Closing the Gap Between Float and Posit Hardware Efficiency

Aditya Anirudh Jonnalagadda
Rishi Thotli
John L. Gustafson

Abstract

- **Motivation**: Posits outperform IEEE floats at the same bit width but incur high hardware cost of decode/encode cost compared to floating-point
- **Idea**: Bound the regime to 6 bits (b-posit), limiting regime size variability so decode–encode maps to simple multiplexers-allowing for simpler parallel hardware.
- **Numeric range**: With eS = 5 and n > 12, dynamic range ≈ 2^−192 to 2^192 (~10−58 to 10^58)
- **Hardware Performance Relative to Standard Posit**:
  - Decoder: −79% power, −71% area, −60% latency
  - Encoder: −68% power, −46% area, −44% delay
- **Hardware Performance Relative to IEEE FP32**:
    - Matches or exceeds speed and area
    - slight worst-case power increase due to higher speed.
- **Hardware Scalability**: Advantages grow with precision; outperforms standard posit hardware at 64-bit as well.
- **Takeaway**: B-posit delivers posit accuracy and mathematical properties without typical power/area/latency penalties and should inform future standard revisions.

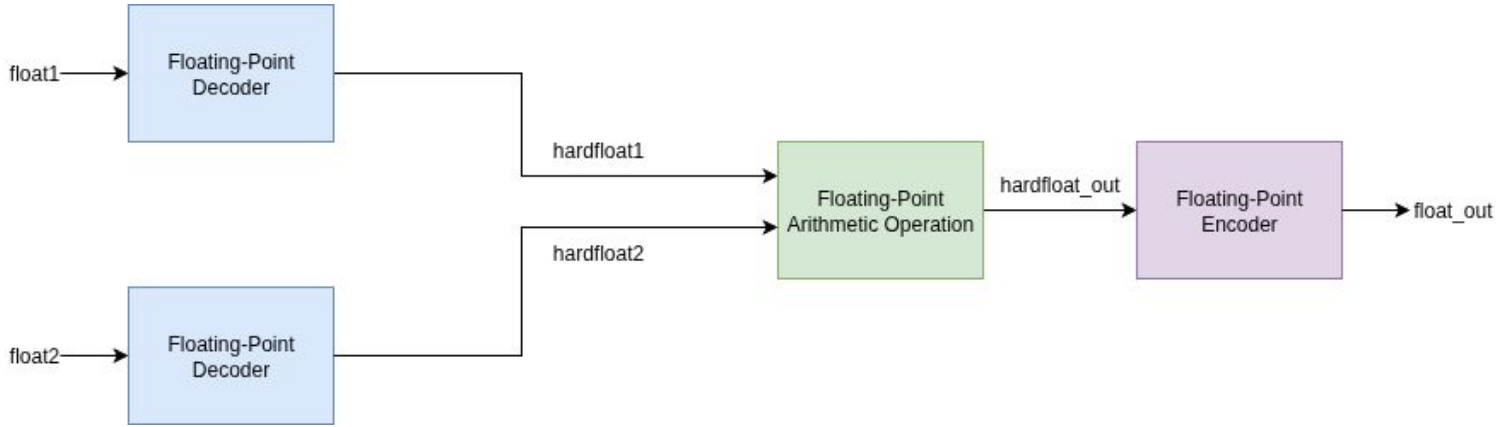
Background - Floating Point Arithmetic

IEEE-754 exceptions and subnormals

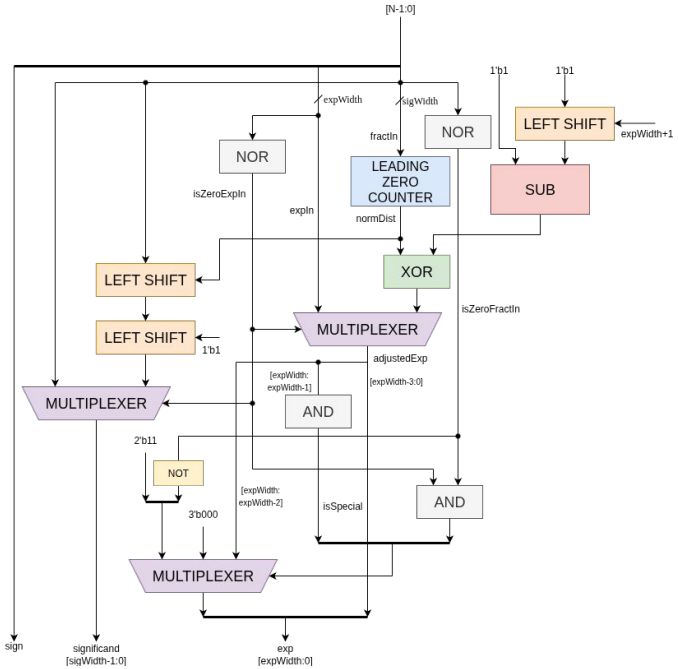
- **Exception classes**: The standard defines five exceptions—invalid operation, division by zero, overflow, underflow, and inexact.
- **Subnormals:** Numbers with exponent field = 0 and a nonzero significand to extend the range toward zero.
- **Practical cost**: Hardware to detect exceptions and handle subnormals is non-trivial, however is left unaccounted in performance models.

Berkeley HardFloat

- Implementation by UC Berkeley that includes hardware support for exception detection and subnormal handling
- Performs floating-point operations in three stages- Decode, Arithmetic and Encode
- Decode handles exceptions, subnormals by converting float to an intermediate “HardFloat” format and encode does the reverse



