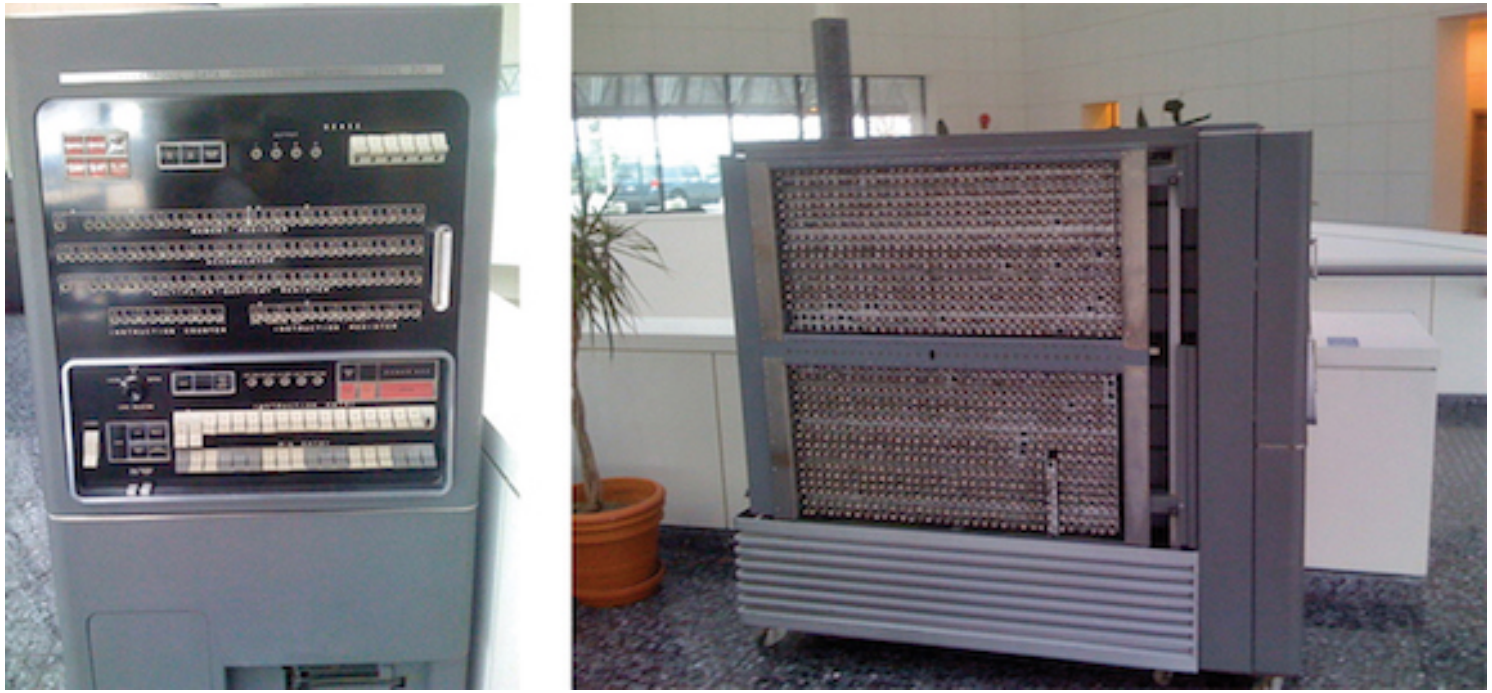
Versus native binary, 8 bits:

10010000

$2^7 + 2^4 = 144$

Decimal computers took about 7 times as many gates as binary computers.

History’s Next Mistake: A Separate “Sign Bit”



IBM 701, 1953

Negative integers were originally stored in “sign-magnitude” form, imitating the way humans write + and – before digit strings.

+5 –5

`0 101` `1 101`

BAD idea. Why? Well, here’s one reason:

+0 –0

`0 000` `1 000`

Welcome to the joys of “negative zero.”

Remember adding signed numbers in school?

To add nonzero signed integers *m* and *n*:

Are they the same sign or different sign?

    If they are the same sign, add their magnitudes.

        Apply that sign to the resulting sum, **DONE**.

    Else if they have different signs,

        Find out which magnitude is bigger.

        If *m* has bigger magnitude,

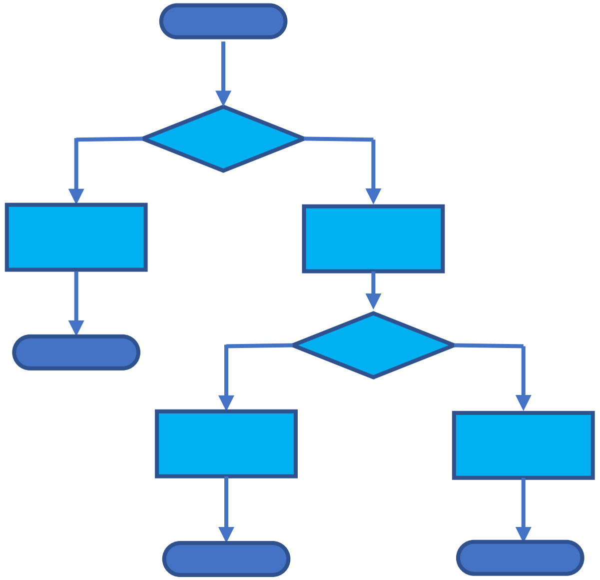
            Subtract *n*’s magnitude from *m*’s magnitude.

