UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Posit arithmetic and its applications

Aritmática posit y sus aplicaciones

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Raúl Murillo Montero

Directores

Alberto Antonio del Barrio García
Guillermo Botella Juan

Madrid

Universidad Complutense de Madrid

Facultad de Informática

Tesis doctoral

Posit Arithmetic and its Applications

Aritmética Posit y sus Aplicaciones

Memoria para optar al grado de Doctor

presentada por

Raúl Murillo Montero

Directores

Alberto Antonio Del Barrio García

Guillermo Botella Juan

Posit Arithmetic and its Applications

Aritmética Posit y sus Aplicaciones

Memoria que presenta para optar al título de Doctor en Ingeniería Informática

Raúl Murillo Montero

Dirigida por los Doctores:

Alberto Antonio Del Barrio García
Guillermo Botella Juan

Facultad de Informática
Universidad Complutense de Madrid

Febrero 2024

This thesis has been partially supported by the EU (FEDER) and Spanish MINECO (grant No. RTI2018-093684-B-I00), the Spanish CM (grant No. S2018/TCS-4423), the BBVA Foundation (grant No. PR2003_20/01), the Spanish MCIN/AEI/ 10.13039/501100011033 and the ERDF (grant No. PID2021-123041OB-I00), and the HiPEAC network (grant No. ICT-2019-871174).

A mi familia, directores
y todos los que me han
acompañado en este camino

Agradecimientos

Una vez me dijeron: "Los agradecimientos serán la única parte de tu tesis que todo el mundo lea". Así pues, espero no defraudar a nadie con estas líneas en las que pongo todo mi esmero.

Recuerdo perfectamente cuando Alberto y Guillermo me ofrecieron continuar trabajando en mi proyecto de fin de carrera. Sin embargo, poco o nada me imaginaba lo que aquella decisión supondría para mí. Lo que fue mi trabajo de fin de grado se convirtió en mi trabajo de fin de máster, y poco a poco ese proyecto fue creciendo hasta convertirse en la presente tesis, 5 años después. Pero echando la vista atrás, la tesis no ha sido sólo un largo proyecto que culmina con este documento, sino muchos momentos y, sobre todo, personas que, de una u otra manera, me han acompañado en este largo camino, y que por ello quiero dedicarles estas líneas de agradecimiento.

En primer lugar, tengo que mencionar a mis directores, Alberto y Guillermo, por la oportunidad que me ofrecieron y por su constante apoyo profesional y personal durante todos estos años. Dicen que el director de tesis es el *padre académico* del investigador. Yo he tenido dos, y no han podido ser mejores.

A toda la gente del grupo ArTeCS, por su acogida y siempre buen ambiente. A todo el personal de administración y servicios de ambas facultades que me ayudaron desde el momento en el que llegué al grupo. Y, por supuesto, a los compañeros y amigos del laboratorio, con quienes he compartido tantos cafés, comidas, más cafés, y toda clase de buenos momentos. A Luisma, que en este periodo ha sido como un hermano mayor al que acudir; a David, que me ha ayudado a materializar los proyectos con los que yo sólo alcanzaba a imaginar; a Dani, Nico, Tomasso, Sandra y Youssef, que han vivido esta aventura conmigo desde el principio.

A mi familia y amigos, por su paciencia y apoyo constante, pero en especial a mis padres y a Sara, que han estado ahí siempre, apoyándome de forma incondicional desde el momento en el que decidí embarcarme en este viaje.

Y, por último, a toda esa gente que, de una u otra manera, ha compartido conmigo una parte de esta travesía y que no he mencionado.

A todos vosotros, sólo puedo decir: gracias.

Madrid, 1 de diciembre de 2023

Resumen

Aritmética Posit y sus Aplicaciones

El fin del escalado de Dennard, así como la llegada de la era *post-Moore* han supuesto una desaceleración en la mejora del rendimiento de los sistemas computacionales modernos. Como resultado, los desafíos en la arquitectura de computadores son, a día de hoy, aún más difíciles, dado que hemos perdido los recursos que antes crecían de manera exponencial gracias al escalado de Dennard y la ley de Moore. Todo esto ha dado lugar a áreas críticas en esta nueva era. Entre ellas, destaca el co-diseño de hardware/software para lenguajes de alto nivel y de dominio específico, que permite aumentar tanto el rendimiento como la eficiencia energética.

Asociada a la especialización de las arquitecturas de hardware, han surgido en los últimos años nuevos formatos de representación para incrementar el rendimiento de las aplicaciones modernas. Específicamente, el formato IEEE 754 de representación de números en coma flotante es actualmente utilizado en la mayoría de los computadores modernos para representar y operar con números reales. No obstante, en esta nueva era, la supremacía de este formato se ha cuestionado debido a que no resulta óptimo para todo tipo de aplicaciones.

En este contexto, en el año 2017 se presentó una propuesta innovadora: un nuevo formato aritmético denominado *posit* para la representación de números reales en computadoras. Este formato emergente fue concebido con el propósito de reemplazar el estándar IEEE 754 de números de coma flotante, ofreciendo ventajas tales como un rango dinámico más amplio, mayor precisión, resultados idénticos bit a bit en todos los sistemas y un diseño hardware más sencillo, entre otras. Las propiedades que presenta el formato posit parecen ser sumamente útiles en algunas aplicaciones modernas. Sin embargo, para obtener un rango dinámico mayor, el formato posit sacrifica precisión al representar valores de gran magnitud, lo cual podría resultar perjudicial en ciertos casos. Además, la adopción de un nuevo formato aritmético implica el desarrollo de la tecnología necesaria, lo que lleva un coste significativo. Este escenario añade una capa adicional de complejidad al discernir si el formato posit ofrece un rendimiento superior al de coma flotante.

El objetivo de esta tesis es la exploración de las propiedades y características del formato posit para evaluar su viabilidad como alternativa al formato de coma flotante IEEE 754, que prevalece en las aplicaciones modernas. Además, también se propone el diseño de unidades aritméticas para examinar el impacto hardware de este formato.

Para abordar estos objetivos, en esta tesis se emplean diversas técnicas de simulación de este nuevo formato, ejecutando aplicaciones de alto nivel con diferentes configuraciones de aritmética posit. De esta manera, se puede verificar cómo un cambio sencillo en la aritmética subyacente utilizada en los cálculos puede, en muchos casos, producir resultados más precisos o con un menor margen de error en comparación con el formato estándar implementado en los computadores. Por otro lado, con el propósito de evaluar el impacto que tendría la implementación de este nuevo formato en sistemas reales, en esta tesis se aborda el diseño y la evaluación de diversos componentes hardware dedicados a realizar operaciones aritméticas básicas, como la suma, la multiplicación o la división en formato posit. Estos componentes, diseñados con un enfoque especial en la eficiencia energética, presentan un coste adicional en comparación con las unidades análogas de coma flotante, aunque mejoran las implementaciones previas presentadas en la literatura sobre aritmética posit. Adicionalmente, aprovechando los diseños realizados para las unidades posit, estos se integran en una herramienta para la generación automática de aceleradores hardware destinados a aplicaciones específicas basadas en aritmética posit, permitiendo su ejecución en hardware real. Finalmente, con el objetivo de encontrar un mejor equilibrio entre la precisión demostrada por el formato posit y un menor coste de implementación, en esta tesis también se exploran nuevos formatos aritméticos derivados de la aritmética posit.

Los resultados obtenidos en esta tesis muestran que el formato posit ofrece ventajas significativas en aplicaciones modernas como redes neuronales y computación de alto rendimiento, proporcionando resultados con mayor precisión numérica que el formato de coma flotante IEEE 754. Sin embargo, también se observa que este nuevo formato, con las implementaciones actuales, presenta un mayor consumo de energía y de recursos y un mayor tiempo de ejecución que el formato estándar de coma flotante. Por último, se demuestra que los formatos derivados de la aritmética posit pueden ofrecer un mejor equilibrio entre precisión y coste de implementación.

En esencia, aunque adoptar la aritmética posit de forma generalizada puede suponer enfrentarse a retos como el uso de más energía y recursos de hardware, su potencial para tareas de precisión es evidente. Al utilizar formatos basados en posit, los investigadores y profesionales pueden manejar mejor las complejidades del cálculo numérico. Esto podría cambiar nuestra forma de hacer cálculos, haciéndolos más precisos, eficientes y prácticos en el mundo digital actual.

Abstract

Posit Arithmetic and its Applications

The end of Dennard scaling, as well as the arrival of the *post-Moore* era, has resulted in a slowdown in the improvement of modern computer systems' performance. As a result, the challenges in computer architecture are even more difficult nowadays, given that we have lost the resources that previously grew exponentially thanks to Dennard scaling and Moore's law. All of this has led to critical areas in this new era. Among them, hardware/software co-design for high-level and domain-specific languages stands out, which allows for both performance and energy efficiency to be increased.

Associated with the specialization of hardware architectures, new representation formats have emerged in recent years to increase the performance of modern applications. Specifically, the IEEE 754 format for representing floating-point numbers is currently used in most modern computers to represent and operate with real numbers. However, in this new era, the supremacy of this format has been questioned because it is not optimal for all types of applications.

In this context, an innovative proposal was presented in 2017: a new arithmetic format called *posit* for the representation of real numbers in computers. This emerging format was conceived with the purpose of replacing the standard IEEE 754 of floating-point numbers, offering advantages such as a wider dynamic range, greater precision, bit-identical results on all systems, and simpler hardware design, among others. The properties presented by the posit format seem to be extremely useful in some modern applications. However, to obtain a wider dynamic range, the posit format sacrifices precision when representing values of great magnitude, which could be detrimental in certain cases. Additionally, the adoption of a new arithmetic format implies the development of the necessary technology, which carries a significant cost. This scenario adds an additional layer of complexity when discerning whether the posit format offers superior performance to that of floating-point.

The objective of this thesis is to explore the properties and characteristics of the posit format to evaluate its viability as an alternative to the floating-point IEEE 754 format, which prevails in modern applications. Additionally, the design of arithmetic units is proposed to examine the hardware impact of this format.

To address these objectives, various simulation techniques of this new format are employed in this thesis, executing high-level applications with different posit arithmetic

configurations. In this way, it can be verified how a simple change in the underlying arithmetic used in calculations can, in many cases, produce more accurate results or with a lower margin of error compared to the standard format implemented in computers. On the other hand, with the purpose of evaluating the impact that the implementation of this new format would have on real systems, this thesis addresses the design and evaluation of various hardware components dedicated to performing basic arithmetic operations, such as addition, multiplication, or division in posit format. These components, designed with a special focus on energy efficiency, present an additional cost compared to analogous floating-point units, although they improve on previous implementations presented in the literature on posit arithmetic. Additionally, taking advantage of the designs made for the posit units, they are integrated into a tool for the automatic generation of hardware accelerators intended for specific applications based on posit arithmetic, allowing their execution in real hardware. Finally, with the aim of finding a better balance between the precision demonstrated by the posit format and a lower implementation cost, this thesis also explores new arithmetic formats derived from posit arithmetic.

The results obtained in this thesis show that the posit format offers significant advantages in modern applications such as neural networks and high-performance computing, providing results with higher numerical precision than the IEEE 754 floating-point format. However, it is also observed that this new format, with current implementations, presents higher energy and resource consumption and longer execution time than the standard floating-point format. Finally, it is demonstrated that formats derived from posit arithmetic can offer a better balance between precision and implementation cost.

In essence, although adopting posit arithmetic widely might mean dealing with challenges like using more energy and hardware resources, its potential for precision tasks is clear. By using posit-based formats, researchers and practitioners can handle the complexities of numerical computing better. This could change the way we do computations, making them more accurate, efficient, and practical in today's digital world.

Contents

Resumen III

Abstract V

Acronyms XV

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Background and definitions . . . 3
  - 1.2.1. Floating-point arithmetic . . . 4
  - 1.2.2. Posit arithmetic . . . 7
  - 1.2.3. Properties of posit arithmetic . . . 10
- 1.3. Objectives . . . 13
- 1.4. Document structure . . . 14

2. State of the art 17
- 2.1. Posits in software . . . 17
  - 2.1.1. Software libraries . . . 17
  - 2.1.2. Scientific applications . . . 18
  - 2.1.3. Deep learning . . . 19
- 2.2. Posit implementation solutions . . . 20
  - 2.2.1. Basic arithmetic units . . . 20
  - 2.2.2. Domain-specific accelerators . . . 21
  - 2.2.3. Posit-based RISC-V implementations . . . 22
- 2.3. Other posit number representations . . . 23

I Posit Arithmetic in Software Applications 25

3. Use of posit arithmetic in scientific computing 27
- 3.1. A motivational example: Goldberg's thin triangle problem . . . 27
- 3.2. Decimal accuracy . . . 28

3.3. Error-sensitive applications . . . 30
3.3.1. Benchmark description . . . 30
3.3.2. Results . . . 31
3.4. Conclusions . . . 32

4. Use of posit arithmetic in deep learning 35
4.1. Background on deep learning . . . 35
4.1.1. Neural networks and deep learning . . . 36
4.1.2. Model training and execution . . . 37
4.1.3. Common deep learning architectures . . . 39
4.1.4. Benefits of posit arithmetic in deep learning . . . 40
4.2. Emulating posit arithmetic in DNNs . . . 41
4.2.1. DNN training with posits . . . 42
4.2.2. DNN inference with low-precision posits . . . 42
4.2.3. Fused arithmetic for low-precision DNN inference . . . 44
4.3. Experimental evaluation . . . 46
4.3.1. Experimental setup . . . 46
4.3.2. Training evaluation . . . 47
4.3.3. Quantization evaluation . . . 51
4.4. Conclusions . . . 52

II Hardware Design for Posit Arithmetic 55

5. Functional units design 57
5.1. Posit decoding . . . 58
5.2. Posit encoding and rounding . . . 59
5.3. Posit addition and subtraction . . . 60
5.4. Posit multiplication . . . 62
5.5. Posit division . . . 62
5.5.1. Non-restoring division . . . 64
5.5.2. Newton-Raphson division . . . 65
5.5.3. Goldschmidt division . . . 66
5.5.4. Implementation of posit dividers . . . 66
5.6. Posit square root . . . 67
5.7. Evaluation . . . 68
5.7.1. Comparison with state-of-the-art posit implementations . . . 68
5.7.2. Comparison with floating-point operators . . . 72
5.8. Conclusions . . . 73

6. Fused posit arithmetic unit 75
6.1. The quire accumulator . . . 75
6.2. Posit MAC unit design . . . 77
6.2.1. Exact accumulation of products . . . 78

6.2.2. Quire to posit conversion . . . 78
6.2.3. Posit to quire conversion . . . 78
6.3. Evaluation . . . 79
6.3.1. Accuracy . . . 80
6.3.2. Hardware comparison with non-fused operations . . . 80
6.3.3. Implementation alternatives . . . 82
6.3.4. Comparison with state-of-the-art implementations . . . 83
6.4. Conclusions . . . 85

7. Generation of posit-based accelerators **87**
7.1. Domain-specific hardware accelerators . . . 88
7.1.1. High-level synthesis . . . 88
7.2. Posit-aware high-level synthesis . . . 90
7.2.1. Library of posit operators . . . 91
7.2.2. Memory customization . . . 92
7.2.3. Integration of posits into HLS tool . . . 92
7.2.4. Limitations . . . 95
7.3. Hardware evaluation . . . 96
7.3.1. Experimental setup . . . 96
7.3.2. Accuracy evaluation . . . 98
7.3.3. Operation-level evaluation . . . 98
7.3.4. Application-level evaluation . . . 102
7.4. Conclusions . . . 107

III Alternative Arithmetic Formats Based on Posit **109**

8. Logarithmic-approximate posit arithmetic **111**
8.1. Approximating posits in the logarithmic domain . . . 112
8.2. Posit logarithmic approximate multiplication . . . 113
8.3. Posit logarithmic approximate division . . . 114
8.4. Posit logarithmic approximate square root . . . 116
8.5. Approximation error . . . 116
8.6. Application evaluation . . . 119
8.6.1. Digital image processing . . . 119
8.6.2. Machine learning . . . 120
8.6.3. Deep learning . . . 121
8.7. Hardware evaluation . . . 123
8.8. Conclusions . . . 125

9. Half-unit-biased posit arithmetic **129**
9.1. Half-unit biased formats . . . 130
9.1.1. The posit HUB format . . . 130
9.2. Posit HUB architectures . . . 131

9.2.1. Posit HUB adder . . . 133
9.2.2. Posit HUB multiplier . . . 133
9.2.3. Comments about conversion and rounding . . . 133
9.3. Error analysis . . . 135
9.3.1. Experimental setup . . . 135
9.3.2. Results and discussion . . . 135
9.4. Hardware evaluation . . . 137
9.4.1. Synthesis results and comparison . . . 137
9.5. Conclusions . . . 139

10. Conclusions **143**
10.1. Conclusions and main contributions . . . 143
10.2. Related publications . . . 145
10.2.1. Directly related publications . . . 145
10.2.2. Indirectly related publications . . . 146
10.3. Open-source repositories . . . 146
10.4. Open research lines . . . 147

A. Interpretations of posit arithmetic **149**
A.1. Sign-magnitude interpretation . . . 149
A.2. Two's complement interpretation . . . 150
A.3. Equivalence between interpretations . . . 150
A.4. Hardware evaluation . . . 151

B. Third-party software **155**
B.1. Posit emulation software . . . 155
B.2. Hardware design tools . . . 158

References **176**

List of Figures

1.1. Encoding of the binary IEEE 754 floating-point formats 5
1.2. Comparison of alternative floating-point formats 6
1.3. Previous unum types 8
1.4. General binary Posit$\langle n, es \rangle$ representation with all fields explicit 8
1.5. Visual representation in the real projective line of different posit formats . . 12
1.6. Significant binary digits precision of different number formats 13

3.1. Goldberg's thin triangle problem 28
3.2. MSE per iteration for different ODEs 32

4.1. Schematic representation of the mathematical operation of an artificial neuron 37
4.2. General architecture of a neural network 38
4.3. Histogram of the parameter distribution for different DNN models 41
4.4. Scheme of the training flow for the proposed framework 43
4.5. DNN forward data flow for low-precision GEMM operation with fused posit arithmetic 45
4.6. LeNet-5 training on MNIST with different number formats 48
4.7. LeNet-5 training on Fashion-MNIST with different number formats 49
4.8. CifarNet training on SVHN with different number formats 49
4.9. CifarNet training on CIFAR-10 with different number formats 50

5.1. Generic computation flow for posits 58
5.2. Block diagram of the proposed posit decoder 59
5.3. Block diagram of the proposed posit encoder 60
5.4. Block diagram of the proposed posit adder 61
5.5. Block diagram of the proposed posit multiplier 63
5.6. Basic scheme for posit division 64
5.7. Implementation scheme for iterative dividers 67
5.8. Basic scheme for posit square root 68
5.9. Synthesis results for different Posit$n$ adder designs 69
5.10. Synthesis results for different Posit$n$ multiplier designs 70

5.11. Synthesis results for different Posit$n$ divider designs 71

6.1. Quire format encoding for Posit$\langle n, es \rangle$ 76
6.2. Basic scheme for posit quire MAC operation 77
6.3. Basic scheme for quire to posit conversion 79
6.4. Hardware comparison between standard and fused posit units 81
6.5. Synthesis results for different designs of Posit16 MAC units with 256-bit quire 83
6.6. Synthesis results for different designs of Posit32 MAC units with 512-bit quire 84

7.1. HLS workflow with support for posit arithmetic and memory customization 90
7.2. Scheme of the different approaches that use float and/or posit arithmetic in memory and hardware accelerator 91
7.3. HLS design flow and modifications to accommodate posit units 94
7.4. PolyBench benchmarks error comparison for different number formats . . . 99
7.5. Bambu HLS results for basic arithmetic operators 101
7.6. HLS results for posit-float converters 102
7.7. Vitis HLS results for basic arithmetic MArTo operators 103
7.8. Bambu HLS results for PolyBench benchmarks 104
7.9. Vitis HLS results for PolyBench benchmarks 106

8.1. Resource distribution of a Posit32 multiplier 113
8.2. Hardware implementation scheme for the proposed logarithm approximate posit multiplication 115
8.3. Hardware implementation scheme for the proposed logarithm approximate posit square root 116
8.4. Relative error plots between logarithmic approximate and exact posit operations 118
8.5. Resource usage of the approximate operators compared to the exact ones . . 126

9.1. Architecture of a posit HUB decoder and decoder modules 132
9.2. Histograms of the rounding error for arithmetic operations 136
9.3. Synthesis results for conventional and HUB Posit$n$ adders 138
9.4. Comparison of conventional and HUB Posit32 adder implementations . . . 139
9.5. Synthesis results for conventional and HUB Posit$n$ multipliers 140
9.6. Comparison of conventional and HUB Posit32 multiplier implementations . 141

A.1. Hardware synthesis for Posit32 adders 152
A.2. Hardware synthesis for Posit32 multipliers 153

B.1. Simplified overview of the FloPoCo class hierarchy 158
B.2. VHDL generation flow with FloPoCo 159
B.3. Bambu compilation flow 160

List of Tables

1.1. Posit8 format decodification 10
1.2. Span and precision of IEEE 754 floating-point and posit formats 14

2.1. Summary of posit arithmetic hardware implementations 22

3.1. Result and accuracy for different benchmarks 29
3.2. Decimal accuracy for representation of physical constants 30
3.3. Global MSE for different benchmarks 31

4.1. Supported features in Deep PeNSieve 44
4.2. Accuracy results for DNNs trained with different data formats 48
4.3. Training time per epoch for each model and numeric format 50
4.4. Accuracy results with post-training quantization 51

5.1. Synthesis results for Posit$n$ square root units 72
5.2. Synthesis results for float and posit arithmetic operators 72

6.1. Quire parameters for different posit formats 76
6.2. GEMM MSE comparison between IEEE 754 floating-point and posit numbers 80
6.3. Comparison with state-of-the-art Posit$n$ MAC units 85

8.1. Image processing accuracy results 120
8.2. K-NN accuracy results 121
8.3. Top-1 accuracy results (%) for the inference stage 122
8.4. ImageNet validation set accuracy (%) 123
8.5. Synthesis results for exact and approximate posit operators 124

9.1. Posit6 format decodification 131
9.2. Statistical parameters of rounding error distribution 137

Acronyms

AI artificial intelligence . . . 2
ASIC application-specific integrated circuit . . . 64
ALU arithmetic logic unit . . . 57
ADP area-delay product . . . 123
BKA Brent-Kung adder . . . 82
CNN convolutional neural network . . . 19
CPU central processing unit . . . 3
DNN deep neural network . . . 15
DSA domain-specific architecture . . . 3
DSP digital signal processor . . . 99
FF flip-flop . . . 99
FMA fused multiply-add . . . 11
FU functional unit . . . 9
FP floating-point arithmetic . . . 2
FPU floating-point unit . . . 2
FPGA field-programmable gate array . . . 3
GEMM general matrix-matrix multiplication . . . 41
GPU graphics processing unit . . . 3
HDL hardware description language . . . 68
HLS high-level synthesis . . . 15
HPC high-performance computing . . . 2
HUB Half-Unit-Biased . . . 129
IoT internet of things . . . 3
ISA instruction set architecture . . . 22
iLSB implicit least significant bit . . . 130

IEEE Institute of Electrical and Electronics Engineers . . . 2
KSA Kogge-Stone adder . . . 82
LSB least significant bit . . . 8
LUT lookup table . . . 18
MAC multiply-accumulate . . . 19
MSB most significant bit . . . 58
MSE mean squared error . . . 31
NaN Not a Number . . . 2
NaR Not a Real . . . 7
ODE ordinary differential equation . . . 30
PSNR peak signal-to-noise ratio . . . 119
PTQ post-training quantization . . . 43
PLAU posit logarithm-approximate unit . . . 113
QAT quantization aware training . . . 43
RCA ripple-carry adder . . . 82
RTL register-transfer level . . . 88
RISC reduced instruction set computer . . . 22
SIMD single instruction, multiple data . . . 37
SIMT single instruction, multiple threads . . . 37
SSIM structural similarity index . . . 119
TPU tensor processing unit . . . 3
unum universal number . . . 7
ULP unit in the last place . . . 27

1

Introduction

1.1. Motivation

Since the dawn of time, human beings have asked some fundamental questions: *how to count quantities? how to manipulate them?* This practical need for counting, elementary measurements and calculations became the reason for the emergence of arithmetic. This is the fundamental branch of mathematics that studies numbers and their operations. The term "arithmetic" has its root in the Latin term "*arithmetica*" which derives from the Ancient Greek words ἀριθμός (*arithmós*), meaning "number" or "counting", and τέχνη (*tékhnē*), meaning "art".

Throughout history, numerous numeral systems have been proposed for the expression and manipulation of numbers, each with its own set of advantages and disadvantages. One such example is the Roman numeral system, a non-positional system that originated in Ancient Rome and was employed throughout the Roman Empire for more than two millennia. While this system offers a concise representation in certain contexts by using combinations of letters to represent numbers, it presents complexities when used for calculations, often requiring the use of tools such as an abacus. Furthermore, Roman numerals do not have a symbol for zero. As a result, the Hindu-Arabic numeral system supplanted Roman numerals in the 15^{th} century, relegating the latter to purely aesthetic applications.

The Hindu-Arabic numeral system is a positional base-10 numeral system that marked a significant leap forward in human interpretative and computational capacities. Actually, the decimal numeral system, which is nowadays the standard system for denoting integer and non-integer numbers, is an extension of the former. However, since these numerals also adhere to a base-10 system, challenges arise when attempting to integrate them into computer systems. Devising an effective numeral system, as demonstrated, presents a nuanced challenge.

Most modern computer hardware and software systems commonly use a binary (or base-2) representation internally, i.e., they utilize only two symbols, typically "`0`" (zero) and "`1`" (one). This framework gave rise to computer arithmetic, which encompasses the

study of number representations and algorithms for operations on numbers in computer systems [38], and is the main area in which this Ph.D. thesis is framed.

Representing integer numbers in computers can follow a similar approach to the decimal system, but dealing with real numbers introduces additional complexities. Unlike integers, which form a discrete set allowing for straightforward mapping to devices, real numbers belong to a continuous set, making it impossible to create a one-to-one mapping to represent a subset of them. Fixed-point arithmetic is a method used to represent fractional numbers by storing a fixed number of digits of their fractional part. However, this format is constrained by limitations in precision and/or the range of values it can represent. In 1914, the Spanish engineer Leonardo Torres Quevedo published "Essays on Automatics" [160], where he introduced the idea of floating-point arithmetic. Initially, various representations were used for floating-point numbers in computers. Subsequently, in 1938, Konrad Zuse of Berlin completed the Z1, the first binary, programmable mechanical computer using a binary floating-point number representation [145]. The popular and mass-produced IBM 704 followed in 1954, introducing the use of a biased exponent. For several decades thereafter, floating-point hardware remained an optional feature. It was not until 1985 that the Institute of Electrical and Electronics Engineers (IEEE) established what we know nowadays as the IEEE 754 Standard for Floating-Point Arithmetic. This first standard is followed by almost all modern machines, widely used in computer hardware and languages. It defines several sizes and specifications of floating-point numbers. While this standard has become prevalent, it has been found to have several deficiencies since its adoption, including issues such as subtractive cancellation, redundancy of Not a Numbers (NaNs), signed zeros, or inconsistency of results across machines due to the different rounding methods [45]. To address these shortcomings, a new data type called posit was proposed in 2017 as a direct replacement for IEEE 754 floating-point numbers [52].

In its early days, floating-point arithmetic (FP) was primarily used for scientific applications. However, as time has passed, an increasing number of diverse applications have come to rely on this type of computation. Nowadays, the vast majority of modern computers are equipped with an floating-point unit (FPU). Consequently, software has advanced significantly, enabling the handling of increasingly complex tasks. One area benefitting from this evolution is machine learning, and in particular deep learning. These artificial intelligence (AI) techniques and algorithms have incredibly transformed various fields such as computer vision, speech and language processing, as well as making significant transformations in medicine, the automotive industry, and finance over the past decade. This, coupled with its potential to address future societal challenges, has emphasized the growing significance of deep learning. The impressive advancements that this technology has achieved in recent years can be largely attributed to the significant expansion of available datasets and the computational power required for their analysis [5]. However, the increasing complexity of deep learning models is starting to confront the limits of Moore's law and Dennard scaling [56]. The projection made by Gordon Moore in the mid-1960s began to lose accuracy around the turn of the century, and in recent years, the disparity between the actual number of transistors on a chip and Moore's prediction has widened, thereby leading to the so-called *post-Moore* era. Similarly, Dennard scaling experienced a significant slowdown starting in 2007, diminishing to nearly zero by 2012. Another area that has experienced a similar evolution since the advent of the Exascale era is high-performance computing (HPC), which uses supercomputers and computer clusters to solve complex computation problems

at extremely high speeds. HPC applications, like autonomous driving, weather forecasting, or computational fluid dynamics, among others, take advantage of hardware and software architectures that spread computation across resources. Indeed, over the past decade, deep learning has emerged as a primary application within modern HPC contexts.

Nowadays, maintaining performance improvement is crucial for enabling new software capabilities, such as deep learning or HPC. A compelling research focus in computer architecture involves the use of domain-specific architectures (DSAs), which are processors tailored for a specific domain or class of applications. Examples of DSAs include graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and Google's tensor processing units (TPUs) [74], each closely customized to the requirements of particular applications. As a result, DSAs, also known as accelerators, can deliver superior performance and energy efficiency compared to general-purpose central processing units (CPUs). In the context of machine learning applications, recent studies have revealed that the standard floating-point format is notably inefficient for these purposes [157, 10]. Consequently, companies like Google, ARM, or Microsoft have developed their own accelerators based on alternative formats to IEEE 754 floating-point for their AI services. More efficient formats are necessary for these types of applications to operate on mobile, internet of things (IoT), or embedded devices [34]. Also, recent advances in hardware accelerator design within the HPC field have provided efficient solutions to meet the performance requirements of various applications. This includes not only GPUs and TPUs-based accelerators, but also DSAs such as FPGAs-based and ASIC-based accelerators [151]. The favorable characteristics of posit and its recent advancements render this format a suitable alternative to the IEEE 754 standard for these domains. A simple shift to a new number system could significantly enhance the scale and cost-effectiveness of these applications, thereby contributing to the Exascale revolution.

In summary, despite the long-standing use of the IEEE 754 Standard for Floating-Point Arithmetic in modern computers, the need for more efficient formats within the realm of HPC is becoming more and more apparent. Investigating the potential of alternative formats, such as posit arithmetic, in this domain is a highly compelling challenge that will be addressed throughout the remainder of this document.

1.2. Background and definitions

Computer arithmetic is a field of computer science that investigates how computers should represent numbers and perform operations on them. All computer hardware, and practically all software, performs arithmetic by representing every number as a fixed-length sequence of `1` s and `0` s, or bits. However, there are a few differences when dealing with integers or real values in computers. Integers are often represented as a single sequence of bits, each representing a different power of two, with a single bit indicating the sign. Under this representation, integer arithmetic operates according to the "normal" (or symbolic) rules of arithmetic. On the other hand, there are multiple arithmetic formats for encoding real numbers in computers, and this is the case on which this thesis will focus.

In scientific computing, most operations happen between real numbers. There are two classical approaches for storing real numbers (i.e., numbers with fractional components) in modern computing. These are fixed-point arithmetic and floating-point arithmetic. While

the former of these formats can run on devices without the need for specific hardware for decimal arithmetic and provides high computational speed, the latter has greater precision and is able to represent a much wider range of numbers. Indeed, floating-point arithmetic is ubiquitous in modern computing systems, and the preferred way computers approximate real numbers.

1.2.1. Floating-point arithmetic

Floating-point numbers (often called *floats*), as in scientific notations, are represented using an exponent (normally in base two) and a significand, except that this significand has to fit on a certain amount of bits.

The representation of a float is similar in concept to scientific notation and consists of

- A signed number, referred to as the *significand*, *mantissa*, *coefficient*, or ambiguously *fraction*. This number is encoded as a digit string of a given length.
- A signed integer exponent (normally in base two), which modifies the magnitude of the number.

To obtain the value of the floating-point number, the significand or mantissa is multiplied by the base raised to the power of the exponent. The way in which the significand (including its sign) and exponent are stored in a computer is implementation-dependent, but it is quite common that the first bit represents the sign, the next bits the exponent, and finally the significand.

The term *floating point* refers to the fact that the radix point of a number can "float"; that is, it can be placed anywhere relative to the significant digits of the number, similarly as in common scientific notation. A floating-point system can be used to represent, with a fixed number of digits, numbers with different orders of magnitude, so it can be often found in systems that include very small and very large real numbers. However, since it uses only a finite number of bits, not all numbers can be represented under this format. Throughout the years, a variety of floating-point representations have been used in computers, but in 1985, the IEEE 754 Standard for Floating-Point Arithmetic was established. Since then, the most commonly encountered representations are those defined by the IEEE.

The IEEE 754 Standard for Floating-Point Arithmetic

Some decades ago, issues like the length of the mantissa and the rounding behavior of operations for floating-point numbers could differ between computer manufacturers, and even between models from the same manufacturer. Obviously, this led to problems with the portability and reproducibility of results. The IEEE Standard for Floating-Point Arithmetic (namely, IEEE 754) solved these shortcomings by providing definitions for arithmetic formats, rounding schemes, operations, representation of special numbers, and exception handling [64].

The original IEEE 754-1985, which is implemented in the vast majority of modern computers, stipulates two fundamental formats, namely single (sometimes called Float32, FP32, or binary32, as in the latest revision of the standard) and double (alternatively known as Float64, FP64, or binary64). Note that this nomenclature is frequently used in basic data

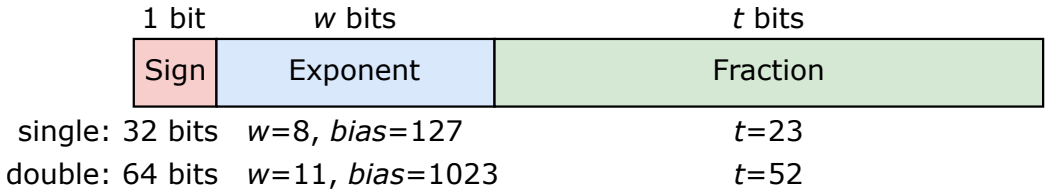


Figure 1.1: Encoding of the binary IEEE 754 floating-point formats.

types from high-level, general- purpose programming languages like C++, Java or Python. Representations of numbers in these formats are encoded in n bits in the following three fields, as shown in Figure 1.1:

- 1-bit sign S,
- w-bit biased exponent $E = e + bias$,
- t-bit trailing significand field digit string $F = f_1 f_2 \dots f_t$; the leading bit of the significand, f_0, is implicitly encoded in the biased exponent E.

The value v of a floating-point number is inferred from the aforementioned fields as follows:

a) If $E = 2^w - 1$ and $F \neq 0$, then v is NaN, and f_1 shall distinguish between qNaN (quiet) and sNaN (signaling).

b) If $E = 2^w - 1$ and $F = 0$, then $v = (-1)^S \times (+\infty)$.

c) When $1 \leqslant E \leqslant 2^w - 2$ is the most common case, and the value of the corresponding floating-point number is given by Equation (1.1); thus normal numbers have an implicit leading significand bit of 1.

$$v = (-1)^S \times 2^{E-bias} \times (1 + 2^{-t} \times F). \tag{1.1}$$

d) If $E = 0$ and $F \neq 0$, the implicit leading significand bit of the number is 0, as shown in Equation (1.2). Such numbers are called *subnormal* or *denormalized*, since they are non-zero numbers with a smaller magnitude than the smallest normal number.

$$v = (-1)^S \times 2^{emin} \times (0 + 2^{-t} \times F). \tag{1.2}$$

e) If $E = 0$ and $F = 0$, then $v = (-1)^S \times (+0)$ (that is, signed zero).

Note that this format represents signed infinity and zeros, and multiple NaN exceptions. Denormalized numbers are non-zero numbers with a magnitude smaller than the smallest normal number, and provide a *gradual underflow*, filling this gap around zero in floating-point arithmetic.

The IEEE 754 original standard from 1985 has two revisions, one in 2008 [65], which extended the previous version with decimal floating-point arithmetic (radix 10), and another in 2019 [66], which is a minor revision of IEEE 754-2008.

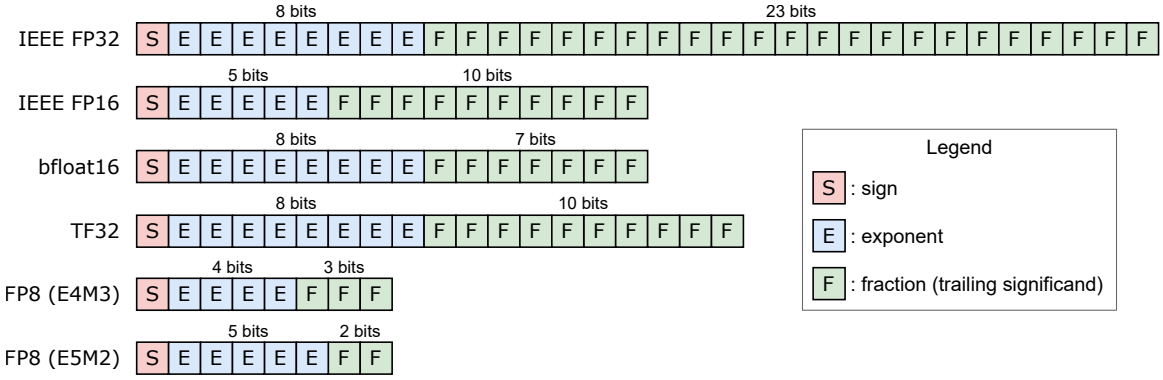


Figure 1.2: Comparison of alternative floating-point formats.

Other floating-point formats

In addition to the widely used floating-point formats defined in the IEEE 754-1985 standard, there exist alternative formats for performing floating-point computations. The revised version of the standard, IEEE 754-2008, introduced definitions for storage formats of various lengths, including 16-bit and 128-bit formats. However, recent developments in computer arithmetic have led to the emergence of alternative arithmetic formats that are not compliant with IEEE 754-2008. Notably, leading companies have also contributed to the development of these formats, making them of particular interest to the research community. The most relevant of such formats are summarized in Figure 1.2.

- **IEEE half-precision 16-bit float.** Also called Float16, FP16, or binary16, is a binary format defined by the IEEE 754-2008 standard [65] and is commonly used in applications where storage space and computational efficiency are important considerations. The format uses 16 bits to represent a floating-point number, with 1 bit for the sign, 5 bits for the exponent, and 10 bits for the mantissa. Compared to other floating-point formats, IEEE half-precision is less precise but requires less storage space and computation time. It is used in a wide range of applications, including graphics processing, mobile devices, and machine learning [107].

- **Brain floating point.** The brain floating point format, or bfloat16 for short, developed by Google, uses 16 bits to represent a floating-point number. In this format, 1 bit is used for the sign, 8 bits are used for the exponent, and 7 bits are used for the significand. It is designed to strike a balance between the precision of a traditional 32-bit floating-point format (both have the same exponent bit length) and the memory efficiency of a 16-bit format, making it useful in a wide range of applications, including machine learning and HPC [76, 57]. Although non-standard, this format is supported by many modern processors, including those from Intel, Arm, and Google [70, 4, 46], as well as GPUs [134].

- **8-bit floating point.** Also known as Float8 or FP8, is a natural progression from 16-bit floating-point data types, reducing the bit length down to just 8 bits [109]. In contrast with other formats, this one consists of two encodings:

  - E4M3, with 4 exponent bits, 3 mantissa bits, and 1 sign bit.
  - E5M2, with 5 exponent bits, 2 mantissa bits, and 1 sign bit.

  Although initially designed for educational purposes, the FP8 data format has shown promise in accelerating both AI training and inference. As a matter of fact, its implementation has resulted in halving data storage requirements and doubling throughput when compared to FP16 or bfloat16 [155]. As a result, this novel data type has already been implemented in some of the most advanced computer accelerators [136], further validating its potential as a valuable tool for improving AI performance.

- **TensorFloat-32.** This format, sometimes called TF32, is a hybrid between the single- and half-precision standard formats. It uses the same 10-bit mantissa as the FP16, having enough precision for AI workloads, but adopts the same 8-bit exponent as FP32, so it can support the same numeric range. This format, proposed by NVIDIA, shows remarkable speedups on AI training when compared to FP32 [135]. It is implemented on the NVIDIA Ampere generation of GPUs. However, due to its irregular size, all memory accesses remain FP32.

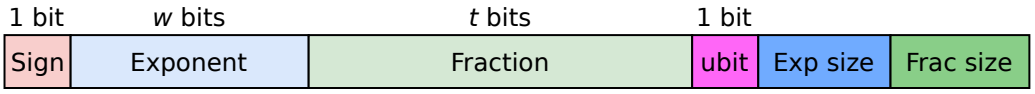
1.2.2. Posit arithmetic

As an alternative to the IEEE 754 arithmetic standard, John L. Gustafson proposed the concept of universal number (unum) [49], which has several forms.

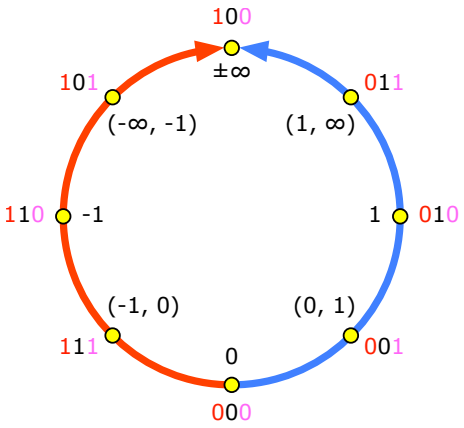
The original "Type I" unum is a superset of the IEEE 754 Standard floating-point format. It uses a *ubit* at the end of the fraction to indicate whether the number corresponds to an exact value or lies in an open interval. Type I unum takes the definition of the sign, exponent and fraction bit fields from IEEE 754, but the last two have a variable width. The bit lengths of such fields are indicated after the ubit, as depicted in Figure 1.3a. Although this format provides a more straightforward way to express interval arithmetic, its variable length demands extra management.

The "Type II" unum [50] was designed to resolve some of the shortcomings that Type I had, presenting a mathematical design based on the projective reals, but abandoning compatibility with the IEEE 754. The 3-bit format is illustrated in Figure 1.3b. This version has ideal mathematical properties, such as symmetrical distribution for negative and inverse values, but relies on look-up tables for most operations. This limits the scalability to about 20 bits or fewer. Furthermore, fused operations such as dot product, which is used in a vast amount of applications, are quite expensive in this format. These drawbacks served as a motivation for searching a new format that would keep many of the Type II unum properties, but also be "hardware-friendly", which means, easily implementable on hardware.

The "Type III" unum, better known as posit, was introduced by John L. Gustafson in 2017 "as a direct drop-in replacement for IEEE 754 standard for floating-point numbers" [52]. Posit numbers take the concept from the previous unum type while relaxing the perfect symmetry condition in order to perform hardware computations with similar logic to the existing floating-point format. The posit number system is a floating-point encoding scheme with tapered precision. Initially, a posit format Posit$\langle n, es\rangle$ was defined by its total bit length (n) and maximum width of the exponent field (es) [52], being Posit$\langle 8, 0\rangle$, Posit$\langle 16, 1\rangle$ and Posit$\langle 32, 2\rangle$ the most widespread posit formats in literature [32, 126, 127]. Posit numbers only distinguish two special cases: zero and Not a Real (NaR), which are represented as



(a) Type I unum.



(b) Type II unum.

Figure 1.3: Previous unum types.

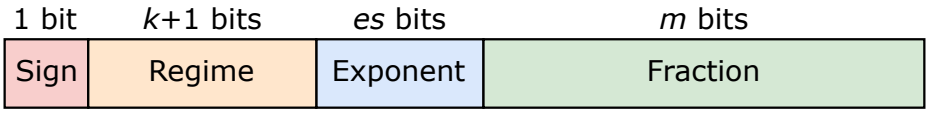


Figure 1.4: General binary Posit$\langle n, es\rangle$ representation with all fields explicit.

`00...0` and `10...0` respectively. The rest of the representations are composed of four fields as shown in Figure 1.4:

- **Sign.** As for floats or signed integers, the first bit stands for the sign (s): 0 for positive numbers, 1 for negative numbers. In contrast to floats, which use a sign-magnitude representation, posits are encoded in two's complement.

- **Regime.** This field is unique to this number format. The regime consists of a sequence of k identical bits R terminated either with the negation of such value ($1 - R = \overline{R}$) or with the least significant bit (LSB) of the posit. This sequence encodes the scaling factor r, given by conditional Equation (1.3). For example, when the regime is 4-bits, pattern `1110` encodes $r = 2$, while `0001` stands for $r = -3$.

$$r = \begin{cases} -k & \text{if } R = 0 \\ k - 1 & \text{if } R = 1 \end{cases}. \tag{1.3}$$

- **Exponent.** The following es bits encode another scaling factor e. Unlike with floats, the exponent is unbiased. As the length of the regime field is variable, there exists the possibility that some exponent bits (or even all of them) are shifted out of the bit-string. In such cases, missing bits are considered as `0`.

- **Fraction.** The remaining bits after the exponent correspond to the fraction field. Its value $0 \leqslant f < 1$ is given by dividing the unsigned integer encoded in this field by 2^m. This is similar to the case of floats, with the only difference being that the hidden bit depends on the sign rather than on the exponent, so denormalized numbers do not exist in the posit format.

The real value p of a generic Posit$\langle n, es \rangle$ is given by Equation (1.4)

$$p = (-1)^s \times \left(2^{2^{es}}\right)^r \times 2^e \times (1 + f). \tag{1.4}$$

At this point, it is noteworthy to highlight the difference between the two interpretations that have been made of posit numbers to date: in the first works studying posit arithmetic [52], the numerical value of a normal posit datum was defined by Equation (1.4). Under this approach, if a value is negative (when the sign bit is 1), its two's complement must be computed before extracting the regime, exponent, and fraction, so values r, e and f in Equation (1.4) are always considered from the absolute value of the posit.

Notice that this interpretation, usually known as sign-magnitude, is quite similar to the one for classical floating-point numbers: it deals with a sign bit, a signed exponent (regime and exponent can be gathered in a single scaling factor) and a significand with a hidden bit. As a consequence, the circuit design for both arithmetic formats would be similar too. Indeed, this sign-magnitude representation is the one used by most of the posit arithmetic units in the literature [52, 15, 72].

However, posits (and Type II unums) were designed from the key observation that signed (two's complement) integers map elegantly to the projective reals [52]. Trying to implement posits by first forcing them to look more like floats (taking the two's complement before decoding) and then converting back does not seem optimal [48]. Instead, by considering posit numbers to be in two's complement representation much more efficient designs can be obtained. In this sense, we propose Equation (1.5) as an alternative toEquation (1.4) to obtain the value of a posit number:

$$p = ((1 - 3s) + f) \times 2^{(1-2s)\times(2^{es}r+e+s)}. \tag{1.5}$$

In this thesis, we prove that both approaches are equivalent from a mathematical sense (for more details, see Appendix A). Indeed, this is the representation under which posit numbers are commonly interpreted throughout this thesis, and it serves as the basis for presenting the designs of the functional units (FUs) in Chapter 5. However, this implies that many of the existing units for FP cannot be reused, since this format just considers positive significands, and can not exploit two's complement properties. This means that arithmetic units have to be re-designed to accommodate the alternative posit representation. Actually, the community is still in the early stages of discovering new representations and circuit shortcuts that leverage the posit arithmetic format.

Another noteworthy aspect of posit arithmetic is that in the latest Standard for Posit™ Arithmetic (2022) [51] the value of es was fixed to 2. This has the advantage of simplifying the hardware design and facilitates the conversion between different posit sizes. Therefore, from now on we will be using the Posit$n$ to refer to posits of length n bits with 2 exponent bits.

There are some main differences with the IEEE 754 floating-point format. First, the use of an unbiased exponent. Another difference is in the value of the fraction hidden bit.

Table 1.1: Posit8 format decodification.

Binary	s	r	e	f	value
00000001	0	-6	0	0	$5.960\,46 \times 10^{-8}$
00001011	0	-3	1	0.5	$0.000\,732\,421\,875$
01000000	0	0	0	0	1.0
01101011	0	1	2	0.75	112
01111111	0	6	0	0	$16\,777\,216$
10000001	1	-6	0	0	$-16\,777\,216$
10010101	1	-2	1	0.25	-112
11000000	1	0	0	0	-1.0
11110101	1	2	2	0.5	$-0.000\,732\,421\,875$
11111111	1	6	0	0	$-5.960\,46 \times 10^{-8}$

Usually, in floating-point arithmetic, the hidden bit is considered to be 1. However, when considering the representation of posits given by Equation (1.5), it might be either 1 if the number is positive, or -2 if the number is negative. This simplifies the decoding stage of the posit representation [48, 124]. But probably the major difference with standard floats is the existence of the regime field. This variable-length field acts as a long-range dynamic exponent, as can be seen in Equation (1.5), where it is multiplied by 2^{es} or, equivalently, shifted left by the number of exponent bits. Since it is a dynamic field, it can occupy more bits to represent larger numbers or leave more bits to the fraction field when looking for accuracy in the neighborhoods of ± 1. Table 1.1 shows several examples of posit values and field decodification. However, detecting these variable-sized fields adds some hardware overhead, as will be shown in Chapter 5.

1.2.3. Properties of posit arithmetic

The posit format provides compelling advantages over IEEE 754 floating-point numbers. Such advantages range from a more elegant design (in a mathematical sense) to mechanisms that reduce the rounding error in calculations. Below, some unique properties that make posits an interesting alternative to floats are presented.

Special cases

As aforementioned, posit numbers are encoded in two's complement notation, in contrast to floats, which use the sign-magnitude representation. Hence, posit format has no representation for negative zero, and thus does not suffer from its associated difficulties. The single representation for zero value in posit arithmetic is, as one may expect, with all bits set to 0. The other bit pattern that has no complementary value is a 1 bit preceded by all 0 bits, which is used to represent the NaR exception. This value is used for denoting anything not mathematically definable as a unique real number, like division by zero or the square root of a negative number. There is no representation for the infinite value in posit arithmetic, since it is not a useful value for arithmetic calculations, but denotes a non-representable result.

According to posit standard [51], any real number should be rounded to the closest posit value, so there is no overflow or underflow in such format, and thus infinity representation is not needed.

Floats present two different representations for positive and negative infinities, again according to the sign bit. In addition, the IEEE 754 format provides NaN exceptions, which are not the same as infinity, although both are typically handled as special cases. In the floating-point standard, bit strings with all the exponent bits set to 1 represent a NaN. Just the first fraction bit is used to determine the type of NaN (*quiet* or *signaling*), while the rest of the bits are often ignored in applications. The representation of NaNs specified by the standard has some unspecified bits that could be used to encode the type or source of error, but there is no standard for that encoding. Besides this, any implementation compliant with the IEEE 754 standard must incorporate mechanisms to detect NaNs in arithmetic and comparison operations. Moreover, when a calculation produces any of these values, an exception also occurs. NaN is unordered: it is not equal to, greater than, or less than anything, including itself. The fact that the posit format has a single exception value allows not only to represent more values with the same number of bits but to tremendously simplify the exception handling when designing posit units.

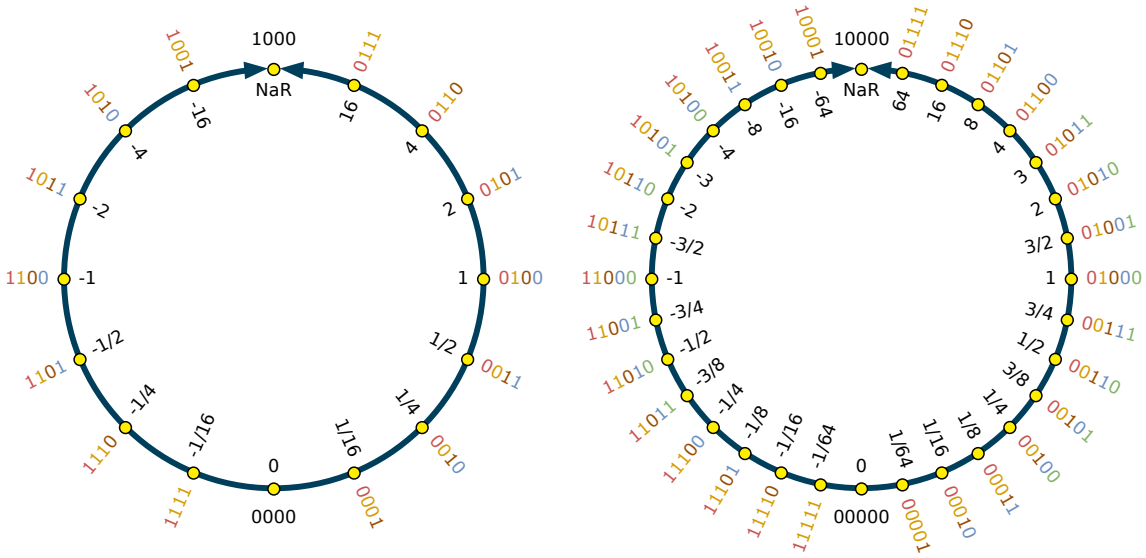
Overflow, underflow, rounding and reproducibility

As mentioned earlier, in posit arithmetic there are no overflow to NaR and underflow to 0 phenomena. When the result of an operation is greater (respectively lower) than the maximum (respectively minimum) representable posit number, the result is rounded to that extreme value. For all other cases where a real number is not exactly representable in a posit format, the standard specifies the use of the *round-to-nearest-even* scheme, i.e., if two posits are equally near, take the one with binary encoding ending in 0. This single rounding rule ensures obtaining bitwise identical results across systems, which is not guaranteed in the IEEE 754 as it defines up to five different rounding rules.

Associativity and fused operations

In the realm of integer computation, operations are generally exact. However, the same cannot be said for floating-point numbers and posits, which possess limited mathematical precision. Therefore, during any operation involving these numbers, the final result is typically rounded to conform to the original format. This characteristic of digital floating-point arithmetic results in a potential loss of accuracy due to round-off errors. As a consequence, the associative laws of algebra do not necessarily hold for this arithmetic format.

Nevertheless, if the rounding process were postponed until the final operation in computations requiring multiple operations, the associative laws of algebra would still apply to computer number systems. To address this issue, the concept of *fused operations* has been introduced in computer arithmetic. Fused operations combine multiple arithmetic operations into a single operation, eliminating the need for intermediate result rounding. The most common type of fused operation is the fused multiply-add (FMA), which computes the product of two numbers and adds the result to a third number in one step, with only one rounding taking place. The FMA can speed up and improve the accuracy of many



(a) Posit⟨4, 1⟩. **(b)** Posit⟨5, 1⟩.

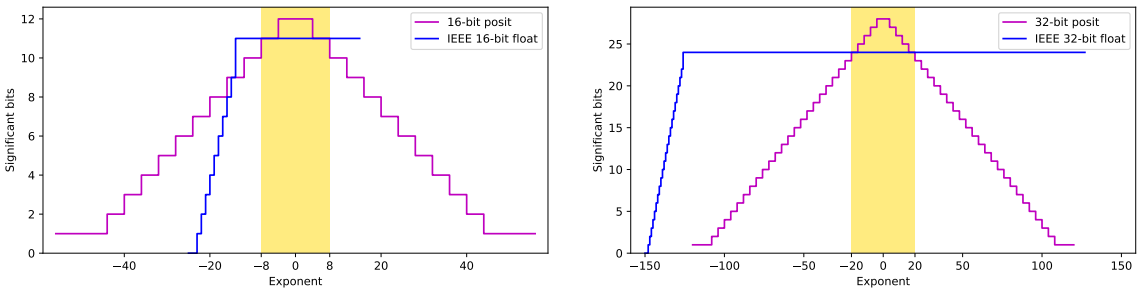
Figure 1.5: Visual representation in the real projective line of different posit formats.

computations that involve the accumulation of products, such as dot products, matrix multiplications or convolutions [112]. It was not until the revision of the IEEE 754 standard in 2008 [65], that the standard included the FMA operation in its requirements. The posit arithmetic goes one step further and includes fused expressions as part of the standard [51]. Such expressions must be distinct from non-fused expressions in source code, and any standard-compliant implementation must be able to evaluate accurately any expression that can be written as a product of vectors of length less than 2^{31}, with a single final rounding to posit format. To that end, posit arithmetic introduces the concept of *quire*: a fixed-point two's complement format capable of storing exact sums and differences of products of posits. For each posit precision, there is a corresponding quire format with $16n$ bits of precision. Such accumulator is based on the concept presented by Kulisch [87], but in contrast with previous implementations, the quire must be accessible to the programmer, so it might be possible to perform load/store and basic arithmetic (addition/subtraction) operations on it.

Visualization of posits in the projective real line

Posits were created as a modification of Type II unum to be more similar to the floating-point numbers. Therefore, the posit format inherits one of the concepts of the previous unum type, the design based on the projective reals. Posits can be mapped into the real projective line[1] as depicted in Figure 1.5, which helps to visualize and understand this novel format. Apart from the elegant mathematical design, these geometric representations reveal some properties of the posit format.

[1]In geometry, a real projective line is an extension of the usual concept of line that incorporates a point at infinity; i.e., the one-point compactification of $\mathbb{R}$.



(a) 16-bit formats.

(b) 32-bit formats.

Figure 1.6: Significant binary digits precision of different number formats.

