Tampere University

Juho Kylväjä

INVESTIGATION OF POSITS AND IEEE-754 FLOATING POINTS

In hardware implementations of addition and multiplication operations

Master's Thesis
Tampere University
Examiners: Prof. Karri Palovuori
Prof. Jukka Vanhala
April 2023

ABSTRACT

Juho Kylväjä: Investigation of posits and IEEE-754 floating points
Master's Thesis
Tampere University
Master's degree Programme in Electrical Engrineering
April 2023

This thesis aims to investigate a relatively new alternative presentation for floating-point arithmetic the type-3 UNUM, posit for a replacement of the widely used IEEE 754 floating-point standard. The thesis's main focus is on arithmetic operations of addition and multiplication. First, literature check of posit and IEEE 754 floating-point standards formats, special cases, overflow and underflow operations, and rounding methods are conducted. Then the arithmetic implementation steps of posit and IEEE 754 addition and multiplication operations on hardware are shown. In addition, the tools used to analyze the chosen designs and the designed testbench flow for behavioral verification of the designs is described. Finally, the results were examined, followed by the conclusion.

The thesis concludes that posits could replace the currently widely used IEEE 754 standard due to having better accuracy around one and better dynamic range with 8, 16 and 32-bit numbers. However, the synthesis results show that FPU achieves better area, delay and power scores than the posit designs chosen in this thesis. Furthermore, implementing compatible processors for posits would require lots of work and time. Overall, posits have great potential to replace the IEEE 754 standard. It is interesting to see how future studies on posits will affect the future of floating-point arithmetic in hardware.

Keywords: UNUM, posit, arithmetic, IEEE 754, floating-point

TIIVISTELMÄ

Juho Kylväjä: Investigation of posits and IEEE-754 floating points
Diplomityö
Tampereen yliopisto
Sähkötekniikan diplomi-insinööri tutkinto-ohjelma
Huhtikuu 2023

Tämän tutkielman tarkoituksena on tutkia suhteellisen uutta vaihtoehtoista esitystapaa liukulukuaritmeettiselle laskennalle, tyypin 3 UNUM:ia, joka voisi korvata laajalti käytetyn IEEE 754 -liukulukustandardin. Työn pääpaino on yhteenlaskun ja kertolaskun aritmeettisissa operaatioissa. Ensin tarkastellaan kirjallisuutta posit- ja IEEE 754 -liukulukustandardien formaateista, erikoistapauksista, yli- ja alivuoto-operaatioista sekä pyöristysmenetelmistä. Sen jälkeen esitetään posit- ja IEEE 754 -liukulukujen yhteen- ja kertolaskuoperaatioiden aritmeettiset toteutusvaiheet järjestelmäpiireissä. Lisäksi kuvataan työkalut, joita käytetään valittujen mallien analysointiin, sekä testipenkki vuo, jota on käytetty verifioimaan mallien funktionaalisuus. Lopuksi tarkastellaan tuloksia, minkä jälkeen esitetään johtopäätökset.

Tutkielmassa todetaan, että posit voisi korvata nykyisin laajalti käytetyn IEEE 754 -standardin, koska sen tarkkuus on parempi lähellä yhtä sekä sen dynaaminen alue on parempi 8-, 16- ja 32-bittisillä luvuilla. Synteesitulokset osoittavat kuitenkin, että IEEE 754 liukulukuyksikkö saavuttaa paremmat pinta-ala-, viive- ja tehonkulutustulokset kuin tässä tutkielmassa valitut posit-mallit. Lisäksi yhteensopivien prosessoreiden toteuttaminen posit-malleille vaatisi paljon työtä ja aikaa. Kaiken kaikkiaan posit-standardilla on suuri potentiaali korvata IEEE 754 -standardi. On mielenkiintoista nähdä, miten posit-malleja ja standardia koskevat tulevat tutkimukset vaikuttavat järjestelmäpiirien liukulukuaritmetiikan tulevaisuuteen.

Avainsanat: UNUM, posit, aritmetiika, IEEE 754, liukuluku

PREFACE

This thesis was done for Nordic Semiconductor Finland Oy. I want to thank everyone involved from Nordic: M.Sc Svein Henninen, M.Sc Alexander Skavnes and M.Sc Tommi Räikkönen. In addition, I want to thank examiners from the University of Tampere Prof. Karri Palovuori and Prof. Jukka Vanhala.

Finally, I want to thank my family and friends for their support.

Tampere, 24th April 2023

Juho Kylväjä

CONTENTS

ABBREVIATIONS

ASIC	Application specific integrated circuit
C-NN	Convolution neural network
DUT	Design under test
EDA	Electronic design automation
FPGA	Field-programmable gate array
FPU	Floating-point unit
IEEE	Institute of Electrical and Electronics Engineers
LEC	Logic equivalence check
LOD	Leading one detector
LSB	Least significant bit
LUT	Lookup table
LZD	Leading zero detector
MSB	Most significant bit
MUX	Multiplexer
RTL	Register transfer level
SAIF	Spatial Archive and Interchange Format
SoC	System on a chip
UNUM	Universal number

1. INTRODUCTION

Computer systems produce real numbers encoded as binary and they have their finite limit in digital system hardware. Different computer systems need to produce similar results from floating-point operations, thus a standard is needed. The most used standard in computer architecture is the IEEE-754 floating-point format. The first IEEE-754 standard was made in 1985 and newer revisions have been made since. Currently the widely used standard is IEEE-754-2008. However, could there be a better alternative standard for floating-point arithmetic in computer architectures?

This thesis will investigate this question by comparing the before mentioned IEEE-754-2008 floating-point standard to type-3 UNUMs also known as posits. Posits were introduced in 2017 by John Gustafson and it has supposed benefits over the IEEE-754 standard by having higher precision and better dynamic range with same bitwidths. This thesis investigates and analyzes these differences and architecture designs using analysis tools used in a system-on-chip design. The thesis aims to get results from two different posit designs and compare achieved results from the tools to corresponding IEEE-754 floating-point units results. The chosen arithmetic operation units were addition and multiplication.

First posits and IEEE-754 standard formats, special cases and rounding methods are introduced in chapter 2. Then, their corresponding hardware architecture designs are shown for addition and multiplication operations in chapters 3 and 4. In chapter 5 posit accuracy compared to IEEE-754 floating-points is examined from related works by John Gustafson. In chapter 6 the tools used for analysis, and the designed testbench flow for the behavioral verification for design units are introduced. In chapter 7 the results from the different analysis tools and behavioral verification are shown and compared. Finally, conclusion is formed in chapter 8.

2. POSIT AND IEEE 754-2008 FLOATING POINTS

Floating-point arithmetic is a fundamental aspect of computing. In this chapter two different ways of presenting floating-point arithmetic posits and IEEE-754-2008 standards are described. In addition to presenting their formats and number systems, their associated special cases are covered when invalid or special operations are performed. Furthermore, the overflow and underflow section will showcase when numbers get extremely small or large and how the standards perform in these conditions. Finally, possible rounding methods are introduced and their effect on the final rounded result.

2.1 Type-3 posits

Posits consist of four fields: sign bit, regime, exponent and fraction. The sign bit is the MSB, and determines whether the posit value is positive (0) or negative (1). After sign bit comes the regime bit/bits. The first regime bit determines the regime sign and also the termination bit. The termination bit is the first bit with the opposite value of the regime sign bit. The regime field consists of consecutive regime sign bits terminated by the termination bit. After the termination bit there are exponent bits. Exponent bit field length is up-to beforehand determined *es* bits. If the exponent bit size *es* is two, the next two bits after termination bit determine the exponent value. The rest of the bits are reserved for fraction bits. [1] [2]

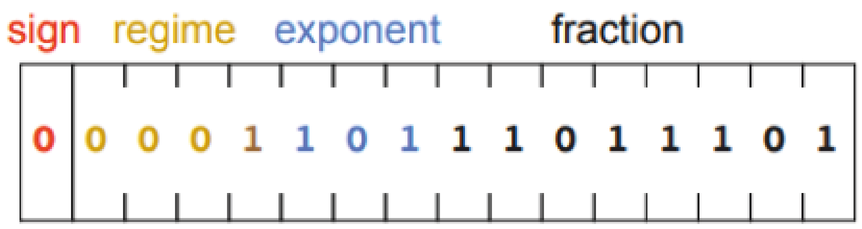

Figure 2.1. *Four bit fields of 16-bits type-3 posit with es=3.*

An example of a 16-bit posit field is shown in figure 2.1. From the figure 2.1 it can be determined that the sign bit is "0", the regime bits are "000", the exponent bits are "101" and finally, the fraction bits are "1101 1101". Notice that the termination bit "1" with the brown color is not part of any of the fields. Regime value is determined by the run-length of the regime bits and by the regime sign bit. If the regime sign bit is 0, the regime value *k* will be determined by *k = -m* where *m* is the run-length of regime bits. If the regime sign bit is 1, the regime value *k* will be determined by *k = m-1*. Regime value *k* is also used to indicate the scale factor of so-called $useed^k$ value, where *useed* = $2^{2^{es}}$. The *useed* is used to determine the maximum and minimum values of N-bit posit. In addition *useed* is used to calculate the real value of posit.

Exponent value *e* is an unsigned integer and is determined by the value of the *es* bits and has the scaling effect of 2^e to the real value of posit. [1, 2] How different bit fields affect the real value of posit are later shown in cp. 4.1 in formula 4.1. Because of the nature of regime bits varying in length, there can be posit values that do not have exponent or fraction bits. An illustrative examples of such situations are shown below with 8-bit posit when exponent size is $es = 1$. Different fields are separated with dash.

$$0 - 1111111$$

In the first example the posit value consist of the sign bit 0 and seven bit regime field "1111111". Because the posit is 8-bit in length there are no exponent or fraction bits in the posit value left after regime bits. Below is an example of 8-bit posit with exponent bits when $es = 1$.

$$0 - 1111 - 0 - 1$$

Now the posit consist of the sign bit 0, four regime bits "1111", termination bit 0 and exponent bit 1. In this example the regime bit field is terminated by the termination bit 0 and there are still bits left. In this example the exponent bit size is up-to one bit ($es = 1$) so the next bit after termination bit is the exponent field. Similar way the fraction bits would appear if the regime field would be only three bits long and exponent size would again be $es = 1$. More examples of how regime can affect other fields sizes are shown in cp. 4.1. Posit number system can be visualized by a circle.

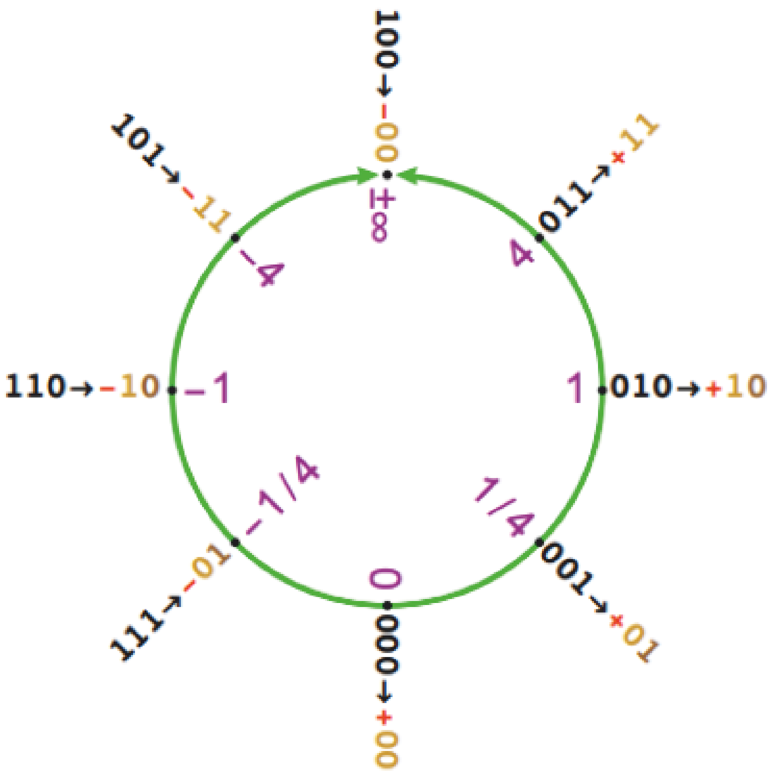

Figure 2.2. *Type-3 posit 3-bit number system with es = 1.*

Type-3 posit's 3-bit number system is shown in figure 2.2. On the right side of the circle are positive reals and on the left side are negative reals. Posit values always increase to *infinity* on the right side and decrease to *-infinity* on the left side. Zero is always on the bottom part of the circle. *Infinity* and *-infinity* in type-3 posits are both represented by 1 followed by only zeros. The value zero is represented by 0 followed by only zeros. From figure 2.2 can also be noticed that -1 is always located in the leftmost place on the circle and 1 on the rightmost place. One feature of the posit number system is that the opposite values are on opposite sides of the circle. In addition to posits having the feature of opposite numbers on opposite sides, posits have the feature of being reciprocal. In the posit number system, every real value has its reciprocal value on the circle.

