STUDY OF THE POSIT NUMBER SYSTEM: A PRACTICAL APPROACH

Raúl Murillo Montero

DOBLE GRADO EN INGENIERÍA INFORMÁTICA – MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Grado en Ingeniería Informática – Matemáticas

Curso 2018-2019

Directores:

Alberto Antonio del Barrio García
Guillermo Botella Juan

Este documento está preparado para ser imprimido a doble cara.

STUDY OF THE POSIT NUMBER SYSTEM: A PRACTICAL APPROACH

Raúl Murillo Montero

DOUBLE DEGREE IN COMPUTER SCIENCE – MATHEMATICS
FACULTY OF COMPUTER SCIENCE
COMPLUTENSE UNIVERSITY OF MADRID

Bachelor's Degree Final Project in Computer Science – Mathematics

Academic year 2018-2019

Directors:

Alberto Antonio del Barrio García
Guillermo Botella Juan

To me, mathematics,
computer science,
and the arts
are insanely related.
They're all creative expressions.

Sebastian Thrun

Acknowledgements

I have to start by thanking Alberto Antonio del Barrio García and Guillermo Botella Juan, the directors of this amazing project. Thank you for this opportunity, for your support, your help and for trusting me. Without you any of this would have ever been possible.

Thanks to my family, who has always supported me and taught me that I am only limited by the limits I choose to put on myself.

Finally, thanks to all of you who have accompanied me on this 5 year journey. Sharing classes, exams, comings and goings between the two faculties with you has been a wonderful experience I will always remember.

Abstract

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) has been for decades the standard for floating-point arithmetic and is implemented in a vast majority of modern computer systems. Recently, a new number representation format called *posit* (Type III unum) introduced by John L. Gustafson – who claims this new format can provide higher accuracy using equal or less number of bits and simpler hardware than current standard – is proposed as an alternative to the now omnipresent IEEE 754 arithmetic.

In this Bachelor dissertation, the novel posit number format, its characteristics and properties – presented in literature – are analyzed and compared with the standard for floating-point numbers (floats). Based on the literature assertions, we focus on determining whether posits would be a good "drop-in replacement" for floats. With the help of Wolfram Mathematica and Python, different environments are created to compare the performance of IEEE 754 floating-point standard with Type III unum: posits. In order to get a more practical approach, first, we propose different numerical problems to compare the accuracy of both formats, including algebraic problems and numerical methods. Then, we focus on the possible use of posits in Deep Learning problems, such as training artificial Neural Networks or preforming low-precision inference on Convolutional Neural Networks. To conclude this work, we propose a low-level design for posit arithmetic multiplier using the FloPoCo tool to generate synthesizable VHDL code.

Keywords

Computer arithmetic, Floating point, Posit number system, Numerical error, Neural networks, Multiplier.

Resumen

El estándar del IEEE para aritmética en coma flotante (IEEE 754) ha sido durante décadas el formato estándar para la aritmética en coma flotante y está implementado en la gran mayoría de los sistemas informáticos modernos. Recientemente un nuevo formato de representación numérica llamado *posit* (Type III unum) presentado por John L. Gustafson (quien afirma que este nuevo formato puede proporcionar una mayor precisión empleando una cantidad igual o menor de bits y un hardware más sencillo que el actual estándar) ha sido propuesto como alternativa al actualmente omnipresente estándar IEEE 754.

En este Trabajo de Fin de Grado el nuevo formato numérico posit, sus características y propiedades (presentadas en la literatura) son analizados y comparados con el estándar para números en coma flotante. Basándonos en las afirmaciones presentes en la literatura, nos centramos en determinar si los posits son un buen reemplazo directo para los números en coma flotante. Con la ayuda de Wolfram Mathematica y Python, diferentes entornos de simulación son creados para comparar el desempeño del estándar IEEE 754 con los posit. Con el objetivo de conseguir un enfoque más práctico, en primer lugar se proponen diversos problemas numéricos para así comparar la precisión de ambos formatos, incluyendo tanto problemas algebraicos como métodos numéricos. Luego nos centramos en la posible utilización de los posits en problemas de aprendizaje profundo, como el entrenamiento de redes neuronales artificiales o la realización de inferencia de baja precisión en redes neuronales convolucionales. Para finalizar este trabajo, presentamos un diseño de bajo nivel para un multiplicador en formato posit empleando la herramienta FloPoCo para generar código VHDL sintetizable.

Palabras clave

Aritmética computacional, Coma flotante, Sistema numérico posit, Error numérico, Redes neuronales, Multiplicador.

Contents

List of Figures

List of Tables

List of Acronyms

CNN Convolutional Neural Network 43

DL Deep Learning 3

DNN Deep Neural Network 43

FMA fused multiply–add 2

FPGA Field-Programmable Gate Array 53

GPU Graphics Processing Unit 18

HPC high-performance computing 1

LNS logarithmic number system 49

LSB least-significant bit 53

LUT lookup table 60

MAC multiplier–accumulator 16

MSB most-significant bit 18

MSE mean squared error 45

NaN Not-A-Number 2

NN Neural Network 18

SF scaling factor 54

ULP Unit in the Last Place 30

Chapter 1

Introduction

1.1. Motivation

We live in an era where the trade-off between cost, performance and energy consumption is crucial in the area of computer science. The current demand of analyzing huge amounts of data, the high-performance computing (HPC), or the limited computing resources available on the more and more frequent embedded systems are developing new computer paradigms. Over the years, some famous disasters caused by floating-point numerical errors have occurred, such as the Patriot Missile failure (February 25, 1991), the change of parliament makeup on the German elections (April 5, 1992) or the explosion of the Ariane 5 (June 4, 1996). These fatal errors were produced by issues of the floating-point format design, such as overflow or rounding error problems [1].

Multiple floating-point representations have been used in computers over the years, although the IEEE Standard for Floating-Point Arithmetic (IEEE 754) [2] is the most common implementation that modern computing systems have adopted. Since it was established in 1985, the standard has only been revisited in 2008 (IEEE 754-2008) [3], but it remains the main characteristics of the original to keep compatibility with existing implementations and it is not adopted by all computer systems. However, multiple shortcomings have been identified in the IEEE 754 standard, which are listed below [4]:

- Different computers using the same IEEE floating-point format are not required pro-

duce the same results. When a computation does not fit into the chosen number representation, the number will be rounded. Even in the last revision of the standard they introduce the *round-to-nearest, ties away from zero* rounding scheme and provide recommendations for computations reproducibility, hardware designers are not coerced to implement them. Therefore, identical computations can lead to multiple results across different computing platforms [5].

- Multiple bit patterns are used for handling exceptions such as the Not-A-Number (NaN) value, which indicates that a value is not representable or undefined – for example dividing by zero results in a NaN. The problem is that the amount of bit patterns that represent NaNs may be more than necessary, making hardware design more complex and decreasing the available number of exactly representable values.

- IEEE 754 makes use of overflow – accepting ∞ or $-\infty$ as a substitute for large-magnitude finite numbers – and underflow – accepting 0 as a substitute for small-magnitude nonzero numbers. Thus, major problems can be produced, as the above mentioned.

- Rounding is performed on individual operands of every calculation, so associativity and distributivity properties are not always held in floating-point representation. The last revision of the standard tries to solve this issue including the fused multiply–add (FMA) operation. However, again this may not be supported by all computer systems.

The above listed shortcomings led to the idea of developing a new number system that can serve as a replacement for the now ubiquitous IEEE 754 arithmetic. At the beginning of 2017, John L. Gustafson introduced the *posit* number representation system, a format that has no underflow, overflow or wasted NaN values. Gustafson claims that posits are not only a suitable replace for the current IEEE Standard for Floating-Point Arithmetic, but also provide more accurate answers with an equal or smaller number of bits and simpler hardware [6].

1.2. Objectives

The primary purpose of this dissertation is to determine whether the posit number format might be a suitable replacement for the current IEEE 754 floating-point number format. As this question may be too broad, we will focus our efforts into the following goals:

- To understand the posit number format.
- To determine the differences between Type III unums (posits) and IEEE Standard for Floating-Point Arithmetic in both theoretical and practical contexts.
- To explore the use of posits in the area of Deep Learning and compare it to the IEEE 754 floating-point standard in terms of accuracy and performance.
- To design a posit operator using reconfigurable logic to compare the hardware design with the equivalent for IEEE 754 arithmetic.

1.3. Document Structure

This document is a faithful reflection of the research process carried out from the very beginning. The rest of the document is structured as follows. In Chapter 2 the *unum* number format and in particular posits are widely explained. Then in Chapter 3, both posit and the standard floating-point formats are compared in terms of accuracy. Next, in Chapter 4 we propose the use posits for Deep Learning (DL). Afterwards, in Chapter 5, the state of art of the posit hardware design is presented, and we introduce our first posit synthesizable component. Finally, in Chapter 6 the conclusions and the future work are shown. To complement the reading of the document, the main concepts of IEEE 754 standard are explained in Appendix A.

Chapter 2

The Unum Number Format

The universal number (*unum*) format is an arithmetic format similar to floating point that is gaining interest as an alternative to the IEEE 754 arithmetic standard. This chapter describes in detail the posit number format (Type III unum), a possible replacement for floats proposed by Gustafson [6] in 2017. To better understand posits, it is also useful to have prior knowledge of Type I and II unums. That is why they are described first, after which posits are explained. Finally, we present some properties and advantages of this new number format.

2.1. Type I and Type II Unums

The concept of *unum* was proposed by John L. Gustafson in [4] as an alternative to the IEEE Standard for Floating-Point Arithmetic that has been the standard for decades (IEEE 754). The unum number format is used for expressing both real numbers and ranges of real numbers. This arithmetic framework has evolved over the last years, dividing the *unum* into three different types.

The original Type I unums are to floats what floats are to integers: A superset. They can represent either an exact float or an open interval between adjacent floats, when a computation is not able to provide the numerically exact answer and in standard floating-point arithmetic rounding should be performed. To do this, *unum* include a "ubit" (uncertainty bit) at the end of the fraction to indicate whether it corresponds to an exact value or an

interval, if the ubit is equal to 0 or 1, respectively.

The Type I unum format also takes the rest of components of the IEEE 754 floating-point scheme – the sign, exponent and fraction (or mantissa) bit fields. However, in this format the exponent and fraction field lengths are variable, from a single bit up to some maximum set by the user. Thus, the *exponent size* and *fraction size* fields are added to the *unum* scheme in order to annotate the widths of the corresponding exponent and fraction fields. This format specification can be visualized in Figure 2.1. Type I unums provide a natural way to expand floats into *interval arithmetic*, but their variable length would require special management at hardware implementation. More about the proposal and justification of this format can be found in [4].

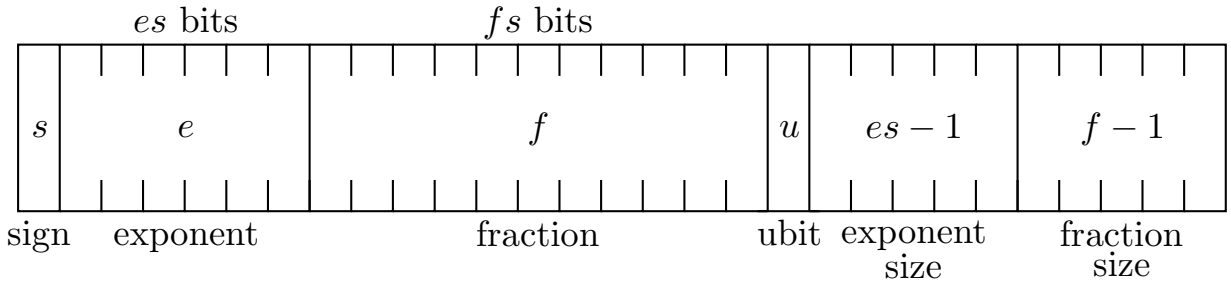


Figure 2.1: Type I unum bit fields.

The Type II unum was proposed to resolve some of the shortcomings that the first version had, such as the complexity of the hardware implementation or the fact that certain values can be represented in different ways. This second version is no more compatible with IEEE floats. Instead, Type II unums present a clean, mathematical design based on the mapping of values onto the real projective line, which is the set $\widehat{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$. The key concept is that the point where signed (two's complement) numbers change from positive to negative is the point where positive real numbers turn to negative numbers, and the same ordering, and that point represents the value $\pm\infty$. The structure of Type II unums is shown in Figure 2.2. The upper right quadrant of the circle has an ordered set of real numbers x_i, while the upper left quadrant has the negatives of those x_i, a reflection about the vertical axis.

The lower half of the circle holds the reciprocals of the numbers on the top half, a reflection about the horizontal axis. This way, given a certain value, we can get the opposite and the reciprocal values by vertical and horizontal reflections, respectively. Again, as Type I, Type II unums ending in 1 (the ubit) represent the open interval between the surrounding reals, represented by the *unums* ending in 0. As one can imagine, Type II unums have many ideal mathematical properties based on the geometry of projective real numbers, but in practice those properties rely on look-up tables for most operations, which limits the scalability of this ultra-fast format to about 20 bits or less, for current memory technology [6, 7]. Moreover, fused operations such as dot product is quite expensive in this format. These drawbacks served as motivation of a search for a new format that would keep many of the Type II unum properties, but also be more "hardware-friendly".

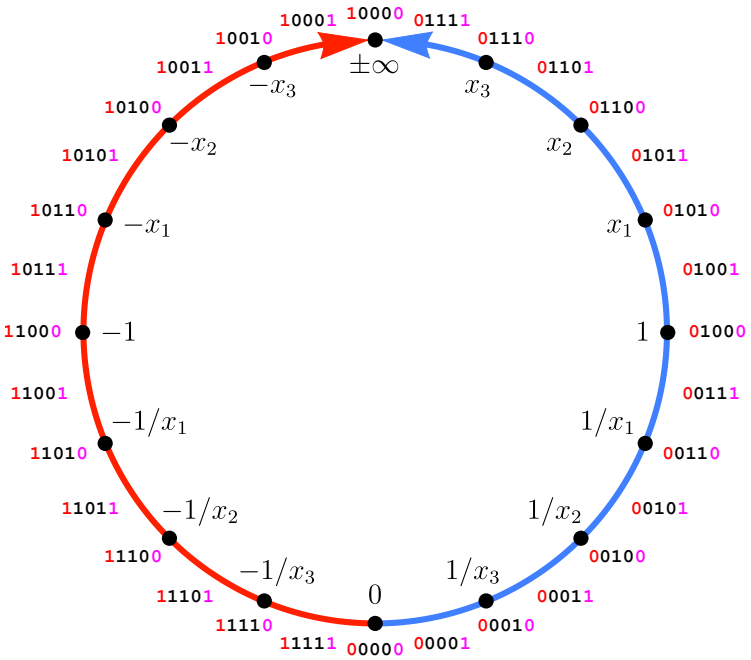


Figure 2.2: Visual representation of the projective real number line of Type II unums.

2.2. Type III Unum: Posits

The Type III unum data type (also known as *posit*) is, in words of the authors, "designed as a direct drop-in replacement for IEEE 754 standard for floating-point numbers" [6]. The idea of Type III unum is therefore, just like Type II, based on the real projective line, although the hardware implementation for this format would be similar to the existing logic used for IEEE 754 floating-point arithmetic [8]. This is archived by relaxing the perfect reflection rule to get the reciprocals – now only follows for 0, $\pm\infty$ and integer powers of 2. Thus, all the numbers are of the form $m \cdot 2^k$, where m and k are integers, and there are no open intervals.

A *valid* is the interval arithmetic version of the posit. It consists of a pair of equal-size posits, each ending in a ubit indicating the bounds. However, valids are not the focus of this dissertation – in addition, details of this format have not yet being officially presented by Gustafson.

2.2.1. The Posit Format

Compared to Type I and Type II unums, the format of this new type changed thoroughly. Figure 2.3 shows the structure of an n-bit posit with es exponent bits.

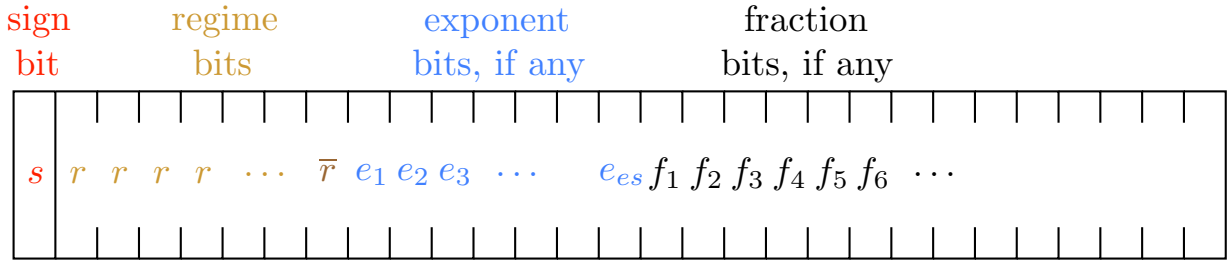


Figure 2.3: Generic posit format.

The posit format consists of the sign, regime, exponent and fraction fields.

- **Sign** The sign bit is as in signed floats or integers: 0 for positive numbers, 1 for

negative numbers. In the case of negative, the 2's complement of the remaining bits has to be taken before extracting the regime, exponent, and fraction fields.

- **Regime** This field is exclusive of this numeric format. It is used to calculate a scale factor of $useed^k$, where $useed = 2^{2^{es}}$. The value k is determined by the number of identical bits (color-coded in **amber**), terminated with an opposite bit (in **brown**), if any. Let m be the number of identical bits; if the regime field consists of leading `0`'s, then $k = -m$; if they are `1`'s, then $k = m - 1$.

- **Exponent** The exponent bits (color-coded **blue**) encode the value e, and represent the scaling factor 2^e. Unlike with floats, there is no bias. As the length of the regime is variable, there can be up to es exponent bits, as the first bit of this field is located directly after the regime field (so the possibility of no exponent bits exist).

- **Fraction** The bits remaining after the exponent correspond to the fraction field, and they represent the fraction value f. This is the same as for the IEEE floats (also called *significand* or *mantissa*), but there exist a big difference – in the case of posits the hidden bit is always `1`, and there are no subnormal numbers with a hidden bit of `0` as in the standard IEEE 754.

Therefore, the decimal value of a posit is given by

$$(-1)^{sign} \times useed^k \times 2^e \times (1 + f),$$

where:

$useed$	is the scaling factor, equal to $2^{2^{es}}$,
k	is the value of the **regime** field,
e	is the value of the **exponent** field,
f	is the value of the **fraction** field.

As can be seen, every posit configuration is completely determined by the total length of bits (n) and the maximum number of exponent bits (es). Therefore, it is common to use the notation Posit$\langle n, es\rangle$ to denote a configuration of n-bit posit with es exponent bits.

To conclude this section, we present a comparison between the three existing types of *unum* in Table 2.1 [9].

