TÉCNICO LISBOA

Parameterizable ALU for Deep Neural Networks

Luís Miguel Marques Crespo

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Doutor Nuno Filipe Valentim Roma
Doutor Pedro Filipe Zeferino Aidos Tomás

January 2021

Contents

Chapter 1

Introduction

The widespread format used to represent floating-point numbers is the IEEE-754 standard, released in 1985 and revised in 2019 [1]. However, an alternative representation has recently emerged: the posit format [2], released in 2017. This new representation features improved accuracy and larger dynamic range [2] when compared to the IEEE-754 standard. Since then, many application domains have been exploring the use of this numbering format as stated in [3] – Machine learning, graphics rendering, some Monte Carlo methods, integration-based methods, where the magnitude of the result can be framed. However, due to its run-time varying fields, an overhead of time and resources is imposed, in addition to the design challenge of developing such arithmetic architectures.

One of the hottest and compelling fields of computer science is Deep Learning (DL). Machine learning algorithms have proved powerful to multiple applications. However, it is a highly demanding computational tool. Many deep neural networks (DNNs) presently use 16-bit or 32-bit floating-point operations. However, motivated by the substantial time, energy and memory cost, low-precision arithmetic alternatives started being considered. First for inference and more recently, for training. In particular, low-precision posits started being used in DL inferring and training due to many of this numbering format interesting properties:

- Some posit configurations can do a fast approximation to the sigmoid function, which is an important function for neural network training.
- Numbers around 0 have more accuracy than extremely large or extremely small numbers (tapered precision), which is the same distribution that DNN weight parameters usually follow (more grouped around 0).
- Defines a Kulisch-like large accumulator [4] (quire) designed to contain exact sums of products of posits, which is particularly useful for the frequent dot products in DL.

Furthermore, the greatest advantage of posits is the fact that they can be used on the training phase of DNNs with fewer bits. In fact, the work that motivated this thesis ([5]), a DNN framework[1], supporting both training and inference, was developed by exploiting low-precision posits with as few as 8-bits to

[1] Available at: `https://github.com/hpc-ulisboa/posit-neuralnet`

obtain similar results as 32-bit IEEE-754 floats. This impressive results not only suggest an energy save for this kind of costly applications, but also a performance boost.

On the downside, there are situations where posits are worse than floating-point, as it is also refered in [3]. One such case occurs with particle physics simulations and integration methods, where the result is unbounded a priori. Considering this and given the fact that posit was designed to be a drop-in replacement for the IEEE-754 standard, being both representations similar, the present work proposes designing a configurable ALU architecture that supports both representations.

With this in mind, the requirements of the prospectiveted ALU are detailed in the following section.

1.1 ALU Requirements

The unit shall be parameterizable and configured to use either IEEE-754 or posit format.
Configurable parameters:

- Format
- Precision
- Exponent (only for posit)

Supported operations:

- Addition/subtraction
- Multiplication
- Division
- Fused Multiply-accumulate

1.2 Objectives

The key goals of this project is to design a run-time parameterizable ALU architecture that supports both representations. The conceived dynamic precision ALU will be described in VHDL and will be thoroughly evaluated with a Field Programmable Gate Array (FPGA) device and, if possible, will be synthetesized to a Application Specific Integrated Circuit (ASIC) prototype. Convenient support for vectorization will be one architectural optimization to consider if time allows so.

Other objective is to derive convenient models of the conceived architecture, by performing a Power, Performance, and Area (PPA) analysis on the target technologies. This model will allow to gain a precise idea of how the implementation would be if it was realized.

Finally the ALU will be integrated in a in-order processor based on RISC-V instruction set architecture (ISA). The floating-point ALU will be replaced by the posit unit in order to evaluate DNNs applications without creating new instructions. This is, the specific native instructions of the floating-point will be directly used.

Chapter 2

State of the art revision

This chapter presents a brief overview of the IEEE-754 floating-point standard and the novel posit number system. For this work, the operations to develop are: addition/subtraction, multiplication, division and fused multiply-accumulate. Therefore, a revision of the operators' structure is presented followed by some hardware architectures that have been develop for both representations. A implementation comparison is also presented between the operators available for both representations. Finally, a brief summary where the use of posit arithmetic units is justified. These will serve as a knowledge base for the future development of this work operators.

2.1 Floating-point Numerical Formats

This work addresses both formats, therefore a formal review is presented.

2.1.1 IEEE 754 Floating-point

The IEEE Standard for Floating-Point Arithmetic was established in 1985 and it was recently updated in 2019 [1]. The standard supports radix of 2 and 10, although the second is not relevant in the scope of this work. The representation of binary floating-point (FP) data consist of three fields – sign (S), exponent (E) and trailing significand (T). In Figure 2.1 it is represented the structure of an n-bit float.

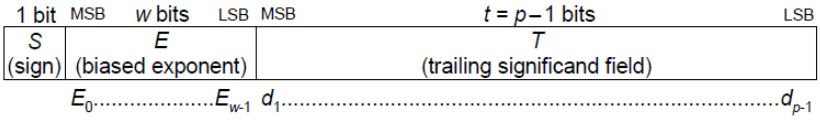

Figure 2.1: Binary floating-point format [1].

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: $E = e + bias$. The significand field has $(t = p - 1)$-bits and its leading bit, d_0, is implicitly encoded in the exponent. When representing a normal number

($1 \leq E \leq 2^{w-2}$), d_0 is set to 1. When a subnormal number ($E = 0$ and $T \neq 0$) is represented, d_0 is set to 0. The p parameter stands for precision and corresponds to the actual number of bits in the significand. The basic binary formats defined by IEEE-754 standard and its corresponding parameters are represented in Table 2.1.

Parameter	$FP32$	$FP64$	$FP128$
Whole representation (n)	32	64	128
Sign (S)	1	1	1
Exponent (w)	8	11	15
$bias(E-e)$	127	1023	16383
Precision (p)	24	53	113

Table 2.1: Number of bits used by the parameters defining basic format binary floating-point numbers.

The value (fp) of the represented floating-point datum and its exceptions are given by

$$fp = \begin{cases} (-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T), & \text{if normal number} (1 \leq E \leq 2^{w-2}) \\ (-1)^S \times 2^{E-bias} \times (0 + 2^{1-p} \times T), & \text{if subnormal number} (E = 0 \text{ and } T \neq 0) \\ (-1)^S \times (+0), & \text{if } E = 0 \text{ and } T = 0 \\ (-1)^S \times (+\infty), & \text{if } E = 2^{w-1} \text{ and } T = 0 \\ NaN, & \text{if } E = 2^{w-1} \text{ and } T \neq 0 \end{cases} . \quad (2.1)$$

To implement rounding operation, the 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.
- *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.