- As aforementioned, and in contrast with the IEEE 754 standard, posits have a single representation of the zero and NaR values, and the latter is the only exception value used in this format.
- Posit numbers with their corresponding bit strings constitute a totally ordered set. As there are no two posit numbers with the same bit representation, it is a strict total order relation. Thanks to this property, posits comparison can be done in the same manner as for signed integers. However, this is not possible in the case of floats, where different bit strings may represent the same real value (like $0 = -0$).
- Generating a posit format of a bigger size consists of appending a `0` bit on the right to the existing posits, and filling the gaps with posits ending in `1`, as depicted in Figure 1.5. Note that appending a `0` on the right does not modify the original value of the posit, so conversion between posit formats with the same number of exponent bits just requires padding with zeros, or chopping (and rounding) the rightmost bits.
- For every posit configuration, about half of the representable values concentrate within the interval $[-1, 1]$. This makes posit values to follow a normal distribution centered at 0, which is known as tapered precision. This contrasts with the flat precision of the floating-point format, as shown in Figure 1.6 and in Table 1.2. Note that, a total precision of X bits is equivalent to $log_{10}(2^X)$ decimal digits. For this reason, posits present a higher accuracy in that interval, where most of the computations from different areas take place, such as deep learning, digital signal processing or numerical analysis [15, 148, 168].

1.3. Objectives

With the aim of addressing the aforementioned scenarios, this dissertation primarily focuses on the following objectives:

(I) *"To explore the characteristics and properties of the posit format, assessing its viability as an alternative to the prevailing IEEE 754 floating-point format in modern applications."*

(II) *"To propose designs for posit arithmetic units that offer hardware support for this format, with a special emphasis on enhancing energy-efficiency and performance."*

Table 1.2: Span and precision of IEEE 754 floating-point and posit formats.

Format	Min. Subnormal	Min. Normal	Max. Finite	Max. Precision
Float16	$2^{-24} \approx 5.96 \times 10^{-8}$	$2^{-14} \approx 6.1 \times 10^{-5}$	$(2 - 2^{-10}) \times 2^{15} = 65520$	11
Posit16	–	$2^{-56} \approx 1.39 \times 10^{-17}$	$2^{56} \approx 7.21 \times 10^{16}$	13
Float32	$2^{-149} \approx 1.4 \times 10^{-45}$	$2^{-126} \approx 1.18 \times 10^{-38}$	$(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$	24
Posit32	–	$2^{-120} \approx 7.52 \times 10^{-37}$	$2^{120} \approx 1.33 \times 10^{36}$	28
Float64	$2^{-1074} \approx 4.9 \times 10^{-324}$	$2^{-1022} \approx 2.23 \times 10^{-308}$	$(2 - 2^{-52}) \times 2^{1023} = 1.8 \times 10^{308}$	53
Posit64	–	$2^{-248} \approx 2.21 \times 10^{-75}$	$2^{248} \approx 4.52 \times 10^{74}$	60

These general goals can be further divided into the following specific objectives:

- To thoroughly study the posit number system, including its key characteristics and distinctions from other commonly used formats in computer arithmetic.
- To investigate the potential advantages of posit arithmetic over other number formats in diverse applications and scenarios.
- To develop a platform capable of conducting both training and inference phases in deep learning systems. The platform will facilitate computations using either floating-point or posit arithmetic, enabling a fair and comprehensive comparison between the two formats without requiring modifications to the models.
- To design and implement units in the posit format for basic arithmetic operations, including addition/subtraction, multiplication, division, and square root. In addition, multiplication with accumulation is a computation common in many scientific applications that will be considered in the realm of fused arithmetic.
- To integrate the aforementioned designs into hardware development tools to enable the execution of posit computations on real hardware platforms, such as FPGAs.
- To evaluate the accuracy, performance, resource utilization, and energy efficiency of the proposed approaches, and to compare the achieved results with other state-of-the-art solutions.

1.4. Document structure

Clearly, there are two strongly differentiated approaches in this thesis: one with a more software-centric approach and the other purely hardware. This document is structured into two initial introductory chapters, and three main parts, the first two gathering all the approaches proposed, plus a third part presenting alternative formats derived from the standard posit.

Chapter 1 (this chapter) provides the motivation for this thesis, describes the main and specific objectives addressed, and summarizes the different approaches described in this dissertation. Chapter 2 reviews the current state of the art about the realm of posit arithmetic, showing different approaches to both software and hardware topics, the former with a special focus on the field of deep learning.

Part I explores the properties of the posit format in a wide variety of applications and scenarios. The experiments in both chapters are performed using software to emulate position computations. Chapter 3 motivates the usage of posits in scientific applications of multiple areas, including linear algebra or dynamic systems. For each application, the error of the posit format is compared with that obtained under floating-point computation. Chapter 4 is specifically focused on the usage of posits in the computations of deep neural networks (DNNs). First, a deep learning framework is presented, allowing to emulate all the internal computations in different posit formats at both inference and training phases. Then, the results of performing training and low-precision inference under floating-point and posit formats are compared across multiple models and datasets.

In contrast, Part II is devoted to designing a complete suite of hardware operators for performing computations with posit arithmetic. The proposed solutions aim to be energy-efficient and competitive against current floating-point operators. The first two chapters within this part follow the same structure, introducing the operation that wants to compute, a detailed description of the design proposed, and a complete evaluation of the hardware implementation and comparison with previous solutions. Chapter 5 presents functional unit designs for fundamental operations in posit format, encompassing addition/subtraction, multiplication, division, and square root. Comparative analyses between the proposed posit units, contemporary designs, and floating-point standard units are presented. Moreover, a detailed study of the approach employed in the proposed designs is presented in Appendix A. In Chapter 6, the novel concepts of fused arithmetic and the quire accumulator are explored, introducing a functional unit designed for the computation of exact operations with no rounding. In addition, diverse implementation alternatives are investigated for this unit, elucidating multiple trade-offs between area and performance. Finally, Chapter 7 presents a complete solution to generate and synthesize custom accelerators for FPGAs from behavioral specifications employing posit numbers instead of conventional FP. This is achieved through high-level synthesis (HLS), which transforms a behavioral description of a digital system into an RTL model that realizes the given behavior.

The last part of this document, Part III, deals with some alternative formats that are derived from the initial idea of posit arithmetic. These formats attempt to reduce the hardware implementation cost of the standard posit format, at the cost of sacrificing some accuracy or compatibility with the standard posit format. The paradigm of approximate computing is explored in Chapter 8. Although approximate computing refuses to guarantee exact computations, it can still achieve acceptable results in certain applications. Another different format, HUB, is explored in combination with the posit number system. This format is known for diminishing the implementation resources while not compromising the accuracy of the computations.

Finally, Chapter 10 presents the general conclusions of this thesis, a list of the publications and open-source repositories derived from it, and a set of open research lines.

2

State of the art

Since John L. Gustafson introduced the posit number system in 2017 [52], there has been a surge in interest regarding scenarios where this format can outperform traditional floating-point representations. Numerous research works have extensively evaluated the advantages and limitations of posits across various domains within computer science. These evaluations range from high-level applications, including physical simulations and neural networks, to the intricate landscape of hardware implementation.

This chapter provides a comprehensive overview of the current state of the art in posit arithmetic, synthesizing key contributions published to date. We categorize these contributions into two main streams: those predominantly focused on applications and software, and those situated within the realm of hardware implementation.

2.1. Posits in software

The applications of posit arithmetic span a wide spectrum of computational domains. Researchers have explored its utility in high-level applications, such as physical simulations [32], HPC [20], or neural networks [153]. This section provides a comprehensive overview of the diverse applications of posit arithmetic, discussing its impact on computational performance, accuracy, and overall efficiency.

2.1.1. Software libraries

Posit arithmetic exhibits many implementation differences when compared to classical floating-point. Naturally, the appropriate hardware is not always available, and developing new hardware is always much more expensive than performing software simulations. Therefore, various software libraries for emulating this arithmetic have been proposed. The two most well-known ones so far are SoftPosit [97] and Universal [137].

SoftPosit is a C library that follows the philosophy of the Berkeley SoftFloat[1] software implementation of binary IEEE floating-point. It supports basic arithmetic operators and

[1] https://github.com/ucb-bar/berkeley-softfloat-3

quire functionality for Posit⟨8, 0⟩, Posit⟨16, 1⟩, and Posit⟨32, 2⟩ formats, as well as Posit⟨n, 2⟩ with $n \in [2, 32]$. Bindings of this library to different programming languages like Python[2], Haskell[3], Rust[4], Racket[5], and Julia[6] have also been done.

The other main library is known as the Universal Numbers Library. It is written as a template C++ library and provides emulation support for posit formats of (virtually) any bit length and exponent size, including quire functionality. Additionally, it supports other alternative formats to the IEEE 754 floating-point, such as arbitrary configuration floating-point, fixed-point, or logarithmic number systems, among others.

Another software library that emulates posit operations in a much less faithful way is cppPosit [25]. This library supports posits with a total number of bits ranging from 4 to 64. However, posit operations are emulated with the help of pre-computed lookup tables (LUTs) (when the number of bits is no more than 12) or using conversion to either floating-point or fixed-point. Also, it considers certain naive operations that can be directly executed in hardware without emulation, such as multiplication and division by two.

Lastly, it must be noted that some works have focused on evaluating several numerical aspects of this format, developing software frameworks for such purposes. In [99] posits are compared with other formats with variable-length coding for exponents and significands. A tool for detecting errors in computation using posits is presented in [22], whereas a suite for the verification of posit arithmetic units is proposed in [88]. On the other hand, regarding the resiliency and robustness of this number format, some works have evaluated it [2, 149, 41], showing that the posit format is considerably more resilient compared to classical IEEE 754 compliant representation. That is, posit format demonstrates to be less vulnerable and prone to errors such as bit flips.

2.1.2. Scientific applications

Multiple studies have compared the performance of posit arithmetic against the classical IEEE 754 floats in a wide variety of areas and real-world applications.

The use of 16-bit precision formats is investigated for weather and climate modeling [80, 81] and computing the solutions of chaotic dynamical systems [79]. In such simulations, posits clearly outperform floats and appear very promising for accelerating these kinds of computations.

Another potential application for posit arithmetic is optical flow. As demonstrated in [148], there is a clear advantage in using posits instead of floats for calculating optical flow using classical methods like Lucas-Kanade [7].

Neuromorphic computing, an approach inspired by the structure and function of the human brain, utilizes a type of artificial neural network known as spiking neural networks. In [14], the presynaptic region of neurons was implemented in hardware, incorporating posit arithmetic for that purpose. The posit-based implementation outperformed other operating IEEE 754 modes in terms of error, while reducing the area and power consumption compared to double-precision floating-point implementation.

[2] https://gitlab.com/cerlane/SoftPosit-Python
[3] https://hackage.haskell.org/package/posit
[4] https://crates.io/crates/softposit
[5] https://github.com/DavidThien/softposit-rkt
[6] https://github.com/milankl/SoftPosit.jl

Finally, other studies demonstrate the suitability of the posit format for different well-known physical applications and problems, such as dynamical systems [118], the fast Fourier transform [147, 98], Kalman filtering [67], or radio astronomy tasks [47]. All of these works show that using posit numbers produces better results than the corresponding floats of the same size.

2.1.3. Deep learning

Among the various fields of research, deep learning stands out as the domain that currently attracts the most attention for utilizing posit arithmetic. Notably, the original article introducing the posit format [52] highlights a property of this format that proves highly advantageous for neural networks: 8-bit posit with $es = 0$ can approximately compute the sigmoid function through straightforward bit manipulation.

Researchers have been investigating the advantages of posit arithmetic for the inference of DNNs, with the potential to leverage the higher accuracy of this format to reduce data bit length. This exploration began with [73] focusing on 7-9 bits, where also the use of the Kulisch accumulation (the precursor of the fused multiply-accumulate (MAC) introduced in the posit Standard [51]) is evaluated in conjunction with the MAC operation in the log domain. Subsequently, similar investigations into the use of low-precision posit formats for inference are conducted in [12]. In such a work, a precision-adaptable FPGA soft-core for performing exact posit MACs named Deep Positron is used to demonstrate that the 8-bit posit precision achieves a comparable accuracy to that obtained with a 32-bit floating-point implementation. Similarly, [114] examines quantization to 8-bit posits and the use of fused MAC for inference, drawing similar conclusions to other works. Collectively, these studies have demonstrated that 8-bit (and even smaller) posits can effectively serve for the inference of DNNs.

In contrast, exploiting the use of posits during the training phase is a more compelling avenue, given the computational intensity of this stage. The first time posits were used in this context was in [128], where a fully connected neural network is trained for a binary classification problem using posits of varying precisions (8, 10, 12, 16, 32 bits). Subsequent studies [89, 90] extend this approach to training fully connected neural networks for MNIST and Fashion MNIST, employing 16 and 32-bit posits for all DNN parameters. Additionally, [102, 101] explore the training of convolutional neural networks (CNNs) using mixed precision under 8 and 16-bit posits, initially relying on floats for the first epoch and layer computations, with posits emulated through conversion. In contrast, the software framework Deep PeNSieve [114] is used to demonstrate the training of a CNN for CIFAR-10 using only 16 and 32-bit posits, with no discernible accuracy degradation, emulating all computations in such formats. More recently, PositNN, a mixed-precision framework incorporating fused arithmetic for both training and inference with posits of any precision, is presented in [143]. This work showcases the successful use of common CNNs with 8-bit posits, exhibiting no significant loss of accuracy on classical datasets such as MNIST, Fashion MNIST, and CIFAR-10.

Other innovative works that explore the application of posit arithmetic in deep learning encompass the use of posits to reduce storage requirements while performing computations in a more resource-efficient format like fixed-point [130], the utilization of adaptive quantization techniques [91], the emulation of posit arithmetic on GPUs [58, 59], and the

implementation of approximate computing, either in matrix multiplication [121, 125] or for processing the activation functions [24].

2.2. Posit implementation solutions

Efficient hardware implementations are crucial for the practical adoption of posit arithmetic. This section delves into the various implementation solutions devised by researchers to overcome challenges associated with hardware realization and software support. It explores optimization strategies, computational capabilities, and other key aspects that contribute to the successful integration of posit arithmetic into computational frameworks.

2.2.1. Basic arithmetic units

Since its initial introduction, there has been considerable interest in hardware implementations of posit arithmetic. A plethora of standalone posit arithmetic units with different degrees of capabilities have been described in the literature. These units serve as the fundamental building blocks for the execution of posit arithmetic operations.

The first works presenting posit arithmetic units are [141, 15]. Both present parametric designs for posit adders and multipliers, but are not open-source. On the other hand, authors in [72] propose PACoGen, an open-source suite of parameterized posit adder, multiplier and divider units (for $es > 0$) in Verilog, along with a set of pipelined architectures. None of these works support the quire. The work [161] presents MArTo, an open-source hardware implementation for addition, subtraction and multiplication of custom-size posits, in the form of a C++ library compatible with Vivado HLS. This library improves previous posit implementations in terms of latency and resource consumption. Other open-source designs of posit adders and multipliers improving previous implementations are presented in [113], in which FloPoCo, a framework for the generation of arithmetic datapaths [33], is used to generate parameterized units with automatic pipeline according to the design constraints and target FPGA device (see Appendix B.2). Moreover, by leveraging the variable-length nature of posit numbers, a power-efficient posit multiplier is presented in [174], which only enables the required portions of the signified multiplier at run-time. Finally, a faithful evaluation of the hardware requirements of posit adders and multipliers when implemented either as floating-point units, using sign-magnitude coding, or implemented from scratch exploiting the unique properties offered by this new format, i.e. with two's complement coding, is presented in [124]. This study shows a substantial improvement in terms of latency and resource consumption when using the former approach.

In contrast to addition and multiplication, implementing binary division and square root is a much more complex task, and not much work has addressed it. As mentioned above, PACoGen includes an open-source posit divider design [72]. It relies on the Newton-Raphson algorithm. Moreover, the works [144, 171] propose functional units for both division and square root, since the algorithms for the two operations are pretty similar. Indeed, the unit presented in [144] supports the two operations in the same hardware design. However, none of these two works are open-source. While the units from [171] are based on the Newton-Raphson algorithm, the fused division and square-root architecture from [144] is iterative, as it implements the non-restoring algorithm. Lastly, authors in [119] compare

different implementations for posit division, including both iterative and multiplicative methods. Such dividers improve previous posit implementations in terms of latency and resource consumption.

It is noteworthy to mention that several research efforts have been made to develop the fused operations introduced by the posit standard. The first ASIC implementation of a fused posit MAC unit is proposed in [173]. A 5-stage pipeline strategy is also presented to make the design meet the speed requirements of modern processors. Another of the earliest papers to feature a posit MAC unit with quire was [161]. It compares the equivalent floating-point Kulisch accumulator on which this unit is actually based. Results reveal that the cost and performance of a Posit32 quire and a Kulisch accumulator for Float32 are almost identical. Another set of energy-efficient designs for fused posit MAC is presented in [123], where different design alternatives for quire accumulation and multiple pipeline strategies are compared. Finally, a fused posit MACs with dynamic support for variable exponent sizes and a 5-stage pipeline is presented in [131]. This is the first unit able to compute fused posit operations for different exponent sizes within the same hardware.

A summary of prior research studies proposing posit arithmetic hardware implementations is presented in Table 2.1. The works are sorted by publication date. All the works present parametric designs (some of them do not support $es = 0$, but this is not possible to determine for those that are not open-source). However, not all of them have pipelining. Designs implemented in high-level tools such as Vivado HLS or FloPoCo support automatic pipelining, while those implemented in Verilog present a fixed pipeline scheme (and not all designs have pipelining at all, like in the case of PACoGen [72]).

2.2.2. Domain-specific accelerators

One of the most popular current trends in computer architecture is the design of DSAs, rather than general-purpose hardware [56]. Besides the functional units designed for the aforementioned posit arithmetic operations, several works have explored the usage of this format in hardware accelerators that target specific applications.

Authors in [17] propose a matrix multiplier unit for Posit32 numbers. It consists of 64 quire accumulators, so it can compute vector dot products exactly. Another accelerator designed to exploit data parallelism is presented in [132]. Authors propose a 2D array of reconfigurable processing elements, each implementing a 4-stage pipeline SIMD fused MAC unit (with quire) of variable-precision ranging from 8 to 64 bits. Thus, each processing element supports 1 × 64, 2 × 32, 4 × 16, and 8 × 8-bit posit vector operations using the same hardware resources. This work is extended in [29], which presents a vector MAC unit with shared support for the posit and IEEE 754 formats. Such a unit comprises a vectorized variable-precision datapath for 1 × 32-bit, 2 × 16-bit, and 4 × 8-bit vector operations. In addition, to mitigate the hardware overheads associated with the quire, the proposed unit only provides an exact accumulation for low-precision scenarios with standard Posit8 format, by using a quire of 128-bits, and with 8-bit minifloat variant for IEEE 754.

Some hardware accelerators have already been proposed that seek the adoption of the posit format to DNNs. Deep Positron is a DNN kernel accelerator that includes an FPGA soft-core with an exact posit MAC unit for performing inference with $\leqslant$8-bit posit precisions. Thanks to the exact MAC operation, this work demonstrates that the 8-bit posit precision achieves an accuracy comparable to those obtained with a 32-bit IEEE 754 floating-point

Table 2.1: Summary of posit arithmetic hardware implementations.

	Operations	Parametric	Pipeline	Quire support	Implementation	Device	Open-source
Podobas *et al.* [141]	$\pm, \times$	✓	✓	✗	Custom (output VHDL)	Stratix V FPGA	✗
Chaurasiya *et al.* [15]	$\pm, \times$	✓	✗	✗	–	Zynq-7000 FPGA & ASIC	✗
Zhang *et al.* [173]	MAC	✓	✓	✓	Verilog	ASIC	✗
Jaiswal *et al.* [72]	$\pm, \times, \div$	✓	✓*	✗	Verilog	Virtex-7 FPGA & ASIC	✓
Uguen *et al.* [161]	$\pm, \times$, MAC	✓	✓	✓	Vitis HLS	Kintex-7 FPGA	✓
Raveendran *et al.* [144]	$\div, \sqrt{}$	✓	✓	✗	Verilog	Virtex UltraScale FPGA	✗
Zhang *et al.* [174]	$\times$	✓	✗	✗	Verilog	ASIC	✗
Xiao *et al.* [171]	$\pm, \times, \div, \sqrt{}$	✓	✗	✗	–	Virtex-7 FPGA	✗
Murillo *et al.* [113]	$\pm, \times$	✓	✓	✗	FloPoCo	ASIC	✓
Neves *et al.* [131]	MAC	✓	✓	✓	–	Virtex-7 FPGA & ASIC	✗
Murillo *et al.* [123]	MAC	✓	✓	✓	FloPoCo	ASIC	✓
Murillo *et al.* [124]	$\pm, \times$	✓	✓	✗	FloPoCo	ASIC	✓
Murillo *et al.* [119]	$\div$	✓	✓	✗	FloPoCo	ASIC	✓

implementation. This work is later extended in [90], where the authors present *Cheetah*, a framework for low-precision arithmetic to both DNN training and inference. The *Cheetah* software framework evaluates the performance of various numerical formats (fixed-point, floating-point, and posit) with different quantization approaches for both feedforward and CNNs on various datasets. It includes the FPGA soft-core from Deep Positron.

2.2.3. Posit-based RISC-V implementations

The integration of posit arithmetic units into hardware cores represents a crucial step in the execution of entire posit algorithms and an in-depth exploration of the efficiency of this numerical representation. When designing such a core and its arithmetic operations, a critical decision lies in selecting the instruction set architecture (ISA) to implement. RISC-V [167], an open-source ISA based on established reduced instruction set computer (RISC) principles, stands out as a promising choice that has garnered significant attention in both academic and industrial circles. Thanks to its openness and flexibility, multiple RISC-V cores have been developed in recent years, targeting diverse purposes.

The first works integrating posit arithmetic into RISC-V cores did it by replacing the existing FPU with the corresponding units in posit format. PERC [146] integrates a posit processing unit into the Rocket Chip generator, overloading the already existent 32 and 64-bit FPU. However, this work did not include quire support, as it is constrained by the "F" and "D" RISC-V extensions for IEEE 754 floating-point numbers. More recently, PERI [159] adds a tightly coupled posit processing unit into the SHAKTI C-class core, a 5-stage in-order

RV32IMAFC core. This proposal did not include quire support either, as it reuses the "F" extension instructions. Nonetheless, it allows dynamically switching the exponent size of posits (es) between 2 and 3. In [23] authors include a posit unit named POSAR into a RISC-V Rocket Chip core. Again, this proposal did not include quire support and replaces the FPU present in Rocket Chip to reuse the floating-point instructions.

A different approach is presented in [150]. The proposed framework, CLARINET, introduces custom instructions for incorporating fused operations on the quire data type into a RV64GC 5-stage in-order core. However, not all posit capabilities are included in this work. Most operations are performed in floating-point format, and the values are then converted to posit when using the quire. The only posit functionalities included within the core are fused MAC with quire, fused divide and accumulate with quire and conversion instructions. Another work that presents a RISC-V extension for posit is [26]. Nevertheless, this extension only allows the conversion between 8 or 16-bit posits and floating-point or fixed-point formats, while performing the real computation in such backends, therefore using posit format as lossy compressed information storage. Similarly, the same authors propose a RISC-V vector extension for posits, but again relying on the cppPosit library, and thus limited to floating-point conversion or simple operations that can be executed as integers, like multiplication by 2 [25].

Finally, PERCIVAL [105, 106] integrates a posit processing unit into the CVA6 core without replacing the existing FPU. This work incorporates support for all fundamental posit arithmetic operations, conversions, and fused operations using the quire register. Such a design choice renders PERCIVAL a platform suitable for a fair comparison between posit and floating-point formats. To facilitate the utilization of PERCIVAL, a RISC-V custom extension, Xposit, is integrated into the LLVM backend to allow the compilation of high-level applications targeting posit arithmetic. The initial version of PERCIVAL, with support for just 32-bit posits, was subsequently extended to 64-bit posits [103].

2.3. Other posit number representations

While the standard posit format serves as the foundation, researchers have explored alternative posit number representations. This section examines the variations and extensions to the standard posit format, discussing their implications and potential advantages in specific contexts. By surveying the landscape of posit number representations, we gain valuable insights into the adaptability and versatility of this emerging numerical paradigm.

One of the first approaches that researchers applied to posit arithmetic to reduce delay and energy consumption of hardware units is approximate computing. The first work proposing the use of this paradigm within posit arithmetic is [121], which presents an approximate multiplier based on computations in the logarithmic domain. Furthermore, the authors in [121] train different DNNs using the Posit$\langle 16, 1\rangle$ format for various datasets, and during inference, they replace the accurate posit multipliers with their proposed approximate multipliers, resulting in similar output accuracy. Another approximate posit multiplier, this time iterative, is presented in [133]. Other approaches that leverage posit approximate computing in DNNs were later presented in [166, 69]. In [166], a posit-based processing element is proposed for simplifying the MAC operations required during DNN training. However, the proposed unit still relies on the floating-point format for the accumulation

of intermediate results. On the other hand, authors in [69] explore multiple operator-level approximations, including posit addition, multiplication and MAC. A different approximate posit multiplier is proposed in [175], which modifies the exact fraction multiplier by truncating the LSB positions. This approach is also extended in the logarithmic domain, and both approximate posit multipliers are evaluated for the inference of various DNNs with 16-bit posits, resulting in negligible accuracy degradation. Finally, the work from [121] is extended in [125] with support for approximate posit division and square root.

To address the limitations inherent in the posit format, authors in [44] establish a fixed configuration for the number of bits in the regime and exponent fields of the posit representation, introducing the *fixed-posit* format as a 3-tuple (bit length, exponent bits, and regime bits). This alternative format is used for DNN inference in [165], resulting in faster and more energy-efficient computations, and in [43], where the authors present an approximate fixed-posit multiplier that reduces area, and energy, increases the speedup and achieves the same accuracy compared with a 16-bit floating-point format.

Lastly, the HUB format, previously employed in the domain of floating-point representation [62, 61] is integrated with posit arithmetic for the first time in [122]. Although incompatible with the standard posit, this alternative format offers hardware implementations with reduced resource consumption compared to conventional posit units while maintaining a similar level of accuracy.

Part I

Posit Arithmetic in Software Applications

3

Use of posit arithmetic in scientific computing

The utilization of low-precision formats, such as bfloat16, or novel arithmetic systems like posit, has garnered increasing interest not only in domains such as machine learning and HPC but also in general-purpose numerical algorithms. The adoption of these non-standard formats has the potential to reduce hardware requirements and enhance accuracy. However, the availability of dedicated hardware for emerging arithmetics, such as posit, is not always guaranteed, and the development of such specialized hardware incurs substantial expenses. Consequently, simulating diverse numerical precisions becomes imperative to experiment with alternative formats that have not yet been implemented in hardware.

In this chapter, a comprehensive comparison of multiple scientific applications is presented, highlighting the outcomes when computed under various precision levels and arithmetic formats distinct from the IEEE 754 standard. The computations involving posit arithmetic are conducted through software, leveraging the Universal Numbers Library [137]. This approach enables a thorough exploration of the performance implications associated with different numerical precisions and arithmetic models, providing insights into the feasibility and advantages of adopting these non-standard formats in various scientific applications.

Specifically, Section 3.1 motivates the importance of selecting an appropriate number format for certain computations. Section 3.2 presents the concept of decimal accuracy, and provides a comparison across different benchmarks. A more in-depth overview of how the error introduced by intermediate roundings can affect applications is shown in Section 3.3. Finally, Section 3.4 closes the chapter with the main contributions and motivation for the following chapters.

3.1. A motivational example: Goldberg's thin triangle problem

Let us consider the following elementary-school algebra problem: computing the area of a triangle of known sides. Assume sides a, b, and c, with the condition $a \approx b + c$, so the triangle is very thin [45]. Specifically, let sides b and c be just 3 unit in the last places (ULPs) longer than half of the longest side a (see Figure 3.1).

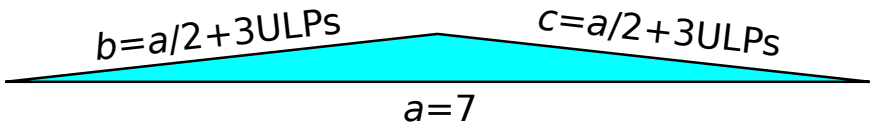


Figure 3.1: Goldberg's thin triangle problem.

The area of a triangle can be directly expressed in terms of the lengths of its sides a, b, and c, as given by the formula:

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \text{ where } s = (a+b+c)/2. \tag{3.1}$$

Equation (3.1) poses a risk of incorrect rounding, especially for a very thin triangle, where s is very close to a. In other words, the term $(s-a)$ subtracts two nearby numbers, one of which may have a rounding error. For this analysis, 32-bit IEEE 754 floats (single-precision) are employed, setting $a = 7$, $b = c = 7/2 + 3 \times 2^{-22}$. Then the same formula is evaluated using Posit32. The correct answer (obtained using an arbitrary precision system like SageMath[1]) and the computed results are as follows:

- Correct answer: $7.831550660045\ldots \times 10^{-2}$
- 32-bit IEEE float result: $9.043096564710\ldots \times 10^{-2}$
- 32-bit posit result: $7.8315506689 25\ldots \times 10^{-2}$

As observed, not a single correct decimal digit is obtained with floating-point representation. Instead, posits yield a significantly more accurate result. This outcome is not unexpected, as single-precision posits exhibit higher accuracy than floats within the dynamic range from 2^{-16} to 2^{16}.

This illustrates that, for specific problems or calculations, the choice of arithmetic format can profoundly impact the accuracy of the results.

3.2. Decimal accuracy

One of the claims of posit arithmetic is that it offers superior accuracy compared to the standard IEEE 754 floating-point numbers (floats). Consider, for example, the following 2×2 linear system of equations presented in [52]:

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

It is not difficult to verify that the solution to System (3.2) is $x = -1$, $y = 2$. Since the system is 2×2, Cramer's rule provides a straightforward method for computing the solution. To mitigate rounding errors during the conversion of decimal numbers to binary representation, System (3.2) can equivalently be expressed as:

$$\begin{pmatrix} 25510582 & 52746197 \\ 80143857 & 165707065 \end{pmatrix} \frac{1}{2^8} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 79981812 \\ 251270273 \end{pmatrix} \frac{1}{2^8}. \tag{3.3}$$

[1] `https://www.sagemath.org`

Table 3.1: Result and accuracy for different benchmarks.

	Thin triangle area	$x^n/n!$	Quadratic formula (r_1)	Quadratic formula (r_2)
Correct answer	0.007831550660045...	0.032797155836709...	-0.020012014421636...	-33.313321318911696...
Float32	0.009043096564710...	0.032797154039144...	-0.020011901855468...	-33.313320159912109...
Posit32	0.007831550668925...	0.032793700695037...	-0.020012060878798...	-33.313321590423583...
Decimal accuracy				
Float32	1.204	7.623	5.612	7.820
Posit32	9.307	4.339	5.996	8.451

The values of the revised System (3.3) can be accurately represented using double-precision (64-bit) IEEE floats. However, when applying Cramer's rule with double-precision IEEE floats, the computed result is $x = 0$, $y = 2$. This substantial rounding error is produced due to an underflow on the numerator expression x. Achieving the exact answer necessitates the use of quadruple-precision IEEE floats (128 bits). Nevertheless, the exact solution can be obtained by conducting the same computation using Posit$\langle 64, 3\rangle$ format. As illustrated, in certain circumstances, high-precision floats can be safely replaced by lower-precision posits.

When dealing with computations that do not produce the exact result, it is necessary to measure how good the computed result is. It is thus necessary to define a metric for error analysis. Accuracy is the inverse of error. Decimal accuracy refers to the precision or correctness of a numerical result in terms of the number of decimal places [52]. It is a measure of how closely a calculated or observed value aligns with the true or expected value, specifically focusing on the digits after the decimal point. The formula for decimal accuracy is defined by Equation (3.4),

$$decimal\ accuracy = -\log_{10}(|\log_{10}(x/y)|), \tag{3.4}$$

where x and y are respectively the correct value and the computed value when using a rounding system like floats and posits. Obviously, the higher this metric is, the better accuracy the numerical format offers.

For example, if the exact result of an operation is 3.141 59, but the computed result is rounded to two decimal places (3.14), the decimal accuracy would be 3.66. However, if the result is rounded to three decimal places (3.142), a higher decimal accuracy is obtained, 4.25.

Next, the decimal accuracy of floating-point and posit formats are compared. Table 3.1 presents the results of multiple benchmarks from the literature [49, 52, 45], as well as the decimal accuracy with respect to the correct answer. These benchmarks are:

- the area of the thin triangle presented in Section 3.1, with sides $a = 7$, $c = b = a/2 + 3 \times 2^{-22}$,
- the fraction $x^n/n!$ for $x = 7$, $n = 20$,
- the two roots from the quadratic formula for $a = 3$, $b = 100$, $c = 2$.

The incorrect decimal digits are highlighted in red. Just the second benchmark is favorable to IEEE 754. The reason for this lies in the computation of the factors x^n and $n!$, which for the case $n = 20$ turn to be values that fall out of the golden zone of posits, thus resulting in lower accuracy. As stated in [32], this is the same explanation why Posit32 is unable to represent most of the common physical constants accurately, as demonstrated in Table 3.2.

Table 3.2: Decimal accuracy for representation of physical constants.

	Planck constant	Avogadro number	Speed of light	Electron charge	Boltzmann constant
Float32	8.727	8.075	7.839	8.004	7.782
Posit32	1.184	4.091	6.969	4.974	4.037

A solution to this would be applying scaling. As mentioned in [32], physicists sometimes use scaled units rather than the International System of Units. On the other hand, it must be noted that for such scientific computations, double-precision floats are usually preferred over 32-bit formats. In any case, there are some studies that report the superiority of 64-bit posits with regard to IEEE 754 double-precision when applied to algebraic problems, such as solving linear systems [103].

3.3. Error-sensitive applications

As shown in the preceding section, posits demonstrate superior accuracy under specific circumstances when compared to traditional floating-point formats. This inherent advantage makes them particularly well-suited for domains where even the slightest deviation can have significant consequences. This is particularly pertinent in error-sensitive applications, where precision and accuracy are of paramount importance. In this context, the adoption of posits presents itself as a promising avenue.

To verify that hypothesis, this section takes a comprehensive evaluation of both floating-point and posit formats across various applications where errors exert a substantial impact. Through this analysis, we aim to discern the efficacy of posits in mitigating errors and enhancing precision in scenarios where the consequences of inaccuracies are notably pronounced.

3.3.1. Benchmark description

A variety of time-dependent ordinary differential equations (ODEs) are proposed. More precisely, this evaluation focuses on systems that exhibit sensitive dependence on initial conditions for certain parameters; that is, tiny differences in the starting conditions (or intermediate steps) for the system rapidly become magnified [177]. Each experiment is computed using different formats for floating-point and posit number formats. The following problems are considered:

- Lorenz system with parameter values $\beta = 8/3$, $\rho = 28$ and $\sigma = 10$, and initial conditions $x = 8$, $y = 1$, $z = 1$. The model is a system of three ODEs, which describe the rate of change of three quantities with respect to time: convection, horizontal temperature and vertical temperature variation.

- Duffing oscillator, a non-linear second-order differential equation used to describe the motion of damped oscillators. This second-order ODE can be rewritten as a system of two first-order ODEs. If the parameters of the system are $\alpha = -1$, $\beta = 1$, $\gamma = 0.5$, $\delta = 0.3$ and $\omega = 1.2$, the system exhibits a chaotic behavior.

Table 3.3: Global MSE for different benchmarks.

	Lorenz	Duffing	Double pendulum
Float32	2.650×10^{1}	2.358×10^{-2}	1.043×10^{3}
Posit32	1.938×10^{1}	6.385×10^{-5}	6.594×10^{2}
Float64	7.334×10^{-15}	5.595×10^{-19}	2.339×10^{2}
Posit64	1.114×10^{-16}	1.275×10^{-24}	9.641×10^{1}

- Double-rod pendulum (also known as a chaos pendulum), which is a pendulum with another pendulum attached to its end. In this case, both pendulums are considered to have the same mass and length (equal to 1), with no initial velocity, and initially located at an angle of 120° and –10° degrees, respectively for the first and second pendulum. The motion of a double pendulum can be modeled as a system of four equations that compute the center of mass of each pendulum (in two dimensions).

To solve the aforementioned systems, the Runge-Kutta fourth-order method is used iteratively. Each system undergoes computation for 200 000 iterations with different time steps, namely 200 µs, 500 µs, and 1 ms for the Lorenz, Duffing, and double pendulum problems, respectively. As posit formats are not supported in commercial general-purpose processors, such computations are emulated via software. Given that the primary emphasis of this evaluation is on accuracy rather than timing performance, the utilization of a software library is not considered inconvenient.

3.3.2. Results

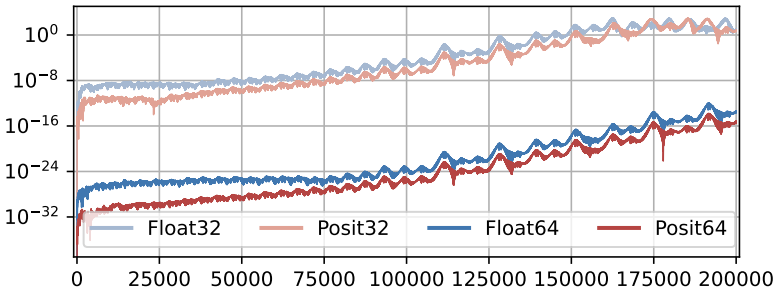
As a baseline result for comparison, the aforementioned systems were computed using the x86 extended precision format (80-bit floating-point), which minimizes round-off errors in intermediate calculations.

The solutions to these problems may correspond to data in different numbers of dimensions, depending on the number of equations in each system. Consequently, the mean squared error (MSE) serves as a metric for comparing the computed results with the baseline, adjusted to the dimension of the solution for each system. The MSE metric becomes particularly valuable in comparing the precision of different arithmetic formats, effectively quantifying the deviation between two computations by penalizing large errors and assigning less importance to small differences.

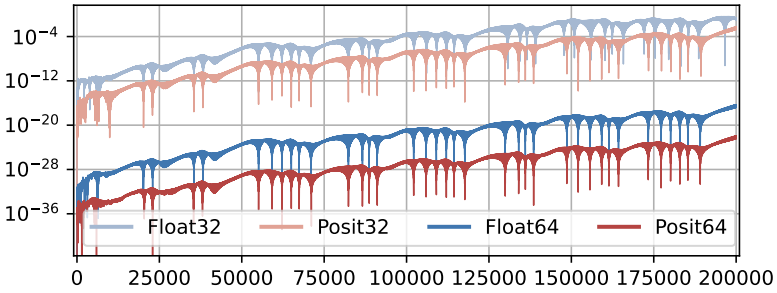
Considering that the systems of ODEs under examination in these experiments are dynamical systems with time-dependent solutions, the evaluation of errors is conducted across all time steps, and the calculation of the global average error is also performed.

The average MSE of dynamical systems computed with different arithmetic formats is presented in Table 3.3. Computations using lower bit lengths did not converge into valid solutions. In all the cases, posits exhibit lower errors compared to their corresponding floating-point formats. This distinction is particularly noteworthy in the case of 64-bit precision, where posits provide answers with multiple orders of magnitude greater accuracy.

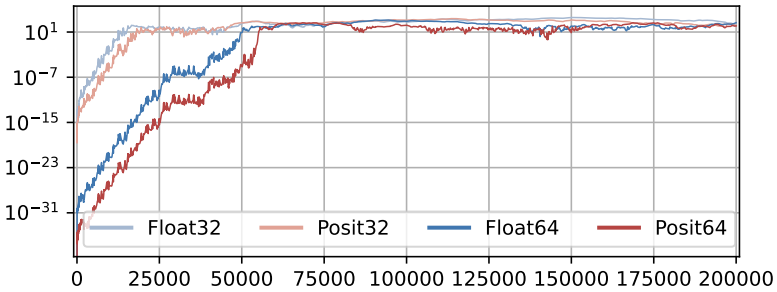
For a more comprehensive understanding of the results, the error per iteration is depicted in Figure 3.2. As illustrated, posit formats demonstrate significantly lower errors than their



(a) Lorenz system.



(b) Duffing oscillator.



(c) Double pendulum.

Figure 3.2: MSE per iteration for different ODEs.

floating-point counterparts during the initial steps. However, after several iterations, the inherent chaotic nature of these systems of ODEs leads to higher errors for both formats. This is especially notable in the double pendulum experiment, in which the positions of the pendulums at the end of each simulation are completely different across number formats.

3.4. Conclusions

Posit arithmetic claims to outshine the standard IEEE 754 floating-point numbers in terms of accuracy, a feature crucial in various problem-solving scenarios. This chapter has corroborated these claims through evaluations across multiple scientific applications. By scrutinizing error rates and decimal accuracy under different configurations of posits and

floats, the study covered classic issues from literature and simulations of time-dependent systems.

It is important to note that while posits generally outperform floating-point formats in accuracy, their superior performance is constrained to a specific range of precision, known as the golden zone. Beyond this range, computations involving values outside the golden zone may lead to increased errors in the case of posits. Thus, the programmer should be aware of this to perform data normalization, if required.

The subsequent chapter extends the assessment of posit format to a distinct domain—deep learning. Given the computational intensity of applications in this field, the study explores the potential of posits as a viable alternative to traditional floating-point representations.

4

Use of posit arithmetic in deep learning

Since its introduction in 2017, posit format has been the subject of multiple studies exploring its benefits over the standard floating-point format and its suitability in real applications. However, most of these studies have focused on deep learning, demonstrating the potential of this format over floats [73, 130, 143, 58, 101]. One major challenge that posits present is that there is currently no hardware support for such a format, leaving only two options: building custom hardware for posit arithmetic [12] or relying on software emulation. The latter option is often preferred for its faster deployment time, although it comes at the cost of reduced performance.

Based on such observations, a framework for entirely performing both inference and training of deep neural networks (DNNs) employing posit arithmetic through all the computations is proposed in this chapter. The framework, so-called Deep PeNSieve, relies on software emulation for faithfully performing computations under Posit$\langle 32, 2\rangle$, Posit$\langle 16, 1\rangle$ and Posit$\langle 8, 0\rangle$ formats. In addition, it includes support for performing post-quantization to Posit$\langle 8, 0\rangle$ as well as performing inference with fused MAC operations, leveraging the quire accumulator. Under this approach, we aim to accurately evaluate the effects of using such an arithmetic format in a variety of models and datasets.

The rest of the chapter is structured as follows. Section 4.1 undertakes a comprehensive review of the fundamentals of deep learning and common architectures in the area, as well as motivates the usage of posit arithmetic in this kind of applications. The proposed framework for the computation of posit-based DNNs is presented in Section 4.2. In Section 4.3, the usage of posit arithmetic is evaluated under different sets of models and datasets. This includes training with both 32 and 16-bit formats, as well as low-precision inference. The chapter culminates with Section 4.4, where the principal contributions and concluding remarks on this evaluation are detailed.

4.1. Background on deep learning

Deep learning is a subfield of machine learning that has gained significant attention in recent years due to its success in a wide range of applications, including computer vision,

natural language processing, and speech recognition [94, 156]. The main advantage of deep learning over traditional machine learning techniques is its ability to automatically extract and learn features from raw data, without the need for human domain expertise.

Deep learning has its roots in the development of artificial neural networks, which are computational models inspired by the structure and function of the human brain. Neural networks consist of interconnected nodes or neurons that are organized into layers, with each layer responsible for processing a different aspect of the input data. In recent years, the development of more powerful computing hardware, such as GPUs and TPUs, has enabled the training of much larger and more complex DNNs [74, 34]. This has led to several improvements in the state of the art in areas like image classification, object detection, natural language processing, image generation [86, 9, 142] and many other domains such as game playing, drug discovery and genomics [152, 75].

In this section, a brief overview of the basics of machine learning and neural networks will be provided, and then focus on the specific aspects of deep learning that are relevant to the research problem. The architecture of DNNs, the training process, common deep learning architectures, and some current challenges in the field are discussed as well. Overall, deep learning represents a significant advance in the field of machine learning, with the potential to revolutionize many industries and applications. Understanding the fundamentals of deep learning is crucial for researchers and practitioners in the field, and is essential for the development of new and innovative applications.

4.1.1. Neural networks and deep learning

Neural networks are a type of machine learning algorithm inspired by the structure and function of the human brain. The basic building block of a neural network is the artificial neuron (also known as perceptron), which takes a weighted sum of the inputs and passes the result through an activation (nonlinear) function to produce an output. Figure 4.1 shows a schematic representation of the mathematical model of an artificial neuron, i.e., the weighted sum of the inputs x_i multiplied by the weights w_i plus a bias b is passed by an activation function f to obtain the corresponding output y. Multiple artificial neurons can be combined to form a layer, and multiple layers can be stacked together to form a neural network. For that reason, the activation function of the neurons is a nonlinear differentiable function, such as ReLU, sigmoid or hyperbolic tangent.

As neural networks are typically composed of multiple layers of neurons, the output of each layer is propagated as input to the next layer, as shown in Figure 4.2. That is, every layer l takes as inputs ($x_i^{[l-1]}$) the outputs of the previous layer ($y_i^{[l-1]}$). This way, the information is processed hierarchically to learn increasingly complex representations of the input data.

Within the domain of neural networks, the term *deep learning* refers to the use of neural networks with multiple layers, i.e., more than one hidden layer. Such networks are often referred to as deep neural networks (DNNs). Today, the number of network layers used in deep learning can vary from five [95] to more than hundreds. The increased depth of DNNs allows them to learn high-level features that are more complex and abstract than those learned by shallower networks, enabling them to perform more sophisticated tasks.

One of the key features of DNNs is that the operations of individual neurons are primarily composed of weighted sums, as shown in Figure 4.2. As a result, when thousands of neurons are stacked in multiple layers to form a DNN, the bulk of the computation

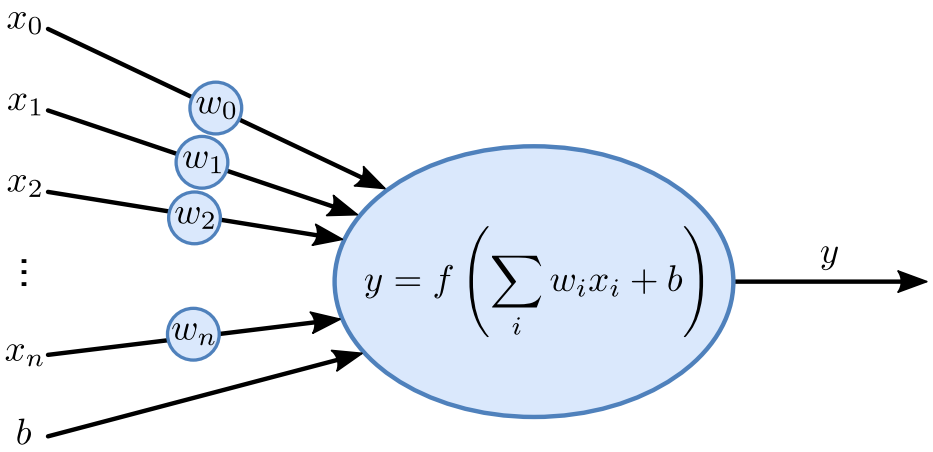


Figure 4.1: Schematic representation of the mathematical operation of an artificial neuron.

involved is matrix multiplication. To optimize for this computationally intensive process, various parallelization techniques, such as single instruction, multiple data (SIMD) or single instruction, multiple threads (SIMT), have been developed. These techniques can exploit the parallelism inherent in matrix multiplications to accelerate DNN computations. Additionally, hardware accelerators such as GPUs have become increasingly popular for DNN training and inference due to their ability to perform large-scale matrix multiplications in a highly parallelized manner. The use of GPUs and other specialized hardware has significantly reduced the time required to train and evaluate DNNs, enabling researchers and practitioners to tackle increasingly complex problems with larger datasets and deeper architectures.

4.1.2. Model training and execution

DNNs can be incredibly complex, with thousands or even millions of neurons, and it would be nearly impossible to calculate the weights for each neuron by hand. To address this issue, a breakthrough occurred in the mid-1980s when it was discovered that multi-layer architectures could be trained using the chain rule for derivatives, and a process called *backpropagation* was proposed [95]. Backpropagation is an algorithm that allows for the iterative calculation of the weights and biases in each layer during a process called *training* phase.

During training, the algorithm learns from a dataset, and after each forward pass, the backpropagation algorithm propagates errors backward through the network, adjusting the weights of each neuron based on the difference between the predicted output and the actual output. This process is repeated many times until the network reaches a satisfactory level of accuracy. Once the network is trained, it can compute output values using the weights determined during training. This phase is called *inference* or prediction. In this phase, only the forward pass of the inputs is performed, so the computation involved in inference is very similar to that of training. However, the backward computation of training requires higher precision and the preservation of intermediate outputs of the network [108, 107].

Overall, the backpropagation algorithm has proven to be a powerful tool in the training of DNNs. However, despite its effectiveness, DNNs may be difficult to train and frequently

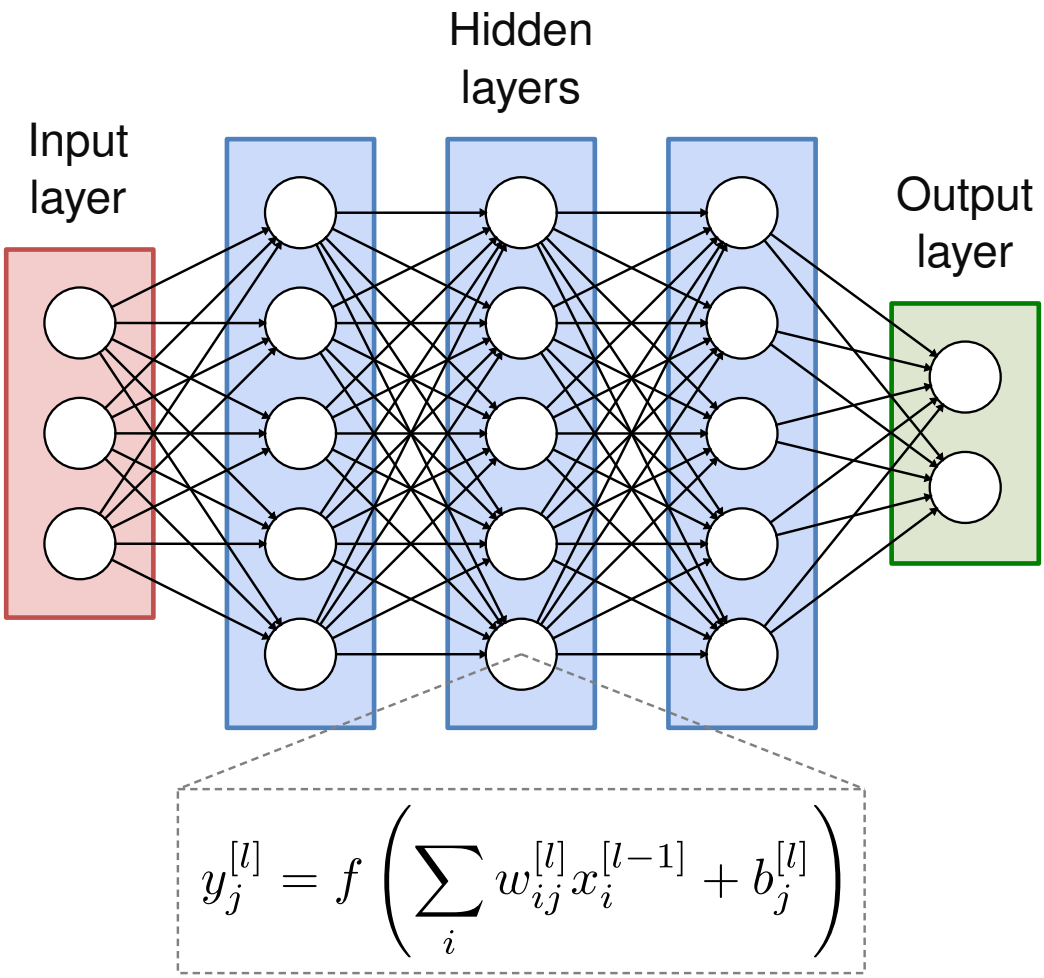


Figure 4.2: General architecture of a neural network.

need large amounts of data and computational resources to achieve good performance. Nevertheless, recent advances in optimization strategies, regularization procedures, and hardware acceleration have aided in tackling some of these obstacles and have allowed DNNs to continue pushing the boundaries of what is possible.

Finally, several deep learning libraries and resources, such as TensorFlow [1] and PyTorch [138], provide core utilities for both inference and training of DNNs, including many pre-trained models and other necessary functions for the building and implementation of such models. These tools enable researchers and practitioners to easily develop and experiment with deep learning models, even if they do not have extensive experience in programming or machine learning. The availability of pre-trained models and other resources has greatly accelerated the adoption of deep learning techniques in a wide range of fields and applications, from autonomous driving to medical imaging to natural language processing.

4.1.3. Common deep learning architectures

In deep learning, the term *neural network architecture* refers to the overall structure and organization of the layers of a network and their connections. An artificial neural network generally consists of several layers of neurons. Each layer processes information in a step-by-step manner, creating a hierarchy to understand more intricate representations of the input data. This section explores the most prevalent types of layers found in neural network architectures: fully connected layers, convolutional layers, and recurrent layers. For a detailed understanding of these and other layers, please refer to [3].

Fully connected layers

Also known as dense layers, represent the simplest layer type in a neural network. They consist of neurons that are entirely connected to the neurons in the preceding layer. Each neuron in a fully connected layer calculates a weighted sum of its inputs and applies a nonlinear activation function to generate the output, as depicted in Figure 4.2.

Typically employed in the concluding stages of a neural network, these layers are used in tasks such as classification or regression. Despite their versatility in modeling a broad array of functions, fully connected layers are characterized by a high number of parameters compared to convolutional and recurrent layers. This parameter abundance makes them more susceptible to overfitting.

Convolutional layers

Convolutional layers are widely employed in computer vision tasks. Their primary function is to extract spatial features from input data. A convolutional layer consists of a set of filters that slide over the input data, applying a convolution operation to each local patch. The result is a set of feature maps, where each map corresponds to a distinct filter, encoding a specific type of feature.

The utilization of convolutional layers serves to decrease the number of parameters in the network, facilitating the learning of spatially invariant representations. In comparison to fully connected layers, convolutional layers have a lower number of parameters, reducing the risk of overfitting. Additionally, they have the ability to capture local spatial correlations and can be stacked to progressively learn more complex features.

Recurrent layers

Recurrent layers are used in sequence modeling tasks such as language modeling, speech recognition, and machine translation. In a recurrent layer, a group of neurons handles a sequence of inputs and maintains a hidden state summarizing information from prior inputs. The output of a recurrent layer depends not only on the current input but also on the previous hidden state, allowing the network to capture temporal dependencies in the input sequence.

Unlike fully connected and convolutional layers, recurrent layers have a memory-like mechanism that allows them to process variable-length sequences of input data. However, recurrent layers have a high number of parameters, which makes them more computationally expensive and more susceptible to overfitting. A common type of recurrent layer is the long

short-term memory unit, or LSTM, widely employed in modern speech recognition and text-to-speech applications.

Other layers

In addition to these three main types of layers, there are also *pooling layers*, *activation layers*, *normalization layers*, and more that can be used in neural network architectures. Each type of layer has its own strengths and weaknesses, and the choice of layer architecture depends on the specific requirements of the task at hand.

For instance, pooling layers help in downsizing the spatial dimensions of the input, reducing computational complexity. Activation layers introduce non-linearities to the model, enabling it to learn complex patterns. Normalization layers contribute to stabilizing and accelerating the training process.

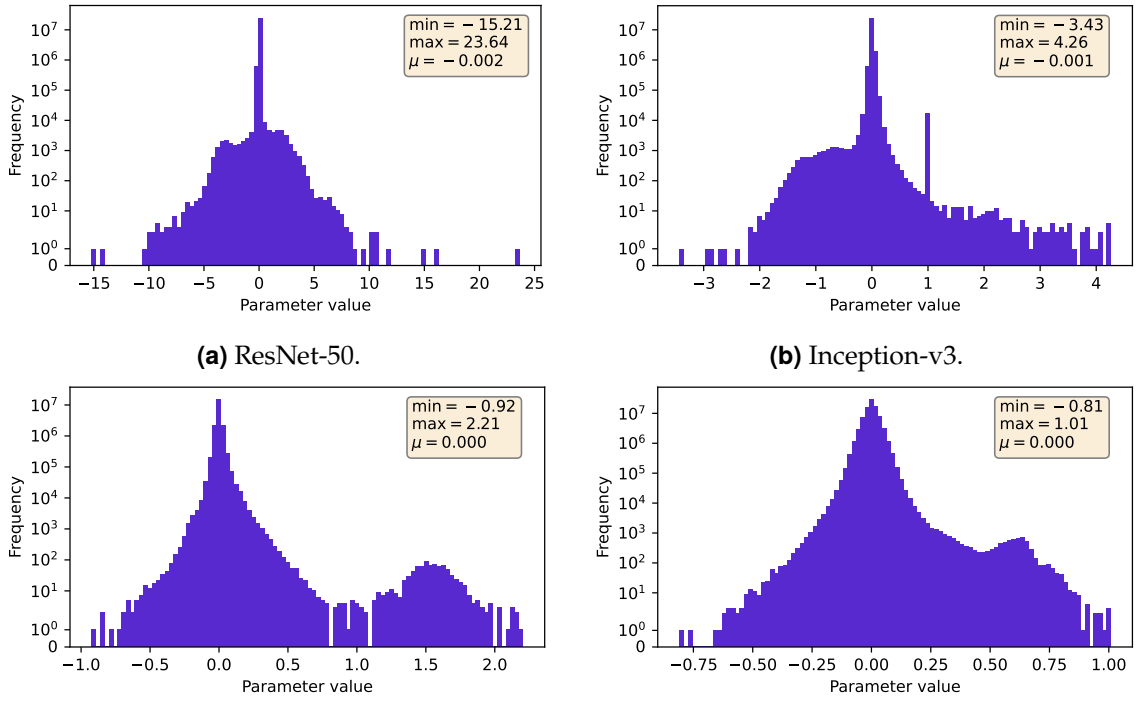
The decision on which layers to incorporate is guided by factors such as the nature of the data, the complexity of the task, and computational considerations. A well-designed architecture can lead to state-of-the-art performance on a wide range of tasks, while a poorly designed architecture can result in poor performance or even failure to converge. Therefore, it is important to carefully choose the appropriate type and number of layers and to tune the hyperparameters of the network for optimal performance.