Unum	**Date Introduced**	**IEEE 754 Compatibility**	**Advantages**	**Disadvantages**
Type I	March 2015 [4]	Yes; perfect superset	Most bit-efficient rigorous-bound representation	Variable width management needed; inherits IEEE 754 disadvantages, such as redundant representations
Type II	January 2016 [10]	No; complete redesign	Maximum information per bit (can customize to a particular workload); perfect reciprocals ($+ - \times \div$ equally easy); extremely fast via ROM table lookup; allows decimal representations	Table look-up limits precision to ~ 20 bits or less; exact dot product is usually expensive and impractical
Type III	February 2017 [6]	Similar; conversion possible	Hardware-friendly; posit form is a drop-in replacement for IEEE floats (less radical change); faster, more accurate, lower cost than float	Too new to have vendor support from vendors yet; perfect reciprocals only for 2^n, 0, and $\pm\infty$

Table 2.1: Comparison between the different types of *unum*.

2.3. Posits as Projective Reals

Posits can be better understood from a geometrical point of view. As mentioned before, Type III posit arithmetic is derived from Type II unums, which are mapped onto the projective reals. However, for Type III, the requirement that all the values have a reciprocal is relaxed – the opposite values of a posit number is obtained again flipping around the vertical axis, but only for 0, $\pm\infty$ and powers of 2 flipping across the horizontal axis yields the reciprocal. Both Type II and Type III unums start with the two-bit template shown in Figure 2.4.

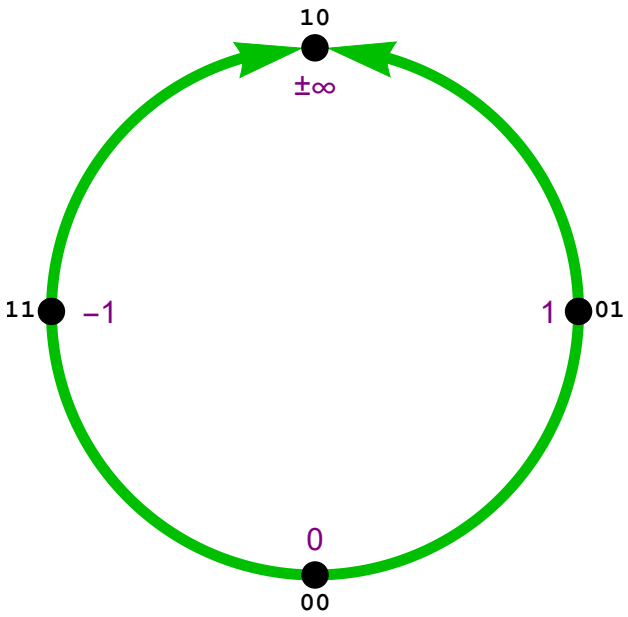


Figure 2.4: Two-bit numbers mapped over the projective reals.

The bit strings around the ring (one can consider them as 2's complement signed integers) change from positive to negative at the same point the real numbers do. This eliminates the "negative zero" that floats consider, but also the $-\infty$ and $+\infty$ are reduced into a single one.

In the above two-bit ring we can insert a value between 1 and $\pm\infty$ (and the corresponding negation and reciprocation obtained by the reflections around the vertical and horizontal axes, respectively) to get a three-bit ring (see Figure 2.5). The value to insert could be any

real number greater than one, but it must be noticed that this choice "seeds" the way the rest of the ring of *unums* gets populated (all the positive values of the ring are powers of that value), so this is why this value is called *useed*.

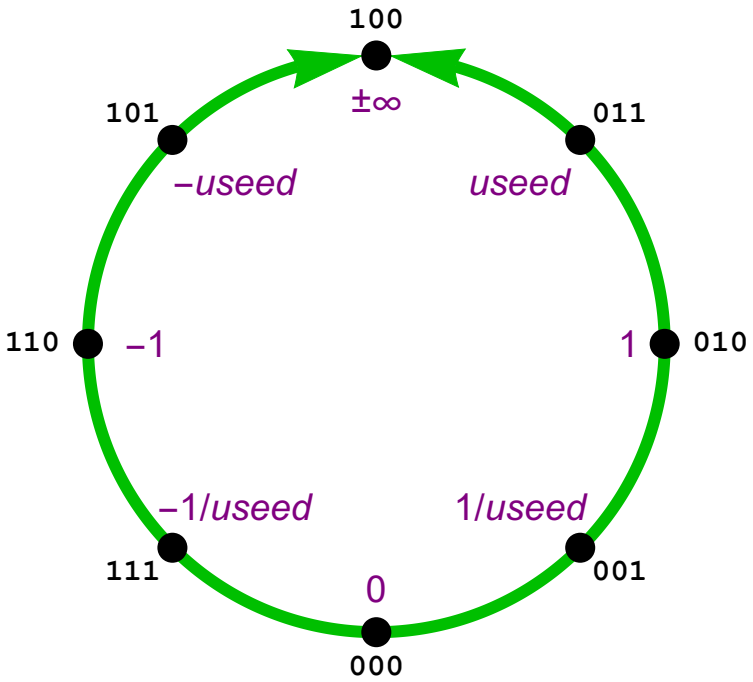


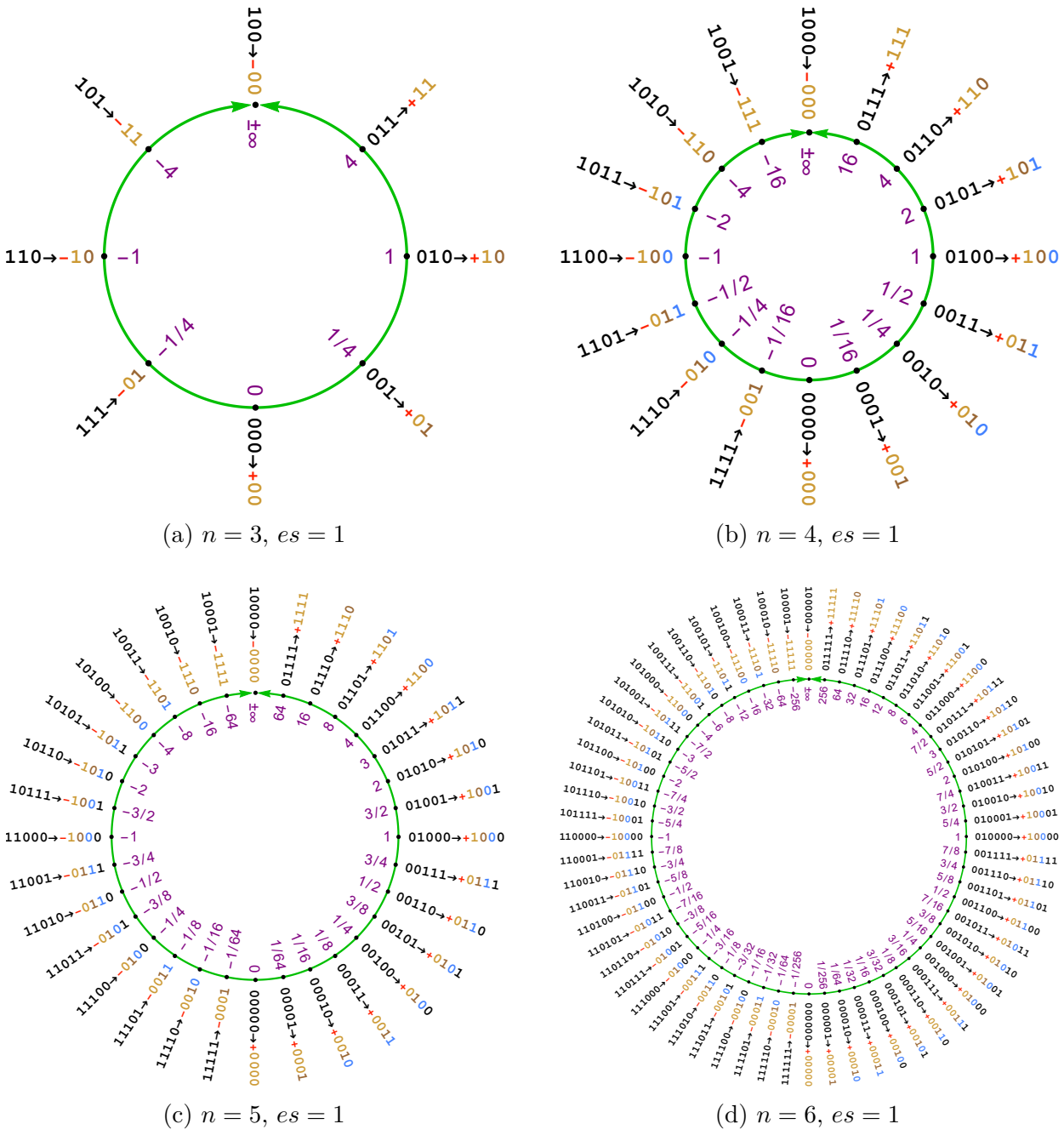
Figure 2.5: Adding the *useed* value between 1 and $\pm\infty$ results on the three-bit ring.

One may try to guess the next step, how to append bits in order to increase posit precision; there is a recursive definition for that. The values remain when a bit `0` is appended. Appending a bit `1` creates a new point in the circle between two existing values. If *maxpos* is the largest positive value and *minpos* the smallest positive one on the ring, the value assigned to the new in-between value follows these interpolation rules:

- Between *maxpos* and $\pm\infty$, the new value is $maxpos \times useed$, and between 0 and *minpos*, the new value is $minpos/useed$ (new **regime** bit).
- Between two existing values $x = 2^m$ and $y = 2^n$, where integers m and n differ by more than 1, the new value is their *geometric* mean, $\sqrt{x \cdot y} = 2^{(m+n)/2}$ (new **exponent** bit).

- Between any other adjacent points x and y, it represents the *arithmetic* mean, $(x+y)/2$ (new **fraction** bit).

Figure 2.6 shows this iterative process from a 3-bit up to a 6-bit posit with $es = 1$.



(a) $n = 3$, $es = 1$

(b) $n = 4$, $es = 1$

(c) $n = 5$, $es = 1$

(d) $n = 6$, $es = 1$

Figure 2.6: Posit construction with one exponent bit, $es = 1$, so $useed = 4$.

2.4. Properties

Posits have been presented as a more elegant format – in a mathematical sense – than the current IEEE 754 floating-point numbers. However, that is not enough to become a replacement for the standard. Therefore, below we present some properties of posits that make this format superior to floats.

2.4.1. Numeric Representation. Zero and NaN

As seen in Section 2.2.1, the way posits are coded is similar as floats are – a sign bit followed by the exponent of a scaling factor and ended with fraction bits. However, the posit codification does not allow a "negative zero" representation, which exists in IEEE floats. On the other hand, and due to the same reason, posits do not have different patterns for ∞ and $-\infty$. One may think that this is a big mistake, but as we will see below, posits do not overflow, so results involving ∞ are not common in posit arithmetic. In addition, one possible solution to this problem may be use *valids* (interval arithmetic) to express signed intervals.

But probably the most interesting property of posits in terms of numeric representation is the lack of NaN patterns. According to the posit standard [11], when finding a NaN the calculation is interrupted and the interrupt handler can be set to report the error. This obviously simplifies the hardware, since there is a maximum of two patterns (0 and ∞) to check for exceptions. The rest of bit patterns are therefore used for representing real numbers, so the amount of representable values using posit format is always higher than using floats for the same number of bits.

One more advantage of the posit format related to hardware implementation is that checking for equality ($a = b$) is as simple as checking if the bit patterns are identical – which is not true for floats since $-0 = 0$ and the corresponding bit patterns are not the same. What is more, posit numbers with same sign and their bit patterns keep a total order relation, i.e., if p_1 and p_2 are two posit numbers with the same sign such that $p_1 < p_2$,

then $bin(p_1) < bin(p_2)$, where $bin(p_i)$ indicates the binary representation of posit p_i. These two tests (equality and relation order) are, in terms of hardware, equivalent to perform the same tests for signed integers, which is obviously faster and easier than performing the same computations with floats.

2.4.2. Underflow, Overflow and Rounding

As early mentioned, the hidden bit from the fraction size in posit format is always `1`, so there are no *subnormal numbers* as in the IEEE 754 standard. Posits do not use "gradual underflow" then, but they use the so-called *tapered precision*, i.e., the values mass around 0 and sparse to higher or lower numbers in less frequency (one can check that for every posit configuration, half of the values are between -1 and 1). This way is provided a similar functionality as gradual underflow, and also a kind of symmetrical gradual overflow.

When a real number is not expressible as a posit, rounding is necessary. The rounding method for posits differs slightly from the default method used at the standard IEEE 754 (*round-to-nearest-even*, also called Banker's Rounding) at the extreme situations. Before formulating the rules that posits follow, we need to formally define two concepts that were previously introduced: *maxpos*, which is the largest real value expressible as a posit, and respectively *minpos*, which is, analogously, the smallest nonzero value expressible as a posit. These two values are reciprocal one to the other (if dealing with absolute values), and their decimal value is given by the expressions $maxpos = useed^{n-2}$ and its reciprocal $minpos = useed^{2-n}$. The rules for rounding a real value x into a posit are the following:

1. If x is exactly expressible as a posit, no rounding is needed.
2. If $|x| > maxpos$, x is rounded to $sign(x) \times maxpos$.
3. If $0 < |x| < minpos$, x is rounded to $sign(x) \times minpos$.
4. For all other values, use the *round-to-nearest-even* scheme, i.e., if two posits are equally near, take the one with binary encoding ending in 0.

Note that posits, unlike floats, do not overflow or underflow. While overflow is just a problem in computations, underflow may not; we will see below how this affect computations.

2.4.3. Fused Operations and Quire

When performing any operation the result if often rounded to fit in the original format. This may cause that computations involving multiple operations lose accuracy due to intermediate roundings. The solution to this problem would be to defer the rounding until the last operation in a computation involving more than one operation – expressions that follow this rule are called *fused operations*. The most recent version (2008) of the IEEE 754 standard [3] includes the FMA in its requirements, and many modern general-purpose processors include multiplier–accumulator (MAC) units. The posit format supports the following fused operations:

- Fused multiply-add $(a \times b) + c$
- Fused add-multiply $(a + b) \times c$
- Fused multiply-multiply-subtract $(a \times b) - (c \times d)$
- Fused sum $\sum a_i$
- Fused dot product (scalar product) $\sum a_i b_i$

Notice that all these operations can be performed from the fused dot product, which is very nice in terms of hardware requirements – all the fused operations can be performed with a MAC unit. According to the posit standard [11], fused operations are distinct from non-fused operations and must be explicitly requested in a posit-compliant programming environment.

For performing fused operations, the posit format introduces the concept of *quire*: a fixed-size scratchpad register that is used as an accumulator for intermediate computations. This register must be wide enough to avoid the need of rounding until the entire expression is

evaluated. The quire width depends therefore on the values n and es of the posit configuration; this is discussed in [9] and sizes for common formats are depicted in Table 2.2.

Posit configuration	$\langle 8,0\rangle$	$\langle 16,1\rangle$	$\langle 32,2\rangle$	$\langle 64,3\rangle$	$\langle 128,4\rangle$	$\langle 256,5\rangle$
Quire size (bits)	64	256	512	2048	8192	32768

Table 2.2: Quire size according to posit configuration.

One may claim that using scratch registers in fused operations is not a new idea. However, the concept of quire is slightly different; in contrast with the actual architectures, the quire register would be accessible by the programmer with the following instructions:

- Clear the quire
- Load the quire from memory
- Store the quire to memory
- Add the product of two posits to the quire
- Subtract the product of two posits from the quire
- Add a quire stored in memory to the quire
- Subtract a quire stored in memory from the quire
- Convert a quire into a posit

More complex instructions are not allowed to perform with the quire register since it is just an accumulator, not an extended precision register to declare variables. The programmer may assume there is only one quire register in a core. Thus, this can provide much more accuracy when instantiating data without rounding than the actual standard do. What is more, in [9] the authors ensure that quire-based operations are about 3 to 6 times faster than rounding after every operation.

2.4.4. 8-bit Posits and Sigmoid Function

While IEEE floats do not define a "quarter-precision" 8-bit float (such configuration is often called "minifloat" [12], but it is not in the standard), there are no fixed sizes in the posit format, so we can perfectly define such precision. In particular, 8-bit posit with $es = 0$ configuration, which is also called *posit8*, has proved to have some really useful properties in the area of Neural Networks (NNs). As we will show below, *posit8* has a nice addition closure near 0. But what is more surprisingly, it can approximate extremely well and easily the *sigmoid function*. This function is well-known in Machine Learning and Deep Learning. The analytic expression for sigmoid function is $f(x) = 1/(1 + e^{-x})$, which is very expensive – many clock cycles – to compute due to the exponential and the division. However, *posit8* can easily simulate this function by just flipping the most-significant bit (MSB) of the posit representation and shifting it two positions to the right, adding `0` bits on the left. Note that this requires only one clock cycle. The result of such transformation, compared with the original function, is shown in Figure 2.7. As can be seen, the highest errors are made outside the critical region of this function – near $x = 0$ – while the slope intersecting the y-axis is correctly approximated.

Currently, half-precision (16-bit) IEEE floats are used by many Machine Learning frameworks such as TensorFlow [13] to perform Graphics Processing Units (GPUs) computations. However, using quarter-precision (8-bit) posits can be 2–4 times faster with a reduced memory footprint and power-consumption.

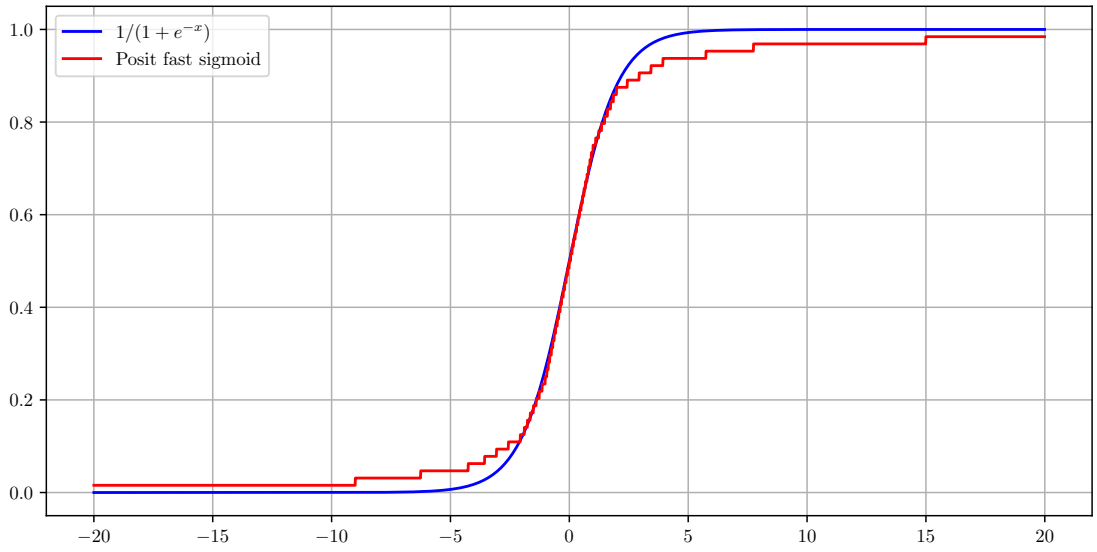


Figure 2.7: Sigmoid function approximation using Posit$\langle 8, 0\rangle$.