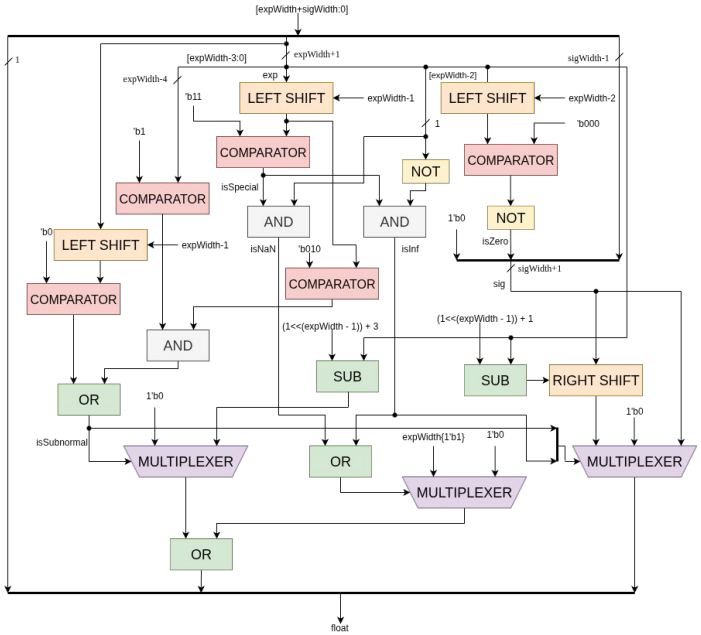
Floating-Point Decode



- Converts IEEE float to HardFloat's recoded format (adds 1 exponent bit for internal ops).
- Subnormal handling: Normalizes subnormals by left-shifting significand and adjusting exponent so the arithmetic datapath treats them like normals.
- Exception Detection: NaN/Inf/Zero detection via exponent field patterns; NaN identified through exponent pattern and payload checks.

	standard format			HardFloat's recoded format		
	sign	exponent	significand	sign	exponent	significand
zeros	s	0	0	s	000xx...xx	0
subnormal numbers	s	0	F	s	$2^k + 2 - n$	normalized F<<n
normal numbers	s	E	F	s	$E + 2^k + 1$	F
infinities	s	1111...11	0	s	110xx...xx	$xxxxxxx$...$xxxx$
NaNs	s	1111...11	F	s	111xx...xx	F

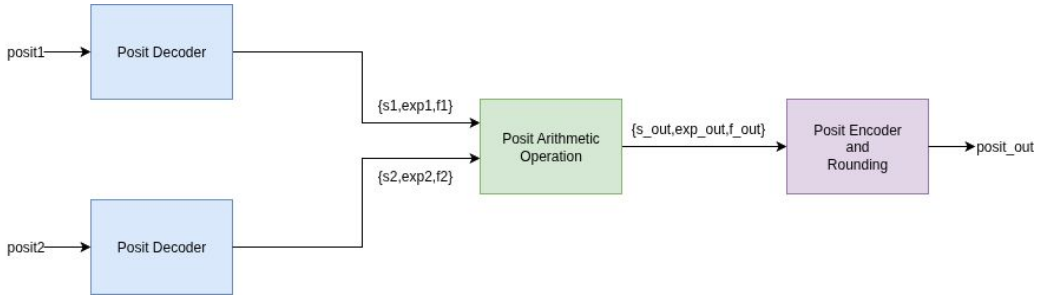
Floating-Point Encode



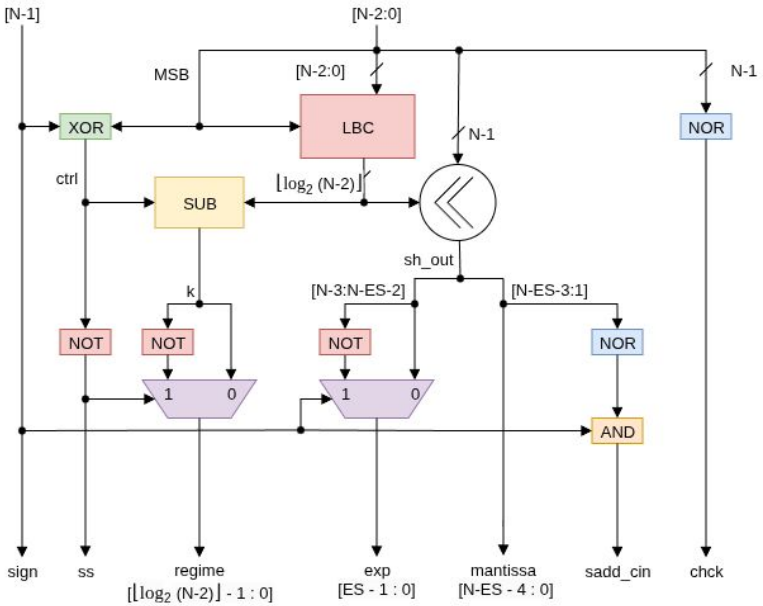
- Takes input as sign, recoded exponent, significand as input and converts it to IEEE float form
- Drives appropriate exponent/fraction bit patterns based on corresponding exception
- Computes right-shift distance, right-shifts significand and sets exponent field to zero in case of subnormals.
- Maps signed exponent to biased exponent in case of normals

Background - Posit Arithmetic

- Similar to floats, posit arithmetic also involves decode and encode steps
- This decode-encode is not to handle subnormals or exceptions, rather to handle the variable regime length of the posit
- A general posit arithmetic operation follows the following steps-
  - Posit decode: Translate the input into an intermediate float-like tuple by first unpacking the regime as a signed integer k from its run-length code, then extracting exponent e and fraction f.
  - Arithmetic: Perform the operation on the intermediate representation with standard alignment, normalization, and rounding logic analogous to floating-point arithmetic.
  - Posit encode: Re-pack the resulting regime, exponent, and fraction into the target posit format