4.1.4. Benefits of posit arithmetic in deep learning

Accuracy is crucial for DNN applications in both inference and training phases. Recall that, as shown in Figure 1.6, the posit format provides a higher precision than the standard floating-point format when representing and operating small magnitude values. Potential applications that would benefit from using posit are those whose data are within that range that posit offers higher precision with the same bit length. In particular, deep learning is one of those potential applications. As it can be seen from Figure 4.3, most of the parameters of the different vision models (up to 86 M in the case of the Transformer model) lie in the $[-1, 1]$ range. Just a few weighs are more than an order of magnitude larger. This analysis of the model parameter distribution suggests that posits might be a suitable format for the representation of weights and activations of DNNs.

Another advantageous aspect of posit arithmetic in deep learning is the lack of overflow or underflow in this representation. Training DNNs with lower data precision (such as Float16 or bfloat16) is a common technique known as *mixed precision training* [107] that can reduce the computational cost. However, storing small magnitude gradients in Float16 may lead to underflow. Typically, loss scaling and gradient scaling methods can prevent this issue by multiplying the loss with a scaling factor. However, utilizing these methods may lead to other problems such as modifying the training workflow or avoiding an overflow situation. Fortunately, since posits cannot underflow, this risk is eliminated.

Finally, it should be noted that certain posit formats allow for the approximation of common activation functions (such as the sigmoid or hyperbolic tangent) through simple bitwise manipulation [52, 24]. This approach can lead to higher efficiency and improved performance compared to computing exact and expensive operations required for such functions.



(a) ResNet-50.

(b) Inception-v3.

(c) DenseNet-121.

(d) Vision Transformer (ViT-B/16).

Figure 4.3: Histogram of the parameter distribution for different DNN models.

4.2. Emulating posit arithmetic in DNNs

Regarding the study and usage of posit arithmetic in DNNs, previous works mainly present two approaches, (1) the training stage is performed with floating-point numbers, while the inference stage is performed in low-precision posit format [92], or (2) performing rounding to posit format before and after each operation at both inference and training [101, 59]. These techniques allow leveraging existing floating-point hardware. However, none of the approaches provides a faithful emulation of the entire computation of DNNs with posits. The former just converts the learned widths to posit format, while the latter involves rounding conversion to posit format at each operation, and the kernel of each layer is always computed using FP. This limitation can lead to discrepancies in results when compared to performing all computations in posit format, especially in the case of DNNs, which rely heavily on general matrix-matrix multiplication (GEMM) operations and produce many intermediate partial results. Therefore, further research is needed to explore the potential benefits and limitations of these different approaches to performing arithmetic operations with posits.

In order to avoid such a conversion and show the whole potential of posits, a software framework named Deep PeNSieve [114] was developed in this thesis. This framework allows entirely performing both training and inference of DNNs employing posit arithmetic during the whole process. Trained models are saved preserving the format to perform inference with the same or even lower precision. In particular, it is capable of performing low-precision inference down to 8-bit posits and fused operations.

Deep PeNSieve is built on top of the well-known machine learning framework TensorFlow [1], which implements all the functionalities necessary to develop DNN models, such as different layers, forward and backward algorithms, optimizers, etc. The posit format is extended to this framework via software emulation with the library SoftPosit [97]. Such a library provides support for Posit⟨32, 2⟩, Posit⟨16, 1⟩ and Posit⟨8, 0⟩ datatypes, as well as for fused arithmetic operations on these formats. Therefore, Deep PeNSieve also implements the algorithms required for DNN inference using fused arithmetic. The whole framework is written in Python programming language, although some internal libraries are in C++. Next, a detailed description follows on the procedures for training and conducting low-precision inference.

To the best of our knowledge, this is the first and only software that allows performing both DNN inference and training entirely using posit arithmetic. However, such accuracy in the computations has certain limitations. Since posit arithmetic operations are emulated via software, the execution of the tool is restricted to the CPUs, and it cannot show compared performance with floating-point computations. Also, as the software relies on SoftPosit, data types are currently limited to Posit⟨32, 2⟩, Posit⟨16, 1⟩ and Posit⟨8, 0⟩.

4.2.1. DNN training with posits

The proposed framework's training flow is illustrated in Figure 4.4. The red boxes in the figure represent input data, including hyperparameters, while the blue boxes correspond to the functionalities performed by the TensorFlow framework, and the green box refers to the output of the model.

Firstly, it must be noted that all the inputs of the neural networks in the proposed framework must be in posit format. Therefore, the outputs of the networks will also be in the posit format. When creating a new DNN, all the hyperparameters are initialized using the posit format. Table 4.1 summarizes the different layers, activation functions and optimizers available for the construction of DNNs. The parameters of the optimizer used to train the model must also be converted to the posit format to ensure consistent computation. Once the network (including all internal parameters and weights) is generated with the selected posit configuration, it is trained as usual, with the single difference that all computations are performed with posit numbers. This allows us to evaluate the real performance of the posit format in this task. Finally, the trained parameters and models are saved to perform the inference stage with the same or lower precision.

Recall that Deep PeNSieve performs posit computations via software emulation, which causes an overhead compared to the default floating-point format (Float32). Therefore, comparing the speed between different numeric formats is beyond the scope of this work, as the lack of dedicated hardware is a major handicap for the posit format.

4.2.2. DNN inference with low-precision posits

In recent years, deep learning has shown remarkable success in solving many complex tasks in various fields. However, this success comes at a high computational cost, which demands efficient processing and storage of data. One of the ways to achieve this is by using low-precision data formats, which reduce memory usage and computational requirements.

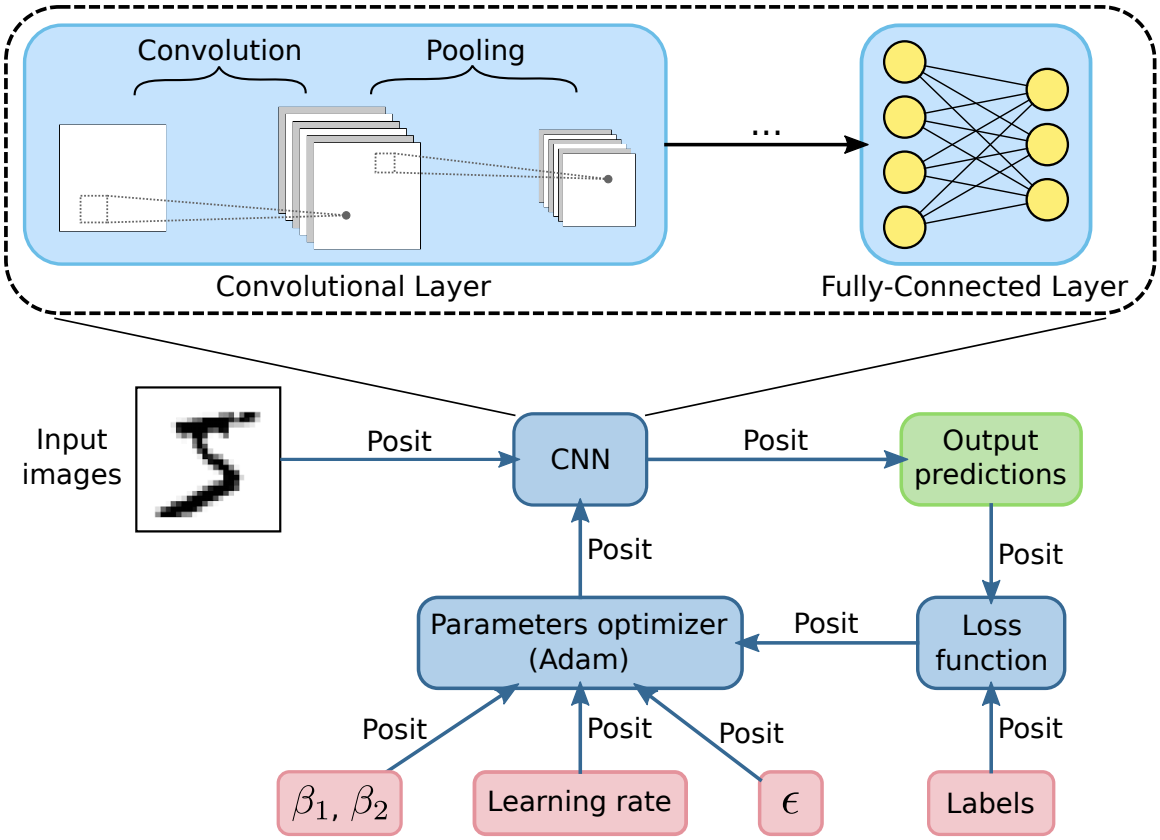


Figure 4.4: Scheme of the training flow for the proposed framework.

In this regard, several numerical formats have been proposed, such as half-precision floats and 8-bit integers.

Quantization is a widely used technique in deep learning that consists of reducing the size of DNNs by reducing the number of bits used to represent the weights and activations of the network without significantly affecting its accuracy [84, 42]. This can provide multiple benefits, such as reducing memory and power footprint, or faster computation. Common lower-precision numerical formats used in quantization include 8-bit integers (INT8), half-precision floating-point format (Float16), and bfloat16. When applying quantization at inference, the parameters in the DNN need to be adjusted. This can either be performed by retraining the model, a process that is called quantization aware training (QAT), or done without re-training, a process that is often referred to as post-training quantization (PTQ). While the former of these methods can provide higher accuracies, in many cases it is desirable to reduce the model size without requiring to re-train. PTQ performs the quantization of weights and activations for faster inference with very low overhead compared with QAT.

In this thesis, emphasis is placed on PTQ for DNN inference. More specifically, the implementation of post-training Posit$\langle 8, 0\rangle$ quantization is carried out within the Deep PeNSieve framework. The trained models can be converted to low-precision by maintaining the model architecture and quantizing the original weights and biases to Posit$\langle 8, 0\rangle$ format, thereby preserving the operations of the models. The proposed solution involves performing

Table 4.1: Supported features in Deep PeNSieve.

Operator Name	Type	Support
Fully connected	Layer	✓
Convolution	Layer	✓
Max pooling	Layer	✓
Dropout	Layer	✓
ReLU	Activation function	✓
Sigmoid	Activation function	Inference only
Tanh	Activation function	Inference only
Mean squared error (MSE)	Loss function	✓
Cross entropy	Loss function	✓
Stochastic gradient descent (SGD)	Optimizer	✓
Momentum	Optimizer	✓
Adam	Optimizer	✓

a linear quantization of model parameters to a lower posit precision format, as shown in Equation (4.1),

$$\tilde{x} = round_{P_{n,es}}(x), \tag{4.1}$$

where x is the original value, and $round_{P_{n,es}}(\cdot)$ performs rounding according to selected Posit$\langle n, es\rangle$ precision. In particular, the proposed framework relies on the $round_{P_{8,0}}(\cdot)$ operation. Note that, since overflow and underflow are not allowed in posit arithmetic, such rounding is saturated. It is worth noting that, in this case, it is not necessary to quantize activation functions since only the precision is reduced, while the arithmetic is maintained. This is an advantage compared to classical quantization processes, which usually require fine-tuning the pre-trained models. For example, when quantizing floating-point models to 8-bit integer/fixed-point precision, the dynamic range of unbounded activations, such as ReLU, needs to be calculated using calibration data [71].

4.2.3. Fused arithmetic for low-precision DNN inference

Using low-precision formats in quantized neural networks might result in notable accuracy degradation. Although the current trend is to use INT8, arithmetic operations in these networks often require internal higher precision to maintain accuracy, as INT8 cannot hold the result of certain operations, such as multiplication and addition. For example, NVIDIA Ampere GPU architecture uses Float32 accumulators for Float16 FMA operations, as well as INT32 accumulators for INT8 operations [134]. To address this issue, we propose using the quire register—an accumulator included in the posit standard [51]—for the fused dot product operations within the GEMM operations present in most of the layers of neural networks. Specifically, convolutional and fully connected layers can be computed using this method in 8-bit posit neural networks.

To implement this, the internal structure of these layers must be modified to perform the GEMM operation with the fused dot product. While TensorFlow does not currently support

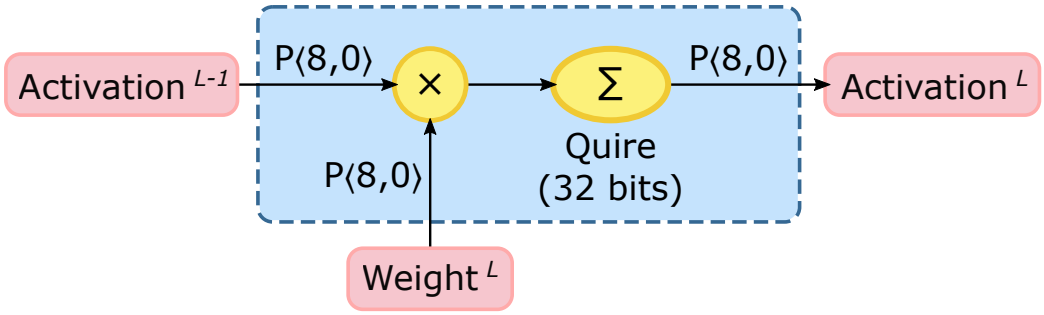


Figure 4.5: DNN forward data flow for low-precision GEMM operation with fused posit arithmetic.

Algorithm 1 Fused GEMM.

procedure GEMM(W, X)
  $rows \leftarrow W.rows$
  $inner \leftarrow W.cols$ ▷ *Same as* $X.rows$
  $cols \leftarrow X.cols$
  $Y \leftarrow posit[rows][cols]$
  for $j = 1, \ldots, rows$ **do**
    for $k = 1, \ldots, cols$ **do**
      $q \leftarrow quire(0)$ ▷ *quire initialization*
      for $i = 1, \ldots, inner$ **do**
        $q \leftarrow q + W[j][i] \times X[i][k]$ ▷ *result accumulated with no rounding*
      $Y[j][k] \leftarrow q$ ▷ *quire to posit rounding*
  return Y

fused posit operations, the SoftPosit library provides functionality for such operations using a quire accumulator [97]. Therefore, the network architecture is re-implemented within this library accordingly for PTQ, and using Posit$\langle 8, 0\rangle$ with fused operations. For such a posit format, the recommended size of the quire is 32 bits [123]. Figure 4.5 shows the implementation of the GEMM operation using the quire during the forward stage, and Algorithm 1 describes this kernel modification using a single quire accumulator. The intermediate results are stored in the quire, so only one rounding is performed at the end of each computation.

As explained above, the operation of fully connected layers essentially consists of performing GEMMs. This also occurs in convolutional layers, albeit less obviously. The current approach to implement convolutional layers is to expand the image into a column matrix (im2col operation) and the convolution kernels into a row matrix. This allows convolutions to be performed as matrix multiplications. Such an approach is exploited to benefit from the quire-based implementation of GEMM in convolutional layers, as well.

4.3. Experimental evaluation

To evaluate the performance of different numerical formats, it is necessary to compare the training and inference results for various DNN architectures and datasets. Typically, deep learning frameworks employ IEEE 754 single-precision floating-point format for training and the same or lower-precision data formats for inference. Recent works suggested using 32 or even 16-bit posit configurations as an alternative to 32-bit floats [101, 114]. These 16-bit posits could replace 32-bit floats, resulting in faster and more efficient processing.

In this study, the performance of different CNNs trained with the Posit$\langle 32, 2\rangle$ and Posit$\langle 16, 1\rangle$ configurations are compared with the baseline Float 32 format, which is the default precision for training DNNs in many deep learning frameworks. Regarding inference, half-precision floats or 8-bit integers can also be used, resulting in reduced memory usage and computational requirements. In this context, such formats are compared with the Posit$\langle 8, 0\rangle$. TensorFlow is used for baseline results, while Deep PeNSieve is utilized to manage the training and inference processes with posits. The following experiments aim to determine the optimal numerical format for different CNN architectures and datasets, with the goal of achieving high accuracy while minimizing computational cost.

4.3.1. Experimental setup

To assess the effectiveness of posits for deep learning tasks, experiments were conducted utilizing multiple datasets and CNN architectures for image classification. The datasets used for training and testing of different DNNs include:

- **MNIST**: This is a dataset of 70 000 28 × 28 grayscale images of handwritten digits from 0 to 9. There are 60 000 training images and 10 000 test images.
- **Fashion-MNIST**: This is a dataset of 70 000 28 × 28 grayscale images of fashion products such as clothing, shoes, and bags (10 different classes). There are 60 000 training images and 10 000 test images.
- **SVHN**: Street View House Numbers dataset consists of 32 × 32 color images of house numbers captured from Google Street View images. It contains 73 257 images for training, 26 032 images for testing. Each image contains a single house number in the range of 0 to 9.
- **CIFAR-10**: This is a dataset of 60 000 32 × 32 color images in 10 classes, with 6000 images per class. There are 50 000 training images and 10 000 test images.

The selection of these datasets was based on their representation of a diverse array of image classification tasks, encompassing varying levels of complexity.

In addition to the datasets, the evaluation of posits included the assessment of performance with the following CNN architectures:

- **LeNet-5**: A classic CNN architecture that was proposed by Yann LeCun et al. in 1998 [95]. It consists of seven layers, including two convolutional layers, two pooling layers, and three fully connected layers. The architecture was originally designed to recognize handwritten digits, but since then it has been used for various image classification tasks. This model is used to classify MNIST and Fashion-MNIST datasets.

- **CifarNet**: A network designed to solve the CIFAR-10 classification problem. It was proposed by Alex Krizhevsky [85], the same researcher who designed the well-known AlexNet architecture [86]. CifarNet consists of nine layers, including convolutional, pooling, and fully connected layers. The size of the pooling layers is slightly modified with respect to the original design. The first layer is a convolutional layer with 64 filters of size 5 × 5. This is followed by a max-pooling layer with a pool size of 3 × 3 and a stride of 2. Next, another convolutional layer followed by another max-pooling layer with the same parameters is appended. The output of the latter max-pooling layer is then flattened and passed to a fully connected layer with 384 neurons, followed by another fully connected layer with 192 neurons. Finally, the output layer consists of 10 neurons for CIFAR-10 (or 100 neurons for CIFAR-100). This model is used to classify SVHN and CIFAR-10 datasets.

4.3.2. Training evaluation

To determine the trade-offs between accuracy and computational cost for training CNNs, the aforementioned architectures are implemented with three different numerical formats: The first format is the default precision for DNN training, which is Float32. The other formats are Posit⟨32, 2⟩ and Posit⟨16, 1⟩. Experimental results showed Posit⟨8, 0⟩ format does not have sufficient precision and dynamic range for this task. All models are trained with a batch size of 128 images and using the Adam optimizer algorithm for 30 epochs. It was observed that training the models for more epochs did not result in better performance.

To eliminate any variability introduced by random initialization of the models, we generate the initial weights of each model using the same initial random seed. This ensures that all models start with the same initial weights and reduces the chances of differences in performance being attributed to differences in random initialization. Input data are normalized into the range $[-1, 1]$. No other regularization techniques, such as normalization and dropout, are used in the training process.

Table 4.2 displays a comparison of inference results for different CNN models trained with either standard Float32 or posits. Multiple training executions were performed with variations in the random seeds, although similar results were obtained. The Top-k metric computes the number of times the correct label is among the top k labels predicted. The results show that models trained with posit format exhibit similar or even better performance to the baseline Float32 format. Interestingly, models employing Posit⟨16, 1⟩ show higher accuracy than those trained under 32-bit formats, particularly for the complex CIFAR-10 dataset. Actually, the Top-1 accuracy achieved with Posit⟨16, 1⟩ is more than 4% higher than that obtained with Float32. While this is a significant improvement, it does not necessarily imply that 16-bit posits outperform floats. The results could be attributed to the specific network samples trained in this experiment. Alternatively, it may be due to the sensitivity of 32-bit formats to small gradient variations, leading to them falling into local minima and being outperformed by 16-bit networks. For the sake of comprehensiveness, deeper models and regularization techniques should also be implemented in the framework, although this is beyond the scope of this work.

For a more comprehensive grasp of the accuracy results and the training process, Figures 4.6 to 4.9 are presented. These figures depict the training process over different epochs for MNIST, Fashion-MNIST, SVHN, and CIFAR-10 datasets, respectively. The CNNs

Table 4.2: Accuracy results for DNNs trained with different data formats.

	MNIST		Fashion-MNIST		SVHN		CIFAR-10	
Format	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Float32	99.17%	100%	89.34%	99.78%	89.32%	98.35%	68.06%	95.15%
Posit⟨32, 2⟩	99.09%	99.98%	89.90%	99.84%	89.51%	98.36%	69.32%	96.59%
Posit⟨16, 1⟩	99.18%	100%	90.17%	99.81%	90.90%	98.72%	72.51%	97.40%

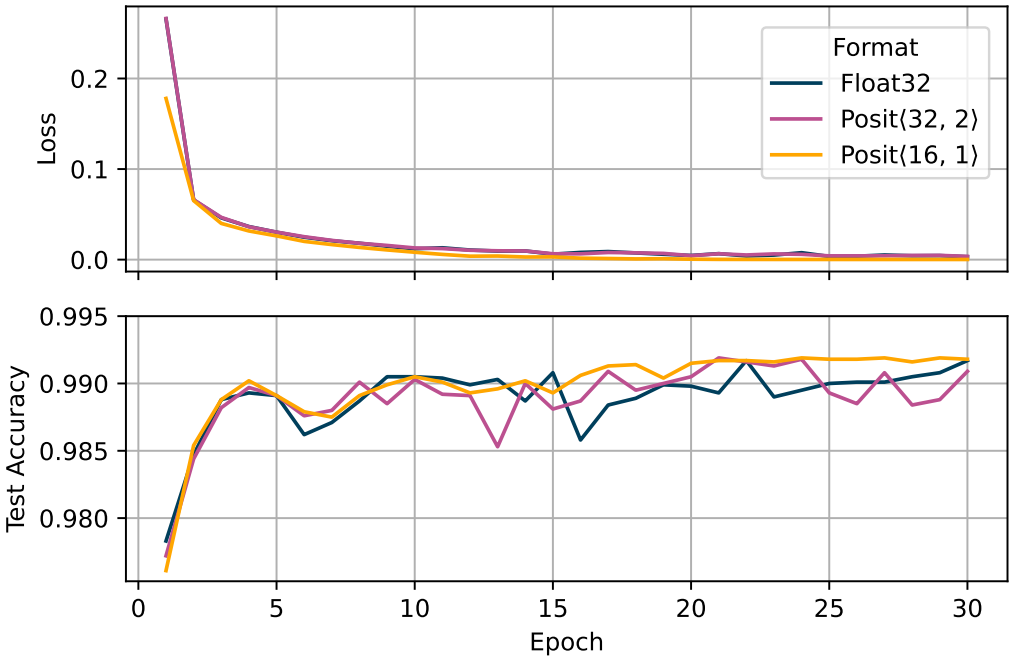


Figure 4.6: LeNet-5 training on MNIST with different number formats.

implemented using posits converge similarly to the floating-point ones. In particular, 32-bit floating-point and posit models present pretty much the same accuracy and nearly indistinguishable loss values. In addition, the training plots show how networks trained on Posit⟨16, 1⟩ exhibit lower error throughout training than the other formats, resulting in higher accuracy, as mentioned earlier.

It is worth noting that, although there are currently some hardware implementations that enable the execution of operations in the posit format [72, 113, 123], there is currently no software support in the form of libraries and compilers that allow the execution of high-level code, such as in the case of DNNs, on said hardware. Indeed, the proposed framework performs all posit computations via software emulation. This has a significant impact on execution times, as shown in Table 4.3. Experiments were run on a computer with an Intel Core i7-9700K processor running at 3.60 GHz using 32 MB of RAM, running Ubuntu 18.04. Similar results were obtained for either 32 and 16-bit posits. The lack of hardware support in this scenario means that posit arithmetic is not competitive with floating-point arithmetic in terms of performance and speedup. However, as hardware-software support for posits

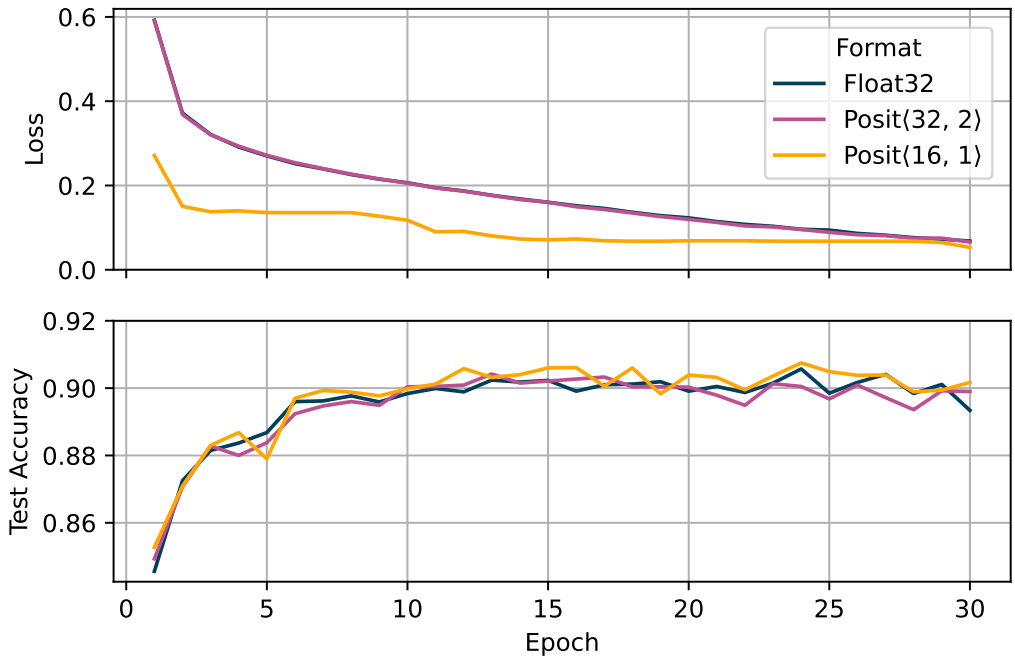


Figure 4.7: LeNet-5 training on Fashion-MNIST with different number formats.

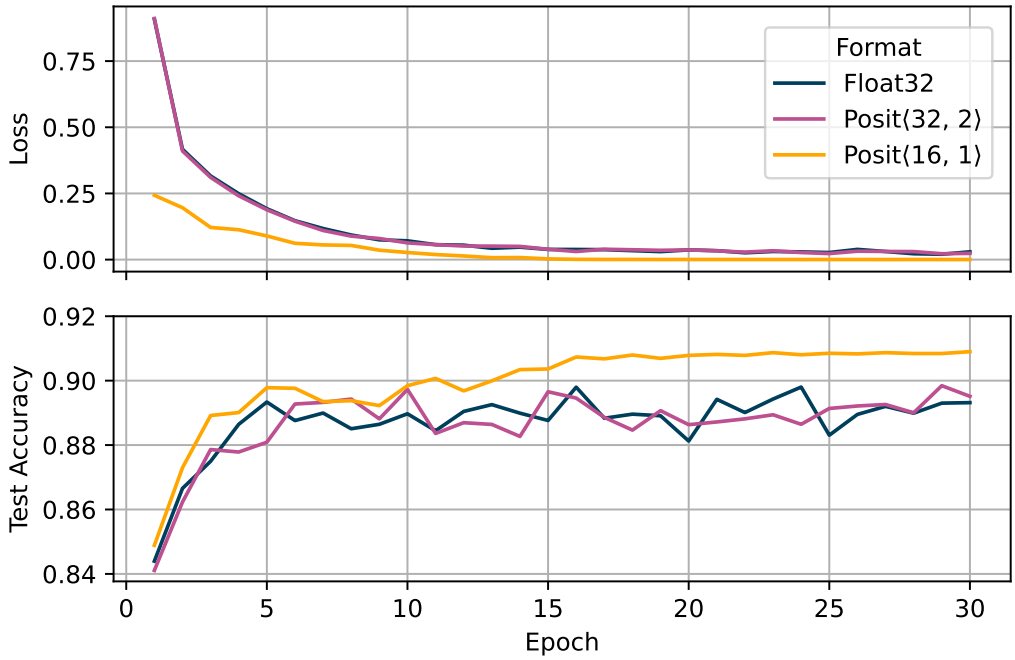


Figure 4.8: CifarNet training on SVHN with different number formats.

becomes available, it may become a more viable alternative to floats for training deep learning models.

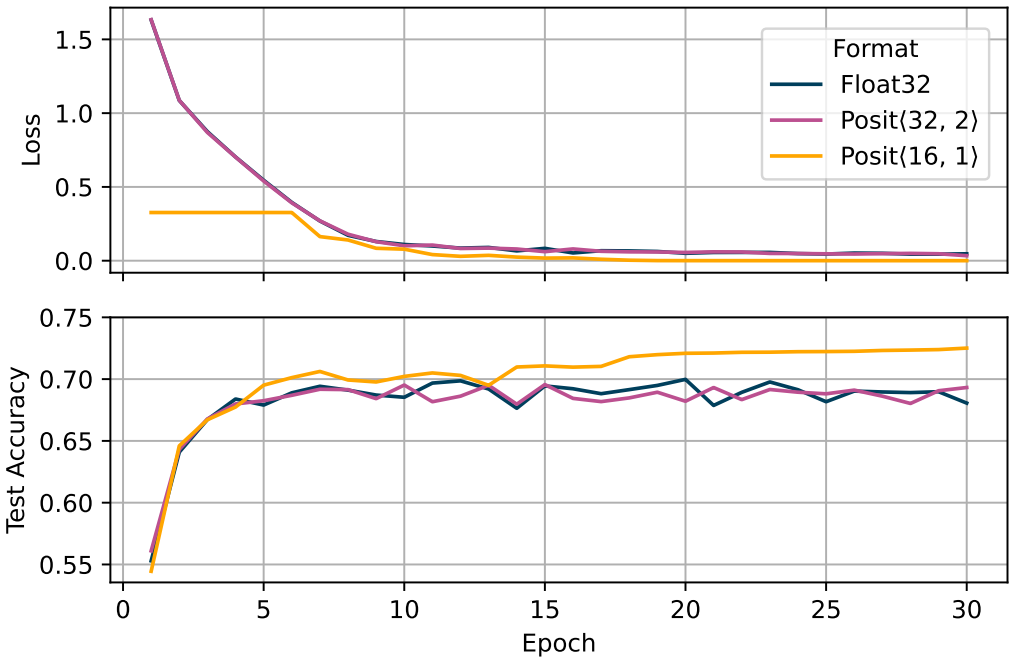


Figure 4.9: CifarNet training on CIFAR-10 with different number formats.

Table 4.3: Training time per epoch for each model and numeric format.

Format	MNIST	Fashion-MNIST	SVHN	CIFAR-10
Float32	00:00:09	00:00:09	00:01:58	00:01:17
Posit	00:05:35	00:05:35	06:13:28	03:47:26

In addition to achieving comparable accuracy to the Float32 format, it is important to note that low-precision formats such as Posit⟨16, 1⟩ can significantly reduce the size of the models. For instance, while 32-bit models require 724 kB and 21 MB when stored on disk for LeNet-5 and CifarNet, respectively, models trained on Posit⟨16, 1⟩ format require just 362 kB and 10.5 MB, respectively. This reduction in model size can enable training larger models or with larger mini-batches, thereby improving the scalability of deep learning algorithms. Furthermore, the use of 16-bit posits can significantly reduce the memory bandwidth of neural networks by 2× when corresponding hardware support is available. This reduction in memory usage could also be accompanied by an improvement in math throughput by using functional units with fewer bits, leading to faster training and lower power consumption, making posits an attractive option for resource-constrained environments such as mobile devices or embedded systems.

While the usage of 16-bit precision for training DNNs is not a novel technique, previous approaches in this regard require keeping gradients in higher precision, as well as performing scaling techniques [108]. However, in the proposed approach, the training flow remains unchanged, except for the replacement of the arithmetic format from 32-bit to 16-bit posits in all parameters and operations. While this improvement is significant, it is important to note

Table 4.4: Accuracy results with post-training quantization.

	MNIST		Fashion-MNIST		SVHN		CIFAR-10	
Format	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Float32	99.17%	100%	89.34%	99.78%	89.32%	98.35%	68.06%	95.15%
Float16	99.17%	100%	89.34%	99.78%	89.32%	98.35%	68.05%	96.15%
INT8	99.16%	100%	89.51%	99.79%	89.33%	98.38%	68.15%	96.14%
Posit$\langle 32, 2\rangle$	99.09%	99.98%	89.90%	99.84%	89.51%	98.36%	69.32%	96.59%
Posit$\langle 8, 0\rangle$	98.77%	99.99%	88.52%	99.82%	81.31%	97.07%	43.89%	86.49%
Posit$\langle 8, 0\rangle_{\text{fused}}$	99.07%	99.99%	89.92%	99.81%	89.13%	98.39%	68.88%	96.47%

that it does not necessarily mean that posits outperform floats in all scenarios. To gain a more complete understanding of the performance of different numerical formats, it is important to consider other factors that can impact the accuracy of the models. Regularization techniques, such as batch normalization or dropout, may be particularly relevant in this regard. However, incorporating these techniques into the framework might require big efforts, so this is left as future work.

Overall, the conducted experiments show that posits, particularly Posit$\langle 16, 1\rangle$, can achieve high accuracy in CNN models while reducing memory usage and computational requirements. However, further research is needed to fully understand the advantages and limitations of posits compared to other numerical formats in the context of deep learning.

4.3.3. Quantization evaluation

In order to perform low-precision inference, the previously trained models are kept unchanged. Thus, common neural network quantization methods are performed for both Float32 and Posit$\langle 32, 2\rangle$ models. For Float32 models, Float16 quantization and integer quantization techniques are employed to obtain models that entirely work in Float16 and INT8 formats, respectively. For Posit$\langle 32, 2\rangle$ models, the models are quantized to Posit$\langle 8, 0\rangle$ format as described in Section 4.2.2.

In addition, inference results are compared between naive posit quantization and employing the fused dot product approach (with Posit$\langle 8, 0\rangle_{\text{fused}}$), which requires modifying convolutional and fully connected layers of the models as described in Section 4.2.3.

Table 4.4 displays the inference results for the different PTQ techniques. The full precision models are included for comparison. It is worth noting that the quantized 16-bit and 8-bit models are 1/2 and 1/4 the size of the corresponding original 32-bit models, respectively. Results confirm the potential of conventional quantization techniques. However, it is well-known that for larger networks such as MobileNets and BERT, integer quantization provides a noticeable loss of accuracy [170].

Regarding the case of posits, the conducted experiments reveal a notable difference in performance when using quire and fused operations. As can be observed, networks using fused dot products with quire achieved much higher accuracy (25% higher top-1 for CIFAR-10) than those that did not use it. This difference becomes more noticeable with more complex networks. In the worst case, for the CIFAR-10 dataset, the accuracy degradation of

Posit$\langle 8, 0\rangle_{\text{fused}}$ was only 0.44% compared to the original 32-bit model, which is considered more than acceptable. The higher accuracy of posit-based models on Fashion-MNIST and CIFAR-10 datasets might be attributed to the fact that the initial models trained with Posit$\langle 32, 2\rangle$ presented better results than the corresponding Float32 models.

Overall, these experiments demonstrate that the presented low-precision posit approach provides similar results to the widely used PTQ techniques provided by the TensorFlow framework. These findings suggest that using posits-based low-precision inference may provide a viable alternative to conventional quantization techniques for reducing the computational cost of deep learning models.

4.4. Conclusions

In this chapter, the use of posit arithmetic in deep learning and its potential as an alternative to FP was explored. Due to the lack of a full hardware-software stack that allows the execution of high-level code (such as DNNs) in posit format, a software framework named Deep PeNSieve was developed. This framework offers posit functionality for the most common layers and functions involved in the generation of DNN models. In contrast with previous works, the presented approach consists of faithfully emulating the posit format via software at every single operation involved in the process of inference and training of neural networks, so that the effects of using such an arithmetic format can accurately be evaluated. The framework Deep PeNSieve has been publicly released under an open license.

The proposed approach has been tested on the scenario of training DNN models for image classification. Several CNN architectures were implemented using three different numerical formats: Float32, Posit$\langle 32, 2\rangle$, and Posit$\langle 16, 1\rangle$. The results reveal that the Posit$\langle 8, 0\rangle$ format does not have sufficient precision and dynamic range for training such models.

The experimental findings indicate that models trained with posit arithmetic exhibited similar or even better performance to those trained with the baseline Float32 format. Interestingly, it was found that models employing Posit$\langle 16, 1\rangle$ showed higher accuracy than those trained under 32-bit formats without the need to modify the training process. The training plots reveal that networks trained on Posit$\langle 16, 1\rangle$ exhibited a lower error throughout training than the other formats, resulting in higher accuracy. It is important to note that this improvement does not necessarily indicate that 16-bit posits are superior to floats in these tasks, and further exploration should be considered to ensure a comprehensive understanding in future work.

To further evaluate the potential of posits, low-precision inference was performed by comparing common neural network quantization methods for both Float32 and Posit$\langle 32, 2\rangle$ models. Float16 and INT8 quantization techniques were employed for Float32 models, and quantized Posit$\langle 32, 2\rangle$ models to Posit$\langle 8, 0\rangle$ format. The experiments revealed that, while using standalone Posit$\langle 8, 0\rangle$ numbers is not enough for performing inference, using Posit$\langle 8, 0\rangle$ with quire for computing the matrix multiplications at inference provides negligible accuracy reduction. This suggests that this approach may be a viable alternative to conventional quantization techniques for reducing the computational cost of deep learning models.

Overall, the experimental results indicate that posits may represent a promising alternative to floats for both training and inference of deep learning models. Since they can

provide similar results with a smaller bit length, posits might be particularly useful in energy and/or memory-constrained environments. However, it is important to acknowledge that the lack of native hardware, compilation, and software support for posits makes it difficult to corroborate this hypothesis with more complex models. The development of dedicated hardware entails substantial costs, and therefore, adopting a software-emulated approach, as proposed in this chapter, serves as a preliminary and convenient step in pursuing this line of research. The forthcoming section will delve into the development and implementation of posit functional units. This represents a significant step towards the development of posits, making this format a more practical and efficient solution for deep learning tasks.

Part II

Hardware Design for Posit Arithmetic

5

Functional units design

In Part I of this thesis (Chapters 3 and 4), the potential benefits of employing posit arithmetic in various applications were demonstrated. However, due to the absence of an existing hardware platform capable of executing all required posit operations, the previous experiments relied on software emulation. While this approach served as an initial method for assessing and exploring the advantages and suitability of the posit arithmetic format, emulating all the arithmetic operations inherently possesses high limitations. Consequently, it is imperative to design arithmetic units in posit format to enable the native execution of programs based on this format, thereby enhancing efficiency.

This chapter introduces functional unit (FU) designs for basic operations in posit format, including addition/subtraction, multiplication, division, and square root. The proposed posit units are compared to both state-of-the-art designs and the floating-point standard units.

At the hardware level, posits were intentionally designed to be "hardware friendly", i.e., to have similar (and even simpler) circuitry compared to existing floating-point units. Notably, comparison operations exemplify this concept. As mentioned in Section 1.2, the comparison of posit numbers can be performed similarly to signed integers, allowing for computation in an arithmetic logic unit (ALU) without requiring dedicated hardware. The primary encoding distinction between float and posit formats lies in the regime, a runtime-varying scaling component. As elaborated in this chapter, addressing this regime is the principal design challenge for posit units.

The common computation flow for posit numbers, which is illustrated in Figure 5.1, closely resembles that of floats. Operand fields must be decoded before operating, and the result must undergo rounding and encoding into the same format. An essential aspect of this process is decoding the regime, exponent and fraction fields of the operands. Unlike classical floating-point formats, the variable length of the regime impedes parallel data decoding. Specifically, the fraction and exponent cannot be extracted until the regime's length is known.

Lastly, it is worth noting that some works have found interesting properties for certain exponent formats, such as $es = 0$ [52, 24, 127]. However, the designs presented in this chapter

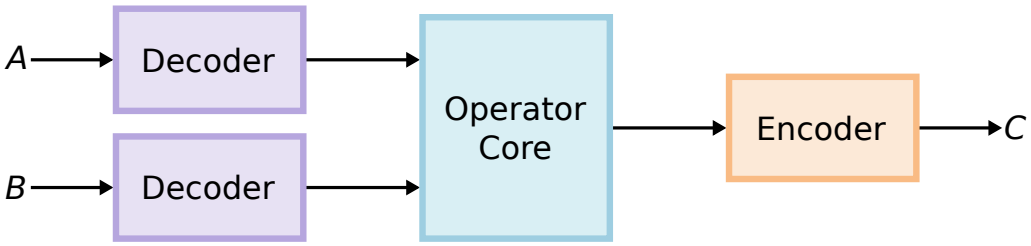


Figure 5.1: Generic computation flow for posits.

are described in a generic way, for any total bit length (n) and exponent size (es). This approach allows for a more accurate assessment of the implementation cost of this format in comparison to the floating-point standard.

The chapter is structured around the design of the different posit units. Section 5.1 presents the module for decoding posit numbers into their different fields, while the encoder module is elaborated in Section 5.2. Next, the core of each arithmetic unit is detailed. Section 5.3 presents de design for addition and subtraction operators, Section 5.4 focuses on the multiplication unit, division is presented in Section 5.5, and Section 5.6 accomplishes the same for the square root operation. Section 5.7 presents a detailed evaluation of the implementation of each of the posit units, comparing them to those proposed in previous works and to floating-point units. Finally, Section 5.8 concludes this chapter.

5.1. Posit decoding

When implementing posit operators in hardware, it is usually necessary to extract the four fields presented in Section 1.2 (s, r, e and f, plus a flag for zero/NaR exceptions) from a compact posit number before starting the real computation, as well as packing again the resulting fields after that.

Due to the variable-length regime of the posit format, the operand fields decoding flow slightly differs from the analogous stage for IEEE 754 numbers. More specifically, the value of the regime, which is encoded by the length of this field, must be unpacked as an integer value to perform arithmetic operations. After that, the extraction of the exponent and fraction is immediate. The component that performs such processes is usually known as the decoder. Thus, the importance of a good posit decoder design is crucial for the overall arithmetic design, since the core operation can not start until the sign, regime, exponent and fraction of the inputs are extracted.

The architecture of the proposed posit decoder is described in Figure 5.2. The most significant bit (MSB) contains the sign of the posit. Next, the regime length is detected using a Leading Zero or One Counter (LZOC). The regime value is then adjusted according to its length and the content of the regime bits. Finally, the regime is shifted out (adding 0's to the right), so the location of the exponent and fraction is known. The main difference between the proposed posit decoder and previous implementations is the use of a LZOC with integrated shifter instead of using separate detectors for decoding both positive and negative regimes [72, 173, 174] or using a single detector but always modifying the operand to deal with the same regime sign [15, 141, 72, 128] and then shifting out the regime. This

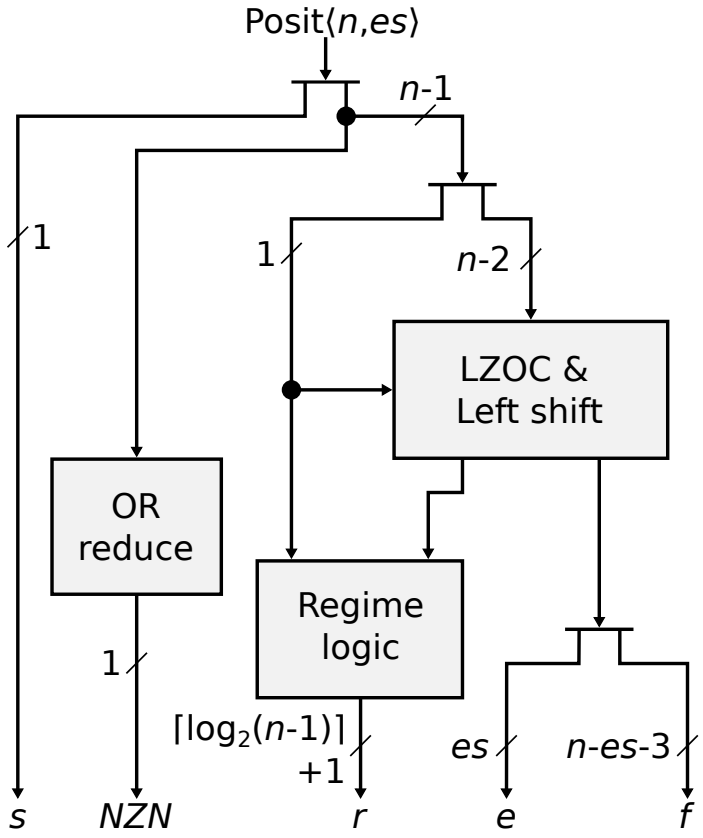


Figure 5.2: Block diagram of the proposed posit decoder.

approach simplifies the datapath substantially. Finally, the two posit arithmetic exceptions can be detected simultaneously by checking if all the bits except the MSB are equal to 0, so a flag bit indicates whether the posit is Neither Zero nor NaR (NZN).

When the posit number is negative, it is necessary to make a final adjustment to accommodate the regime and exponent to the value given by Equation (1.5). This can be done by just flipping the corresponding bits (see Appendix A). Also, notice that the implicit bit of the fraction is not appended, but it can be generated when necessary from the sign bit.

5.2. Posit encoding and rounding

At the end of every operation, the resulting regime, exponent and fraction must be packed into the corresponding posit format. When the exact result of a posit operation needs more digits than there are available in the fraction, rounding is required. These two steps are usually performed simultaneously in the so-called encoder module. The process of encoding a posit result from its different fields is depicted in Figure 5.3, and it mainly consists in performing the decoding process backward, plus handling possible rounding and overflow/underflow situations.

Firstly, the regime value (which has an extra bit for handling the possible overflow) is used to perform a logical right shift of the exponent and fraction fields. Posits, same as

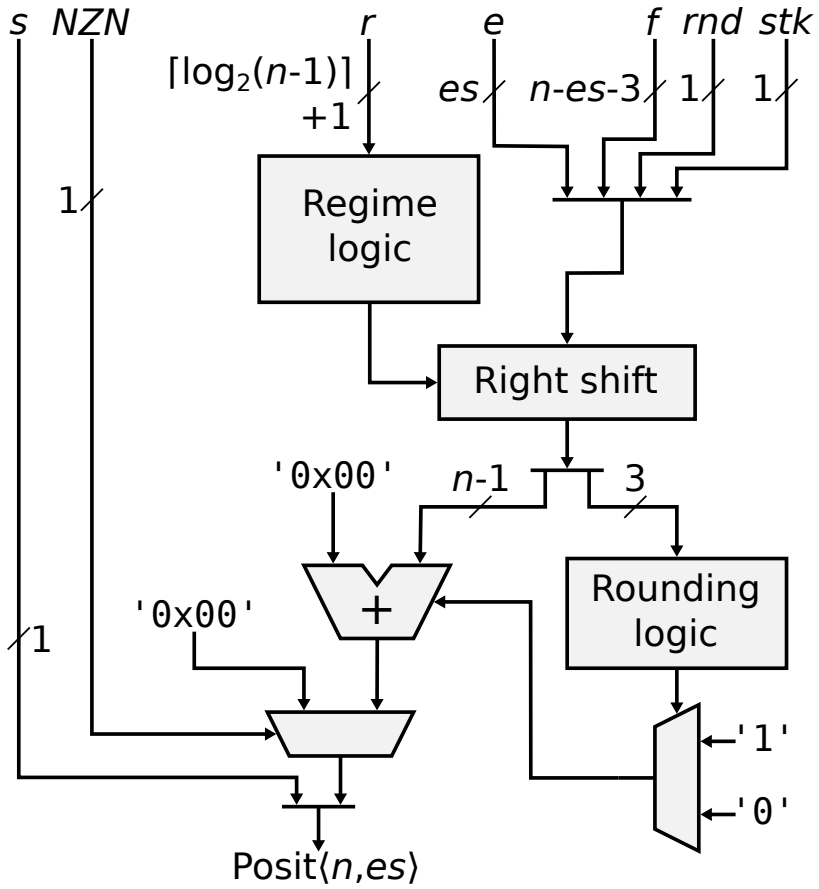


Figure 5.3: Block diagram of the proposed posit encoder.

IEEE 754 floats, follow round-to-nearest-even scheme. Therefore, two extra bits, namely round (rnd) and sticky (stk), must be preserved to perform a correct unbiased rounding. Finally, the result is obtained after considering the final sign bit and special cases.

5.3. Posit addition and subtraction

In posit arithmetic, as well as in the case of floating-point arithmetic, when performing the addition (or subtraction) of two numbers, it is necessary to adjust both operands to the same exponent, as depicted by Equation (5.1):

$$A + B = \begin{cases} \left(((1-3s_A)+f_A)+((1-3s_B)+f_B)\times 2^{E_A-E_B}\right)\times 2^{E_A} & \text{if } E_A > E_B, \\ \left(((1-3s_A)+f_A)+((1-3s_B)+f_B)\times 2^{E_B-E_A}\right)\times 2^{E_B} & \text{if } E_A \leqslant E_B, \end{cases} \tag{5.1}$$

where $E_i = (1-2s_i)\times(2^{es}r_i + e_i + s_i)$.

When translating this to hardware, it can be achieved by shifting one of the fractions, so both exponents are equal. If the first exponent is smaller than the second, the first fraction is shifted to the right by several bits given by the absolute difference of the exponents. Otherwise, the same is done to the second fraction. Then, the aligned significands are

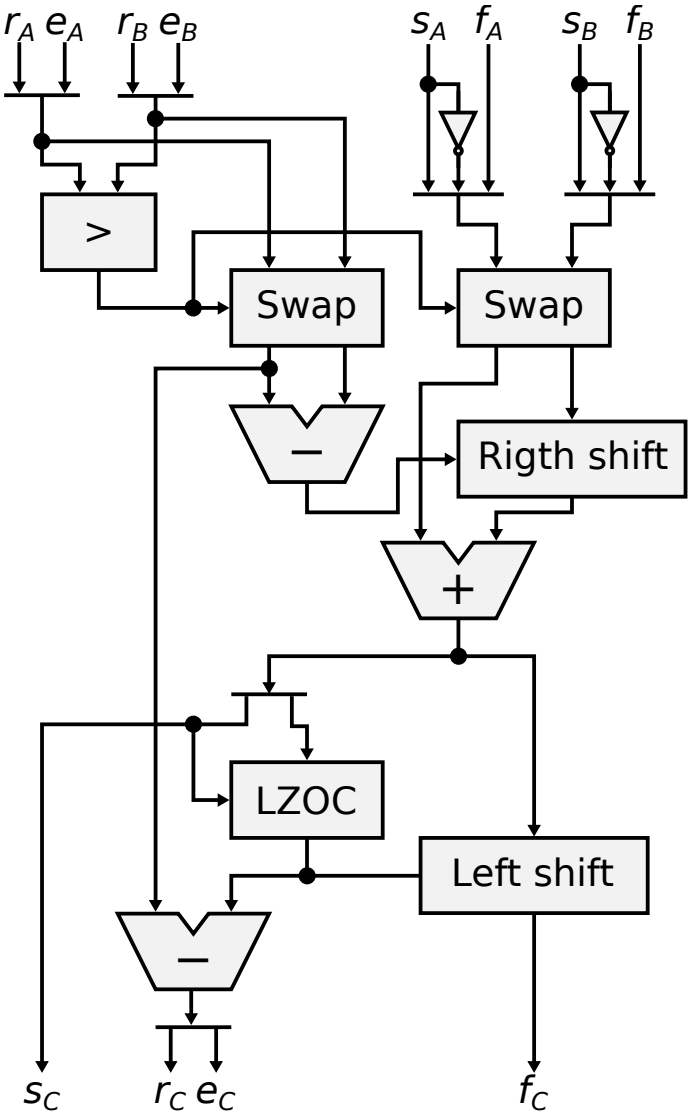


Figure 5.4: Block diagram of the proposed posit adder.

added, and the result is normalized, so the larger exponent is adjusted if needed. Figure 5.4 illustrates the architecture of the proposed unit.

When using a classical sign-magnitude decoding approach, like in FP, some extra logic is needed to handle the sign of the result. But such logic is eliminated when the sign of the operands is explicit in the significands. In such a case, the sign of the result can be inferred from the leftmost bit of the addition of the significands, without initially comparing the magnitude of the inputs. Dealing with signed significands also avoids the need to perform subtraction or taking two's complement when the sign of both addends differ, which makes up for using one extra bit in the addition of significands. On the other hand, normalization of the significand in two's complement deserves special attention, since it needs to count not only the leading zeroes but the leading ones when the result is negative. However, this does not introduce additional hardware overhead for this particular module.

5.4. Posit multiplication

Multiplying two posit numbers A and B is done as in Equation (5.2):

$$A \times B = ((1 - 3s_A) + f_A) \times ((1 - 3s_B) + f_B) \times 2^{((1-2s_A)\times(2^{es}r_A+e_A+s_A))+((1-2s_B)\times(2^{es}r_B+e_B+s_B))}. \tag{5.2}$$

Similarly to addition, posit multiplication draws inspiration from the FP algorithm, as depicted in Figure 5.5. In this process, both significands are multiplied and normalized, and the exponents are added together. It is important to note that, unlike floats, there is no need to consider a bias for the exponents when dealing with posits. Additionally, the result from significand multiplication must be normalized to fit in the corresponding interval, which involves shifting the fraction plus adding the number of shifted bits to the exponent. However, when dealing with signed fractions, even though the multiplication can be performed directly (just one extra sign bit for each operand is necessary), the normalization of the result is more complex in this case. As each multiplicand $(1 - 3s + f)$ is within the range $[-2, -1) \cup [1, 2)$, the result will fall within the range $(-4, -1) \cup [1, 4]$. In terms of fixed-point arithmetic, the operand has two integer bits, so the multiplication result has four bits that represent the integer part of the number and that should be examined in the normalization process (note that the resulting sign is also implicit in the multiplication result). However, this approach introduces one extra case that needs special attention: when the two multiplicands are equal to -2, the result is 4. In such a case, when normalizing the result, it is necessary to increase the exponent by 2, rather than 1.

5.5. Posit division

From a mathematical point of view, the division of two posit numbers A and B is done similarly as the multiplication, like Equation (5.3) shows:

$$\frac{A}{B} = \frac{((1 - 3s_A) + f_A)}{((1 - 3s_B) + f_B)} \times 2^{((1-2s_A)\times(2^{es}r_A+e_A+s_A))-((1-2s_B)\times(2^{es}r_B+e_B+s_B))}. \tag{5.3}$$

As can be seen, in posit division fractions are divided and exponents are subtracted, without the need of taking into account any bias, like for the IEEE 754 numbers.

The division is the most challenging arithmetic operation to implement in hardware design, as it differs from the other three operations in that it typically yields two components: the quotient and the remainder. However, for rational or real numbers, division always produces a single number, which is the inverse of multiplication. Therefore, computer division often involves approximation, since the dividend may not be exactly divisible by the divisor. Also, to reduce complexity, fast division algorithms are commonly used, even at the expense of accuracy.

Posit division consists of a fixed-point division with some extra hardware to take care of the exponents, which makes the divider circuit more complex. The major steps for a posit division are:

- Extracting the sign (s) of the result from the two sign bits.

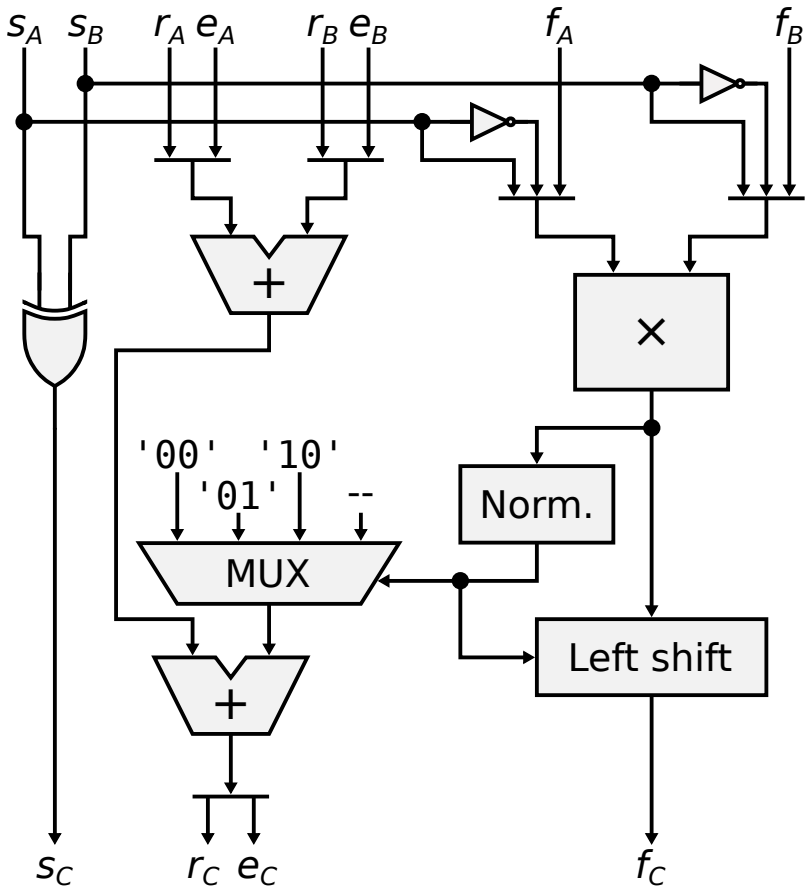


Figure 5.5: Block diagram of the proposed posit multiplier.