Chapter 3

Posits vs Floats: Metric Study

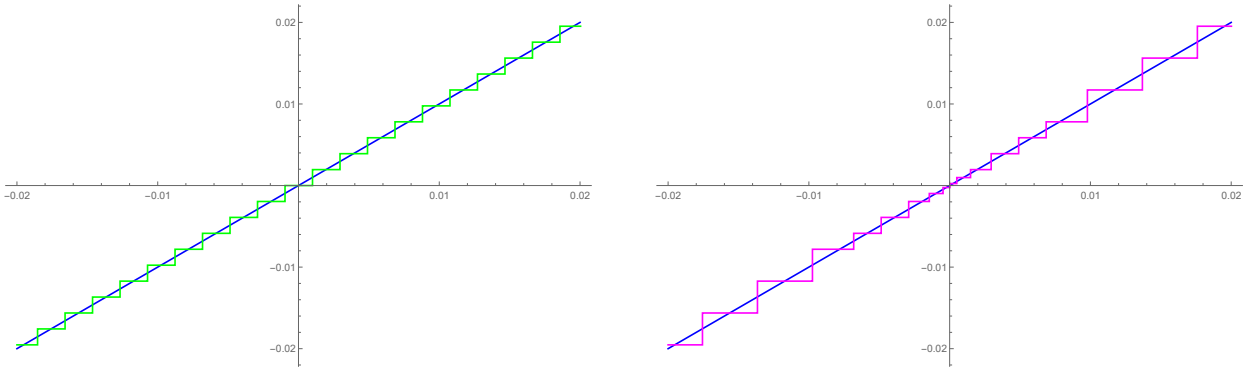
This chapter focuses on how computations based on posit arithmetic are performed, and how accurate are – in terms of numerical error – compared to the different precision formats of the IEEE Standard for Floating-Point Arithmetic. Therefore, we reproduce some of the experiments performed in [6, 9, 14].

Firstly, we present a comparison of the basic operations for both formats. As an exhaustive test is needed in this kind of experiments, all possible values are taken. For this reason, we choose 8-bit length strings of both formats, since a larger amount of bits would take extremely large computation times. Therefore, we compare 8-bit posits with 1 bit of exponent (Posit$\langle 8, 1\rangle$) and 8-bit floats with 4 bits for the exponent (also known as "minifloat" [12]). After this, we perform higher precision problems and compare results obtained from both formats with the analytic solution.

The results are obtained with the help of Wolfram Mathematica – which allows both HPC and symbolic computation – and Python language – which has different libraries for the posit format such as SoftPosit [15] and PySigmoid [16].

3.1. Behavior Around 0

To better understand how computations on both formats are done, let us focus on the rounding error around the critical value 0. To show this, the line $f(x) = x$ is plotted. Then, values of the line are estimated using floats and posits. The result is shown in Figure 3.1.



(a) Floats behavior around 0.

(b) Posits behavior around 0.

Figure 3.1: Estimates of the real values.

As we can appreciate in 3.1a, floats present a regular step shape, which means that the distance between values keeps constant in the surroundings of 0. In addition, it can be shown how values close enough to 0 are rounded to it, so underflow is produced in these situations. On the other hand, posits do not underflow, only 0 value is taken when representing exactly that number, as Figure 3.1b shows. What is more, the step shape is not regular in this case, but the distance between numbers become smaller while getting closer to 0, which means that the relative error of posits is smaller.

In the neighborhood of 0, 8-bit posits work better than floats, since the distance between the representable values gets smaller and closer to 0, so the relative error remains small. Furthermore, from a mathematical point of view, it is very nice property that no rounding is made to 0.

3.2. Single-Argument Operation Comparisons

The purpose of the following computations is to compare the *closure* of some basic unary operations. In this context, the closure of an operation consists on the elements of a set that are exactly representable as a member of the set after performing the operation, i.e., there is no rounding needed. It is also relevant to know what occurs to the values that are not in the closure, what is the result after performing the operation.

In order simulating basic operations of floats and posits and compare them with the exact values, we create test environments of both formats with the help of Wolfram Mathematica [9].

3.2.1. Reciprocal

The percentage of cases where $1/x$ is exactly representable as a float or a posit is shown in Figure 3.2. It also shows how the non-exact values are distributed as finite but inexact (so a rounding would be needed), NaN, underflows and overflows.

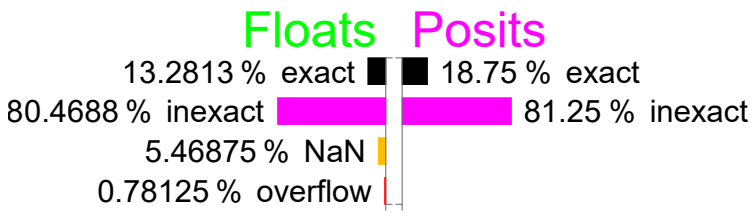


Figure 3.2: Quantitative comparison of floats and posits computing the reciprocal, $1/x$.

As can be seen, a higher percentage of posits has exact reciprocals. But not only exactitude is important – as mentioned before, posits never underflow or overflow, but floats do, in this case due to subnormal numbers. Finally, the float NaN values produce NaN outputs. In this case, posits perform reciprocation better than floats.

3.2.2. Square Root

The obtained results (see Figure 3.3) show that the square root function does not produce underflows or overflows. However, the negative values generate a great amount of NaNs.

3.2.3. Square

In the case of squaring, overflow and underflow are more common than in the previous operation. As Figure 3.4 shows, at almost half of the cases the result produced by floats is useless for computations, while all the posits can be squared.

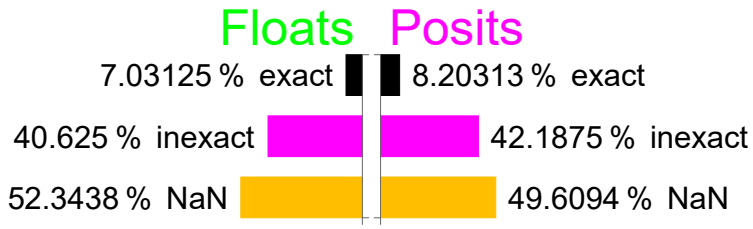


Figure 3.3: Quantitative comparison of floats and posits computing $\sqrt{x}$.

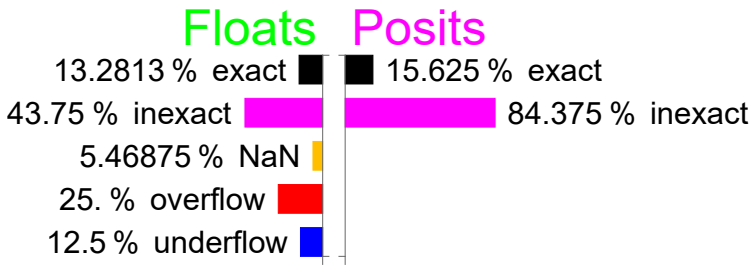


Figure 3.4: Quantitative comparison of floats and posits computing x^2.

3.2.4. Logarithm Base 2

For the closure of operation $\log_2(x)$, a similar result as with the square roots is obtained – about half of the values produce a NaN since the logarithm of negative values is not defined in the reals numbers, but in the complex numbers.

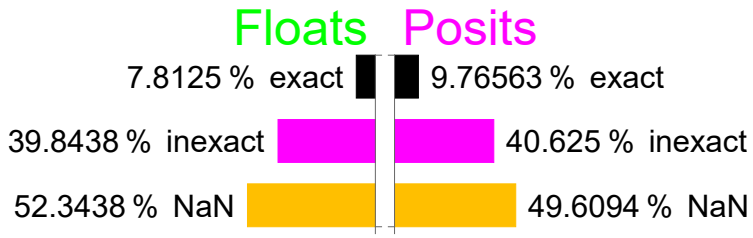


Figure 3.5: Quantitative comparison of floats and posits computing $\log_2(x)$.

As Figure 3.5 shows, although the advantage is slight, posits do better again.

3.2.5. Exponential Base 2

Once 2^x is computed, it is not difficult to change the base to get e^x or 10^x. Thus, the chosen base is not really representative.

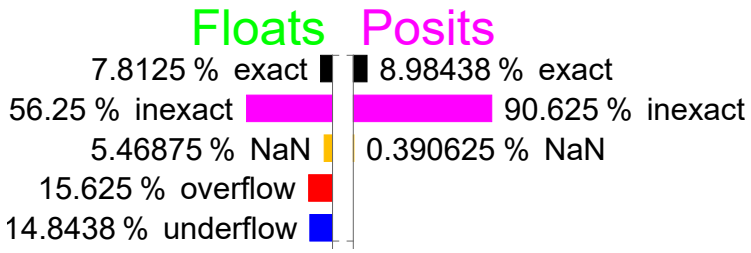


Figure 3.6: Quantitative comparison of floats and posits computing 2^x.

As can be seen on Figure 3.6, performing the exponential operation on posits produces one exception: the exponent $\pm\infty$ results in NaN. However, this situation is not used for real computations, since $2^{+\infty}$ is not computable any more and $2^{-\infty}$ is equal to zero. Even so, posits do not suffer from overflow nor underflow, which means that more values can be properly estimated.

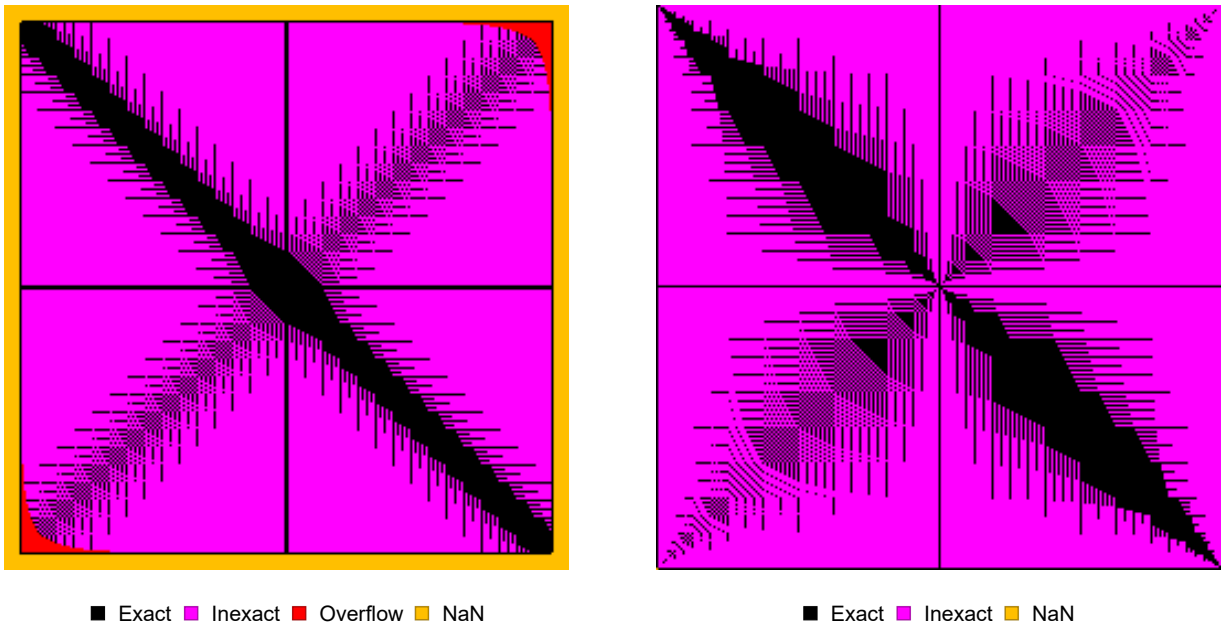
As the above tests show, posits are more accurate when performing common unary operations with the same number of bits. However, the obtained precision for posits is tied closely to the exponent size; for example, if we consider Posit$\langle 8, 0\rangle$ configuration, the amount of exact values would be around 5% lower than the obtained with $es = 1$ (all those values would become inexact, never underflow, overflow or NaN).

3.3. Two-Argument Operation Comparisons

Now we focus on the four elementary arithmetic operations that take two operands. The methodology is as described in the previous section. To help visualize the results, in this case we make use of matrices that show the results of operating the different 256 values.

3.3.1. Addition and Subtraction

Subtraction can be interpreted as addition of a negative value. Therefore, there is no need to study both operations separately. For comparing the addition operation (and the basic operations below), first the exact value of $x + y$ is computed and then it is compared with the result obtained by following the rules of each number system – it can be an exact result, maybe some rounding is needed, it can overflow or underflow, or can be indeterminate like ∞ or $-\infty$, which produces NaN. Each case is distinguished with a different color in the result matrices. Figure 3.7 shows the closure plots for both floats and posits in a way that can easily be compared.



(a) Closure plot for float addition.

(b) Closure plot for posit addition.

Figure 3.7: Complete closure plots for float and posit addition tables.

The result is also quantized in Figure 3.8. It is clear that posits have more additions that are exact. Of course, the boundary of the floats' matrix is full of NaN; this is expected to also happen in all the other operations, since the amount of bit patterns for representing NaN that floats have. Overflow also occurs when adding two large float numbers with the same sign. In the case of posits, there is only a NaN case, produced by operation $(\pm\infty) + (\pm\infty)$.

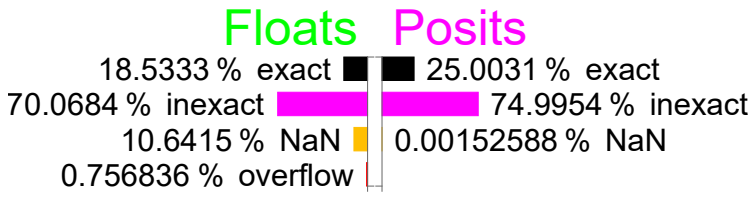
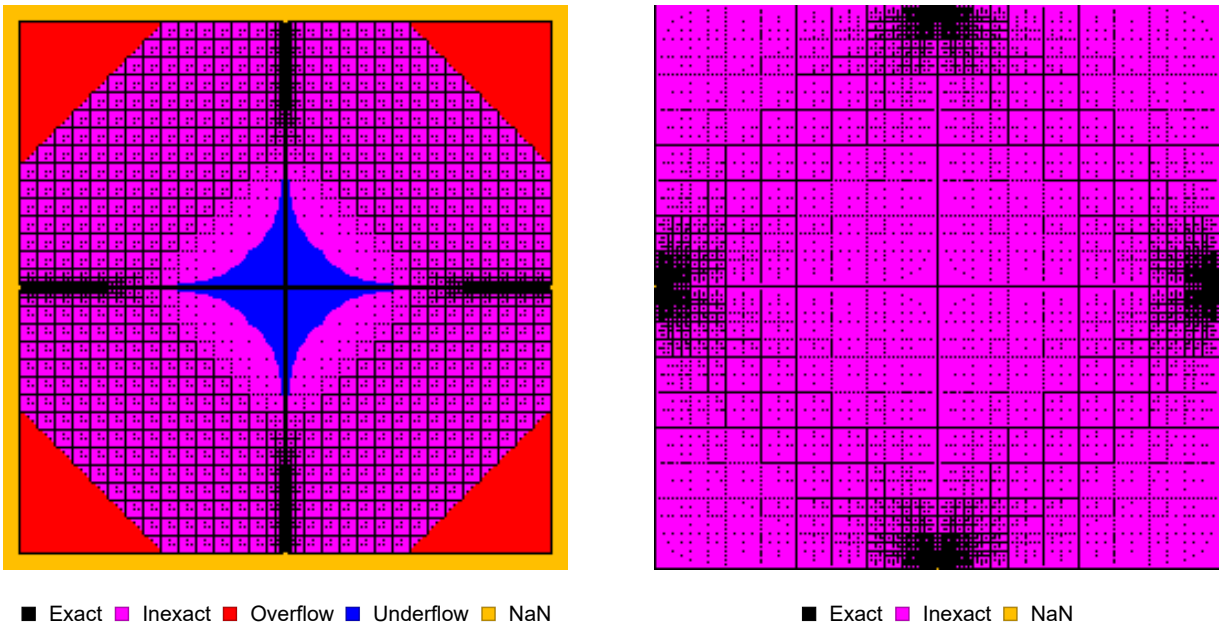


Figure 3.8: Quantitative comparison of floats and posits for addition.

3.3.2. Multiplication

Comparison between floats and posits multiplication is performed in the same way. The difference in this case is that floats can underflow too. Figure 3.9 shows the closure graphs. While floats overflow when multiplying two large numbers (does not matter the sign in this operation) or underflow when multiplying two small numbers – the result is rounded to 0 –, the posits graph has only two NaN values, at the middle of two edges, corresponding to the situations $0 \times \pm\infty$.



(a) Closure plot for float multiplication.

(b) Closure plot for posit multiplication.

Figure 3.9: Complete closure plots for float and posit multiplication tables.

Regarding the number of exact values (see Figure 3.10), this is the first time floats surpass

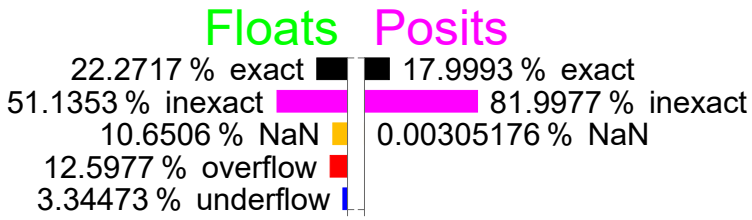
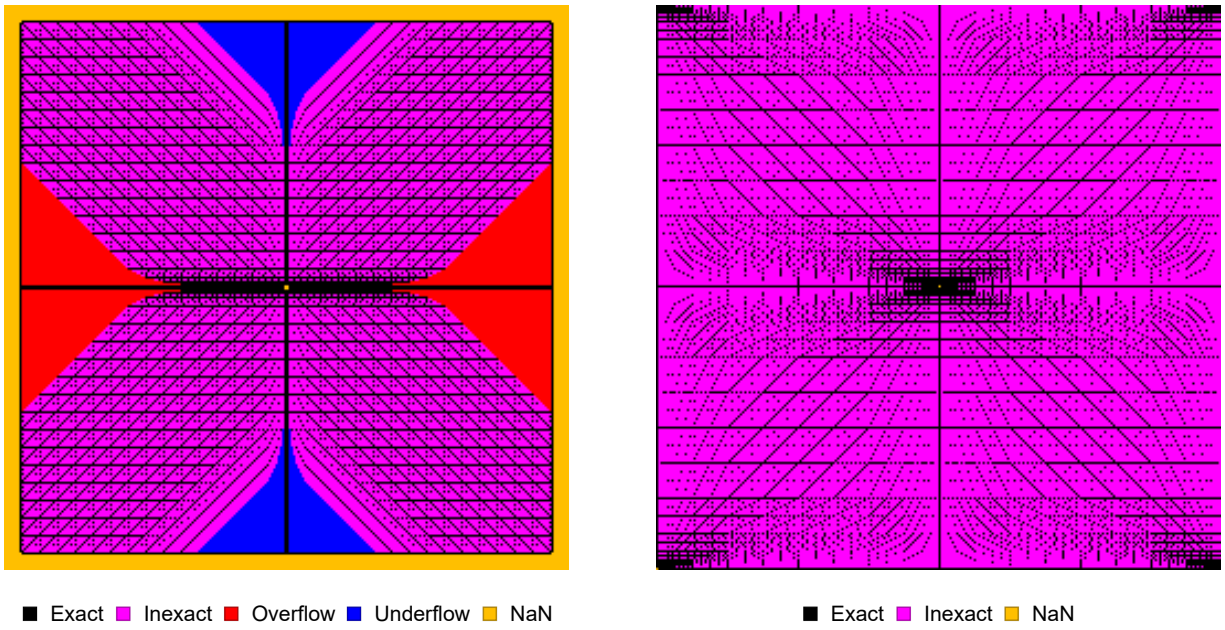


Figure 3.10: Quantitative comparison of floats and posits for multiplication.

posits. However, more than 25% of all float products are useless for computations.