In figure 2.2 $useed$ value is 4 since $2^{2^1} = 4$ if this example posit system of 3-bits would be turned to 4-bit system, only zero would be added at the LSB of every posit on the circle. In figure 2.2 the maximum real value *maxpos* is 4 and the minimum real *minpos* is 1/4. Maxpos value can be determined by $maxpos = useed^{n-2}$, where *n* is the number of posit bits. Minpos can be determined by $minpos = useed^{2-n}$. When increasing the number of bits the new maxpos and minpos can be determined using *useed* $newMaxpos = maxpos \times useed$ and $newMinpos = minpos/useed$. When converting 4-bit posit to a 5-bit posit, new values are inserted halfway on existing posit values. Posit bit amount *n* can range from 2 to *n* and the number of exponents bits can be $n-2$. Currently, the standard for posits is to use 0 exponent bits for 8-bit posits, 1 exponent bit for 16-bit posits and 2 exponent bits for 32-bit posit. [1, 2]

2.2 IEEE 754-2008

The IEEE 754-2008 floating-point standard consists of three fields: sign bit, exponent and significand. Similarly, as in posits, the sign bit is the MSB and determines the sign of the value. If the sign bit is 1, the value is negative; if the sign bit is 0, the value is positive. The main difference between posit and IEEE floating-point bit fields is that the exponent and significand field lengths are determined beforehand in IEEE 754 floating-points unlike in posit where the exponent and fraction bit field lengths can vary due to regime bits. [3, 4] In the figure below, the floating-point bit fields are illustrated.

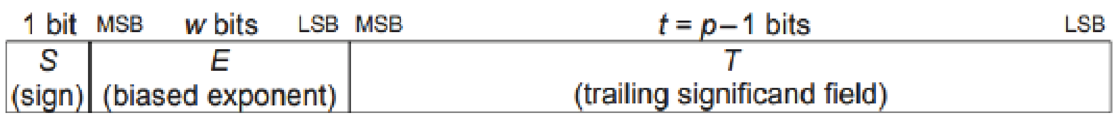

Figure 2.3. *IEEE 754-2008 floating-point bit fields. [4]*

In the figure 2.3 the floating-point bit fields are shown, the MSB *S* is the sign bit, the *E* is the biased exponent and *T* is the significand field. The biased exponent in the floating point standard is defined to be $E = e + bias$, where the *e* is the exponent value and bias is $2^{w-1} - 1$ where *w* is the exponent field width. The bias value is also the maximum value of the exponent field and the minimum exponent value can be computed from the maximum value by $emin = 1 - emax$. The width of the significand field is $t = p - 1$ where *p* is the chosen precision. The value of the significand field *T* is computed from the significand bits [3, 4]. The real value of floating-point when $1 \leq E \leq 2^W - 2$ can be computed from

$$(-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T) \tag{2.1}$$

Suppose the $E = 0$ and $T \neq 0$ the real value of floating-point is computed similarly as above, except the exponent $E - bias$ is replaced by the minimum exponent value $emin$. Floating-point values have corresponding opposite values but unlike posits floating-points do not have a reciprocal value associated with every value.

Floating-point format specifies more special values than posits do. In posits, there are only *-Infinity* and *Infinity* specified by the same binary format where the sign bit is 1 and others are all zeroes. In floating-point standard special values are:

- Two infinities *+Infinity* and *-Infinity*
- Two kinds of NaN (not-a-number) quiet NaN *qNaN* and signaling NaN *sNaN*

- Two kinds of zero values +0 and -0

the IEEE 754 standard specifies that 32-bit numbers should have 8 exponent bits and 16-bit numbers should have 5 exponent bits. Standard does not specify bit sizes for under 16 bits so with the bit width of 8 used in this thesis 3 exponent bits have been used. [3, 4]

2.3 Overflow and underflow

In posits, underflow or overflow do not occur but in IEEE 754-2008 there are a few rules associated with overflow and underflow. [2, 1] In the IEEE 754-2008 standard overflow is specified so that if the result value after rounding exceeds the largest possible finite number, and the exponent field overflows, will the result value be set according to what rounding format is used:

- roundTiesToEven and roundTiesToAway in default sets result to infinite based on the intermediate results sign.
- roundTowardsZero in default sets result to formats largest finite number based on the sign of intermediate result.
- roundTowardNegative in defaults sets positive overflows to formats largest finite number and negative overflows to -infinity.
- roundTowardPositive in defaults sets negative overflow to formats largest finite number and negative overflow to +infinity.

In the case of overflow, the overflow flag will also be raised with the inexact flag. Underflow in floating-point standard occurs when a number is a small non-zero value. Underflow can be detected either before rounding or after rounding. Underflow before rounding occurs when the exponent range is between $+2^{emin}$ or -2^{emin}. After rounding underflow is detected when non-zero values exponent range and precision range lie between $+2^{emin}$ or -2^{emin}. Underflow is handled by always returning a rounded result which can be one of three things: zero, subnormal or between $+2^{emin}$ or -2^{emin}. [4, 3]

2.4 Special cases

In the posit standard there are only three special cases: infinity, zero and NaN for invalid operations such as root of a negative number. [1] In the IEEE 754 floating-points, multiple exception cases are associated with arithmetic and status flags. Floating-point arithmetic throws exception NaN in IEEE 754-2008 standard when performing invalid operations such as adding infinities, dividing by zero or trying to get the square root of a negative number. As specified earlier, there are two kinds of NaNs qNaN and sNaN. sNaNs can be used for example as a uninitialized variables that do not appear in default operational mode. sNaNs signals exception when they are operands. qNaNs are used for debugging by enabling and disabling traps.

Traps are different invalid operations. If the trap is disabled and invalid operations for that given trap occur, result will be qNaN. [3] qNaN and sNaN binary encodings can vary by different processor architectures, but the IEEE 754 standard specifies binary to present qNaN and sNaN when $E = bias$ and $T \neq 0$. In IEEE 754 standard there are two kinds of zeros +0 and -0. They are encoded in binary by all exponent and fraction fields zero. The sign of the zero value is again determined by the sign bit. In IEEE 754 standard there are two kinds of infinities as well: +infinity and -infinite. Infinite is encoded in binary by having all the exponent bits as 1 and all the fraction bits as 0. The sign bit again determines the sign of infinity. In the IEEE 754 floating-point standard there are also numbers called subnormal numbers which are numbers that are non-zero but smaller than the format's smallest normal number. In addition, one special case in the IEEE 754 floating-point standard is the inexact flag that is raised when the result is not the exact. [4] In posit standard there are no flags.

2.5 Rounding

Floating-point arithmetic has different rounding methods due to the nature of floating-point operation results only sometimes being exactly presentable in floating-point systems. [3] In the IEEE 754-2008 standard there are four different rounding methods specified:

- round toward negative
- round toward positive
- round toward zero
- round-to-nearest

The first three rounding algorithms work similarly as in overflow and underflow situations described in cp. 2.4. If round towards negative is used, number is rounded towards a smaller number or equal to the value before rounding. Rounding towards positive produces a larger or equal number to non-rounded value. In floating-point standard these rounding operations can produce +infinite or -infinite. Round toward zero has the same functionality as round toward negative but does not produce -infinity as an answer. [3, 4]

Default rounding used in floating-point arithmetic as well as in posits is the round-to-nearest method [3, 2, 1]. The round-to-nearest method has two options: roundTiesToEven (round to nearest even) or roundTiesToAway. In roundTiesToEven number is rounded to the value closest to the intermediate result. If the intermediate result is equally close to two different numbers, the one with integral significand bit being even is chosen. roundTiesToAway works similarly as roundTiesToEven but when two numbers are equally close to the intermediate result, one that is further from zero is chosen as the result. [4] In the table 2.1 below different rounding method functionality is shown with example values.

Table 2.1. *Different rounding methods and their impact on real value*

Rounding method	11.5	12.5	-11.5	-12.5
to nearest, ties to even	12.0	12.0	-12.0	-12.0
to nearest, ties away from zero	12.0	13.0	-12.0	-12.0
toward 0	11.0	12.0	-11.0	-12.0
toward negative (-infinite)	11.0	12.0	-12.0	-13.0
toward positive (+infinite)	12.0	13.0	-11.0	-12.0

In this thesis the chosen rounding method for floating-point units is round-to-nearest, because posits use that rounding method.

3. IEEE 754 HARDWARE ARITHMETIC

The IEEE 754 standard is a widely used standard for representing floating-point numbers. In computer systems and in arithmetic operations, addition and multiplication are one of the most important operations to be performed. This chapter describes steps for performing addition and multiplication in the IEEE 754 standard. Furthermore, dual-path hardware implementation designs for addition and multiplication are examined.

3.1 Addition

The basic implementation of the addition algorithm for floating-points can be described well in steps. First, the inputs' *x* and *y* exponent fields are compared to ensure that $e_x \geq e_y$ if this does not hold true the inputs are swapped. Then the smaller input's significand will be aligned by right-shifting it by the exponent difference of $e_x - e_y$. After significand alignment, the significand will be either added or subtracted according to the sign bits of the inputs. Simple XOR logic can determine the operation with the input sign bits as follows $s_x \oplus s_y$. If the result significand is negative, then negation will be performed. After these steps result of sum is achieved. This result is then normalized in two cases. [3]

- Carry out in the significand addition
- Cancellation in the significand addition

If the carry out in the significand addition occurs, the result significand will be right-shifted by one digit and the result exponent will be incremented unless the result exponent equals e_{max}. In that case, overflow is signaled. If cancellation occurs, the number of leading zeros will be determined and the result significand is then left-shifted by the number of leading zeros, and the same number will be subtracted from the result exponent. If, after performing these steps the results exponent is smaller than e_{min} the exponent will be set to e_{min} and the result significand will be left-shifted by $e_r - e_{min}$. After normalization, rounding and exception handling will be performed. Possible exceptions in addition operation are invalid operation, overflow, underflow and inexact. [3] Below in figure 3.1 the hardware implementation of dual-path floating-point addition/subtraction is shown.

