Unum Type-IV: A Floating-Point Unit with Dynamically Varying Exponent and Mantissa Sizes

Micaela Moraes Serôdio

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. José João Henriques Teixeira de Sousa
Supervisor: Prof. Horácio Cláudio de Campos Neto

Examination Committee

Chairperson: Prof. Francisco André Corrêa Alegria
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Mário Pereira Véstias

September 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code od Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

This document marks the end of my 5-year journey as a university student and, I would like to start by thanking the Instituto Superior Técnico and the Federal University of Santa Catarina. These two universities gave me the necessary knowledge, material and conditions to develop this project, allowing me to conclude my academic career with success. I would also like to thank all my colleagues and teachers who made this whole journey stimulating and with a lot of personal, academic and social learning. In particular, I would like to thank Professor José Teixeira de Sousa for his knowledge, advice, for reviewing my thesis and also for creating in me a great interest in computer electronics and hardware, to Professor Horácio Neto for the valuable insights and, also, to João Dias Lopes for all the help in the development of this dissertation.

Finally, I would like to give special thanks to my family and friends for all the support and help they gave me throughout this process. This achievement would not be possible without their support.

Abstract

The main objective of this dissertation is to create a new floating-point format, called Unum-IV, that replaces the unary numeral system used in the Regime bits of the Unum Type-III format, aka *Posit* numbers, with a regular binary system. The Unum formats (types I, II and III) introduced the concept of tapered precision: numbers having an absolute value close to 1 are given more significand bits and fewer exponent bits. This way, it is possible to extraordinarily increase the dynamic range or accuracy, depending on the application.

The new format recovers and extends Unum Type-I ideas but, like Unum-III, it is meant to be hardware fiendly, is hence named Unum Type-IV. The main ideas of the new Unum-IV format are to use a dynamically variably-sized significand and exponent with a 1's complement representation of the exponent, and a 2's complement representation of the significand. The Unum-IV format uses one hidden bit both in the exponent and significand representations to prevent redundancy. However, unlike other formats, including Posits, that use a fixed hidden 1 bit only in the significand, Unum-IV uses dynamic but simply computed hidden bits for both the significand and the exponent.

This proposal intends to be a suitable drop-in replacement for the IEEE 754 Floating-Point Standard, using a dynamically sized significand and exponent, which allows the new format to fulfil the precision or dynamic range needs of applications, using fewer hardware resources. This format is aimed at small memory footprint and low energy consumption applications, such as those that can be found in the Internet of Things (IoT). In fact, in most practical cases, Unum-IV can replace IEEE 754 double precison numbers, which means half the memory footprint and 30% less hardware resources.

In this dissertation work, a Unum-IV Floating-Point Unit (FPU) with addition, subtraction, multiplication and division operations is implemented in software and hardware, in the C and Verilog languages, respectively. To study it, the FPU is attached as a peripheral to an open-source RISC-V processor, and the K-Nearest Neighbours (KNN) algorithm, a non-parametric machine learning algorithm, is used as a proof of concept for comparing the Unum-IV and the IEEE Standard 754 Floating-Point formats. The metrics used to compare the different number systems are the precision, dynamic range, resolution, Units of Least Precision (ULP), and integrated circuit implementation results: silicon area, power consumption, operation clock frequency. The percentage of accurate classifications in the KNN application, using the IEEE-754 64-bit double precision floats as reference, is also studied.

Keywords: Computer Arithmetic, Unum Number, Posits, Floating-Point Unit, High Precision Arithmetic, Approximate Computing

Resumo

O objetivo principal desta dissertação é a criação de um novo formato de vírgula flutuante, o Unum-IV, que substitui o sistema numérico unário utilizado nos bits de regime do formato Unum Type-III, também conhecido como Posits, com um sistema binário regular. Os formatos Unum (tipos I, II e III) introduziram o conceito de precisão cónica, isto é, números com um valor absoluto próximo de 1 recebem mais bits de mantissa e menos bits de expoente. Desta forma, é possível aumentar extraordinariamente a gama dinâmica e/ou precisão, dependendo do contexto em que é usado.

O novo formato recupera e estende ideias do formato Unum-I. No entanto, tal como o formato Unum-III, este novo formato Unum-IV é voltado para versão otimizada de termos da utilização de hardware. As principais ideias deste novo formato são a utilização de um significand e expoente de tamanho dinamicamente varíavel com uma representação de complemento para 1 do expoente, e uma representação de complemento para 1 do significando. O formato Unum-IV contém um bit ”escondido” nas representações de expoente e significando para não existir redundância nas representações. No entanto, contrariamente aos restantes formatos, incluindo os Posits, que usam um bit implícito fixo a 1 apenas no significando, o Unum-IV utiliza bits implícitos de forma dinâmica mas simples de calcular tanto para o expoente como para o significando.

Esta proposta pretende ser um substítuto adequado para o Padrão de Vírugla Flutuante IEEE 754, através de um significando e exponente de tamanho dinâmico, o que permite que o novo formato cumora as necessidades de gama dinâmica ou precisão das aplicações, utilzando menos recursos de hardware. O Unum-IV é um formato voltado para aplicações de baixo consumo energético e baixa utilização de memória, como IoT. Na verdade, na maioria dos casos práticos, o Unum-IV pode substituir os números IEEE 754 de precisão dupla, o que se traduz numa redução de memória utlizada, bem como uma redução 30% dos recursos de hardware.

Nesta dissertação, foi implementada em software e hardware uma Unidade de Ponto Flutuante (UPF) com adição, subtração, multiplicação e divisão de números no formato Unum-IV em C e Verilog, respectivamente. Esta UPF foi anexada como periférico a um processador RISC-V de código aberto e um algoritmo de classificação não paramétrico (KNN) foi utilizado como prova de conceito para comparar os resultados das operações com Unum-IV e com IEEE 754 Standard. As métricas usadas para comparar os diferentes sistemas numéricos são a precisão, gama dinâmica, resolução, ULP (”Units of Least Precision”), resultados de implementação ASIC, como a área ocupada, consumo energético e frequência do relógio. Os resultados obtidos com os dois modelos na aplicação KNN são comparados em termos da percentagem de classificações corretas, sendo que a referência é dada pelo formato de 64 bits IEEE 754.

Palavras-chave: Aritmética Computacional, Computação Unum, Unidade de Ponto Flutuante, Aritmética de Alta Precisão

Contents

Acknowledgments . . . v
Abstract . . . vii
Resumo . . . ix
List of Tables . . . xiii
Listings . . . xiv
List of Figures . . . xv
List of Acronyms . . . xix

1 Introduction **1**
1.1 Topic Overview . . . 1
1.2 Motivation . . . 2
1.3 Objectives . . . 3
1.4 Dissertation Outline . . . 3

2 Background **5**
2.1 IEEE Standard 754 Floating-Point . . . 5
2.1.1 Single-Precision . . . 5
2.1.2 Other IEEE 754 Standard Formats and ”Bfloat16” Format . . . 7
2.2 Unum . . . 8
2.2.1 Unum Type-I . . . 8
2.2.2 Unum Type-II . . . 9
2.2.3 Unum Type-III: Posits . . . 9

3 Unum Type-IV **13**
3.1 Unum Type-IV Generic Format . . . 13
3.2 Field Extractions . . . 16
3.2.1 Exponent Extraction . . . 16
3.2.2 Significand Extraction . . . 17
3.2.3 Examples . . . 17
3.3 Features . . . 18
3.3.1 Special Cases . . . 18
3.3.2 Exceptions . . . 18

3.3.3 Rounding . . . 18
3.3.4 Dynamic Range and Precision . . . 19
3.4 Examples . . . 22
3.4.1 Unum-IV to Decimal Conversion . . . 22
3.4.2 Unum-IV<4,1> Enconding . . . 24

4 Hardware Implementation 25
4.1 Unum Type-IV Floating-Point Unit . . . 26
4.2 Functional Units . . . 28
4.2.1 Unpack Unit . . . 28
4.2.2 Processing Units . . . 30
4.2.3 Pack Unit . . . 35
4.2.4 Auxiliary Components . . . 38
4.2.5 Functional Units Pipeline Stages . . . 40

5 Evaluating and Comparing Unum Type-IV to Other Formats 41
5.1 Comparison Metrics . . . 41
5.1.1 Precision Bits . . . 41
5.1.2 Dynamic Range . . . 42
5.1.3 Hardware Resources . . . 42
5.1.4 Decimals of Accuracy . . . 42
5.1.5 Units of Least Precision . . . 42
5.2 Comparing Unum Type-IV Features with Other Formats . . . 43
5.3 Comparing Unum Type-IV Dynamic Range with Other Formats . . . 45
5.4 Comparing Unum Type-IV Precision with Other Formats . . . 46
5.4.1 Unum-IV<8,2> vs. Quarter-Precision IEEE-Style floats . . . 49
5.4.2 Unum-IV<8,2> vs. Posit<8,1> vs. Posit<8,0> . . . 51
5.5 Comparing Unum Type-IV Hardware Resources with Other Formats . . . 52
5.5.1 IEEE 754 and Unum Type-IV Comparison . . . 52
5.5.2 Posits and Unum Type-IV Comparison . . . 55
5.6 Comparison Summary . . . 56

6 Proof of Concept: KNN Application 58
6.1 Algorithm . . . 59
6.2 Implementation . . . 60
6.3 Experimental Results . . . 64
6.3.1 Experiment 1 . . . 64
6.3.2 Experiment 2 . . . 65

7 Conclusions **67**

7.1 Achievements 68

7.2 Future Work 69

Bibliography **71**

List of Tables

2.1 IEEE 754 Standard and ”Bfloat16” Format Features. 8

3.1 Exponent and Significand Extraction for DATA_W=32 and EXP_SZ_W=4. 18
3.2 ”Round to the nearest, ties even” Mode. 19
3.3 Unum Type-IV Dynamic Ranges. 20
3.4 Unum Type IV Precision. 21
3.5 Unum-IV<4,1>. 24

4.1 FPU Interface. 27
4.2 Specification for the Operation Selection. 30
4.3 Functional Units Pipeline Stages 40

5.1 Qualitative Comparison Between IEEE 754 Standard and Unum Type-IV. 43
5.2 Resolution for Different Unum-IV, Posits and IEEE754 Format Configurations. 49
5.3 ASIC Implementation Results. 53
5.4 Silicon Area of the Different Modules using Unum-IV<32,4>. 54
5.5 FPGA Results Comparison Between Different Unum-IV and Posits Configurations. 55
5.6 Silicon Area Comparisons Between Different Unum-IV and Posits Configurations. 55
5.7 Binary32, Bfloat16, Posit<16,3> and Unum-IV<16,3> Comparisons. 56

6.1 Parameters used in the KNN Clustering Application. 64

Listings

3.1 Pseudo-code of the Exponent Extraction Algorithm. 17
3.2 Pseudo-code of the Significand Extraction Algorithm. 17
4.1 Verilog Code to Instantiate the Parameterizable Unum-IV Floating Point Unit. 27
4.2 Rounding Bits Assignment. 37
6.1 Unum-IV and KNN Header File. 60
6.2 Double Random Generator. 62
6.3 Data Structures. 62
6.4 Square Distance Functions. 63
6.5 Insertion in Ordered List. 63
6.6 Classification step code. 64

List of Figures

2.1 IEEE 754 Single-Precision Representation Format. 5
2.2 Unum Type-I Representation Format. 8
2.3 Visual Representation of the Unum Type-II Projective Real Number Line [15]. 10
2.4 Unum Type-III Representation Format. 10
2.5 Unum Type-I,II and III Advantages and Disadvantages. 11

3.1 Generic Unum-IV Representation Format. 14
3.2 Unum-IV<16,2> and Unum-IV<16,3> Precision Bits vs Binade (Exponent). 21
3.3 Unum-IV<32,3> and Unum-IV<32,4> Precision Bits vs Binade (Exponent). 21
3.4 Example 1 of an Unum-IV<32,4> Bit String. 23
3.5 Example 2 of an Unum-IV<32,4> Bit String. 23
3.6 Example 3 of an Unum-IV<32,4> Bit String. 23

4.1 IoB-SoC Block Diagram with the Unum-IV Module Attached as a Peripheral. 25
4.2 Unum-IV FPU. 26
4.3 Unum-IV FPU Datapath. 28
4.4 Unum-IV Unpack Stage Block Diagram. 29
4.5 Unum-IV Addition/Subtraction Unit. 31
4.6 Unum-IV Multiplication Unit. 33
4.7 Unum-IV Division Unit. 34
4.8 Unum-IV Pack Unit. 36
4.9 Unum-IV Exponent Difference Flow Diagram. 38

5.1 Dynamic Range for Different Unum-IV and IEEE754 Format Configurations. 45
5.2 Exponent vs Significand Bits Comparison Between Unum-IV<16,3> & 16-bit IEEE 754 Format. 46
5.3 Exponent vs Significand Bits Comparison Between Unum-IV<32,3> & 32-bit IEEE 754 Format. 47
5.4 Exponent vs Significand Bits Comparison Between Unum-IV<32,4> & 32-bit IEEE 754 Format. 47
5.5 Exponent vs Significand Bits Comparison Between Unum-IV<64,4> & 64-bit IEEE 754 Format. 48

5.6 Exponent vs Significand Bits Comparison Between Unum-IV<128,4> & 128-bit IEEE 754 Format. 48
5.7 Decimals of Accuracy Comparison Between Unum-IV<8,2> & 8-bit Floats. 50
5.8 Decimals of Accuracy Comparison Between Unum-IV<8,2> & Posit<8,0> (LEFT).Decimals of Accuracy Comparison Between Unum-IV<8,2> & Posit<8,1> (RIGHT). 51
5.9 ULP Comparison Between Unum-IV<8,2> & 8-bit Floats. 52

6.1 KNN Application. 59
6.2 Accuracy of Classification for Unum-IV<32,4> and 32-bit Floats. 65
6.3 Accuracy of Classification for Unum-IV<32,4> and 64-bit Floats. 66

List of Acronyms

FPU Floating-Point Unit

FPGA Filed Programmable Gate Array

RISC Reduced Instruction Set Computer

SoC System on Chip

FU Functional Unit

ALU Arithmetic and Logic Unit

FP Floating Point

SP Single Precision

DP Double Precision

NaN Not a Number

AI Artificial Intelligence

ML Machine Lerning

IoT Internet of Things

qNaN Quiet Not a Number

sNaN signaling Not a Number

HPC High Performace Computing

BD Big Data

MSB Most Significant Bit

IEEE Institute of Electrical and Electronics Engineers

DL Deep Learning

ASIC Application-Specific Integrated Circuit

ULP Units of Least Precision

Chapter 1

Introduction

1.1 Topic Overview

We live in an era where the trade-off between performance, cost, resources, and power consumption significantly impacts the computer science area. The increase in the complexity of the algorithms due to Big Data (BD) analysis, the need for High-Performance Computing (HPC) and the never-ending need for energy-efficient computing in fields of Machine Learning (ML), Artificial Intelligence (AI) and the Internet of Things (IoT) are creating a new computing paradigm. The majority of numerical computation algorithms require the capacity to perform arithmetic operations and manipulation of real numbers. In digital electronics, the most extensively adopted approximation of the real numbers is given by the floating-point formats.

Throughout our history, some disasters happened due to floating-point issues, usually because of rounding errors [1], such as the Patriot Missile Failure on February 25, 1991, where an Americal Patriot Missile battery in Dharan, Suadi Arabia failed to intercept an incoming Iraqi Scud missile. That happened due to a rounding error in the time calculation, which costed the life of 38 people.

Another disaster was the explosion of the Ariana-5 rocket launched by the European Space Agency in 1996. The rocket exploded just forty seconds after the lift-off due to a software error in the inertial reference system. One more example of a floating-point error disaster was the Sleipner Oil Platform that collapsed to the ocean floor due to a floating-point error in the structural analysis, costing a nearly 1 billion dollar loss. There were other famous disasters in different areas caused by issues of the floating-point format design, like the overflow, underflow and rounding error problems [2].

During the 1960s and the 1970s, there was no ubiquity in the floating-point format, so each computer manufacture developed its floating-point system, resulting in floating-point inconsistency across platforms. In 1985, the IEEE Standard for Floating-point Arithmetic (IEEE 754) [3] was established by the Institute of Electrical and Electronics Engineers (IEEE), and it is, nowadays, the most common representation of real numbers on computers, including Intel-based PC's, Macs and most Unix platforms. This format has been reviewed and replaced two times, in 2008 [4] and 2019 [5] but there were only minor changes between them to maintain compatibility in existing implementations.

However, some drawbacks have been identified in the IEEE 754 Standard [6, 7, 8], such as:

- Overflow and Underflow: overflowing to $-\infty$ or $+\infty$ and underflowing to 0, increases the relative error by an infinite factor and leads to sign information loss, respectively.
- No Gradual Overflow and Fixed-Accuracy: accuracy is flat across a vast range, then ”falls off a cliff”.
- Wasted bit-patterns: there are too many NaN representations and two bit-patterns to represent 0, the ”negative zero” (0^-) and the ”positive zero” (0^+).
- There is no guarantee of identical results across systems.
- Exponents usually take too many bits.
- Equality verification test between two floating-point is complex due to the presence of redundant representations.

Throughout the years, different number systems and techniques have been proposed to overcome these challenges. In [9], Morris suggested a ”tapered” system to solve the fixed accuracy problem of the floating-points [10].

In 2013, John L. Gustafson introduced a new binary and arithmetic way of representing real numbers, a number system called ”Universal Numbers” [6, 11, 12, 13]. They have so far three different types of representation. Type-I is a super-set of the IEEE 754 Standard floating-point format, and it was introduced in [6]. This format uses a variable-length storage format for the exponent and fraction fields and a ”u-bit” at the end of the fraction that indicates if a ”real” number (u=0) is an exact float or lies in the open interval between two consecutive exact (u=1). The Type -II [14] enables a clean mathematical design based on the *projective reals* and relies on lookup tables. It is a direct map of signed integers to the projective real number line. The last version introduced in 2017 is the Type-III format, also known as Posits [15, 16]. Posits are a hardware-friendly version of Unums [17, 18], with all the advantages from the previous types, but where the problems found in Type-I due to the variable sizes are solved. Posits have a different format than IEEE 754 floats, consisting of 4 fields: the sign, regime, exponent, and fraction field.

Recent studies on Posits suggest it solves many of the drawbacks of the IEEE 754 Standard for floating-point arithmetic [8, 19, 20, 21], and are especially useful for Deep Learning (DL) applications [22, 23, 24].

Over the years, other notable formats have been introduced for DL algorithms [25, 26], for example, the ”Bfloat16” format [27, 28, 29, 30] introduced by a Google research group to replace the 32-bit IEEE 754 floating-point format in energy-efficient applications.

1.2 Motivation

Nowadays, even though many shortcomings have been pointed out upon the IEEE 754 Standard, it is still the most commonly implemented in microcontrollers and general-purpose microprocessors to

perform floating-point arithmetic. However, the trade-off between the dynamic range and the precision of the floating-point formats can be explored to create a new number system that avoids those shortcomings. The Unum system introduced the concept of tapered precision, which means that numbers having an absolute value close to 1 are given more mantissa bits and fewer exponent bits, whereas the other numbers trade-off mantissa bits with exponent bits. This way is possible to extraordinarily increase the dynamic range and accuracy at the said ranges.

The Unum-III number system (Posits) looks very promising as an IEEE 754 replacement, but, in this thesis, we show that it can still be improved to provide more accuracy and dynamic range for the same number of bits to represent the numbers. More specifically, the unary representation of the "regime" bits of the Posits limits its accuracy or dynamic range and beg for optimisation.

Hence, the principal motivation of this dissertation is to design a new floating-point format suitable for replacing Posits, improving its accuracy and dynamic range without requiring more hardware resources.

1.3 Objectives

The main objective of this dissertation is to come up with a number system that can overcome the shortcomings in accuracy and dynamic range identified in the Unum Type-III format.

This proposal intends to be a suitable replacement for the IEEE 754 Standard for floating-point arithmetic in low energy consumption applications by using an exponent and mantissa of dynamic size.

In order to achieve this objective, the state-of-the-art of the IEEE 754 Standard and the different Unum floating-point formats and arithmetic is studied in detail, a new format, Unum-IV, is proposed, and software and hardware models are developed.

The hardware model is a parameterisable Unum-IV Floating-Point Unit (FPU) with four basic operations (addition, subtraction, division and multiplication) is implemented.

To verify the new FPU, the FPU is attached as a peripheral to a RISC-V System on Chip produced by IOBundle, Lda, a Lisbon-based computer architecture company, and simulated. Additionally, the new FPU and an IEEE 754 FPU has been synthesised in an Integrated Circuit flow using the UMC 130nm silicon technology. The synthesis results in terms of the clock frequency, silicon area and power consumption are compared to the IEEE 754 FPU and the published results of a Posits FPU.

To verify the advantages in terms of accuracy and dynamic range, the Unum-IV FPU software model is and tested against the IEEE 754, already implemented in the C Standard library, using a K-Nearest Neighbours (KNN) application [31] as proof of the concept.

1.4 Dissertation Outline

This dissertation is composed of seven chapters, including the present one. In the second chapter, the IEEE 754 Standard formats and the Unum formats are presented. In the third chapter, the new Unum Type-IV format is introduced and detailed. In the fourth chapter, the hardware implementation of a parameterised floating-point unit using the Unum Type-IV number system is fully described. In the fifth

chapter, the IEEE 754 Standard is compared against the Unum Type-IV format. In the sixth chapter, a KNN application is implemented and tested using different number systems, and the experimental results are discussed. Finally, in the seventh chapter, the conclusions are presented, the achievements are pointed out, and the directions of the future work are outlined.

Chapter 2

Background

This chapter describes the existing Floating-Point (FP) Unum formats (Unum Type-I, Unum Type-II and Unum Type-III), the IEEE 754 Standard and the ”Bfloat16” format.

2.1 IEEE Standard 754 Floating-Point

The IEEE 754 Standard, established in 1985 by the IEEE [3, 4, 5], is a standard that specifies formats and methods for floating-point arithmetic in computer programming environments. The format comprises three fields, sign, exponent and significand, exception conditions, rounding modes and their default handling. The implementation of the FP system may be performed in software, hardware or a combination of both. The IEEE 754 standard allows for FP computations to have the same results. It means that the standard defines a family of commercially feasible ways for systems to perform floating-point arithmetics.

The standard specifies formats for binary and decimal FP data, operations, conversions between FP formats and integer formats, exceptions and how to handle them. The formats approached in more detail in this chapter are the Single-Precision (SP) and the Double-Precision (DP) formats, because they are compared and tested against the new number system proposed in this dissertation, using a KNN application.

2.1.1 Single-Precision

The single-precision floating-point is one of the formats belonging to the IEEE 754 family. It has a width of 32 bits and is encoded as shown in Figure 2.1.

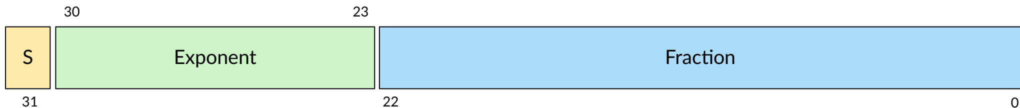


Figure 2.1: IEEE 754 Single-Precision Representation Format.

As represented in Figure 2.1, this format has three representation fields:

- S: the sign bit that indicates the sign of the significand of the signed number representation.
- Exponent: a biased 8-bit unsigned integer with a range between 0 and 255. The bias is 127, meaning that the actual 0 of the exponent is represented by an exponent biased value of 127. The 0 and 255 values of the biased exponent are reserved for ”special” numbers explained in detail in Section 2.1.1. The actual exponent range is [-126:127].
- Fraction: the 23 explicit bits of the significand. The significand has a 23-bit fraction on the right side of the binary point plus an implicit leading bit on the left side. That hidden bit is always 1, unless the biased exponent is 0. As a result, the significand is in the format *0.fraction* if the exponent is equal to -126 and in the *1.fraction* format, otherwise.

