TÉCNICO LISBOA

Vector Multiply-Accumulate Unit for Transprecision Computing

Luís Miguel Marques Crespo

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Filipe Valentim Roma
Prof. Pedro Filipe Zeferino Aidos Tomás

Examination Committee

Chairperson: Doutora Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Nuno Filipe Valentim Roma
Member of the Committee: Prof. Horácio Cláudio de Campos Neto

November 2021

Declaration

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

Acknowledgments

Gostaria de começar por agradecer aos meus pais, por todo o apoio, carinho e educação que sempre me proporcionaram. Agradeço à minha namorada, Inês, por estar sempre presente e por todos os momentos de alegria e afeto. Ao meu irmão, agradeço todas as horas de companhia e amizade. Um obrigado aos amigos que conheci e me acompanharam ao longo deste árduo percurso e por todos os bons e maus momentos que partilhámos.

Agradeço aos meus orientadores Prof. Nuno Roma e Prof. Pedro Tomás, os quais me deram um apoio essencial na elaboração deste trabalho e me desafiaram a ir mais longe. Quero deixar um especial agradecimento ao Nuno Neves, por toda a paciência, disponibilidade e apoio no desenvolvimento desta tese. Sem ele, este trabalho não seria possível.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/CEC/50021/2020, PTDC/EEI-HAC/30485/2017 and PCIF/MPG/0051/2018.

Abstract

Transprecision computing is currently viewed as a potential paradigm to increase performance and energy efficiency in modern computing systems, by allowing the floating-point precision to be tuned to the application requirements. However, most attempts at deploying transprecision architectures often rely on instantiations of multiple different modules to provide support for different precisions. To counteract this issue, recent solutions have started to explore variable-precision units with dynamic datapaths that can support different floating-point precisions with the same hardware resources. This approach not only provides for significant area reductions but also enables straightforward Single Instruction, Multiple Data (SIMD) capabilities. Despite their success, most architectures often have to rely on the IEEE-754 standard and lack support for low-precision arithmetic. To that end, the recent Posit number system presents a non-uniform encoding that is particularly well-suited for low-precision arithmetic. However, for higher precisions, it often incurs in prohibitive hardware requirements. In this Thesis, a new unified Posit/IEEE-754 Vector Multiply-Accumulate Unit is proposed, with variable-precision and SIMD computing capabilities. It implements a fully vectorized datapath with multiple-precision arithmetic capabilities and unique shared support for both the Posit and IEEE-754 formats. The proposed unit was fully described in RTL by considering Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) implementations. Results show that the proposed unit requires 50% less area and 2.9$\times$ less power consumption when compared to a reference transprecision setup.

Keywords

Floating-point Arithmetic, Posit Number System, IEEE-754, Variable-Precision, SIMD, Transprecision Computing

Resumo

O paradigma de computação em transprecisão é atualmente visto, como uma potencial solução para aumentar o desempenho e a eficiência energética em sistemas de computação modernos, através do ajuste da precisão dos números de vírgula flutuante aos requisitos da aplicação. No entanto, a maior parte das implementações resultam da utilização de diversos módulos para suportar as diferentes precisões. Para contrariar esta tendência, arquiteturas mais recentes implementam unidades de precisão variável, com *datapaths* dinâmicos que suportam diferentes precisões com os mesmos recursos de hardware. Esta abordagem, permite reduções de área como também possibilita explorar esquemas de Single Instruction, Multiple Data (SIMD). No entanto, apesar do seu sucesso, estas arquiteturas tendem apenas a suportar o standard IEEE-754, que não contempla aritmética de baixa precisão. Neste sentido, o recente sistema numérico Posit apresenta uma codificação não uniforme que é particularmente adequada para a aritmética de baixa precisão. No entanto, para precisões mais elevadas, o Posit toma proporções de hardware demasiado elevadas. Assim, esta Tese propõe uma nova unidade Vetorial de Multiplicação-Acumulação unificada para os formatos Posit e IEEE-754, oferecendo precisão variável e recursos de computação SIMD. A unidade é totalmente vetorizada, com aritmética de precisão variável e suporte para os formatos Posit e IEEE-754. A unidade proposta, foi inteiramente descrita em RTL e implementada em Application Specific Integrated Circuit (ASIC) e em dispositivos Field Programmable Gate Array (FPGA). Os resultados mostram que a unidade proposta, quando comparada a uma referência de transprecisão, obtém uma área 50% inferior e requere consumo de potência 2.9$\times$ menor.

Palavras Chave

Aritmética de Vírgula Flutuante, Formato Numérico Posit, IEEE-754, Precisão Variável, SIMD, Computação de Transprecisão

Contents

List of Figures

List of Tables

Acronyms

ASIC	Application Specific Integrated Circuit
DL	Deep Learning
DNN	Deep Neural Network
FMA	Fused Multiply-Add
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FP	Floating-Point
IEEE-754	IEEE Standard for Floating-Point Arithmetic
LSB	least significant bit
LZC	leading zero counter
MAC	Multiply-Accumulate
maxpos	maximum positive number
minpos	minimum positive number
MSB	most significant bit
NaN	Not a Number
NaR	Not a Real
qNaN	quiet NaN
SIMD	Single Instruction, Multiple Data
sNaN	signaling NaN
VMAC	Vectorized Multiply-Accumulate

1 Introduction

Contents

1.1 Motivation

The end of Moore's Law and waning of Dennard scaling mark the end of an era in which the computational capacity growth was mainly based on the down-scaling of silicon-based technology [5]. As a result, new research efforts have been shifting to the study of more efficient arithmetic circuits and improved computer technologies to meet the requirements of applications in several emergent domains [5].

Motivated by rapidly evolving algorithms advances in domains, such as Deep Learning (DL) and the ever-increasing amount of data availability, led to the exploration of different approaches to cope with increasing computational demands. In particular, significant advances have been made by lowering the arithmetic precision of Floating-Point (FP) operations [3,6–13], to obtain straightforward acceleration and efficiency. By lowering operand precision, it is possible to reduce memory storage per operand, obtain higher computing bandwidths, while reaching lower power and energy consumptions. In fact, it has been shown that it is possible to use different precisions in the different stages of an application [8, 14]. As an example, a recent study [8] verified that some stages of the training phase of DL applications are more sensitive to numerical errors than others, allowing the adjustment of the precision accordingly. On the other hand, applications such as physics simulation, rely heavily in high precision operations, such as 64-bits or even higher [15, 16]. Hence, it can be concluded that different applications have distinct precision requirements and adjusting the precision to the application can provide significant acceleration and efficiency gains.

Transprecision computing [14] is set on these principles and has received a gradually increasing attention as a viable paradigm to cope with the ever-increasing performance and energy efficiency demands in modern computing systems. Transprecision computing is a step beyond approximate computing [17], in which rather than tolerating errors implied by imprecise computations, systems are designed to deliver just the required precision. However, most transprecision hardware solutions [18] rely on instantiating multiple arithmetic modules to support different precisions, which leads to an increased chip area and power consumption. Moreover, even if the non-used modules are disabled, when a given precision is considered, it still results in a waste of resources [19].

To tackle this issue, recent variable-precision arithmetic units [19–21] introduce dynamic datapaths that can operate in different precisions with the same hardware resources. To do so, they deploy a higher precision arithmetic logic (e.g., 32-bit) and allow parts of the circuit to be turned off to lower the operand precision by as much as it is required by the application (e.g., to 8-bit or lower [15]). While this approach provides for significant area reductions and enables straightforward Single Instruction, Multiple Data (SIMD) capabilities [19], existing solutions are often limited by their adoption of the IEEE-754 standard [20], whose lowest supported precision is only 16 bits. Some major computing market players such as Intel [22], Google [6] and Xilinx [23] have already realized that the adoption of alternative floating-point formats with reduced precision may provide straightforward computing acceleration. However, despite their success, most solutions are tailored for the DL domain.

Alternatively, some recent solutions [19, 21] adopt the recently proposed Posit format [3], since it allows parameterizable precision and dynamic range (exponent size). This format is especially well suited for low-precision operations since it offers a trade-off between a wider dynamic range and an increased precision, which effectively allows a higher accuracy while lowering the operand precision (fewer bits). The Posit format is also particularly suited for fused operations since it adopts an exact accumulator structure (quire) with enough precision to avoid overflow and accuracy losses [4]. While Posit-based implementations traditionally define and fix its parameters at design-time [10–12, 24, 25], it has been shown that it is possible to support runtime-configurable exponent sizes with minimal hardware overheads [13]. This allows making use of the entire representable dynamic range for a given posit precision by specifying the exponent size of the input values. In turn, it also provides the possibility to encode a larger dynamic range, capable of supporting (within the same hardware) both values with high precisions and very large magnitude.

Nevertheless, while these features make posits well suited for low-precision arithmetic and transprecision computing, the hardware overhead associated with the quire becomes prohibitive when the precision and exponent size increase [24, 25]. Additionally, for a more general-purpose context, it is desirable to maintain compatibility with the standard IEEE-754 format, as it still is the most established FP format.

1.2 Objectives

The main objective of this thesis is to investigate and implement new variable-precision architectures for future Transprecision computing systems. Accordingly, the following research directions were considered for the herein presented work:

- Research on new dynamic datapaths to enable variable-precision computing;
- Assess the viability of adopting the Posit format as an alternative to the IEEE-754 standard, for low-precision arithmetic;
- Investigate new mechanisms to mitigate the hardware overheads associated with the adoption of the Posit format for high-precision arithmetic.

The aimed work will also entail the description of all developed hardware modules in RTL and their evaluation in terms of performance and energy efficiency, by considering implementations in Application Specific Integrated Circuit (ASIC) technologies and Field Programmable Gate Array (FPGA) devices.

1.3 Contributions

According to the defined objectives, this thesis proposes a new Posit/IEEE-754 Vector Multiply-Accumulate (MAC) unit for transprecision computing. Besides combining variable-precision arithmetic and SIMD capabilities, it takes a step further from existing solutions by deploying a unified support for the IEEE-754 and Posit formats. It introduces the following contributions and features:

- an efficient variable-precision FP multiply-accumulate (MAC) 32-bit architecture, especially designed for transprecision computing;
- a unified FP arithmetic architecture compatible with both the IEEE-754 and the Posit formats with support for inter-format operation and conversion;
- a fully vectorized datapath to efficiently make use of the released hardware resources in low-precision computing scenarios;
- SIMD decoding/encoding modules with shared support for FP vectors encoded with *i)* dynamic posit formats with configurable exponent size; *ii)* IEEE-754 standard and low-precision non-standard formats; and *iii)* multiple scalar and vector element precisions (including 32/16/8-bit scalars and 2x16/4x8-bit vectors).

Results show that, for an ASIC implementation, the proposed unit requires 50% less area and 2.9$\times$ less power consumption when compared to a reference transprecision system setup. For a FPGA implementation, the proposed unit requires 2.1x less LUTs, 4x less registers, and 3.9x less power consumption when compared to a reference transprecision system setup.

1.4 Thesis Outline

The thesis is organized in the following chapters:

- **Chapter 2** provides background on floating-point formats, by focusing on the IEEE-754 standard and the Posit number system. Then, the main arithmetic structures for each format are presented in detail, followed by a revision of the literature. The chapter is concluded by covering the main problems and solutions that can be found in the literature;
- **Chapter 3** describes the proposed variable-precision architecture, by introducing the adopted vector data formats and structures. Then, it is presented the proposed architecture and its main modules in detail;
- **Chapter 4** describes the main implementation results for ASIC and FPGA, together with a comparison with state-of-the-art solutions;
- **Chapter 5** presents the main conclusions of this work along with a discussion on possible future work directions.

2 Background

Contents

Transprecision computing [14] is set on the principle that different application domains have different precision requirements. However, most transprecision hardware solutions [18] rely on the instantiation of multiple modules to support different precisions and only support the IEEE Standard for Floating-Point Arithmetic (IEEE-754). More recently, some floating-point architectures have been adopting the Posit format [3], which has been gaining attention as a possible complement to the IEEE-754 standard. The unified support of both formats can be the next step for transprecision computing.

This chapter presents some background on floating-point arithmetic with the IEEE-754 standard and Posit formats. The first section introduces relevant floating-point formats, starting with the definition of floating-point, the IEEE-754 standard for floating-point including some of his variants, and the recent Posit number system. The second section describes the main computing structures and operators for IEEE-754 floats and posit operations. The third section presents the most relevant IEEE-754 and Posit arithmetic unit architectures in the literature. Finally, a discussion is presented regarding IEEE-754 floats and posit as well as transprecision implementation approaches.

2.1 Floating-point formats

Since the early years of computing, several ways of approximating and representing real numbers have been introduced, each providing different compromises between the complexity of its manipulation and the involved approximation error. A relatively straightforward approach is to adopt a fixed-point representation, where the numbers are processed as integers with a fixed scale factor. They have three components: integer part, binary point (implied), and fractional part. They have simple associated hardware, however, they have a limited dynamic range, caused by the fixed radix point position.

Other proposed representations involve storing the logarithm of a number and doing multiplication by adding the logarithms or using a pair of integers (x,y) to represent the fraction x/y. However, floating-point arithmetic is by far the most widely used way of approximating real number arithmetic to perform numerical calculations on modern computers.

Floating-point was created as a means to have an approximation of real numbers while supporting both very small and very large real numbers (i.e. a larger dynamic range) with a limited number of bits. It is a “semi-logarithmic” representation with a fixed-point component (significand or mantissa) that is scaled by a factor (exponent). A floating-point number is represented as

$$sign \times significand \times base^{exponent}. \tag{2.1}$$

The most popular format for floating-point arithmetic is the IEEE-754 [1], a standard established in 1985 that has been expanded and improved by the Institute of Electrical and Electronics Engineers over the years. The standard defines encodings, operations, exception handling, and rounding rules. The standard supports base 2 and 10, although the second is not relevant in this work.

2.1.1 IEEE-754 standard

The IEEE Standard for Floating-Point Arithmetic was established in 1985 with the goal to improve the portability of floating-point computations and it was recently updated in 2019 [1]. The representation of binary floating-point data consists of three fields – sign, exponent, and trailing significand. In Figure 2.1 it is represented the structure of an n-bit float. The standard defines several formats of widths, 16, 32, 64, and 128 bits, and in general for any multiple of 32 bits of at least 128 bits. In Table 2.1 are represented the first four.

The sign bit is 0 for positive numbers and 1 for negative. The exponent is w-bit wide and it is represented as an unsigned integer with a bias: $E = exp + bias$. The significand field is represented as a fixed-point number with t-bits. Normalized numbers are represented as

$$(-1)^{sign} \times 2^{exp} \times 1.significand, \tag{2.2}$$

where the '1' denotes the hidden bit, which is implicitly encoded in the exponent. When the number is subnormal, the hidden bit is set to 0 and $exp = exp_{min}$. Apart from these encodings, the standard defines several expectations, such as Not a Number (NaN) and infinity, all represented in Table 2.2.

The IEEE-754 standard defines not only a number, but also several guidelines and rules in what concerns:

- Operations (e.g., Arithmetic operations, conversions and sign manipulation)
- Rounding
- Exceptions

Most arithmetic operations do not result in a number that can be exactly represented. In such cases, the result needs to be rounded. To that end, The IEEE-754 standard defines five rounding modes:

- *roundTiesToEven* - rounds to the nearest value; if the two nearest are equally near, rounds to the one with an even least significant digit; if that is not possible, rounds to the larger in magnitude.
- *roundTiesToAway* - rounds to the nearest value; if the two nearest are equally near, rounds to the larger in magnitude.

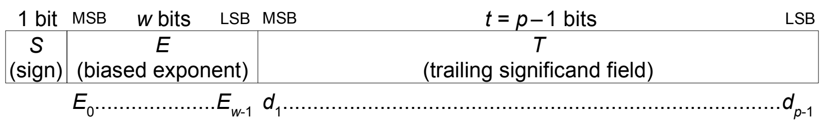

1 bit
MSB
w bits
LSB
MSB
t = p−1 bits
LSB
S
(sign)
E
(biased exponent)
T
(trailing significand field)
E0....................Ew-1
d1..................................dp-1

Figure 2.1: Binary IEEE-754 floating-point format [1].

Format	Bits (n)	Exponent Bits (w)	Bias	Significand Bits (t)
Half-precision (FP16)	16	5	15	10
Single-precision (FP32)	32	8	127	23
Double-precision (FP64)	64	11	1023	52
Quadruple-precision (FP128)	128	15	16383	113

Table 2.1: Haf, single, double and quadruple precision floating-points encodings.

Encoding	Sign	Biased Exponent	Fraction
Positive zero	0	0 (all 0's)	0 (all 0's)
Negative zero	1	0 (all 0's)	0 (all 0's)
Positive infinity	0	(all 1's)	0 (all 0's)
Negative infinity	1	(all 1's)	0 (all 0's)
qNaN	-	(all 1's)	any number (non-zero)
signaling NaN (sNaN)	-	(all 1's)	any number (non-zero)
Positive nonzero (Subnormal)	0	0 (all 0's)	0.any number
Negative nonzero (Subnormal)	1	0 (all 0's)	0.any number
Positive nonzero (Normalized)	0	any number (non-zero)	1.any number
Negative nonzero (Normalized)	1	any number (non-zero)	1.any number

Table 2.2: Encodings of the IEEE-754 standard.

- *roundTowardPositive* - rounds to the closest value towards positive infinity.
- *roundTowardNegative* - rounds to the closest value towards negative infinity.
- *roundTowardZero* - truncation.

It is also stated in the standard that a binary format implementation shall provide the first rounding mode as default (*roundTiesToEven*) and the last three as user selectable. The mode *roundTiesToAway* is optional.

The standard specifies five kinds of exceptions that shall be signaled with the corresponding status flag and default result. The exceptions covered by the standard are:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact

Some disadvantages (also argued in [3]) are: repeated patterns to represent NaN values, $\pm\infty$ and ± 0 values, the possibility of overflow/underflow, the added complexity of using normalized/subnormal numbers, misused exponent size and the lack of reproducibility guarantees across systems. The latter occurs since, there are various optional mechanisms that are covered in the standard, including the subnormal support, the diagnostic information on invalid operation exception, which is provided in the quiet NaN (qNaN) output, etc.

Variations

Several variations to the IEEE-754 standard have been recently proposed, mostly by the ever-increasing proliferation of deep learning applications. One of the most most popular is the ”Brain Floating Point” (bfloat16) [2], which is a 16-bit truncated version of the F32. It aims to improve hardware efficiency while maintaining the ability to train accurate deep learning models, all with minimal switching costs from FP32. In Figure 2.2, is represented the structure of: (a) FP32, (b) FP16 and (c) bfloat16.

Since the FP16 format is prone to overflowing and Deep Learning (DL) is not a strongly precision-bound application, the bfloat16 format has an exponent of the same size as a FP32, at the cost of fraction accuracy. Furthermore, bfloat16 multipliers require nearly half the chip area size of an FP16 multiplier. However, this custom floating-point format is not suited for all domain applications, since it was designed specifically to DL.

There is also an 8-bit floating-point format [26], also known as microfloats. It has 1 sign bit, 4 exponent bits, and 3 significand bits. This 8-bit variation of IEEE 754 floating-point stresses some of its limitations. Specifically, it has 14 representations for NaN, $\pm\infty$, and ± 0 which make up to approximately 6.6% of useless values.

One of the most recent is the TensorFloat-32 (TF32) [27], introduced in the NVIDIA Ampere architecture. Despite the name, this format is 19-bit long and is a combination of FP16 and bfloat16. The mantissa is 10-bit long (similar to FP16), while the exponent is 8-bit long (similar to bloat16 and FP32). As a consequence, this format has the same dynamic range of FP32 and the same precision of FP16.

Most of these variations are tailored for the Deep Learning domain, contrarily to the Posit number system, which is gaining attention as a possible alternative (or complement) to the IEEE-754 Standard.

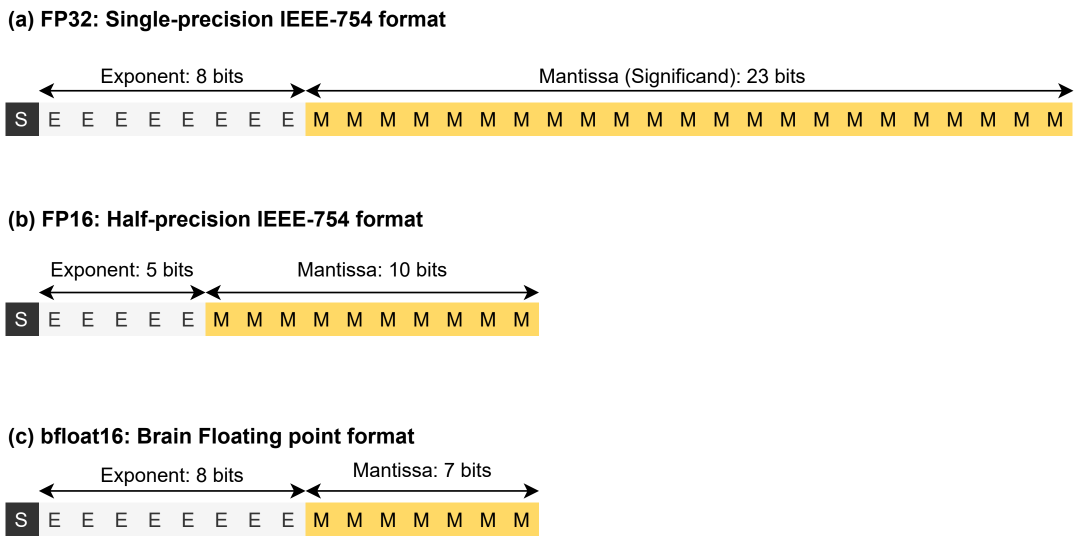

(a) FP32: Single-precision IEEE-754 format
Exponent: 8 bits
Mantissa (Significand): 23 bits
S E E E E E E E E M M M M M M M M M M M M M M M M M M M M M M M
(b) FP16: Half-precision IEEE-754 format
Exponent: 5 bits
Mantissa: 10 bits
S E E E E E M M M M M M M M M M
(c) bfloat16: Brain Floating point format
Exponent: 8 bits
Mantissa: 7 bits
S E E E E E E E E M M M M M M M

Figure 2.2: Structure of: (a) FP32, (b) FP16 and (c) bfloat16 [2].

2.1.2 Posit Format

The Posit format [3] is part of the latest revision of the unum (universal number) arithmetic framework. The original ”Type I” unum was proposed in 2015 [28], and it is a superset of the IEEE-754 Standard. Following the Type I, the “Type II” unum [29] was proposed and corresponds to a completely new design based on the projective reals and abandons compatibility with the IEEE Standard. Motivated by the fact that Type II unums rely on table look-up for most operations and they are not adequate to fused operations, the ”Type III” unum emerged in 2017 [3] – with the designation of posit. The Posit format is a “hardware friendly” version of Type II unums that keeps many of its merits while relaxing some mathematical properties. A draft of its standard [4] was made available in 2021.

A posit is defined by the pair $< n, es >$, where n represents the word size and es the exponent size. In Figure 2.3 it is represented the structure of an n-bit posit, with es exponent bits.

The sign bit is 0 for positive numbers and 1 for negative. However, for negative numbers, it is necessary to take the 2’s complement before decoding the following fields: regime, exponent, and fraction.

The Posit format presents a non-uniform encoding where the length of its parameters varies depending on the magnitude of the represented number. The regime is a sequence of identical bits, r, terminated by the opposite bit, $\bar{r}$ (or by the end of the posit) and has numerical meaning, k, as demonstrated in Table 2.3. Let m be the number of identical bits in the regime. Then, k is given by:

$$k = \begin{cases} m-1 & \text{, if } r_0 = 1 \\ -m & \text{, otherwise.} \end{cases} \tag{2.3}$$

The encoded value indicates a scale factor (sf) of magnitude $useed^k$ where $useed = 2^{2^{es}}$ (or $2^{k2^{es}}$).

The exponent can have up to es exponent bits (depending on how many bits remain to the right of the regime) and is represented as an unsigned integer, e. Contrarily to IEEE-754 floats, there is no bias. Hence the encoded value indicates a scale factor of 2^e.

The fraction, f, is represented by the remaining bits that are not used by the regime and exponent fields. Similarly to the IEEE-754 significand field, there is a hidden bit. However, there are no subnormal numbers, that is, the hidden bit is always ’1’, and the encoded value is $1.f$.

Similarly to the floating-point standard, the Posit format also defines exception values. However, it only provides a single representation for 0 (all ’0’ bits) and one $\pm\infty =$ Not a Real (NaR) (’1’ followed by

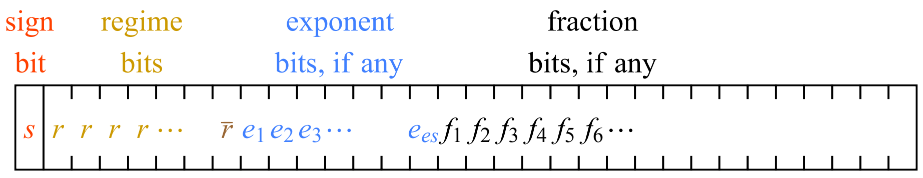


Figure 2.3: Posit format highlighting its components (sign, regime, exponent and fraction) [3].

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	-4	-3	-2	-1	0	1	2	3

Table 2.3: Example of the run-length meaning k of the regime field with 4 regime bits.

all '0' bits). It does not have NaN representation, thus all remaining bit patterns are used to represent actual numbers. Furthermore, the sf can be heavily affected by the chosen value of es, which will, in turn, establish the maximum positive number (maxpos) and minimum positive number (minpos) representable as posits. Unlike IEEE-754 floats, posits do not overflow nor underflow, but saturate to $\pm$maxpos or $\pm$minpos, respectively. Therefore, a number (x) encoded as a posit has a decoded value p given by Equation 2.4.

$$p = \begin{cases} 0 & \text{if } x = 000 \ldots 0, \\ \pm\infty = \text{NaR} & \text{if } x = 100 \ldots 0, \\ (-1)^{sign} \times useed^{k} \times 2^{e} \times 1.f & \text{all other } x. \end{cases} \tag{2.4}$$

Example 2.1.1 illustrates the decoding process for a posit, according to Equation 2.4. To understand the example it is important to define the posit sf, which is analogous to the exp from IEEE-754 floats. It is the combination of the regime and exponent factors and is obtained by rearranging Equation 2.4 as

$$p = (-1)^{sign} \times 2^{e+k2^{es}} \times 1.f, \tag{2.5}$$

therefore, $sf = e + k2^{es}$.

Example 2.1.1 Posit decoding

Consider the following posit bit strings for the $< 8, 2 >$ configuration:

1. 01100101;
2. 11001101;
3. 01111101.