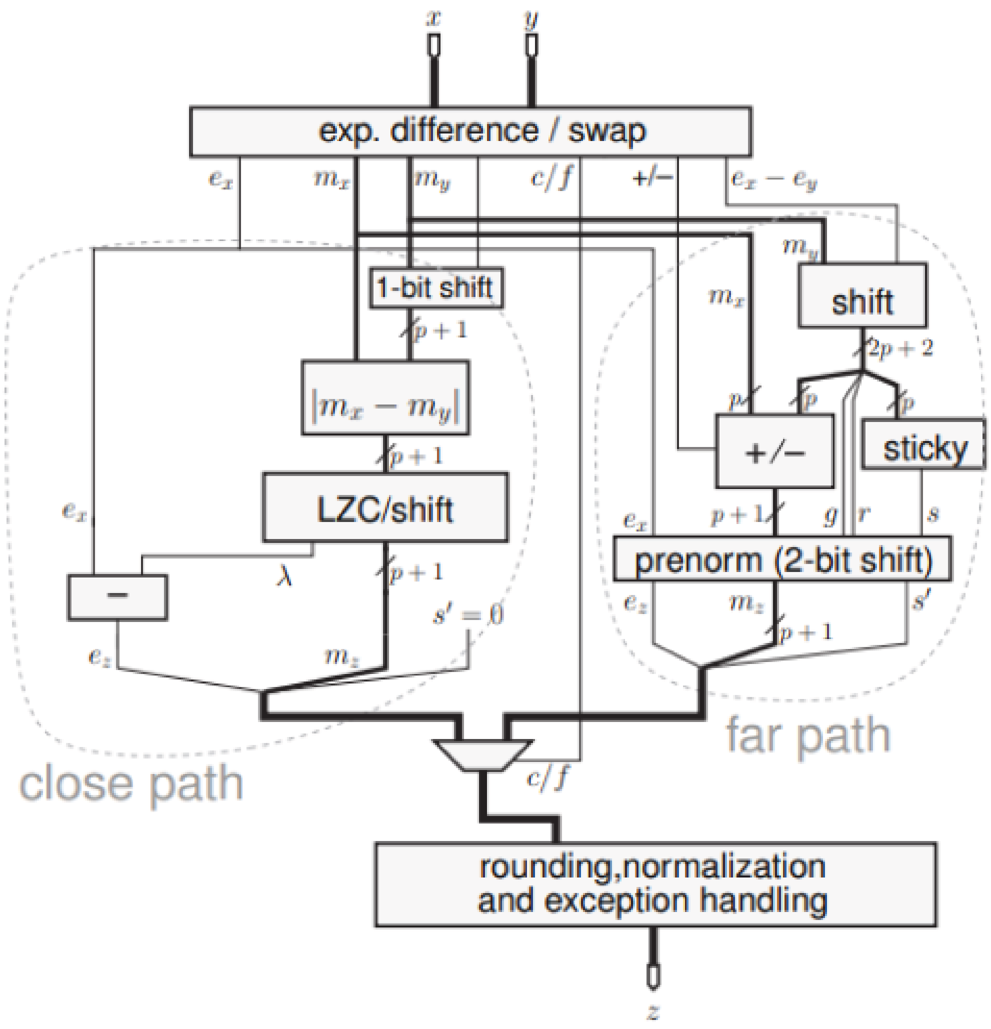

Figure 3.1. *a dual-path floating-point addition hardware implementation. [3]*

This dual-path hardware implementation works in steps similarly as mentioned before. First, input exponents are compared and possibly swapped. Then, from inputs, the extracted information, the exponent and the significand fields are transmitted to according places. The exponent difference calculation determines the amount of shifting on the smaller input on the 'far path'. After the shifting operation, the aligned and larger inputs significand is added or subtracted. Then the intermediate result with possible carry out together with least significand bits of the significand sum and the *g* guard bit, *r* round bit and *s* sticky bit will be inputted to prenorm. In prenorm the following action is decided by the two leading bits. If carry out has occurred (MSB=1), the result exponent will be set to $e_z = e_x + 1$ and new sticky bit s' will be computed using the guard, the round and sticky bit from the previous step by $s' = g \vee r \vee s$. If the leading bits are "01", the result significand is obtained by removing the leading zero from the MSB, and guard bit is concatenated to the LSB and the result exponent will be set to the exponent of the larger input and $s' = r \vee s$. If leading bits are "00", the result exponent will be set to $e_x - 1$ and $s' = s$. After prenorm the result exponent, significand and sticky bit from the far-path will be transmitted to the multiplexer before final procedures.

The 'close path' is for cases where the exponent difference between inputs is at most 1 and cancellation occurs. In the close path, the subtraction of the significands will always be performed. If the resulting significand is a negative, negation of the significand will be done. LZC (leading zero detector) and shifting will be done for the cancellation case as mentioned before. Multiplexer then chooses which of the two paths results are chosen as the intermediate result before rounding. Finally, normalization and error handling phase is performed. [3]

3.2 Multiplication

Multiplication in floating-points is more simple than addition. Multiplication result of the inputs *x* and *y* is achieved by the product of the significands m_x and m_y multiplied by the base β which is 2 for binary number system and 10 for decimal number system. The power of β is the exponent sums e_x and e_y as follows [3]:

$$m_x m_y \times \beta^{e_x+e_y} \tag{3.1}$$

From the formula above, the computation of the multiplication result can be divided into two parts: one for mantissa multiplication and the other for exponent sum. Below in figure 3.2 such an implementation is illustrated. [3]

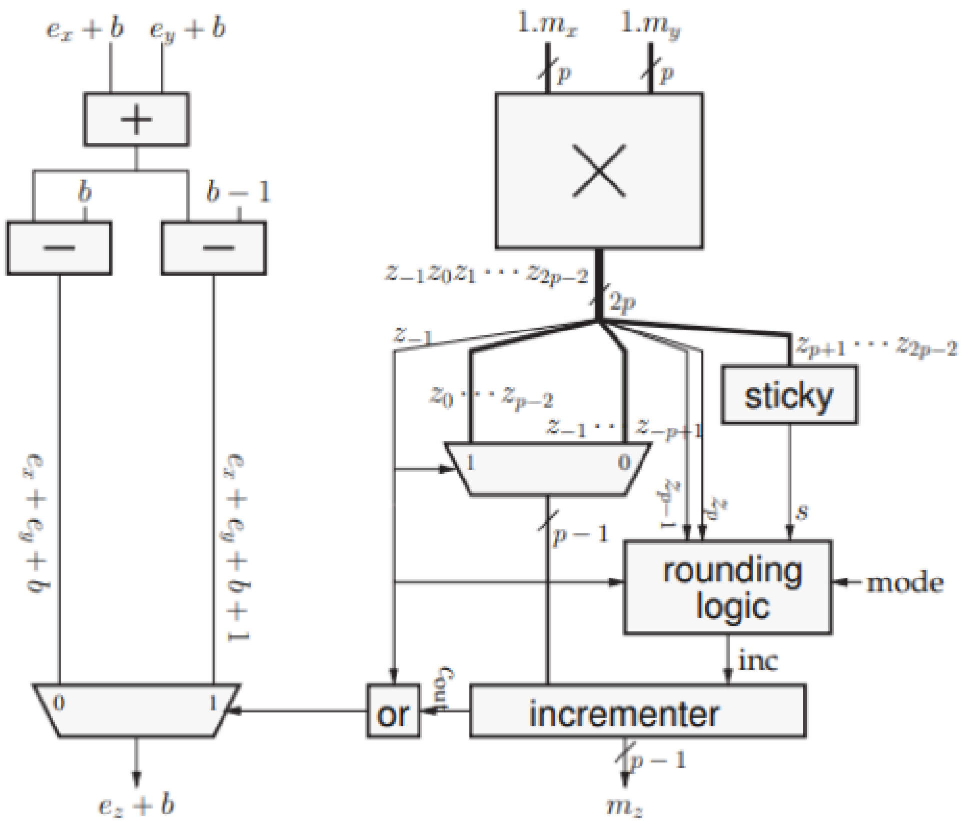

***Figure 3.2.** Multiplication of floating-points. [3]*

On the left side of the figure 3.2 occurs exponent computation and on the right side mantissa multiplication and rounding. Exponent computation is quite a straightforward operation of adding *x* and *y* inputs exponents together with bias *b*. After the exponent sum, the bias is subtracted from the result of addition and in the case of the carry bit, one is subtracted. These two computations in this given implementation example occur parallely and both computations are transmitted to a multiplexer controlled by OR-port from the mantissa multiplication side. On the multiplication side the signficands m_x and m_y are first multiplied and the sticky bit is used for rounding and can be obtained from that operation directly. After rounding, final normalization will be performed and if there is a carry-out the exponent result with the added one is chosen from the exponent computation side. Possible exceptions in multiplication are invalid the operation, overflow, underflow, and inexact [3]

4. POSIT HARDWARE ARITHMETIC

Posits have been recently developed as an alternative to traditional floating-point arithmetic. In this chapter, the steps for performing addition and multiplication operations are described. Posit hardware implementation usually consists of three parts: decoding the posit, operational core and encoding the posit. In posit decoding, the different bit fields are separated for the operational core. In the operational core, the wanted operation is performed and finally in the encoding part different parts are decoded to achieve the final posit result. Furthermore, the conversion from posit to a real value formula is shown with examples.

4.1 Conversion from posit to a real number

To better understand how different parts of the posit format affect the corresponding real value of the posit number, conversion from posit to the real formula is presented. The posit number real value can be achieved by the following formula

$$s \times (2^{(2^{es})})^k \times 2^e \times f \tag{4.1}$$

where *s* is the sign bit value, *es* exponent bit amount, *e* exponent value, *k* regime value and *f* fraction value. The fraction field also includes hidden bit 1. If the posit value is negative, a 2's complement of the posit value will be taken. After possible 2's complement posit fields and their values will be decoded from the posit. [5, 6, 7] Few examples of 8-bit posit conversion to a real value are shown below with a two different number of exponent bits *es*. Fields are separated with a dash.

ES=2

$$10001111 \rightarrow 2'c \rightarrow 01110001 \rightarrow 0{-}1110{-}00{-}1 \rightarrow -(2^{(2^2)})^2{*}2^0{*}(1{+}(1.0/2)) = -384$$

$$00001111 \rightarrow 0 - 0001 - 11 - 1 \rightarrow +(2^{(2^2)})^{-3} * 2^3 * (1 + (1.0/2)) = 0.0029296875$$

ES=3

$$00000111 \rightarrow 0 - 00001 - 11 \rightarrow +(2^{(2^3)})^{-4} * 2^6 * (1 + 0.0) = 1.4901210^{-8}$$

$$11101011 \rightarrow 2'c \rightarrow 00010101 \rightarrow 0-001-010-1 \rightarrow$$
$$-(2^{(2^3)})^{-2} * 2^2 * (1+(1.0/2)) = -0.0000915527$$

In the first example the sign bit is one; thus the 2's complement must be taken. After 2's complement the value is "0111 0001" next the regime, the exponent and the fraction bits need to be determined. The first regime bit is 1 after the sign bit. The regime field consists of consecutive bits of first regime bit and is then terminated by the termination bit. The termination bit is the opposite value of the regime bit. From the example value "0111 0001", the regime is the "111" after the sign bit "0" terminated by the "0" thus the run length is 3. The value *k* can be computed from run length minus 1 since the regime sign bit is 1, *k=RunLength-1=3-1=2*. If the regime sign bit would have been 0, the *k* value would be just the run length of consecutive zeros. Next, the exponent bits and their corresponding values are determined. The *es* value is two, thus there can be up-to two exponent bits. Exponent bits appear after termination bit. In this example the next two bits are "00" thus, the corresponding exponent value *e* is 0. The rest of the bits are fraction bits. The only bit left is the fraction bit 1. The calculation shows that the fraction value is calculated by adding the hidden one bit to the fraction value divided by 2^{fBits} where *fBits* is the amount of fraction bits. [5, 6, 7]

4.2 Extracting information from a posit number

In posits the information extracting part is always done first. This part is also one of the main differences between IEEE standard floating-points and posits in their hardware implementations. In posits, first the special cases of zero and infinity are checked. After that, the sign bit is taken from the MSB place: if the sign bit is one, then the 2's complement will be taken from the rest of the posit value - except the sign bit part. After the possible 2's complement the regime sign bit is determined from the next bit of the MSB. Regime sign bit is then used to determine the run length of the regime using a leading one detector, leading zero detector or parametrizable detector. In these detectors, the number of consecutive regime sign bits is computed. If the regime sign bit is 1 the run length of the regime is the number of consecutive bits minus one. If the regime sign bit is 0, the run length value is the number of consecutive bits. After the regime field is achieved the posit value is left-shifted by the run length value to determine the exponent and fraction bits. The MSB after shifting is the first exponent bit. The number of exponent bits is known. Thus, the exponent bits can be computed from the MSB to the right by the number of available exponent bits. Finally, rest of the bits, if any are left, are fraction bits. [8, 5, 6]