3.3.3. Division

In a similar way the closure for division operations can be compared. As Figure 3.11 shows, the regions are permuted in contrast to the multiplication result.



(a) Closure plot for float division.

(b) Closure plot for posit division.

Figure 3.11: Complete closure plots for float and posit division tables.

The results from Figure 3.12 show the same result as for the multiplication with the posit format. On the other hand, the number of exact float values has decreased.

As happened in the previous section with the single-argument operations, the high

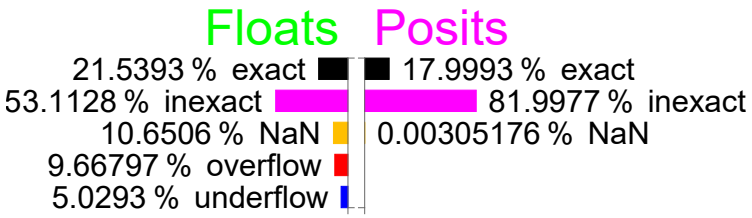


Figure 3.12: Quantitative comparison of floats and posits for division.

accuracy of posits depends on the maximum number of bits for the exponent. However, compared again with the format used in these tests, the 8-bit posit with $es = 0$ shows a very interesting behavior: while the number of exact values after performing the multiplication and division decays around the 5%, Figure 3.13 shows what happens in the case of addition.

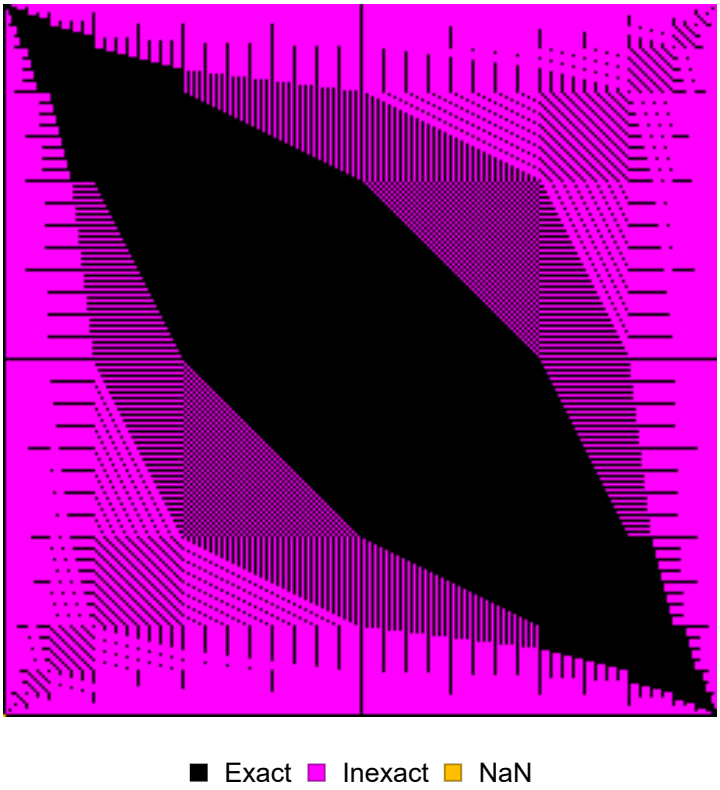


Figure 3.13: Closure plot for Posit$\langle 8, 0\rangle$ addition.

In this case the amount of exact values is over 45%, really high compared to the 25%

that posits with 1 exponent bit reach and the poor 18.5% that floats archive. But there is more on this result. As can be seen at Figure 3.13, there is a thick continuous black diagonal band which denotes that this configuration of posits performs exact additions in a wide continuous domain and, in particular, in a neighborhood of 0, that is, according to Gustafson, "where most calculations occur".

3.4. Algebraic Problems

Algebra is one of the oldest and main branches of Mathematics, and is based on the study of mathematical symbols and the rules for manipulating them. Elementary Algebra for problem solving makes use of Arithmetic and, from a mathematical point of view, this is enough to obtain exact solutions. However, when trying to solve these kind of problems computationally on a floating-point arithmetic-based system, computations may produce some exceptions such as overflow, and is responsibility of the programmer to modify the original problem in a way these exceptions are avoided, if possible.

This section presents some examples of how posits perform better than floats on solving algebraic problems when they are implemented in a naive way.

3.4.1. The Thin Triangle Problem

Let us propose the following elementary-school algebra problem: compute the area of a very flat triangle, i.e., whose base is much greater than its height. This is known as the "thin triangle" problem [1]. Suppose sides a, b, c, such that $a \approx b + c$. In particular let us take sides b and c just 3 Units in the Last Place (ULPs) longer than the half of the side a (see Figure 3.14).

Instead of using the well-known formula $A = bh/2$, which makes use of the height and therefore it would be needed to perform more computations such as squaring, let us use the classic Heron's formula:

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \tag{3.1}$$

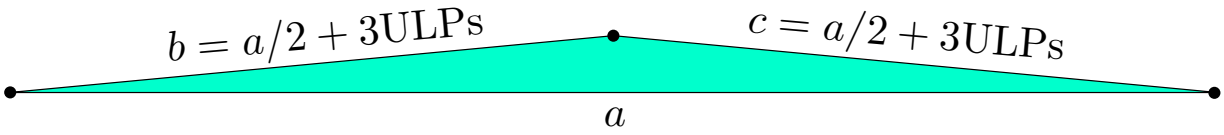


Figure 3.14: Thin triangle problem.

where $s = (a + b + c)/2$ is the semiperimeter.

Formula (3.1) has risk of wrong rounding in the sense that for a very thin triangle s is very close to a. Let us choose values $a = 7$, $b = c = 7/2 + 3 \times 2^{-111}$ and use 128-bit (quad-precision) IEEE floats and 128-bit posits ($es = 7$). The results are shown in Table 3.1.

Correct answer	$3.14784204874900425235885265494550774498\ldots \times 10^{-16}$
128-bit IEEE float answer	$3.63481490842332134725920516158057682879\ldots \times 10^{-16}$
128-bit posit answer	$3.14784204874900425235885265494550774439\ldots \times 10^{-16}$

Table 3.1: Computations of the thin triangle problem.

The correct answer can be obtained using extended precision such as Mathematica provides. When comparing the different solutions, is clear that posits have much more accuracy than floats in quad-precision – the answer provided by floats is only one decimal digit correct, while the posits one is 37 decimals. Even converting this quad-precision result into single-precision would be far more accurate using posits than floats. Similar results can be obtained computing the height and multiplying by the half of the base, but 3 less decimal digits are correct while using posits.

3.4.2. Linear Systems

An interesting example presented in [4] is the following 2-by-2 linear system:

$$\begin{pmatrix} 0.25510582 & 0.52746197 \\ 0.80143857 & 1.65707065 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.79981812 \\ 2.51270273 \end{pmatrix} \tag{3.2}$$

One can easily check that the answer is $x = -1$, $y = 2$. However, the problem with this system is that its condition number is extremely large[1].

To compute the solution, since the system is only 2-by-2, the Cramer's rule is a simple method for that. In this case, let us use 64-bit (double-precision) IEEE floats and posits ($es = 3$). However, even double precision floats have 15 decimal digits of precision, some rounding error is produced when converting the decimals to binary representation. In order to avoid that, let us try to solve the following equivalent system to (3.2):

$$\begin{pmatrix} 25510582 & 52746197 \\ 80143857 & 165707065 \end{pmatrix} \frac{1}{2^8} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 79981812 \\ 251270273 \end{pmatrix} \frac{1}{2^8}$$

These values have now no representation problem with double-precision IEEE floats. Using a solver method in Mathematica one can confirm that the exact answer is $x = -1$, $y = 2$. Nonetheless, the result of performing Cramer's rule with double-precision IEEE floats produces a terrible rounding error: $x = 0$, $y = 2$. This is due to an underflow on the x numerator expression. On the other side, the computation with 64-bit posits ($es = 3$) gives the correct solution. And what is more, if we change the scaling factor by $1/2^{26}$ in the previous system, we can turn down the posit length up to 59 bits and still get the correct result.

3.5. Newton-Raphson Method

At the beginning of this chapter, we thoroughly tested 8-bit floats and posits in Mathematica from a technical point of view. In the previous section, some more practical problems were computed with different precisions, closer to real situations. In this section, we present a completely applicable case of use to compare both posit and IEEE float formats. Python is used instead of Mathematica. The choice of this programming language is due to its high popularity[2], the facility to visualize results graphically and the support to posit arithmetic.

[1] The condition number of a linear system measures how the solution will change with respect to a change in the inhomogeneous term. Thus, a large condition means a small error (or rounding) in the right-side terms may cause large error in the solution. Well-posed systems have a condition number near to 1.

[2] At the date of this dissertation, Python is one of the most popular programming languages at *GitHub*.

We will use the SoftPosit package from S.H. Leong [15] to perform computations on posits. This package has been integrated in the Python scientific computing library NumPy [17]. Along this chapter we will also use a 32-bit environment for both floats (single-precision) and posits.

3.5.1. Explanation of the Method

The tests from previous section are scenarios that show the effects of wrong rounding and underflow in the case of floats. Below we provide some more scenarios to compare how extremely large/low values affect posits and floats computations.

The Newton-Raphson method has been chosen because this method seeks roots (or zeroes) and there is therefore some risk of underflow.

The basic version consist on taking a single-variable function f defined for a real variable x and an initial guess x_0 for a root of f. If the function is differentiable and the initial guess is close, then the subsequent iterations

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

provide better approximations of a root of f.

Let us take the function $f(x) = x^n - a$. This function has been chosen for two reasons – it is easy to implement and derivate, and its zeroes are on the form $\sqrt[n]{a}$, so this method is used to compute n^{th} roots numerically.

3.5.2. Computing a Common Root

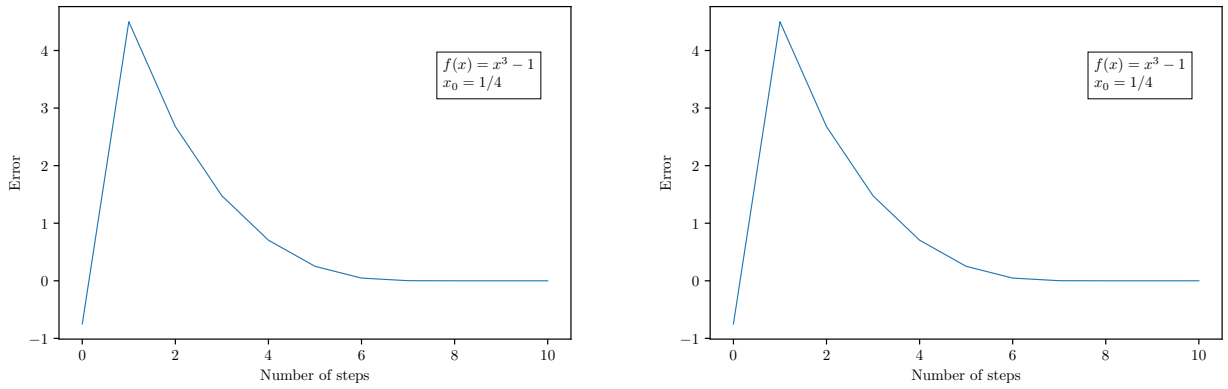
The first test is to calculate the 3^{rd} root of 1 starting at 1/4. One can expect that there will not be any problems in this case, since calculations do not lead to extremely large or small numbers.

The package that is used has some limitations for posits operations, such as integer multiplication or power. Therefore, and in order to check that there are not intermediate casts, those operations have to be performed manually.

With these modifications, the Newton-Raphson method can be applied to the function $f(x) = x^3 - 1$ with the initial guess $x_0 = 1/4$. The results of the first 10 steps are shown in Table 3.2 and Figure 3.15.

Step	Posit	Posit Error	Float	Float Error
0	0.25	-0.75	0.25	-0.75
1	5.5	4.5	5.5	4.5
2	$3.677\ldots$	$2.677\ldots$	$3.677\ldots$	$2.677\ldots$
3	$2.476\ldots$	$1.476\ldots$	$2.476\ldots$	$1.476\ldots$
4	$1.705\ldots$	$7.053\ldots \cdot 10^{-1}$	$1.705\ldots$	$7.053\ldots \cdot 10^{-1}$
5	$1.251\ldots$	$2.514\ldots \cdot 10^{-1}$	$1.251\ldots$	$2.514\ldots \cdot 10^{-1}$
6	$1.047\ldots$	$4.715\ldots \cdot 10^{-2}$	$1.047\ldots$	$4.715\ldots \cdot 10^{-2}$
7	$1.002\ldots$	$2.092\ldots \cdot 10^{-3}$	$1.002\ldots$	$2.092\ldots \cdot 10^{-3}$
8	$1.000\ldots$	$4.358\ldots \cdot 10^{-6}$	$1.000\ldots$	$4.410\ldots \cdot 10^{-6}$
9	1.0	0.0	1.0	0.0
10	1.0	0.0	1.0	0.0

Table 3.2: Estimated value and error per step for the function $f(x) = x^3 - 1$ with $x_0 = 1/4$.



(a) Error of Newton-Raphson for floats.

(b) Error of Newton-Raphson for posits.

Figure 3.15: Error from the Newton-Raphson method of the function $f(x) = x^3 - 1$ with initial value $x_0 = 1/4$.

Both posits and floats approach the exact solution very quickly – after 9 steps. This is because the number that is approximated is 1, which is exactly representable by both formats and easy to compute. Furthermore, the chosen n is small, which causes relatively small rounding errors[3]. Figure 3.15 shows that both formats behave the same in this test. The

[3]In these tests, the error computed is the absolute error, i.e., the difference between the inferred value

error at the first step is due not to the rounding errors, but to the Newton-Raphson method itself (just check that the analytic value of the first iteration is $x_0 - f(x_0)/f'(x_0) = 5.5$, which is equal to the value computed by floats and posits).

3.5.3. Computing a Root of 0

If the value to be calculated is 1, then posits and floats work as expected. Now we take a function that is almost the same as the previous one, but the answer that we are now looking for is the 3$^{\text{rd}}$ root of 0. The function is therefore $f(x) = x^3$ and the initial value $x_0 = 1/4$. The test uses largely the same methodology as presented in the previous test.

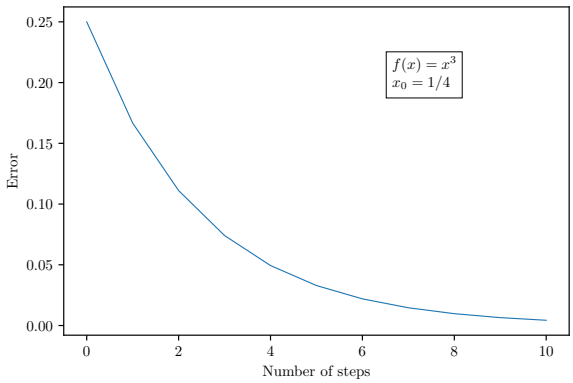
Since posits around 0 do not round because they have no underflow, it is expected that posits can calculate to higher accuracy than floats. The results of this test are shown in Table 3.3 and Figure 3.16.

Step	Posit	Posit Error	Float	Float Error
0	0.25	0.25	0.25	0.25
1	$0.166\ldots$	$0.166\ldots$	$0.166\ldots$	$0.166\ldots$
2	$1.111\ldots \cdot 10^{-1}$	$1.111\ldots \cdot 10^{-1}$	$1.111\ldots \cdot 10^{-1}$	$1.111\ldots \cdot 10^{-1}$
3	$7.407\ldots \cdot 10^{-2}$	$7.407\ldots \cdot 10^{-2}$	$7.407\ldots \cdot 10^{-2}$	$7.407\ldots \cdot 10^{-2}$
4	$4.938\ldots \cdot 10^{-2}$	$4.938\ldots \cdot 10^{-2}$	$4.938\ldots \cdot 10^{-2}$	$4.938\ldots \cdot 10^{-2}$
5	$3.292\ldots \cdot 10^{-2}$	$3.292\ldots \cdot 10^{-2}$	$3.292\ldots \cdot 10^{-2}$	$3.292\ldots \cdot 10^{-2}$
6	$2.194\ldots \cdot 10^{-2}$	$2.194\ldots \cdot 10^{-2}$	$2.194\ldots \cdot 10^{-2}$	$2.194\ldots \cdot 10^{-2}$
7	$1.463\ldots \cdot 10^{-2}$	$1.463\ldots \cdot 10^{-2}$	$1.463\ldots \cdot 10^{-2}$	$1.463\ldots \cdot 10^{-2}$
8	$9.755\ldots \cdot 10^{-3}$	$9.755\ldots \cdot 10^{-3}$	$9.755\ldots \cdot 10^{-3}$	$9.755\ldots \cdot 10^{-3}$
9	$6.503\ldots \cdot 10^{-3}$	$6.503\ldots \cdot 10^{-3}$	$6.503\ldots \cdot 10^{-3}$	$6.503\ldots \cdot 10^{-3}$
10	$4.335\ldots \cdot 10^{-3}$	$4.335\ldots \cdot 10^{-3}$	$4.335\ldots \cdot 10^{-3}$	$4.335\ldots \cdot 10^{-3}$

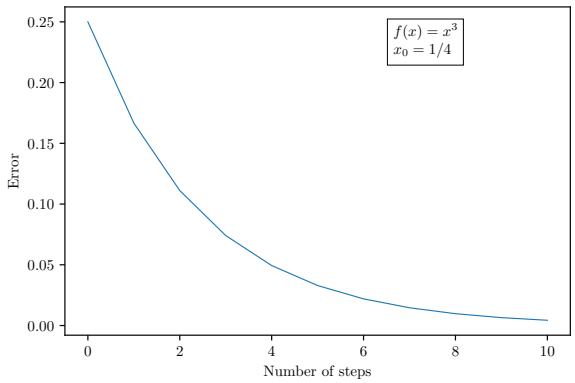
Table 3.3: Estimated value and error per step for the function $f(x) = x^3$ with $x_0 = 1/4$.