The formula to convert the single-precision IEEE 754 FP format into the decimal representation is given by the following equation

$$X = \begin{cases} (-1)^{\text{S}} \times 2^{\text{Exponent}-127} \times 1.\text{fraction}, & \text{if Exponent} \neq 0 \\ (-1)^{\text{S}} \times 2^{-126} \times 0.\text{fraction}, & \text{otherwise} \end{cases} \tag{2.1}$$

Special Cases

There are four types of ”special” representation patterns: the ”subnormal” numbers, the ”Not a Number” patterns (NaNs), the $+\infty$ and the $-\infty$.

The 32-bit combinations where the biased exponent field is encoded with all zeros are the”subnormal” numbers. These numbers fill the underflow gap around zero. In those cases, the implicit leading bit is 0 instead of 1, and even though the biased exponent is 0, the actual exponent is -126 as shown in Equation 2.1; it is interpreted with the value of the smallest allowed exponent. These denormalized numbers are used to create a gradual underflow.

The other special representations are bit patterns where the exponent is encoded with all ones. If fraction=0, there are 2 possible combinations: S=0, which is used to represent $+\infty$, and S=1 which represents $-\infty$. The other possible combinations with fraction$\neq 0$ are used to represent the NaNs, interpreted as an undefined value. For example, dividing zero by zero results in an ”undefined real” number and, therefore, computed as a NaN in computing systems.

The NaNs are separated into two types: the quiet NaN (qNaN) and the signalling (sNaN). The difference between them is that the qNaNs are used to propagate errors resulting from invalid operations, and the sNaNs signal an invalid operation exception. NaNs are a symbolic entity encoded in FP format, produced by the following operations: $\infty - \infty$, $-\infty + \infty$, $0 \times \infty$, $0/0$ and ∞/∞. The sNaNs correspond to a bit pattern of the fraction where the most significant bit is set to zero, and at least one of the remaining bits set to one. For the qNaNs, the fraction needs to have the most significant bit set to 1.

Rounding

The IEEE 754 provides four rounding modes: the round to nearest or ties to even; round toward $+\infty$; round towards $-\infty$; round towards 0. The default rounding mode provided by the standard is the first one, round to nearest.

Operations

The operations required by the IEEE 754 standard are the addition, subtraction, division, square root, fused multiply-add, remainder calculation, conversions between the IEEE 754 supported formats, comparisons, sign manipulation, scaling and quantizing, miscellaneous operations and total-ordering operations.

Exceptions

In the IEEE 754 standard, there are five types of exceptions: invalid operations, divide by zero, overflow, underflow and inexact results. When any exception occurs, it must be signalled by a status flag and issue a trap. The invalid operation occurs when the operation result is a qNaN, for example, an operation that has a NaN as an input. The divide by zero operation must return a signed ∞ if the number to be divided is different from 0. Overflow occurs when the number is larger than the maximum possible value of the representation format. On the other hand, underflow occurs when the result of an operation is a number that has a smaller absolute value than the smallest representable value of the format used. Finally, the inexact exception happens when the rounded result of an operation is not exact and, in that case, no trap occurs.

2.1.2 Other IEEE 754 Standard Formats and "Bfloat16" Format

The IEEE 754 Standard [5] includes four formats: half-precision (16 bits), single-precision (32 bits), double-precision (64 bits) and quadruple-precision (128 bits). The different formats features and behaviour is similar, with a difference in the number of bits of the format, fields size and bias.

The exponent field has 5 bits for the 16-bit, 8 bits for the 32-bit, 11 bits for the 64-bit and 15 for the 128-bit formats. The fraction field has 10, 23, 53 and 112 bits, and the bias is equal to 15, 127, 1023, 16383, respectively.

Table 2.1 summarises all the IEEE 754 Standard Floating-Point Formats plus the "Bfloat16" format. "Bfloat16" [30] is a 16-bit format, with one sign bit, eight exponent bits and seven fraction bits. The 8-bit exponent allows this format to have an identical dynamic range compared with the 32-bit float, so it is a truncated version of the IEEE 754 single-precision format. The main idea was to replace the 32-bit IEEE-754 with a format that occupies half of the computer memory, making it faster, simpler, and cheaper, with the downside of losing precision. However, as stated before, research show [32, 33, 34, 27] that many ML models tolerate this trade-off between speed and precision, as they increase the performance using less memory and reach the same results without degradation.

Table 2.1: IEEE 754 Standard and ”Bfloat16” Format Features.

Format	Width [Bits]	Exponent [Bits]	Exponent Bias	Significand [Bits]	Dynamic Range [Decades]
IEEE 754 Half-Precision	16	5	15	10 explicit + 1 implicit	12.04
Google Bfloat16	16	8	127	7 explicit + 1 implicit	83.47
IEEE 754 Single-Precision	32	8	127	23 explicit + 1 implicit	83.39
IEEE 754 Double-Precision	64	11	1023	52 explicit + 1 implicit	631.56
IEEE 754 Quadruple-Precision	128	15	16383	112 explicit + 1 implicit	9882.51

2.2 Unum

The ”Universal Numbers”, or Unums, introduced by John L. Gustafson in 2013 are a binary and arithmetic representation format for real numbers. Unums have so far three different types of representation. Type-I is a superset of the IEEE 754 Standard floating-point format, introduced in [6]. Type-II enables a clean mathematical design based on the *projective reals* and relies on lookup tables. This format was explained and detailed by Gustafson in an 2016 interview [14]. It is a direct map of signed integers to the projective ”real” number line. The last version, introduced in 2017 is the Type-III, aka, Posits [15]. Type III is a hardware-friendly version with all the advantages from the previous types, and where the problems found in Type I due to the variable sizes are solved.

The main idea of John L. Gustafson was to create a new number system for computers that could be a suitable replacement to the IEEE 754 format. This implies a system that could give more accurate results using fewer or the same number of bits, respects all the algebraic laws that IEEE 754 breaks, and saves memory resources, energy and power consumption.

2.2.1 Unum Type-I

As described in [15], Unum Type-I is a superset of IEEE 754. The representation format is as shown in Figure 2.2. This format uses a ”u-bit” (U) to indicate if the number is an exact float (U=0) or if the number lies in the open interval between two consecutive floats (U=1).

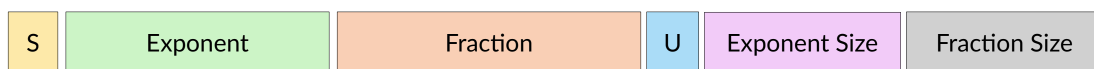


Figure 2.2: Unum Type-I Representation Format.

The sign (S) bit indicates if the number is a negative or a positive value. If the sign bit is 1 the number is negative, otherwise it is positive or zero. The exponent and fraction fields also have a similar definition as in the IEEE 754 format.

However, in the IEEE 754 format, the fields have a fixed size, whereas in Unum Type I, they have variable size, depending on the exponent size and significand size fields. They can go from 1 bit to the maximum set by the user, meaning that they can vary in size to save storage and bandwidth.

This format is a compact way to express interval arithmetic. However, it requires extra management

at the hardware implementation due to the variable width of the exponent and significand field. It also can avoid rounding errors, underflow and overflow because of the interval arithmetic approach.

However, there are disadvantages of the Unum Type-I format inherited from the IEEE 754, in particular, the existence of redundant representations, later solved by Unum Type-III as will be shown below. The Unum Type-I format is explained in more detail in [6].

2.2.2 Unum Type-II

Unum Type-II, introduced in 2016, intended to solve some of the drawbacks of Unum Type-I, such as the existence of redundant representations. While Unum Type-I is a superset of IEEE 754, Unum Type-II is a complete redesign, losing compatibility with IEEE 754.

Unum Type-II format enables a clean mathematical design based on projective reals. They are a direct-map of signed integers onto the "real" projective line, relying on look-up tables. This format has many ideal mathematical properties based on the projective reals. As John L. Gustafson explained in an interview [14], Unum Type-II shares about 80 % of the mathematical advantages of Unum Type-I, such as the ability to avoid overflows, underflows and rounding errors. Unum Type-II is a configurable accuracy, fast and simple format, which allows the user to design a custom number system for a particular workload, particularly interesting for the deep learning community.

Figure 2.3 shows the visual representation of a 5-bit Unum-II projective "real" number line. The upper right quadrant has an ordered set of real numbers (x_i), whereas the upper left quadrant has the symmetric of those values ($-x_i$). As for the lower quadrants, they hold the reciprocal values of the upper quadrants. This geometry of the projective reals is what gives the format many ideal mathematical properties.

The last bit of the 5-bit representation format is the "u-bit"; if U=0 the representation is exact, otherwise (U=1), the number represents the open interval between consecutive exact numbers. The most significant bit is the sign bit.

One of the main disadvantages of this format is that they rely on look-up tables, which are limited to 20 bits or less to be hardware efficient, which results in low precision for certain operations. This format is explained in more detail in [6] and [14].

2.2.3 Unum Type-III: Posits

Unum Type-III, or Posits as they are also known, were introduced by John L. Gustafson in 2017 [15]. They are determined by the format size (n) and by the number of bits necessary to represent the exponent size (es). Therefore, the notation used in this dissertation to express the configuration is Posit<n,es>. Figure 2.4 shows the generic representation format of Posits with four fields.

- S: follows the same definition as Unum Type-I: S=0 if the number is positive and S=1 if the number is negative.
- Regime: used to calculate a scale factor of $useed^{Regime}$. The number of regime bits is variable and

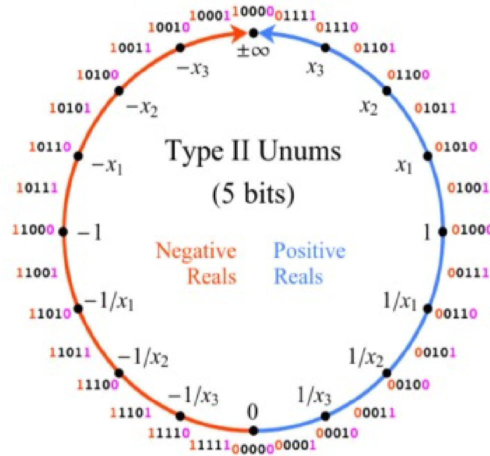


Figure 2.3: Visual Representation of the Unum Type-II Projective Real Number Line [15].

can land between 1 and n-1 bits. It is determined by a run of identical bits, ending either because there are no more bits in the n-bit word or because it runs into an opposite bit. The *useed* value is a described as a "batched" form of powers of two, which depends on m the number of bits of the exponent, and is given by $useed = 2^{2^{m}}$.

- Exponent: represents the scaling factor of 2^{Exponent}. Unlike the floats, there is no bias in the exponent representation. This field has a variable length depending on the Regime field, after which it follows.
- Fraction: represents the significant digits on the right side of the significand's binary point. It follows the same logic as the IEEE 754 floats, but the main difference is that the implicit leading bit is always 1, as there is no sub-normal mode.

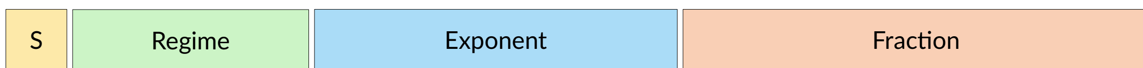


Figure 2.4: Unum Type-III Representation Format.

The formula to convert a Type-III value into its decimal representation is given by the following equation

$$X = (-1)^{\text{S}} \times \text{useed}^{\text{Regime}} \times 2^{\text{Exponent}} \times 1.\text{fraction} \tag{2.2}$$

Type-III has no NaN bit representation. Instead interrupts are advocated for this and other special cases. The interrupts are handled by an interruption handler that can be set to either report the error or

work around it to continue computing. That means that Posits can use more bit patterns to represent real numbers and that fewer hardware resources are used. In terms of the bit patterns available, the Posits architecture has only one representation for 0 compared with the two zero representations in IEEE floats, the 0^+ and 0^-. This is an advantage because it follows the mathematical logic of the concept of 0. On the other hand, the Type-III format does not have signed ∞ representations. There are no subnormal numbers in Posits, since the hidden bit is always 1. It means that there is no ”gradual underflow”.

With Posits, the numbers close to 1 are given more significand bits (more precision bits) and fewer exponent bits. This is an important concept called *tapered precision*.

The dynamic range can be larger than the IEEE 754 dynamic range because of the variably-sized exponent. Gustafson intended to have a drop-in replacement for the IEEE floats, with a faster, more accurate, more hardware-friendly and lower cost number system. Figure 2.5 is extracted from [16] and sums up the advantages and disadvantages of all Unum types.

Unum	Date Intro-duced	IEEE 754 Compati-bility	Advantages	Disadvantages
Type I	March 2015	Yes; perfect superset	Most bit-efficient rigorous-bound representation	Variable width management needed; inherits IEEE 754 dis-advantages, such as redundant representations
Type II	January 2016	No; complete redesign	Maximum infor-mation per bit (can customize to a particular workload); perfect reciprocals (+ − × ÷ 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	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 $\pm\infty$

Figure 2.5: Unum Type-I,II and III Advantages and Disadvantages.

From the figure, it is possible to conclude that Unum Type-III is the most promising Unum format for replacing IEEE 754, and some studies show that Posits can have better behaviour in DL applications compared to floats [8, 23, 22, 21, 29, 19].

Even though this format is the most promising, there are features of Unum Type-I that might be interesting to keep exploring, such as the exponent size field, the level of compatibility with IEEE 754, the use of variably-sized significand and exponent, and its bit-efficiency.

The Regime field in Type-III field behaves as a ”super exponent”, which does not appear to be the best approach to increase the dynamic range or the precision. In fact, those bits can be more useful if used to extend the significand or exponent width. This is the weakest aspect of the Type-III system.

There are other minor weaknesses of this format, such as the fact that there is no difference between the $-\infty$ and $+\infty$ representations, 64-bit Posits can not represent the maximum corresponding float [35], and the level of compatibility with IEEE 754 is lower than Unum Type-I.

In the following chapters, a new Unum format, called Unum-IV, is proposed and explained in detail. This new format explores and combines ideas from both Unum Type-I and the Unum Type-III formats, specifically, the use of an exponent size field, the use of variably-sized significand and exponent. Unum-IV also dispenses with NaN, $-\infty$ and $+\infty$, and does not have two different patterns to represent 0. However, the new format solves the main problem of Posits: the space wasting of the Regime bits as explained above.

The new Unum-IV format uses regular exponent and significand fields, but resorts to an elaborate scheme to avoid redundant representations by introducing a dynamic hidden bit not only in the significand but also in the exponent. The significand’s hidden bit also serves as the sign, which needs no specific field.

Chapter 3

Unum Type-IV

This chapter introduces a new floating-point representation format developed during this thesis, called Unum Type-IV. The new Unum Type-IV, shortly Unum-IV, replaces the unary numeral system used in the type-III format, aka Posit numbers, with a regular binary system. The Unum formats (types I, II, and III) introduced the concept of tapered precision: numbers having an absolute value close to 1 are given more significand bits and fewer exponent bits, and the very large or very small numbers give up significand bits and use more exponent bits. This way, it is possible to extraordinarily increase the dynamic range and accuracy at the said ranges. The main idea in this new representation is to use a hidden bit in the exponent and a hidden bit in the significand to prevent redundant representations. This new format recovers and extends Unum Type-I ideas while being hardware friendly like the Unum Type-III format. Hence, it is named Unum type-IV.

This new format, like Posits, uses the balance between two aesthetics for calculation involving real numbers: (1) the non-rigorous but cheap, fast and ”good enough” approach, and (2) the mathematically rigorous and expensive approach, both in execution time and storage.

3.1 Unum Type-IV Generic Format

The Unum-IV generic format has three fields: the exponent, the fraction and the exponent size. In order to avoid redundant representations, a hidden bit is added to the exponent as well as to the significand. These hidden bits optimise the use of all bit patterns as representable real numbers. In both cases, exponent and significand, the hidden bit is the Most Significant Bit (MSB) and the extraction of exponent and significand is explained in Section 3.2. The significand and exponent in this format are both signed, with the hidden bits providing the sign information. The exponent is in 1’s complement format, so there is no need for a bias as in IEEE 754 and the significand is in 2’s complement format, which dispenses with the sign bit field used in other formats such as the IEEE-754 format.

The Unum-IV configuration and enconding is determined by the format size (DATA_W) and by the number of bits necessary to represent the exponent size (EXP_SZ_W). Therefore, the notation

Unum-IV<DATA_W,EXP_SZ_W>

is used here to denote a DATA_W-bit Unum-IV with EXP_SZ_W bits for the exponent size field. The generic Unum-IV floating-point format is encoded as shown in Figure 3.1, and the next subsections explain each of the fields.

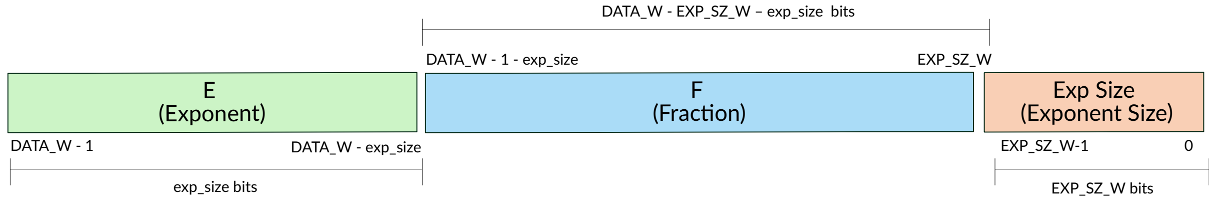


Figure 3.1: Generic Unum-IV Representation Format.

Using this encoding, and processing the fields as explained below, one can obtain the significand s and the exponent e of the real number r represented in the Unum-IV<DATA_W,EXP_SZ_W> format and given by

$$r = s \times 2^e \tag{3.1}$$

Exponent Size (Exp Size)

The Exponent Size (Exp Size) is a small unsigned integer, which has a width defined by the parameter EXP_SZ_W, and can assume a value in the range from 0 to $2^{\text{EXP_SZ_W}} - 1$. The Exp Size field allows for tapered accuracy since it represents the number of explicit bits necessary to represent the exponent of the number in the 1's complement format. It establishes a higher accuracy for numbers that are close to 1, and a lower accuracy for very large or very small numbers.

Denoting the Exp Size bits by ExpSize_i, the explicit exponent size in bits $ExpSz$ is given by

$$ExpSz = \sum_{i=0}^{\text{EXP_SZ_W}-1} \text{ExpSize}_i 2^i \tag{3.2}$$

Exponent (E)

The Exponent (E) field holds the explicit exponent, as indicated by its name. Unlike the IEEE 754, the exponent is not biased and is represented as a signed integer in the 1's complement format with a hidden (implicit) most significant bit. The exponent hidden bit is always the negation of the most significant bit of the E field. Moreover, the 1's complement representation breaks for the widest exponent and most negative 1's complement exponent: $(1)00...00$, where the hidden bit is shown in brackets followed by EXP_SZ_W zeros. This combination is reserved for subnormal representations to allow for a graceful underflow. Subnormal representations are also used in the the IEEE 754 standard, though with a different implementation.

The E field width is ExpSz, therefore it is variably-sized. If $ExpSz = 0$, the E field is not present. Following the logic of the 1's complement signed integers, if the hidden bit is zero the exponent is positive and negative otherwise.

When E is filled with zeros and Exp Size is filled with ones, indicating the widest possible exponent, the subnormal representation applies, as explained above. In this mode, the exponent is valued $-2^{ExpSz}+2$, which is one unit more than the normal 1's complement valuation of $-2^{ExpSz}+1$. Hence, the exponent's magnitude falls in the following range of values: [-$2^{ExpSz}+2 : 2^{ExpSz}-1$].

Without the subnormal representation, there would be a noticeable gap between 0 and smallest negative and positive numbers. By filling this gap around 0, the logarithmic distance between the numbers when approaching zero increases but not as abruptly as with a simple flush to zero approach. This allows computation results to lose precision slowly when very small. Hence, in the new Unum-IV format, the exponent value is given by

$$e = \begin{cases} 0, & \text{if } ExpSz = 0 \\ -2^{ExpSz}+2, & \text{if } ExpSz = 2^{\text{EXP_SZ_W}}-1 \wedge E = 0 \\ \overline{E_{ExpSz-1}}(-2^{ExpSz}+1) + \sum_{i=0}^{ExpSz-1} E_i 2^i, & \text{otherwise} \end{cases} \tag{3.3}$$

The exponent hidden bit E_{HB} is introduced to avoid redundant representations. It does not need to be stored in memory, and is given by

$$E_{HB} = \begin{cases} absent, & \text{if } ExpSz = 0 \\ \overline{E_{ExpSz-1}}, & \text{otherwise} \end{cases} \tag{3.4}$$

Fraction (F)

The Fraction (F) field represents the significant digits on the right side of the significand binary point. Like in the IEEE 754 format, in the Unum-IV format, the significand has an implicit leading bit. However, unike in the IEEE 754 format, this hidden bit is not always 1.

In the Unum-IV case, the significand is represented by a signed 2's complement number, and the hidden bit is always the complement of the fraction's MSB, except for the subnormal representation (see previous section), where the hidden bit equals the fraction's MSB. Thus, if the hidden bit is 0, the represented number is positive or zero; if the hidden bit is one, the number is negative. The F field bit-width $FracSize$ depends of course on the format width and exponent size and is given by

$$FracSize = \text{DATA_W} - ExpSz - \text{EXP_SZ_W} \tag{3.5}$$

Hence, in the new Unum-IV format, the significand is given by

$$s = \begin{cases} -F_{-1} + \sum_{i=1}^{FracSize} F_{-i}2^{-i}, & \text{if } ExpSz = 2^{\text{EXP_SZ_W}}-1 \wedge E = 0 \\ -\overline{F_{-1}} + \sum_{i=1}^{FracSize} F_{-i}2^{-i}, & \text{otherwise} \end{cases} \tag{3.6}$$

The significand's hidden bit F_{HB} in the new Unum-IV format dispenses with the sign bit as in the IEEE-754 format and is given by

$$F_{HB} = \begin{cases} F_{MSB}, & \text{if } ExpSz = 2^{\text{EXP_SZ_W}} - 1 \wedge E = 0 \\ \overline{F_{MSB}}, & \text{otherwise} \end{cases} \tag{3.7}$$

Tapered Precision

The tapered precision is easily verified after having explained the format's fields in the previous sections. The sum of the exponent and fraction sizes is constant and given by

$$ExpSz + FracSize = DATA_W - EXP_SZ_W \tag{3.8}$$

On the one hand, the very large or very small numbers require more exponent bits to represent, get fewer fraction bits and therefore less precision. On the other hand, the numbers closer to magnitude 1 require less exponent bits, get more fraction bits and therefore more precision. The numbers whose exponent equals 0 do not require any exponent bits and get all the available DATA_W - EXP_SZ_W bits for precision.

3.2 Field Extractions

In this section, the exponent and significand binary format extraction is explained and exemplified. As mentioned before, the exponent and significand of the Unum-IV format are both variably-sized, so the field extractions can't be performed in parallel. In these conditions, the extractions are sequenced.

Firstly, the exponent size is extracted because it only depends on the second parameter in the Unum-IV<DATA_W, EXP_SZ_W> notation, that is, EXP_SZ_W.

The *ExpSize* field is extracted by taking the EXP_SZ_W least significant bits of the Unum-IV number. Using the Unum-IV<32,4> configuration as an example, the *ExpSize* field is given by the four least significant bits of the 32-bit Unum-IV word, and it is interpreted as an unsigned 4-bit integer.

After extracting the *ExpSize* value, the exponent and significand can be both extracted in parallel.

3.2.1 Exponent Extraction

The exponent extraction follows the algorithm shown in the Listing 3.1. The hidden bits are shown in brackets and the exclamation mark expresses the bit inversion. For example, (1) means that the implicit hidden bit is 1.

There are three different scenarios in the exponent extraction. The first scenario is when *ExpSize*=0, which implies that the exponent is also zero. These are the numbers close to 1 and do not require an exponent. The second scenario is when the $ExpSize = 2^{EXP_SZ_W} - 1$, the maximum exponent size allowed by the Unum-IV<DATA_W, EXP_SZ_W> configuration, and the exponent E field is all zeros. This is the subnormal representation, where the significand's hidden bit is flipped compared to the normal

representations, and the exponent is incremented by 1 to fill the gap around zero of the Unum-IV normal representation.

For example, in the Unum-IV<32,4> case, this scenario corresponds to $ExpSize = 15$ and the E field with 15 zeros. The third and normal scenario is when the exponent is extracted by left-concatenating the exponent and significant hidden bits to the E and fraction bits.