            Apply *m*’s sign to the result. **DONE**.

        Else

            Subtract *m*’s magnitude from *n*’s magnitude.

            Apply *n*’s sign to the result. **DONE**.



Modern signed integers are *2’s complement*

0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	–8	wraparound
1001	–7	
1010	–6	
1011	–5	
1100	–4	
1101	–3	
1110	–2	
1111	–1	

Not as human friendly…

+5 → 0 101

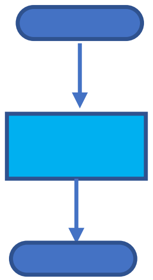
–5 → 1 011

??

(*flip the bits and add 1*)

…but mathematically and computationally *far* better. Here’s the addition algorithm for 2’s complement:

To add signed integers *m* and *n*:
Add like unsigned integers. **DONE**

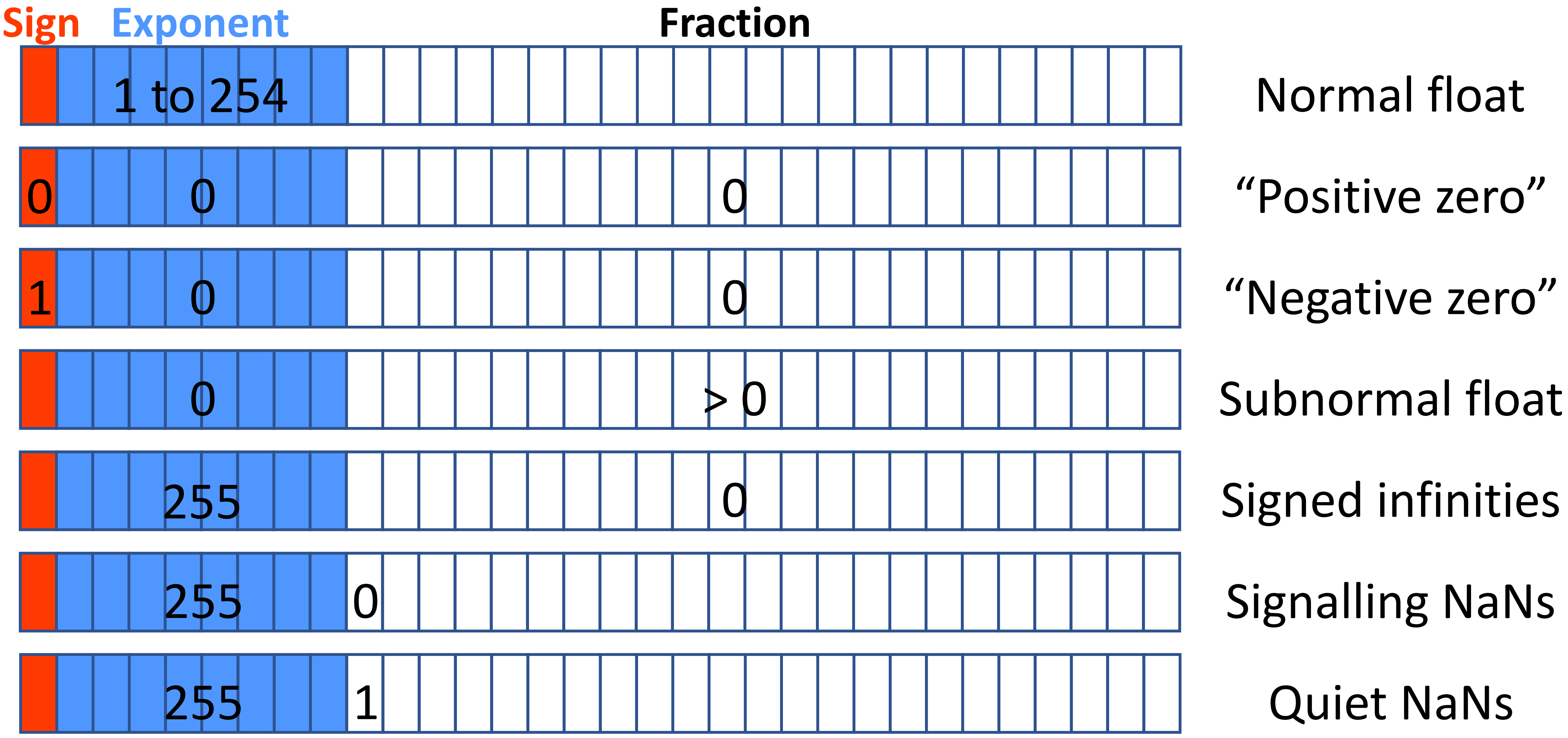


The fundamental idea of floating point format

Express values as $m \times 2^j$, where m and j are integers.
Use a fixed set of bits for m and a fixed set for j.

- Notice we need both negative and positive m and j.
- Before the IEEE 754 Standard (1985), there were **many** different schemes for where the bits go in a word, and how to interpret the bits as signed integers.
- Larger dynamic range (more bits for j) means less accuracy (fewer bits for m). And vice versa.
- Note: in practice, the most common numbers have j near zero.