2.1.2 Posit

The posit format is part of the latest revision of the unum (universal number) arithmetic framework [2]. The original ”Type I” unum was proposed by Gustafson [6], in 2015, and it is a superset of the IEEE 754 Standard. Following the Type I, the “Type II” unum [7] 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 [2] – 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 [8] was made available in 2018.

A posit format is defined by the pair $< n, es >$, where n represents the word size and es the exponent size. In Figure 2.2 it is represented the structure of an n-bit posit, with es exponent bits.

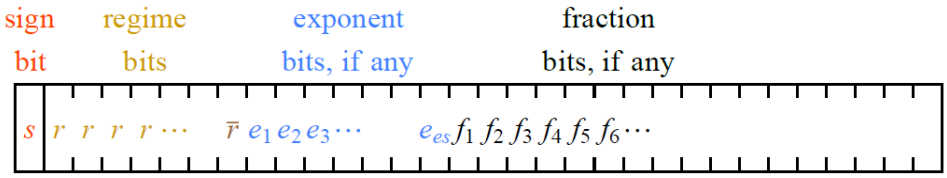

Figure 2.2: Posit format highlighting its components (sign, regime, exponent and fraction) [2].

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

The length of the other parameters varies depending on the magnitude of the represented number. The regime is the sequence of identical bits r, terminated by the opposite bit $\bar{r}$ (or by the end of the posit) and has numerical meaning k. Let m be the number of identical bits in the regime. Then, k is given by: $k = -m$, if the identical bits are 0s and: $k = m - 1$, if the identical bits are 1s. Basically, it works like a primitive way to record integers (with marks) but allows representing negative and positive values. The encoded value indicates a scale factor of magnitude $2^{k2^{es}}$ in the posit, where es denotes the number of bits of the exponent field.

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 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. Just like floats significand field, there is a hidden bit. However, there are no subnormal numbers, that is, the hidden bit is always 1 being the encoded value $1.f$.

Therefore, the encoded posit value p is given by:

$$p = (-1)^S \times 2^{k2^{es}} \times 2^e \times 1.f. \tag{2.2}$$

Similarly to the floating-point standard, posit also defines exception values. However, it only has one single representation for 0 (all 0 bits) and one $\pm\infty$ (1 followed by all 0 bits). It does not have any Not-a-Number (NaN) representation, thus all remaining bit patterns are used to represent actual numbers. Regarding fused operations, the posit format requires the utilization of a quire, which is basically a 2's complement fixed-point accumulator based on the Kulisch accumulator [4]. In Figure 2.3 it is represented the quire format.

The quire parameters bit width are the following:

- length: $n^2/2$ bits.

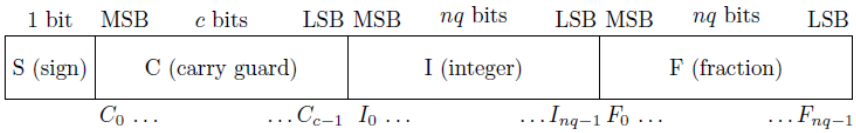

Figure 2.3: Binary quire format [8].

- fraction: $nq = ((1/4)n^2 - (1/2)n)$ bits.
- integer: nq bits.
- carry guard: $n - 1$ bits.

The value of the quire datum is given by the 2's complement signed integer represented by all bits, divided by 2^{nq} and has an exception (NaR) when S=1 and all other fields contain only 0 bits.

Despite the size and exponent parameters of a posit format being arbitrary, there are 4 standardized configurations and its parameters are represented in Table 2.2.

Parameter	$posit8$	$posit16$	$posit32$	$posit64$
whole representation (n)	8	16	32	64
max exponent es	0	1	2	3
quire ($n^2/2$)	32	128	512	2048

Table 2.2: Parameters of standardized posit formats.

For rounding, the standard defines only one rounding method, 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. The posit format does does not overflow to $\pm\infty$ nor underflow to 0, it is rounded to the maximum or minimum representable value, respectively.

2.2 Floating-point structures

A typical floating-point ALU structure is represented in Figure 2.4. In the decoding stage, exception cases are detected and the fields of the operands are extracted. In the particular case of IEEE-754 it corresponds to the: sign, exponent and significand; in the case of posit it includes the: sign, regime, exponent and significand.

While the IEEE-754 decoding is simple, since the fields location is know a priori, the posit decoding is more complex, since the location of the fields varies depending on the magnitude. In particular, the common posit decoding modules use a leading zero detector (LZD) to obtain the run-length of the regime value; a dynamic shifter to align the exponent and the fraction. Some encode modules attach the regime and exponent value to one single field, known as scale factor.

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 properly packed. Due to the variable-length encoding of posits, the position to which a value must be rounded is only known when performing this conversion.

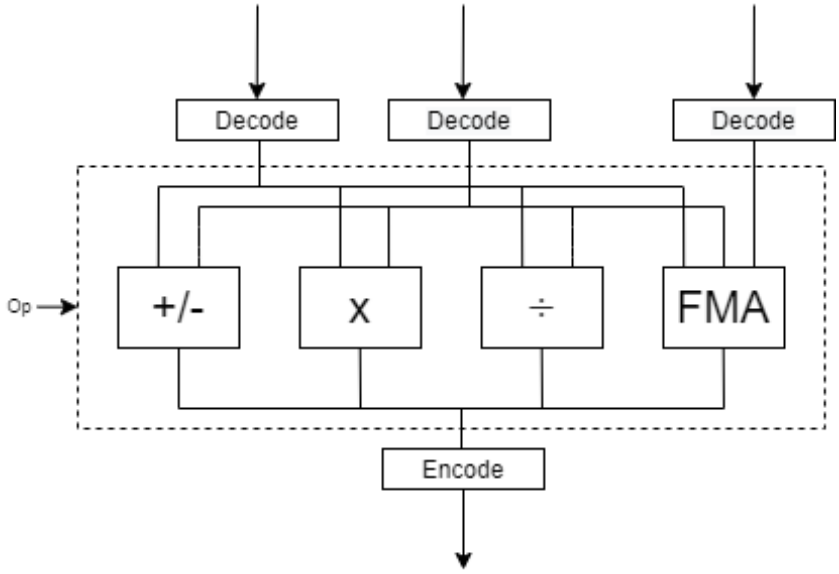

Figure 2.4: Floating-point ALU structure.

