University of Southampton
Faculty of Engineering and Physical Sciences Electronics and Computer Science

The Unum Number Format: Implementation and Comparison to IEEE 754 Floating-point number

by
Xiao Li

September 2022

Supervisor: Professor Mark Zwolinski
Second Examiner: Professor Stephen Gabriel

A dissertation submitted in partial fulfilment of the degree
MSc Microelectronics Systems Design

Abstract

Posit is designed as a replacement of IEEE-754 floating-point numbers. The aim of this project is to compare accuracy, power consumption, performance and area used between IEEE-754 single-precision floating-point format and 8-bit Posit. This paper demonstrates an approach to compute Posit in terms of addition and multiplication, and an assumption relevant to the Posit adder algorithm. This is a working theory which is based on 'Type 3' Unum, proposed by John L. Gustafson. With the use of simulation tool like Modelsim and synthesis tool like Quartus, the accuracy, power consumption, performance and area used between two number formats are compared. The result of the experiment indicates that 8-bit Posit is either saving energy and area or providing higher performance than IEEE-754 floating-point single-precision format, but the accuracy is lesser than the IEEE-754 floating-point number.

Keywords: Unum, Posit, IEEE-754 floating-point number, Posit adder, Posit multiplier, accuracy, power consumption, area, performance, energy saving

Acknowledgements

First, I would like to express my deep and sincere gratitude to Professor Mark Zwolinski for the assistance at every stage of this MSc project. Also, I am extending my thanks to Professor Stephen Gabriel, who gives me useful advises for the dissertation during the demonstration.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk

Contents

List of Figures 5
1 Introduction 6
2 Background 7
2.1 Type 1 8
2.2 Type 2 8
2.3 Type 3 Posit 9
2.4 Related work 11
3 Design 12
3.1 Posit adder 12
3.1.1 Stage 1: Posit data extraction 12
3.1.2 Stage 2: Posit core adder arithmetic 15
3.1.3 Stage 3: Posit data composition and rounding 17
3.2 Assumption 19
3.2.1 Algorithm 7: Core arithmetic (combine regime and exponent part together) 19
3.2.2 Algorithm 8 Posit composition and rounding 22
3.2.3 Algorithm 9: sticky_bit_adjustor 23
3.3 Posit multiplier 23
3.3.1 Algorithm 10 Proposed Posit Multiplier Computational Flow 23
4 Result and Analysis 26
4.1 Accuracy test and comparison 26
4.2 Cost comparison 31
4.3 Performance test 34
5 Conclusion 36
6 Reference 37
7 Appendix 38
7.1 Gantt chart 38
7.2 Code 38

List of Figures

Figure 2.1 32-bit floating-point number format 7

Figure 2.2 IEEE-754 double-precision energy cost 7

Figure 2.1.1 Type 1 Unum number format 8

Figure 2.2.1 Type 2 Unum number format circle 9

Figure 2.3.1 Type 3 Unum (Posit) number format 9

Figure 2.3.2 Posit regime value representation 9

Figure 2.3.3 Posit exponent value representation 10

Figure 2.3.4 right half circle from 3-bit to 5-bit posit with es=2 11

Figure 3.1.3.1 Posit data composition and rounding 18

Figure 4.1.1 Posit adder exceptional case: zero add zero 27

Figure 4.1.2 Posit assumption adder exceptional case: zero add zero (part1) 27

Figure 4.1.3 Posit assumption adder exceptional case: zero add zero (part2) 28

Figure 4.1.4 Posit adder exceptional case: adding infinity 28

Figure 4.1.5 Posit assumption adder exceptional case: adding infinity (part1) 29

Figure 4.1.6 Posit assumption adder exceptional case: adding infinity (part2) 29

Figure 4.1.7 Posit adder random test 30

Figure 4.1.8 Posit adder accuracy test 30

Figure 4.1.9 Floating-point adder accuracy test 31

Figure 4.2.1 Floating-point adder cost test 31

Figure 4.2.2 Posit adder cost test 32

Figure 4.2.3 Floating-point multiplier cost test 33

Figure 4.2.4 Posit multiplier cost test 33

Figure 4.3.1 Floating-point multiplier performance test 34

Figure 4.3.2 Posit multiplier performance test 35

Figure 7.1.1 Gantt chart 38

1 Introduction

The Unum is a superset of IEEE-754. Comparing with the IEEE-754, Unum has no overflow to infinity and no rounding to zero, which means fewer limitations [1]. There are two special representations of Unum: zero and infinity, which are represented by all bits with zero and all bits but the sign bit are zero respectively. By not including 'NaN' (not a number), for an N-bit FP mantissa, Posit saves $2 * (2^N - 1)$ 'NaN' combinations. Also, Unum obeys algebraic laws and is safe to compute in parallel. Moreover, it has fewer bits than the floating point. The universal number system was first published at the year of 2015 by John L. Gustafson. The 'Type 3 (Posit)' is the latest version of Unum. Interestingly, the Posit format is more similar to the IEEE-754 floating-point number format than 'Type 1' and 'Type 2'. Posit can provide better dynamic range and accuracy within the same bit width.

The reason why this project is useful is that using the Posit instead of floating-point might save energy or provide better performance. As is known to all, energy means money, and that is what the vendors care the most about. Energy saving is what makes this project meaningful.

The aim of this project is to compare accuracy, power consumption, performance and area used between IEEE-754 floating point number and Unum (Posit). As we all know, the IEEE-754 floating-point number is widely learnt and used in current design field. The floating-point arithmetic use numbers of specific fields to represent a real number. Take 32-bit floating-point number as an example, it consists of sign bit (1 bit), exponent width (8 bits) and mantissa (23 bits). The maximum value of the IEEE-754 32-bit floating-point variable is approximately 3.4028235×10^{38}. However, just using an 8-bit Posit, with es = 4, the maximum value of it could be 7.9228162×10^{28}.

In the rest of the dissertation, background information is first presented, including potential problems about floating point, a brief introduction of 'Type 1', 'Type 2' and 'Type 3' Unum. After that, the related work is presented to show what has been done. Second, the design chapter suggests the computational flow of Posit adder and Posit multiplier, providing detail explanations for each line in the flow. Third, the simulation results are shown, including exceptional cases (zero, infinity), random number test and Quartus synthesis.

2 Background

In the background section, it shows the floating- point number composition and the potential problems of the IEEE-754 floating-point numbers. More importantly, the development progress of the Unum is illustrated.

Nowadays, almost every hardware floating-point units are using the IEEE-754 standard format. The following figure shows an example layout for 32-bit floating-point number.

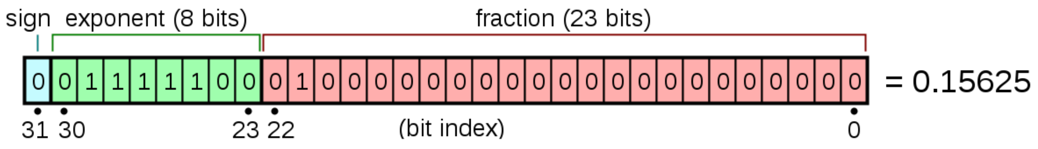
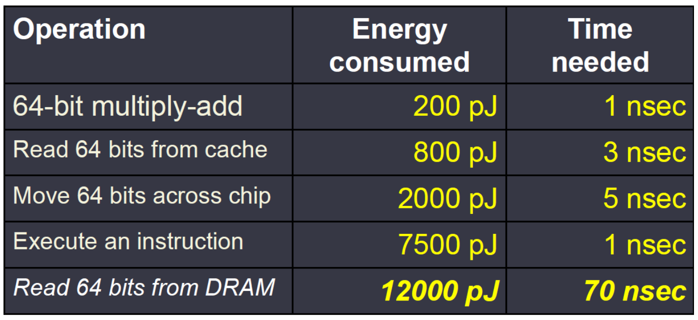


Figure 2.1 32-bit floating-point number format

0.15625_{10} is represented in binary as $0.00101_2 = \frac{1}{8} + \frac{1}{32}$, which can be written as: $1.01 * 2^{-3}$. Biased by 127, the exponent part -3 is stored as 127-3 = 124_{10} = 01111100_2. Leading one is not store for the mantissa.

Currently, several big problems we faced in terms of computing. John L. Gustafson proposes the following problems, which vendors care the most. First, the current calculation consumed too much energy and power. For example, huge heat sinks are needed to cool down the chip. When heat causing the design becomes bigger, which will increase the latency, therefore, limiting the speed of the whole design. The data centre needs loads of electric bills to maintain, the mobile device's battery life is a potential problem. Second, the bandwidth is not enough. The following diagram shows the one size fits all format like double-precision wasting energy, storage and bandwidth.



Operation	Energy consumed	Time needed
64-bit multiply-add	200 pJ	1 nsec
Read 64 bits from cache	800 pJ	3 nsec
Move 64 bits across chip	2000 pJ	5 nsec
Execute an instruction	7500 pJ	1 nsec
Read 64 bits from DRAM	***12000 pJ***	***70 nsec***

Figure 2.2 IEEE-754 double-precision energy cost

The energy cost of 'Read 64-bits from DRAM' under the frequency of 2GHz is 24 watts, which most of it cost for nothing. That is a clear weakness of this format. Third, floating-point prevent the use of parallelism computing. For instance, it seems wrong that $(a + b) + (c + d) \neq ((a + b) + c) + d$. However, computer programmers adopt serial rather than parallel, the IEEE-754 floating-point report rounding, overflow and underflow in the register that hide in the whole process without showing.

The Unum is developing from 'Type 1' to 'Type 3 (Posit)'. The 'Type 1' and 'Type 2' are introduced in detail at the following parts.

2.1 Type 1

Proposed in March 2015, the Unum (universal number) arithmetic system was first presented to the world at the year of 2013. The original version is titled “Type I” Unum, which is a superset of IEEE 754 standard floating-point format. In this version, it has a special bit at the end of the fraction called ‘ubit’, which indicates whether this number is an exact float or within an open interval between near floats. With the inclusion of ‘ubit’, rounding, overflow and underflow are eliminated. Instead of overflow, the ‘ubit’ marks the number as lying in the open interval between the largest number of the dynamic range and infinity [6]. Instead of underflow, the ‘ubit’ marks the number as lying in the open interval between zero and the smallest number in the dynamic range. There is a misunderstanding that higher precision will produce more accuracy. However, it cannot be guaranteed. The meaning of sign, exponent and mantissa field has the same definition compared with IEEE 754 standard. The bit width of the exponent and mantissa field can be changed manually by the user through varing the number of ‘esize’ and ‘fsize’. ‘utag’ is suggested to represent the combination of ‘ubit’, ‘esize’ and ‘fsize’. Adding ‘utag’ into the bit string can save total bit width, comparing with the floating-point. The format is as follows:

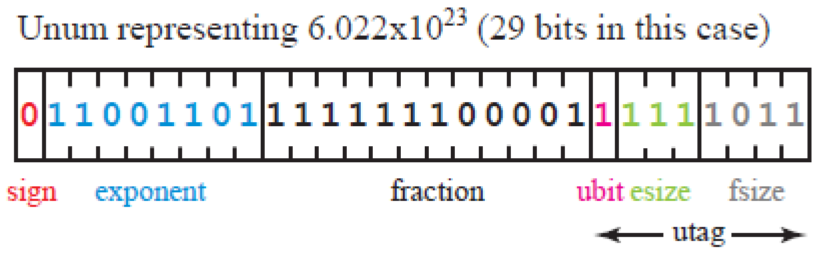


Figure 2.1.1 Type 1 Unum number format

Because the Unum has flexible dynamic range and precision, it eliminates the chance for the designer to select oversize data type with the aim of solving all the numerical problems. The “oversize fits all” data type obviously wastes storage and bandwidth. Since the “oversize fits all” precision is only needed for small amounts of calculations, designers choosing double precision format as an over-insurance seems not an optimum option.

2.2 Type 2

Introduced in January 2016, the ‘Type 2’ Unum is a brand-new version, which abandons compatibility with floating points, allowing an obvious, more efficient design based on the projective reals. All signed integers within the dynamic range are mapped around projective reals, from positive to infinity to negative numbers, with the same ordering. The structure of the 5-bit ‘Type 2’Unum is shown below in figure 2. The main principle of where the Unum number set is within the ‘Type 2’ circle, the upper left side, aligned 2^{n-3}-1 real numbers, which could be named as x_i. At the upper left-hand side, aligning the same amounts of numbers in negative value of x_i. Reflection through the horizontal axis, the lower half of the circle aligns reciprocals value of the upper half. ‘Type 2’ Unum inherits the idea of ‘ubit’ within ‘Type 1’ Unum. Ending 1 suggests an uncertain value between two real numbers, which represent by two Unum numbers with ending 0.