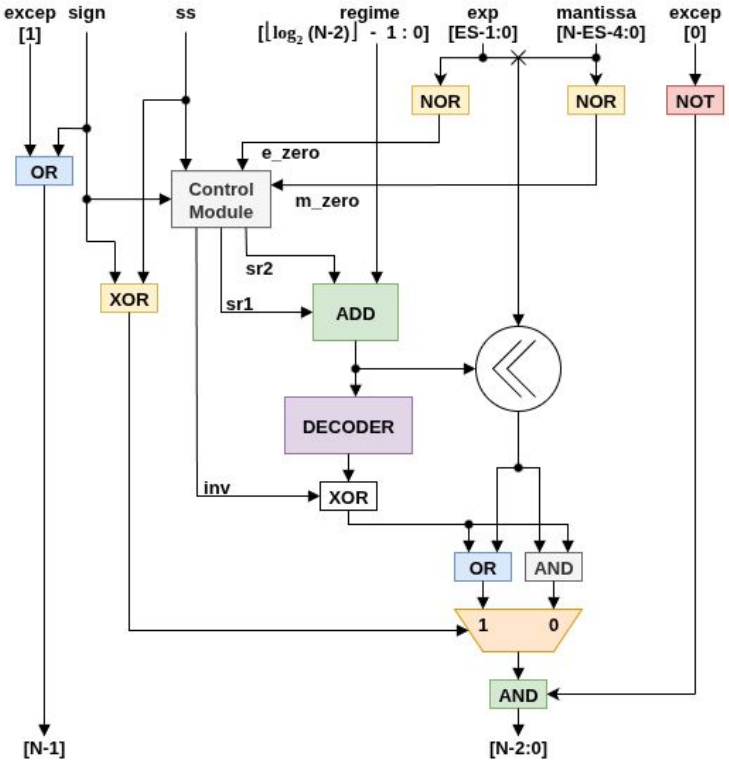
Posit Decode



Converts posit to float-like format in the following steps

- Regime width detection: Use a Leading Bit Counter (LBC) on the run of identical regime bits to determine regime length and signed value k.
- Field extraction: Feed the computed regime width to a left shifter; after aligning, slice out exponent ‘e’ and significand ‘f’ from the shifted word.
- Exception detect: A reduction-NOR over all data bits generates a simple “chck” flag that asserts for zero or NaR, gating downstream arithmetic and encode paths.

Posit Encode

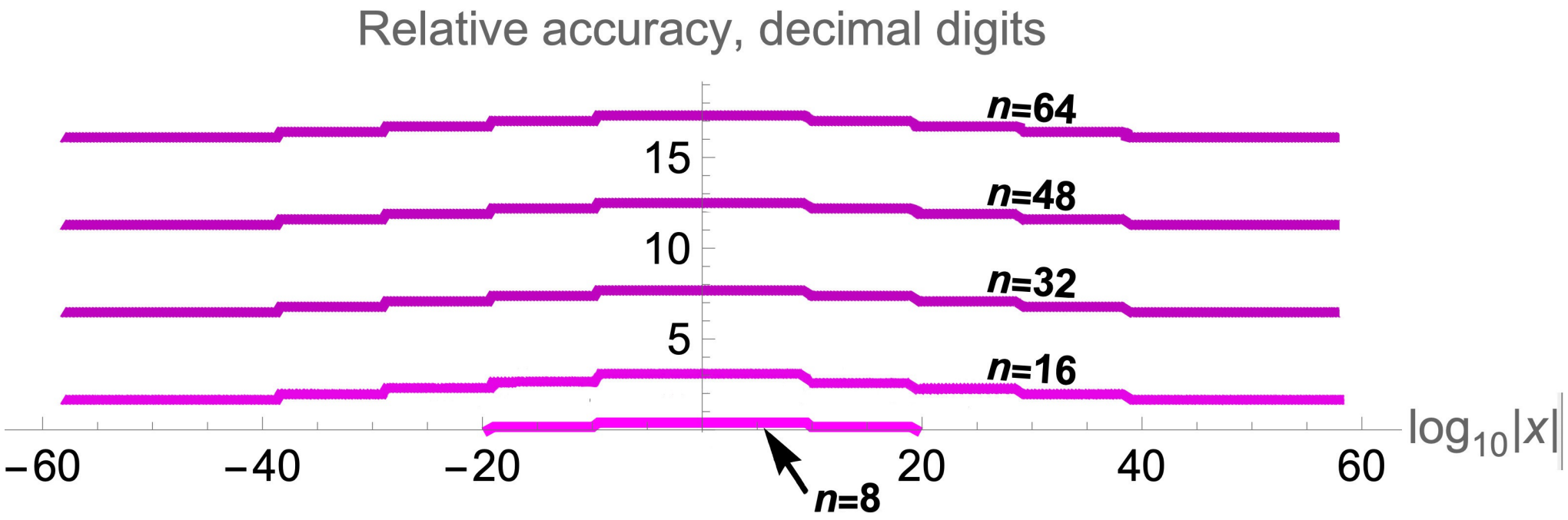


Takes inputs- sign, regime, exponent, fraction, ‘excep’ bits and packs them into posit form in following steps-

