Posit Arithmetics

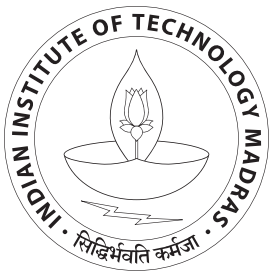
A Project Report

submitted by

P.NAVEEN KUMAR

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2018

THESIS CERTIFICATE

This is to certify that the thesis entitled **Posit Arithmetics**, submitted by **P.Naveen Kumar**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. V.Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 10 May 2018

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my guide Dr. V. Kamakoti for all the valuable advice and motivation he gave from time to time. Despite of a very busy schedule, he always gave me a patient hearing. His knowledge and an extraordinary ability to lighten the students with his positive approach towards things always infused me with great energy and positivity. The invaluable inputs and suggestions from him enabled me to achieve the desired goals during the project work.

I would like to extend a special thanks to my faculty advisor Prof. Saurabh Saxena who continuously supported me throughout my course with his knowledge and advise. I would like to extend special thanks to Arjun Menon, Vinod and my lab-mates who have been very supportive during the course of this project. They have enriched the project experience with their knowledge of the subject matter, active participation, deep understanding and invaluable suggestions.

I would also like to mention special thanks to Dr. John L. Gustafson, the man behind the idea of this posit representation for his constant support through e-mails for whatever doubts I have posed to him during the course of this project.

Last but not the least, I would like to thank my parents and all of my friends for their constant encouragement and support.

ABSTRACT

KEYWORDS: Computer Arithmetic; Floating points; Posits; Unum computing; Valid arithmetic.

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

The HDL that has been used to implement the posit operations is Bluespec System Verilog (BSV).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
NaN	Not-a-Number
FLOPS	Floating Operations Per Second
POPS	Posit Operations Per Second
BSV	Bluespec System Verilog
HDL	Hardware Description Language
FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuits

CHAPTER 1

Floating Point Arithmetic

1.1 Introduction

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rational, and represent every number as the ratio of two integers. Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.ABC might be represented as $1.23ABC \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of ”sliding window” of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

1.2 Scientific Notation of Floating Point Numbers

Floating-point notation can be used conveniently to represent both large as well as small rational and mixed numbers. This makes the process of arithmetic operations on these numbers relatively much easier. Floating-point representation greatly increases the range of numbers, from the smallest to the largest, that can be represented using a given number of digits. Floating-point numbers are in general expressed in the form

$$N = m \times b^e \tag{1.1}$$

where m is the fractional part, called the significand or mantissa, e is the integer part, called the exponent, and b is the base of the number system or numeration. Fractional part m is a p-digit number of the form ($\pm d.dddd..$) with each digit d being an integer between 0 and b-1 inclusive. If the leading digit of m is nonzero, then the number is said to be normalized.

In the case of decimal, hexadecimal and binary number systems will be written as follows:

Decimal System:

$$N = m \times 10^e \tag{1.2}$$

Hexadecimal System:

$$N = m \times 16^e \tag{1.3}$$

Binary System:

$$N = m \times 2^e \tag{1.4}$$

For example, decimal numbers 0.0003754 and 3754 will be represented in floating-point notation as 3.754×10^{-4} and 3.754×10^{3} respectively. A hex number 257.ABF will be represented as $2.57ABF \times 16^{2}$. In the case of normalized binary numbers, the leading digit, which is the most significant bit, is always '1' and thus does not need to be stored explicitly. Also, while expressing a given mixed binary number as a floating-point number, the radix point is so shifted as to have the most significant bit immediately to the right of the radix point as a '1'. Both the mantissa and the exponent can have a positive or a negative value. The mixed binary number $(110.1011)_2$ will be represented in floating-point notation as $.1101011 \times 2^3 = .1101011e^{+0011}$. Here, .1101011 is the mantissa and e^{+0011} implies that the exponent is +3. As another example, $(0.000111)_2$ will be written as $.111e^{-0011}$, with .111 being the mantissa and e^{-0011} implying an exponent of -3. Also, $(-0.00000101)_2$ may be written as$-.101 \times 2^{-5}$ =-.$101e^{-0101}$, where -.101 is the mantissa and e^{0101} indicates an exponent of -5.

1.3 Desirable properties

Specifying a floating-point arithmetic (formats, behavior of operators, etc.) requires us to find compromises between requirements that are seldom fully compatible. Among the various properties that are desirable, one can cite:

- Speed: Tomorrow's weather must be computed in less than 24 hours;
- Accuracy: Even if speed is important, getting a wrong result right now is about as bad as getting the correct one too late;
- Range: We may need to represent big as well as tiny numbers;
- Portability: The programs we write on a given machine must run on different machines without requiring modifications;
- Ease of implementation and use: If a given arithmetic is too arcane, almost nobody will use it.

CHAPTER 2

IEEE Standard 754 Floats

2.1 Evolution

The world of numerical computation changed much in 1985, when the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic was released [1]. This standard specifies various formats, the behavior of the basic operations and conversions, and exceptional conditions. As a matter of fact, the Intel 8087 mathematic co-processor, built a few years before, in 1980, to be paired with the Intel 8088 and 8086 processors, was already extremely close to what would later become the IEEE 754-1985 standard. Now, most systems of commercial significance offer compatibility with IEEE 754-1985. This has resulted in significant improvements in terms of accuracy, reliability, and portability of numerical software. William Kahan played a leading role in the conception of the IEEE 754-1985 standard and in the development of smart algorithms for floating-point arithmetic.

IEEE 754-1985 and 854-1987 have been under revision since 2001. The new revised standard, called IEEE 754-2008 in this book, merges the two old standards and brings significant improvements. It was adopted in June 2008 [2].

The standard defines:

- Arithmetic formats: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
- Interchange formats: encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form

- Rounding rules: properties to be satisfied when rounding numbers during arithmetic and conversions
- Operations: arithmetic and other operations (such as trigonometric functions) on arithmetic formats
- Exception handling: indications of exceptional conditions (such as division by zero, overflow, etc.)

2.2 IEEE Standard 754 Representation

An integer can be represented as a binary string of certain amount of bits which is a combination of three fields: sign, exponent and fraction as below:

S	Exponent	Fraction

$$(X)_{10} = (-1)^S \times 2^{Exponent-Bias} \times (1.Fraction)_2 \tag{2.1}$$

The MSB of the bit string is the sign(S) of the number. If MSB is 0, the number is positive and if it is 1 the number is negative. The n-bit exponent field is an unsigned integer. The exponent field should be able to represent both positive and negative exponent value. To achieve this we add a bias of $2^{n-1} - 1$ to the actual exponent while representing as IEEE float. When it comes to their precision and width in bits, the standard defines four basic formats : Half precision, Single precision, Double precision and Quadruple precision. The Bias coefficient is equal to 127 for an 8bit exponent of a single precision format and 1023 for 11 bit exponent of 64 bit double precision format. This allows useage of exponents in the range of -127 to +128 which corresponds to 0 to 255 for an 8 bit exponent of single precision format and -1023 to +1024 corresponding to 0 to 2047 for a 11bit exponent of double

precision format. The single-precision format offers a range from 2^{127} to 2^{127}, which is equivalent to 10^{-38} to 10^{38} and it ranges from $2^{-1023}$to 2^{1023}, which is equivalent to the range of 10^{-308} to 10^{308}in case of double precision format.[3]

Precision	No.of Bits	Exponent Bits	Fraction Bits	Exponent Bias
Half Precision	16	5	10	15
Single Precision	32	8	23	127
Double Precision	64	11	52	1023
Quadruple Precision	128	15	112	16383

Table 2.1: Characteristic parameters of IEEE-754 formats

2.3 Exceptions of IEEE 754 standard

The standard reserves all 0s and all 1s of the exponent field to handle some exceptional cases:

- **Zero** As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Because of the sign bit there is a distinct representations for -0 and +0, though they both compare as equal.
- **Denormalized Numbers** If the exponent have all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.
- **Infinity** The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

- **NaN** The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN). They have the following format, where s is the sign bit:
  - QNaN: s 11111111 10000000000000000000000
  - SNaN: s 11111111 00000000000000000000001

  A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined. An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage. Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.

2.4 Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

Table 2.2: Special results for some special combination of inputs

Operation	Resut
$n \div \pm Infinity$	0
$\pm Infinity \times \pm Infinity$	$\pm Infinity$
$\pm non - zero \div 0$	$\pm Infinity$
$\pm Infinity + \pm Infinity$	$\pm Infinity$
$\pm 0 \div \pm 0$	NaN
$Infinity - Infinity$	Nan
$\pm infinity \div \pm infinity$	NaN
$\pm infinity$	Nan

2.5 Issues With IEEE Floats

- **Wasted Bit Patterns** - 32-bit IEEE floating point has around sixteen million ways to represent Not-A-Number (NaN) while 64-bit floating point has nine quadrillion. A NaN is an exception value for invalid operations such as division by zero.
- **Mathematically Incorrect** - The format specifies two zeroes, a negative and positive zero which have different behaviors.
- **Overflows to ±inf and underflows to 0** - Overflowing to ±**inf** increases the relative error by an infinite factor, while underflowing to 0 loses sign information.
- **Complicated Circuitry** - One of the reasons why IEEE floating points have complicated circuitry is because the standard defines support for denormalized numbers. Denormalized floating point numbers have a hidden bit of 0 instead of 1.
- **No Gradual Overflow and Fixed Accuracy** - If accuracy is defined as the number of significand bits, IEEE floating point have fixed accuracy for all numbers except denormalized numbers because the number of signficand digits is fixed. Denormalized numbers are characterized by a decreased number of significand digits when the value approaches zero as a result of having a zero hidden bit. Denormalized numbers fill the underflow gap (i.e. the gap between zero and the least non-zero values). The counterpart for gradual underflow is gradual overflow which does not exist in IEEE floating point.

CHAPTER 3

Posit Arithmetcis

Whats wrong with IEEE 754?

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

3.1 Background: Type I and Type II Unums

The unum (universal number) format is a format similar to floating point, proposed by John Gustafson as an alternative to the now ubiquitous IEEE 754 format.

3.1.1 Type-I Unums

”Type I” unum is a superset of IEEE 754 Standard floating-point format[4]. It uses a ”ubit”at the end of the fraction to indicate whether a real number is an exact float or lies in the open interval between adjacent floats. While the sign, exponent, and fraction bit fields take their definition from IEEE 754, the exponent and fraction field lengths vary automatically, from a single bit up to some maximum set by the user. Type I unums provide a compact way to express interval arithmetic, but

their variable length demands extra management. They can duplicate IEEE float behavior, via an explicit rounding function.

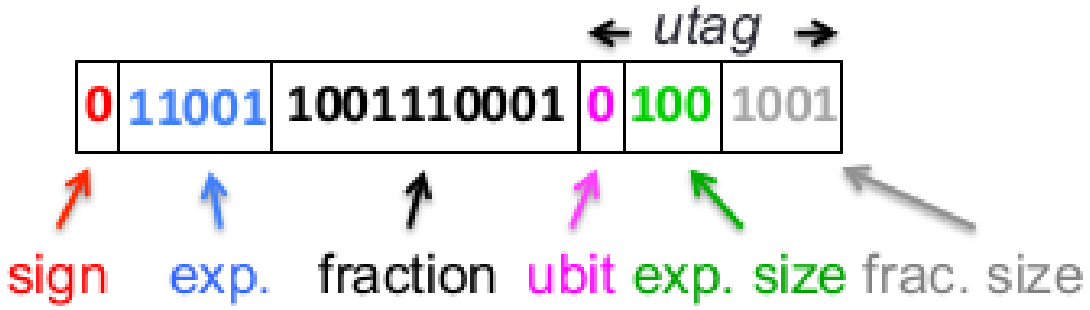


Figure 3.1: Type I unum representation

A u-bit which determines whether the unum corresponds to an exact number (u=0), or an interval between consecutive exact unums (u=1). In this way, the unums cover the entire extended real number line $[-\infty, +\infty]$.