- Finding the magnitude of the difference between two exponents/scaling factors.
- Dividing the fraction of A (f_A) by the fraction of B (f_B), considering the hidden bits (these are also known as significands).
- Normalizing the result by shifting left, if it is out of the possible range of values.
- Due to the normalization, the exponent is to be decremented according to the number of left shifts.

The general architecture for posit division is shown in Figure 5.6. There are two subtractors used in the divider architecture, one for the exponent subtraction and one for the correction of exponents. The main hardware block is the divider block, whose implementation will be discussed below. If the result of the divider is not in the correct range of values then the normalizing step is executed. The significand of any posit number is in the interval $[-2,-1) \cup [1,2)$, so the result from the division of two significands will be in the interval $[-2,-0.5] \cup (0.5,2)$. Thus, up to two bits should be shifted from the result (shifting two bits is necessary only when the quotient is equal to -0.5), and subtracted to the final exponent.

Obviously, the most complex module within such an operator is the divider block. Implementing binary division is not straightforward. It often requires numerous hardware

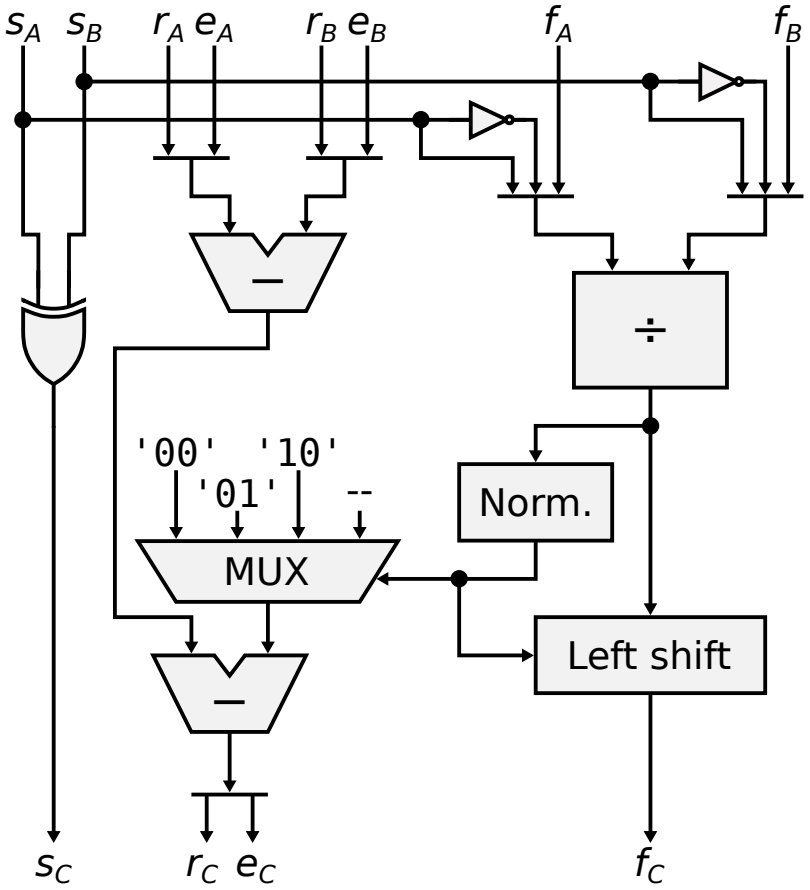


Figure 5.6: Basic scheme for posit division.

resources and takes multiple clock cycles. Actually, such a component is usually not included in FPGA or application-specific integrated circuit (ASIC) libraries. There are different algorithms that can be used to implement such a block, which will be discussed below. However, it is important to mention that, unlike in the case of floating-point, in this case, the numerator and denominator (fractions of posits), and therefore the quotient, are all signed values, so the corresponding algorithm must be adapted to handle signed numbers. The selected division algorithm will determine the accuracy, area, and delay of the final FU.

There are three main families of division algorithms: digit-recurrence, multiplicative, and polynomial-based methods [38]. In this work, one of the most common digit-recurrence algorithms (non-restoring) and two well-known multiplicative (also known as iterative) methods (Newton-Raphson, and Goldschmidt) are presented, and the necessary modifications to adapt them for posit arithmetic are discussed as well. Hereafter, we refer to the division operands as numerator $N = f_A$, denominator $D = f_B$, quotient Q and remainder R, so that $N = D \times Q + R$.

5.5.1. Non-restoring division

This is a slow division method based on the recurrence of the partial remainder R_i [38]. As depicted in Algorithm 2, it generates a single quotient bit per iteration. Thus, it requires

Algorithm 2 Non-restoring posit division.

```
1:  procedure NonRestoring(N, D)
2:      R_0 ← N
3:      for i = 1, ..., k do
4:          if 2R_{i-1}, D have the same sign then
5:              q_i ← 1
6:              R_i ← 2R_{i-1} − D
7:          else
8:              q_i ← −1
9:              R_i ← 2R_{i-1} + D
10:     Q ← 00.q_1 q_2 ... q_k
11:     return Q
```

at least as many iterations k as fraction bits of the numerator N. That is, the number of steps in digit-recurrence algorithms is linearly proportional to the number of bits k. However, the answer produced by this method is exact, rather than the one produced by faster algorithms. Each iteration of the non-restoring division just performs an addition or subtraction and a multiplication by 2 (which can be implemented by a left shift of one position). It is noteworthy that this algorithm uses the digit set $\{-1, 1\}$ for the quotient digits q_i instead of $\{0, 1\}$, but it can be easily converted to conventional two's complement representation [83].

Non-restoring division assumes that $|N| < |D|$. To ensure that, the numerator N is always shifted one position to the right, and the result Q is then shifted one bit to the left. For that reason, one extra iteration of the loop must be computed. Another extra three iterations must be performed—one for the case the result must be normalized, as depicted in Figure 5.6, and two additional steps to get the round and guard bits. Special attention is deserved in the case where the numerator $N = -2$ and the denominator $D = 1$. In such a case, the right shifting of N does not satisfy the previous inequality, so the final result must be corrected. Also, note that line 10 of Algorithm 2 adds two zeros to the quotient. This corresponds to the two integer bits that fractions of posit numbers have. This differs from the conventional implementation of non-restoring division for fixed/floating-point formats, which assume a single integer bit and just one zero would be added in such a case. Finally, the quotient Q must be corrected to ensure that the final remainder R_n and the numerator N have the same sign [83].

5.5.2. Newton-Raphson division

This is based on the Newton-Raphson iterative method to obtain the zero of a function [38], in this case, of the function $f(X) = 1/X - D$ (whose zero is $1/D$). This method is also known as division by reciprocation, as it first calculates the reciprocal of the denominator D and then multiplies it by the dividend to obtain the final quotient. Such reciprocal is obtained by successive more accurate approximations $X_1, X_2, \ldots, X_k$ obtained by Equation (5.4),

$$X_i = X_{i-1} \times (2 - D \times X_{i-1}). \tag{5.4}$$

The recurrence requires an initial approximation X_0 for the reciprocal of the denominator, and each iteration performs two multiplications and one subtraction. Nevertheless, in

Algorithm 3 Newton-Raphson posit division.

```
1: procedure NewtonRaphson(N, D)
2:     X_0 ← 1/D
3:     for i = 1, ..., k do
4:         X_i ← X_{i-1} × (2 − D × X_{i-1})
5:     Q ← N × X_k
6:     return Q
```

Algorithm 4 Goldschmidt posit division.

```
1: procedure Goldschmidt(N, D)
2:     X_0 ← 1/D
3:     N_{-1} ← N
4:     D_{-1} ← D
5:     for i = 0, ..., k do
6:         N_i ← N_{i-1} × X_i
7:         D_i ← D_{i-1} × X_i
8:         X_{i+1} ← 2 − D_i
9:     Q ← N_k
10:    return Q
```

contrast to the digit-recurrence algorithms, in the iterative/multiplicative methods, the number of steps is proportional to $\log_2(n)$ [83]. Algorithm 3 depicts this process for k steps. The initial guess X_0 directly affects the accuracy of the final solution, and it is usually obtained from a LUT, or by another algorithm. Implementation details are given in Section 5.7.

5.5.3. Goldschmidt division

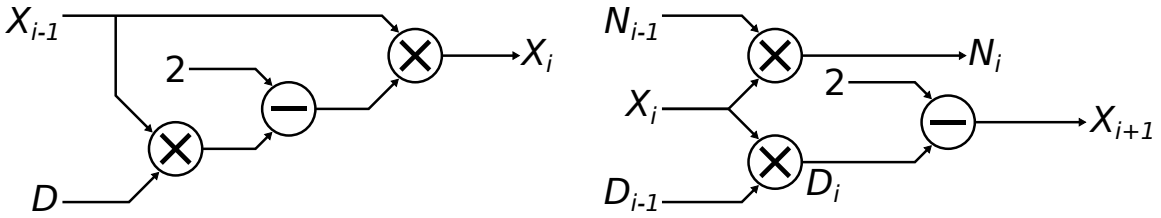
Also known as division by convergence [83], this method uses an iterative process of repeatedly multiplying both the numerator and denominator by a common factor, X_i for $i = 0, 1, \ldots, k$, such that, for sufficiently large k, the denominator converges to 1 and hence the resulting numerator converges towards the desired quotient, as shown in Equation (5.5):

$$Q = \frac{N}{D} = \frac{N}{D} \times \frac{X_0}{X_0} \times \frac{X_1}{X_1} \times \ldots \times \frac{X_k}{X_k} \longrightarrow \frac{Q}{1}. \tag{5.5}$$

The iterative process is depicted in Algorithm 4. Similarly, as in the Newton-Raphson division, this method requires an initial factor X_0 (a good choice for this is again an estimate for the reciprocal $1/D$), and each iteration performs two multiplications and a single subtraction from 2, as depicted in Figure 5.7. However, notice that in this case, the two multiplications are independent; consequently, they can be done in parallel, increasing the performance of the divider.

5.5.4. Implementation of posit dividers

The implementation of the aforementioned division algorithms has been done in a combinational manner. The number of iterations per algorithm, as well as the initial guess



(a) Newton-Raphson. **(b)** Goldschmidt.

Figure 5.7: Implementation scheme for iterative dividers.

for the reciprocal of the denominator, follow the recommendations from [72]: 1 and 2 iterations for 16 and 32-bit formats, respectively, and the approximation for the reciprocal is obtained from a LUT. This also allows obtaining a fairer comparison between the proposed implementations and the one presented in [72] (known as PACoGen). In contrast with PACoGen, in this work, posit numbers are decoded in two's complement format, rather than in sign-magnitude. Therefore, it is necessary to check for one extra bit in the fraction (corresponding to the sign bit) for having the same accuracy as the divider from PACoGen has, so a LUT of size $2^9 \times 10$ is required. Since combinational units are implemented, and hardware components are not reused through successive iterations, rectangular multipliers are selected to save resources. In each Newton-Raphson/Goldschmidt iteration, two $(8i + 2) \times (wF + 2)$ signed integer multipliers are used, where i is the iteration number and wF is the width of the fraction (without the hidden bits). This configuration offers sufficient accuracy with a minimum of hardware resources. For smaller posit bit length, just LUT approximation can be used (note that, for example, 8-bit posits only have 3 fraction bits), and a single final multiplication must be performed.

5.6. Posit square root

The square root is the only one of the basic operations that considers a single input operand. As shown by Equation (5.6), the square root of a non-negative posit number A requires obtaining the square root of the significand and producing the scaling factor of the result.

$$\sqrt{A} = \sqrt{1+f} \times 2^{\frac{(1-2s)\times(2^{es}r+e+s)}{2}}. \tag{5.6}$$

If the exponent is odd, it is decremented by 1, and the significand is multiplied by 2.

The implementation of this operation is very similar to posit division. More precisely, the square root extraction is conceptually similar to the division methods discussed in the previous section [38, 83]. However, in this case, the significand is always non-negative, otherwise, the result of the operation is NaR. Figure 5.8 illustrates the general architecture of the posit square root unit. In this thesis, the square root is implemented using the binary non-restoring algorithm. To obtain the root, the algorithm requires the operand to be in the range $[0.25, 1)$, so the input to the square root submodule is divided by 2 (of by 4, if the exponent is odd) to guarantee such a condition. This is done by a 1-bit (or 2-bit) operand left-shift.

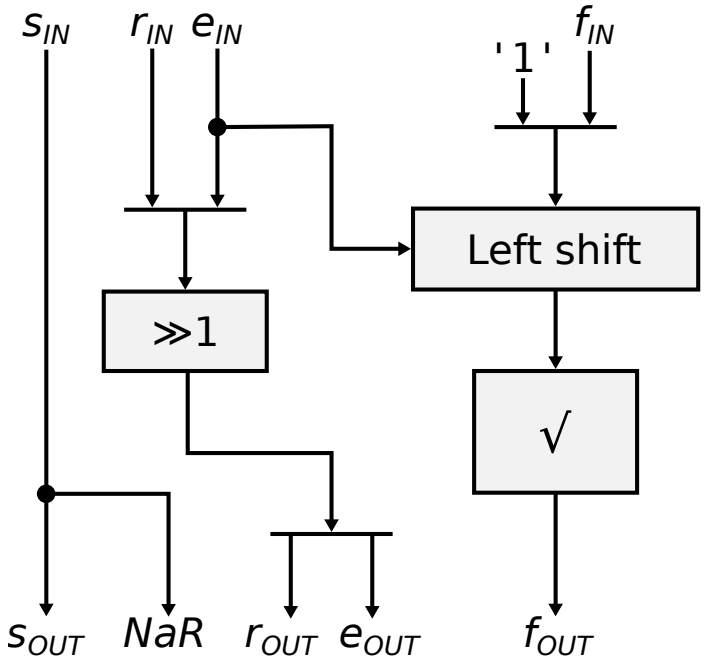


Figure 5.8: Basic scheme for posit square root.

5.7. Evaluation

The proposed generic designs have been implemented into FloPoCo, which generates synthesizable VHDL for any Posit$\langle n, es \rangle$ operator. Simulations with extensive testing vectors are performed to verify the functionality of the proposed designs. These testing vectors have been generated with the Universal reference library [137].

To evaluate the impact of proposed designs in terms of hardware resources, designs with different bit lengths have been synthesized using Synopsys Design Compiler® with a 45 nm target-library from TSMC with typical case parameters and without placing any timing constraint.

5.7.1. Comparison with state-of-the-art posit implementations

The proposed designs have been compared with the state-of-the-art designs from PACoGen [72][1] and MArTo [161][2], which are open-source and publicly available. PACoGen is presumably one of the first complete implementations of posit arithmetic units, including division operation. Their designs are based on the sign-magnitude interpretation of posit arithmetic (see Appendix A). On the other hand, MArTo, which is based on the two's complement interpretation of posits, is a library that provides custom-sized arithmetic types for HLS, including posits. Thus, the FUs from MArTo are obtained using Vitis HLS 2021.2 for C++ to hardware description language (HDL) compilation with default options. Unfortunately, designs from PACoGen are restricted to $es > 0$, and no automatic pipeline is allowed. The former of these drawbacks is solved by the proposed designs, while FloPoCo

[1]Source code from `https://github.com/manish-kj/PACoGen/tree/5f6572c`.
[2]Source code from `https://github.com/lforg37/marto/tree/7a34237`.

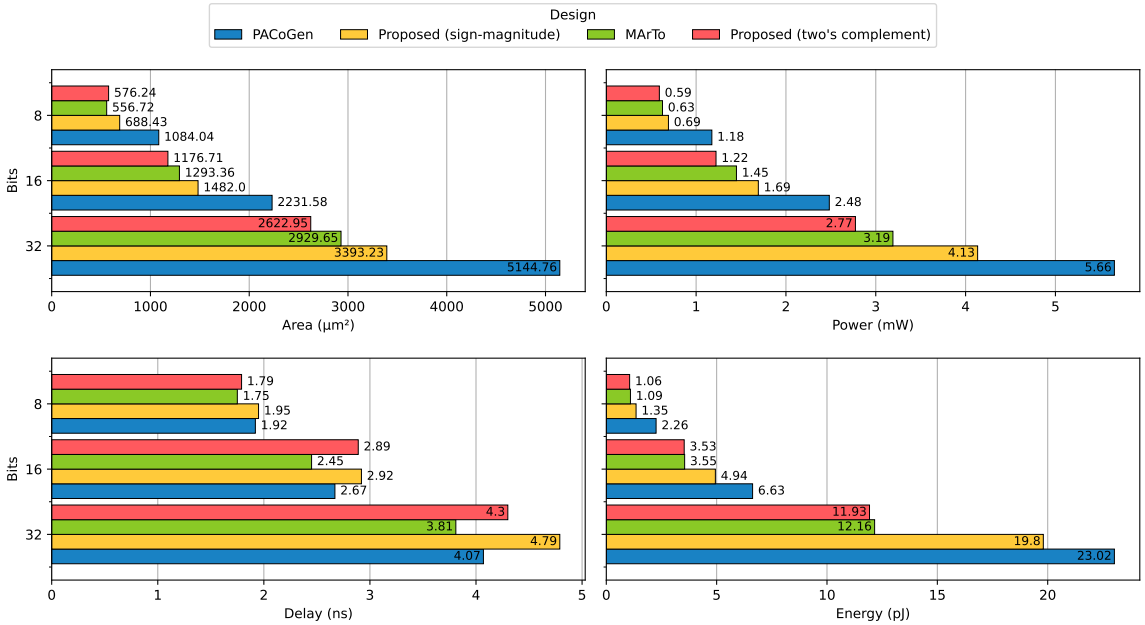


Figure 5.9: Synthesis results for different Posit$n$ adder designs.

solves the latter, as it can automatically generate pipelined operators for any given frequency. Thus, to have a fair comparison with previous works, combinational operators with $es = 2$ are generated (although similar results are obtained for different es values). In particular, posit operators of standard formats and precisions are compared, this is Posit8, Posit16 and Posit32. For the sake of completeness, posit adders and multipliers are implemented using both sign-magnitude and two's complement interpretation of posits (see Appendix A.4 for a more detailed comparison between both implementations).

Addition/subtraction

Synthesis results of posit adder designs are shown in Figure 5.9. Under the same interpretation, the proposed adder designs generally present better figures than those of previous work (except for the delay). The proposed two's complement adders present better overall results. Just the MArTo 8-bit adders require slightly less area than the proposed one, but such a design does not scale as well as the one proposed in this thesis as the bit length increases. Nevertheless, in all cases, the proposed adder designs have a lower energy consumption.

Multiplication

The multiplier hardware resources are depicted in Figure 5.10. In this case, the results follow the same trend as for the adders. The proposed two's complement designs provide more efficient implementations than the rest of the state-of-the-art implementations in practically all the tested cases. The delay results present a clear gap between the designs based on the sign-magnitude posit decoding (including PACoGen) and the ones using the

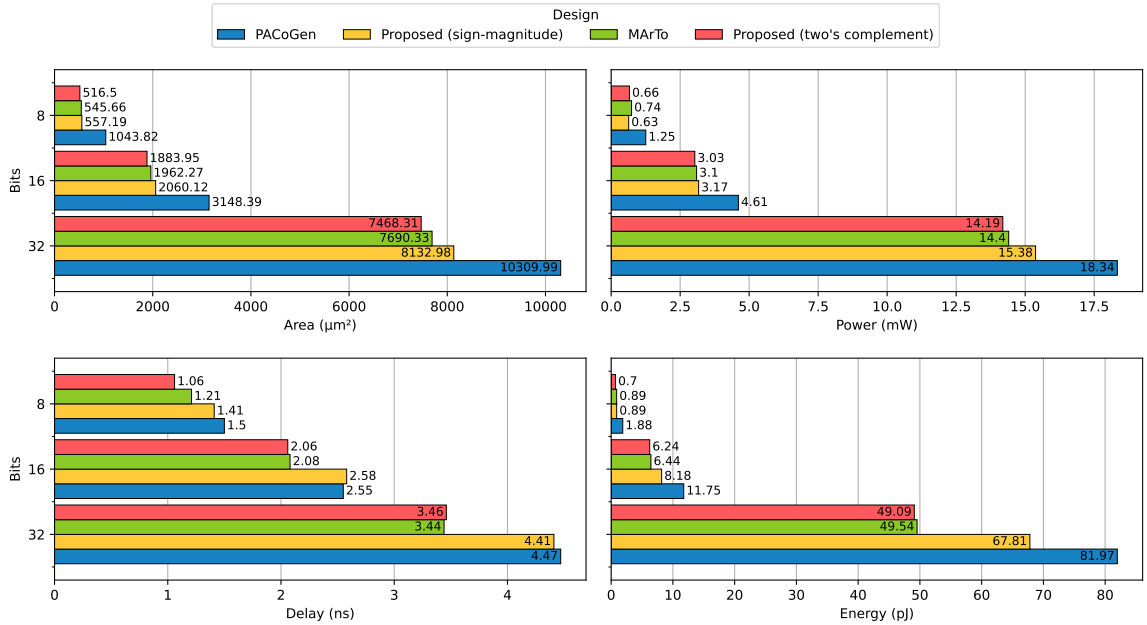


Figure 5.10: Synthesis results for different Posit$n$ multiplier designs.

two's complement scheme (including MArTo). This is especially remarkable for the 16 and 32-bit cases, where operators using sign-magnitude decoding present around 24% and 29% more delay, respectively, in comparison with the two's complement multipliers.

Division

From the aforementioned posit implementations, only PACoGen provides an implementation for posit division (which is based on the Newton-Raphson algorithm). At the time of writing, to the best of our knowledge, this is the only available open-source implementation of posit division. Figure 5.11 presents a comparison between the design of previous work and the new designs proposed in this thesis, for both 16- and 32-bit implementations. As Posit8 iterative dividers have only 3 fraction bits, the division can be computed immediately with a small LUT, rendering this precision uninteresting. Conversely, both 16- and 32-bit iterative dividers are implemented using the same LUT for the initial guess, with the former taking only one iteration of the corresponding algorithm, and the latter taking two.

Non-restoring dividers demonstrate a clear advantage over iterative implementations in terms of area and power consumption. Specifically, results show that non-restoring dividers require only about 53% of the area and 65% of the power of the proposed Newton-Raphson implementation, which is the most area-efficient iterative divider. It should be noted, however, that non-restoring dividers are slower than multiplicative operators and require a larger number of iterations. This results in a performance penalty of approximately 3 to 4 times slower than multiplicative operators. As a consequence, the energy consumption of non-restoring dividers can be up to twice as high, as shown in Figure 5.11.

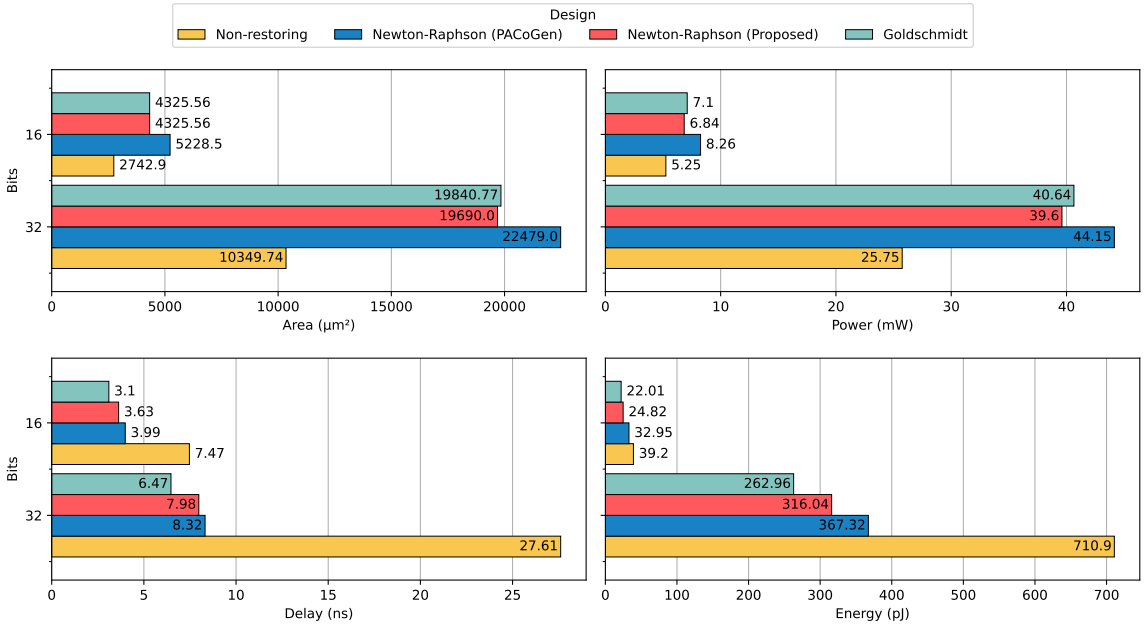


Figure 5.11: Synthesis results for different Posit$n$ divider designs.

Both dividers based on multiplicative methods yield comparable results in terms of area and power consumption, with the Goldschmidt divider exhibiting superior performance due to its non-dependent multiplications. As discussed in Section 5.5, the parallel nature of the Goldschmidt algorithm allows for increased throughput and faster operation.

The Newton-Raphson implementation proposed in this thesis outperforms the PACoGen implementation in all respects. Despite requiring a larger LUT and wider multipliers to accommodate signed fractions, the use of two's complement notation for posit numbers offsets the extra hardware cost dedicated to fraction division. As a result, the presented 32-bit implementation achieves reductions in area, delay, power, and energy consumption by 12.4%, 4.1%, 10.3%, and 14%, respectively (with even greater savings achieved for the 16-bit case).

Square root

To the best of our knowledge, at the moment of writing this thesis only a few works have presented an implementation of the square root for posit numbers [144, 171, 23]. The two former integrate the unit with the division operation, and are not open-source, while the latter presents a separate module for the square root operation, written in Chisel. All of them implement the non-restoring algorithm. Although the work from [23] is open-source, the presented FUs are integrated within a whole posit arithmetic unit, which makes comparison difficult. What is more, such units are implemented under non-standard formats (Posit$\langle 8, 1\rangle$ and Posit$\langle 32, 3\rangle$), which makes the comparison even more unfair.

Table 5.1: Synthesis results for Posit$n$ square root units.

Bits	Area (μm^2)	Power (mW)	Delay (ns)	Energy (pJ)
8	444.06	0.57	1.06	0.60
16	1923.00	3.00	3.61	10.84
32	8066.18	15.75	13.84	217.94
64	32509.81	73.79	52.70	3888.93

Table 5.2: Synthesis results for float and posit arithmetic operators.

Operator	Bits	Format	Area (μm^2)	Power (mW)	Delay (ns)	Energy (pJ)
Addition	32	Float	1697.20	1.74	3.40	5.92
		Posit	2622.95 (1.55×)	2.77 (1.59×)	4.30 (1.26×)	11.91 (2.01×)
	64	Float	3522.12	3.40	6.04	20.54
		Posit	5409.60 (1.54×)	5.52 (1.62×)	6.76 (1.12×)	37.32 (1.82×)
Multiplication	32	Float	4949.78	9.09	2.83	25.72
		Posit	7468.31 (1.51×)	14.19 (1.56×)	3.46 (1.22×)	49.10 (1.91×)
	64	Float	23710.51	44.58	5.15	229.59
		Posit	33532.70 (1.41×)	65.69 (1.47×)	6.55 (1.27×)	430.27 (1.87×)
Division	32	Float	5324.93	10.98	13.43	147.46
		Posit	10349.74 (1.94×)	25.75 (2.35×)	27.61 (2.06×)	710.96 (4.82×)
	64	Float	19038.03	48.73	47.20	2300.06
		Posit	39751.39 (2.09×)	109.48 (2.25×)	101.96 (2.16×)	11162.58 (4.85×)
Square root	32	Float	2779.83	6.01	10.95	65.81
		Posit	8066.18 (2.90×)	15.75 (2.62×)	13.84 (1.26×)	217.98 (3.31×)
	64	Float	12811.81	33.05	42.56	1406.61
		Posit	32509.81 (2.54×)	73.79 (2.23×)	52.70 (1.24×)	3888.73 (2.76×)

Synthesis results for square root posit units are shown in Table 5.1. Since the non-restoring algorithm requires one iteration per digit in the fraction field, there is a notable area and delay increment when doubling the bit length.

5.7.2. Comparison with floating-point operators

As posits are considered as a potential replacement for IEEE 754 floating-point computations, it is essential to provide a detailed comparison of the operators implemented in each format. Table 5.2 presents a comparative analysis of posits and floats of the same size across the four basic arithmetic operations described in this chapter. The overhead of posit units is displayed for the sake of clarification.

The floating-point units under consideration have been generated using FloPoCo. While the floating-point addition implemented by this tool aligns completely with the IEEE 754 standard, it is noteworthy that the remaining operations adhere to mainstream floating-

point implementations[3] but deviate from the standard. Specifically, they lack support for subnormal numbers (which are flushed to zero) and do not provide complete exception handling. Despite these deviations, this framework suffices for establishing a reference point between posit operators and their floating-point counterparts.

It is evident that the posit FUs result in higher hardware cost than their floating-point counterparts (due to sequential decoding and larger implicit fraction). Whereas posit adders and multipliers provide an area and delay overhead of 1.5× and 1.25×, respectively, posit division and square root units are not so well optimized. According to the results, posit division units require about double the area and consume much more energy (4.8×) per operation compared to the corresponding FPUs. A higher area overhead is observed in the case of square root, almost the triple for 32-bit and 2.5× for 64-bit, while in this case, the delay and energy overhead are lower, up to 1.26× and 3.31× respectively. These results reflect the fact that the division and square root units are significantly more intricate than adders and multipliers. In fact, while the corresponding designs for floating-point have been optimized through decades of research, those presented in this paper are among the first designs for posit division and square root. As a consequence, there are still opportunities for further research to develop superior designs and datapaths for these units.

5.8. Conclusions

In this chapter, we provided a comprehensive exploration of the four fundamental arithmetic operations—addition, subtraction, multiplication, and division—as well as the square root within the context of posit arithmetic. The posit functional units (FUs) have been designed in a parametric manner, accommodating any possible Posit$\langle n, es \rangle$ format. The design process was facilitated through the use of FloPoCo, which also enables the automatic creation of pipelined units.

These FUs were implemented and compared with prior units documented in the literature. Synthesis evaluations demonstrate that the proposed units surpass previous posit implementations, exhibiting improvements in terms of both area and energy consumption, and, in some instances, reduced delay.

Furthermore, a comparative analysis between the proposed posit FUs and FPUs of equivalent size has been conducted. Results indicate that posit arithmetic introduces a certain degree of hardware overhead compared to classical floating-point. This trade-off is necessary to accommodate the enhanced accuracy offered by the novel posit format.

Additionally, posits extend the concept of fused arithmetic, enabling precise results for a wide range of operations. This unique capability will be thoroughly explored in the subsequent chapter.

[3]Division is implemented using the radix 4 SRT algorithm, and square root is implemented using the binary non-restoring algorithm.

6

Fused posit arithmetic unit

In addition to the previously discussed properties of the posit format, the standard [51] introduces the so-called *fused operations*, which allow computations with more than two operands (such as the dot product) to be performed without intermediate rounding. This capability reduces the error associated with such calculations. To enable this, posit arithmetic introduces the concept of *quire*, a fixed-point accumulator of sufficient size to allow for the exact accumulation of posit products.

In this chapter, we delve into the concept of fused posit arithmetic and propose a hardware design for the computation of fused multiply-accumulate (MAC). Specifically, a unit for the computation of fused MAC is presented and evaluated in terms of accuracy and hardware resources. Given the potentially high cost of this unit, various implementation alternatives are explored, illustrating multiple area-performance trade-offs.

Specifically, Section 6.1 introduces the concept of fused arithmetic and explains the intricacies of the quire accumulator. The proposed design for the posit MAC unit is detailed in Section 6.2. Auxiliary operations such as posit-quire format conversion are also elaborated. Section 6.3 evaluates both the accuracy and hardware implementation of the fused MAC operation and explores the effects of alternative designs. Finally, Section 6.4 concludes the chapter with a description of the principal contributions and comments.

6.1. The quire accumulator

The concept of quire is similar to the Kulisch accumulator for floating-point format [87]. Instead of rounding the result after each product and before accumulating such value, Kulisch suggested the usage of a higher-precision register in which partial products are accumulated without rounding, reducing this way the impact of the rounding errors. However, this is not included in the IEEE 754 standard. The only similarity to fused posit operations in the standard floats is the FMA operation, which allows to perform a single product and accumulation without rounding, but was not included in the standard until the 2008 revision [65].

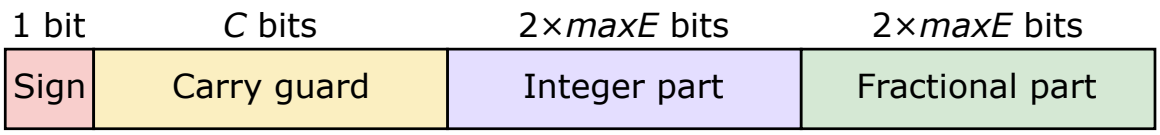


Figure 6.1: Quire format encoding for Posit$\langle n, es \rangle$.

Table 6.1: Quire parameters for different posit formats.

Posit			Quire	
n	es	$maxE$	C	$qSize$
8	0	6	7	32
8	1	12	15	64
8	2	24	31	128
16	1	28	15	128
16	2	56	31	256
32	2	120	31	512
64	2	248	31	1024

According to the posit standard, fused expressions must be explicit in the source code, and the quire accumulator must be accessible to the programmer. A posit-compliant system needs to support quire operations, as well as rounding from quire to posit and conversion of posit to quire in the matching posit format. The precision of the quire format is determined by the dynamic range of the corresponding Posit$\langle n, es \rangle$ format. The largest and smallest positive posit values are 2^{maxE} and 2^{-maxE}, respectively. The largest exponent for such a format is given by Equation (6.1),

$$maxE = (n-2) \times 2^{es}, \tag{6.1}$$

which corresponds to the cases without exponent and fraction bits. To correctly represent a posit number in fixed-point format, $maxE$ bits are needed for the fractional part, and $maxE + 1$ bits for the integer part. Thus, a total of $4 \times maxE$ bits is not enough to hold the result of multiplying any two posit numbers in fixed-point format (multiplying the biggest possible number by itself can not be represented this way). Adding extra C bits guarantees that up to $2^C - 1$ sums of products can be evaluated exactly without overflow risk. The quire format corresponding to a Posit$\langle n, es \rangle$ format is depicted in Figure 6.1.

A common rule of thumb is to take C carry bits so that the total length (including the sign bit) of the quire ($qSize$) is a power of two. Table 6.1 summarizes the quire sizes and relevant parameters for different posit formats. The latest version of the posit standard [51] particularizes this for $es = 2$, so that $qSize = 16n$ bits and the amount of carry bits is always 31.

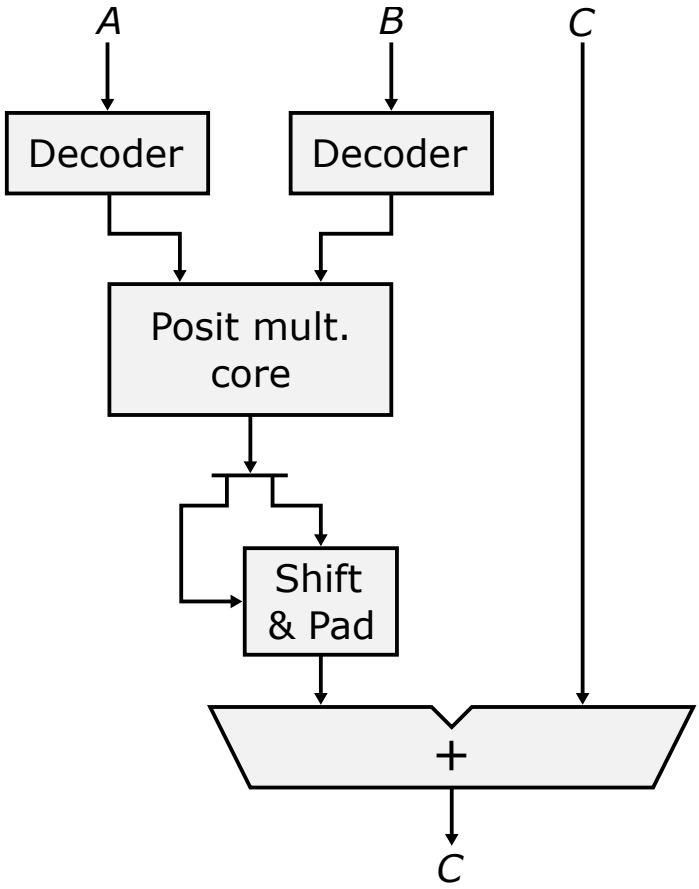


Figure 6.2: Basic scheme for posit quire MAC operation.

6.2. Posit MAC unit design

The MAC operation is a common step that computes the product of two numbers and adds that product to an accumulator. For a posit implementation to be compliant with the standard it must support full functionality for fused expressions that can be written in the form of a dot product of vectors of length less than 2^{31}. This is much more powerful than simple FMA floating-point operation[1] and thus, a new MAC unit that leverages the quire accumulator is necessary to satisfy such a requirement in hardware[2]. In addition to the core of the fused operation, special operations are required by the posit standard to convert from posit to quire format and vice versa. These units can be independent of the fused operations, but are explained in this section as well.

[1]Fused expressions (such as fused multiply-add and fused multiply-subtract) need not be performed using quire representations to be posit-compliant.

[2]According to the standard [51], the decision of how to satisfy the requirements is up to the implementer. As a consequence, the computation of fused expressions could be implemented in software, at the expense of compromising its performance.

6.2.1. Exact accumulation of products

The architecture of the proposed fused posit MAC unit is shown in Figure 6.2. For the sake of clarification, signals related to special case detection are omitted in the datapath diagram. The design does not depend on the total bit length or the number of exponent bits. As in general MAC operations, three operands are taken as inputs. In this case, only two of them are in posit format, while the remaining is the quire accumulator, which holds an initial value for the accumulation. This allows to perform even dot products in a fused manner, with a single rounding.

The first step in the operation is to decode the fields of the posit inputs. Then, the product of the two posits is computed by multiplying both fractions and adding the regime and exponent factors, which are grouped to form the corresponding scale factors. In order to transform the intermediate result into a quire representation, the product is shifted by a number of bits given by the addition of scaling factors. Finally, both quires (the one received as input and the one converted from the previous product) are added together, so that the result of the operation is a new quire in which successive products can be accumulated in the same manner.

Since the quire encoding is a fixed-point two's complement format, the final addition of the MAC unit can be performed in the usual way. However, given the bit lengths shown in Table 6.1, such an adder is the module with the highest resource consumption within the unit. For example, in a Posit$\langle 32, 2\rangle$ format, the 512-bit quires to be added are much larger than the 29-bit fractions (including hidden bits) to be multiplied. Thus, the quire adder is a critical component for the efficiency of the whole fused MAC unit. Previous works have addressed this issue, either by detecting the smallest amount of bits that are necessary to add (only for 32-bit posits) [17], or by segmenting the quire into smaller words (at the cost of a larger number of cycles for correct carry propagation) [161]. In this thesis, several variants of quire adders and segmented units will be evaluated in Section 6.3.

6.2.2. Quire to posit conversion

At the end of a fused operation, it is necessary to convert the value stored in the accumulator back to the corresponding posit format. Rounding is inherent in this process. First, it is necessary to detect any possible overflow by checking if any of the carry guard bits are different from the sign bit. Next, the regime and exponent values of the corresponding posit are obtained by counting the leading zeros (or leading ones in the case of negative values) in the integer part. Then, the fraction of the posit can be directly extracted from the quire. Finally, this fraction must be correctly rounded according to the remaining bits from the quire, also avoiding underflow. The process is depicted in Figure 6.3.

While this process seems to be computationally expensive, it must be noted that it only needs to be done once, at the end of the fused operation. In contrast, standard operations perform rounding and encoding of the results after every single calculation, which can increase the latency of calculations that require numerous operands [112].

6.2.3. Posit to quire conversion

The quire accumulator encodes numbers in fixed-point format, as shown in Figure 6.1. Therefore, converting a posit value into a quire format is straightforward when using the

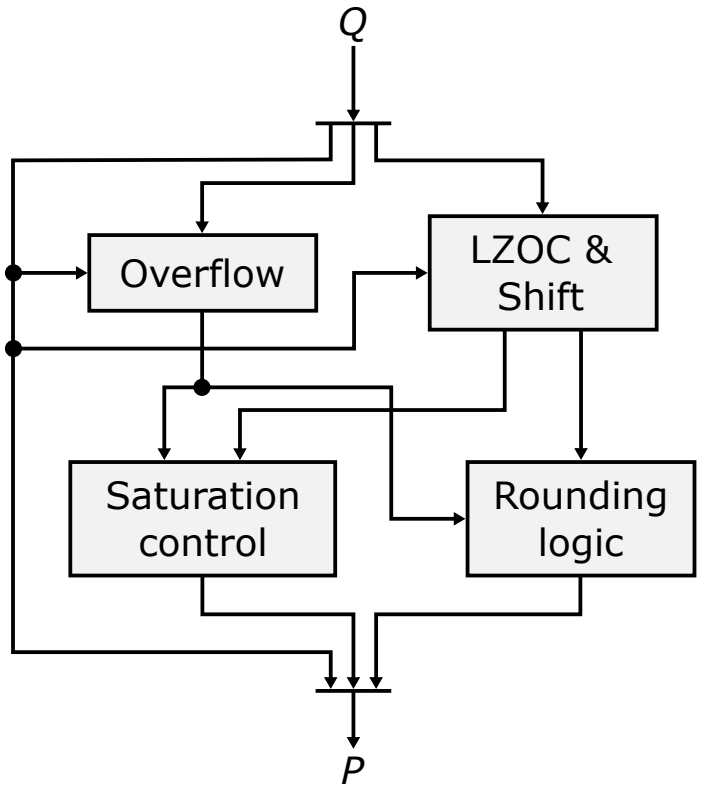


Figure 6.3: Basic scheme for quire to posit conversion.

proposed two's complement interpretation of posits. The posit fraction (including the hidden bits) must be shifted to the correct position according to its scaling factor. As the quire is prepared to hold any possible result of the multiplication of two posits, converting a single posit value to quire encoding would require padding the shifted fraction with 0's on the right and extending the sign bit on the left.

6.3. Evaluation

The design for posit-fused MAC units presented in the previous section was implemented using FloPoCo. In order to verify the correctness of the proposed architectures, extensive tests were generated using the Universal software library [137] for 8, 10 and 12-bit posits, and corner case tests for 16 and 32-bit posits, with different exponent and quire sizes. All of these tests were successful.

This section evaluates the accuracy and hardware cost of the proposed MAC units. For the first experiments, a software library is used to emulate fused operations. For the hardware evaluation, standard cell synthesis is performed under the same conditions as in Chapter 5, using Synopsys Design Compiler® with a 45 nm library from TSMC with typical case parameters. In addition, several alternative implementations of the proposed MAC units are considered.

Table 6.2: GEMM MSE comparison between IEEE 754 floating-point and posit numbers.

Input values	Format	Matrix size				
		16 × 16	32 × 32	64 × 64	128 × 128	256 × 256
[-0.1, 0.1]	Float32	1.515×10^{-18}	4.752×10^{-18}	1.566×10^{-17}	6.524×10^{-17}	2.432×10^{-16}
	Posit32	2.146×10^{-20}	6.726×10^{-20}	2.371×10^{-19}	7.805×10^{-19}	2.203×10^{-18}
	$\text{Posit32}_{\text{fused}}$	3.157×10^{-21}	6.110×10^{-21}	1.158×10^{-20}	2.014×10^{-20}	3.497×10^{-20}
[-1, 1]	Float32	1.324×10^{-14}	4.637×10^{-14}	1.686×10^{-13}	6.246×10^{-13}	2.416×10^{-12}
	Posit32	5.028×10^{-17}	1.727×10^{-16}	6.457×10^{-16}	2.447×10^{-15}	9.870×10^{-15}
	$\text{Posit32}_{\text{fused}}$	1.138×10^{-17}	2.355×10^{-17}	4.729×10^{-17}	9.430×10^{-17}	1.937×10^{-16}
[-10, 10]	Float32	1.300×10^{-10}	4.304×10^{-10}	1.708×10^{-9}	6.026×10^{-9}	2.447×10^{-8}
	Posit32	3.878×10^{-12}	1.341×10^{-11}	7.500×10^{-11}	3.282×10^{-10}	1.41×10^{-9}
	$\text{Posit32}_{\text{fused}}$	8.549×10^{-13}	1.475×10^{-12}	3.055×10^{-12}	6.355×10^{-12}	1.295×10^{-11}
[-100, 100]	Float32	1.293×10^{-6}	5.052×10^{-6}	1.595×10^{-5}	6.503×10^{-5}	2.440×10^{-4}
	Posit32	3.077×10^{-7}	1.230×10^{-6}	4.295×10^{-6}	2.804×10^{-5}	1.569×10^{-4}
	$\text{Posit32}_{\text{fused}}$	4.819×10^{-8}	8.266×10^{-8}	1.760×10^{-7}	6.150×10^{-7}	1.506×10^{-6}
[-1000, 1000]	Float32	1.675×10^{-2}	4.815×10^{-2}	1.644×10^{-1}	6.323×10^{-1}	2.433
	Posit32	4.168×10^{-2}	1.570×10^{-1}	5.669×10^{-1}	2.365	9.586
	$\text{Posit32}_{\text{fused}}$	5.293×10^{-3}	8.573×10^{-3}	1.900×10^{-2}	3.746×10^{-2}	8.265×10^{-2}

6.3.1. Accuracy

When compared with standard operations, using fused MAC operations provides clear benefits in terms of computation accuracy, but it comes with higher area and power costs when implemented in hardware. As these operators can perform numerous multiplications and additions without rounding, GEMM is a proper benchmark to evaluate the accuracy of the standard against fused operators. In particular, square matrices with uniformly distributed random values in intervals of the form $[-10^i, 10^i],\ i \in \{-1, 0, 1, 2, 3\}$ are generated. These intervals allow for a study of the impact of the input data range on the target application. The results obtained using the double-precision IEEE 754 floating-point numbers are considered as the golden solution and used to compute the MSE of the Posit32 result. For the sake of completeness, the MSE obtained using single-precision IEEE 754 floats is also included.

The MSE results are shown in Table 6.2 for different matrix sizes and input ranges. As can be seen, for 256 × 256 matrices, the difference between MSE is around two orders of magnitude when using fused operations. This gap increases by up to four orders when compared to the standard IEEE 754 format. By comparing the scaling of MSE as matrix size increases, it becomes evident that posit numbers exhibit superior behavior, attributed to the presence of a quire register. This holds true across all input value ranges. Across all the tested cases, the impact of the quire is notable, justifying its additional cost. In summary, the benefits of employing the quire register make posit numbers a better arithmetic format in terms of MSE, regardless of the input value range.

6.3.2. Hardware comparison with non-fused operations

Figure 6.4 presents the area and power synthesis results for both standard and fused posit units across various bit lengths. For this evaluation, combinational arithmetic units

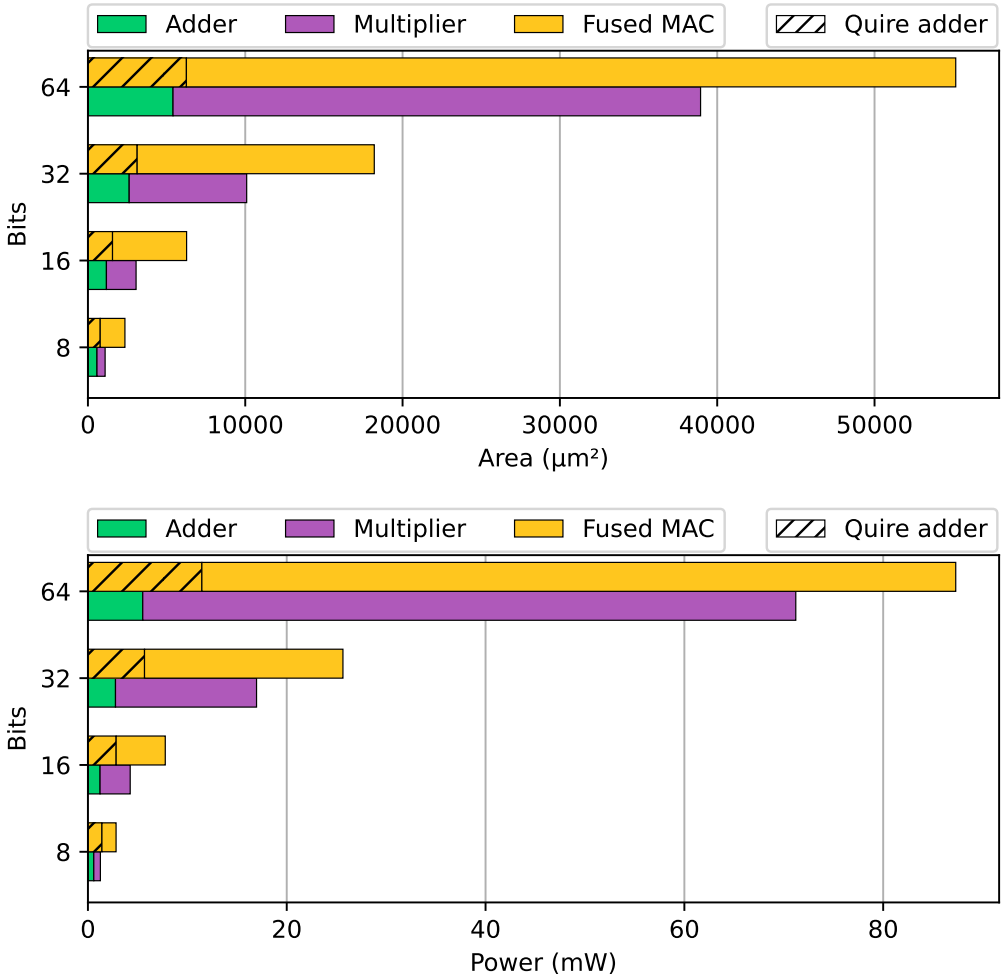


Figure 6.4: Hardware comparison between standard and fused posit units.

were taken into account during the synthesis process. Results illustrate that an exact accumulator consumes significantly more resources than standard operators. This discrepancy is contingent on the bit length, with an overhead of approximately $2\times$ for an 8-bit scenario. However, this overhead diminishes in larger data lengths. For instance, in the 64-bit case, the area and power overhead is reduced to $1.4\times$ and $1.2\times$, respectively.

For a more comprehensive understanding, Figure 6.4 also highlights the hardware requirements associated with quire accumulation. Notably, the $16n$-bit fixed-point adders (for the quire accumulator) require slightly more resources than their corresponding full Posit$n$ adders. This suggests that the primary resource utilization within fused MAC units resides in other components of the unit. Thus, reducing this logic could be a point of interest for future optimizations.

Conversely, utilizing fused operations with an exact accumulator can markedly reduce calculation errors, as demonstrated in the preceding subsection. Furthermore, the results presented in this evaluation focus solely on stand-alone operators. When incorporating these units into a complete core, other resources must be taken into account, such as registers

and the number of instructions. In this context, there are evident advantages to performing a single operation for multiplication and accumulation rather than two separate operations.

6.3.3. Implementation alternatives

As mentioned in Section 6.2, the addition of quires is the most resource-intensive part within the fused MAC operation. The quire accumulator can be quite large even for low bit length posits. Such a large adder is the main component that restricts the maximum frequency achievable by these arithmetic units due to the long carry propagation. Therefore, this module deserves special attention when designing fused operators.

In this section, different designs for the final quire adder depicted in the MAC architecture of Figure 6.2 are compared. In particular, the quire adder has been implemented with three different designs offering different area/performance trade-offs, namely the ripple-carry adder (RCA), Brent-Kung adder (BKA) [8] and Kogge-Stone adder (KSA) [82][3]. The RCA is the simplest design, composed of multiple full adders. However, this adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. More precisely, RCAs take $O(n)$ time for carry to reach MSB. The BKA and KSA adders are parallel prefix adders, so they provide better performance ($O(\log_2(n))$) at the cost of more area.

In order to improve the throughput of the MAC posit unit in practical applications, multiple pipeline approaches can be employed to design this operator. In this case, a 2-stage pipeline architecture separating quire addition from the rest of the operation, and a 3-stage pipeline architecture that also separates posit multiplication from posit-to-quire shifting at different stages are compared.

Unconstrained synthesis results for Posit16 and Posit32 are graphically shown in Figures 6.5 and 6.6, respectively, although similar ratios are obtained with other bit lengths. By using a BKA for quire addition instead of a RCA, the area is slightly increased ($\sim$5.6% and $\sim$4.8% for 16 and 32-bit cases, respectively), which is expected, since the area of this adder is $O(n)$ [35]. However, the power and datapath delay are reduced, resulting in average energy savings of more than 26% for the Posit16 and about 40.5% for the Posit32. This is due to the delay of the BKA, $O(2\log_2(n)) - 2$, lower than the $O(n)$ delay of the RCA. A larger delay reduction is achieved by using the KSA design, since it has an asymptotic delay of $O(\log_2(n))$. In this case, for Posit16 a delay reduction of 63.9% is achieved with the combinational design and about 87.1% in the pipelined cases (correspondingly, 70.9% and 91.5% delay reduction for Posit32), resulting in energy savings of 59.6% and 86.2%, respectively (correspondingly, 68.2% and 90.9% energy reduction for Posit32). On the other hand, and in contrast to the previous design, there is a considerable area increment ($\sim$38.8% and $\sim$32.8% for Posit16 and Posit32, respectively). This is also expected, since the area of KSA is $O(n\log(n))$ [35].

It is noteworthy that the critical path of the pipelined units is the same for both cases. As hypothesized, the final quire adder is the component with the largest delay, even in the case of the KSA design. For this reason, the 2-stage pipeline is the most energy-efficient design option.

[3]Designs obtained from `https://github.com/albertodbg/vhdl-arithmetic`.

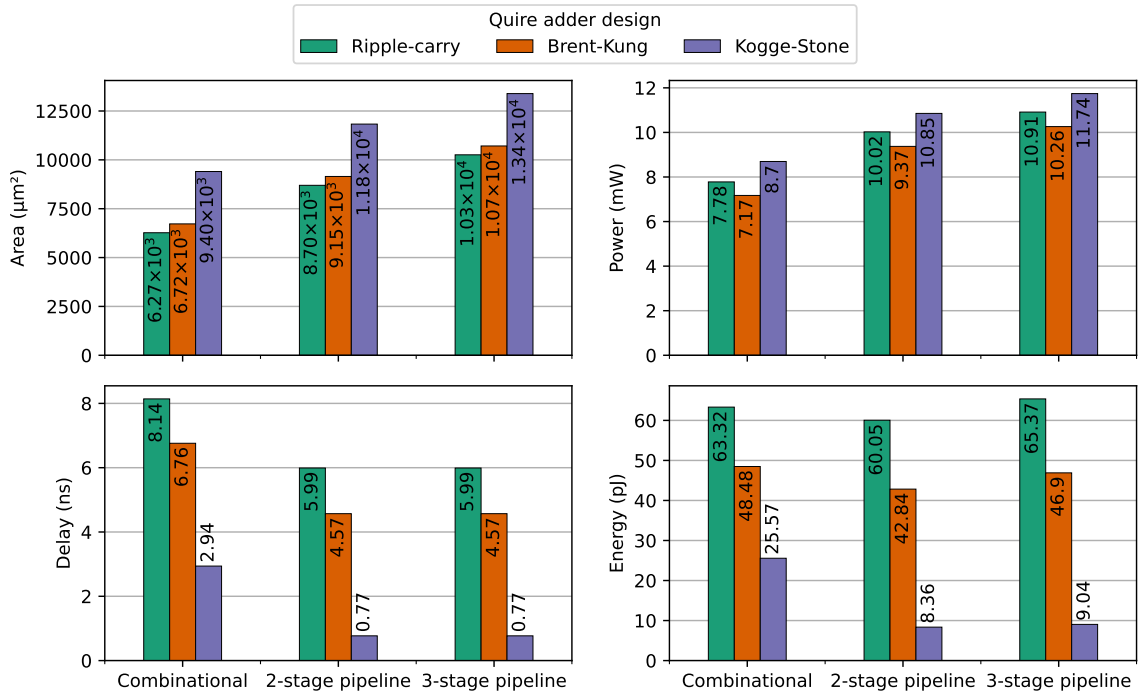


Figure 6.5: Synthesis results for different designs of Posit16 MAC units with 256-bit quire.

6.3.4. Comparison with state-of-the-art implementations

Deep Positron is a DNN kernel accelerator that works with posit arithmetic [12]. The architecture includes an exact MAC posit unit[4] for performing DNN inference with 8 bits or less with comparable accuracy to 32-bit floating-point. This exact MAC unit makes use of the quire, but both input and output are in posit format, so quire-to-posit conversion is incorporated in the unit. This implementation is not compliant with the standard draft, since the quire is not accessible to the programmer, it is just for internal accumulation. A different open-source design of exact accumulator with quire implemented in Chisel is SmallPositHDL[5]. In contrast with the previous one, this unit presents similar inputs/outputs as the one proposed in this work. Finally, authors in [161] presented MArTo, a library for generating parameterized posit-compliant units, including exact accumulation of products with quire[6].

The posit MAC units presented in those works are pipelined into three stages, generally comprised of 1) posit multiplication, 2) quire alignment, and 3) quire summation. To make a relatively fair comparison, the proposed 3-stages pipelined designs are selected. Also, the same number of exponent ($es = 2$) and carry ($C = 31$) bits are selected for all the designs, so the total bit length of the quire is the same in each case.