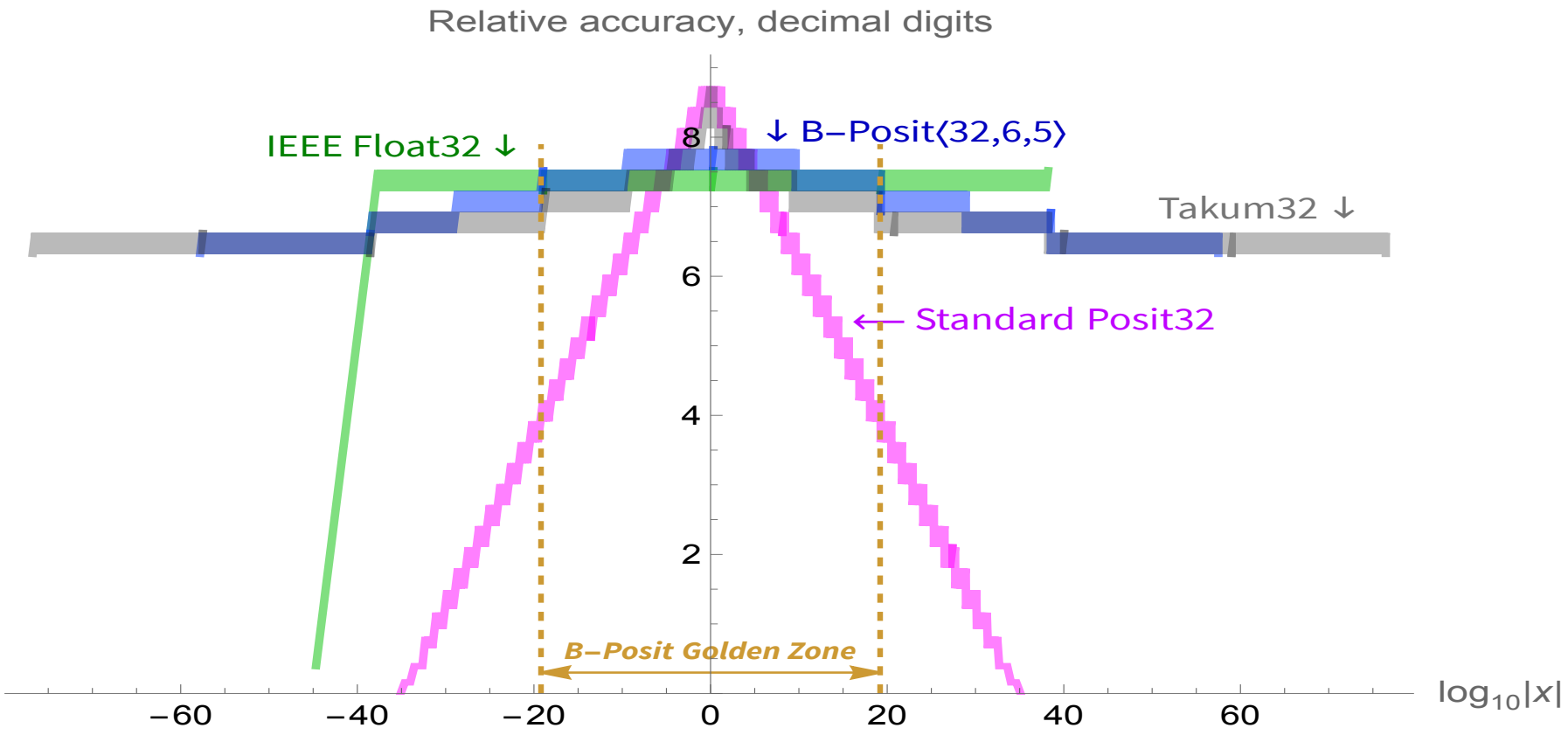
- Shift magnitude: A small decoder selects shift amount from {sr1, sr2} to align e and m with the finalized regime length.
- Regime handling: Apply inv when needed to correct regime polarity after shifts; combine with processed regime count to emit the unary run plus terminator bit.
- Exponent/mantissa: Concatenate {e,m}, apply selected barrel shift, and mask/OR based on inv to preserve boundary between regime and e.
- Final pack: Concatenate sign with {regime bits, e, m}; on excep force all zeros; on excep force 10…0

B-Posit Properties

- Relative accuracy has a guaranteed minimum within dynamic range.
- Expresses scientific constants from Cosmological constant to mass of universe.
- Preserves theorems of numerical analysis, while restoring associative addition.
- **Much more efficient hardware**

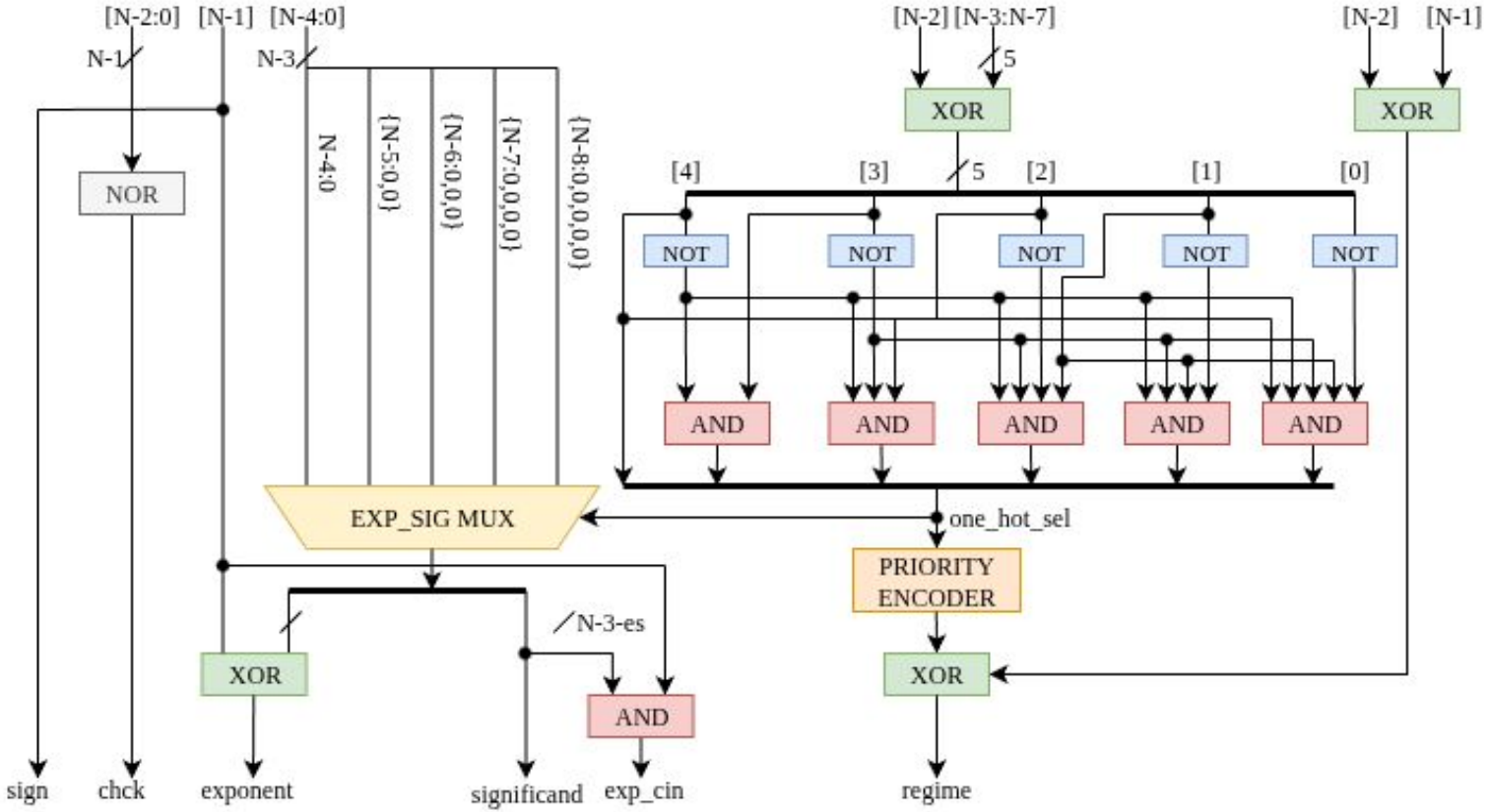


Comparison with 32-bit standard posit, float, takum



Only b-posits and takums satisfy minimum relative accuracy demands of numerical analysis. The b-posits have a huge "Golden Zone", about 10^{-20} to 10^{20}.

