# Design/ Practical Experience [EEN1010]
# Department of Electrical Engineering

**Academic year:** 2021-2022

**Semester:** Summer Term 2021-2022

**Date of submission of report:** 20/07/2022

**1) Name of the student:** Kethireddy Harshith Reddy & Karan Jain

**2) Roll number:** B20AI018 & B20AI016

**3) Title of the project:** ‘’Posit Arithmetic Representation’’

**4) Project category:** 3

**5) Target Deliverables:**
- Analyzing of Posit arithmetic representation
- Documentation and evaluation

## 6) Work done:

**Brief Explanation of Problem :**

In posit mode, a unum 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. So here in this we analyze posit arithmetic representation.

**Theory/Literature Review :**

A new data type called a posit is designed as a direct drop-in replacement for IEEE Standard 754 floating-point numbers (floats). Unlike earlier forms of universal number (unum) arithmetic, posits do not require interval arithmetic or variable size operands; like floats, they round if an answer is inexact. However, they provide compelling advantages over floats, including larger dynamic range, higher accuracy, better closure, bitwise identical results across systems, simpler hardware, and simpler exception handling. Posits never overflow to infinity or underflow to zero and NaN indicates an action instead of a bit pattern. A posit processing unit takes less circuitry than an IEEE float FPU. With lower power use and smaller silicon footprint, the posit operations per second (POPS) supported by a chip can be significantly higher than the FLOPS using similar hardware resources.

The posit number system is composed of a run-time varying exponent component, which is defined by a composition of varying length “regime-bit” and “exponent-bit” (with a maximum size of ES bits, the exponent size). This in effect also makes the fraction part to vary at run-time in size and position. These run-time variations inherit an interesting hardware design challenge for posit arithmetic architectures. The posit number system, being at an infant stage of its development, possesses very limited hardware solutions for its arithmetic architectures. Verilog HDL code respective posit arithmetic for any given posit word width (N) and exponent size (ES), as defined under the posit number system.

## Work Required to be Done:

- We need to implement operations like adding, subtraction, multiplication and Divide between two floating numbers.
- After that we need to show our results and documentation of all things.

## Methodology :

- First we figured out what’s the problem with IEEE754 standard.
- Then we figured Type1,Type2 Unums and about their formats and design.
- After that we figured out the Posit format and understood n-bit Posit.
- We dynamic range of Posit and show this in the report.
- We show how to decode a posit number.
- Then we did operations on posits. In coding first we convert a floating point to posit number and extract sign bit, regime bits, exponent bits, and fraction bits.
- After these we did operation addition/subtraction, multiplication, division on posit numbers and then decode into the final result.

- # 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 .

## Background: Type I and Type II Unums :

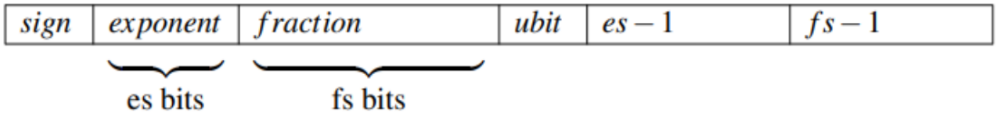
### 1) Type I Unums :

The first type of Unums were developed as a superset of floating point representation. The main goal was to create a dynamic format that would only use as much data space as needed. The reasoning behind this is fairly straightforward: don’t waste memory you don’t need. Other additions include an error flag as well as a system for implementing interval arithmetic.

**Format:**

The format for Type I Unums (abbreviated to T1 from this point), as a superset for Floating Point, becomes a fair bit more complicated. The following sections describe the individual components that make of a T1 number.

→ **type 1 unum bit pattern :**



### ubit:

One of the first steps to help the user of the Type I system (shortened to T1 from this point) was to implement a flag to easily determine if rounding, and thus error, has occurred. This bit is called the ubit and is simply appended to the least significant bit of a floating point style number. This additional bit can prevent costly errors based on the assumption that the calculation is correct.

### Exponent and fraction bit respectively:

In order for a number to have variable size, it is necessary to include information bits along with the actual data. For T1 representation, these take the form of the exponent and fraction size bits. The sizes of these two values are the only values necessary to create a computing environment. As defined in, a T1 environment is set up by {x,y}, where x is the bit-width of the es field and y is the bit-width of the f s field known as esizesize and f sizesize respectively.

