Beating Floating Point at its Own Game: Posit Arithmetic

John L. Gustafson
Professor, A*STAR and
National University of Singapore

Agency for Science, Technology and Research
SINGAPORE

Supercomputing Frontiers 2017

Why worry about floating point?

Find the scalar product $a \cdot b$:

$$a = (3.2e8,\ 1,\ -1,\ \ 8.0e7)$$
$$b = (4.0e7,\ 1,\ -1,\ -1.6e8)$$

Note: All values are integers that can be expressed *exactly* in the IEEE 754 Standard floating-point format (single or double precision)

Single Precision, 32 bits: $a \cdot b = 0$

Double Precision, 64 bits: $a \cdot b = 0$

Correct answer: $a \cdot b = 2$

Most linear algebra is unstable with floats!

What's wrong with IEEE 754? A start:

- **No guarantee of identical results across systems**
- It's a *guideline*, not a *standard*
- Breaks the laws of algebra:

$$a + (b + c) \neq (a + b) + c \qquad a \cdot (b + c) \neq a \cdot b + a \cdot c$$

- Overflow to infinity, or underflow to zero, create an infinite *loss of accuracy.*

IEEE floats are weapons of math destruction.

What *else* is wrong with IEEE 754?

- Exponents usually take too many bits
- Accuracy is flat across a vast range, then falls off a cliff
- Subnormal numbers are a headache (“gradual underflow”)
- Divides are messy and slow
- Wasted bit patterns: “negative zero,” too many NaN values

Do we really need 9,007,199,254,740,990 ways to say something is *Not a Number*??

Contrasting Calculation "Esthetics"

	Rounded: cheap, uncertain, "good enough"	**Rigorous: more work, certain, mathematical**
IEEE Standard (1985)	Floats, $f = n \times 2^m$ m, n are integers	Intervals $[f_1, f_2]$, all x such that $f_1 \le x \le f_2$
Type 1 Unums (2013)	"Guess" mode, flexible word size	Unums, ubounds, sets of uboxes
Type 2 Unums (2016)	"Guess" mode, fixed word size	Sets of Real Numbers (SORNs)
Type 3 Unums (2017)	***Posits***	***Valids***

If you ***mix*** the two esthetics, you end up satisfying ***neither***.

posit | ˈpäzət |

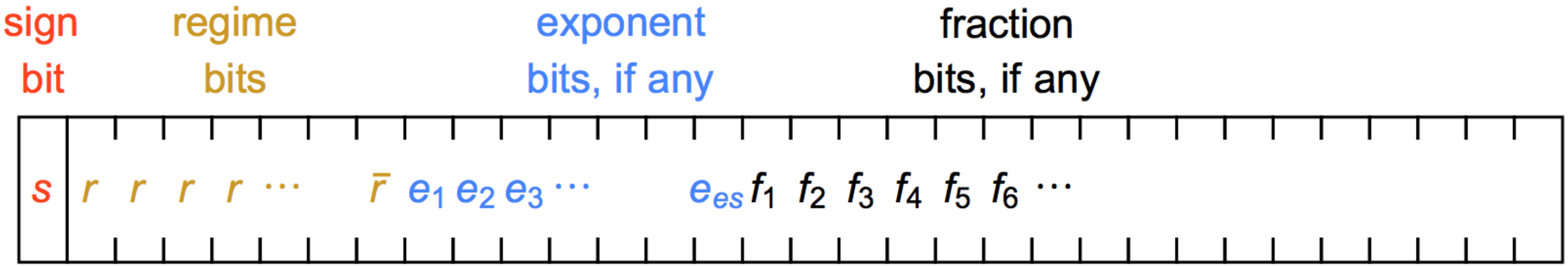
noun *Philosophy*

a statement that is made on the assumption that it will prove to be true.

Metrics for Number Systems

- Decimals of Accuracy $-\log_{10}(\log_{10}(x_j / x_{j+1}))$
- Dynamic range $\log_{10}(maxreal / minreal)$
- Percentage of operations that are *exact* (closure under + – × ÷ √ etc.)
- Average accuracy loss when *inexact*
- Entropy per bit (maximize information)
- Accuracy benchmarks: simple formulas, linear equation solving, math kernels…

Posit Arithmetic:
Beating floats at their own game



Fixed size, *nbits*.

Note: The "ubit" is only for the ***valid*** type;
posits round, if necessary.

es = exponent size = 0, 1, 2,… bits.

Posit Arithmetic Example

sign bit	regime bits	exponent bits	fraction bits
0	0 0 0 1	1 0 1	1 1 0 1 1 1 0 1
$+$	256^{-3}	$\times\ 2^{5}$	$\times\ (1 + 221/256)$

$$= 3.55\cdots\times 10^{-6}$$

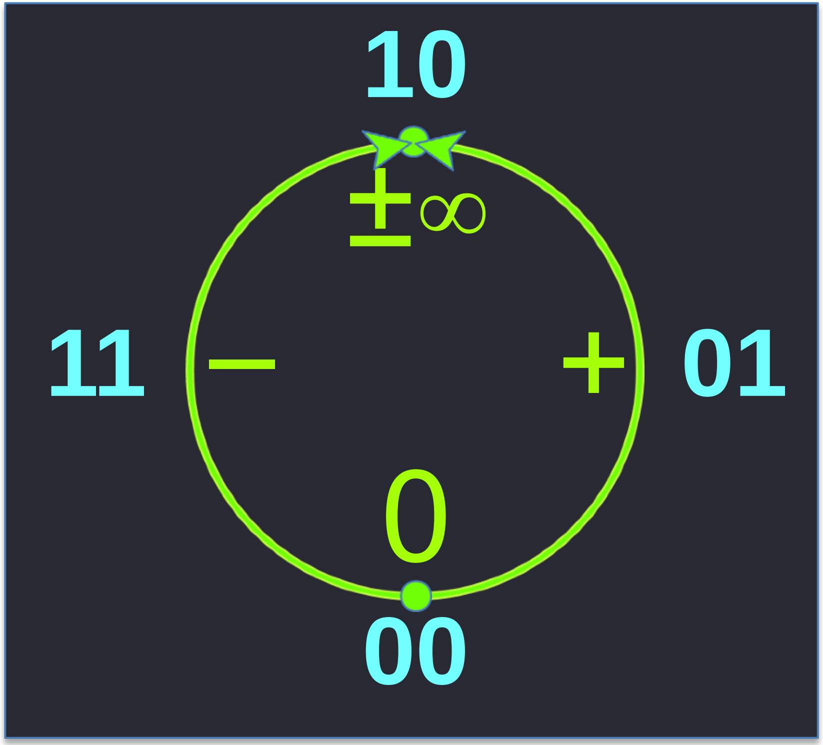
Here, *es* = 3. Float-like circuitry is all that is needed (integer add, integer multiply, shifts to scale by 2^k)

Posits **do not underflow or overflow**. There is no NaN.

Simpler, smaller, faster circuits than IEEE 754

Posits use the Projective Reals

- Like Type 2 unums, posits map reals to standard *signed integers*.
- This eliminates "negative zero" and other IEEE float issues



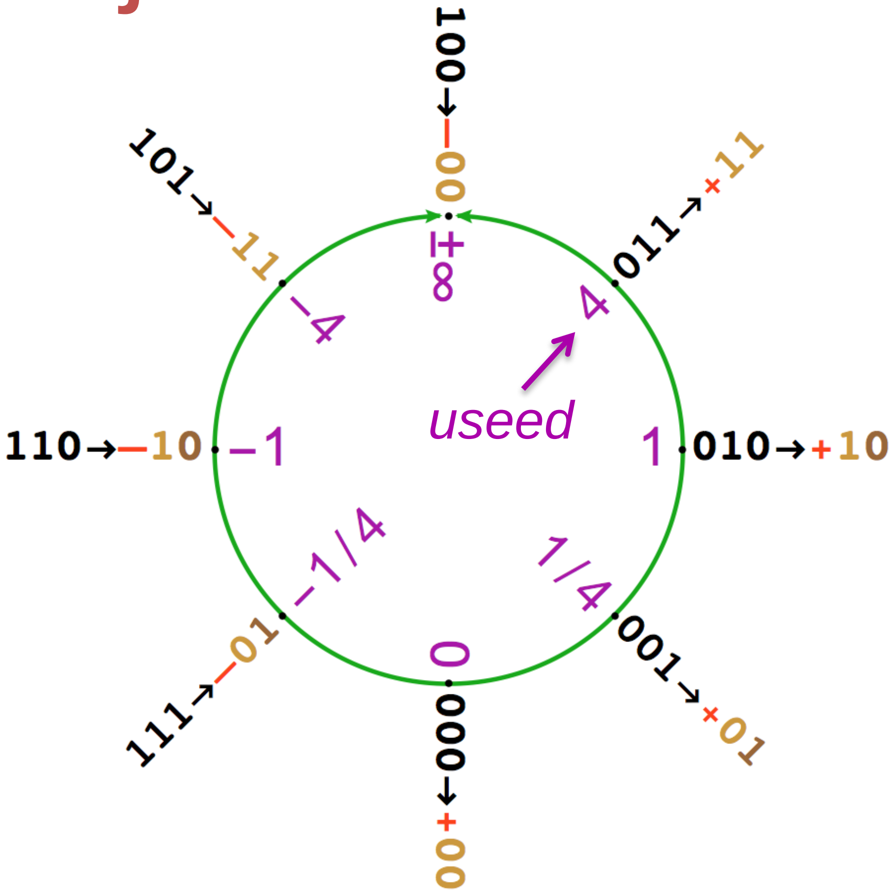
Mapping to the Projective Reals

Example with
nbits = 3, *es* = 1.

Value at 45° is always

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

If bit string < 0, set sign to –
and negate integer.