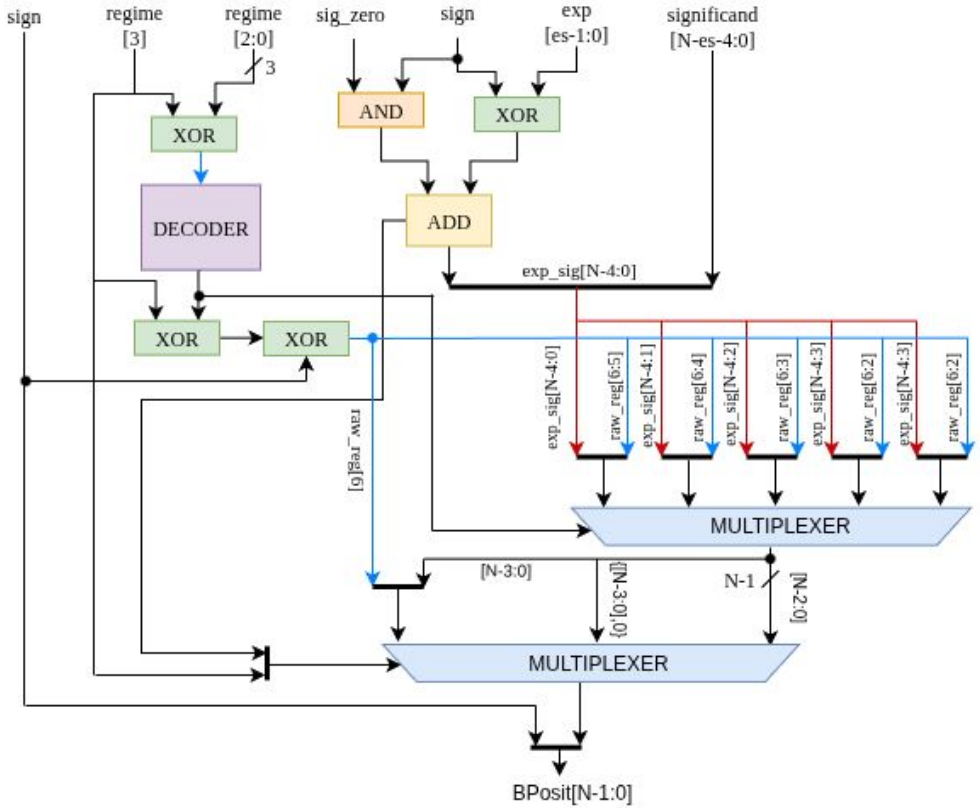
Proposed B-Posit Decoder



Proposed B-Posit Decoder

- Reduces decode to two simple steps:
    - Step 1: Simple combinational logic deduces regime size from the regime string.
    - Step 2: A fixed 5-input multiplexer selects among pre-tapped b-posit alignments based on the regime size, while the priority encoder runs in parallel to compute k.
- Critical path: XOR → NOT → AND → priority encoder / multiplexer, with the encoder and mux parallelized so they do not lengthen the sequential depth.
- Scaling behavior: Because regime length is capped at 6, the regime-deduction logic and priority encoder structures do not grow with N; the mux remains 5-input, only its datapath width increases, which raises power/area but keeps delay nearly constant across precisions.
- Contrast to standard posits: Conventional posit decoders chain leading-bit detectors, barrel shifters, multiplexers, and adders; their logic depth grows with input width since LZD trees and shifter fan-in expand, pushing both gate count and propagation delay upward.
- Practical takeaway: Bounded-regime decoding removes the LZD+barrel-shift sequential chain from the critical path, yielding near-constant delay vs. precision; only power/area scale with N due to wider parallel datapaths.

Proposed B-Posit Encode



Proposed B-Posit Encoder

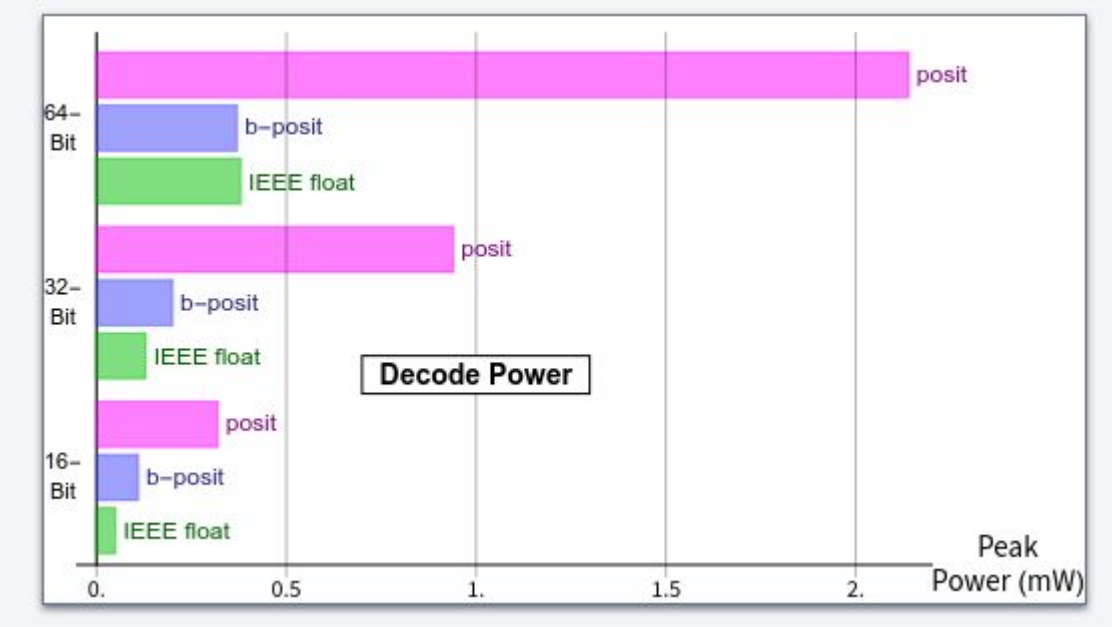
- Reduces encode also to two simple steps:
    - Step 1: Simple combinational logic to deduce regime string length from the regime value.
    - Step 2: Regime string used indirectly as multiplexer select chooses one of five layouts: regime size {2,3,4,5,6}, trimming fraction width as regime grows, while exponent placement remains same
- Critical path: Three XORs → one binary decoder → two small multiplexers; no barrel shifters or adders on the hot path.
- Precision scaling: Only the bus width of MUX inputs grows with N; fan-in stays fixed (5-input), so delay is nearly constant, while power/area scale roughly with width.
- Contrast: Standard posit encoders typically include NOR/control logic, adder, shifter, binary decoder, ANDs, and a MUX—depth and fan-in increase with precision; b-posit avoids these growth drivers by bounding regime length.