Since the configuration is $< 8, 2 >$, $useed = 2^{2^2} = 16$ for all examples.

The first two examples correspond to decoding a simple positive and negative number. The third example shows some of the peculiarities due to the variable-length parameters. An analysis of each bit string is presented below, with the parameters discriminated (where the variable length is well observed) according to the color scheme from Figure 2.3:

1. 0 110 01 01

   The sign bit 0 means the value is positive. The regime bits 110 have a run-length of two

Continued

1s, which means k is 1, corresponding to a contribution by the regime of 16^1. The exponent bits, 01, represent 1 (as an unsigned binary integer), and contribute another scale factor of 2^1. Lastly, the fraction bits 01 represent 1 (as an unsigned binary integer), so the fraction is $1 + \frac{1}{4}$. This translates in: $p = 1 \times 16^1 \times 2^1 \times 1.25 = 40$.

2. 1 1001101 (01 10 011)

   The sign bit 1 means the value is negative, consequently, it is necessary to take the 2's complement to decode the remaining fields: 01 10 011.

   The regime bits 01 have a run of one 0, which means k is -1, corresponding to a contribution by the regime of 16^{-1}. The exponent bits, 10, represent 2 (as an unsigned binary integer), and contribute another scale factor of 2^2. Lastly, the fraction bit 011 represent 3 (as an unsigned binary integer), resulting in the fraction $1 + \frac{3}{8}$. This translates in: $p = -1 \times 16^{-1} \times 2^2 \times 1.375 = -0.34375$.

3. 0 111110 1

   The sign bit 0 means the value is positive. The regime bits 111110 have a run of five 1s, which means k is 4, corresponding to a contribution by the regime of 16^4. For the exponent, in this case, there is only one 1 bit left in the bit string. However, the exponent field is 2 bits wide, therefore, the represented exponent corresponds to 10, which represents 2 (as an unsigned binary integer), and contributes another scale factor of 2^2. Since there are no bits left, the fraction is $1 + 0$. This translates in: $p = 1 \times 16^4 \times 2^2 \times 1.0 = 262144.0$.

When comparing the structure of a posit against the IEEE-754, the main difference corresponds to the presence of the regime field and the variable length of the parameters. The variable length of the regime allows numbers near to 0 to have more accuracy than extremely large or extremely small numbers (tapered precision), which is the same distribution that Deep Neural Network (DNN) weight parameters usually follow (more grouped around 0). As a result, posit presents more accuracy in this "golden zone", as highlighted on Figure 2.4. However, out of this zone, posits become less accurate than IEEE-754 of the same precision.

Finally, the Posit format only defines one rounding method, where the value is rounded to the nearest binary value and if two posits are equally near, the one with binary encoding ending in 0 is selected.

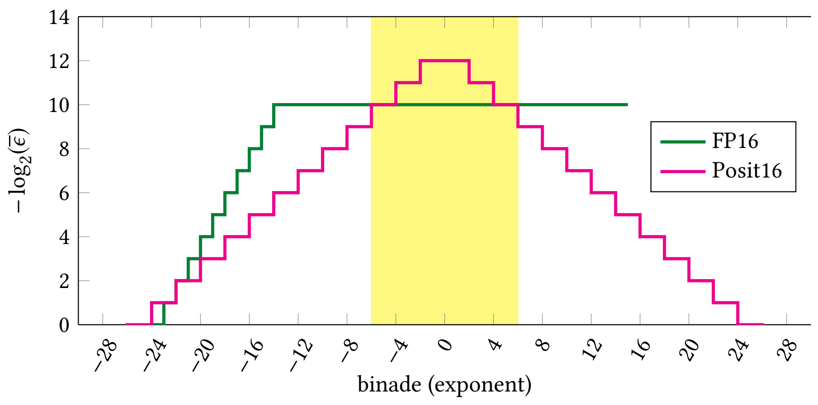


Figure 2.4: Accuracy comparison of FP16 and posit $< 16, 1 >$ with the posit ”golden zone” highlighted [24].

Quire

Besides its non-uniform encoding, the most important feature of the Posit format is the adoption of an exact accumulator for fused operations. This structure, named quire, is a 2’s complement fixed-point accumulator based on the Kulisch accumulator [30]. It is capable of storing sums of products of posits without rounding, up to at least $2^{n-1} - 1$ (dependent of the carry guard) number of such products. As a consequence, the quire format is particularly useful to implement the frequent dot products present in DNN computations, such as convolutions, matrix multiplications, etc. In Figure 2.5 it is represented the quire format.

The quire is composed of the following fields:

- length: $qsize = 16n$ bits.
- fraction: $nq = (8n - 16)$ bits.
- integer: nq bits.
- carry guard: $cg = 31$ bits (allows at least $2^{30} - 1$ sums of products).

The value of the quire datum is given by the 2’s complement signed integer represented by all bits, divided by 2^{nq} and also includes the NaR exception.

Despite the size and exponent parameters of a posit configuration being arbitrary, there are 3 standardized configurations and its parameters are represented in Table 2.4.

The equations that define the quire parameters bit-width on the standard do not adjust to the dynamic

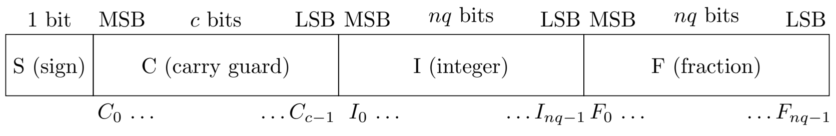


Figure 2.5: Binary quire format [4].

Parameter	$posit8$	$posit16$	$posit32$
whole representation (n)	8	16	32
max exponent es	2	2	2
quire $(16n)$	128	256	512

Table 2.4: Parameters of standardized posit configurations.

range of all the configurations, in fact, they are simplifications tailored for the configurations of Table 2.4. The generic size of the quire for a given configuration is given by Equation 2.6. The 1 is for the sign, the cg is self-explanatory, the remaining factor corresponds to the integer and fraction parameters $(nq = 2^{es+1} \times (n-2))$.

$$qsize = 2^{es+2} \times (n-2) + 1 + cg \tag{2.6}$$

2.2 Floating-point arithmetic structures

This section presents an overview of the most relevant IEEE-754 and Posit floating-point arithmetic operations and their computing architecture. These include common operations, such as addition/subtraction, multiplication, division, as well as fused operations, such as Fused Multiply-Add (FMA) and Multiply-Accumulate (MAC). The latter are particularly used in deep learning applications for dot products, matrix operations and convolutions.

A typical Floating-Point Unit (FPU) structure is depicted in Figure 2.6. The computation is usually in three main phases: *i)* decoding; *ii)* operation; and *iii)* encoding. In the decoding stage, special encodings (e.g. NaN for IEEE-754 floats and NaR for posits) are detected and the fields of the operands are extracted. After properly decoded, the respective arithmetic operation is conducted, whose result must be normalized and the exponent/scale factor properly adjusted. In the encoding stage, the result is rounded and packed, and exception flags are generated. While the various arithmetic operations are mostly identical and independent from the format, the decode and encode stages are naturally dependent on it. In particular, those for the Posit format are more complex, therefore, they will be addressed independently.

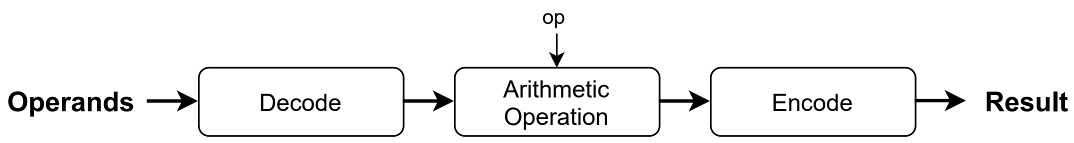


Figure 2.6: Floating-point unit structure.

2.2.1 IEEE-754 Format Decoding

To decode IEEE-754 floats, it is only necessary to unpack all the fields (sign, exponent, and significand), since they can be directly obtained from the encoded binary, only exception cases (± 0, $\pm\infty$ and NaN) need to be processed. Nevertheless, first, the three fields are extracted and the hidden bit (obtained through the exponent) is concatenated with the significand. Figure 2.7 depicts how to detect the various special encodings. Specifically, if the exponent is 0, the hidden bit is 0, otherwise, the hidden bit is 1. This logic does not influence the $\pm\infty$ and NaN decoding since they are usually decoded to flags. Furthermore, as there are two types of NaN and since the standard does not provide details on how to distinguish between a qNaN and a sNaN, they depend on the implementation.

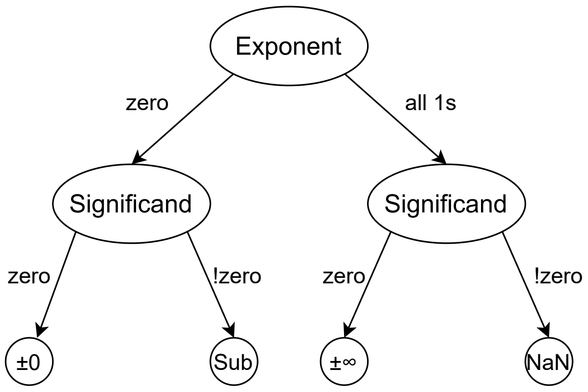


Figure 2.7: Decision tree representation of the IEEE-754 special encodings.

2.2.2 Posit Decode

In contrast with the IEEE-754 format, where the location of the fields is known a priori, the location of the fields in a posit varies depending on the magnitude of the encoded number. Therefore, the decoding of a posit number is much more complex. This can be observed in Figure 2.8, where the common structure of a posit decoding module is illustrated.

The decode stage translates a posit value to their corresponding sign (`s`), scale factor (`sf`) and fraction (`frac`) fields, while signaling the exceptions NaR and zero (`z`) through flags. Posit decoding is a process that includes several steps. First, the sign is extracted from the most significant bit (MSB) and the 2's complement is taken according to the sign value. Next, the regime run-length (`zc`) is decoded by means of a leading zero counter (LZC). However, if the run-length starts with '1' the value ($\mathtt{r}_0$='1') is first inverted. Alternatively, the run-length can be obtained with a LZC in parallel with a leading one counter, being the result selected according with $\mathtt{r}_0$.

Then, the value encoded in the regime (`k`) is calculated according to equation 2.3, which is controlled by the $\mathtt{r}_0$ bit. At the same time, the regime is shifted out of the 2's complemented posit value according

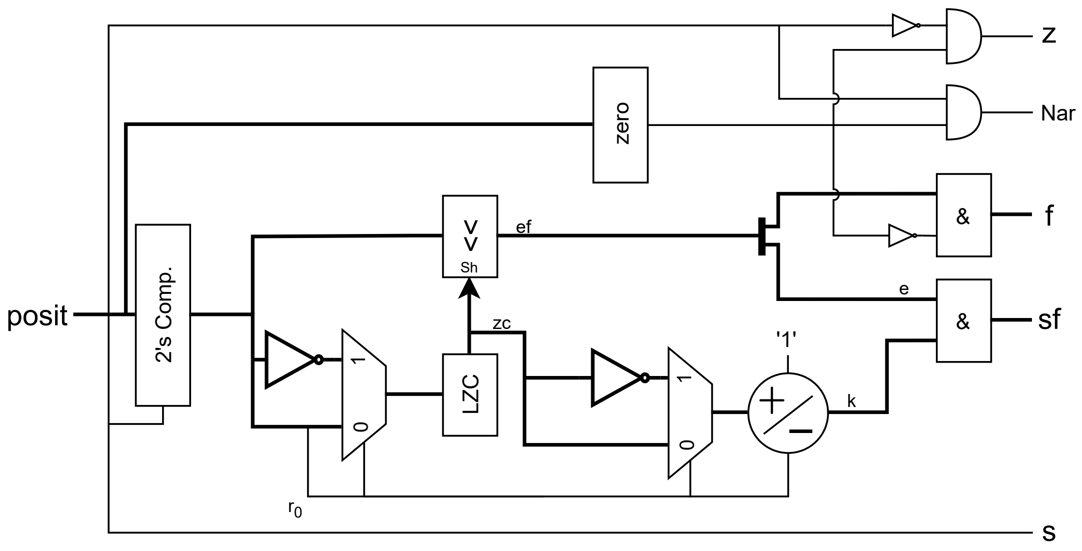


Figure 2.8: Typical posit decoding module.

to the `zc`, leaving the exponent and fraction (`ef`). Since the exponent size is known (defined by es), the exponent and fraction can be easily detached. Finally, the `k` value is concatenated with the exponent to obtain `sf`, and the hidden bit is concatenated, resulting in the fraction (`f`). The hidden bit is '1' if the posit is not an exception (zero or NaR).

2.2.3 Addition/subtraction

Floating-point addition/subtraction is regarded as a very complex operation involving several steps. A simplified version of the classical structure for floating-point addition/subtraction [31, 32] is represented in Figure 2.9.

Suppose we want to add the 8-bit binary numbers 1.1010110×2^{10} and 1.1011001×2^{5}. Mathematically, to add two exponential numbers, their exponents need to be equal for the significands to be added. In case they are not, there are two possible ways to achieve this. The largest exponent can be decremented (significand left-shifted) or the smallest exponent can be incremented (significand right-shifted). Since there is a finite number of bits, there will be a loss of precision. Left shifting the significand affects the MSBs of the significand, while right shifting affects the LSBs. Loss of MSBs implies more precision loss, therefore the smaller exponent is usually incremented and the correspondent significand right shifted. Moreover, pre-shifting is typically applied to only one of the operands, therefore, swap capability is also provided. With the significands aligned, the addiction/subtraction can be conducted. Subsequently, if the operation produces a carry-out bit, the resulting significand value of the addition corresponds to a number in the interval [0,4[. In such a case, the result must be normalized to the interval [1,2[, with corresponding exponent adjustments.

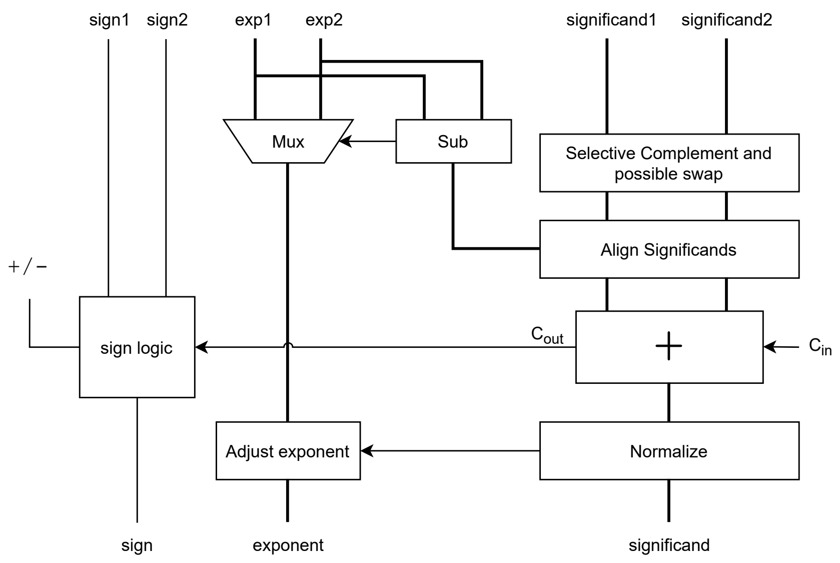


Figure 2.9: Floating-point addition/subtraction structure.

The posit addition/subtraction is very similar to the IEEE-754 operator, it solely differs on the way exponent and scale factor are handled. In Posit format, the scale factor is usually in 2's complement, therefore, 2's complement comparison is needed. However, the detection of larger and smaller operands is done with a simple integer comparison of the posit values (complemented if negative).

2.2.4 Multiplication and Division

The floating-point multiplication and division operations are mathematically simpler than addition/subtraction operations and their implementation is much more straightforward. The classical structure for the floating-point multiplication/division [31, 32] is represented in Figure 2.10. The significands are multiplied/divided, the exponents are added/subtracted (with the proper bias subtraction) to calculate a temporary exponent, and the sign is computed with a XOR operation.

The significand division corresponds to an integer divisor and can be performed with different algorithms. The most common are digit recurrence algorithms and functional iteration algorithms [33]. The first class is a slow division algorithm that produces one digit of the final quotient per iteration, converging linearly to the result. The second class is a fast division algorithm that represents the division or reciprocal operation as a function and uses function-solving techniques such as the Newton-Raphson method [33] to converge to the quotient or reciprocal.

For posit multiplication, the only difference is the scale factor calculation, which does not involve a bias.

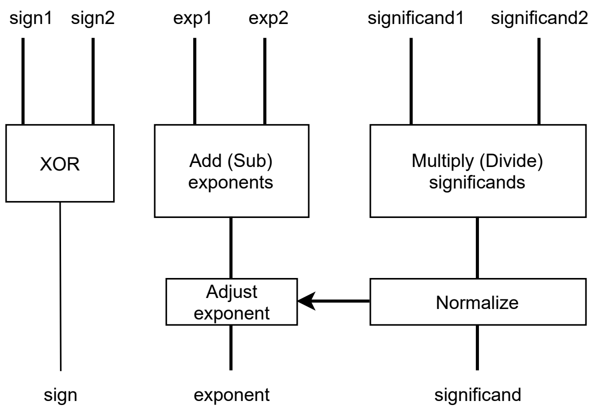


Figure 2.10: Floating-point multiplication/division structure.

2.2.5 Fused Multiply-Add

A fused multiply–add (FMA) operator (defined as $(A\times B)+C$) is a floating-point operation that performs a multiplication followed by an addition between three operands, without intermediate normalization/rounding (fusing). A simplified version of a FMA structure [32, 34] is represented in Figure 2.11.

Since the FMA operator is a combination of a multiplier and an adder/subtractor without the intermediate rounding step, it is necessary to include a larger adder (when compared to a single addition operation) and alignment logic. Similarly, to the single multiplication, the significands are multiplied; the exponents are added, and the sign is computed with a XOR operation. With the resulting exponent, alignment is conducted with the third operand exponent. This concludes the multiplication. The resulting significand is added/subtracted with the aligned significand of the third operand and a leading zero/one

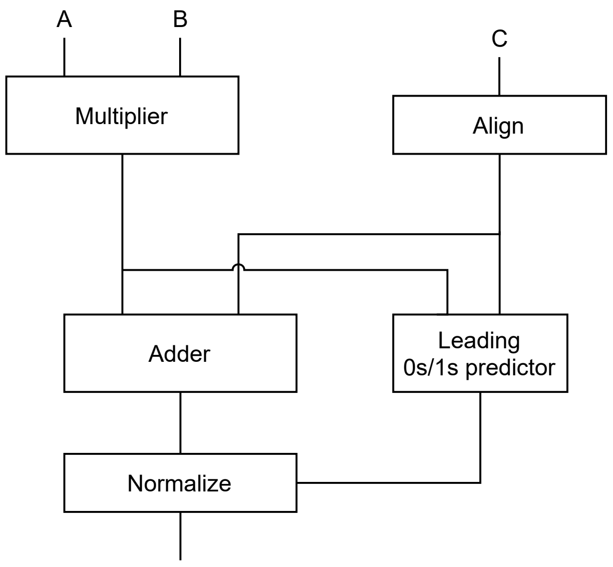


Figure 2.11: Floating-point FMA structure.

predictor is used to shorten the critical path of the normalization. After normalizing the significand, the signals enter the encoding module.

Remembering the reader that the posit standard defines a long accumulator for fused operations, the quire, however, its use is not mandatory. In fact, the standard [4] states that fused expressions (such as fused multiply-add and fused multiply-subtract) do not need to be performed in the quire to be compliant. Therefore, this structure can be applied in the posit environment.

2.2.6 Multiply-Accumulate

This operation implements the multiplication of two elements followed by a continuous accumulation of the multiplication results (defined as $R \leftarrow R + (A \times B)$). This structure is particularly useful to compute several dot products with minimal deviations from the exact result. While there are different solutions for accumulations, the most relevant for this work is the Long Accumulator.

The first Long Accumulator (Kulisch accumulator) was proposed by Ulrich W. Kulisch in [30], which corresponds to a 2's complement fixed-point accumulator to process the dot product of IEEE-754 floats with full accuracy. The structure of a basic long accumulator is depicted in Figure 2.12.

The multiplication result is converted to a 2's complement fixed-point notation which is accumulated with the value stored in the register. The result of each accumulation is normalized and propagated to

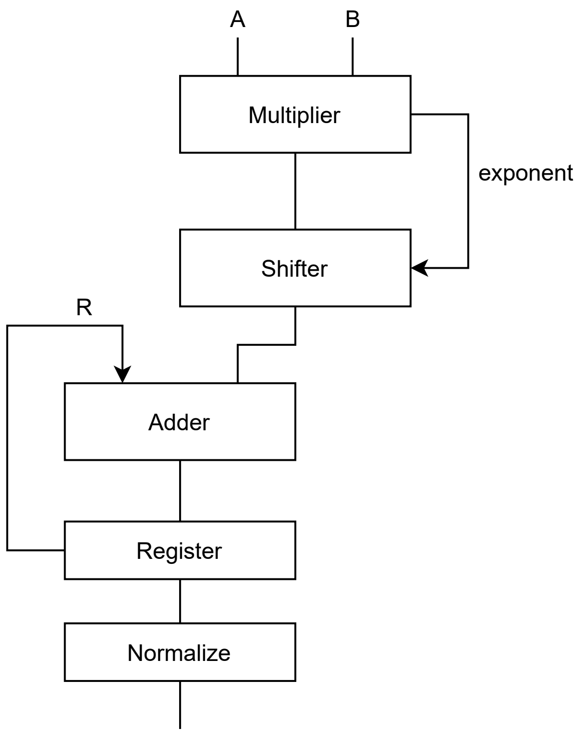


Figure 2.12: MAC structure with Long Adder and Long Shifter.

be encoded back to the IEEE-754 format.

While this operation is not included in the IEEE-754 standard, the Posit standard defines a similar structure named quire, which is specific for accumulation. Nonetheless, several IEEE-754 accumulators have already been proposed and differ from the Posit format mainly in the accumulator sizes, since the formats have different dynamic ranges. While the size of the quire is given by Equation 2.6, an analogous for IEEE-754 is given by Equation 2.7, for the Kulisch accumulator:

$$asize = 2e_{max} + 2n + 2|e_{min}| + cg \tag{2.7}$$

2.2.7 IEEE-754 Format Encoding

Floating-point arithmetic operation results are encoded to the IEEE-754 format by translating the sign, exponent, and significand fields to a binary format. Since the position to which a value must be rounded is known when performing this conversion, the encoding is fairly straightforward. It is only necessary to apply a rounding scheme and result selection, according to the special cases defined by the format. Some encodings may also produce some of the following status flags:

- Inexact: set if the result is different from the mathematically exact result of the operation.
- Underflow: set if the rounded value is tiny and inexact.
- Overflow: set if the absolute value of the rounded value is too large to be represented.
- Divide-by-zero: set if the result is infinite given finite operands.
- Invalid: set if a real-valued result cannot be returned (e.g., $0 \times \infty$, $+\infty - \infty$).

Some of the status flags have an associated result. The overflow outputs $\pm\infty$ (depending on the sign), the divide-by-zero and invalid outputs qNaN.

2.2.8 Posit Format Encoding

To encode a posit value, it is necessary to translate the sign (`s`), scale factor (`sf`), and fraction (`f`) fields to a Posit binary format. In Figure 2.13 is represented the common structure of a posit encoding module.

The encoding process starts by detaching the regime value (`k`) from the exponent (`e`), according to the exponent size es. The `k` value's 2's complement is taken and the corresponding encoded regime value is shifted-in (according to `k`'s sign) to the exponent and fraction fields that are concatenated in parallel (`ef`). The resulting signal (`ref`) is rounded. Since posits do not overflow and underflow, when the `k`'s value is greater in the module than the maximum regime value, no round-up is allowed. The rounded value is then 2's complemented according to `s`. Finally, the result is obtained considering the special cases zero and NaR.

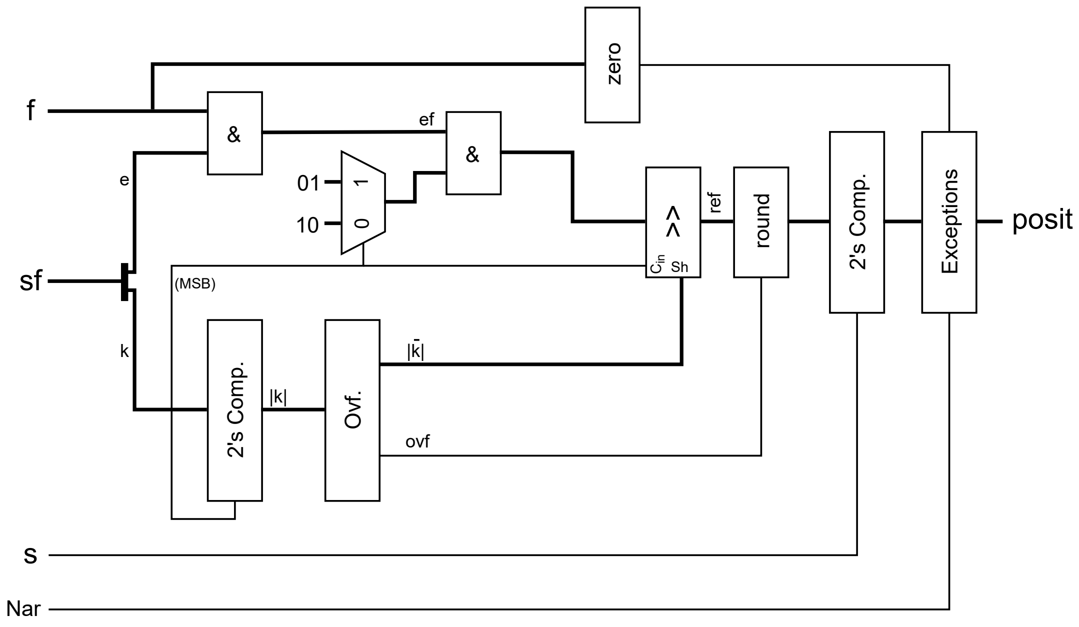


Figure 2.13: Typical posit encoding module.

2.3 Floating-point Hardware Units

Alternative floating-point formats with reduced precision have been gaining attention in the DL and artificial intelligence application domains. In fact, several implementations already exploit low-precision/custom floating-point formats. Such is the case of NVIDIA's Graphics Processing [7] Units and Google's Tensor Processing Units [35], both supporting the FP16 and bfloat16 formats. However, some DL operations do not need as much precision, resulting in the growing interest in exploiting low-precision posits.