Listing 3.1: Pseudo-code of the Exponent Extraction Algorithm.

```
if(ExpSize == 0)
  Exponent = 0
else if (ExpSize == 2^EXP_SZ_W-1 && E == 0) // Exception
  Exponent = (!E[ExpSize-1])E[ExpSize-1:0] + 1
else
  Exponent = (!E[ExpSize-1])E[ExpSize-1:0]
```

3.2.2 Significand Extraction

The significand extraction follows the algorithm shown in the Listing 3.2. As in the exponent extraction, the hidden bit is shown in brackets, the exclamation mark expresses the bit inversion, and the dot represents the binary point of the significand.

There are two different extraction scenarios for the significand, In both scenarios, the significand is extracted by concatenating the implicit hidden bit to the left of the binary point, and the fraction bits to the right of the binary point. The first scenario corresponds to the subnormal numbers when the E field is all zeros and ExpSize filed all ones. In this scenario, the hidden bit is the same as the fraction's MSB. The second scenario is for the normal representation and the hidden bit is the complement of the fraction's MSB.

For example, in the Unum-IV<8,2> case, the first scenario corresponds to $ExpSize = 3$ and the E field with 3 zeros. If the F is field with 3 zeros, then the significand is given by (0).000. An example for the second scenario is when F is equal to 10000, the significand is (0).10000.

Listing 3.2: Pseudo-code of the Significand Extraction Algorithm.

```
if(ExpSize == 2^EXP_SZ_W-1 && E ==0) //Exception
  Significand = (F[DATA_W-ExpSize-EXP_SZ_W-1]).F[DATA_W-ExpSize-EXP_SZ_W-1:0]
else
  Significand = (!F[DATA_W-ExpSize-EXP_SZ_W-1]).F[DATA_W-ExpSize-EXP_SZ_W-1:0]
```

3.2.3 Examples

Table 3.1 exemplifies the significand and exponent extraction for a 32-bit Unum-IV format with a 4-bit exponent size field. From Table 3.1 it is possible to conclude that the exponent minimum width is 0, and the maximum is 16, including the implicit leading hidden bit. The significand has a width between 14 and 29 bits, also with the hidden bit.

Table 3.1: Exponent and Significand Extraction for DATA_W=32 and EXP_SZ_W=4.

Exponent Size	**Exponent**	**Significand**
0(min)	0	$(\overline{F_{27}}).F_{27}...F_1F_0$
1	$(\overline{E_0})E_0$	$(\overline{F_{26}}).F_{26}...F_1F_0$
2	$(\overline{E_1})E_1E_0$	$(\overline{F_{25}}).F_{25}...F_1F_0$
3	$(\overline{E_2})E_2E_1E_0$	$(\overline{F_{24}}).F_{24}...F_1F_0$
$\ldots$		
i	$(\overline{E_{i-1}})E_{i-1}...E_1E_0$	$(\overline{F_{27-i}}).F_{27-i}...F_1F_0$
$\ldots$		
8	$(\overline{E_7})E_7...E_1E_0$	$(\overline{F_{19}}).F_{19}...F_1F_0$
$\ldots$		
$15 \wedge E \neq 0$	$(\overline{E_{14}})E_{14}...E_1E_0$	$(\overline{F_{12}}).F_{12}...F_1F_0$
$15 \wedge E = 0$	$(\overline{E_{14}})E_{14}...E_1E_0 + 1$	$(F_{12}).F_{12}...F_1F_0$

3.3 Features

3.3.1 Special Cases

The Unum Type-4 format does not have any special numbers such as ”NaN”, $-\infty$, $+\infty$, ∞, 0^+ or 0^-. Instead of having a range of representations locked for those special cases (losing space for real representable numbers), every combination of bits is used to represent a real number. Hardware exceptions may be implemented when interesting conditions occur. That simplifies the hardware and also extends the dynamic range.

3.3.2 Exceptions

There are three different types of exceptions in the present Unum-IV implementation: the *divide by zero*, *overflow* and *underflow* exceptions. The divide by zero exception occurs when in a division operation the divisor is zero. The overflow exception happens when some operation results in a number that would require a larger exponent than allowed by the Unum-IV format used. The underflow exception occurs when some operation produces a result that would require an exponent more negative than allowed by the format.

The exceptions are handled using flags that monitor the various stages for producing the results. If particular conditions happen, the respective flags are activated, simplifying the hardware used. The exception can then be tread by special hardware circuits or software routines.

3.3.3 Rounding

There are two rounding modes in the present Unum-IV implementation: “the round to the nearest”, ties even” and the “truncation mode”. The rounding or truncation is necessary since operations with

floating-point numbers can often result in non-representable numbers in terms of precision. Thence, the result needs to be rounded to the nearest representable Unum-IV value or truncated to the last representable digit. The default mode is the ”round to the nearest, ties even”.

As in IEEE754, the ”round to the nearest, ties even” mode uses three extra bits of less significance than the significand bits, a *guard* bit, a *round* bit and a *sticky* bit. The most significant bit is the *guard* bit, and the least significant bit is the *sticky* bit.

In this mode, if the exact result can not be represented by an Unum-IV number, the result is rounded to the nearest of two possible values. If there is a tie between the two possibilities, then the even alternative is chosen. The action to be taken according to those three bits is shown in Table 3.2.

Table 3.2: ”Round to the nearest, ties even” Mode.

Guard Bit	**Round Bit**	**Sticky Bit**	**Action**
0	X	X	Round Down
1	0	0	If the significand LSB is 1: Round Up Otherwise: Round Down
1	0	1	Round Up
1	1	0	Round Up
1	1	1	Round Up

The X (”Don’t Care”) means that the bit can either be 0 or 1. In either case, it will have no impact on the action to be taken. In terms of the rounding action: rounding up implies adding 1 bit to the significand, and rounding down does not require any action.

3.3.4 Dynamic Range and Precision

The length of the significand determines the precision of the representable floating-point number. The ratio between the smallest and the largest positive number determines the dynamic range of the number system in evaluation. Since the Unum-IV number system has a variably-sized significand and exponent, the Unum-IV dynamic range and precision depend on the value of the parameter EXP_SZ_W. The larger this parameter is, the greater the exponent contribution in the format, and therefore the dynamic range will increase.

The relation between the exponent size field and the dynamic range is: increasing the number of bits available to represent the exponent increases the dynamic range, and vice-versa. However, increasing the EXP_SZ_W means decreasing the maximum number of bits available to the fraction field since they are the remaining bits of the format. Therefore, if the maximum number of fraction bits decreases, so does the precision.

The principal advantage of the Unum-IV format is the ability to choose the parameters, DATA_W and EXP_SZ_W, to adjust the trade-off between the dynamic range and precision to meet the performance

needs of an application. If the application needs more accurate answers with numbers of small magnitude, it will need more fraction bits and fewer exponent bits. On the other hand, if the application works within a range of extremely small or large values, it will need more exponent bits with less fraction bits available, increasing the dynamic range. This trade-off can make a huge difference in applications that process large amounts of data.

In the Unum-IV generic format, the smallest and largest positive representable number, ***minpos*** and ***maxpos*** respectively, are given by

$$\begin{cases} \textbf{\textit{minpos}} = 2^{\left[-2^{\left[2^{\text{EXP_SZ_W}}-1\right]}+2\right]} \times 2^{-\left[\text{DATA_W}-\left(2^{\text{EXP_SZ_W}}-1\right)-\text{EXP_SZ_W}\right]} \\ \textbf{\textit{maxpos}} = 2^{\left[2^{\left[2^{\text{EXP_SZ_W}}-1\right]}-1\right]} \times \left(1-2^{-\left[\text{DATA_W}-\left(2^{\text{EXP_SZ_W}}-1\right)-\text{EXP_SZ_W}\right]}\right) \end{cases} \quad (3.9)$$

Hence, the dynamic range, measured in decades, is given by the following formula

$$\textbf{\textit{Dynamic Range}} = log_{10}\left(\frac{maxpos}{minpos}\right) \quad (3.10)$$

Table 3.3, summarises and exemplifies the relation between the generic parameters of the Unum Type-IV number system and the dynamic ranges.

Table 3.3: Unum Type-IV Dynamic Ranges.

Unum-IV$<$ n, k $>$	***minpos*** **($\approx$)**	***maxpos*** **($\approx$)**	**Dynamic Range [Decades]**
n=8, k=2	1.95×10^{-3}	1.12×10^{2}	4.76×10^{0}
n=16, k=2	7.63×10^{-6}	1.28×10^{2}	7.22×10^{0}
n=16, k=3	1.84×10^{-40}	1.67×10^{38}	7.80×10^{1}
n=32, k=3	2.80×10^{-45}	1.70×10^{38}	8.28×10^{1}
n=32, k=4	3.45×10^{-9868}	7.08×10^{9863}	1.97×10^{4}
n=64, k=4	8.03×10^{-9878}	7.08×10^{9863}	1.97×10^{4}
n=128, k=4	3.06×10^{-9883}	7.08×10^{9863}	1.97×10^{4}

The number of precision bits (number of fraction bits) for each representable Unum-IV value ***p_bits*** is given by

$$\textbf{\textit{p_bits}} = \text{DATA_W} - ExpSz - \text{EXP_SZ_W} \quad (3.11)$$

where

- $ExpSz$: number of exponent bits, excluding the implicit leading bit.

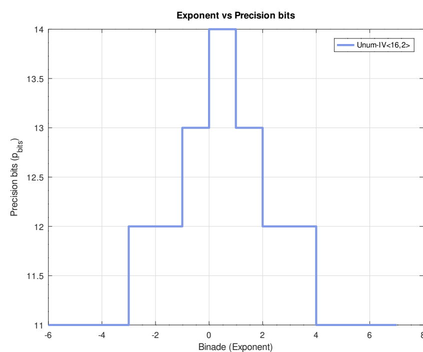
Therefore, the number of precision bits can go from DATA_W-$2^{\text{EXP_SZ_W}}$+1-EXP_SZ_W to DATA_W-EXP_SZ_W bits.

Table 3.4 shows some examples of the relationship between the Unum-IV parameters and the number of precision bits.

Table 3.4: Unum Type IV Precision.

Unum-IV <n,k>	***p_bits***
n=8, k=2	from 3 to 6
n=16, k=2	from 11 to 14
n=16, k=3	from 6 to 13
n=32, k=3	from 22 to 29
n=32, k=4	from 13 to 28
n=64, k=4	from 55 to 60
n=128, k=4	from 109 to 124

Figs. 3.2 and 3.3 help visualize the tapered precision of the Unum-IV format with different parameters, where the *p_bits* are plotted in function of the binades. A *binade*, in software engineering, is the set of numbers of a format with the same base 2 exponent, i.e., the group of bit patterns that have the same exponent bits. Note that the word *decade* denotes the set of numbers with the same base 10 exponent.



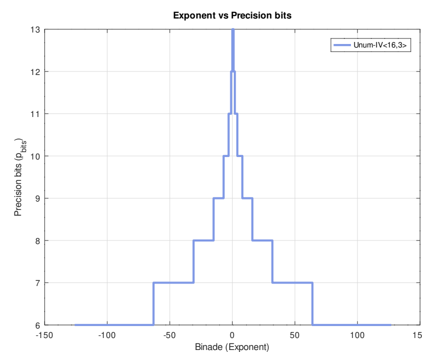
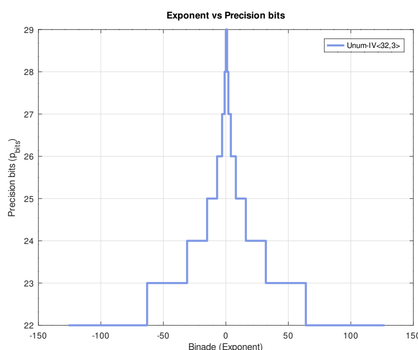


Figure 3.2: Unum-IV<16,2> and Unum-IV<16,3> Precision Bits vs Binade (Exponent).



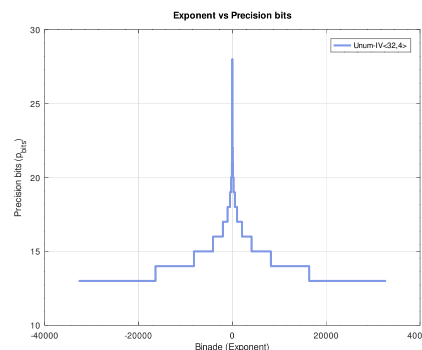


Figure 3.3: Unum-IV<32,3> and Unum-IV<32,4> Precision Bits vs Binade (Exponent).

When the numbers have exponents near 0, they have a higher resolution because the Unum-IV system has more space for fraction bits. As they run far from 0, either to the positive or negative side, the precision decreases symmetrically because the number of bits used to represent the exponent increases. The gradual underflow achieved by the denormalized numbers also enables a gradual loss

of precision.

3.4 Examples

3.4.1 Unum-IV to Decimal Conversion

Three examples of the decoding of the Unum-IV format are presented in Figs. 3.4, 3.5, 3.6 and their mathematical meaning and field extraction are explained in detail. Each example is a 32-bit string Unum-IV, DATA_W=32, with a 4-bit exponent size field, EXP_SZ_W=4.

These examples were selected to give a more concrete way of understanding how the encoding of Unum-IV takes place and express the tapered precision, the wide dynamic range and the gradual underflow attached to this format. The first example is the smallest positive number, and the second is the largest positive number representable by the Unum-IV$<$32,4$>$ format. In both examples, the exponents use the maximum number of exponent bits allowed, 15 exponent explicit bits, and the significand has the minimum number of precision bits (13 fraction bits). The last and third example represents the closest number to 1, which has the maximum number of precision bits (28 fraction bits) since there is no exponent bit in the format, meaning that the exponent is equal to 0.

Following the tapered precision concept, where the numbers with a magnitude near 1 have more precision bits than the numbers extremely small or large numbers and analysing the exponent magnitude of the examples, it is possible to conclude that the last example (near 1) has more precision bits than the previous examples.

The first example is an extremely small number with the minimum fraction width allowed by the chosen parameters and falls on the exponent range lower bound (the exception expressed in the extraction algorithm). This exception case uses the fraction most significant as the implicit leading bit and increments by one the exponent to ensure the closure around 0 of the Unum Type IV numerical system (gradual underflow).

The smallest and largest positive number clearly show the wide dynamic range achieved with the variably-sized exponent and significand compared with the IEEE 754 floats. As can be seen in Table 3.3, Unum-IV$<$32,4$>$ has a dynamic range in order of 20 thousand (1.97×10^4) of decades, while the IEEE 754 floats have a dynamic range of 83 decades, approximately. The difference can be explained by the fact that IEEE 754 floats have a biased exponent of 11 bits, whereas the exponent of the Unum-IV$<$32,4$>$ format can be up to 16 bits, counting with the implicit leading bit.

Smallest Positive Number

In the first example, the exponent size is the maximum allowed by the parameter EXP_SZ_W=4, $2^4 - 1 = 15$ (colour-coded orange). In terms of the exponent, the explicit bits are filled with all 0's 000000000000000 (colour-coded green) and the hidden bit is the complement of exponent field MSB. Following the algorithm of the exponent extraction in Listing 3.1, the exponent falls in the exceptional case criteria. Therefore, to the exponent (1)000000000000000, a bit is added, resulting in (1)0000000000

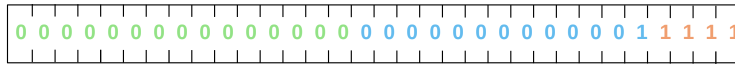


Figure 3.4: Example 1 of an Unum-IV<32,4> Bit String.

00001. The exponent represents -32766 in 1's complement format. The remaining $32-4-15=13$ bits represent the explicit fraction bits. In the exceptional case, the hidden bit is given by the fraction MSB resulting in a (0).0000000000001 significand (colour-coded blue). The significand is given by 2^{-13} and the decimal value of this example is $2^{-13} \times 2^{-32766} \approx -3.45 \times 10^{-9868}$.

Largest Positive Number

Figure 3.5: Example 2 of an Unum-IV<32,4> Bit String.

Since the exponent size is the same as in the previous example the exponent size is 15, meaning that the exponent has 15 explicit bits. The exponent bits are the most significant bits of the Unum-IV bit string, 111111111111111, and the implicit bit is the complement of the MSB of the exponent explicit bits, 0. Thus, the exponent bits (0)111111111111111, represent 2^{32767} as a 1's complement signed integer. Lastly, the remaining 13 bits, 1111111111111, are the fraction bits. The significand hidden bit is the complement of MSB of the fraction field, 0. Then, the significand is (0).1111111111111 and represents $0+2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-10}+2^{-11}+2^{-12}+2^{-13}$ as a 2's complement signed integer. Using the formula of the decimal representation given by the Equation 3.1, the first example results in $\approx 7.08 \times 10^{9863}$.

Largest Number Smaller Than One

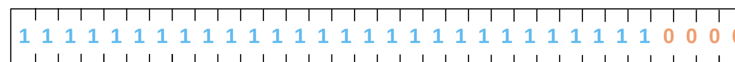


Figure 3.6: Example 3 of an Unum-IV<32,4> Bit String.

Lastly, the exponent size is 0. As consequence, there is no exponent bits in the Unum-IV bit string and the exponent contributes with a scale factor of 2^0 in the decimal conversion equation. In this case, the fraction field has the maximum available width, 28 bits. Evaluating (0).1111111111111111111111111111,

the significand is evaluated as $\sum_{i=1}^{28} 2^{-i}$ and the decimal representation of the last example is $2^0 \times \sum_{i=1}^{28} 2^{-i} = 0.999999996274709701538085937 5$.

3.4.2 Unum-IV<4,1> Enconding

The encoding of all Unum-IV<4,1> bit pattern combinations (16) is shown in Table 3.5. This format expresses compactly the fact that the Unum-IV uses all the bit combinations to represent a numerical value, whereas formats like the IEEE have reserved bit patterns for the 0^+, 0^-, +∞, -∞ and the NaN representations, leaving a lot of redundant representations. As can be seen in the 4-bit Unum-IV<4,1> encoding, there are no redundant representations in the Unum-IV number system. The exponent varies between 0 and 1 explicit bits, and the fraction varies between 2 to 3 bits. There is only one bit pattern combination to represent the 0 (0001), and the gradual underflow is achieved by the following combinations: 0001, 0011, 0101 and 0111. The brackets are used to represent the implicit leading bits of both exponent and significand fields.

Table 3.5: Unum-IV<4,1>.

Unum-IV<4,1>	**Exponent Size**	**Fraction Size**	**Exponent**	**Significand**	**Unum-IV<4,1> Decimal Conversion**
0000	0	3	0	(1).000	-1
0001	1	2	(1)1	(0).00	0
0010	0	3	0	(1).001	-0.875
0011	1	2	(1)1	(0).01	0.25
0100	0	3	0	(1).010	-0.75
0101	1	2	(1)1	(1).10	-0.5
0110	0	3	0	(1).011	-0.675
0111	1	2	(1)1	(1).11	-0.25
1000	0	3	0	(0).100	0.5
1001	1	2	(0)1	(1).00	-2
1010	0	3	0	(0).101	0.625
1011	1	2	(0)1	(1).01	-1.5
1100	0	3	0	(0).110	0.75
1101	1	2	(0)1	(0).10	1
1110	0	3	0	(0).111	0.875
1111	1	2	(0)1	(0).11	1.5

Chapter 4

Hardware Implementation

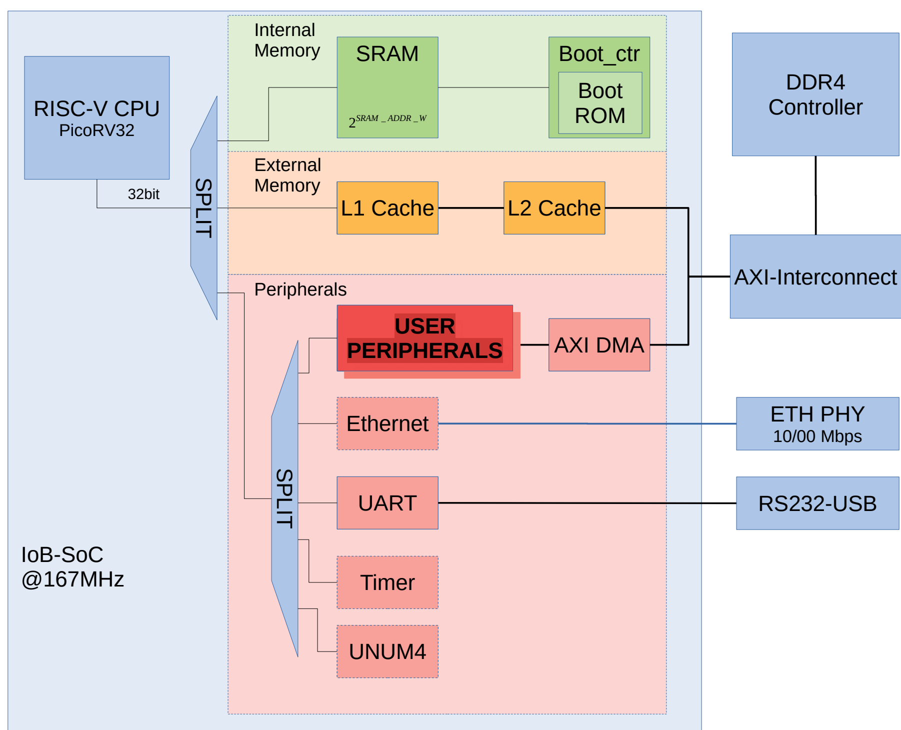


Figure 4.1: IoB-SoC Block Diagram with the Unum-IV Module Attached as a Peripheral.

This chapter presents an Unum Type-IV Floating-Point Unit (Unum-IV FPU) developed during this thesis [36]. The proposed Unum-IV FPU was implemented in Verilog and was attached as peripheral to an open-source IoB-SoC equipped with a RISC-V CPU, an internal SRAM memory subsystem, a UART (iob-uart), and an optional external DDR memory subsystem [37]. The Unum-IV FPU IP core was added to the list of peripherals, as shown in Figure 4.1.

4.1 Unum Type-IV Floating-Point Unit

In this thesis, the design of a parameterized and pipelined Floating-Point Arithmetic Unit based on the new Unum Type-IV format was explored and implemented in Verilog. The hardware implementation supports four types of operations: addition, subtraction, division and multiplication, and it also supports two rounding modes: the truncation and the "round to the nearest, ties even".

Unum-IV FPU is parameterized so that any Unum-IV$< \text{DATA_W}, \text{EXP_SZ_W} >$ arithmetic can be supported, with the option of having different rounding modes associated. Consequently, the FPU has three parameters:

- **DATA_W**: the size of the Unum-IV operands.
- **EXP_SZ_W**: the number of bits available to represent the exponent size.
- **ROUNDING**: rounding mode selector. If 1, the rounding mode is "Round to the Nearest, Ties Even". Otherwise, the only rounding procedure performed to the result is the truncation of the significand.

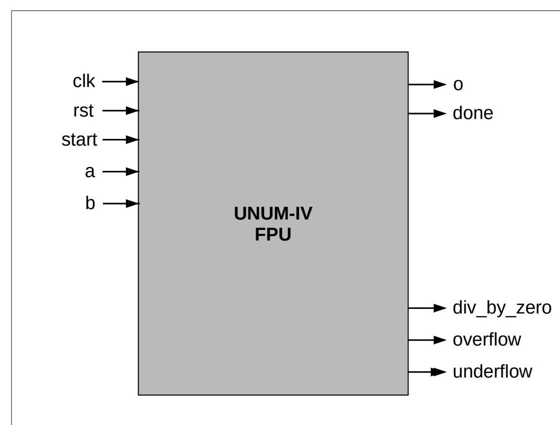


Figure 4.2: Unum-IV FPU.