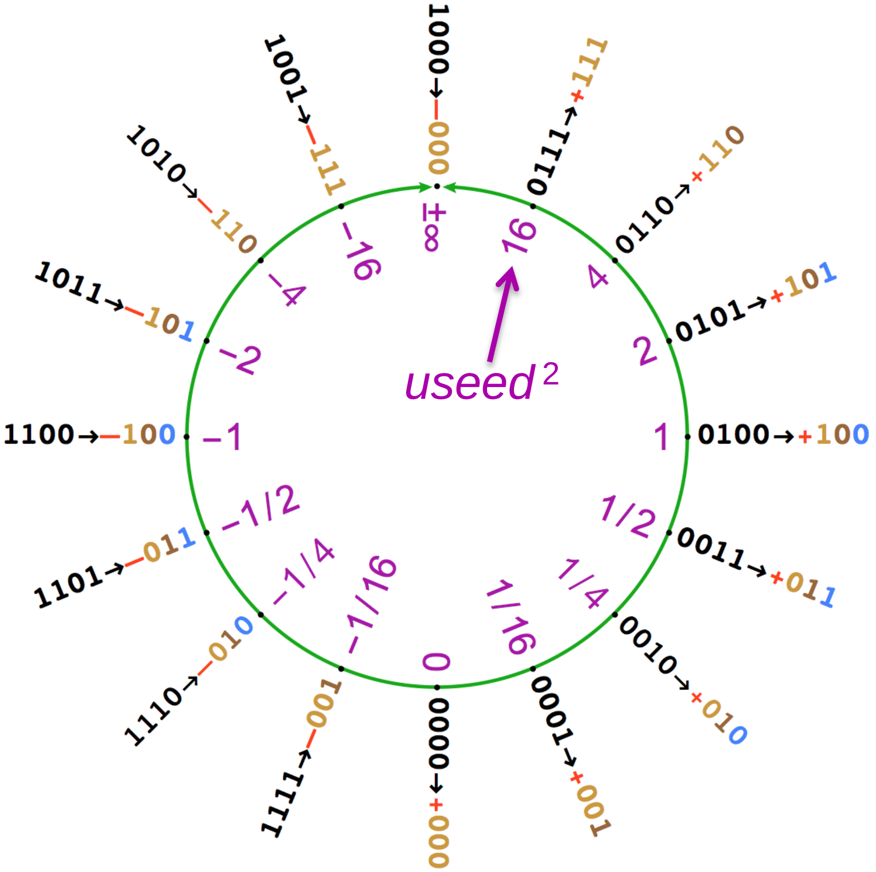


Rules for inserting new points

Between ±*maxpos* and ±∞,
scale up by *useed*.
(New **regime** bit)

Between 0 and ±*minpos*,
scale down by *useed.*
(New **regime** bit)

Between 2^m and 2^n where
$n - m \geq 2$, insert $2^{(m+n)/2}$.
(New **exponent** bit)

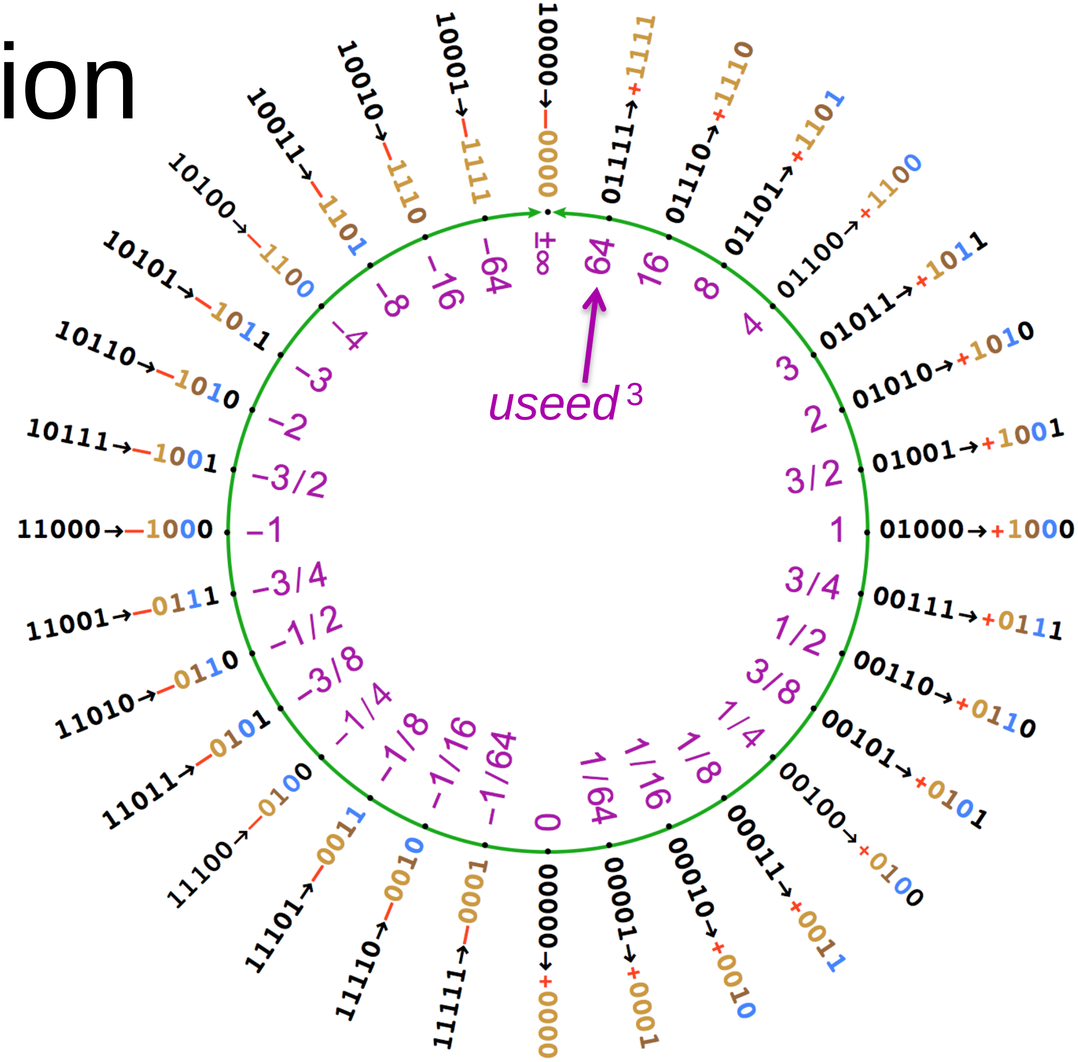


At *nbits* = 5, fraction bits appear.

Between x and y where $y \le 2x$, insert $(x + y)/2$.

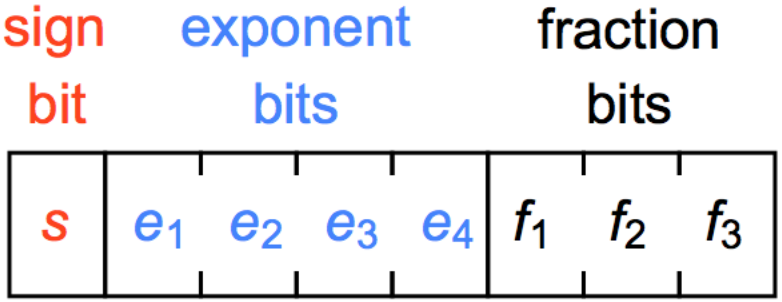
Existing values stay put as *trailing* bits are added.

Appending bits increases ***accuracy*** east and west, ***dynamic range*** north and south!



Posits v. Floats: a *metrics-based* study

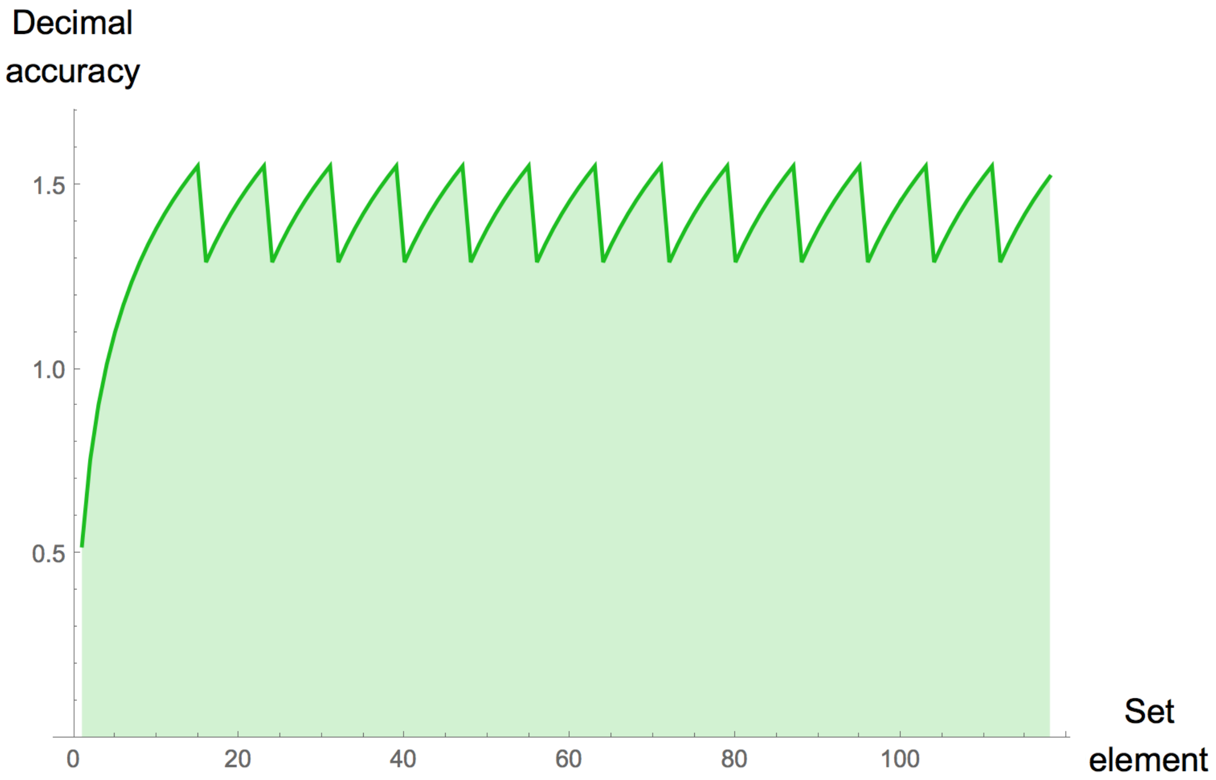
- Compare *quarter-precision* IEEE-style floats
- Sign bit, 4 exponent bits, 3 fraction bits



- *smallsubnormal* = 1/512; *maxfloat* = 240.
- Dynamic range of five orders of magnitude
- Two bit patterns that mean zero
- Fourteen bit patterns that mean “Not a Number” (NaN)