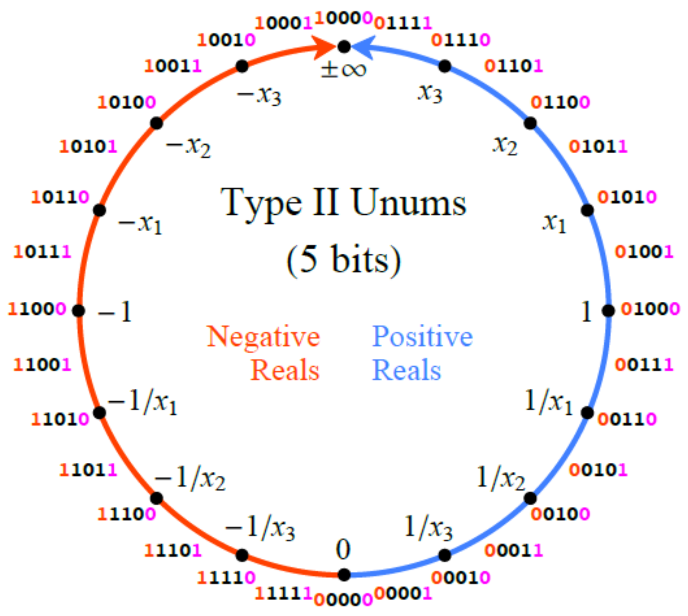


Figure 2.2.1 Type 2 Unum number format circle

2.3 Type 3 Posit

Posit is the latest version of Universal Number (Unum), which was published in February 2017. It can be seen as a potential counterpart of floating-point number system. Furthermore, Posit provides many advantages over floating-point, which include dynamic range flexibility, accuracy, smaller storage space and exact arithmetic computation. Both ‘Type 1’ and ‘Type 2’ Unum are using the idea of ‘ubit’. However, if the Unum number string keep adopting this uncertain function in every operation, it is better to use this last bit as an extra fraction bit in order to achieve more accuracy. ‘Type 3’ Unum is named as ‘Posit’. Merriam-Webster dictionary defines Posit as: to assume or affirm the existence. Posit is similar to ‘Type 2’ Unum in some extends. It is also a hardware-friendly version of Unum, which problems facing in ‘Type 1’ Unum due to changeable size are solved. The following graph shows the format of Posit.

$$\overbrace{s}^{sign}\quad \underset{run-length}{\overbrace{r\,r\,r\,\ldots\,\overline{r}}^{regime\,bits}}\quad \underset{max.\,ES\,bits}{\overbrace{e_1,e_2,\ldots,e_{es}}^{exponent\,bits,\,if\,any}}\quad \underset{remaining\,bits,\,if\,any}{\overbrace{m_1,m_2,\ldots}^{mantissa\,bits,\,if\,any}}$$

Figure 2.3.1 Type 3 Unum (Posit) number format

As it shows in the above figure, the Posit format can be divided into four parts: sign, regime, exponent and mantissa. Before anything to be written in code, the bit width (N) and exponent size (es) need to be defined first. The regime bits are varied during run-time, providing more flexible dynamic range and precision. Flexibility is a good technique to ensure Posit can be a useful number format that can be adopted by applications that have different groups of requirements, dynamic range and accuracy. The following figure shows how the regime bits work.

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	−4	−3	−2	−1	0	1	2	3

Figure 2.3.2 Posit regime value representations

The binary regime bits string begin with a couple of all zero or all one and terminate with one opposite number appearing in the next bit or if the string reaches the end. The number of identical bits can be represented as ‘i’. If the identical bit string is started with 1, then k = i – 1. Otherwise, k = -i. For example, in the above table, ‘0000’ is started with 0 to the end, therefore, i is equal to 4, and k = -i = -4. ‘10xx’ is started with 1, terminating with 0 at the second bit. In that situation, leaving the remaining bit ‘xx’, thus, ‘i’ is equal to 1, k = i – 1 = 0. The regime value k suggests a power factor of useed^k. ‘useed’ is a scale factor related to the exponent bit, which is represented by ‘useed’ = $2^{2^{es}}$. The following figure shows the example of the ‘useed’ value.

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

Figure 2.3.3 Posit exponent value representations

The next bit field is the exponent bits, which are unsigned integers. Different from the IEEE-754 floating-point numbers, Posit does not need bias in the final exponent calculation. As it is defined before, there can be up to es exponent bits, but it will depend on how many bits are left after the regime bits. This is a good technique to demonstrate tapered accuracy, which means numbers around one have more accuracy than extremely huge or small numbers.

If there are any bits remaining besides sign bit, regime bits and exponent bits, it is on behalf of the mantissa bits. Due to this reason, a short bit width may lose some accuracy. This part is the same compared with the IEEE-754 floating-point number. It could be illustrated by the format of ‘1.mantissa’, with a hidden bit 1 always in front of the mantissa bits. Subnormal numbers with a hidden bit 0 in floats cannot exist in the Posit number format.

The Posit value can be calculated by the following equation, including the sign bit (s), regime value (k), exponent value (e), and mantissa value (f), which includes a hidden bit 1.

$$\text{Posit value} = s * (2^{2^{es}})^k * 2^e * f \qquad (1\text{-}1)$$

If the remaining bits are lesser than ‘es’ bits, then those bits will be treated as MSB of the exponent field. Take N = 8, es = 4 Posit as an example, for the bit string 01111101, sign bit is 0, the regime sequence is 111110, therefore, only one bit left in the exponent field, which will be treated as 1000 by adding zeros on the right side. Hence, the Posit value is equal to $+ (2^{2^4})^4 * 2^8 * 1$ = 4,722,366,482,869,645,213,696.

Posit can be illustrated by a circle populating numbers, and the posit precision can be increased by adding extra bits. Appending bits with the value of zero will make no difference to the former value, new bit string remains where it is on the circle as before. However, adding ones to the bit string will create new values between two original values on the circle. The following figure shows only the right half of the circle from 3-bit to 5-bit posit with es = 2.

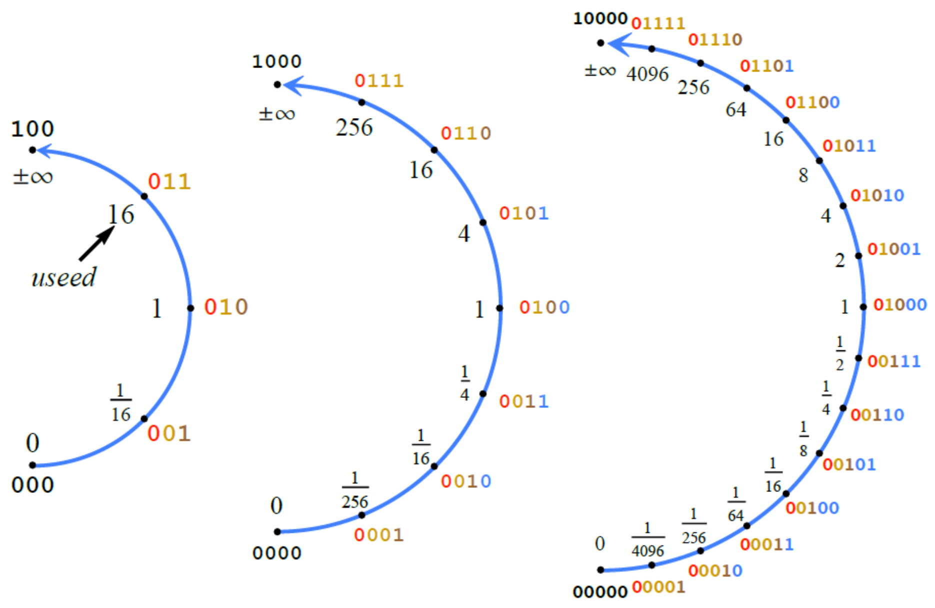


Figure 2.3.4 right half circle from 3-bit to 5-bit posit with es=2

Using simple example of multiplication can express why Posit can reduce energy consumption. Take -1024 * 0.046875 as an example, result is easily calculated, which is -48. If use IEEE-754 single-precision floating-point format, this equation can be transformed to 1_10001001_00000000000000000000000 * 0_01111010_10000000000000000000000 = 1_10000100_10000000000000000000000. However, when calculate in Posit, which is 8-bit width with es=4, this equation can be interpreted as 1_01_0110_0 * 0_01_1011_1 = 1_01_1010_1. It is obvious to discover that number format wastes lots of bits and the mantissa part, which will cost power for nothing.

2.4 Related work

The universal number system has undergone a progressive evolution since 'Type 1' is presented [2] – [4]. 'Type 2' [5], [6]. Since Posit was proposed, there were some papers attracting people's attention, increasing their interest like [1], [2], [6]. The Posit number system can be seen as a simple replacement of IEEE-754 floating-point format [7]. Numbers of articles were writing about the hardware implementation of the Posit [8] – [11]. A parameterized Posit arithmetic generator, including floating-point to Posit converter, Parameterized leading one detection (LOD), leading zero detection (LZD), dynamic left shifter (DLS), Posit to FP converter, Posit adder/multiplier computational flow is presented in [8]. Similar problem is not solved in the later paper [11]. They both lack the details about rounding, which is just mentioned by 'round-to-zero' or barely noticed. This is an important factor that will cause the reduction of accuracy. [9], [10] present a fixed width 32-bits Posit adder and Posit multiplier. The recent article [12] mentioned the details about rounding, which makes the design more rigorous.

3 Design

In this chapter, the details about the computational flow of Posit adder and Posit multiplier are presented. Also, there is an assumption about changing the algorithm into another form, which might change the total cost of the Posit adder.

3.1 Posit adder

In this part, a detail description of parameterized computational flow of a Posit adder is proposed. The flow can be further subcategorized as data extraction, leading one detection (LOD), dynamic left/right shifter (DLS/DRS), core arithmetic flow, computation of final regime and exponent, rounding and Posit construction. Basically, this computational flow can be categorized into three stages: Posit data extraction, Posit core arithmetic, rounding and Posit construction.

3.1.1 Stage 1: Posit data extraction

Posit data extraction is the first step of the whole algorithm, which is presented in Algorithm 1. The algorithm is executed by first setting the parameter, including Posit width (N) and exponent bits (es). Because the regime bits are varied during run-time, it can be up to (N-1) bits except for the sign bit, which can be stored in 'Bs' bits (Bs = $\log_2 N$). The given inputs are 'in1' and 'in2', first and foremost is to check whether those two inputs are special cases like zero and infinity. The binary representation of those two special cases are mentioned in chapter 1. After checking for the special cases, data could be extracted from two inputs. The extractor extracts signs, regimes, exponents and mantissa. However, there are certain rules about extraction. The extraction flow is as follow (line 3 – 9). First, extracting the MSB of both inputs as sign bits and storing in the register named s1 and s2 (line 12). Second, if the sign bit is equal to 1, it indicates the input is a negative operand, otherwise, the input is positive. On the one hand, for negative inputs, all the remaining bits are encoded in 2's complement form in the Posit number system. On the other hand, in terms of positive inputs, it will remain unchanged (line 13). Third, according to the Posit number format, after the sign bit are the regime bits. The remaining bits of 'in1' and 'in2' will be stored in (N-2) bit registers 'xin1' and 'xin2' respectively. The first bit of 'xin' regime check (RC) is used for checking the regime bits are beginning with one or zero. Therefore, following the explanation in chapter 1, the regime value is decided. However, to execute in hardware, it requires the LZD and LOD technique to find the terminating bit and the regime left shift amount 'Lshift'. However, if the regime check (rc) is equal to 1, 'xin' can be complemented to avoid using both LOD and LZD, therefore, same more hardware (line 14 - 17). Finally, in order to extract the exponent and mantissa bits, numbers in the 'xin' registers need to be left shifted by 'LOP + 1' bits, which means all the regime bits are flushed out (line 18). Obviously, now the MSB es bits are the exponent bits and all the other bits are the mantissa bits (line 19 -20). The details of the data extraction are shown below in Algorithm 1:

3.1.1.1 *Algorithm 1 Proposed Posit Data Extraction Flow*

```
Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size)

Inputs: in1, in2

Outputs: s1, s2, rc1, rc2, regime1, regime2, exp1, exp2, mant1, mant2, zero, inf

1: Check for exceptions (infinity (inf), zero (zero))

2: Check for infinity

3: inf1 = in1[N-1] & ~(|in1[N-2:0]), inf2 = in2[N-1] & ~(|in2[N-2:0])     // all bits except sign bit are 0

4: inf = inf1 | inf2

5: Check for zero

6: zero1 = |in1, zero2 = |in2                                              // all bits are 0

7: zero = zero1 & zero2

8: Data extraction: sign (s), remaining bits (xin), regime check (rc), regime value (regime), exponent (exp), mantissa (mant).

9: Extraction from in1:

10: s1 = in1[N-1]

11: xin1 = s1 ? ~in1[N-2:0] : in1[N-2:0]                                   // 2's Complement if sign is 1

12: rc1 = xin1[N-2]                                                        // Regime check

13: xint = rc ? ~xin : xin

14: LOP = Leading one detection (LOD) (xin1[N-2:0])

15: regime1 = rc1 ? LOP-1 : ~LOP (Effective Regime Value)

16: xin = xin << LOP + 1 (Flush out regime sequence)

17: exp1 = MSB es-bits of xint (Exponent)

18: mant1 = remaining bits of xint (do not forget the hidden bit)

19: Extraction from in2: s2, xin2, regime2, exp2, mant2
```

In the data extraction algorithm, leading one detector (LOD) finds the position of leading one bit (LOP) from ‘xint’. The detail of the LOD algorithm is shown below:

3.1.1.2 *Algorithm 2 Leading One Detector (LOD)*

Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size)

Inputs: A (the width of A is donated by L_A)

Outputs: Position of leading one (LOP)

```
1: Set default value: LOP = 0 (lead one position initialization), t = 0, found = 0
2: for i = 0 to L_A – 1
3:     if (A[L_A - i -1]) == 1
4:         found = 1
5:     t = t | found
6:     if (t == 0)
7:         LOP = LOP + 1
```

After the value of LOP is known, ‘xin’ needs to be left-shifted by ‘LOP + 1’ bits through the use of dynamic left shifter. In the next algorithm, a parameterized dynamic left shifter (DLS) is presented. Following the given Posit size and the shifting amount ‘LOP’. Using the basic principle of barrel shifter, which requires ‘LOP’ numbers 2 to 1 multiplexers with the bit width of (N-1). Algorithm 3 below shows the detail of DLS.