4.3 Addition core

Posit addition core works very similarly as depicted in IEEE 754 floating points in cp 3.1 after decoding of input posit operands into corresponding parts sign, regime, exponent and fraction is performed. First, the operation is computed from the sign bits by xor-operation. In addition, special cases of infinity and zero are checked if not already done in the decoding part. Then scaling factors are computed by concatenating input operands' regime and exponent bits and comparing the absolute values of inputs. After larger and smaller operands are found, the offset is used to right-shift the smaller operands' fraction bits. The offset is computed by subtracting the scaling factors and taking the absolute value of the result. Next, fraction bits are either subtracted or added depending on the operation. Then possible overflow is checked and the normalization of fraction bits will be performed by inserting the intermediate fraction bit value to LZD where the number of leading zeros once again shifts fraction bits and then the scaling factor will be adjusted by adding larger scaling factors fraction bits with possible overflow bit and subtracting the leading zero count from normalization LZD. [8, 5]

4.4 Multiplication core

The posit multiplication core is much simpler than addition. First, special cases are checked and scaling factors are constructed similarly as in addition core by concatenating regime and exponent bits from the decoder. After that, multiplication operation of fraction bits is performed, the possible overflow is checked, and the multiplication fraction result is normalized. Finally, the scaling factors are added together with the overflow bit value, which can be either 0 for no overflow or 1 for the overflow case. [5]

4.5 Constructing a posit number

Posit numbers after the core operation are usually transmitted to a new algorithm that encodes the parts produced in the core operation back to the final posit value. First, the exponent bits and regime bits are extracted from the adjusted scaling factor and the regime sign bit is determined from the MSB of the regime. If the regime bit is 1, the absolute value of the regime is taken and one is subtracted from the regime value. Exponent and normalized fraction are concatenated and then shifted by the regime value. After that, rounding bits LSB, G, R and S are determined from the shifted intermediate value. LSB is the bit that is the final bit of n-value posit but after the LSB there are still more bits for the rounding purpose. After LSB, in the given order is G guard, R round and rest of the bits are sticky bits S and the value of the sticky bit is computed by bitwise or-operation for the rest of the bit array. The rounding value is then computed from these rounding bits as follows:

$$Round = G \wedge (LSB \vee R \vee S) \tag{4.2}$$

Finally, the posit value is constructed by concatenating the result sign bit with the intermediate result and adding a rounding bit. [8, 5]

5. ARITHMETIC ACCURACY ESTIMATION

There are a few direct studies on accuracy differences between posits and the IEEE-754 floats. Supposedly, posits are more precise and have a higher dynamic range than floats of the same bit width. Even smaller bit width posits can provide more precise results than larger bit width floats. [9] In some machine learning examples using C-NN, 16-bit posits have produced more precise results than 32-bit floats and even better efficiency [10]. Fewer bits needed in computations would reduce the amount of needed memory. The reason for better accuracy in posits is that fewer bits are reserved for the exponent and thus, more is left for fraction bits. For example, in 16-bit floats there are 5 bits reserved for the exponent and 10 for fraction bits. In 16-bit posits on the other hand, only one bit is reserved for the exponent, and the fraction bits can thus be up to 12 bits. The difference in the number of fraction bits increases when using 32-bit posits and floats [11]. In the 32-bit numbers floats have 8 exponent bits and thus 23 bits for fraction. In 32-bit posits there are only 2 exponent bits; thus, the fraction bits can be up to 27 bits. In addition to the fraction bit amount, the amount of special cases in posits is only two, leading to more number representation with the same amount of bits compared to floats where there are multiple different special cases [10].

The amount of fraction bits varies in posit numbers due to regime bits. Regime bits are used to scale the values with exponent bits and get posits to have tapered accuracy. [2] Tapered accuracy means that the accuracy of posits will decrease towards maximum and minimum values but be more accurate around zero where most of the computations in general occurs. [11, 2]

In addition to better accuracy, posits have better dynamic range than floats, at least from 8 to 32 bits. Dynamic range can be calculated using the smallest positive finite value and the largest positive finite value. The formula for the dynamic range is

$$log_{10}(maxpos/minpos) \tag{5.1}$$

where *maxpos* is the largest positive finite value and *minpos* the smallest positive finite value. If dynamic range for 8-bit posits and floats would be calculated using one exponent bit for posits and using so-called quarter-precision floats where there are 4 exponent bits, would the dynamic range of floats be 5.1 decades and for posits 7.2

decades. Posits thus have 2 decades better dynamic range compared to floats in quarter-precision example. The table below lists dynamic ranges of floats and posits for bit widths from 8 to 256. Even though floats do not have an 8-bit standard, for comparison purposes floats with 3 exponent bits are used.

Table 5.1. *Dynamic ranges of floats and posits with different bit widths [2]*

Size, Bits	IEEE Float Exp. Size	Approx. IEEE Float Dynamic Range	Posit *es* Value	Approx. Posit Dynamic Range
8	3	$1.56 * 10^{-2}$ to 15.5	0	$1.52 * 10^{-2}$ to 64
16	5	$6 * 10^{-8}$ to $7 * 10^{4}$	1	$4 * 10^{-9}$ to $3 * 10^{8}$
32	8	$1 * 10^{-45}$ to $3 * 10^{38}$	3	$6 * 10^{-73}$ to $2 * 10^{72}$
64	11	$5 * 10^{-324}$ to $2 * 10^{308}$	4	$2 * 10^{-299}$ to $4 * 10^{298}$
128	15	$6 * 10^{-4966}$ to $1 * 10^{4932}$	7	$1 * 10^{-4855}$ to $1 * 10^{4855}$
256	19	$2 * 10^{-78984}$ to $2 * 10^{78913}$	10	$2 * 10^{-78297}$ to $3 * 10^{78296}$

From the table 5.1, posits have noticeably better dynamic range compared to floats in 8, 16 and 32 bits. After that, the difference decreases slightly. The posit dynamic range could be made larger using more exponent bits, but the accuracy aspect would then worsen. Accuracy is determined in posits and floats using decimal accuracy. Decimal accuracy determines how many decimals the floating-point values are accurate to the correct answer. Decimal accuracy is calculated in decals by the following formula in posit and float-like rounding systems

$$-log_{10}(|log_{10}(x/y)|) \tag{5.2}$$

where *x* is the correct value and *y* the computed value. Decimal accuracy for a given 8-bit example is shown in figure 5.1 below.

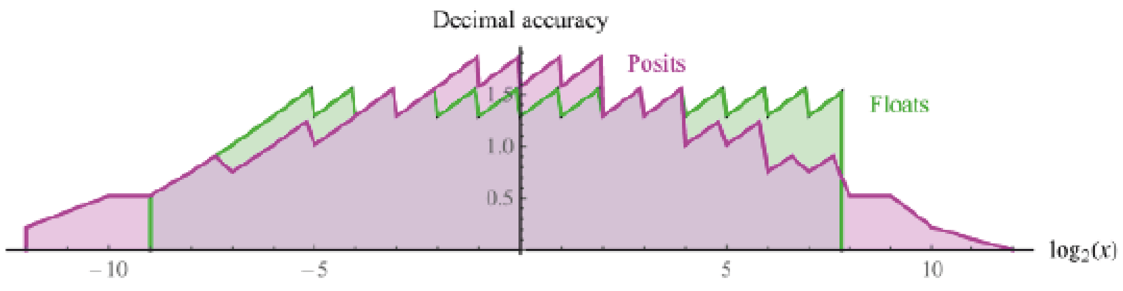

Figure 5.1. *Posit and IEEE-754 floating point decimal accuracy illustrated [2]*

From fig. 5.1, the decimal accuracy of posits is better near the middle, and floats have a slightly better accuracy outside the middle. The figure also shows the nature of tapered accuracy in posits towards the minimum and maximum area. In floats, on the other hand, decimal accuracy drops directly down when numbers get towards maximum due to many different NaN representations near maximum value. Towards minimum, floats also decrease in accuracy due to the gradual overflow of floats. In tapered accuracy, smaller term computation suffers loss in precision but in these given situation the mantissa bits would anyway be shifted out. Therefore posits are more efficient than IEEE floats, since IEEE floats do extra computation with the mantissa bits that would be shifted out in the end anyways [11].[2]

5.1 Addition and subtraction

Posits and floats can both operate addition and subtraction with the same implementation, so in the given example addition with negative input operands is used to perform subtraction. This same 8-bit example of posits and floats can be illustrated on the following two figures to see which parts produce exact, inexacts, overflows, underflows and NaN's in 8-bit all exhaustive addition test. In addition to special cases, the decimal loss is illustrated. In the fig. 5.2 below the computations of all exhaustive inputs addition tests for floats and posits are shown.

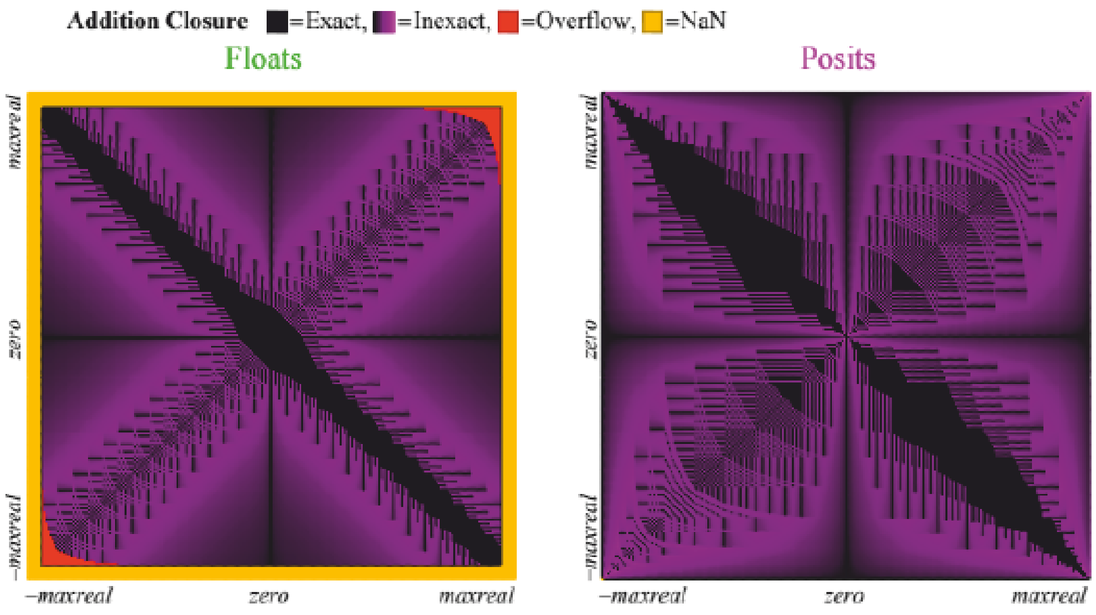

***Figure 5.2.** Posit and IEEE-754 floating point complete closure plot for 8-bit all exhaustive test in addition/subtraction [2]*

The figure shows that overflow occurs in floats in maximum negative real and in maximum positive real areas. In posits overflow does not occur. Diagonally from maximum positive real to other operands maximum positive real occurs all the exact answers in both posits and floats. In posits the exact answer diagonal is larger at different points. Noticeably, floats have NaN rectangle rounding the closure plot; this illustrates that floats have NaN operation all around the exhaustive closure plot, when compared to posits where there are no NaN's. Below in fig. 5.3 the decimal loss and percentage of different output flags are shown..