Results- Comparison of B-Posit Decode with Posit and Floating-Point Decode

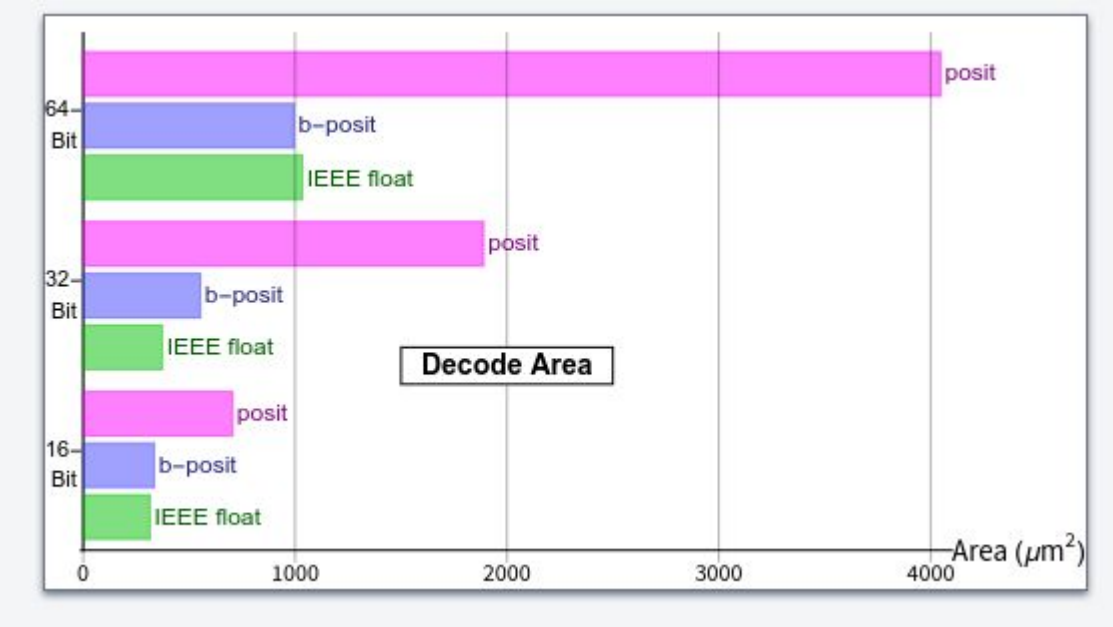
Configuration	Design	Peak Power (mW)	Area(μm^2)	Delay(ns)
16	Floating-Point Decoder	0.05	**315**	0.44
$\langle 16, 6, 5 \rangle$	B-Posit Decoder	0.11	**335**	0.39
$\langle 16, 2 \rangle$	Posit Decoder	0.32	**705**	0.71
32	Floating-Point Decoder	0.13	**373**	0.75
$\langle 32, 6, 5 \rangle$	B-Posit Decoder	0.20	**553**	0.52
$\langle 32, 2 \rangle$	Posit Decoder	0.94	**1890**	1.28
64	Floating-Point Decoder	0.38	**1034**	1.16
$\langle 64, 6, 5 \rangle$	B-Posit Decoder	0.37	**994**	0.65
$\langle 64, 2 \rangle$	Posit Decoder	2.14	**4047**	1.50

- 16-bit: B-posit vs posit: −52% area, −45% delay, −66% power; still higher power and slower than FP at 16-bit.
- 32-bit: B-posit vs posit: −79% power, −71% area, −60% latency; B-posit delay is 69% of FP (≈39% faster than FP32), with power/area overheads vs FP.
- 64-bit: B-posit is >2× faster than FP with slightly less power and area; vs posit: ≈3× faster, <1/6 power, ≈4× smaller.

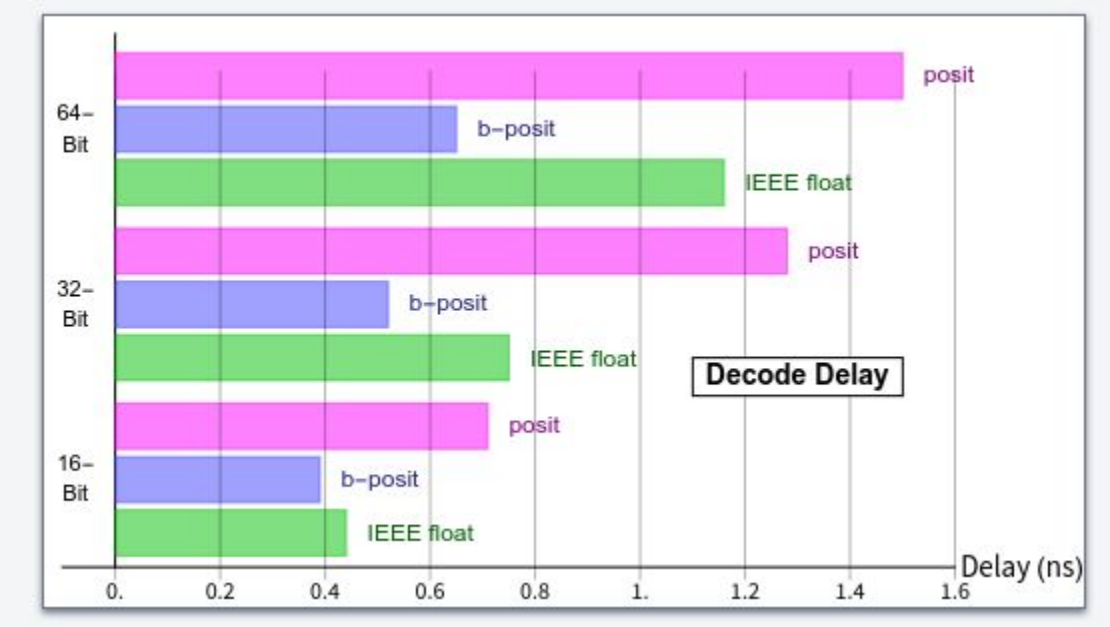
Results- Comparison of B-Posit Decode with Posit and Floating-Point Decode