The IEEE 754 Standard greatly complicated the fundamental idea of a floating point format



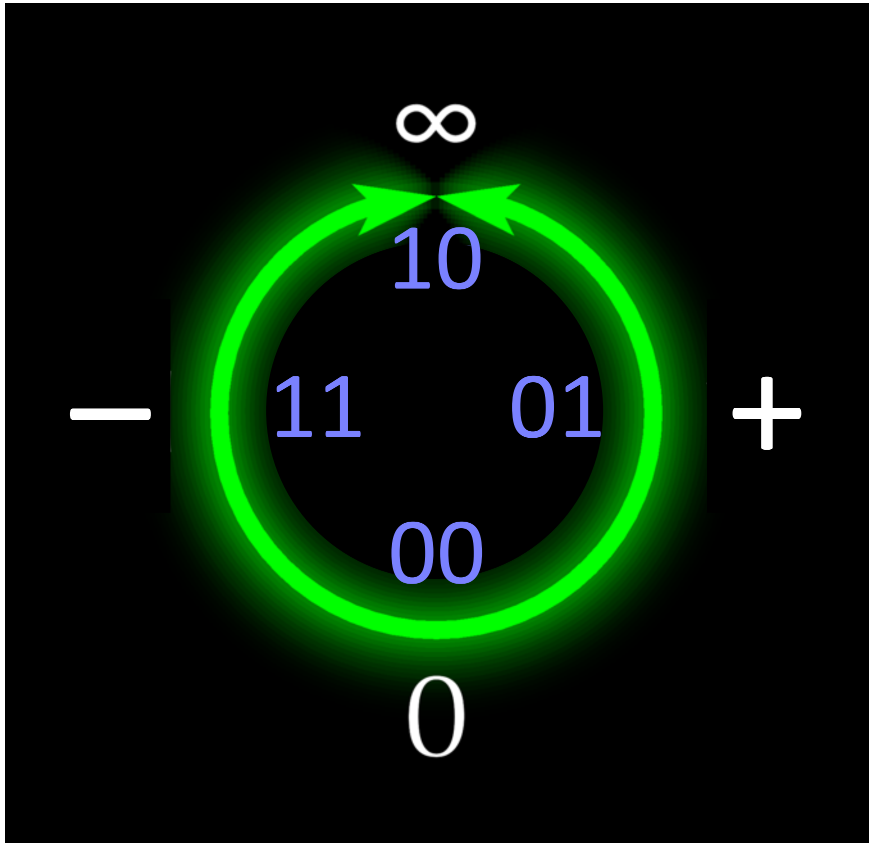
The fundamental idea of *posit* format

Express values as $m \times 2^j$, where m and j are integers.
Use *regime format* for j, so a small j **takes fewer bits.**

- Creates 1-to-1 map of reals to integers, monotone
- The j is represented as the sum of two integers:
  - power-of-2 exponent e
    (unsigned integer ranging from 0 to $2^{es}-1$)
  - a "regime exponent" k, the power of *useed* = $2^{2^{es}}$.
  - Put more simply: the 2^j scaling is $2^{k2^{es}+e}$.

Some Prefer the Geometric Explanation

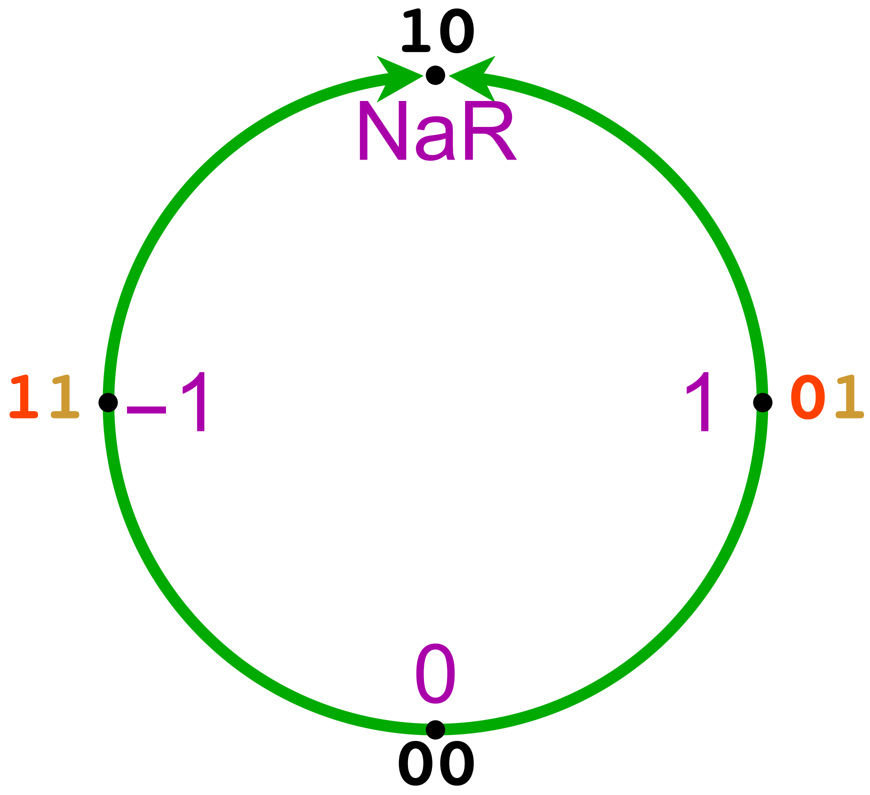
(This is how posits were invented)



Instead of the real number line, the *projective reals* put "the point at infinity" at the top of the circle… the first step to making an infinite line map to a finite-state computer format.

Big positives wrap to big negatives, just like (2's complement) signed integers.