3.1.1.3 *Algorithm 3 Parameterized Dynamic Left Shifter (DLS)*

Given parameters: N: Posit width, Bs: log2 (N) (Posit regime value storage size)

Inputs: in[N-1:0], b[Bs-1:0]

Output: OUT

```
[N-1:0] t [Bs-1:0]
1: t[0] = b[0] ? in << 7’d1 : in;
2: genvar i
3: generate
4: for (i=1; i<Bs; i=i+1) begin
5:     t[i] = b[i] ? (t[i-1] << 2**i) : t[i-1]
6: end
7: endgenerate
8: OUT = t[Bs-1]
```

With the help of LOD and DLS the data extraction process can be completed, all the variables for further calculation are extracted.

3.1.2 Stage 2: Posit core adder arithmetic

3.1.2.1 Algorithm 4 Posit Adder Core Arithmetic Flow

```
Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size)
Inputs: s1, s2, rc1, rc2, regime1, regime2, exp1, exp2, mant1, mant2
Outputs: exp_o, e_o, r_o, add_m
1: large checking and assignment
2: op =  s1 xor with s2                                   // Effective operand
3: Check for large input and small input
4: in1 greater than in2 = xin1 >= xin2 ? 1 : 0                    // Operands comparison
5: Large (l) component: ls, lrc, lr, le, and lm
6: ls = in1 greater than in2 ? s1 : s2
7: lrc = in1 greater than in2 ? rc1 : rc2
8: lr = in1 greater than in2 ? regime1 : regime2
9: le = in1 greater than in2 ? e1 : e2
10: lm = in1 greater than in2 ? m1 : m2
11: Small (s) component: src, sr, se, and sm
12: ss = in1 greater than in2 ? s2 : s1
13: src = in1 greater than in2 ? rc2 : rc1
14: sr = in1 greater than in2 ? regime2 : regime1
15: se = in1 greater than in2 ? e2 : e1
16: sm = in1 greater than in2 ? m2 : m1
17: Regime difference: r_diff, r_diff0, r_diff1, r_diff2
18: r_diff0 = lr - sr
19: r_diff1 = lr + sr
20: r_diff2 = sr – lr
21: r_diff = lrc ? (src ? r_diff0 : r_diff1) : r_diff2
22: effective exponent difference: diff, exp_diff                // lower mantissa shift amount
```

```
23: diff = {r_diff,le} – {Bs+1{1'b0},se}
24: exp_diff = (|diff[Bs+es:Bs]} ? {Bs{1'b1}} : diff[Bs-1:0]
25: sm_tmp  = sm >> exp_diff                          // right shift sm by exp_diff amount
26: add lm and sm_tmp                                 // Mantissa addition/subtraction
27: add_m = op ? lm - sm_tmp : lm + sm_tmp
28: Movf  = add_m[MSB]                                // Mantissa overflow check
29: add_m = Movf ? add_m : add_m << 1
30: Normalization of add_m:
31: Nshift = LOD of add_m                             // Check for Leading-One
32: add_m = add m << Nshift                           // Dynamic left shifting by Nshift
33: Final exponent (e_o) and regime (r_o) computation:
34: exp_o  is the concatenation of lr, le, movf and Nshift
35: lr1 = lrc ?  lr : -lr
36: exp_tmp = {lr1, le} - Nshift
37: exp_o = exp_tmp + Movf
38: exp_o1 = exp_o[MSB] ? – exp_o  : exp_o
39: e_o: based on +/- of exp_o, compute LSB es bits from exp_o
40: e_o = (exp_o[MSB] & | exp_o1[es-1:0]) ? exp_o[es-1:0] : exp_o1[es-1:0]
41: r_o:  based on +/- of exp_o, compute MSB Bs bits from exp_o
42: r_o = ~ (exp_o[MSB]) | (exp_o[MSB] & (exp_o1[es-1:0])) ? exp_o1[es+Bs-1:es] + 1 :
exp_o1[es+Bs-1:es]
```

This algorithm is used for calculating the mantissa part of the Posit. In order to attain the process, several values need to be determined. First of all, whether the mantissa calculation is addition or subtraction is related to the sign between two Posit numbers. The effective arithmetic operation (op) between two mantissa inputs is decided by a XOR operation between two operands' sign bit, if op = 1, which means the sign of two Posit numbers are opposite, therefore, a subtraction operation is operated. Otherwise, an addition operation is executed (line 2). Second, to perform a mantissa arithmetic operation, which input is larger than the other one needs to be find out first. The easiest way to achieve that is to compare the value of 'xin1' and 'xin2'. Therefore, the sign, regime, exponent and mantissa of the larger input will be assigned to 'ls', 'lrc', 'lr', 'le', 'lm' respectively. And the smaller input's components will be assigned to 'ss', 'src', 'sr', 'se', 'sm' respectively (line 3 - 16). Third, after decided all the components, the smaller mantissa needs to be right-shifted to achieve decimal alignment. The right-shift amount is determined by the difference of effective exponent, which contains the difference of the regime value and the exponent value (line 17 - 24). Algorithm 5 below is a parameterized dynamic right shifter, which has the same basic idea used in Algorithm 3.

3.1.2.2 Algorithm 5 Parameterized Generation of Dynamic Right Shifter (DRS)

```
Given parameters: N: Posit width, Bs: log2 (N) (Posit regime value storage size)
Inputs: in[N-1:0], b[Bs-1:0]
Output: OUT
[N-1:0]t[Bs-1:0]
1: t[0] = b[0] ? in >> 1 : in;
2: genvar i
3: generate
4: for (i=1; i<Bs; i=i+1) begin
5:   t[i] = b[i] ? (t[i-1] >> 2**i) : t[i-1]
6: end
7: endgenerate
8: OUT = t[S-1]
```

The arithmetic operation of the aligned small mantissa and the larger mantissa can be calculated using an adder or a subtractor unit (line 26 - 27). By detecting the MSB of the adding result, which will be evaluated if there is an overflow (Movf), if there is an overflow, the value of final total exponent will be increased by 1. Otherwise, the 'add_m' mantissa will be left-shift by one bit. This step requires a two to one multiplexer (line 28 - 29). In terms of the subtraction operation for the mantissa, if two number does not have huge difference, it might affect the result since the 'add_m' will lose several MSB bits. Due to this reason, a stage of normalization is required, which needs a LOD on the 'add_m' to get the shift amount for later left shifting (line 31 - 32). After the process of normalization, the value of final regime value (r_o) and final exponent (e_o) are calculated and performed. By calculating the effective exponent (exp_o), the first step is to check the sign of 'lrc', if it is true, then keep the former value. Otherwise, convert it into 2's complement. Then combining with the large exponent 'le', subtract it by the underflow amount (Nshift) and add it by the mantissa overflow (Movf) (line 33 - 37). If the sign of 'exp_o' is equal to 1, it is converted into 2's complement as exp_o1 (line 38). The final regime (r_o) and final exponent (r_o) are calculated in line 39 – 42.

3.1.3 Stage 3: Posit data composition and rounding

When every final component is ready, this stage is to compose them all together and perform rounding. The detail of the composition is demonstrated in the following figure.

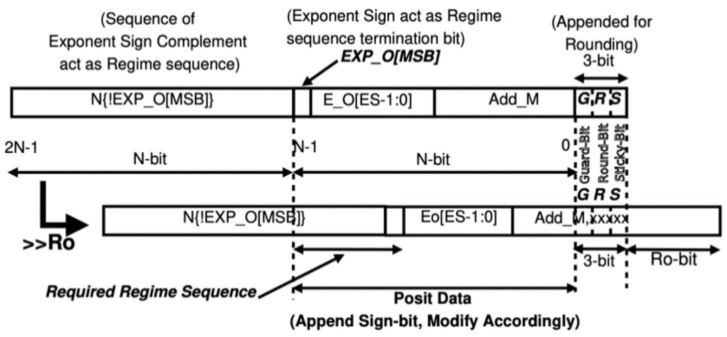


Figure 3.1.3.1 Posit data composition and rounding

According to the figure, the Posit is first composed of the complement of the sign bit of the effective exponent (exp_o) at the MSB N-bit act as regime sequence. Interestingly, this arrangement creates a regime sequence that is desired, which fits the regime value calculation. Because when the regime is beginning with 1, the exponent will be positive, on the contrary, the exponent will be negative. Therefore, using the complement of sign bit of the real exponent can act as regime sequence. Furthermore, in calculating the regime value process, the termination bit is the opposite value as the former bits, hence, the sign bit without complement will act as a regime sequence termination bit. The next es bits are from the final exponent output (e_o), after that is followed by the mantissa part 'add_m'. Finally, three bits are initialized to zero appending at the LSB of the Posit sequence, which are guard bit (G), round bit (R) and sticky bit (S) aimed for rounding (line 2). The details of the Posit composition and rounding algorithm are presented at Algorithm 6.

3.1.3.1 Algorithm 6 Posit composition and rounding

```
Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size)
Inputs: ls, r_o, exp_o, e_o, add_m, zero, inf
Outputs: out
1: Posit data composition: regime bits, exponent bits and mantissa bits packing:
2: REM = {N{!exp_o[MSB]}, exp_o[MSB], e_o, add_m, 3'b0}    // remaining bits except for sign bit
3: REM = REM >> r_o                                         // right shift by r_o btis
4: G = REM[2]
5: R = REM[1]
6: S = REM[0]
7: $L_B$ = REM[3]
8: Rounding: round to nearest even
9: If (r_o < N-es-2)
10:    ulp_add = (G & (R | S)) | ($L_B$ & G & (!(R|S)))
11:    REM =  REM + {(N-1) {1'b0}, ulp_add}
```

12: REM_N = ls ? -REM : REM

13: Combine ls with N-2 bits of rounded REM to perform Output while considering exceptions (infinity and zero):

14: out = inf ? {inf, {N-1 {1'b0}}} : (zero ? {N {1'b0}} : {ls, REM_N})

In the step of rounding, there is only one method for Posit arithmetic, which is round to the nearest even. To use that technique, rounding needs to generate a 'ulp_add' bit using the combination of variables like: LSB of the mantissa (L_B), the next bit is guard bit (G), the next bit is round bit (R), the next bit is sticky bit (S) (line 4 - 8). If r_o is lesser than N-es-2, then through a few numbers of logic gates, executing the equation like line 11, the 'ulp_add' bit is added to REM (line 12). There might be a problem when using the principle at [3] to decide the value of sticky bit, since in that article the sticky bit is determined by OR of remaining all bits as sticky bit, but in SystemVerilog it is wrong to make a register's width an inconstant value like r_o. therefore, Algorithm 9 in the assumption might be a solution to this problem. Then the final REM, which is composed of the regime part, exponent part and the mantissa part. Furthermore, for true sign bit (ls == 1), REM is 2's complemented, otherwise, it will remain unchanged (line 13). The exceptions check is taken to make sure they will not affect the final output. For the case of zero, all the output bits are 0. And for the infinity case, all the output bis are 0, except for the sign bit. In normal cases, the output is the concatenation of 'ls' and 'REM' (line 14 - 15).

3.2 Assumption

It is interesting to make an assumption that when combining the regime and exponent part together through the calculation, it might save some space that will cause lesser area, which will lead to lesser delay. There are some slightly differences about the data extraction stage, where the regime bits and the exponent bits are combined together to perform effective exponent 'eff_e'. The most important part of this assumption is in stage 2: core arithmetic (Algorithm 7).

3.2.1 Algorithm 7: Core arithmetic (combine regime and exponent part together)

Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size)

Inputs: sign bit (s1, s2), regime part (regime1, regime2), exponent part (exp1, exp2), mantissa part (mant1, mant2), effective exponent (eff_e1, eff_e2)

Outputs: mantissa with higher effective exponent is assigned to M_B, mantissa with lesser effective exponent is assigned to M_S, added mantissa M_I (by adding M_B and M_S), final mantissa C, higher effective exponent is assigned to Eeff

// Finding difference of regimes

regime difference: r_diff1 = regime1 – regime2, r_diff2 = regime2 – regime1, r_diff = r_diff1

// Finding difference of exponents

exponent difference: exp_diff1 = exp1 – exp2, exp_diff2 = exp2 – exp1, exp_diff = exp_diff1

```
a = 0
Sr1 = regime1[MSB]; Sr2= regime2[MSB]                              // Sign of regimes
if (Sr1 ^ Sr2 == 1)                                                // Regimes are of different sign
   if (Sr1 == 0)                                                   // If input 1 is positive
      $M_B$ = mant1; $M_S$ = mant2; Eeff = eff_e1
   else                                                            // If input 2 is positive
      $M_B$ = mant2; $M_S$ = mant1; Eeff = eff_e2; a = 1; r_diff = r_diff2
else                                                               // Regimes are of same sign
   if (|r_diff [Bs:0] == 0)                                        // If two regimes have the same value
      if (exp_diff[MSB] == 1)                                      // If exp1 > exp2
         $M_B$ = mant1; $M_S$ = mant2; Eeff = eff_e1
      else                                                         // If exp1 < exp2
         $M_B$ = mant2; $M_S$ = mant1; Eeff = eff_e2; a = 1; r_diff = r_diff2
   else
      if (r_diff[msb] == 1)                                        // If regime1 > regime2
         $M_B$ = mant1; $M_S$ = mant2; Eeff = eff_e1
      else                                                         // If regime1 < regime2
         $M_B$ = mant2; $M_S$ = mant1; Eeff = eff_e2; a = 1; r_diff = r_diff2
if (a == 1)
   exp_diff = exp_diff2
if (exp_diff[MSB] == 0)
   if (|r_diff == 1)
      r_diff = r_diff - 1
   else
      exp_diff = exp_diff2;
shift = {r_diff[Bs:0], exp_diff[es-1:0]}
if (shift >= N-es+3)
   $M_S$ = 0; shift = 0
else
   $M_S$ = sticky_bit_adjustor($M_S$, shift)
$M_B$ = {0, $M_B$}; $M_S$ = {0, $M_S$}
if (s1 ^ s2 == 1)
```

```
    if (s1 == 1)
        $M_B$ = -$M_B$
    else
        $M_S$ = -$M_S$
else
    if (a == 0)
        $M_S$ = -$M_S$
    else
        $M_B$ = -$M_B$
// adding two mantissas
$M_I$ = $M_B$ + $M_S$
// determine the final sign bit $S_F$ and mantissa C
$S_F$ = (s1 ^ s2) ? (!$M_I$[MSB]) : s1
C = (s1 ^ s2) & $S_F$ ? -$M_I$ : $M_I$
```