The top-level module has six input and five output signals as illustrated by the Unum-IV FPU symbol in Figure 4.2. These interface inputs and outputs size, direction and description are explained in Table 4.1. The operand A and B signals have a DATA_W-bit width and are interpreted as Unum-IV$< \text{DATA_W}, \text{EXP_SZ_W} >$ representation numbers; the operation selection controlled by the 2-bit op input interpreted as in Table 4.2. This signal has 2 bits because there are four supported operations; the start, clk and rst signals are 1-bit sized and belong to the control logic of this design. As for the outputs, the o is the Unum-IV$< \text{DATA_W}, \text{EXP_SZ_W} >$ result of the operation performed; the done is a flag that signals the conclusion of the calculation; the div_by_zero, underflow, overflow are exception flags activated if the results fall outside of the format limits or if the operation is invalid.

Table 4.1: FPU Interface.

Signal	Size	Direction	Description
clk	1	Input	System Clock
rst	1	Input	Asynchronous active high reset
start	1	Input	Strobe to start calculation
op	2	Input	Operation selection
a	DATA_W	Input	Operand A
b	DATA_W	Input	Operand B
o	DATA_W	Output	Calculation Result
done	1	Output	Strobe to signal the end of calculation
div_by_zero	1	Output	Strobe to signal an invalid operation ($x/0$)
underflow	1	Output	Strobe to signal an underflow result
overflow	1	Output	Strobe to signal an overflow result

Listing 4.1 is an example Verilog code used to instantiate a parameterizable Unum-IV FPU based in Unum-IV<32,4> format. The rounding parameter is 1, so the arithmetic results will round using the "Round to the Nearest, Ties Even" mode.

Listing 4.1: Verilog Code to Instantiate the Parameterizable Unum-IV Floating Point Unit.

```
fpu   #(.DATA_W(32),.EXP_SZ_W(4),.ROUNDING(1)) u0 (
      .clk(clk),
      .rst(rst),
      .start(en),
      .a(a),
      .b(b),
      .op(op),
      .o(o),
      .overflow(overflow),
      .underflow(underflow),
      .div_by_zero(div_by_zero),
      .done(done)
      );
```

The hardware implementation of Unum-IV arithmetic is based on three hardware stages, as shown in Figure 4.3. The first stage is the unpacking module, where the Unum-IV exponent and significand are extracted from each Unum-IV operand, following the algorithms shown in Listings 3.1 and 3.2. The intermediate stage is the processing module, where the four basic supported operations are performed. The remaining and final stage is the packing module, where the computed exponent and significand are packed in the Unum-IV format. As can be seen in Figure 4.3, Unum-IV FPU has a hierarchical architecture, comprising three main stages. The hierarchy flow is controlled by the Control Logic (CL), preventing the propagation of errors and allowing exceptions handling.

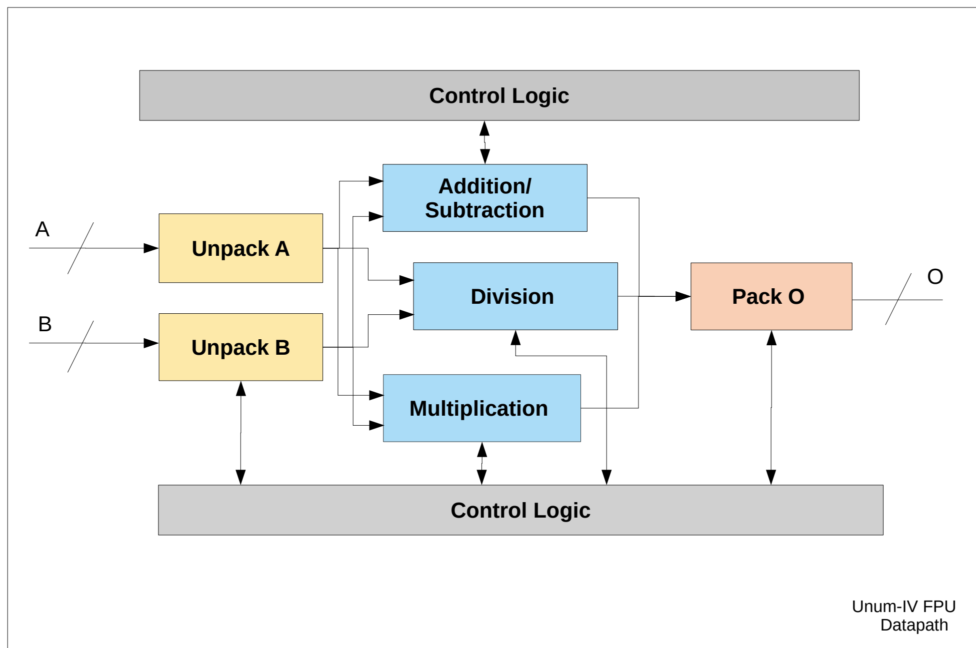


Figure 4.3: Unum-IV FPU Datapath.

The CL block consists of a two-bit register ”op”, containing the operation to be executed, and a ”start and ”done” flag for each module to ensure that the result is correctly propagated and to prevent loss of data.

4.2 Functional Units

The proposed Unum-IV FPU has five principal parameterizable Functional Units (FUs) that make usage of five auxiliary units: a barrel shifter, an adder and subtractor, an exponent difference module, leading zeros/one's detector, a shift and subtract serial divider and a multiplier.

4.2.1 Unpack Unit

The Unpack (unpack) unit used in Unum-IV FPU is parameterized and has four parameters. Two of those parameters are propagated from the fpu top-level module, DATA_W and EXP_SZ_W, and the others are local parameters obtained in fpu top-level module, EXP_MAX_W and MAN_MAX_W, corresponding to the maximum exponent and significand width allowed by the Unum-IV format used, respectively. EXP_MAX_W is obtained directly for the EXP_SZ_W parameter as $2^{\text{EXP_SZ_W}}$ and MAN_MAX_W is computed as $\text{DATA_W} - \text{EXP_SZ_W} + 1$.

This unit has four input ports (clk, rst, start and x). The input x is the DATA_W-bit Unum-IV operand; the clk and rst are propagated from the top-level module; the unpack start signal results of the propaga-

tion of the fpu module start signal. The Unpack unit has three output ports (e, m, done). The e port is the extracted exponent from the x input word with EXP_MAX_W bits; the m port is the extracted significand with MAN_MAX_W width; the done is the strobe that indicates that the unpacking stage is finished.

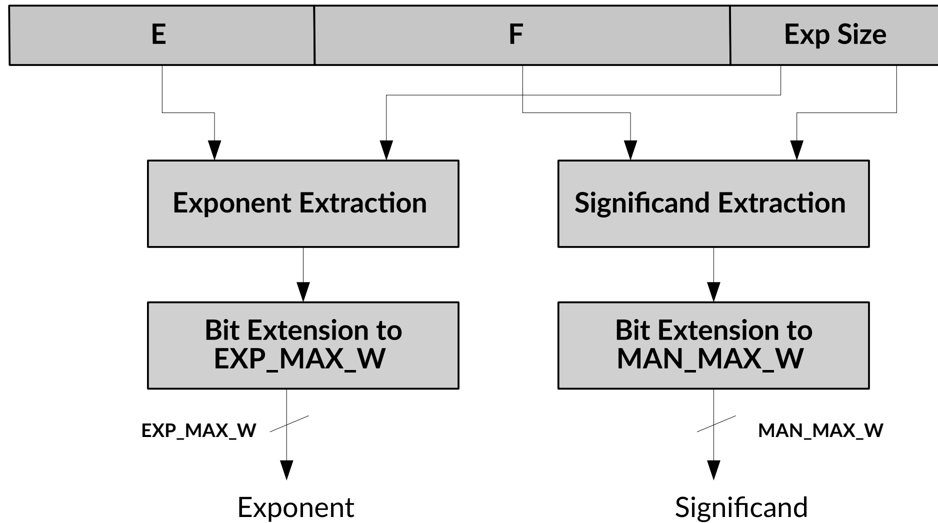


Figure 4.4: Unum-IV Unpack Stage Block Diagram.

The unpack unit is responsible for taking an Unum-IV input and extracting the exponent, fraction and exponent size fields. The data flow is performed as shown in the block diagram presented in Figure 4.4. When the start is high, the unpack is activated, and the EXP_SZ_W least significant bits of the input X is extracted as an unsigned integer, corresponding to the $ExpSize$ field. Then, the exponent and significand are extracted from the input X using extraction algorithms presented in Listings 3.1 and 3.2, respectively. Since the exponent is a signed integer in 1's complement representation, the exponent is converted to the 2's complement representation. This conversion is performed because the main difference between the two representation formats is that 1's complement has two representations of zero, the negative zero and the positive zero, whereas, in 2's complement, there is only one representation for zero. The operations between the 1's complement are also more complex and require more hardware than the 2's complement. For those reasons, the exponent is converted to the 2's complement representation format to be manipulated in the processing unit and then converter back to the 1's complement format in the packing stage. To do this conversion, if the exponent is negative, then the exponent is incremented by one. On the packing unit, the reverse process occurs.

Lastly, the exponent and significand are extended to the maximum width of each field, if necessary, and stored in the e and m output registers. The done signal is activated in this last pipeline stage, signalling the end of the unpacking stage.

4.2.2 Processing Units

Two operation bits are used for this FPU, meaning that up to four operations are available. The processing stage can perform additions, subtractions, divisions and multiplications, as shown in Table 4.2. The addition and subtraction are performed by the same unit, while the division and multiplication are performed by different units.

In the processing unit, if the rounding mode selected is the "round to the nearest, ties even", then the significand receives three extra bits for the least significant part (all set to zero) at the beginning of the processing stage. In that stage, the significand suffers many operations, for example, the shifts, that can change those bits.

Table 4.2: Specification for the Operation Selection.

Operation Bits	**Description**	**Operation**
00	Addition	A+B
01	Subtraction	A-B
10	Division	A/B
11	Multiplication	A*B

Addition/Subtraction Unit

The Addition/Subtraction (adder) unit used in Unum-IV FPU is parameterized and has five parameters. Two of those parameters are propagated from the fpu top-level module, DATA_W and EXP_SZ_W, and the remaining others are local parameters obtained in the fpu top-level module, EXP_MAX_W, MAN_MAX_W, EXTRA, corresponding to the maximum exponent and significand width allowed by the Unum-IV format used and the extra bits needed by the rounding mode, respectively. EXP_MAX_W is obtained directly for the EXP_SZ_W parameter as $2^{\text{EXP_SZ_W}}$, MAN_MAX_W is computed as DATA_W $-$ EXP_SZ_W $+ 1$ and EXTRA is given by $3 \times$ ROUNDING.

This unit has eight input ports (clk, rst, start, e_a, e_b, m_a, m_b, op). The inputs e_a and e_b are the exponents of operand A and B, both extracted in the unpacking stage. The m_a and m_b are the significands of the operand A and B, also obtained from the unpacking stage. The clk and rst are propagated from the top-level module, and the start is the result of a logical AND between the done signals of the two operands unpack stages. Lastly, the op input is the least significant bit of the operations selector, which is used to determine the kind of operation to be performed. If it is 0, then the addition is performed. Otherwise, a subtraction is performed.

The module has five output ports (e_o, m_o, done, over and under). The e_o port is the result exponent from the addition or subtraction between A and B, with EXP_MAX_W bits, the m_o port is the result significand with MAN_MAX_W width, the done is the strobe to indicate to the top module that the processing stage is finished, and the over and under outputs are the exception flags, set to one when an overflow or underflow happened, respectively.

The block diagram of the Unum-IV proposed addition and subtraction unit is shown in Figure 4.5.

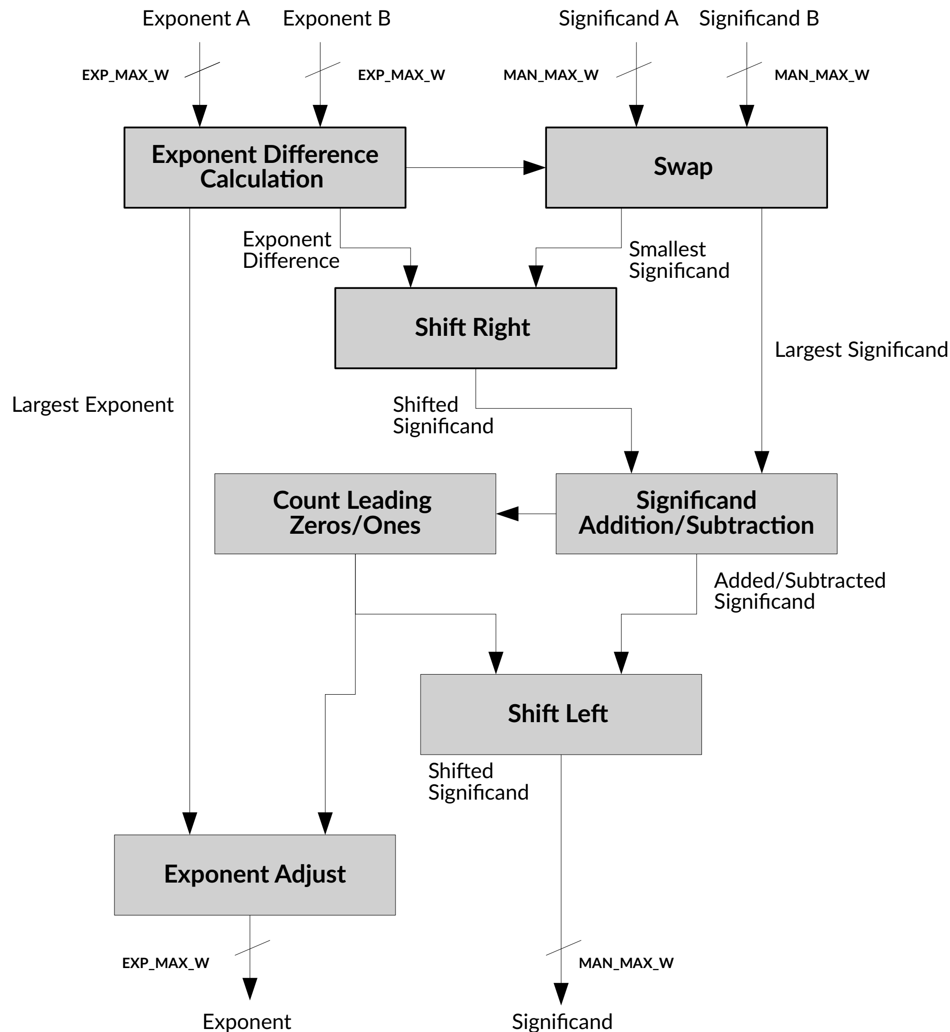


Figure 4.5: Unum-IV Addition/Subtraction Unit.

The unit has six pipeline stages. In the first stage, the extracted exponents and significands of each operand are stored in registers. The exponent difference between operand A and B is computed using a parameterized exponent difference module implemented as described in 4.2.4. The exponent difference module will output the absolute difference and the greatest exponent. In the second stage, a swap between the significands takes place. If the greatest exponent belongs to operand A, then the significand of B needs to be right-shifted to align the exponents and vice-versa. A parameterized leading zeros/ones detector gathers the shift size for the alignment, as described in 4.2.4. Since the significands are signed, the shift is arithmetic. In the third stage, the significands are stored in registers accordingly

with the shifted significand. If the operand A exponent is greater than the operand B exponent, the first operand is the non-shifted significand A and the other is the shifted significand B. On the other hand, if operand B has the largest exponent, the first operand will be the non-shifted B, and the second is the significand A.

In the fourth stage, the significands are added or subtracted in an adder/subtracter module as explained in 4.2.4. If an overflow occurs, the significand overflow is set to 1. If the significand result is an overflow, the significand is right-shifted by one, and the greatest exponent is incremented by one. Whenever the exponent suffers an adjustment, an overflow detection happens. These significand and exponent adjustments take place in the fifth stage.

The last stage is composed by the normalization, where the added or subtracted significand is left-shifted if necessary. To verify the need for shifting a leading zeros/one's detection is performed. If the exponent is the smallest possible by the Unum-IV configuration set by the FPU, then no shift is performed nor any exponent adjustments, meaning there is no normalization. If the exponent falls outside the exponent range, then the overflow or underflow output is set to 1. Finally, if none of those two conditions is met, then the exponent is subtracted by the shift size and the significand is left-shifted by the shift size obtained by the leading zeros/one's detector.

Multiplication Unit

The multiplication (mult) unit used in Unum-IV FPU is parameterized and has five parameters. The parameters are the same for all processing units: DATA_W, EXP_SZ_W, MAN_MAX_W, EXP_MAX_W and EXTRA.

This unit has seven input ports (clk, rst, start, e_a, e_b, m_a, m_b) and five output ports (e_o, m_o, done, over and under). The inputs e_a and e_b are the exponents of operand A and B, both extracted in the unpacking stage. The m_a and m_b are the significands of the operand A and B, also obtained from the unpacking stage. The clk and rst are propagated from the top-level module, and the start is the result of a logical and between the done signals of the two operands unpack stages. The e_o output port is the multiplication exponent between A and B, with EXP_MAX_W bits, the m_o port is the result significand with MAN_MAX_W width, the done is the strobe to indicate to the top module that the processing stage is finished, and the over and under outputs are the exception flags, set to one when an overflow or underflow happened, respectively.

The block diagram of the Unum-IV proposed multiplication unit is shown in Figure 4.6.

This unit is composed of five pipeline stages. The multiplication module is more simple than the addition and subtraction module. In this module, an addition between the extracted exponents stored in registers takes place.

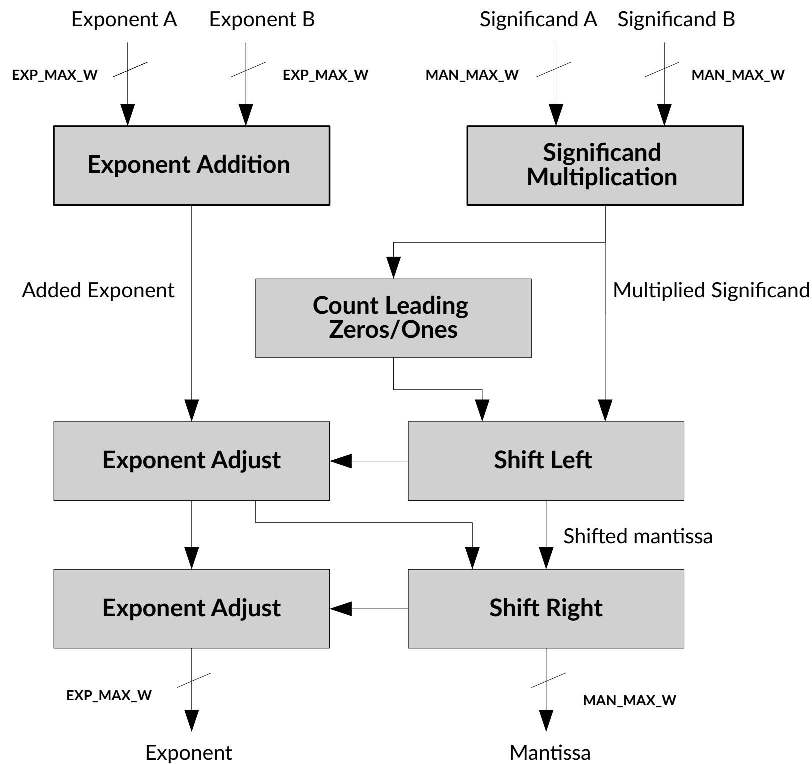


Figure 4.6: Unum-IV Multiplication Unit.

For this, the parameterized adder/subtracter module is used and set in the adder mode, with an EXP_MAN_MAX+1-bit result. The significands are multiplied in this unit using a 2*MAN_MAX_W-bit multiplier. These operations perform in parallel since there are no dependencies between them. Then, if both significands have the same MSB, the added exponent is incremented by one, and the multiplication partial result MAN_MAX_W+EXTRA most significant bits of the multiplication result are stored in a signed significand register. Otherwise, the multiplication result stored is from the second MSB to the MAN_MAX_W+EXTRA-1 bit, resulting in MAN_MAX_W+EXTRA bits saved in the signed significand register. In this case, there are no adjustment needs for the added exponent. Finally, the normalization might take three different steps.

In this first scenario, if the exponent result is equal to the smallest exponent allowed by the Unum-IV configuration, then no normalization is performed, and the exponent and significand are ready to be outputted by the unit along with the overflow and underflow flags set to 0.

In the second scenario, a leading zeros/ones detector is used for the stored significand result to compute the left-shift size needed by the significand. Then, to adjust the exponent, the added exponent is subtracted by the shift size.

In the last scenario, after the first normalization described in the previous situation, if the added exponent is lower than the smallest exponent and the difference between that exponent and the added exponent is smaller than the significand size (MAN_MAX_W+EXTRA), then the second normalization takes place. In this normalization, the exponent is set to the smallest exponent and the significand

is arithmetical right-shifted by that difference. Any other scenario where the exponents fall outside of the exponent range set by the Unum-IV configuration parameters is an exception. Either it can be an overflow if the exponent falls outside the exponent range upper bound or an underflow if the normalized exponent falls outside of the lower bound.

Division Unit

The parameterized division (divide) unit has the same five parameters as the other processing units (DATA_W, EXP_SZ_W, MAN_MAX_W, EXP_MAX_W and EXTRA). It has seven inputs ports (clk, rst, start, e_a, e_b, m_a, m_b) and six output ports (e_o, m_o, over, and div_by_zero). The inputs e_a and e_b are the extracted exponent from the unpacking stage, and the inputs m_a and m_b are the extracted significands. These inputs are propagated as the result of the unpacking stage. The clk and rst are both propagated from the top-level module, and the start input is the result of a logical and between the done signals of the two operands unpack stages. The e_o output is the division exponent between A and B, with EXP_MAX_W bits, the m_o port is the result significand with MAN_MAX_W width, the done is the strobe to indicate to the top module that the processing stage is finished, the over and under outputs are the exception flags, set to one when an overflow or underflow happened, respectively. There is one more exception flag that is only used for the division, the div_by_zero. This flag is set to one when operand B is equal to zero, being handled as an invalid operation.

The block diagram of the Unum-IV proposed division unit is shown in Figure 4.6.

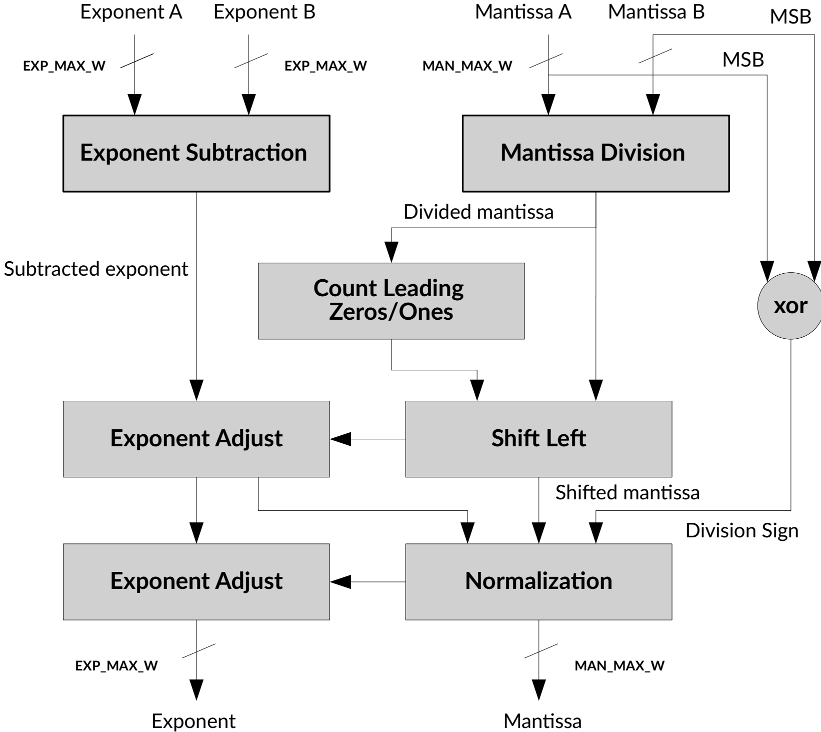


Figure 4.7: Unum-IV Division Unit.

The division operation between two operands takes 5+MAN_MAX_W+1 clock cycles. In the first stage of the division module, the extracted exponents and significands from the unpacking stage are stored in registers, and operand B is evaluated. If its significand is equal to zero, the flag div_by_zero is set to one, and the operation is interrupted. Assuming that it is not interrupted, the exponents are subtracted between themselves using the parameterizable adder/subtracter module in the subtraction mode with EXP_MAX_W-bit signed operand inputs and EXP_MAX_W+1-bit output result.