Subtle change: Treat ∞ the way floats treat "NaN"

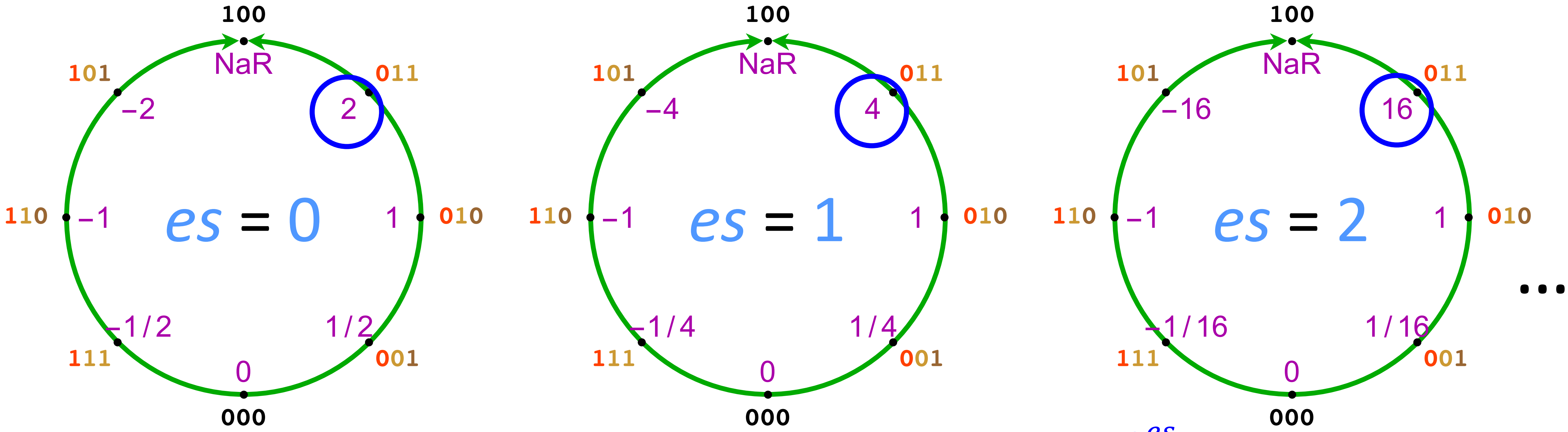


Not a Real (NaR) is the catch-all for $\sqrt{-1}$, 0/0, arcsin(3), etc.

Unlike ±∞, NaR always propagates, so 1/NaR = NaR, not 0.

Why "NaR" and not "NaN" (Not a Number"? Because imaginary and complex numbers are **numbers**! They're just not *real* numbers.

The *es* value controls the dynamic range.