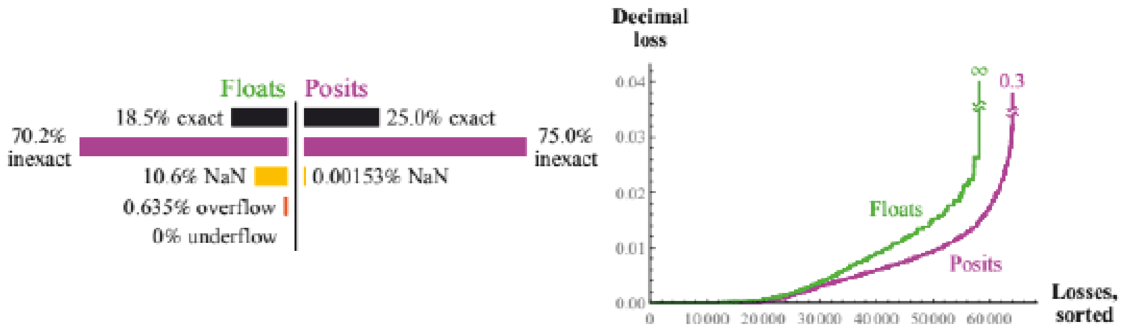

Figure 5.3. Posit and IEEE-754 floating point decimal loss for 8-bit all exhaustive test in addition/subtraction [2]

The figure shows that the floats decimal loss seems to overflow to infinity when the losses sorted amount increases and the posits on the other hand seem to gap to 0.3. The figure shows that the decimal loss of posits is significantly less than in floats. Also, the exact answer percentage is over 7% higher than in floats. Furthermore, the amount of NaN's in floats is over 10% which is inefficient compared to zero NaN operation of posits.

5.2 Multiplication

The same 8-bit exhaustive test was also done for multiplication. Similarly, the closure plot and the decimal accuracy figures and the output flag percentage are shown below.

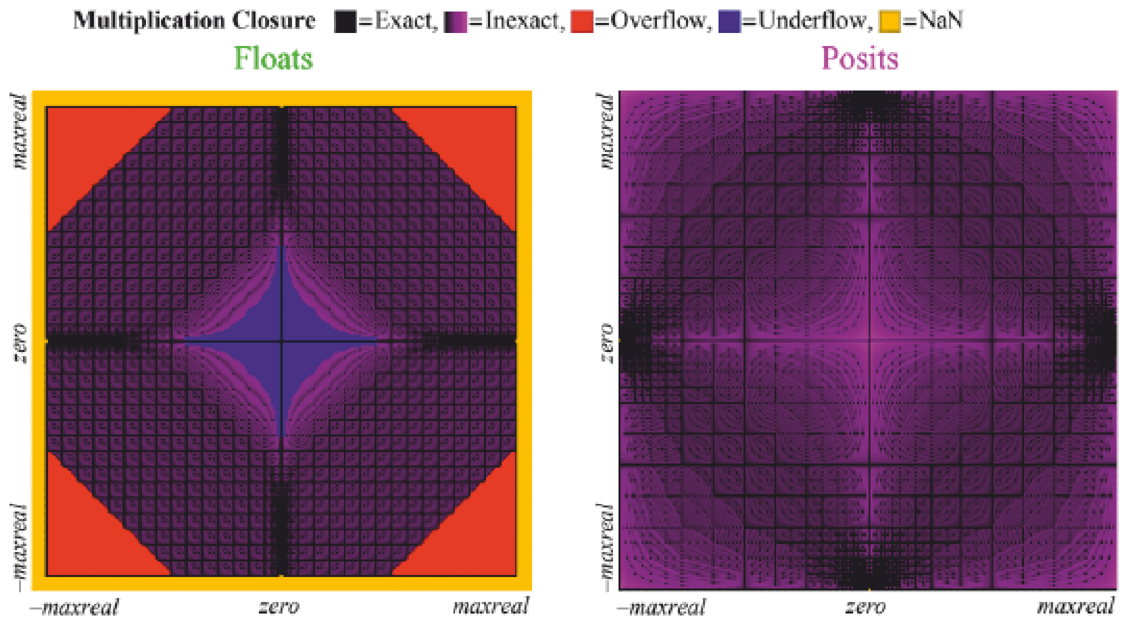

***Figure 5.4.** Posit and IEEE-754 floating point closure plot for 8-bit all exhaustive test in multiplication [2]*

From the figure 5.4 Can be seen the same NaN rectangle on the border of floats as in addition. Also, there are a lot more overflow cases in the corners. Furthermore, now there are underflows as well. Underflows appear in the middle - except in the direct center there is the effect of gradual underflow. In posits there are again no major overflow, underflow and NaN cases. Two NaN points cannot be directly identified from the graph but they occur from $infinity \times zero$ and $-infinity \times zerooperations$. Below the percentages and decimal loss graph is shown.

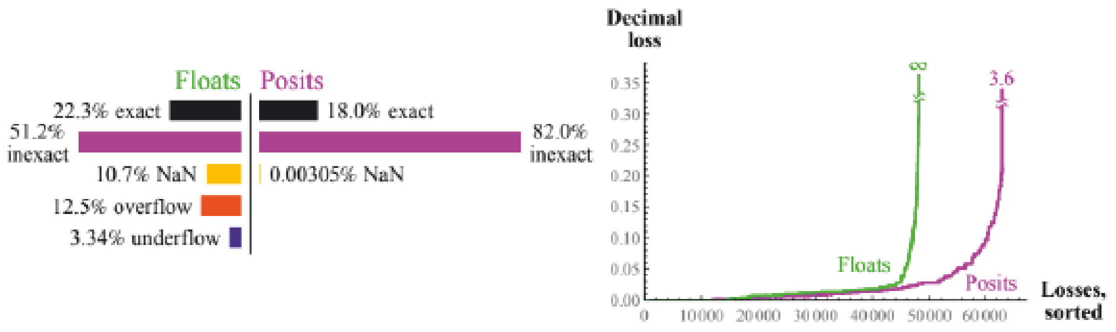

***Figure 5.5.** Posit and IEEE-754 floating point decimal loss for 8-bit all exhaustive test in multiplication [2]*

From the figure 5.5 can be seen that the exact percentage is now in favor of floats. However, there are also a 12.5% of overflow, 3.34% of underflow, and 10.7 % of NaN's. In total, in floats there are no usable results for over 25% of the computed answers. On the other hand, in posits almost all the cases produce a usable result. Decimal loss is also again in favor of posits. Floats again overflow to infinite and posits gap to 3.6 decades. This rounding error happens in the worst possible case of $maxpos \times maxpos$.

6. TESTING ENVIRONMENTS AND TOOLS

Verification is a big part of the system-on-chip design flow, and the time consumed to verification increases as designs get bigger and more complex. Different tools in a system on a chip design flow are used to detect problems early in designs. This chapter shows the implemented testbench for behavioral verification of chosen designs and the design flow. In addition, ASIC and FPGA synthesis are described and the difference between them. Furthermore, the importance of power analysis tools and use case in the thesis is described. Also, a brief introduction to static verification and two tools used in the thesis LINT and LEC are presented.

6.1 Testbench flow in simulations

Testbench used in the thesis was implemented to test the addition and multiplication units of floats and posits. Bit widths tested were 8, 16 and 32-bits. For 8-bit numbers all exhaustive test was used meaning all possible input combinations, negative and positive were covered. For 16-bit and 32-bit numbers similar all exhaustive tests would require much time and memory space. Thus, for these two bit widths, it was decided to use random 500k test cases to save time and still try to cover a good amount of possible cases.

First, the inputs were generated for all three used bit widths. For the 8-bit case python script was used to generate all the different 8-bit values and write them to a two different text files for *x* and *y* inputs. For 16-bit and 32-bit the same python generator was used to generate random values, including special cases. Then the correct values needed to be generated to compare them to the results from the tested designs. The softposit library was used for generating correct results for posits, and for correct floats, softfloat library was used [12, 13]. The posit library does not accept posit values as input for addition or multiplication, so the generated input bits needed to be converted to real values before using them as a parameter in the addition and multiplication functions.

Converting of bits to their corresponding posit real value was done by first using the extracting algorithm for posits to extract the bit fields from the posit. Then these extracted values were inserted into a formula 4.1. Python script then converted all the bits used in the testing to their corresponding real value. Similarly floats were converted to their corresponding real value using the formula 2.1. These real values were then again written to a two input text files.

Then these real values were read from the text files and inserted into the corresponding python libraries to get the correct result for comparison purposes. The correct results were then again written to a new text file. After that the testbench was implemented so that different designs are instantiated on the top-level and the wanted design can be chosen with an integer parameter. The testbench worked by reading the inputs from the generated input text files, and results were then computed using the computation unit. After that, computed values were subtracted from the correct result produced by the python libraries. If the result was zero the computed value was correct, and if not result was incorrect. Below in the fig. 6.1 the created top-level of the implemented testbench is shown with the needed parameters.

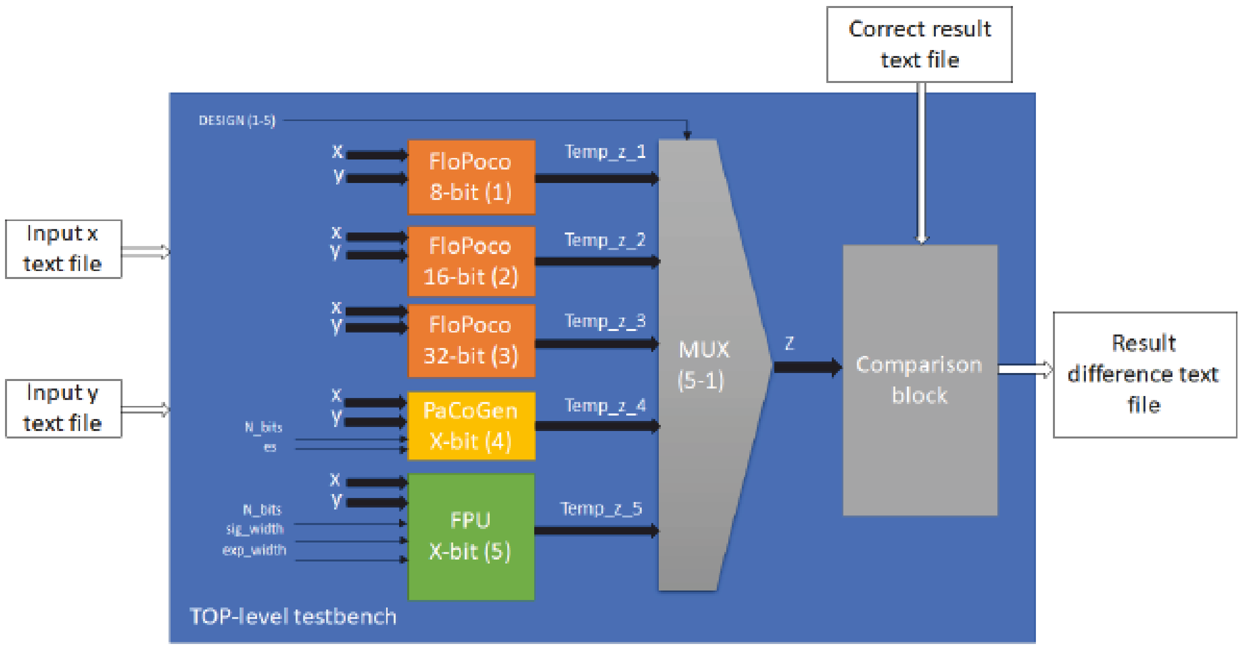

***Figure 6.1.** Testbench flow*

From the figure, five different DUT instances are selected via 5-1 MUX with corresponding design parameters. All the instances get their data *x* and *y* from the input text files. Datapaths in the figure are shown with thicker arrows and logical signals with a thinner arrows. From the figure can be seen the needed parameters: the number of bits *nBits*, the exponent bit amount *es*, the FPU exponent width *expWidth* and the significand width of FPU *sigWidth*. The *tempZ* is the temporary result inserted to the MUX and which is then selected as the final result *z*. The final result is then compared to the correct result from a correct result text file. The difference between the two results is then rewritten to the result difference text file. The same top-level design was done to adder and multiplication separately. In addition, simulation tools were used to identify if there were problems with implementing the testbench design or with the design itself.