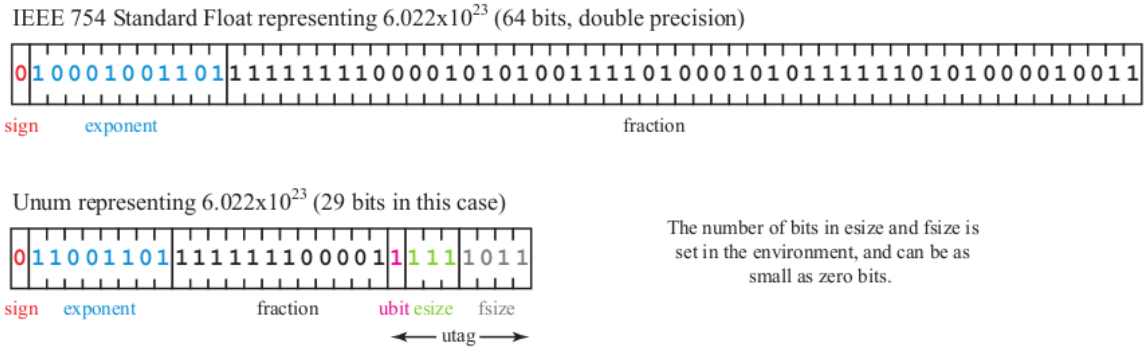


Figure 3.2: Comparison of IEEE 754 64-bit float and a Type 1 unum representing the same value

The inclusion of the "uncertainty bit" (ubit) at the end of the fraction eliminates rounding, overflow, and underflow by instead tracking when a result lands between representable floats. Instead of underflow, the ubit marks it as lying in the open interval between zero and the smallest nonzero number. Instead of overflow, the ubit marks it as lying in the open interval between the maximum finite float value and infinity.

3.1.2 Type II Unums

The "Type II" unum [5] abandons compatibility with IEEE floats, permitting a clean, mathematical design based on the projective reals. The key observation is that signed (two's complement) integers map elegantly to the projective reals, with the same wraparound of positive numbers to negative numbers, and the same ordering. Like Type I, Type II unums ending 1(ubit) represent the open interval between adjacent exact points (unums ending in 0).

The structure of 5-bit Type II unums is shown in fig3.3. For an n-bit Type II unum, the upper right quadrant corresponds to an ordered set of $2^{n-3} - 1$ real numbers x_i. The upper left quadrant represents the negative values of x_i, the reflection about the vertical axis represents the opposite numbers of the upper right quadrant. The lower semi circle, the reflection about the horizontal axis represents the reciprocals of the numbers in the upper half circle. This horizontal projection makes $\times and \div$ symmetrical operation like + and -.

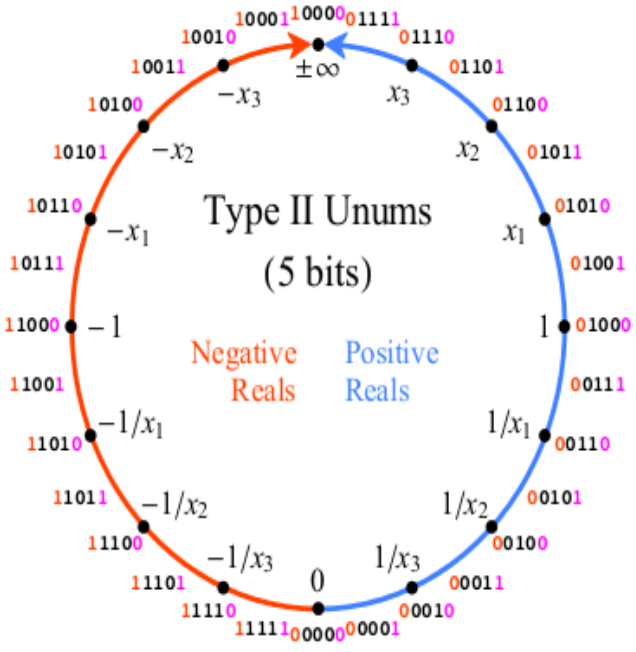


Figure 3.3: Projective real number line mapped to 4-bit two's complement integers

3.2 The Posit Format

The below fig3.4 represents the structure of an n-bit posit with es exponent bits.

The n-bits of posit consists of four fields sign, regime, exponet and fraction.

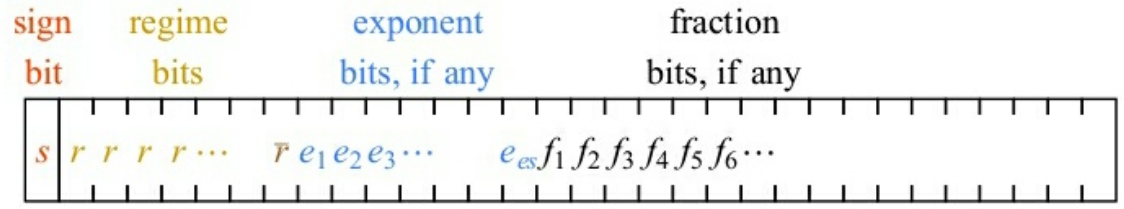


Figure 3.4: Posit format for finite non-zero values

3.2.1 Understanding a n-bit posit

- **Sign**- The MSB of the n-bits represents the sign of the number represented. If it is 0, the number is positive and if 1 the number is negative. If sign bit is 1, take 2's complement of the entire posit before decoding it.
- **Regime bits**- The identical bits after the sign bit either terminated by opposite bit or the end of the binary string is reached contribute to the regime bits. Let there be 'm' identical bits in the regime. If the m bits are 0, then runlength(k)= -m; if they are 1s, then k=m-1. Table below gives more clarification about regime bits and numerical meaning of runlength(k).

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

Table 3.1: Numerical meaning k of regime bits

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

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

Table 3.2: useed as a function of es

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

- **Fraction bits** - The bits left after the exponent bits represent the fraction, just like the fraction $1.f$ in a float, with a hidden bit 1. Posits doesn't support the de-normalized format with hidden bit 0.

3.2.2 Exceptions of Posits

There only two exceptions for posits.

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

The decimal equivalent of a posit is calculated as below:

$$(X)_{10} = (-1)^S \times useed^k \times 2^{exp} \times (1.Fraction)_2 \tag{3.1}$$

3.2.3 Representation of posits as projective reals

For understanding, let us start with the simple case of 3-bit posit. The fig3.5 below shows the right half(positive values) of the projective reals.

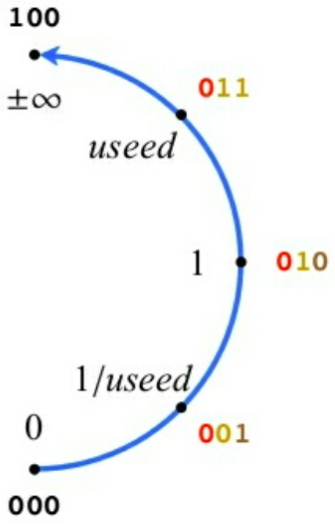


Figure 3.5: Positive values represented by 3-bit posit

The precision of posits increases by appending bits. The value remains where it is on the circle when a 0 is appended. Adding a 1 bit creates a new value between two existing values on the circle. Let *maxpos* be the the largest positive value and *minpos* be the smallest positive value on the ring defined with a bit string. In the above fig3.5, maxpos = useed and minpos = 1/useed.

The interpolation rules are as follows:

- Between the maxpos and $\pm\infty$, the new value is maxpos $\times$ useed; and between 0 and minpos, the new value is minpos / useed.
- Between existing values $x = 2^m$ and $y = 2^n$ where m and n differ by more than 1, the new value is their geometric mean, $\sqrt{x.y} = 2^{(m+n)/2}$ (new exponent bit).
- Otherwise, the new value is midway between the existing x and y values next to it, that is, it represents the arithmetic mean (x + y)/ 2 (new fraction bit).

As an example, fig3.6 below shows a build up from a 3-bit to a 5-bit posit with es=2, so useed = 16.

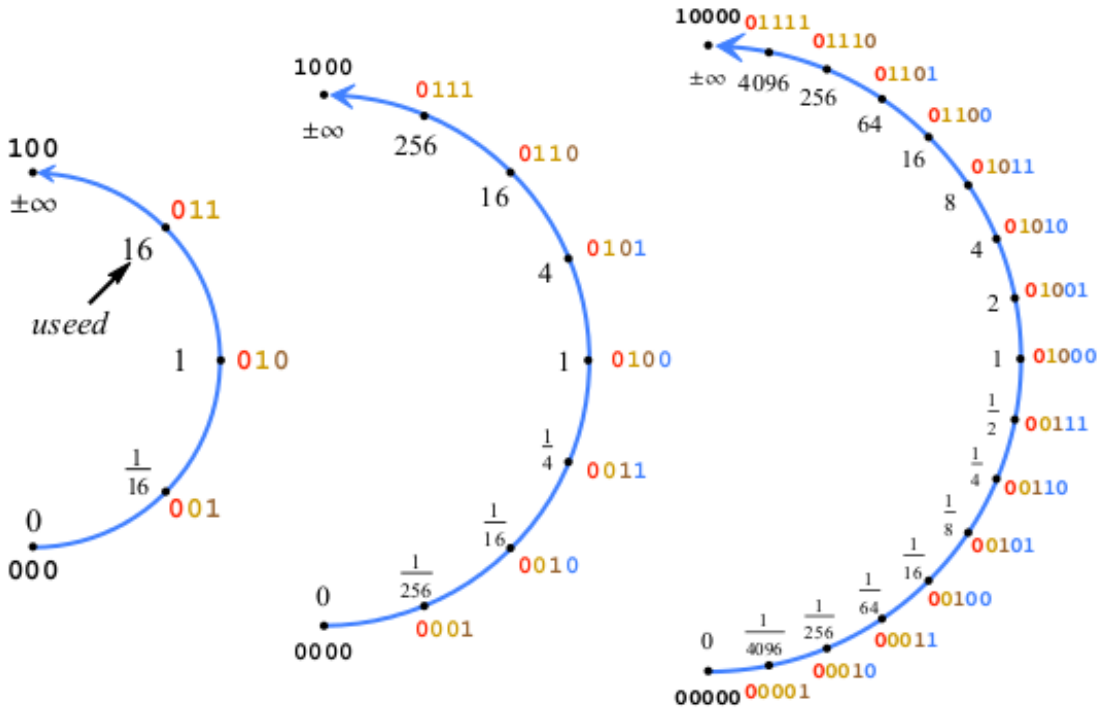


Figure 3.6: Construction of posits with two exponent bits, es=2, useed = $2^{2^{es}}$ = 16

3.2.4 Decoding a Posit

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

The regime and es bits combined serve the function of exponent bits of ieee floats. Together they set the power-of-2 scaling of fraction where each useed increment is a batch shift of 2^{es} bits. The maxpos is $useed^{n-2}$ and minpos is $useed^{2-n}$. In the Fig. 3.7 below an example for decoding a posit shown.

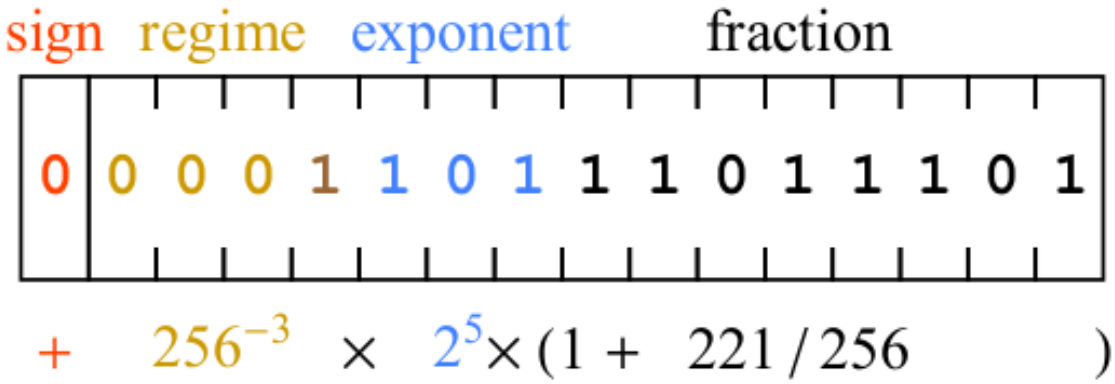


Figure 3.7: Example for mathematical meaning of a posit with es=3

The MSB=0 hence the number is positive. The regime bits are 0001, results in runlength k=-3 and es=3 hence $useed = 2^{2^3} = 256$ hence, the exponent contributed by the regime bits is $256^{?3}$. The exponent bits, 101, the binary number represented is 5 and contribute another scale factor of 2^5. Finally, the fraction bits 11011101 represent 221 as an unsigned binary integer, so the fraction is 1+(221/256). It works out to $1.86328125 \times 2^{-19} = 3.55393 \times 10^{-6}$.

CHAPTER 4

Posits versus IEEE Floats: A Metric-Based Study

4.1 Comparison between and Float and Posit Formats

Posits doesn't support 'NaN'(Not-a-number)s. Instead, the calculation is interrupted and the interrupt handler can be set to report the error and invoke a workaround and continue computing. This simplifies the hardware considerably. Similarly, posits lack a separate $+\infty$ and $-\infty$ like floats have.

There is no "negative zero" in posit representation but IEEE standard supports separate representation for "negative zero". With posits, when a=b, f(a) = f(b). The IEEE 754 standard says that the reciprocal of "negative zero" is $-\infty$ but the reciprocal of "positive zero" is ∞, but also says that negative zero equals positive zero. Hence, floats imply that $-\infty = \infty$.