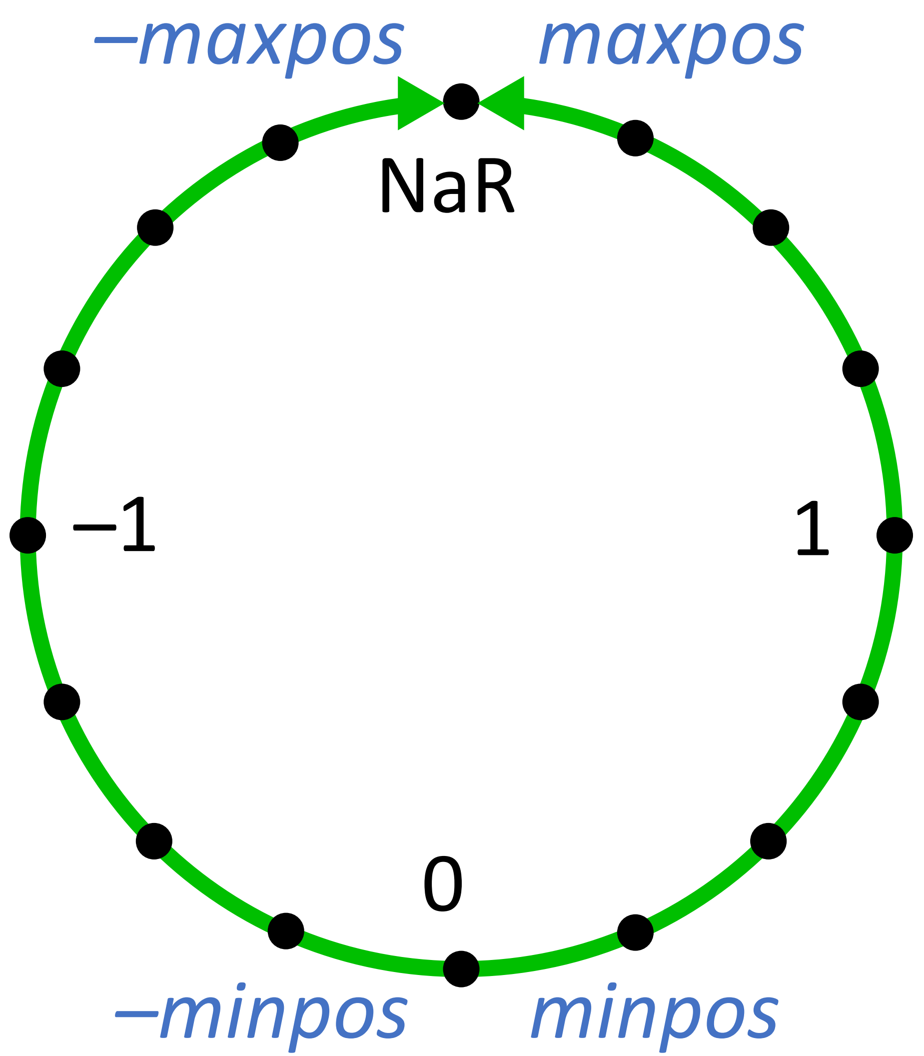


$$useed = 2^{2^{es}}$$

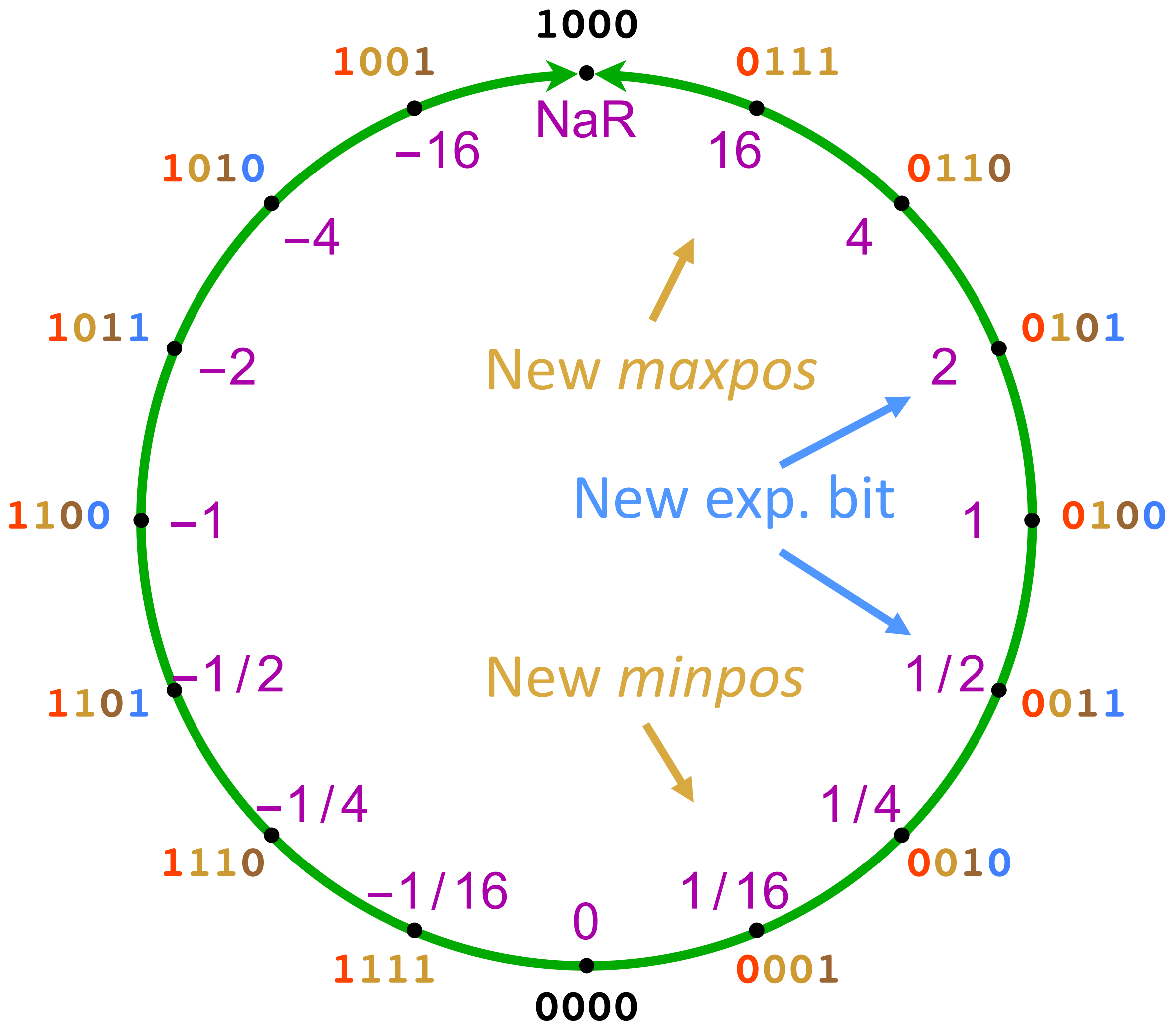
The northeast value is called the *useed.*
Start with *useed* = 2, then do *useed* ← *useed*^{2}, *es* times.

Instead of overflow and underflow

- *maxpos* is the largest magnitude positive posit value.
- *minpos* is the smallest magnitude positive posit value.
- *–maxpos* is the largest magnitude negative posit value.
- *–minpos* is the smallest magnitude negative posit value.