Float accuracy tapers only on left

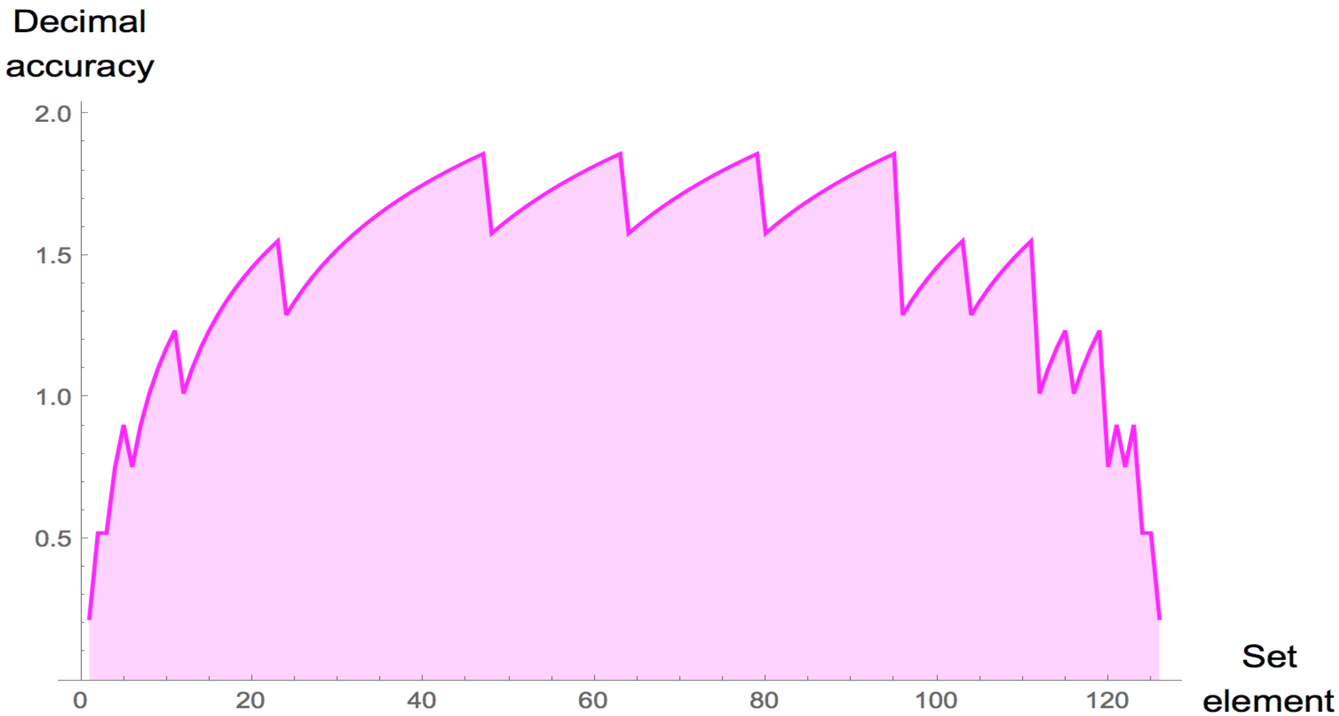
- Min: **0.52** decimals
- Avg: **1.40** decimals
- Max: **1.55** decimals



Graph shows decimals of accuracy from *minfloat* to *maxfloat*.

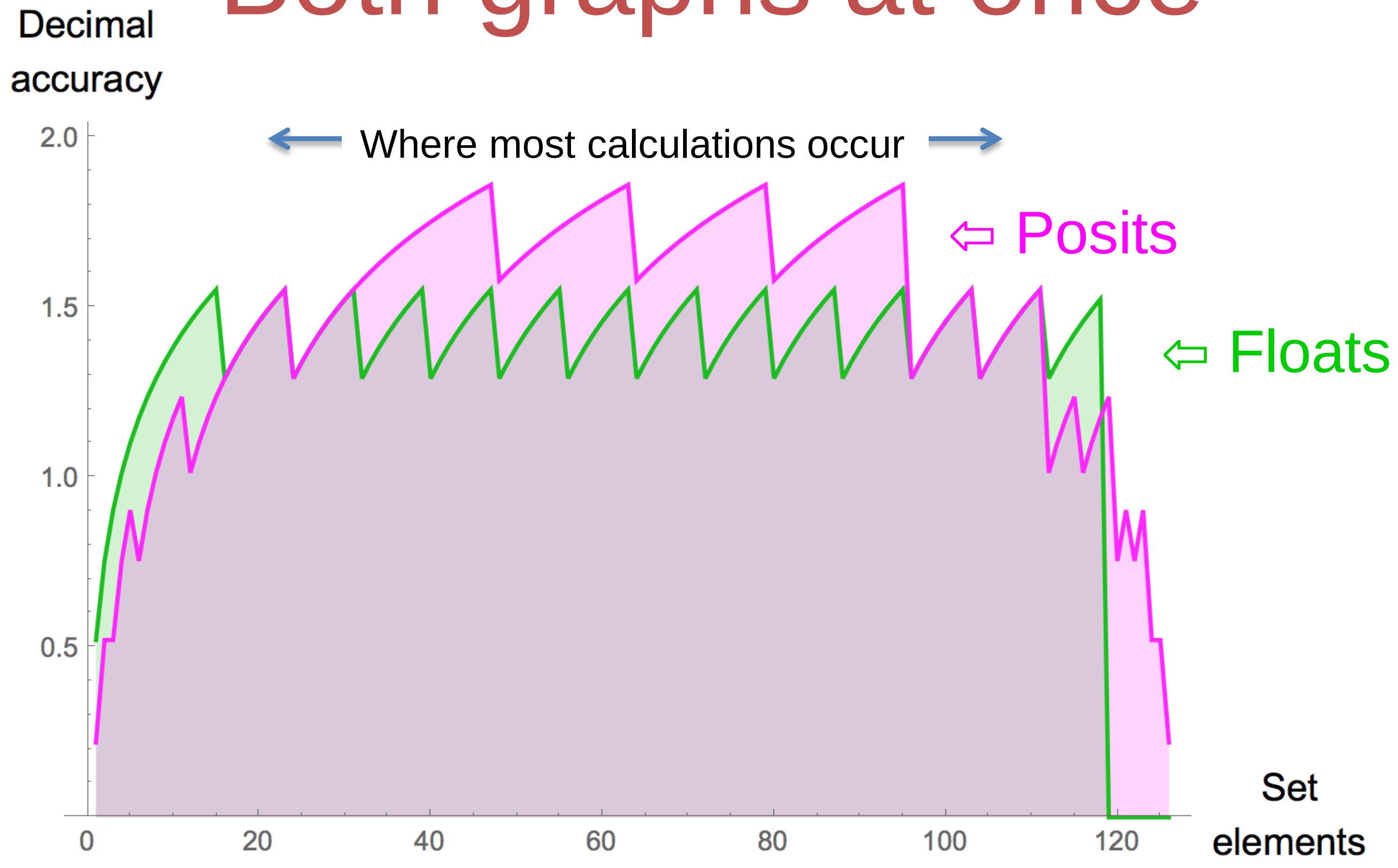
Posit accuracy tapers on both sides

- Min: **0.22** decimals
- Avg: **1.46** decimals
- Max: **1.86** decimals



Graph shows decimals of accuracy from *minpos* to *maxpos*.
But posits cover *seven* orders of magnitude, not five.

Both graphs at once



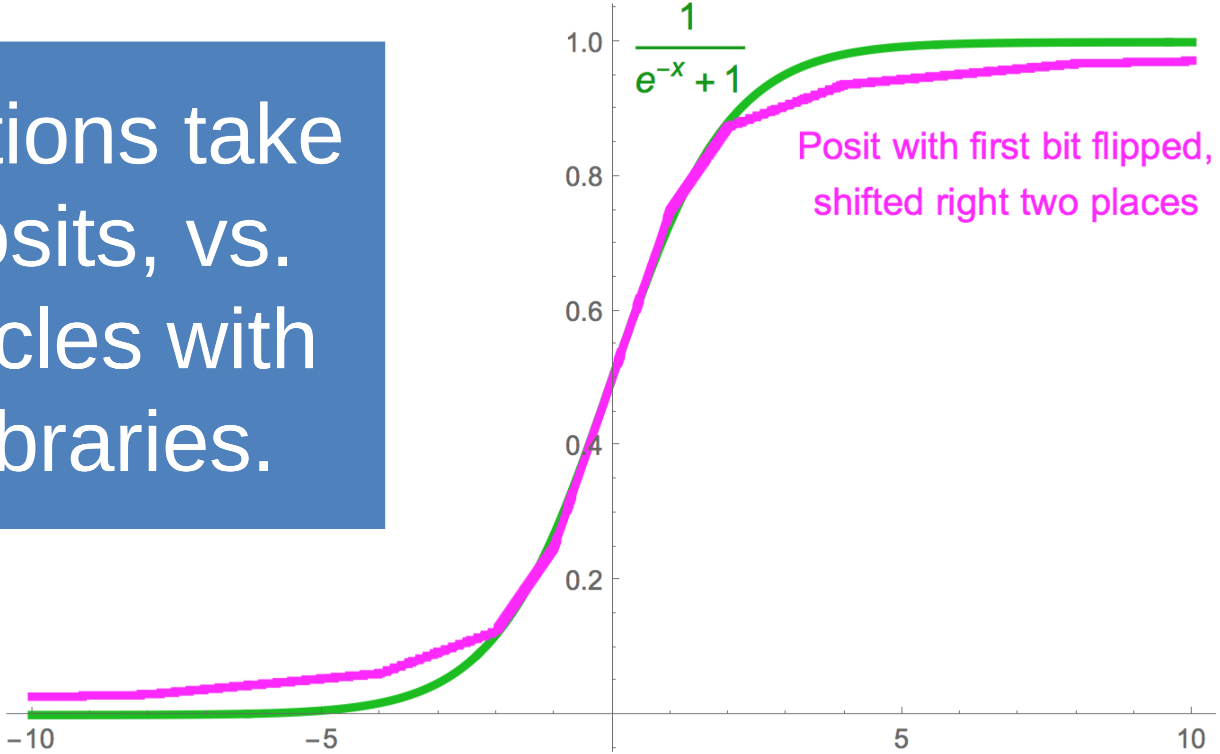
Matching float dynamic ranges

Size, bits	Float exponent size	Float dynamic range	Posit es value	Posit dynamic range
16	5	6×10^{-8} to 7×10^{4}	1	4×10^{-9} to 3×10^{8}
32	8	1×10^{-45} to 3×10^{38}	3	6×10^{-73} to 2×10^{72}
64	11	5×10^{-324} to 2×10^{308}	4	2×10^{-299} to 4×10^{298}
128	15	6×10^{-4966} to 1×10^{4932}	7	1×10^{-4855} to 1×10^{4855}
256	19	2×10^{-78984} to 2×10^{78913}	10	2×10^{-78297} to 5×10^{78296}

Note: Isaac Yonemoto has shown that 8-bit posits suffice for neural networks, with *es* = 0

8-bit posits speed *neural nets*

Sigmoid functions take 1 cycle in posits, vs. dozens of cycles with float math libraries.