Floats have a complicated test for equality, a=b. If either a or b are NaN, the result is always false even if the bit patterns are identical. If the bit patterns are different, it is still possible for a to equal b, since negative zero equals positive zero! With posits, the equality test is exactly the same as comparing two integers: if the bits are the same, they are equal. If any bits differ, they are not equal. Posits share the same a ¡ b relation as signed integers; as with signed integers, you have to watch out for wraparound, but you really don't need separate machine instructions for posit comparisons if you already have them for signed integers.

There are no subnormal numbers in the posit format, that is, special bit patterns indicating that the hidden bit is 0 instead of 1 . Posits do not use ”gradual underflow.” Instead, they used tapered precision, which provides the functionality of gradual underflow and a symmetrical counterpart, gradual overflow. (Instead of gradual overflow, floats are asymmetric and use those bit patterns for a vast and unused cornucopia of NaN values.)

Floats have one advantage over posits for the hardware designer: the fixed location of bits for the exponent and the fraction mean they can be decoded in parallel. With posits, there is a little serialization in having to determine the regime bits before the other bits can be decoded. There is a simple workaround for this in a processor design, similar to a trick used to speed the exception handling of floats: Some extra register bits can be attached to each value to save the need for extracting size information when decoding instructions.

4.2 Dynamic Range of Floats Vs Posits

Dynamic range of a number system is defined as the number of decades from the smallest to the largest finite positive number(minpos to maxpos) that can be represented.

$$DynamicRange = \log_{10} maxpos - \log_{10} minpos = \log_{10}(maxpos/minpos) \tag{4.1}$$

For example, an 8-bit posit with es=0, has a value of useed=2, hence the minpos=1/64 ($useed^{2-n}$) and maxpos=64 ($useed^{n-2}$). So the dynamic range is $\log_{10}(64^2)$ is about 3.6decades. The posits with es=0 are simple and elegant but their higher bit versions like 16, 32 and 64 have less dynamic ranges compared to the corre-

sponding IEEE floats. 32 bit posit system with es=0 has dynamic range of about only 18 decades whereas in the case of 32 bit single precision IEEE floats it is about 83 decades.

4.2.1 Using the Useed to Match or Exceed the Dynamic Range of IEEE floats

As we append the exponent bits, the useed keep on scaling by factor of 2^{es}. Consider the case of 32-bit posits with es=1, has useed=4 and the dynamic range is $60 \times \log_{10} 4$ turns out to be around 36 decades. Similarly, for 32-bit posit with es=3, useed=256 and dynamic range is $60 \times \log_{10} 256$ which works out to around 144 decades. This clearly surpasses the dynamic range of single precision IEEE floats.

Bit size	IEEE Float Exp.Size	Approx. IEEE Float Dynamic Range	Posit es value	Approx. Posit Dynamic Range
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	$2 \times 10^{-299}$to 1×10^{298}
128	15	6×10^{-4966} to 1×10^{4932}	7	1×10^{-4855} to 1×10^{4855}

Table 4.1: Dynamic range of Floats Vs Posits for same number of bits

These exponent sizes for floats do not follow any mathematical pattern, but reflect intensely argued compromises by the IEEE committee. Trying to match the 1985-era choices of the IEEE committee results in an equally inexplicable set of es values for the posits. Frankly, the reason for such enormous dynamic ranges is that they were trying to save transistors instead of provide what users really need. Multiplying and dividing floats only requires integer addition and subtraction of

the exponent field, but the fraction field needs an integer multiplier, and the cost of that can grow almost as the square of the number of fraction bits. So while almost no one strays outside the range 10^{-13} to 10^{13} in real applications. But the IEEE 754 Standard proudly lets you go from about 10^{-78984} to 10^{78913}.

4.3 Decimal Accuracy

Decimal error defined as the absolute value of number of decades from the computed value to the exact value.

$$\text{Decimal Error} = |\log_{10}(X_{computed}) - \log_{10}(X_{exact})| = |\log_{10}(X_{computed}/X_{exact})| \tag{4.2}$$

Notice that the absolute value makes $x_{computed}$ and x_{exact} interchangeable in the above definition. Also notice that it produces the same result whether you use $x_{computed}$ and x_{exact} as inputs, or $1/x_{computed}$ and $1/x_{exact}$. So that looks like a mathematically sound definition. We have choose base 10 logarithm because it measures the error in decades. For example, $x_{computed} = 0.001$ and $x_{exact} = 0.0001$; the decimal error is 1. That means it's a decade off.

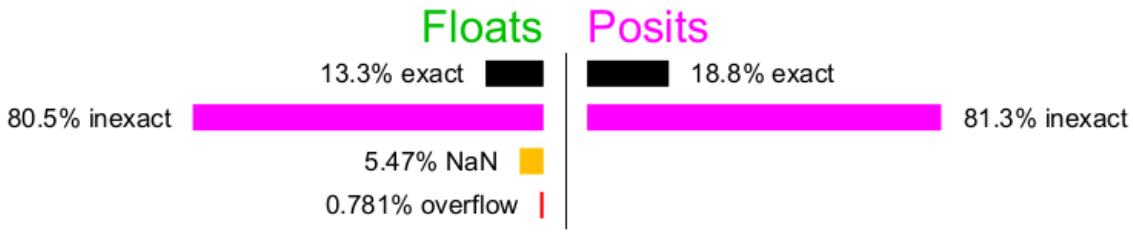
The decimal error can be used to define decimal accuracy. Accuracy is the inverse of error. If we want to know the number of decimals of accuracy, we again take the log base 10.

$$\text{Decial Accuracy} = \log_{10}(\frac{1}{\text{Decimal Error}}) = -\log_{10}|\log_{10}(\frac{X_{computed}}{X_{exact}})| \tag{4.3}$$

4.4 Comparing floats with posits performing unary operations

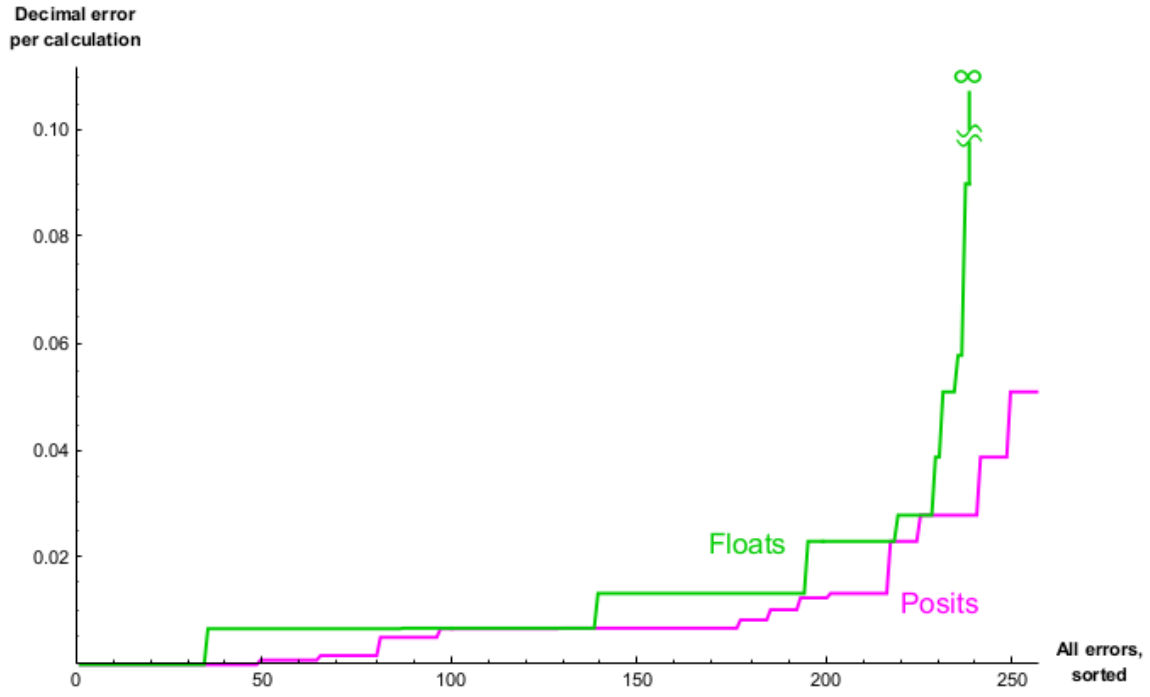
4.4.1 Reciprocation

We can compare the reciprocal closure, that is, the percentage of cases where 1/x is exactly representable as a member of the set. Compare the percentage of the entire set of floats or posits for which a reciprocal is exact, finite but inexact, produces a NaN, overflows, or underflows:



Only 34 of the float values have exact reciprocals. In contrast, 48 of the 256 unum values have exact reciprocals, and never experience catastrophic loss of accuracy through overflow. The IEEE float definition is a ”kludge” in that it has subnormal numbers at the low end (the reciprocals of which incorrectly overflow to infinity), but replaces the high end numbers with NaN values.

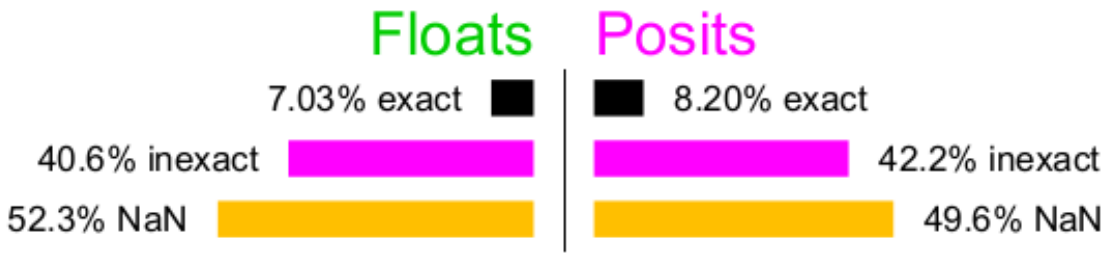
The following graph makes it much easier to visualize the relative performance of floats and posits. The entire set of decimal losses in computing 1/x is sorted from smallest to largest, and plotted.



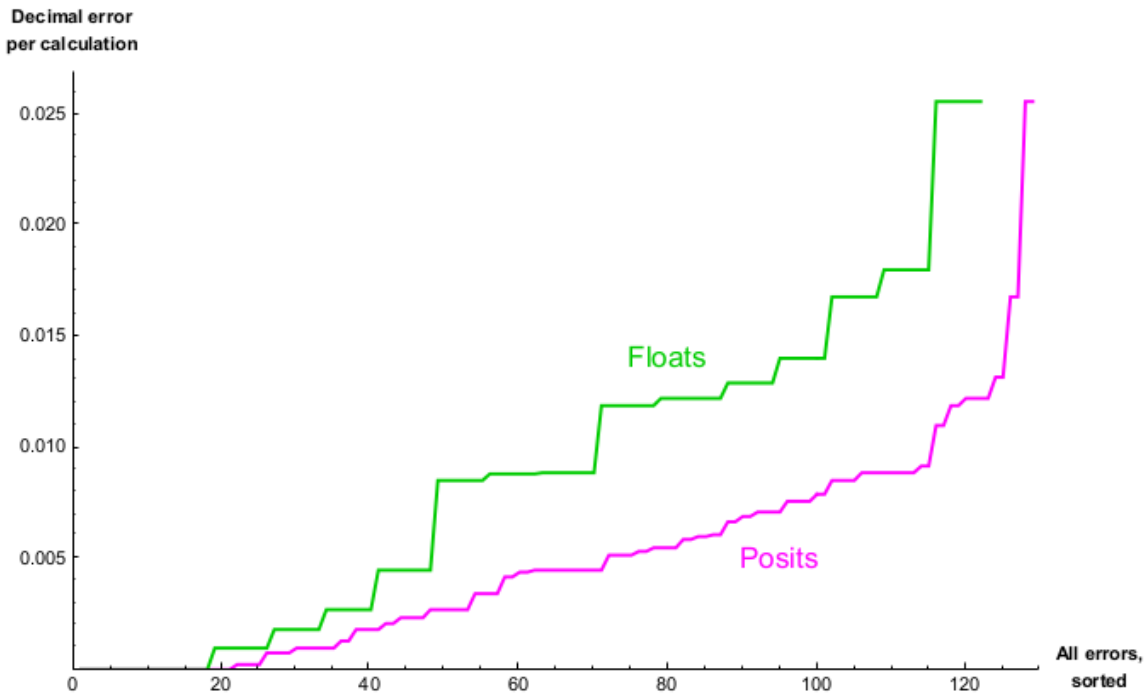
The graph of posit decimal losses grows more slowly than that for floats, and never goes to infinity.

4.4.2 Square Roots

We can also compare square root closure, that is, the percentage of cases where $\sqrt{x}$ is exactly representable as a member of the set. The sqrtbarchart routine also finds out the fraction of the time a square root is exact, finite but inexact, or produces a NaN. The square root operation cannot overflow or underflow. Negative inputs produce NaN results, but since the posit "±∞" really means unsigned infinity, the square root of "±∞" is "±∞" and thus is closed for that input.