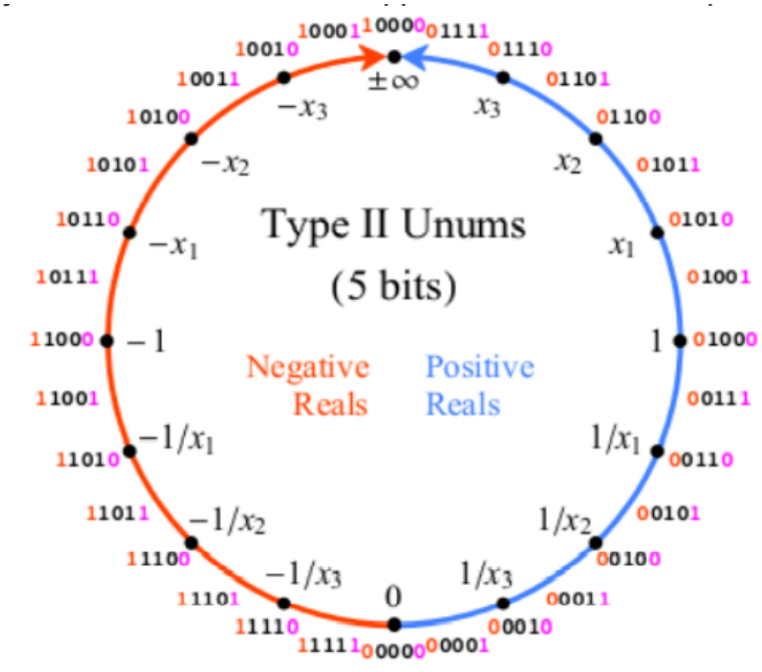
## 2) Type II Unums :

In order to counteract the hardware intensity of Type I Unums, Type II was developed to operate much more efficiently in hardware. These operate as a method of pointer arithmetic rather than operating on actual data. This format completely breaks with traditional floating point for the purposes of speed.

### Design:

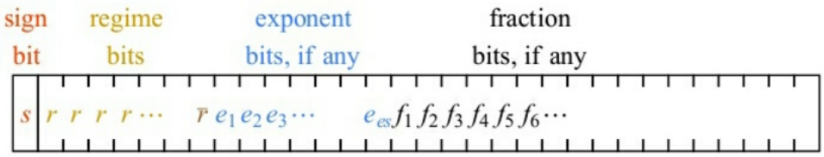
Type II Unums are based on the idea of the number line centered around zero with negative infinity on the far left and positive infinity on the far right with all real values on the line. This implementation uses the U-bit as described in above which is used in the least significant position. Values with a set ubit are shown as being in between exact values. As per most systems, the most significant bit is allocated as the sign bit with negative numbers being on the left half and positive on the right. Thus negating values can be seen as being flipped around the vertical axis. Two's complement suffices for the actual negation.

→ Projective real number line mapped to 4-bit two's complement integers




- **The Posits Format:**

  →Posit format for finite non-zero values:




- **Understanding an n-bit posit:**

  - **Sign bit:**

    The MSB of the n-bits represents the sign of the number represented. If it is 0, the number is positive and if 1 the number is negative. If the sign bit is 1, take 2's complement of the entire posit before decoding it.

## ● Regime Bits:

The identical bits after the sign bit either terminated by the opposite bit or the end of the binary string is reached contribute to the regime bits. Let there be 'm' identical bits in the regime.

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 |

**Scaling factor:**

The regime contributes to a scaling factor of useed , where useed = $2^{2^{es}}$

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

## ● Exponent bits:

Bits after regime bits are exponent bits. There can be up to es exponent bits. These are regarded as an unsigned integer. They represent a scaling factor of 2e.There is no bias as there is for floats.

## ● Fraction bits:

The bits left after the exponent bits represent the fraction, just like the fraction 1. If in a float, with a hidden bit 1. Posits doesn't support the denormalized format with hidden bit 0.