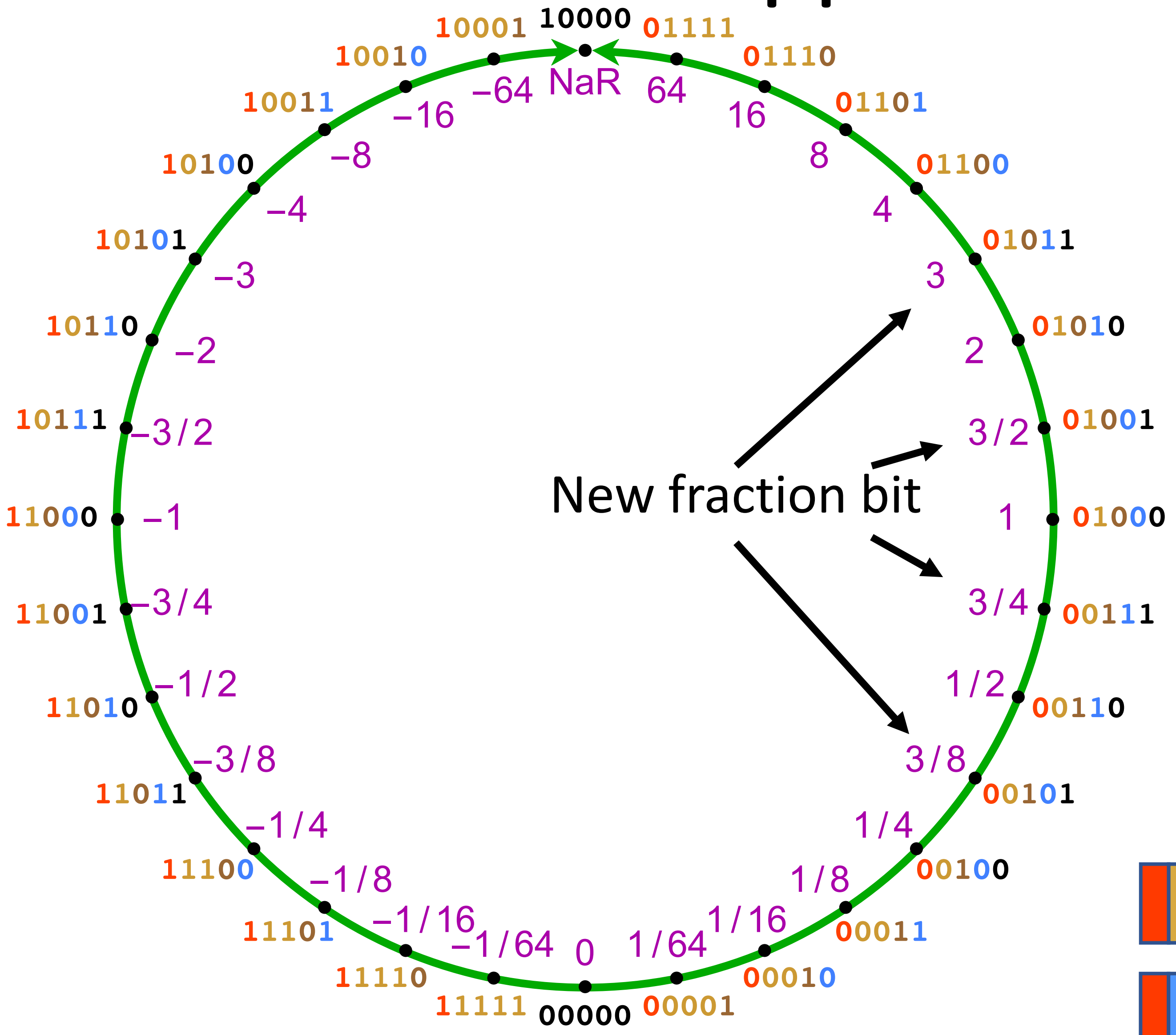
Pick *es* = 1 and append another bit.



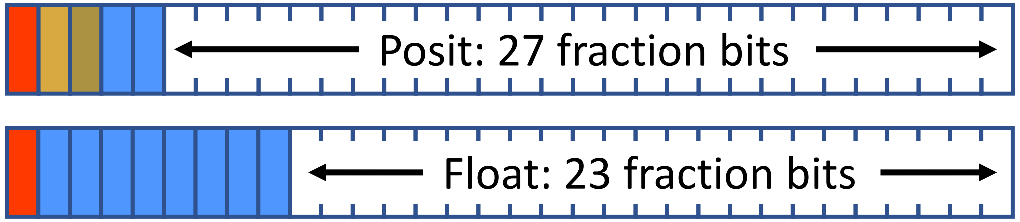
If you can squeeze another integer power of 2 between two adjacent points of the previous, do so. The appended bit is an exponent bit.

The new *maxpos* is the old *maxpos* times *useed*. The new *minpos* is the old *minpos* divided by *useed*. The appended bit is a regime bit.

Fraction bits appear



- If a new intermediate value cannot be an integer power of two, use the average. The appended bit is a **fraction bit**.
- Unusual property: Posits increase both dynamic range **and** decimals of accuracy by adding bits on the right.
- Notice that **half** of all posits use only *two* regime bits!



Making posits look more like IEEE floats:

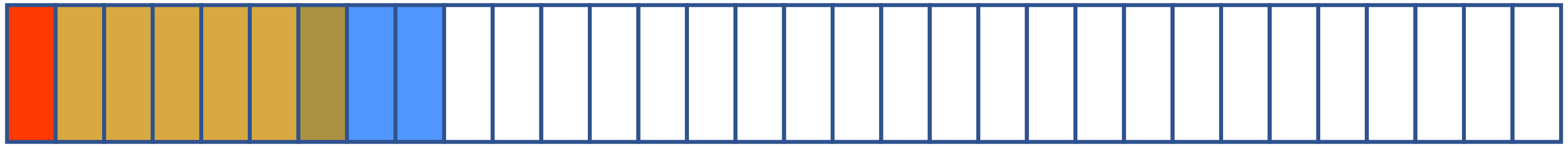
Total posit size is *ps* bits

Only two Exceptions:

Zero: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Not a Real (NaR): | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Else, 2's complement sign, with accuracy that tapers automatically:



Exponent size is *es* bits

Fraction size is *fs* bits

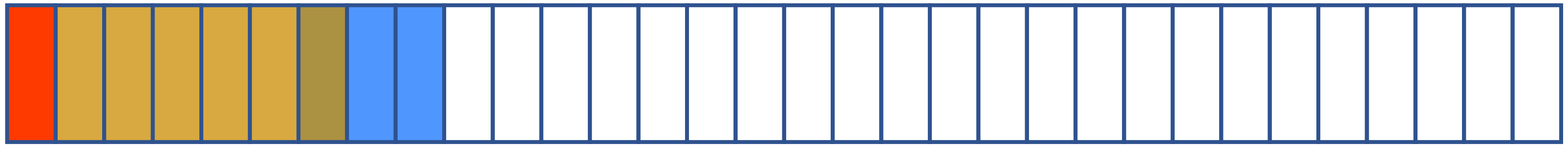
Sign bit ***s*****.**
0 for positive,
1 for negative.

"Regime" bits
Signed unary integer representing k, the power of $2^{2^{es}}$.

Exponent bits ***e***
unsigned integer. Size is
$es = 0,\ 1,\ 2, \ldots$ for
$ps = 8, 16, 32, \ldots$

Fraction bits ***f***
Fraction in positional notation. "Hidden bit" is **always** 1.

Human decoding or hardware decoding?



Easier to understand:

Check for 0 and NaR case first.
Else record s. If s = **1**, find magnitude by 2's complement.
Find k, e, and f of the magnitude.
Posit represents the number $(-1)^s \cdot useed^k \cdot 2^e \cdot (1 + f/2^{fs})$.

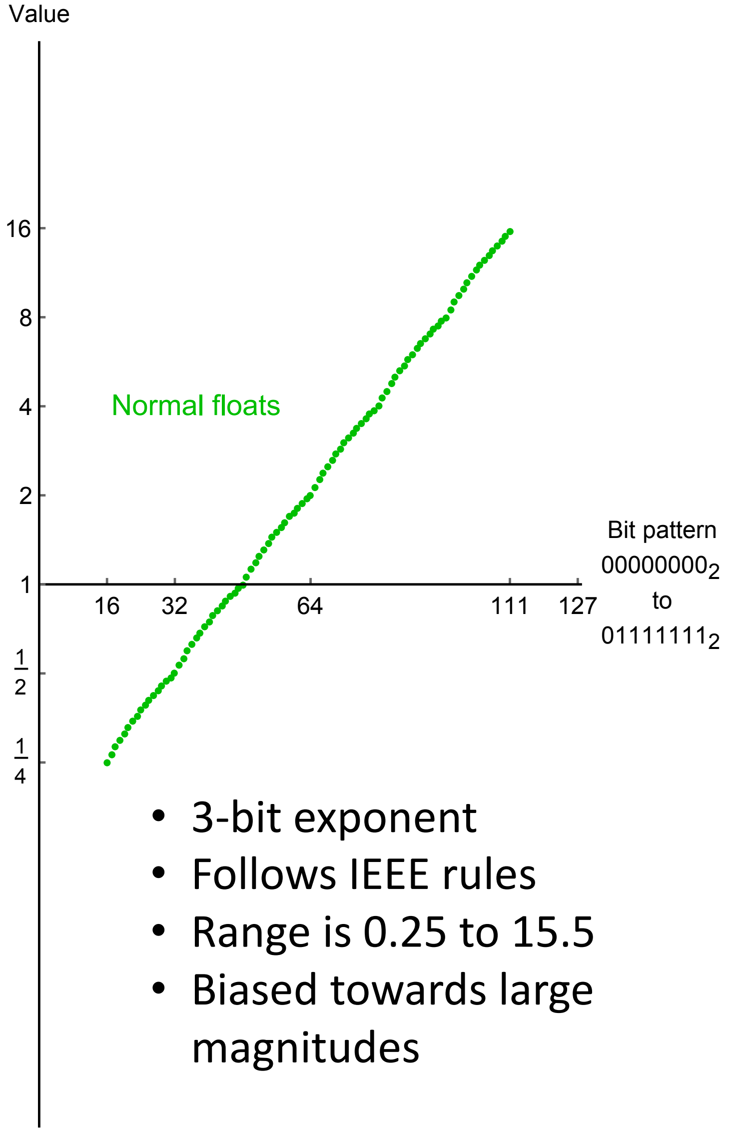
Better for circuit design, but cryptic:

Check for 0 and NaR cases while finding s, k, e, f concurrently.