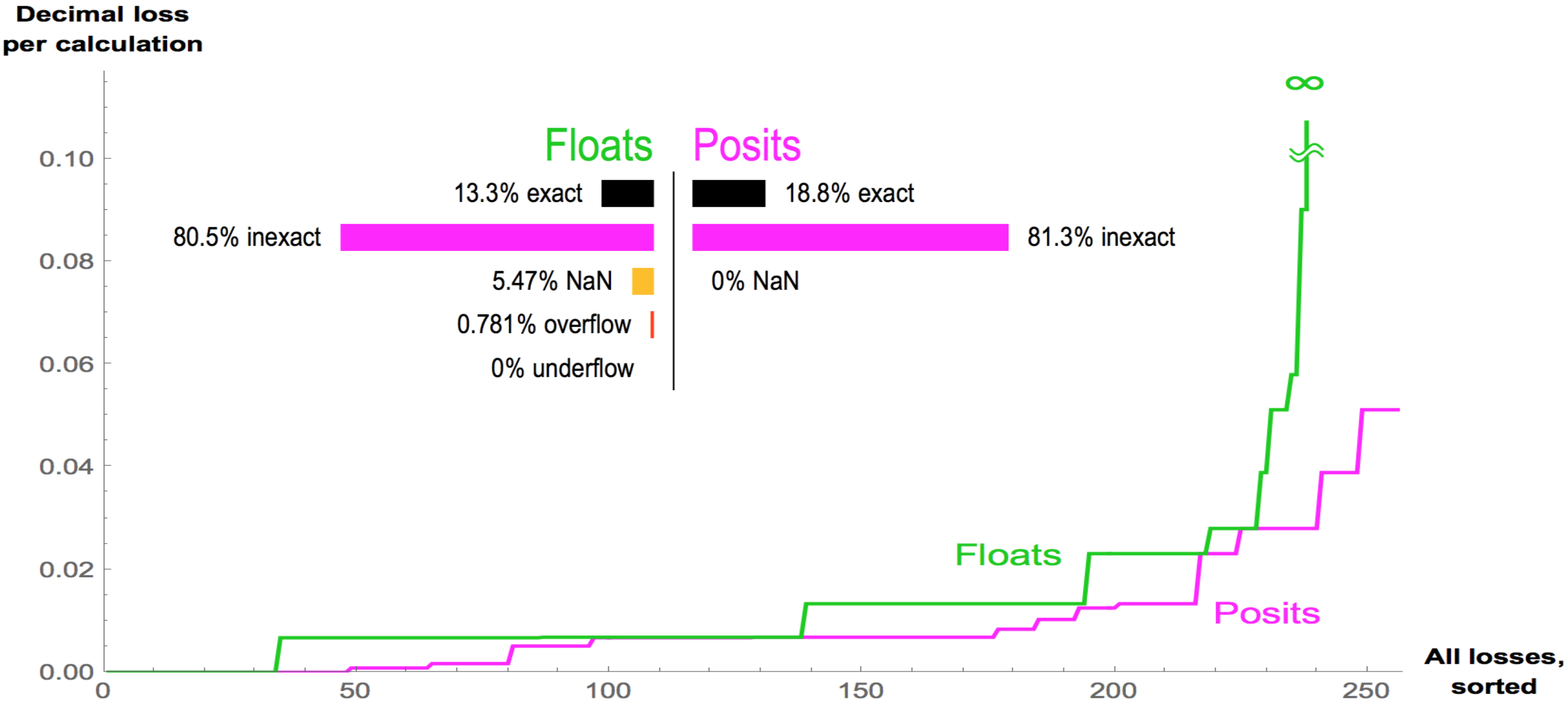
(Observation by I. Yonemoto)