Before addressing each specific hardware unit implementation, a generic understanding of the operations' structure in both formats is essential. The addition/subtration, multiplication and division structures are similar for both the IEEE-754 and posit formats. Therefore, they will be explained using the classical structures. The fused multiply-accumulate is more format dependent and since the proposed architecture will adopt some posit ideas (as will be seen in Chapter 3) a posit architecture will be used to explain the operation.

2.2.1 Addition/subtraction

Floating-point addition/subtraction is regarded as a very complex operation involving several steps. The classical structure for floating-point addition/subtraction [9] is represented in Figure 2.5. Both operands are first unpacked and separated in its sign, exponent and fraction/significand fields. To sum the significands, it is necessary to first align them. To accomplish this, the exponents of the two operands are compared and the significand of the smaller is right shifted. Since pre-shifting is typically applied to only one of the operands, swap capability is also provided. Additionally, if the signs differ, the operand that is not pre-shifted is complemented. With the significands aligned, the addiction/subtraction (adjusted by the carry in) is conducted. This may produce a carry out, used in the result sign logic, and produces the fixed-point result (in the interval [0,4[). This result must be normalized to interval [1,2[with the proper exponent adjustments. If the result is negative, it has also to be complemented. To pack the result, it has to be properly rounded, which may imply another normalization and the corresponding exponent adjustment. After this final step, the sign, exponent and fraction are properly packed.

The posit arithmetic architectures are very similar to the floating-point operator, being the major difference the operand unpacking/decoding and packing/encoding. The significand addition is similar in both cases (considering that both are unsigned) and the sale factor is similar to the exponent.

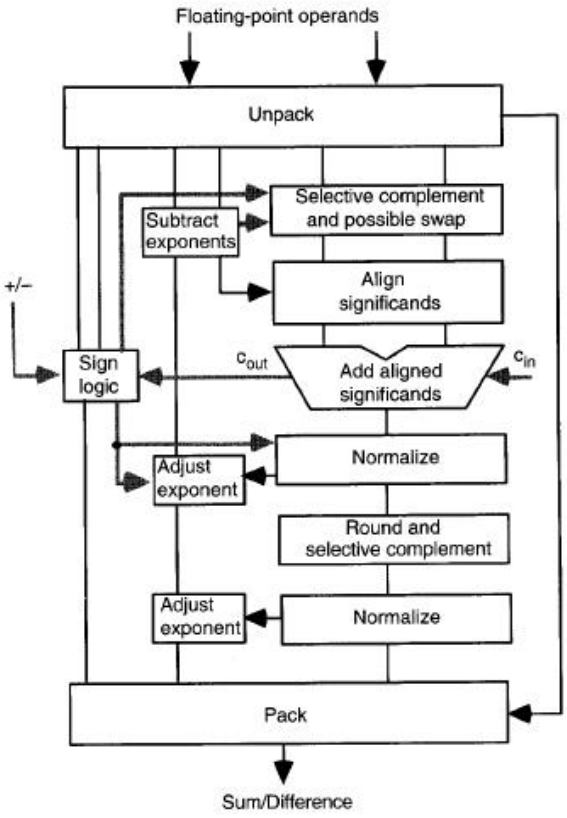

Figure 2.5: Floating-point addition/subtraction structure.

2.2.2 Multiplication

The floating-point multiplication operation is much simpler than the addition/subtraction operation. The classical structure for the floating-point multiplication [9] is represented in Figure 2.6. Both operands are first unpacked and separated in sign, exponent and fraction/significand. The significands are then multiplied using ordinary integer multiplication, because floating-point numbers are stored in sign magnitude form. Therefore, the multiplier deals only with unsigned numbers. At the same time, the exponents are added (with the proper bias subtraction) to calculate the provisory exponent. The sign is computed with a XOR operation. The product result can have up to twice the bits of the operands significand, varying in the range of [1, 4[. This must be normalized with the proper exponent adjustments, where the number of significand bits is adjusted in the rounding step. The rounding may imply another normalization. With this, the sign, exponent and fraction are properly packed.

Similar to the Adder/Subtractor, the posit Multiplier is very similar, being the major difference the operand unpacking/decoding and packing/encoding, that in posit are much more complex. The significand multiplication is similar in both cases (considering that both are unsigned). The sale factor is similar to the exponent.

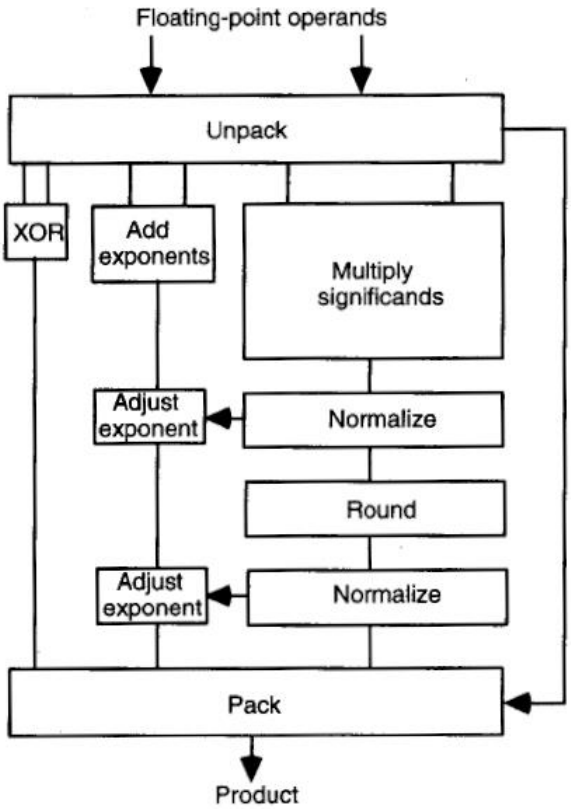

Figure 2.6: Floating-point multiplication structure.

2.2.3 Division

The division structure depicted (Figure 2.7) has a similar flow to Multiplication. The operands start being unpacked/decoded. Then, the sign bit is computed thought a XOR gate, the exponent of the dividend is subtracted by the divisor exponent and the fraction/significand is computed. The result is then normalized and the exponent properly adjusted. To pack the result, it has to be properly rounded, which may imply another normalization and the corresponding exponent adjustment. After this final step, the sign, exponent and fraction are properly packed. The most complex part of this operator is the significand division. In fact, four different classes of algorithms for significand division are described in [10] – digit recurrence, functional iteration, very high radix and variable latency.