[4]Source code from `https://github.com/craymichael/Low-Precision-EMACs/tree/b6fe0d1`.
[5]Source code from `https://github.com/starbrilliance/SmallPositHDL/tree/c862e91`.
[6]Source code from `https://gitlab.inria.fr/lforget/marto/tree/0bc08ce8`.

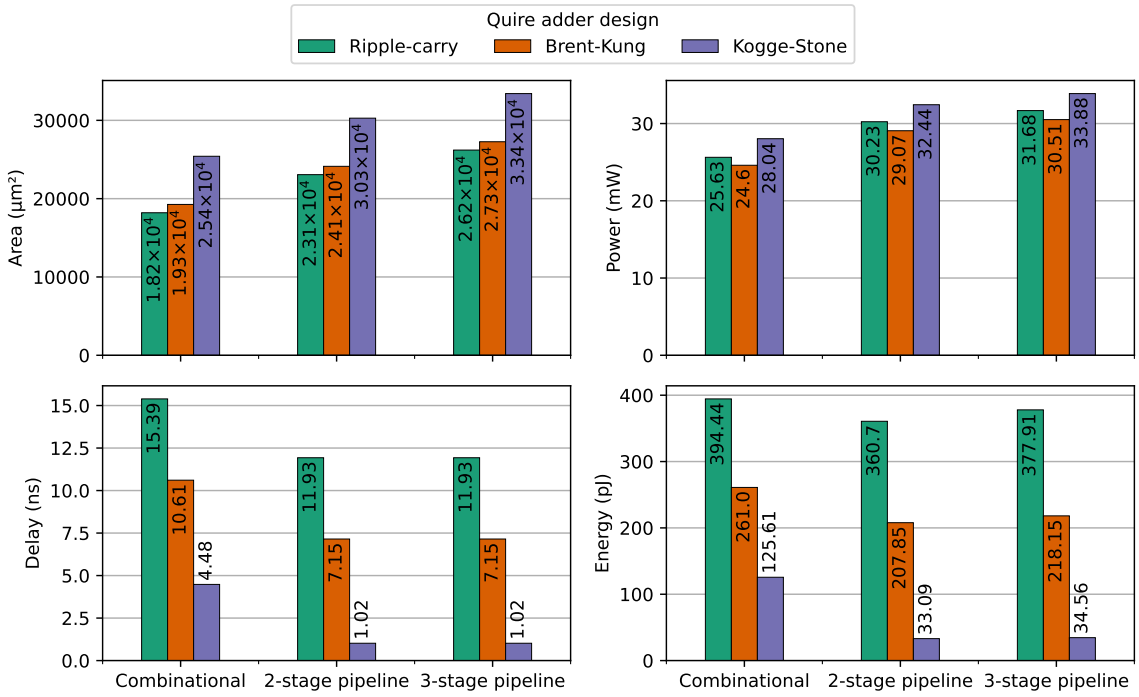


Figure 6.6: Synthesis results for different designs of Posit32 MAC units with 512-bit quire.

Table 6.3 lists the detailed comparison of standard cell synthesis between the different designs proposed in the literature. As can be seen, Deep Positron does not provide competitive arithmetic units, especially for large bit lengths, when compared with previous and proposed designs. The quire-to-posit converter included in these MAC units dramatically increases the hardware requirements, so keeping those modules separately seems to be a better design choice, as well as being compliant with the standard. The rest of the works follow that criteria, and they get results that are more similar to each other.

In virtually all cases, each of the proposed designs obtains the best results in each of the synthesis sections. The units using a RCA provide less area (except compared with the 8-bit unit from MArTo), power and delay than previous implementations, resulting in a more energy-efficient design. There is an exception for the 32-bit unit from MArTo, which has approximately the same delay as the 16-bit one. The explanation for this is found in the pipeline of the unit. In the 32-bit case, the 512-bit quire addition is split into two phases, using a 256-bit adder in each. This way, the datapath delay is the same as for the 16-bit unit, which also uses a 256-bit adder to add the quires. Such optimization is not taken in the proposed designs but could be considered for future works. As can be seen, among the implementations from previous works, SmallPositHDL offers the lowest energy consumption for the 8-bit operator, while MArTo provides the best results for the 16 and 32-bit cases. Thus, for each bit length, these operators are considered as the baseline for comparison.

Regarding the proposed implementations, and as the previous experiments show, using a KSA for quire addition results in the fastest and most energy-efficient fused MAC units. In

Table 6.3: Comparison with state-of-the-art Posit$n$ MAC units.

Design	n	Area (μm^2)	Power (mW)	Max Delay (ns)	Energy (pJ)
Deep Positron [12]	8	5349.15	4.46	4.38	19.51
	16	16147.42	12.18	11.70	142.48
	32	58937.59	40.07	42.01	1683.30
SmallPositHDL [169]	8	4590.87	4.73	3.02	14.30
	16	11263.02	12.37	5.99	74.11
	32	28366.06	34.79	11.94	415.39
MArTo [161]	8	**4234.78**	4.78	3.16	15.10
	16	11011.83	11.88	6.09	72.36
	32	29045.79	33.28	6.00	199.69
Proposed with RCA	8	4346.26	4.47	3.01	13.45
	16	**10257.78**	10.91	5.99	65.37
	32	**26204.34**	31.68	11.93	377.91
Proposed with BKA	8	4525.95	**4.18**	2.66	11.11
	16	10710.77	**10.26**	4.57	46.90
	32	27267.44	**30.51**	7.15	218.15
Proposed with KSA	8	5585.53	4.84	**0.81**	**3.92**
	16	13391.11	11.74	**0.77**	**9.04**
	32	33417.22	33.88	**1.02**	**34.56**

this case, the proposed design reduces energy consumption by 72.60%, 87.50% and 82.70% for the 8, 16 and 32-bit units, respectively, when compared with previous works. On the other hand, this energy reduction is accompanied by an increase in area of 21.66%, 21.61% and 15.05%, respectively, when compared with the same alternatives. Recall that Table 6.3 compares 3-stage MAC units, but the results shown in Figure 6.6 reveal that the 2-stage design is more efficient than the 3-stage one in terms of area, power and energy. In such case, the energy reduction would be slightly similar (75.49%, 88.45% and 83.43%, respectively) while the area of the units would be just 4.97%, 7.44% and 4.24% more than that of the state-of-the-art operators. In addition, the latency of the operators is reduced by 73.18%, 87.36% and 83.00%, respectively.

Finally, performing the quire addition with a BKA is an intermediate compromise between the other two solutions. This design choice requires quite less power, delay and energy than the implementations from previous works, while still taking a similar or even smaller area.

6.4. Conclusions

Throughout Chapters 3 and 4, the posit format has demonstrated to be a promising alternative to the IEEE 754 floating-point standard in a variety of areas such as deep learning

and physical simulations. Nevertheless, there are circumstances where even greater accuracy is required. As a solution, the posit arithmetic comprises fused operations.

In this chapter, a fused posit multiply-accumulate (MAC) unit design is proposed to enhance the development of the posit number format, as well as facilitate its utilization in computationally intensive applications. The experimental results show that using the fused posit MAC, instead of standard multiplication and addition, for large matrix multiplication significantly mitigates computational error arising from intermediate rounding by two orders of magnitude. The parameterized design is implemented with varying posit configurations, featuring a 1, 2 and 3-stage pipeline, and employing different adder designs, including ripple-carry adder, Brent-Kung adder and Kogge-Stone adder. Compared to other posit hardware solutions, the proposed implementation achieves notable reductions in energy and latency reduction up to 88.45% and 87.36%, respectively, with a modest 7.44% increase in area.

This chapter, along with the previous one, concludes the design of the posit arithmetic units. Based on these basic building blocks, it would be possible to design a complete posit arithmetic unit capable of executing posit instructions within a processor [106]. Nonetheless, to incorporate such a unit, adjustments to the instruction set and/or the processor data path are necessary. An alternative is to develop hardware accelerators for specialized posit-based applications. This idea will be further elaborated in the following chapter.

7

Generation of posit-based accelerators

In the contemporary computing landscape, characterized by the ubiquity of data-intensive applications driven by advances in deep learning and the escalating demand for high-performance computing, the challenge of sustaining performance improvement emerges with the culmination of Moore's law and Dennard scaling [56]. Recent studies underscore the significance of domain-specific architectures (DSAs) or accelerators and specialized processors, such as GPUs, FPGAs, and ASICs, as promising avenues for augmenting both performance and energy efficiency [34]. In this context, the advantages of posit arithmetic position it as an appealing candidate for DSAs, offering potential mitigation against the impacts of the conclusion of Moore's law and Dennard scaling. While preceding studies have extensively scrutinized the hardware costs associated with posit arithmetic in individual operations, there persists a paucity of information concerning the interconnection and integration of these components as a cohesive whole.

This chapter builds upon previously introduced operators and introduces a comprehensive high-level synthesis (HLS) flow that supports posit arithmetic. This facilitates the design of custom accelerators for FPGAs directly from a given behavioral specification, utilizing posit numbers instead of traditional floating-point arithmetic. The proposed approach leverages the higher accuracy inherent in posits. Additionally, a novel memory customization approach that employs posit numbers in memory while retaining accelerator logic in floating-point format is presented. This can be a useful solution when the designers want to optimize local memories and data transfers, but still use legacy HLS tools that only support traditional floating-point notations.

Section 7.1 reviews some basic concepts and motivates the utility of HLS in the design and development of DSAs. In Section 7.2, the proposed methodology and strategies for seamlessly integrating posit operators into an HLS tool are explained in detail. The evaluation of the proposed approach, inclusive of benchmarks, implementation results, and analyses, is thoroughly discussed in Section 7.3. Concluding the chapter, Section 7.4 offers pertinent conclusions derived from the presented material.

7.1. Domain-specific hardware accelerators

In the ever-evolving landscape of computing, to meet computing needs and overcome power density limitations, the computing industry has entered the era of parallelization. Such highly parallel, general-purpose computing systems, however, still face serious challenges in terms of performance, area and energy consumption. The demand for specialized and efficient processing capabilities has given rise to the concept of domain-specific architectures (DSAs). Traditional general-purpose processors, while versatile, may not always offer the optimal performance for specific tasks or applications. DSAs address this challenge by providing targeted, high-performance solutions tailored for particular workloads or domains [31, 19].

At their core, domain-specific accelerators are hardware components designed to execute specific types of computations, such as signal processing, AI, cryptography, or graphics rendering. Unlike general-purpose processors that are built to handle a wide range of tasks, DSAs are finely tuned to deliver superior performance for a particular set of applications.

In domains where computations involve complex mathematical operations, such as scientific simulations, numerical analysis, or machine learning, floating-point DSAs can be particularly valuable. These accelerators enhance the performance of algorithms that rely on extensive floating-point calculations. Examples of floating-point DSAs include GPUs, TPUs, DSPs, FPGAs, and custom ASICs tailored to specific domains.

In this regard, posit-based accelerators could be designed to handle computations with a wide dynamic range and precision, making them suitable for applications where numerical accuracy is crucial.

7.1.1. High-level synthesis

High-level synthesis (HLS) is an automated design process that, starting from the high-level description of an application, a register-transfer level (RTL) component library, and specific design constraints, identifies an RTL structure that implements the given behavior [28, 27]. The main steps executed by an HLS tool are the following:

1. **Compilation**. The HLS process begins with the compilation of the functional specification. This initial step transforms the high-level input description (typically ANSI C/C++) into a formal representation. Usually, it often involves several code optimizations, such as data dependency resolution, dead-code elimination, or loop transformations.

2. **Allocation**. The type and the number of hardware resources (e.g., FUs, storage, or connectivity components) are defined based on design constraints. These components are selected from the RTL component libraries. The utilization of numerical representations with lower bit length directly impacts the hardware resources of the final design, as it results in the allocation of less memory and smaller FUs.

3. **Scheduling**. All operations required in the specification model must be scheduled into control steps. This is done considering the functional components, operation priorities, and dependencies. Operations can be chained or scheduled to execute in parallel, provided there are no data dependencies and sufficient available resources. Different numerical representations might result in different scheduling results. Typically,

reducing the bit length of the signals also reduces the datapath delay, yielding faster circuits.

4. **Binding**. Within the computed schedule, each variable carrying values across control steps must be bound to a storage unit or register. Variables with non-overlapping life intervals may share the same register. Similarly, operations must be bound to the capable FUs, preventing those that execute concurrently from sharing the same resource instance. Register and FU binding also depends on interconnection binding, which introduces steering logic or connection units (such as buses or multiplexers) to facilitate transfers between components. The impact of numerical representation in the preceding steps directly influences the binding stage. Smaller data might require smaller or fewer FUs, registers, and hardware resources in general.

5. **Netlist generation**. The final architecture, derived from allocation, scheduling, and binding tasks, is translated into an RTL model of the synthesized design in a HDL like Verilog or VHDL. This process accesses the resource library, embedding the RTL implementation of each resource. The result is target-dependent, leading to potentially different hardware descriptions for various technologies.

HLS raises the design abstraction level and allows rapid generation of optimized RTL hardware for performance, area, and power requirements. For DSAs, which are designed to accelerate specific computations, this high-level abstraction facilitates a more productive and intuitive design process. Designers can focus on the algorithmic and functional aspects of the accelerator without delving into low-level hardware details. Thus, HLS serves as a bridge between high-level software descriptions and low-level hardware implementations, enabling designers to explore and generate hardware designs more efficiently. In addition, HLS tools enable designers to explore a wide range of design alternatives and optimizations at a high level of abstraction. This way, designers can evaluate the performance of DSAs in the early stages of development without the need for extensive low-level hardware design. More specifically, allocation, scheduling, and binding can be performed simultaneously or in a specific sequence depending on the strategy and algorithms used.

In summary, HLS plays a crucial role in the design and development of DSAs by providing a high-level abstraction for hardware design, enabling efficient exploration of design space, supporting rapid prototyping, and facilitating the integration of hardware accelerators with software components. This synergy contributes to the overall success of hardware-software co-design in the realm of DSAs.

At the time of writing this thesis, limited published works delve into the application of posit arithmetic in DSAs and HLS. Notably, authors in [161] introduced MArTo, a C++ library for posit arithmetic compliant with Xilinx Vitis HLS. It is built on a custom internal representation that leverages the Vitis HLS arbitrary precision integer type, and supports addition, subtraction and multiplication of posit datatypes, as well as the exact accumulation of posit products. Although this library is designed from a higher abstraction level, it requires adapting the source codes to use it, and its usage is currently limited to the aforementioned operations. Moreover, experiments in [161] evaluated just the performance of standalone posit operators, but no results on posit-based accelerators generated from HLS are given. On the other hand, authors in [17] present a posit accelerator for matrix multiplication, and

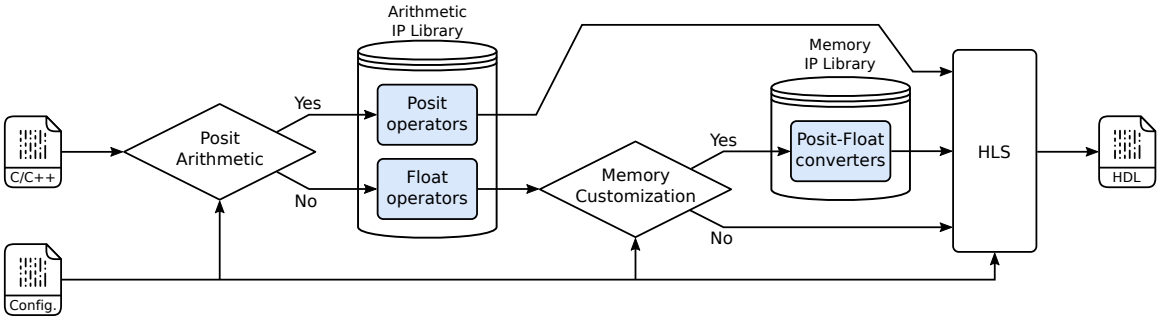


Figure 7.1: HLS workflow with support for posit arithmetic and memory customization.

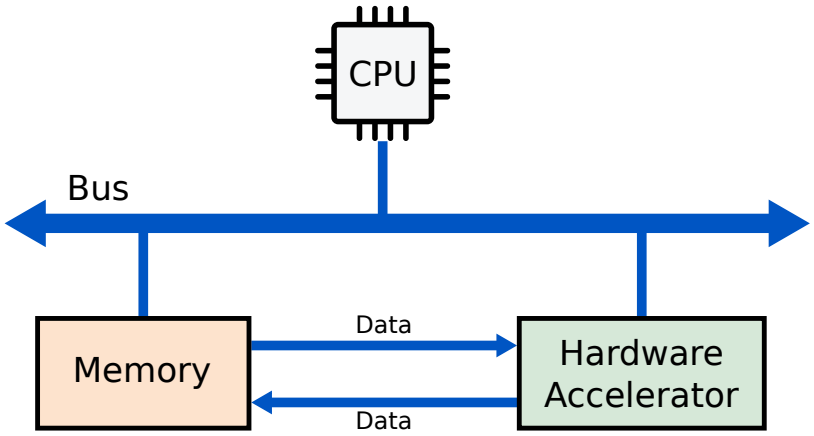
a posit accelerator for L1 BLAS routines and for pair Hidden Markov Model are presented in [162].

7.2. Posit-aware high-level synthesis

The work presented in this chapter is focused on the integration of posit arithmetic into the HLS flow. To achieve this objective, the initial phase involves the implementation of the required RTL components. Subsequently, specific stages of the synthesis process are modified to accommodate such an arithmetic format.

A scheme of the proposed tool flow is depicted in Figure 7.1. To add support for posit arithmetic to the HLS flow, it is necessary to 1) design an RTL library of posit operators with support for different design restrictions and provides equivalent functionality to floating-point operations, and 2) integrate the library into an HLS tool with a selection/substitution mechanism that preserves the memory infrastructure of the device. Although certain implementation aspects of the proposed approach are closely tailored to the software technology, the fundamental concepts of this approach can be extended and potentially applied to any HLS tool.

Furthermore, prior research has demonstrated that storing data in posit format can lead to reduced memory requirements while preserving accuracy [93, 32]. Such an approach is also useful when the designers want to customize communication by reaping the benefits of posits while still using legacy HLS tools that do not support non-standard formats. In light of this observation, this study also aims to investigate the implications of adopting such a design alternative, specifically examining the impact of employing posit numbers in memory with accelerators that do not inherently support this arithmetic format. By doing so, we intend to gain insights into the potential benefits and challenges associated with incorporating posit-based memory in systems utilizing non-posit arithmetic accelerators. Figure 7.2 illustrates the different strategies evaluated in this work. Case #1 represents the classical floating-point approach, and it serves as a reference point. It is important to note that in case #3, where just floats are used in the accelerator, certain data conversion processes would be necessary to ensure compatibility with the data originally stored in posit format. For each case, both 32 and 64-bit precision are considered.



Device	Case #1	Case #2	Case #3
Memory	Float	Posit	Posit
Accelerator			Float

Figure 7.2: Scheme of the different approaches that use float and/or posit arithmetic in memory and hardware accelerator.

7.2.1. Library of posit operators

HLS tools transform a high-level specification into an RTL design. Realistic hardware implementation thus requires the conversion of floating-point and integer variables into bit-accurate data types of a specific length (not a standard byte or word size, as in software) with acceptable computation accuracy. This is done in the resource allocation step of the HLS process (see Figure 7.3 below), and usually requires RTL libraries to map the variables and structures at higher abstraction levels to specific hardware components.

In Chapter 5, posit units for addition, subtraction, multiplication and division are designed and implemented using FloPoCo. Such FUs have empirically demonstrated to be efficient in terms of performance and energy. A complete HLS flow might require other operations, such as square root, comparison of operands and conversion with integer arithmetic. Additionally, conversions between floating-point and posit formats are necessary for the case when the memory and the accelerator work with different data types (case #3 in Figure 7.2). The required FUs were implemented in FloPoCo as well. The advantage of designing posit operators with FloPoCo is that this tool can generate such operators for any given bit length. In this work, 32 and 64-bit precision are considered, so the same design is used for the different bit lengths; FloPoCo automatically adjusts the size of the internal wires and signals according to the specified data width. The tool also allows pipelining the FUs automatically according to the specified parameters and target frequency. The implementation of parameterized designs with FloPoCo makes the creation of the library of posit operators independent of the design of the final accelerator.

In order to verify the correctness of the proposed architectures, exhaustive tests were generated with the reference software library Universal [137] for 8, 10 and 12-bit posits, as well as random/corner case tests for 16, 32 and 64-bit posits. All these tests were successful.

7.2.2. Memory customization

Modern HLS tools allow the use of a wide variety of memory allocation policies and memory accesses. They automatically infer the memory infrastructure according to the constraints, commands and types of the operands (e.g., integer, float, etc.). However, the posit data type is neither available in high-level languages nor commercial hardware devices, so some extra effort is needed to customize the memory with posit format. This work considers the 32-bit and 64-bit precisions of the different arithmetics. Therefore, float (32-bit) and double (64-bit) C types are used as replacements for Posit32 and Posit64, respectively, as they have the same bit length, and therefore memory accesses do not change. Using a different number of bits would require changing either the memory inside the accelerator or the compiler (so that it understands a different kind of floating-point operation), which is out of the scope of this work.

Posit arithmetic claims to have higher accuracy using the same number of bits, or sufficient accuracy using fewer bits. When storing data in fewer bits, the total memory footprint is reduced, reaching lower power and energy consumption. But also, when transferring data from external memory to devices such as FPGAs, using smaller bit lengths might allow transferring more data simultaneously under the same bandwidth, increasing the SIMD vectorization and achieving higher throughput.

In this work, a comparison of the effects of using 32-bit posits in memory with regard to 64-bit floats is presented. While this clearly halves the size of the external memory, it is important to evaluate its impact on the accuracy of the results.

Additionally, a hybrid scenario is proposed wherein the memory is customized with posit format while the accelerator logic is kept unchanged in floating-point. The usefulness of this approach lies in storing the data in a posit format with a smaller bit length than the floating-point format in which the computations are performed. From the memory point of view, this is not different from the case when all data is in posit format. However, under this approach, the input data must be converted into floating-point format before performing computations in the accelerator, and the results must be stored in memory using posit format. The conversion between n-bit posits and floating-point numbers with e exponent bits and m fraction bits is depicted in Algorithm 5, and the reverse analogous process is done for float-to-posit conversion.

7.2.3. Integration of posits into HLS tool

In this work, we consider Bambu, an open-source HLS research framework [139, 40]. The tool receives as input a behavioral description of the specification, written in C/C++ language, and generates the HDL description of the corresponding RTL implementation as output, which is compatible with commercial RTL synthesis tools. Bambu supports different target FPGAs, so the generated accelerator can be flashed into a board. In addition, it is designed in an extremely modular way and supports floating-point operations through

Algorithm 5 Posit to IEEE 754 float conversion.

```
1:  procedure POSIT2FLOAT(X)
2:      sign ← x[n − 1]
3:      val ← x[n − 2 : 0]
4:      if val = 0 then
5:          if sign = 0 then
6:              y ← 0
7:          else
8:              y ← NaN
9:      else
10:         if sign = 0 then
11:             abs_val ← val
12:         else
13:             abs_val ← −val                          ▷ Take 2's complement
14:         regime, exp, frac ← extract_fields(abs_val)
15:         biased_exp ← {regime, exp} + bias           ▷ IEEE float exponent bias = 2^(e−1) − 1
16:         y ← {sign, biased_exp, frac}                ▷ Pad y with 0's to the right if necessary
17:     return y
```

FloPoCo. The choice of Bambu as HLS tool for this work is motivated by its open-source philosophy and its integration with FloPoCo.

Performing HLS with a custom arithmetic format such as posit is not straightforward, since the data format and the RTL components interfere in multiple steps of the HLS design flow. As mentioned in Section 7.1, the synthesis process starts with the compilation of the C/C++ source code. However, the posit data type is not supported in such programming languages or compilers, in contrast to the float or double types. Incorporating compiler support for new data types is quite a complex task that is out of the scope of this work. Consequently, the decision was made to leave that aspect unchanged in the process, opting instead for subsequent modifications in the synthesis stages. To accomplish this, it is necessary to modify the allocation and scheduling stages of the HLS flow, as indicated in Figure 7.3.

At this point, two possible alternatives arise for the integration of posits into the HLS flow. Either replacing the floating-point functionality with posit arithmetic or maintaining both formats and introducing any control mechanism to select which format to use. While the former of the options has a simpler implementation, we aim to compare the results obtained for each format, as depicted in Figure 7.2. Thus, to keep the two arithmetic formats available as well as to facilitate the usage of different back-end arithmetic libraries for end users, the existing Bambu option `--flopoco` is modified to accept either the value `float` (which remains the default functionality), or the value `posit`. When the `--flopoco=posit` flag is activated, floating-point operations in the source code are intended to be computed in posit arithmetic. This approach allows handling posit arithmetic without the need to modify the C/C++ source code nor the compiler. From the point of view of the programmer, the use of posit arithmetic for the computation of real numbers just requires selecting between single or double-precision floating-point and activating the corresponding flag.

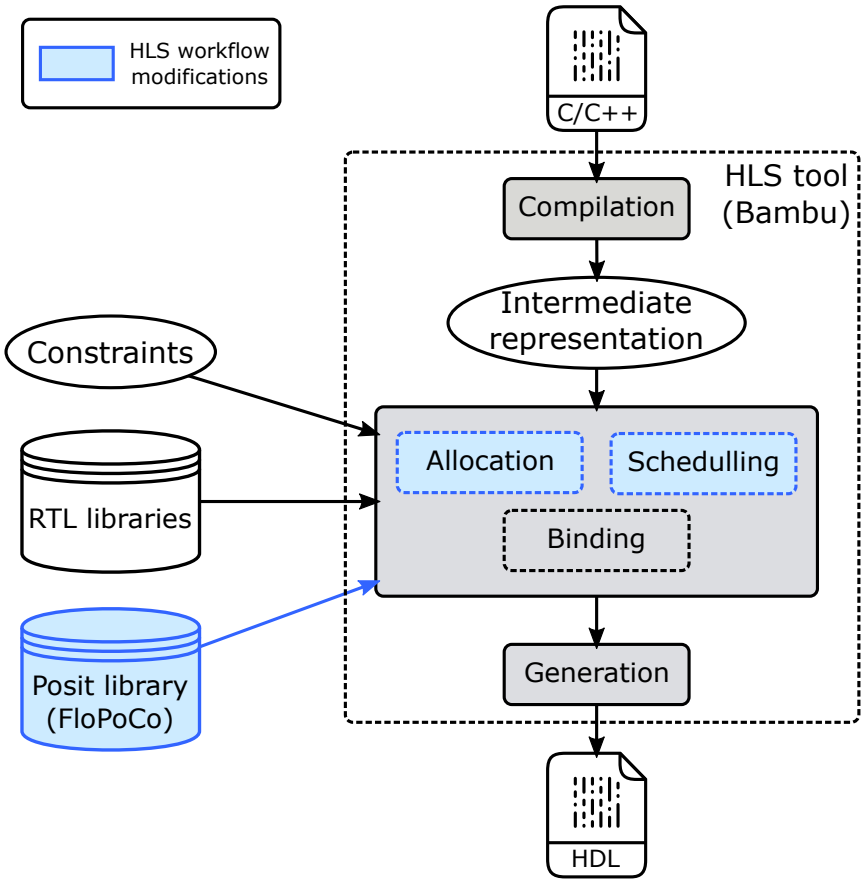


Figure 7.3: HLS design flow and modifications to accommodate posit units.

After the compilation step, Bambu detects all the arithmetic operations that are required in the source code. When a floating-point operation is detected, Bambu calls FloPoCo with the corresponding parameters to generate the required RTL operators according to the design constraints (allocation step). At this point, a different implementation from the IP library can be selected, for example, a posit adder instead of a floating-point adder circuit. However, the HLS tool must be aware of the latency of such components in order to properly generate the entire accelerator (during the scheduling and binding phases).

Regarding the allocation stage, certain modifications were done to incorporate the posit RTL library. This involved ensuring that the appropriate library, whether it is the float or posit library, is called based on the specified configuration of the HLS and the `--flopoco` flag. When this flag is used with the posit value, floating-point operations in the source code are translated in the corresponding RTL for posit operations defined in FloPoCo. The allocation step maps them on the set of available FUs: their characterization includes information, such as latency, area, and the number of pipeline stages. The pipeline of the operators is automatically driven by FloPoCo according to the specified design constraints such as the clock period. In addition to FUs, also memory resources are allocated, in this case as detailed in Section 7.2.2. Along the entire HLS process, especially during resource allocation and scheduling, it is necessary to have certain information about each

component in order to perform synthesis optimizations. For this reason, Bambu adopts a pre-characterization approach. The latency and resource occupation of every posit FU are obtained by synthesizing them for multiple combinations of bit lengths, frequencies and target devices. This has a direct impact on the scheduling stage of the HLS design flow, and is a common approach adopted by other HLS tools such as LegUp [11]. It is worth noting that this pre-characterization needs to be performed only once, and allows performing aggressive optimizations. Then, such characterizations are used to select the most appropriate configuration for each posit FU and schedule the operations according to the design constraints.

As already mentioned, this work also proposes the use of posit arithmetic in memory (case #3 in Figure 7.2), while keeping the logic of the accelerator in floating-point. Using posits as lossy compressed information storage (in 32-bit precision) can reduce the amount of data transferred up to a factor of 2 with respect to the double-precision accelerator while maintaining decent accuracies. Also, such an approach is compatible with commercial HLS tools that do not support posit arithmetic. When the data in memory is in a different format or precision with respect to the logic of the accelerator, data conversion must be performed before and after performing computations in the accelerator. For that reason, posit-to-float and float-to-posit units are designed to convert input data and output results from the accelerator, respectively. Such FUs are implemented in FloPoCo (as well as the rest of the units of the proposed library), so the same parameterized design is used for different bit lengths. In Bambu, it is possible to map C functions to handwritten HDL modules, which makes this process straightforward. What is more, such modules could be also appended to accelerators generated using floating-point, which makes this a suitable option for those HLS tools without support for posit arithmetic. However, when using this hybrid approach, the corresponding conversions must be considered in the HLS scheduling stage, as data are converted before and after the real computation.

It is apparent that 32-bit posits boast higher accuracy (or precision bits) compared to their 32-bit floating-point counterparts but fall short of the precision provided by 64-bit floats. Consequently, converting from a more accurate format to a less accurate one necessitates meticulous handling of proper rounding to mitigate potential errors. While decreasing the bit count in memory yields evident advantages in accelerator design—enabling higher computing bandwidths and resulting in lower power and energy consumption—it comes at a cost. This trade-off involves the need for additional hardware dedicated to data conversion, and there is the potential for increased error due to rounding.

7.2.4. Limitations

With the aforementioned modifications, the HLS tool Bambu demonstrates the capability to generate posit-based accelerators on par with its floating-point counterparts. However, it must be noted the posit arithmetic introduces distinct operations not found in the IEEE 754, such as fused operations beyond FMA [123]. The current approach does not directly support these operations as it would necessitate the introduction of new operations in the C language. Nevertheless, leveraging the capability of Bambu to map C functions to handwritten HDL modules opens up the possibility of incorporating such functions within the HLS process.

An additional limitation of the proposed approach is that Bambu allows for the simulation of the generated design before initiating synthesis to verify its correctness. Unfortunately,

this functionality relies on the compilation and execution of the C source code, making it unsupported in the case of posit arithmetic.

One potential approach to address these limitations is to leverage arbitrary precision types supported by certain HLS tools. This is the case of MArTo and Vitis HLS. Nonetheless, adopting this solution introduces additional challenges, including the necessity to re-implement algorithms utilizing proprietary libraries and to adjust the source code to integrate such implementations.

These considerations highlight the need for further exploration and potential enhancements in the integration of posit arithmetic within HLS tools like Bambu. Addressing these challenges could unlock the full potential of posit-based accelerators, ensuring both accurate simulation and seamless incorporation of posit-specific operations into the HLS workflow.

7.3. Hardware evaluation

While previous works have already analyzed the hardware cost of posit arithmetic in the context of individual arithmetic operations, there is not much information about how all these pieces fit together. In this section, an analysis was conducted to assess the impact of various arithmetic formats on the design of hardware accelerators for real applications.

7.3.1. Experimental setup

The effects that each of the schemes depicted in Figure 7.2 has in terms of hardware resources and latency when performing HLS, were evaluated. It is also important to have an understanding of the accuracy of each approach. For cases #1, #2 and #3, both 32 and 64-bit precision were considered. When the memories and accelerator logic have different representations, 32-bit posit format (Posit32) is used on the memory side.

To compare the performance of the different approaches, a set of numerical benchmarks from the PolyBench 4.2 suite[1]was employed. This suite includes many common algorithms in fields such as linear algebra, data mining, and image processing. Specifically, the following representative benchmarks were selected:

- **3mm**: Linear algebra kernel that consists of three matrix multiplications arranged in the form $G = ((AB)(CD))$.
- **cholesky**: Cholesky decomposition of a positive-definite matrix A into a lower triangular matrix L such that $A = LL^T$.
- **covariance**: Computes the covariance of N data points, each with M attributes.
- **fdtd-2d**: Simplified finite-difference time-domain method for 2D data. It models electric and magnetic fields based on Maxwell's equations.
- **gemm**: General matrix-matrix product from BLAS, $C = \alpha AB + \beta C$.
- **ludcmp**: LU decomposition followed by forward and backward substitutions to solve a system of linear equations.

[1] https://sourceforge.net/projects/polybench/

PolyBench implements each benchmark in a single file, with some header parameters and a series of compile-time directives, including the data format and dataset size. PolyBench was configured to use 32-bit and 64-bit precision for all the experiments. For numerical accuracy evaluation, the code structure was kept unchanged, including the initialization phase which populated the input data to the algorithms, just modifying the data format and test size for different experiments.

For hardware evaluation, emphasis is placed on the HLS-generated accelerators for the main kernel computation. Consequently, the initialization phase is excluded during synthesis, with the kernel core remaining unaltered. Instead, a wrapper is introduced around the kernel, facilitating the creation of a local copy of the data within the accelerator. This has two consequences. First, when working with arrays, it is faster to access the accelerator's local memory rather than the host machine's memory, which significantly reduces the latency of the accelerators. On the other hand, in case #3, where the memory data is assumed to be in posit format but the computation is performed in floating-point, it is necessary to convert the data to the latter format before operating, and back to the former at the end of the computation. Although this conversion could be applied each time an operation is performed, in cases such as GEMM, where the same piece of data is used for several intermediate computations, this approach has a negative impact on operator latency. However, performing this conversion only once per single datum and storing it in local memory allows for a reduction of the number of conversions (and clock cycles), at the cost of higher hardware resource costs. To have a fair comparison across the proposed approaches, this approach is considered in all the experiments. Performing data conversions at every single operation might reduce the amount of memory required in the FPGA at the cost of increasing the data transfers between the accelerator and the host device, but such a study is out of the scope of this work.

The HLS for each application was executed using Bambu, targeting a Xilinx Artix-7 (XC7A100T-1CSG324C) FPGA device. Notably, the HLS with Bambu incorporated the options `--no-iob` (ensuring that primary ports from the I/O buffers are disconnected, and large arrays can be instantiated in the target device) and `--experimental-setup=VVD` (which provides similar settings for RTL synthesis as the commercial solution Vivado HLS). Under this approach, all objects and internal variables that need to be stored in memory are allocated on BRAMs rather than on external memory. To select a suitable target frequency for the HLS, comprehensive tests were conducted for individual arithmetic operators, targeting different maximum clock frequencies, which allowed us to obtain more details in this regard. Xilinx Vivado 2021.2 was used to perform the logic synthesis for the comparison of hardware resources.

To generate floating-point logic for the accelerators, the option `--flopoco=float` was used, so the FPUs are the ones provided by FloPoCo. However, such units are non-compliant with the IEEE 754 standard: although the memory format is in IEEE 754 format, subnormals are flushed to zero to save resources. This could produce inaccurate results in applications that make use of such small-magnitude data. Also, exceptions are handled in a much simpler way as required by the standard, and just a single rounding mode is implemented (round to nearest, ties to even), rather than the five rounding rules defined in the standard. Therefore, it should be kept in mind that a fully IEEE 754-compliant implementation would incur a much higher overhead than the current one. On the other hand, Bambu was extended with

the option `--flopoco=posit` to allocate posit FUs in the final accelerator. Such units are fully compliant with the current posit standard [51].

Lastly, it is important to mention that all programs in this evaluation were compiled with the `-O3` optimization option, which applies a standard set of optimizations. By adopting this approach, the focus is placed squarely on the capabilities and limitations of the HLS tool itself, without introducing additional custom optimization strategies. This allows for a clear assessment of the baseline performance achievable through compiler optimizations alone.

7.3.2. Accuracy evaluation

Before proceeding with the synthesis evaluation, it is important to ascertain the benefit of each of the encodings proposed in this work in terms of numerical accuracy. To evaluate the error of each approach, software simulations were conducted for each experiment across multiple dataset sizes, ranging from MINI to LARGE. The error is computed by the Frobenius norm[2] against the result obtained with an extended precision format. Such a metric becomes useful when comparing the precision of different arithmetic formats, as it effectively measures how much two simulations deviate from each other by penalizing large errors and giving less importance to minor differences. Indeed, it can be used for either scalars or matrices and vectors, which is the case in the PolyBench applications.

To obtain these metrics, the results of the three cases depicted in Figure 7.2 (under both 32 and 64-bit precision) were compared to the same algorithm computed using the double extended 80-bit format present in x86 processors. The relative error results (with respect to the norm of the baseline) are shown in Figure 7.4. Note the logarithmic scale on the Y-axis. The trend in every benchmark is a significantly lower error when using posit numbers. This is up around one order of magnitude for 32 bits, and between two and three orders less in the case of 64 bits, depending on the benchmark.

Case #3, which mixes both arithmetic formats reveals that using Posit32 in memory rather than Float64 and performing computations in the former precision can be a useful option when posit circuitry is not available, and provides less error than using Float32 and even Posit32. On the other hand, results show that, from the point of view of accuracy, there is no benefit in performing computations with a less accurate format than the stored data.

After characterizing the error of the various proposed formats and approaches, the subsequent step involves evaluating the performance of specific accelerators using HLS.

7.3.3. Operation-level evaluation

Prior to the HLS of the PolyBench applications, a synthesis evaluation of the basic arithmetic operators has been conducted as the initial step in the assessment of the FPGA implementation results. This evaluation aimed to provide a more fine-grained and detailed analysis of the outcomes, isolating as much as possible the library of posit arithmetic operators from the rest of the HLS tool. By focusing on individual operations within the hardware design, in-depth insights can be gained into the performance, efficiency, and

[2]The Frobenius norm is an extension of the Euclidean or element-wise 2-norm, i.e., it is computed as the square root of the sum of the square of all the elements, where each element is the computed difference from the baseline.

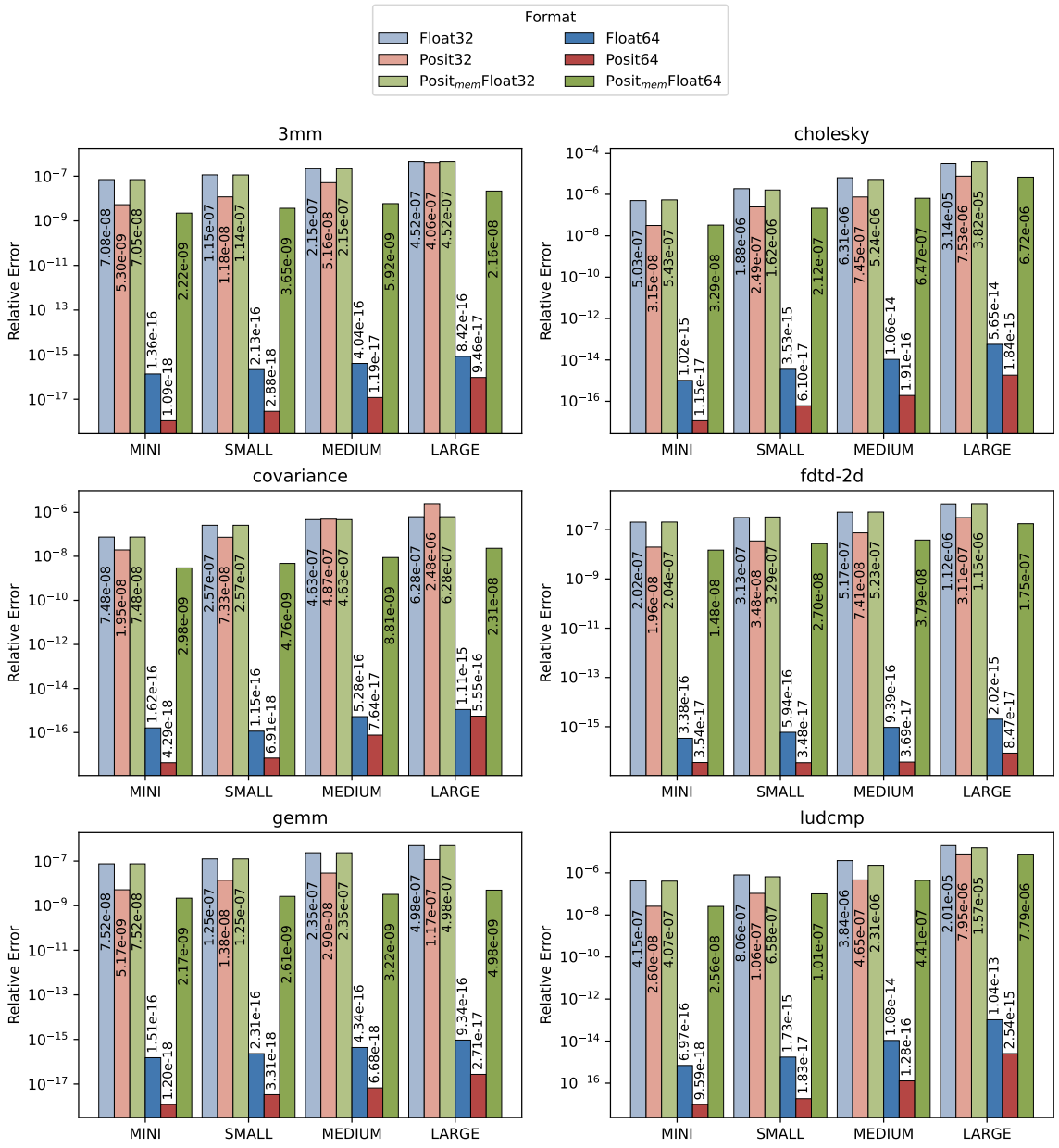


Figure 7.4: PolyBench benchmarks error comparison for different number formats.

potential bottlenecks at a granular level. The findings and observations obtained from this operation-level evaluation will guide the subsequent evaluation of complete applications.

FPGA synthesis reports usage of lookup tables (LUTs), flip-flops (FFs) and digital signal processors (DSPs). The synthesis results for each arithmetic unit, as well as the clock cycles obtained by the RTL simulation, are reported in Figure 7.5. Posit adders require about 1.5× hardware resources (LUTs and FFs) than the corresponding float units, while this overhead is between 2× and 6× for the rest of the operators. Nonetheless, the amount of resources

required by Posit32 is always fewer than by Float64 units. Regarding the frequency, all the FUs except the Float64 multiplier satisfy the timing target conditions up to 150 MHz. For a target frequency of 200 MHz a few operators violate the timing constraint, and none of them reach 300 MHz. Therefore, 150 MHz is a clear candidate as the target frequency for the HLS of complex applications. Finally, it must be noted how the iterative algorithm used for division and square root has a direct impact on the latency of such units as the target frequency increases, especially for the Posit64 format.

In addition to the resources shown in Figure 7.5, HLS results show that, independently of the target frequency, the 32 and 64-bit floating-point multipliers require 2 and 9 DSPs, respectively, and the corresponding posit multipliers make use of 2 and 12 DSPs, respectively. Also, the design of the floating-point division includes a table for fast computation, which requires 7 and 14 extra BRAMs when synthesizing the 32 and 64-bit designs, respectively.

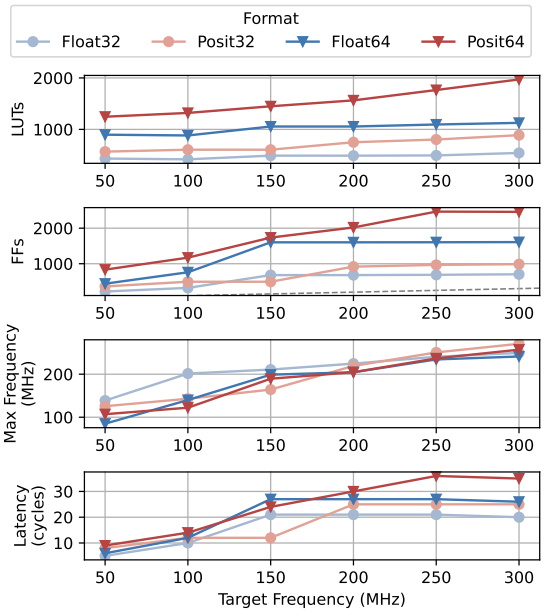
Case #3 proposed in this work considers input data to be in Posit32 format, while the computation done within the accelerator is in floating-point. For this hybrid scenario, input and output data conversion must be done, so it is important to evaluate separately the hardware overhead of such conversions. Synthesis results are reported in Figure 7.6. As can be seen, the library of posit converters can be synthesized with Bambu up to 300 MHz seamlessly. In addition, the units exhibit quite a low latency (3 cycles or less) when targeting up to 150 MHz.

Comparison with previous works

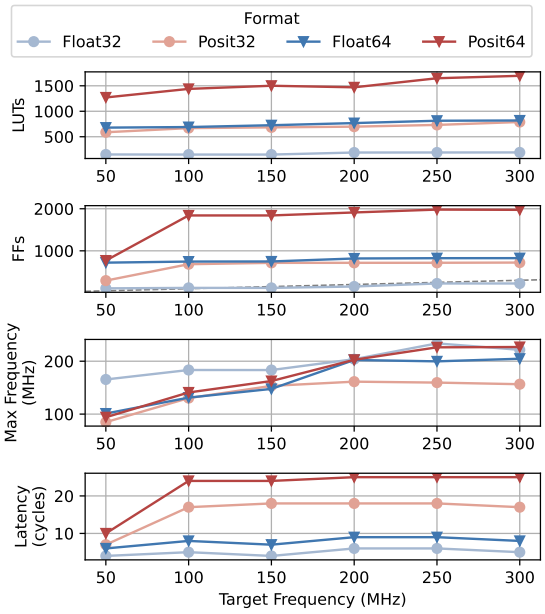
To the best of our knowledge, MArTo [161] is the only library that provides posit arithmetic support for HLS up to date. It consists of a templatized C++ library compliant with Vitis HLS. It currently offers standalone posit adders, subtracters, and multipliers, but does not support operations such as division, square root, or comparison, unlike the proposed work. Vitis HLS 2021.2 is employed to perform HLS of the designs, targeting the same device (Artix-7) and frequency (150 MHz). The RTL synthesis is performed by Xilinx Vivado with the default configuration. Although the focus of this work is posit arithmetic, floating-point designs are generated as well for better comparison and understanding of the results. It is worth mentioning that while MArTo provides arithmetic units designs for posit arithmetic, the floating-point units are from the proprietary Xilinx Floating-Point Operator IP.

In the same manner as with Bambu, experiments are firstly conducted for single arithmetic operators. Note that, in this case, not only a different posit operator library is being used, but also the HLS tool changes. Therefore, such an operator-level evaluation allows to isolate as much as possible the differences between different HLS tools and to be able to analyze in more detail the differences between the two posit operator libraries. The synthesis results for posit addition and multiplication units provided by MArTo under different target frequencies are depicted in Figure 7.7.

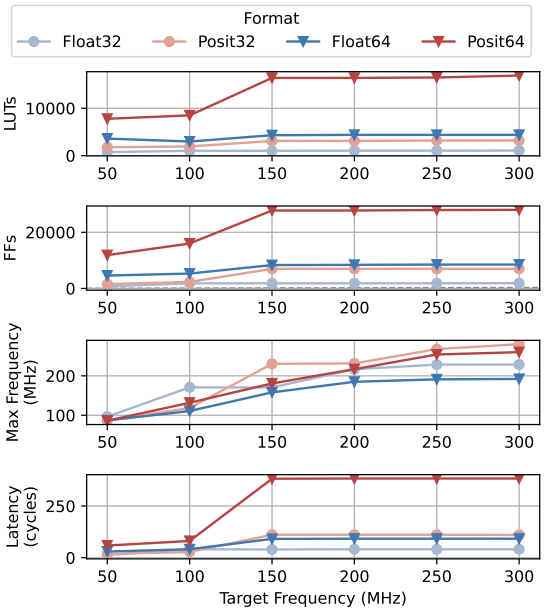
In contrast with the proposed library of posit operators, in this case, the Posit32 units require even more area than the corresponding Float64 operators. When compared with Figure 7.5, on average the 32-bit floating-point adders generated with FloPoCo/Bambu require up to 1.5× more LUTs and 1.33× more FFs than the ones generated with MArTo/Vitis (respectively, 1.26× and 1.36× for 64-bit adders). However, the situation is very different with respect to posit adders. The units generated with the latter tools require on average



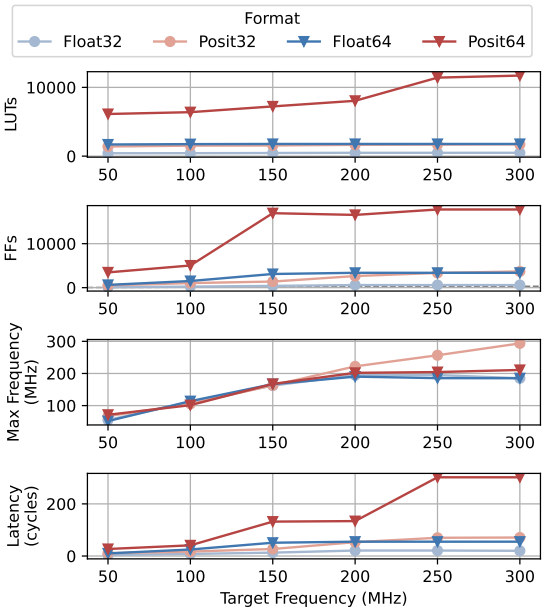
(a) Addition.



(b) Multiplication.



(c) Division.



(d) Square root.

Figure 7.5: Bambu HLS results for basic arithmetic operators.

4.14× more LUTs and 2.66× more FFs than the proposed 32-bit units (respectively, 3.7× and 2.25× for 64-bit posit adders). The same behavior is observed in the multiplier units, which also make use of DSP units, again with an overhead for those multipliers defined at the MArTo library.

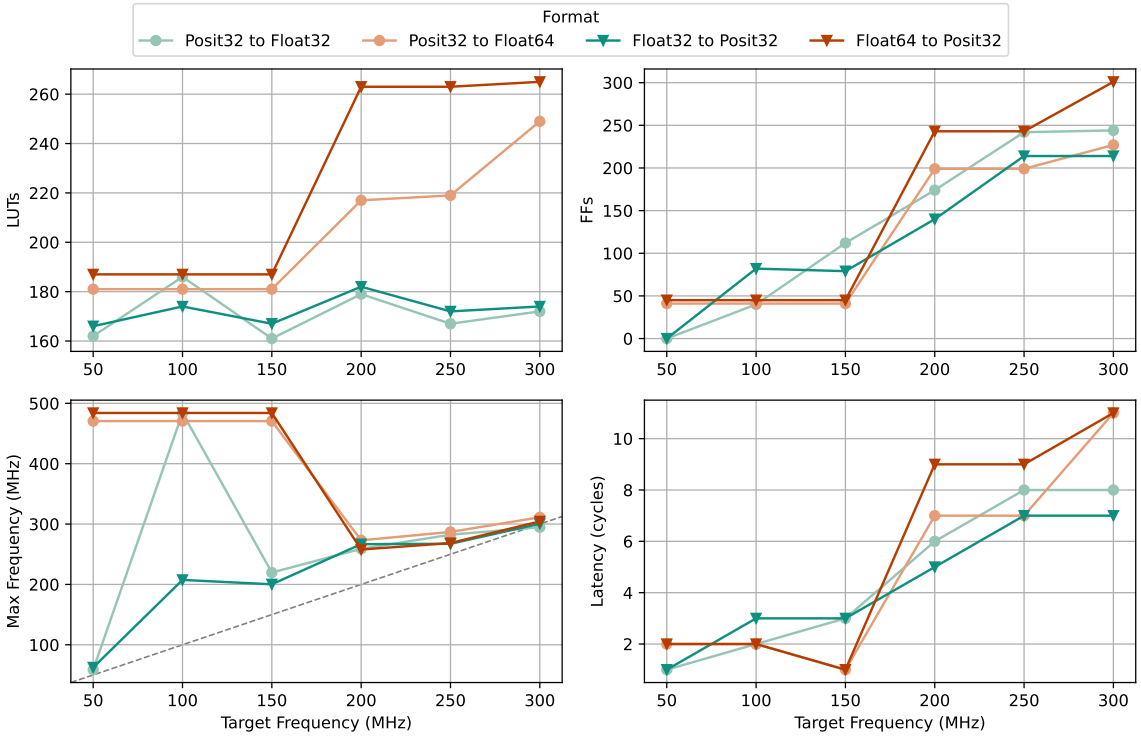


Figure 7.6: HLS results for posit-float converters.

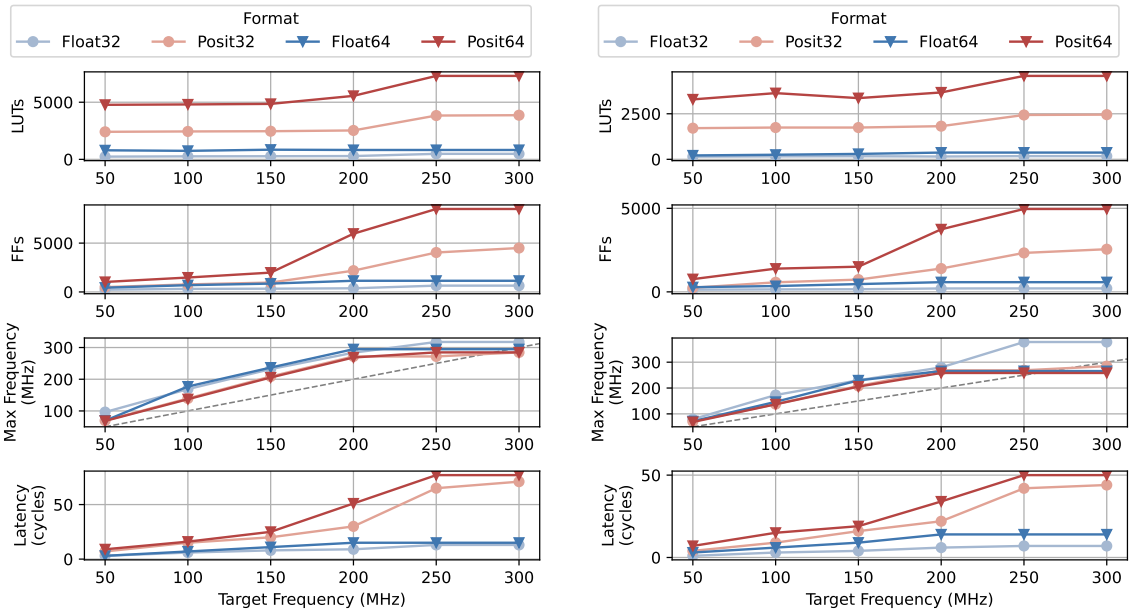
If comparing the latency of the operators, noticeable differences are found between the posit libraries. The 32-bit adders from MArTo require 1.74× more cycles, on average, than the proposed units (1.54× more for 64-bit adders), and for the 32-bit multipliers, the overhead is 1.36× more cycles, on average (1.25× for the 64-bit units).

Overall, it can be concluded that at the operation level, the library of posit operators for HLS proposed in this work presents more efficient units in terms of both area and performance than those proposed in previous works.

7.3.4. Application-level evaluation

By considering the behavior of a diverse range of applications, we aim to assess the suitability and efficacy of utilizing floating-point and posit formats in accelerators for real-world applications. The chosen PolyBench programs were synthesized with Bambu. This evaluation leverages the frequency results obtained from the operation-level analysis. In consideration of the above results, a target frequency of 150 MHz has been set for all experiments.

Synthesis results for the multiple benchmarks under the MINI dataset size are depicted in Figure 7.8. Larger sizes mainly affected the latency of the accelerators, in the same proportion for all the cases. Although each benchmark yields different results, similar patterns can be found in each of the metrics. In addition, the geometric mean across the proposed benchmarks is included for the sake of comprehension.



(a) Addition.

(b) Multiplication.

Figure 7.7: Vitis HLS results for basic arithmetic MArTo [161] operators.

As can be seen, when comparing LUTs and FFs between cases #1 and #2, there are two patterns clearly distinguishable. For the applications that only require addition and multiplication, the posit overhead is slightly higher than for the floating-point case, between 1.06× and 1.45×. On the other hand, for cholesky, covariance and ludcmp benchmarks, the overhead of posit accelerators ranges from 1.58× to 3.96×. This corresponds to the applications using division and square root. As shown in Figure 7.5, those posit operators required much more hardware resources in comparison with the floating-point ones, especially in the case of Posit64. Such a discrepancy suggests that the posit division and square root operators are not optimized as well as the corresponding floating-point units. Overall, the Posit32 accelerators have an average overhead of 1.46× and 1.72× in terms of LUTs and FFs, respectively, and similarly, for the 64-bit case, the LUTs and FFs overhead is 1.73× and 2.12×, respectively.

As for the use of DSPs, all benchmarks show exactly the same result, which in turn is the same as the one requiring a single multiplier unit, as mentioned above. This reveals that the amount of floating-point or posit multipliers is constant (and equal to one) across the experiments. This is in line with the behavior of Bambu, which allocates a single instance of each function and/or floating-point operation.