For the significand division, an unsigned division is performed. Therefore, since the operands are both 2's complement signed numbers, the significands are converted to theirs unsigned format before entering the division module. The division is performed by a subtract and shift serial division module explained in detail in 4.2.4. This module takes MAN_MAX_W+1+EXTRA clock cycles to obtain the unsigned division result. The sign of the division is given by a logical XOR between the MSB of both operands. After the division is executed, the normalization process begins, where, firstly, the quotient of the significand division is right-shifted, and the exponent is adjusted by adding the shift size to the exponent. Then, the MAN_MAX_W+EXTRA significand bits of the quotient are stored as the significand result of the division and reconverted to the 2's complement format is the sign bit is set to one.

As in the multiplication unit, there are different normalization steps depending on the Unum-IV format. In the first scenario, if the exponent result is equal to the smallest exponent allowed by the Unum-IV configuration, then no normalization is performed, and the exponent and significand are ready to be outputted by the unit along with the overflow and underflow flags set to 0.

In the second scenario, a leading zeros/ones detector is used for the stored significand result to compute the left-shift size needed by the significand. Then, to adjust the exponent, the added exponent is subtracted by the shift size.

In the last scenario, after the first normalization described in the previous situation, if the added exponent is lower than the smallest exponent and the difference between that exponent and the added exponent is smaller than the significand size (MAN_MAX_W+EXTRA), then the second normalization takes place. In this normalization, the exponent is set to the smallest exponent and the significand is arithmetical right-shifted by that difference. Any other scenario where the exponents fall outside of the exponent range set by the Unum-IV configuration parameters is an exception. Either it can be an overflow if the exponent falls outside the exponent range upper bound or an underflow if the normalized exponent falls outside of the lower bound.

4.2.3 Pack Unit

Two different modules can be generated to pack the FPU operation result into the Unum-IV format depending on the rounding mode chosen using the ROUNDING parameter. If the parameter is set to one, the selected is the "round to the nearest, ties even" procedure, which is more complex than the truncation mode (ROUNDING=0). The block diagram of the Unum-IV proposed packing unit is shown in Figure 4.8.

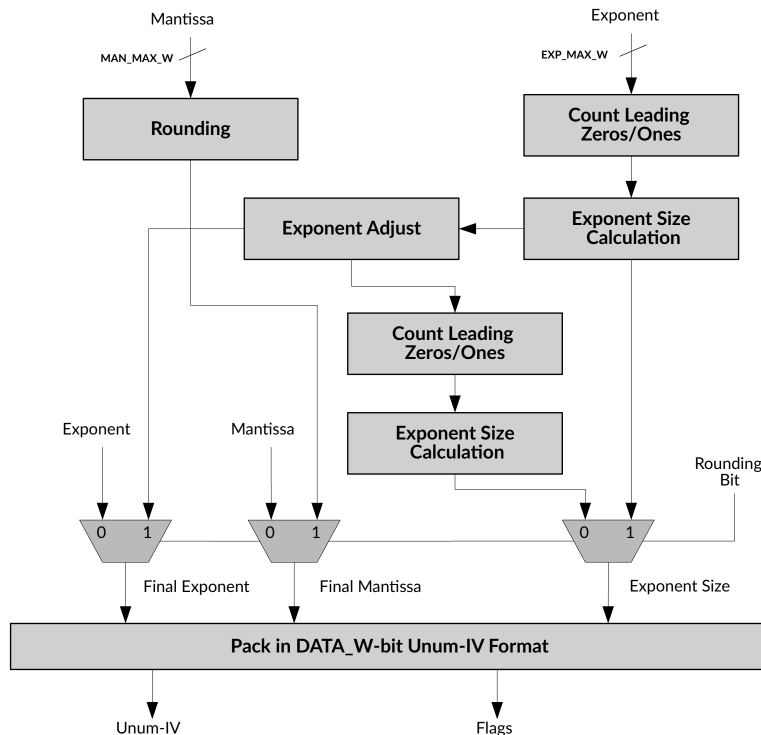


Figure 4.8: Unum-IV Pack Unit.

For the truncation mode, the generated module has four parameters (DATA_W, EXP_SZ_W, MAN_MAX_W and EXP_MAX_W), five inputs (clk, rst, start, exp and mant) and two output ports (o and done). On the other hand, for the ”round to the nearest, ties even” mode, the generated module has five parameters (DATA_W, EXP_SZ_W, MAN_MAX_W, EXP_MAX_W and EXTRA), five input ports (clk, rst, start, exp and mant) and three output ports (o, done and over).

The module with the truncation mode (pack0) takes 3 clock cycles to generate the result. In the first stage, the exponent and mantissa propagated from one of the processing units are stored in registers. Since the exponent was converted to the 2’s complement format during the unpacking stage, the exponent is reconverted to the 1’s complement format. For that purpose, if the MSB of the exponent is equal to one, then the exponent is decremented by one. In the exceptional case, where the MSB and the second MSB of the significand are equal, the exponent is decremented by two. This happens when the exponent is the smallest possible allowed by the Unum-IV format used, and the significand is not normalized as explained in the previous chapter. The exponent reconverted in 1’s complement format is stored in a pipeline register afterwards, and the exponent size is calculated using a parameterized leading zeros/one’s detector for the exponent. The number of leading zeros or ones obtained is then subtracted to the maximum exponent size minus one (EXP_MAX_W-1).

Lastly, each field is stored in a DATA_W-bit register accordingly to the Unum-IV¡DATA_W, EXP_SZ_W¿ format used. So, the exponent size is stored in the EXP_SZ_W least significant bits of the register. From the MAN_MAX_W-bit significand, only a portion of those bits are stored depending on the exponent size (truncation) determined in the previous step. The MSB of both exponent and significand is not stored in the format as they are implicit leading bits. Then the missing field is the exponent, which is placed on

the remaining bits of the Unum-IV register. The done is set to one, and the Unum-IV output is ready in the last and third pipeline stage.

The module with the "round to the nearest, ties even" mode (pack) is very similar to the pack0 module. The main difference is the rounding logic, which requires more hardware. So, this module receives as an input a significand with MAN_MAX_W+3 bits. The first stage follows the same logic since it only manipulated the exponent. So, the exponent is reconverted to its 1's complement format, and the exponent size is calculated using a leading zeros/one's detector. Then, the rounding bits are assigned accordingly with the exponent size (exp_size) determined. Those bits are assigned as described in Listing 4.2, where the rounding logic is also described.

Listing 4.2: Rounding Bits Assignment.

```
//Rounding Bits
  assign m_lsb = mant_reg[3+exp_size];
  assign guard_bit = mant_reg[2+exp_size];
  assign round_bit = mant_reg[1+exp_size];
  assign sticky_bit = mant_reg[exp_size+1-1];
// Round
  assign round = (~guard_bit)? 1'b0: (~(round_bit | sticky_bit) & ~m_lsb) ? 1'b0: 1'b1;
  assign extra = ({{MAN_MAX_W-1{1'b0}},1'b1} << exp_size);
  assign mantissa = (round)? mant_reg[MAN_MAX_W-1+EXTRA:3] + extra:
  mant_reg[MAN_MAX_W-1+EXTRA:3];
```

The mant_reg is a register where the propagated significand of the processing unit is stored. Whereas the m_lsb, guard_bit, round_bit, sticky_bit, round, extra and mantissa are all wires. The m_lsb is the least significant bit of the significand without taking into account the extra bits, the guard bit (guard_bit) is the MSB of the extra bits, the round bit (round_bit) is the intermediate bit, and the sticky bit (sticky_bit) is the least significant. The round bit is a wire that verifies the need for rounding. It is set to one if the significand needs to be rounded. This rounding logic was described in Table 3.2. Since the significand size stored in the format varies with the exponent size, the extra wire has a MAN_MAX_W-bit width with the LSB set to one, which is left-shifted to ensure that the extra rounding bit is incremented to the least significant bit of the explicit significand.

After that, the rounded significand needs to be normalized, and the exponent needs to be consequentially adjusted, if necessary. There are two different scenarios for this stage. Either the significand is positive and becomes negative with the rounding, meaning that it positively overflowed, or the significand is negative and becomes positive, meaning that it negatively overflowed. In the first case, the exponent is logical right-shifted by one, and the exponent is also incremented by one. In this case, the exponent might overflow, so the overflow output can be set to one. Otherwise, the exponent is decremented by one and the rounded significand is shifted by one. Since the exponent might suffer an adjustment, the exponent size needs to be recalculated with the resource of a leading zeros/one's detector as in the other module. Finally, the packing of each field into the Unum-IV format is done exactly like in the pack0 module. The pack module takes more hardware than the pack0 module, and also takes more clock cycles since it has four pipeline stages (instead of three).

4.2.4 Auxiliary Components

Exponent Difference Module

This module is parameterizable, having the maximum exponent size as a parameter and has two inputs (e_a and e_b), corresponding to the extracted exponents from the operand A and from the operand B, respectively. A signed subtraction is performed between those two inputs. Then, the outputs are obtained from that subtraction. The diff_out output is the absolute difference between the two exponents and the e_larger is a result of a multiplexer, where the e_a and e_b are the inputs and the decision is made based in their value. If e_b is smaller, then the e_larger is e_a and vice-versa.

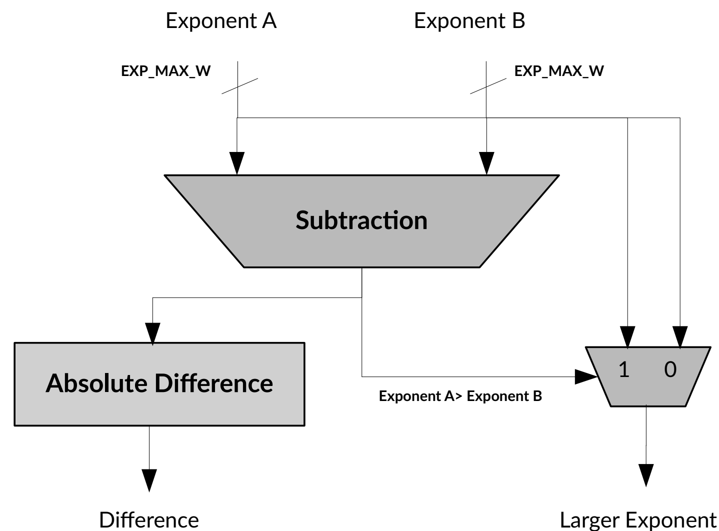


Figure 4.9: Unum-IV Exponent Difference Flow Diagram.

Barrel Shifter

The parameterized barrel shifter module was designed to perform left and right shifts. The shift right operation can only be configured as arithmetic, this is, with sign extension. The barrel shifter was implemented to perform significand shifts, so, logically, the two configuration parameter of the module are the significand length (MAN_MAX_W) and the extra rounding bits (EXTRA). Considering that the barrel shifter only performs two types of shifts, there is a 1-bit input of the module that defines which is performed (left if 1, right if 0). The other two inputs are the value to be shifted and the shift size (0 to EXP_MAX_W).

Leading Zeros/Ones Detector

A parameterized leading zeros or one detector module was adopted in the Unum-IV FPU to obtain the normalization shift size and detect the exponent width. Since the significand and exponent are signed, the detector needed to detect the number of leading zeros if the input is positive or zero or

to detect the number of leading ones if it is negative. The Leading Zeros/One's Detector is the key to perform shiftings and the normalization process in all operations available in the FPU. This unit compares sequentially two adjacent bits from the most significant to the least and counts the number of times that those bits are equal. The comparison stops when two adjacent bits are different, and the number of leadings bits is reported. Thus, the unit has one input and one output. The input can either be the exponent or the significand, as previously mentioned, so the configuration parameters are the exponent width (EXP_MAX_W), the significand (MAN_MAX_W) width and the extra bits required by the rounding mode(EXTRA) if any. The only output of this unit will be the number of leading zeros or ones detected if any.

Serial Subtract and Shift Divider

To perform divisions, the FPU division unit uses a fixed-point serial divider to divide the significands. The divider added as a parameterizable module, takes MAN_MAX_W+1+EXTRA cycles to complete the division due to the fact that the divider is a shift and subtract serial divider. The module takes MAN_MAX_W+1+EXTRA cycles because of the data width (DATA_W) parameter of the divider, which can be 2*MAN_MAX_W or 2*MAN_MAX_W+EXTRA depending on the rounding mode. The EXTRA is a 0 is there the rounding mode only requires truncation of the significand and 3 if the ohter mode is set in the instatiation of the FPU. The divider it can be set as an unsigned or signed divider, however in the division unit of this FPU the selected mode to perform the significand division is the unsigned one. This module has two registers for the operands, the divider and divisior, and the results are other two registers, the quotient and the remainder.

Adder/Subtracter

The adder/subtracter module was used in the proposed FPU to add and subtract significands in the addition or subtraction module, respectively, to subtract the operand exponents if the operation performed by the FPU is a division and to add the exponents if it is a multiplication. Hence the module receives two operands as inputs and a 1-bit operation selector to indicate which operation will be performed, an addition or a subtraction. If the operation bit set to 0, it selects the subtracter mode. Otherwise, the module will stage as an adder. Since the exponent and significand are variably-sized, instead of implementing a fixed-sized adder/subtracter with a wide bit length, a parameterized unit was adopted. One of the parameters is the operand width, which depends on the exponent width, EXP_MAX_W, or the significand operand width, MAN_MAX_W. The other parameter is the extra bits used for the rounding mode, which can either be equal to 0 or 3. The sum of those two parameters comprises the bit width of the parameterized adder or subtracter.

The output result of the addition or subtraction has an extra bit to save the carry bit of the operation result. Also, an overflow flag is set to signal out the occurrence of an overflow. Since both exponent and significand fields are signed, the adder/subtracter is also signed. Therefore, the logic behind it is the same as for signed integer operations. If it is an addition and the operands have the same MSB and the

result MSB is different, an overflow occurs. On the other hand, if it is subtractions, the operands have different MSBs, and the subtraction outcome MSB is different from the operand to be subtracted MSB, an overflow occurs.

4.2.5 Functional Units Pipeline Stages

Each FU has a latency due to pipelining: 3 pipeline stages for the Unpacking FU, 6 stages for the Addition/Subtraction FU, 4 stages for the Multiplication FU, 5+MAN_MAX_W+1+EXTRA stages for the Division Unit and 3 or 4 stages for the Packing FU, depending on the rounding mode choosen. Therefore, if an addition or subtraction is performed between two Unum-IV numbers, the Unum-IV FPU will take 3+6+3+(1) stages to output the result. On the other hand, if a multiplication is selected by the operation selector, the FPU will take 3+4+3+(1) stages to process the result. Finally, the operation with more latency is the division, which would take 3+5+MAN_MAX_W+1+EXTRA+3+(1) clock cycles to output the result. The brackets represent an extra pipeline stage that is only added to the contabilization if the Packing FU generated uses the ”round to the nearest, ties even” mode. The latency of each FU is presented in Table 4.3.

Table 4.3: Functional Units Pipeline Stages

Functional Unit	**Pipeline Stages**
Unpacking Unit	3
Addition/Subtraction Unit	6
Multiplication Unit	4
Division Unit	5+MAN_MAX_W+1+EXTRA
Packing Unit (pack0 / pack)	3 / 4

Chapter 5

Evaluating and Comparing Unum Type-IV to Other Formats

As stated previously, the main goal of this work is to introduce a new number system that can be a replacement for IEEE 754. This chapter presents and explains the main differences between the Unum Type-IV and other formats.

5.1 Comparison Metrics

In this section, the metrics used to evaluate and compare the Unum Type-IV with other formats, such as the IEEE 754 Standard, Unum Type-III (Posits) and ”Bfloat16”, are introduced.

5.1.1 Precision Bits

The number of precision bits introduced in Chapter 3 is a metric of study used to compare the precision variation along with the range of representable values of each format. The number of precision bits is equivalent to the number of explicit significand bits. If the significand has a fixed width, then the number of precision bits is constant. Otherwise, the number of precision bits will vary depending on the other fields of the respective format. For the Unum Type-IV, as described in Chapter 3, the number of precision bits depends on the exponent size ($ExpSz$) and the EXP_SZ_W parameter, and it is given by

$$\textit{p_bits} = \text{DATA_W} - ExpSz - \text{EXP_SZ_W} \tag{5.1}$$

where:

$ExpSz$: number of explicit exponent bits.

5.1.2 Dynamic Range

The dynamic range, also introduced in Chapter 3, is a metric of study used to compare the range of representable values of each format. This metric, measured in decades, is the logarithmic base 10 of the ratio between the largest (maxpos) and smallest (minpos) representable positive values, given by the following formula

$$\textit{Dynamic Range} = log_{10}\left(\frac{maxpos}{minpos}\right) \tag{5.2}$$

5.1.3 Hardware Resources

The hardware resource metrics used in this dissertation to compare Unum Type-IV with the other floating-point formats are given by the silicon area implemented or estimated, the power consumption and clock frequency, both obtained from the implementation of the respective FPUs in the ASIC. In Section 5.5, these metrics are used and compared in detail for different configurations.

5.1.4 Decimals of Accuracy

The metric of study used to compare the accuracy is the *Decimal Accuracy* proposed by John L. Gustafson in [15] given by the inverse of the decimal error in a decibel scale. The *Decimals of Accuracy* are the number of accurate digits to the right of the decimal point between a correct number and a computed number in the respective format. The formula is given by

$$\textit{Decimals of Accuracy} = log_{10}(\frac{1}{\textit{Decimal Error}}) = -log_{10}(|log_{10}(\frac{x_{computed}}{x_{exact}})|) \tag{5.3}$$

where:

Decimal Error$= |log_{10}(\frac{x_{computed}}{x_{exact}})|$

In this dissertation, the formula adapts to the worst case, where it calculates the *Decimals of Accuracy* between two consecutive representable values. This strategy gives a clear vision of the accuracy from the smallest positive value and the largest positive value of each format. Therefore, the formula used is

$$\textit{Decimals of Accuracy} = -log_{10}(|log_{10}(\frac{x_i}{x_{i+1}})|) \tag{5.4}$$

where:

x_i and x_{i+1} represent two consecutive representable values, either in Unum-IV or IEEE 754 format.

5.1.5 Units of Least Precision

Another measure used in this dissertation to evaluate the resolution of the formats is the Units of Least Precision (ULP), which associates to a representable value the weight of the last bit of the significand. This measure gives the spacing between two consecutive floating-point numbers, and it is variable

depending on the exponent and precision. The ULP of a x_i number is given by

$$\text{ULP}(x_i) = 2^{-p} \times 2^{e} \tag{5.5}$$

where:

p is the number of explicit significand bits of x_i (precision bits) and e is the exponent of x_i.

5.2 Comparing Unum Type-IV Features with Other Formats

Table 5.1 presents a qualitative comparison between the most relevant features separating Unum Type-IV, Unum Type-III and IEEE 754 number systems. The Unum-IV and Posits formats have some distinctive features which can not be found in the IEEE 754 number system.

Table 5.1: Qualitative Comparison Between IEEE 754 Standard and Unum Type-IV.

Features	**IEEE 754 [n-bit]**	**Posits [n-bit]**	**Unum-IV [n-bit]**
Portability/Reproducibility	No	Yes	Yes
Redundant Representations	Many	None	None
NaNs Representations	$2^{n-e}-2$ w/ $e=\{5,8,11,15\}$	None	None
Infinity Representations	2 ($-\infty/+\infty$)	1 (∞)	None
Zero Representations	2 ($0^-;0^+$)	1 (0)	1 (0)
Real Number Representations	Exceptions: NaNs,$+\infty$,$-\infty$	Exceptions: ∞	2^n
Overflow	"Falls of a Cliff"	Gradual (tapered accuracy)	Never. Exceptions: $\frac{1}{0}=\infty$
Underflow	Gradual	Gradual	Gradual
Exponent	Fixed-Size; Biased; Unsigned; 0 Implicit Leading Bits	Variable-Size; Unsigned 0 Implicit Leading Bits	Variable-Size; Signed (1's Complement); 1 Implicit Leading Bit
Significand	Fixed-Size; Unsigned; 1 Implicit Leading Bit	Variable-Size; Signed (2's Complement) 1 Implicit Leading Bit	Variable-Size; Signed (2's Complement); 1 Implicit Leading Bit
Precision Bits	Fixed	Variable	Variable

There are existing works [6, 7, 8] that point out the main disadvantages of the IEEE 754 Standard, as the portability/reproducibility feature in Table 5.1 because there is no guarantee of identical results across systems, meaning that there is no portable or repeatable behaviour. The IEEE 754 standard specifies the binary format and the semantics of the operations. Although, the standard leaves plenty of space for the compilers to diversify the implementation of IEEE 754. Therefore, the same code may produce slightly different results on different systems.

In terms of redundant representations, there are no redundant representations in the Unum-IV and Posits formats, unlike IEEE 754 that has many due to the NaN bit pattern representations. Whereas the IEEE 754 floating points are polluted with NaN values, Unum-IV uses all those patterns to represent real numbers. To understand the NaN problem, let's compare the number of NaN representations in the 32-bit IEEE format, which has 16777214 NaN combinations, meaning that any 32-bit Unum-IV configuration format has at least more 16777214 representable real numbers than the IEEE format, corresponding to 0.4% of the total number of values. There are no wasted representations for the Unum Type-IV format.

Unum-IV has no representation for infinities, whereas IEEE 754 has two distinct bit patterns represen-

tations, $-\infty$ and $+\infty$. As stated in some works that evaluate the IEEE 754 disadvantages, the overflow is captured by the infinity representations, creating infinite relative errors. As for the zero representation, the IEEE has two combinations, 0^- and 0^+, while the Unum-IV has only a bit pattern representation for zero. The Posits format has just one representation for the infinity and another for zero. The existence of two representations for zero leads to an undesired complexity in mathematical equality. Unlike IEEE 754, the Unum-IV and Posits number systems verify the equality of two numbers by simply checking their bit sequence. If two Unum-IV or Posit numbers have the same bit-pattern combination, then their equality is proved. In IEEE 754, equality is more complex due to the redundancy in the format and the 0^+ and 0^-. Mathematically, even though the two representations of zero are distinct, they compare as equal. The other issue with equality is the existence of NaN, represented by the various bit sequences with an all-ones exponent and non-zero significand. These reasons make it impossible for IEEE 754 to verify the equality of two numbers by only comparing their bit patterns.

When an overflow occurs, the IEEE 754 Standard assigns the result to infinity. In practical terms, if a number exceeds the maximum representable number of the format, then the compiler using IEEE 754 will be overflowed to infinity, which will create an infinite relative error. For the underflow case, IEEE 754 assigns the result to zero. On the other hand, the Unum-IV arithmetic uses flags to be interpreted by an interruption handler to interrupt the calculation if any exception flags are activated. This strategy is adopted to avoid propagating errors due to the lack of precision or range of the chosen formats. Both underflow scenarios are gradual due to the presence of ”subnormals” numbers, although Unum-IV uses tapered precision on top of the subnormals to delay the underflow. For the overflow, IEEE 754 ”falls off a cliff” because it does not gradually overflow. Instead, it uses a vast combination of bit patterns to represent the NaN values. Discordantly, the Unum-IV is more symmetrical, so it also gradually overflow through the tapered precision achieved by the variably-sized exponent and significand. This lack of symmetrically in IEEE 754 compared with the Unum Type-IV format can be seen in Figure 5.7. The Unum Type-III number system never overflows to infinity or underflows to zero [15]. There is just one exception described in Table 5.1.

These main differences between the two formats are all reached by the different formats of exponents and significands. As for the IEEE 754 Standard, the exponent has a fixed number of bits (5 for half-precision; 8 for single-precision; 11 for double-precision; 15 for quad-precision), and it is biased and unsigned. The significand also has a fixed number of explicit bits (10 for half-precision; 23 for single-precision; 52 for double-precision; 112 for quad-precision), has an implicit leading bit, and it is unsigned. For the Unum-IV format, both exponent and significand have a hidden bit, are signed, 1’s complement and 2’s complement, respectively, and have a variable size. Finally, the Posits significand is signed (2’s complement), has an implicit leading bit always equal to 1, whereas the exponent is an unsigned integer, both with a variable size.