Similar to the multiplier, the posit divider is very similar, being the major difference the operand unpacking/decoding and packing/encoding, that in posit are much more complex. The significand division is equal in both cases (considering that both are unsigned) and the scale factor (exponent and regime) is similar to the IEEE-754 exponent.

2.2.4 Fused Multiply-Accumulate

A fused multiply–add operator is a floating-point multiply–add operation performed in one single step, with a final rounding at the destination format. The posit fused multiply-accumulate architecture, represented in Figure 2.8, corresponds to the architecture proposed in [11]. The operands are first decoded in sign, scale factor (exponent and regime) and fraction. Two of the operands are multiplied, by following

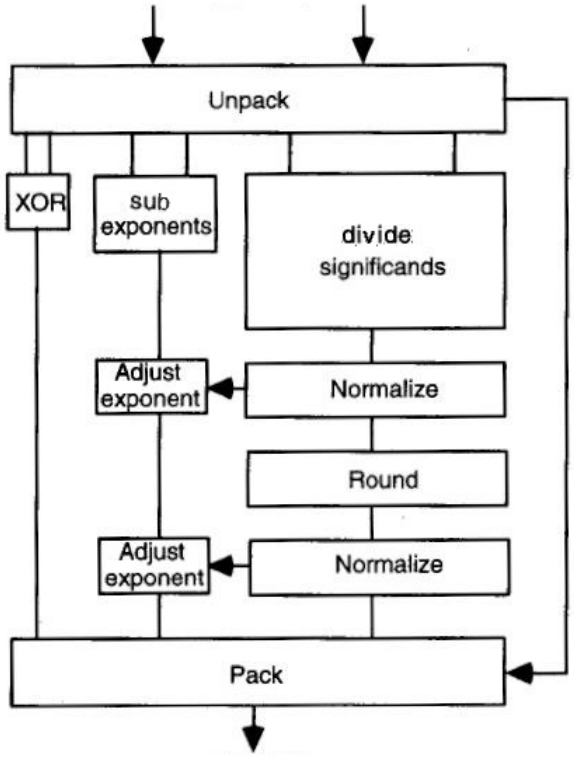

Figure 2.7: Floating-point division structure.

the typical floating-point core arithmetic: sign bit XOR gate, exponent adder, and significand multiplier. The multiplier is followed by an overflow protection circuit and the third operand is propagated. Typical floating-point fraction alignment logic for addition/subtraction is adopted in the quire stage and the accumulation is conducted (or the third operand is added to the quire (fixed-point)). The result is then normalized and then encoded with proper rounding.

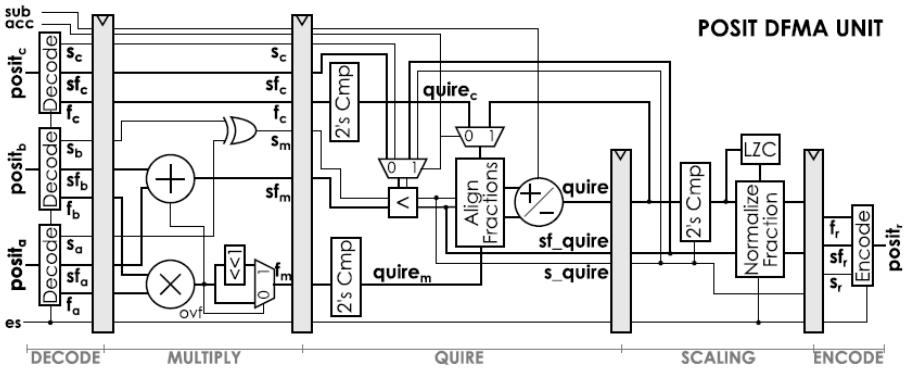

Figure 2.8: Fused Multiply-Accumulate architecture from [11].

The IEEE-754 analogue arithmetic structure some of the structures of this posit operator, being the major difference the operand unpacking/decoding and packing/encoding, that in IEEE-754 are much simpler. The significand addition and multiplication, are similar in both cases (considering that both are

unsigned) and the exponent is similar to the scale factor. Since accumulation is not supported in IEEE-754, the fraction alignment, is typically done in parallel with the multiplication. The other major difference is the quire register. To notice that this architecture supports dynamic exponent adjustment which imply additional logic (shifters and overflow logic), in the case of IEEE-754, this is not used since the exponent field is well defined and immutable.

2.3 Related floating-point hardware units

Most common hardware units provide support for the elementary and a fused operator: addition, subtraction, multiplication, division and fused multiply-accumulate. Both standards include the first four. However, in what regards to fused operators, the IEEE-754 standard [1], supports fused multiply-add that oblige a rounding step at every multiply-add operation, which contrasts with the posit standard [8], which has a special structure (quire) for fused operations that specialize on continuous accumulation with an insignificant rounding error.

To the best of the author knowledge, there are no available hardware supporting both representations. However, there is a wide range of studies regarding floating-point operators. The novel posit format, naturally, has fewer hardware studies. Hardware works for IEEE-754 include [12–22] and posit solutions include [11, 23–28] as represented in Table 2.3.

Format	Reference	Operators	Compliant	Device	Open source
IEEE-754	[12]	Add	Yes	FPGA	No
	[13]	Add	No	ASIC	No
	[14]	Add	Yes	ASIC	No
	[15]	Add/Mult/Div/FMA	No	FPGA	No
	[16]	Add/Mult/FMA	n.a.	FPGA	No
	[17]	Add/Mult	Yes	FPGA	Yes
	[18]	Add/Mult/Div	Yes	FPGA	Yes
	[19]	Mult	No	FPGA	No
	[20]	Mult	n.a.	FPGA	No
	[21]	Div	No	FPGA	No
	[22]	FMA	Yes	ASIC	No
posit	[23]	Add/Mult/Div	Yes	FPGA/ASIC	Yes
	[24]	Add/Mult	Yes	FPGA/ASIC	No
	[25]	Add/Mult	Yes	ASIC	No
	[26]	Add/Mult/Div	Yes	FPGA	No
	[27]	Add/Mult/FMA	Yes	FPGA	Yes
	[28]	FMA	Yes	ASIC	No
	[11]	FMA	Yes	FPGA/ASIC	No

Table 2.3: Arithmetic unit architectures for IEEE-754 floating-point and posit.