For BRAMs evaluation, again two different patterns can be distinguished between the applications that use division and those that do not. As has been mentioned, floating-point division units require 7 and 14 extra BRAMs for 32 and 64-bit designs, respectively. For the rest of the applications, the amount of BRAMs required by posit-based accelerators is the same as that used by floating-point kernels since they are encoded with the same number of bits.

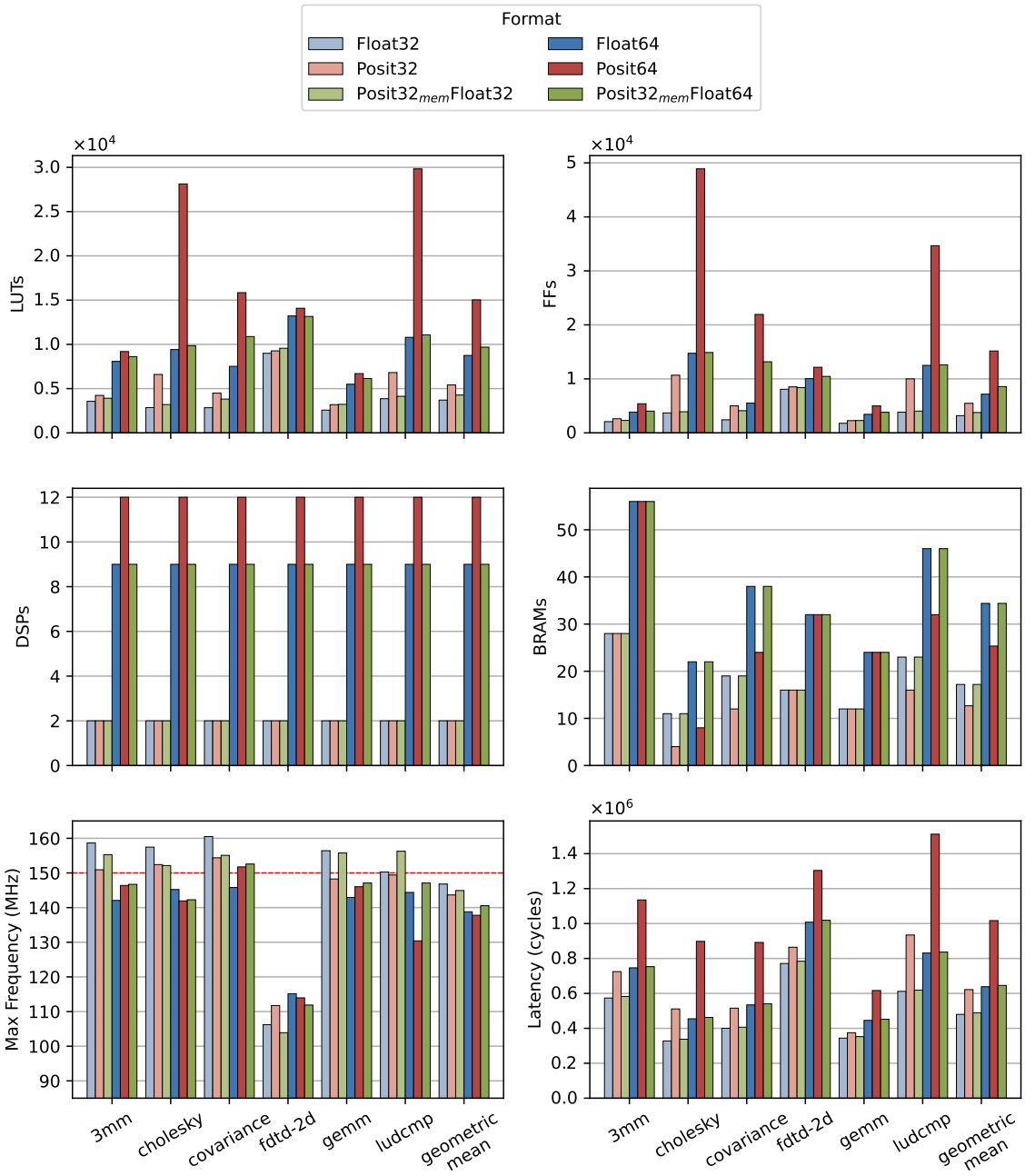


Figure 7.8: Bambu HLS results for PolyBench benchmarks.

Although the target frequency is set at 150 MHz, as shown in Figure 7.8 this cannot be satisfied by all benchmarks. In particular, only 32-bit benchmarks satisfy this timing condition in most cases. The other time-related metric, latency, presents more differences between number formats. Posit32 requires between 1.09× up to 1.56× more cycles than Float32 accelerators (1.30× more on average), while the overhead for 64-bit formats ranges

from 1.29× to 1.98× (1.59× more on average). Again, the higher increments are on the benchmarks that make use of division and square root operators.

Case #3 is worth mentioning separately. The amount of DSPs and BRAMs is exactly the same as their floating-point counterpart, since the arithmetic units employed in this hybrid scenario are the same as for case #1. However, as depicted in Figure 7.6, such conversions introduce certain hardware overhead (about 10-15% more LUTs for 32-64 bits, and about 18-19% more FFs for 32-64 bits, respectively). In addition, the posit-float conversion also requires some extra clock cycles, but this is just 1 or 2 cycles per conversion, which results in less than 3% of overhead when considering any of the PolyBench programs.

Comparison with previous works

For a further comparison, MArTo was used to generate posit-based accelerators for the aforementioned PolyBench applications supported by this tool, i.e., excluding cholesky, covariance and ludcmp because they use division and square root, which is not supported. Within MArTo, posit multiplications return an internal data format that must be converted back into posit, and subtractions must be re-written as additions with negative numbers, among other changes that must be done in the source code. Again, Xilinx Vitis was used to perform the HLS of the designs.

As can be seen in Figure 7.9, the designs obtained with Vitis HLS lead to very different synthesis results than the ones generated with Bambu. At first glance, it can be seen that in this scenario the Posit32 accelerators generally require more LUTs (and in some cases more FFs) than the Float64 accelerators, and also exhibit higher latency than the latter. In contrast to the previous case, the synthesis results for Vitis vary considerably among the benchmarks. Although there are important differences in the floating-point case, let us focus on the comparison in the posit designs between the two HLS tools. Comparing the Vitis accelerators with the one generated by Bambu, it can be seen that, except for the fdtd-2d application (where the posit designs do not use any DSPs), the designs generated by Vitis generally require more hardware resources (between 2.6× and 8.8× more LUTs, 2.1× and 16.3× more FFs, and 0.91× and 8.5× more DSPs, depending on the application and format). On the other hand, the number of BRAMs used for Vitis designs is half that of Bambu designs.

This hardware overhead has certain advantages in terms of frequency, as the designs generated with MArTo and Vitis meet the target frequency in all the benchmarks, even for the 64-bit scenario. The latency is also lower than for the Bambu accelerators. This may seem to contradict the conclusions of the previous operator-level comparison. However, there are two aspects that need to be highlighted here. First, as can be seen, not only is the latency of the posit-based accelerators lower, but so is that of the floating-point accelerators. Also, according to the official Xilinx documentation [172], *the default operation of Vitis HLS is to first maximize performance*, while Bambu generates a single instance of each function or floating-point operation (even if there are multiple call points in the program), thus providing more area-efficient designs.

The reason for this discrepancy in results is therefore the scheduling and binding algorithms used by the different HLS tools, rather than differences in RTL design. However, it should be remembered that the purpose of this study is to analyze the impact of the different encodings, rather than to compare the performance of different HLS tools. In this

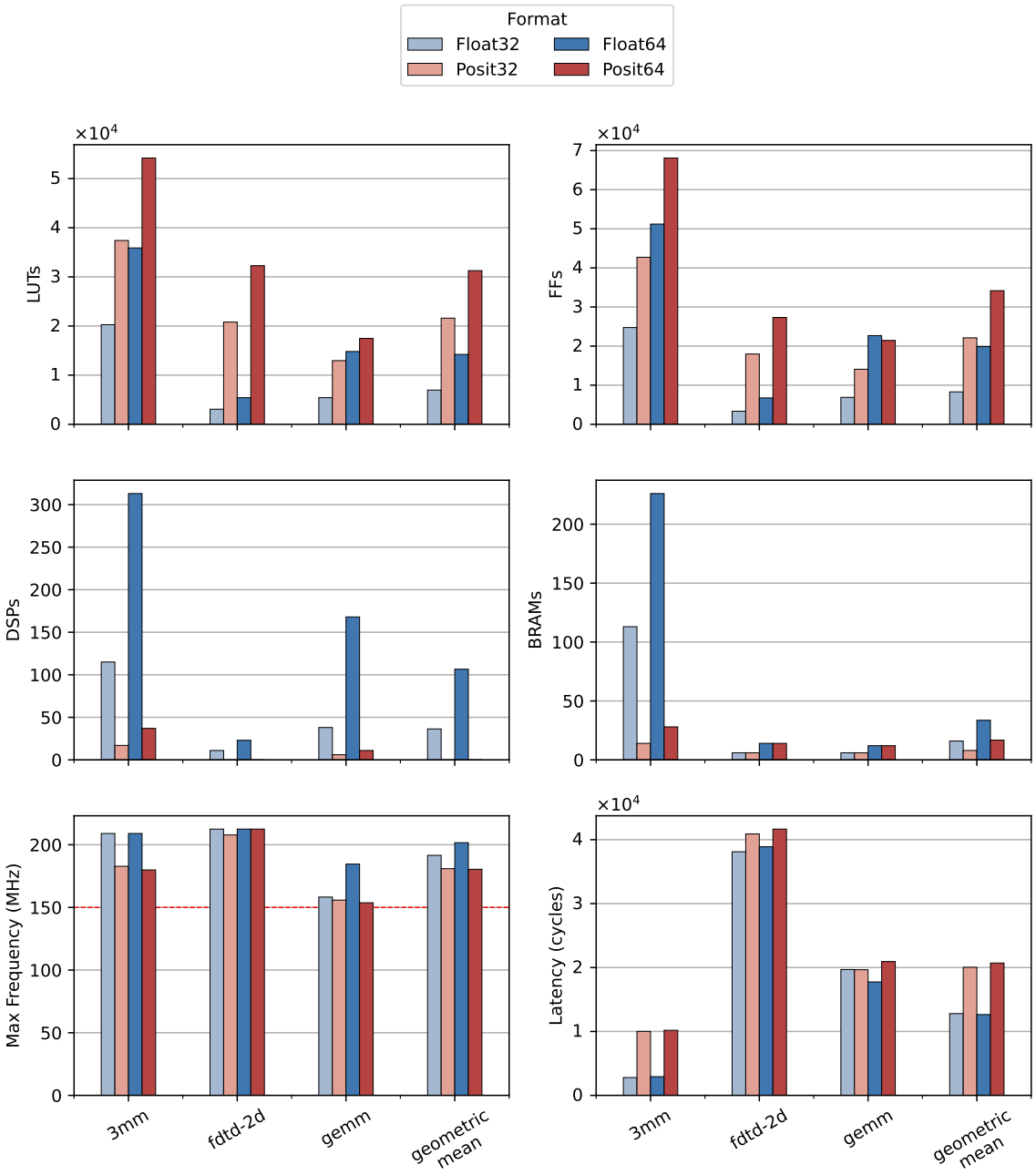


Figure 7.9: Vitis HLS results for PolyBench benchmarks.

sense, the results obtained with Vitis for the PolyBench application follow the same patterns observed in the previous analysis for the basic arithmetic operations, i.e., the use of posit arithmetic requires higher latency and more hardware resources than even Float64.

Despite the potential benefits of Vitis HLS, the evaluations indicate that the proposed posit library leads to higher performance and lower area under the same HLS tool. Specif-

ically, the conducted experiments show that the Posit32 (respectively, Posit64) designs produced by MArTo require, on average, 1.57× (respectively, 1.64×) more clock cycles than the corresponding float designs. However, the same set of Posit32 (respectively, Posit64) accelerators proposed in this work exhibit a lower latency overhead of 1.15× (respectively, 1.40×) compared to the float designs. Also, in terms of area requirements, the posit designs from MArTo present a higher increment factor with respect to the corresponding designs in floating-point. For example, the Posit32 designs from previous work use on average 3.1× more LUTs and 2.67× more FFs than the Float32 designs, while the increment in the proposed work is just of 1.15× and 1.19× for LUTs and FFs, respectively. Similar figures are obtained for 64-bit precision.

7.4. Conclusions

Building upon the operators introduced in Chapter 5, this chapter outlines a comprehensive HLS flow that incorporates support for posit arithmetic. This enables the creation of custom accelerators harnessing the superior accuracy inherent in the posit arithmetic format. Furthermore, a novel memory customization approach is introduced, employing posit numbers in memory while maintaining the logic of accelerators in floating-point. This approach capitalizes on posit benefits in scenarios where support for this alternative format is lacking. The implementation leverages open-source tools, utilizing FloPoCo for the RTL library of posit operators and Bambu for HLS. The outcome of this work is publicly accessible through an open-source GitHub repository[3].

To showcase the capabilities of the proposed workflow and analyze the impact of different arithmetic formats, floating-point and posit kernels for computer-intensive applications were deployed. Results consistently demonstrate that posit arithmetic outperforms classical floating-point notations in terms of accuracy, all without increasing memory usage or requiring additional bits for numerical representation. This finding is particularly valuable in scenarios where memory is a critical resource, or where the cost of expanding memory outweighs the benefits of improved accuracy.

However, the extra accuracy comes with associated costs. Specifically, the 32-bit posit-based accelerators require about 1.46× more LUTs, 1.73× more FFs, and 1.30× more clock cycles than their corresponding floating-point counterparts.

In comparing the proposed HLS flow with previous works relying on commercial HLS tools, the FPGA synthesis outcomes reveal that the posit designs exhibit reduced overhead compared to those from earlier studies. Overall, these findings emphasize a clear trade-off between precision and hardware resources in posit arithmetic. Consequently, the decision to use and implement a specific format should be left to the discretion of hardware designers, considering their specific requirements and constraints.

Finally, in the three chapters of this part, the bottom-up process of designing fundamental hardware components for posit arithmetic has been systematically demonstrated, culminating in the generation of hardware DSAs centered around this alternative format. Despite the evident advantages in accuracy showcased in various evaluations, it is important to acknowledge the associated trade-off: a higher hardware overhead. The field of research on posit arithmetic is still in its early stages. While ongoing efforts continue to uncover new

[3] https://github.com/artecs-group/posit-hls

benefits and innovative ways to leverage and design with this format, it remains more costly than its floating-point counterpart. A potential avenue to address this challenge is exploring alternative representations and operations for posits that could mitigate implementation costs, even at the expense of some precision inherent in the posit format. The upcoming chapters delve into the extension of these concepts, considering novel arithmetic formats built upon the standard posit. This exploration reflects a dynamic approach, seeking ways to strike a balance between implementation costs and precision, as the research community navigates the evolving landscape of posit arithmetic.

Part III

Alternative Arithmetic Formats Based on Posit

8

Logarithmic-approximate posit arithmetic

As elucidated in Chapter 5, the variable-length field introduced by the posit format inherently introduces a hardware overhead. This concern becomes particularly notorious when addressing energy-efficient and/or high-performance designs.

Approximate computing emerged as a paradigm that efficiently addresses this challenge by allowing for energy savings and performance enhancement through the relaxation of accuracy requirements for error-tolerant applications [53, 111]. Given the existence of human perceptual limitations and redundant input data, numerous applications inherently tolerate errors, including multimedia signal processing, machine learning, and pattern recognition [39, 21]. The key to approximate computing is to carefully use estimations only on non-critical information. Attempting to estimate important data (such as control operations) may lead to disastrous outcomes, like program crashes or erroneous output.

Several research efforts have delved into approximation techniques, exploring approaches such as lower-bit precision or inexact multipliers in both fixed-point [100, 78] or floating-point arithmetic [68, 18]. These techniques aim to reduce the power consumption of arithmetic computations while minimizing accuracy degradation in applications like image filtering or deep learning.

In this regard, a bold step is taken in this chapter by suggesting the integration of approximated posit units. More precisely, the objectives pursued in this chapter are:

1. Algorithms for performing the costly operations of posit multiplication, division, and square root in an approximate manner are proposed. Specifically, logarithmic approximations of the previously mentioned posit operations are proposed to reduce the hardware cost of such operators. Additionally, their maximum expected error is analyzed.

2. It is demonstrated how the proposed approximations can be applied to error-tolerant applications. To illustrate this, multiple use cases from signal processing, machine learning, and deep learning are computed with the approximate posit operators.

3. The proposed algorithms are implemented in hardware to evaluate their cost and demonstrate the resource reduction that can be obtained in comparison with exact posit units.

Section 8.1 presents the basic properties on which the proposed logarithmic approximation relies. Sections 8.2, 8.3 and 8.4 explains how such approximation is adapted in the case of multiplication, division, and square root, and hardware implementation schemes are presented for each operation. The error of each approximate operation is studied in Section 8.5. In Section 8.6, the proposed posit approximated units are evaluated under multiple error-tolerant applications. The proposed algorithms are implemented and evaluated through ASIC synthesis in Section 8.7, as well as compared with the exact posit operators. Finally, Section 8.8 shows some conclusions and final remarks.

8.1. Approximating posits in the logarithmic domain

Typical number formats like fixed-point or floating-point, as well as posit, follow the rules within the linear domain, where arithmetic operations are computed as we are used to. However, in the logarithmic domain, operations like multiplication and division can be performed with simple addition and subtraction, respectively.

To approximate posit numbers in the logarithmic domain, we will follow the approach proposed by Mitchell in [110]. He proposes using an approximation in the conversion process between the linear domain and the logarithmic domain, so some operations can be performed in the latter domain with very few hardware resources, but with some error in the result introduced by the approximated conversion.

The approximated conversion relies on the property of logarithms depicted by Equation (8.1),

$$\log_2(1+x) \approx x, \text{ for } 0 \leqslant x \leqslant 1. \tag{8.1}$$

To take advantage of this property in posit arithmetic, some preliminary considerations must be taken into account:

- The fraction of f posit numbers must be within range $[0, 1)$. This is met for all posit numbers apart from the exceptions.
- The hidden bit of the fraction must be always one. Considering a negative hidden bit might cause issues when taking logarithms. Therefore, the sign-magnitude notation of posit numbers will be considered throughout this chapter (for more information about different posit notations, see Appendix A).
- The sign bit can be taken separately, so it does not interfere when calculating logarithms. The (absolute) value can be considered in the logarithmic domain, to which the sign is then added.

Based on these conditions, we proceed to approximate posit numbers in the logarithmic domain. Recall that the value of a non-exceptional posit number X is given by Equation (1.4). Therefore, taking logarithms on both sides, such a posit can be approximated in the logarithmic domain as shown by Equation (8.2):

$$\log_2 X = (-1)^s \times 2^{es} \times r + e + \log_2(1+f) \approx (-1)^s \times 2^{es} \times r + e + f. \tag{8.2}$$

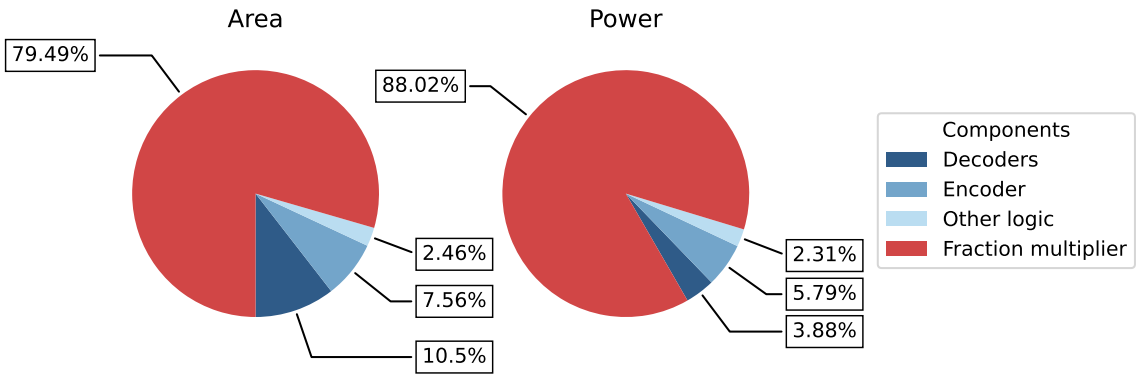


Figure 8.1: Resource distribution of a Posit32 multiplier.

For notational simplicity, the scaling factors formed by the regime and the exponent can be reassembled, so that $2^{es} \times r + e = k$. Hence, $k + f$ is the approximation of the logarithm value of X, where k and f are usually named *characteristic* and *fraction*, respectively [110].

By the aforementioned approximation, arithmetic operations of approximated values can be performed in the logarithmic domain and the result can be obtained back to the linear domain without the need for conversion, as detailed in [110]. The next sections will detail the procedure for performing multiplication, division and square root based on this approximation method, as well as propose implementation algorithms for the different posit logarithm-approximate units (PLAUs).

8.2. Posit logarithmic approximate multiplication

Multiplication in hardware can be computationally expensive, particularly when dealing with real numbers, as it necessitates additional steps beyond the multiplication of fixed-point values. As illustrated in Figure 8.1, specifically tailored for the posit format, hardware multiplication constitutes the primary contributor to power consumption. To mitigate the hardware complexity associated with multiplication, an alternative approach known as logarithm multiplication has been proposed in the literature for either integer [78, 77, 140] and floating-point arithmetic [18, 16]. This approach eliminates the need for hardware multipliers by approximating multiplication as fixed-point addition. This section introduces the posit logarithm approximate multiplier and presents an implementation algorithm. The operators explained in subsequent sections, namely division and square root, follow a similar rationale.

Let us consider A and B to be two posit numbers in the sign-magnitude form $(-1)^s \times 2^k \times (1 + f)$. The exact result of the product of such numbers, $P = A \times B$, is given by Equation (8.3),

$$P_{exact} = (-1)^{s_A+s_B} \times 2^{k_A+k_B} \times (1 + f_A) \times (1 + f_B). \tag{8.3}$$

Notice that, in order to properly accommodate the result into a posit number, a correction must be applied if the product of fractions is greater or equal to one.

Recall that in posit arithmetic there are no special cases to be taken care of, as the denormal numbers in the case of IEEE 754 formats, a single rounding mode, i.e. round to nearest even, and unique representations for zero and infinite values. Thus, Equation (8.3) allows obtaining the product of any two posit numbers (except for the product by zero or NaR, which is trivial). More details on the implementation of posit exact multiplication can be found in Section 5.4.

Once the exact posit multiplication is explained, the insights on how to approximate this operation by taking advantage of the logarithmic multiplication will be described in detail. Firstly, it is worth noting that in the multiplication operation, the computation of the sign is independent of the computation of the other fields. Therefore, let us focus on positive posit numbers from now on (under the sign-magnitude approach we are considering in this chapter, computation with negative numbers just differs on the sign).

Converting numbers to the logarithmic domain allows to compute the multiplication as the addition of fixed-point numbers. In such a case, the product is obtained by taking the anti-logarithm of the sum of the logarithms of both operands. However, using the approximation of the logarithm given by Equation (8.1) eliminates the need for the anti-logarithm step, as depicted in Equation (8.4):

$$P_{approx} = \begin{cases} (-1)^{s_A+s_B} \times 2^{k_A+k_B} \times (1 + f_A + f_B) & \text{if } f_A + f_B < 1, \\ (-1)^{s_A+s_B} \times 2^{k_A+k_B+1} \times (f_A + f_B) & \text{if } f_A + f_B \geqslant 1. \end{cases} \tag{8.4}$$

Notice that the sign computation is independent of the value, and to leverage logarithmic approximation the addition of fractions must be in the range $[0, 1)$, so a correction needs to be applied (if the addition of fractions is greater or equal to one) to pack the result back to a posit number.

Now, let us focus on the hardware implementation of the proposed logarithm approximate posit multiplier. Firstly, notice that the one that is added to the fractions in the first case of Equation (8.4) corresponds to the implicit bit of the resulting posit number C. Thus, the resulting fraction (f_C) is always obtained from the addition of the input fractions. The second case of Equation (8.4) increments the scaling factor by one, as the sum of fractions is too large ($\geqslant 1$). However, note that in such a case the carry bit from the resulting fraction field can be directly added as a carry-in to the exponent addition, that is, the resulting scaling factor is corrected as $k_A + k_B + 1$. This technique can be implemented in hardware with the scheme shown in Figure 8.2. Note that, for such implementation, the implicit 1 bit of the fraction does not need to be considered.

8.3. Posit logarithmic approximate division

From a mathematical point of view, performing the quotient of two posit numbers, $Q = A/B$, $B \neq 0$, is analogous to the multiplication process. The exact result is obtained by dividing the scale factors (subtracting the exponents) and the fractions with the implicit bit, as depicted in Equation (8.5):

$$Q_{exact} = (-1)^{s_A-s_B} \times 2^{k_A-k_B} \times (1 + f_A)/(1 + f_B). \tag{8.5}$$

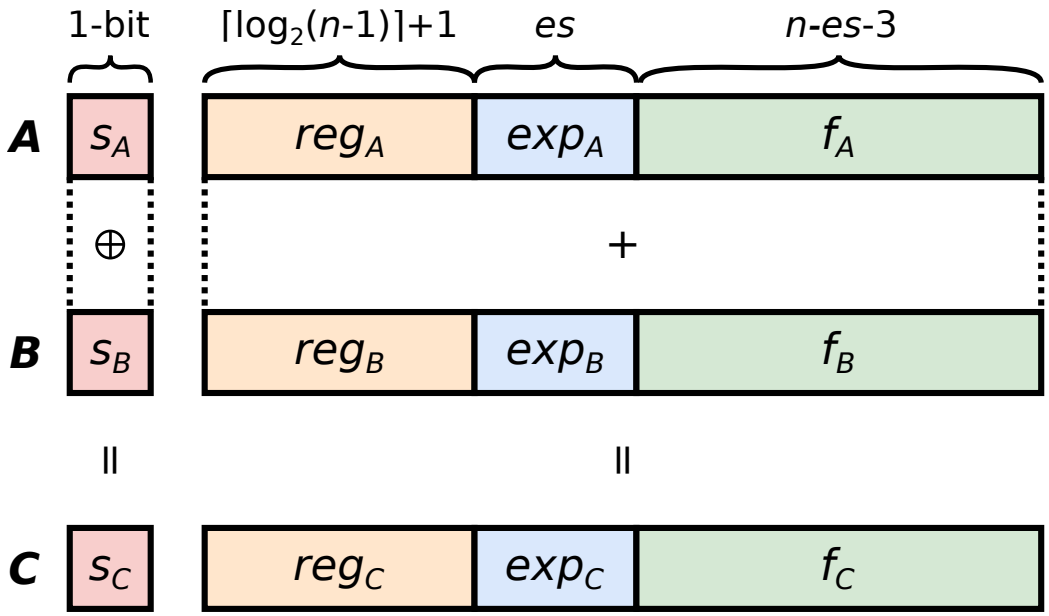


Figure 8.2: Hardware implementation scheme for the proposed logarithm approximate posit multiplication.

However, implementing the division of fixed-point numbers in hardware is extremely costly [38]. As shown in Section 5.5, it usually consists of successive subtractions (and even multiplications, to accelerate the computation).

On the other hand, the logarithm of a quotient is equal to the difference between the logs of numerator and denominator. Using this property, the approximated quotient is given by Equation (8.6).

$$Q_{approx} = \begin{cases} (-1)^{s_A - s_B} \times 2^{k_A - k_B} \times (1 + f_A - f_B) & \text{if } f_A - f_B \geqslant 0, \\ (-1)^{s_A - s_B} \times 2^{k_A - k_B - 1} \times (2 + f_A - f_B) & \text{if } f_A - f_B < 0. \end{cases} \tag{8.6}$$

As can be seen, the only difference concerning the approximate multiplier is the use of subtraction instead of addition. Therefore, the hardware implementation of the proposed logarithm approximate posit divider is essentially the same as shown in Figure 8.2, simply replacing the addition with subtraction in the scheme. Again, the proposed scheme allows handling both conditions of Equation (8.6) simultaneously: when $f_A - f_B < 0$, a bit is subtracted from the subsequent fields (exponent and regime) and, in such a case, it is necessary adding 2 to correct the subtraction $f_A - f_B$, which is negative. But this just affects the implicit fraction bit, while remains unchanged the explicit part of the fraction.

Finally, notice how this approach can reduce the hardware resources and delays tremendously, since the multiple iterations needed to compute the division of fractions precisely are replaced by a single subtraction.

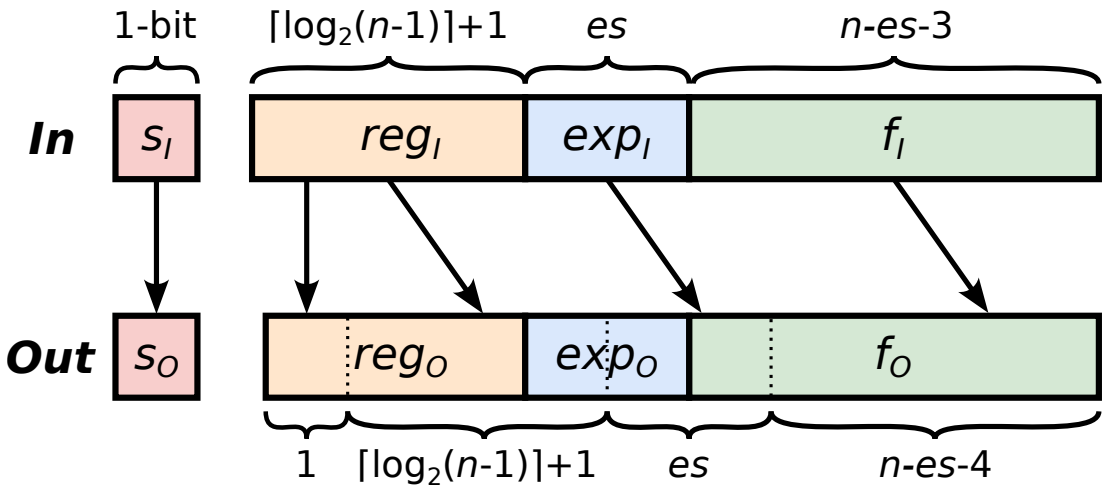


Figure 8.3: Hardware implementation scheme for the proposed logarithm approximate posit square root.

8.4. Posit logarithmic approximate square root

The exact square root of a non-negative positive number is mathematically defined by Equation (8.7),

$$R_{exact} = \sqrt{2^k \times (1+f)} = 2^{k/2} \times \sqrt{1+f}. \tag{8.7}$$

Similar to division, the computation of the square root of a fixed-point fraction demands substantial hardware resources [38]. However, leveraging the conversion into the logarithmic domain simplifies the calculation of this operation. It is important to note that square roots can be expressed as powers with an exponent of $1/2$. Furthermore, the logarithm of a number raised to a power is equal to the exponent multiplied by the logarithm of the base number. Therefore, an approximation of the square root of a posit can be obtained by using Equation (8.8), where the logarithm of the square root is approximated as half of the fraction,

$$R_{approx} = 2^{k/2} \times (1 + f/2). \tag{8.8}$$

It is worth mentioning that both Equation (8.7) and Equation (8.8) involve dividing specific values by 2. However, when implementing these operations in software, multiplying and dividing by powers of 2 can be accomplished by shifting the binary representation of the values. Specifically, division by 2 can be achieved by right-shifting one bit (while preserving the regime sign). Consequently, the hardware implementation of the posit logarithmic approximation for square root, as depicted in Figure 8.3, can employ binary shifting operations. Notably, the LSB of the fraction can be utilized as a rounding bit, thereby reducing the overall error.

8.5. Approximation error

As mentioned earlier, the approximation of logarithms of posit numbers introduces a small error in the computation of values in the linear domain. To analyze this error, the

relative error formula defined in Equation (8.9) can be employed:

$$E_{rel} = \frac{approx - exact}{exact} = \frac{approx}{exact} - 1. \tag{8.9}$$

This formula quantifies the error that may arise when arithmetic operations are performed using the proposed approximation schemes. The resulting error is not only independent of the magnitude of the value but also provides information about whether the approximation overestimates or underestimates the actual value.

Substituting in Equation (8.9) the exact and approximate expressions of Equations (8.3) and (8.4), Equations (8.5) and (8.6), and Equations (8.7) and (8.8), yields error Equations (8.10) to (8.12), respectively.

$$E_{mul} = \begin{cases} \dfrac{1 + f_A + f_B}{(1 + f_A)(1 + f_B)} - 1 & \text{if } f_A + f_B < 1, \\ \dfrac{2(f_A + f_B)}{(1 + f_A)(1 + f_B)} - 1 & \text{if } f_A + f_B \geqslant 1. \end{cases} \tag{8.10}$$

$$E_{div} = \begin{cases} \dfrac{(1 + f_A - f_B)(1 + f_B)}{(1 + f_A)} - 1 & \text{if } f_A - f_B \geqslant 0, \\ \dfrac{(2 + f_A - f_B)(1 + f_B)}{2(1 + f_A)} - 1 & \text{if } f_A - f_B < 0. \end{cases} \tag{8.11}$$

$$E_{sqrt} = \frac{1 + f/2}{\sqrt{1 + f}} - 1. \tag{8.12}$$

Let us now examine the Equations (8.10) to (8.12) to determine their maximum and minimum values. It is worth noting that the relative errors of approximation, denoted as E_{mul}, E_{div}, and E_{sqrt}, are functions that solely rely on the fraction of the respective operands. It is important to recall that the fraction f of any posit number is a value ranging between zero and one.

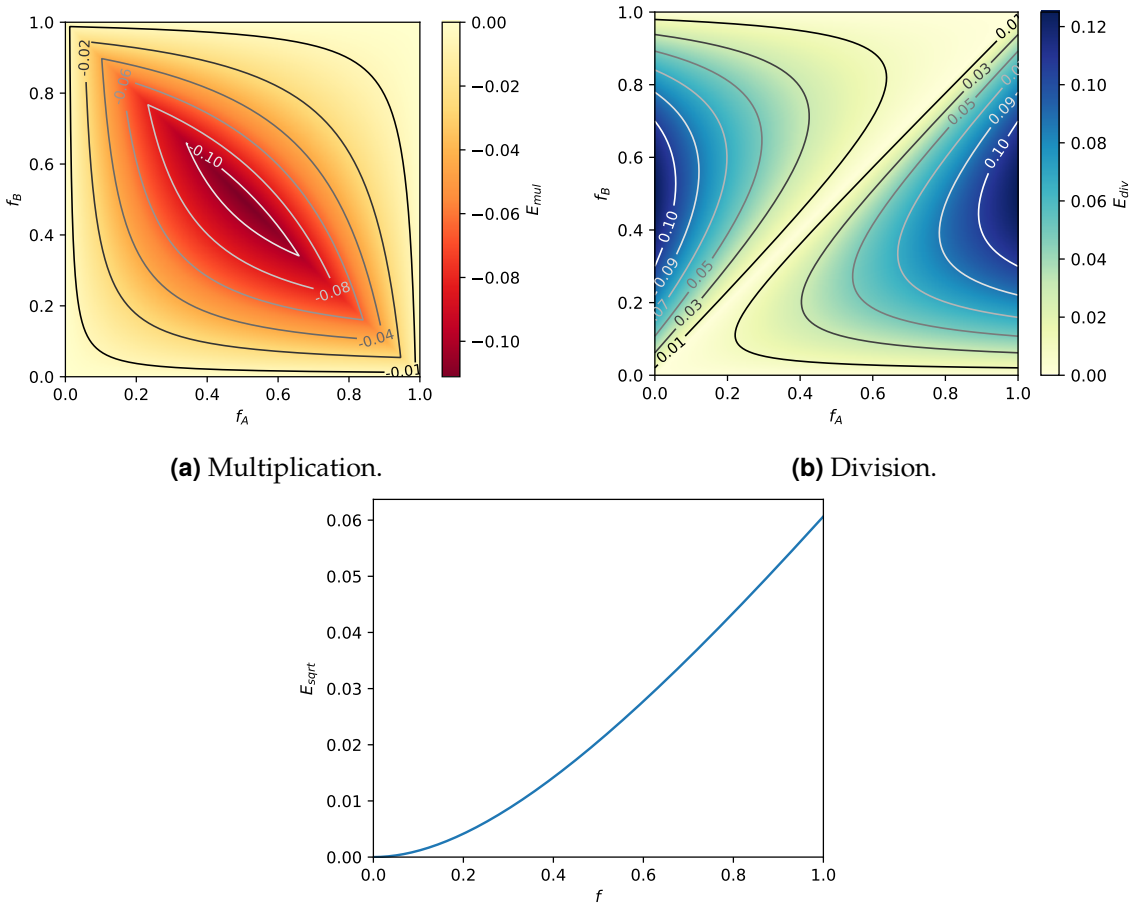
Equations (8.10) to (8.12) can be verified to be continuous and differentiable. By computing the derivatives of these equations, one can determine the critical points of the aforementioned functions. Further details regarding these computations can be found in [110].

Figures 8.4a and 8.4b illustrates the contour map of the error functions E_{mul} an E_{div} according to the fractions f_A and f_B. Additionally, Figure 8.4c depicts the plot of the error function E_{sqrt} with respect to the fraction f.

The multiplication error function is non-positive, and its minimum is found at $f_A = f_B = 1/2$. Substituting these values into Equation (8.10) leads to $E_{mul} = -1/9$, or -11.1%. The maximum error in multiplication is zero and occurs when either f_A or f_B is zero or one.

The maximum possible error in division is $E_{div} = 1/8$ or 12.5%, which occurs when $f_A = 0$, $f_B = 1/2$, or $f_A = 1$, $f_B = 1/2$. This function is non-negative, and it reaches the minimum (zero) when f_B is zero or one, or when f_A and f_B are equal.

The error in square root approximation is a monotonically increasing function. The maximum error, when $f = 1$, is given by $E_{sqrt} = 3/sqrt(8) - 1$, resulting in an error of 6.06%. The minimum error in square root operation occurs when the fraction is zero, and in such cases, the error is also zero. It is important to note that the fraction of any posit can never reach unity since $0 \leqslant f < 1$.



(a) Multiplication.

(b) Division.

(c) Square root.

Figure 8.4: Relative error plots between logarithmic approximate and exact posit operations.

The largest error observed among the proposed approximation methods is 12.5% in the case of division. However, it is important to note that in error-tolerant applications such as image filters or machine learning, these errors can be acceptable and have minimal impact on the quality of the results, as will be demonstrated in Section 8.6.

Furthermore, it is worth considering the effect of tapered precision in posit arithmetic on the proposed approximate computing. Notably, extreme magnitudes in posit arithmetic do not possess fraction bits, which are the source of error in the proposed methods. Hence, from an analytical perspective, no error is introduced when dealing with such values in the case of square root or multiplication. This holds true regardless of which operand has an extreme magnitude, as evident in Equation (8.10) and Figure 8.4a.

However, the case of division is slightly different, as the error function exhibits non-symmetry concerning the fractions of the numerator and denominator. When the denominator value represents an extreme magnitude (thus lacking fraction bits), the error is zero. Conversely, an extreme magnitude in the numerator does not limit the relative error of the division, but rather depends on the fraction value of the denominator. This behavior is clearly illustrated in Figure 8.4a, where the leftmost vertical region of the map encompasses

all possible values in the range $[0, 0.125]$. Naturally, the same behavior can be observed when exponent bits are shifted out.

For instance, consider the approximate computation of $1.329\,23 \times 10^{36} = 2^{120}$ (the maximum positive value) divided by $1.237\,94 \times 10^{27} = 2^{90}$ in 32-bit posits. The result is $1.073\,74 \times 10^{9} = 2^{30}$, which precisely matches the exact result. However, when the denominator is altered to $1.856\,91 \times 10^{27}$ (with a fraction value of 0.5), the result becomes $8.053\,06 \times 10^{8}$, which is 12.5% greater than the exact quotient of $7.158\,28 \times 10^{8}$.

8.6. Application evaluation

This section presents a comprehensive evaluation of the proposed PLAUs in a range of error-tolerant applications to assess their accuracy and usability. For ease of use, these approximate operators were implemented in software using Universal library [137]. The software modules were meticulously tested to ensure a one-to-one match with the hardware designs discussed in Section 8.7.

Considering that not all applications can benefit from approximate computing, a meticulous selection of multiple applications from diverse domains, encompassing image processing, machine learning, and deep learning, was undertaken. In these applications, achieving approximate results proves to be sufficient for their intended purpose.

8.6.1. Digital image processing

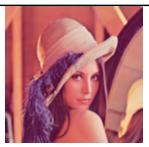
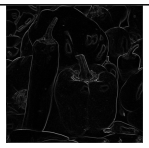
As a first use case, the implementation of image filters and edge detectors, commonly employed techniques in the field of computer vision for digital image processing, was carried out.

In such experiments, attention was directed toward two specific techniques: the blur filter and the Sobel operator. The blur filter, also known as average smoothing, is used to reduce image noise by applying a filter across all pixels in the image through a convolution operation. It involves the computation of the mean of values, which requires the use of the division operator. On the other hand, edge detection is employed to identify points in a digital image with discontinuities. It relies on calculating directional derivatives for a specific orientation, involving convolution, Euclidean distance calculation, and a subsequent normalization step.

Table 8.1 presents the results of applying these techniques to various 512 × 512 images using both exact and approximate posit operators. To assess the quality of the results obtained with approximate operators, the peak signal-to-noise ratio (PSNR) and structural similarity index (SSIM) metrics were employed, comparing them to the results obtained with exact operators.

As evident from the table, the exact and approximate images are practically indistinguishable to the human eye, and both metrics consistently yield high values. However, upon closer inspection of the blurred images, one may observe a slight difference: the approximately calculated images appear slightly brighter. This effect can be attributed to the positive error introduced by the approximate division operator, where higher results lead to brighter tones, closer to the color white.

Table 8.1: Image processing accuracy results.

	Cameraman	Lenna	Peppers	Baboon
Exact Computation				
Approximate Computation				
	PSNR: 27.6019 SSIM: 0.98217	PSNR: 29.0023 SSIM: 0.98281	PSNR: 28.2892 SSIM: 0.98318	PSNR: 28.7468 SSIM: 0.97396
Exact Computation				
Approximate Computation				
	PSNR: 51.0439 SSIM: 0.99909	PSNR: 44.5427 SSIM: 0.99629	PSNR: 45.6212 SSIM: 0.99629	PSNR: 38.2694 SSIM: 0.99503

8.6.2. Machine learning

The second case study in this section focuses on the K-Nearest Neighbors (K-NN) algorithm, which is a supervised classification or regression machine learning algorithm [158]. K-NN is categorized as a lazy learner algorithm, in contrast to eager learners like DNNs, as it does not involve a traditional training stage to generate a model. Instead, it selects one or more examples from the training data to determine the predicted value for the given sample. This approach eliminates the need for model generalization.

In this application, the computation of the Euclidean distance between data points is required, involving multiplication and square root operations. Consequently, these operations are substituted with their corresponding approximate counterparts.

To evaluate the performance of the algorithm, four well-known datasets for classification were chosen [36]. Each dataset was split into training and validation sets using a 70-30% ratio. The optimal number of neighbors, denoted as k, for each case was determined using the elbow method.

The total accuracy achieved by the algorithms is depicted in Table 8.2. Surprisingly, when comparing the use of approximate operators to the use of exact ones, the final accuracy remains the same in both cases. Interestingly, slight differences were observed in the intermediate results obtained with exact and approximate operators; however, these

Table 8.2: K-NN accuracy results.

Dataset	Instances	Attributes	Classes	k	Exact	Approx.
Iris	150	3	3	5	95.55%	95.55%
Wine	178	13	3	7	67.92%	67.92%
Glass	214	9	7	5	59.38%	59.38%
Breast Cancer	569	30	2	13	94.74%	94.74%

variations do not impact the overall accuracy of the algorithm. This result demonstrates the robustness of the K-NN algorithm to the incorporation of approximate operations.

8.6.3. Deep learning

For the third use case, an evaluation of the impact of approximate posit multiplication and division is conducted within the context of DNN inference.

Posit logarithmic approximate multiplication

To illustrate the effect of the proposed approximate posit multiplication in DNN inference, various CNN models were trained using posit arithmetic. Specifically, the LeNet-5 architecture was employed for the MNIST and Fashion MNIST (FMNIST) datasets, while the Cuda-Convnet architecture was used for the CIFAR-10/100 datasets. Once trained, the exact multiplications performed in the convolutional and fully connected layers of these models were replaced with posit logarithm approximate multiplication for inference purposes.

All models were trained for 50 epochs using the Adam optimizer and a batch size of 128. Prior research [114, 101] has demonstrated that 16-bit posits can be employed for DNN training without any loss in accuracy compared to the baseline 32-bit floating-point format (Float32). Hence, each model was trained using the Posit$\langle 16, 1 \rangle$ format, and during the inference stage, approximate posit multiplication was applied to the convolutional and fully connected layers. Furthermore, a comparison was made between the inference results obtained from the trained models using Float32 and bfloat16 data formats.

For performing computations in the posit format, software emulation through Deep PeNSieve, the framework introduced in Chapter 4, was employed. The framework was extended to accommodate the posit logarithm approximate multiplication operation for both scalar and matrix multiplication. It is important to note that, due to the absence of native hardware support for posit arithmetic, computations in this format were limited to CPU-only, making it challenging to train larger DNN models within a reasonable time frame.

In order to demonstrate the impact of approximate posit multipliers on deeper networks, the Pre-ResNet-20 and ResNet50 models [54, 55] were adopted for the CIFAR-10 and CIFAR-100 datasets, respectively. For training these models, Qtorch+ [59] was used to emulate training with posits while using a GPU environment. Subsequently, after training, the models were ported to Deep PeNSieve, enabling the utilization of the approximate multiplier during inference, as detailed above.

Table 8.3: Top-1 accuracy results (%) for the inference stage.

Model	Dataset	Float32	bfloat16	Posit$\langle 16, 1\rangle$	Posit$\langle 16, 1\rangle_{\text{approx}}$
LeNet-5	MNIST	99.25	99.13	99.29	99.23
LeNet-5	FMNIST	90.33	88.36	90.11	89.65
Cuda-Convnet	CIFAR-10	81.21	81.28	81.52	81.37
Cuda-Convnet	CIFAR-100	52.44	51.56	52.36	51.80
Pre-ResNet-20	CIFAR-10	87.23	84.81	87.58	85.97
Pre-ResNet-20	CIFAR-100	59.29	49.53	59.96	54.81
ResNet-50	CIFAR-10	93.12	90.12	92.39	89.87
ResNet-50	CIFAR-100	73.11	69.16	73.32	65.95

The Top-1 accuracy achieved during inference when applying approximate posit multipliers to matrix multiplication within the DNNs is shown in Table 8.3. For the smaller models trained on Deep PeNSieve, the utilization of approximate posit multiplication results in minimal accuracy degradation (less than 0.6%) compared to exact posit multiplication at the inference stage. However, in the case of the ResNet models, notable accuracy drops are observed when employing low-precision formats.

Two potential hypotheses were identified to explain this phenomenon. Firstly, the introduced error by the approximate operators may accumulate through layers, leading to a higher accuracy drop in deeper models. Secondly, the behavior of Qtorch+ may contribute to this effect. Since Qtorch+ internally utilizes Float32 precision for computations (with posits emulated by adding quantization layers before and after each operation), it may generate suboptimal weight values, particularly when replacing the matrix multiplication operation, which involves calculating numerous intermediate values in higher Float32 precision.

However, it is important to note that further research is required to determine the underlying cause of this phenomenon, although it falls beyond the scope of this thesis.

Finally, it should be noted that in the bfloat16 format, intermediate products of matrix multiplication are accumulated using Float32 accumulators [108], while posit-based computations are performed entirely with 16 bits, lacking quire accumulator for fused arithmetic or other methods to extend precision.

Posit logarithmic approximate division

Although the inference step mainly consists of matrix multiplications, it is frequent to incorporate pooling layers within DNNs to reduce the number of parameters to learn and summarize information. Specifically, the average pooling layers (AvgPool in short) compute the average value for patches of a feature map. In this scenario, the exact position division operation within the AvgPool layers is replaced with the approximated version. To achieve this objective, pre-trained widely recognized models [5] for image classification are chosen. Next, all weights and activations are quantized into posit format, and the default average pooling layers are then exchanged with a customized approximated version. The PyTorch framework is used to generate the pre-trained models and run the inference process. For fast posit format emulation, Qtorch+ is used [59]. In particular, pre-trained weights are converted

Table 8.4: ImageNet validation set accuracy (%).

Format	Float32		Posit16		$\text{Posit16}_{\text{approx}}$	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
AlexNet	56.556	79.088	56.546	79.084	56.546	79.084
GoogLeNet	69.772	89.534	69.772	89.546	69.830	89.526
ResNet-18	69.760	89.082	69.746	89.090	69.648	89.076
ResNet-50	80.854	95.438	80.840	95.440	80.776	95.430
MobileNetV3	75.258	92.572	75.278	92.578	74.674	92.302
EfficientNet-B0	77.698	93.524	77.704	93.524	77.200	93.366

into 16-bit posit format by just rounding the values to the nearest posit number. No extra quantization or fine-tuning techniques are applied, as it has been shown that 16-bit posits have sufficient accuracy to perform inference (and even training) of DNNs [114, 153, 96].

Table 8.4 presents the inference results of various models on the well-known ImageNet dataset [86]. For completeness, the accuracy of each pre-trained model is reported in floating-point format. As shown, in almost all cases, accuracy barely decreases when approximating the division of AvgPool layers. Only in MobileNetV3 and EfficientNet-B0 networks, a greater degradation of accuracy is observed (0.6% and 0.5% less, respectively). The reason behind this might be the depth of the networks. These models have a higher number of AvgPool layers that are being computed approximately, so it is expected that the error will increase in these cases. Meanwhile, the remaining models only have one AvgPool layer at the end of all convolutional layers, and indeed, in such models, the decrease in accuracy is much lower, not decreasing even by 0.1% in the worst case.

8.7. Hardware evaluation

After assessing the accuracy of the proposed PLAUs across various applications, this section presents synthesis results that highlight the hardware advantages of using approximate units.

The proposed architectures have been implemented in FloPoCo, which generates the corresponding VHDL code. To conduct a comprehensive evaluation of these units, several metrics such as area, power, energy, and area-delay product (ADP) are compared against their exact module counterparts. The designs of the exact posit units presented in Chapter 5 are used for such a comparison. All the designs are implemented as combinational circuits with a 45 nm standard cell library from TSMC. Standard cell synthesis has been performed using Synopsys Design Compiler®.

Table 8.5 compares the synthesis results for various posit formats with 2 exponent bits. Significant resource savings are observed in nearly all scenarios, except for the specific multiplier units with smaller bit lengths (8 and 16 bits), which exhibit slightly less delay compared to their approximate counterparts. This can be attributed to the disparity in the posit interpretation and decoding between the two types of units. While the approximate units employ a sign-magnitude interpretation, the exact units utilize the two's complement

Table 8.5: Synthesis results for exact and approximate posit operators.

	Design	Bits	Area (μm^2)	Power (mW)	Delay (ns)	ADP ($\mu m^2 \cdot ns$)	Energy (pJ)
Multiplication	Exact	8	516.50	0.66	1.06	547.49	0.70
		16	1883.95	3.03	2.06	3880.94	6.24
		32	7468.31	14.19	3.46	25840.34	49.09
		64	29429.64	63.42	6.22	183052.33	394.50
	Approx.	8	418.66	0.51	1.18	494.01	0.60
		16	972.08	1.15	2.07	2012.21	2.38
		32	2207.59	2.79	3.21	7086.35	8.96
		64	4535.60	5.63	5.82	26397.17	32.78
Division	Exact	8	841.78	1.05	2.75	2314.90	2.88
		16	2742.90	5.25	7.47	20489.48	39.20
		32	10349.74	25.75	27.61	285756.34	710.90
		64	39751.39	109.48	101.96	4053051.43	11162.92
	Approx.	8	423.36	0.52	1.18	499.56	0.61
		16	983.14	1.16	2.06	2025.26	2.40
		32	2230.64	2.82	3.21	7160.34	9.04
		64	4581.93	5.68	5.81	26621.02	33.02
Square root	Exact	8	444.06	0.57	1.06	470.70	0.60
		16	1923.00	3.00	3.61	6942.01	10.84
		32	8066.18	15.75	13.84	111635.99	217.94
		64	32509.81	73.79	52.70	1713267.22	3888.93
	Approx.	8	262.95	0.24	0.81	212.99	0.20
		16	600.00	0.55	1.49	893.99	0.81
		32	1396.38	1.47	2.25	3141.86	3.31
		64	2848.51	2.94	3.94	11223.12	11.57

approach, which is more efficient from the hardware perspective (see Appendix A). Moreover, for such small bit lengths, the hardware multipliers and adder units exhibit comparable performance, further contributing to the reduced delay of the exact operators in these particular scenarios. Nevertheless, the ADP and energy metrics, which can be used to represent energy efficiency, indicate that resource savings are still achieved with the use of approximate multipliers. On the other hand, for Posit32, the proposed approximate multipliers can achieve 70% area reduction and 80% power reduction, and even bigger savings for the 64-bit case, namely 85% and 91% less area and power, respectively.

The savings become even more striking when considering division and square root operations, both of which utilize the non-restoring algorithm in the case of exact operations. The implementation of this algorithm necessitates one iteration for each bit of the fraction,

resulting in a relatively slow process. Conversely, the logarithmic approximation of these operations exhibits exceptional speed. Indeed, the approximate division unit achieves a substantial reduction in delay, ranging from 57% (for Posit8) to 94% (for Posit64). This significant reduction in delay, combined with the corresponding decrease in area and power requirements, makes the approximate division unit highly energy-efficient.

Likewise, the logarithm-approximate square root operation demonstrates substantial improvements in terms of delay, area, and power consumption. The approximate square root unit can reduce the delay by 24% for 8-bit precision up to 93% for 64-bit precision. Additionally, it achieves a reduction in area ranging from 41% to 91% and a decrease in power consumption ranging from 58% to 96% for 8-bit and 64-bit precision, respectively.

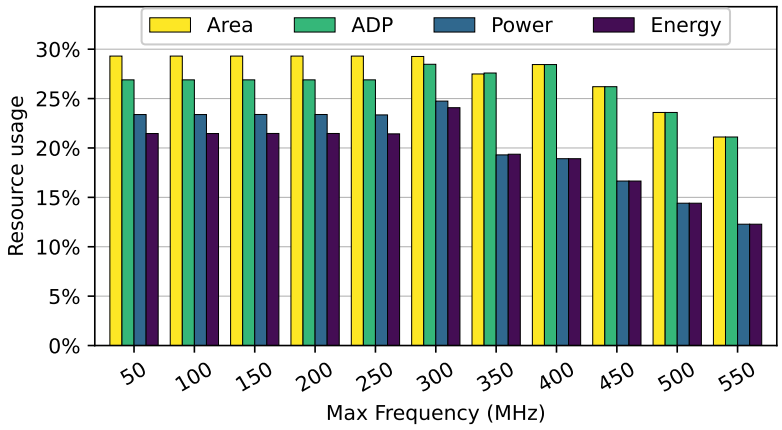
To conduct a more comprehensive evaluation, the different combinational units for the Posit32 format have been synthesized, targeting a wide range of maximum clock frequencies from 50 MHz to 1200 MHz. Remarkably, all the PLAU designs successfully synthesized at clock frequencies up to 1150 MHz, surpassing the frequency of their exact counterparts by more than double. In comparison, the exact posit multiplier reached a clock frequency of only 550 MHz, while the divider and square root unit achieved frequencies of 150 MHz and 250 MHz, respectively.

To analyze the results for each operator, Figure 8.5 presents the comparative cost of the approximate posit units in relation to their exact counterparts at different operating frequencies. As depicted in Figure 8.5a, the ADP and energy savings increase for higher frequencies, ranging from 73.1.7% to 78.9% and from 78.5% to 87.7%, respectively. Figures 8.5b and 8.5c exhibit a similar trend, with the proportion of area and power resources required by approximate units compared to exact units decreasing as frequencies rise. Notably, as observed in the previous unconstrained synthesis, both the approximate division and square root units demonstrate greater resource savings than the approximate multiplier unit. Specifically, the division unit exhibits ADP and energy reductions ranging from 94.2% to 97% and from 95.7% to 98.2%, respectively. Similarly, the approximate square root unit reduces ADP by 94.7% up to 96.9% and energy by 95.7% to 98%.

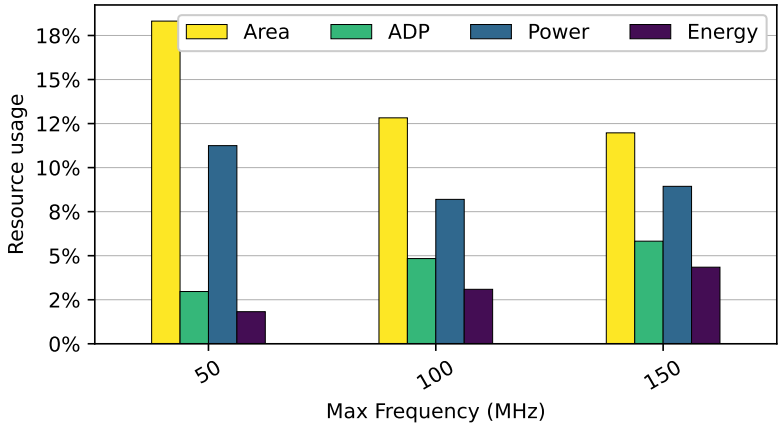
8.8. Conclusions

The preceding chapters have highlighted a fundamental trade-off in posit arithmetic: while this format offers higher accuracy, it comes at the expense of increased hardware overhead. To address this challenge, this chapter has explored the usage of approximate computing techniques within posit arithmetic, highlighting its potential to strike a trade-off between accuracy and efficiency in computer arithmetic.