ROUND 1

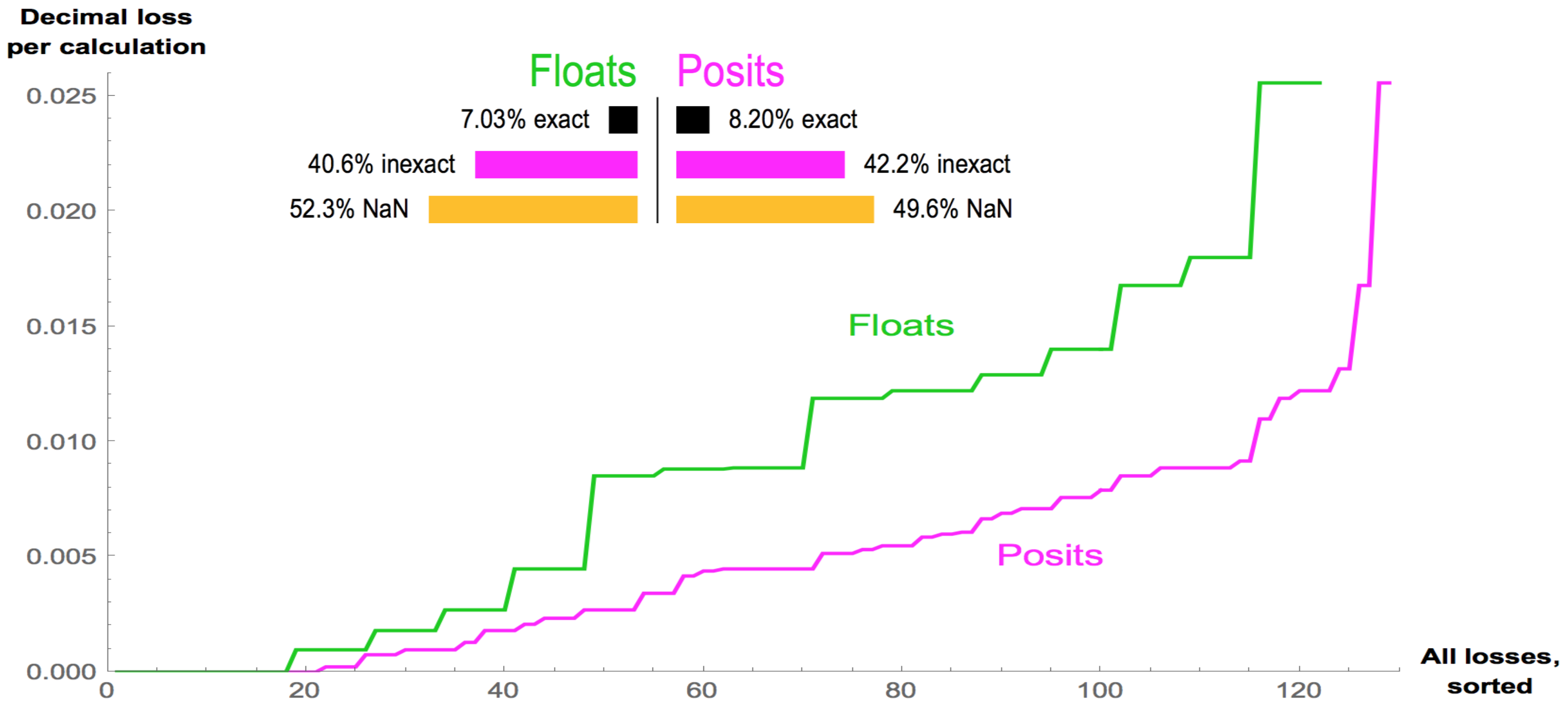
Unary Operations

$1/x$, $\sqrt{x}$, x^2, $\log_2(x)$, 2^x

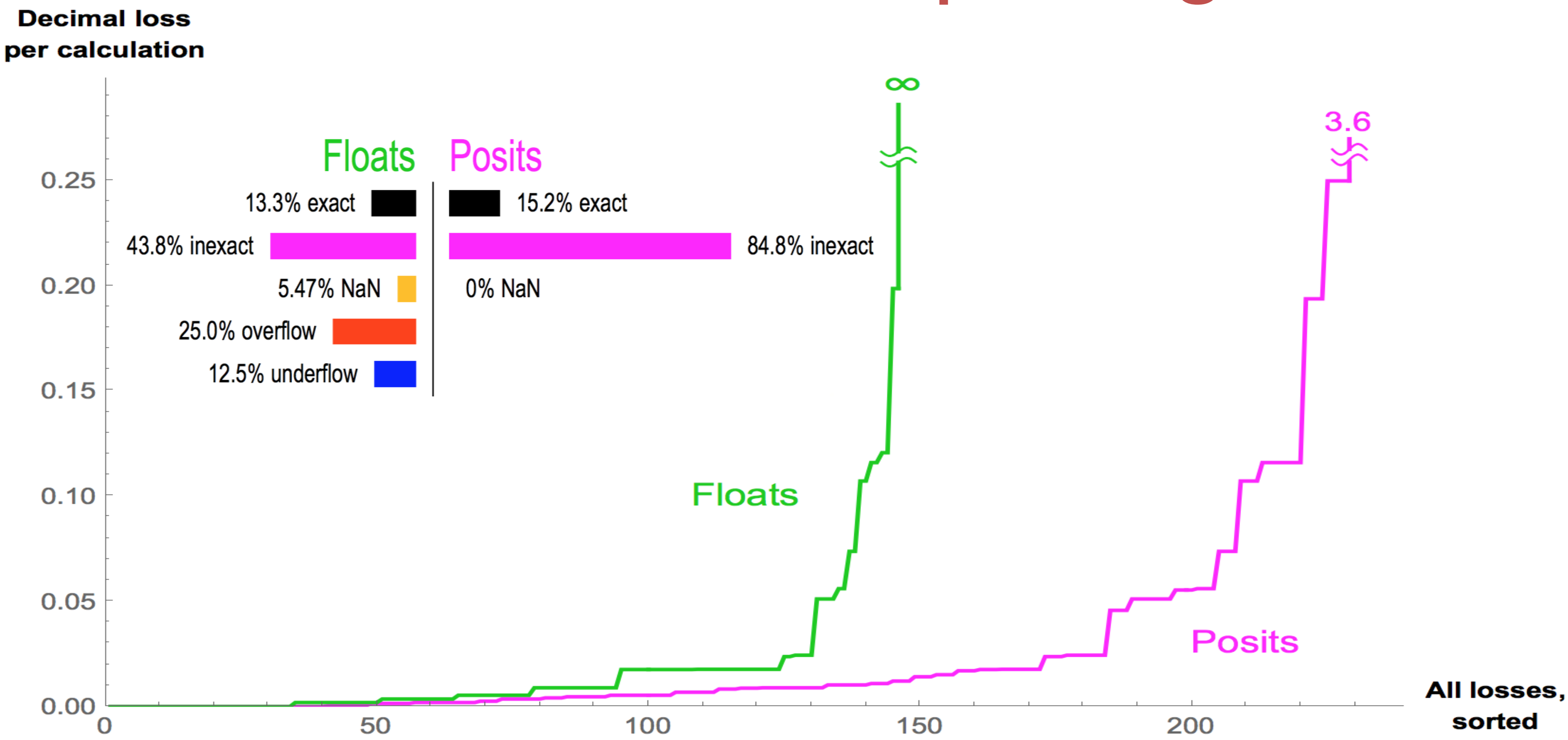
Closure under Reciprocation, 1/*x*



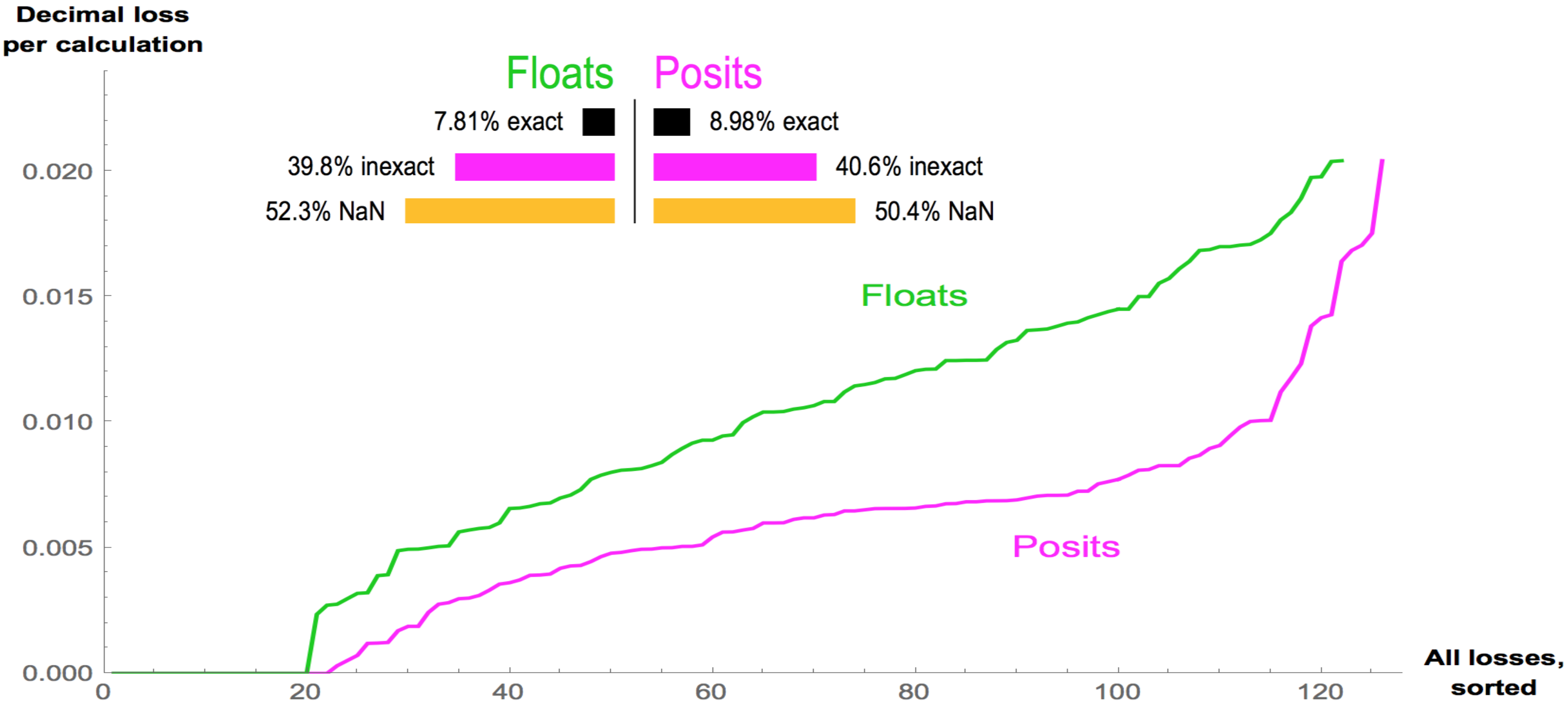
Closure under Square Root, √x



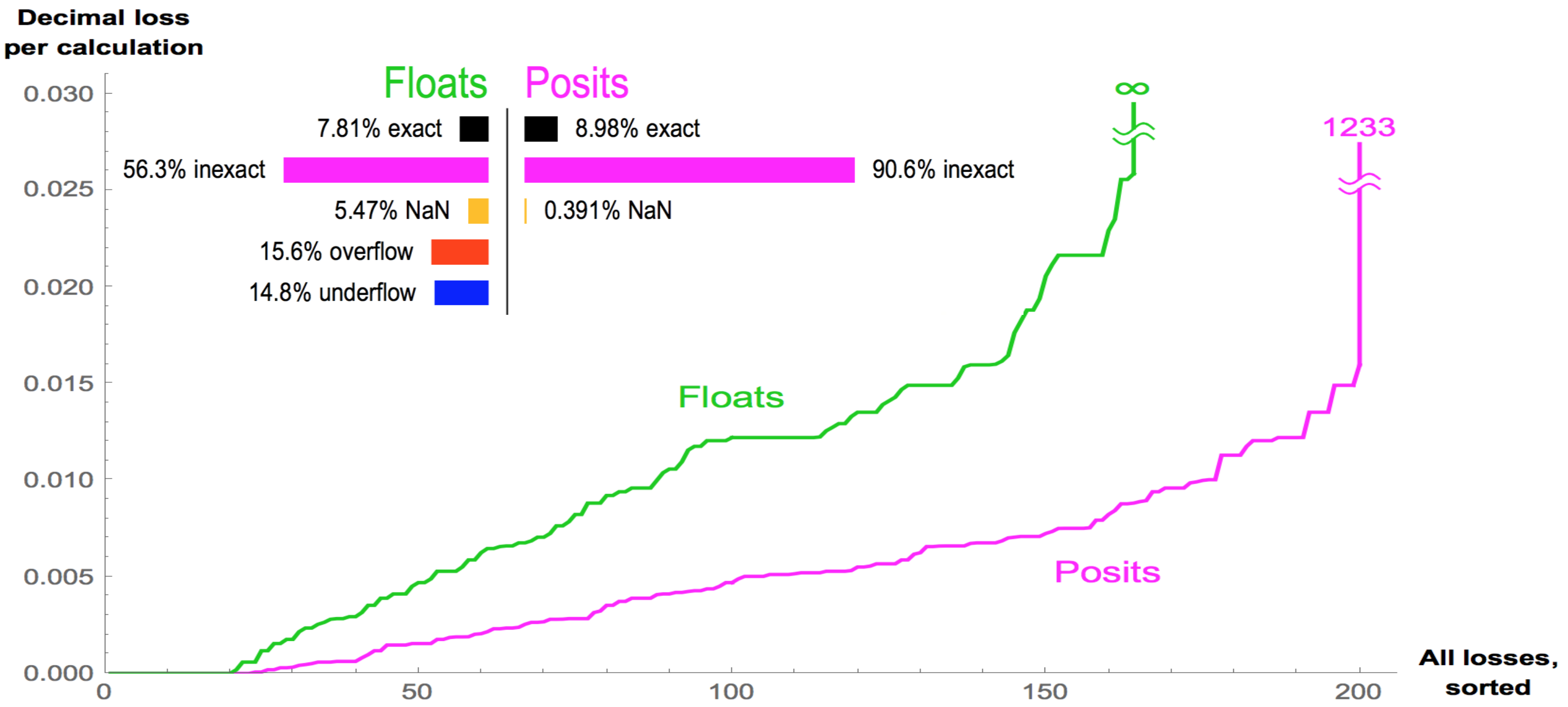
Closure under Squaring, x^2



Closure under $\log_2(x)$



Closure under 2^x



ROUND 2

Two-Argument Operations

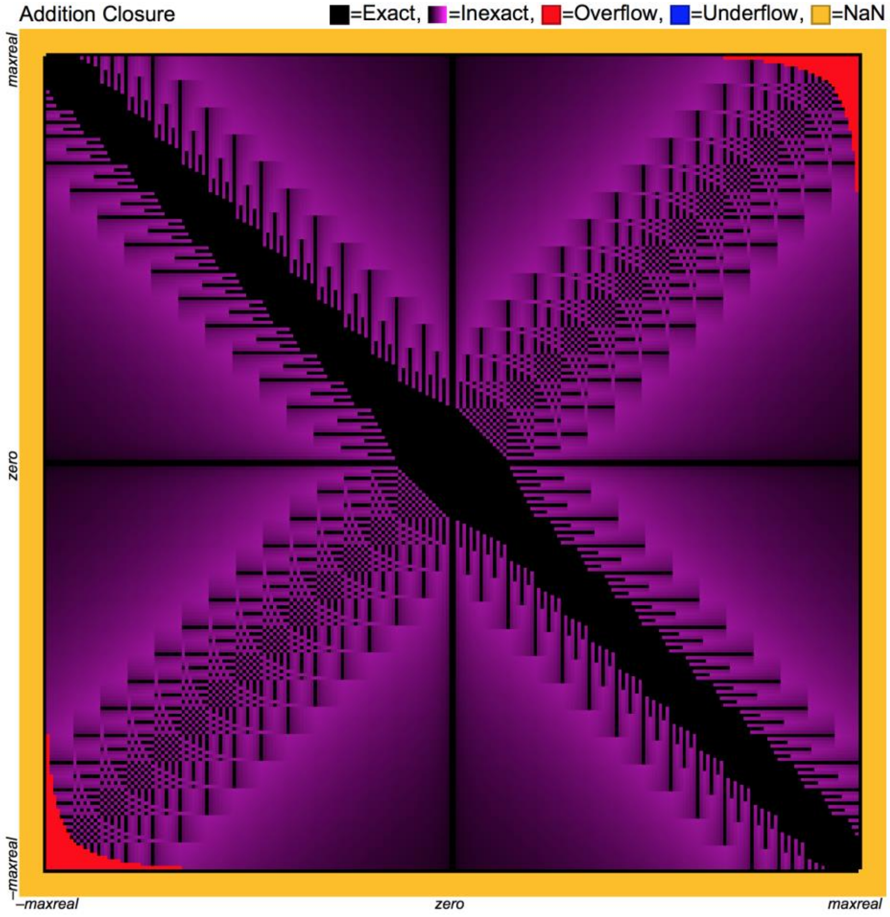
$x + y$, $x \times y$, $x \div y$

Addition Closure Plot: Floats

18.533%	exact
70.190%	inexact
0.000%	underflow
0.635%	overflow
10.641%	NaN

Inexact results are magenta; the larger the error, the brighter the color.

Addition can overflow, but cannot underflow.