The DL inference stage tends to have low sensitivity to errors, often allowing good performance to be achieved with very low precisions (e.g. 8-bit posits [36, 37]). Conversely, the training phase is more challenging and the most computationally demanding stage. Recent results suggest that posits can consistently achieve similar accuracies to IEEE-754, with precisions as low as half of those used by the IEEE-754 standard [8, 38, 39]. However, resources, energy, and performance optimizations are not guaranteed if the posit hardware is more costly. To assess the posit potential, it is necessary to compare posit hardware implementations with IEEE-754 implementations.

Most common hardware units provide support both for the elementary and fused operators, such as addition, subtraction, multiplication, division, and fused multiply-add. Both standards include the first four. However, in what regards fused operators, the IEEE-754 standard [1], only supports fused multiply-add, which requires a rounding step at every multiply-add operation. In contrast, the Posit standard [4], supports FMA and MAC, making use of the quire for fused operations, specializing in continuous accumulation, with rounding errors being introduced only after the last operation.

Format	**Operator**	**Ref.**	**Sup. Device**	**FPGA**	**Lat.**(cycle)	**LUTs**	**Reg.**	**DSPs**	**Freq.**(MHz)
IEEE-754	Add	[44]	FPGA	Zynq 7000 SoC	5	388	252	0	107.5
		[46]	FPGA	Zynq 7000 SoC	6	503	290	0	83
		[47]	FPGA	Zynq 7000 SoC	7	448	360	0	126.6
	Mult	[44]	FPGA	Zynq 7000 SoC	4	263	234	1	161.3
		[46]	FPGA	Zynq 7000 SoC	4	328	153	1	120.5
		[47]	FPGA	Zynq 7000 SoC	12	610	445	0	126.6
	Div	[44]	FPGA	Zynq 7000 SoC	12	823	539	0	92.6
		[47]	FPGA	Zynq 7000 SoC	35	627	525	0	126.6
	FMA	[44]	FPGA	Virtex-7	6	258	296	2	196
		[45]	FPGA	Virtex-7	19	989	1210	3	493
	MAC	[50]	FPGA	Zynq 7000 SoC	12	5300	1900	2	90
Posit	Add	[51]	FPGA/ASIC	Zynq 7000 SoC	0	1103	0	0	31.7
			FPGA/ASIC	Zynq 7000 SoC	5	884	254	0	113.6
		[10]	FPGA/ASIC	Zynq 7000 SoC	0	981	0	0	25
		[25]	FPGA	Zynq 7000 SoC	0	745	0	0	41.7
			FPGA	Kintex-7	22	738	811	0	376
	Mult	[51]	FPGA/ASIC	Zynq 7000 SoC	0	616	0	4	36.1
			FPGA/ASIC	Zynq 7000 SoC	6	802	204	1	108.7
		[10]	FPGA/ASIC	Zynq 7000 SoC	0	572	0	4	30.3
		[25]	FPGA	Zynq 7000 SoC	0	469	0	4	37
			FPGA	Kintex-7	21	544	710	4	413
	Div	[51]	FPGA/ASIC	Zynq 7000 SoC	12	922	538	5	129.9
			FPGA/ASIC	Virtex-7	0	4,050	0	0	21.7
		[53]	FPGA	Virtex-7	0	828	0	0	4.7
	MAC	[25]	FPGA	Kintex-7	40	5068	6256	4	112
		[13]	FPGA/ASIC	Virtex-7	6	4134	1580	4	85
	FMA	[54]	FPGA	Artix-7	0	1740	0	0	18
		[55]	FPGA	Artix-7	0	1797	0	0	21

Table 2.5: Arithmetic unit architectures for IEEE-754 and Posit.

While the hardware for floating-point operators has been extensively studied in the literature [20, 40–50] and the Posit format has arisen the interest of many researchers in the community and there is already a considerable amount of studies for hardware implementations [10, 12, 13, 19, 21, 25, 51–55]. Table 2.5 presents some of the most relevant works with the respective hardware metrics for 32-bits operators. The aim of this table is to assess the posit potential, not to list all the implementations, therefore, only the comparable implementations, in terms of architecture or technology differences, are listed.

IEEE-754 floating-point unit architectures

Since the IEEE-754 standard was established in 1985, there are plenty of hardware implementations for addition/subtraction [40–44, 46, 47], multiplication [44–47] and division [44, 47, 49]. Regarding fused operations, the implementations from [44, 45] correspond to classical FMA architectures, while [20] goes a little further, with native support for vectorization (1×128-bit, 2×64-bit, 4×52-bit or 8×16-bit precision parallel operations). Although accumulation is not compliant with the IEEE-754 standard, there are plenty of MAC units, one recent example was proposed in Fiolhais et al. [50]. There are some differences between the architectures, however, the most relevant corresponds to the subnormal number support, which is only provided by [20,40,42,46,47].

Posit-based architectures

Regarding the posit hardware implementations, studies [10,25,51,52] implement the fundamental arithmetic operators (adder/subtractor and multiplier) while others implement dividers [51, 53]. Regarding fused operations, Forget et al. [25] implement a MAC unit based on the Kulish accumulator, using the quire to store accumulations. Classical FMA architectures [12,54,55] were also implementation. In [13] it is proposed a MAC architecture, which supports both accumulation or addition with a third operand. They also proposed a 64-bit vectorized MAC posit unit (1$\times$64-bit, or 2$\times$32-bit, or 4$\times$16-bit, or 8$\times$8-bit parallel operations) integrated in a reconfigurable tensor unit [19]. More recently, Zhang et al. [21] proposed a Posit vectorized variable precision architecture, however, it only performs multiplication.

For this format, the architectural differences are more significant, in particular, Jaiswal et al. [51] does not concatenate the regime and the exponent, Forget et al. [25] propose a C++ template library and uses a scale factor with bias (just like IEEE-754 floats), Xiao et al. [53] keeps the 2's complement format, and Neves et al. [13] supports different exponents values dynamically through a set of shifters.

IEEE-754 vs Posit

Observing the results represented in Table 2.5 for the referred implementations, it can be concluded that the posit hardware has an overhead in terms of time and resources in both operators. This occurs because of the complexity of the encode and decode steps, as a result of the format's non-uniform encoding scheme. Additionally, the quire has considerable overheads of resources and timing, as expected of a long accumulator.

In terms of division operators, a fair comparison between the formats is not possible since the implementations use different division algorithms, which would only highlight the differences between them, not the differences between the formats. Therefore, regarding the digit recurrence methods [44] and [53] (see Table 2.5) have similar resource utilization. However, the posit divider [53] is not pipelined therefore, a totally fair comparison is not possible. Nonetheless, it can be estimated that both standards have similar resources utilization. In terms of timing, a comparison is not possible because of the referred differences.

The metrics of the IEEE-754 fused operators [44] and [50] (see Table 2.5) reveal the trade-off of having exact accumulation without loss of precision. As such, a significant difference is visible in resource utilization and maximum operating frequency. The same situation occurs if a IEEE-754 FMA implementation is compared with a Posit implementation supporting exact accumulation (see metrics from [25] and [13] in Table 2.5). This reveals a problem for the Posit format, and in general, of exact accumulators, it is an extremely costly architecture, almost prohibitive with more than 32-bits.

In conclusion, for the same precision, posit arithmetic is slightly more costly because of its variable size fields. The quire is also extremely costly in terms of resources, since its size grows exponentially with the precision. However, posits can be used with fewer bits. Which suggests an energy and memory save, and performance boost for several applications, such as DL.

2.4 Discussion

As it was discussed in the previous sections, the Posit format presents several advantages over IEEE-754 floats, such as improved accuracy and larger dynamic ranges with lower precision (fewer bits). These make posits especially suited for low-precision arithmetic. However, posits represent numbers with a non-uniform format. In particular, for smaller values (in magnitude), the Posit format can provide more accuracy and for large values (in magnitude) the accuracy is reduced. This makes it suitable for applications such as DL, but unsuited for other domains, such as particle physics simulations and integration methods, where the numerical values of the result is often unbounded [24]. Moreover, modern computing systems have the IEEE-754 standard deeply embedded in compilers and low-level software routines. Therefore, a drastic replacement of the IEEE-754 standard to the Posit format may be unfeasible [24]. In terms of hardware overhead, for the same precision, Posit arithmetic is slightly more costly because of its variable size fields (i.e., regime). The quire is also extremely costly in terms of resources since its size grows exponentially with Posit precision. Instead of looking at the Posit format as an alternative to the IEEE-754 standard, it is more suited to look at it as a complement. Since their intermediate format (after decoded) is quite similar, an arithmetic unit with a shared datapath that supports both representations is accomplishable and may provide important steps towards the proliferation of transprecision computing.

Regarding the topic of transprecision computing, it was discussed that most hardware solutions [18] attempt to support different precisions by instantiating multiple arithmetic modules to support different precisions. However, recently developed architectures, deploy variable precision units [19–21] with dynamic datapaths that can operate in different precisions with the same hardware resources. They deploy a higher precision arithmetic logic (e.g., 32-bit) and allow parts of the circuit to be turned off to lower the operand precision. This approach not only provides for significant area reductions but also enables straightforward Single Instruction, Multiple Data (SIMD) capabilities.

Accordingly, the work presented in this thesis aims at exploring such opportunities to develop an efficient transprecision architecture deploying a unified Posit and IEEE-754 that that takes advantage of the quire for accumulations in low-precision and configurable to perform operations for other precisions.

2.5 Summary

This chapter starts by introducing fundamental topics on floating-point arithmetic and an introduction to the most relevant formats, including the standard for floating-point (IEEE-754), some of its variants for DL, and the novel Posit format. Next, it is presented a discussion on the structure of the addition/subtraction, multiplication, division, fused multiply-add and multiply-accumulate operators.

Subsequently, it is discussed the recent research work regarding the referred operators for both formats and it is presented a comparison between both formats to assess the potential of adopting the posit format for low precision arithmetic. To conclude the chapter, the problems and solutions were discussed.

3 Proposed Architecture

Contents

In this Chapter, a new Multiply-Accumulate (MAC) architecture for variable-precision floating-point is proposed. It was particularly designed for future transprecision computing and takes a step further from existing solutions by *(i)* deploying a dynamic datapath capable of supporting multiple-precision within the same hardware while making an efficient use of hardware resources by introducing vectorization capabilities; and *(ii)* introducing an unique support for both the Posit and the IEEE-754 floating-point formats. This allows the proposed unit to simultaneously leverage the capabilities of the Posit format to deploy low-precision operations, and maintain support for the well-established IEEE-754 format, while mitigating the hardware overheads imposed by high-precision posit operations.

Accordingly, this chapter begins by introducing the proposed architecture, by providing an overview of its main features and discussing the main design decisions. The second Section describes the adopted vector data formats and details the introduced vectorization by detailing the designed vector arithmetic operators. The last Section provides a detailed description of the proposed MAC unit architecture and its individual modules.

3.1 Proposed Architecture Overview

This section provides an overview of the proposed Vector MAC unit architecture and its main features. In accordance with the opportunities identified in Chapter 2, the herein proposed architecture combines variable-precision arithmetic and dynamic vectorization capabilities, while providing an unified support for the Posit and IEEE-754 formats. The proposed unit features the following properties:

1. Posit-based Variable-Precision Architecture: The proposed unit features a 32-bit Posit fused multiply-accumulate datapath, that targets low-precision arithmetic while providing support to some degree of high-precision, achieved with 32-bit floating-point numbers. To provide variable-precision capabilities, all modules are designed to allow reducing their arithmetic precision at runtime to alternate between 32, 16, and 8-bit operations (as illustrated in Figure 3.1.A). Remembering the reader that the Posit MAC architectures use a long accumulator, the quire, which has significant hardware overheads in high precision. To mitigate these overheads associated with the use of a quire, the proposed unit only provides exact accumulation for low-precision scenarios with standard [4] 8-bit posits ($es = 2$) and 128-bit quire (as opposed to the 512 bits that would be necessary for 32-bit posit accumulation). As such, a scale factor value is paired with the quire to ensure the correct representation of accumulations for all the supported precisions.

2. Dynamic Vectorization: All arithmetic operators and logic modules are fully vectorized and configurable at runtime to support 1x32-bit, 2x16-bit, and 4x8-bit vector operations within the same hardware (see Fig. 3.1.B). This allows resources that are freed (when precision is reduced) to be reused for additional parallel computations, in turn offering increased throughput. To support the introduced

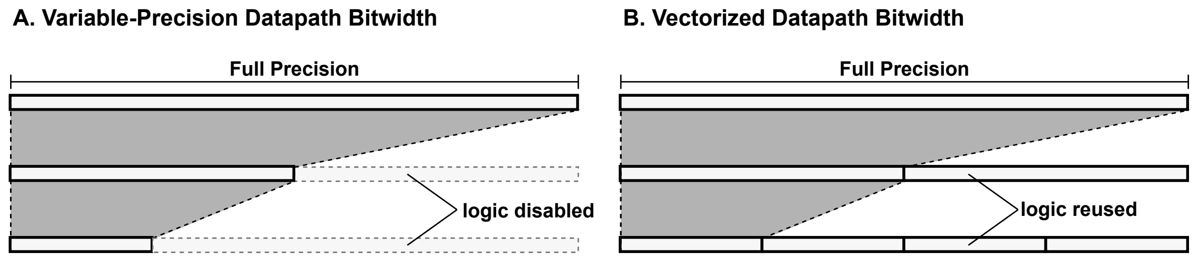

A. Variable-Precision Datapath Bitwidth
Full Precision
logic disabled
B. Vectorized Datapath Bitwidth
Full Precision
logic reused

Figure 3.1: (A) variable-precision vs proposed (B) vector datapath configuration scheme.

variable-precision vectorization, input 32-bit vectors are decoded into three unified vector formats that gather the sign (s), scaling factor (sf), and fraction (f) that compose each vector element precision (more details in Section 3.2.1), according to the $(-1)^s \times 2^{sf} \times 1.f$ generic floating-point format (see Equation 2.1). These vectors are also paired with additional flag vectors to represent special encodings necessary for the floating-point formats.

3. Variable-Exponent Posit Configuration: The Posit format, apart from different precisions, supports different exponents, which allows the adjustment of the dynamic range (see Section 2.1.2 from Chapter 2). In the proposed unit, the Posit exponent size can be defined at runtime (instead of being fixed at design time), allowing most of the dynamic range for a given posit precision to be representable. Since, as mentioned above, the quire is already paired with a scaling factor value, the arithmetic logic can already support dynamic ranges larger than that which can be represented by the quire precision. Accordingly, it is only necessary to include a set of shifters to decode/encode the posit format according to the configured exponent size (described in Sections 3.3.2 and 3.3.6). To optimize the scale factor bitwidth (see Fig. 3.1.B), the maximum exponent range is restrained. Namely, 8-bit Posit can only use an exponent configuration of up to 3, while 16 and 32-bit posits can use an exponent configuration of up to 7.

4. Floating-Point (FP) Format Unification: While the Posit and IEEE-754 formats are fundamentally different in their representation, after being decoded, both represent a FP number in the generic exponential format. As such, the logic to perform typical arithmetic operations (such as multiplication and addition/subtraction) is virtually the same for both formats (as seen in Section 2.2 from Chapter 2). Conversely, to add IEEE-754 support in a Posit base architecture, it is only necessary to include decoding/encoding logic and minimal detection for IEEE-754 mathematical exceptions (not represented in the Posit format). For the particular case of 8-bit precision operations, to match the equivalent Posit precision, it is also adopted an 8-bit format (since the IEEE-754 standard does not define lower than 16-bit precisions). The 8-bit format adopted is the microfloat (see Section 2.1.1 from Chapter 2).

5. Inter- and intra-format Operation and Conversion: The introduced unified FP format also allows the proposed unit to perform inter-format operations between equivalent Posit and IEEE-754 pre-

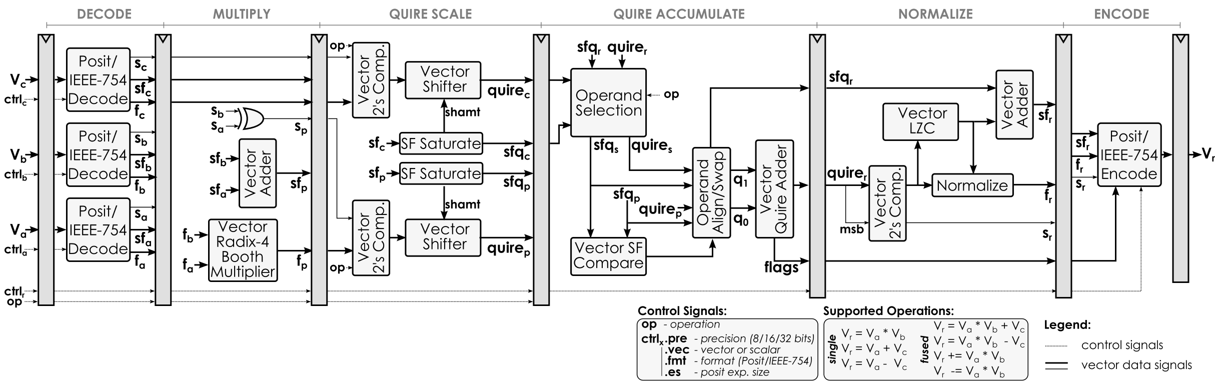


Figure 3.2: Proposed Posit/IEEE-754 unit architecture diagram.

cisions. Since the unit's internal representation is compatible with both formats, it is only necessary to decode each operand according to their specific format (controlled by dedicated configuration signals – see Section 3.3.1). Similarly, the format of the output can also be configured independently of the input formats. As such, it is also possible to perform conversions between formats (with or without performing arithmetic operations). Additionally, the introduced dynamic exponent support also allows the proposed unit to perform intra-format operations and conversions between posits with different exponent configurations. by applying the same methodology of inter-format operations and conversions. An independent signal for each decode and encode module.

The proposed unit (depicted in Figure 3.2) comprises a fully pipelined architecture, supporting vector variable-precision FP addition, subtraction, and multiplication, together with fused multiply-add and multiply-accumulate operations. Accordingly, the unit accepts three input vector operands (V$_a$, V$_b$ and V$_c$), and outputs one result vector (V$_r$). To deploy a datapath with variable-precision support and Single Instruction, Multiple Data (SIMD) capabilities, the data formats and basic arithmetic operators must be defined.

3.2 Arithmetic Vectorization

To efficiently support variable-precision hardware and vectorization capabilities, the data formats used in each configuration must be well defined. Additionally, an arithmetic floating-point MAC unit is made up of various basic structures such as shifters and adders that in a vectorized approach must be reformulated to support the several vector configurations.

3.2.1 Vector Data Formats

To efficiently support the aimed variable-precision hardware and vectorization capabilities, an unified vector data format was devised. The adopted number format can represent 1×32-bit, 2×16-bit, or

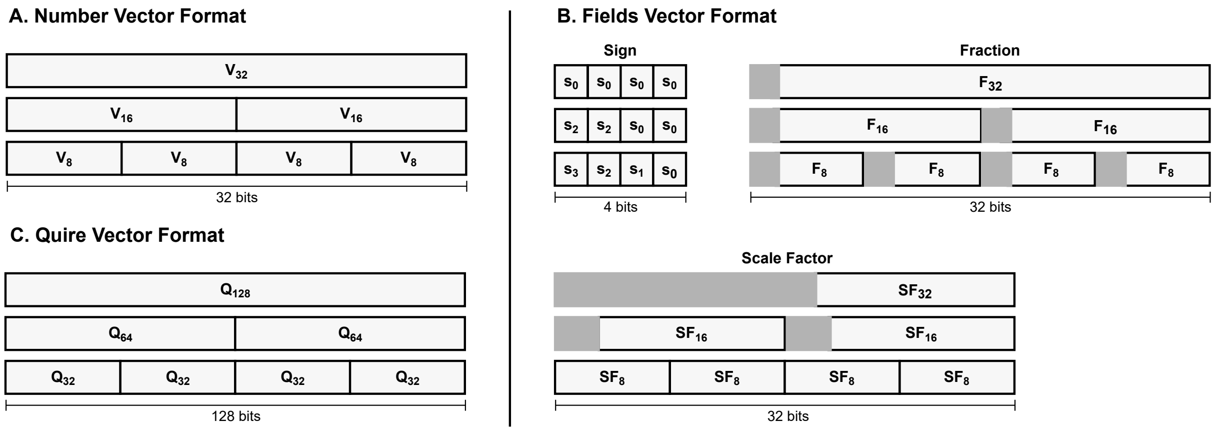


Figure 3.3: Vector data formats for (A) input/output vectors, (B) decoded fields vectors, and (C) quire vectors. Grey areas represent unused bits.

4×8-bit numbers in the same vector (see Figure 3.3.A). Additionally, it is also capable of operating with 32/16/8-bit scalar values, which is particularly useful for full accuracy accumulation with 8 and 16-bit operands. Despite the fact that the proposed unit supports both posits and IEEE-754 floats, each individual vector can only support one of the representations at the same time (e.g., V$_a$ cannot have one 16-bit posit number and one 16-bit float number within the same vector). This cross-format support allows operations between formats (e.g., V$_a$ can be a posit vector and V$_b$, V$_c$ and V$_{result}$ float vectors), as referred above.

Accordingly, the input vectors (V$_a$,V$_b$ and V$_c$) are decoded into three unified vector formats that gather the sign, scale factor and fraction components for each supported vector element precision (see Figure 3.3.B). The sign of each operand is organized into a 4-bit signal. For 8-bit operations, each bit represents the sign of one sub-operand. For a 16-bit and 32-bit operations, the sign is extended to the adjacent sub-element (see Figure 3.3.B).

The scale factor of each operand is organized into a 32-bit signal. This is due to the fact that, for 8-bit operations, the minimum scale factor bitwidth is 4 bits ($es = 0$), however, the unit supports variable exponent configurations. As a consequence, the 8-bit posit configuration supports an exponent (es) of up to 3, which requires four additional bits in each scale factor (3 for the exponent and 1 for overflow protection). Similarly, for 16 and 32-bit precisions, the unit supports a posit configuration with an exponent of up to 7, resulting in scale factors of 13 and 14 bits, respectively.

The fraction of each operand is also organized into a 32-bit signal. Each fraction element has a bitwidth of $n - 2$ bits, where n is the precision. Every vector element has a padding of 2 bits, protecting each element from overflowing to its left neighbor and allowing parallel arithmetic operations. Finally, any unused bits are set to '0' (grey areas in Figure 3.3), with the exception of the scale factor, which is sign extended.

The sign, exponent, and significand components of the IEEE-754 numbers fit in the adopted vector configurations. In fact, the exponent and significand are padded to match the bitwidth of the Posit scale factor and fraction, respectively. Additionally to the referred signals, the flags that are used to represent special encodings (such as Not a Real (NaR), signaling NaN (sNaN) and infinity) adopt the same configuration of the sign vector format.

As referred in Section 3.1, to mitigate the hardware overheads associated with the use of the quire, the proposed unit adopts a reduced quire and introduces a paired scale factor value to ensure the correct number representation. The motivation behind the chosen configuration is discussed below.

Quire Vector Format

The main objective of the quire structure is to represent the entire dynamic range of a given posit configuration for accumulation with full accuracy and overflow protection. As a result, its size is mathematically tied to the precision and the exponent of the posit configuration (see Equation 2.6). To put the quire dimensions in perspective, to support a 32-bit posit configuration with $es = 7$, the quire would have 15392 bits, which is clearly prohibitive in terms of hardware resources and in the context of transprecision computing. In fact, even 8-bit posits with $es = 3$ would take prohibitive dimensions, with 224 bits for every element, resulting in 896 bits for the full four elements. Even if the maximum exponent configuration of every precision was limited to 2 (according to the Posit standard [4]), the quire would require a total of 512 bits: 32-bits posits – 1x512 quire bits; 16-bits posits – 2x256 quire bits; 8-bits posits – 4x128 quire bits. Furthermore, the quire size that would be necessary to perform exact accumulation for IEEE-754 floats would have to be even larger (see Equation 2.7).

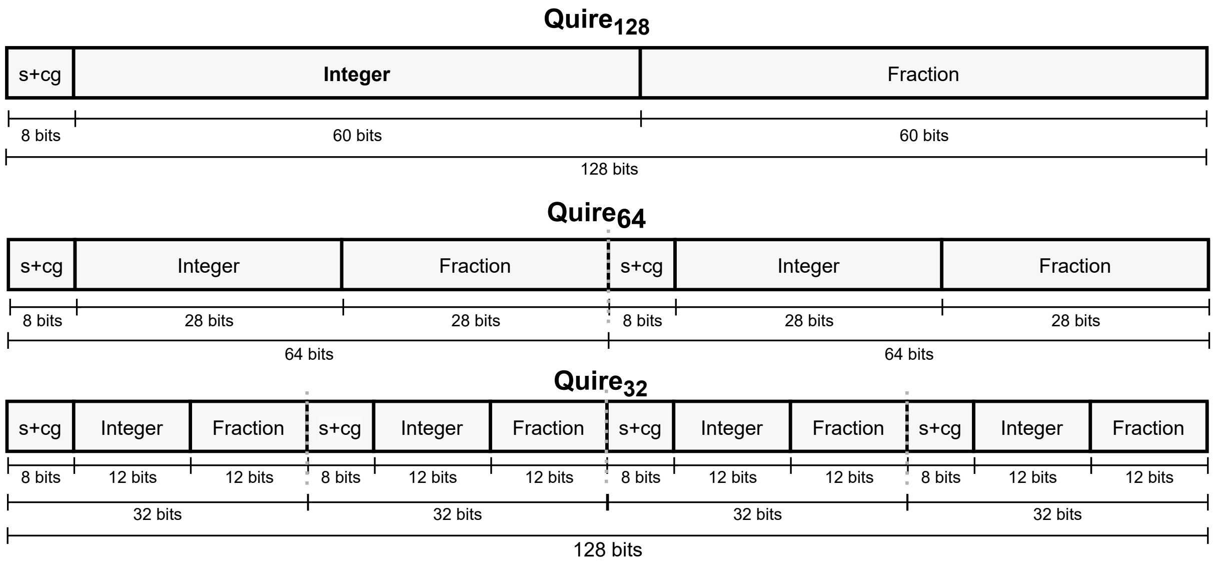


Figure 3.4: Adopted quire vector formats.

To mitigate the quire footprint, while maintaining some of its benefits, the quire size was limited to 128 bits (see Figure 3.4). Hence, to support the defined exponent configurations and the Posit/IEEE-754 unified datapath, the quire maintains an associated scale factor, requiring alignment logic for addition and accumulation.