2.3.1 IEEE-754 Arithmetic units

Regarding IEEE-754, it can be observed that most references are not fully compliant with the standard. In fact, in most cases, hardware solutions do not support subnormal numbers due to added complexity and delay. It can also be observed that most references are not open source and the evaluation

technology is based on FPGAs in most of these studies.

Reference [12] only supports single precision. Although it is compliant with the standard, it only implements the standard floating-point adder algorithm [9]. On the contrary, papers [13, 14] propose an alternative to the standard designs of floating-point adders regarding their architecture parallelization and the optimization of the adder. These papers are from the same authors and the implementations are fully compliant with the standard except the fact that in [13] subnormal numbers are not supported. They use a parallel adder-subtractor structure, having the results of the addition and subtraction operations computed simultaneously. This way, the correct result is utilized once the sign is known, avoiding complementing the result after the operation. For datapath evaluation, all inputs are converted from a lower precision to the largest precision. Additionally to the subnormal difference, [13] only supports single and double precision, while [14] supports half, single and double precision.

The Xilinx floating-point IP core [15] supports half, single, double and even custom precisions but not at the same time, as the previous studies. It also has more operations than the listed ones and because it does not support subnormal numbers as well, as all the rounding modes, the implementation is not fully compliant with the standard.

Reference [16] implements a single precision floating point arithmetic unit with support for multiplication, addition and fused multiply-add. It also supports other operations but are not relevant for this work. The architecture consists of a fused pipelined multiplier-adder unit using DSPs that supports all elementary operations. Hence, the addition is performed by multiplying one of the operands by 1 and adding the result; the multiplication is performed by adding the result to 0. To achieve an increased precision, the whole 48 bits from the multiplier are used in the adder.

References [17] and [18] correspond to open-source code available online and described in VHDL. Both implementations follow the standard floating-point adder and multiplication algorithms [9] and both implementations are compliant with the standard. [18] also implements a divider using a Digit Recurrence algorithm [10]. While [18] presents an architecture that implements all the operators, reference [17] has all operators separated.

Jaiswal and So [19] propose a single, double, double-extended and quadruple precision floating-point multiplier architectures that do not support subnormal numbers. The study targets the utilization of DSP48E IP blocks in FPGAs, with the objective of a better hardware utilization. This type of architecture is focused on FPGA and, therefore, it is only useful if an architecture targeting the same FPGA technology is needed.

In [20], it is used the standard floating point multiplication algorithm and three types of single precision floating point multiplier implementations are compared: Array, Wallace and Vedic. This is specially relevant because the algorithm for floating-point multiplication is simple and the critical section is the significand multiplication, whose acceleration is crucial. This information might be useful for future architecture improvements.

Reference [21] implements three single precision floating-point multiplicative inverse and dividers architectures based on three different algorithms. The used algorithms are the Newton-Raphson division, Goldschmidt division and one of its variants – Goldschmidt division with binomial simplification. These

three fast computation algorithms provide a good performance, lower latency but at a cost of hardware resources. To note that the implemented Newton-Raphson architecture only supports multiplicative inverse. To implement the divider, it would be needed an additional multiplication by the multiplicative inverse result.

Paper [22] propose a half, single, double and quadruple precision FMA architecture compliant with the IEEE-754 standard. The multiple-precision FMA has native support for vectorization, by implementing one quadruple-precision operation, or two parallel double-precision operations, or four single-precision operations, or eight parallel half-precision operations. In addition, the architecture supports a mixed-precision operation, providing a 2-term dot-product accumulating to a higher precision. Additionally, the FMA unit can also be used to perform basic multiplication or addition operations.

2.3.2 Posit Arithmetic units

The posit numbering system has arisen the interest of many researchers in the community. As a consequence there are already a considerable amount of studies for hardware implementations, being the most relevant and recent ones listed in Table 2.3. Studies [23–27] implement the fundamental arithmetic operators (adder/subtractor and multiplier) while [23, 26] go a little bit further, implementing a divider. Regarding the quire, [11] is the most relevant one because the implementation supports a dynamic exponent field. All the listed studies are compliant with the posit standard [8] and only [23] and [27] are open source. However, access will be granted to study [11] by its authors.

Jaiswal and So [23] propose an open source ([29]) parameterizable posit arithmetic hardware generator. These authors have two previous studies regarding posit hardware [30, 31] being [23] their most recent posit implementation that replaces the leading one detector (LOD) and leading zero detector (LZD) by only a single LZD. The units work mostly like floating-point units and the divider unit uses the Newton–Raphson method [10]. It should be noted that their implementation does not support posits with format $< n, 0 >$, this is, $es = 0$.

Just like [23], Chaurasiya et al. [24] like [23] propose a parameterizable posit arithmetic hardware generator that only uses one LZD. However, it is not open source and only supports addition, subtraction and multiplication.

Reference [25] proposes two parameterized algorithms to perform addition and multiplication, by using the FloPoCo framework to generate the HDL code with an automatic pipelining of the operators.

Study [26] propose a different pair of decoding and encoding modules, that keep the two's complement format of the posit, as apposite to the previous works that convert the posit numbers to sign and magnitude. Additionally, it proposes a divider that uses an digit recurrence method. They present metrics for the addition and multiplier units whose implementations were not specified in the paper.

Reference [27] propose an open-source template based on a C++ library compliant with Vivado HLS that is capable of generating parameterized posit operators and their associated quire. Due to the variable length of the posit field, most posit related work convert the posit to a more hardware-friendly version. In this case, they formally define this intermediate format which is similar to IEEE-754 (having

even a biased exponent), differing on the fact that the significand is stored in two's complement.

Reference [28] propose the first hardware implementation of a posit fused multiply-accumulate operator, introducing a MAC unit for deep learning applications. Their architecture is parameterized and has a combinatorial implementation and a 5-stage pipeline design.

Neves et al. [11] proposed a dynamic fused multiply-accumulate unit, where the es field can be set at run-time with minimal resources and delay overhead over previous FMA implementations ([27, 28]). This unit not only supports the ordinary addition, subtraction and multiplication arithmetic operations, but it also implements used multiply-add and multiply-accumulate operations.