Results- Comparison of B-Posit Decode with Posit and Floating-Point Decode



Results- Comparison of B-Posit Decode with Posit and Floating-Point Decode



Results- Comparison of B-Posit Encode with Posit and Floating-Point Encode

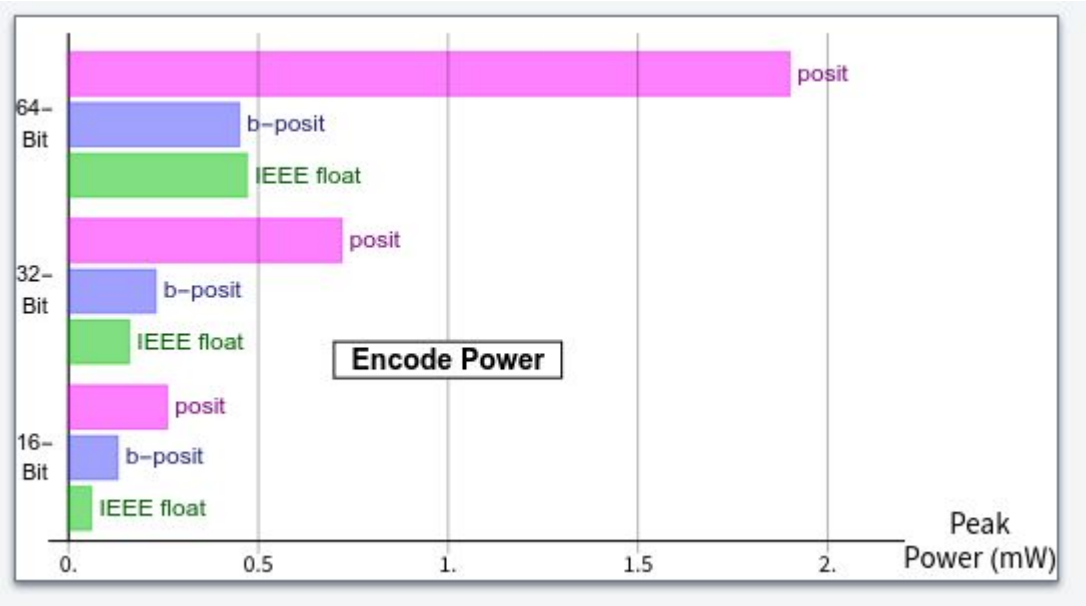
Configuration	Design	Peak Power (mW)	Area(μm^2)	Delay(ns)
16	Floating-Point Encoder	0.06	**297**	0.29
$\langle 16, 6, 5 \rangle$	B-Posit Encoder	0.13	**418**	0.39
$\langle 16, 2 \rangle$	Posit Encoder	0.26	**610**	0.71
32	Floating-Point Encoder	0.16	**777**	0.4
$\langle 32, 6, 5 \rangle$	B-Posit Encoder	0.23	**711**	0.43
$\langle 32, 2 \rangle$	Posit Encoder	0.72	**1330**	0.77
64	Floating-Point Encoder	0.47	**1878**	0.53
$\langle 64, 6, 5 \rangle$	B-Posit Encoder	0.45	**1278**	0.46
$\langle 64, 2 \rangle$	Posit Encoder	1.9	**3093**	1.17

16-bit: B-posit encoder uses about half the power and is nearly 2× faster than the posit encoder, but is slightly worse than the float encoder in power, area, and delay.

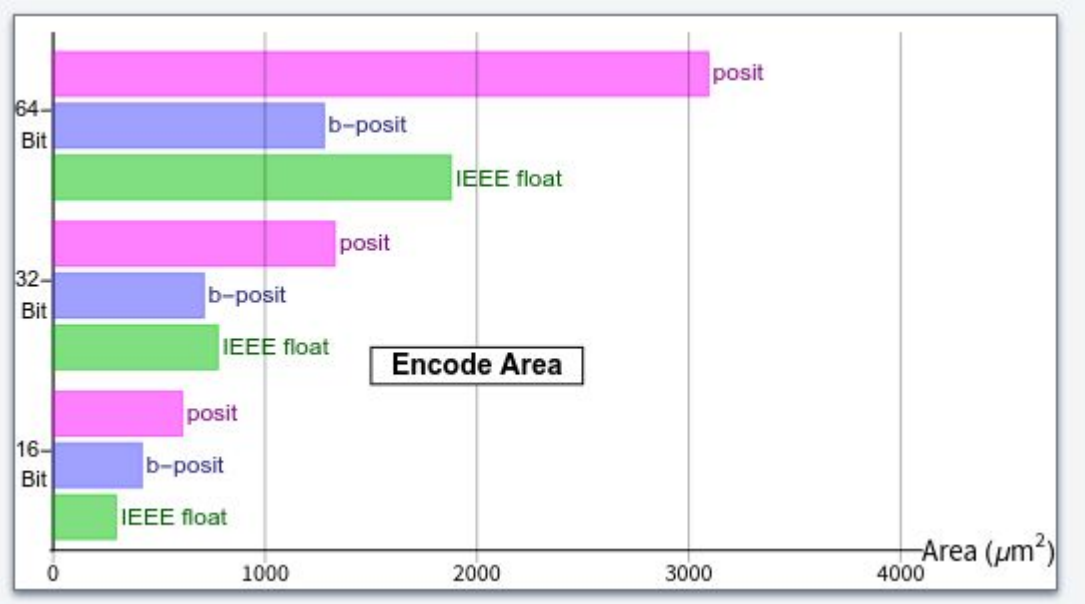
32-bit: Versus posit, B-posit −68% power, −46% area, −44% delay; versus float, B-posit has +43% peak power, ~8% smaller area, and similar delay.