At this stage, the first step is to find the difference of both the regime and the exponent. Using outputs from the data extraction stage, the regime difference and the exponent difference are calculated in parallel. The regime and exponent of 'in1' are subtracted by the regime and exponent part from 'in2' irrespective of which one has the higher value. Because this result could be adjusted to make sure the effective exponent has an overall positive value. Based on the sign of the regime, the bigger mantissa and smaller mantissa are assigned to M_B and M_S respectively. If two regimes' signs are opposite, the operand has the positive regime value assigned their mantissa, effective exponent part to M_B and Eeff. Another operand's mantissa part is assigned to M_S. 'a' is assigned the value of 1 and 'r_diff' is 2's complemented to convert it into a positive value when 'regime2' has a positive value. In the circumstance that 'Sr1' and 'Sr2' have the same sign, then based on the value of 'r_diff' and 'exp_diff', the value of 'M_B', 'M_S', 'Eeff', 'a', 'r_diff' can be determined. If two regimes have the same value (|r_diff = 0), then based on the MSB of exp_diff, the value of 'M_B', 'M_S', 'Eeff' is assigned. When 'exp1 < exp2', then 'mant2', 'eff_e2' and 'mant1' are assigned to 'M_B' , 'Eeff' and 'M_S' respectively. Moreover, 'a' is set to 1 and 'r_diff' is converted into 2's complement to become a positive value. When 'exp1 > exp2', then 'mant1', 'eff_e1' and 'mant2' are assigned to 'M_B' , 'Eeff' and 'M_S' respectively. If two regime values are not the same (r_diff $\neq$ 0), then if r_diff[MSB] = 0, which means the effective exponent value of 'in1' is greater than 'in2', thus, 'mant1', 'eff_e1' and 'mant2' are assigned to 'M_B' , 'Eeff' and 'M_S' respectively. if r_diff[MSB] = 1, this implies the effective exponent value of 'in1' is lesser than 'in2', then 'mant2', 'eff_e2' and 'mant1' are assigned to 'M_B' , 'Eeff' and 'M_S' respectively, 'a' is set to 1 and 'r_diff' is converted into 2's complement to become a positive value. After the former process is finished, if 'a' is set to 1 , the 'exp_diff' is converted into 2's complement. If 'exp_diff' is a negative value, 'r_diff' and itself need to be adjusted. If exp_diff[MSB] = 1 and r_diff is not equal to zero, 'r_diff' is decreased by 1, else 'exp_diff' is converted into 2's complement form.

After the above part of the algorithm, the 'r_diff' and the 'exp_diff' are both positive values, the alignment can be proceed using the sticky bit adjustor at Algorithm 7. The 'shift' value can be determined only by the concatenation of ‘r_diff’ and ‘exp_diff’. The maximum shift amount can be limited by the value of N-es+3. If the shift amount is greater than the maximum value, both the shifted mantissa ‘M_S’ and shift are set to zero. Otherwise, ‘M_S’ is shifted by ‘shift’ amount using Algorithm 9. Based on the arithmetic operation and the value of ‘a’, both ‘M_B’ and ‘M_S’ need to be adjusted. For instance, if the operation is addition, if ‘s1’ and ‘a’ are both equal to zero, ‘M_S’ is converted into its 2’s complement. Further details are illustrated in Algorithm 8. After decided the value of ‘M_B’ and ‘M_S’, ‘M_I’ is equal to the result of ‘M_B’ + ‘M_S’. Using the M_I[MSB] and ‘s1’ the final sign ‘S_F’ and mantissa C is performed, and C could act as the ‘add_m’ in Algorithm 4, which can be further used in normalization, data composition and rounding.

For the data composition and rounding, the normalized mantissa ‘C’ is rounded, and the final output ‘out’ is generated. The methodology is similar to the rounding technique in Algorithm 6, except for using Algorithm 9. The sticky bit adjustor right-shifted the input and adjusts the sticky bit (LSB). The sticky bit is initialized to 1, and the input A is padded with L_{pad} numbers of zero at the right-hand side to make ‘t’ fits the bit width. Then the padded input is right-shifted by ‘B’ amount. The sticky bit is set to 0 if there are no bits with the value of one in the right -shifted part. Finally, remove the padded bits and update the LSB bits if the sticky bit is equal to 1.

3.2.2 Algorithm 8 Posit composition and rounding

```
Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size)
Inputs: $S_F$, r_o, exp_o, e_o, C, zero, inf
Outputs: out
1: Posit data composition: regime bits, exponent bits and mantissa bits packing:
2: Tp = {N{!exp_o[MSB]}, exp_o[MSB], e_o, C}.          // remaining bits except sign bit
3: Tp = sticky_bit_adjustor(Tp, r_o)                   // right shift by r_o btis
4: G = Tp [2]
5: R = Tp [1]
6: S = Tp [0]
7: $L_B$ = Tp [3]
8: $I_P$ = Tp [N+1:3]
8: Rounding: round to nearest even
9: If (r_o < N-1)
10:     ulp_add = (G & (R | S)) | ($L_B$ & G & (!(R|S)))
11:     $I_P$ = $I_P$ + ulp_add
12: $I_P$ = $S_F$ ? - $I_P$ : $I_P$
```

13: Combine ls with N-2 bits of rounded REM to perform Output while considering exceptions (infinity and zero):

14: out = inf ? {inf, {N-1 {1’b0}}} : (zero ? {N {1’b0}} : { S_F, I_P})

3.2.3 Algorithm 9: sticky_bit_adjustor

Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size), width of the pad (L_{pad}), width of the input A (L_A)

Inputs: A, B

Output: C

```
sticky bit adjustor
begin
    sticky_bit = 1
    t = {A, $L_{pad}${1′b0}}
    t = t >> B
    if t[$L_{pad}$ −1:0]==0
        sticky_bit = 0
    sticky_bit = sticky_bit|t[$L_{pad}$]
    C = {t[$L_A + L_{pad} − 1 : L_{pad} + 1$], sticky_bit}
end
```

3.3 Posit multiplier

The Posit multiplier is very similar to the Posit adder in terms of the algorithm. They both adopt 3 processing stages, which are data extraction, core arithmetic, data composition and rounding. The algorithm for the Posit multiplier computational flow is shown in Algorithm 9.

3.3.1 Algorithm 10 Proposed Posit Multiplier Computational Flow

Given parameters: N: Posit width, es: Posit exponent size, Bs: log2 (N) (Posit regime value storage size)

Inputs: in1, in2

Output: out

1: Posit Data Extraction: Algorithm 1

2: Data from in1: xin1, s1, rc1, regime1, exp1, mant1, inf1, zero1

```
3: Data from in2: xin2, s2, rc2, regime2, exp2, mant2, inf2, zero2
4: Check for zero and infinity: zero = zero1 | zero2, inf = inf1 | inf2
5: Posit Multiplier Core Arithmetic:
6: Sign determination:
7: mult_s = s1 xor s2
8: Mantissa Multiplication Processing:
9: mult_m =  mant1 * mant2                              //Mantissa multiplication
10: mult_m_ovf = mult_m[MSB]                            //Check mantissa overflow
11: mult_mN = mult_m_ovf ? mult_m : mult_m << 1         //1-bit mantissa shifting for overflow
12: Effective regime computation:
13: regime1 = rc1 ? regime1 : -regime1                  //Effective regime1 value
14: regime2 = rc2 ? regime2 : -regime2                  //Effective regime2 value
15: Final exponent (e_o) and regime (r_o) processing:
16: Using full-adder to calculate total exponent value:
17:  mult_e[es+Bs+1:0] = {regime1, exp1} + {regime2, exp2} + mult_m_ovf
18: mult_eN[es+Bs:0] = mult_e[es+Bs+1] ? - mult_e: mult_e      // Absolute value of total exponent
18: e_o[es-1:0] = mult_e[es-1:0]                        // Exponent output
20: r_o[Bs:0] = !( mult_e[es+Bs+1]) | (|mult_eN[es-1:0]) ? mult_eN[es+Bs:es] + 1'b1 :
mult_eN[es+Bs:es]                                       // Absolute regime value
21: Posit Construction, Rounding and Final Processing:
22: Using the same strategy as Algorithm 6: regime, termination bit, exponent, mantissa, GRS
23: Posit Data Composition:
24: Regime bits, exponent bits and mantissa bits packing:
25: REM = {{N{!mult_e[MSB]}}, mult_e[MSB],e_o, mult_mN[MSB N-es-1 bits], mult_mN[Next 2
bits], 3'b0}
26: REM = REM >> r_o
27: Rounding: round to nearest even
28: If (r_o < N-es-2)
29: ulp_add = (G & (R | S)) | (L & G & (!(R|S)))
30: REM = REM + {(N-1) {1'b0}, ulp_add}
31: REM_N = ls  ? -REM : REM
32: Final Output:
```

```
33: Combine ls with N-2 bits of rounded REM to perform Output while considering exceptions
(infinity and zero):
34: out = zero ? {N{1'b0}} : (inf ? {inf,{N-1{1'b0}}} : {mult_s, REM_N[N-1:1]})
```

The Posit multiplication data extraction stage is mostly the same as in Algorithm 1. The extracted datas are including sign bits (s1, s2), 2’s complemented form inputs (xin1, xin2), regime check bits (rc1, rc2), absolute regime values (regime1, regime2), exponent bits (exp1, exp2), mantissa bits (mant1, mant2), the flags of zero and infinity (zero1, zero2, inf1, inf2) (line 2 - 3). What is slightly different from algorithm 1 is the zero flag. Because in the multiplication process, only one operand equal to zero could cause the result to be zero, therefore, the zero flag here should be: ‘zero = zero1 | zero2’ (line 4).

The extracted data are then processed by the core arithmetic unit. The final sign bit is computed by using a 1-bit XOR gate among s1 and s2 (line 7). The mantissa multiplication processing use a multiplier to calculate ‘mant1 * mant2’ and stored in the result register ‘mult_m’ (line 9). The result is then checked for overflow by detecting the MSB bit of ‘mult_m’ (line 10). If multiplication result is not overflowed, the result mantissa is left-shifted by 1-bit for later normalization (line 11). The final regime and exponent computation start with computing the absolute regime value using the regime check bit (line 13 - 14). Combining with the exponent bits (exp1, exp2), these absolute regime values are then added with ‘mult_m_ovf’ to provide the final output effective exponent ‘mult_e’ for each input (line 17). Using the same strategy in Algorithm 4, the absolute total exponent value and ‘mult_eN’, final regime value ‘r_o’ and final exponent value ‘e_o’ is computed in line 18 – 20. After the core arithmetic unit is finished, the Posit composition and rounding unit are presented just like in Algorithm 6.

4 Result and Analysis

In this chapter, all the test results are included, which contain Modelsim simulation for exceptional cases, random number test and Quartus synthesis. More importantly, all the test results are analyzed and compared. The accuracy test of an 8-bit Posit adder result is got by using well-designed testbench and the Modelsim simulation. The comparison of area used between Posit and IEEE-754 floating-point is given by Quartus total ALMs used.

4.1 Accuracy test and comparison

In this section, the accuracy and the comparison test between two number formats are provided. Using examples of exceptional case and random normal case of the Posit number format could test whether the calculation is correct. Also, the accuracy comparison between two number formats is by both computing the largest number 8-bit Posit can represent, then analyzing the result.

The result of Posit adder (exceptional cases) is shown in the below figure, including the Posit size, exponent width, inputs, intermediate variables and outputs. First, testing the exceptional cases: zero and infinity. As it is shown below, these three figures show the result of zero condition, also some of the numbers plus zero. Simply from the results, it implies the calculation is correct, which means any number plus zero is equal to itself. However, the regime value seems incorrect in this exceptional case, in principle, the regime value should be -7, because it does not have a termination bit. Moreover, the regime bits are not set to 'signed'. This is fixed at the assumption Posit adder by adding two variables 'Rmax' and 'Rmin' concerning the exceptional cases and setting regime to 'signed'. The following figures show the result for the Posit adder with the exceptional case: adding zero.