According to this design decision, the carry guard field of each element of the quire vector is fixed to 7 bits and the integer and fraction fields are divided equally from the remaining bits. By reducing the quire carry guard, only the maximum number of accumulations with overflow protection are reduced. A quire with 128 bits allows support for accumulation without loss of accuracy (i.e., according to the Posit standard [4]) to 8-bit configurations with exponent sizes up to 2 and for 16-bit configurations with exponent sizes up to 1, in full precision mode (scalars). Additionally, the introduction of the associated scale factor and alignment logic allows accumulation with all configurations, albeit with the possibility of accuracy loss, depending on the magnitude of the elements.

The main advantage of the adopted quire format, in addition to a straightforward reduction in hardware resources and increased flexibility, is the fact that for low-precision, most of the benefits of the quire are maintained, while still allowing for high precision, operations without prohibitive overheads. Specifically, it allows a gradual loss of accumulation capacity and accuracy for higher precisions in order to keep the accumulator smaller (128-bit instead of 512-bit) while maintaining support for higher exponent configurations and IEEE-754 floats.

3.2.2 Vector Arithmetic Operators

To fully vectorize the floating-point multiply-accumulate datapath of the aimed unit, the main fundamental arithmetic structures used on floating-point arithmetic (such as adders, barrel shifters, leading zero counters, multipliers) must be redesigned to support variable-precision and vector operations.

The following paragraphs detail the design and architecture of each vectorized structure used in the proposed unit.

Vector Adder

A vector adder can be simply designed by solely breaking a typical adder carry-chain with strategically placed single-bit multiplexers, as represented in Figure 3.5. Such a structure is capable of performing the addition of either 1×n-, 2×n/2-, or 4×n/4-bit vectors. Depending on the configuration, the carry-in of each sub-adder is selected with a multiplexer, between the carry-out of the previous sub-adder or to an input carry-in. This is controlled with split signals, which value depends on the configuration. Specifically, for 4×n/4-bit vectors each split signal is '1', for 2×n/2-bit vectors the signal `splitH` is '1', and for 1×n-bit vectors all split signals are '0'.

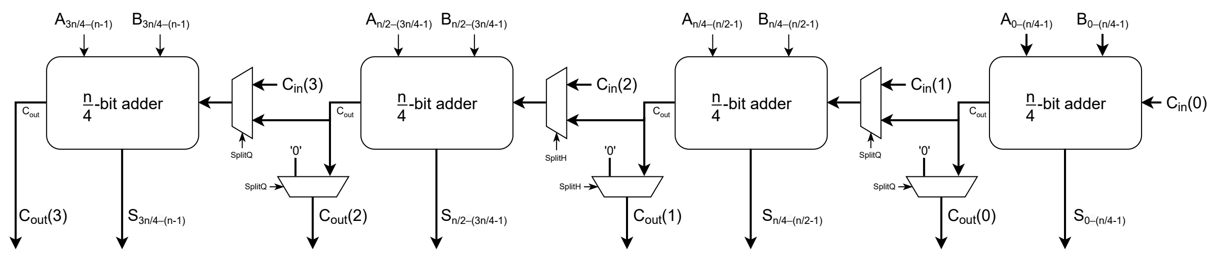


Figure 3.5: Vectorized adder structure.

To do subtraction in this type of structure, first, the subtrahend is inverted and the corresponding carry-in set to '1', according to the vector configuration.

Vector Barrel Shifter

A barrel shifter is one of the most common dynamic shifters, allowing shifting a word by a varying amount. It has a control input (shift amount) that specifies the number of bit positions of an input data word to shift. Figure 3.6 depicts a generic left barrel shifter with a multiplexer-based architecture..

The first step to vectorize a left barrel shifter is to first dimension a barrel shifter for the minimum size of each vector element (e.g. n/4 bits for a n-element vector). Each of these shifters has an associated shift amount (see Figure 3.7), allowing each vector element to be individually shifted.

When considering the aimed variable-precision, it is necessary to add support for configurations with 2×n/2 bit elements. Hence, it is first necessary is to determine which shift amount corresponds to each barrel shifter. Since one n/2-bit value occupies the same bidwidth as two n/4-bit values, two barrel shifters are used to shift each n/2 vector, and the shift amount of each barrel shifter must be the same. However, a n/2-bit element implies one more bit of shift amount (see Figure 3.7). This can be easily performed with an additional shifting level, with the remaining bit.

However, while a normal barrel shifter typically discards the bits that are shifted out (see Figure 3.6), in this case, such bits may contain data for the higher level vector elements. To solve this problem, in

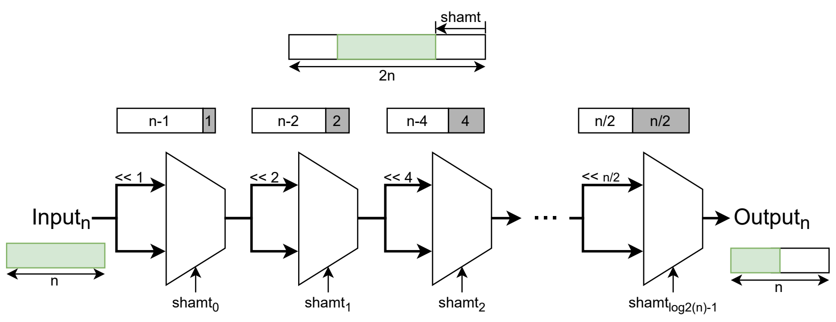


Figure 3.6: Left Barrel Shifter architecture based on multiplexers.

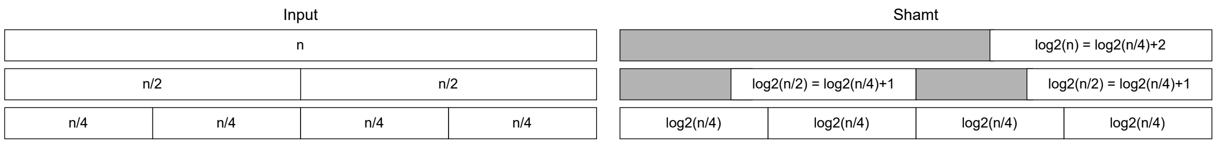


Figure 3.7: Input and shift amount generic vectors format.

each barrel shifter, all data is maintained (each n/4-bit input vector element results in a n/2-bit output signal). These elements must then be properly put together to form the correct signal and correctly shift the higher precision values. The described solution is depicted in Figure 3.8. Note that each `shamt` and each bitwise OR is controlled according to the configuration. This way, each vector element is either cropped or OR'ed with the adjacent vector element to form the high-level element vector. The bitwise OR is the mechanism that allows forming high-level elements, and it is based on the fact that each element is shifted by the same amount, therefore, the interception is between data and 0's.

A similar scheme can also be applied to vectorize right barrel shifters. However, contrarily to left shifting, it is necessary to account for the differences between logic right shifting and arithmetic right shifting, which implies shifting in the sign. In the latter case, the sign is introduced strategically depending on the configuration.

Vector leading zero counter

A leading zero counter (LZC) is a critical structure used both in the decoding/encoding of posits and in the floating-point normalization. It allows counting the number of zeros (from the most significant bit (MSB)) up to the position of the first one in a binary number, outputting the zero count and a flag (zero), which, as the name implies, shows if the input signal has all bits to '0'. A simple generic architecture comprises designing a base LZC module of 4-bits (using simple combinational logic) and arranging them in a tree-like structure to construct higher-order counters.

For example, a 8-bit LZC can be constructed by using two 4-bit modules (see Figure 3.9.A). In the same way, two 8-bit LZCs can construct a 16-bit LZC (and so on). This assembly is done by *(i)* placing an AND gate between the zero flags of each module to get the corresponding higher-order zero flag; *(ii)* extending the zero count of each module with the number of zeros of each adjacent module (concatenating a '0' on the most significant count and an '1' on the least significant count); and *(iii)* selecting the zero count where the first '1' bit appears based on the zero flag of the most significant LZC module.

With such a modular design, to vectorize a LZC, it is only necessary to capture intermediate results to obtain zero counts for each supported precision. Figure 3.9.B depicts the structure of a scalar 32-bit LZC. As can be observed, depending on the desired configuration, each partial result can be obtained and combined in a unified vector format (see Figure 3.9.C). Other length vectorized LZCs can

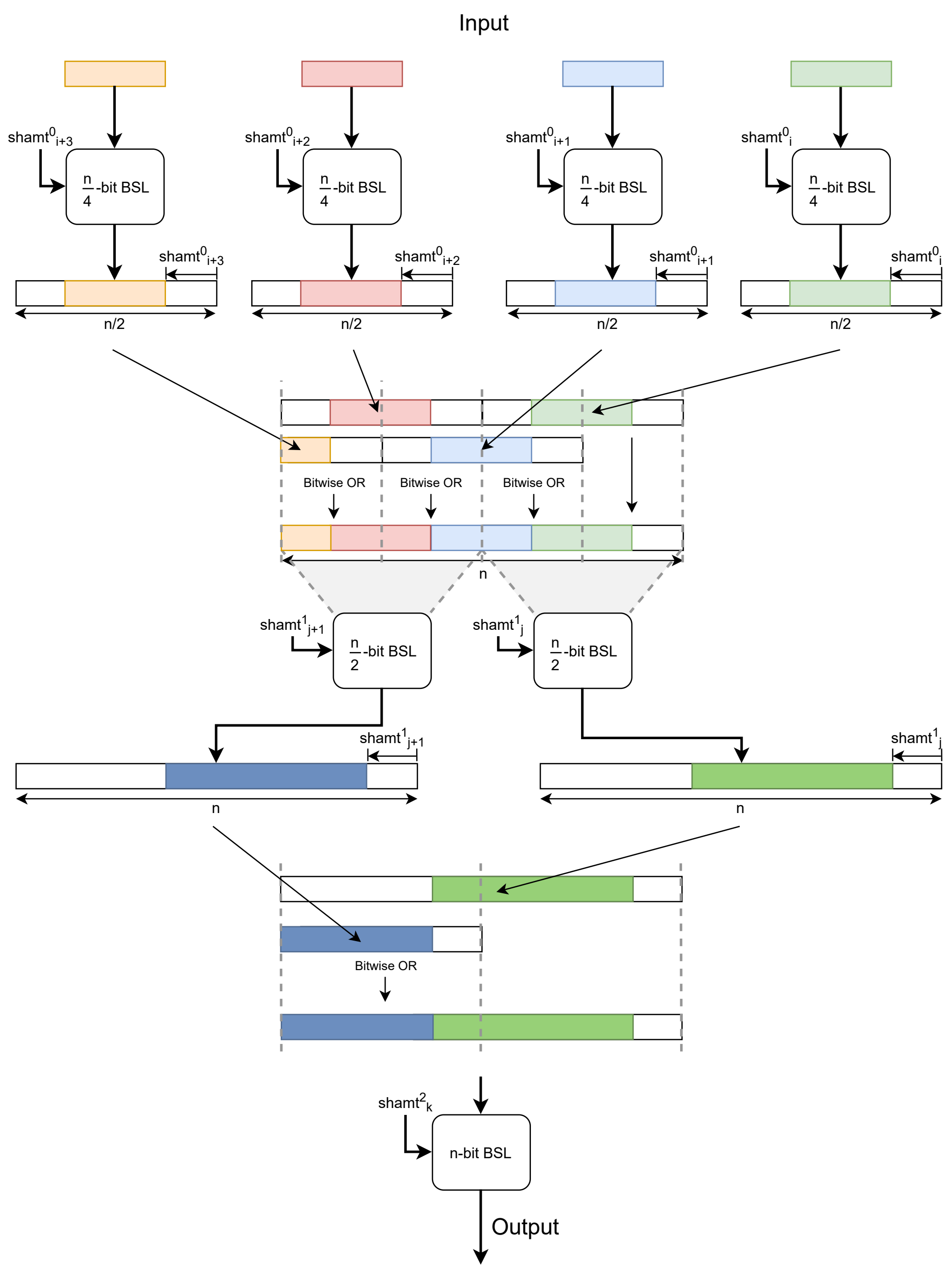


Figure 3.8: Overview of the vectorized barrel shifter architecture. The block diagram illustrates the multi-level architecture of a n-bit left barrel shifter with a maximum of four elements. It shows how partial results can be extracted between each level to obtain shifted value for each supported vector configuration. It also demonstrates the unification of partial results via a bitwise OR operation and their propagation to a subsequent level.

be designed in the same way. For example, in the normalization phase (described below), a 128-bit vectorized LZC with minimum vectors of 32-bits is needed. Instead of unifying the partial results of 8-bits, only 32-bits partial results are unified, up to the 128-bit result.

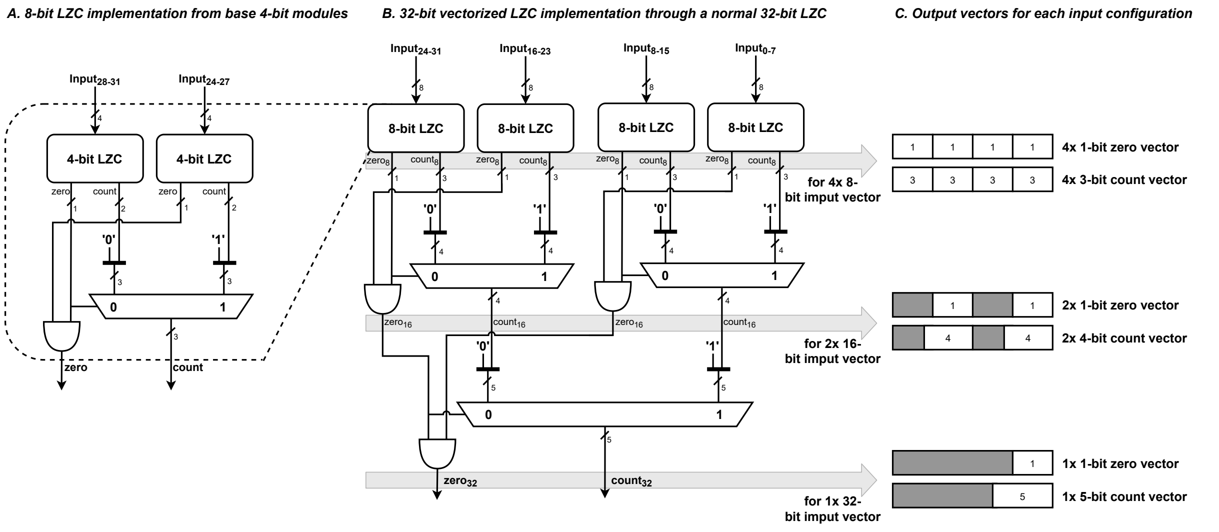


Figure 3.9: Vectorized leading zero counter architecture, illustrating (A) a 8-bit module construction from two basic 4-bit LZC modules, (B) a 32-bit vectorized LZC from four 8-bit LZC modules and (C) the corresponding outputs depending on the precision.

Vectorized Radix-4 Booth Multiplier

The most classic method for multiplying long integer numbers is based on calculating partial products, shifting them to the left, and then adding them together. While in binary, each digit is either 0 or 1. The same logic is still applied (shift and add). To illustrate the multiplication algorithm, an example using unsigned integers is shown in Figure 3.10, for decimal (A) and binary (B) representations.

It can be observed that the number of partial products is exactly the number of digits of the multiplier

Multiplicand (M) = 190_{10} = 10111110_2
Multiplier (X) = 170_{10} = 10101010_2

A. Long decimal multiplication

```
    190
  x 170
    000
   1330
 + 190
  32300
```

B. Long binary multiplication

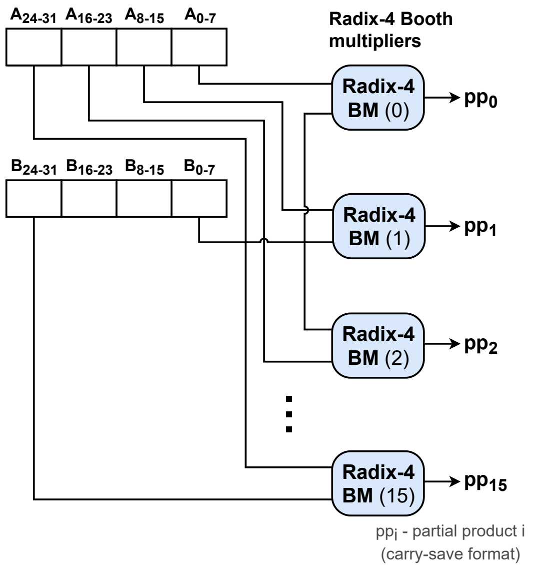
```
         10111110
       x 10101010
         00000000
        10111110
       00000000
      10111110
     00000000
    10111110
   00000000
 + 10111110
  0111111000101100
```

Figure 3.10: Long (A) decimal and (B) binary multiplication example of 190×170.

(i.e., a n-digit multiplier implies n partial products). In general, the partial product section is proportional to the amount of hardware required to sum the partial products and obtain the final product. While it can be reduced by using time multiplexing, it implies a longer operation time. The latency is also related to the height of the partial product section, which may vary depending on the adder algorithm. Nevertheless, adding fewer partial products is naturally a better option, since reducing the number of partial products may reduce the hardware cost and improve performance. This is the idea behind the technique of radix-4 Booth recoding (or encoding), which can halve the total number of partial products in a binary multiplication. Higher radix recoding (radix-8, radix-16, etc) can result in even fewer partial products compared to a radix-4 Booth multiplier. However, the encoding logic is more complex and it is not worthwhile for the desired multiplier. The basics of a radix-4 Booth multiplier are discussed in Appendix A.

For the proposed architecture, a vectorized multiplier is implemented with a 4$\times$4 structure of 8-bit radix-4 Booth multipliers, where segments of 8-bits are multiplied between them, resulting in 16 partial products (see Figure 3.11). Each resulting partial product is gathered and maintained in carry-save format. The obtained carry-save values are then added through a Wallace tree-like structure, resulting in a 64-bit value (see Figure 3.12). Since the multiplier must support all vector configurations, some segments cannot be multiplied at all times – configured by activating and deactivating specific encoders depending on the precision configuration through bitwise ANDs.

Since every fraction element has a padding of 2 bits, according to the adopted vector format (see Figure 3.3), it is not necessary to account for potential overflows to the left neighbor. However, since each Booth multiplier output is in carry-save format, the carry-out might affect the adjacent partial products.



$A_{24\text{-}31}$ $A_{16\text{-}23}$ $A_{8\text{-}15}$ $A_{0\text{-}7}$
$B_{24\text{-}31}$ $B_{16\text{-}23}$ $B_{8\text{-}15}$ $B_{0\text{-}7}$
Radix-4 Booth multipliers
Radix-4 BM (0)
Radix-4 BM (1)
Radix-4 BM (2)
Radix-4 BM (15)
pp_0
pp_1
pp_2
pp_{15}
pp_i - partial product i (carry-save format)

Figure 3.11: Vectorized radix-4 Booth multiplier partial product generation scheme for 32-bit vectors.

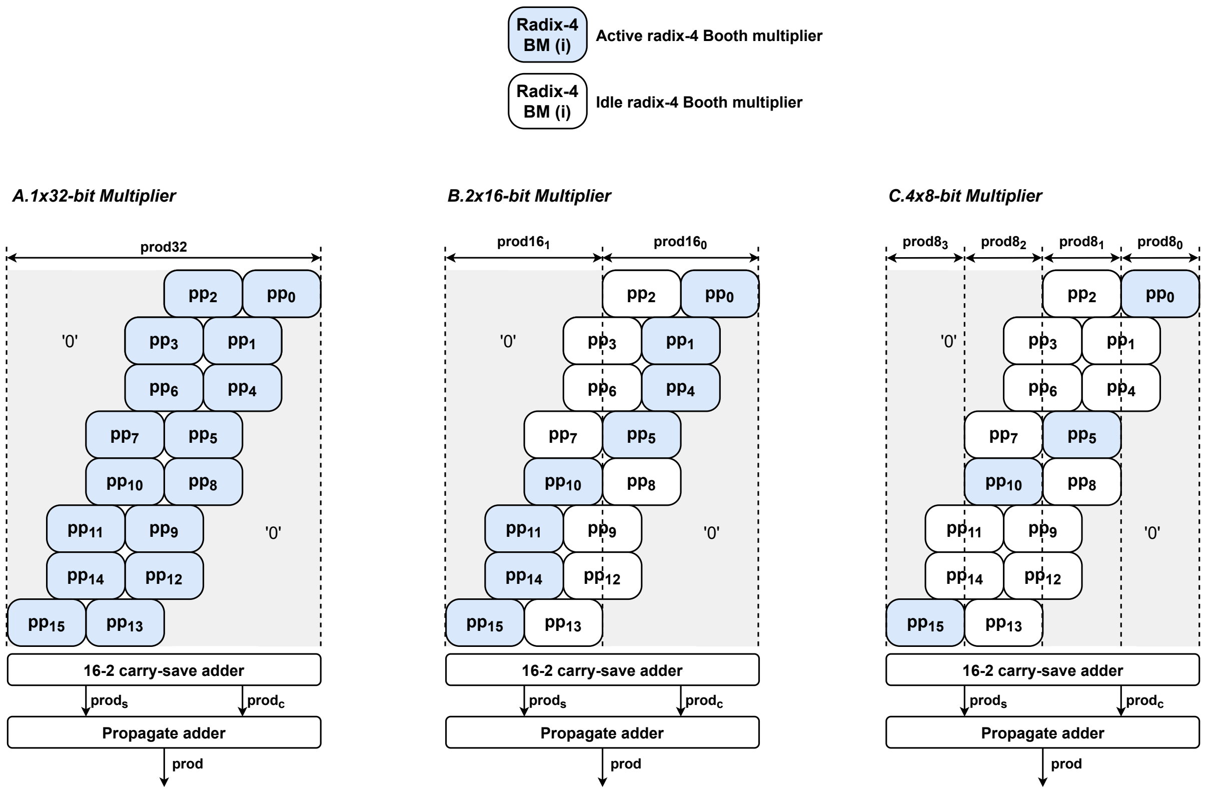


Figure 3.12: Vectorized radix-4 Booth multiplier architecture for 32-bit vectors. The block diagram shows the partial product scheme with radix-4 booth multipliers and each encoder activation for each supported vector configuration: (A) 1x32-bit, (B) 2x16-bit, and (C) 4x8-bit.

This carry-out should not be considered as a product bit, it is resultant of the Booth algorithm and would be discarded if the result was not in carry-save. Therefore, when combining the partial products of each Booth multiplier, these carry-outs must be removed. In the proposed vector multiplier (see Figure 3.12), the sum vector of each carry-save format Booth multiplier result is extended with multiple 1s, in specific positions (not represented in the Figure 3.12) to propagate the undesired carry-outs outside the larger product range.

3.3 Proposed Vector MAC Unit

The proposed unit (depicted in Figure 3.2) comprises a fully pipelined architecture, supporting vector variable-precision FP addition, subtraction, and multiplication, together with fused multiply-add and multiply-accumulate operations. Accordingly, the unit deploys a 32-bit SIMD datapath with unified support for Posit and IEEE-754 FP formats, implemented by a 6-stage pipeline : *i)* **Decode**; *ii)* **Multiply**; *iii)* **Quire Scale**; *iv)* **Quire Accumulate**; *v)* **Normalize**; and *vi)* **Encode**. The unit accepts three input vector operands (V$_a$, V$_b$ and V$_c$), and outputs one result vector (V$_r$), and is capable of operating with

32/16/8-bit scalar values or with 2x16/4x8-bit vectors.

In the **Decode** stage, the input vectors are decoded, according to the control signals, into three unified vector formats that gather the sign, scale factor, and fraction fields. These signals enter the **Multiply** stage, where V$_a$ and V$_b$ are multiplied and V$_c$ are propagated to the next stage. The multiplication is performed using typical floating-point multiplication, however, with vectorized structures. In the **Quire Scale** stage, the multiplication result and the third vector V$_c$ are converted to the adopted quire format with an associated scale factor. In the **Quire Accumulate** stage, addition or accumulation with a previously stored quire value is performed (depending on the operation) after the proper alignment. The result is then re-normalized to fraction and scale factor in the **Normalize** stage. Finally, in the **Encode** stage, the result vectors are transformed in a posit or IEEE-754 result vector, according to the control signals.

Before addressing each stage is in detail, the signals that control the unit functioning such as precision and operation first are addressed.

3.3.1 Control Signals

The several unit configurations and operations are defined through a set of control signal. They include: operation (`op`), precision (`pre`), full precision (`vec`), format (`fmt`) and exponent (`es`) (see Figure 3.2). The operation signal define the arithmetic operation between the input vectors, and the precision defines the vector configuration, 1$\times$32-bit, 2$\times$16-bit, or 4$\times$8-bit. Their encodings and respective function are represented in Table 3.1.

The proposed unit supports operations with three different precisions (8, 16, and 32-bits), which are controlled by the `pre` 2-bit signal. The supported addition, subtraction, multiplication, fused multiply-add, and multiply-accumulate operations are controlled by the `op` 3-bit signal. The MSB of the `op` signal encodes the accumulation operation. The fused multiply-add has the same encoding as the addition operation and the same occurs with the fused multiply-sub and subtraction operations. This is possible since multiplying vector V$_a$ by the fundamental value 1 and adding/subtracting V$_c$ is the same as a fused operation. Forcing V$_b$ to the value 1 is left to external control coverage by the user or the processing system when the unit may be deployed. However, for multiplication, forcing V$_c$ to 0 and performing the operation as if it is a fused operation may produce an incorrect result, when considering the signed zeros of the IEEE-754 format. Therefore, a special operation encoding is established for multiplication. Nonetheless, V$_c$ must still be forced to 0 by external facilities as in the previous case.

The `vec` is a 1-bit control signal that allows operations with scalars. In this mode (full precision), a sub-element vector uses the full quire bitwidth. As an example, an 8-bit operation with a `vec` signal set to high performs the operation encoded in the `op` signal with only one 8-bit element from each input vector. This feature is particularly relevant for continuous accumulation with full accuracy. The same occurs for

Signal	Encoding	Function
precision	0X	32-bit precision
	10	16-bit precision
	11	8-bit precision
operation	000	$V_r = V_a \times V_b + V_c$ or $V_r = V_a + V_c$
	001	$V_r = V_a \times V_b - V_c$ or $V_r = V_a - V_c$
	010	$V_r = V_a \times V_b$
	100	$V_r + = V_a \times V_b$
	101	$V_r - = V_a \times V_b$

Table 3.1: Precision and operation signal encodings and respective function.

16-bit operations. For 32-bit operations, this signal is ignored, since the quire is already being fully used.