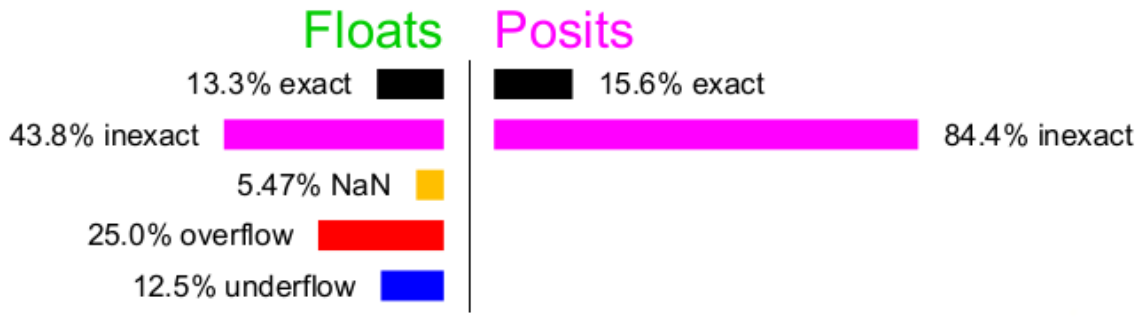
Posits do better, but at first glance it looks like the advantage is slight. The bar chart does not reveal just how much more inexact the floats are. The difference in the sorted losses is more dramatic than it was for computing 1/ x. Here are the sorted losses for every $\sqrt{x}$ value that is not a NaN:



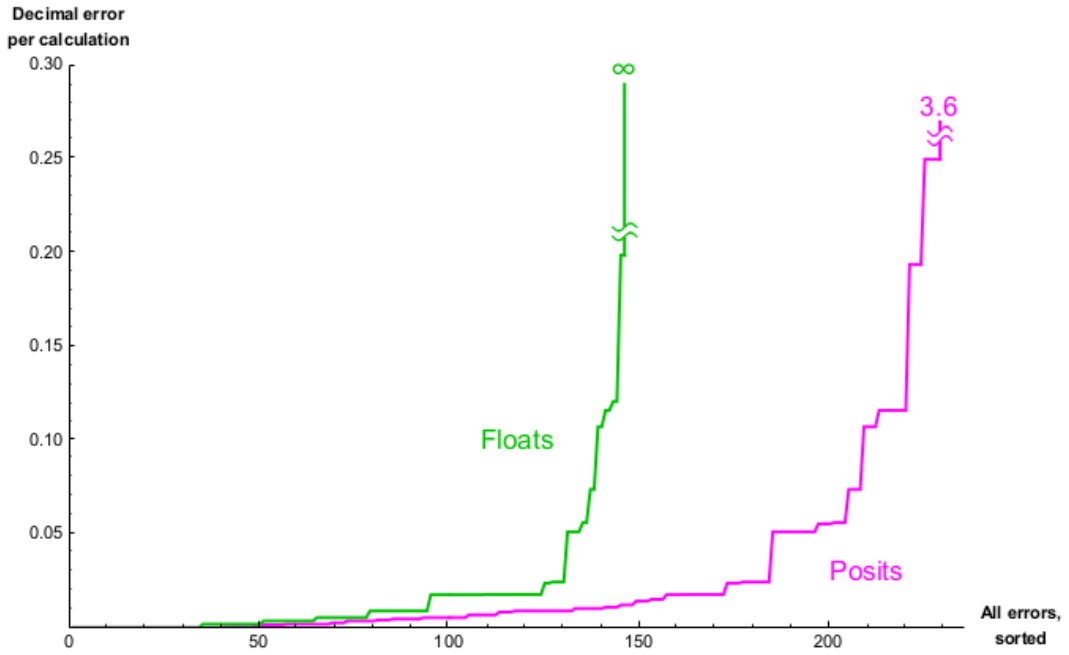
The posit errors are about half those of the floats (for the results that are not indeterminate).

4.4.3 Square

Another common unary operation is x^2 . Overflow and underflow are a common disaster when squaring floats. Posits experience their largest decimal loss for squares that would overflow or underflow by IEEE rounding rules, but at least the loss is a few decimals and not infinite.



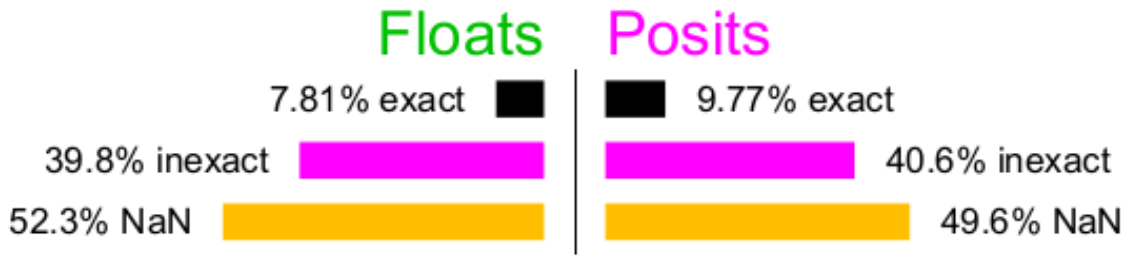
Posits do much better, mainly by not having any exception cases at all.



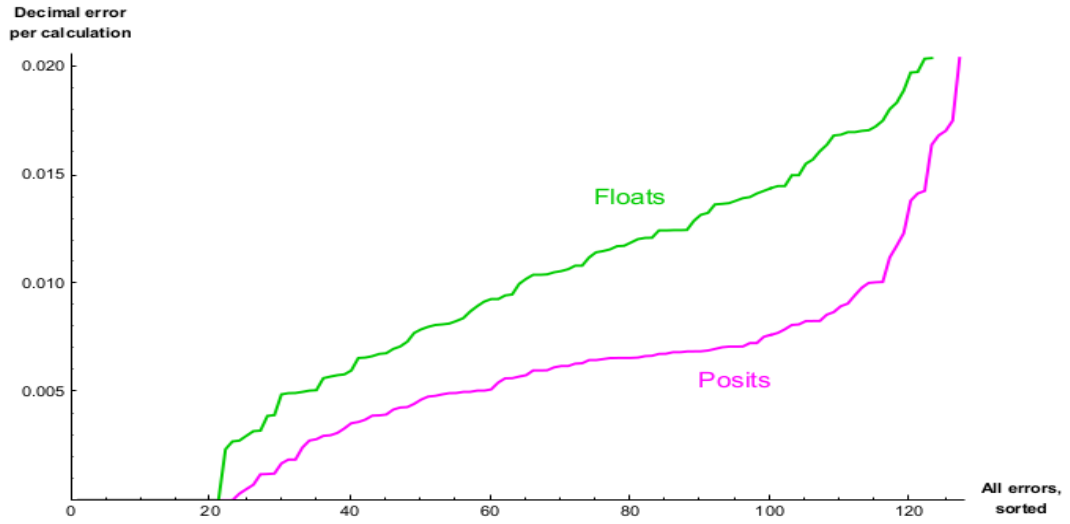
Every posit can be squared. (The square of unsigned infinity is again unsigned infinity.) In contrast, almost half the squarings of floats result in complete loss of information about the result.

4.4.4 Logarithm (base 2)

We can also compare the logarithm base 2 closure, that is, the percentage of cases where $\log_2(x)$ is exactly representable as a member of the set. As with square roots, about half the values produce a NaN since the logarithm of a negative value is a complex number. Note: we allow posits to return $\pm\infty$ as the logarithm of zero and the logarithm of $\pm\infty$. Remember, posit infinity is unsigned, like zero.



Posits do better, and again at first glance it looks like the advantage is slight. There are more integer powers of 2 in the posit environment, for which the logarithm base 2 is expressible exactly. Here are the sorted losses for every value that is not a NaN or infinity:



The posit errors are again about half those of the floats.

CHAPTER 5

Bluespec System Verilog

The hardware decription language used for coding the arithmetic operations on posits is BSV.

5.1 Introduction

Bluespec System Verilog is a Hardware Description Language (HDL), which is used for specification, synthesis, modeling and verification of ASIC and FPGA design. With a radically different approach to highlevel synthesis, bluespec offers significantly higher productivity. It allows designers to express intended hardware through high-level constructs, where all behavior is described as a set of guarded atomic actions.

5.2 Limitataion of Verilog

Verilog focusses more on simulation than logic synthesis. The source text of verilog often explicitly contains aspects of circuit that could be readily determined by the compiler, such as size of registers, width of busses etc. This makes the design less portable. Handling concurrency in hardware is relatively difficult in verilog as the designer should manage all the aspects of handshaking between combinational circuits. Shared use of register and other memory resources should also

be elaborated. The behavioral specification of design in verilog often consumes multiple clock cycles. Attempts to resolve this problem results in a highly unreadable code with possible bugs. In practice,this problem is solved by separating the combinational and sequential parts of the circuit. Due to these shortcomings, the synthesis and verification of hardware in verilog is slowed down. This is a huge problem during the design of SOC.

5.3 Bluespec

Bluespec is based on atomic transactions, which increases the level of concurrency abstraction above SystemC and RTL without compromising the control over hardware design. It enables automatic synthesis of complex control logic, which is the source of many bugs. This results in highly adaptable, reusable and re-configurable designs. Control adaptive parametrization in bluespec provides flexibility, where a significantly different micro-architecture can be generated by changing the parameters in the design with the associated control structures generated automatically. Bluespec allows user defined data types and static type checking. It provides several features of the modern high level languages and all of them can be synthesized.

In recent times, several attempts have been made to move the hardware design language towards a more software like specification of the circuit behaviour. Languages like C, C++ are used to express designs as sequential programs. However, the semantic gap between the software model and the hardware results in sub optimal designs with unpredictable speed and area. Bluespec System Verilog tackles this problem by building upon the traditional hardware semantics. It exploits

advanced concepts from software only for static elaboration and static verification. It uses the standard hardware structure model of verilog such as modules, module instances, hierarchy etc. For communication between modules it uses the System verilog model of interfaces and interface instances. These added with the advanced features of the high level languages, makes designing and verification in bluespec much faster.

Design activities by BSV

- Virtual Platforms - Virtual platforms written in BSV are much faster and accurate than software virtual platform.
- Architectural Modeling - It is very difficult to decide analytically that a system component should be hardware or software. BSV gives architecture model as IP 5 blocks for exploration and validation.
- Design and Implementation - BSV enables designing of IP blocks at a much higher level of abstraction and with better maintainability.
- Verification Environments - BSV is used for writing synthesizable transactors, test benches and both system and reference models.
- Executable Specifications.

5.4 Features of BSV

BSV is feature rich Hardware Descriptor Language and at the same time uses important feature of C/C++, and Object Oriented Programming. It includes all major data types present in a High Level Language. Instead of ports as in Verilog, it uses Interfaces. Each functional component is defined as Module. Coordination and Control among different modules are done through interfaces. Methods are responsible for transferring data and control signals across different modules. Atomicity and Concurrency is achieved by Rules. Functions written in C and Verilog can also embedded in BSV.

5.4.1 Compilation of BSV code

1. A designer writes a BSV program. It may optionally include Verilog, System Verilog, VHDL and C components.
2. The BSV program is compiled into a Verilog or Bluesim specification. This step has two distinct stages:
   - Pre-elaboration - parsing and type checking.
   - Post-elaboration - code generation.
3. The compilation output is either linked into a simulation environment or processed by a synthesis tool.

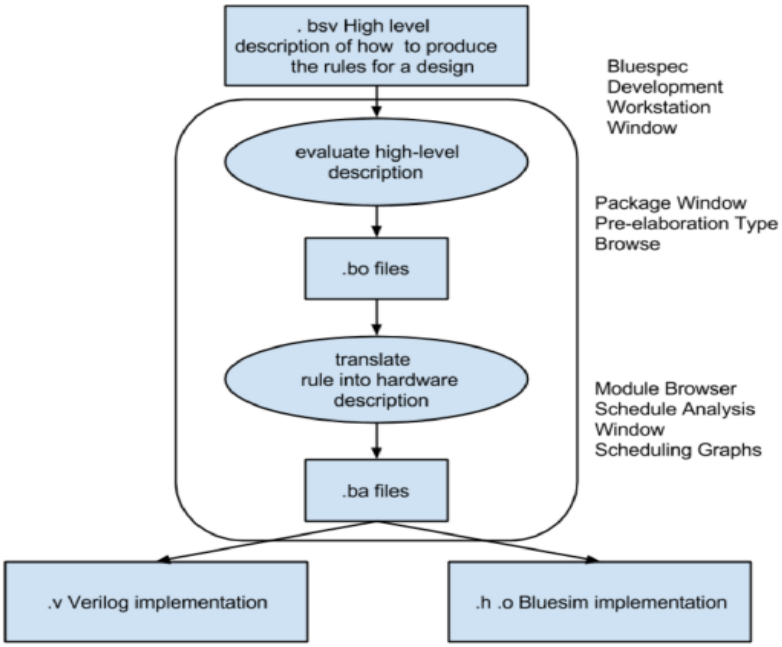


Figure 5.1: Compilation Process of BSV

5.4.2 Modules and Interfaces

Module is the basic element of the hardware design hierarchy in bluespec. A module can be instantiated multiple times, and also different parameters can be passed during every instantiation. Unlike verilog, bluespec does not have input, output and in-out pins as interface to modules. Methods are used to drive signals

and busses in and out of modules. These methods are grouped together into interfaces. Modules contain rules, which use methods in other modules.

