Posit Arithmetic Representation

By,
Karan Jain (B20AI016)
Harshith Kethireddy (B20AI018)

Deliverables

- Analysing Posit-Arithmetic Representation.

- Documentation and Evaluation.

Posit Arithmetics

What's wrong with IEEE 754?

- It's a guideline, not a standard.
- No guarantee of identical results across systems.
- Invisible rounding errors; the inexact flag is useless.
- Breaks algebra laws, like a+(b+c) = (a+b)+c
- Overflows to infinity, underflows to zero.
- No way to express most of the real number line.

Example of invisible Rounding errors

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: $a \cdot b = 2$

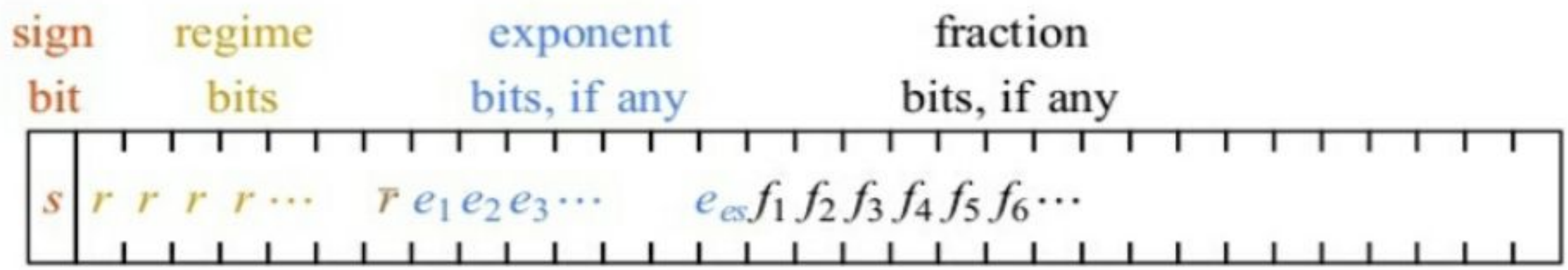
Most linear algebra is unstable with floats!

Reference: https://www.youtube.com/watch?v=N05yYbUZMSQ

What is a Posit?

Posit is a unum that behaves much like a floating-point number of fixed size, rounding to the nearest expressible value if the result of a calculation is not expressible exactly; however, the posit representation offers more accuracy and a larger dynamic range than floats with the same number of bits, as well as many other advantages.

The Posits format



Regime bits

K = -m if m bits are 0

= m-1 if m bits are 1

Binary	0001	001x	01xx	10xx	110x	1110
Run-length,k	-3	-2	-1	0	1	2

The regime contributes to a scaling factor of useed^k, where useed = 2^2^es.

es	0	1	2	3	4
useed	2	$2^2 = 4$	$4^2 = 16$	$16^2 = 256$	$256^2 = 65536$

- **Posit Dynamic Range :**

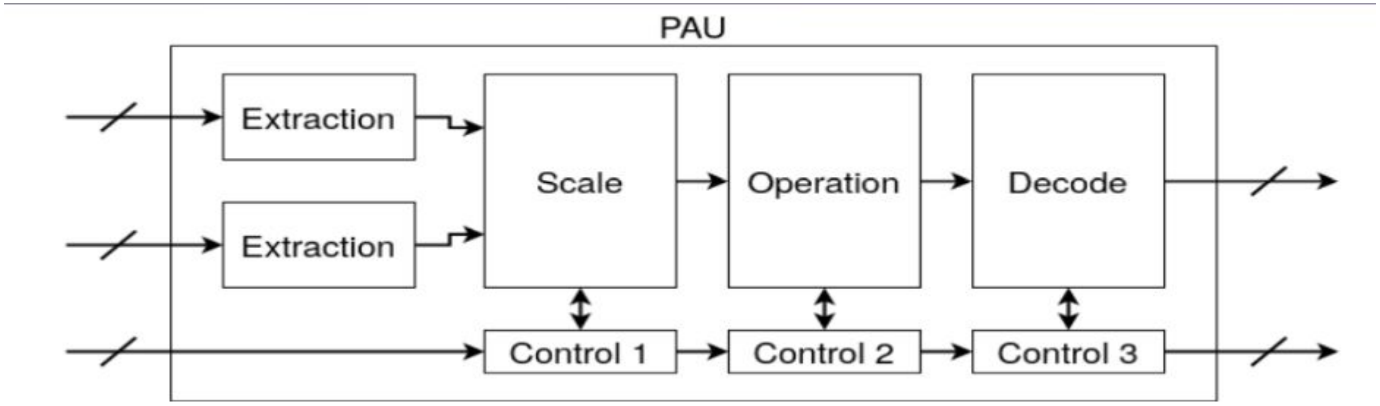
Width		Posit *es*	Posit Dynamic Range
16		1	$4*10^{-9} - 3*10^{8}$
32		3	$6*10^{-73} - 2*10^{72}$
64		4	$2*10^{-299} - 4*10^{298}$