The remaining two signals, format, and exponent, are four independent signals, one for each vector (input and output). In particular, `fmt`, is a 4-bit signal, in which bit 0 corresponds to the result format, bit 1 to the V$_c$ format, bit 2 to the V$_b$ format, and bit 3 to the V$_a$ format. The `es` signal is a 12-bit signal, constituted by segments of 3-bits corresponding to each vector. This signal is only used for posit operands. For 8-bit operations, only the first 2 bits of each segment are used. A configurable exponent of 2-bits allows the posit format to be used with an exponent configuration of up to 3 ($es = 3$) (minimum number of bits to represent the standard configuration $es = 2$). This design decision was taken taking into consideration the resulting scale factor vector bitwidth, as addressed in Section 3.2.1. Higher precisions can use higher exponent configuration, up 7.

3.3.2 Decode

The decode module (see Figure 3.13) translates an input vector (`V`$_i$) to the corresponding sign (`s`), scale factor (`sf`), and fraction (`f`) vectors. Furthermore, it signals, through flags, special encodings – not a real (NaR) and zero (`z`), for the Posit format and `NaN` (`qNaN` and `sNaN`), infinity (`inf`) and zero (`z`). for the IEEE-754 format. Since the proposed vector unit supports both formats, the input vector of each decode module has to be processed in dedicated sub-modules. The format is selected with the signal `fmt`$_i$, which may differ between decoding modules, allowing cross format operations, as seen above. Similarly, the `es`$_i$ signal may also differ in each decoding module (even from the encode module), allowing Posit cross exponent operations. These features provide an intrinsic support for conversions, either between posits configurations or between posits and IEEE-754 floats.

Regarding the flags, two aspects stand out in Figure 3.13. One is the fact that IEEE-754 floats have more exception flags than posits. The other is that there is no `qNaN` flag output in the decode module. In fact, the `qNaN` flag is merged with the `NaR` flag. This is possible, since, according to the posit standard [4], NaR converts to quiet NaN (qNaN). It is also stated that all forms of infinity and Not a Number (NaN) convert to NaR. However, only the referred conversion is applied in the decode stage, since including

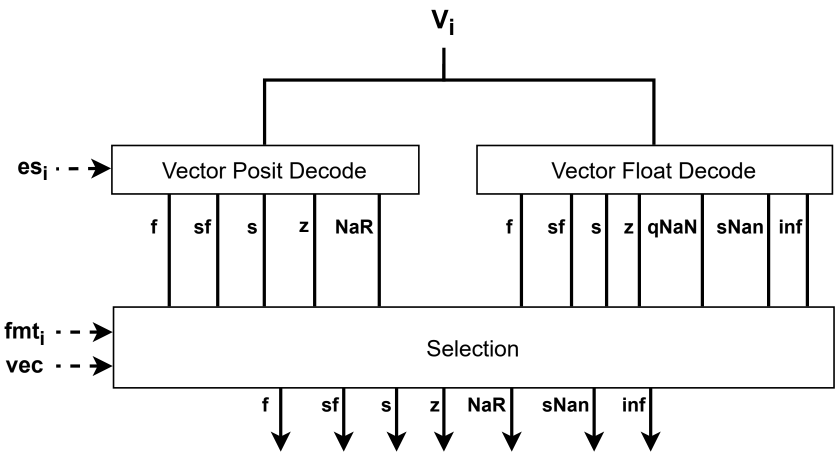


Figure 3.13: Proposed Posit/IEEE-754 decode module.

additional convergence would invalidate IEEE-754 floats and cross-format arithmetic. As such, after the format selection, the `z` and `NaR` flags correspond to a posit or an IEEE-754 float (depending on the input), just like the sign, scale factor, and fraction. The `sNaN` and `inf` flags only have an useful value if the input vector is in the IEEE-754 format, otherwise the bits are set to '0'.

The `vec` signal is used in this stage to adjust the decoded fields for the full precision operation mode. It extends the sign and zero signals as in a 32-bit operation (see Figure 3.3.B), and changes the fraction (according to the precision) to the most significant positions. Additionally, all signals have their unused sections set to '0', through a register reset, which does not apply to the sign and zero signals. The objective of these changes is to perform operations as if they were 32-bit operations, however, only for the fraction and the signals that directly affect it (zero and sign).

Posit Decode

The posit decode stage translates an input posit vector (V_{posit}) to the corresponding sign (`s`), scale factor (`sf`) and fraction (`f`) vectors, while signaling the special encodings Not a Real (`NaR`) and zero (`z`) through flags. In Figure 3.14 it is represented the proposed posit decode module. Its architecture is based on the decode module proposed by Neves et al. [13] which supports dynamic exponent. For each posit vector, the fields extraction follows the procedure from Section 2.2.2, however the used structures are fully vectorized (as described in Section 3.2.2), and two additional shifters and some local ORs are used to deal with the dynamic exponent support.

The `NaR` and `z` flags and the hidden bit logic are also generated according to the unit configuration. The `z` flags are detected by a tree-like structure with three levels. In the first level, each of the four 8-bit segments is checked for zero, which corresponds to the zero signal for 8-bit operations. With these signals, in the second level, each 16-bit segment zero is obtained by two zero signals for 8-bit. In the

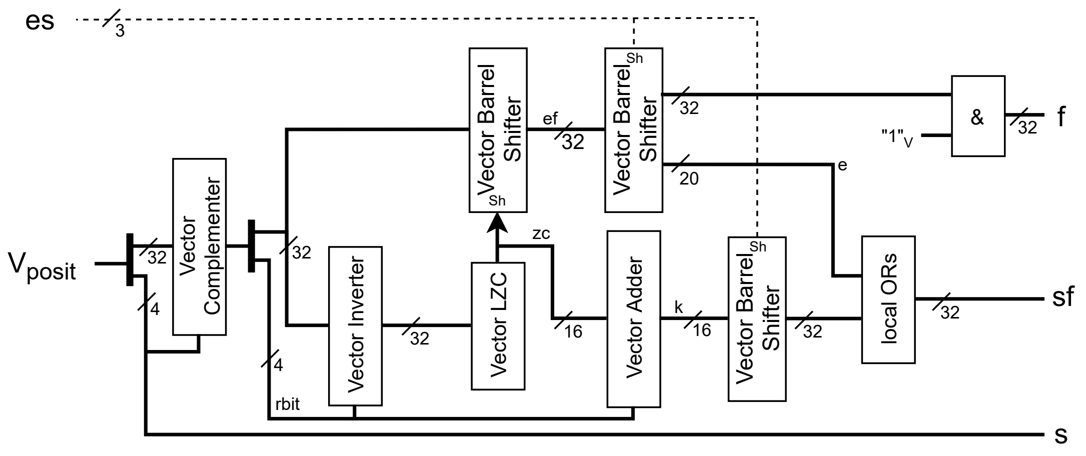


Figure 3.14: Proposed posit decode module.

last level the two zero signals for 16-bit form the zero signal for 32-bit. Depending on the precision, the corresponding zero signals are selected. Since in the Posit format the MSB is the differing factor between a zero and a NaR, the s bit of each vector element in each level is not analyzed, but must be considered in the following levels of the tree. The z signal is latter used to condense the hidden bit and, together with s, derive the NaR signal.

The vector Posit decode module starts by taking the 2's complement according to the s bit of each posit element. The 2's complement corresponds to inverting the signal and adding 1. To invert correctly according to the precision, each s bit controls (through a bitwise XOR) the corresponding 8-bit segment, as if it was a 8-bit operation. This is possible due to the adopted extended sign format (see Figure 3.3.B). To add 1, a vector adder (as described in Section 3.2.2) is used, by taking the inverted posit vector and adding it with 0 and with a carry-in set to the s vector.

Then, to obtain the number of regime bits, each vector element is inverted according to each regime's MSB bit (rbit), with the same logic as the 2's complement inversion step. The inverted signal enters a vectorized LZC (see Section 3.2.2) which counts the number of zeros (number of regime bits) of each vector element. The zero count (zc) has a bitwidth of 16-bits, however, its minimum value is 12-bits since the base 2 logarithm of 8 is 3 ($3\times4=12$). Nonetheless, since the zc will ultimately lead to the scale factor, which is a 2's complement number, each vector element must be extended with 0s according to the precision in the vectorized LZC, hence the 16-bits.

The zc is then simultaneously used to shift out the regime bits from each vector element with a vector left barrel shifter and to obtain the value k encoded in each regime, which is a 2's complement number. The latter is calculated using a vector adder, which uses as inputs zc and rbit signals. The vector adder inputs are defined by analyzing the expression to obtain k (see Equation 2.3). It can be concluded that for a regime starting with 0, zc must be 2's complemented, which involves inversion and adding 1 ($k = not(zc) + 1$). Each segment of the zc inversion is controlled by each rbit bit. However, in this case, it is inverted if the `rbit` is 0, instead of 1, therefore a XNOR is used (instead of a XOR). Adding 1 is done, through the carry-in using the negated `rbit` signal. When the regime starts with 1, `zc` must be subtracted by 1 ($k = zc - 1$). Since one input of the vector adder remains unused, it is added a binary string of 1s, obtained by extending each `rbit` bit to a 4-bit segment.

After shifting out the regime bits, the shifted vector contains the exponent and fraction fields (`ef`) for each element, which are then split according to the exponent configuration (`es`), using a vector left barrel shifter. This shifter is slightly different from others since the vector elements that are shifted out correspond to the exponent (`e`) value. Hence, the shifted out bits must be saved and are not discarded. Additionally, the `es` signal is a scalar shift amount input, which simplifies the shifter, having only one shifting level. The `k` is also shifted according to `es`, however, with another modified shifter. Which, first, extends the input to 32-bits with zeros according to the precision, and then shifts it, forming the regime value.

Subsequently, to obtain `sf`, it is necessary to join the regime value with the exponent. The regime value has the same format of the scale factor (see Figure 3.3.B), while the exponent has a special format, chosen to allow a uniform fusion between both values across all precisions. The fusion can be performed solely through bitwise ORs since overlap between the regime and exponent is mathematically impossible (see Equation 2.4). In Figure 3.15, it is represented the exponent format and the fusion logic. As described in Section 3.2.1 for 8-bit operations, each exponent can only have up to 3 bits, however, for 16- and 32-bit operations, each exponent can have up to 7 bits. Accordingly, bitwise ORs are used to

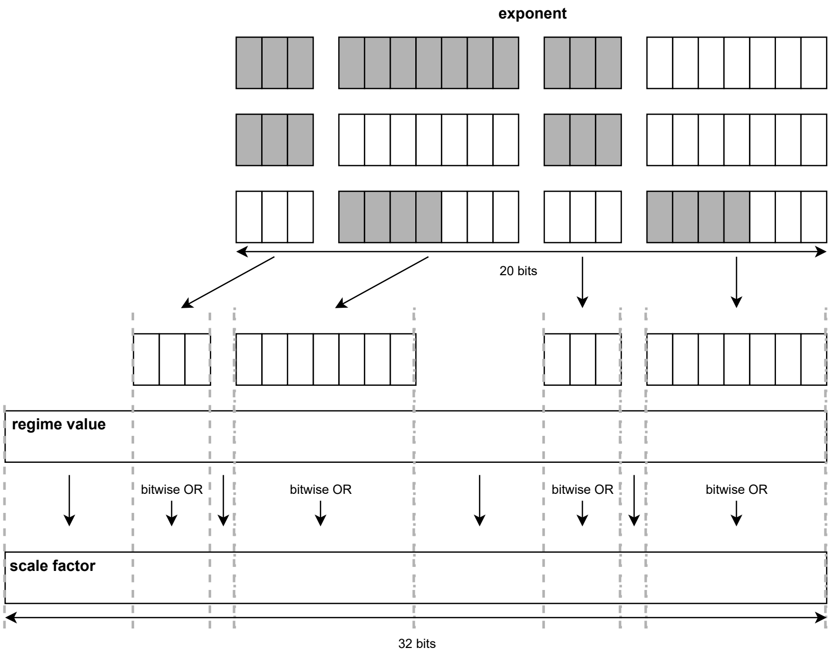


Figure 3.15: Scale factor construction logic. Each rectangle corresponds to a bit. The grey bits are '0'.

fill the overlapable exponent sections with 0s for all precisions. In the sections that do not have bitwise ORs, the regime is directly concatenated to the scale factor.

Finally, the `f` is obtained by concatenating two 0s (since each vector element has a padding of 2 bits for overflow protection in the multiplication stage) and the negated `z` signal to each vector element according to the precision. The negated `z` signal corresponds to the hidden bit.

IEEE-754 Decode

The IEEE-754 decode sub-module extracts the signs, exponents and significands from the input float vector ($\mathtt{V}_{float}$) to form the sign (`s`), scale factor (`sf`) and fraction (`f`) vectors. It also signals the special encodings qNaN, sNaN, infinity and zero through flags. In Figure 3.16 it is depicted a scheme of the proposed IEEE-754 decoding sub-module. Similarly to the Posit decode sub-module, the signs are extracted from the MSB of each vector element, organized and extended depending on the precision in an unified format (see the sign in Figure 3.3.B).

The exponents are directly extracted, grouped in an unified format, and analyzed for potential special encodings: all 0s (`expZ`) – zero or subnormal; all 1s (`expF`) – infinity or NaN. The `sf` vector is formed by adding a negative bias vector to the exponent vector, selected from constant vectors according to the precision. To deal with subnormal numbers, the carry-in of the vector adder is connected with the `expZ` signal, since $exp_{min} = E - bias + 1$ when the value is subnormal. While this affects the exponent for zero inputs, during the following stages, the `z` flag vector is properly analyzed, eliminating any potential errors. In addition, to follow the defined scale factor format (see Figure 3.3.B), the 16-bit unbiased exponent vector must be sign extend to a 32-bit vector, according to the precision.

The `f` vector is obtained by concatenating two 0s and the negated `expZ` signal to each significand element (corresponding to the hidden bit). Similarly to the format adopted for posits, each vector element

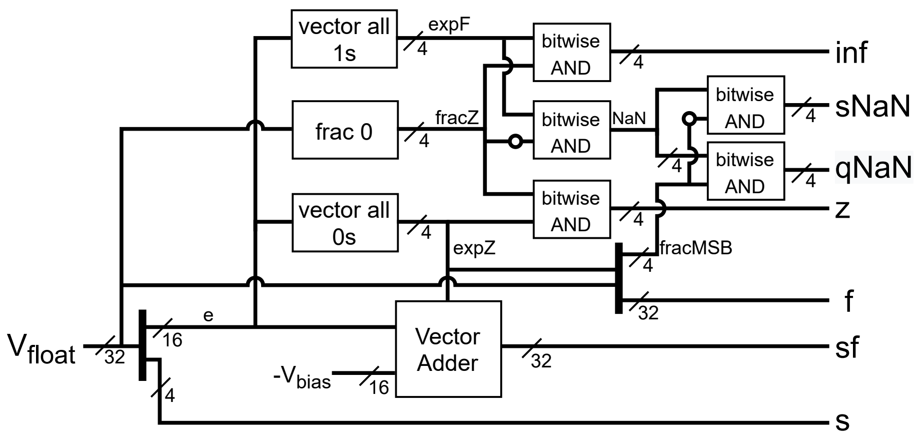


Figure 3.16: Proposed IEEE-754 decode module.

has a padding of 2 bits for overflow protection in the multiplication stage. Due to the bitwidth size differences between the significand of each precision (8-bit – 3 significand bits, 16-bit – 10 significand bits, 32-bit – 23 significand bits) and the adopted fraction format (see Figure 3.3.B), the significand is also prefixed with zeros to fill the 32-bit bitwidth.

Since the IEEE-754 standard has several special encodings, each exponent and significand must be analyzed for special combinations (as it was shown in Table 2.2, from Chapter 2). The exponent is analyzed for all 0s and for all 1s, and the result represented in the already referred signals `expZ` and `expF`, respectively. Each significand is analyzed for all 0s (`fracZ`), directly performed on the input vector. The `z` flag is obtained from the `expZ` and `fracZ` signals, and the infinity (`inf`) from the `expF` and `fracZ` signals, both by a bitwise AND between them. Additionally, the standard also defines two types of NaNs, whose representation can differ depending on the implementation. In the proposed unit, qNaN is distinguished from sNaN by the MSB of the significand (the qNaN has the MSB to '1', while the sNaN has the MSB to '0'). As such, the two different NaN flags are obtained from the `expF`, negated `fracZ`, and `fracMSB`.

3.3.3 Multiply

The Multiply stage (see Figure 3.2) performs the multiplication of the decoded $\texttt{V}_a$ and $\texttt{V}_b$ input vectors, while propagating the decoded $\texttt{V}_c$ vectors to the next stage. Multiplication is performed using the typical floating-point scheme, by performing a bitwise XOR between the signs, adding the scale factors, and multiplying the fractions. However, since the adopted solution provides variable-precision through vectorization, all the necessary arithmetic operators are also vectorized (as described in Section 3.2.2). Hence, to add the scale factors, a 32-bit vector adder is used. Similarly, the fraction components are multiplied with a vectorized multiplier, which is implemented with the same vectorized radix-4 Booth multiplier as described in Section 3.2.2.

A detailed representation of the multiplication is depicted in Figure 3.2 from a scalar perspective. Besides this logic, the unit requires additional logic to deal with the special encodings in the flags and a correction block to ensure a correct conversion to the quire format (see Figure 3.4), as will be described in the Quire Scale module below.

Correction block: Regarding the correction block, the result of the multiplication is in a fixed-point format with 2 integer bits, Q2.x (where x depends on the precision), that must be normalized to the format Q1.y (y=x+1). If the MSB of the fraction multiplication (`ovf`) is '0', the fraction is shifted left. Contrarily, if it is '1', the scale factor is adjusted with the carry-in of the scale factor addition. This method avoids right-shifting the least significant bit (LSB) out in the case of overflow by left-shifting out the '0' instead, solely consisting of a manipulation of the implicit fraction point.

$\lvert x\rvert \times \lvert y\rvert$		$\lvert y\rvert$			
		+0	$+\infty$	qNan	sNan
$\lvert x\rvert$	+0	+0	sNan	qNan	sNan
	$+\infty$	sNan	$+\infty$	qNan	sNan
	qNan	qNan	qNan	qNan	sNan
	sNan	sNan	sNan	sNan	sNan

Table 3.2: Specification of multiplication for IEEE-754 float special encodings.

Special encodings arithmetic: Conversely, the special encodings logic must follow both formats while sharing some of its signals. In the case of the posit multiplication, if one operand is NaR, the result is NaR, if one operand is 0, the result is 0 (except if the other operand is NaR). Since priority can be assigned to one case over the other, there is no need to consider the exception, and an OR between both flags produces the resulting flags.

The IEEE-754 standard is more complex in what concerns the special encodings. The specifications for $|x| \times |y|$ are summarized in Table 3.2. The zero and qNaN are equal to the Posit format and priority can also be applied (sNaN $>$ qNaN $>$ ∞ $>$ 0). The logic for the sNaN and infinity flags is obtained by following Table 3.2 and considering the corresponding. However, since the proposed architecture performs multiplication followed by addition, a special case must be taken care of, the infinity flag, that cannot be generated if the result is a qNaN. This happens due to the addition/subtraction logic with infinity, which may generate a sNaN incorrectly, motivating the final AND gate for the infinity flag.

As seen in Section 2.1.1, the IEE-754 standard specifies five status flags, which are processed in the encoding stage. Hence, the *Invalid Operation* status flag corresponds to mathematically impossible operations (such as $0 \times \infty$) or those involving signaling NaNs. This said, in the proposed architecture,

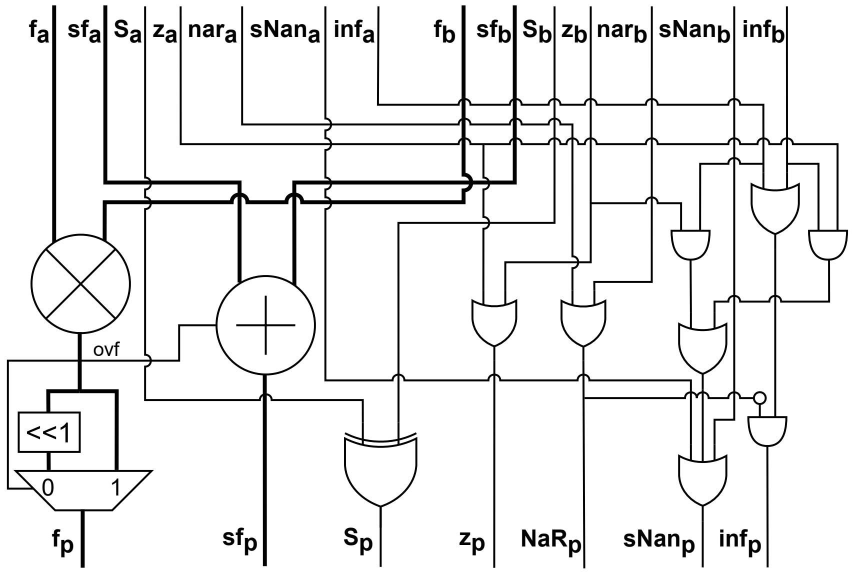


Figure 3.17: Complete Multiply stage in a scalar perspective.

the sNaN flag encodes any invalid operation during the arithmetic stage, which is then evaluated in the encode stage to signal the *Invalid Operation* status flag and the respectively associated result qNaN.

Vectorization: Apart from the vectorization of the adder, multiplier, and XOR, the vectorization of the other logic components was not discussed. However, the signals share the sign format (see Figure 3.3, therefore, the logic gates that process the flags are all bitwise. The correction block is single-bit left shifter controlled by the `ovf` signal, vectorized with a similar scheme as described in Section 3.2.2.

3.3.4 Quire Arithmetic

The Quire Arithmetic module (see Figure 3.2) is divided into two consecutive pipeline stages: *(i)*: Quire Scale; and *(ii)* Quire Accumulate. This design decision was taken to try to keep the critical path to a minimum, given the complexity of the logic necessary to implement the quire and its main structures.

Accordingly, in the first stage (Quire Scale), the operands are converted to a mixed fixed-point quire format (see Figure 3.4) with an associated scale factor. Next, in the second stage (Quire Accumulate), the values are added/subtracted or accumulated (based on control signals), by first selecting and properly aligning the correct operands, between the product vector from the Multiply stage and the operand V$_c$ or an accumulated quire vector.

Quire Scale

The Quire Scale stage is responsible for converting the product vector and operand V$_c$ to the quire fixed-point vector format. The necessary logic is replicated for both operands to allow their parallel conversion and is divided into two steps. First, the fraction is converted to the quire format, and then it is adjusted according to the scale factor.

Conversion: The conversion process starts by taking the 2's complement of the fraction vectors, and then sign extending and right padding them (according to the precision) to form a 128-bit vector. In Figure 3.18 it is illustrated this process for the product vector. The sign vectors are not directly used in this process, since they must first be processed with the operation signal and the zero flag.

Typical floating-point addition or Fused Multiply-Add (FMA) operations do not convert the fractions to a 2's complement format (such as the quire). In fact, the addition or subtraction operations are often controlled by a selective complement of one of the operands and the carry-in of the adder. In this case, since the hardware is the same for all operations, and all fractions are converted to the adopted quire format, the necessary 2's complement step can be used to define the operation (addition or subtraction). Since the unit supports sole multiplication, the default operation, in this case, is addition (with 0).

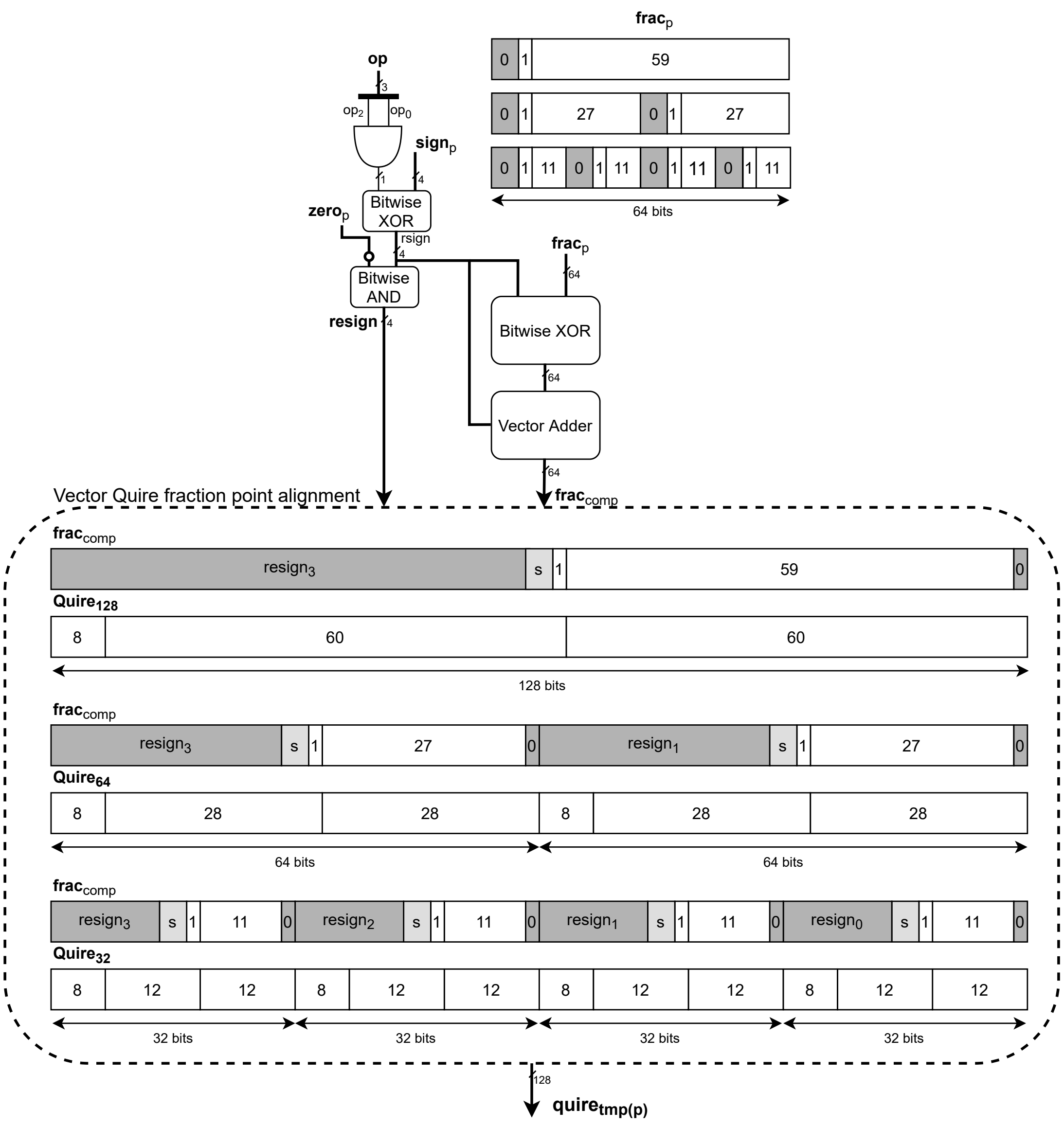


Figure 3.18: Quire conversion of Quire Scale module for the multiplication result.