In BSV, the interface declaration is done separately, outside the module definition. This allows declaration of common interfaces which can be used in multiple modules,without having to declare them repeatedly. All the modules which share the same interfaces also share same methods and therefore share same number and type of inputs and outputs.

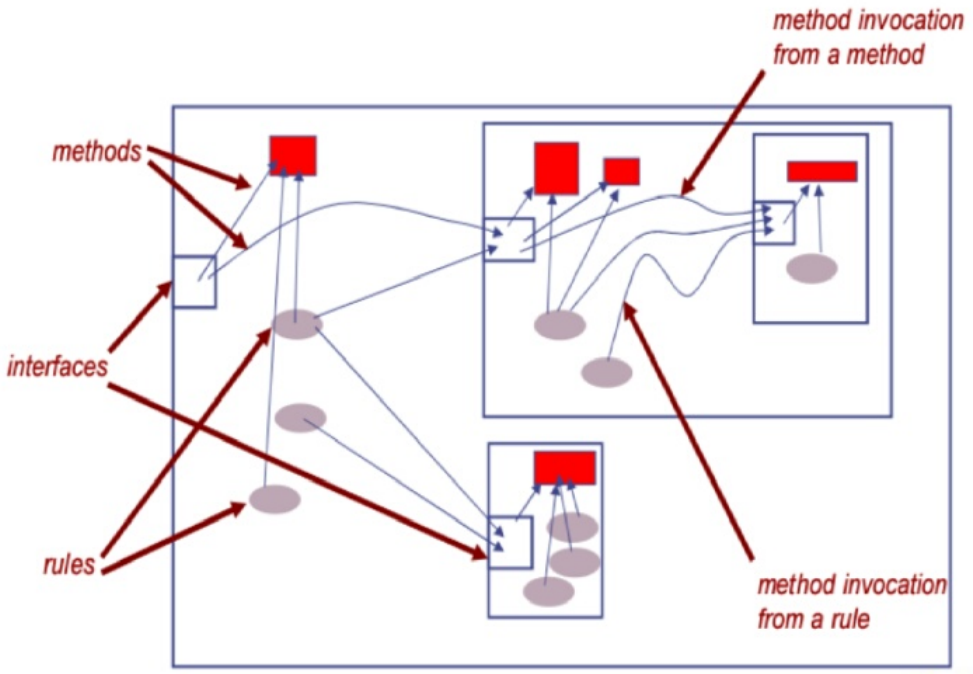


Figure 5.2: Representation of Methods, Interfaces and Rules in a module hierarchy

5.4.3 Data Types

In BSV, every variable has a type and only the values of compatible types can be assigned to a variable. The BSV compiler provides a strong, static type-checking environment. Type checking is done before the program elaboration and it ensures that the object types are compatible and the conversion functions are valid for the

context. Bluespec also allows the usage of user-defined types. BSV has a type class which can be considered as a set of types. It implements overloading across related data types. Overloading is the ability to use a common name for a collection of types, with the specific type for the variable being chosen by the compiler based on the types on which it is actually used. Functions and operators are shared by all the data types within a type class.

Some common scalar types used in Bits type class are Bit(n), Bool, UInt(n) and Int(n). The values stored in registers, FIFOs and other memory elements and also the values passed by wires, must be in the Bits type class. Other common data types include Integer, which belongs to the Arith type class and String, which belongs to the Literal type class etc.

Apart from basic data types BSV has user define data types that really helps in modeling wires of circuit.

- **Typedef** - ”typedef” is used to define new and more reliable synonym.
  typedef existingDataType NewDataType
  typedef bit[63:0] Data;
  typedef bool flag;
- **Enum** - ”enum” defines new data type to a set of scalar values with symbolic name. Due to strong type-checking enum variables can not be compared with its equivalent numeric value.
  typedef enumIdentifier1 identifier2 ... NewType
  typedef enum Reset, Count, Decision State deriving(Bits, Eq);
- **Structs**- A structure or record (struct) is a composite type made up of a collection of members. Each member has a particular type and is identified by a member, or field, name.
  typedef structIdentifier1, Identifier2, ... NewType deriving (Bits,Eq)
  typedef structint x; int y; Coord;
  typedef structAddr pc; RegFile rf; Memory mem; Proc;
- **Tagged Union** - A tagged union is a composite type made up of a collection of members. A union value only contains one member at a time while a structure value contains all of its members.

5.4.4 Rules

Rules manage the movement of data from one state to another, within the module.It consists of two parts: rule conditions and rule body. Rule conditions are boolean expressions which decide whether the rule can be fired. Rule body is a set of actions for state transitions. Rules in BSV are atomic. The actions within the rule completely describes the state transition. The process of determining the functional correctness of a design is greatly simplified by one-rule-at-a-time semantics. That is, because of the atomic property of rules, each rule can be looked at in isolation, without considering the actions of the other rules to determine functional correctness. Multiple rules can be executed concurrently in the hardware implementation.

The actions in a rule are executed simultaneously. This can be thought of as similar to the execution of non-blocking statements in always blocks of verilog. Also, as the rule has atomic property, the entire body of rule is executed and there is no partial exe- cution of a rule. When there are several rules within a module, the execution of rules is ordered by the compiler. No two rules can execute simultaneously. The ordering of the rules by the compiler is called scheduling.

BSV does not have always blocks like Verilog. Rules execute as logically ”instantaneous”, ”complete”, and ”ordered” with respect to the execution of all other rules. By ”instantaneous” it means that all the actions in a rule body occur at a single common instant, there is no sequencing of actions within a rule. By ”Complete” it means, when it is fired, the entire rule body executes; there is no concept of ”Partial” execution of a rule body. By ”ordered” it means that each rule execution conceptually occurs either before or after every other rule execution, never simultaneously[4].

Rules are made up of two components.

- **Rule Condition** - A boolean expression which determines, if the rule body is allowed to execute.
- **Rule Body** - A set of actions which describe the state updates that occur when the rule fires.

5.4.5 Methods

Method is a procedure which takes arguments and returns a value. It could also return a value without taking any arguments. It becomes a bundle of wires when translated into RTL. The method definition is written within the definition of the interface and it can be different in different modules sharing a common interface. A method also contains implicit conditions which are handshaking signals and logic automatically generated by the compiler. They do not alter any state within the module. Action methods cause actions to occur. They create state changes within the module. Action value methods are a combination of value methods and action methods. They cause state changes and also return values.

Methods are carrier for sending requests and getting responses across Modules. Methods carry data and commands on wires. Methods are different from functions because they are associated with some interface, whereas functions are independent. Methods are of following types:

- Action Method
  method Action MethodName(parameter1, parameter2, ...)
- Action Value Method
  method ActionValue (returnType) MethodName(parameter1, parameter2, ...)
- Value Method
  method Value(returnType) MethodName(parameter1, parameter2, ...)

CHAPTER 6

Operations on Posits

Once the standard has been explained it is time to start with the implementation of the arithmetic operations using the posit representation. In this chapter we deal with different arithmetic operations such as Addition, Subtraction, Multiplication, Division and Square root operations and theri corresponding algorithms.

6.1 Addition/Subtraction

Addition is the fastest of the fundamental arithmetic operations, but also surprisingly complex. The change from fixed to floating-point turns the simple addition into a 10 step process:

1. Convert to internal representation
2. Find exponent difference
3. Allign significands
4. Add/Subtract significands
5. Round
6. Detect leading one
7. Normalize result
8. Adjust exponent
9. Convert to Posit format

First the input posits are decoded and exponents and mantissa bits are seperated. The exponents are compared and the the absolute difference between them is found. The mantissa with the lower exponent is aligned so that they have same exponents. The result exponent will be $max(Exp_1, Exp_2)$. Then the two mantissas are added or subtracted depending on the effective addition/subtraction operation.

After the addition/subtraction the significand may not fit in the format anymore as it may be a digit too large, or it may be unnormalized. A Leading One Detector(LOD) finds the position of first non-zero digit and a shifter normalizes the result to a format obeying the input format. The exponent is updated with the shift. After normalization inexact results must be rounded according to the current rounding modes. As the rounding may overflow the significand has to be normalized again and the exponent must be updated accordingly. Finally the internal format must be converted to posit format.

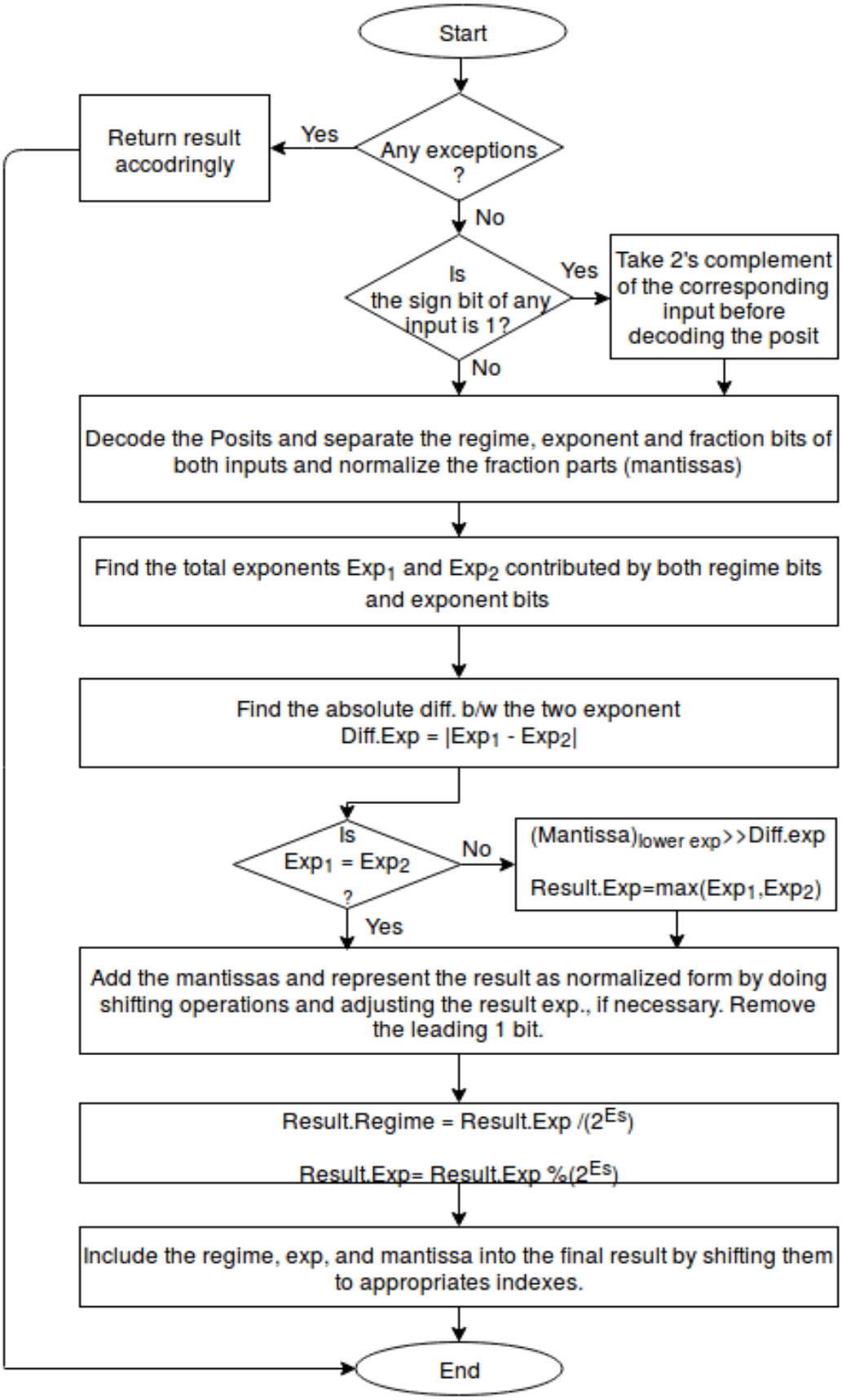


Figure 6.1: Steps involved in implementing addition operation

6.2 Multiplication

Floating-point multiplication is much simpler in concept compared to addition, but it is also slower. In total it is a 7 step process where step 2 to 4 conceptually differ from addition:

1. Convert to internal representation
2. Multiply significands and add exponents in parallel
3. Detect leading one
4. Normalize result
5. Round
6. Adjust exponent
7. Convert to posit format

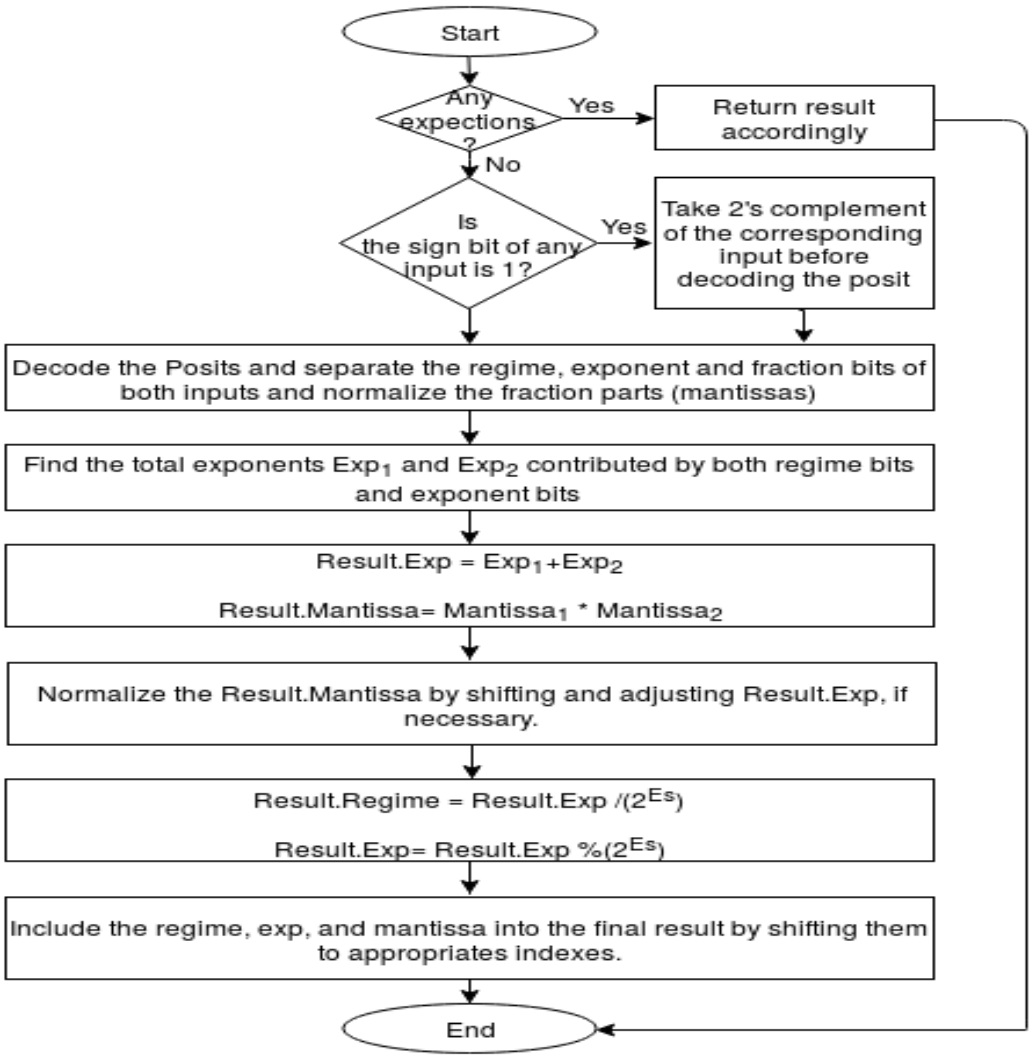


Figure 6.2: Steps involved in implementing multiplication operation

After decoding the input posits, the exponents are added and is assigned as the result exponent. The product of the mantissas is found and then leading zero is detected from the product for normalizing. The product is normalized by shifting and adjusting the result mantissa. Rounding the result of the mantissas product is done if necessary according to the round rules followed. The regime bits and the exponent bits are deducted from the result exponent. The leading 1 one from the mantissas product is removed before converting into posit format.

6.3 Division

Like floating-point subtraction is similar to addition, division is similar to multiplication. Conceptually only the second step is different.

1. Convert to internal representation
2. Divide significands and subtract the exponents in parallel
3. Detect leading one
4. Normalize result
5. Round
6. Adjust exponent
7. Convert to posit format

But the algorithms for division of the significands is not similar to multiplication at all. Where a fixed-point multiplier consists of three simple often combinatorial steps, fixed-point division is a sequencial operation with many possible implementation. There are three common algorithms which are called SRT, Newton-Raphson reciprocal approximation and multiplicative normalization.

6.3.1 Division Alogrithm

Step 1: Set the value of Divisor in register Divisor

Step 2: Set the value of Dividend in register Dividend

Step 3: Set the value of register Quotient as 0

Step 4: Intialize a new register Ext.Dividend with a size of doble the number of bits in Divisor and set it to 0.

Step 5: Add Dividend to Ext.Dividend (zero extended version of dividend)

Step 6: Repeat this step for ”n” number of times (n is no. of bits in divisor)

If (Ext.Dividend ≥ Divisor)

$Ext.Dividend = Ext.Dividend - Divisor$

$Quotient = (Quotient << 1)|1$

Else

$Ext.Dividend = Ext.Dividend << 1$

If Ext.Dividend is still less than Divisor, repeat the follwoing steps until $Ext.Dividend \geq Divisor$

$Ext.Dividend = Ext.Dividend << 1$

$Quotient = (Quotient << 1)$

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

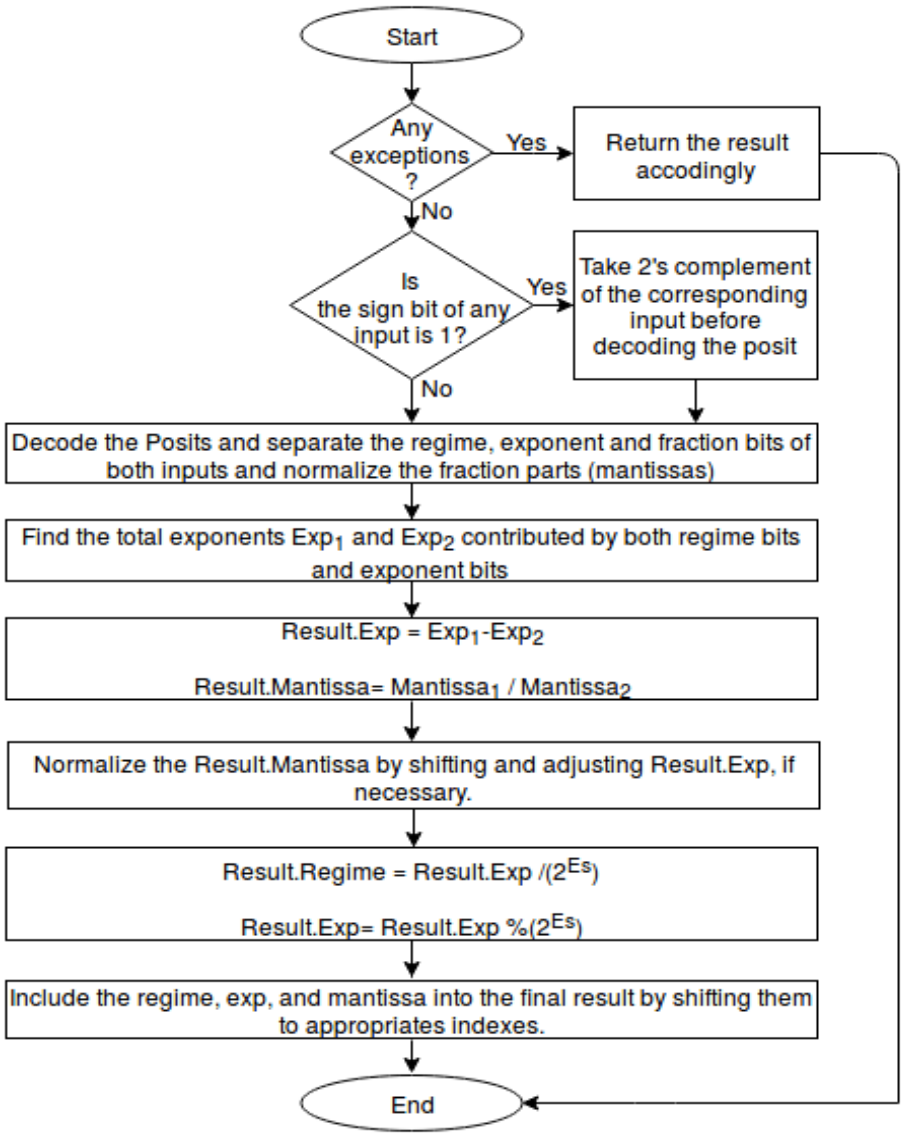


Figure 6.3: Steps involved in implementing division operation

6.4 Square Root

The non-restoring square root determination algorithm focuses on the ”partial remainder” with every iteration and not on ”each bit of the square root” [11]. At each iteration, this algorithm requires only one traditional adder or subtractor, i.e., it does not require other hardware components, such as multipliers, or even multiplexors. It generates the correct result even for the last bit position. Based on the result of the last bit, a precise remainder is obtained immediately without any addition or correction operation. It can be implemented at very fast clock rate as it has very simple operations at each iteration [12]. The alogorithm is as follows:

Initial conditions:

- Set value of register Remainder as value 0
- Set the value of register Quotient as value 0
- Set the register D as the value of the number whose square root is to be obtained.

Do the following for ”n” times (n is half the number of bits in D)

Step 1: If the value of register Remainder is greater than or equal to 0, do

Set the value of register Remainder as $(Remainder << 2)|((D >> (2 * i))\&3)$

Then set the value of register Remainder as $Remainder((Quotient << 2)|1)$

Step 2: Else do

Set the value of register Remainder as $(Remainder << 2)|((D >> (2 * i))\&3)$

Then set the value of register Remainder as $Remainder+((Quotient << 2)|3)$

Step 3: If the value of register Remainder is greater than or equal to 0 then do

Set the value of Quotient as $((Quotient << 1)|1)$

Step 4: Else do

Set the value of Quotient as $((Quotient << 1)|0)$

Step 5: If the value of register Remainder is less than 0 then do,

Set the value of register Remainder as $Remainder + ((Quotient << 1)|1)$

Finally the value of square root is obtained from the register Q and the value of remainder is obtained from the register Remainder. The algorithm is generating a correct bit of result in each iteration including the last one. For each iteration addition or subtraction is based on the sign of the result obtained from previous iteration. The partial remainder is generated in each iteration which is used in the successive iteration even if it is negative (satisfying the meaning of non-restoring our new algorithm). In the last iteration, if the partial remainder is positive, it will become the final remainder. Otherwise, we can get the final remainder by addition to the partial remainder.

CHAPTER 7

Simulations and Synthesis Results

The arithmetic operations are implemented in Bluespec System Verilog (BSV) Hardware Description Language(HDL) and the complete functionality was verified by writing a number of testcases.

7.1 Verification Setup

The Bluespec code that is written in .bsv format is given to the Bluespec compiler in the Bluespec Development Workstation (BSW). The Bluespec Compiler compiles and generates the Verilog code in .v format. The Verilog code is synthesized in Xilinx Vivado to get the clock frequency and to know about the amount of hardware generated by the design. Figure 7.1 shows the verification steps in Bluespec.

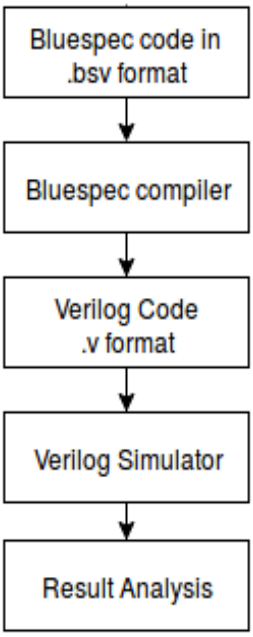


Figure 7.1: Verification Steps in Bluespec

The hardware utilization reports and the simulation results for sample test cases for different operations are shown in the following sections.