2.4 Performance and hardware efficiency comparison

To distinguish the best hardware units available a comparison of their implementations is essential. This comparison will be done between IEEE-754 and posit implementations. A comparison between both standards implementations will also be presented.

2.4.1 Adder/Subtractor unit

Floating-point addition/subtraction is regarded as a very complex operation involving several steps. Hence, it is costly in terms of hardware and timing. Due to this fact, several algorithms have been developed. As an example [32] studies three different algorithms and concludes that the standard one is area efficient but has larger delays in its pipeline stages and presents a larger overall latency when compared to the other algorithms. Most of the IEEE-754 listed above implement the standard algorithm for addition. Even the posit work uses the same flow of the standard algorithm.

This section starts with a comparison of FPGA implementations by considering IEEE-754 references. Then, a similar comparison will be presented by considering posit FPGA implementations. Since there are implementations that only target ASICs, a comparison in this technology is also presented for both representations. Finally, a comparison between IEEE-754 and posit will be also presented.

A – IEEE 754 comparison

FPGA technologies

All considered references [12, 17, 18] adopt a pipeline structure for single-precision architectures. Consequently, they will be compared with a state of art Xilinx core [15] (with single-precision) for the same FPGA (Table 2.4). The results of the comparison between [12] and the Xilinx adder ([15]) were presented in [12] being the considered metrics for [15] from a previous version. Both implementations have similar resources utilization but present a significant difference in terms of the maximum operating frequency. This is explained by the logic required for handling subnormal numbers, which increases the critical path in [12].

Regarding [17], it can be observed a significant difference in terms of the resource utilization and maximum operating frequency, when compared against the Xilinx core adder. Implementation from [18]

has one more pipeline stage but, nevertheless, it is a slightly better implementation over [17], using less resources and a better overall delay. It uses more registers due to its additional pipeline stage. In what concerns the maximum operating frequency, [18] is faster than [17] and much closer to the Xilinx core. It is worth noticing that this implementation, as said (see section 2.3.1), implements all its operators in the same unit, having some resource overhead for this extra functionality.

Reference	FPGA	Latency(cycle)	LUTs	Registers	Frequency (MHz)
[15] V7.0	Virtex-5	8	432	558	420
[12]	Virtex-5	8	441	433	263
[15]	Zynq 7000 SoC	6	394	387	185
[17]	Zynq 7000 SoC	6	503	290	83
[18]	Zynq 7000 SoC	7	448	360	126.6

Table 2.4: Comparison of IEEE-754 floating-point FPGA adder implementations versus the Xilinx design.

ASIC technologies

As it said in section 2.3.1, the papers [13] and [14] are from the same authors and implement similar architectures, differing only on the supported precisions and on the support for subnormal numbers. Both papers compare their implementation with the state of the art Synopsys' DesignWare (DW) floating-point adder/subtractor [33] being the results represented in Table 2.5. Implementation [13] occupies more area when compared to DW design, since it supports single and double precision. It also includes the necessary hardware for the internal conversions, while DW design does not include it since it only supports double precision. It also presents a lower propagation delay and consumes less power due to the design optimizations that minimizes logic through a separate adder and subtractor. In addition to single and double precision, reference [14] also supports half precision, which added to the support for subnormal numbers justifies the difference in area utilization when compared with the other implementations. Despite this, it has a propagation delay similar to the DW reference and consumes less power. Architectures [13] and [14] are basically equal, differing only in the support for subnormal numbers and one precision support. However, they present very different results, mainly justified by the subnormal numbering support.

Reference	Technology [nm]	Area [μm^2]	Delay [ps]	Total Power [mW]
[33]	32	6,652.6	616.21	12.4313
[13]	32	7,499.1	481.44	6.7290
[14]	32	9,671.9	634.81	7.7570

Table 2.5: Comparison of IEEE-754 floating-point ASIC adder implementations.

B – Posit comparison

FPGA technologies

Table 2.6 presents an analysis of the combinatorial architectures for the four listed studies – [23], [24], [26] and [27], targeting the Zynq 7000 Soc FPGA. All architectures offer parameterizable structurs since posit can have any desired configuration. Architecture [26] was only evaluated for the configurations $<$

$16, 1 >$ and $< 32, 1 >$. Therefore, the resource utilization and maximum operation frequency comparison also include $< 32, 1 >$ for some of the studies.

A comparison between [23] and [24] is highlighted in [23]. The authors argue that although their implementation uses more resources, it has a higher maximum frequency. As a consequence, in what concerns a comparison based on the product of area (LUT) $\times$ period (ns), it is concluded that [23] outperforms [24] in various posit configurations. However, I did not obtain the same results for the Zynq 7000 SoC FPGA. In fact, the metrics presented in Table 2.6 for [23] are those that were obtained by a synthesis of the available HDL ([29]) and it can be seen that [23] implementation offers more frequency in exchange of resources, but this is merely a trade-off. From the results presented in Table 2.6, [27] not only uses less resources than other implementations but is also has the higher frequency. The other implementations have similar metrics with a slight advantage to [26].

Reference	Configuration	LUTs	Frequency (MHz)
[23]	$< 16, 1 >$	486	40,3
	$< 32, 1 >$	969	30.8
	$< 32, 2 >$	1,103	31,7
[24]	$< 16, 1 >$	391	30.9
	$< 32, 1 >$	934	26.3
	$< 32, 2 >$	981	25.0
[26]	$< 16, 1 >$	383	36.7
	$< 32, 1 >$	939	27.9
[27]	$< 16, 1 >$	320	47.6
	$< 32, 2 >$	745	41.7

Table 2.6: Comparison of posit FPGA adder implementations.

ASIC technologies

Papers [23], [24] and [25] also present ASIC implementations. However, all their implementations are prototyped using different technologies and due to the fact that [23] and [24] were already compared in their FPGA implementations, only a comparison of [23] and [25] is presented. This is possible since authors of [25] implemented the open source code of [23] in the technology they use based on a 65 nm standard cell library. The comparison was done using a non-pipeline architectures, since [23] does not have pipeline architectures for every posit configuration. The collected metrics are presented in Table 2.7.

Reference	Configuration	Area [μm^2]	Delay [ns]	Total Power [μW]
[23]	$< 16, 1 >$	3,228.48	5.34	1,637.6
	$< 32, 2 >$	7,615.08	7.94	3,828.3
[25]	$< 16, 1 >$	2,176.92	6.23	1,133.1
	$< 32, 2 >$	4,880.88	9.48	2,811.1

Table 2.7: Comparison of posit ASIC adder implementations in 65 nm.