The target value is 0, so the error is equal to the value found in both cases. As can be seen, again posits and floats behave the same, but this time 10 steps are not enough to reach the solution. We will therefore increase the number of steps until maximum precision is achieved. In this case that is 100 steps. The results are shown in Table 3.4.

of the solution x_n and its actual value. In addition, no absolute value is taken, since it is important to distinguish situations where the numerically computed value stabilizes and where it oscillates around the actual solution, taking both positive and negative errors.



(a) Error of Newton-Raphson for floats.



(b) Error of Newton-Raphson for posits.

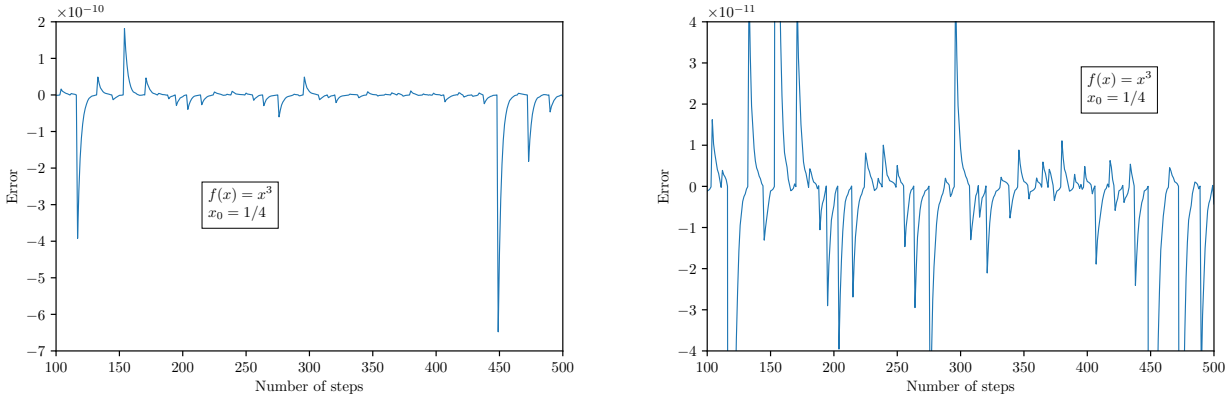
Figure 3.16: Error from the Newton-Raphson method of the function $f(x) = x^3$ with initial value $x_0 = 1/4$.

Step	Posit	Float
0	0.25	0.25
10	$4.335\ldots \cdot 10^{-3}$	$4.335\ldots \cdot 10^{-3}$
20	$7.518\ldots \cdot 10^{-5}$	$7.518\ldots \cdot 10^{-5}$
30	$1.304\ldots \cdot 10^{-6}$	$1.304\ldots \cdot 10^{-6}$
40	$2.260\ldots \cdot 10^{-8}$	$2.261\ldots \cdot 10^{-8}$
50	$3.923\ldots \cdot 10^{-10}$	$3.921\ldots \cdot 10^{-10}$
60	$6.367\ldots \cdot 10^{-12}$	$6.799\ldots \cdot 10^{-12}$
70	$-4.095\ldots \cdot 10^{-12}$	$1.179\ldots \cdot 10^{-13}$
80	$1.729\ldots \cdot 10^{-10}$	$2.036\ldots \cdot 10^{-15}$
90	$2.548\ldots \cdot 10^{-12}$	$8.535\ldots \cdot 10^{-16}$
100	$-1.143\ldots \cdot 10^{-12}$	$8.535\ldots \cdot 10^{-16}$

Table 3.4: Estimated value and error per step for the function $f(x) = x^3$ with $x_0 = 1/4$.

As can be seen, floats reach maximum accuracy after approximately 90 steps. Then, the computed value stays constant. However, posits not only are less accurate in this case, they also continue to change values and even sign. This happens because, in contrast to floats, posits do not underflow. Therefore, the quotient $f(x)/f'(x)$ is never rounded to 0 and new values continue to be calculated. To better understand what happens, let us view more steps.

In Figure 3.17 the behavior of the posit test is represented graphically from steps 100 to 500. The posits continue fluctuating and do not stabilize. The outliers are very large, as



(a) Error in the interval $[100, 500]$.

(b) Zoomed error in the interval $[100, 500]$.

Figure 3.17: Error from the Newton-Raphson method of the function $f(x) = x^3$ with initial value $x_0 = 1/4$ for posits.

can be seen in Figure 3.17a. Zooming further into Figure 3.17b, it can be seen that even if we ignore these large outliers, the values fluctuate considerably.

In the case of approximate the zeroes of the function $f(x) = x^3$, floats are more accurate than posits, since they stabilize thanks to underflow.

3.5.4. Computing Roots in Extreme Situations

If the result must be 0, then floats are a better choice, since they underflow while posits do not. However, one may ask if this underflow and overflow are always so useful. To answer that, let us take the function $f(x) = x^{120} - 2^{-120}$. The choice of term 2^{-120} is based on the fact that this is the smallest value that can be represented exactly by a 32-bit posit with 2 exponent bits. In this case the initial value is set to $x_0 = 4$ in order to see how posits and floats behave when computing with both large and small numbers. The solution this time is $x = 1/2$. Proceeding as above, the result of Table 3.5 follows.

Floats overflow after first step. However, what happened is that both $f(x)$ and $f'(x)$ produce an overflow, so the computer calculates ∞/∞ here, which results in a NaN, and consequently all the subsequent steps are NaN.

This does not apply to posits, since they do not overflow. However, they do not reach

Step	Posit	Posit Error	Float	Float Error
0	4.0	3.5	4.0	3.5
20	$8.674\ldots \cdot 10^{-1}$	$3.674\ldots \cdot 10^{-1}$	NaN	NaN
40	$7.337\ldots \cdot 10^{-1}$	$2.337\ldots \cdot 10^{-1}$	NaN	NaN
60	$6.208\ldots \cdot 10^{-1}$	$1.208\ldots \cdot 10^{-1}$	NaN	NaN
80	$7.108\ldots \cdot 10^{-1}$	$2.108\ldots \cdot 10^{-1}$	NaN	NaN
100	$6.009\ldots \cdot 10^{-1}$	$1.009\ldots \cdot 10^{-1}$	NaN	NaN
120	$6.871\ldots \cdot 10^{-1}$	$1.871\ldots \cdot 10^{-1}$	NaN	NaN
140	$5.797\ldots \cdot 10^{-1}$	$7.973\ldots \cdot 10^{-2}$	NaN	NaN
160	$6.645\ldots \cdot 10^{-1}$	$1.645\ldots \cdot 10^{-1}$	NaN	NaN

Table 3.5: Estimated value and error per step for the function $f(x) = x^{120} - 2^{-120}$ with $x_0 = 4$.

the solution and error does not decrease. To understand this phenomenon, let us focus on first 500 steps. The result is graphically represented in Figure 3.18.

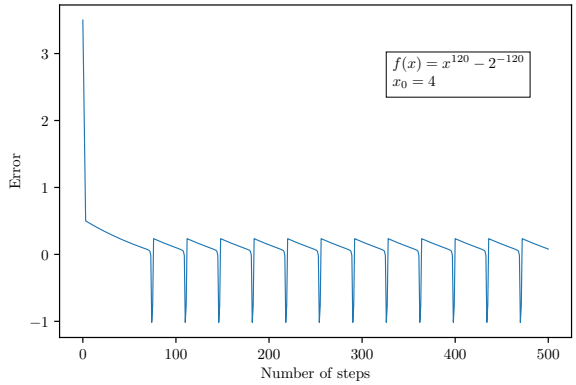


Figure 3.18: Error from the Newton-Raphson method of the function $f(x) = x^{120} - 2^{-120}$ with initial value $x_0 = 4$ for posits.

As can be seen, the error decreases to 0, but at a certain moment the sign turns and the error becomes considerably larger. This process is repeated forever. In this case, posits perform better than floats, but the results found oscillate around the solution with a big error.

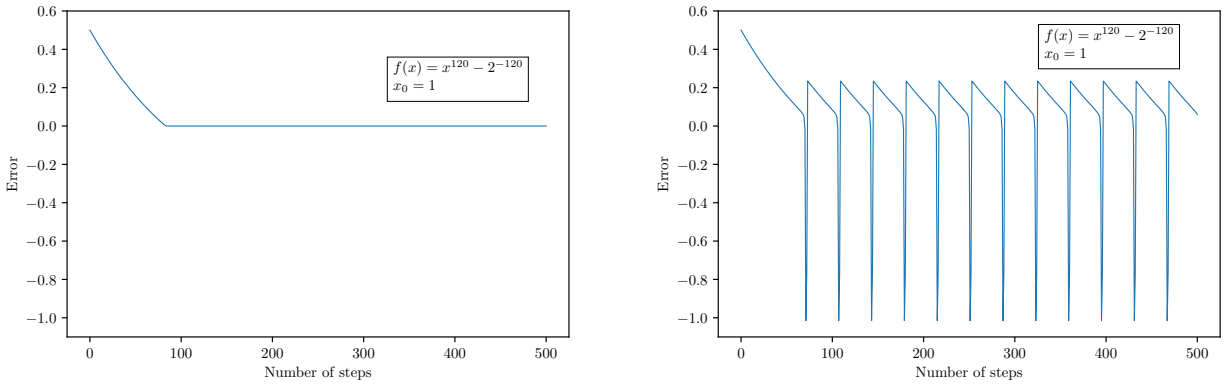
After this test, where floats were unable to give a result due to overflow, one may ask if floats can make it better if the initial value is chosen in such a way there is no overflow. So

if choosing $x_0 = 1$ as initial value the problem of overflow is avoided. The results are shown in Table 3.6.

Step	Posit	Posit Error	Float	Float Error
0	1.0	0.5	1.0	0.5
10	$9.197\ldots \cdot 10^{-1}$	$4.197\ldots \cdot 10^{-1}$	$9.197\ldots \cdot 10^{-1}$	$4.197\ldots \cdot 10^{-1}$
20	$8.459\ldots \cdot 10^{-1}$	$3.459\ldots \cdot 10^{-1}$	$8.459\ldots \cdot 10^{-1}$	$3.459\ldots \cdot 10^{-1}$
30	$7.780\ldots \cdot 10^{-1}$	$2.780\ldots \cdot 10^{-1}$	$7.780\ldots \cdot 10^{-1}$	$2.780\ldots \cdot 10^{-1}$
40	$7.155\ldots \cdot 10^{-1}$	$2.155\ldots \cdot 10^{-1}$	$7.155\ldots \cdot 10^{-1}$	$2.155\ldots \cdot 10^{-1}$
50	$6.581\ldots \cdot 10^{-1}$	$1.581\ldots \cdot 10^{-1}$	$6.581\ldots \cdot 10^{-1}$	$1.581\ldots \cdot 10^{-1}$
60	$6.054\ldots \cdot 10^{-1}$	$1.054\ldots \cdot 10^{-1}$	$6.053\ldots \cdot 10^{-1}$	$1.053\ldots \cdot 10^{-1}$
70	$4.850\ldots \cdot 10^{-1}$	$-1.500\ldots \cdot 10^{-2}$	$5.567\ldots \cdot 10^{-1}$	$5.667\ldots \cdot 10^{-2}$
80	$6.932\ldots \cdot 10^{-1}$	$1.932\ldots \cdot 10^{-1}$	$5.121\ldots \cdot 10^{-1}$	$1.213\ldots \cdot 10^{-2}$
90	$6.376\ldots \cdot 10^{-1}$	$1.376\ldots \cdot 10^{-1}$	0.5	0
100	$5.843\ldots \cdot 10^{-1}$	$8.427\ldots \cdot 10^{-2}$	0.5	0

Table 3.6: Estimated value and error per step for the function $f(x) = x^{120} - 2^{-120}$ with $x_0 = 4$.

As with the previous initial value, posits do no reach the solution – they behave the same, oscillating around it. Floats, on the other hand, not only do not suffer from overflow this time, but they get to the exact solution (they are taking advantage of underflow again). To have a better knowledge of what occurs, the errors are depicted in Figure 3.19.



(a) Error of Newton-Raphson for floats.

(b) Error of Newton-Raphson for posits.

Figure 3.19: Error from the Newton-Raphson method of the function $f(x) = x^{120} - 2^{-120}$ with initial value $x_0 = 4$.

With this we conclude the performance study for floats and posits with 32 bits (single-

precision). For common situations, none of the formats stands out from the other. However, in extreme corner cases floats perform better since underflow is an advantage in these cases. Posits, in contrast, have problems when approximating 0 – as underflow never occurs, the answer provided oscillates around – and when large or small numbers are involved – posit answers with a huge error loop. Even so, it is important to recall that posits always produce an answer, and even it may sometimes be far from the solution, it is still more accurate than the NaN or infinite value that floats can provide when overflowing.

3.5.5. Computations with Half-Precision

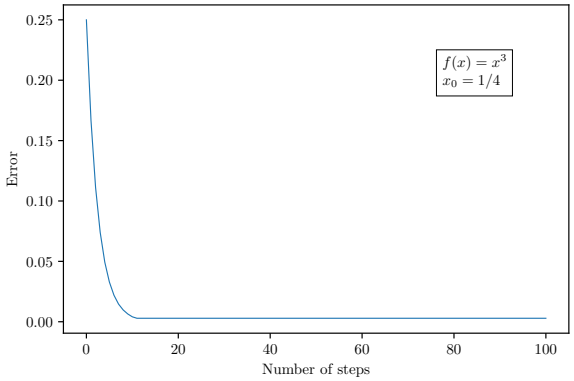
To conclude our study of numerical precision, let us repeat the Newton-Raphson method comparing half-precision (16-bit) floats with Posit$\langle 16, 1\rangle$, also known as *posit16*. We just adjust the precision in the script and run the same tests. Below we present the results and compare them between both formats and with the above computations using 32 bits.

For the root of $f(x) = x^3 - 1$ with $x_0 = 1/4$ there are almost no differences with the 32-bit cases, just both formats reach the solution one step earlier.

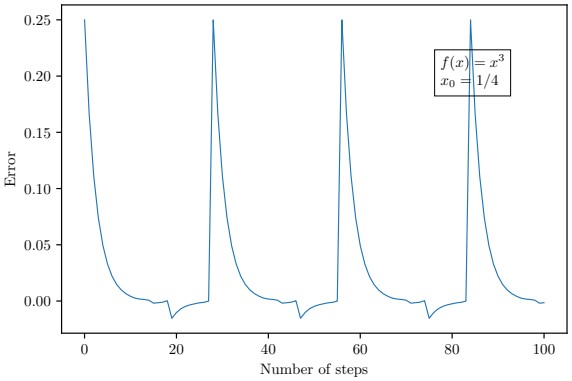
The results of computing the root of $f(x) = x^3$ with $x_0 = 1/4$ are depicted in Figure 3.20. As can be seen, floats have apparently the same behavior (but the error this time is higher, it reaches $2.884\ldots \cdot 10^{-3}$), but posits do not converge. Instead, they experiment an oscillating behavior already seen in a previous case: the error decreases to 0, but at a certain moment it changes sign and then becomes larger in a kind of loop.

The results obtained in the calculation of $f(x) = x^{120} - 2^{-120}$ are almost the same whether the initial value is 4 or 1: floats overflow in both cases (earlier if the initial value is 4) and posits, in contrast to the 32-bit case, stabilize very near to the solution, reaching the value $0.452\ldots$. Figure 3.21 illustrates this situation. Therefore, in this case 16-bit posits behave better than 32-bit posits and, obviously, outperform floats.

Recall that the choice of term 2^{-120} was due to the fact that it is the minimum value that can be exactly representable using Posit$\langle 32, 2\rangle$. Therefore it is reasonable to wonder if

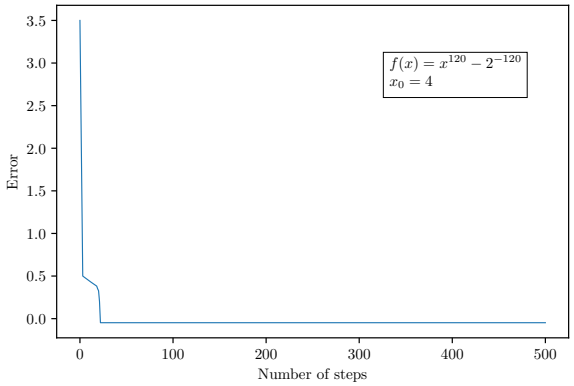


(a) Error of Newton-Raphson for floats.

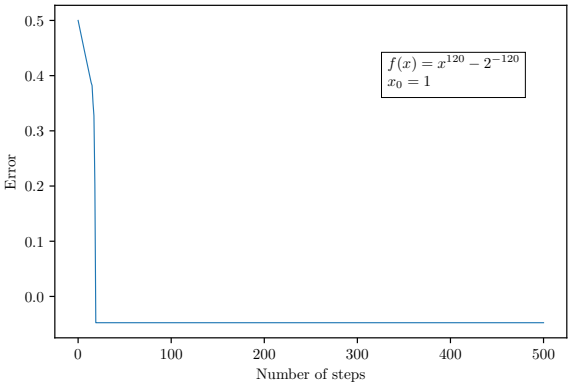


(b) Error of Newton-Raphson for posits.

Figure 3.20: Error from the Newton-Raphson method of the function $f(x) = x^3$ with initial value $x_0 = 1/4$ using 16 bits.



(a) Error of Newton-Raphson for posits with $x_0 = 4$.

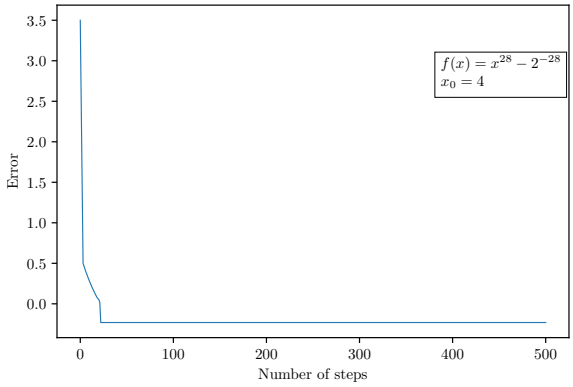


(b) Error of Newton-Raphson for posits with $x_0 = 4$.