The tapered precision of these two Unum formats presents an advantage, also because one of the points referred to as an issue for IEEE 754 is that the exponent usually takes too many bits.

5.3 Comparing Unum Type-IV Dynamic Range with Other Formats

As referred to in Chapter 3, the dynamic range is the ratio between the largest and the smallest positive values that a specific number system configuration can assume in decades. The Unum Type-IV number system can have greater or matching dynamic ranges with the IEEE 754 Standard formats depending on the configuration adopted. The half-precision IEEE 754 has a dynamic range of about 12 decades, single-precision has about 83 decades, double-precision has nearly 652 decades, and the quad-precision format has about 9882 decades. These four formats are the formats defined in the standard, having 5, 8, 11 and 15 exponent bits, respectively.

The Unum Type-III Standard Draft [35] includes the Posit<16,1>, Posit<32,2>, Posit<64,3> and Posit<128,4> configuration. The Posit<16,1> configuration has a dynamic range of about 16 decades, Posit<32,2> has about 73 decades, Posit<64,3> has nearly 299 decades, and, finally, 128-bit Posit<128,4> has, approximately, 1214 decades.

Figure 5.1 shows the dynamic range, in decades, for the different IEEE 754 and Posits standards, and some of the most significant configurations of Unum-IV, namely, Unum-IV<16,3> with about 78 decades, Unum-IV<32,3> with nearly 83 decades, Unum-IV<32,4> with, approximately, 19731 decades, Unum-IV<64,4> and Unum-IV<128,4> with 19741 and 19746 decades, respectively.

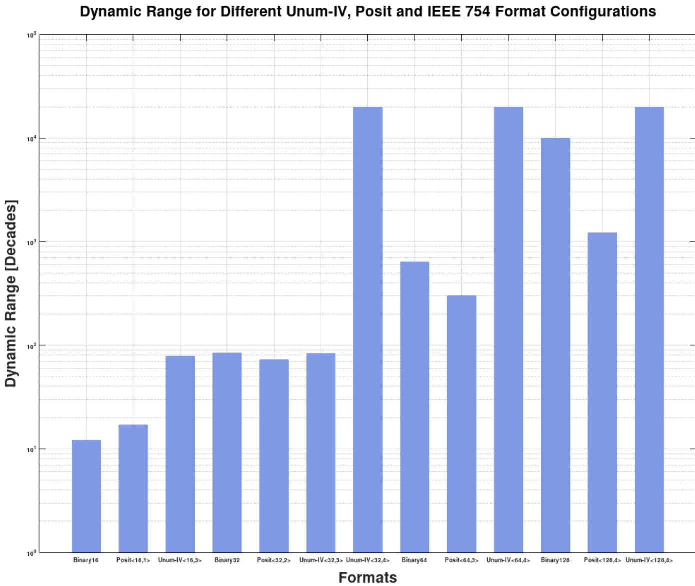


Figure 5.1: Dynamic Range for Different Unum-IV and IEEE754 Format Configurations.

As can be seen in Figure 5.1, the Unum-IV$< 16,3 >$ configuration has almost the same dynamic range as the 32-bit IEEE 754 float, which is stored occupying twice of the computer memory. Unum-IV$< 32,4 >$ has more than 237 times the dynamic range compared with the IEEE 754 bit string with the same width (single-precision), has more than 30 times the dynamic range of the 64-bit IEEE 754 format, which uses the double of the format memory, and it also has a slightly better dynamic range that the quadruple-precision floats, with 128 bits of memory.

Figure 5.1 also shows that Unum Type-IV not only has a wider dynamic range for the same bit width

as Posits but also with using half of it.

5.4 Comparing Unum Type-IV Precision with Other Formats

In computer arithmetic, besides the dynamic range of the data representation, accuracy and performance are critical features that must be considered.

In Figures 5.3, 5.4, 5.2, 5.5 and 5.6, the number of significand bits is shown as a function of the exponent range of each format. The highlighted areas are the ”golden zone” where the Unum-IV format in question has at least the same resolution as floats.

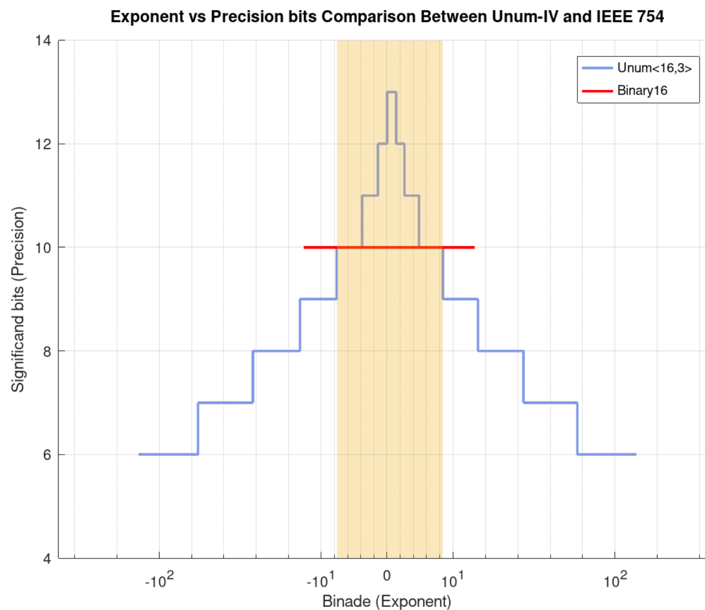


Figure 5.2: Exponent vs Significand Bits Comparison Between Unum-IV<16,3> & 16-bit IEEE 754 Format.

Since IEEE 754 uses fixed-size exponent and fraction, the number of fraction or precision bits, as described in Chapter 3, is constant. For example, the 16-bit floats, the fraction field has ten explicit bits. As for the remaining formats defined in the standard, in the 32-bit, 64-bit and 128-bit floats, the fraction has 23, 52 and 112 bits, respectively. The Unum-IV format has variable resolution, having more precision bits for exponents near 0 and fewer precision bits for numbers with a higher magnitude. Therefore, the ”precision” plot of Unum-IV has a sine form as expected.

Figure 5.2 shows that Unum-IV<16,3> has a wider dynamic range and, on average, outperforms the 16-bit floats in terms of accuracy. The ”golden area” covers almost all the exponent range covered by the binary16 format, and out of this zone, the Unum-IV precision suffers a reduction of only one bit.

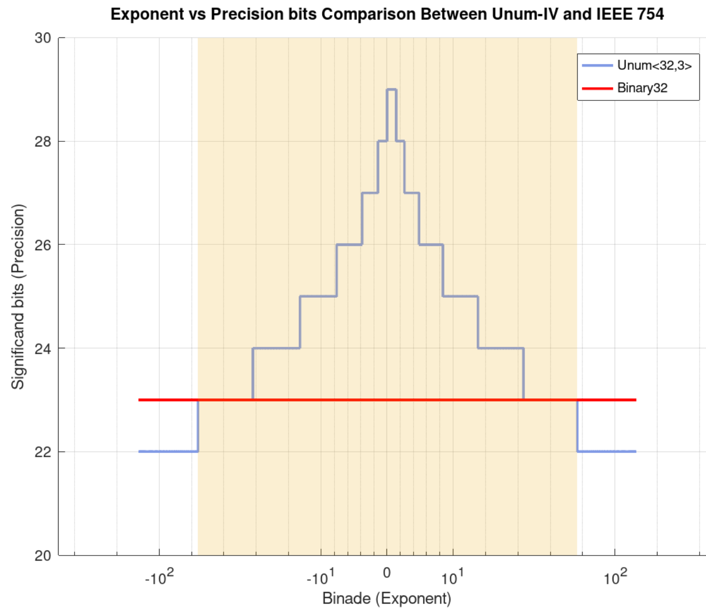


Figure 5.3: Exponent vs Significand Bits Comparison Between Unum-IV<32,3> & 32-bit IEEE 754 Format.

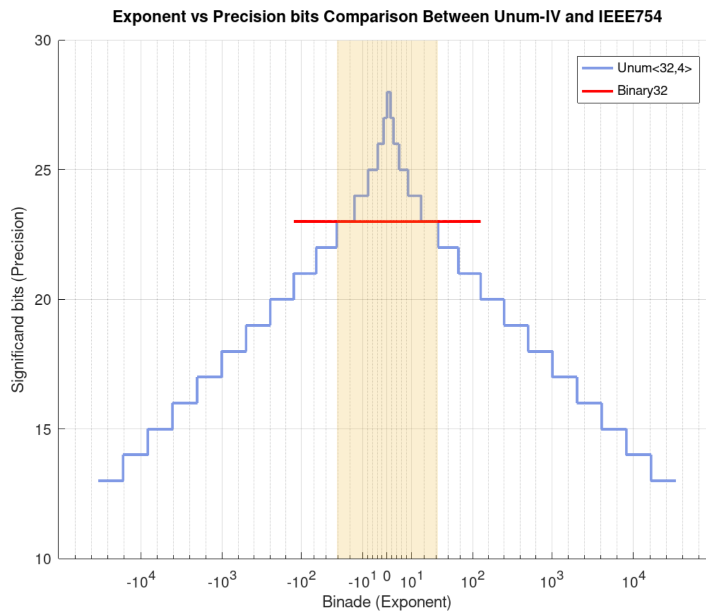


Figure 5.4: Exponent vs Significand Bits Comparison Between Unum-IV<32,4> & 32-bit IEEE 754 Format.

As for the 32-bit floats, Figures 5.3 and 5.4 show that for the same size, both Unum-IV<32,3> and Unum-IV<32,4> have exponent ranges where they have more fraction bits than the binary32 format. The "golden area" occupied by Unum-IV<32,3> is wider than Unum-IV<32,4> because of the smaller dynamic range, having space for more precision bits. Near 0, Unum-IV<32,3> gives a maximum of 29 precision bits, outperforming the 32-bit floats by 6 bits, while Unum-IV<32,4> has just more 5 bits. On the edges where the floats overflow and underflow, outside the golden area, Unum-IV<32,3> has 22 bits, having a smaller resolution by one bit and Unum-IV<32,4> has 21 bits of precision.

Therefore, Unum-IV<32,3> has a similar dynamic range to the 32-bit floats and covers a larger area where numbers have at least the same resolution than floats compared with Unum-IV<32,4>. In terms of accuracy, it is a better option to replace 32-bit floats if the range is enough. The advantage of Unum-IV<32,4> is that besides having, on average, better resolution in the binary32 range than floats, it also has a massive difference in the dynamic range, covering the 64-bit and 128-bit floats ranges.

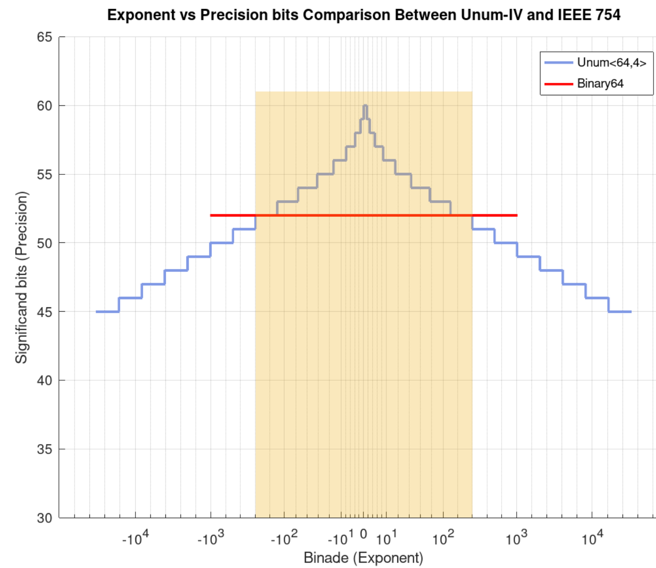


Figure 5.5: Exponent vs Significand Bits Comparison Between Unum-IV<64,4> & 64-bit IEEE 754 Format.

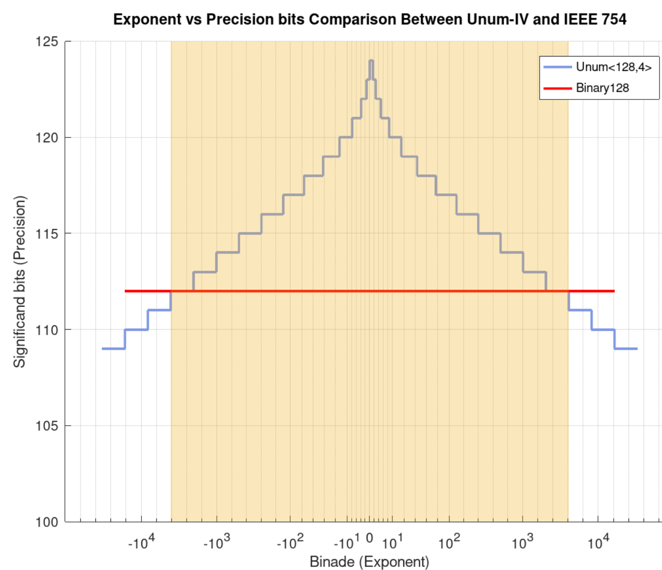


Figure 5.6: Exponent vs Significand Bits Comparison Between Unum-IV<128,4> & 128-bit IEEE 754 Format.

Figure 5.5 shows the comparison between two double-precision formats, the 64-bit float and Unum-IV<64,4>. As can be seen, the 64-bit Unum-IV configuration has a wider dynamic range, covering exponents from -32766 to 32767 and a comfortable ”golden area”, where it can have a higher resolution

than the 64-bit floats with 52 bits of precision. Finally, Figure 5.6 also shows that the Unum-IV<128,4> numbers have the same or better resolution for a vast area of exponents, as well as a greater dynamic range.

Table 5.2: Resolution for Different Unum-IV, Posits and IEEE754 Format Configurations.

Formats	**ULPMin**	**ULPofOne**	**ULPMax**
Binary16	2^{-24}	2^{-10}	2^{5}
Posit<16,1>	2^{-28}	2^{-12}	2^{28}
Unum-IV<16,3>	2^{-132}	2^{-13}	2^{121}
Binary32	2^{-149}	2^{-23}	2^{104}
Posit<32,2>	2^{-120}	2^{-27}	2^{120}
Unum-IV<32,3>	2^{-148}	2^{-29}	2^{105}
Unum-IV<32,4>	2^{-32778}	2^{-28}	2^{32755}
Binary64	2^{-1074}	2^{-52}	2^{971}
Posit<64,3>	2^{-496}	2^{-58}	2^{496}
Unum-IV<64,4>	2^{-32811}	2^{-60}	2^{32722}
Binary128	2^{-16494}	2^{-112}	2^{16271}
Posit<128,4>	2^{-2016}	2^{-118}	2^{2016}
Unum-IV<128,4>	2^{-32875}	2^{-124}	2^{32658}

Table 5.2 displays the resolution of the formats used in Figure 5.1, using the ULP metric. The ULPMin gives the minimum spacing between two consecutive representable values, the ULPofOne gives the weight of the last bit of the significand for numbers with exponent equal to zero (Exponent=0), and the ULPMax gives the maximum spacing. From Table 5.2 it is possible to verify that Unum Type-IV has a better resolution compared with the IEEE 754 and Posits formats using the same bit width.

These comparisons highlight a crucial feature of the Unum-IV, which is the possibility of replacing IEEE 754 formats with Unum-IV formats that use less storage in the computer memory.

5.4.1 Unum-IV<8,2> vs. Quarter-Precision IEEE-Style floats

In Figure 5.7, a quarter-precision IEEE-style float format (not standardized) is tested against Unum-IV<8,2> to compare both formats. These two low precision configurations are selected because they present comparable dynamic ranges and are practical to analyse the whole range since both sets have only 256 elements. The 8-bit IEEE-style format used in this comparison follows the IEEE 754 Standard rules even though this format does not make part of the standard formats. It has a sign bit, a 4-bit exponent and a 3-bit fraction field. It has a total of 14-bit patterns that represent NaN values and a dynamic range of about five decades, where the smallest positive value is $\frac{1}{512}$, and the largest is 240.

The Unum-IV configuration chosen, Unum-IV<8,2>, implies that the format has a total of 8 bits, the exponent ranges between 0 and 6 explicit bits, and the fraction is set between 3 and 6 bits, depending

on the exponent size. Therefore, Unum-IV<8,2> has a dynamic range of about 4.8 decades, where the smallest positive value is also $\frac{1}{512}$ and the largest is 112.

Figure 5.7 shows the application of this metric of study to analyse the accuracy between the positive range of the 8-bit IEEE-style float number system and Unum-IV<8,2>. The xx axis represents the index i of a set of x_i representable values of each format, both ordered from the smallest (minpos) to the largest positive (maxpos) number.

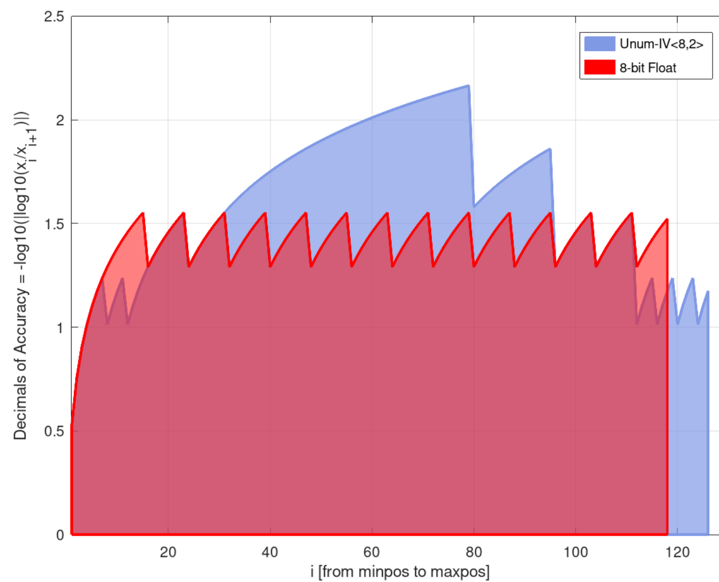


Figure 5.7: Decimals of Accuracy Comparison Between Unum-IV<8,2> & 8-bit Floats.

These graphs reveal that Unum-IV<8,2> is more accurate on average than the 8-bit floats. The results show that Unum-IV<8,2> has a minimum of 0.52, a maximum of 2.17 and an average of 1.58 decimals of accuracy. The 8-bit float format has a minimum of 0.52, a maximum of 1.55 and an average of 1.40 decimals. The decimal accuracy is at the highest in the centre of the graph for Unum-IV, which is where the most common numbers used in the computations occur. As expected, due to the tapered precision of this format, the accuracy tends to decrease in both directions. At the centre of the graph are the numbers with smaller exponents, in terms of magnitude, requiring fewer exponent bits and using more fraction bits. As the numbers run from the centre, the exponent magnitude increases, demanding more exponent bits, which provides less accurate results. Figure 5.7 shows that the floats have tapered accuracy on the left as they use subnormals to obtain a gradual underflow. On the right side, the floats "fall of a cliff" to accommodate all the NaN values (14 in this case). On the other hand, the results show that the Unum-IV format has tapered accuracy on both sides, becoming closer to symmetrically tapered accuracy.

5.4.2 Unum-IV<8,2> vs. Posit<8,1> vs. Posit<8,0>

Here two Posits configurations (Posit<n,es>) are chosen to compare with Unum-IV<8,2>, Posit<8,0> and Posit<8,1>.

Posit<8,1> has a total of 8 (n) bits, a maximum of 1 exponent (es) and of 4 explicit significand bits. Therefore, Posit<8,1> has a dynamic range of about 7.23 decades, where the smallest positive value is also $\frac{1}{4096}$ and the largest is 4096. Posit<8,0> has a total of 8 bits, with 0 exponent bits and maximum of 5 explicit significand bits. Therefore, Posit<8,1> has a dynamic range of about 3.61 decades, where the smallest positive value is also $\frac{1}{64}$ and the largest is 64.

Figure 5.8 shows the application of this metric of study to analyse the accuracy between the positive range of Unum-IV<8,2>, Posit<8,0> and Posit<8,1>.

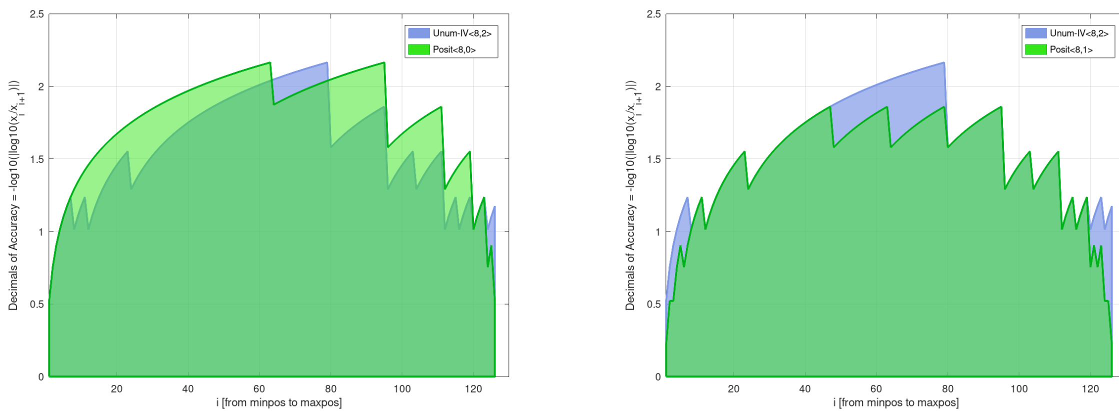


Figure 5.8: Decimals of Accuracy Comparison Between Unum-IV<8,2> & Posit<8,0> (LEFT).Decimals of Accuracy Comparison Between Unum-IV<8,2> & Posit<8,1> (RIGHT).

As can be seen in Figure 5.8, Unum-IV<8,2> is more accurate on average than the Posit<8,1> format due to the higher resolution of Unum-IV<8,2>. It is possible to take that Posit<8,1> has a minimum of 0.22, a maximum of 1.8605 and an average of 1.46 decimals of accuracy. However, Posit<8,1> has a greater dynamic range than the Unum-IV format, with a difference of nearly two decades. As Posit<8,0> is concerned, Unum-IV<8,2> is less accurate because Posit<8,0> format has a higher resolution at the cost of having a smaller dynamic range. Posit<8,0> has a minimum of 0.52, a maximum of 2.17 and an average of 1.7625 decimals of accuracy.

Figure 5.9 shows the ULP variation of the 8-bit float, Posit<8,1> and Unum-IV<8,2> format from the smallest (minpos) to the largest positive representable value (maxpos). The ULP expresses the distance between two consecutive numbers and measures the resolution of a floating-point format.

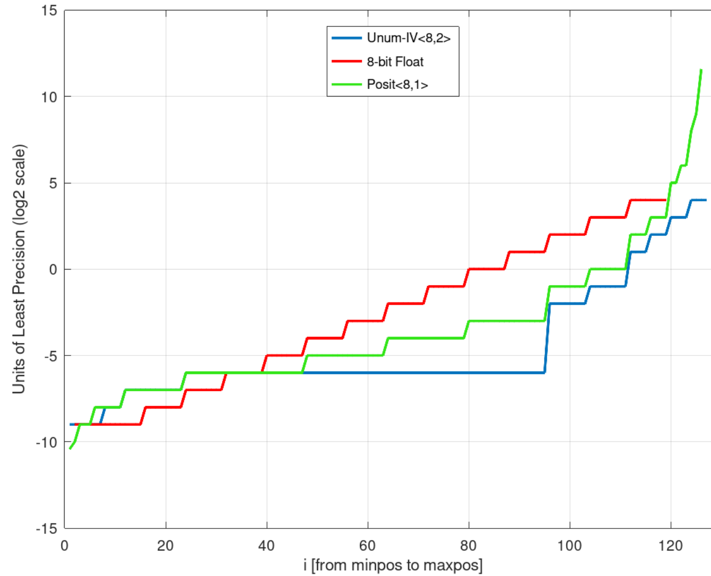


Figure 5.9: ULP Comparison Between Unum-IV<8,2> & 8-bit Floats.

Since the number of precision bits of the 8-bit floats is fixed and equal to 3, the ULP expression is given by $2^{-3} * 2^{e}$, depending exclusively on the exponent. Whereas The Unum-IV<8,2> and Posit<8,1> ULP variation depends both on the exponent and precision. The Unum-IV and Posits ULP variation slows down as the numbers get closer to one because they receive more bits of precision.