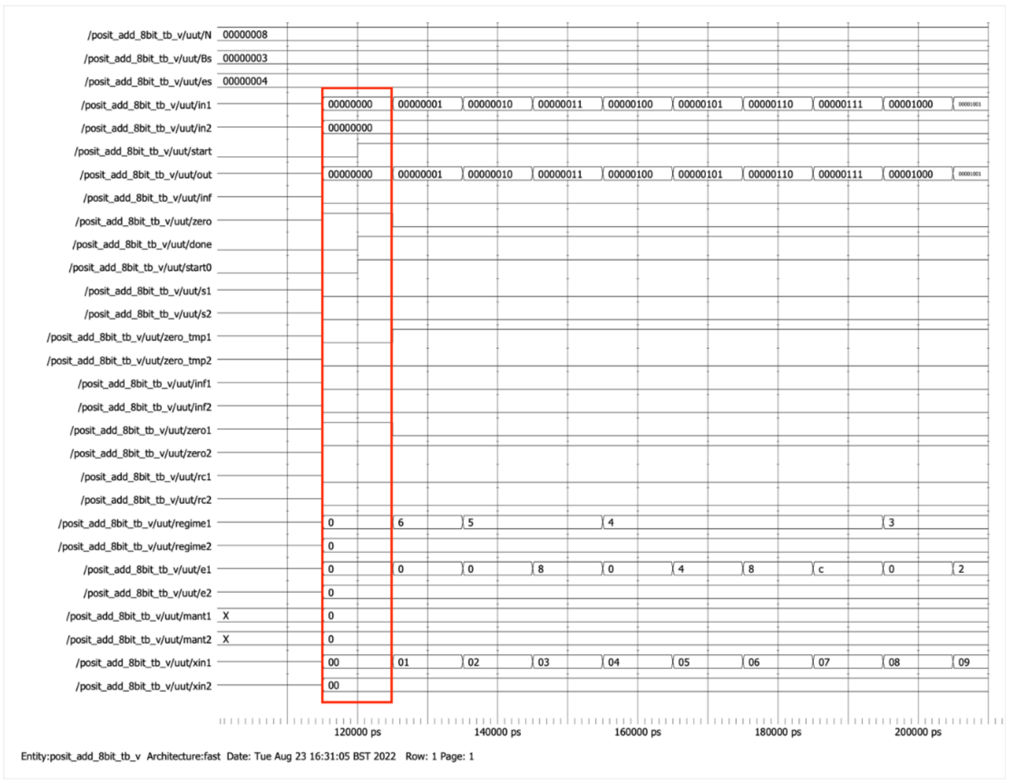

/posit_add_8bit_tb_v/uut/N 00000008
/posit_add_8bit_tb_v/uut/Bs 00000003
/posit_add_8bit_tb_v/uut/es 00000004
/posit_add_8bit_tb_v/uut/in1
/posit_add_8bit_tb_v/uut/in2
/posit_add_8bit_tb_v/uut/start
/posit_add_8bit_tb_v/uut/out
/posit_add_8bit_tb_v/uut/inf
/posit_add_8bit_tb_v/uut/zero
/posit_add_8bit_tb_v/uut/done
/posit_add_8bit_tb_v/uut/start0
/posit_add_8bit_tb_v/uut/s1
/posit_add_8bit_tb_v/uut/s2
/posit_add_8bit_tb_v/uut/zero_tmp1
/posit_add_8bit_tb_v/uut/zero_tmp2
/posit_add_8bit_tb_v/uut/inf1
/posit_add_8bit_tb_v/uut/inf2
/posit_add_8bit_tb_v/uut/zero1
/posit_add_8bit_tb_v/uut/zero2
/posit_add_8bit_tb_v/uut/rc1
/posit_add_8bit_tb_v/uut/rc2
/posit_add_8bit_tb_v/uut/regime1
/posit_add_8bit_tb_v/uut/regime2
/posit_add_8bit_tb_v/uut/e1
/posit_add_8bit_tb_v/uut/e2
/posit_add_8bit_tb_v/uut/mant1
/posit_add_8bit_tb_v/uut/mant2
/posit_add_8bit_tb_v/uut/xin1
/posit_add_8bit_tb_v/uut/xin2
120000 ps
140000 ps
160000 ps
180000 ps
200000 ps
Entity:posit_add_8bit_tb_v Architecture:fast Date: Tue Aug 23 16:31:05 BST 2022 Row: 1 Page: 1

Figure 4.1.1 Posit adder exceptional case: zero add zero

The following figures show the results for the assumption Posit adder with the exception case: adding zero.

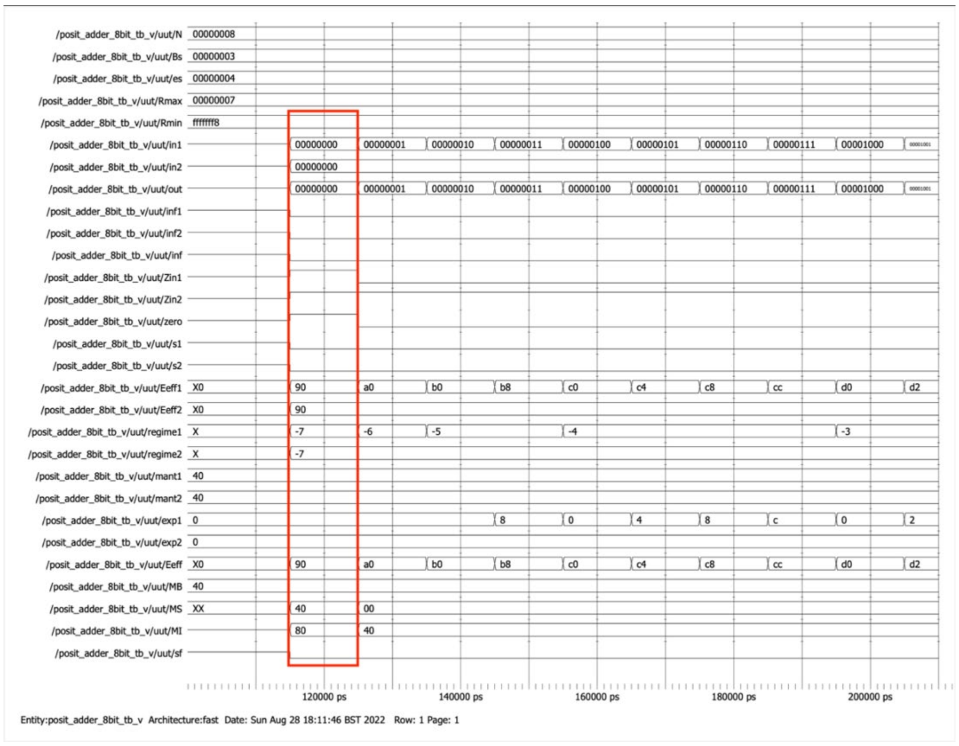

/posit_adder_8bit_tb_v/uut/N 00000008
/posit_adder_8bit_tb_v/uut/Bs 00000003
/posit_adder_8bit_tb_v/uut/es 00000004
/posit_adder_8bit_tb_v/uut/Rmax 00000007
/posit_adder_8bit_tb_v/uut/Rmin fffffff8
/posit_adder_8bit_tb_v/uut/in1
/posit_adder_8bit_tb_v/uut/in2
/posit_adder_8bit_tb_v/uut/out
/posit_adder_8bit_tb_v/uut/inf1
/posit_adder_8bit_tb_v/uut/inf2
/posit_adder_8bit_tb_v/uut/inf
/posit_adder_8bit_tb_v/uut/Zin1
/posit_adder_8bit_tb_v/uut/Zin2
/posit_adder_8bit_tb_v/uut/zero
/posit_adder_8bit_tb_v/uut/s1
/posit_adder_8bit_tb_v/uut/s2
/posit_adder_8bit_tb_v/uut/Eeff1
/posit_adder_8bit_tb_v/uut/Eeff2
/posit_adder_8bit_tb_v/uut/regime1
/posit_adder_8bit_tb_v/uut/regime2
/posit_adder_8bit_tb_v/uut/mant1
/posit_adder_8bit_tb_v/uut/mant2
/posit_adder_8bit_tb_v/uut/exp1
/posit_adder_8bit_tb_v/uut/exp2
/posit_adder_8bit_tb_v/uut/Eeff
/posit_adder_8bit_tb_v/uut/MB
/posit_adder_8bit_tb_v/uut/MS
/posit_adder_8bit_tb_v/uut/MI
/posit_adder_8bit_tb_v/uut/sf
120000 ps
140000 ps
160000 ps
180000 ps
200000 ps
Entity:posit_adder_8bit_tb_v Architecture:fast Date: Sun Aug 28 18:11:46 BST 2022 Row: 1 Page: 1

Figure 4.1.2 Posit assumption adder exceptional case: zero add zero (part1)

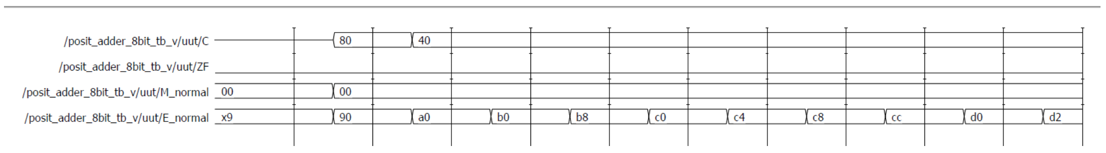
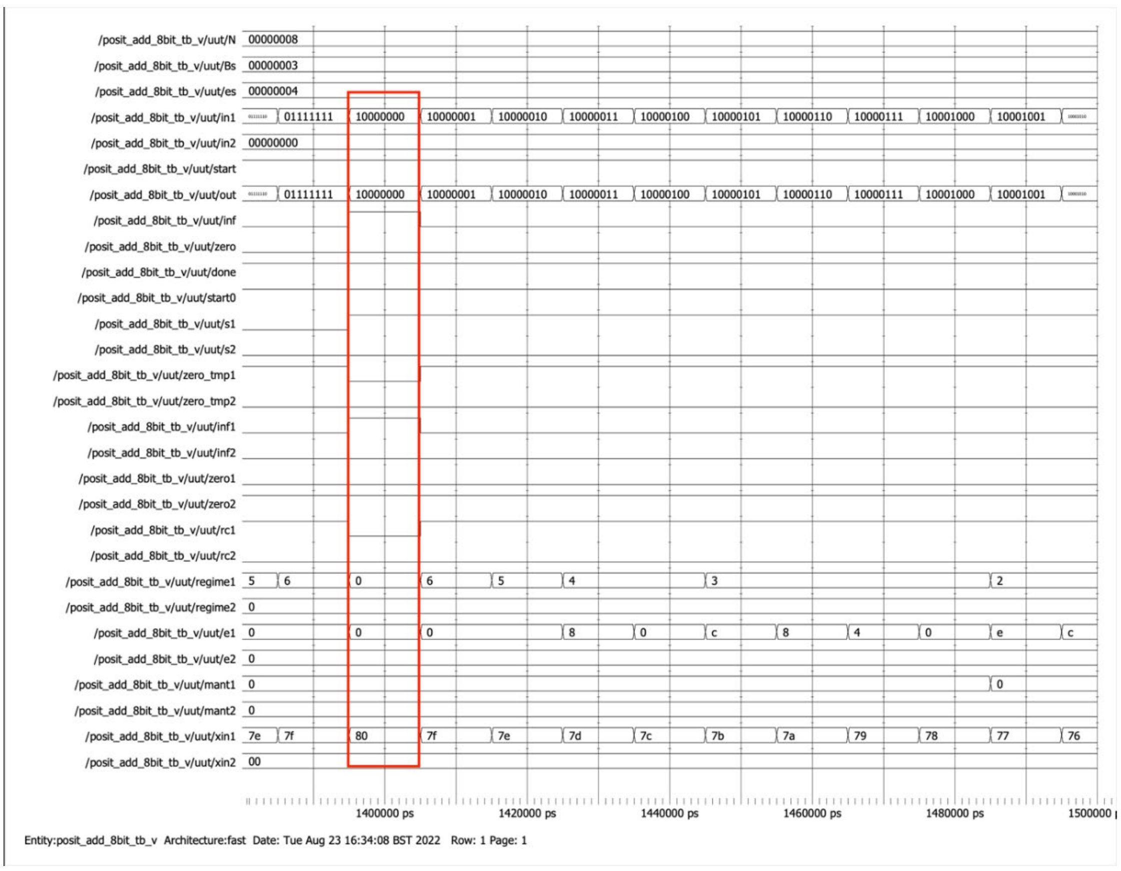


Figure 4.1.3 Posit assumption adder exceptional case: zero add zero (part2)

As figure 4.1.2 and figure 4.1.3 show above, the regime is set to its smallest number $R_{min} = -7$, the extracted exponent bits are zero and mantissa bits are appended with the hidden one. Moreover, the regime part and the exponent part are combined together to perform the effective exponent 'Eeff', bigger and smaller mantissa are chosen and assigned to 'M_B' and 'M_S' respectively. And after the normalization step, the 'E_normal' and 'M_normal' is performed for the final rounding purpose.

The following figures show the results for the Posit adder with the exception case: adding infinity.