- **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:**

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

# ● Operations on Posits:

## 1) Addition/Subtraction :

### Algorithm:-

//write module

//check is there any exceptions, (if yes -> result accordingly)

//if no -> check if the sign bit of any input is 1?

// if yes-> take 2's complement of the corresponding input before decoding the posit. and then(if no)

//Decode the posits and separate the regime, exponent, and fraction bits of both inputs and normalize the fraction parts(mantissas)

//after that find the total exponent Exp1 and Exp2 contributed by both regime bits and exponent bits

//find the absolute diff. between the two exponents-> Diff.Exp=|Exp1-Exp2|

//is Exp1=Exp2, if no-> (Mantissa)lower exp >> Diff.Exp => Result Exp=max(Exp1,Exp2)

//then (if yes) add the mantissas and represent the result as normalized form by doing shifting operations and adjusting the result exp.., if necessary. Remove the leading 1 bit.

//Result.Regime=Result.Exp/(2^Es)

Result.exp=Result.Exp%(2^Es)

//Include regime, exp, and mantissa into the final result by shifting them to appropriate indexes.

//Result.

**Code:-**

```
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;
}
```

```
struct unpacked_t op2_add(struct unpacked_t a, struct unpacked_t b)
{
    if (a.neg == b.neg) {
        return add(a, b, a.neg);
    } else {
        return sub(a, b, a.neg);
    }
}
```

```
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;
}
```

```
struct unpacked_t op2_sub(struct unpacked_t a, struct unpacked_t b)
{
    if (a.neg == b.neg) {
        return sub(a, b, a.neg);
    } else {
        return add(a, b, a.neg);
    }
}
```

## 2)**Multiplication :**

### **Algorithm:-**

//write module

//check is there any exceptions, (if yes -> result accordingly)

//if no -> check if the sign bit of any input is 1?

// if yes-> take 2's complement of the corresponding input before decoding the posit. and then(if no)

//Decode the posits and separate the regime, exponent, and fraction bits of both inputs and normalize the fraction parts(mantissas)

//after that find the total exponent Exp1 and Exp2 contributed by both regime bits and exponent bits

//Result.Exp=Exp1+Exp2

Result.mantissa=mantissa1*mantissa2

//Normalize the Result.mantissa by shifting and adjusting Result.Exp, if necessary

//Result.Regime=Result.Exp/(2^Es)

Result.exp=Result.Exp%(2^Es)

//Include regime, exp, and mantissa into the final result by shifting them to appropriate indexes.

//Result

**Code:-**

```
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;
}
```

# 3) Division:

## Algorithm:-

//After n iterations the value in Quotient is the resultant quotient of the division of two mantissas.

//write module

//check is there any exceptions, (if yes -> result accordingly)

//if no -> check if the sign bit of any input is 1?

// if yes-> take 2's complement of the corresponding input before decoding the posit. and then(if no)

//Decode the posits and separate the regime, exponent, and fraction bits of both inputs and normalize the fraction parts(mantissas)

//after that find the total exponent Exp1 and Exp2 contributed by both regime bits and exponent bits

//Result.Exp=Exp1-Exp2

Result.mantissa=mantissa1/mantissa2

//Normalize the Result.mantissa by shifting and adjusting Result.Exp, if necessary

//Result.Regime=Result.Exp/(2^Es)

Result.exp=Result.Exp%(2^Es)

//Include regime, exp, and mantissa into the final result by shifting them to appropriate indexes.

//Result

## Code:-

```
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;
}
```

- # Exception of posits :

There are only two exceptions for posits:

- **Zero :** All n-bits are 0s.
- **± Infinity :** All the n-bits except the MSB are 0s. (1 followed by all 0 bits). There is no separate representation for positive and negative infinity.

## Concluding Remarks:

We have done posit number representation and additionally finded way for adding, subtracting and multiplying posit numbers in C. We have learnt how posit numbers work in real life and how accurate they are as compared to the floating point numbers.

## References:


- Beating Floating Point at its Own Game
- Better floating point
- https://posithub.org/docs/Posits4.pdf
- https://eescholars.iitm.ac.in/sites/default/files/eethesis/ee16m048.pdf
- https://ieeexplore.ieee.org/document/8731915
- https://www.sciencedirect.com/topics/engineering/exponent-bit


**—----------THE END----------**