6.2 ASIC synthesis

ASIC (Application Specific Integrated Circuit) synthesis or logic synthesis is one of the steps in the design flow in SoCs (System-On-Chips). In ASIC synthesis, the RTL design is converted into a technology-specific gate-level netlist. Successful gate-level netlist has the same logical operation as the RTL but can also be implemented on the SoC. The synthesis is performed by EDA (Electronic Design Automation) tool. EDA tool takes as an input the RTL design, design constraints and the technology libraries and proceeds to create similar logic as in the RTL given and optimize the synthesized design according to the design specific constraints and technology libraries. Design constraints are, for example, related to timing and what specifications need to be met from the synthesized design. Technology libraries are used to optimize the design area and delay further. [14]

In synthesis phase the EDA tools examine the design for things such redundant logic, clock speed and timing constraints. This same examination occurs in FPGA synthesis as well. [15] In this thesis the timing constraints of the design were chosen so that timing violations did not occur. Also two different ASIC synthesis were performed for each design. One with the fastest possible clock speed without timing violations reported by the EDA and one with the slowest clock speed in its corresponding bit width to better compare the areas and power between designs. In the thesis, every design was successfully synthesized and area, delay and power were achieved from all the designs.

6.3 FPGA synthesis and place & route

FPGA (Field-Programmable Gate Array) is a chip where unlike in ASIC the internal logic can be changed after manufacturing. FPGA consists of three configurable components: the logic blocks, the routing elements and the input/outputs blocks. The logic blocks have the wanted logic of the system and routing elements connect the logic to I/O. Most of the FPGA logic blocks consist of LUTs (Look-up tables). LUTs are components that with specific input combinations produce beforehand computed answers. LUTs reduce computation power but require area to store input combinations and outputs. [16, 17] In addition to LUTs the logic blocks contain flip-flops to allow for combinational (non-clocked) or synchronous (clocked) designs.

ASIC chips are a cost-efficient way of manufacturing chips in large amounts but require significant capital to start the production. [17] To reduce the risk of incorrect design, FPGAs are a good way to prototype the design and verify the physical design before starting the production of the ASIC chip. The synthesis in FPGAs works similarly to the ASIC synthesis except the resources available are slightly different than in ASIC and depend on the target FPGA used. In this thesis the target FPGA used for synthesis and place & route is Xilinx kintex-7k325tfbg676-1

FPGA synthesis results are not accurate until the resources are placed and routing elements are routed [15]. Due to this reason, place & route was also run for the FPGA analysis. Place & route operation in this thesis was done only by the EDA automation tool within constraints given to it. Further optimization could be achieved by continuing the place & route analysis. In the thesis, number of LUTs and delay was achieved for all the designs except for the FPU multiplier. The reason for this is unclear but it could be that the tool used in the place and route operation did not support the FPU design from another vendor.

6.4 Power analysis

Power analysis is an essential factor when designing a chip. Low power usage increases battery life if the device is powered by a battery. In addition, low power usage reduces the chip's temperature and thus the chip's reliability is better. [18] The needed amount of power in chips is increasing because more transistors are added to the chips to increase performance. In addition to this, the leakage powers of chips are increasing due to every generation's lowering threshold voltages of transistors. Leakage power means power when the system is in the off state. In addition to leakage power, the total power usage consists of activity power and switching power. [18]

In this thesis, the power analysis is done using a power analyzing tool that with given constraints produces a SAIF file and power related reports. SAIF file is ASCII format to store the switching activity of the design [18]. This SAIF file is then used in a wave tool to examine the waveform of the power on the design. All exhaustive tests for 8-bit posits and floating-points are illustrated in the waveform to see how different input combinations consume power. Only FloPoCo posit design was used to compare to the floating-point unit [8].

6.5 LEC

LEC (Logic Equivalence Check) is one of the technologies of static verification tools used in SoC development. Static verification tools mathematically tests different inputs for signal to show possible problems within the design. This is different to simulations since simulations require input set vector and testbench. In static verification the used tool generates the inputs. Static verification tools thus save time from designing the testbench and in addition static verification tools can cover corner cases that testbenches do not. Best coverage nevertheless is achieved by the combination of both. [19]

The underlying idea in using LEC in verification is to prove that two versions of the design are functionally equivalent [19]. This thesis uses the LEC to verify that the synthesized designs are functionally equivalent to the corresponding RTL design. In the thesis, all the designs passed were equivalent to their design description. LEC runs are generally more useful in bigger designs [19]. The designs used in the thesis are relatively small

6.6 LINT

Lint, in addition to LEC, is a static verification tool used in SoC development. Lint helps speed up the verification flow when developing larger or smaller designs by running the tool while the design is done without needing a testbench. In lint tools, EDA will check RTL design for violations set by rulesets given to the tool. The rulesets can be about unconnected input/outputs, existing loops for example combinational loops, bit-width mismatches on operation etc. [20]

Lint also understands design structures well and for more complicated designs, schematics created by the tool can be viewed to verify that the design looks like supposed to. Also, the schematic view can be used to fix bugs/problems. [20] In this thesis designs were checked before synthesis for violations with the lint tool. No issues were found in any of the chosen designs.

7. RESULTS OF ANALYSIS AND BEHAVIORAL VERIFICATION

In this chapter, testbench ran results for every design, their synthesis results, and power analysis are shown. Area, delay and power are important factors when designing hardware implementations. Thus, these are the main factors examined in this thesis. Behavioral verification results show if the designs have the correct functionality of the standard. This chapter shows that FPU has better area, delay and power results than the chosen posit designs. Furthermore, the FPU produces no errors with the input vectors used. In posit designs, FloPoCo is better at following the posit standard functionality and synthesis scores than PaCoGen.

7.1 Behavioral verification of chosen arithmetic units

In this thesis two different posit design and one FPU design was chosen for comparison. Chosen posit designs were FloPoCo ad PaCoGen [8, 6]. They are different in that the PaCoGen is a parametrizable design where the parameters are amount of bits and exponent bit size and FloPoCo is not parametrizable and has its own leading zero detector unit which also performs shifting [8]. These posits designs were chosen due to already made studies between them and since they are both open-source [21, 22]. The chosen FPU design is from a known vendor. First, to ensure correct results from analysis tools, all the designs were tested with testbench flow described in cp 6.1 and the results of that test are shown in a table 7.1 below.

Table 7.1. Design specific error amounts

Design	Size, Bits	Adder Error Amount	Multiplication Error Amount
FloPoCo [8]	8,0	1	0
	16,1	0	0
	32,2	0	0
PaCoGen [6]	8,0	X	X
	16,1	47 670	442
	32,2	16 425	0
FPU	8,3	0	0
	16,5	0	0
	32,8	0	0

From the table 7.1 above, FloPoco has 1 error in all exhaustive test case for 8-bit numbers, and PaCoGen does not support 0 exponent bits. Thus, results are not achievable from 8-bit PaCoGen. In 16-bit tests, even though using 500k randomized test inputs like described in cp. 6.1, PaCoGen produces almost 50k errors with an adder unit and FloPoCo zero. In 16-bit multiplication the error amount is far less 442 but compared to FloPoCo where the error amount is again zero, it can be determined that there are many error combinations in PaCoGen algorithm. In the 32-bit test PaCoGen adder has with the given 500k tests cases fewer errors than in the PaCoGen 16-bit adder. In 32-bit adder unit PaCoGen produces 16 425 errors compared to FloPoCo zero errors. On the other hand, in multiplication units with 32-bit numbers PaCoGen does not produce an error with the given test cases. FPU unit does not produce errors with the test cases used.

The testbench's error definition is that, if the computed answer from the design unit differs even one-bit from the reference values created by the posit python library [12] or float library [13] is the answer incorrect thus error. All the errors from the table were one-bit off from the value generated by the python libraries. The one-bit error occurred randomly in the result values, so there where no specific position for the error bit. The first idea was to check for flaws with the rounding algorithm because the rounding algorithm adds one-bit to the intermediate value to achieve to final result. However, further investigation showed that the problem is outside the rounding method. The problem seems to occur with some specific input values, but further investigation of this behavior was not continued in the thesis. Nevertheless, the algorithms still need development to perform computation perfectly according to the posit standard. The one-bit difference can be huge in some cases for example the sign of the answer can change.

It is also noticeable that the coverage is incomplete with random test cases for 16-bit and 32-bit cases, so it is possible that in not covered input combinations lies more errors. Because of this the error amounts gotten from the testbench, are not all the errors that can be produced for 16-bit and 32-bit cases, but for 8-bit all the possible input cases are covered. The multiplier results are better because of the simpler design compared to addition.

7.2 Comparison of addition hardware metrics

This chapter shows the ASIC and FPGA synthesis results of addition operation in two posit implementations and one FPU for bit widths of 8, 16 and 32. The results show that area and power increase significantly when the number of bits increases and that FPU achieves better scores than posit designs. However, posit designs have better dynamic range and thus present more numbers with the same bit widths than IEEE 754 floats. Furthermore, the power analysis tool results are shown for FloPoCo and FPU in all exhaustive 8-bit test.

7.2.1 ASIC

Below in table 7.2 is shown the optimized adder results for all the designs. Optimized results mean that the clock constraint was decided by trying the fastest clock speed without timing violation slack.

***Table 7.2.** ASIC synthesis results for optimised adder design units.*

Adders	**Size, Bits** (N, es)	**Area** (μm^2)	**Delay** (ns)	**Power** (μW)
FloPoCo [8]	8,0	324	1.10	118
	16,1	867	1.41	214
	32,2	1779	1.83	328
PaCoGen [6]	8,0	X	X	X
	16,1	1111	1.51	227
	32,2	2525	1.78	424
FPU	8,3	148	1.11	60
	16,5	826	1.43	205
	32,8	1096	0.95	380

From the table, the area score between 8-bit posit FloPoCo and FPU seems to be almost half in FPU compared to posit and in addition to area the power score is half in FPU compared to posit. In 16-bit designs best area, delay and power score is in FPU but the area score is still over five times the 8-bit area. The area increase from 8-bit to 16-bit design is quite siginicant in FPU design. When comparing posit designs FloPoCo seems to have quite a noticeable better area and a better delay score than PaCoGen. In power values the difference on the other hand is not that big. In 32-bit bit designs again, the best area and delay score is with FPU design, but now a slightly better power score is in FloPoCo. It is worth noticing that the FPU design delay score is much better with 32-bit numbers when compared to any other design FPU design delay. The reason for this is not clear. In posit designs area and power score is much better with FloPoCo but the delay is slightly better with PaCoGen. Below in table 7.3 non-optimized ASIC synthesis results are shown. Non-optimized synthesis results mean that the delay is chosen with the same bit-width designs by finding which designs have the slowest clock speed and using that found value with every design. The reason for this is to compare design metrics better.

Table 7.3. *ASIC synthesis results for non-optimised adder design units.*

Adders	**Size, Bits** (N, es)	**Area** (μm^2)	**Delay** (ns)	**Power** (μW)
FloPoCo [8]	8,0	332	1.11	119
	16,1	708	1.51	173
	32,2	1779	1.83	328
PaCoGen [6]	8,0	X	X	X
	16,1	1111	1.51	227
	32,2	2356	1.83	392
FPU	8,3	148	1.11	61
	16,5	324	1.51	82
	32,8	699	1.83	137

From the table 7.3 can be seen quite similar results as in the optimized table 7.2 except that with bigger delay values the area scores get better. This can be best seen in the FPU design. The reason for the area score to get better with a bigger delay is that the synthesis tool does not try to optimize the design according to the delay and tries to optimize the area more.