7.2 Simulation and Synthesis Results of Addition/-Subtraction

7.2.1 Addition

Resource	Utilization	Available	Utilization %
LUT	3274	63400	5.16
FF	100	126800	0.08
DSP	9	240	3.75
IO	101	210	48.10
BUFG	1	32	3.13

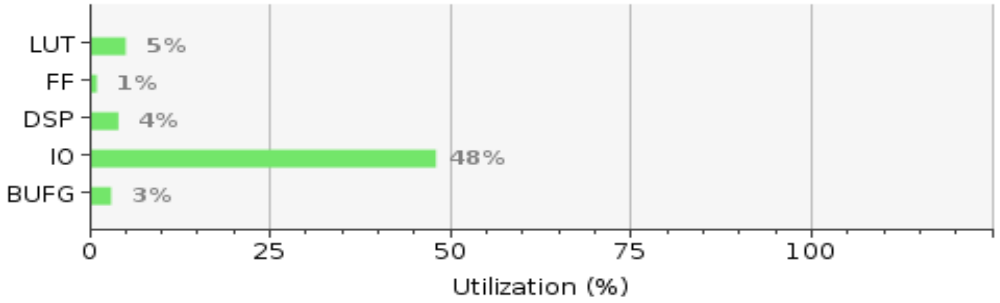


Figure 7.2: Hardware utilization report of Addition on Nexys 4 DDR board

Sample test case1:

$Input_a$: $01110100011110000000000000000000 = (74780000)_{16} = (1540096)_{10}$

$Input_b$: $01100111010100000000000000000000 = (67500000)_{16} = (3392)_{10}$

Sum : $01110100011110001101010000000000 = (7478D400)_{16} = (1543488)_{10}$

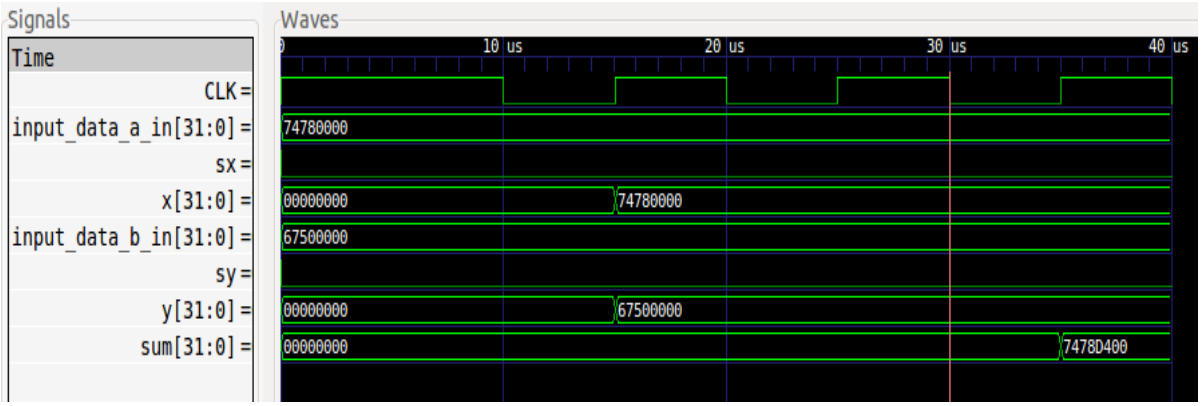


Figure 7.3: Simulation result of addition operation for a sample test-case:1

Sample test case2:

$Input_a$: $10010011010101000000000000000000 = (93540000)_{16} = (-21888)_{10}$

$Input_b$: $01100111010100000000000000000000 = (67500000)_{16} = (3392)_{10}$

Sum : $10010011101111100000000000000000 = (93BE0000)_{16} = (-18496)_{10}$

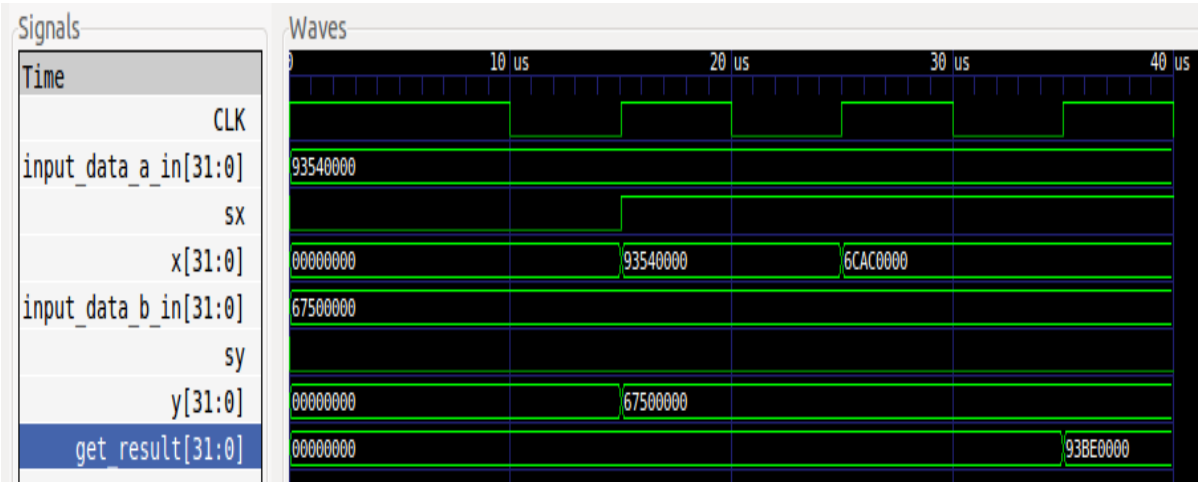


Figure 7.4: Simulation result of addition operation for a sample test-case:2

7.2.2 Subtraction

Resource	Utilization	Available	Utilization %
LUT	3222	63400	5.08
FF	100	126800	0.08
DSP	9	240	3.75
IO	101	210	48.10
BUFG	1	32	3.13

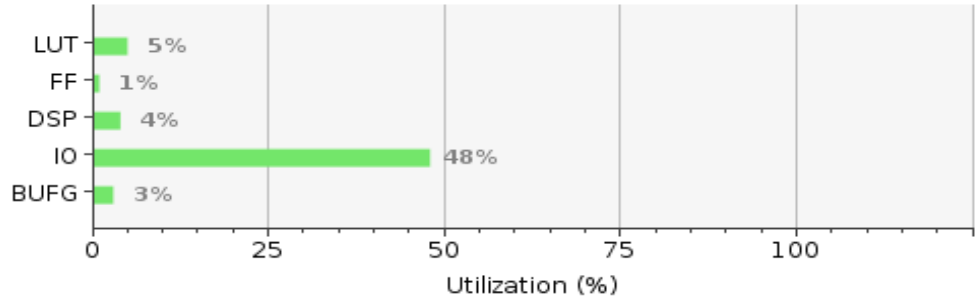


Figure 7.5: Hardware utilization report of Subtraction on Nexys 4 DDR board

Sample test case1:

$Input_a$: $01110100011110000000000000000000 = (74780000)_{16} = (1540096)_{10}$

$Input_b$: $01100111010100000000000000000000 = (67500000)_{16} = (3392)_{10}$

$Subtract$: $01110100011101110010110000000000 = (74772C00)_{16} = (1536704)_{10}$

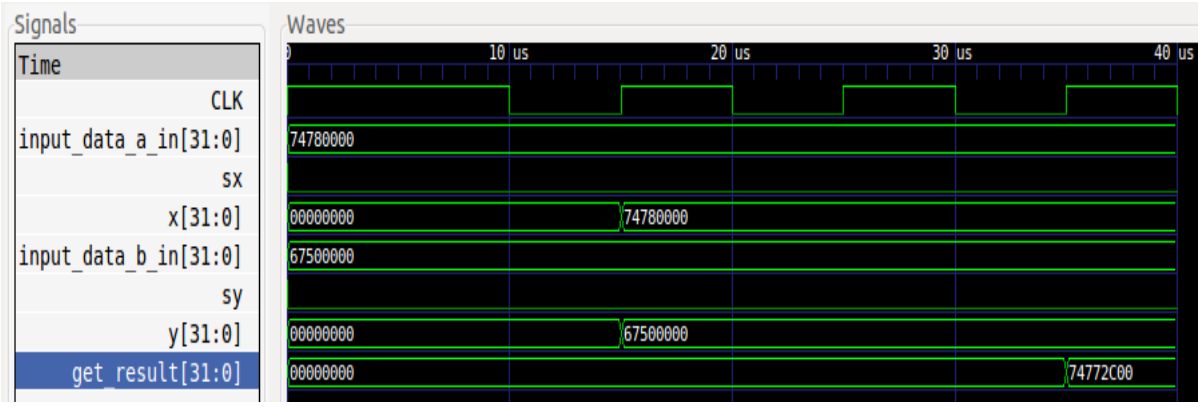


Figure 7.6: Simulation result of subtraction for sample test-case:1

Sample test case2:

$Input_a$: 00010000101010100000000000000000 = $(10AA0000)_{16}$ = $(2.032518387 \times 10^{-5})_{10}$

$Input_b$: 11110000101010100000000000000000 = $(F0550000)_{16}$ = $(-1.272559166 \times 10^{-5})_{10}$

$Subract$: 00010010001010101000000000000000 = $(122AB000)_{16}$ = $(3.305377553 \times 10^{-5})_{10}$

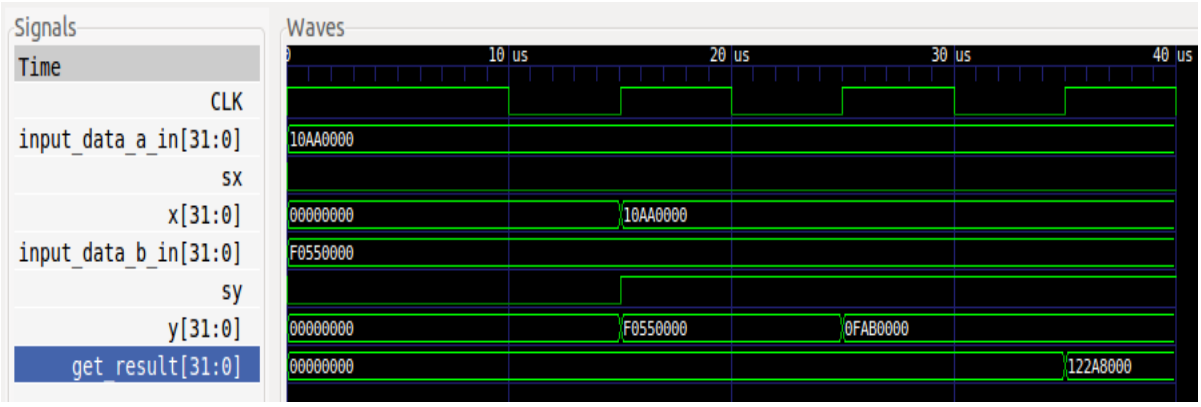


Figure 7.7: Simulation result of subtraction for a sample test-case:2

7.3 Multiplication

Resource	Utilization	Available	Utilization %
LUT	2574	63400	4.06
FF	100	126800	0.08
DSP	13	240	5.42
IO	101	210	48.10
BUFG	1	32	3.13

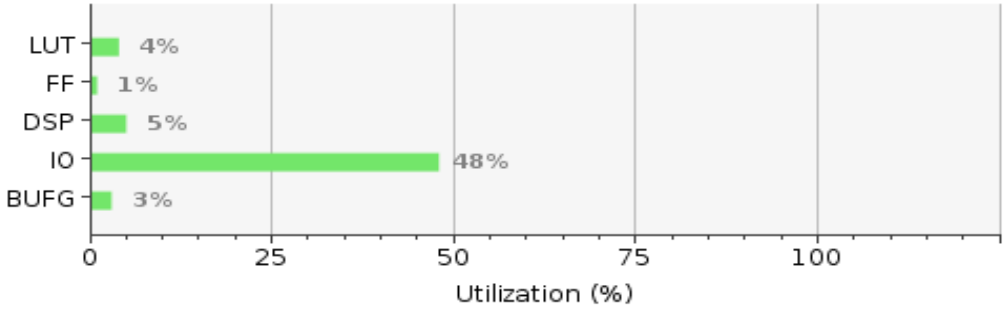


Figure 7.8: Hardware utilization report of Multiplication on Nexys 4 DDR board

Sample test case1:

$Input_a$: $01110100011110000000000000000000 = (74780000)_{16} = (1540096)_{10}$

$Input_b$: $01100111010100000000000000000000 = (67500000)_{16} = (3392)_{10}$

$Product$: $01111100000011011101100000000000 = (7C0DD800)_{16} = (5224005632)_{10}$