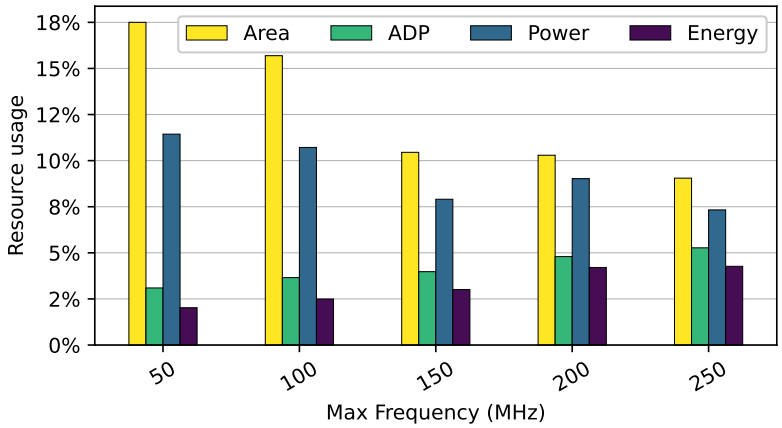
Specifically, the logarithmic approximation paradigm has been adapted to posit multiplication, division, and square root—three of the most energy-intensive arithmetic operations. Through meticulous algorithm design, implementation strategies for the various posit logarithm-approximate units (PLAUs) have been proposed, and the error on such approximations has been examined. The conducted analysis has also provided insightful boundaries for the approximation errors associated with each operation. Notably, the approximated multiplication exhibits a negative error not lower than -11.11%, while the division and square root approximations have positive relative errors bounded by 12.5% and 6.06%, respectively.



(a) Multiplication.



(b) Division.



(c) Square root.

Figure 8.5: Resource usage of the approximate operators compared to the exact ones.

In order to validate the error limitations and assess the practical implications of our proposed approximate operators, extensive experiments were conducted on fault-tolerant

applications. These applications encompassed image filters, machine learning algorithms, and DNNs. Digital image processing and machine learning applications showed no degradation in accuracy or quality of results when using approximate operators. This highlights the resilience of these applications to approximation computing techniques. Regarding DNNs, different PLAUs have been used during inference for reducing the computational requirements of this phase. In smaller models, negligible errors were observed with the introduction of approximate operators. However, as the depth of the models increased, a slight degradation in performance was observed. This could potentially be attributed to the accumulation of errors across successive layers. This phenomenon underscores the need for careful consideration of the approximation impact when dealing with deeper neural networks.

Furthermore, the synthesis evaluation conducted in this study has demonstrated the competitiveness of the approximate units compared to their exact counterparts, particularly in terms of energy efficiency. The approximate Posit32 multiplication, division, and square root units demonstrated their ability to synthesize designs for maximum frequencies that surpassed those achievable by their exact counterparts, with frequency improvements exceeding double (or even seven times higher in the case of division). Moreover, when targeting the same frequency, the approximate units exhibited significant reductions in ADP and energy, ranging from 78.9% to 97% and from 87.7% to 98.2%, respectively. These findings firmly establish the energy-saving potential of the proposed PLAUs. In summary, the results of this chapter demonstrate that the proposed PLAUs strike a compelling trade-off between accuracy and efficiency when compared to their non-approximate counterparts. The tested fault-tolerant applications, along with the synthesis evaluation, support the notion that these units provide a feasible solution for enhancing computational performance while reducing energy consumption.

In the next chapter, a different approach for diminishing the hardware resources of posit arithmetic units is presented. Unlike the approximate method presented in this chapter, such another approach does not compromise the accuracy of the results. Consequently, it is not confined to error-tolerant applications, although the reduction in resources may not be as pronounced.

9

Half-unit-biased posit arithmetic

While posit formats share similar elements with FP formats, as demonstrated in Chapter 5, their arithmetic units are currently more costly. The contribution outlined in Chapter 8 addresses this challenge by employing approximate computation techniques, sacrificing a certain degree of accuracy in the results. Nevertheless, one of the defining characteristics of the posit format is its inherent high accuracy, surpassing that of the floating-point format and yielding more precise results.

This prompts the question of whether any technique exists to curtail the hardware requirements of the posit format without compromising result accuracy. Given the relative novelty of posits, a thorough exploration of their hardware implementations may not yet be exhaustive. Numerous contemporary techniques employed in FUs have the potential for application to posit units, offering avenues to reduce their hardware costs.

Among these techniques, the Half-Unit-Biased (HUB) approach recently emerged as a viable strategy to alleviate the implementation cost of arithmetic units dealing with real numbers subjected to rounding to the nearest [164].

This chapter studies the usage of the HUB formats to implement the computation within posit-based systems. To that end, addition and multiplication FUs for this new format are designed and evaluated.

The main objective of the chapter is to demonstrate whether the HUB approach can be applied to the posit format to provide similar hardware reduction as to other formats, while maintaining the same accuracy. Specifically, in this chapter:

1. The posit HUB format is defined. To the best of our knowledge, this is the first work that proposes the combined implementation of the HUB and posit formats.

2. Functional units for addition and multiplication of posit HUB numbers are carefully designed.

3. The error of standard posit and posit HUB formats are analyzed experimentally for the aforementioned operations.

4. The proposed posit HUB designs are implemented and synthesized for comparison with conventional posit adders and multipliers presented in Chapter 5.

The chapter is structured around the list of contributions outlined below. Section 9.1 undertakes a comprehensive review of the fundamentals of the HUB format and introduces the conceptual framework of posit HUB numbers. Section 9.2 delves into architectures tailored for implementing posit HUB operations, with a specific focus on extending posit addition and multiplication to the proposed posit HUB format. Additionally, the section investigates rounding mechanisms and the conversion process between formats. Section 9.3 conducts an experimental error analysis to assess the viability of utilizing HUB formats as opposed to conventional ones. Section 9.4 undertakes a comparative examination of ASIC implementations for the proposed adders and multipliers. Finally, Section 9.5 culminates the chapter by presenting the principal contributions and providing concluding remarks.

9.1. Half-unit biased formats

HUB is a number format used in computer arithmetic. It is an emerging format that shifts the represented numbers by half ULP, i.e., it considers an implicit least significant bit (iLSB). Thanks to this iLSB, this format simplifies two's complement and round-to-nearest operations by preventing any carry propagation, which saves power consumption, time, and area.

The HUB format was initially defined in [62]. Until now, the efficiency of using the HUB formats has been demonstrated in either fixed-point representation [60, 129, 30] or the floating-point format [61, 63, 163]. However, this technique could be applied to almost any representation of real numbers, as long as it uses rounding to the nearest integer. This is exactly the case with posit arithmetic, whose only rounding mode is the one mentioned above.

In this section, the ideas used in [62] are formalized and extended to optimize posit processing datapaths. A new representation system is defined, which allows simplifying the computation of round-to-nearest rounding and complement operation.

9.1.1. The posit HUB format

In the case of floating-point, a HUB number is defined as an FP number whose significand is a HUB fixed-point number and its exponent is a conventional one. That means the significand has an iLSB set to one, and consequently, each floating-point HUB number corresponds to the middle point of two consecutive floats. That allows performing rounding-to-nearest by an actual truncation and two's complement by bit inversion. This simplification at the logic level produces remarkable savings in the hardware implementation of FPUs [61, 63].

Posits only use round-to-nearest, making them a great candidate for using the HUB approach. However, unlike floating-point HUB, which only applies the HUB approach to the significand (the only field rounded in FP), posits require the HUB approach to be applied to the entire posit number since posits might also require rounding the exponent. Considering this, we define a posit HUB as a posit number with an iLSB set to one, except for the zero value. Recalling Figure 1.5, the generation of a posit format of one bit larger size involves

Table 9.1: Posit6 format decodification.

Binary	Conventional					HUB				
	s	r	e	f	value	s	r	e	f	value
000011	0	−3	2	0	0.000 976 562 5	0	−3	3	0	0.001 953 125
001011	0	−1	1	0.5	0.1875	0	−1	1	0.75	0.218 75
010000	0	0	0	0	1.0	0	0	0	0.25	1.25
011111	0	4	0	0	65 536	0	5	0	0	1 048 576
100001	−1	−4	0	0	−65 536	−1	−4	2	0	−16 384
110000	−1	0	0	0	−1.0	−1	0	0	0.25	−0.875
110101	−1	0	2	0.5	−0.1875	−1	0	2	0.75	−0.156 25
111101	−1	2	2	0	−0.000 976 562 5	−1	2	3	0	−0.000 488 281 25

adding a 0 bit to the right of the existing posit, and filling the gaps with posits ending in 1. Hence, an n-bit posit HUB is like a conventional $(n+1)$-bit posit but the LSB is always set to one. Consequently, the iLSB is part of either the regime (if there is no negated regime bit in the posit number), the exponent (if such a field has less than two bits), or the fraction (if the posit number has the maximum number of exponent bits).

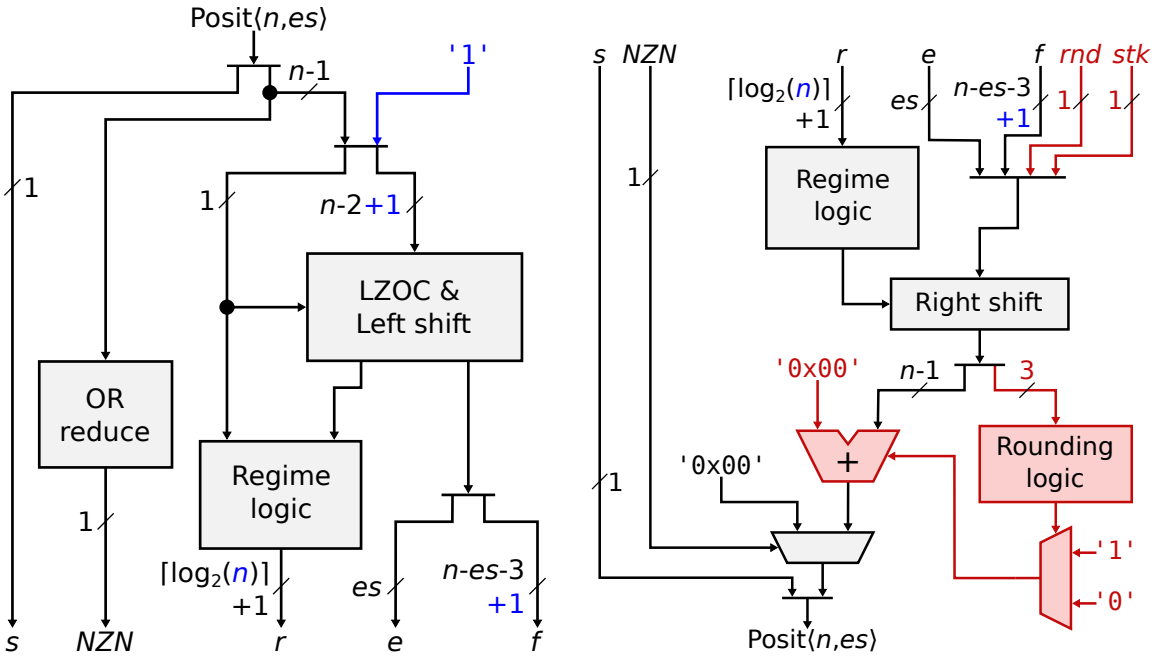
The introduction of the iLSB produces that, in the HUB version, the numbers exactly represented by the conventional posit format are shifted to the midpoint between those and the next posit number (in logarithmic scale when affecting the exponent). Consequently, round-to-nearest is obtained by truncating either the fraction or the exponent when coding a posit HUB number.

Table 9.1 shows several examples comparing the values represented for conventional and HUB 6-bit posit numbers. The s, r, e, and f columns represent the corresponding values in Equation (1.5) extracted from the bit representation, namely, sign, regime, exponent and fraction values. Recall that the HUB format considers an iLSB. Notice how the same bit patterns always result in greater values when interpreted under the HUB format. This is observed in both positive and negative cases, since all values are biased in the same direction.

Like in the case of floating-point, the HUB format for posit allows performing rounding-to-nearest by an actual truncation and two's complement by bit inversion. However, the two's complement of a posit number results in the additive inverse (or opposite) number. Therefore, HUB also simplifies posit subtraction.

9.2. Posit HUB architectures

As detailed in Section 5.2, posit arithmetic units initially decode the operands to acquire the exponent and the fraction. This process involves detecting the size of the regime and shifting the exponent and fraction bits accordingly to the left. Consequently, the sizes of the exponent and fraction are expanded to their regular dimensions. In contrast to the floating-point implementation, where the inclusion of the iLSB can be deferred until just before the fixed-point operation [163], in posit arithmetic, the iLSB must be included at this stage to append it to the correct position—whether it be the fraction, the exponent, or the



(a) Decoder. **(b)** Encoder.

Figure 9.1: Architecture of a posit HUB decoder and decoder modules.

regime. This is depicted in Figure 9.1a, which highlights in blue the extra hardware that is required in comparison with the conventional posit decoder illustrated in Figure 5.3.

After the inclusion of the iLSB, the resulting value becomes a regular number but with one additional bit. It is worth noting that when utilizing a power of 2 as the bit length of the posit, the length of the regime does not increase; only the fraction field is expanded. Consequently, the corresponding number can be processed with regular posit circuits, considering the presence of an extra bit. However, depending on the specific operation, certain simplifications are applied to these circuits to enhance the implementation results, as elaborated below.

The second significant difference arises during the encoding/packing phase. Here, the regime is generated based on the scaling factor of the result, and the exponent and fraction bits are shifted to the right accordingly. In the conventional implementation, depicted in Figure 5.2, the discarded bits play a role in rounding the fraction (or the exponent if all the fraction bits are discarded). This rounding necessitates additional logic and an addition operation. However, in the HUB implementation, no supplementary operation is needed after the shifting, as values are rounded through truncation. Figure 9.1b visually represents the logic that is omitted for the HUB circuit, highlighted in red.

Beyond adjustments to the decoder and encoder logic, specific modifications may be applied to the datapath, depending on the implemented operation. These modifications serve to simplify the circuit or address corner cases. The subsequent subsections provide a detailed explanation of additional adjustments to the datapath of both the multiplier and adder.

9.2.1. Posit HUB adder

To implement addition, the significands are added after aligning the one with the smallest exponent to the right. Three additional bits, including a sticky bit, are retained from the shifted significand for subsequent normalization and rounding steps to achieve proper rounding to the nearest value. In the case of the HUB version, these additional bits are unnecessary since no rounding is performed. However, an extra bit is needed in the fixed-point adder due to the inclusion of the iLSB. Furthermore, specific corner cases require special attention in the HUB version:

- **Alignment of a Negative Number.** If a negative number is aligned for more bits than the size of the significand (i.e., the smallest operand is smaller than the weight of the LSB of the other operand), it results in an incorrect effective subtraction of one ULP from the larger significand. This arises due to sign extension and the limited number of bits in the adder. In the conventional implementation, this situation is spontaneously resolved by rounding. However, it poses an issue in the HUB version as rounding is eliminated. To address this, a logical right-shifting is employed instead of arithmetic right-shifting when this situation is detected.
- **Catastrophic Cancellation after Right-Shifting.** In scenarios where catastrophic cancellation occurs after right-shifting the significand of the smallest number by one bit for alignment, the discarded LSB is required for normalization. In the conventional case, this bit is retained in the rounding bit as a side effect. For the HUB version, this bit must be appended to the result of the addition before normalization.

9.2.2. Posit HUB multiplier

To implement multiplication, once the posit operands are decoded, the significands are constructed by appending the implicit leading bits (1 or -2) to the fraction bits. Subsequently, both significands undergo multiplication, and the exponents are summed. In the case of the HUB version, it necessitates a fixed-point multiplier that is one bit wider due to the presence of the iLSB. However, the computation of the sticky bit and the guard bit is rendered unnecessary since rounding is eliminated during the encoding step. As will be detailed in Section 9.4, this implies that HUB introduces a marginal increase in area for low frequencies. Nevertheless, this increment is offset as frequency rises, thanks to the shorter datapath.

9.2.3. Comments about conversion and rounding

Conversion between numbers of different sizes

The process of converting between posit formats of differing sizes, while maintaining the same exponent size, is inherently uncomplicated. In the design philosophy of posits, expanding a posit format to a wider bit length is a straightforward task, involving the mere padding of zeros. Specifically, in the context of the posit HUB, it is crucial to preserve the information encapsulated in the iLSB of the original number, which aligns with the MSB of the appended bits.

Contrastingly, the conversion to a narrower bit length necessitates a rounding operation. Notably, the posit arithmetic framework supports a rounding-to-nearest tie-to-even approach,

wherein examination of the last bit and determination of whether the discarded bits are all 0 (sticky bit) is imperative. This scrutiny governs the decision to round up, prompting the incremental adjustment of the truncated posit bitstring when necessary. It is pertinent to acknowledge that these conversion operations engender a substantial hardware overhead.

Furthermore, it is essential to highlight that the rounding-up operation, inherent in the conversion process, may potentially result in an overflow, which is incompatible with posit arithmetic design principles. However, under the HUB format such a complexity is evaded, since the truncation of the bitstring is sufficient to obtain an accurately rounded conversion. It is noteworthy that the iLSB of the original number plays an important role in ensuring the sticky bit is always set to 1, thus eliminating the existence of tie cases in the conversion process.

Conversion between conventional and HUB formats

The process of converting numbers between a conventional format and its corresponding HUB format, wherein both formats share the same number of explicit bits and exponent size, unfolds in a distinct manner. This conversion scenario becomes relevant when data interchange transpires between systems operating in these disparate formats. Notably, owing to the inherent dissimilarity in exactly represented numbers for each format (excluding zero and NaR instances), this conversion invariably introduces rounding and, consequently, a rounding error.

Moreover, in the context of this conversion, each exactly represented number in one format consistently finds itself positioned equidistant between two numbers of the alternate format, which creates a tie case. This characteristic simplifies the hardware requirements, as the computation of the sticky bit becomes unnecessary. However, it must be noted that the magnitude of the rounding error consistently stands at 0.5 ULP.

Rounding in HUB formats

Following each arithmetic operation, such as addition or multiplication, a crucial step involves rounding the resultant value. Leveraging its iLSB, the posit HUB format executes a round-to-nearest operation through a straightforward truncation process, thereby streamlining the hardware complexity. However, it is important to acknowledge that a potential bias in the rounding may manifest under certain conditions, specifically when a tie situation arises.

In the realm of HUB numbers, a tie condition occurs when the outcome of an operation, after normalization, exhibits a bit 0 in the position of the iLSB, with all subsequent bits to its right being 0. In other words, when the bits discarded after truncation amount to zero. This circumstance leads to a rounded tie-to-away significand, as the significand consistently rounds up.

Addressing this concern involves the application of a technique analogous to that employed for HUB floating-point numbers [164]. This entails detecting the tie condition and rectifying the result through a rounding-down process. However, it is crucial to note that the implementation of such unbiased operators inevitably leads to an increase in both area and power consumption. This, however, falls beyond the immediate scope of this chapter.

9.3. Error analysis

In this section, the experimental evaluation focuses on the error introduced by the posit HUB format, comparing it with the error associated with conventional posits. The purpose of this analysis is to verify that using HUB does not compromise the accuracy when dealing with posits. This is especially relevant in the case of posits, since because of their variable-length fields, the iLSB might affect not just the fraction, but also the exponent or regime, so the value of the half ULP is not constant, as in the case of fixed- or floating-point arithmetic.

To validate the proposed approach across a representative spectrum of scenarios, all experiments were conducted encompassing the various arithmetic operations proposed in the previous section.

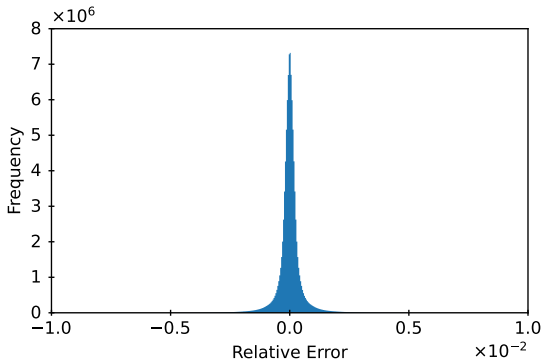
9.3.1. Experimental setup

An empirical error study by Monte Carlo simulation is provided to demonstrate that the posit HUB could be used instead of the conventional posit format in real numbers-specific applications while guaranteeing the same level of precision. To test the arithmetic operations, two randomly uniform Posit24 numbers within the range $[-2^{20}, -2^{-20}] \cup [2^{-20}, 2^{20}]$ were utilized. This range is deemed acceptable for data from real applications and offers variability in the width of the regime. In such a data range, neither overflow nor underflow will occur in the products. They were converted into conventional 16-bit posit and HUB format, using round-to-nearest, so both formats have an initial error with the same probability. The exact result of the operation is computed using twice the number of bits and compared with the obtained result in both conventional and HUB 16-bit posit formats by calculating the relative error. The Universal library [137] was used to emulate the computation of such formats via software.

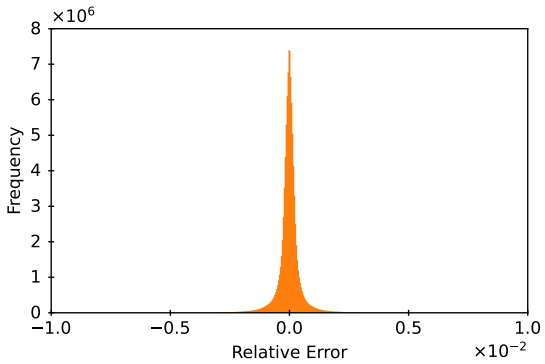
9.3.2. Results and discussion

The relative error with respect to the exact result is computed for addition and multiplication, as well as for the conversion operation from Posit24 to Posit16 (this is to check that no extra error is introduced in the conversion, so the inputs for both HUB and conventional formats have the same error, on average). Each experiment was repeated 10^8 times for each tested operation. In the case of addition, the results that produced zero and subtractive cancellations, which were less than 0.01%, were excluded. The high relative errors produced by these cases would hide the behavior of the error in the cases of interest.

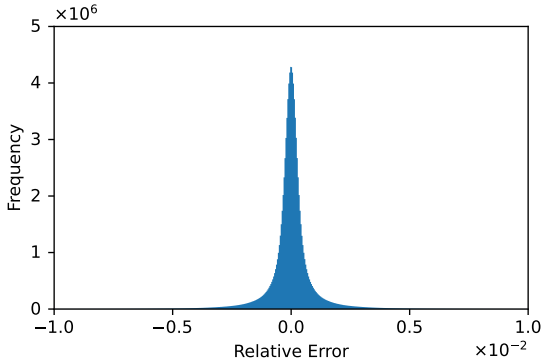
Figure 9.2 shows the histograms of the computed rounding error for each operation, whereas Table 9.2 shows the main statistical parameters corresponding to these errors. Although the rounding error values corresponding to both approaches are always different, the probability distributions of these errors are quite similar, as shown by the histogram and the statistical parameters. For all the operations, statistical parameters are almost the same, but the mean because the rounding implemented for the HUB version may produce some bias. Nonetheless, the value is still very low and enough for many applications. Unbiased rounding could also be implemented [164], but its study is out of the scope of this thesis and will be studied in future work.



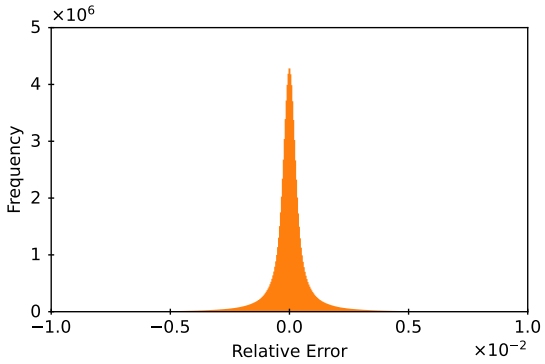
(a) Conventional posit addition.



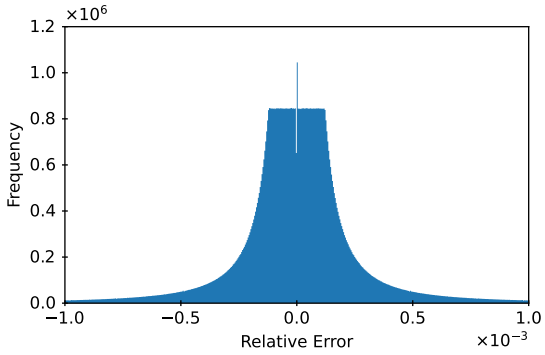
(b) Posit HUB addition.



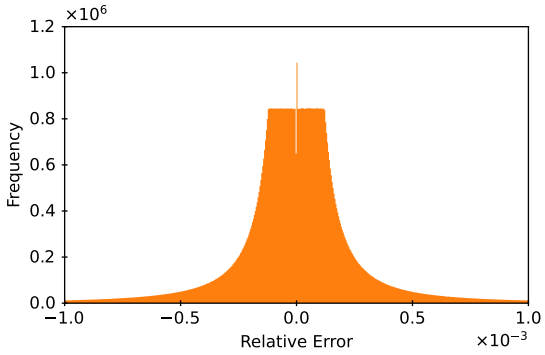
(c) Conventional posit multiplication.



(d) Posit HUB multiplication.



(e) Posit24 to Posit16 conversion.



(f) Posit24 to HUB Posit16 conversion.

Figure 9.2: Histograms of the rounding error for arithmetic operations.

The outcomes from both theoretical and experimental investigations demonstrate that HUB formats can serve as an alternative to the traditional posit representation for handling real-number computations. It is important to note that utilizing the HUB format for solving real-number calculations may not produce identical outputs compared to those obtained using standard posits. Yet, the precision of the calculations would remain unchanged.

Table 9.2: Statistical parameters of rounding error distribution.

Format	min	mean	max	σ	SNR_{dB}
Addition					
Posit	−0.9922	4.921e−07	0.9542	2.475e−03	56.02
HUB	−0.9655	−1.697e−05	0.9532	2.465e−03	56.00
Multiplication					
Posit	−0.1158	9.290e−08	0.1163	1.951e−03	25.71
HUB	−0.1298	−4.074e−06	0.1302	1.954e−03	25.81
24-to-16 precision conversion					
Posit	−3.891e−03	−5.405e−08	3.891e−03	3.987e−04	56.64
HUB	−7.812e−03	−1.751e−06	3.876e−03	3.987e−04	56.64

9.4. Hardware evaluation

In this section, the main hardware implementation results of the posit HUB circuits presented in Section 9.2 are presented and compared with the conventional units designed in Chapter 5.

To get a detailed evaluation of the hardware cost of the application of HUB format in posit arithmetic, the circuits presented in Section 9.2 for posit HUB numbers, along with the posit units presented in Chapter 5, have been modeled in FloPoCo to generate synthesizable VHDL.

9.4.1. Synthesis results and comparison

The previous posit HUB adders and multipliers have been synthesized using Synopsys Design Compiler® with a 45 nm standard cell library from TSMC. To provide a more comprehensive evaluation, two different kinds of synthesis have been performed:

- First, units for different bit lengths (8, 16 and 32 bits) have been synthesized with no timing constraints. These results are split into components, i.e. encoder, decoder and arithmetic core for a detailed evaluation. This evaluation shows how the HUB format affects the different submodules within each design, and how that scales with different bit lengths.
- Then, for the case of Posit32 units, the same units have been synthesized targeting different maximum clock frequencies, ranging from 200 MHz to 800 MHz. This evaluation attempts to illustrate the performance that the HUB format can achieve.

Posit HUB adder

Figure 9.3 illustrates the area, power, and delay characteristics reported by the synthesis tool for both approaches across various bit lengths of adders. The observed behavior remains consistent across all bit lengths. The iLSB in HUB has a negligible impact on decoding and the addition core, while the complexity of the encoder is significantly reduced, courtesy of

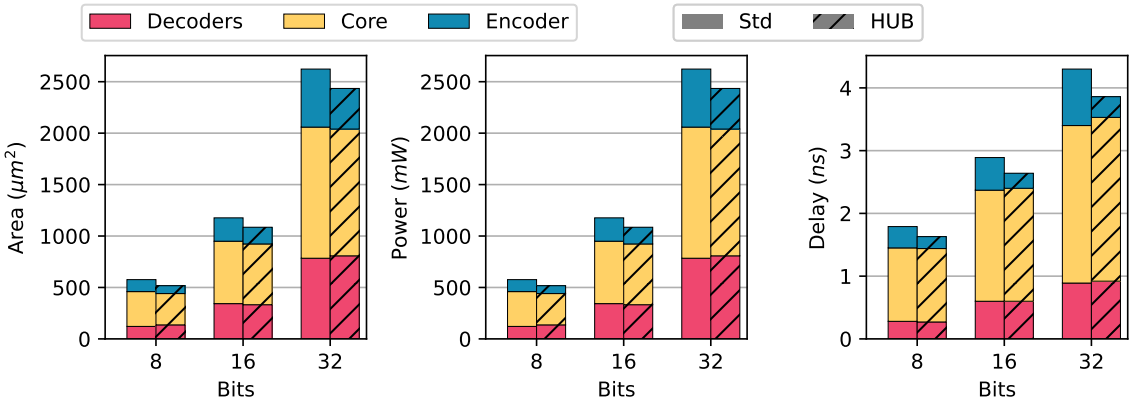


Figure 9.3: Synthesis results for conventional and HUB Posit$n$ adders.

rounding removal. On a global average, the HUB format exhibits an 8.4% reduction in area, a 9.0% reduction in power, and a 9.3% reduction in delay.

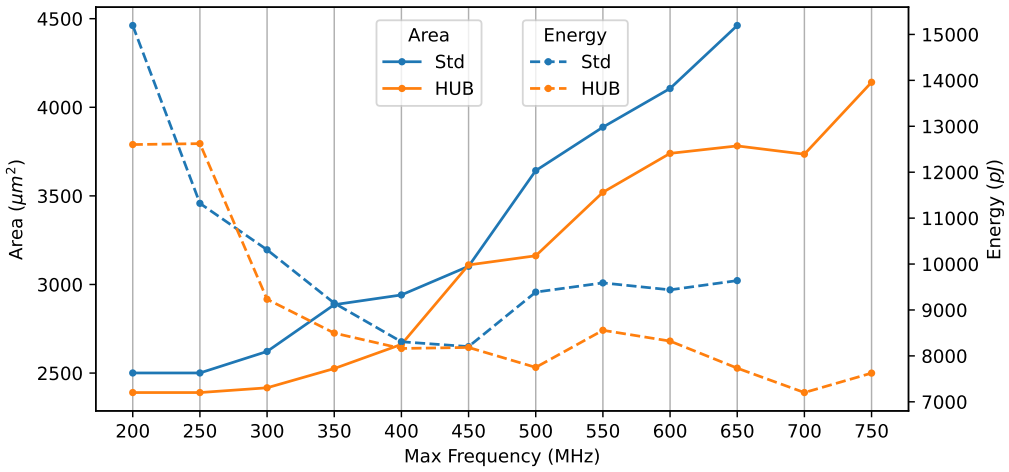
For a more comprehensive evaluation, Posit32 adders were synthesized to target different maximum clock frequencies. Figure 9.4a presents a comparison of area and energy results for both approaches. Similar to the unconstrained scenario, posit HUB adders generally exhibit substantially smaller area requirements compared to conventional ones. This behavior is mirrored in energy consumption, with most HUB results clearly surpassing conventional counterparts. Notably, HUB adders achieve a clock frequency of 750 MHz, outpacing conventional ones, which reach only 650 MHz. This discrepancy arises from HUB savings in the rounding logic, a critical component of the path.

To facilitate comparison, Figure 9.4b presents a normalized version of these results, incorporating the ADP. As anticipated, the reductions in area and energy are highly significant, peaking at 15% for area and 20% for energy at the maximum clock frequency. The average reductions for area and energy stand at 8.5%. Consequently, the HUB version achieves a reduction in ADP of up to 15%, with a mean reduction of 10%.

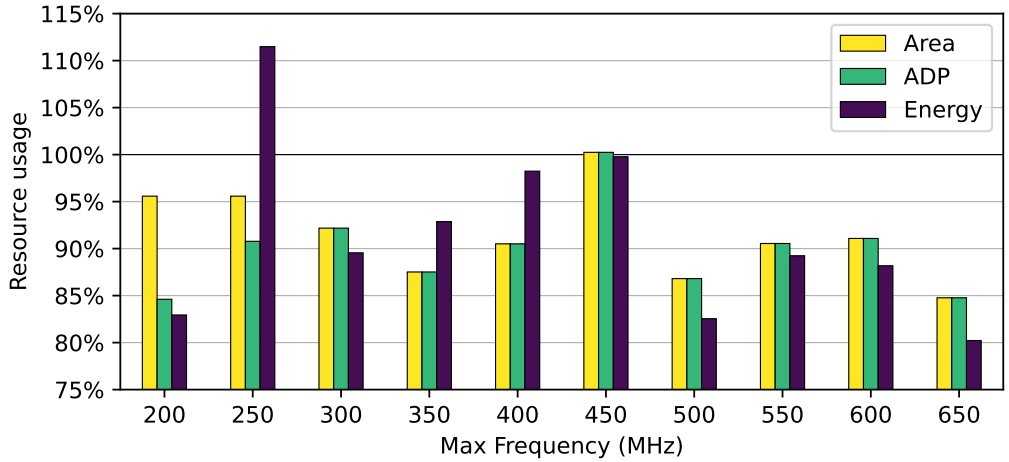
Posit HUB multiplier

The area, power, and delay results from the synthesis of various multipliers are depicted in Figure 9.5, where subcomponents are distinguished. A consistent trend is observed across diverse bit lengths. While the HUB format streamlines the logic for the posit encoder (thanks to rounding removal), this simplification does not fully offset the area and power increments induced by HUB in the multiplication core, averaging 6.1% and 10.7%, respectively. Nevertheless, the delay of HUB multipliers experiences a substantial average reduction of 12.9%, compensating for the area and power increase, particularly at high frequencies.

In-depth comparison of Posit32 conventional and HUB multipliers, targeting various maximum clock frequencies, is presented in Figure 9.6a. Unlike the more pronounced differences observed in adders, distinctions in multipliers are subtler. Conventional posit multipliers exhibit less area when targeting lower frequencies, while the HUB case generally demonstrates slightly lower area requirements at higher frequencies. Conversely, energy



(a) Area and energy synthesis results.



(b) Resource usage of the posit HUB units compared to the conventional ones.

Figure 9.4: Comparison of conventional and HUB Posit32 adder implementations.

consumption and the ADP are generally lower for HUB multipliers, particularly at lower frequencies, where the reduction exceeds 12%, as illustrated in Figure 9.6b. This aligns with the findings in Figure 9.6a, showcasing lower delays for HUB operators. On average, energy is reduced by 3.7%, and ADP by 4.8%. Notably, similar to adders, HUB multipliers achieve a clock frequency of 650 MHz, surpassing conventional counterparts limited to 600 MHz.

9.5. Conclusions

This chapter delves into the exploration of the Half-Unit-Biased (HUB) approach within posit-based systems, presenting a novel perspective on mitigating the implementation cost of arithmetic units dealing with real numbers under rounding to the nearest. As an integral part of this endeavor, addition and multiplication FUs for the posit HUB format were meticulously designed and evaluated.

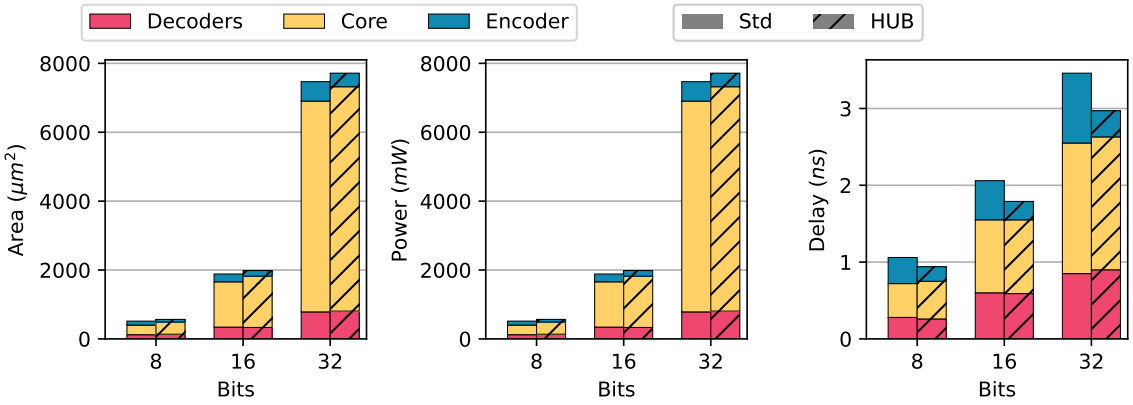
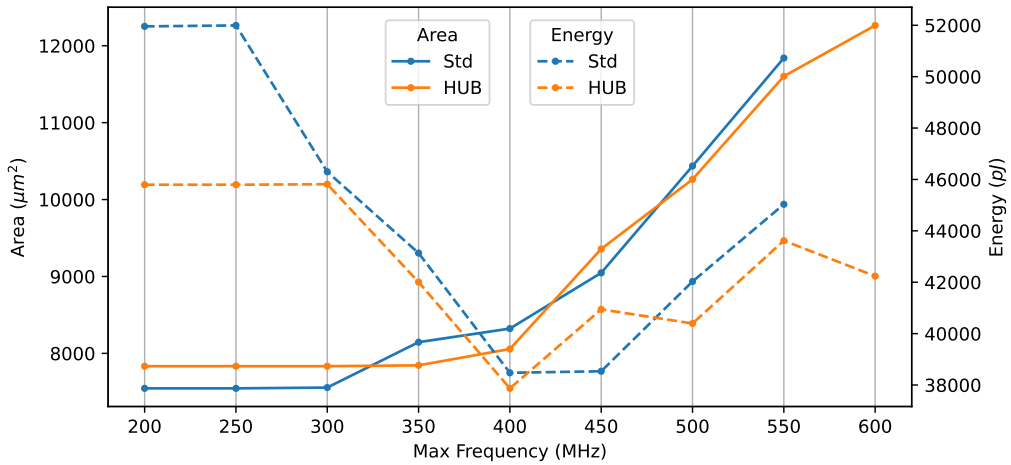


Figure 9.5: Synthesis results for conventional and HUB Posit$n$ multipliers.

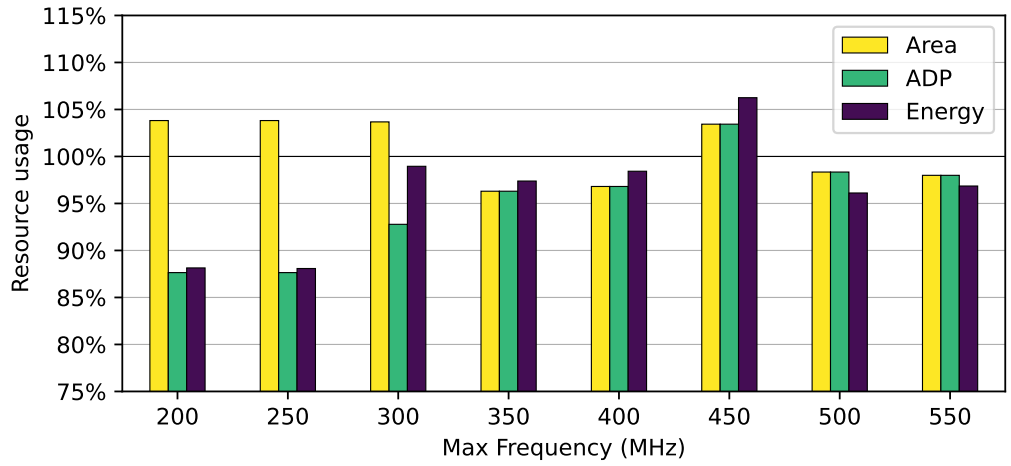
The primary objective of this chapter was to determine if the HUB approach, a technique originally developed for FPUs, could be successfully applied to posit units. The aim was to achieve a reduction in hardware costs similar to other formats while preserving accuracy. This study stands as the pioneering work proposing the integration of HUB with posit arithmetic and corresponding FUs.

The findings from the comprehensive evaluation highlight the substantial benefits of adopting the posit HUB format. Particularly noteworthy is the significant enhancement observed in adders, especially when targeting higher frequencies. The advantages extend to multipliers, primarily attributed to the observed reduction in energy consumption. Importantly, the posit HUB format demonstrates an increased capability to reach higher frequencies, further affirming their viability. While the savings in posit arithmetic units using the HUB approach may not match those in FPUs, the demonstrated value is unmistakable. These savings are achieved without compromising accuracy, maintaining the same level attained with conventional posits.

In conclusion, this chapter underscores the promising potential of the posit HUB format in the realm of posit arithmetic units. The observed hardware reduction, coupled with the preservation of accuracy and improved frequency capabilities, positions posit HUBs as a valuable avenue for future research. Subsequent work may explore the implementation of more complex operators, such as fused multiply accumulation units, to further unravel the full spectrum of possibilities offered by posit HUBs in advanced computing architectures.



(a) Area and energy synthesis results.



(b) Resource usage of the posit HUB units compared to the conventional ones.

Figure 9.6: Comparison of conventional and HUB Posit32 multiplier implementations.

10

Conclusions

10.1. Conclusions and main contributions

As described in the motivation of this thesis (Section 1.1), within the current trend in the specialization of computer architectures, the suitability of IEEE 754 floating-point numbers for all kinds of applications is questioned nowadays. To achieve performance improvements in the post-Moore era, new arithmetic formats have been proposed for their usage in specific applications. Among the various alternative formats proposed, the posit format is perhaps the most promising to replace the floating-point standard in different modern HPC applications. However, this novel format currently lacks the necessary hardware support, which difficulties its adoption and the investigation of its capabilities.

In this context, the primary objectives of this Ph.D. thesis were: (I) *"to explore the characteristics and properties of the posit format, assessing its viability as an alternative to the prevailing IEEE 754 floating-point format in modern applications"*, and (II) *"to propose designs for posit arithmetic units that offer hardware support for this format, with a special emphasis on enhancing energy-efficiency and performance"*.

To achieve these goals, both software and hardware approaches were considered targeting different scenarios, all of them with the previous goals as common objectives. The main contributions of this thesis can be summarized as follows:

- We have shown that the posit number format usually provides higher accuracy in common scientific applications than the IEEE 754 floating-point format when using the same amount of bits, or precision.
- We have demonstrated how, for the case of deep learning, training classical DNNs with Posit$\langle 16, 1 \rangle$ format provides no accuracy degradation compared with 32-bit formats with no modification of the training flow. In addition, 8-bit posit formats are suitable for low-precision inference of such models, provided that the matrix multiplications are performed without rounding using fused operations.
- We have determined the hardware implementation cost of posit arithmetic units. The designs proposed outperformed previous implementations, corroborating the quality

of the results. In comparison with conventional floating-point units, the corresponding posit operators present an overhead in terms of area and performance, but not as much as doubling the bit length. A demonstration has also been provided to show the scalability of these implementations in designing custom accelerators based on this arithmetic format, while achieving similar output metrics.

In addition to these general contributions, each proposal explores a different scenario, and therefore, each of them has some specific contributions:

Posit arithmetic in deep learning applications

A framework to accurately simulate DNN inference and training using the posit format for all internal operations has been developed targeting different bit lengths. Evaluations of different models show that 16-bit posits provide similar accuracy as 32-bit formats on training. In addition, the framework has the capability of successfully performing low-precision (8-bit) inference with exact accumulation of products using fused arithmetic.

Hardware design for posit arithmetic

A full set of posit functional units has been developed for the basic arithmetic operations of addition, subtraction, multiplication, division, and square root. Also, a unit for fused MAC operation has been proposed to perform the exact accumulation of a large number of products. The proposed designs explore different implementation alternatives to provide competitive units. Experimental results reveal how, compared with implementations from previous works, the proposed approach increases energy efficiency consistently in all the tested scenarios.

Leveraging the proposed arithmetic operations, posit arithmetic has been incorporated into an HLS tool to automatically generate application-specific accelerators based on this format. This approach is proven to work on a wide variety of applications, with similar capabilities as floating-point. The evaluations show a similar trend as in the case of standalone operators.

Alternative arithmetic formats based on posit

Approximate computing techniques have been incorporated into posit arithmetic to notably reduce the hardware requirements of multiplication, division, and square root units, in compensation for a possibly inaccurate result. Digital image processing, machine learning and deep learning are application areas that have proven to benefit greatly from this type of computation.

Another contribution of this dissertation is the development of the HUB posit format, which integrates the benefits of the HUB approach into the original posit arithmetic proposed by John Gustafson. This format has proven to reduce the hardware complexity by suppressing the rounding logic, while maintaining the same accuracy and the rest of the properties of conventional posit format.

10.2. Related publications

The contributions of this thesis are supported by the publication of multiple articles in different peer-reviewed international conferences and journals. Publications are classified as directly and indirectly related to the content of the dissertation. Below they are listed in chronological order, along with the importance metrics when appropriate.

10.2.1. Directly related publications

Murillo, R., Hormigo, J., Del Barrio, A. A., and Botella, G. HUB Meets Posit: Arithmetic Units Implementation. *IEEE Transactions on Circuits and Systems II: Express Briefs 71*, 1 (2024), 440–444 — JOURNAL, JIF'22: 4.4 (Q2)

Murillo, R., Del Barrio, A. A., Botella, G., and Pilato, C. Generating Posit-Based Accelerators With High-Level Synthesis. *IEEE Transactions on Circuits and Systems I: Regular Papers 70*, 10 (2023), 4040–4052 — JOURNAL, JIF'22: 5.1 (Q1)

Murillo, R., Del Barrio, A. A., and Botella, G. A Suite of Division Algorithms for Posit Arithmetic. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2023), vol. 2023-July, IEEE, pp. 41–44 — CONFERENCE PROCEEDINGS, GGS Class: 3 (B)

Murillo, R., Mallasén, D., Del Barrio, A. A., and Botella, G. PLAUs: Posit Logarithmic Approximate Units to Implement Low-Cost Operations with Real Numbers. In *Conference for Next Generation Arithmetic (CoNGA)* (2023), vol. 13851, pp. 171–188 — CONFERENCE PROCEEDINGS

Murillo, R., Del Barrio, A. A., and Botella, G. Efectos de la Precisión Numérica en las Aplicaciones Científicas. In *Jornadas SARTECO* (2022), pp. 663–669 — JOURNAL

Murillo, R., Del Barrio, A. A., and Botella, G. The Effects of Numerical Precision In Scientific Applications. In *2022 Annual Modeling and Simulation Conference (ANNSIM)* (2022), IEEE, pp. 152–163 — CONFERENCE PROCEEDINGS

Murillo, R., Mallasén, D., Del Barrio, A. A., and Botella, G. Comparing Different Decodings for Posit Arithmetic. In *Conference for Next Generation Arithmetic (CoNGA)* (2022), vol. 13253, pp. 84–99 — CONFERENCE PROCEEDINGS

Murillo, R., Del Barrio Garcia, A. A., Botella, G., Kim, M. S., Kim, H., and Bagherzadeh, N. PLAM: a Posit Logarithm-Approximate Multiplier. *IEEE Trans. on Emerging Topics in Computing 10*, 4 (2022), 2079–2085 — JOURNAL, JIF'22: 5.9 (Q1)

Murillo, R., Mallasén, D., Del Barrio, A. A., and Botella, G. Energy-Efficient MAC Units for Fused Posit Arithmetic. In *2021 IEEE 39th International Conference on Computer Design (ICCD)* (2021), pp. 138–145 — CONFERENCE PROCEEDINGS, GGS Class: 2 (A-)

Murillo, R., Del Barrio, A. A., and Botella, G. Posit Arithmetic Units for Deep Neural Networks. In *Jornadas SARTECO* (2021), pp. 617–622 — CONFERENCE PROCEEDINGS

Murillo, R., Del Barrio, A. A., and Botella, G. La Aritmética del Futuro: una Reflexión sobre los Planes de Estudio. *Enseñanza y Aprendizaje de Ingeniería de Computadores* (2020), 49–60 — CONFERENCE PROCEEDINGS

CONFERENCE PROCEEDINGS GGS CLASS: 2 (A-)

MURILLO, R., DEL BARRIO, A. A., AND BOTELLA, G. Customized Posit Adders and Multipliers using the FloPoCo Core Generator. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (2020)

JOURNAL JIF'20: 3.381 (Q2)

MURILLO, R., DEL BARRIO, A. A., AND BOTELLA, G. Deep PeNSieve: A deep learning framework based on the posit number system. *Digital Signal Processing: A Review Journal 102* (2020), 102762

10.2.2. Indirectly related publications

CONFERENCE PROCEEDINGS

MALLASÉN, D., MURILLO, R., DEL BARRIO, A. A., BOTELLA, G., PIÑUEL, L., AND PRIETO-MATIAS, M. PERCIVAL: Deploying Posits and Quire Arithmetic into the CVA6 RISC-V Core. In *Proceedings of the 20th ACM International Conference on Computing Frontiers* (2023), Association for Computing Machinery, pp. 375–376

CONFERENCE PROCEEDINGS

MALLASÉN, D., MURILLO, R., DEL BARRIO, A. A., BOTELLA, G., PIÑUEL, L., AND PRIETO-MATIAS, M. Customizing the CVA6 RISC-V Core to Integrate Posit and Quire Instructions. In *2022 37th Conference on Design of Circuits and Integrated Circuits (DCIS)* (2022), IEEE, pp. 01–06

JOURNAL

MALLASÉN, D., MURILLO, R., DEL BARRIO, A. A., BOTELLA, G., PIÑUEL, L., AND PRIETO-MATIAS, M. PERCIVAL: Open-Source Posit RISC-V Core With Quire Capability. *IEEE Transactions on Emerging Topics in Computing 10*, 3 (2022), 1241–1252

MURILLO MONTERO, R., DEL BARRIO, A. A., AND BOTELLA, G. Template-Based Posit Multiplication for Training and Inferring in Neural Networks. *arXiv e-prints* (2019)

10.3. Open-source repositories

In addition to the scientific contributions, the research conducted in this Ph.D. thesis has yielded several source code projects. The code for all these projects has been compiled into repositories, and they have been released under open licenses, ensuring that the code is accessible to the public, fostering collaboration and allowing anyone to utilize or extend it for their own purposes:

- The software repository for scientific applications (Chapter 3): `https://github.com/RaulMurillo/ANNSIM_22`.
- The Deep PeNSieve software repository (Chapter 4): `https://github.com/RaulMurillo/deep-pensieve`.
- The FloPoCo-posit hardware repository, containing the posit units generated by FloPoCo for addition, subtraction, multiplication (Chapter 5) and fused MAC (Chapter 6), as well as the comparison between posit encodings (Appendix A): `https://github.com/RaulMurillo/Flo-Posit`.
- The posit-HLS repository (Chapter 7): `https://github.com/RaulMurillo/posit-hls`.

- The approximate-posit hardware/software repository (Chapter 8): `https://github.com/RaulMurillo/CoNGA_23`.
- The posit-HUB repository (Chapter 9): `https://github.com/RaulMurillo/hub-posit`.

It is worth mentioning that two of the above repositories have been included in official sites. The FloPoCo sources have been added to the official FloPoCo repository[1]. Also, the proposed Deep PeNSieve framework is mentioned on the official site for posit contributions[2] as a software solution that employs posits.

10.4. Open research lines

The research developed in this Ph.D. thesis has addressed the evaluation of posit arithmetic in software applications, as well as the design and development of the corresponding arithmetic units to provide hardware support for this format. The techniques proposed have tackled different scenarios, from software emulation of the format in consideration, to RTL design and implementation of energy-efficient operators. However, some interesting points of future research have emerged during the evolution of this thesis.

Some of the open research lines are:

- The computation of DNNs in real hardware platforms such as FPGAs. The use of software emulation of posit computation was the main limitation in the experiments conducted in Chapter 4. Computing posits in a hardware platform would increase throughput, enabling the evaluation of larger models with a greater number of parameters.
- The fused MAC operation stands out as one of the key innovations introduced by the posit format. Although this thesis has delved into the design of this unit, the findings indicate that while the accuracy of this component is commendable, it comes at a substantial hardware cost. Hence, optimizing the hardware datapath of this operation holds the potential for noteworthy enhancements in advancing the utilization of this arithmetic format.
- The approach presented in Chapter 7 extends HLS with posit support for all the operations found in floating-point. Nonetheless, the inclusion of the fused MAC operation is currently absent in the suggested workflow. Looking ahead, future work will explore the integration of fused posit arithmetic, utilizing the quire accumulator, into the HLS flow to achieve even more accurate results. Furthermore, the proposed workflow will undergo further optimization to narrow the current gap between posit and floating-point arithmetic.
- One interesting area of future research is the development of posit-based accelerators for deep learning. The work presented in Chapter 7 could be extended in this direction. For instance, ONNX is an existing standard format for exchanging neural network

[1] `https://gitlab.com/flopoco/flopoco/-/tree/posit_utils`
[2] `https://en.wikipedia.org/wiki/Unum_(number_format)`

models. Bambu can be utilized to generate hardware accelerators from ONNX models. Alternatively, the hls4ml tool [37] proposes an approach where the pre-processing step relies on a library of C++ components optimized for HLS. The intermediate representation produced by hls4ml is highly specialized and dependent on Vivado HLS. However, it will be possible to adapt it, so that Bambu can be used instead.

- The posit units presented in this thesis have already been implemented into a RISC-V core with the capability to execute posit-based programs in C language [106]. Future work will explore the implementation of posit units into other RISC-V cores with parallelization capabilities like Sargantana [154] or Ara [13].

A

Interpretations of posit arithmetic

Posit arithmetic has caught the attention of the research community as one of the most promising alternatives to the IEEE 754 Standard for Floating-Point Arithmetic. However, the recentness of the posit format makes its hardware less mature and thus more expensive than the floating-point hardware. Most approaches proposed so far decode posit numbers similarly to classical floats.

However, the posit format was proposed as a hardware-friendly version of Type II unums, keeping reciprocation between negative and positive numbers, in the same manner as signed (two's complement) integers do. By following this design principle, in this thesis, a novel decoding approach for posit numbers is proposed, which in contrast with the previous one, considers negative posits to have a negative fraction. This alternative interpretation is proven to be equivalent to the float-like decoding proposed in previous works. Also, in this chapter, a generic implementation for the latter is presented and offers comparisons of posit addition and multiplication units based on both schemes.

ASIC synthesis reveals that this alternative approach enables a faster way to perform operations while reducing the area, power and energy of the FUs. What is more, the proposed posit operators are shown to improve the state-of-the-art implementations in terms of area, power and energy consumption.

A.1. Sign-magnitude interpretation

Posit arithmetic is a floating-point format for representing real numbers. Thus, the numerical value p of a normal posit datum was initially defined in [52] by Equation (A.1)

$$p = (-1)^s \times \left(2^{2^{es}}\right)^r \times 2^e \times (1+f), \tag{A.1}$$

where s is the sign, e is the integer encoded by the exponent field, r is the value encoded in the regime field as indicated by Equation (1.3), and f is the normalized fraction (this is, the value encoded by the fraction bits normalized, so $0 \leqslant f < 1$). Under this decoding approach, if a value is negative (when the sign bit is 1), its two's complement is computed

before extracting the regime, exponent, and fraction, so values e, r and f in Equation (A.1) are always considered from the absolute value of the posit.

The main differences with the standard floating-point format are the utilization of an unsigned and unbiased exponent, the hidden bit of the significand is always "1" (no subnormal numbers are considered), and the existence of the variable-length regime field. However, notice that this decoding is quite similar to the one for classical floating-point numbers: it deals with a sign bit, a signed exponent (regime and exponent can be gathered in a single factor) and a significand with a hidden bit. As a consequence, the core of the circuit design for both arithmetic formats would be similar too. Indeed, this float-like decoding scheme is the one used by most posit arithmetic units from the literature [15, 72, 113], as well as by the approximate posit units proposed so far [121, 133].

A.2. Two's complement interpretation

The previous decoding scheme of posit numbers deals with negative numbers in a similar manner as signed integers do. From a hardware perspective, converting posits to their absolute value before decoding them adds an extra area and performance overhead, especially when compared with IEEE 754 floats. To address this issue, Isaac Yonemoto, co-author of [52], proposes a different way of decoding posit numbers: for negative values, the most significant digit of the significand is treated as "-2" instead of "1". The rest of the fields remain the same, but under this approach, there is no need to compute the two's complement (absolute value) of each negative posit. This is consistent with the way posits were initially intended, as a mapping of the signed (two's complement) integers to the projective reals. The value p of a posit number is now given by Equation (A.2)

$$p = ((1 - 3s) + f) \times 2^{(1-2s)\times(2^{es}r+e+s)}. \tag{A.2}$$

When considering this approach, the significand $(1 - 3s + f)$ belongs to the interval $[-2, -1)$ for negative posits and to $[1, 2)$ for positive ones, so such signed fixed-point representation requires two integer (or hidden) bits that depend on the sign of the posit. More precisely, in this case, negative posits prepend `10` to the fraction bits as the two's complement hidden bits, while positive posits prepend `01`. Note how this contrasts with the unsigned fixed-point representation of the significand in the floating-point and previous sign-magnitude posit formats interpretation. Therefore, this approach eliminates complexity in the decoding and encoding stages but requires redesigning some logic when implementing posit operators.

A.3. Equivalence between interpretations

Previous works have examined the effect of two's complement notations in floating-point arithmetic [6]. However, in such a case, some features or properties are lost with respect to the IEEE 754 standard for floats. Next, a proof is presented establishing the equivalence of both sign-magnitude and two's complement interpretations of posit numbers. Therefore, all properties are preserved regardless of the used approach. The impact of each approach is found on the hardware implementation, as will be discussed in Appendix A.4.