In particular, to select the operation between addition or subtraction a bitwise XOR is performed between the signs and the LSB of the operation signal (see Table 3.1). However, the products fraction can only be subtracted if the operation is accumulation. This requires an additional bitwise AND with the MSB of the operation signal. The resulting sign vector corresponds to the real sign vector (`rsign`) of the operands, according to the operation (addition – same sign; subtraction – inverted sign). After this step, each fraction element is 2's complemented. The 2's obtained value is then sign extended and right padded to correctly converted to the adopted quire format.

In parallel with the conversion, it is necessary to handle the zero flag. In particular, there is the possibility that elements with a value of zero may have a negative real sign. This is particularly problematic in the sign extension step, since a 0 value would be extended with 1s, which is incorrect. This would only happen for the Posit format in the multiplication of a negative element by a zero element. However, the IEEE-754 standard has positive and negative zeros, making it impossible to manipulate the sign prior to the addition. Therefore, the sign extension is performed by the `resign` signal, obtained by a bitwise AND between `rsign` and the negated zero flag.

Adjustment: At this point, it remains to adjust the integer and fractional part according to the scale factor of each element. For positive scale factors the quire must be left-shifted, while for negative scale factor, it must right shifted. However, the proposed unit supports IEEE-754 floats and several Posit exponent configurations with a reduced quire format, when left shifting, the shift amount cannot be greater than the integer size, and the right shift cannot be performed, since it could shift out the value before the addition. Therefore, the shift amount is calculated from the scale factor dynamic range with the aid of a *saturation module*, which saturates the shifting amount (adjusting the scale factor accordingly) whenever the fixed-point value overflows.

By only left shifting and maintaining the scale factor associated with the quire format and introducing floating-point alignment logic in the next stage, it is possible to support the aimed features. The scale factor saturation logic and quire adjustment are processed in the following way:

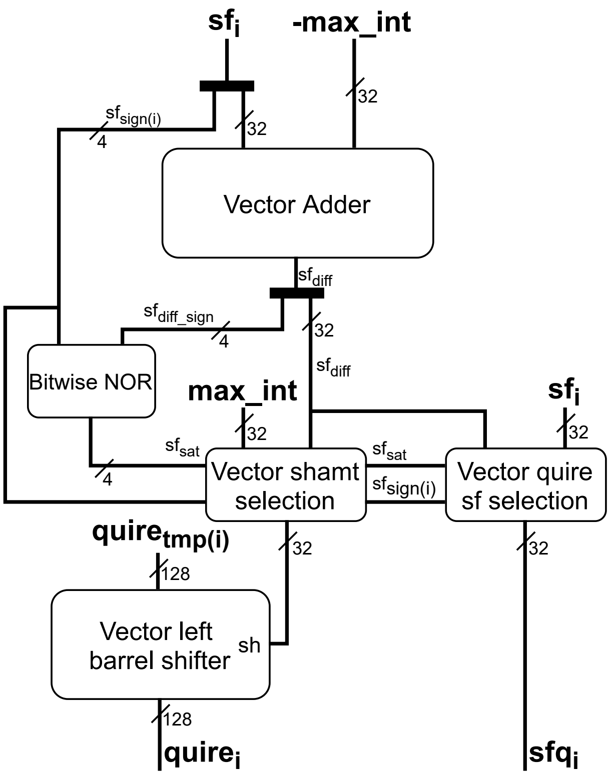


Figure 3.19: Quire adjustment of the Quire Scale module for the *i* operand.

- if the scale factor is negative, the temporary quire (output of the conversion step, see Figure 3.18) is not shifted and the quire scale factor remains unchanged;
- if the scale factor is positive and less than the maximum quire integer value (i.e., it fits the quire fixed-point precision), the temporary quire is left-shifted by the value encoded in scale factor and the quire scale factor is set to 0;
- if the scale factor is positive and greater than the maximum quire integer value (i.e., it does not fit the quire fixed-point precision), the temporary quire is left-shifted by the maximum quire integer value and the quire scale factor is set to the difference.

Figure 3.19 depicts the introduced saturation logic and quire adjustment. As it is shown in the schematic, the scale factor ($\texttt{sf}_i$, where i is the operand, product or V_c) is subtracted by the maximum quire integer value ($\texttt{max\_int}$), according to the precision. Since $\texttt{max\_int}$ is a constant, its negative constant value ($\texttt{-max\_int}$) is added to the scale factor. The resulting scale factor difference ($\texttt{sf}_{diff}$), $\texttt{max\_int}$ and $\texttt{sf}_i$ are then selected to form the quire shift amount ($\texttt{quire}_{sh}$) and the quire scale factor ($\texttt{sfq}_i$).

The selection is performed in segments of 8-bits and each segment is controlled by the scale factor sign ($\texttt{sf}_{sign(i)}$) and the scale factor overflow signal ($\texttt{sf}_{sat}$). The latter is obtained with a bitwise NOR between $\texttt{sf}_{sign(i)}$ and the scale factor difference sign ($\texttt{sf}_{diff\_sign}$). This only occurs if both scale factors are positive, any other case, there is no saturation. Next, in the vector shift amount selection, if the scale factor is negative ($\texttt{sf}_{sign(i)}$='1'), the corresponding 8-bit segment of the $\texttt{quire}_{sh}$ is 0, if there is saturation ($\texttt{sf}_{sat}$='1'), it is $\texttt{max\_int}$, any other case, it is $\texttt{sf}_i$. Then, the $\texttt{quire}_{sh}$ enters a vector left barrel shifter to adjust the temporary quire ($\texttt{quire}_{tmp(i)}$), resulting in the operand fraction converted to the adopted quire format. In the vector quire scale factor selection, if the scale factor is negative ($\texttt{sf}_{sign(i)}$='1'), the corresponding 8-bit segment of the $\texttt{sfq}_i$ is $\texttt{sf}_i$, if there is saturation ($\texttt{sf}_{sat}$='1'), it is $\texttt{sf}_{diff}$, any other case, it is 0.

Quire Accumulate

The Quire Accumulate stage is responsible for selecting the correct operands and performing the required operations (see Figure 3.2). To do so, the registered quire ($\texttt{quire}_r$) or the $\texttt{quire}_c$ (and respective complementary signals, such as the scale factor and the flags) are first selected according to the operation (accumulation or addition, respectively). This selection is performed according to the MSB of the `op` signal. Afterwards, typical floating-point addition alignment is conducted. While in typical FMA architectures, the alignment is performed in parallel with the multiplication, since the proposed unit supports accumulation, the registered value may need alignment. Therefore, the alignment is performed only after the selection. In Figure 3.20 is represented in detail the swap, alignment, and addition logic.

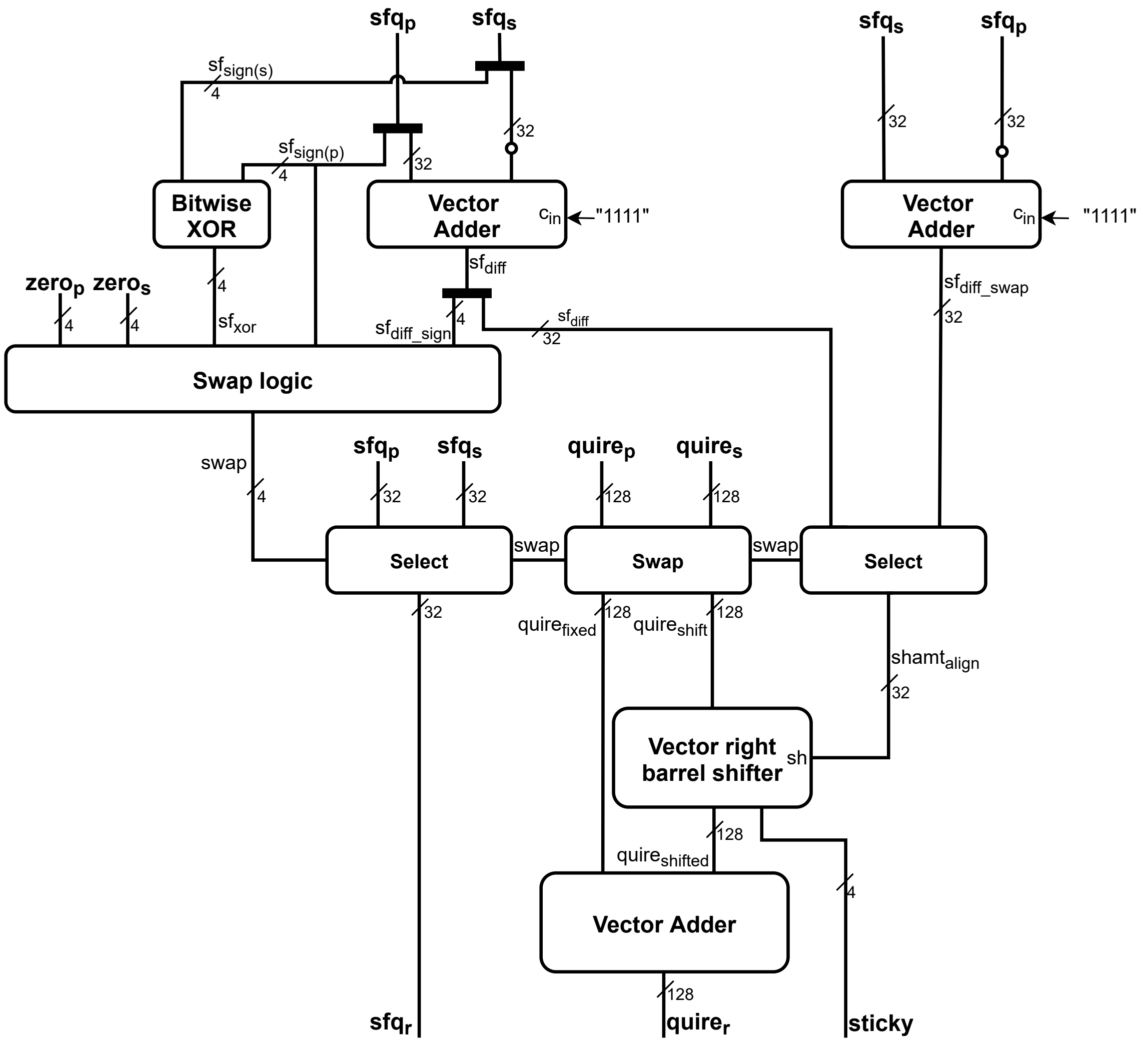


Figure 3.20: Detailed Quire Accumulate arithmetic stage.

Swap logic: The first step for alignment is to determine which operand has the lower scale factor so that the corresponding quire element can be shifted. Similarly to typical floating-point addition, swapping capability is provided so that shifting is only applied to one of the signals (in this case, the one coming from the quire). However, since the scale factor elements are in 2's complement, the swap logic is slightly more complex than typical floating-point arithmetic. In particular, it must be considered the sign of the multiplicand quire scale factor ($\texttt{sf}_{sign(p)}$), the bitwise XOR between the quire scale factors ($\texttt{sf}_{xor}$), the zero flags ($\texttt{zero}_p$ and $\texttt{zero}_s$), and the sign of the difference between each scale factor element ($\texttt{sf}_{diff\_sign}$). The logic behind the swapping is represented in the flowchart from Figure 3.21.A. In Figure 3.21.B the swap logic is represented for the LSB of the `swap` signal.

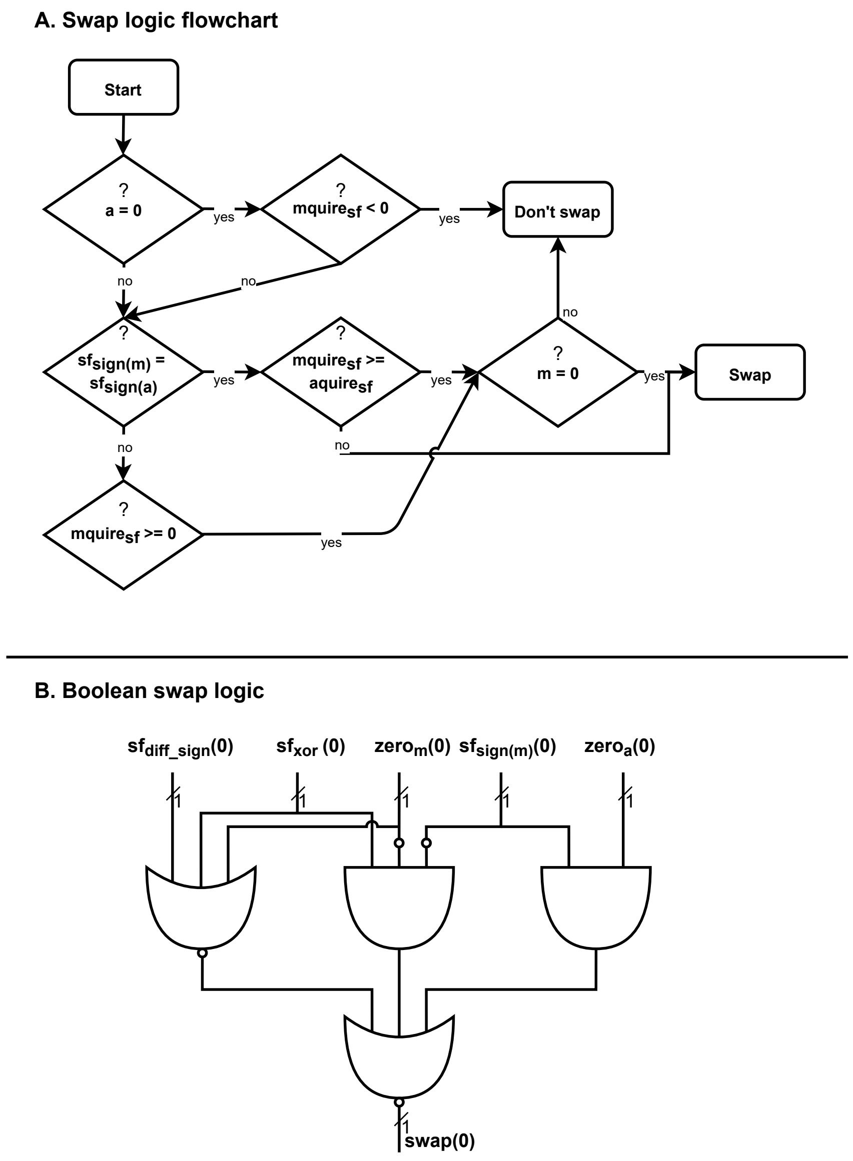


Figure 3.21: (A) flowchart and (B) Boolean swap logic.

Alignment logic: After the swap step, the quire elements with lower scale factors are shifted in a vectorized right barrel shifter (as described in Section 3.2.2). This structure is similar to the vector left barrel shifter, differing in the bitwise ORs positions and the shifted in bits. In this case, since the quire is in 2's complement, the corresponding sign is shifted in. The shifted out bits are condensed in a sticky bit vector, essential for rounding in the encoding stage. The shifting amount is given by the difference between the scale factor elements. Since the difference can be negative, both operations are performed (i.e., $\mathtt{sfq}_s$ is subtracted to $\mathtt{sfq}_p$ and $\mathtt{sfq}_p$ is subtracted to $\mathtt{sfq}_s$). The correct value ($\mathtt{shamt}_{align}$) is selected with the `swap` signal. Additionally, the higher quire scale factor elements are also selected with the `swap` signal, corresponding to the scale factor associated with the result quire ($\mathtt{sfq}_r$). The latter is obtained

$x+y$		y					
		$+0$	-0	$+\infty$	$-\infty$	qNan	sNan
x	$+0$	$+0$	$+0$	$+\infty$	$-\infty$	qNan	sNan
	-0	$+0$	-0	$+\infty$	$-\infty$	qNan	sNan
	$+\infty$	$+\infty$	$+\infty$	$-\infty$	sNan	qNan	sNan
	$-\infty$	$-\infty$	$-\infty$	sNan	$-\infty$	qNan	sNan
	qNan	qNan	qNan	qNan	qNan	qNan	sNan
	sNan	sNan	sNan	sNan	sNan	sNan	sNan

$x-y$		y					
		$+0$	-0	$+\infty$	$-\infty$	qNan	sNan
x	$+0$	$+0$	$+0$	$-\infty$	$+\infty$	qNan	sNan
	-0	-0	$+0$	$-\infty$	$+\infty$	qNan	sNan
	$+\infty$	$+\infty$	$+\infty$	sNan	$+\infty$	qNan	sNan
	$-\infty$	$-\infty$	$-\infty$	$-\infty$	sNan	qNan	sNan
	qNan	qNan	qNan	qNan	qNan	qNan	sNan
	sNan	sNan	sNan	sNan	sNan	sNan	sNan

Table 3.3: Specification of addition and subtraction for IEEE-754 float special encodings.

by adding the the shifted quire ($\texttt{quire}_{shifted}$) with the fixed quire ($\texttt{quire}_{fixed}$).

Special encodings arithmetic: Besides the actual arithmetic operation, it is necessary to include additional logic to deal with special encoding cases. A detailed representation of the exception logic is presented in Figure 3.22, by considering a scalar perspective. The special encodings logic must follow both formats while sharing the same signals. For the Posit format, if one operand is NaR, the result is also NaR, otherwise, the result is given by the arithmetic result. An OR between each NaR flag can detect any operation involving NaRs. The logic between the zero flags is not used for the Posit format, since adding two 0s is not the only operation resulting in 0. Hence a zero result can be detected in the later normalization stage. In fact, the AND between the zero flags is to deal with signed zero operations with IEEE-754 floats (described below).

All the logic represented in Figure 3.22, with the exception of the NaR, is required to deal with the

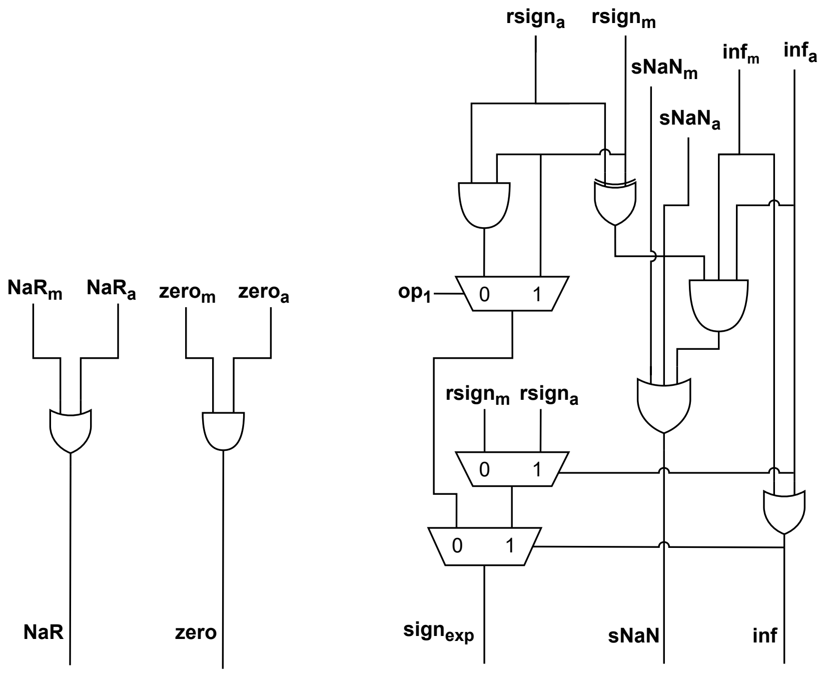


Figure 3.22: Special encodings arithmetic.

IEEE-754 standard special encodings. This logic was developed according with the specifications for $x + y$ and $x - y$ that are summarized in Table 3.3 (for the default rounding mode – round to nearest value). The qNaN (merged in the NaR flag) logic is equivalent to the Posit format, for any operation involving a qNaN results in a qNaN with the exception of the sNaN. Moreover, the priority described for the Multiply stage is also applicable here (sNaN $>$ qNaN $> \infty > 0$). However, it is not necessary to deal with overlapping special encoding operations.

Additionally, any mathematically impossible operations, or those involving sNaN, result in a sNaN. In the addition/subtraction case, this is refered to operations between infinities with opposite signals (see Table 3.3). This is verified with an AND between the infinity flags and the XOR between the real signs (`rsign`, see Figure 3.18).

Finally, to deal with signed special encoding operations (specifically, signed zeros and signed infinities) the signs are processed according to Table 3.3, where the operation (addition or subtraction) is distinguished by $\texttt{op}_1$, which corresponds to the bit 1 of the `op` control signal. Its use is necessary since, in this stage, for the sole multiplication case, the product result is added with $+0$. However, if the product result is -0, adding $+0$ changes the result. Therefore, if the operation is multiplication, the sign of the multiplication is selected. In any other case, the signs are processed with an AND gate, complying with Table 3.3 regarding signed zero operations. The infinity has priority over zero, therefore, the corresponding sign is selected as the exception sign ($\texttt{sign}_{exp}$). This signal is then selected in the normalization stage as the result sign, according to the zero and infinity flags (i.e., if either one of the flags is '1', the sign of the result is the one encoded in the $\texttt{sign}_{exp}$ signal).

3.3.5 Normalize

The Normalize stage (see Figure 3.2) is responsible for re-normalizing the quire vector and extracting the sign (`s`), scale factor (`sf`), and fraction (`f`) vectors. First, the sign of the quire is extracted from the MSB of each quire basic vector element. This sign is used to convert the quire to an unsigned value through 2's complement. However, as referred in the previous section, the signs for signed zero and infinity encodings are packed in the $\texttt{sign}_{exp}$ signal. As a consequence, the sign of the result is selected between the sign extracted from the quire and the $\texttt{sign}_{exp}$ through the zero and infinity flags.

With a vectorized LZC (as described in Section 3.2.2), the number of positions to normalize the unsigned quire are obtained (zero count). With the obtained zero count, the unsigned quire vector is left-shifted by a vectorized left barrel shifter, which structure is similar to the one in the first stage of the Quire Arithmetic. It frames the result to a format similar to the decoded fraction (see Figure 3.3) and condenses any discarded bits in a sticky bit vector. The normalized fraction vector is represented in Figure 3.23, where the L is the integer (leading) bit.

At this step, the scale factor must be adjusted according to the fraction normalization. Specifically, it

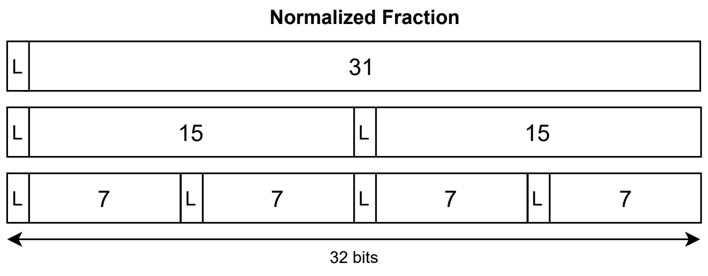


Figure 3.23: Normalized fraction vector.

is summed with a normalization factor, which is obtained from the zero count and an offset. The logic to implement the normalization factor is presented in Figure 3.24, for an example 32-bit operation and by considering positive and negative normalization factors. This logic is used to convert any excess (or shortage) of integer bits to the scale factor, representing a typical conversion from fixed-point to exponential formats.

In parallel with the quire sign being extracted, the value of each quire element is checked for zero. This is necessary for the alignment in the Quire Accumulation, since if the addition/accumulation result is zero due to one of the operands being the inverse of the other, the quire value is also zero, but the corresponding scale factor may be different than zero. When aligning this operand for the accumulation, if the scale factor of the second operand is smaller, it would be incorrectly shifted. The zero flag is then OR'ed with that coming from the Quire Accumulate stage, forming the final zero flag.

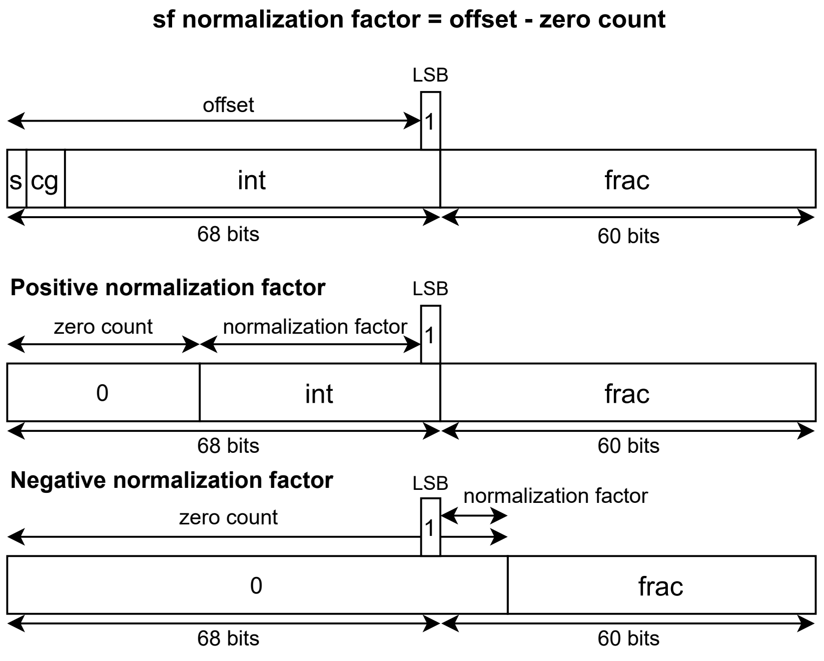


Figure 3.24: Scale factor encoded in the quire.

Additional logic is also necessary in this stage to the NaR flag in case of a quire overflow. In particular, if the number of continuous accumulations exceeds the quire capacity, the NaR flag must be signaled. To do so, the overflow flags are first obtained with typical 2's complement overflow logic (the carry-out vector of the quire addition and the quire sign are XOR'ed and the signs of the operands checked) and then OR'ed with the NaR flag.

3.3.6 Encode

The Encode stage provides the necessary logic for encoding the proposed unit output vectors to the Posit and IEEE-754 formats (see Figure 3.25). The logic is fully vectorized and translates the sign (`s`), scale factor (`sf`), and fraction (`f`) vectors of the result to the selected format vector. Since the proposed vector unit supports both posits and IEEE-754 floats, the result vectors are processed in parallel in the sub-decode modules of each representation and the correct result is selected. The result is selected through the signal *fmt*, which may differ from the decoding modules. Similarly, the `es` signal may also differ from the decoding modules, allowing cross exponent operations. As referred in the Decode module, these features provide intrinsic support for conversions, either between posits (differing on the es) or between Posit and IEEE-754 formats.