- **Decoding a posit:**

$$X_{10} = \begin{cases} 0, & \text{p=0} \\ \pm\infty, & \text{p=}-2^{n-1} \\ (-1)^S \times useed^k \times 2^{exp} \times (1.Fraction), & \text{all other p.} \end{cases}$$

Posit Core Design

This effectively splits the design into several stages:



- Extraction.
- Scaling
- Operation
- Decoding along with the necessary control logic.

Extraction

Positive Extraction Algorithm

```
If negative
  Take 2's complement
Check for exception values
Remove sign bit
If regime>0
  Count leading 0
  regime=zero_count - 1
else
  negate input
  Count leading zeros
  regime =zero_count
temp=input<<(zero_count - 1)
exp=temp[top:top - es +1]
frac = {1'b1,temp[top - es:end]}
```

Scale

Scale factor = (regime << es) + exp

Addition/Subtraction Scaling

```
shit_value=|sf_a - sf_b|
If op_a > op_b
  greater_frac =a_frac
  smaller_frac = b_frac
  greatest_scaling_factor = sf_a
else
   greater_frac = b_frac
   smaller_frac = a_frac
   greatest_scaling_factor=sf_b
```

Operations

We can do many operation in posit here we have done fout of them:

1) Addition
2) Subtraction
3) Multiplication
4) Division

Decoding block

Control block

Floating Point to Posit Conversion

```
check for exception values
if (x<0)
  negate x
convert decimal value to binary
remove sign bit
if (MSB=1)
  count leading ones
  k = ones − 1
else
   Count leading zeros
   k = −ones
regime = useed^k
```

```
if ( regime is entire bit pattern )
  return max value
remove regime from binary string
exp = 2 ^ ( $es$ MSB’s of binary string )
fraction = remaining bits
compute exact fraction
pos = sign * regime * exp * frac
```

Addition

```c
static struct unpacked_t add(struct unpacked_t a, struct unpacked_t b, bool neg)
{
    struct unpacked_t r;

    POSIT_LUTYPE afrac = HIDDEN_BIT(a.frac);
    POSIT_LUTYPE bfrac = HIDDEN_BIT(b.frac);
    POSIT_LUTYPE frac;

    if (a.exp > b.exp) {
        r.exp = a.exp;
        bfrac = RSHIFT(bfrac, a.exp - b.exp);
    } else {
        r.exp = b.exp;
        afrac = RSHIFT(afrac, b.exp - a.exp);
    }

    frac = afrac + bfrac;
    if (RSHIFT(frac, POSIT_WIDTH) != 0) {
        r.exp++;
        frac = RSHIFT(frac, 1);
    }

    r.neg = neg;
    r.frac = LSHIFT(frac, 1);

    return r;
}
```