The design proposed in [25] reduces area and power consumption relative to the proposed in [23]. However, it has a higher delay. Nevertheless, it is a slightly more optimized architecture, since it has a better trade-off between area and delay.

IEEE-754 vs posit

This comparison is inevitable in all posit related work since this representation was designed to substitute floating-point. In fact, references [23], [24] and [27] actually do this comparison. Comparison metrics for [15], [17], [18] and [23] pipeline architectures using the Zynq 7000 SoC FPGA are represented in Table 2.8. The corresponding metrics for [27] target Kintex 7 FPGA. The floating-point references use single precision, while posit references correspond to the $< 32, 2 >$ configuration, in order to compare both representations for similar dynamic ranges. The first half of the table correspond to floating-point implementations and the second half correspond to posit implementations. ASIC implementations are not considered because different technologies were used. Reference [24] is not considered since a pipeline implementation of their architecture was not evaluated.

Reference	Standard	FPGA	Latency(cycle)	LUTs	Registers	Freq. (MHz)
[15]	IEEE-754	Zynq 7000 SoC	5	388	252	107.5
[17]	IEEE-754	Zynq 7000 SoC	6	503	290	83
[18]	IEEE-754	Zynq 7000 SoC	7	448	360	126.6
[23]	posit $< 32, 2 >$	Zynq 7000 SoC	5	884	254	113.6
[27]	posit $< 32, 2 >$	Kintex 7	22	738	811	376

Table 2.8: Comparison of IEEE-754 floating-point adder implementations versus posit adder implementations in FPGA.

Regarding the maximum operation frequency, [23] and [15] are very similar. However, in what concerns hardware resources, [23] uses more than double the resources. Comparing [23] with the IEEE-754 compliant architectures [17] and [18], its maximum operating frequency is superior. However, the resource utilization has a significant difference. Reference [27] has a deeper pipelined architecture and their implementation targets a different FPGA. Nonetheless, a fair comparison is possible in terms of resources, since both FPGAs have 6-input LUTs. This said, [27], similarly to [23], uses considerably more resources than the floating-point counterparts.

In sum, for similar configurations, posit implementations compete with state of the art IEEE-745 implementations in terms of frequency. However, they have a considerable resource overhead.

2.4.2 Multiplier unit

As it was referred in section 2.2.2 the implementation of the floating-point of the floating-point multiplication is simpler than the implementation of the addition/subtraction operation. However, the computation of the significand is still costly in terms of hardware and timing. As a consequence, all IEEE-754 above listed work implements the standard algorithm for multiplication, and even the posit work uses the same flow of the standard algorithm.

By following the same approach that was adopted in section 2.4.1 a comparison between IEEE-754 floating-point implementations will be presented, followed by a comparison of posit implementations. The first will only consider FPGA technologies, since the listed work only addresses such devices. The second will compare both FPGA and ASIC implementations. Finally, the inevitable comparison between IEEE-754 and posit implementations will be discussed.

A – IEEE 754 comparison

References [15, 17, 18] adopt pipelined single-precision architectures and present implementation metrics targeting Zynq 7000 SoC FPGA, considering no usage of DSPs. These results are presented in the first part of Table 2.9. On the second part of Table 2.9 are represented metrics for [15, 17, 19] using 1 DSP element. Metrics for [19] correspond to a implementation targeting a Virtex-5 FPGA. On the third part of Table 2.9 are represented the metrics for [15] and [16] using 2 DSPs. Reference [20] won't be considered since it is a study of significand multiplication methods.

Regarding the implementations without DSPs, [15] and [17] have practically the same amount of LUTs utilization but with a significant maximum operating frequency difference. This is explained by the logic required to handle subnormal numbers that increases the critical path. Concerning [18], a resource utilization difference can be observed against [15] and [17] with a maximum operating frequency difference very close to [15]. However, [18] has a much higher latency due the used significand multiplication method.

Reference	FPGA	Latency(cycle)	DSPs	LUTs	Registers	Frequency (MHz)
[15]	Zynq 7000 SoC	4	0	750	384	140
[17]	Zynq 7000 SoC	4	0	754	170	76.9
[18]	Zynq 7000 SoC	12	0	610	445	126.6
[15]	Zynq 7000 SoC	4	1	263	234	161.3
[17]	Zynq 7000 SoC	4	1	328	153	120.5
[19]	Virtex-5	5	1	392	195	331
[15]	Virtex-7	8	2	91	166	457
[16]	Virtex-7	8	2	183	249	525

Table 2.9: Comparison of IEEE-754 floating-point FPGA Multiplier implementations.

Regarding implementations with 1 DSP, [17] uses more resources than [15] but with a smaller maximum operating frequency difference. Concerning [19], a fair comparison is only possible in what concerns the resources since both FPGAs have 6-input LUTs. However, the same sis not possible in terms of frequency. This said, [19] uses more LUTs than [15] and [17]. Nevertheless, the architectures proposed by [19] are more relevant for higher precision arithmetic, since it uses less DSPs comparing to other state of art architectures.

Comparing implementations [16] and [15] for the Virtex-7 on the third part of the Table, implementation [16] has a superior maximum operating frequency; however it uses more resources.

B – Posit comparison

FPGA technologies

An analysis of non-pipeline architectures is presented for [23], [24], [26] and [27] targeting the Zynq 7000 SoC FPGA in Table 2.10. All architectures are parameterizable since posit can have any desired configuration and in [26] only metrics for the configurations $< 16, 1 >$ and $< 32, 1 >$ are presented, therefore, the resource utilization and maximum operation frequency comparison also include $< 32, 1 >$ for some of the studies.

Reference	Configuration	DSP	LUTs	Frequency (MHz)
[23]	$< 16, 1 >$	1	245	54.0
	$< 32, 1 >$	4	643	37.0
	$< 32, 2 >$	4	616	36.1
[24]	$< 16, 1 >$	1	218	41.6
	$< 32, 1 >$	4	576	32.2
	$< 32, 2 >$	4	572	30.3
[26]	$< 16, 1 >$	1	201	47.9
	$< 32, 1 >$	4	571	34.2
[27]	$< 16, 1 >$	1	253	55.6
	$< 32, 2 >$	4	469	37.0

Table 2.10: Comparison of posit FPGA multiplier implementations.