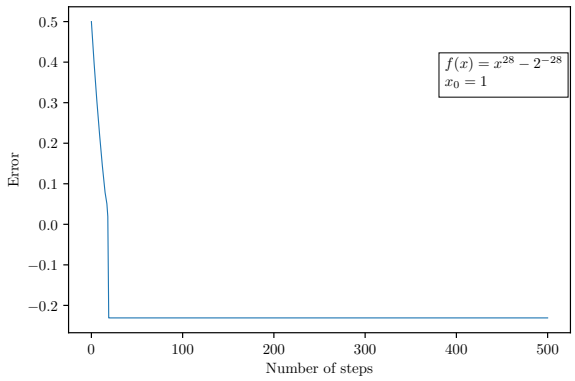
Figure 3.21: Error from the Newton-Raphson method of the function $f(x) = x^{120} - 2^{-120}$ for 16-bit posits.

an analogous result would be produced using Posit$\langle 16, 1\rangle$ and the minimum representable value, i.e., 2^{-28}. However, the answer is no. In fact, the results of computing the roots of $f(x) = x^{28} - 2^{-28}$ with 16-bit formats, shown in Figure 3.22, are very similar to the obtained on the previous test – floats overflow again and posits converge to the value $0.269\ldots$, which in this case is a bit far from the real solution 0.5 but in contrast with the analogous situation with 32 bits, posits do not keep oscillating around the solution.

With this we conclude our numerical precision study of standard floating-point numbers



(a) Error of Newton-Raphson for posits with $x_0 = 4$.



(b) Error of Newton-Raphson for posits with $x_0 = 4$.

Figure 3.22: Error from the Newton-Raphson method of the function $f(x) = x^{28} - 2^{-28}$ for 16-bit posits.

and posit number system with 16 bits. Based on the results of the tests above, we can conclude that *posit16* outperform half-precision floats and do not present an oscillating behavior in extreme corner cases as the analogous posit format with 32 bits.

The purpose of this section was to compare the performance of float and posit number systems in numerical methods under different precision formats. In particular, the Newton-Raphson method has been used to calculate different zero points. From the obtained results we can conclude that even posits can provide an answer when floats overflow, there are cases when, thanks to underflow, floats can provide better answers than posits, so none of the formats is more suitable than the other for all the situations.

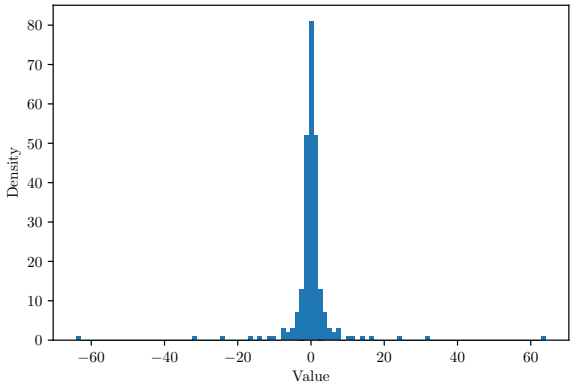
Chapter 4

Neural Networks with Posits

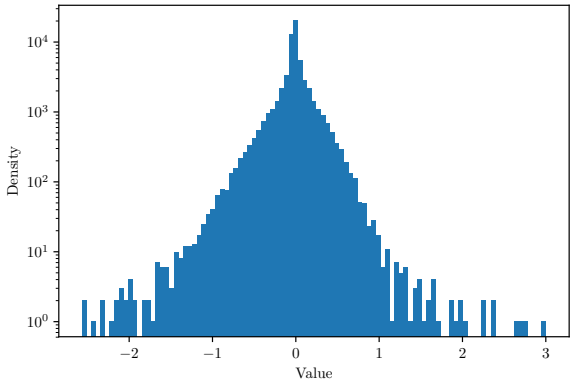
The recent surge of interest in Artificial Intelligence – and in particular in DL – together with the limitations this sector currently has in terms of power consumption and memory resources make us wonder if posits can be helpful in this field. As shown in this dissertation, the posit number system has many interesting properties, such as lack of underflow or overflow (Section 2.4.2), the support of fused operations, in particular the dot product (Section 2.4.3), and the nice and fast approximation of sigmoid function that some configurations of posits can do (Section 2.4.4). This, along with another property mentioned in Section 2.4.2 and explained below, suggests that posits may be suitable for performing deep learning tasks.

This last property that can be of interest is the so-called *tapered precision* [18]. As mentioned before, in a format with tapered precision the values mass around 0 and sparse to higher or lower numbers in less frequency, so representation of small values is more accurate than using other formats. When we use a number system with tapered precision, such as posits, the values follow a normal distribution centered in 0. That is the same distribution that Deep Neural Network (DNN) weight parameters usually follow, but even more grouped around 0. Figure 4.1 illustrates this concept, which suggests that using posits for DNNs may provide more accurate results.

This chapter focuses on implementing different deep learning algorithms using the posit number system, and compare the obtained results with the float based ones. For simplicity, the tests start with a simple multilayer perceptron and follow with a Convolutional Neural



(a) Distribution of Posit$\langle 8, 0\rangle$ values.



(b) LeNet-5 weight distribution for CIFAR-10.

Figure 4.1: Distributions of posit values and Neural Network weights.

Network (CNN).

4.1. Neural Network Training

A first approach to NNs implementation in the posit format is done with the help of Python language and the posit-arithmetic library PySigmoid [16]. The choice of this particular package is due to the fact that it allows working with specific posit configurations, not only the "common" *posit8* (Posit$\langle 8, 0\rangle$), *posit16* (Posit$\langle 16, 1\rangle$) and *posit32* (Posit$\langle 32, 2\rangle$), and that it has a function that simulates the hardware operation for fast sigmoid, which approximates the original function when posits have $es = 0$, in particular when using *posit8* configuration (Section 2.4.4).

To measure how well posits perform at deep learning tasks, we propose a simple binary classification problem which is depicted in Figure 4.2 – samples consist only of two features and classes are obviously separated by a non-linear boundary.

The task consists in training a simple DNN with two hidden layers of 4 and 8 neurons, respectively, to solve the above binary classification problem and check if posits are suitable for training NNs. We will set different Posit$\langle n, es\rangle$ configurations varying the number of bits n and fixing the exponent size $es = 0$ in order to use the fast sigmoid function and check

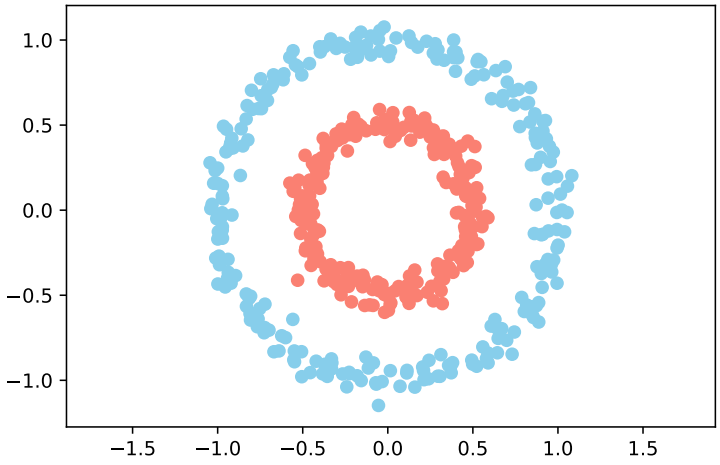


Figure 4.2: Classification problem for posit Deep Neural Network.

how this approximation performs. 32-bit floats and posits with another exponent sizes will be explored too.

Although there are multiple libraries and frameworks for Machine Learning in Python that accelerate and simplify these kind of tasks, our test requires that all the internal computations involving parameters of the network are done in the posit format. Hence, the only option is to implement the NN from scratch, casting the input into posit type and replacing all the internal operands by the ones from PySigmoid library. In this way we can also use the fused dot product with the quire accumulator. The weights and biases are initialized randomly, and the activation function we use is the sigmoid for the cases when fast sigmoid approximation cannot be performed, so comparison is as fairest as possible. We use mean squared error (MSE) as loss function to compare the output of the network with the real solution.

We set a training of 2500 epochs and compare the losses of different configurations along the whole training. In this manner we can compare for the different configurations not only whether the network converges or not, but how fast, which makes a thorough study of results. Table 4.1 and Figure 4.3 depict the obtained results.

As can be seen, there is almost no difference in using single or double-precision floats.

	Epochs					
Configuration	0	250	500	750	1000	1250
32-bit Float	0.3701	0.2346	0.1726	0.0839	0.0023	0.0010
64-bit Float	0.3701	0.2346	0.1727	0.1124	0.0023	0.0010
Posit$\langle 8, 0\rangle$	0.3681	0.1882	0.1491	0.1530	0.1530	0.1530
Posit$\langle 10, 0\rangle$	0.3653	0.2129	0.1359	0.0938	0.1478	0.1264
Posit$\langle 12, 0\rangle$	0.3650	0.2467	0.1758	0.1684	0.0140	0.0081
Posit$\langle 16, 0\rangle$	0.3648	0.2817	0.1716	0.1622	0.0645	0.0035
Posit$\langle 16, 1\rangle$	0.3337	0.1772	0.1453	0.0440	0.0019	0.0011
Posit$\langle 32, 2\rangle$	0.3337	0.1758	0.1658	0.0328	0.0017	0.0009

Table 4.1: Loss function along the Neural Network training.

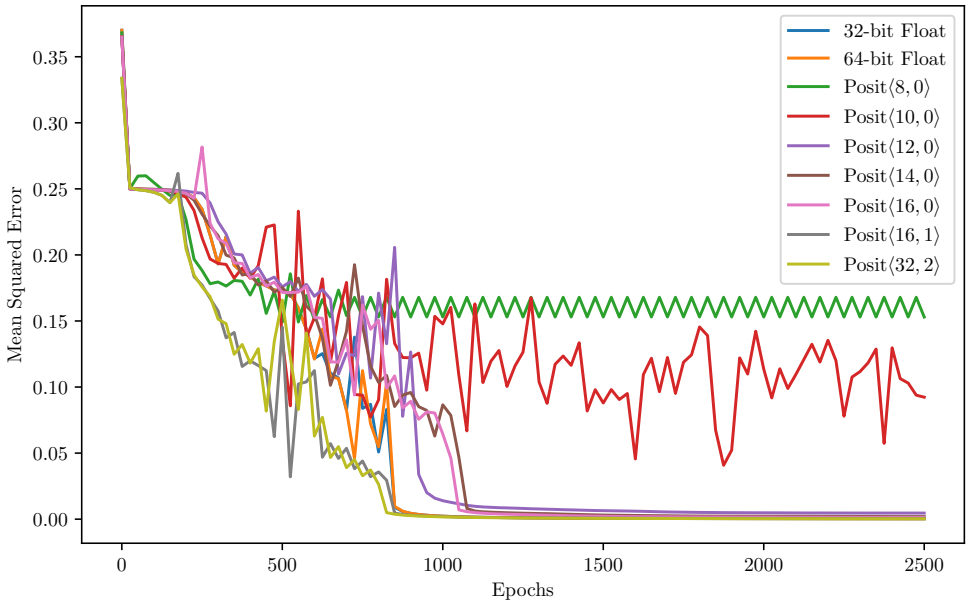


Figure 4.3: Loss function along the Neural Network training.

In addition, both Posit$\langle 32, 2\rangle$ and Posit$\langle 16, 1\rangle$ present the same behavior as floats, but even with less MSE along the fists epochs. All the networks using these four data types converge very fast – in less than 1000 epochs. Also when using Posit$\langle 16, 0\rangle$ the network converges, but this time it requires some extra epochs. The results are more interesting when using less bits: using Posit$\langle 12, 0\rangle$ the network converges even faster than with Posit$\langle 14, 0\rangle$ or Posit$\langle 16, 0\rangle$, which have similar behavior. However, if we reduce the precision to Posit$\langle 10, 0\rangle$, the network

tries to decrease the error, but as Figure 4.3 shows, it remains to high and it does not reach convergence. The case with Posit$\langle 8, 0\rangle$ is completely different; it starts decreasing as the others, but at a certain point, the MSE gets alternating values in a loop and the network never converges. This behavior has already appeared in Section 3.5, and very likely the reason behind such oscillation is the same as before – the lack of underflow.

Based on the obtained results, we can conclude that posits are as good as floats for training NN under the same conditions, and can even be suitable for the training stage using only 12 bits.

4.2. Low-Precision Deep Learning Inference

The results obtained on NNs training are not enough for posits to be a replacement for floats. As some research papers show [19–21], it is difficult to apply lower numerical precision to the training of NNs, especially when using less than 16 bits.

However, many research papers have shown that it is possible to apply low-precision computing to the inference stage of NNs after training with exact arithmetic [22–24]. These results led to the idea of using *posit8* for performing Deep Learning inference. Edge computing is nowadays a key area of research, and performing low-precision inference can be extremely helpful in embedded systems and applications that make use of DL techniques such as Autonomous Driving [25].

In addition to the properties mentioned at the beginning of this chapter, there are some characteristics of the posit format, and in particular of *posit8*, that can be an advantage when using this format in CNNs:

- The comparison of posits uses the same hardware as for comparing integers, which is much faster than floats comparison (Section 2.4.1). Thus, the pooling layers implemented with max pooling can be optimized if running on posit format.

- If using *posit8*, the sigmoid function can be approximately calculated on hardware by

just flipping a bit and shifting two positions, this is extremely fast – only one clock cycle – and cheap to compute, and can speed the inference step.

- Input values for NNs are usually normalized. Therefore, the tapered precision of the posit format is an advantage in this situations.

- The addition of two *posit8* numbers is pretty accurate near the 0 (Section 3.3.1).

The aforementioned items suggest that the *posit8* format may be suitable for performing low precision inference of CNNs. Below we explore the effects this number system has on the accuracy of different image classification problems.

The performance of *posit8* format is evaluated on two datasets: MNIST and CIFAR-10. For the same reason as in the previous tests, the whole network must be implemented from scratch, which motivated us to pick the simple and well-known architecture LeNet-5 [26] instead of others more complex. However, in this case we are only interested in the inference process and to compare it with the accuracy obtained with floats. Therefore, the networks are firstly trained on floating-point arithmetic format and then the weights are stored, converted to posit format and used for inference on the same dataset. In order to reduce the training time, such floating-point nets are implemented using Keras [27] and TensorFlow [13] frameworks. Also for the inference process it is important to reduce the runtime – the datasets are quite large – so for these tests we simulate the posit computations with the help of a NumPy library version which includes a posit data type [17] – computations are much faster than using PySigmoid library, but there is not a fast sigmoid implementation in this package, so simple ReLU is used as activation function instead. The obtained results are shown in Table 4.2.

As can be seen, this time there is not a clear winner. The posit number system outperformed the floating-point when using the MNIST dataset in terms of accuracy (Top-1), but just by a 0.1% and the Top-5 is slightly worse. Also the result archived for a more complex dataset as CIFAR-10 is lower (around 12% in accuracy and 4% in Top-5) when using *posit8*

	32-bit Float		Posit$\langle 8,0\rangle$		Posit$\langle 8,0\rangle$ (only addition)	
Dataset	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
MNIST	99.22%	100.00%	99.32%	99.94%	99.40%	100%
CIFAR-10	68.04%	96.47%	56.11%	92.42%	58.92%	95.62%

Table 4.2: Performance on Convolutional Neural Network inference.

than floats. Nonetheless, we must emphasize that we are using posits of only 8-bit length in contrast to 32-bit floats. This can be immediately translated in terms of power savings in the datapath design and indirectly in terms of memory footprint – after converting the pre-trained weights, and the models themselves, from float to posit format the file sizes are reduced into approximately a quarter of the original sizes.

The last two columns of the table show the results obtained with a hybrid posit-float architecture. Recall that *posit8* has very nice addition closure. This, together with good results using logarithmic number system (LNS) obtained in recent literature [28, 29], motivated us to implement a hybrid system where additions are performed on posit numeric format and the rest of computations using 32-bit floats. The accuracy obtained with this combination of formats is higher for every test that using only posits.

The results obtained in CNN inference with Posit$\langle 8,0\rangle$ are quite satisfactory. On one hand, in some cases posits outperformed floats, even getting higher accuracy than only converting the weights into posit format and performing computations in floating-point arithmetic as in [30]. On the other hand, although using *posit8* decreased accuracy, the gain in memory resources is considerable. Thus, one should consider whether getting lower accuracy is not a great loss compared to the power and memory consumption reduction that 8-bit posits may provide.

Chapter 5

Hardware Implementation

Some of the ideas that motivated to create posits from Type II unums were to design a more "hardware friendly" format, so its architecture would be similar to the floating-point one, and to perform better with the same precision, not only in accuracy terms, but also in silicon footprint and power consumption. The importance of hardware design is therefore in the spotlight when comparing posit and float formats.

In this chapter, first we describe the state of the art comparing the proposed implementations for posit arithmetic. Then we present a fully functional and parameterized posit multiplier, developed as a result of this dissertation.

5.1. Related Works

Since the posit number system was introduced, the interest on a hardware implementation for this format has increased rapidly. Due to the short life time of posits, only a few hardware implementations have been proposed since 2017.

The posit arithmetic unit proposed in [31], and more detailed in [32], includes floating-point to posit conversion, posit to floating-point conversion, addition/subtraction and multiplication. Although this work seems quite complete for a first approach, it is not completely parameterized, since it is not possible to synthesize any posit configuration with zero exponent bits. Furthermore, any of the posit arithmetic units described in this work apply any rounding scheme – they just truncate the fraction values according to the posit size.

Another general design for posit arithmetic unit that includes posit adder and multiplier is presented in [33]. In contrast to the implementations shown in [7, 31], the posit decoder proposed in this architecture uses only a leading zero detector for decoding the regime, while others use a leading one detector too [34]. This, together with other optimizations obtain better area and energy footprint results of posit adders and multipliers. However, this work does not provide any implementation and the algorithms description are poorly detailed in some cases. We provide a clearer implementation based mainly on the one presented in [33].

Since the posit standard includes fused operations such as the fused dot product, and due to the importance of this operation for matrix calculus, some research and development for this kind of implementations has been done. Different matrix-multiply units for posits are presented in [7, 35, 36]. They make use of the quire register to accumulate the partial additions that are involved in the dot product, so the result is rounded only after the whole computation. Therefore, they obtain better accuracy on their computations.

5.2. Proposed Posit Multiplier

At hardware level, posits were designed to be easy to compute, i.e., to have a circuitry similar to the existing floating point. The main encoding difference between float and posit formats is the fact that the second one includes a run-time varying scaling component – the regime and the available exponent bits. This leads to a format that has no fixed fields at run-time, which is a hardware design challenge. Below we present a fully functional posit multiplier operator and show that the hardware design for this module is not so different in comparison with the floating-point arithmetic design.

An important point to consider when designing hardware is the length of the operands and their encoding. However, the posit number format has no length nor exponent fixed size. We could therefore focus in a particular format, or we could create a completely parameterized design, which only depends on the total length of bits (n) and the maximum number of exponent bits (es), i.e., the only two parameters that determine every posit configuration.