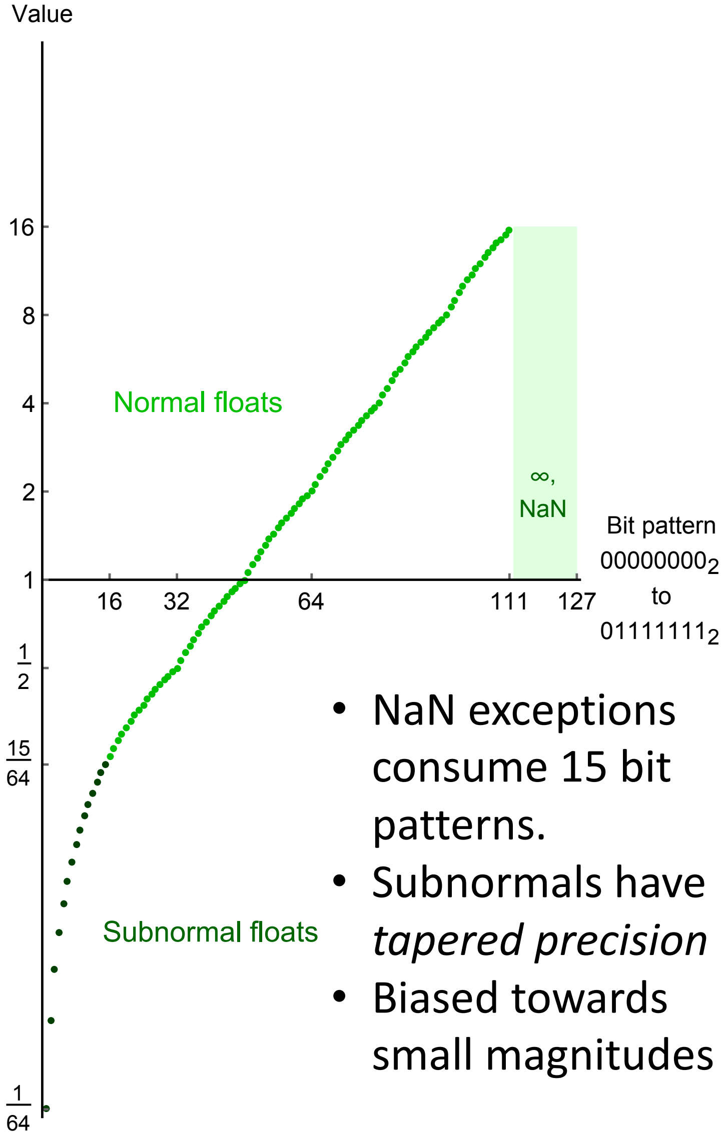
Posit represents $(1 - 3s + f) \cdot 2^{(-1)^s(k \cdot 2^{es} + e + s)}$

From 1 to 2 if s = **0**
From –2 to –1 if s = **1**

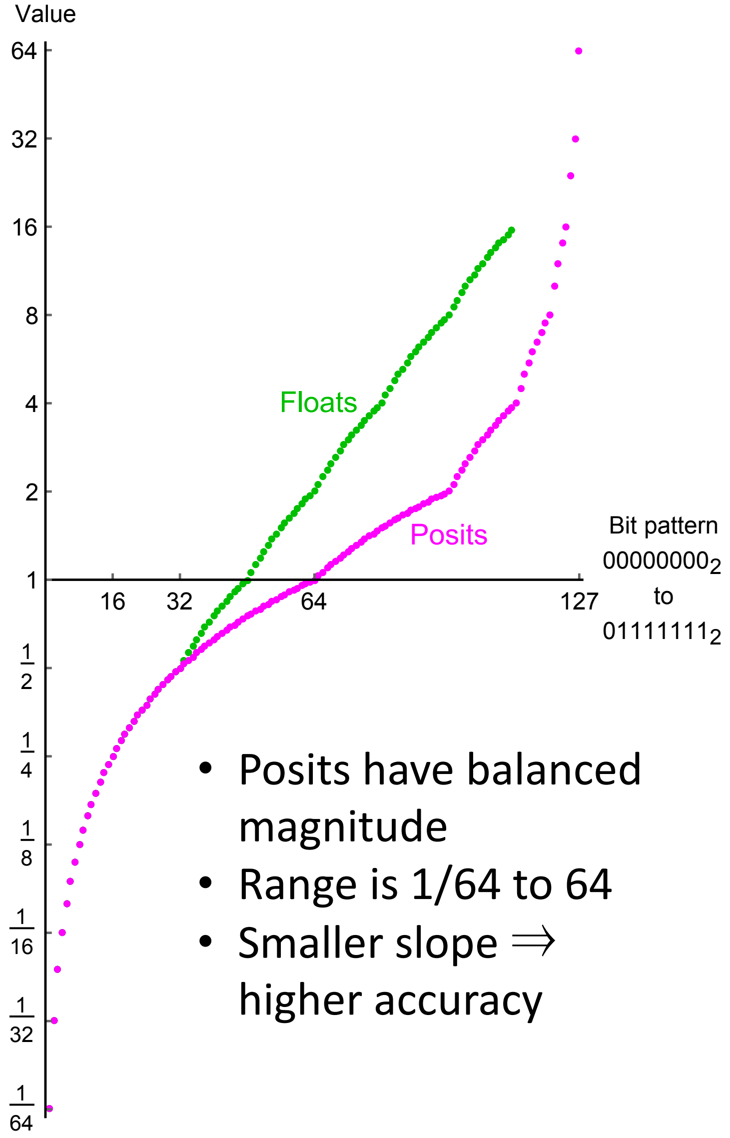
Visualization of 8-bit floats vs 8-bit posits



- 3-bit exponent
- Follows IEEE rules
- Range is 0.25 to 15.5
- Biased towards large magnitudes



- NaN exceptions consume 15 bit patterns.
- Subnormals have *tapered precision*
- Biased towards small magnitudes



- Posits have balanced magnitude
- Range is 1/64 to 64
- Smaller slope ⇒ higher accuracy

End of
Posit Arithmetic Introduction