Any operation with IEEE-754 as the destination format is processed normally, even if the input vectors are in the Posit format. However, the same does not occur in case the input is in the IEEE-754 format and the result is in the Posit format, due to the IEEE-754 special encodings. If any of the input operands is a IEEE-754 float and the result format is a posit, any special encodings from the IEEE-754 format

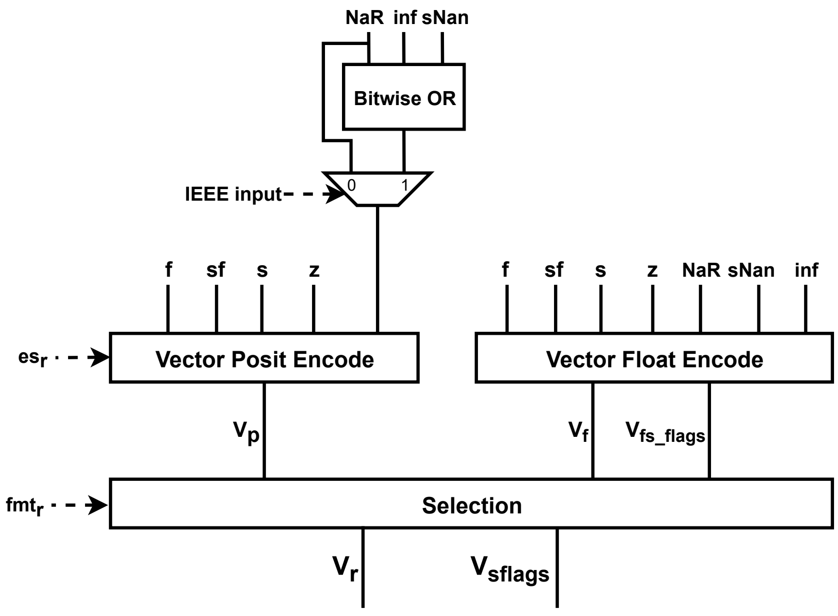


Figure 3.25: Proposed Posit/IEEE-754 encode module.

(sNaN, qNaN and infinity) must be processed as a NaR. As referred in the Decode stage, all forms of infinity, sNaN, and qNaN are merged with NaR. This is accomplished by a bitwise OR between all the corresponding flags.

Posit Encode

The Posit Encode sub-module translates the result vectors to a posit vector. Any special encoding result (NaR or zero) is directly encoded in the posit vector, according to the Posit format [4]. In Figure 3.26, it is represented the Posit encode sub-module. The sub-module is fully vectorized and follows the typical posit encode scheme (see section 2.2.8), while accounting for the variable exponent capabilities introduced in the proposed unit.

Accordingly, the process starts by concatenating the full scale factor and fraction according to the precision. The resulting value is then right-shifted by the `es` value, with the aid of a vectorized right barrel shifter, that is especially designed so that any discarded bits are condensed in a sticky bit vector and to allow the scale factor sign to be shifted in.

The obtained value is then split into the `ef` and `k` vectors, each with 32-bits. Next, the k value is 2's complemented, resulting in the module of `k` (`|k|`), while each element is checked for overflow. In particular, if `|k|` is greater to the maximum that is supported by the precision, the resulting posit saturates at the maximum supported value.

After this step. the two base regime combinations (01 and 10) are concatenated to each element of the `ef` signal according with the respective `k` sign. This signal is then shifted-in according to the saturated value encoded in `k` ($|\bar{k}|$). The shifted-in bits are given by the `k` sign and the result represents the unrounded posit value (`ref`). The discarded bits during shifting correspond to the guard, round, and

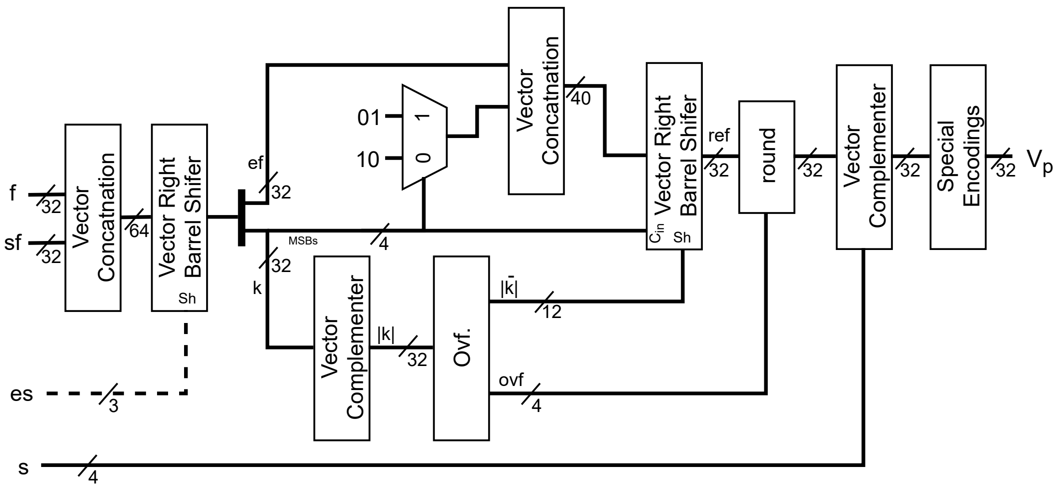


Figure 3.26: Proposed Posit encode module.

sticky bit vectors. Additionally, a '0' is concatenated in the MSB of each vector element (for the sign).

Finally, the `ref` signal is rounded with a round to the nearest even scheme (using the LSB, guard, round, and sticky bits). Afterwards, the 2's complement is taken according to the sign and the correct sign is placed in the MSB. The resulting posit values are then selected between the 2's complement rounded results and special encodings (according to the flags NaR and zero).

IEEE-754 Encode

The IEEE-754 format encode sub-module translates the result vectors to a IEEE-754 float vector, and signals the corresponding status flags. In Figure 3.27, it is represented the IEEE-754 encode sub-module. Since the proposed unit arithmetic datapath is shared for the Posit and IEEE-754 formats, the subnormal cases are only processed in this sub-module.

To encode a result to the IEEE-754 format first the bias corresponding to the adopted precision is added to the scale factor. This results in the biased scale factor signal $\mathtt{sf}_{biased}$, which is then verified if it corresponds to a subnormal, to generate the corresponding subnormal flag (`sub`). In case the value is subnormal, the fraction is adjusted to nullify the exponent and the hidden bit. This is done with the aid of a vectorized right barrel shifter, which saturates any element whose absolute value is greater than the maximum shift amount supported by each element, according to the precision. The correct normalized fraction and exponent ($\mathtt{exp}_{biased}$) elements are then selected between `f` and $\mathtt{sub}_{frac}$, and between $\mathtt{sf}_{biased}$ and zero, respectively, according to the `sub` signal.

Next, the obtained sign, fraction, and $\mathtt{exp}_{biased}$ are concatenated in a 32-bit signal (`prefloat`) according to the precision. Any discarded $\mathtt{exp}_{biased}$ bits have no meaning in the `prefloat` since no overflow or special encoding is generated with $\mathtt{sf}_{biased}$. However, overflow can occur when `prefloat` is rounded (with a round to the nearest scheme) and is handled accordingly. The correct result is then selected be-

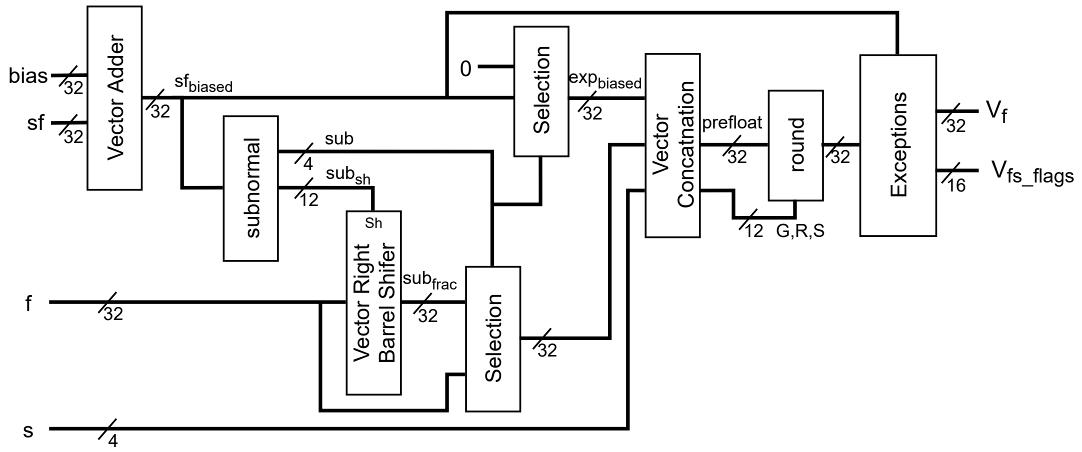


Figure 3.27: Proposed IEEE-754 encode module.

tween the rounded result, zero, infinity, or canonical NaN, according to the flags generated by previous stages and the scale factor value. In the zero and infinity cases, the corresponding sign is concatenated and the sNaN and qNaN flags always correspond to the canonical NaN output and are distinguished by the status flags.

Status flags logic: Regarding the status flags logic (see Figure 3.28), the `sNan` flag is directly connected to the `invalid` status flag. As such any impossible arithmetic operation (such as $0 \times \infty$ and $\infty - \infty$) or involving signaling NaNs results in a signaled `invalid` status flag. The `overflow` status flag is obtained by checking the $\texttt{sf}_{biased}$ bits that stay out of the exponent bitwidth and the exponent of the rounded value exponent (for the cases in which the NaN and infinity flags are not signaled). It is important to note that the `overflow` status flag is not signaled for operations with infinities, it is only signaled if any ”normal” arithmetic operation that exceeds the maximum representable IEEE-754 float value for the corresponding precision. When the `overflow` status flag is signaled, the `inexact` flag is also signaled. Additionally, any loss of accuracy that may occur (`discarded`) also signals the `inexact` status flag (for the cases in which the NaN and infinity flags are not signaled). Any loss of accuracy is encoded in the round, guard, and sticky bit vectors.

The remaining status flag is the `underflow` status flag, which is raised when the result of an operation is inexact and tininess is detected. While the inexact condition was previously resolved (`discarded`), the proposed unit detects tininess after rounding since it results in fewer spurious underflow signals. To determine tininess, the sub-encode module rounds a complementary result to the destination precision without regard for the exponent range. If this result is not in the normal exponent range for the destination precision, then the tininess condition occurs. Specifically, a result is tiny (`tininess`) if the $\texttt{sf}_{biased}$

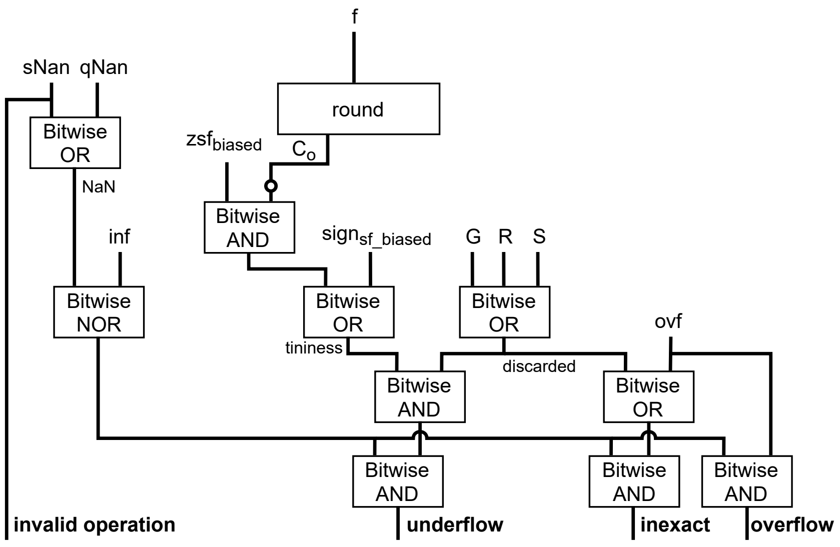


Figure 3.28: IEEE-754 encode status flags logic.

is negative or if the $\texttt{sf}_{biased}$ is zero. However, in the last case, the fraction is also be taken into consideration, since it can produce a round-up that influences the scale factor. Namely, if the $\texttt{sf}_{biased}$ is zero and in the fraction rounding, a carry-out is produced, then tininess does not occur, otherwise, tininess occurs.

3.4 Summary

In this chapter, a vectorized variable-precision arithmetic unit with unified Posit and IEEE-754 format support was proposed. First, an overview of the functionalities of the architecture was presented. Following, it was presented the adopted vector data formats and detailed descriptions of the low-level vectorized components used in the proposed unit. Finally, the proposed unit architecture was presented by detailing each individual pipeline stage and its functionality. The next Chapter evaluates the proposed unit implementation in Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) devices and compares it to other state-of-the-art solutions.

4

Implementation Results

Contents

This chapter presents an evaluation of the proposed unified Posit/IEEE-754 Multiply-Accumulate (MAC) unit architecture in what concerns its resources, timing, and energy efficiency, by considering an Application Specific Integrated Circuit (ASIC) synthesis and an Field Programmable Gate Array (FPGA) implementation.

In the first section, it is discussed the evaluation methodology. The second section shows and compares the proposed architecture with several reference architectures and state-of-the-art solutions.

4.1 Evaluation Methodology

The 32-bit 6-stage pipeline architecture of the proposed Unified Posit/IEEE-754 Vectorized MAC unit was described in RTL, using VHDL, and synthesized using a 28nm UMC standard cell technology [56] and was implemented on a Xilinx Virtex-7 FPGA device (xc7vx485tffg1761-2).

The functionality of the proposed architecture and all its modules was validated using testing vectors generated with the Sigmoid Numbers Julia library [57] and TestFloat [58]. TestFloat corresponds to a set of applications to test whether the implementation of IEEE-754 arithmetic conforms to the standard. It generates millions of test cases and the corresponding results. The test cases were used as inputs in the proposed unit and the respective outputs (result and status flags) were cross-checked for errors. The Sigmoid Numbers library corresponds to a software implementation of the Posit number system endorsed by its developers. It allows generating random inputs and the corresponding outputs. Special tests for Posit arithmetic were also performed for corner cases such as 0, Not a Real (NaR) and the smallest and largest positive and negative numbers that can be represented.

Reference Architectures: To evaluate the proposed architecture, several reference architectures were implemented alongside the proposed unit:

- Three Posit standard MAC architectures with 8-, 16- and 32-bits precisions. All with exponent size of 2, as defined in the Posit standard [4]. These 8, 16, and 32-bit precisions imply the use of 128, 256, 512-bit internal quires, respectively;
- A 32-bit Posit MAC architecture with a dynamic exponent as proposed by Neves et al. [13]. However, with a maximum exponent configuration of 7 and a dynamic configuration for every decode/encode module, as defined in the proposed architecture for a fairer comparison. This unit maintains a standard 512-bit quire;
- A 32-bit MAC architecture with unified support posits and IEEE-754 floats, with a 512-bit quire, a dynamic exponent (up to 7), and cross-format support similar to the proposed architecture.

The Posit standard MAC architectures correspond to units that implement the standard posit configurations. Additionally to being the main reference for comparison (32-bit) with the proposed Vectorized Multiply-Accumulate (VMAC), they will also be used to simulate the cost of typical Posit MAC transprecision implementations, with the same precision mix as the proposed unit (4x8-bit + 2x16-bit + 1x32-bit). The remaining reference architectures will be used to compare the gains of adopting a reduced quire. Namely, to verify if the freed resources are enough to mitigate the vectorization approach. All these architectures were implemented in the same technologies of the proposed VMAC.

State-of-the-art Architectures: The proposed unit will also be compared with state-of-the-art implementations in ASIC technology, including typical Fused Multiply-Add (FMA), dynamic and variable-precision units. Namely, with a 32-bit Posit typical FMA implementation [12] (TFMA), a 32-bit Posit dynamic MAC [13] with configurable exponent size (DMAC), with a 64-bit Posit variable-precision MAC [19] (VMAC), with a 64-bit IEEE-754 variable-precision FMA [20] (VFMA), and a 32-bit Posit variable-precision multiplier [21] (VMULT).

Most state-of-the-art architectures that implement variable-precision datapaths with Single Instruction, Multiple Data (SIMD) capabilities are implemented in ASIC, and the available FPGA metrics only correspond to scalar units. Nevertheless, the proposed VMAC was also implemented and compared with the available state-of-the-art implementations in FPGA technology. Namely, with the 32-bit IEEE-754 Xilinx FMA implementation [44], with a 32-bit IEEE-754 MAC implementation [50], with two 32-bit Posit FMA implementations [54, 55], with a 32-bit Posit MAC implementation [25], and a 32-bit Posit dynamic MAC [13] with configurable exponent size (DMAC).

4.2 Implementation Results

4.2.1 ASIC Implemenatation

The proposed Unified Posit/IEEE-754 Vectorized MAC unit was described in RTL and synthesized using a 28nm UMC standard cell technology [56], by targeting an operating frequency of 667 MHz, under typical operating conditions (1.05 V, 25° C). Chip area and power estimation results were obtained with Cadence Genus 19.11. The synthesis results for the proposed VMAC and all the considered reference and state-of-the-art units are presented in Table 4.1.

The obtained ASIC synthesis results (see Table 4.1) show that the proposed VMAC has a maximum delay of 1.5 ns, which is imposed (critical path) by the Quire Accumulate stage (see Figure 3.2 from Chapter 3). This is mainly due to the logic complexity and bitwidth of the required operations. It has an area of 51563 μm^2 and total power of 99 mW.

UNIT	NUM. BITS	PIPEL. STAGES	ASIC TECH.	DELAY (ns)	AREA (μm^2)	POWER (mW)
Ref. Posit Std. MAC	8	5	28 nm	0.65	7598	21
Ref. Posit Std. MAC	16	5	28 nm	0.8	17384	47
Ref. Posit Std. MAC	32	5	28 nm	0.91	39767	108
Ref. Posit Transp. MAC	8/16/32	5	28 nm	0.91	104927	286
Ref. Posit Dyn. MAC	32	6	28 nm	1.45	54109	134
Ref. Posit/IEEE-754 MAC	32	6	28 nm	1.47	54849	137
Proposed VMAC	**8/16/32**	**6**	**28** nm	**1.5**	**51563**	**99**
Posit TFMA [12]	32	-	28 nm	1.6	10750	10
Posit DMAC [13]	32	5	45 nm	1.5	112350	370
Posit VMAC [19]	8/16/32/64	6	45 nm	1.25	799000	702
IEEE-754 VFMA [20]	16/32/64	3	90 nm	1.5	180610	44
Posit VMULT [21]	8/16/32	-	90 nm	2.3	91861	64

Table 4.1: Comparison of the proposed VMAC with reference architectures and state-of-the-art solutions for ASIC technology.

Comparison with references: When comparing the reference architectures, an increased chip area of 36% and an increased delay of 60% is needed to add dynamic exponent support to the standard Posit reference architecture. In terms of power, this implies an increase of 27%. It can be seen that adding support for IEEE-754 floats to the reference architecture with dynamic exponent support implies minimal overheads. Specifically, an increased area and delay of 1%, and consequently, an increase of power consumption of 2%.

When compared to the reference 32-bit Posit standard MAC unit, it can be seen that despite the introduced variable-precision and unified Floating-Point (FP) functionality, the proposed VMAC only presents a 30% chip area increase, while showcasing a similar power consumption. Furthermore, although a lower operating frequency was expected as a result of the increased complexity, the critical path is still majorly mitigated by limiting the size of the quire to 128 bits (as opposed to the reference 512-bit quire). This is particularly evident when comparing with the reference Posit Dynamic and Posit/IEEE-754 MAC units, which also adopt a 512-bit quire and have similar features as the proposed VMAC (dynamic exponent support and floating-point alignment logic). Furthermore, the maximum operating frequency difference is negligible while the freed resources resultant from the quire reduction allow a 5% and 6% reduction of chip area, and 26% and 28% less power consumption, respectively.

Comparison with state-of-the-art: When compared to a typical scalar FMA architecture (TFMA), it can be observed that the proposed VMAC presents a 4.8x chip area and a 10x power consumption increase while showcasing a similar delay. As would be expected of a FMA architecture, which does not support accumulation. As a consequence, the TFMA has a maximum datapath bitwidth of 62 bits, while the proposed VMAC a maximum datapath bitwidth of 128 bits with alignment logic. Since the TFMA does not provide area and power metrics for a pipelined implementation, the differences in area and power consumption are even larger. In terms of functionality, the proposed unit shows much higher flexibility, in addition to the features provided by the TFMA. In particular, the proposed VMAC supports accumulation, posits with dynamic exponent configuration, and IEEE-754 floats in a shared variable-precision datapath.

Functionality comparison with state-of-the-art: While direct comparisons with other state-of-the-art solutions are hardly possible due to the distinct implementation technologies (28 nm vs. 45 and 90 nm), it is still possible to compare the proposed VMAC in what concerns the offered functionality:

- The DMAC [13] supports more exponent configurations when compared to the proposed VMAC, however, the degree of support is excessive (up to 29) and presents a fixed-precision datapath that lacks the IEEE-754 format support.
- The VMAC [19] is integrated into a hardware accelerator where several 64-bit variable precision Posit units with SIMD capabilities (1$\times$64-bit, 2$\times$32-bit, 4$\times$16-bit, 8$\times$8-bit parallel operations) are implemented, together with a 2048-bit quire, according to an older version of the Posit standard. When compared to the proposed VMAC, it still lacks the Posit dynamic configuration and the IEEE-754 support.
- The VFMA [20] presents a variable-precision architecture, also with similar SIMD capabilities, however, it is bound by its sole adoption of the IEEE-754 format, and cannot perform 8-bit low-precision operations.
- The more recent VMULT [21] also presents low-precision Posit support and variable-precision capabilities (similar to the proposed VMAC), however, it only implements the multiplier datapath and lacks the same flexibility of the VMAC in what concerns the configurable exponent size and support of the IEEE-754 standard.

Comparison with a transprecision setup: Finally, when considering the integration of the proposed VMAC in a typical transprecision architecture [18] to support a variable-precision datapath it is observed that it requires 50% less area and 2.9$\times$ less power than an alternative combination of the reference standard Posit MAC units that offer the same precision mix (4x8-bit + 2x16-bit + 1x32-bit MAC). Additionally, the VMAC offers increased flexibility, by supporting unified support for Posit and IEEE-754 formats with dynamic exponent configurations.

Energy Efficiency Study

To further study the proposed VMAC, an energy efficiency study was also performed (see Figure 4.1) for the proposed VMAC, the reference Posit/IEEE-754 MAC, and a typical transprecision setup (simulated

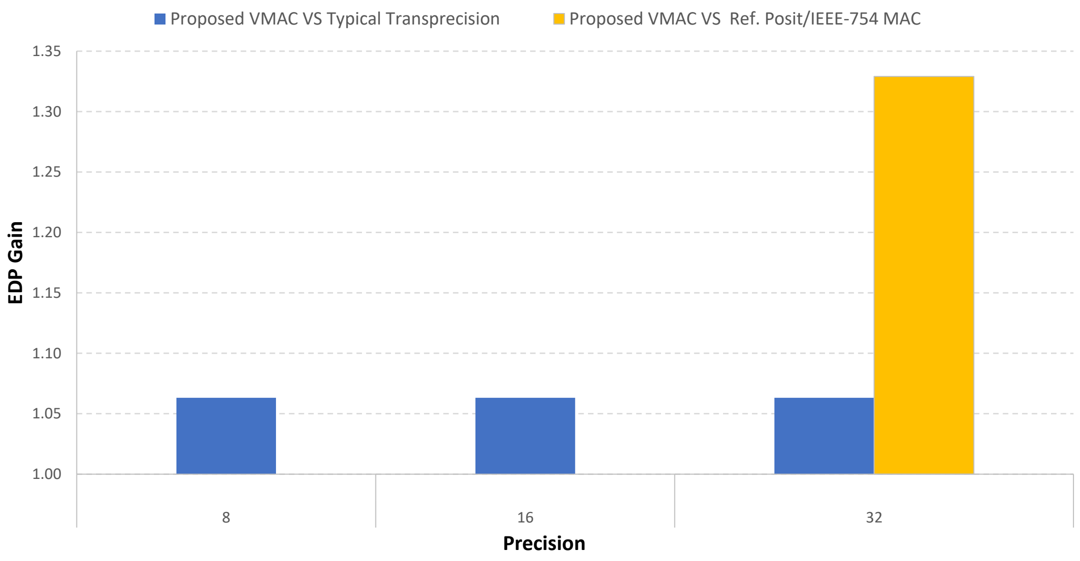


Figure 4.1: ASIC implementation energy-delay product gain against a scalar version of the VMAC and a typical transprecision setup.

by combining the reference standard Posit with the equivalent precision mix). Energy efficiency is herein characterized by an energy-delay product (EDP) metric (see Equation 4.1), which quantifies the trade-off between raw performance and energy consumption. The obtained results for the considered ASIC implementation show that adopting a multiple-precision architecture deploying SIMD capabilities results in considerable gains compared to typical transprecision systems setups.

$$EDP = power \times delay^2 \tag{4.1}$$

It can be seen, that despite all the added functionalities and flexibility, in addition to having less area and power consumption compared with a scalar architecture with similar functionalities, the proposed VMAC presents increased gains of up to 33%. Also, when compared to a typical trasnprecision setup, the proposed unit presents a 6% energy efficiency gain.

4.2.2 FPGA Implementation

The proposed Unified Posit/IEEE-754 VMAC architecture was also implemented on a Xilinx Virtex-7 FPGA device (xc7vx485tffg1761-2). Synthesis and place-and-route results were obtained with the Vivado 2020.1 suite. The implementation results for the proposed VMAC, the reference architectures, and all the considered state-of-the-art units are presented in Table 4.2.