/posit_add_8bit_tb_v/uut/N
/posit_add_8bit_tb_v/uut/Bs
/posit_add_8bit_tb_v/uut/es
/posit_add_8bit_tb_v/uut/in1
/posit_add_8bit_tb_v/uut/in2
/posit_add_8bit_tb_v/uut/start
/posit_add_8bit_tb_v/uut/out
/posit_add_8bit_tb_v/uut/inf
/posit_add_8bit_tb_v/uut/zero
/posit_add_8bit_tb_v/uut/done
/posit_add_8bit_tb_v/uut/start0
/posit_add_8bit_tb_v/uut/s1
/posit_add_8bit_tb_v/uut/s2
/posit_add_8bit_tb_v/uut/zero_tmp1
/posit_add_8bit_tb_v/uut/zero_tmp2
/posit_add_8bit_tb_v/uut/inf1
/posit_add_8bit_tb_v/uut/inf2
/posit_add_8bit_tb_v/uut/zero1
/posit_add_8bit_tb_v/uut/zero2
/posit_add_8bit_tb_v/uut/rc1
/posit_add_8bit_tb_v/uut/rc2
/posit_add_8bit_tb_v/uut/regime1
/posit_add_8bit_tb_v/uut/regime2
/posit_add_8bit_tb_v/uut/e1
/posit_add_8bit_tb_v/uut/e2
/posit_add_8bit_tb_v/uut/mant1
/posit_add_8bit_tb_v/uut/mant2
/posit_add_8bit_tb_v/uut/xin1
/posit_add_8bit_tb_v/uut/xin2
1400000 ps
1420000 ps
1440000 ps
1460000 ps
1480000 ps
1500000 ps
Entity:posit_add_8bit_tb_v Architecture:fast Date: Tue Aug 23 16:34:08 BST 2022 Row: 1 Page: 1

Figure 4.1.4 Posit adder exceptional case: adding infinity

As the above figure 4.1.4 shows, when one operand is equal to infinity, the adding result is infinity. Even though the result in the figure is correct, however, it still has the same problem as the zero case above. The regime value is not correct according to the principle. The assumption Posit adder fix this problem either, the simulation waveforms are shown below:

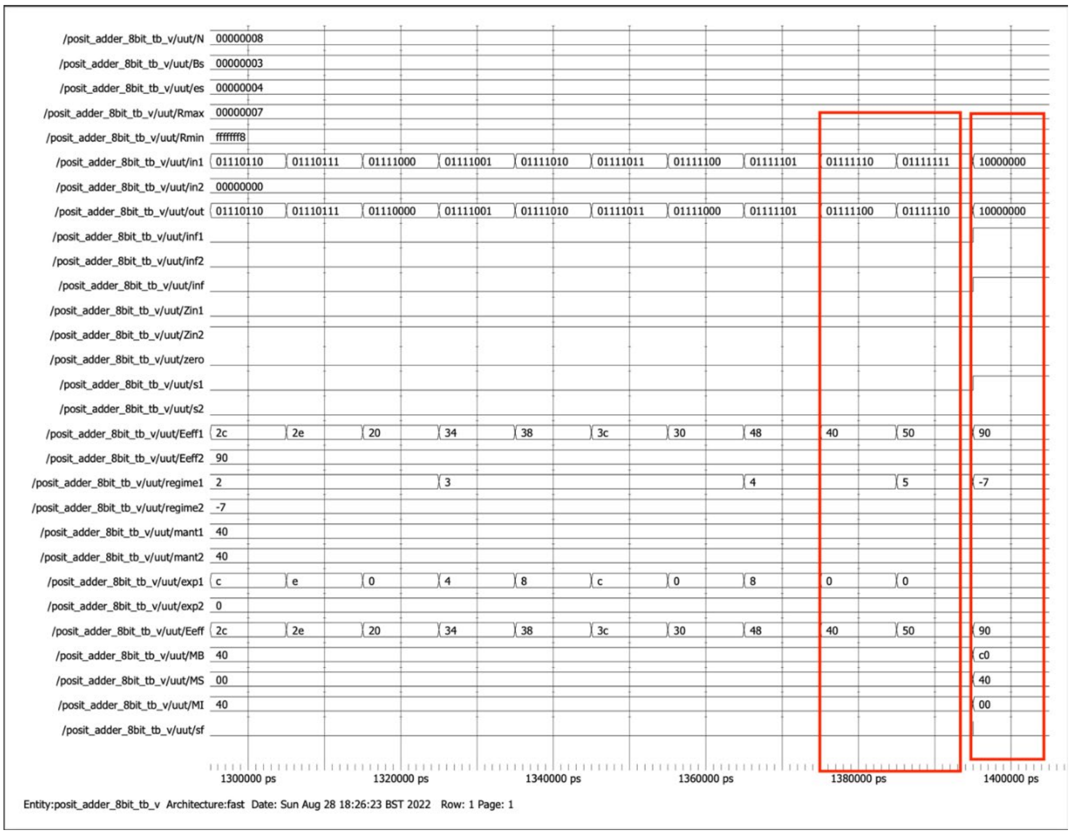

/posit_adder_8bit_tb_v/uut/N 00000008
/posit_adder_8bit_tb_v/uut/Bs 00000003
/posit_adder_8bit_tb_v/uut/es 00000004
/posit_adder_8bit_tb_v/uut/Rmax 00000007
/posit_adder_8bit_tb_v/uut/Rmin ffffff8
/posit_adder_8bit_tb_v/uut/in1 01110110 01110111 01111000 01111001 01111010 01111011 01111100 01111101 01111110 01111111 10000000
/posit_adder_8bit_tb_v/uut/in2 00000000
/posit_adder_8bit_tb_v/uut/out 01110110 01110111 01110000 01111001 01111010 01111011 01111000 01111101 01111100 01111110 10000000
/posit_adder_8bit_tb_v/uut/inf1
/posit_adder_8bit_tb_v/uut/inf2
/posit_adder_8bit_tb_v/uut/inf
/posit_adder_8bit_tb_v/uut/Zin1
/posit_adder_8bit_tb_v/uut/Zin2
/posit_adder_8bit_tb_v/uut/zero
/posit_adder_8bit_tb_v/uut/s1
/posit_adder_8bit_tb_v/uut/s2
/posit_adder_8bit_tb_v/uut/Eeff1 2c 2e 20 34 38 3c 30 48 40 50 90
/posit_adder_8bit_tb_v/uut/Eeff2 90
/posit_adder_8bit_tb_v/uut/regime1 2 3 4 5 -7
/posit_adder_8bit_tb_v/uut/regime2 -7
/posit_adder_8bit_tb_v/uut/mant1 40
/posit_adder_8bit_tb_v/uut/mant2 40
/posit_adder_8bit_tb_v/uut/exp1 c e 0 4 8 c 0 8 0 0
/posit_adder_8bit_tb_v/uut/exp2 0
/posit_adder_8bit_tb_v/uut/Eeff 2c 2e 20 34 38 3c 30 48 40 50 90
/posit_adder_8bit_tb_v/uut/MB 40 c0
/posit_adder_8bit_tb_v/uut/MS 00 40
/posit_adder_8bit_tb_v/uut/MI 40 00
/posit_adder_8bit_tb_v/uut/sf
1300000 ps
1320000 ps
1340000 ps
1360000 ps
1380000 ps
1400000 ps
Entity:posit_adder_8bit_tb_v Architecture:fast Date: Sun Aug 28 18:26:23 BST 2022 Row: 1 Page: 1

Figure 4.1.5 Posit assumption adder exceptional case: adding infinity (part1)

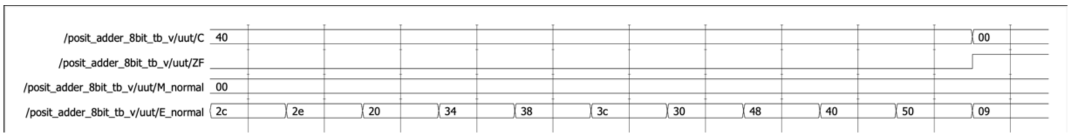

/posit_adder_8bit_tb_v/uut/C 40 00
/posit_adder_8bit_tb_v/uut/ZF
/posit_adder_8bit_tb_v/uut/M_normal 00
/posit_adder_8bit_tb_v/uut/E_normal 2c 2e 20 34 38 3c 30 48 40 50 09

Figure 4.1.6 Posit assumption adder exceptional case: adding infinity (part2)

For the assumption Posit adder, although the result of this exceptional case is correct, some of the other extracted components are not correct. For example, the in the bigger highlight square, it is obviously the wrong answer. The reason why this problem appears is the incorrectness of rounding technique, which might be the limitation of this assumption.

Then, the normal case is tested, the waveform is shown below, 10011000 * 00101110 = 10011000, which is -16777216 + 0.001953125 = -16777216. Using the rounding technique, this example is correct. The simulation waveform is shown below.

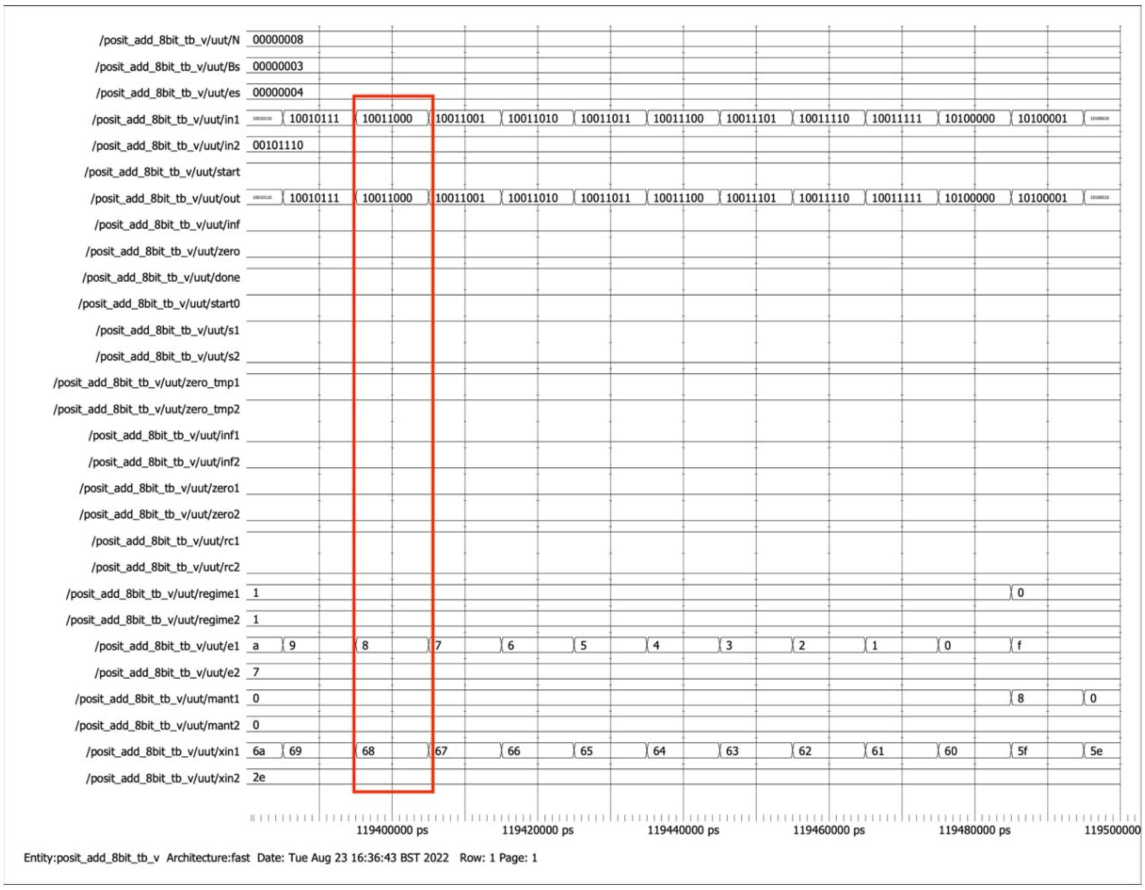


Figure 4.1.7 Posit adder random test

To compare the accuracy between two number formats, use the biggest number the 8-bit Posit with 4-bit exponent bits can represent added to itself, which is 01111111 + 01111111. The result of this addition is equal to 158,456,325,028,528,675,187,087,900,672 in decimal, which can be represent as 0_11100000_00000000000000000000000 in the form of single-precision floating point number. However, this result cannot be presented by the 8-bit Posit, because it is beyond its dynamic range. The simulation result is shown below.

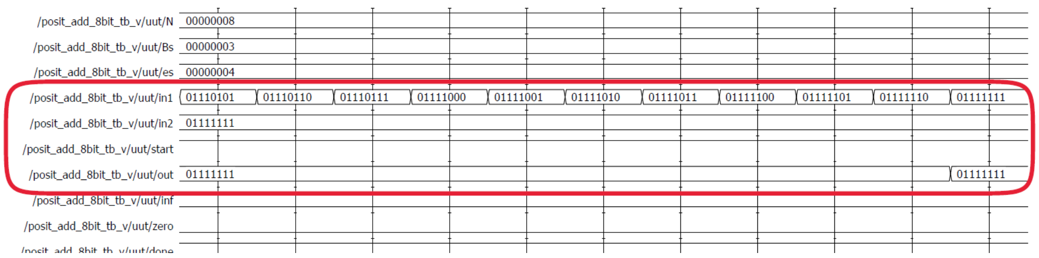
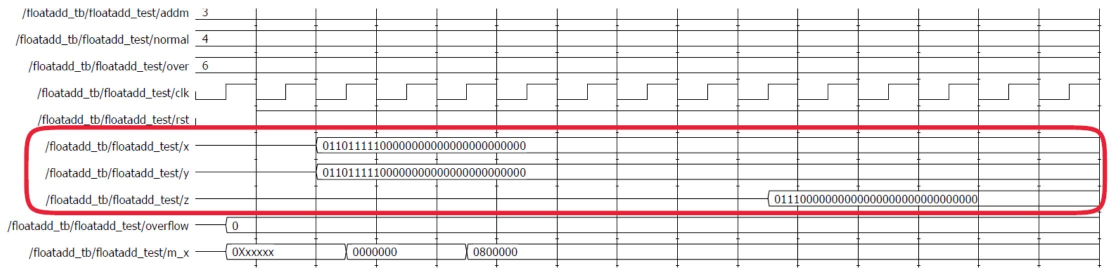


Figure 4.1.8 Posit adder accuracy test

In this simulation waveform above, all the numbers added '01111111' is rounded to '01111111', which means the accuracy is not enough to represent the adding result. In the below figure, in comparison, using the same operands the Posit represent in the above figure 01111111 + 01111111, in the floating-point format, this could be transferred to 01101111100000000000000000000000 + 01101111100000000000000000000000, and the result is correctly shown as 01110000000000000000000000000000



```
/floatadd_tb/floatadd_test/addm      3
/floatadd_tb/floatadd_test/normal    4
/floatadd_tb/floatadd_test/over      6
/floatadd_tb/floatadd_test/clk
/floatadd_tb/floatadd_test/rst
/floatadd_tb/floatadd_test/x         0110111110000000000000000000000
/floatadd_tb/floatadd_test/y         0110111110000000000000000000000
/floatadd_tb/floatadd_test/z         01110000000000000000000000000000
/floatadd_tb/floatadd_test/overflow  0
/floatadd_tb/floatadd_test/m_x       0Xxxxxx  0000000  0800000
```