Subtraction

```
static struct unpacked_t sub(struct unpacked_t a, struct unpacked_t b, bool neg)
{
    struct unpacked_t r;

    POSIT_UTYPE afrac = HIDDEN_BIT(a.frac);
    POSIT_UTYPE bfrac = HIDDEN_BIT(b.frac);
    POSIT_UTYPE frac;

    if (a.exp > b.exp || (a.exp == b.exp && a.frac > b.frac)) {
        r.exp = a.exp;
        bfrac = RSHIFT(bfrac, a.exp - b.exp);
        frac = afrac - bfrac;
    } else {
        neg = !neg;
        r.exp = b.exp;
        afrac = RSHIFT(afrac, b.exp - a.exp);
        frac = bfrac - afrac;
    }

    r.neg = neg;
    r.exp -= CLZ(frac);
    r.frac = LSHIFT(frac, CLZ(frac) + 1);

    return r;
}
```

Result of Addition&Subtraction

	Float	Posit
Exact	18.5%	25.0%
Inexact	70.1%	75.0%
NaN	10.6%	0.00153%
Overflow	0.757%	0.0%

Multiplication

```c
struct unpacked_t op2_mul(struct unpacked_t a, struct unpacked_t b)
{
    struct unpacked_t r;

    POSIT_LUTYPE afrac = HIDDEN_BIT(a.frac);
    POSIT_LUTYPE bfrac = HIDDEN_BIT(b.frac);
    POSIT_UTYPE frac = RSHIFT(afrac * bfrac, POSIT_WIDTH);
    POSIT_STYPE exp = a.exp + b.exp + 1;

    if ((frac & POSIT_MSB) == 0) {
        exp--;
        frac = LSHIFT(frac, 1);
    }

    r.neg = a.neg ^ b.neg;
    r.exp = exp;
    r.frac = LSHIFT(frac, 1);

    return r;
}
```

Division

```
struct unpacked_t op2_div(struct unpacked_t a, struct unpacked_t b)
{
    struct unpacked_t r;

    POSIT_LUTYPE afrac = HIDDEN_BIT(a.frac);
    POSIT_LUTYPE bfrac = HIDDEN_BIT(b.frac);
    POSIT_STYPE exp = a.exp - b.exp;

    if (afrac < bfrac) {
        exp--;
        bfrac = RSHIFT(bfrac, 1);
    }

    r.neg = a.neg ^ b.neg;
    r.exp = exp;
    r.frac = LSHIFT(afrac, POSIT_WIDTH) / bfrac;

    return r;
}
```

Result of Multiplication&Division

	Float	Posit
Exact	22.3%	18.0%
Inexact	51.2%	82.0%
NaN	10.7%	0.00305%
Overflow	12.5%	0.0%
Underfow	3.34%	0.0%

Posit to Floating point Conversion

//Here, we can find the Posit to Floating-Point converter module. It includes the following files.

//created a Top-module which takes N (posit word size), E (FP exponent size) and es (posit exponent size) as parameters.

//after we created the Dynamic right shifter sub-module and Dynamic left shifter sub-module.

//and then we created the Leading-One-Detector sub-module and Leading-Zero-Detector sub-module.

//and we already define all theories in the above section how we find regime, exponent, mantissa in posit number so here we computed all these things like that.

//and then decode the output and calculate the final answer.

Disadvantages of Posit

- Real-world hardware doesn't support posits (so far).
- Posits don't distinguish between positive infinity, negative infinity and NaN.
- Since there is no NaN, "the calculation is interrupted, and the interrupt handler can be set to report the error".

Contribution

<u>Karan Jain (B20AI016):</u>

Conversion to floating point, Arithmetic operation on posits

<u>Kethireddy Harshith Reddy (B20AI018):</u>

Conversion to posit numbers, advantage of posit over floating point numbers

Reference:-

1. https://posithub.org/docs/Posits4.pdf
2. https://ieeexplore.ieee.org/document/8731915
3. https://eescholars.iitm.ac.in/sites/default/files/eethesis/ee16m048.pdf
4. https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=11516&context=theses
5. https://www.sciencedirect.com/topics/engineering/exponent-bit
6. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/posits/

THANK YOU