The obtained FPGA implementation results (see Table 4.2) show that the proposed VMAC has a maximum operating frequency of 76 MHz, which is imposed (critical path) by the Quire Accumulate stage (see Figure 3.2 from Chapter 3). It uses 8892 LUTs, 1493 registers, and 0 DSPs, which corresponds to

UNIT	NUM. BITS	PIPEL. STAGES	FPGA	DELAY (ns)	NO. LUTS	NO. REGS	NO. DSPS	D. POWER (mW)
Ref. Posit Std. MAC	8	5	Virtex-7	6.0	1252	341	0	68
Ref. Posit Std. MAC	16	5	Virtex-7	7.4	2951	1073	0	156
Ref. Posit Std. MAC	32	5	Virtex-7	9.0	7389	2462	0	376
Ref. Posit Transp. MAC	8/16/32	5	Virtex-7	9.0	18299	5972	0	960
Ref. Posit Dyn. MAC	32	6	Virtex-7	12.5	9984	1727	0	326
Ref. Posit/IEEE-754 MAC	32	6	Virtex-7	12.8	11321	1754	0	361
Proposed VMAC	**8/16/32**	**6**	**Virtex-7**	**13.2**	**8892**	**1493**	**0**	249
IEEE-754 Xilinx FMA [44]	32	6	Virtex-7	5.1	258	296	2	19
IEEE-754 MAC [50]	32	12	Zynq 7000 SoC	11.1	5300	1900	2	-
Posit FMA [54]	32	-	Artix-7	54.4	1740	0	0	-
Posit FMA [55]	32	-	Artix-7	47.5	1797	0	0	-
Posit MAC [25]	32	40	Kintex-7	8.9	5068	6256	4	-
Posit DMAC [13]	32	6	Virtex-7	11.7	4134	1580	4	-

Table 4.2: Comparison of the proposed VMAC with state-of-the-art solutions for FPGA technology.

2.9% and 0.25% utilization of the FPGA's available LUTs and registers, respectively. Although several of the listed state-of-the-art architectures are implemented in different FPGAs, all the FPGAs have 6-input LUTs, therefore a fair comparison is possible in terms of used resources.

Comparison with reference setups: When compared to the reference 32-bit Posit standard MAC unit, it can be seen that, despite the introduced variable-precision and unified FP functionality, the proposed VMAC only presents a 20% increase of LUTs and due to the reduced quire, a 75% decrease in registers. A 13% decrease in power consumption is also visible. Furthermore, it has a 47% increased delay as a result of the increased complexity. However, the critical path is still majorly mitigated by limiting the size of the quire to 128 bits (as opposed to the reference 512-bit quire). This is evident when comparing with the reference Posit Dynamic and Posit/IEEE-754 MAC units, which also adopt a 512-bit quire and have similar features as the proposed VMAC. As in the ASIC results, the freed resources resultant from the quire reduction allow a reduction of resources and reduced power consumption when compared with the Posit dynamic MAC and the Posit/IEEE-754 MAC.

Comparison with state-of-the-art FMA solutions: When compared to the FMA architectures from the literature, it can be seen that the proposed VMAC presents an higher utilization of resources and lower operation frequency, similarly to the ASIC results. Although the decode and encode modules of Posit architectures impose considerable overheads of both time and resources [25, 51, 55], most of the difference between the Xilinx FMA and the Posit FMA implementations [54,55] is the use of DSPs, which not only reduces the number of LUTs but also a decreases the delay. Hence, when compared to the latter implementations, the proposed VMAC presents an increased number of LUTs.

Since the Posit implementations [54, 55] are not pipelined and are deployed in different FPGAs, a fair comparison in terms of delay is hard to accomplish. Nevertheless, is to be expected an inferior total latency of the units when compared with the proposed VMAC due to the accumulation support. In fact, the accumulation support implies a datapath with operands with more bits (more than double). Despite all the observed increase in terms of hardware resources and timing, it is important to notice that the proposed unit has much more functionalities and higher flexibility, supporting the IEEE-754 standard and the Posit format with different exponent configurations.

Comparison with state-of-the-art MAC solutions: When compared to state-of-the-art MAC implementations, it can be observed that the proposed VMAC presents an increased number of LUTs and has a greater delay, however, their implementations have more pipeline stages and use DSPs, which reduces the multiplier footprint. Since the proposed vector multiplier is intrinsically implemented with logic, architectures with variable-precision support cannot make use of the FPGAs DSPs.

While the Posit MAC units from [13, 25] deploys a 564-bit quire, the IEEE-754 MAC unit [50] deploy a 564-bit accumulator, equivalent to the quire. The latter implements a segmented accumulator which is split into several pipeline stages resulting in a lower delay. The Posit MAC [25] implements a heavy pipeline architecture, which is clearly inefficient compared to the proposed VMAC architecture.

Nonetheless, the adopted quire format allows mitigating the critical path imposed by the quire size, evident by comparing the maximum operating frequency of the proposed VMAC and the MAC units [13, 25, 50]. On the other hand, although the IEEE-754 MAC [50] and the Posit MAC [25] implement their architectures in different FPGAs, they have more than the double of pipeline stages. Which, with the topology adopted in the proposed VMAC, cannot be changed, since it would imply stalling the pipeline for the accumulation.

Comparison with a transprecision setup: When considering the FPGA implementations of the proposed VMAC and the reference Posit standard architectures arranged in a typical transprecision [18] setup, it can be observed that the proposed VMAC requires 2.1x less LUTs, 4x less registers, and 3.9x less power consumption than an alternative combination of reference standard Posit units that offer the same precision mix (4x8-bit + 2x16-bit + 1x32-bit MAC). Similar to the ASIC implementation, the VMAC FPGA implementation offers increased flexibility, by supporting unified support for Posit and IEEE-754 formats.

Energy Efficiency Study

The obtained results for the proposed FPGA implementation showed that adopting a variable-precision architecture deploying SIMD capabilities results in considerable gains compared to typical transprecision systems setups. To further study the proposed VMAC, an energy efficiency study was also performed (see Figure 4.2) for the proposed VMAC, the references Posit dynamic MAC and Posit/IEEE-754 MAC,

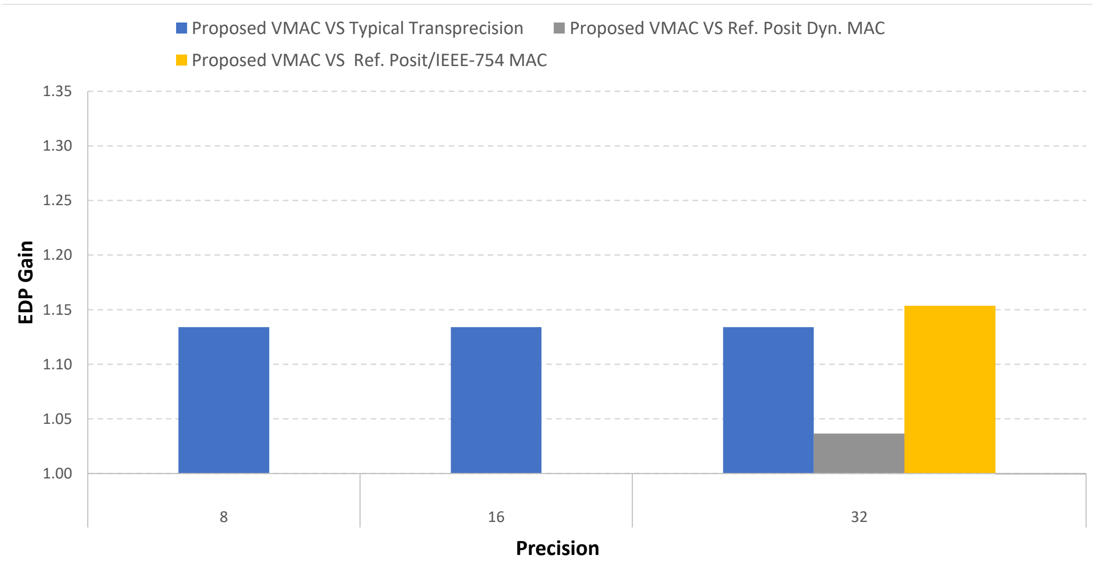


Figure 4.2: FPGA implementation energy-delay product gain against a scalar version of the VMAC and a typical transprecision setup.

and a typical transprecision setup (simulated by combining the reference standard Posit with the equivalent precision mix).

It can be seen, that despite all the added functionalities and flexibility, the proposed unit in addition to having less area and power consumption compared with the reference Posit dynamic MAC and Posit/IEEE-754 MAC, it provides 4% and 15% energy efficiency gains, respectively. Also, when compared to a typical setup, it results in a 13% energy efficiency gain.

In an FPGA, beyond the logic delay, there are significant net delays, associated with the reconfigurable logic. Since the reference Posit/IEEE-754 MAC has more LUTs and registers (i.e., more logic area) than the reference Posit dynamic MAC, the introduced net delay is also more significant. Which explains the differences in efficiency.

4.3 Summary

This chapter provided an in-depth evaluation in terms of resources utilization, power consumption, and performance of the proposed VMAC by considering an ASIC and FPGA implementation. In the first section, the evaluation methodology of the proposed unit was addressed. Specifically, the methods used to test the functionality of the proposed unit, and the architectures used for comparison purposes (developed references and state-of-the-art architectures).

The second section of this chapter presented the ASIC and FPGA metrics for the proposed VMAC, references, and state-of-the-art architectures. A comparison was performed between the proposed unit and the considered architectures in terms of metrics and functionality.

5 Conclusion

Contents

Research efforts have been shifting to the study of more efficient arithmetic circuits to cope with rapidly evolving algorithms advances and the ever-increasing amount of data availability. Transprecision computing has been receiving increasing attention as a viable paradigm to achieve the current computing demands. It is set on the principle that different application domains have different precision requirements. With these architectures, significant performance and efficiency gains can be obtained by adjusting the arithmetic precision of floating-point operations to the application requirements.

As a consequence, the lower the precision requirements of the application, the greater the gains. This approach is being currently used in Deep Learning (DL), where floating-point numbers with precision as low as 4 bits can be used to train neural networks. For floating-point arithmetic, transprecision solutions often rely on the IEEE-754 standard, however, an alternative representation has been gaining attention for low-precision arithmetic, the Posit number system. This format is especially well-suited for low-precision operations since it offers a trade-off between a wider dynamic range and an increased decimal precision, this effectively allows a higher decimal accuracy while lowering the operand precision (fewer bits). It also allows parameterizing the precision and dynamic range (exponent size).

As a result of such properties, it is often observed that posits are more accurate than IEEE-754 floats near zero and less accurate for higher values. A survey on the state-of-the-art Posit and IEEE-754 was carried out and was observed that the Posit standard has an overhead of both timing and resources compared to the equivalent IEEE-754 standard operators, which is a result of its non-uniform encoding. Hence, while this difference is not noticeable when considering low-precision Posit arithmetic, it becomes prohibitive in high-precision scenarios due to the use of a quire for exact accumulation.

A state-of-the-art survey showed that there are recent solutions that propose variable-precision architectures with Single Instruction, Multiple Data (SIMD) capabilities, which are particularly well suited for transprecision computing. They allow resources that are freed (when precision is reduced) to be reused for additional parallel computations, in turn offering increased throughput. Contrarily to most transprecision hardware solutions that rely on instantiating multiple arithmetic modules to support different precisions.

Hence, it was identified an opportunity to explore variable-precision arithmetic with dynamic vectorization capabilities while providing unified support for the Posit and IEEE-754 formats. However, to mitigate the hardware overheads associated with the adoption of the Posit format for high-precision arithmetic, the use of a reduced quire with associated saturation and alignment mechanisms was considered. The deployed mechanisms allow maintaining the accumulation benefits for low-precision arithmetic while gradually losing accumulation accuracy for higher precisions, ultimately enabling hardware and power savings. Additionally, the deployed mechanisms also allow the introduction of dynamic exponent support for posits by adding a set of registers in the respective encode and decode steps.

Finally, an Multiply-Accumulate (MAC) unit architecture was proposed that takes a step further in the transprecision paradigm, by deploying variable-precision arithmetic datapath with SIMD capabilities (1×32-bit, 2×16-bit or 4×8-bit precision parallel operations) which can adapt its precision and floating-point format to the application. Accordingly, it is capable of performing low- and high-precision floating-point operations with the IEEE-754 and Posit formats. The deployment of the shared Posit/IEEE-754 datapath not only provides support for operations with both formats in the same unit but also cross-format and intrinsic support for conversions.

The proposed architecture was successfully implemented in Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) technologies and compared to the state-of-the-art and typical transprecision setups. The 28nm ASIC implementation resulted in 50% less area and 2.9× less power consumption when compared with a typical transprecision system topology. The FPGA implementation resulted in 2.1x less LUTs, 4x less registers, and 3.9x less power consumption when compared with a typical transprecision system topology. Moreover, an energy efficiency study was also conducted, where the proposed unit outperformed the typical transprecision setup in both technologies by 6% and 13%, respectively.

5.1 Future Work

As a result of the work presented in this Thesis, several potential future research directions and improvements to the architecture can be considered:

- Optimize the use of the Quire stages by developing specialized operation encodings for fused multiply-add, addition, and multiplication with associated power gating mechanisms (for ASIC implementations);
- Introduce more operations to the architecture such as sign injection, minimum and maximum detection, and other supported operations in the floating-point RISC-V ISA [59];
- Introduce cross precision conversion mechanisms;

The flexibility that is provided by the proposed architecture capabilities allows for the integration in several applications such as accelerators, embedded systems, edge computing, internet-of-things, and even in a RISC-V processor.

Accordingly, the proposed VMAC can replace the VMAC designed for the Reconfigurable Tensor Unit (RTU) [19] accelerator since it provides increased flexibility and support for more general computations. Despite all the added functionalities, the proposed unit does not support 64-bit operations (double-precision) as the VMAC from the RTU. If double-precision is necessary, two solutions can be proposed. The first consists of supporting more vector configurations (1×64-bit, 2×32-bit, 4×16-bit, 8×8-bit parallel operations), extending the proposed architecture for double-precision, which would imply a quire with 256 bits. This solution would result in a much smaller than the deployed in the RTU (2048 bits) while supporting accumulation with the same accuracy for low-precision. The other solution is to add support for IEEE-754 double-precision operations (which decode and encode require less hardware resources than the Posit analogous modules) with minimal hardware logic addition. Since the Quire Scale and Quire Accumulate have 128-bit bitwidth, by deploying a bigger multiplier, support for 64-bit IEEE-754 floats is possible due to the resulting product fraction size (106 bits).

The RISC-V is an open-source ISA that natively supports computations on single and double-precision IEEE-754 floats. Furthermore, the ISA allows adding support for other instructions by extending the ISA. In addition to the native single-precision floating-point extension of the RISC-V ISA, several useful extensions to integrate the proposed architecture in the RISC-V ISA are available, such as an ISA for Posit [55] and a vector (RVV) [60] extension. With these as starting points, the proposed architecture can be integrated into a RISC-V processor.

Bibliography

[1] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[2] Shibo Wang. (2019) BFloat16: The secret to high performance on Cloud TPUs. Accessed on 9-July-2021. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[3] J. L. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, 2017.

[4] P. W. Group, "Posit Standard Documentation," *Release 4.12-draft*, Jul. 2021.

[5] L. Sousa, "Nonconventional computer arithmetic circuits, systems and applications," *IEEE Circuits and Systems Magazine*, vol. 21, no. 1, pp. 6–40, 2021.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[7] NVIDIA, "Nvidia tesla v100 gpu architecture." *White paper. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/voltaarchitecture-whitepaper.pdf*, 2017.

[8] G. Raposo, P. Tomás, and N. Roma, "Positnn: Training deep neural networks with mixed low-precision posit," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 7908–7912.

[9] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 7686–7695.

[10] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 334–341.

[11] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, “Deep positron: A deep neural network using the posit number system,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1421–1426.

[12] H. Zhang, J. He, and S. Ko, “Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications,” in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[13] N. Neves, P. Tomás, and N. Roma, “Dynamic Fused Multiply-Accumulate Posit Unit with Variable Exponent Size for Low-Precision DSP Applications,” in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, 2020, pp. 1–6.

[14] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn, “The transprecision computing paradigm: Concept, design, and applications,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1105–1110.

[15] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, “Ultra-low precision 4-bit training of deep neural networks,” *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[16] D. H. Bailey, “High-precision floating-point arithmetic in scientific computation,” *Computing in science & engineering*, vol. 7, no. 3, pp. 54–61, 2005.

[17] Q. Xu, T. Mytkowicz, and N. S. Kim, “Approximate computing: A survey,” *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

[18] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, “A transprecision floating-point platform for ultra-low power computing,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1051–1056.

[19] N. Neves, P. Tomás, and N. Roma, “Reconfigurable stream-based tensor unit with variable-precision posit arithmetic,” in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 149–156.

[20] H. Zhang, D. Chen, and S. Ko, “Efficient Multiple-Precision Floating-Point Fused Multiply-Add with Mixed-Precision Support,” *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 1035–1048, 2019.

[21] H. Zhang and S.-B. Ko, “Efficient multiple-precision posit multiplier,” in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[22] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, “Flexpoint: An adaptive numerical format for efficient training of deep neural networks,” *arXiv preprint arXiv:1711.02213*, 2017.

[23] E. Delaye, A. Sirasao, C. Dudha, and S. Das, “Deep learning challenges and solutions with xilinx fpgas,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 908–913.

[24] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, “Posits: The good, the bad and the ugly,” in *Proceedings of the Conference for Next Generation Arithmetic 2019*, ser. CoNGA’19. Association for Computing Machinery, 2019.

[25] L. Forget, Y. Uguen, and F. De Dinechin, “Hardware cost evaluation of the posit number system,” in *Compas’2019 - Conférence d’informatique en Parallélisme, Architecture et Système*, Anglet, France, Jun. 2019, pp. 1–7.

[26] Robert Munafo. Survey of Floating-Point Formats. Accessed on 9-July-2021. [Online]. Available: http://www.mrob.com/pub/math/floatformats.html

[27] Paresh Kharya. (2020) TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. Accessed on 9-July-2021. [Online]. Available: https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/

[28] J. L. Gustafson, *The End of Error: Unum Computing*. CRC Press, 2015.

[29] ——, “A radical approach to computation with real numbers,” *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, pp. 38–53, 2016.

[30] U. Kulisch, *Computer Arithmetic and Validity: Theory, Implementation, and Applications*, ser. De Gruyter Studies in Mathematics. De Gruyter, 2013, vol. 33.

[31] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann Publishers Inc., 2017.

[32] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2010.

[33] S. F. Oberman, S. F. Oberman, M. J. Flynn, and M. J. Flynn, “An Analysis Of Division Algorithms And Implementations,” *IEEE Transactions on Computers*, vol. 46, pp. 833–854, 1995.

[34] R. K. Montoye, E. Hokenek, and S. L. Runyon, “Design of the ibm risc system/6000 floating-point execution unit,” *IBM Journal of research and development*, vol. 34, no. 1, pp. 59–70, 1990.

[35] S. Wang and P. Kanwar, “Bfloat16: The secret to high performance on cloud tpus,” *Google Cloud Blog*, 2019.

[36] H. F. Langroudi, Z. Carmichael, J. L. Gustafson, and D. Kudithipudi, “Positnn framework: Tapered precision deep learning inference for the edge,” in *2019 IEEE Space Computing Conference (SCC)*. IEEE, 2019, pp. 53–59.

[37] H. F. Langroudi, V. Karia, J. L. Gustafson, and D. Kudithipudi, “Adaptive posit: Parameter aware numerical format for deep learning inference on the edge,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 726–727.

[38] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, “Evaluations on deep neural networks training using posit number system,” *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 174–187, 2020.

[39] R. Murillo, A. A. Del Barrio, and G. Botella, “Deep pensieve: A deep learning framework based on the posit number system,” *Digital Signal Processing*, vol. 102, p. 102762, 2020.

[40] M. Shirke, S. Chandrababu, and Y. Abhyankar, “Implementation of ieee 754 compliant single precision floating-point adder unit supporting denormal inputs on xilinx fpga,” in *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*. IEEE, 2017, pp. 408–412.

[41] B. Mathis and J. Stine, “A Novel Single/Double Precision Normalized IEEE 754 Floating-Point Adder/Subtracter,” in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 278–283.

[42] B. Mathis and J. E. Stine, “A Well-Equipped Implementation: Normal/Denormalized Half/Single/-Double Precision IEEE 754 Floating-Point Adder/Subtracter,” in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, 2019, pp. 227–234.

[43] Synopsys, *Floating-Point Adder*, https://www.synopsys.com/dw/ipdir.php?c=DW_fp_add.

[44] Xilinx. (2019) Logicore ip floating-point operator v7.1. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf

[45] W. José, A. R. Silva, H. Neto, and M. Véstias, “Efficient implementation of a single-precision floating-point arithmetic unit on fpga,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.

[46] G. Marcus. (2004) Floating point adder and multiplier (fpuvhdl). [Online]. Available: https://opencores.org/projects/fpuvhdl

[47] J. Al-Eryani. (2006) Fpu (fpu100). [Online]. Available: https://opencores.org/projects/fpu100

[48] M. K. Jaiswal and H. K. . So, “DSP48E efficient floating point multiplier architectures on FPGA,” in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, 2017, pp. 1–6.

[49] P. Malík, “High Throughput Floating-Point Dividers Implemented in FPGA,” in *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2015, pp. 291–294.

[50] L. Fiolhais and H. Neto, “An efficient exact fused dot product processor in fpga,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 327–3273.

[51] M. K. Jaiswal and H. K. . So, “PACoGen: A Hardware Posit Arithmetic Core Generator,” *IEEE Access*, vol. 7, pp. 74 586–74 601, 2019.

[52] R. Murillo, A. D. Barrio, and G. Botella, “Customized posit adders and multipliers using the flopoco core generator,” *2020 IEEE International Symposium on Circuits and Systems*, 2020.

[53] F. Xiao, F. Liang, B. Wu, J. Liang, S. Cheng, and G. Zhang, “Posit arithmetic hardware implementations with the minimum cost divider and squareroot,” *Electronics*, vol. 9, no. 10, p. 1622, 2020.

[54] M. Arunkumar, S. G. Bhairathi, and H. G. Hayatnagarkar, “Perc: Posit enhanced rocket chip,” in *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*, 2020.

[55] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti, “Peri: A configurable posit enabled risc-v core,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–26, 2021.

[56] UMC, “UMC’s 28nm high-k/metal gate stack HPCU,” https://www.umc.com/upload/media/05_Press_Center/3_Literatures/Process_Technology/28nm_Brochure.pdf, 2021.

[57] I. Yonemoto, “”sigmoid numbers”,” *[Online]. https://github.com/interplanetary-robot/SigmoidNumbers*, 2018.

[58] J. Hauser, “Berkeley testfloat.” *[Online]. Available: http://www.jhauser.us/arithmetic/TestFloat.html*, 2018.

[59] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document version 20191213,” 2019.

[60] A. Waterman and K. Asanovic, “RISC-V ”V” Vector Extension,” 2019.

[61] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[62] G. W. Bewick, "Fast multiplication: Algorithms and implementation," Ph.D. dissertation, Stanford University, 1994.

[63] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on electronic Computers*, no. 1, pp. 14–17, 1964.

A

Radix-4 Booth Multiplier

The radix-4 Booth recoding is a modification of the Booth algorithm [61], named after its creator, which was first proposed for speeding up signed radix-2 multiplication in early digital computers. The idea of a radix-4 Booth multiplier is instead of shifting and adding for every digit of the multiplier term and multiplying by 0 or 1, we only take every second digit, and multiply by 0, $\pm$1, or $\pm$2. This is, M $\times$ X is implemented as M $\times$ Y where Y is a Booth representation of X, such that y$_i \in \{-1, -2, 0, 1, +2\}$.

To recode the multiplier term, the bits are grouped in blocks of three, with overlapping between blocks. Grouping starts from the least significant bit (LSB), and since the first block does not have a previous block to overlap, it is extended with a ’0’. Each block is then decoded to select a single partial product as per Table A.1. In general, there will be $\lfloor \frac{n+2}{2} \rfloor$ partial products, where n is the operand length. Each partial product is shifted 2-bit positions with respect to its neighbors. These partial products are then added to form the result.

First, to implement the multiplications in each partial product as described in Table A.1, the multiplicand is extended with a ’0’ since numbers as large as 2 times the multiplicand must be dealt with. Depending on the block, the respective partial product is then selected. Multiply by 0 correspond to selecting a bit string of only 0s. To multiply by 1, the extended multiplicand signal is selected. To multiply

Multiplier Bits Block	Partial Product
000	$0\times$ M
001	$1\times$ M
010	$1\times$ M
011	$2\times$ M
100	$-2\times$ M
101	$-1\times$ M
110	$-1\times$ M
111	$0\times$ M

Table A.1: Radix-4 Booth recoding.

by 2, it is left-shifted. Negative multiplications, in 2's complement, can be obtained by complementing the signal, and, adding the sign bit in the least significant position of the partial product. This sign can be simply concatenated in the correct position of the following partial product to avoid additional additions. Since the partial products can be negative, they need to be sign-extended. However, this can be simplified by clearing the sign extension bits for positive partial products with the same technique of the literature [20, 62] (extending with the sign and 1s). This said the encoding module of each block outputs the partial product and the sign of the resulting partial product. Figure A.1 shows a diagram for a 8-bit radix-4 Booth multiplier (the empty bit positions correspond to 0s).

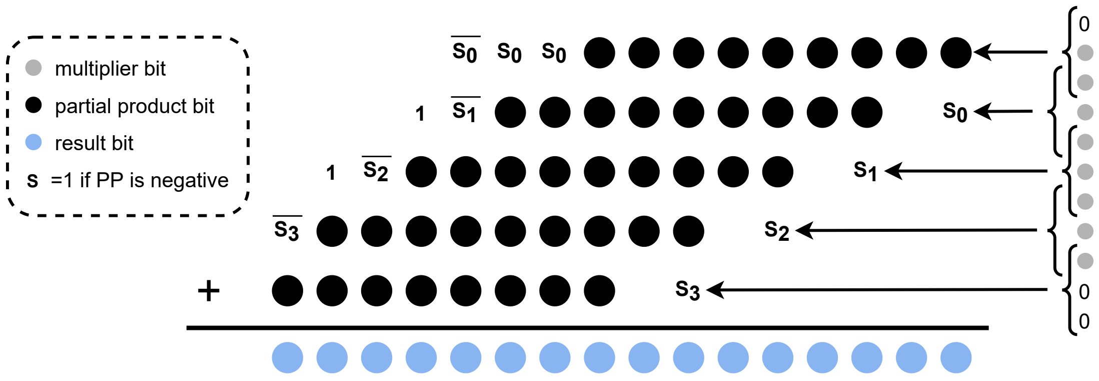


Figure A.1: 8 bit modified radix-4 Booth multiplier.

Propagate adders such as ripple-carry adders, carry-lookahead adders, or some other method are not convenient to use to add the partial products due to the high number of operands. In fact, one of the most important advances in improving the multiplication is the use of carry-save adders [63]. The carry-save adder reduces the addition of three numbers to the addition of two numbers. To obtain the result, a propagate adder is used at the end of the carry-save adder which adds the two final numbers (Wallace tree-like structure). In conclusion, a radix-4 Booth multiplier uses a radix-4 Booth encoder to generate the partial products and a Wallace tree-like structure to add them to the result.