64-bit: B-posit is >2× faster and ~¼ the power of posit; B-posit and float have nearly identical power and delay, with B-posit ~32% smaller area.

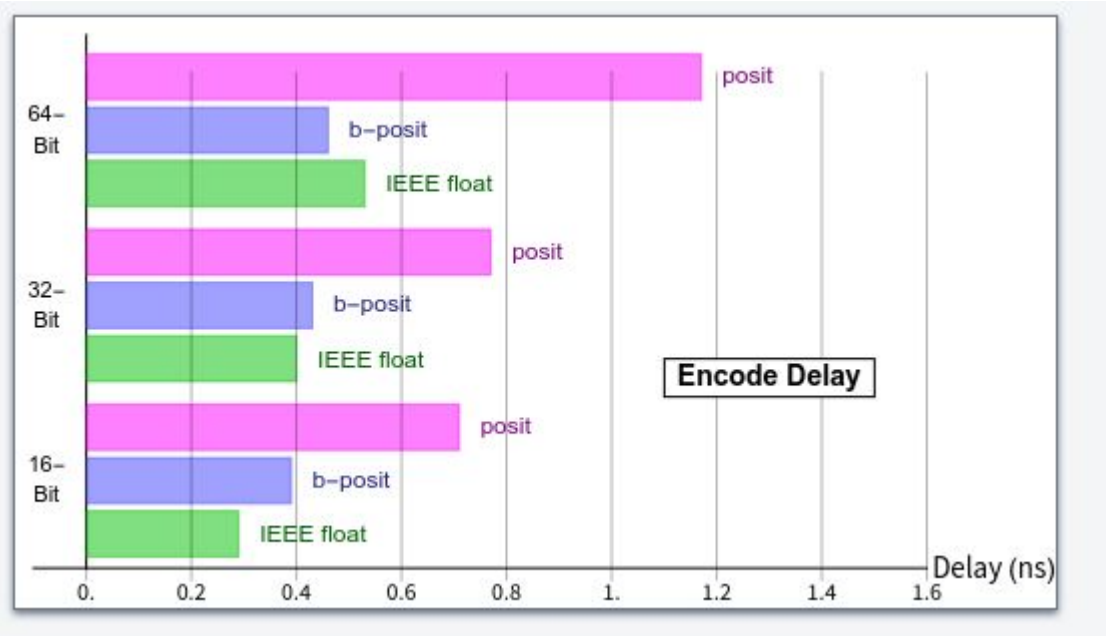
Results- Comparison of B-Posit Encode with Posit and Floating-Point Encode



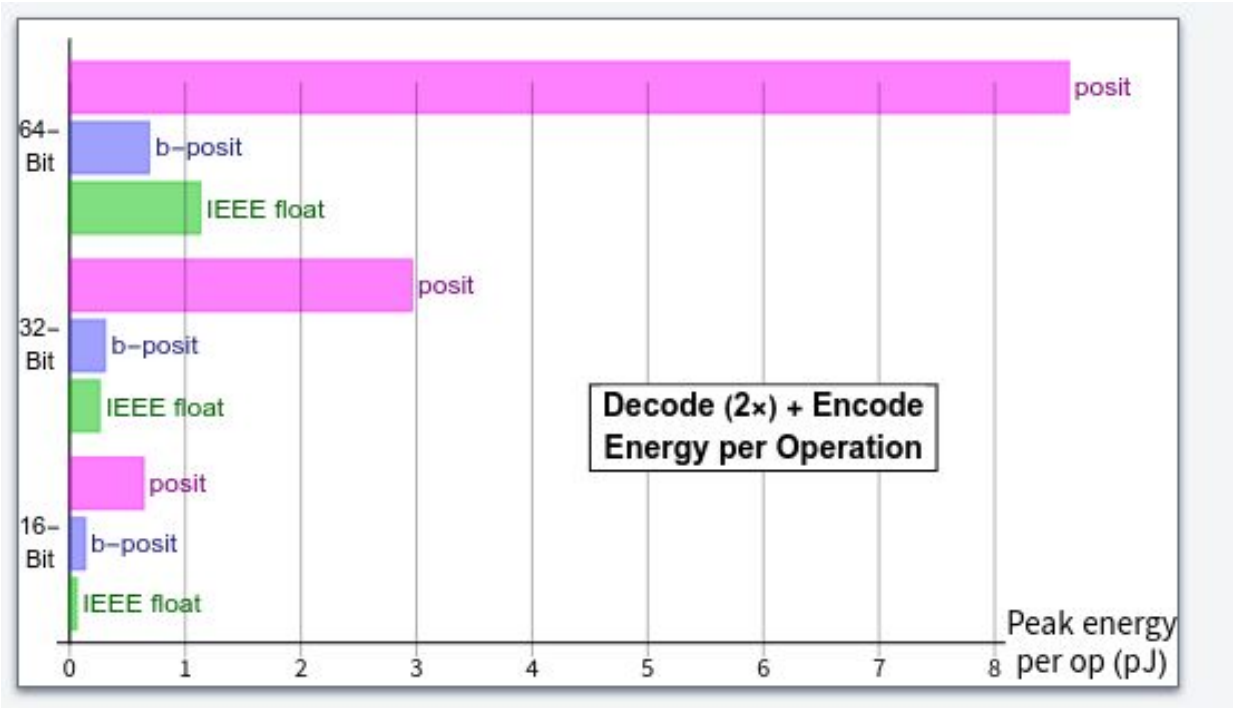
Results- Comparison of B-Posit Encode with Posit and Floating-Point Encode



Results- Comparison of B-Posit Encode with Posit and Floating-Point Encode



Results- Worst Case Energy



Conclusions

- The b-posits beat floats at their own game, *including hardware of the same precision*, for the first time since posits were introduced in 2017. At least for 32-bit precision and higher.
- Combines best-of-both-worlds advantages of floats and posits.
- Preserves theorems of numerical analysis, while restoring associative addition.
- Experiments needed to compare b-posits with takums.