7.2.2 FPGA

From the table 7.4 below the FPGA synthesis results after automated place & route operation for multiplier designs can be seen. The chosen values were achieved by trying clock constraints so that timing violation slack did not occur.

Table 7.4. *FPGA synthesis results for adders.*

Adders	Size, Bits (N, es)	Area, LUT	Delay (ns)
FloPoCo [8]	8,0	150	10.1
	16,1	376	16.6
	32,2	904	22.6
PaCoGen [6]	8,0	X	X
	16,1	533	16.6
	32,2	1151	22.6
FPU	8,3	118	8.2
	16,5	244	11.5
	32,8	547	16.6

From the table 7.4 can be seen that the FPU design again has a better area and delay scores compared to the posit designs. In posit designs FloPoCo again has a slight advantage over PaCoGen design in terms of area. In delays the designs do not have differences.

7.2.3 Power

The power analysis tool results of adder designs for posit and FPU are shown in this section. The results are from an 8-bit all exhaustive test where all the possible combinations of inputs and outputs are tested. Below is the table 7.5, the results for posit adder are shown separately for register and combinational power groups. Also, their internal, switching, leakage and total power are shown. In addition to that the percentage of the total power in register and combinational power groups are shown.

Table 7.5. Power analysis results for posit adder.

Posit Adder [8] Power Group	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)	%
Register	224	109	69	402	37
Combinational	36	11	651	698	63
Reg + Comb	260	120	720	1100	100

From the table 7.5 can be seen that power consumption occurs mainly in the combinational power group. All the designs in the thesis are entirely combinationals but their inputs and outputs are registered in flip-flops for the purpose of getting delay results. Most of the combinational power is consumed in leakage power. In the register power group internal and switching powers consume the most power. In the figure 7.1 below waveform of one of the power spikes of the posit design is illustrated to see how different input changes affect the power.

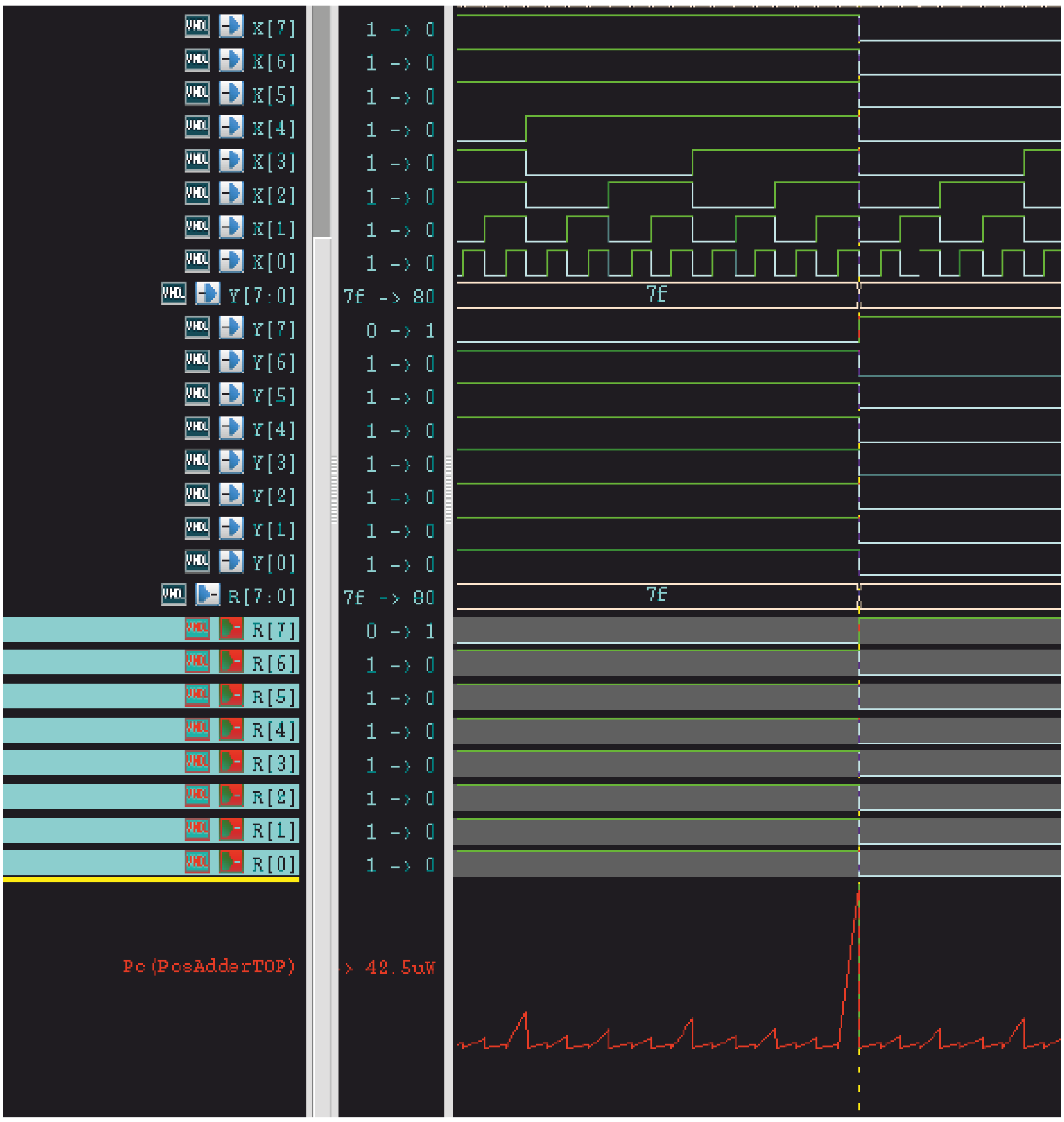

Figure 7.1. *Power graphs of 8-bit all exhaustive test in posit standard.*

From the figure 7.1 can be seen the input *x* and *y* values and their produced output *z* value and the power needed for that computation in the red graph. In this figure the power spike occurs due to many input bits transitions simultaneously seen by the blue arrows. The change in data value bits corresponds to more computation power. The value of this peak in the design is 42,5 μW. Similarly, the FPU adder results from the power analysis tool are shown in the table below.

Table 7.6. Power analysis results for FPU adder.

FPU Adder Power Group	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)	%
Register	0	0	51	51	12
Combinational	79	100	187	367	88
Reg + Comb	79	100	238	418	100

From the table 7.6 can be seen that the registers' internal and switching power appears to be zero and in combinational again the leakage power is the biggest power consumer. The total power of the FPU is less than in posits; this could be because IEEE-754 has so many special cases and thus, there is less computation for actual answers. From the figure 7.2 below power peak of the FPU adder is shown similarly as in the posit adder.

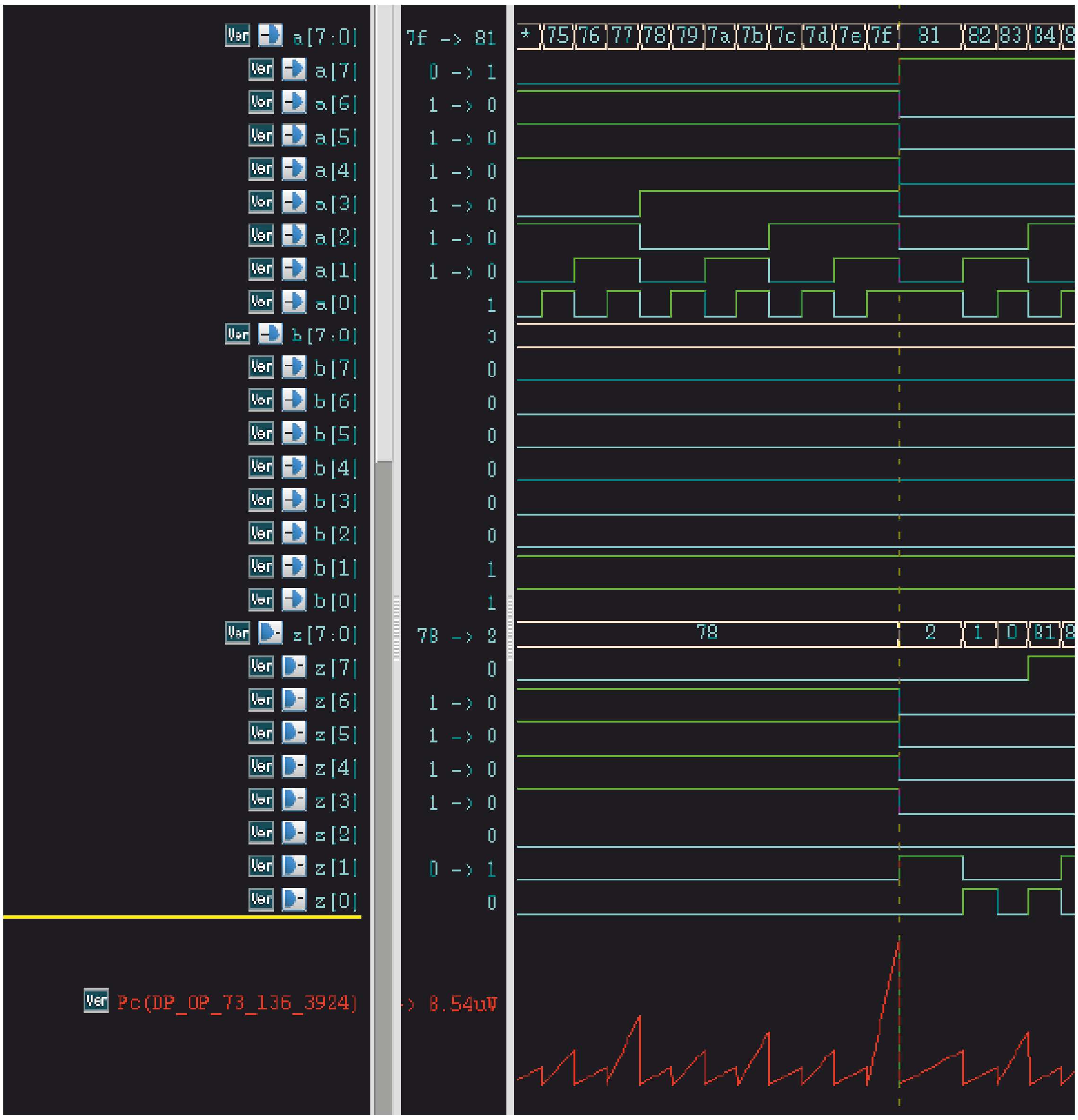

Figure 7.2. Power graphs of zero case in 8-bit IEEE-754 floating-point standard.

From the figure 7.2 can be seen that the peak occurs similarly as in posits when many input values change simultaneously. The power peak value in the figure is 8,54 μW.

7.3 Comparison of multiplication hardware metrics

In this chapter, the results for two different posit designs and one FPU design are shown in multiplication. Synthesis results again show that the FPU achieves better results in synthesis than posits. Furthermore, FloPoCo outmatches PaCoGen. The area scores are slightly lower in 8 and 16-bit cases in posits but rise significantly when bit width increases to 32 bits. Also, the power analysis tool shows that multiplication uses less power than addition.

7.3.1 ASIC

Below similarly as in adders the optimized design results of ASIC synthesis in multipliers are shown in the table 7.7. The definition of optimized is similar to adders. Optimized designs were found by trying different clock speeds and those values that were fastest without timing violations were chosen for each design.

Table 7.7. ASIC synthesis results for optimised multiplier design units.