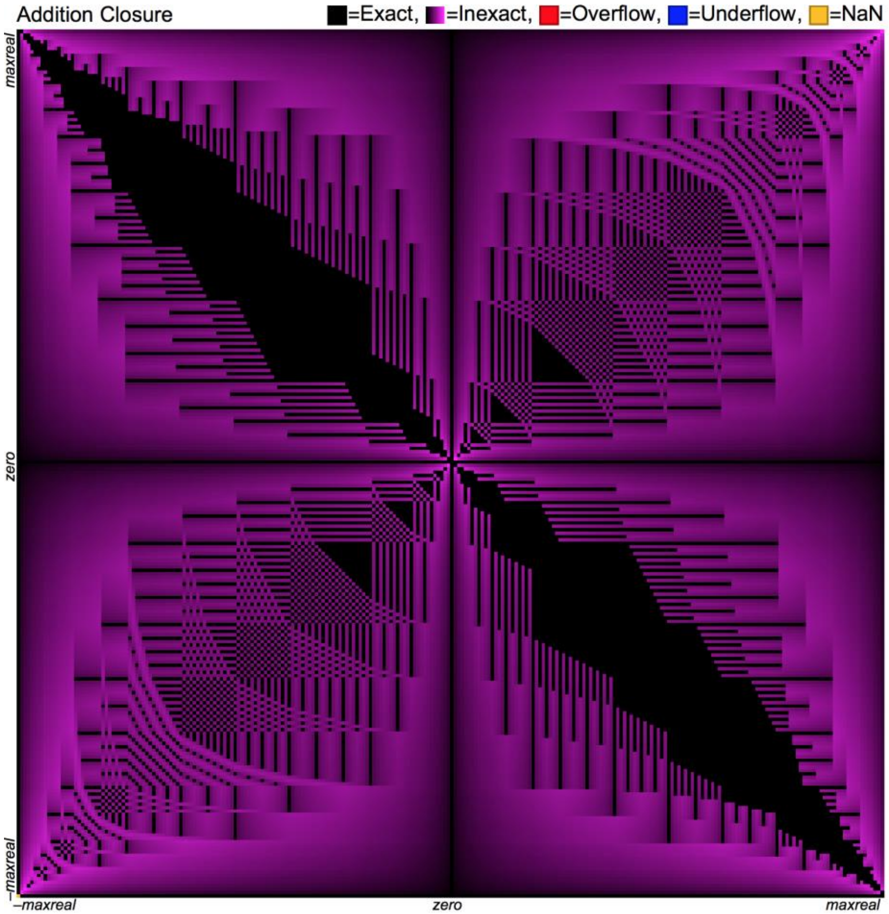
Addition Closure Plot: Posits

25.005%	**exact**
74.994%	inexact
0.000%	underflow
0.000%	overflow
0.002%	NaN

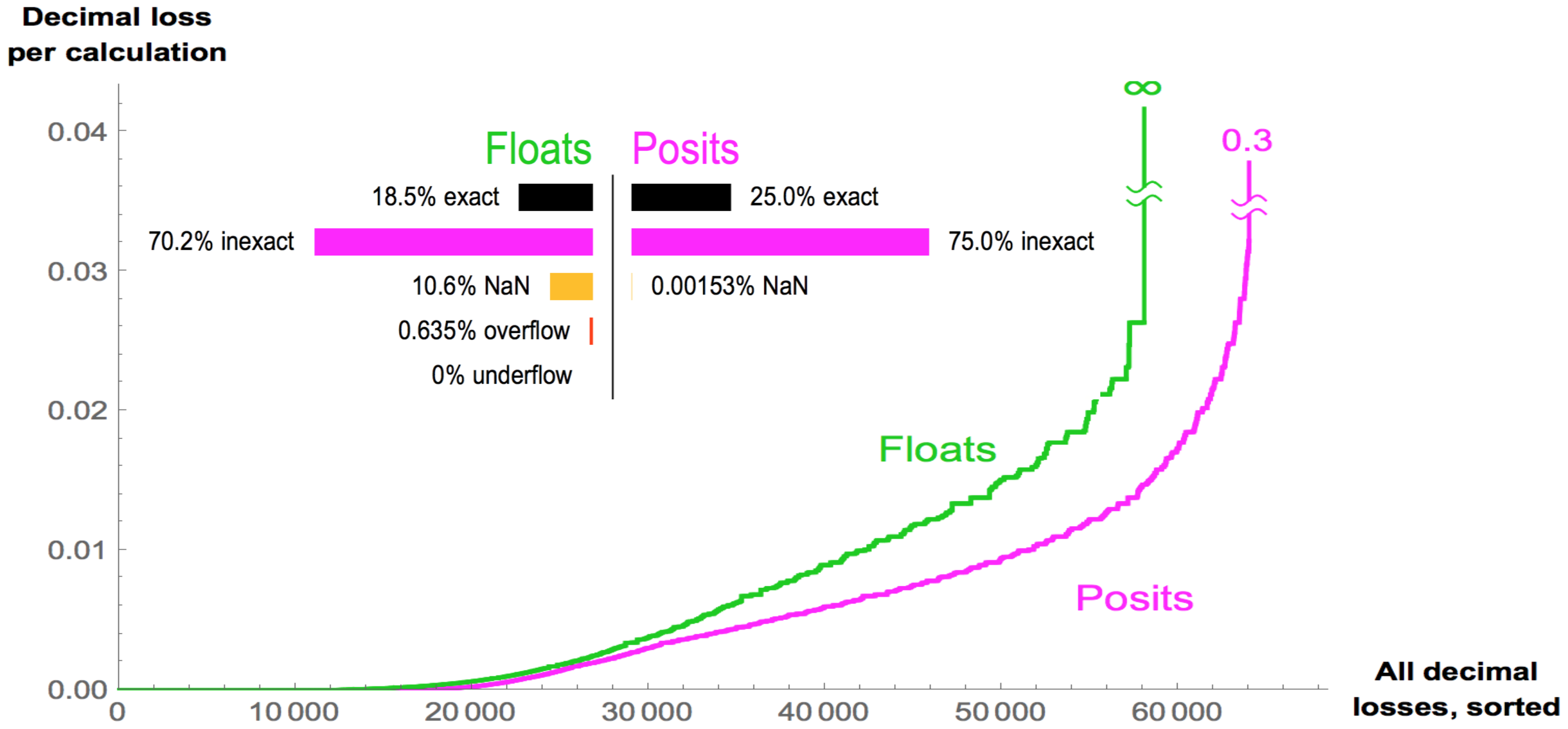
Only one case is a NaN:

$$\pm\infty + \pm\infty$$

With posits, a NaN always *stops the calculation*. (Valids handle NaNs as *sets*.)



All decimal losses, sorted

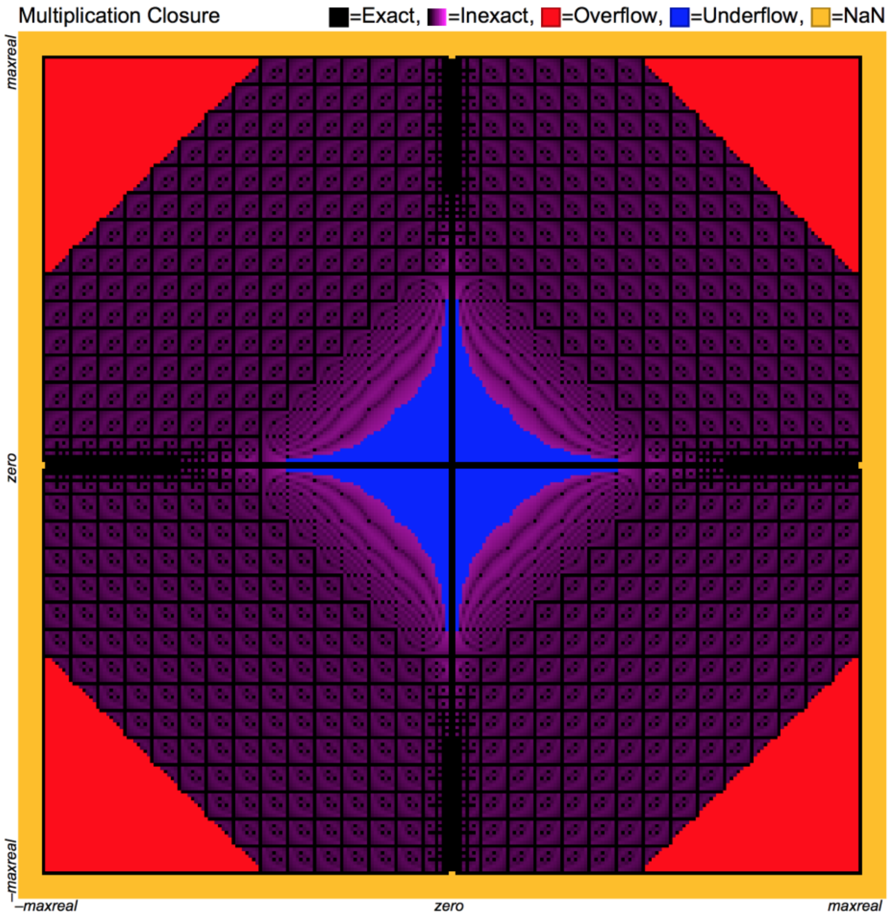


Multiplication Closure Plot: Floats

22.272%	exact
58.279%	inexact
2.475%	underflow
6.323%	overflow
10.651%	NaN

Floats score their first win: more exact products than posits…

but at a **terrible cost**!