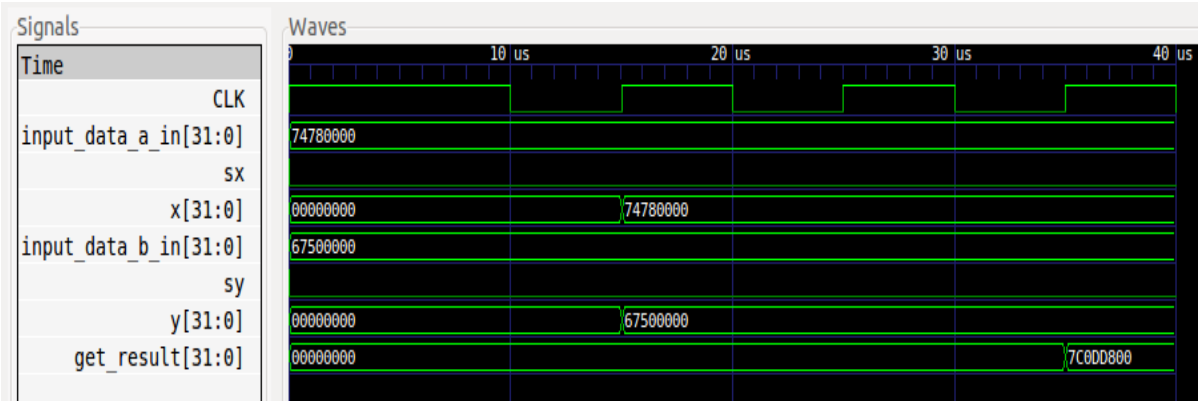


Figure 7.9: Simulation result of Multiplication for sample test-case:1

Sample test case2:

$Input_a$: $01110000010101010000000000000000 = (70550000)_{16} = (87296)_{10}$

$Input_b$: $10010000101010100000000000000000 = (90AA0000)_{16} = (-54656)_{10}$

$Product$: $10000011111110001110011100100000 = (83F8E720)_{16} = (-4771250176)_{10}$

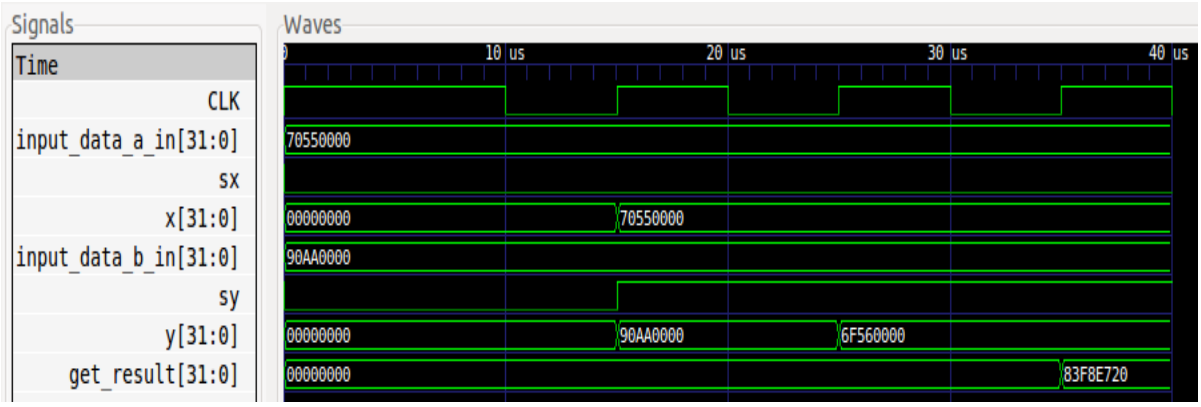


Figure 7.10: Simulation result of Multiplication for sample test-case:2

7.4 Division

Resource	Utilization	Available	Utilization %
LUT	7806	63400	12.31
FF	101	126800	0.08
DSP	9	240	3.75
IO	101	210	48.10
BUFG	1	32	3.13

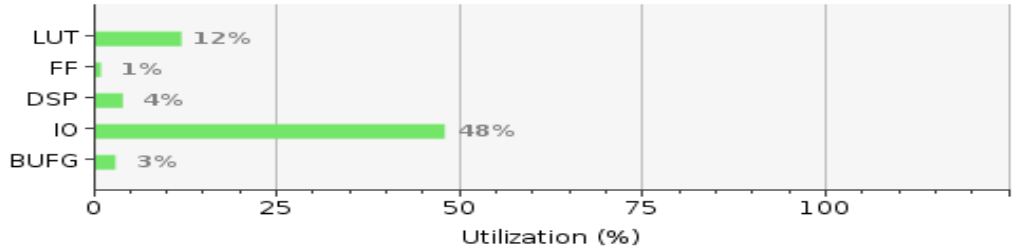


Figure 7.11: Hardware utilization report of Division on Nexys 4 DDR board

Sample test case1:

$Input_a$: $01100111010100000000000000000000 = (67500000)_{16} = (3392)_{10}$

$Input_b$: $01110100011110000000000000000000 = (74780000)_{16} = (1540096)_{10}$

$Division$: $00011110010000010101110010011000 = (1E415C98)_{16} = (2.202460077 \times 10^{-3})_{10}$

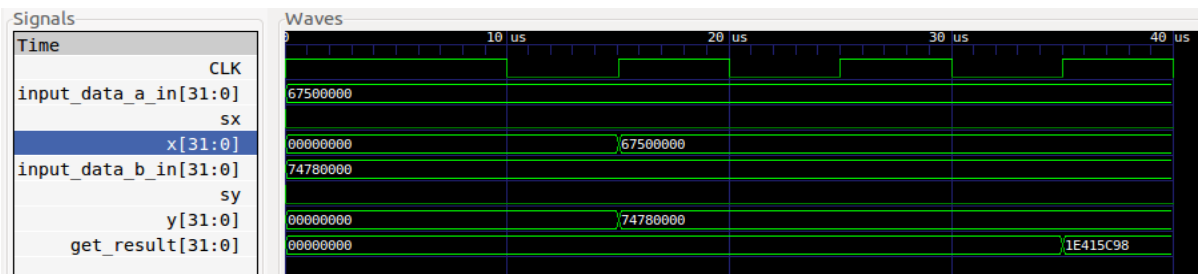


Figure 7.12: Simulation result of Division for sample test-case:1

Sample test case2:

$Input_a$: $01110000010101010000000000000000 = (70550000)_{16} = (87296)_{10}$

$Input_b$: $10010000101010100000000000000000 = (90AA0000)_{16} = (-54656)_{10}$

$Division$: $10111101100111000111101010000000 = (BD9C7A80)_{16} = (-1.59718895)_{10}$

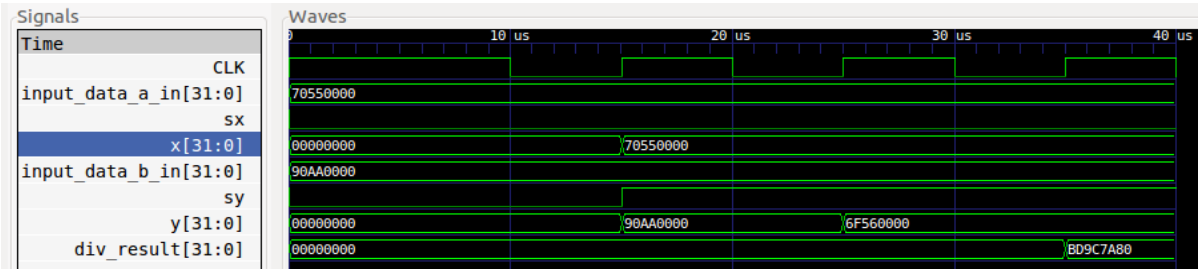


Figure 7.13: Simulation result of Division for sample test-case:2

7.5 Square Root

Resource	Utilization	Available	Utilization %
LUT	1735	63400	2.74
FF	66	126800	0.05
DSP	6	240	2.50
IO	69	210	32.86
BUFG	1	32	3.13

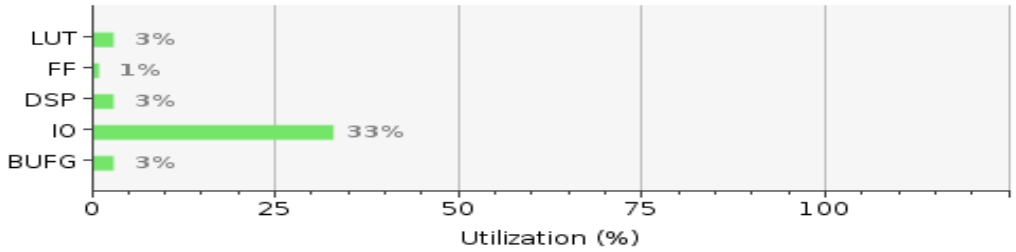


Figure 7.14: Hardware utilization report of Square Root on Nexys 4 DDR board

Sample test case1:

$Input_a$: $01100111010100000000000000000000 = (67500000)_{16} = (3392)_{10}$

$Sqrt$: $01010111010001111011000000000000 = (5747B000)_{16} = (58.24023438)_{10}$

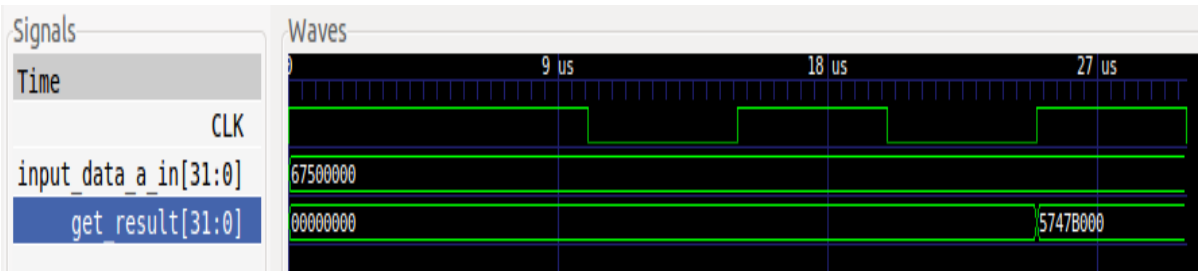


Figure 7.15: Simulation result of Square Root for sample test-case:1

Sample test case2:

$Input_a$: 00001101101010000000000000000000 = $(0DA80000)_{16}$ = $(3.159046173 \times 10^{-6})_{10}$

$Sqrt$: 00011101101000111101100000000000 = $(1DA3D800)_{16}$ = $(1.77350903 \times 10^{-3})_{10}$

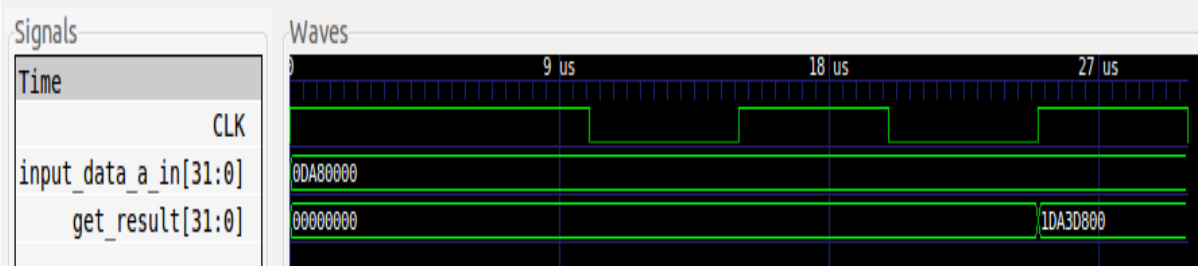


Figure 7.16: Simulation result of Square Root for sample test-case:2

7.6 Conclusions

Posits provide higher accuracy, larger dynamic range, and better closure. They can replace floats for producing better answers with the same number of bits as floats, or an eually good answer with fewer bits.

Because they work like floats, not intervals, they can be regarded as a drop-in replacement for floats, as demonstrated here. If an algorithm has survived the test of time as stable and "good enough" using floats, then it will run even better with posits. The fused operations available in posits provide a powerful way to prevent rounding error from accumulating, and in some cases may allow us to safely use 32-bit posits instead of 64-bit floats in high-performance computing. Doing so will generally increase the speed of of a calculation 2 to 4 times, and save energy and power as well as the cost of storage.

Unlike floats, posits produce bitwise-reproducible answers across computing systems, overcoming the primary failure of the IEEE 754 float definition. The simpler and more elegant design of posits compared to floats reduces the circuitry needed for fast hardware. As ubiquitous as floats are today, posits could soon prove them obsolete.

REFERENCES

[1] American National Standards Institute and Institute of Electrical and Eectronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Standard 754-1985, 1985.

[2] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic.* IEEE Standard 754-2008, August 2008. available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[3] 754-2008 - IEEE Standard for Floating-Point Arithmetic. available at https://ieeexplore.ieee.org/servlet/opac?punumber=4610933

[4] **John L Gustafson.** *The End of Error: Unum Computing, volume 24.* CRC Press, 2015.

[5] **John L Gustafson.** A radical approach to computation with real numbers. *Supercomputing Frontiers and Innovations,* 3(2):3853, 2016. available at http://dx.doi.org/10.14529/jsfi160203.

[6] **Bluespec-Inc,** *Bluespec System Verilog BSV by Examples.* Bluespec Inc, 2010.

[7] **Bluespec-Inc,** *Bluespec System Verilog Refeerence Guide.* Bluespec Inc, 2014.