From Figure 5.9, it is possible to infer that the Unum-IV<8,2> format has a higher resolution for a considerable portion of the format range because the ULP is smaller, meaning that they have smaller spacing between consecutive floating-point numbers.

5.5 Comparing Unum Type-IV Hardware Resources with Other Formats

5.5.1 IEEE 754 and Unum Type-IV Comparison

In this section, the ASIC implementation results for the new Unum-IV FPU and an existing IEEE 754 FPU, both designed using a UMC 130nm process, are presented and discussed.

Table 5.3 compares the ASIC implementation results of the Unum-IV FPU developed in this dissertation with the IEEE 754 FPU developed by a colleague in the IObundle company [38]. Both Unum-IV FPU and IEEE 754 FPU are parameterizable, so they are compared with different configurations in terms of the silicon area (Area), frequency (Clock Frequency) and power consumption (Power).

The results show that the Unum-IV FPU exponent size(EXP_SZ_W) width configuration influences the power consumption and silicon area. For the same data width (DATA_W), the power consumption and silicon area used grows with the number of exponent size bits, as can be seen in Table 5.3.

The Unum Type-IV configurations selected to implement and compare with IEEE 754 take into account their dynamic range and precision, as they intend to replace the following IEEE 754 formats. The 32-bit IEEE 754 format has an exponent ranging from -126 to 127 and 23 precision bits. For those reasons, the Unum-IV<16,3> and Unum-IV<32,3> are implemented with an EXP_SZ_W=3, to cover the same exponent range. However, the Unum-IV<16,3> precision varies between 6 and 13 bits, whereas Unum-IV<32,3> varies between 22 to 29 bits. On the other hand, the 64-bit IEEE 754 exponent ranges between -1022 to 1023. However, the Unum-IV parameters do not specify this range. The smallest EXP_SZ_W that covers the 64-bit float range is 4, which gives an exponent ranging between -32766 and 32767. Thus, Unum-IV<32,4> and Unum-IV<64,4> configurations are selected. Unum-IV<32,4> is a low-precision version (ranging from 13 to 28 bits) of Unum-IV<64,4> (ranging from 55 to 60 bits).

Another Unum-IV<DATA_W, EXP_SZ_W> configuration that has a minor influence in the implementation results is the Unum-IV FPU rounding mode configuration (Rounding Mode). When the truncation mode is selected (Rounding Mode=0), the silicon area and power consumption are smaller than when using the "round to the nearest, ties even" rounding mode (Rounding Mode =1). These differences are justified because the "round to the nearest, ties even" rounding mode uses extra registers and logic gates to perform rounding operations, and also because all the significand registers have three extra bits.

Table 5.3: ASIC Implementation Results.

FPU	Data Width	ExpSize Width	Rounding Mode	Area [mm^2]	Power [mW]	Clock Frequency [MHz]
Unum-IV	16	3	0	45.19	6.26	200
			1	49.89	7.01	200
	32	3	0	98.31	13.30	200
			1	104.58	14.08	200
		4	0	112.36	14.25	200
			1	123.95	15.38	200
	64	4	0	304.70	40.13	175.19
			1	344.93	38.94	190.25
IEEE 754	32	——	1	67.66	7.44	200
	64	——	1	267.34	27.97	169.15

To make a fair comparison between the Unum-IV FPUs and the IEEE 754 FPUs, they must use the same rounding mode. Therefore, the selected rounding mode to make the comparisons is the "round to the nearest, ties even" because it is the default mode of IEEE 754.

In terms of silicon area, the results show that if the data width is the same between the FPUs, then the IEEE 754 FPUs are smaller than the Unum-IV FPUs. These area differences are explained by the fact that the Unum-IV FPUs exponents and significands are extended to their maximum size allowed by the configuration (EXP_MAX_W and MAN_MAX_W). Thus, all the modules like the barrel shifters and adders will be as large as the configuration parameters. The other reason is that the field extractions require more logic since Unum-IV has variably-sized exponent and significand. Thus, the unpacking and packing require more hardware.

Despite that, the main focus of this dissertation was to develop an FPU with a configurable exponent and significand sizes. It is also relevant to compare the FPUs based on their number system configuration and dynamic range to verify if it was possible and advantageous to replace a format with another using less memory storage.

Let us take Unum-IV<16,3> into consideration. This format has the same exponent range as the single precision IEEE 754 (binary32), having a similar dynamic range. In this scenario, the Unum-IV$< 16,3 >$ FPU is 1,36x smaller than the binary32. Unum-IV<32,4> has a higher dynamic range compared with the double-precision IEEE 754 (binary64), and its FPU is 2,16x smaller.

Table 5.4 shows the silicon area used by each module using a Unum-IV<32,4> configuration.

Table 5.4: Silicon Area of the Different Modules using Unum-IV<32,4>.

Unum-IV<32,4> FPU Modules	**Area (mm^2)**
Unpack A	4.96
Unpack B	4.96
Addition/Subtraction	27.09
Multiplication	37.68
Division	33.16
Pack O	12.49

The processing modules have a similar silicon area use, using more than 81% of the total area. The multiplication module is the largest in terms of the silicon area because it uses the system 64-bit multiplier, which is not that efficient in this terms. The division module has a smaller area because the significand division is performed by an implemented pipelined subtract and shift component, which reduces the silicon area. The pack and unpack modules use almost the same components, with the main difference of the pack module having an extra rounding procedure, increasing its silicon area by $2.5\times$.

As far as power is concerned, the Unum-IV FPUs consumes more than the IEEE FPUs for the same data width. However, if the comparison is made based on formats with similar dynamic ranges, then the Unum-IV FPUs consumes less or almost the same power. For the same examples used for the silicon area comparison, it is possible to verify that the power consumption of the Unum-IV<16,3> FPU is slightly smaller (7.01 mW) than the binary32 FPU (7.44 mW). The other example is between binary64 and Unum-IV<32,4>, where the Unum-IV configuration has a higher dynamic range compared to binary64 and where the binary64 FPU consumes 1,82x more power.

Both FPUs make use of the pipeline to maximize the clock frequency, and their frequency is comparable. However, there is still room for optimizations as the double precision is not maximized to 200 MHz.

5.5.2 Posits and Unum Type-IV Comparison

In the comparison between the Unum-IV and Posits hardware resources utilization, the published FPGA results in [39]are used to estimate the ASIC area of a Posits FPU. In that article, a pipelined FPU with an adder and multiplier is implemented in an FPGA using the Posits standard configurations.

Unum Type-IV and Posits with the same bit-width and without the division unit are compared using FPUs that implement the multiplication and addition/subtraction, using pipeline and the same rounding mode to ensure that the comparison is fair. The FPGA sysnthesis results are first shown in Table 5.5.

Table 5.5: FPGA Results Comparison Between Different Unum-IV and Posits Configurations.

Format	Configuration	LUTs	FFs	DSPs	Estimated Gates
Posit	<16,1>	533	208	1	6205
	<32,2>	1162	658	4	19100
	<64,3>	2775	2018	16	63965
Unum-IV	<16,3>	721	517	1	8690
	<32,3>	1288	854	4	20710
	<64,4>	2854	1651	10	47525

The FPGA synthesis results clearly show that the Posits and Unum-IV implementations have a similar size. From these results, the silicon area of a Posits FPU can be estimated. First the number of NAND2 equivalent system gates is estimated from the FPGA results for the Posits FPU, assuming 6 gates per LUT and 2500 gates for each DSP unit. Then the silicon area of the Posits FPU is calcuated based on the area of a NAND2 gate.

Table 5.6 compares the silicon area of the Unum-IV FPU to the estimated silicon area of the Posits FPU for the following standard configurations: Posit<16,1>, Posit<32,2> and Posit<64,3>. For the Unum-IV, the ASIC results are obtained by directly implementing the Unum-IV<16,3>, Unum-IV<32,3> and Unum-IV<64,4> FPU configurations, using a UMC 130nm process. It could be shown that the silicon area estimated from the Unum-IV FPU FPGA results and its actual silicon area are similar. However, the actual results are shown since they are available.

Table 5.6: Silicon Area Comparisons Between Different Unum-IV and Posits Configurations.

Format	Configuration	Maximum Precision [Bits]	Dynamic Range [Decades]	Estimated ASIC Area [mm^2]	ASIC Area [mm^2]
Posit	<16,1>	12	16.86	31.77	—
	<32,2>	27	72.25	97.79	—
	<64,3>	58	298.62	327.5	—
Unum-IV	<16,3>	13	77.96	—	34.42
	<32,3>	29	82.78	—	74.14
	<64,4>	60	19740.9	—	240.46

The results in Table 5.6 show that the Unum-IV area is similar or smaller than the Posits area. Given that the accuracy and dynamic range of the Unum-IV has been shown superior to those of the Posits,

one concludes that the Unum-IV format is a better replacement for the IEEE 754 format than the Posits.

For the same bit width, Unum-IV<16,3> has a similar area, a wider dynamic range by almost 30 decades and a greater maximum number of precision bits compared with Posit<16,1>. Unum-IV<32,3> uses 25% less area than Posit<32,2> while having a similar dynamic range and supporting more precision bits. Finally, Unum-IV<64,4> uses nearly 36% less area, has $66x$ more decades of dynamic range than the Posit<64,3> configuration, and also supports more precision bits.

5.6 Comparison Summary

These metrics show that Unum-IV configurations can translate to more accurate computation due to their variable precision. In general, Unum-IV formats have a ”golden area”, where they have as many precision bits as floats of the same width, with the advantage of also having greater dynamic ranges. This can be advantageous for applications like machine learning applications. Studies [32, 33, 34] show that many ML algorithms can tolerate formats with lower precision without ruining the results. For example, Google developed a 16-bit format called ”Brain Floating Point”, shortly, ”Bfloat16” [28]. This idea came from ”Google Brain”, an artificial intelligence research group from Google. They pointed that the 16-bit float format does not have enough dynamic range for most deep learning applications, so they created the ”Bfloat16”, which solves the range problem by having a similar dynamic range as 32-bit floats do.

Table 5.7 compares the single-precision IEEE 754 format with the ”Bfloat16”, the Unum-IV<16,3> and the Posit<16,3> format. The four formats are compared in terms of exponent size, fraction size, dynamic range and resolution.

Table 5.7: Binary32, Bfloat16, Posit<16,3> and Unum-IV<16,3> Comparisons.

Formats	Exponent [Bits]	Significand [Bits]	minpos (≈)	maxpos (≈)	Dynamic Range [Decades] (≈)	ULP Min	ULP ofOne	ULP Max
Binary32	8	23 explicit + 1 implicit	1.40×10^{-45}	3.40×10^{38}	83.39	2^{-149}	2^{-23}	2^{104}
Bfloat16	8	10 explicit + 1 implicit	1.15×10^{-41}	3.40×10^{38}	83.47	2^{-136}	2^{-10}	2^{117}
Posit<16,3>	[0-3]	[0-10] explicit + 1 implicit	1.93×10^{-34}	5.19×10^{33}	67.43	2^{-112}	2^{-10}	2^{112}
Unum-IV<16,3>	[0-7] explicit + 1 implicit	[6-13] explicit + 1 implicit	1.43×10^{-42}	1.70×10^{38}	77.96	2^{-132}	2^{-13}	2^{121}

As ”Bfloat16”, Unum-IV<16,3> has an identical dynamic range as 32-bit floats using half of the memory size. The main difference between these two formats is the tapered precision attached to the Unum-IV format, whereas ”Bfloat16” has a fixed 8-bit fraction field, Unum-IV<16,3> has a fraction field that floats between 6 and 13 fraction bits. Thus, the Unum-IV has a range of numbers where it outperforms the brain floats in precision terms. In the worst-case scenario, where Unum-IV has less resolution than ”Bfloat16”, they only have an extra fraction bit. As for the best-case (numbers near 1), Unum-IV has an addition of six extra fraction bits. For those reasons, as ”Bfloat16” appeared to replace the 32-bit floats in machine learning applications to increase performance and reduce memory usage, Unum-IV<16,3> can also be a valuable alternative to the ”Bfloat16”, producing better or the same results.

Posit<16,3> has both a smaller dynamic range and resolution compared with Unum-IV<16,3>. This

format can be efficiently replaced by the proposed Unum-IV<16,3> format, producing results at least in the same or in a more extensive range of values without losing precision.

In Chapter 6, a machine learning application is tested using Unum-IV<32,4> against the 32-bit floats to verify the Unum-IV advantages over IEEE 754 format, such as higher dynamic range and accuracy.

Chapter 6

Proof of Concept: KNN Application

In this chapter, a K-Nearest Neighbours (KNN) application [31] is implemented using three different data types: the IEEE 754 double-precision format, which is used as a reference, the IEEE 754 single-precision format, and the Unum-IV<32,4> format. Unum-IV<32,4> is suitable to be used in ML applications such as the KNN and tested against the 32-bit IEEE 754 floats because these types of applications use a high amount of floating-point operations and have a level of accuracy tolerance that others do not. It is expected that Unum-IV<32,4> exceed both 32-bit and 64-bit floats in terms of dynamic range. In terms of accuracy, the variable-precision format should also exceed the 32-bit floats for numbers near 1. Therefore, two different tests are made to compare the accuracy of the classification results for benchmarks that cover those features, using the 64-bit float results as a reference.

The K-Nearest Neighbours [40, 41] algorithm is one of the simplest supervised ML algorithms, which can be used for both classification, regression and search problems. This type of supervised ML algorithms uses labelled data to produce a method to predict the label of given unlabeled data. The KNN algorithm is simple, easy to understand. It is one of the most used algorithms in ML applications and is also robust to the noisy training data and non-parametric, as there is no assumption for underlying data distribution.

The algorithm finds the K closest labelled data to the data point to be classified using the distance as criteria, and then it predicts the test point classification by the majority class voted by its K neighbours. Therefore, the class with the most votes is held as the predicted class of the unlabeled data point.

The number of neighbours is user-defined and very important because as the number of K increases up to a certain point, the predictions become more stable and accurate.

6.1 Algorithm

The KNN algorithm follows four main steps:

- Init Step: firstly, all the dataset structures are initiated with randomly generated coordinates and labels. Then, all the set of data points to be classified are assigned random coordinates. The goal of the algorithm is to predict the class label of each of those test points.
- Calculation Step: for all the test points to be classified, the distance to each dataset point is computed. The metric used is the square distance between two points a and b, which is the Squared Euclidean Distance (SED). The Euclidean Distance (ED) and the Manhattan Distance (MD) are the most common metrics applied in this kind of applications. The first step to employing the SED metric is to initiate all the K neighbours with infinite distance. Then, for all the dataset points, the distance to each test point is computed.
- Insertion Step: for each test point, the distances are inserted in an ordered array of neighbours.
- Classification Step: the neighbours vote, and the best voted data class is assigned to each test point label.

The algorithm ends when all test points are classified.

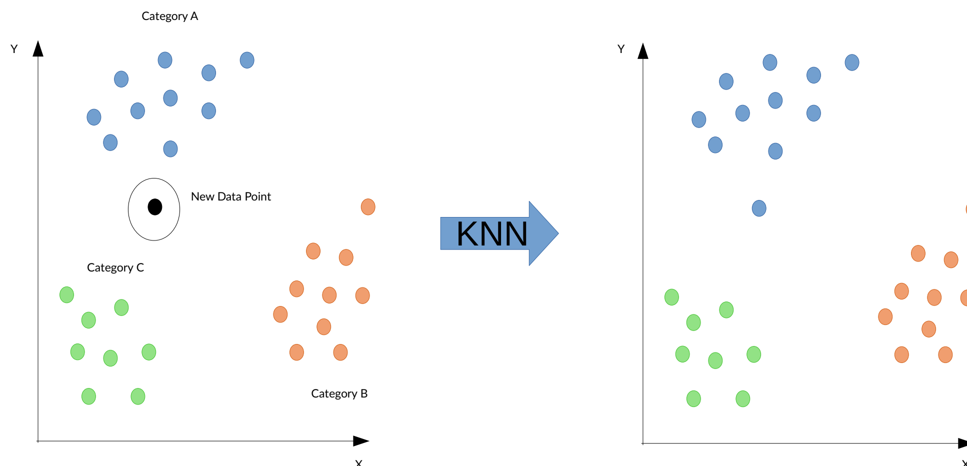


Figure 6.1: KNN Application.

In the KNN task:

- N: the dataset size;
- K: the number of neighbours;
- C: the number of data classes;
- M: the number of samples to be classified, the test points.

6.2 Implementation

The software code that performs the KNN algorithm given in the previous subsection is implemented for three different data types: the double, the float and the Unum-IV<32,4> number system. Float is a data type used to represent the floating-point numbers, given by the 32-bit IEEE 754 single-precision floating-point numbers. Double is also a data type used to express the 64-bit IEEE 754 double-precision floating-point numbers. These two are defined in the C standard library.

To compare the accuracy of the classification results using different data types, we also implemented the Unum-IV FPU in software, with the addition of the data type conversion between the reference datatype (Double) and the Unum-IV<32,4>. Thus an Unum-IV library is implemented, where the data type is defined as an integer with 32 bits and the unpacked fields of Unum-IV are defined as a structure (exponent, significand and exponent size) in the Unum-IV header file shown in Listing 6.1.

Listing 6.1: Unum-IV and KNN Header File.

```
//Define
...
#include <stdint.h>
//Typedef Unum-IV Format

typedef struct {
  int32_t exponent;
  int32_t significand;
  int32_t exp_size;
  }Unum4Unpacked;
typedef int32_t unum4;

// Unum-IV Functions

Unum4Unpacked unum4_unpack(unum4 input);
Unum4Unpacked unum4_add_sub(Unum4Unpacked a, Unum4Unpacked b, int32_t*overflow, int32_t op);
Unum4Unpacked unum4_mul(Unum4Unpacked a, Unum4Unpacked b, int32_t*overflow, int32_t*underflow);
Unum4Unpacked unum4_div(Unum4Unpacked a, Unum4Unpacked b, int32_t*overflow, int32_t*underflow,
 int32_t*div_by_zero);
unum4 unum4_pack(Unum4Unpacked o, int32_t*overflow);
unum4 double2unum4(int64_t input, int32_t*failed);
unum4 float2unum4(int32_t input, int32_t*failed);

//Knn Functions

void knn_double (double random[], double test_points[], unsigned char label_rand[],
int votes_acc[], int double_class[]);
void knn_unum4(double random[], double test_points[], unsigned char label_rand[],
int votes_acc[], int unum4_class[]);
void knn_float(double random[], double test_points[], unsigned char label_rand[],
int votes_acc[], int float_class[]);
```

The implemented Unum-IV library has six functions. The functions are the following:

- **unum4_unpack()**: function that unpacks a 32-bit Unum-IV strings into a structure that contemplates the three Unum-IV fields, exponent, significand and exp_size, respectively.

- **unum4_add_sub()**: function that performs an addition or subtraction between two unpacked 32-bit Unum-IV structures. The function returns the Unum-IV unpacked structure as the result of the operation with an overflow flag.

- **unum4_mul()**: function used to perform a multiplication between two unpacked 32-bit Unum-IV structures. The function returns the Unum-IV unpacked structure and the overflow and underflow flags.

- **unum4_div()**: function that perform a division between two unpacked 32-bit Unum-IV structures. The function returns the same structure and flags of the multiplication with a "divide by zero" extra flag.

- **unum4_pack()**: a function that receives the Unum-IV unpacked structure returned from the previous functions, rounds the result using the "Round to the Nearest, Ties Even" mode and packs the result into a 32-bit Unum-IV string. This function returns the 32-bit Unum-IV string and the rounding overflow flag.

- **double2unum4()**: function that performs the conversion from a 64-bit IEEE double-precision floating-point number to a 32-bit Unum-IV<32,4> number. Therefore, the function returns a 32-bit Unum-IV string and a "failed" flag. The "failed" flag is set to 1 if the conversion fails.

- **float2unum4()**: function that performs the conversion from a 32-bit IEEE double-precision floating-point number to a 32-bit Unum-IV<32,4> number. Therefore, the function returns a 32-bit Unum-IV string with the "failed" flag.

After the Unum-IV library is implemented in C language, the KNN application is implemented for doubles, floats and the Unum-IV<32,4> number formats. As can be seen in Listing 6.1, there are three functions:

- **knn_double()**: the function that performs the KNN algorithm using doubles as the datatype. The classification results of this function are used as a reference for the other data types.

- **knn_unum4()**: the function that performs the KNN algorithm using the Unum-IV<32,4> format as the datatype. The classification results of these functions are evaluated in terms of accuracy against the 32-bit floats.

- **knn_float()**: function that performs the KNN algorithm using 32-bit floats as the datatype. This function is implemented the same way as the 64-bit double format.

In the main function, the three last functions are called for the same labeled data points (random[]) and unlabeled data points (random[]). All the data points are generated using the double random generator function shown in Listing 6.2.

Listing 6.2: Double Random Generator.

```
//Double Random Generator
double random_double(double min, double max) {
  double d = (double) rand() / ((double) RAND_MAX + 1);
  return (min+ d*(max-min));
}
```

Since the data is randomly generated using the double data type, it is converted to the respective format inside each KNN function. For the floats, there is no cast needed to convert the doubles into floats, whereas for the Unum-IV<32,4> the *double2unum4()* function is called to convert each data point before performing the algorithm described in the previous section.

The labelled dataset and the neighbour information are assigned to specific structures as shown in Listing 6.3. In the labeled dataset structure, the x and y coordinates are stored along with the class label of each data point. The neighbour information, composed by the distance to the test data point and the index in the dataset array, is inserted in the ordered neighbour structure of that specific test point.

In Listing 6.3, only the Unum-IV format structures are shown because they are implemented the same way in the other datatypes. The main difference is in the data types used for the two-dimensional coordinates and the distance.

Listing 6.3: Data Structures.

```
//labeled dataset (unum4)
struct datum_unum4 {
  unum4 x;
  unum4 y;
  unsigned char label;
} data_unum4[N], x_unum4[M];

//neighbour info (unum4)
struct neighbor_unum4 {
  unsigned int idx; //index in dataset array
  unum4 dist; //distance to test point
} neighbor_unum4[K];
```

As mentioned before, different distance metrics can be implemented using the KNN algorithm. However, the Squared Euclidean Distance is selected as a metric because there is no need to square root the distances to compare them and, also, because the Unum-IV FPU has only four basic two-argument operations (addition, subtraction, division and multiplication).

The Squares Euclidean Distance (SED) for two dimensional (A,B) problems is given by

$$SED = (A_x - B_x)^2 + (A_y + B_y)^2 \tag{6.1}$$

Listing 6.4 shows the difference in the SED implementation using floats or doubles, both defined in the C standard library, and the implementation using the Unum-IV library developed in this dissertation. Three steps need to be followed to perform an Unum-IV operation: the unpacking of the operands, the

operation execution and the packing of the result into the 32-bit Unum-IV string.

Listing 6.4: Square Distance Functions.

```
//square distance between 2 points a and b (float)
float sq_dist_float( struct datum_float a, struct datum_float b) {
  float X = a.x-b.x;
  float X2=X*X;
  float Y = a.y-b.y;
  float Y2=Y*Y;
  return (X2 + Y2);
}
//square distance between 2 points a and b (unum4)
unum4 sq_dist_unum4( struct datum_unum4 a, struct datum_unum4 b) {
  int32_t overflow=0, underflow =0;
  Unum4Unpacked X = unum4_add_sub(unum4_unpack(a.x), unum4_unpack(b.x), &overflow, 1);
  unum4 x = unum4_pack(X,&overflow); //a.x-b.x
  Unum4Unpacked X2= unum4_mul(unum4_unpack(x),unum4_unpack(x), &overflow, &underflow);
  unum4 x2 = unum4_pack(X2,&overflow); //x^2
  Unum4Unpacked Y = unum4_add_sub(unum4_unpack(a.y), unum4_unpack(b.y), &overflow, 1);
  unum4 y = unum4_pack(Y,&overflow); //a.y-b.y
  Unum4Unpacked Y2=unum4_mul(unum4_unpack(y),unum4_unpack(y), &overflow, &underflow);
  unum4 y2 = unum4_pack(Y2,&overflow); //y^2
  return unum4_pack(unum4_add_sub(unum4_unpack(x2),unum4_unpack(y2), &overflow, 0), &overflow);
}
```

As for the insertion step, Listing 6.5 shows how the insertion of the neighbour's distance and index into the ordered neighbour's structure is implemented for the Unum-IV numbers. Every time a distance of a dataset point to a test point is computed, that dataset point is compared against the neighbours already inside the neighbour's structure of that specific test point.