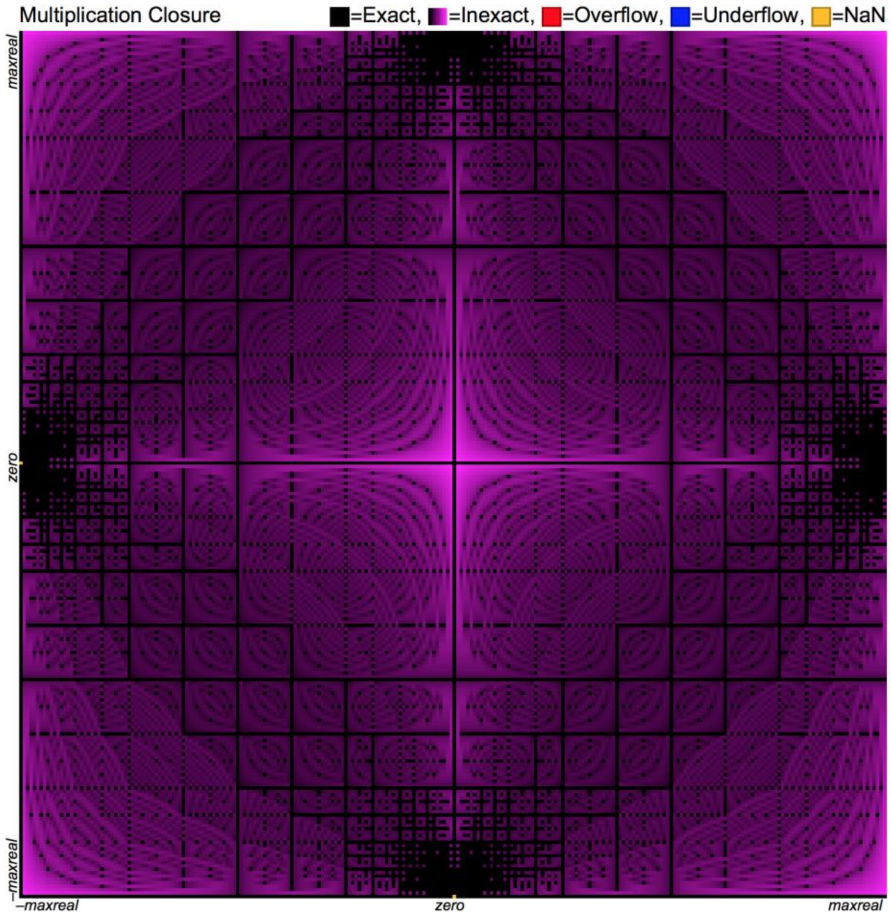


Multiplication Closure Plot: Posits

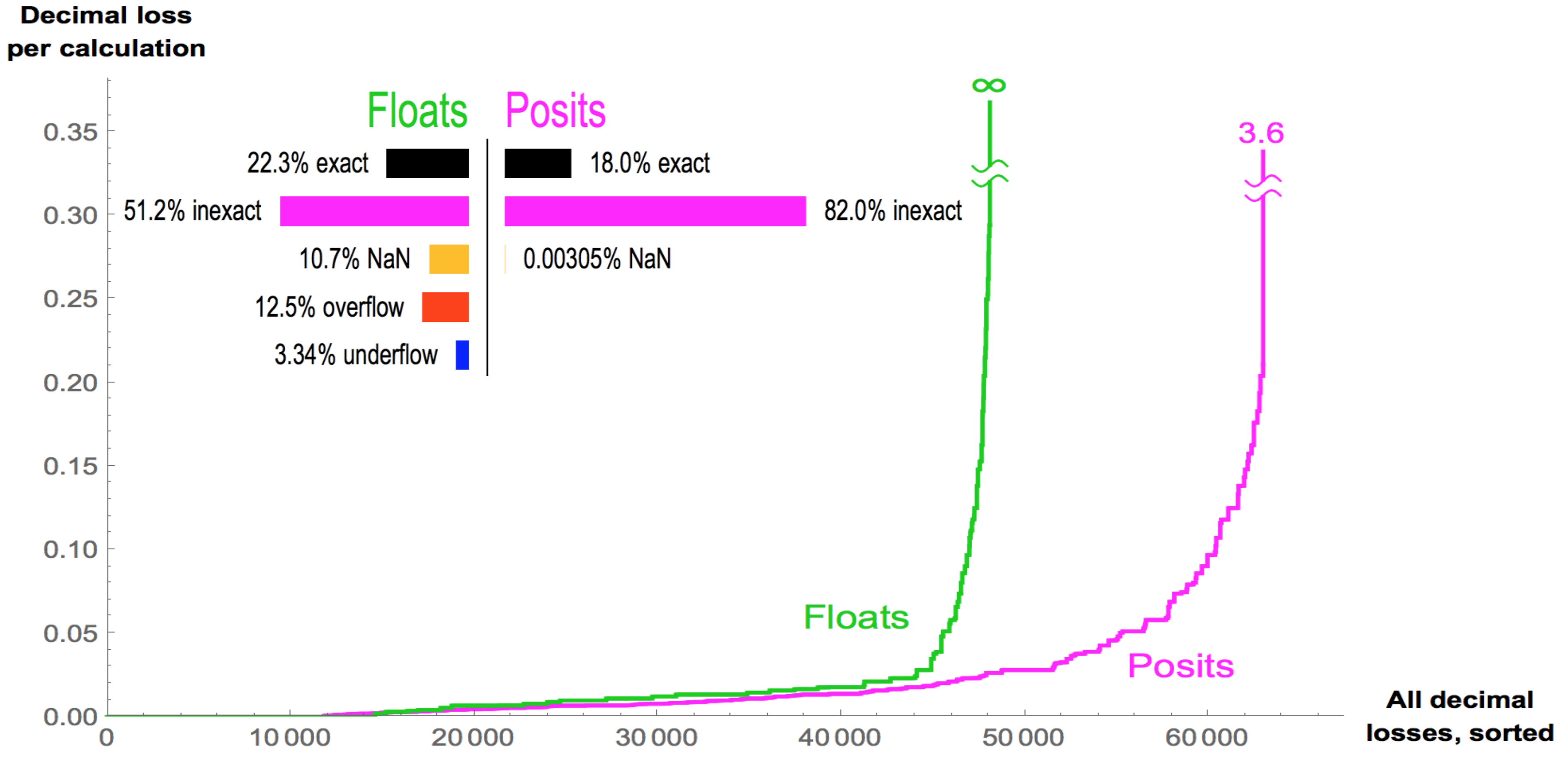
18.002%	exact
81.995%	inexact
0.000%	underflow
0.000%	overflow
0.003%	NaN

Only two cases produce a NaN:

$$\pm\infty \times 0$$
$$0 \times \pm\infty$$



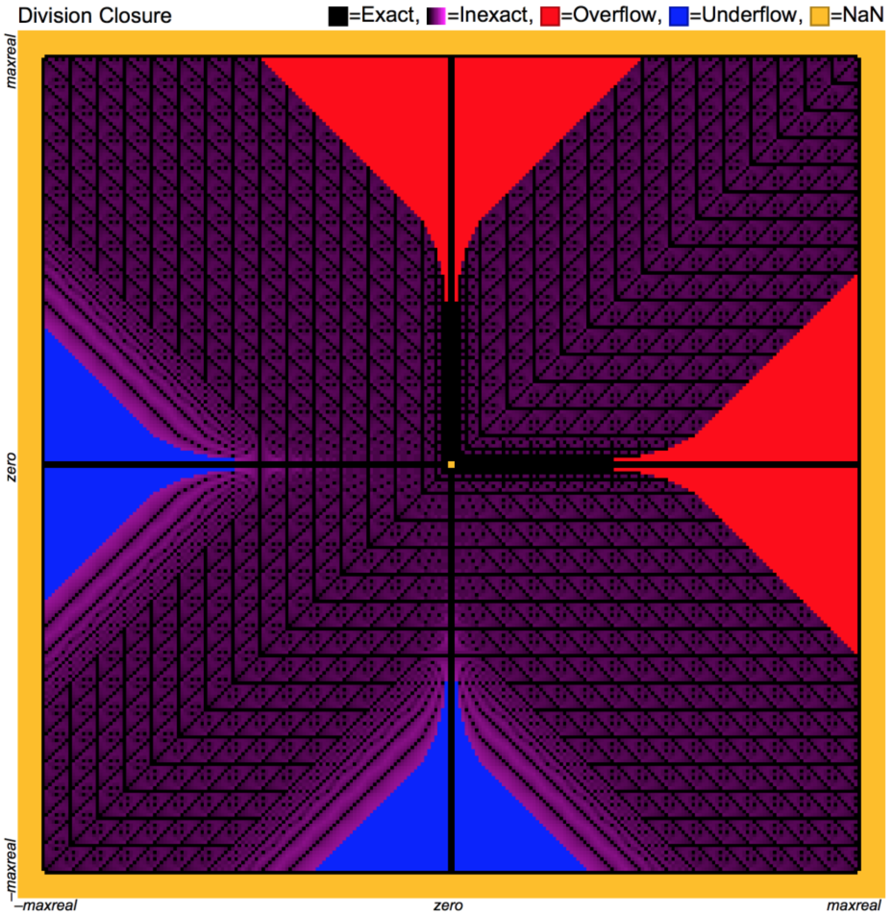
The sorted losses tell the real story



Division Closure Plot: Floats

22.272%	exact
58.810%	inexact
3.433%	underflow
4.834%	overflow
10.651%	NaN

Like multiplication, but with less symmetry.



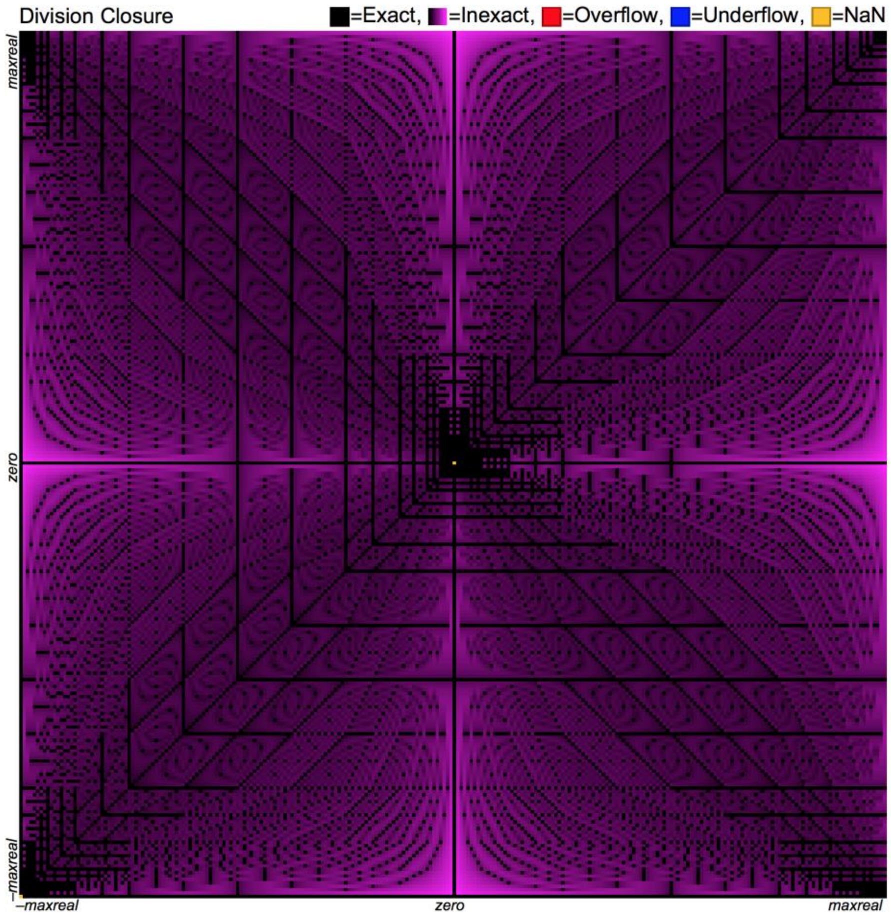
Division Closure Plot: Posits

18.002%	exact
81.995%	inexact
0.000%	underflow
0.000%	overflow
0.003%	NaN

Posits do not have denormalized values. Nor do they need them.

Hidden bit = 1,

always. Simplifies hardware.



ROUND 3

Higher-Precision Operations

32-bit formula evaluation
128-bit triangle area calculation
LINPACK solved with… 16 bits!

Accuracy on a 32-Bit Budget

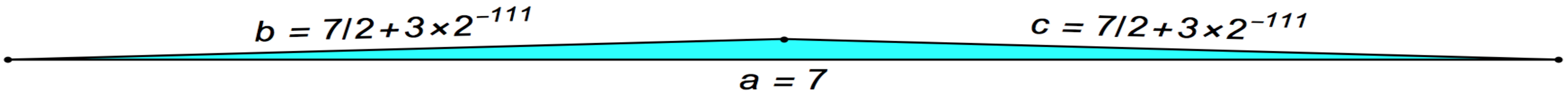
Compute: $$\left(\frac{27/10 - e}{\pi - \left(\sqrt{2}+\sqrt{3}\right)}\right)^{67/16} = 302.8827196\ldots$$ with ≤ 32 bits per number.

Number Type	Dynamic Range	Answer	Error or Range
IEEE 32-bit float	2×10^{83}	302.912⋯	0.0297⋯
Interval arithmetic	10^{12}	[18.21875, 33056.]	$3.3\cdots\times10^{4}$
Type 1 unums	4×10^{83}	(302.75, 303.)	0.25
Type 2 unums	10^{99}	302.887⋯	0.0038⋯
Posits, *es* = 3	3×10^{144}	302.88231⋯	0.00040⋯
Posits, *es* = 1	10^{36}	302.8827819⋯	0.000062⋯

Posits beat floats at both dynamic range and accuracy.