As one may guess, the second option seems more promising when working with posits. For this reason we decided to use the FloPoCo framework [37]. FloPoCo (**Flo**ating-**Po**int **Co**res, but not only) is an open-source C++ framework for the generation of arithmetic datapaths which provides a command-line interface that inputs operator specifications and outputs synthesizable VHDL specially suited for Field-Programmable Gate Arrays (FPGAs). Therefore, writing a generic datapath for a certain operator as a C++ class and then automatically generating the VHDL code for a concrete posit configuration simplifies both the task and the resulting circuitry. Next, we present the proposed posit multiplier algorithms, which are based on the specifications of [31, 33, 36].

Analogously to performing computations with IEEE floats, it is necessary to decode the operands fields before carrying out any computation. Therefore, we first present the posit decoding process in Algorithm 1, as the decoder is a common module for many arithmetic modules, not only the multiplier. The explanation of such algorithm is as follows:

- Sign and special cases are detected checking the MSB and ORing the remaining bits, respectively (lines 2–5).

- Since posit arithmetic uses 2's complement for representing negative numbers, dealing with the absolute value simplifies the data extraction process. Therefore, 2's complement of input posit is obtained, only if it is necessary, by XORing the input with the replicated sign bit – works as a 1's complement if sign is nonzero – and adding the sign to the least-significant bit (LSB) (line 6).

- The $twos[N-2]$ bit aids to determine the regime value. In order to use only a leading zero detector [33], we invert the bits of *twos* if the regime consists on a sequence of ones (line 8). Then we count the sequence of `0` bits terminating in a `1` bit using a leading zero detector module (line 9).

- For extracting the exponent and the fraction bits, the regime is shifted out from *twos*, so the exponent is aligned to the left (line 10).

- The first es bits of the shifted string (if $es = 0$ this instruction is omitted) correspond to the exponent bits (line 11), while the remaining bits correspond to the fraction (line 12) – here the hidden bit is appended as the MSB.

- The regime depends on the sequence of identical bits that constitute this field – regime value is $zc - 1$ when the bits are `1` (positive regime) or $-zc$ when it consists on a sequence of `0` bits (negative regime). Note that the sign bit has to be added (line 13).

Algorithm 1 Posit data extraction

```
1:  procedure DECODE(in)
2:      nzero ← ⋁ in[N − 2 : 0]                          ▷ Reduction OR
3:      sign ← in[N − 1]                                  ▷ Extract sign
4:      z ← ¬(sign ∨ nzero)
5:      inf ← sign ∧ ¬(nzero)
6:      twos ← ({N − 1{sign}} ⊕ in[N − 2 : 0]) + sign     ▷ Input 2's complement
7:      rc ← twos[N − 2]                                  ▷ Regime check
8:      inv ← {N − 1{rc}} ⊕ twos
9:      zc ← LZD(inv)                                     ▷ Count leading zeros of regime
10:     tmp ← twos[N − 4 : 0] ≪ (zc − 1)                  ▷ Shift out the regime
11:     exp ← tmp[N − 4 : N − es − 3]                     ▷ Extract exponent
12:     frac ← nzero & tmp[N − es − 4 : 0]                ▷ Extract fraction
13:     reg ← rc ? '0' & zc − 1 : −('0' & zc)             ▷ Select regime
14:     return sign, reg, exp, frac, z, inf
15: end procedure
```

The process of posit multiplication is almost the same as for floating-point multiplication, i.e. the scaling factors are added and the fractions are multiplied and rounded. There are few differences when multiplying posits due to the regime field – as it has a variable length it is not trivial to compute the resulting regime. The pseudocode for posit multiplication is shown in Algorithm 2 and the explanation of the flow is as follows:

- When the two operands are decoded (lines 2–3), the sign and special cases are handled easily (lines 4–6).

- The scaling factor (SF) of each operand consists on the regime and the exponent values,

Algorithm 2 Proposed Posit Multiplier Algorithm

1: **procedure** PositMult(in_A, in_B)
2: $sign_A, reg_A, exp_A, frac_A, z_A, inf_A \leftarrow$ Decode(in_A)
3: $sign_B, reg_B, exp_B, frac_B, z_B, inf_B \leftarrow$ Decode(in_B)
4: $sign \leftarrow sign_A \oplus sign_B$ ▷ Sign computation
5: $z \leftarrow z_A \vee z_B$ ▷ Special cases computation
6: $inf \leftarrow inf_A \vee inf_B$
7: $sf_A \leftarrow reg_A \;\&\; exp_A$ ▷ Gather scale factors
8: $sf_B \leftarrow reg_B \;\&\; exp_B$
9: $frac_{mult} \leftarrow frac_A \times frac_B$ ▷ Fractions multiplication
10: $ovf_m \leftarrow frac_{mult}[MSB]$ ▷ Adjust for overflow
11: $norm_{frac} \leftarrow ovf_m \;?\; \texttt{'0'} \;\&\; frac_{mult} : frac_{mult} \;\&\; \texttt{'0'}$ ▷ Normalize fraction
12: $sf_{mult} \leftarrow (sf_A[MSB] \;\&\; sf_A) + (sf_B[MSB] \;\&\; sf_B) + ovf_m$ ▷ Add scaling factors
13: $sf_{sign} \leftarrow sf_{mult}[MSB]$ ▷ Get regime's sign
14: $nzero \leftarrow \bigvee frac_{mult}$
15: $exp \leftarrow sf_{mult}[es-1:0]$ ▷ Unpack scaling factors
16: $reg_{tmp} \leftarrow sf_{mult}[MSB-2:es]$
17: $reg \leftarrow sf_{sign} \;?\; -reg_{tmp} : reg_{tmp}$ ▷ Get regime's absolute value
18: $ovf_{reg} \leftarrow reg[MSB]$ ▷ Check for regime overflow
19: $reg_f \leftarrow ovf_{reg} \;?\; \texttt{'0'} \;\&\; \{\lceil \log_2(N) \rceil \{\texttt{'1'}\}\} : reg$
20: $ovf_{regf} \leftarrow \bigwedge reg_f[MSB-2:0]$
21: $exp_f \leftarrow (ovf_{reg} \vee ovf_{regf} \vee \neg nzero) \;?\; \{es\{\texttt{'0'}\}\} : exp$
22: $tmp1 \leftarrow nzero \;\&\; \texttt{'0'} \;\&\; exp_f \;\&\; norm_{frac}[MSB-3:0] \;\&\; \{N-1\{\texttt{'0'}\}\}$ ▷ Packing
23: $tmp2 \leftarrow \texttt{'0'} \;\&\; nzero \;\&\; exp_f \;\&\; norm_{frac}[MSB-3:0] \;\&\; \{N-1\{\texttt{'0'}\}\}$
24: $shift_{neg} \leftarrow ovf_{regf} \;?\; reg_f - 2 : reg_f - 1$
25: $shift_{pos} \leftarrow ovf_{regf} \;?\; reg_f - 1 : reg_f$
26: $tmp \leftarrow sf_{sign} \;?\; tmp2 \gg shift_{neg} : tmp1 \gg shift_{pos}$ ▷ Final answer with extra bits
27: $LSB, G, R \leftarrow tmp[MSB-(N-1) : MSB-(N+1)]$ ▷ Unbiased rounding
28: $S \leftarrow \bigvee tmp[MSB-(N+2):0]$
29: $round \leftarrow (ovf_{reg} \vee ovf_{regf}) \;?\; \texttt{'0'} : G \wedge (LSB \vee R \vee S)$
30: $result_{tmp} \leftarrow \texttt{'0'} \;\&\; (tmp[MSB : MSB-(N-1)] + round)$
31: $result \leftarrow inf \;?\; infinity : z \;?\; zero : sign \;?\; -result_{tmp} : result_{tmp}$
32: **return** $result$
33: **end procedure**

one after the other (lines 7–8). This is due to how posit decimal values are computed using regime and exponent.

- The resulting fraction field is the outcome after multiplying the two operands fractions as if they were integer values (line 9). Recall that multiplying two integers of n bits

length results on an integer of $2n$ bits of maximum length. In addition, the decoder module returns fractions with the hidden bit as MSB, so the first two bits of the fractions multiplication do not strictly belong to the fraction field of the result, since they correspond to the multiplication of the hidden bits plus the possible carry bit due to fraction overflow. Therefore, the MSB of the result aids to detect any overflow at fractions multiplication (line 10).

- If fraction overflow occurs, the resulting fraction has to be normalized shifting one bit to the right. In order to avoid losing any bit for rounding, instead of shifting, we just append a `0` bit as MSB, or as LSB if there is no overflow (line 11).

- The resulting scaling factor is obtained by adding both operands scales, plus the possible fraction overflow. The result of adding two bit strings of same size may overflow, and in this case that carry bit indicates the sign of resulting regime, so it is necessary to replicate the MSB of both scaling factors before adding them (lines 12–13).

- Exponent and regime are extracted from the scaling factors addition. Nevertheless, the obtained regime may be negative, so it is more suitable to handle absolute values (lines 15–17).

- Adding two high-magnitude regimes may result in overflow, so in that case the regime is truncated to the maximum possible value and the exponent is set to 0 (lines 18–21).

- Once the resulting fields have been computed and adjusted, they have to be packed in the correct order. To construct the regime correctly, the packed fields have to be right-shifted as a signed integer according to the sign and value of the regime. It is important not to lose any fraction bit to round correctly, so an amount of `0` bits has to be appended to the right (lines 22–26).

- Posits, same as IEEE 754 floats, follow a *round-to-nearest-even* scheme. To perform

a correct unbiased rounding, the LSB, G (guard), R (round) and S (sticky) bits are needed [38] (lines 27–29). The rounded result is finally adjusted according to the sign and exceptions.

Based on the above algorithm, we created a new class in the FloPoCo framework that implements a parameterized posit multiplier. The design process for this new operator using FloPoCo is as follows: Firstly, we write the above algorithm as a FloPoCo new operator – extending the `Operator` virtual class, in C++ language – with n and es as operator parameters. Then, using the command `flopoco <options> <operator specification list>`, FloPoCo will generate a single synthesizable VHDL file [37]. Figure 5.1 illustrates this process for the command `flopoco PositMult N=8 es=1`, with which we obtain the VHDL code for a Posit$\langle 8, 1\rangle$ multiplier, and changing the values on `N` and `es` we can obtain a new multiplier for any other posit configuration.

It is important to mention that, in contrast with the work presented in [31–33] which only provide implementations with a non-zero value for es, we designed a generic template that can be used to automatically generate multipliers for any posit configuration, not only those with $es > 0$; in particular we can generate a multiplier for *posit8*, the same posit configuration used in Chapter 4 for performing CNN inference. Let us emphasize in the relevance of our contribution: the designed template can provide, with only introducing the values of n and es, a new functional unit in a matter of seconds, without the need of rewriting an adapted version of the algorithm for a specific configuration. What is more, it is possible to generate combinational and sequential and even FPGA-customized versions of the multiplier by just changing the options when invoking FloPoCo. This will be shown in Section 5.3.

The verification of the posit multiplier module has been done as follows: The golden solution was obtained in a similar way as in Section 3.3 – with the help of a Mathematica environment the corresponding decimal values of posits are obtained, then computed against all the possible values, and the results are finally rounded back to posits. After that, we

```
addFullComment("Special Cases");
vhdl << tab << declare("nzero") << " <= '0' when Input" << range(N-2, 0)
 << " = 0 else '1';" << endl;
addComment("1 if Input is zero");
vhdl << tab << "z <= Input" << of(N-1) << " NOR nzero;" << endl;
addComment("1 if Input is infinity");
vhdl << tab << "inf <= Input" << of(N-1) << " AND (NOT nzero);" << endl;

addFullComment("Extract Sign bit");
vhdl << tab << declare("my_sign") << " <= Input" << of(N-1) << ";" << endl;
vhdl << tab << "Sign <= my_sign;" << endl;

addFullComment("2's Complement of Input");
vhdl << tab << declare("rep_sign", N-1) << " <= (others => my_sign);" << endl;
vhdl << tab << declare("twos", N-1) << " <= (rep_sign XOR Input"
 << range(N-2,0) << ") + my_sign;" << endl;
vhdl << tab << declare("rc") << " <= twos" << of(N-2) << ";" << endl;
```

(a) Source code in `PositMult.cpp` file.

```
signal nzero : std_logic;
signal my_sign : std_logic;
signal rep_sign : std_logic_vector(6 downto 0);
signal twos : std_logic_vector(6 downto 0);
signal rc : std_logic;
signal rep_rc : std_logic_vector(6 downto 0);
signal inv : std_logic_vector(6 downto 0);
signal zero_var : std_logic;
signal zc : std_logic_vector(2 downto 0);
signal zc_sub : std_logic_vector(2 downto 0);
signal shifted_twos : std_logic_vector(13 downto 0);
signal tmp : std_logic_vector(4 downto 0);
begin
------------------------------ Special Cases -------------------------------
  nzero <= '0' when Input(6 downto 0) = 0 else '1';
  -- 1 if Input is zero
  z <= Input(7) NOR nzero;
  -- 1 if Input is infinity
  inf <= Input(7) AND (NOT nzero);
---------------------------- Sign bit Extraction ---------------------------
  my_sign <= Input(7);
  Sign <= my_sign;
------------------------ 2's Complement of Input ------------------------
  rep_sign <= (others => my_sign);
  twos <= (rep_sign XOR Input(6 downto 0)) + my_sign;
  rc <= twos(6);
```

(b) Generated VHDL code.

Figure 5.1: Generation of synthesizable VHDL from C++ code with FloPoCo.

perform the test bench with Xilinx Vivado Design Suite [39]. It is observed that for every input combination, the results of our implementation exactly matches the solution obtained with Mathematica. Different posit configurations have been successfully tested.

5.3. Synthesis Results

In order to get comparable results from this work, several multipliers are synthesized using Synopsys Design Compiler with a 65 nm target-library [40] and without placing any timing constraint. We measured the delay, area, power and energy of the different multipliers. Not only multiple posit configurations have been synthesized, but also for each configuration three different designs are taken – sequential (or pipelined) design, combinational one and combinational with no hard multipliers nor DSP blocks. These three designs are obtained using the different options that FloPoCo provides for generating the VHDL code. Table 5.1 presents the delay, area, power, and energy of the posit multipliers after the synthesis. In case of sequential designs, the number of stages is indicated between parenthesis next to delay value.

		Posit$\langle n, es\rangle$ configuration				
		$\langle 8,0\rangle$	$\langle 8,1\rangle$	$\langle 8,2\rangle$	$\langle 16,1\rangle$	$\langle 32,2\rangle$
Delay (ns)	Sequential	0.8 (8)	0.79 (8)	0.78 (7)	1.06 (10)	2.3 (15)
	Combinational	3.59	3.52	3.17	6.2	10.34
	Combinational, No hm	3.36	3.23	3.18	6.2	9.6
Area (µm)	Sequential	2799	2745	2481	6898	24299
	Combinational	1488	1483	1415	3865	15459
	Combinational, No hm	1271	1152	1048	3865	21894
Power (µW)	Sequential	397	384.3	313.7	862.1	2269.2
	Combinational	631.3	562.1	428.4	2609.6	12693.6
	Combinational, No hm	612.4	503.9	424	2609.6	13053.3
Energy (pJ)	Sequential	0.317	0.303	0.244	0.913	5.219
	Combinational	2.266	1.978	1.358	16.179	131.251
	Combinational, No hm	2.057	1.627	1.348	16.179	125.311

Table 5.1: Posit multipliers synthesis results.

A first conclusion we can extract is the fact that sequential designs are not optimized

– as we used the FloPoCo automatic-generation tool, it decides the datapath latency and pipelining. For example, 8-bit multipliers require at least 7 stages, which is a lot for this kind of components. As expected, in combinational designs the area and energy consumption are smaller, but the delay increases compared with the corresponding pipelined designs. This area and consume reduction is also expected to occur when comparing combinational designs with and without hard multipliers, but to a lesser extent. However, as Table 5.1 shows, there are few exceptions: Posit$\langle 16, 1\rangle$, which provide the same results for both combinational multipliers, and Posit$\langle 32, 2\rangle$, whose area – and therefore power – increases when not using hard multipliers.

Unfortunately, a fair comparison with the results from [33] cannot be done since the used library differs from the used in this work. However, we can use Xilinx Vivado for implementing the multipliers on a ZedBoard Zynq-7000 SoC – same FPGA model as used in literature – and compare the synthesis results on LUT and DSP utilization. This comparison is shown in Table 5.2.

Datapath	Posit$\langle 16, 1\rangle$		Posit$\langle 32, 2\rangle$	
	Slice LUT	Used DSP	Slice LUT	Used DSP
Literature	**218**	**1**	**572**	**4**
Sequential	321	1	891	2
Combinational	266	1	927	2
Combinational, No hm	266	1	1640	0

Table 5.2: Comparison of posit multipliers synthesis area results.

Unfortunately, the results are not as good as the ones presented in [33]. Nonetheless, recall that we are using an automatic tool for generating the operators, so one cannot expect a highly optimized solution when using this kind of tools. Our design could be improved if, for example, directly coding in VHDL. Note that the number of lookup tables (LUTs) is lower in the combinational than in the sequential design when using 16 bits, unlike using 32 bits – the combinational circuit requires more LUTs than sequential ones, and even more if not using DSP blocks, what was also shown in Table 5.1. Based on the information displayed

on Table 5.2, one may think that in the case of 32 bits, the sequential design is better in terms of area than the combinational one – which is counterintuitive. However, the implementation report shows that sequential 32-bit design uses many other resources: 65 LUTRAMs, 910 FFs and one BUFG. This is now consistent with the previous cell synthesis.

Chapter 6

Conclusions and Future Work

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) has been used for representing floating-point numbers for over 30 years. Nonetheless, the recently introduced posit number system is seen as a direct alternative to the now ubiquitous IEEE standard.

In this dissertation, we investigated the capabilities and limitations of the two arithmetic formats to check if Type III unum (also known as posits) is a suitable drop-in replacement for the current IEEE Standard for Floating-Point Arithmetic.

6.1. Discussion of Results

In this work, we analyzed the performance – in terms of accuracy – of the posit arithmetic and compared it with the IEEE 754 standard along multiple numerical problems. Based on this analysis, we can conclude that posits have higher accuracy, larger dynamic range, and better closure. Furthermore, in some cases they can produce more accurate answers with the same number of bits as floats, or use fewer bits to archive similar answers.

After theoretical and empirical analyses of accuracy of the posit number format, we have investigated about the use of this newer format in the area of DL. With the obtained results we can claim that posits can be as good as floats when performing NNs training and inference under the same conditions. In addition, posits may be suitable for low-power computing and, in particular, for low-precision neural networks. Since all our computations were simulations of the posit arithmetic using software libraries, it will be interesting to get a fully functional

posit architecture to perform all these computations and compare with the simulations in terms of accuracy and time-consumption. As such a first posit-based architecture has not been designed yet, this is a key area of research.

To conclude this work, and as one of the intentions was to provide a practical approach, a design for a posit arithmetic multiplier have been proposed and implemented. In fact, we created a new class in the FloPoCo framework that can generate VHDL code for any posit configuration, including those with $es = 0$ that are not allowed in previous works. Also some clarifications of the algorithm for low-level multiplier has been done.