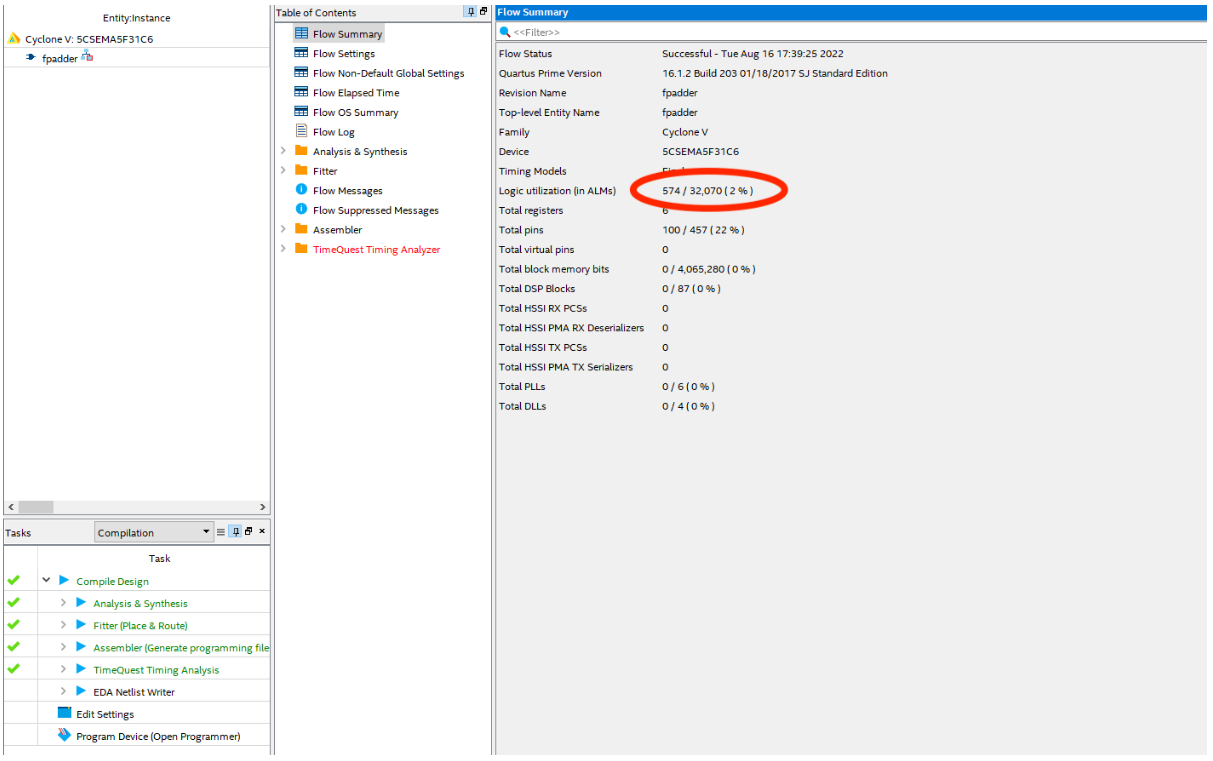
Figure 4.1.9 Floating-point adder accuracy test

This comparison shows the accuracy of 8-bit Posit is lesser than IEEE-754 floating-point format in terms of huge numbers.

4.2 Cost comparison

The cost testing using the synthesis tool Quartus, which will show the logic utilization (in ALMs) of each design.

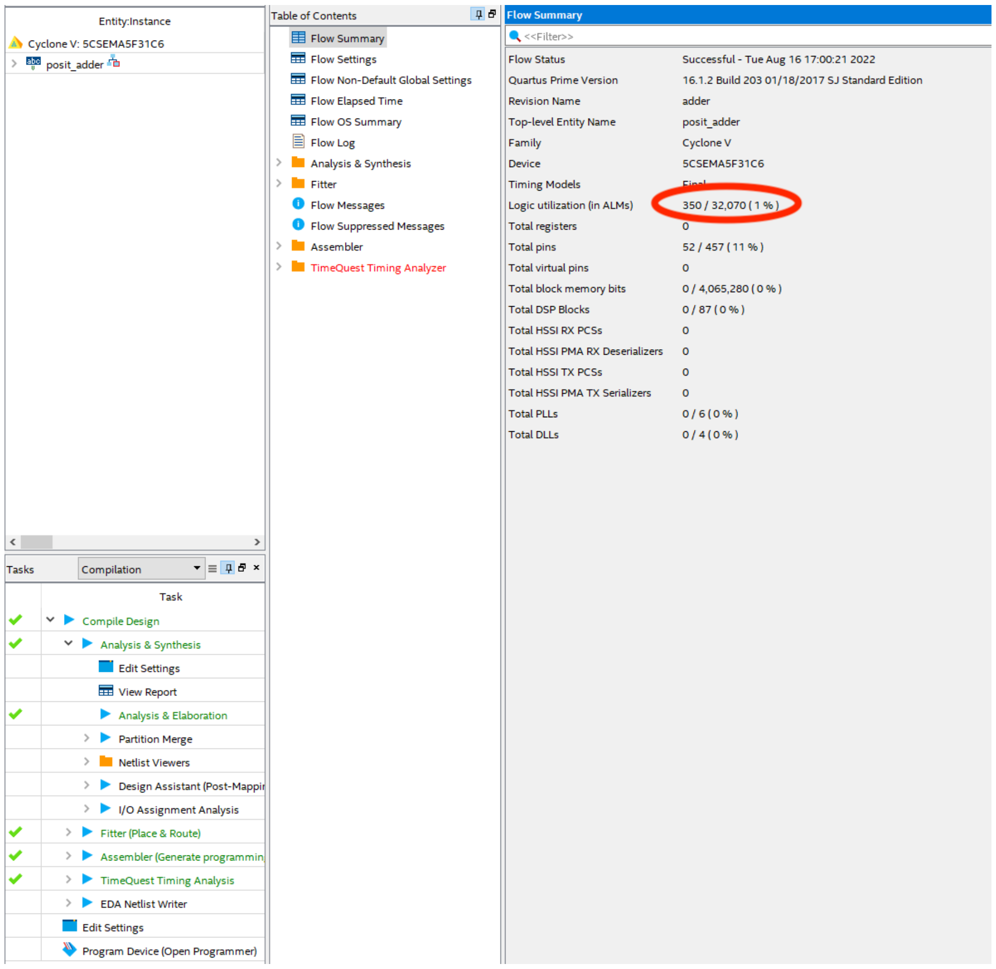
The cost comparison between two number formats and two addition algorithms are shown below. For the single-precision floating-point number, the logic utilization (in ALMs) for an adder is 574, which is shown in the below figure.



Flow Status	Successful - Tue Aug 16 17:39:25 2022
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Standard Edition
Revision Name	fpadder
Top-level Entity Name	fpadder
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	574 / 32,070 (2 %)
Total registers	6
Total pins	100 / 457 (22 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 4.2.1 Floating-point adder cost test

For the Posit, the logic utilization (in ALMs) for an adder is 350, which is shown in the below figure.



Entity:Instance

Cyclone V: 5CSEMA5F31C6

posit_adder

Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Flow Messages
- Flow Suppressed Messages
- Assembler
- TimeQuest Timing Analyzer

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Aug 16 17:00:21 2022
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Standard Edition
Revision Name	adder
Top-level Entity Name	posit_adder
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	350 / 32,070 (1 %)
Total registers	0
Total pins	52 / 457 (11 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

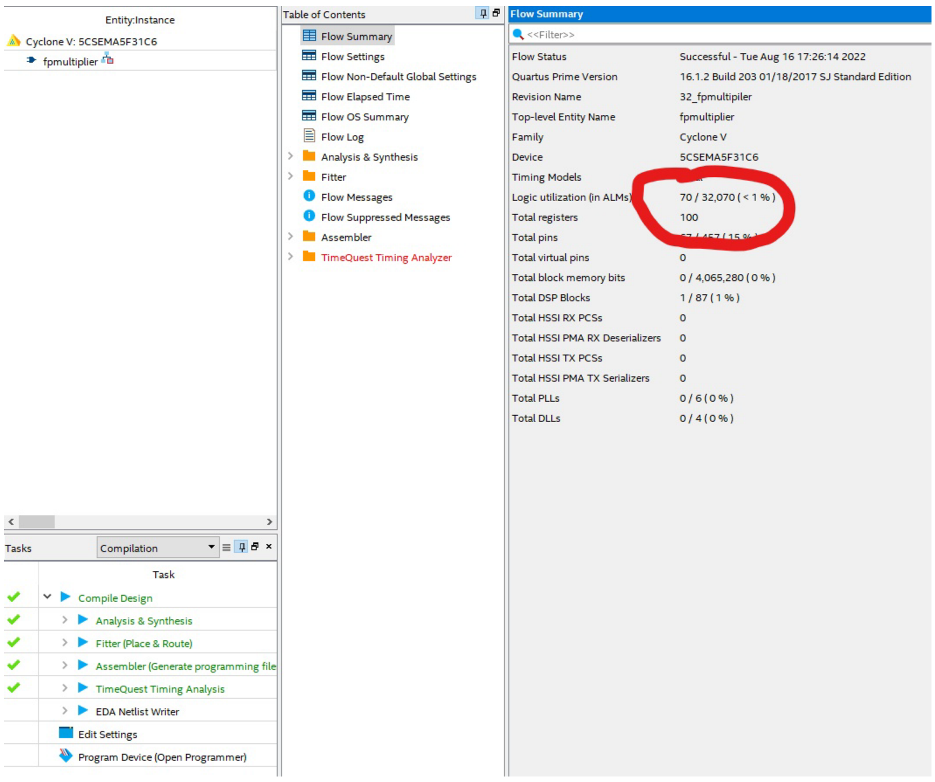
Tasks Compilation

	Task
✓	Compile Design
✓	Analysis & Synthesis
	Edit Settings
	View Report
✓	Analysis & Elaboration
	Partition Merge
	Netlist Viewers
	Design Assistant (Post-Mappi
	I/O Assignment Analysis
✓	Fitter (Place & Route)
✓	Assembler (Generate programmin
✓	TimeQuest Timing Analysis
	EDA Netlist Writer
	Edit Settings
	Program Device (Open Programmer)

Figure 4.2.2 Posit adder cost test

The result of two tests above clearly shows that Posit adder saves approximately 40% of the logic unit in terms of hardware, which suggests Posit can use lesser resources than the floats to finish the add operation.

The cost comparison between two number formats in terms of multiplication is shown below. For the single-precision floating-point number, the logic utilization (in ALMs) for a multiplier is 70, which is shown in the below figure.



Entity:Instance
Cyclone V: 5CSEMA5F31C6
fpmultiplier

Table of Contents
Flow Summary
Flow Settings
Flow Non-Default Global Settings
Flow Elapsed Time
Flow OS Summary
Flow Log
Analysis & Synthesis
Fitter
Flow Messages
Flow Suppressed Messages
Assembler
TimeQuest Timing Analyzer

Flow Summary

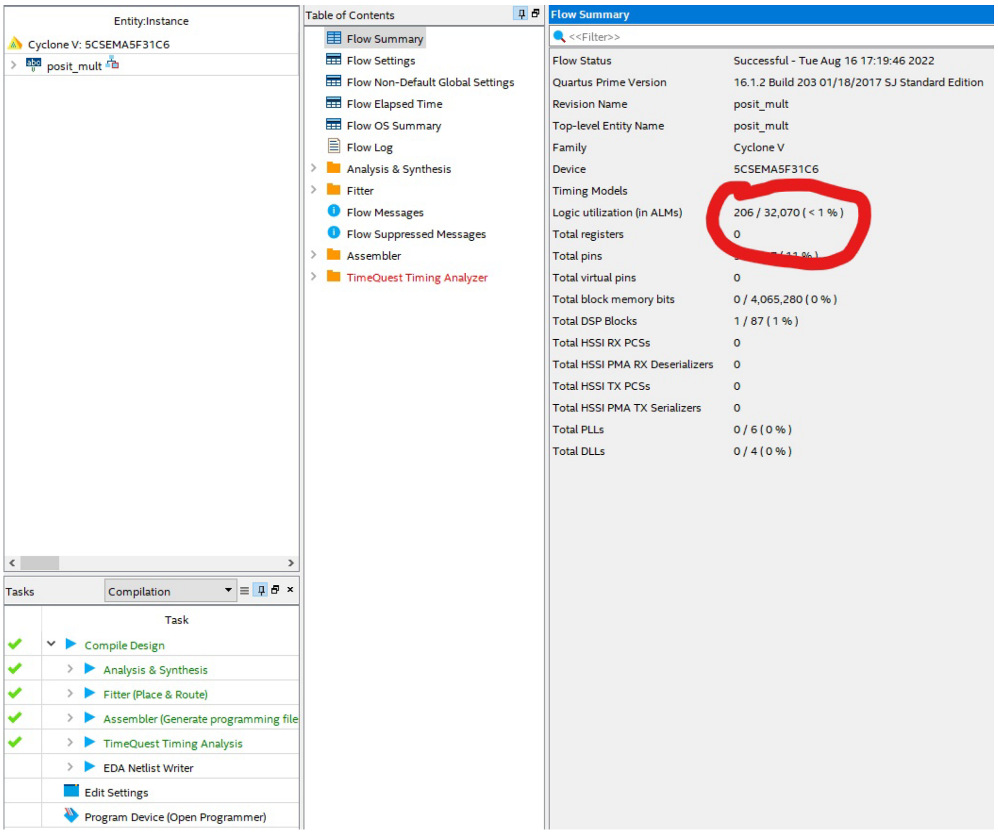
Flow Status	Successful - Tue Aug 16 17:26:14 2022
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Standard Edition
Revision Name	32_fpmultipiler
Top-level Entity Name	fpmultiplier
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	[illegible]
Logic utilization (in ALMs)	70 / 32,070 (< 1 %)
Total registers	100
Total pins	67 / 457 (15 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Tasks: Compilation

	Task
✓	Compile Design
✓	Analysis & Synthesis
✓	Fitter (Place & Route)
✓	Assembler (Generate programming file
✓	TimeQuest Timing Analysis
	EDA Netlist Writer
	Edit Settings
	Program Device (Open Programmer)

Figure 4.2.3 Floating-point multiplier cost test

For the Posit the logic utilization (in ALMs) for a multiplier is 206, which is shown in the below figure.