Thin Triangle Area

Find the area of this thin triangle



using the formula $$s = \frac{a+b+c}{2};\ A = \sqrt{s(s-a)\,(s-b)\,(s-c)}$$

and 128-bit IEEE floats, then 128-bit posits.

Answer, correct to 36 decimals:
$3.14784204874900425235885265494550774\cdots\times 10^{-16}$

From "What Every Computer Scientist Should Know About Floating-Point Arithmetic," David Goldberg, published in the March, 1991 issue of *Computing Surveys*

A Grossly Unfair Contest

IEEE quad-precision floats get only *one digit* right:

$3.6348149084233213472592051615805768\cdots \times 10^{-16}$

128-bit posits get **36** digits right:

$3.1478420487490042523588526549455077\cdots \times 10^{-16}$

To get this accurate an answer with IEEE floats, you need the *octuple* precision (256-bit) format.

Posits don't even need 128 bits. They can get a very accurate answer with only *119* bits.

LINPACK: A*x* = *b*
16-bit posits versus *16-bit* floats

- 100 by 100; random A_{ij} entries in (0, 1)
- *b* chosen so *x* should be all 1s exactly
- Classic LINPACK method: LU factorization with partial pivoting. Allow refinement using residual.

IEEE 16-bit Floats	**16-bit Posits**
Dynamic range: 10^{12}	Dynamic range: 10^{16}
Maximum error: 0.011	Maximum error: NONE
Decimal accuracy: 1.96	Decimal accuracy: ∞

Note: work funded in part by DARPA under contract BAA 16-39

64-bit Float versus 16-bit posit

64-bit IEEE Floats	16-bit Posits
1.0000000000000124344978758017532527446746826171875	1
0.9999999999999837907438404727145098149776458740234375	1
1.0000000000000193178806284777238033711910247802734375	1
0.99999999999998501198916756038670428097248077392578125	1
0.999999999999991118215802998747676610946655273437 5	1
0.9999999999999990007992778373591136187314987182617 1875	1
⋮	⋮

Remember this from the beginning?

Find the scalar product $a \cdot b$:

$$a = (3.2\text{e}8,\ 1,\ -1,\ \ 8.0\text{e}7)$$
$$b = (4.0\text{e}7,\ 1,\ -1,\ -1.6\text{e}8)$$

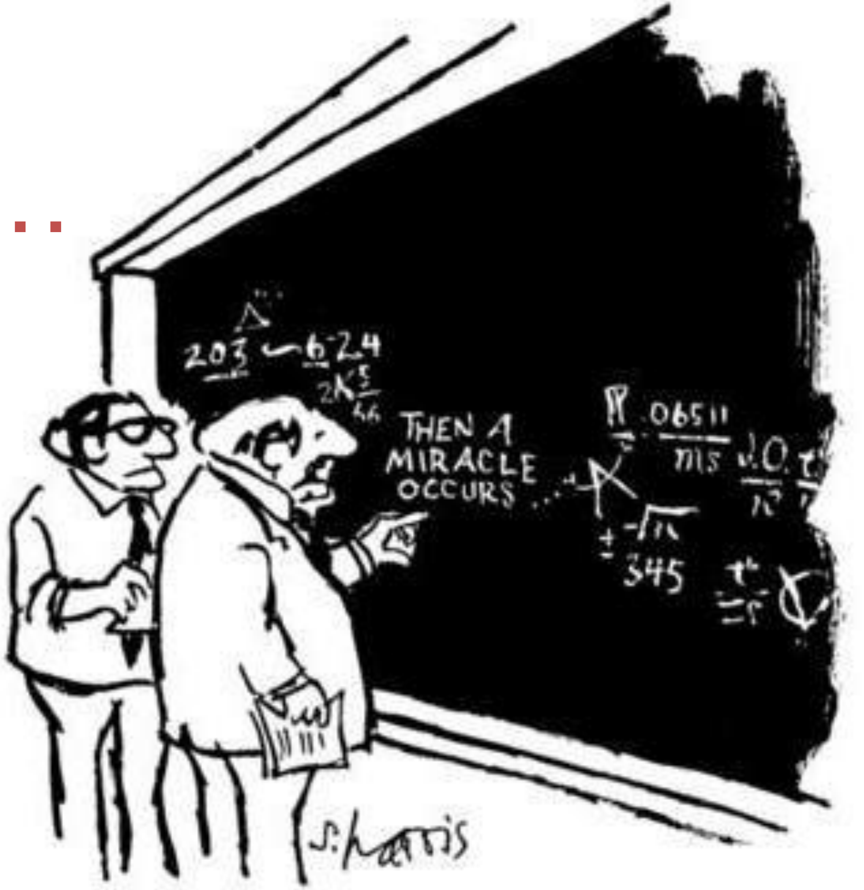
Posit answer: $a \cdot b = 2$ (correct)

IEEE floats require 128-bit precision to get it right.
Posits ($es = 3$) need only 25-bit precision to get it right.
Fused dot product is 3–6 times **faster** than the float method.*

* "Hardware Accelerator for Exact Dot Product,"
David Biancolin and Jack Koenig, ASPIRE Laboratory, UC Berkeley

Type 1, 2 unum hardware challenges…

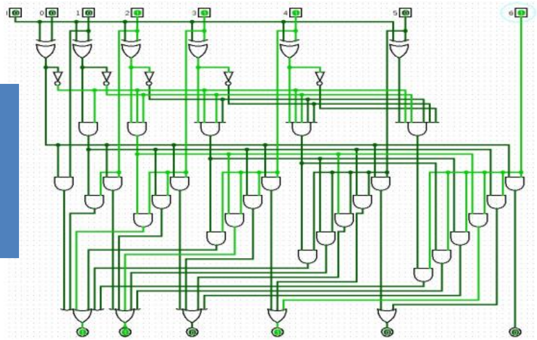
- Type 1 unums need *variable size*; require unpacked form for simple indexing
- Type 2 unums need *table look-up*; only scale to about 20 bits
- But: Posits and Valids are ready to go **now**!

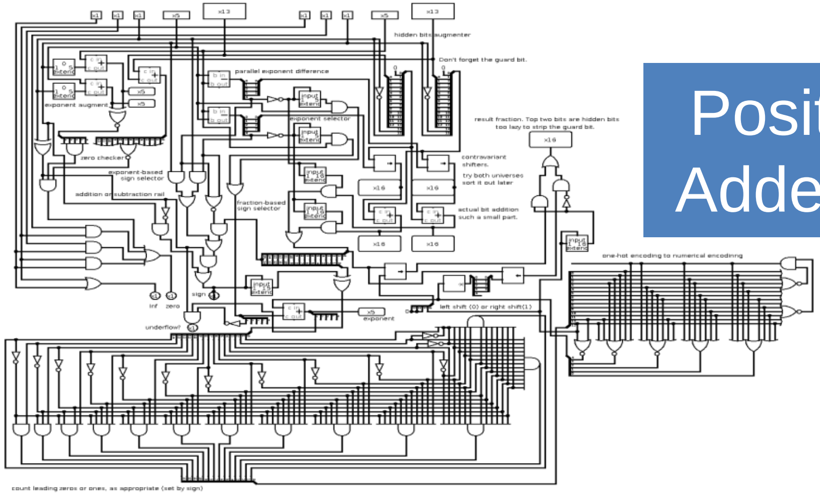


"I think you should be more explicit here in step two."

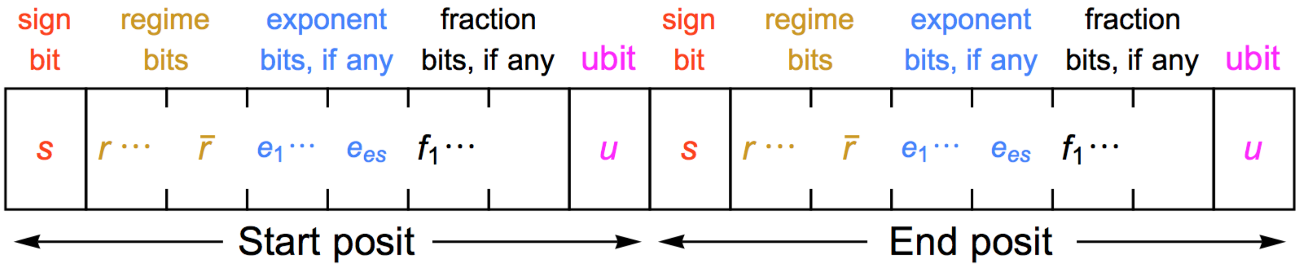
Building posit chips: The race is on

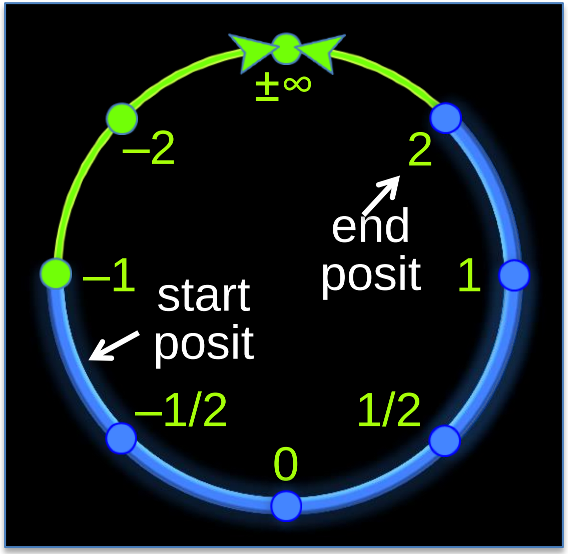
- Like IEEE floats, but *simpler* and *less area* (!)
- Multiplier, adder designs are done
- REX Computing, and a handful of startups are working on it; Intel is showing interest
- Looks ideal for GPUs; more arithmetic per chip





Posit pairs beat *intervals* at their own game, too: **Valid** mode





"*Posit*" *mode*: Round unum if operation yields an inexact.

"*Valid*" *mode*: Rigorous bounds; "NaN" answers are *sets*

32-bit precision may suffice now!

- Early computers used 36-bit floats.
- IBM System 360 went to 32-bit.
- It wasn't quite enough.
- *What if 32-bit posits could replace 64-bit floats for structural analysis, circuit simulation, etc.?*
- Potential 2x shortcut to exascale. Or more.

Summary

- Posits beat floats at their own game: superior accuracy, dynamic range, closure
- Bitwise-reproducible answers (at last!)
- Proven *better answers* with same number of bits
- …or, equally good answers with *fewer* bits
- Simpler, more elegant design can reduce silicon cost, energy, and latency.

Who will produce the first posit arithmetic chip?