Multipliers	Size, Bits (N, es)	Area (μm^2)	Delay (ns)	Power (μW)
FloPoCo [8]	8,0	385	0.90	150
	16,1	1032	1.26	332
	32,2	3118	1.55	867
PaCoGen [6]	8,0	X	X	X
	16,1	1279	1.38	354
	32,2	3478	1.64	856
FPU	8,3	236	0.53	165
	16,5	583	0.77	288
	32,8	1727	0.94	744

Similarly to adders, the FPU design in the 8-bit case has a better area and delay score than the 8-bit posit FloPoCo design. Compared to optimized results in adders in table 7.2 the area scores are smaller and delay scores are noticeably better in both FPU and FloPoCo. Better scores are due to the simplicity of the multiplier design compared to the adder design. Conversely, the power score is slightly better in 8-bit FloPoCo posit design. In 16-bit designs again the best area and delay score is with FPU and also the power score is better. In 16-bit posit designs FloPoCo has a better area, delay and power score than PaCoGen. In 32-bit designs again best area, delay and power score is with FPU. The area scores are significantly higher in 32-bit cases in posits compared to their 16-bit implementation. The difference is almost triple in both FloPoCo and in PaCoGen. Below in table 7.8 the non-optimized synthesis results are shown for multiplier designs, as in cp.7.2 with adder designs.

Table 7.8. ASIC synthesis results for non-optimised multiplier design units.

Multipliers	Size, Bits (N, es)	Area (μm^2)	Delay (ns)	Power (μW)
FloPoCo [8]	8,0	385	0.9	150
	16,1	827	1.38	245
	32,2	2710	1.64	733
PaCoGen [6]	8,0	X	X	X
	16,1	1280	1.38	354
	32,2	3478	1.64	856
FPU	8,3	144	0.9	68
	16,5	468	1.38	131
	32,8	1482	1.64	377

A similar analysis can be made from the non-optimized table 7.8. FPU has the best area and power scores in the different bit-width designs. The same thing can also be noticed in that the area scores of posit designs increase significantly with the 32-bit design.

7.3.2 FPGA

From the table 7.9 below the FPGA synthesis results after automated place & route operation for multiplier designs can be seen. The chosen values were achieved by trying clock constraints so that timing violation slack did not occur. The FPU design unit multiplier results were not achievable because the FPGA synthesis tool did not support the FPU design.

Table 7.9. FPGA synthesis results for multipliers.

Multipliers	Size, Bits (N, es)	Area, LUT	Delay (ns)
FloPoCo [8]	8,0	100	10.0
	16,1	210	14.0
	32,2	543	21.2
PaCoGen [6]	8,0	X	X
	16,1	231	14.0
	32,2	544	21.2
FPU	8,3	X	X
	16,5	X	X
	32,8	X	X

From the table 7.9, FloPoco has a slightly better area score than PaCoGen in 16-bit design and the area scores in 32-bit design are almost exact. In delay scores the difference was so slight that they have also been chosen to be exact when generating the synthesis. From these results, the designs are similar regarding area and delay in FPGA implementation.

7.3.3 Power

In this section the power analysis tool results for multiplier design are shown similarly as in adder. Below in the table 7.10 and 7.11 internal, switching, leakage and total power for register and combinational power groups are shown. The power waveform peaks of the multipliers were similar to in the adder design, so they are not illustrated in this section to save space.

Table 7.10. Power analysis results for posit multiplier.

Posit Multiplier [8] Power Group	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)	%
Register	135	122	53	310	45
Combinational	2	0	377	379	55
Reg + Comb	137	122	430	689	100

Table 7.11. Power analysis results for FPU multiplier.

FPU Multiplier Power Group	Internal Power (μW)	Switching Power (μW)	Leakage Power (μW)	Total Power (μW)	%
Register	193	83	30	306	71
Combinational	44	9	75	128	29
Reg + Comb	237	92	105	434	100

From the table 7.10 can be seen the posit multiplier power analysis tool results. Most power again is consumed in the combinational power group in leakage power. The part of register power is higher in multiplier than an adder. The total power is less as well in multiplier compared to an adder. In the table 7.11 the results for the FPU multiplier can be seen. The register power is 71% of the whole design and the total power is less than in the posit multiplier.

8. CONCLUSION

In chapter 2 the type-3 unum posit and IEEE 754-2008 floating-point standards were presented. Posit consists of four bit fields: sign, regime, exponent and fraction. The IEEE 754-2008 floating-point consists of three fields: sign, exponent and fraction. The main difference in the formats lies in the fourth field of posits the regime field. The regime field like described in cp. 2.1 can vary in length. The regime field consists of consecutive bits of the first bit after the sign bit terminated by the termination bit which is the opposite bit of the first regime bit. Because the regime field varies in length, there are sometimes no exponent or fraction bits, unlike in IEEE 754 where the number of exponents and fraction bits are fixed. In posits, only parameter value besides bit length is how many bits at most are reserved for the exponent bits. In addition to the number systems and formats, special cases were also introduced. In IEEE 754 there are many special cases compared to posits. IEEE 754 special cases are: qNaN, sNaN, +0, -0, +infinity and -infinity. In addition to these special cases, IEEE 754 has subnormal numbers that are less than the standard's minimum value but are not zero. In posits there are only two special cases, infinity and zero but there is also NaN for illegal operations. Furthermore, operations on overflow and underflow situations were introduced for IEEE 754 floating-point. In posits overflow or underflow do not occur. Finally, in chapter 2 different rounding method functionalities for floating-points were shown. Posits use the round-to-nearest even rounding method and in this thesis chosen FPU performed round to nearest even as well.

In chapters 3 and 4 steps for performing addition and multiplication with IEEE 754 and posits were shown. Posit and IEEE 754 perform addition and multiplication in a primarily similar way. The main difference is that posits require sequential logic on encoding the bit fields due to the varying length nature of the regime field. The operational core works primarily similarly in both of the standards. Rounding logic is also slightly different than in IEEE 754 and the rounding formula was shown in 4.2. In addition to the hardware implementation description the formula for conversion of posit to real value was shown in 4.1. The reason for this was to illustrate how different bit fields affect the posit real value. Similarly, for floats formula for conversion of IEEE floating-point to real value was shown in 2.1.

In chapter 5 accuracy results of Gustafson studies comparing IEEE 754 floats to posits were examined. In addition, different ways of calculating accuracy and precision were shown with formulas. Furthermore, dynamic ranges of floats and posits were shown with different bit widths in table 5.1. From Gustafson, studies can be determined that posits have better decimal accuracy near 0 and due to tapered accuracy, the accuracy decreases towards minimum and maximum values. IEEE 754 floats have steady accuracy for larger area but the accuracy near zero, where most of the computation occurs is less than with posits. In IEEE 754 floats, accuracy suddenly drops toward maximum values. The table 5.1 also showed that posits have significantly better dynamic range compared to floats in 8, 16 and 32-bit cases. After that with the used exponent sizes the difference in dynamic range decreases slightly in favor of floats. In addition and multiplication studies in chapters 5.1 and 5.2 can also be seen that the floats have a significant amount of not presentable numbers in addition and multiplication compared to posits such as NaN, overflow and underflow. Because of this the posits are more reliable in producing a number after computations than floats that produce invalid numbers.

In chapter 6 the designed flow of testbench and how input vectors were created was described in addition to what tools were used and why for analyzing the posit and FPU designs. Finally, the tools' results were described in chapter 7. In chapter 7 showed that from the testbench runs PaCoGen produced a significant number of errors in addition operation and multiplication runs with the used input vectors compared to FloPoCo posit design. The chosen FPU did not produce any errors in the testbench runs. Furthermore, from the ASIC and FPGA synthesis metrics, the FloPoCo design outperforms PaCoGen in metrics considering area, delay and power. In addition, PaCoGen cannot perform calculations with 0 exponent bits. However, the FPU design outperformed FloPoCo in the same metrics. In addition to the synthesis results power analysis was done for all exhaustive 8-bit test using the FloPoCo posit design and the FPU design. The power analysis shows what consumes the most power from internal, switching and leakage power and their total power value. Also, from the figures 7.1 and 7.2 can be seen that simultaneous change in signals produce spiked in the waveforms.

In this thesis the goal was to investigate and compare IEEE 754 floating-point standard to posit type-3 unum posit standard in addition and multiplication operations and to determine if posit would be a good replacement for widely used IEEE 754 floating-points. This goal was reached and the conclusion is that posit has the potential to replace floats due to having better accuracy around one and dynamic range than floating-points. However, the synthesis results show that the FPU design has a significantly smaller area and better delay and power scores than any of the designs used in the analysis. Furthermore, the replacement would be difficult because many processors would need to be compatible for posit arithmetic. Future investigation should concern more on how to get posit designs more efficient and further analysis in accuracy compared to floats. Overall, type-3 unums represent an exciting new development in computer arithmetic, and they have the potential to revolutionize the way we perform calculations in floating-point arithmetic.

REFERENCES

[1] John Gustafson. *Notebook on posit*. Aug. 2017. URL: https://posithub.org/docs/Posits4.pdf.

[2] John Gustafson and I. Yonemoto. “Beating Floating Point at its Own Game: Posit Arithmetic”. In: *Supercomputing Frontiers and Innovations* 4 (June 2017), pp. 71–86.

[3] Jean-Michel. Muller. *Handbook of Floating-Point Arithmetic*. 1st ed. 2010. Boston, MA: Birkhäuser Boston, 2010.

[4] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (2008).

[5] Rohit Chaurasiya et al. “Parameterized Posit Arithmetic Hardware Generator”. In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018, pp. 334–341.

[6] Manish Kumar Jaiswal and Hayden K.-H. So. “PACoGen: A Hardware Posit Arithmetic Core Generator”. In: *IEEE Access* 7 (2019), pp. 74586–74601.

[7] Manish Kumar Jaiswal and Hayden K.-H. So. “Architecture Generator for Type-3 Unum Posit Adder/Subtractor”. In: *Proceedings - IEEE International Symposium on Circuits and Systems*. Vol. 2018-. 2018.

[8] Raul Murillo, Alberto A. Del Barrio, and Guillermo Botella. “Customized Posit Adders and Multipliers using the FloPoCo Core Generator”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5.

[9] Alapati Madhu Sravya, N. Swetha, and Asisa Kumar Panigrahy. “Hardware Posit Numeration System primarily based on Arithmetic Operations”. In: *2022 3rd International Conference for Emerging Technology (INCET)*. 2022, pp. 1–8.

[10] Stefan Dan Ciocirlan et al. “The Accuracy and Efficiency of Posit Arithmetic”. In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 2021, pp. 83–87.

[11] Florent de Dinechin et al. “Posits: The Good, the Bad and the Ugly”. In: CoNGA’19. Singapore, Singapore: Association for Computing Machinery, 2019.

[12] Cerlane Leong. *SoftPosit-Python*. 2020. URL: https://gitlab.com/cerlane/SoftPosit-Python/-/tree/master/softposit.

[13] Berkeley. *Berkeley Softfloat 3*. 2020. URL: https://gitlab.com/qemu-project/berkeley-softfloat-3.

[14] Weng Fook Lee. *Verilog Coding for Logic Synthesis*. 1st edition. Wiley Interscience Imprint, 2003.

[15] Gina Smith. *FPGAs 101: Everything You Need to Know to Get Started*. Elsevier Science & Technology, 2010. ISBN: 1856177068.

[16] Eduardo Augusto. Bezerra. *Synthesizable VHDL Design for FPGAs*. 1st ed. 2014. Springer International Publishing, 2014.

[17] Peter Wilson. *Design Recipes for FPGAs: Using Verilog and VHDL*. Oxford: Elsevier Science, 2015.

[18] Rakesh Chadha and J Bhasker. *An ASIC Low Power Primer: Analysis, Techniques and Specification*. 1. Aufl. Springer-Verlag, 2013.

[19] Ashok B Mehta. "Static Verification (Formal-Based Technologies)". In: *ASIC/SoC Functional Design Verification*. Springer International Publishing AG, 2017, pp. 193–220.

[20] "Increasing designer productivity at the RT level". In: *Electronic Engineering* (Feb. 2001), p. 22.

[21] RaulMurillo. *Flo-Posit*. 2022. URL: https://github.com/RaulMurillo/Flo-Posit.

[22] manish-kj. *PACoGen*. 2019. URL: https://github.com/manish-kj/PACoGen.