Theorem 1. *For any given posit bit string that encodes a number other than zero or NaR, Equation* (A.1) *and Equation* (A.2) *are equivalent.*

Proof. When the sign bit is 0 (i.e. the bit string encodes a positive number), it is trivial that both expressions evaluate the same.

On the other hand, the case when $s = 1$ requires more attention. First, note that in such a case, two's complement of the bit string must be computed before evaluating Equation (A.1). Hence, all the bits at the left of the rightmost 1 are inverted. Let us denote by $\tilde{x}$ the two's complement of the bit field x, and by $\bar{x}$ the one's complement (or bitwise negation) of x. Three cases have to be considered, depending on which field that bit belongs to.

(I) If the rightmost 1 bit belongs to the fraction field, the fraction $f \neq 0$. In such a case, when computing the two's complement of the bitstring, both regime and exponent are bitwise flipped (so we have $\bar{r}$ and $\bar{e}$); just the two's complement of f ($\tilde{f}$) is required. Hence, Equation (A.1) turns into Equation (A.3)

$$p = (-1)^1 \times \left(2^{2^{es}}\right)^{\bar{r}} \times 2^{\bar{e}} \times (1 + \tilde{f}). \tag{A.3}$$

Due to the fact that $0 \leqslant f < 1$, it yields $\tilde{f} = 1 - f$. On the other hand, regime and exponent fields are bitwise flipped, so from equation Equation (1.3) we have that $\bar{r} = -r - 1$, and since the exponent is a non-negative value represented with es bits, then $\bar{e} = 2^{es} - 1 - e$. Substituting this expression in Equation (A.3) yields to Equation (A.2).

(II) If the rightmost 1 bit belongs to the exponent field, then $f = 0$ and $e \neq 0$. For the same reason as in the previous case, we have $\bar{r} = -r - 1$. But now the exponent field for expression Equation (A.1) is in two's complement rather than inverted, so we have $\tilde{e} = \bar{e} + 1 = 2^{es} - e$. On the other hand, $f = 0$, so the significand in expression Equation (A.1) evaluates 1, while Equation (A.2) evaluates the significand as -2. This compensates for the difference in the exponents.

(III) If the rightmost 1 bit belongs to the regime, then $e = f = 0$. Note that such a bit corresponds either to the last regime bit (when the regime is a sequence of 0's) or to the second last bit of the regime (when it is a sequence of 1's). In both cases, taking the two's complement of the regime field before computing Equation (A.1) has the effect of reversing the sequence of identical bits and modifying its size by one bit. This, according to Equation (1.3), yields $\tilde{r} = -r$. Substituting this in Equation (A.1) gives an exponent of $-2^{es}r$. On the other hand, substituting $s = 1$ and $e = f = 0$ in Equation (A.2) results in $-2^{es}r - 1$ as exponent. Nevertheless, since the fraction is zero, in this latter case the significand is evaluated as -2, which compensates for the difference of exponents.

□

A.4. Hardware evaluation

There are not many works that implement posit numbers under the two's complement interpretation. The first implementation of posit adders and multipliers based on this

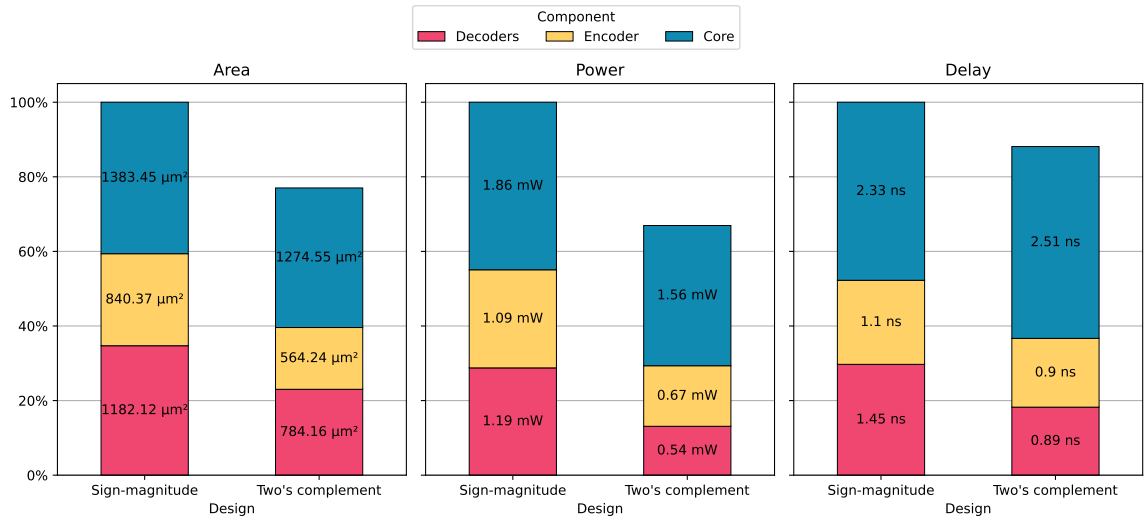


Figure A.1: Hardware synthesis for Posit32 adders.

interpretation appeared in [161], and more details about such a scheme were introduced in [48]. Also, [123] presents different energy-efficient fused posit MAC units that follow the same approach as [161].

This section evaluates the hardware impact of each posit interpretation. To achieve an accurate evaluation, combinational units for each interpretation approach are implemented following state-of-the-art optimizations. The units based on the two's complement interpretation are the ones presented in Chapter 5. The designs are synthesized targeting a 45 nm TSMC standard-cell library with no timing constraint and typical case parameters using Synopsys Design Compiler®.

Synthesis results for Posit32 addition units are shown in Figure A.1. In order to provide a more fine-grained evaluation, the hardware requirements of each individual component are shown separately. As can be seen, the two's complement approach reduces substantially the hardware requirements for both decoder and encoder modules. But also, using two's complement interpretation simplifies the core of the addition operator: in contrast with the sign-magnitude approach, which requires detecting the larger operand and performing addition or subtraction according to the signs of the inputs, with the two's complement notation it just needs to check for the value with larger binade, and the resulting sign can be extracted from the addition of fractions, since those are already signed. This results in reductions of 23%, 33% and 12% in terms of area, power and delay, respectively.

The results for multiplier units are shown in Figure A.2. In this case, although the savings in hardware resources in the decoder and encoder modules are preserved when using a two's complement design, the total savings are less than in the previous case of the addition. The reason for this is in the core of the multiplication unit. When using a two's complement design, not only can significant improvements not be made to the core of the multiplier, but the fractions to be multiplied have an extra hidden bit, so the hardware multiplier will have to be larger, which will have a negative impact on the area of the unit. Anyway, the improvements introduced in the decoder and encoder modules of the two's complement

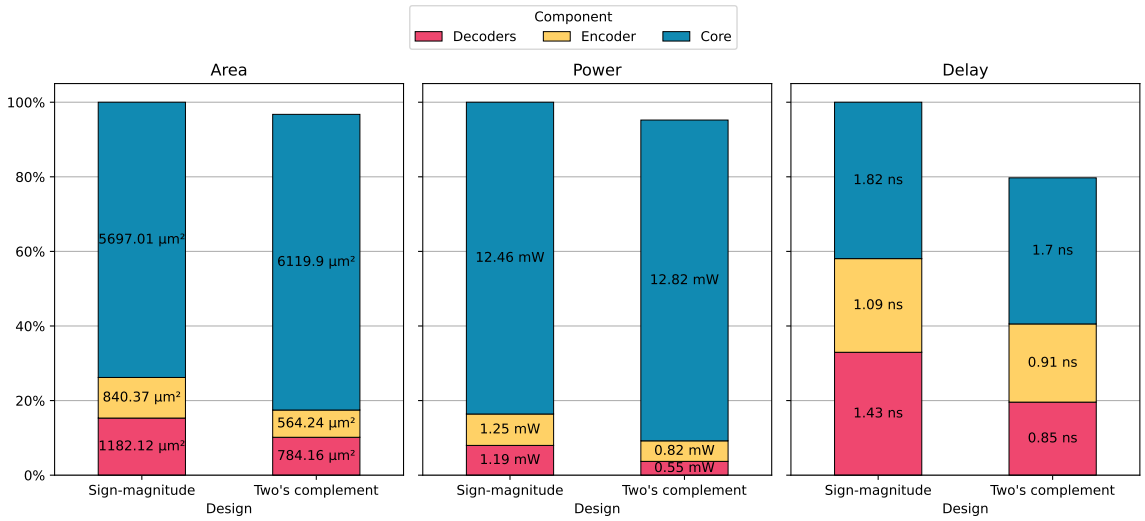


Figure A.2: Hardware synthesis for Posit32 multipliers.

designs compensate the multiplier overhead, thus obtaining reductions of 3.3%, 4.8% and 20% in the area, power and delay of the whole unit.

B

Third-party software

The software used in this Ph.D. thesis can be divided into two different groups, each related to a different part of this document. All the experiments carried out in Part I, related to software applications were done by emulating posit computation via software, while the experiments shown in Part II relied on hardware design tools. To evaluate the new data formats presented in Part III both software emulation and hardware-oriented tools were used.

B.1. Posit emulation software

The main library used to emulate computations with posit arithmetic used along this thesis (primarily in the experimental section of Chapter 3) is Universal [137]. Universal Numbers Library, or simply Universal, is a header-only C++ template library that provides implementations of various number representations and standard arithmetic operations on arbitrary configurations. It includes posit and floating-point number systems, as well as other less common formats like fixed-points, linear floats, SORNs, or interval and adaptive-precision integers. As data types, operators and implicit type conversions are properly defined in the library, it is just necessary to include the necessary parts of the library to emulate the desired arithmetic types, as shown in Listing B.1.

Due to the template-based design of the library, it is straightforward to evaluate the same algorithms under a wide variety of arithmetic formats and precisions. This library has also been used to generate the corresponding testbenches for verification of the proposed arithmetic units in Part II. Moreover, the library provides functionality for manipulating the emulated data types at the bit-level. Therefore, it has also been used to carry on the experiments in Chapters 8 and 9.

```
// bring in the parameterized type of interest, in this case
// a fixed-sized, arbitrary configuration posit number system
#include <universal/number/posit/posit.hpp>

// define your computational kernel parameterized by arithmitic type
template<typename Real>
Real MyKernel(const Real& a, const Real& b) {
    return a * b;  // replace this with your kernel computation
}

constexpr double pi = 3.14159265358979323846;

int main() {
    using Posit32 = sw::universal::posit<32,2>; // 32-bit posit with 2 exponent bits

    Posit32 a = sqrt(2);
    Posit32 b = pi;
    // finally, call your kernel with your desired arithmetic type
    std::cout << "Result: " << MyKernel(a, b) << std::endl;
}
```

Listing B.1: Example program using posits with Universal.

Deep learning

Regarding computations with DNNs, two different tools have been used in this thesis. While both of them allow us to perform either inference or training of deep learning models, it is worth mentioning the differences among them.

The framework proposed in this thesis, Deep PeNSieve, relies on other two libraries: SoftPosit [97] and a modified version of TensorFlow [1][1]. This customized version of TensorFlow relies on SoftPosit as well, and allows implementing neural network architectures using posit datatype for the weights and biases, as shown in Listing B.2. Models declared in Deep PeNSieve can be trained with the usual TensorFlow commands. Such an implementation ensures that all the internal computations within the inference/training phase are emulated in posit arithmetic with SoftPosit.

Moreover, Deep PeNSieve allows performing inference using the quire accumulator for matrix multiplications. This is implemented on top of SoftPosit functionality for fused MAC, while wrapped as a layer module within TensorFlow. Therefore, it just requires substituting the fully connected and convolutional layers within the original model with the corresponding fused layers, while the weights and biases remain unchanged. This simplifies the programming a lot. However, although TensorFlow supports running computations on GPU, the posit is emulated, so the computations are just on CPUs and the performance of this framework is not comparable to that obtained with floating-point data, and the models that can be evaluated on inference/training are limited to a couple of hidden layers.

On the other hand, Qtorch+ [59] is used to train larger DNN models on GPU. This tool is based on QPyTorch [176], a low-precision arithmetic simulation framework, which is also built on top of PyTorch [138]. Qtorch+ introduces posit data support with round to nearest mode into QPyTorch, including Posit$\langle n, es \rangle$, $n \leqslant 16$, formats. QPyTorch supports quantizing different numbers in the training process with customized low-precision formats.

[1] https://github.com/xman/tensorflow/tree/posit-1.11.0

```
import tensorflow as tf

tf_type = tf.posit16

def LeNet(x):
    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), dtype=tf_type))
    conv1_b = tf.Variable(tf.zeros(6, dtype=tf_type))
    conv1 = tf.nn.conv2d(x, conv1_W,
                         strides=[1, 1, 1, 1], padding='VALID') + conv1_b
    conv1 = tf.nn.relu(conv1)
    # Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='VALID')
    # Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), type=tf_type))
    conv2_b = tf.Variable(tf.zeros(16, dtype=tf_type))
    conv2 = tf.nn.conv2d(conv1, conv2_W,
                         strides=[1, 1, 1, 1], padding='VALID') + conv2_b
    conv2 = tf.nn.relu(conv2)
    # Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='VALID')
    # Flatten. Input = 5x5x16. Output = 400.
    fc0 = tf.contrib.layers.flatten(conv2)
    # Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1_W = tf.Variable(tf.truncated_normal( shape=(400, 120), dtype=tf_type))
    fc1_b = tf.Variable(tf.zeros(120, dtype=tf_type))
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b
    fc1 = tf.nn.relu(fc1)
    # Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W = tf.Variable(tf.truncated_normal( shape=(120, 84), dtype=tf_type))
    fc2_b = tf.Variable(tf.zeros(84, dtype=tf_type))
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b
    fc2 = tf.nn.relu(fc2)
    # Layer 5: Fully Connected. Input = 84. Output = 10.
    fc3_W = tf.Variable(tf.truncated_normal( shape=(84, 10), dtype=tf_type))
    fc3_b = tf.Variable(tf.zeros(10, dtype=tf_type))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

Listing B.2: Example of DNN model declaration with Deep PeNSieve.

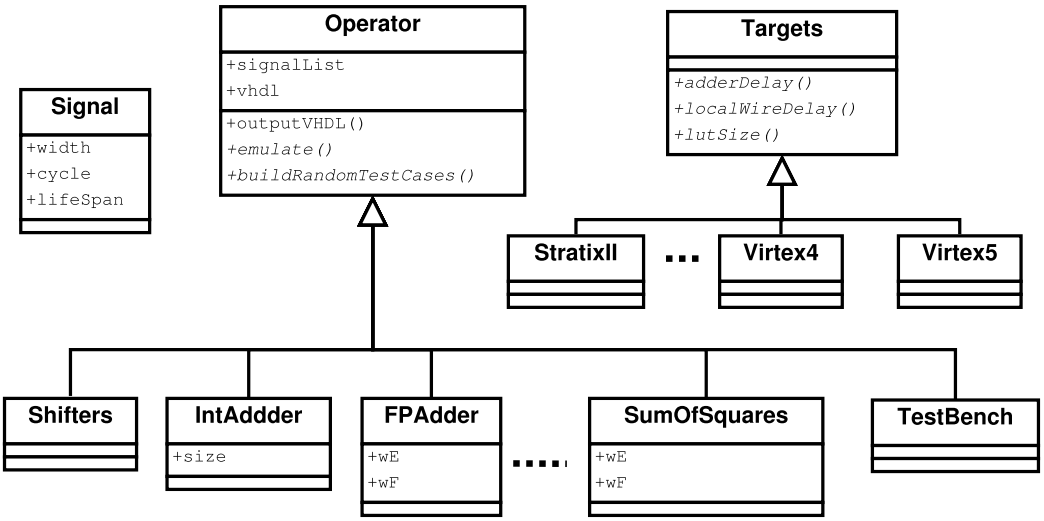


Figure B.1: Simplified overview of the FloPoCo class hierarchy.
Source: [33].

This is achieved with pre- and post-quantization layers, which can be applied to activations, weights and gradients without the need to modify the original model.

However, QPyTorch relies on PyTorch functions for the underlying computation, such as matrix multiplication. This means that the actual computation is done in single-precision floating-point. Therefore, QPyTorch is not intended to be used to study the numerical behavior of different accumulation strategies. Consequently, although Qtorch+ can leverage the fast GPU computation, it does not offer precise results. The authors of this tool argue that they assume the dot product is done using the quire and the output format has enough precision to hold the output value, while this might not be as accurate as using an actual quire accumulator or any software that emulates it like Deep PeNSieve.

B.2. Hardware design tools

The proposed designs have been implemented into FloPoCo, an open-source C++ framework for the generation of arithmetic datapaths. FloPoCo works with a command-line interface that inputs operator specifications and outputs synthesizable VHDL [33].

This tool facilitates the automatic generation of operators with specified parameters, enabling the creation of posit operators for arbitrary values of Posit$\langle n, es \rangle$ while maintaining a consistent base design. FloPoCo adopts a design approach based on simple object-oriented concepts. As depicted in Figure B.1, each datapath in the design is represented as an `Operator`. To implement posit units within FloPoCo, new instances are created by inheriting from the `Operator` class. Additionally, various pre-implemented subcomponents, such as shifters and integer adders, can be utilized as needed. When FloPoCo is executed, it constructs a list of `Operators` based on those specified on the command line, along with all their sub-components. Subsequently, it generates VHDL code for all of them. The tool also builds up pipeline information, according to the specified frequency and target device. Then the `outputVHDL()` method merges the VHDL stream with the pipeline information to

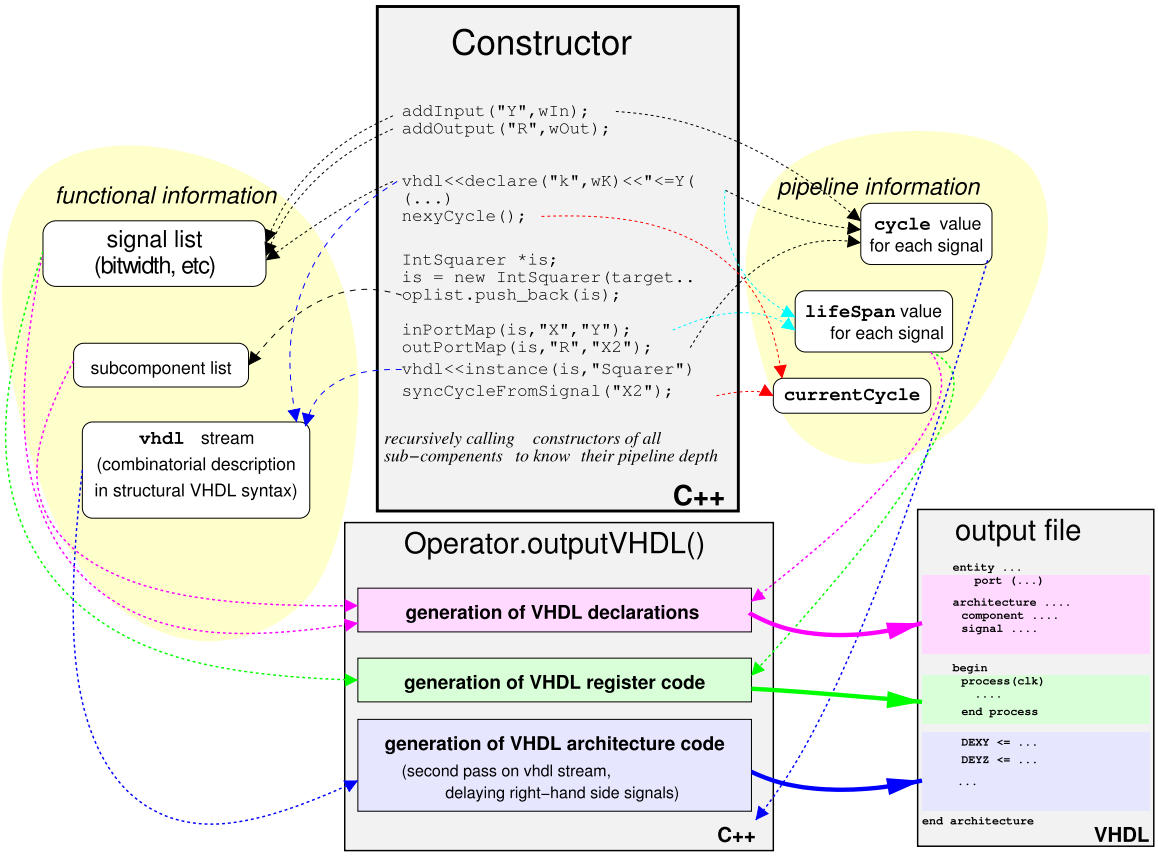


Figure B.2: VHDL generation flow with FloPoCo.
Source: [33].

produce the final VHDL code for the pipelined datapath. Additionally, this method declares all necessary VHDL signals, entities, components, and other elements, streamlining the design process for VHDL code architects. Figure B.2 provides an overview of this VHDL generation flow.

The proposed designs for posit FUs, which include support for correct rounding, have been integrated into the official FloPoCo git repository, and the generated VHDL instances are publicly available to facilitate their use and dissemination. Simulations with extensive testing vectors are performed to verify the functionality of the proposed designs. These testing vectors have been generated with the Universal reference library [137].

Chapter 7 leverages another tool for hardware-software codesign, Bambu, which is an open-source research framework for HLS [139, 40]. Bambu supports various target FPGAs, allowing the generated accelerator to be programmed onto a board. Notably, it is designed with a high degree of modularity and includes support for floating-point operations through integration with FloPoCo. The selection of Bambu as the HLS tool for this work is motivated by its open-source philosophy and seamless integration with FloPoCo.

Bambu takes a behavioral description of the specification, typically written in C/C++, as input and generates the HDL description of the corresponding RTL implementation. The

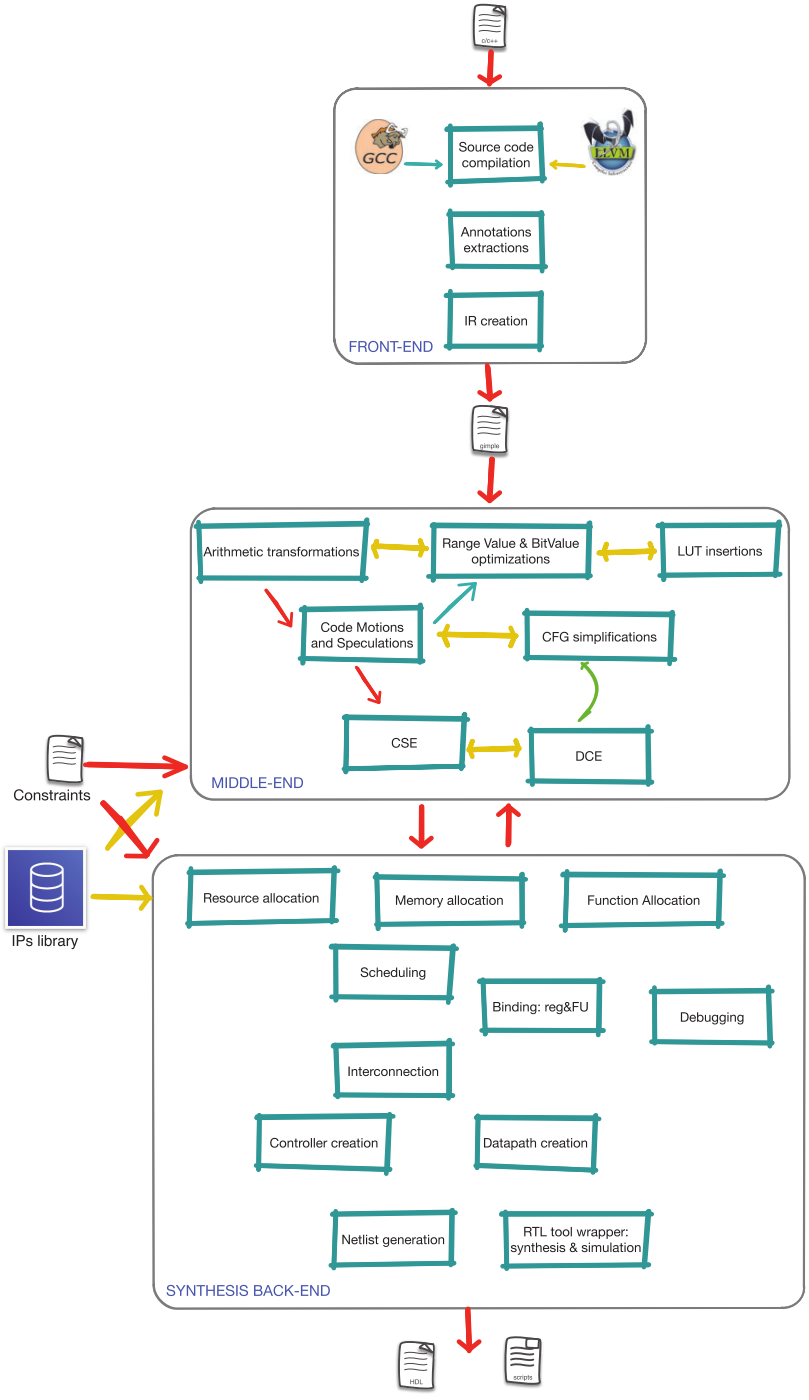


Figure B.3: Bambu compilation flow.
Source: [40].

output is compatible with commercial RTL synthesis tools, accompanied by a testbench for simulating and validating the behavior. Such a workflow is depicted in Figure B.3.

Similar to the software compilation flow, this HLS flow comprises three distinct phases: front-end, middle-end, and back-end. In the front-end, the input code undergoes parsing and translation into an intermediate representation, which is then utilized in the subsequent parts of the flow. The middle-end encompasses target-independent analyses and optimizations. Finally, in the back-end, the actual HLS of the specification is executed. It is during this phase that the workflow modifications detailed in Chapter 7 have been implemented.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A system for large-scale machine learning. In *2016 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016), USENIX Association, pp. 265–283.

[2] Alouani, I., Ben Khalifa, A., Merchant, F., and Leupers, R. An Investigation on Inherent Robustness of Posit Data Representation. *Proceedings of the IEEE International Conference on VLSI Design 2021-Febru* (2021), 276–281.

[3] Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J. I., Fadhel, M. A., Al-Amidie, M., and Farhan, L. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data 8*, 53 (2021).

[4] Arm Ltd. BFloat16 processing for Neural Networks on Armv8-A, 2019.

[5] Bianco, S., Cadene, R., Celona, L., and Napoletano, P. Benchmark Analysis of Representative Deep Neural Network Architectures. *IEEE Access 6* (2018), 64270–64277.

[6] Boldo, S., and Daumas, M. Properties of two's complement floating point notations. *International Journal on Software Tools for Technology Transfer 5* (2004), 237–246.

[7] Botella, G., Garcia, A., Rodriguez-Alvarez, M., Ros, E., Meyer-Baese, U., and Molina, M. C. Robust Bioinspired Architecture for Optical-Flow Computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 18*, 4 (2010), 616–629.

[8] Brent, and Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers C-31*, 3 (1982), 260–264.

[9] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language Models are Few-Shot Learners. *arXiv e-prints* (2020).

[10] Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., and Mansell, D. Bfloat16 Processing for Neural Networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)* (2019), pp. 88–91.

[11] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (2011), ACM, pp. 33–36.

[12] Carmichael, Z., Langroudi, H. F., Khazanov, C., Lillie, J., Gustafson, J. L., and Kudithipudi, D. Deep Positron: A Deep Neural Network Using the Posit Number System. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2019), IEEE, pp. 1421–1426.

[13] Cavalcante, M., Schuiki, F., Zaruba, F., Schaffner, M., and Benini, L. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 28*, 2 (2020), 530–543.

[14] Chatterjee, R., Chowdhury, S., Mondal, S., Raha, A., Paluh, J., and Mukherjee, A. PreSyNC: Hardware realization of the Presynaptic Region of a Biologically Extensive Neuronal Circuitry. *Proceedings of the IEEE International Conference on VLSI Design 2021* (2021), 228–233.

[15] Chaurasiya, R., Gustafson, J. L., Shrestha, R., Neudorfer, J., Nambiar, S., Niyogi, K., Merchant, F., and Leupers, R. Parameterized Posit Arithmetic Hardware Generator. In *2018 IEEE 36th International Conference on Computer Design (ICCD)* (2018), IEEE, pp. 334–341.

[16] Chen, C., Qian, W., Imani, M., Yin, X., and Zhuo, C. PAM: A Piecewise-Linearly-Approximated Floating-Point Multiplier With Unbiasedness and Configurability. *IEEE Transactions on Computers 71*, 10 (2022), 2473–2486.

[17] Chen, J., Al-Ars, Z., and Hofstee, H. P. A matrix-multiply unit for posits in reconfigurable logic leveraging (Open)CAPI. In *Conference for Next Generation Arithmetic (CoNGA)* (2018), ACM.

[18] Cheng, T., Masuda, Y., Chen, J., Yu, J., and Hashimoto, M. Logarithm-approximate floating-point multiplier is applicable to power-efficient neural network training. *Integration 74* (2020), 19–31.

[19] Chi, Y., Qiao, W., Sohrabizadeh, A., Wang, J., and Cong, J. Democratizing Domain-Specific Computing. *Communications of the ACM 66*, 1 (2022), 74–85.

[20] Chien, S. W. D., Peng, I. B., and Markidis, S. Posit NPB: Assessing the Precision Improvement in HPC Scientific Applications. In *Parallel Processing and Applied Mathematics (PPAM)* (2020), vol. 12043 LNCS, pp. 301–310.

[21] Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Analysis and characterization of inherent application resilience for approximate computing. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2013).

[22] Chowdhary, S., Lim, J. P., and Nagarakatte, S. Debugging and detecting numerical errors in computation with posits. In *2020 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2020), ACM, pp. 731–746.

[23] Ciocirlan, S. D., Loghin, D., Ramapantulu, L., Tapus, N., and Teo, Y. M. The Accuracy and Efficiency of Posit Arithmetic. In *2021 IEEE 39th International Conference on Computer Design (ICCD)* (2021), IEEE, pp. 83–87.

[24] Cococcioni, M., Rossi, F., Ruffaldi, E., and Saponara, S. Fast Approximations of Activation Functions in Deep Neural Networks when using Posit Arithmetic. *Sensors 20*, 5 (2020), 1515.

[25] Cococcioni, M., Rossi, F., Ruffaldi, E., and Saponara, S. Vectorizing posit operations on RISC-V for faster deep neural networks: experiments and comparison with ARM SVE. *Neural Computing and Applications 33*, 16 (2021), 10575–10585.

[26] Cococcioni, M., Rossi, F., Ruffaldi, E., and Saponara, S. A Lightweight Posit Processing Unit for RISC-V Processors in Deep Neural Network Applications. *IEEE Transactions on Emerging Topics in Computing 10*, 4 (2022), 1898–1908.

[27] Coussy, P., Gajski, D., Meredith, M., and Takach, A. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers 26*, 4 (2009), 8–17.

[28] Coussy, P., and Morawiec, A. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008.

[29] Crespo, L., Tomas, P., Roma, N., and Neves, N. Unified Posit/IEEE-754 Vector MAC Unit for Transprecision Computing. *IEEE Transactions on Circuits and Systems II: Express Briefs 69*, 5 (2022), 2478–2482.

[30] Da Silva, S. S., Cardoso, M., Nardo, L., Nepomuceno, E., Hübner, M., and Arias-Garcia, J. A New Chaos-Based PRNG Hardware Architecture Using the HUB Fixed-Point Format. *IEEE Transactions on Instrumentation and Measurement 72* (2023).

[31] Dally, W. J., Turakhia, Y., and Han, S. Domain-Specific Hardware Accelerators. *Communications of the ACM 63*, 7 (2020), 48–57.

[32] de Dinechin, F., Forget, L., Muller, J.-M., and Uguen, Y. Posits: the good, the bad and the ugly. In *Conference for Next Generation Arithmetic (CoNGA)* (Singapore, Singapore, 2019), ACM.

[33] de Dinechin, F., and Pasca, B. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers 28*, 4 (2011), 18–27.

[34] Dean, J., Patterson, D., and Young, C. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro 38*, 2 (2018), 21–29.

[35] Del Barrio, A. A., Hermida, R., and Memik, S. O. Exploring the energy efficiency of Multispeculative Adders. In *2013 IEEE 31st International Conference on Computer Design (ICCD)* (2013), pp. 309–315.

[36] Dua, D., and Graff, C. UCI Machine Learning Repository, 2017.

[37] Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., Ngadiuba, J., Pierini, M., Rivera, R., Tran, N., and Wu, Z. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation 13*, 07 (2018), P07027–P07027.

[38] Ercegovac, M. D., and Lang, T. *Digital Arithmetic*. Elsevier, 2004.

[39] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 301–312.

[40] Ferrandi, F., Castellana, V. G., Curzel, S., Fezzardi, P., Fiorito, M., Lattuada, M., Minutoli, M., Pilato, C., and Tumeo, A. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA, 2021), IEEE, pp. 1327–1330.

[41] Gavarini, G., Ruospo, A., and Sanchez, E. On the resilience of representative and novel data formats in CNNs. In *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2023), IEEE.

[42] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. A Survey of Quantization Methods for Efficient Neural Network Inference. In *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[43] Glint, T., Prasad, K., Dagli, J., Gandhi, K., Gupta, A., Patel, V., Shah, N., and Mekie, J. Hardware-Software Codesign of DNN Accelerators Using Approximate Posit Multipliers. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference* (2023), ASPDAC'23, Association for Computing Machinery, pp. 469–474.

[44] Gohil, V., Walia, S., Mekie, J., and Awasthi, M. Fixed-Posit: A Floating-Point Representation for Error-Resilient Applications. *IEEE Transactions on Circuits and Systems II: Express Briefs 68*, 10 (2021), 3341–3345.

[45] Goldberg, D. What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Computing Surveys (CSUR) 23*, 1 (1991), 5–48.

[46] Google LLC. BFloat16: The secret to high performance on Cloud TPUs, 2019.

[47] Gunaratne, T. K. Evaluation of the Use of Low Precision Floating-Point Arithmetic for Applications in Radio Astronomy. In *Conference for Next Generation Arithmetic (CoNGA)* (2023), vol. 13851 LNCS, Springer Nature Switzerland, pp. 155–170.

[48] Guntoro, A., De La Parra, C., Merchant, F., De Dinechin, F., Gustafson, J. L., Langhammer, M., Leupers, R., and Nambiar, S. Next Generation Arithmetic for Edge Computing. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2020), pp. 1357–1365.

[49] Gustafson, J. L. *The End of Error*. Chapman and Hall/CRC, 2015.

[50] Gustafson, J. L. A Radical Approach to Computation with Real Numbers. *Supercomputing Frontiers and Innovations 3* (2016), 38–53.

[51] Gustafson, J. L., Shin, G. B., Chung, Y., Dimitrov, V., Siew, G. J., Leong, H., Lindstrom, P., Omtzigt, T., Rehr, H., Shewmaker, A., and Yonemoto, I. Standard for Posit™ Arithmetic. Standard, Posit Working Group, 2022. Online.

[52] Gustafson, J. L., and Yonemoto, I. T. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations 4*, 2 (2017).

[53] Han, J., and Orshansky, M. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)* (2013).

[54] He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), vol. 2016-Decem, IEEE, pp. 770–778.

[55] He, K., Zhang, X., Ren, S., and Sun, J. Identity Mappings in Deep Residual Networks. In *European Conference on Computer Vision (ECCV 2016)* (2016), Springer, Ed., Springer International Publishing, pp. 630–645.

[56] Hennessy, J. L., and Patterson, D. A. A New Golden Age in Computer Architecture. *Communications of the ACM 62*, 2 (2019), 48–60.

[57] Henry, G., Tang, P. T. P., and Heinecke, A. Leveraging the bfloat16 Artificial Intelligence Datatype for Higher-Precision Computations. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)* (2019), IEEE, pp. 69–76.

[58] Ho, N.-m., Nguyen, D.-t., Silva, H. D., Gustafson, J. L., Wong, W.-f., and Chang, I. J. Posit Arithmetic for the Training and Deployment of Generative Adversarial Networks. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), IEEE, pp. 1350–1355.

[59] Ho, N.-M., Silva, H. D., Gustafson, J. L., and Wong, W.-F. Qtorch+: Next Generation Arithmetic for Pytorch Machine Learning. In *Conference for Next Generation Arithmetic (CoNGA)* (2022), Lecture Notes in Computer Science, Springer, pp. 31–49.

[60] Hormigo, J., and Villalba, J. Optimizing DSP circuits by a new family of arithmetic operators. In *2014 48th Asilomar Conference on Signals, Systems and Computers* (2014), pp. 871–875.

[61] Hormigo, J., and Villalba, J. Measuring Improvement When Using HUB Formats to Implement Floating-Point Systems under Round-to-Nearest. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 24*, 6 (2016), 2369–2377.

[62] Hormigo, J., and Villalba, J. New Formats for Computing with Real-Numbers under Round-to-Nearest. *IEEE Transactions on Computers 65*, 7 (2016), 2158–2168.

[63] Hormigo, J., and Villalba, J. HUB Floating Point for Improving FPGA Implementations of DSP Applications. *IEEE Transactions on Circuits and Systems II: Express Briefs 64*, 3 (2017), 319–323.

[64] IEEE Computer Society. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985).

[65] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008 (Revision of IEEE 754-1985) 2008* (2008).

[66] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008) 2019* (2019).

[67] Ilangovan P., P., Rayan, R., and Saxena, V. S. Improving the Stability of Kalman Filters with Posit Arithmetic. In *Conference for Next Generation Arithmetic (CoNGA)* (2023), vol. 13851 LNCS, Springer Nature Switzerland, pp. 134–154.

[68] Imani, M., Peroni, D., and Rosing, T. CFPU: Configurable floating point multiplier for energy-efficient computing. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2017).

[69] Immaneni, A., Ullah, S., Nambi, S., Sahoo, S. S., and Kumar, A. PosAx-O: Exploring Operator-level Approximations for Posit Arithmetic in Embedded AI/ML. In *2022 25th Euromicro Conference on Digital System Design (DSD)* (2022), IEEE, pp. 214–223.

[70] Intel Corporation. BFLOAT16 - Hardware Numerics Definition, 2018.

[71] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2018), IEEE Computer Society, pp. 2704–2713.

[72] Jaiswal, M. K., and So, H. K. PACoGen: A Hardware Posit Arithmetic Core Generator. *IEEE Access 7* (2019), 74586–74601.

[73] Johnson, J. Rethinking floating point for deep learning. *arXiv e-prints* (2018).

[74] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ISCA'17.

[75] Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., et al. Highly accurate protein structure prediction with AlphaFold. *Nature 596*, 7873 (2021), 583–589.

[76] Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., and Dubey, P. A Study of BFLOAT16 for Deep Learning Training. *arXiv e-prints* (2019).

[77] Kim, H., Kim, M. S., Del Barrio, A. A., and Bagherzadeh, N. A Cost-Efficient Iterative Truncated Logarithmic Multiplication for Convolutional Neural Networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)* (2019), pp. 108–111.

[78] Kim, M. S., Del Barrio, A. A., Oliveira, L. T., Hermida, R., and Bagherzadeh, N. Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks. *IEEE Transactions on Computers 68*, 5 (2019), 660–675.

[79] Klöwer, M., Coveney, P. V., Paxton, E. A., and Palmer, T. N. Periodic orbits in chaotic systems simulated at low precision. *Scientific Reports 13*, 1 (2023), 11410.

[80] Klöwer, M., Düben, P. D., and Palmer, T. N. Posits as an alternative to floats for weather and climate models. In *Conference for Next Generation Arithmetic (CoNGA)* (New York, NY, USA, 2019), ACM.

[81] Klöwer, M., Düben, P. D., and Palmer, T. N. Number Formats, Error Mitigation, and Scope for 16-Bit Arithmetics in Weather and Climate Modeling Analyzed With a Shallow Water Model. *Journal of Advances in Modeling Earth Systems 12*, 10 (2020).

[82] Kogge, P. M., and Stone, H. S. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers C-22*, 8 (1973), 786–793.

[83] Koren, I. *Computer Arithmetic Algorithms*. A K Peters/CRC Press, 2018.

[84] Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv e-prints* (2018).

[85] Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. Master's thesis, University of Toronto, 2009.

[86] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. *Communications of the ACM 60* (2017), 84–90.

[87] Kulisch, U. *Computer Arithmetic and Validity*. De Gruyter, 2008.

[88] Kulkarni, A., Pattanshetty, S., Raveendran, A., Selvakumar, D., Jean, S., and Desalphine, V. PositGen-A Verification Suite for Posit Arithmetic. In *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)* (2021), IEEE, pp. 204–209.

[89] Langroudi, H. F., Carmichael, Z., and Kudithipudi, D. Deep Learning Training on the Edge with Low-Precision Posits. *arXiv e-prints* (2019).

[90] Langroudi, H. F., Carmichael, Z., Pastuch, D., and Kudithipudi, D. Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge. *arXiv e-prints* (2019).

[91] Langroudi, H. F., Karia, V., Carmichael, Z., Zyarah, A., Pandit, T., Gustafson, J. L., and Kudithipudi, D. ALPS: Adaptive Quantization of Deep Neural Networks with GeneraLized PositS. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (2021), pp. 3100–3109.

[92] Langroudi, H. F., Karia, V., Gustafson, J. L., and Kudithipudi, D. Adaptive Posit: Parameter aware numerical format for deep learning inference on the edge. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2020), IEEE, pp. 3123–3131.

[93] Langroudi, S. H. F., Pandit, T., and Kudithipudi, D. Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)* (Williamsburg, VA, USA, 2018), IEEE, pp. 19–23.

[94] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature 521* (2015), 436–444.

[95] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86* (1998), 2278–2323.

[96] Ledoux, L., and Casas, M. A Generator of Numerically-Tailored and High-Throughput Accelerators for Batched GEMMs. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2022).

[97] Leong, S. H. SoftPosit. `https://gitlab.com/cerlane/SoftPosit`, 2020. Online.

[98] Leong, S. H., and Gustafson, J. L. Lossless FFTs Using Posit Arithmetic. In *Next Generation Arithmetic (CoNGA)* (2023), vol. 13851 LNCS, Springer Nature Switzerland, pp. 1–18.

[99] Lindstrom, P. Variable-Radix Coding of the Reals. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)* (2020), vol. 2020-June, IEEE, pp. 111–116.

[100] Lotrič, U., and Bulić, P. Applicability of approximate multipliers in hardware neural networks. *Neurocomputing 96* (2012), 57–65.

[101] Lu, J., Fang, C., Xu, M., Lin, J., and Wang, Z. Evaluations on Deep Neural Networks Training Using Posit Number System. *IEEE Transactions on Computers 70* (2021), 174–187.

[102] Lu, J., Lu, S., Wang, Z., Fang, C., Lin, J., Wang, Z., and Du, L. Training Deep Neural Networks Using Posit Number System. In *2019 32nd IEEE International System-on-Chip Conference (SOCC)* (2019), IEEE, pp. 62–67.

[103] Mallasén, D., Del Barrio, A. A., and Prieto-Matias, M. Big-PERCIVAL: Exploring the Native Use of 64-Bit Posit Arithmetic in Scientific Computing. *arXiv e-prints* (2023).

[104] Mallasén, D., Murillo, R., Del Barrio, A. A., Botella, G., Piñuel, L., and Prieto-Matias, M. PERCIVAL: Deploying Posits and Quire Arithmetic into the CVA6 RISC-V Core. In *Proceedings of the 20th ACM International Conference on Computing Frontiers* (2023), Association for Computing Machinery, pp. 375–376.

[105] Mallasén, D., Murillo, R., Del Barrio, A. A., Botella, G., Piñuel, L., and Prieto-Matias, M. Customizing the CVA6 RISC-V Core to Integrate Posit and Quire Instructions. In *2022 37th Conference on Design of Circuits and Integrated Circuits (DCIS)* (2022), IEEE, pp. 01–06.

[106] Mallasén, D., Murillo, R., Del Barrio, A. A., Botella, G., Piñuel, L., and Prieto-Matias, M. PERCIVAL: Open-Source Posit RISC-V Core With Quire Capability. *IEEE Transactions on Emerging Topics in Computing 10*, 3 (2022), 1241–1252.

[107] Mellempudi, N., Srinivasan, S., Das, D., and Kaul, B. Mixed Precision Training With 8-bit Floating Point. *arXiv e-prints* (2019).

[108] Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed Precision Training. *arXiv e-prints* (2018).

[109] Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., Mellempudi, N., Oberman, S., Shoeybi, M., Siu, M., and Wu, H. FP8 Formats for Deep Learning. *arXiv e-prints* (2022).

[110] Mitchell, J. N. Computer Multiplication and Division Using Binary Logarithms. *IRE Transactions on Electronic Computers EC-11*, 4 (1962), 512–517.

[111] Mittal, S. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys (CSUR) 48*, 4 (2016).

[112] Muller, J.-M., Brunie, N., de Dinechin, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., and Torres, S. *Handbook of Floating-Point Arithmetic, 2nd edition*. Springer, 2018.

[113] Murillo, R., Del Barrio, A. A., and Botella, G. Customized Posit Adders and Multipliers using the FloPoCo Core Generator. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (2020).

[114] Murillo, R., Del Barrio, A. A., and Botella, G. Deep PeNSieve: A deep learning framework based on the posit number system. *Digital Signal Processing: A Review Journal 102* (2020), 102762.

[115] Murillo, R., Del Barrio, A. A., and Botella, G. La Aritmética del Futuro: una Reflexión sobre los Planes de Estudio. *Enseñanza y Aprendizaje de Ingeniería de Computadores* (2020), 49–60.

[116] Murillo, R., Del Barrio, A. A., and Botella, G. Posit Arithmetic Units for Deep Neural Networks. In *Jornadas SARTECO* (2021), pp. 617–622.

[117] Murillo, R., Del Barrio, A. A., and Botella, G. Efectos de la Precisión Numérica en las Aplicaciones Científicas. In *Jornadas SARTECO* (2022), pp. 663–669.

[118] Murillo, R., Del Barrio, A. A., and Botella, G. The Effects of Numerical Precision In Scientific Applications. In *2022 Annual Modeling and Simulation Conference (ANNSIM)* (2022), IEEE, pp. 152–163.

[119] Murillo, R., Del Barrio, A. A., and Botella, G. A Suite of Division Algorithms for Posit Arithmetic. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2023), vol. 2023-July, IEEE, pp. 41–44.

[120] Murillo, R., Del Barrio, A. A., Botella, G., and Pilato, C. Generating Posit-Based Accelerators With High-Level Synthesis. *IEEE Transactions on Circuits and Systems I: Regular Papers 70*, 10 (2023), 4040–4052.

[121] Murillo, R., Del Barrio Garcia, A. A., Botella, G., Kim, M. S., Kim, H., and Bagherzadeh, N. PLAM: a Posit Logarithm-Approximate Multiplier. *IEEE Trans. on Emerging Topics in Computing 10*, 4 (2022), 2079–2085.

[122] Murillo, R., Hormigo, J., Del Barrio, A. A., and Botella, G. HUB Meets Posit: Arithmetic Units Implementation. *IEEE Transactions on Circuits and Systems II: Express Briefs 71*, 1 (2024), 440–444.

[123] Murillo, R., Mallasén, D., Del Barrio, A. A., and Botella, G. Energy-Efficient MAC Units for Fused Posit Arithmetic. In *2021 IEEE 39th International Conference on Computer Design (ICCD)* (2021), pp. 138–145.

[124] Murillo, R., Mallasén, D., Del Barrio, A. A., and Botella, G. Comparing Different Decodings for Posit Arithmetic. In *Conference for Next Generation Arithmetic (CoNGA)* (2022), vol. 13253, pp. 84–99.

[125] Murillo, R., Mallasén, D., Del Barrio, A. A., and Botella, G. PLAUs: Posit Logarithmic Approximate Units to Implement Low-Cost Operations with Real Numbers. In *Conference for Next Generation Arithmetic (CoNGA)* (2023), vol. 13851, pp. 171–188.

[126] Murillo Montero, R. Study of the posit number system: a practical approach. Bachelor's thesis, Universidad Complutense de Madrid, 2019.

[127] Murillo Montero, R. Leveraging Posit Arithmetic in Deep Neural Networks. Master's thesis, Universidad Complutense de Madrid, 2021.

[128] Murillo Montero, R., Del Barrio, A. A., and Botella, G. Template-Based Posit Multiplication for Training and Inferring in Neural Networks. *arXiv e-prints* (2019).

[129] Muñoz, S. D., and Hormigo, J. Improving fixed-point implementation of QR decomposition by rounding-to-nearest. In *2015 International Symposium on Consumer Electronics (ISCE)* (2015).

[130] Nambi, S., Ullah, S., Sahoo, S. S., Lohana, A., Merchant, F., and Kumar, A. ExPAN(N)D : Exploring Posits for Efficient Artificial Neural Network Design in FPGA-Based Systems. *IEEE Access 9* (2021), 103691–103708.

[131] Neves, N., Tomas, P., and Roma, N. Dynamic Fused Multiply-Accumulate Posit Unit with Variable Exponent Size for Low-Precision DSP Applications. *2020 IEEE Workshop on Signal Processing Systems (SiPS)* (2020).

[132] Neves, N., Tomas, P., and Roma, N. Reconfigurable Stream-based Tensor Unit with Variable-Precision Posit Arithmetic. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2020), IEEE, pp. 149–156.

[133] Norris, C. J., and Kim, S. An Approximate and Iterative Posit Multiplier Architecture for FPGAs. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)* (2021), IEEE.

[134] NVIDIA Corporation. NVIDIA A100 Tensor Core GPU Architecture. Whitepaper, NVIDIA Corporation, 2020. Online.

[135] NVIDIA Corporation. Accelerating AI Training with NVIDIA TF32 Tensor Cores, 2021.

[136] NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Architecture. Whitepaper, NVIDIA Corporation, 2022. Online.

[137] Omtzigt, E. T. L., and Quinlan, J. Universal: Reliable, Reproducible, and Energy-Efficient Numerics. In *Conference for Next Generation Arithmetic (CoNGA)* (2022), vol. 13253, pp. 100–116.

[138] Paszke, A., Gross, S., Massa, F., Lerer, A., Google, J. B., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Xamla, A. K., Yang, E., Devito, Z., Nabla, M. R., Tejani, A., Chilamkurthy, S., Ai, Q., Steiner, B., Facebook, L. F., Facebook, J. B., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *33rd International Conference on Neural Information Processing Systems (NeurIPS)* (2019).

[139] Pilato, C., and Ferrandi, F. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)* (2013).

[140] Pilipović, R., and Bulić, P. On the Design of Logarithmic Multiplier Using Radix-4 Booth Encoding. *IEEE Access 8* (2020), 64578–64590.

[141] Podobas, A., and Matsuoka, S. Hardware Implementation of POSITs and Their Application in FPGAs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2018), IEEE, pp. 138–145.

[142] Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. Zero-Shot Text-to-Image Generation. *arXiv e-prints* (2021).

[143] Raposo, G., Tomas, P., and Roma, N. PositNN: Training Deep Neural Networks with Mixed Low-Precision Posit. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2021), IEEE, pp. 7908–7912.

[144] Raveendran, A., Jean, S., Mervin, J., Vivian, D., and Selvakumar, D. A Novel Parametrized Fused Division and Square-Root POSIT Arithmetic Architecture. In *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)* (2020), pp. 207–212.

[145] Rojas, R. Konrad Zuse's legacy: the architecture of the Z1 and Z3. *IEEE Annals of the History of Computing 19*, 2 (1997), 5–16.

[146] Sai, A. M. V., Bhairathi, G., and Hayatnagarkar, H. G. PERC: Posit Enhanced Rocket Chip. In *4th Workshop on Computer Architecture Research with RISC-V (CARRV'20)* (2020).

[147] Sarkar, S., Velayuthan, P. M., and Gomony, M. D. A Reconfigurable Architecture for Posit Arithmetic. In *2019 22nd Euromicro Conference on Digital System Design (DSD)* (2019), IEEE, pp. 82–87.

[148] Saxena, V., Reddy, A., Neudorfer, J., Gustafson, J., Nambiar, S., Leupers, R., and Merchant, F. Brightening the Optical Flow through Posit Arithmetic. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)* (2021), IEEE, pp. 463–468.

[149] Schlueter, B., Calhoun, J., and Poulos, A. Evaluating the Resiliency of Posits for Scientific Computing. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (New York, NY, USA, 2023), ACM, pp. 477–487.

[150] Sharma, N. N., Jain, R., Pokkuluri, M. M., Patkar, S. B., Leupers, R., Nikhil, R. S., and Merchant, F. CLARINET: A quire-enabled RISC-V-based framework for posit arithmetic empiricism. *Journal of Systems Architecture 135* (2023), 102801.

[151] Silvano, C., Ielmini, D., Ferrandi, F., Fiorin, L., Curzel, S., Benini, L., Conti, F., Garofalo, A., Zambelli, C., Calore, E., Schifano, S. F., Palesi, M., Ascia, G., Patti, D., Perri, S., Petra, N., De Caro, D., Lavagno, L., Urso, T., Cardellini, V., Cardarilli, G. C., and Birke, R. A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms.

[152] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of Go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484–489.

[153] Sommer, L., Weber, L., Kumm, M., and Koch, A. Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2020), pp. 75–83.

[154] Soria-Pardos, V., Doblas, M., Lopez-Paradis, G., Candon, G., Rodas, N., Carril, X., Fontova-Muste, P., Leyva, N., Marco-Sola, S., and Moreto, M. Sargantana: A 1 GHz+ In-Order RISC-V Processor with SIMD Vector Extensions in 22nm FD-SOI. In *2022 25th Euromicro Conference on Digital System Design (DSD)* (2022), IEEE, pp. 254–261.

[155] Sun, X., Choi, J., Chen, C. Y., Wang, N., Venkataramani, S., Srinivasan, V., Cui, X., Zhang, W., and Gopalakrishnan, K. Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems* (2019).

[156] Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial andb Survey. *Proceedings of the IEEE 105*, 12 (2017), 2295–2329.

[157] Tagliavini, G., Mach, S., Rossi, D., Marongiu, A., and Benini, L. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2018), pp. 1051–1056.

[158] Taunk, K., De, S., Verma, S., and Swetapadma, A. A Brief Review of Nearest Neighbor Algorithm for Learning and Classification. In *2019 International Conference on Intelligent Computing and Control Systems (ICCS)* (2019), pp. 1255–1260.

[159] Tiwari, S., Gala, N., Rebeiro, C., and Kamakoti, V. PERI: A Configurable Posit Enabled RISC-V Core. *ACM Transactions on Architecture and Code Optimization 18*, 3 (2021).

[160] Torres Quevedo, L. Essays on Automatics. Its Definition. Theoretical Extent of Its Applications. In *The Origins of Digital Computers: Selected Papers*. Springer, 1982, pp. 89–107.

[161] Uguen, Y., Forget, L., and de Dinechin, F. Evaluating the Hardware Cost of the Posit Number System. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (Barcelona, Spain, 2019), IEEE, pp. 106–113.

[162] van Dam, L., Peltenburg, J., Al-Ars, Z., and Hofstee, H. P. An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM. In *Proceedings of the Conference for Next Generation Arithmetic 2019* (2019), CoNGA'19.

[163] Villalba, J., Hormigo, J., and González-Navarro, S. Fast HUB Floating-Point Adder for FPGA. *IEEE Transactions on Circuits and Systems II: Express Briefs 66*, 6 (2019), 1028–1032.

[164] Villalba-Moreno, J., Hormigo, J., and González-Navarro, S. Unbiased Rounding for HUB Floating-Point Addition. *IEEE Transactions on Computers 67*, 9 (2018), 1359–1365.

[165] Walia, S., Tej, B. V., Kabra, A., Devnath, J., and Mekie, J. Fast and Low-Power Quantized Fixed Posit High-Accuracy DNN Implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 30*, 1 (2022), 108–111.

[166] Wang, Y., Deng, D., Liu, L., Wei, S., and Yin, S. PL-NPU: An Energy-Efficient Edge-Device DNN Training Processor With Posit-Based Logarithm-Domain Computing. *IEEE Transactions on Circuits and Systems I: Regular Papers 69*, 10 (2022), 4042–4055.

[167] Waterman, A., Lee, Y., Patterson, D. A., and Asanović, K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Tech. Rep. UCB/EECS-2016-118, EECS Department, University of California, Berkeley, 2016.

[168] White, J., Adámek, K., Roy, J., Dimoudi, S., Ransom, S. M., and Armour, W. Bits Missing: Finding Exotic Pulsars Using bfloat16 on NVIDIA GPUs. *The Astrophysical Journal Supplement Series 265*, 1 (2023), 13.

[169] Wu, B. SmallPositHDL. `https://github.com/starbrilliance/SmallPositHDL`, 2020. Online.

[170] Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. *arXiv e-prints* (2020).

[171] Xiao, F., Liang, F., Wu, B., Liang, J., Cheng, S., and Zhang, G. Posit Arithmetic Hardware Implementations with The Minimum Cost Divider and SquareRoot. *Electronics 9*, 10 (2020).

[172] Xilinx. *Vitis High-Level Synthesis User Guide*, 2021. v2021.2.

[173] Zhang, H., He, J., and Ko, S.-B. Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)* (2019), vol. 2019-May, IEEE.

[174] Zhang, H., and Ko, S.-B. Design of Power Efficient Posit Multiplier. *IEEE Transactions on Circuits and Systems II: Express Briefs 67*, 5 (2020), 861–865.

[175] Zhang, H., and Ko, S.-B. Efficient Approximate Posit Multipliers for Deep Learning Computation. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems 13*, 1 (2023), 201–211.

[176] Zhang, T., Lin, Z., Yang, G., and Sa, C. D. QPyTorch: A Low-Precision Arithmetic Simulation Framework. *arXiv e-prints* (2019).

[177] Żołądek, H., and Murillo, R. *Qualitative Theory of ODEs: An Introduction to Dynamical Systems Theory*. WORLD SCIENTIFIC (EUROPE), 2022.