Entity:Instance
Cyclone V: 5CSEMA5F31C6
posit_mult

Table of Contents
Flow Summary
Flow Settings
Flow Non-Default Global Settings
Flow Elapsed Time
Flow OS Summary
Flow Log
Analysis & Synthesis
Fitter
Flow Messages
Flow Suppressed Messages
Assembler
TimeQuest Timing Analyzer

Flow Summary

Flow Status	Successful - Tue Aug 16 17:19:46 2022
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Standard Edition
Revision Name	posit_mult
Top-level Entity Name	posit_mult
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	[illegible]
Logic utilization (in ALMs)	206 / 32,070 (< 1 %)
Total registers	0
Total pins	[illegible] (11 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Tasks: Compilation

	Task
✓	Compile Design
✓	Analysis & Synthesis
✓	Fitter (Place & Route)
✓	Assembler (Generate programming file
✓	TimeQuest Timing Analysis
	EDA Netlist Writer
	Edit Settings
	Program Device (Open Programmer)

Figure 4.2.4 Posit multiplier cost test

The result of these two tests shows that the Posit multiplier uses more ALMs than the IEEE-754 floating-point number multiplier, which disobeyed the assumption at first about Posit could save more resources compared with the floating-point. At this circumstance, these two designs should be further analyzed by the performance test below. Combining these two test results together could make the test conclusion more precise.

4.3 Performance test

The performance test uses the simulation tool to analyze how many clock cycles the design needs in order to finish the calculation. With lesser clock cycle means higher speed or performance.

In this small section, using a simple example of multiplication could test the performance, through analyzing how many clock cycle the design is used. In terms of performance, using the multiplication of both formats as an example, -512 * 0.046875 = -24. The simulation form shows the multiplication result using floating-point multiplier.

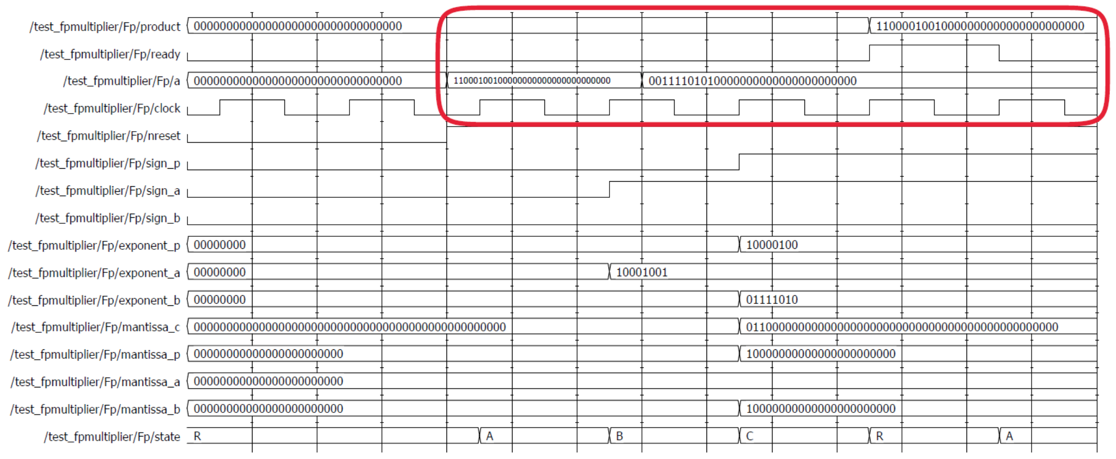


Figure 4.3.1 Floating-point multiplier performance test

As shows in the above figure, the calculation takes three clock cycles to provide the multiplication result. Combining with the test result in the above section, which uses lesser ALMs, suggests using performance trade for cost. With lesser performance could decrease the cost.

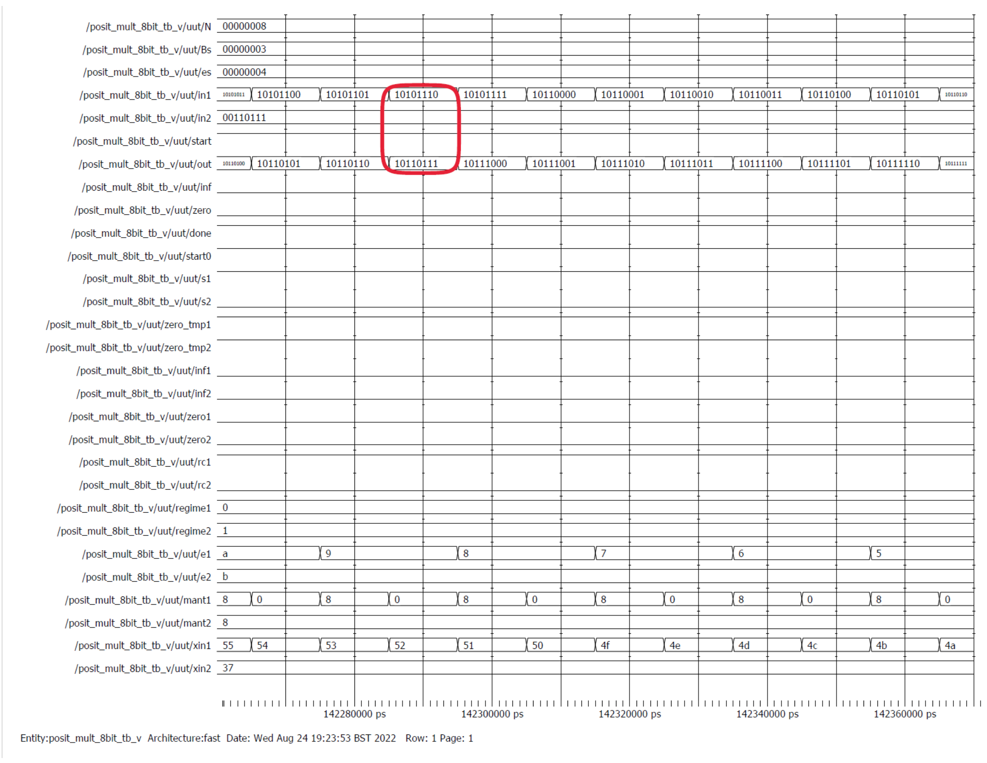


Figure 4.3.2 Posit multiplier performance test

The figure above shows that the calculation in Posit finished in one clock cycle, because it is composed of purely combinational logic, which makes the calculation quicker than the floating-point number format mentioned before.

5 Conclusion

Unum has been through a series of developments. The latest version of Unum is named Posit, which can be the counterpart of IEEE-754 floating-point number. This new number format provides many potential advantages over floating-point number. The project aims to design a Posit adder and Posit multiplier and compare the accuracy, power consumption, performance and area used with the floating-point number adder and multiplier.

This dissertation uses some ideas of recent published algorithms to produce a parameterized Posit adder and multiplier. Also, providing an assume adder try to make some improvements in terms of the hardware use. The result and analysis part of this dissertation shows three findings. First, the 8-bit Posit has lesser accuracy and dynamic range compared with the IEEE-754 single-precision floating-point number. Using the example of adding the largest number of the 8-bit Posit can represent added to itself, the rounding technique of the Posit will make sure the result is not 'NaN' like floating-point numbers when there is an overflow, but the result is not correct. However, these two huge numbers could be calculated correctly using the single-precision floating-point adder, which means the single-precision floating-point number has greater accuracy and dynamic range compared with 8-bit Posit (es=4). Second, the Posit format consumes lesser area and energy than floats, using the information that Quartus provide to decide the cost. Logic utilization is a reasonable parameter which suggests the cost of a design. Comparing the number Quartus provides could determine which number format has the lower cost. In this case, the 8-bit Posit has lesser cost than the single-precision floating-point number. Third, Posit provides higher performance than the floats. In terms of performance comparison, the first thing comes to mind is the speed of the calculation. Using the example of multiplication simulation waveform provided by Modelsim, which holds the Posit could computing the result immediately. However, the floating-point multiplier used in this design takes three clock cycles to provide the result of the result, meaning that the Posit multiplier's performance is better than the floating-point number multiplier.

The limitation of this paper is that in the test and compare stage, using 8-bit Posit to compare with 32-bit floating point is not strong enough to suggest Posit is greater than the floats. For example, although the accuracy and dynamic range of the Posit are lesser than the 32-bit floating-point, when increasing the size of the Posit, this problem can be fixed, and that brings new questions in terms of performance and cost.

To put it in a nutshell, this would be a fruitful area for further work. For example, when Posit provides the result that 32-bit floating-point number cannot represent? Is Posit using lesser energy and providing higher performance? Also, with the same bit width which number format calculator has the better performance or using lesser energy?

6 Reference

[1] J. L. Gustafson and I. Yonemoto, “Beating floating point at its own game: Posit arithmetic,” *Supercomput. Front. Innov.*: Int. J., vol. 4, no. 2, pp. 71–86, Jun. 2017.

[2] J. L. Gustafson, *The End of Error: Unum Computing,* 1st ed. Boca Raton, FL, USA: CRC Press, 2015.

[3] W. Tichy, ‘‘The end of (numeric) error: An interview with John L. Gustafson,’’ *Ubiquity*, vol. 2016, p. 1, Apr. 2016.

[4] Rich Brueckner. (2015). *Slidecast: John Gustafson Explains Energy Efficient Unum Computing. Inside* HPC.

[5] W. Tichy, ‘‘Unums 2.0: An interview with John L. Gustafson,’’ *Ubiquity*, vol. 2016, p. 1, Oct. 2016.

[6] J. L. Gustafson, ‘‘A radical approach to computation with real numbers,’’ *Supercomput. Frontiers Innov.*, vol. 3, no. 2, pp. 38–53, Jul. 2016.

[7] W. Kahan, “IEEE standard 754 for binary floating-point arithmetic,” *Lect. Notes Status IEEE*, vol. 754, no. 1776, p. 11, May 1996.

[8] M. K. Jaiswal and H. K. So, “Universal number posit arithmetic generator on FPGA,” in *2018 Design, Automation & Test in Europe Conference & Exhibition*, DATE *2018, Dresden, Germany, March 19-23, 2018*, 2018, pp. 1159–1162.

[9] J. Hou, Y. Zhu, S. Du, and S. Shong, ‘‘Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on FPGA,’’ *J. Signal Process. Syst.*, pp. 1–12, Nov. 2018.

[10] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, and R. Leupers, ‘‘Parameterized posit arithmetic hardware generator,’’ in *Proc. IEEE 36th Int. Conf. Comput. Design (ICCD)*, Oct. 2018, pp. 334–341.

[11] M. K. Jaiswal and H. K. . -H. So, "Architecture Generator for Type-3 Unum Posit Adder/Subtractor," *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1-5, doi: 10.1109/ISCAS.2018.8351142.

[12] M. K. Jaiswal and H. K. . -H. So, "PACoGen: A Hardware Posit Arithmetic Core Generator," in *IEEE Access*, vol. 7, pp. 74586-74601, 2019, doi: 10.1109/ACCESS.2019.2920936.

7 Appendix

7.1 Gantt chart

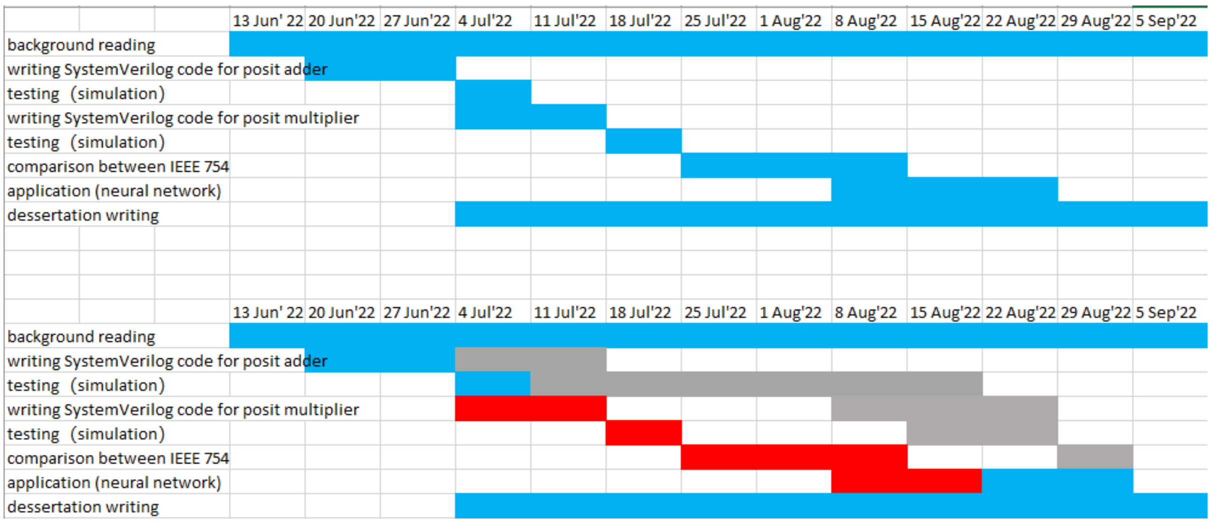


Figure 7.1.1 Gantt chart

There are some uncertainties of the project, therefore the first draft of the plan needs to be changed during the process. Blue stands for the original plan, red representing change the plan to the grey space.

7.2 Code

The code of the project is in an archive file, including:

- Posit adder and testbench
- Posit adder (assumption) and testbench
- Posit multiplier and testbench
- 32-bit floating-point adder and testbench
- 32-bit floating-point multiplier and testbench