Listing 6.5: Insertion in Ordered List.

```
  for (int i=0; i<N; i++) { //for all dataset points
      //compute distance to x[k]
      unum4 d = sq_dist_unum4(x_unum4[k], data_unum4[i]);
      //insert in ordered list
      for (int j=0; j<K; j++) {
        dist = unum4_add_sub(unum4_unpack(d),unum4_unpack(neighbor_unum4[j].dist), &overflow,1);
        if (~overflow && dist.significand <0){
          insert_unum4( (struct neighbor_unum4){i,d}, j);
          break;
        }
      }
```

If a dataset point has a smaller distance to a test point than the dataset points saved in that test point neighbours structure, then the neighbour with the largest distance to the test point is replaced by that dataset point. Therefore, when all the dataset points distances are computed and inserted in the neighbour's list, the neighbour's vote and the classification of the test point is predicted.

In the classification step, the neighbours of each test point vote in their classes and the best-voted is assigned as the test point class as shown in Listing 6.6.

Listing 6.6: Classification step code.

```
//classification of each test point
    //clear all votes
    int votes[C] = {0};
    int best_votation = 0;
    int best_voted = 0;

    //make neighbours vote
    for (int j=0; j<K; j++) { //for all neighbors
      if ( (++votes[data_float[neighbor_float[j].idx].label]) > best_votation ) {
        best_voted = data_float[neighbor_float[j].idx].label;
        best_votation = votes[best_voted];
      }
    }
```

6.3 Experimental Results

The proof of concept application was run for two different sets of benchmarks, using the parameters shown in Table 6.1. The number of neighbours is defined as 10 to provide a more accurate classification, as explained before. To compare the performance between Unum-IV<32,4> and the IEEE single-precision floating-point format, we use the IEEE double-precision format results as a reference.

Table 6.1: Parameters used in the KNN Clustering Application.

Data Set Size [N]	Number of Neighbours [K]	Number of Data Classes [C]	Number of Samples to be Classified [M]
100000	10	4	100

6.3.1 Experiment 1

In the first test, ten different benchmarks are randomly generated. The dataset points range between 0.99999 and 1, and the test points range between 0.9 and 1. This setup provides sparsely dispersed datasets. These benchmarks are suitable to compare and verify if Unum-IV and 32-bit floats have enough resolution to give accurate answers for numbers near 1.

Since the 32-bit floats have a 23-bit significand, the effective resolution of the format lies between six and seven decimal fractional digits. Hence, it is expected that the 32-bit floats might not have enough resolution to give accurate classifications to the test points. On the other hand, for numbers near 1, Unum-IV<32,4> can have a maximum of 28 bits of significand. Therefore, the Unum-IV format can lead to more accurate classifications than 32-bit floats because it supports 6 to 9 decimal fractional digits of resolution.

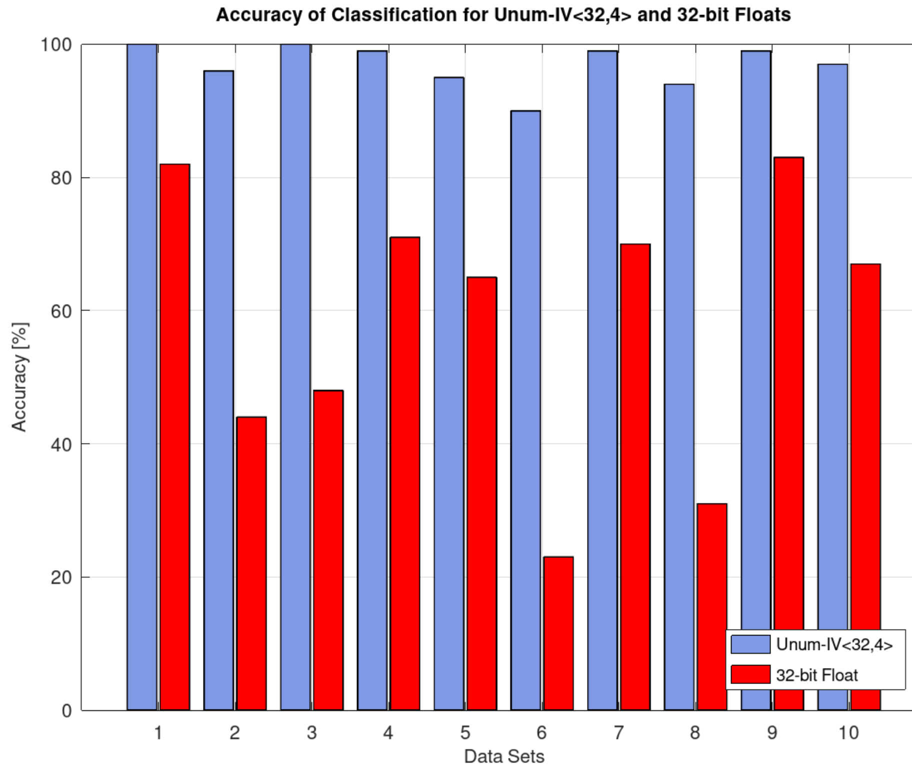


Figure 6.2: Accuracy of Classification for Unum-IV<32,4> and 32-bit Floats.

The results of the first test show that Unum-IV<32,4> produced more accurate classifications than the 32-bit floats for all the ten benchmarks, meaning that they have a higher percentage of correct classifications. The accuracy of Unum-IV is set between 90 to 100 per cent, while the 32-bit floats can only afford results between 23 and 83 per cent of accuracy compared with the 64-bit floats classifications. As expected, Unum-IV<32,4> outperforms the 32-bit floats for numbers with small magnitude.

6.3.2 Experiment 2

In the second test, another ten different benchmarks are randomly generated. However, the dataset points and test points generation use a wider range than the previous test, ranging between 0 and 10^{22}. The main focus of this test is to compare the Unum-IV<32,4> classifications with the results of the 64-bit float. It is expected that in this test, both formats provide similar classifications, considering their dynamic ranges.

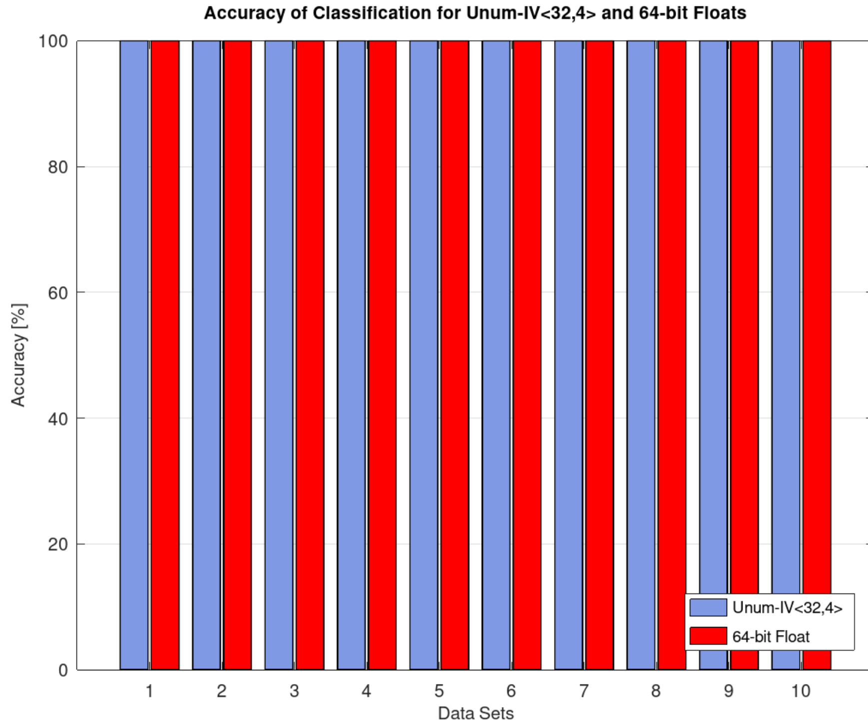


Figure 6.3: Accuracy of Classification for Unum-IV<32,4> and 64-bit Floats.

All the data used in the set of benchmarks are supported by the 32-bit floats, which, nonetheless, may not apply to the square distance between the labelled and unlabeled points, as the computed distance might fall outside the range of the 32-bit floats. For this reason the 32-bit floats have a poor performance, as most of the computed distances overflow, resulting in less accurate classifications. On the other side, we have the Unum-IV format performs correctly in all the benchmarks, which is easily explained by the fact that the Unum-IV$< 32, 4 >$ has a greater dynamic range than 32-bit and 64-bit floats. Another note that can be added to these results is that the behaviour of floats towards the overflow of assigning it to the $+\infty$ can cause mathematical incongruities, as the operations are not interrupted.

Finally, both experimental results show that Unum-IV<32,4> can indeed produce more accurate results than the IEEE 754 single-precision floating-point format for numbers with a small magnitude but also cover all the IEEE 754 double-precision dynamic range with very accurate results. This means that for these type of applications that tolerate some accuracy loss, the 32-bit Unum-IV<32,4> can be a compelling replacement for the 64-bit IEEE 754 format. Unum-IV<32,4> has a wider dynamic range than the 64-bit IEEE 754 using half of the computer memory for the format and can provide near-one results with higher accuracy than the IEEE 754 format with the same word size.

The implementation results obtained in Chapter 4 showed that the Unum-IV FPU using the Unum-IV<32,4> configuration uses nearly half of the silicon area and power consumption than the IEEE 754 double-precision FPU.

With the addition of the conclusions obtained with these two experiments in terms of accuracy, it is possible to conclude that Unum-IV<32,4> can satisfy the application requirements while guaranteeing computational efficiency and lowering the power.

Chapter 7

Conclusions

In this dissertation, Unum Type-IV, a new floating-point number system, is proposed, implemented, tested and compared with the IEEE Standard for Floating-Point Arithmetic, and with the Unum Type-III format, also known as Posits, proposed by John L. Gustafson in 2017 [15].

For over 30 years, the IEEE Standard for Floating-Point Arithmetic (IEEE 754) has been the most widely used format to represent floating-point numbers in computer systems. However, the IEEE 754 has some limitations and drawbacks that have been pointed out by John L. Gustafson, who proposed the Unum Type-I and Unum Type-II formats [6] before proposing Posits.

Like Posits, the proposed Unum Type-IV number system intends to be a suitable drop-in replacement for IEEE 754 floats, targeting low-power consumption applications. The new representation format was named Unum Type-IV because it extends and recovers ideas from the previous Unum formats, and improves upon them. Like its predecessors the Unum Type-IV is a tapered-accuracy numerical format, where numbers that are close to 1 get more significand bits and fewer exponent bits, and very large or very small magnitude numbers get more exponet bits and fewer significand bits.

Unum Type-III or Posits has been so far the most interesting of these proposals, and has gained considerable support in the community. It has been the most viable format for replacing IEEE 754, and many research papers present studies on its properties and possible hardware implementations, which, by the way, are quite competitive with IEEE 754 hardware implementations. Posits' hardware is about 50% larger than IEEE 754 hardware for the same bit width, but Posits offer a lot more precision and dynamic range than same size IEEE 754 numbers. Moreover, smaller-size Posits can replace floats in many applications, making them look interesting even from the hardware size perspective.

The new Unum-IV proposal considerably improves the precision and dynamic range properties of Posits, lowering the barrier to overthrow the IEEE 754 format even more. Moreover, its hardware is competitive with the Unum-III hardware.

7.1 Achievements

In this work, we introduced a new floating-point number system, which can replace the IEEE 754 format offering significantly more precision and dynamic range, and beating the previous Unum-III proposal in terms of precision bits and dynamic range decades per logic gate.

The first step to achieve this goal involved studying the state-of-the-art of the IEEE 754 Standard and the different Unum formats to understand their drawbacks and how the new proposal could add value.

Then, the Unum Type-IV format is proposed, which introduces variably-sized exponent and significand scheme that effectively increases the dynamic range and accuracy compared to Unum-III. The new scheme is based on a dynamic hidden bit for the 1's complement exponent and another dynamic hidden bit for the 2's complement significand, that dispenses with the use of a unary representation for the super exponent called Regime bits in Unum-III (Posits). Compared to Posits, accuracy and dynamic range is increased by using a binary representation for both the exponent and the significand.

A parametrizable Unum-IV Floating-Point Unit (FPU) is developed in Verilog and implemented in FPGA and ASIC technology and tested in different format configurations. The new FPU is compared with IEEE 754 and Posits FPUs in terms of the used silicon. The new FPU has three hardware levels, unpacking, processing and packing, and includes four basic two-argument operations: addition, subtraction, division and multiplication.

A corresponding parametrizable IEEE 754 FPU is also implemented in FPGA and ASIC technology, and the synthesis results show that, for same bit width configurations, the area and power consumption of the Unum-IV FPU is lower. Published results on a Posits FPU have been used for comparison. For equivalent hardware size, Unum-IV has much more maximum precision than IEEE 754 and more maximum precision than Posits. For equivalent hardware size, Unum-IV has much more dynamic range decades than IEEE 754 and more dynamic range decades than Posits. This shows that Unum-IV has a better performance in terms of power consumption and silicon area than IEEE 754 and Posits.

For example, the 32-bit Unum-IV$<$32,4$>$ configuration has 30x the dynamic range in decades compared with 64-bit floats while using 2.1569x less silicon area with 1.8189x less power consumption with a higher maximum clock frequency (200 MHz against 169.15 MHz of the 64-bit floats), using half of the computer memory for the format. This means Unum-IV$<$32,4$>$ can produce similar results as when using 64-bit floats using half the memory and about 30% less silicon.

For the same bit width, the Unum-IV$<$32,4$>$ has a 237x larger dynamic range compared with the 32-bit floats. In terms of accuracy, it can outperform the 32-bit floats as the Unum-IV fraction bits float between 28 and 21 bits, whereas the 32-bit floats have a fixed fraction with 23 bits. Thus, in the best scenario, the Unum-IV$<$32,4$>$ produces answers with five additional accuracy bits, and in the worst case, with fewer two bits of accuracy.

For Posits, the 64-bit Unum-IV$<$64,4$>$ configuration has a dynamic range with more 19442 decades and a significand with 2 extra precision bits compared with Posit$<$64,3$>$ while using 1.32x less silicon area, for example. This means the Unum-IV$<$64,4$>$ can produce better results in terms of the dynamic range and precision than the 64-bit Posit$<$64,3$>$, while using about 32% less silicon.

For equivalent hardware size, Unum-IV<32,3> has more 2 precision bits and about 10 more dynamic range decades than Posit<32,2> format with the same bit width.

Finally, the Unum-IV<32,4> FPU was implemented in C language, and a KNN application was used as a proof of concept to compare the performance of the Unum-IV<32,4> and the 32-bit floats in two different sets of tests, using the 64-bit float results as a reference. As expected, the results showed that the Unum-IV outperforms the 32-bit floats and produces similar results to the 64-bit floats.

It is concluded that the initial goals of this dissertation have been achieved: the proposed floating-point system can be a suitable replacement for the IEEE 754, in particular for the area of HPC and low precision applications, and beats its main competitor, Unum-III (Posits).

7.2 Future Work

As a follow-up to this work, some ideas to improve the proposed numerical system and its performance can be studied and applied. For instance, we provide a few examples are advanced here:

- Use hardware description of a Posits FPU for simulation, synthesis and FPGA testing. In this work published results are used, which affects the precision of the comparisons and reduces the flexibility in devising new experiments.
- Using a variable-precision floating-point arithmetic function (VPA) to evaluate the results obtained with both Unum-IV<32,4> and 32-bit floats, instead of using 64-bit floats as reference. With this, it is possible to gather better evaluations of the performance in terms of accuracy, especially between Unum-IV<32,4> and doubles.
- Adding fused operations in the Unum-IV FPU, such as multiply-add, add-multiply, sum and dot product. These fused operations prevent the rounding error from accumulating by deferring the rounding step until the last iteration when the computation involves more than one operation. Hence, the results could be more accurate and, for example, it may allow for the Unum-IV<32,4> format to safely replace the 64-bit floats in HPC.
- Incorporating the Unum-IV FPU in libraries such as Tensorflow or Keras to increase the testing scale and the performance analysis of the new number system in different datasets and architectures.
- Lastly, as the synthesis results showed, there is still plenty of room for hardware optimization in the parametrizable Unum Type-IV FPU to improve the area, power consumption and clock frequency.

Bibliography

[1] D. Goldberg. What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991. ISSN 0360-0300. doi: 10.1145/103162.103163. URL https://doi.org/10.1145/103162.103163.

[2] A. Di Franco, H. Guo, and C. Rubio-González. A Comprehensive Study of Real-World Numerical Bug Characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 509–519, 2017. doi: 10.1109/ASE.2017.8115662.

[3] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985. doi: 10.1109/IEEESTD.1985.82928.

[4] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008. doi: 10.1109/IEEESTD.2008.4610935.

[5] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[6] J. Gustafson. *The End of Error: Unum Computing*. 02 2015. ISBN 1482239868. doi: 10.1201/9781315161532.

[7] W. Kahan and J. Darcy. How Java's floating-point hurts everyone everywhere. *... 1998 Workshop on Java ...*, pages 1–81, 1998. URL https://people.eecs.berkeley.edu/~wkahan/JAVAhurt.pdf.

[8] E. Ternovoy, M. G. Popov, D. V. Kaleev, Y. V. Savchenko, and A. L. Pereverzev. Comparative Analysis of Floating-point Accuracy of IEEE 754 and Posit Standards. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 1883–186, 2020. doi: 10.1109/EIConRus49466.2020.9039521.

[9] R. Morris. Tapered Floating Point: A New Floating-Point Representation. *IEEE Transactions on Computers*, C-20(12):1578–1579, 1971. doi: 10.1109/T-C.1971.223174.

[10] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating Scientific Computations with Mixed Precision Algorithms. *Computer Physics Communications*, 180(12):2526–2533, Dec 2009. ISSN 0010-4655. doi: 10.1016/j.cpc.2008.11.005. URL http://dx.doi.org/10.1016/j.cpc.2008.11.005.

[11] P. Lindstrom, S. Lloyd, and J. Hittinger. Universal Coding of the Reals: Alternatives to IEEE Floating Point. In *Proceedings of the Conference for Next Generation Arithmetic*. Association for Computing Machinery, 2018. ISBN 9781450364140. doi: 10.1145/3190339.3190344. URL https://doi.org/10.1145/3190339.3190344.

[12] E. Morancho. Unum: Adaptive Floating-Point Arithmetic. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 651–656, 2016. doi: 10.1109/DSD.2016.39.

[13] J. L. Gustafson. A Radical Approach to Computation with Real Numbers. *Supercomputing Frontiers and Innovations*, 3(2), 2016. ISSN 2313-8734. URL https://superfri.org/superfri/article/view/94.

[14] W. Tichy. Unums 2.0: An Interview with John L. Gustafson. *Ubiquity*, 2016:1–16, 10 2016. doi: 10.1145/3001758.

[15] J. Gustafson and I. Yonemoto. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, 4:71–86, 01 2017. doi: 10.14529/jsfi170206.

[16] J. L. Gustafson. Posit arithmetic, [Online], 2017. URL https://posithub.org/docs/Posits4.pdf.

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

[18] A. Podobas and S. Matsuoka. Hardware Implementation of POSITs and Their Application in FPGAs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 138–145, 2018. doi: 10.1109/IPDPSW.2018.00029.

[19] M. Klöwer, P. D. Düben, and T. N. Palmer. Posits as an Alternative to Floats for Weather and Climate Models. In *Proceedings of the Conference for Next Generation Arithmetic 2019*. Association for Computing Machinery, 2019. ISBN 9781450371391. doi: 10.1145/3316279.3316281. URL https://doi.org/10.1145/3316279.3316281.

[20] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen. Posits: the Good, the Bad and the Ugly. Dec. 2018. URL https://hal.inria.fr/hal-01959581.

[21] E. Ternovoy, M. G. Popov, D. V. Kaleev, Y. V. Savchenko, and A. L. Pereverzev. Comparative Analysis of Floating-Point Accuracy of IEEE 754 and Posit Standards. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 1883–186, 2020. doi: 10.1109/EIConRus49466.2020.9039521.

[22] 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. In *2019 Design, Automation Test in Europe Conference Exhibition*, pages 1421–1426, 2019. doi: 10.23919/DATE.2019.8715262.

[23] S. H. Fatemi Langroudi, Z. Carmichael, J. Gustafson, and D. Kudithipudi. PositNN Framework: Tapered Precision Deep Learning Inference for the Edge. pages 53–59, 07 2019. doi: 10.1109/SpaceComp.2019.00011.

[24] J. Johnson. Rethinking Floating Point for Deep Learning. *CoRR*, abs/1811.01721, 2018. URL http://arxiv.org/abs/1811.01721.

[25] T. Trevisan Jost, Y. Durand, C. Fabre, A. Cohen, and F. Pétrot. VP Float: First Class Treatment for Variable Precision Floating Point Arithmetic. pages 355–356, 09 2020. doi: 10.1145/3410463.3414660.

[26] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan. Dlfloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 92–95, 2019. doi: 10.1109/ARITH.2019.00023.

[27] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell. Bfloat16 Processing for Neural Networks. *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 2019. doi: 10.1109/arith.2019.00022.

[28] S. Wang and P. Kanwar. BFloat16: The secret to high performance on Cloud TPUs [Google Cloud Blog], 2019. URL https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus. Accessed on 03-2021.

[29] A. Y. Romanov, A. L. Stempkovsky, I. V. Lariushkin, G. E. Novoselov, R. V. Solovyev, V. A. Starykh, I. I. Romanova, D. V. Telpukhov, and I. A. Mkrtchan. Analysis of Posit and Bfloat Arithmetic of Real Numbers for Machine Learning. *IEEE Access*, pages 1–1, 2021. doi: 10.1109/ACCESS.2021.3086669.

[30] J. Turley. What is bfloat16, Anyway?, Mar 2020. URL https://www.eejournal.com/article/what-is-bfloat16-anyway/.

[31] IObundle. Iob-knn: KNN Application Implementation, . URL https://github.com/IObundle/iob-knn.

[32] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision, 2015.

[33] A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y.-H. Jim, H. Shen, and B. Ziv. Lower Numerical Precision Deep Learning Inference and Training. 2018.

[34] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis. Exploration of Low Numeric Precision Deep Learning Inference Using Intel® FPGAs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–80, 2018. doi: 10.1109/FCCM.2018.00020.

[35] Posit Working Group. Posit Standard Documentation Release 3.2-draft. URL https://posithub.org/docs/posit_standard.pdf.

[36] Unum-IV: Parameterized Floating Point Unit. GitHub. URL https://github.com/IObundle/iob-unum4.

[37] IObundle. RISC-V System on Chip Template Based on the picorv32 Processor, . URL https://github.com/IObundle/iob-soc.

[38] IObundle. Iob-fpu: Parameterized Floating Point Unit, . URL https://github.com/IObundle/iob-fpu.

[39] L. Forget, Y. Uguen, and F. de Dinechin. Comparing posit and IEEE-754 hardware cost. working paper or preprint, Apr. 2021. URL https://hal.archives-ouvertes.fr/hal-03195756.

[40] S. Imandoust and M. Bolandraftar. Application of K-Nearest Neighbor (KNN) Approach for Predicting Economic Events: Theoretical Background. *Int J Eng Res Appl*, 3:605–610, 01 2013.

[41] Okfalisa, I. Gazalba, Mustakim, and N. G. I. Reza. Comparative Analysis of K-Nearest Neighbor and Modified K-Nearest Neighbor Algorithm for Data Classification. In *2017 2nd International conferences on Information Technology, Information Systems and Electrical Engineering (ICITISEE)*, pages 294–298, 2017. doi: 10.1109/ICITISEE.2017.8285514.

[42] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, 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)*, pages 1105–1110, 2018. doi: 10.23919/DATE.2018.8342176.