Posits have shown to be a good alternative for the IEEE Standard for Floating-Point Arithmetic in many situations. While future is uncertain and nowadays floats are supported in almost every modern computer, there is still a lot of research to do in the area of posit arithmetic.

6.2. Future Work

To finish, we briefly comment the future work and research lines. The multiplier obtained in this work is a starting point for design the rest of components. Recall that the posit decoder is a common module for all the posit operators. In addition, synthesis results showed that the already designed components can be still optimized. A future goal will therefore be to design a fully functional Posit Arithmetic Unit (PAU).

As a result of the limitations found when perforng NN training and inference – due to the lack of deep learning frameworks supporting posit arithmetic – as future work, incorporating this new format in such libraries as Tensorflow or Keras would make possible to run tests on larger architectures and datasets and even computing posits on GPUs, until the firsts functional units on posit arithmetic be available.

Another improvement can be done in the use of posit for DL. As the obtained results show, substituting the addition floating-point operation in CNN inference by the *posit8* equivalent may reduce the accuracy of the network in some cases, but a significant improvement in

power and area is obtained. Designing a CNN entirely in posit format, to take advantage of all the properties mentioned would be an interesting future work line. In addition, as results in [28, 29] show, combining LNS and posit arithmetic in a hybrid architecture for CNNs seems also very promising.

Last but not least, thanks to the properties shown in this dissertation, posit arithmetic might not only be useful for improving numerical precision, but also energy efficiency, and therefore be a key element in the new computing paradigms that have risen in the last years. "*Transprecision computing* is rooted into the key intuition of exploiting approximation in both hardware and software to boost energy efficiency" [41], and this is an area where posits could make great contributions.

Bibliography

[1] D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, Mar. 1991. DOI: 10.1145/103162.103163.

[2] IEEE Computer Society Standards Committee and American National Standards Institute, "IEEE Standard for Binary Floating-Point Arithmetic", *ANSI/IEEE Std 754-1985*, 1985. DOI: 10.1109/ieeestd.1985.82928.

[3] "IEEE Standard for Floating-Point Arithmetic", *IEEE Std 754-2008*, pp. 1–70, 2008. DOI: 10.1109/ieeestd.2008.4610935.

[4] J. L. Gustafson, *The End of Error: Unum Computing*. CRC Press, Feb. 5, 2015, vol. 24, ISBN: 9781482239867.

[5] W. Kahan and J. D. Darcy, "How Java's floating-point hurts everyone everywhere", in *ACM 1998 workshop on Java for High–Performance Network Computing*, Stanford University, 1998, pp. 1–81.

[6] J. L. Gustafson and I. T. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic", *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, Jun. 2017. DOI: 10.14529/jsfi170206.

[7] L. van Dam, "Enabling High Performance Posit Arithmetic Applications Using Hardware Acceleration", Master's thesis, Delft University of Technology, the Netherlands, Sep. 17, 2018, ISBN: 9789461869579.

[8] A. A. D. Barrio, N. Bagherzadeh, and R. Hermida, "Ultra-low-power adder stage design for exascale floating point units", *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3s, 150:1–150:24, Mar. 2014. DOI: 10.1145/2567932.

[9] J. L. Gustafson. (Oct. 10, 2017). Posit Arithmetic, [Online]. Available: https://posithub.org/docs/Posits4.pdf (visited on Mar. 13, 2019).

[10] J. L. Gustafson, "A Radical Approach to Computation with Real Numbers", *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, pp. 38–53, Sep. 2016. DOI: 10.14529/jsfi160203.

[11] Posit Working Group. (Jun. 23, 2018). Posit Standard Documentation, [Online]. Available: https://posithub.org/docs/posit_standard.pdf (visited on Apr. 30, 2019).

[12] R. Munafo. (2018). Survey of Floating-Point Formats, [Online]. Available: http://www.mrob.com/pub/math/floatformats.html (visited on Feb. 9, 2019).

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org, [Online]. Available: https://www.tensorflow.org/.

[14] S. van der Linde, "Posits als vervanging van floating-points: Een vergelijking van Unum Type III Posits met IEEE 754 Floating Points met Mathematica en Python", Bachelor's Thesis, Delft University of Technology, Sep. 26, 2018.

[15] S. H. Leong. (2018). SoftPosit-Python, [Online]. Available: https://gitlab.com/cerlane/SoftPosit-Python (visited on Apr. 20, 2019).

[16] K. Mercado. (2017). PySigmoid, [Online]. Available: https://github.com/mightymercado/PySigmoid (visited on Feb. 10, 2019).

[17] SpeedGo Computing. (2018). NumPy (on top of SoftPosit), [Online]. Available: https://github.com/xman/numpy-posit (visited on Apr. 20, 2019).

[18] H. F. Langroudi, Z. Carmichael, J. L. Gustafson, and D. Kudithipudi, "PositNN: Tapered Precision Deep Learning Inference for the Edge", 2018.

[19] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer", in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, Dec. 2014, pp. 609–622. DOI: 10.1109/micro.2014.58.

[20] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision", *CoRR*, vol. abs/1502.02551, Feb. 9, 2015. arXiv: http://arxiv.org/abs/1502.02551v1 [cs.LG].

[21] M. Courbariaux, Y. Bengio, and J.-P. David, "Low precision arithmetic for deep learning", *CoRR*, vol. abs/1412.7024, Dec. 22, 2014. arXiv: http://arxiv.org/abs/1412.7024v1 [cs.LG].

[22] A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y. J. Kim, H. Shen, and B. Ziv, "Lower numerical precision deep learning inference and training", *Intel White Paper*, 2018.

[23] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Dec. 15, 2017, pp. 2704–2713. arXiv: http://arxiv.org/abs/1712.05877v1 [cs.LG].

[24] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations", *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, Sep. 22, 2016. arXiv: http://arxiv.org/abs/1609.07061v1 [cs.NE].

[25] M. Cococcioni, E. Ruffaldi, and S. Saponara, "Exploiting Posit Arithmetic for Deep Neural Networks in Autonomous Driving Applications", in *2018 International Conference of Electrical and Electronic Technologies for Automotive*, IEEE, Jul. 2018, pp. 1–6. DOI: 10.23919/eeta.2018.8493233.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[27] F. Chollet *et al.* (2015). Keras, [Online]. Available: https://keras.io.

[28] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks", *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, May 2019. DOI: 10.1109/tc.2018.2880742.

[29] J. Johnson, "Rethinking floating point for deep learning", Nov. 1, 2018. arXiv: http://arxiv.org/abs/1811.01721v1 [cs.NA].

[30] S. H. F. Langroudi, T. Pandit, and D. Kudithipudi, "Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit", May 22, 2018. arXiv: http://arxiv.org/abs/1805.08624v1 [cs.CV].

[31] M. K. Jaiswal and H. K.-H. So, "Universal number posit arithmetic generator on FPGA", in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, Mar. 2018. DOI: 10.23919/date.2018.8342187.

[32] M. K. Jaiswal and H. K.-H. So, "Architecture Generator for Type-3 Unum Posit Adder/Subtractor", in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, May 2018. DOI: 10.1109/iscas.2018.8351142.

[33] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized Posit Arithmetic Hardware Generator", in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, IEEE, Oct. 2018. DOI: 10.1109/iccd.2018.00057.

[34] M. S. Kim, A. A. D. Barrio, R. Hermida, and N. Bagherzadeh, "Low-power implementation of Mitchell's approximate logarithmic multiplication for convolutional neural networks", in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jeju: IEEE, Jan. 2018, pp. 617–622. DOI: 10.1109/aspdac.2018.8297391.

[35] J. Chen, Z. Al-Ars, and H. P. Hofstee, "A Matrix-multiply Unit for Posits in Reconfigurable Logic Leveraging (Open)CAPI", in *Proceedings of the Conference for Next Generation Arithmetic*, ser. CoNGA '18, New York, NY, USA: ACM, 2018, 1:1–1:5, ISBN: 978-1-4503-6414-0. DOI: 10.1145/3190339.3190340.

[36] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep Positron: A Deep Neural Network Using the Posit Number System", Dec. 5, 2018. eprint: 1812.01762v2.

[37] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo", *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011. DOI: 10.1109/mdt.2011.44.

[38] I. Koren, *Computer Arithmetic Algorithms*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1993, ISBN: 0-13-151952-2.

[39] T. Feist, "Vivado design suite", *White Paper*, 2012.

[40] A. A. D. Barrio, R. Hermida, and S. Ogrenci-Memik, "A Combined Arithmetic-High-Level Synthesis Solution to Deploy Partial Carry-Save Radix-8 Booth Multipliers in Datapaths", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 2, pp. 742–755, Feb. 2019. DOI: 10.1109/tcsi.2018.2866172.

[41] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomas, D. S. Nikolopoulos, E. Flamand, and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications", in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, Mar. 2018. DOI: 10.23919/date.2018.8342176.

Appendix A

IEEE 754: Floating-Point Arithmetic

In computing, floating-point arithmetic is used to obtain a dynamic range of representable real numbers. A floating-point number is represented using two parts, the *significand* (or mantissa) M and the *exponent* E in some fixed base β (normally two), and therefore, the floating-point number F has the value

$$F = M \cdot \beta^E$$

The term " floating point" makes reference to the fact that the decimal point of a number can "float" in a similar way as in common scientific notation. This dynamic range leads to a non uniform sparse of the represented numbers, and the distance between any two consecutive values increases with the scale. Therefore, floating-point numbers are sparser than fixed-point numbers, which results in a lower precision.

Since it is desirable to represent negative numbers, the floating-point format for n bits consists on a sign bit S, e bits of exponent E and m bits of unsigned fraction M as Figure A.1 shows [38]. The value of such a floating-point number is given by

$$F = (-1)^S \cdot \beta^E \cdot M$$

As one may think, this is not the only possibility for representing negative values, but it is the one used in the current standard. Before the standardization of floating-point numbers, a variety of floating-point representations was used in computer systems. This is a problem

Figure A.1: Floating-point format.

when running scientific programs and loading data on different machines. To solve this and other issues, the IEEE 754 was developed.

A.1. The IEEE Floating-Point Standard

The standard for floating-point arithmetic, IEEE 754, was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE) [2]. An earlier version of this standard was published 23 years later, which is known as IEEE 754-2018 [3], and extends the original one. The standard provides definitions for arithmetic formats, rounding scheme, operations, special numbers representation and exception handling.

A.1.1. Formats

The original standard IEEE 754-1985 defines four formats for floating-point numbers in two groups: the basic, with single and double widths, and their corresponding extended formats. We will focus on the foremost group. Let us mention that the revision in 2008 added the "half precision" (16-bit storage format) and "quad precision" (128-bit format) to the standard, which are very common nowadays.

Single-Precision Format

The width of this format is 32 bits, and the encoding is as shown in Figure A.2. An exponent of length 8 bits is selected in order to have a reasonable range.

For all formats defined in IEEE 754 base 2 is selected. Out of all 256 possible values, two are reserved for special cases – the extreme values, 0 and 255. These special cases are

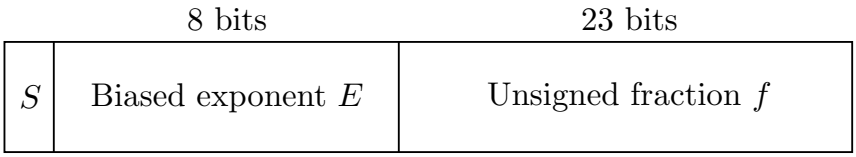


Figure A.2: Single precision floating-point format.

discussed below. Thus, for $1 \leq E \leq 254$, the value of a floating-point numbers is given by

$$F = (-1)^S 2^{E-127}(1.f)_2$$

Notice that the exponent is biased by 127. Hence, the maximum range for the exponent value is $E_{max} = 127$ and the minimum is $E_{min} = -126$. Also the value of f is used as the decimal part of the significand $(1.f)$. Consequently, the larger and smaller positive floating-point numbers in single-precision are

$$F^+_{max} = 2^{254-127} \cdot (2 - 2^{-23}) = (1 - 2^{-24}) \cdot 2^{128}$$

$$F^+_{min} = 2^{1-127} \cdot (1.0) = 2^{-126}$$

Finally, let us focus on the special cases mentioned above. When the exponent value is maximum ($E = 255$), if $f = 0$, the value represented is $(-1)^S \infty$, whether $f \neq 0$ indicates NaN regardless of S. On the other hand, when $E = 0$, if $f = 0$ the value represented is $(-1)^S 0$ (signed zero), and $f \neq 0$ is used to represent denormalized numbers. In computer science, denormalized numbers (also called subnormal numbers) fill the underflow gap around zero in floating-point arithmetic. Any non-zero number smaller than the smallest normal number is "subnormal". The value of denormalized numbers is given by

$$F = (-1)^S 2^{-126}(0.f)_2$$

This way, the smallest representable denormalized number is $F^+_{min} = 2^{-126} \cdot 2^{-23} = 2^{-149}$ instead of 2^{-126}. The use of denormalized numbers is sometimes called gradual underflow or graceful underflow, since instead of losing precision abruptly when discarding all significant

digits, precision is lost slowly this way. Last comment about denormalized numbers is that are not included in all arithmetic units designs following the IEEE standard due to the high cost and complex design that its implementation requires.

Double-Precision Format

The format with the "double" width (64 bits) is used to get a wider range of representable numbers. Therefore, the exponent field increases its length to 11 bits, as can be seen in Figure A.3.

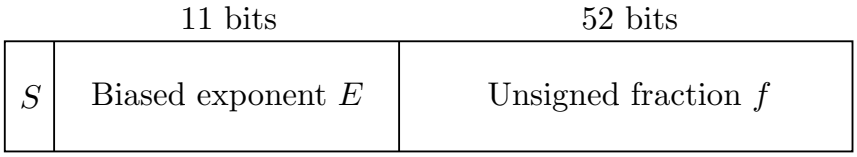


Figure A.3: Double precision floating-point format.

This format is analogous to the previous one. In this case, the bias for the exponent is 1023, and the value for floating point numbers whose exponents E are in the range $1 \leq E \leq 2047$ is given by

$$F = (-1)^S 2^{E-1023} (1.f)_2$$

Table A.1 summarizes both this format and the single-precision format.

Parameter	Single	Double
Word width	32 bits	64 bits
Precision (p)	24 bits	53 bits
Exponent width	8 bits	11 bits
Bias	127	1023
E_{max}	127	1023
E_{min}	-126	-1022

Table A.1: Single and double IEEE precision formats.

A.1.2. Rounding

Rounding consist on fitting a number which is considered as infinitely precise in the specified format. Every operation except binary-decimal conversion must round the produced result. The default rounding mode provided by the standard is round to nearest. If the value to round is in the middle of two representable values, the one with LSB zero should be taken (in IEEE 754-2008 this mode is called *ties to even*, since it includes the alternative *ties away from zero*). However, every result greater or equal $2^{E_{max}}(2 - 2^{-p})$ should be rounded to ∞ with same sign, where E_{max} and p depend on the format (see Table A.1). The standard also provide three user-selectable directed rounding modes: round toward $+\infty$, round toward $-\infty$, and round toward 0.

A.1.3. Operations

Implementations following the standard must provide operations to perform basic arithmetic – add, subtract, multiply, divide and find the remainder – extract the square root, round to integer in floating-point format, convert between different floating point formats, convert between floating-point and integer formats, convert between binary and decimal, and perform comparison – less than, equal, greater than, and unordered.

The IEEE 754-2008 includes new operations such as FMA, explicit conversions, min and max functions and decimal-specific operations among others.

A.1.4. Infinity, NaN and Signed Zero

Infinity Arithmetic

The infinity values are used as limits for real arithmetic with large operands. Therefore, also in the case of floating-point numbers we have that $-\infty <$ (every finite number) $< +\infty$.

Arithmetic on ∞ must be exact, except for invalid operations listed below. When ∞ is created from operands overflow or is an invalid operand, an exception is raised.

Operations with NaNs

Not-A-Number (NaN) is a special code used for indicating an exceptional result occurred. The standard supports two different kinds of NaN: signaling, for uninitialized variables values and arithmetic-like enhancements, and quiet, intended for diagnostic information indicating the source of the NaN.

The Sign Bit

The sign bit of a NaN has no meaning in the standard. The sign of a multiplication/division result is the XOR of both operands' signs. The sign of a sum/difference differs from at most one of the addends' signs, if are different, are compared (negative take 2's complement) and the sign is the same as for the bigger operand. These rules also apply to zero or infinite operands.

If a sum of two opposite signed operands is equal zero, the sign will be + (except when rounding toward $-\infty$, that will be $-$).

Only for `sqrt(-0)` a square root shall have a negative sign (-0).

A.1.5. Exceptions

When an exception is detected, it must be signaled by setting a status flag, taking a trap, or possibly doing both. There is a total of five types of exceptions.

Invalid Operations

The result of an invalid operation will be a quiet NaN when destination has a floating-point format. The invalid operations are

- Any operation with signaling NaN as input
- Addition or subtraction – magnitude subtraction of infinities, such as $(+\infty) + (-\infty)$
- Multiplication – $0 \times \infty$

- Division – $0/0$ or ∞/∞
- Remainder – x `REM` y, where y is zero or x is infinite
- Square root if the operand is less than zero
- Conversion of a binary floating-point number to an integer or decimal format when overflow, infinity, or NaN
- Comparison by way of predicates when the operands are "unordered"

Division by Zero

The operation $x/0$, with x nonzero, will be a correctly signed ∞.

Overflow

Overflow of floating-point numbers occurs when an infinite precision result is so large that should be rounded to a greater value than the format's largest finite number. The result is determined by the rounding mode as follows:

- Round to nearest takes all overflows to ∞ keeping the sign
- Round toward 0 takes all overflows to the format's largest finite number keeping the sign
- Round toward $-\infty$ takes positive overflows to the format's largest finite number, and takes negative overflows to $-\infty$
- Round toward $+\infty$ takes negative overflows to the format's most negative finite number, and takes positive overflows to $+\infty$

Underflow

Underflow may be caused by any of these two events. One is when the result of an operation is a nonzero smaller magnitude that the smallest representable value, i.e., the

result is between $\pm 2^{E_{min}}$, so this tiny value may cause some other exception later. In older designs, all values in this underflow gap were just replaced by zero (which is called *flush to zero*). However, the IEEE 754 introduced denormalized numbers, and in the standard underflow is only signed if there is also a final inexact result or an extraordinary loss of precision in approximation by denormalized numbers. The result of an operation with underflow might be zero, denormalized, or $\pm 2^{E_{min}}$, as the implementor choose.

Inexact

If the rounded result of an operation is not exact or if it overflows (but no trap occurs), an inexact exception must be signed.

Another clause related to exception handling – which we will not detail here – is included in IEEE 754. The revision of the standard incorporates three more clauses corresponding to recommended operations, expression evaluation and reproducibility.