As said in the Posit FPGA comparison from 2.4.1, a comparison between [23] and [24] is highlighted in [23] but I synthesized [23] code ([29]) with the same FPGA (Zynq 7000 SoC) used in [24] and didn't obtain the same results. The results in Table 2.6 for [23] are the ones I obtained and since the other references implemented their architectures with DSPs, the same was done for [23]. Similar to the adder/subtractor, [23] implementation offers more frequency in exchange of resources compared to the other implementations. In study [26] they also compare their implementation with [24] presenting a minimal resource utilization difference but with slight improved maximum operating frequency. They also compare their implementation with [23] but with a virtex-7. Nonetheless, their implementation uses less resources but has slightly inferior maximum operating frequency for the considered posit configurations. Concluding, similar to the adder/subtractor, implementation from [27] is superior.

ASIC technologies

Due to the same reasons that were reported in the adder/subtractor comparison presented in section 2.4.1, only a restricted analysis is presented in Table 2.11 for [23] and [25] implementations in a 65 nm technology process.

Reference	Configuration	Area [μm^2]	Delay [ns]	Total Power [μW]
[23]	$< 16, 1 >$	4,955.76	5.15	3,036.6
	$< 32, 2 >$	15,106.32	8.54	13,027
[25]	$< 16, 1 >$	3,321.72	5.64	2,470.9
	$< 32, 2 >$	11,924.64	8.87	11,926

Table 2.11: Comparison of posit ASIC Multiplier implementations in 65 nm.

The same conclusions reported for the adder/subtractor analysis in section 2.4.1 applies here. Reference [25] reduces area and power consumption relative to the proposed in [23]. However, it has a higher delay. Similarly, [25] has a slightly more optimized architecture since it has a better trade-off of area against delay.

IEEE-754 vs posit

References [15] and [17] adopt pipelined IEEE-754 single-precision architectures, while [23] adopts a 32-bit posit pipeline architecture. However, [23] architecture uses 1 DSP therefore, the synthesis of the

Xilinx core multiplier [15] and the available HDL for [17] was conducted using also 1 DSP. The implementation metrics for the referred implementations targeting a Zynq 7000 SoC FPGA are presented in Table 2.12. Since the floating-point references use single precision, posit reference was implemented with $< 32, 2 >$ configuration, in order to compare both representations for similar dynamic ranges. Similarly to the adder/subtractor (see section 2.4.1), [27] has a deeper pipelined architecture for a different FPGA. Consequently, it was not considered in this comparison.

Reference	Standard	Latency(cycle)	LUTs	Registers	Frequency (MHz)
[15]	IEEE-754	4	263	234	161.3
[17]	IEEE-754	4	328	153	120.5
[23]	Posit $< 32, 2 >$	6	802	204	108.7

Table 2.12: Comparison of pipeline architectures for IEEE-754 floating-point Multiplier implementations versus a posit Multiplier implementation in FPGA.

Regarding the maximum operating frequency and resource utilization, [23] is clearly inferior to [15] and [17]. This said, for similar configurations, posit implementations is inferior to the IEEE-745 architectures in terms of frequency and resources.

2.4.3 Divider unit

Floating-point division is a much more complex operation when compared to its inverse operation (multiplication). The above listed work (see Table 2.3) implements, in both representations, the division operation using either a digit recurrence algorithm ([10]) or a functional iteration algorithm ([10]). 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 to converge (faster than linearly, typical quadratically) to the quotient or reciprocal.

Similarly to the above operators, a comparison of IEEE-754 implementations will be presented, followed by a comparison between posit implementations. Finally, a comparison between IEEE-754 and posit will also be presented.

A – IEEE 754 comparison

References [15, 18, 21] implement single-precision architectures and their resource utilization and maximum operating frequency is presented in Table 2.13. The results corresponding to references [15] and [18] were obtained for Zynq 7000 SoC, while the results for reference [21] were obtained by targeting a Virtex-5 FPGA. Since Zynq 7000 SoC have 6-input LUTs, while Virtex-5 has 4-input LUTs, a fair comparison is not possible. Nonetheless, the algorithm adopted in [15] algorithm is different from [18] being the latter a serial implementation of the significand division and clearly inferior to [15].

In [21], a Goldschmidt variation is also presented. However, it is inferior to the standard Goldschmidt. Their Newton-Raphson implementation only does the multiplicative inverse. Hence, a full division, would need an extra 24$\times$24 integer multiplier. Nevertheless, the Goldschmidt divsion unit is superior.

Reference	Algorithm	Latency(cycle)	LUTs	Registers	Frequency (MHz)
[15]	Digit Recurrence	12	823	539	92.6
[18]	Digit Recurrence	35	627	525	126.6
[21]	Newton-Raphson	15	2,374	291	66.858
	Goldschmidt	11	2,374	193	67.150

Table 2.13: Comparison of IEEE-754 floating-point FPGA Divider implementations.

B – Posit comparison

Reference [23] adopts a pipeline architecture for the posit division. However [26] does not. Therefore, the implementation metrics, represented in Table 2.14, only correspond to the combinatorial logic, which were obtained from [26] for a Virtex-7 FPGA. Both architectures are parameterizable, since posit can have any desired configuration. However, [26] only presents the results for the $< n, 1 >$ configurations. Therefore, the presented resource utilization and maximum operation frequency comparison uses the same configurations of the previous operators comparison ($< 16, 1 >$ and $< 32, 1 >$).

Reference [23] uses much more resources when compared to [26]. However, it has a much higher maximum operating frequency. In fact, architecture [26] implements a divider and square root based on the alternating addition and subtraction method (Digit recurrence method), which justifies the resource and frequency differences, since Jaiswal and So [23] architecture implements a divider based on the Newton-Raphson method (functional iteration method).

Reference	Configuration	LUTs	Frequency (MHz)
[23]	$< 16, 1 >$	1,069	40
	$< 32, 1 >$	4,050	21.7
[26]	$< 16, 1 >$	379	11.1
	$< 32, 1 >$	828	4.7

Table 2.14: Comparison of posit FPGA Divider implementations.

IEEE-754 vs posit

References [15] and [23] adopt pipelined architectures, and their implementation metrics, obtained for Zynq 7000 SoC, are presented in Table 2.15. Since the floating-point references use single precision, posit reference was implemented with $< 32, 2 >$ configuration, in order to compare both representations for similar dynamic ranges.

Reference	Standard	Latency(cycle)	DSPs	LUTs	Registers	Frequency (MHz)
[15]	IEEE-754	12	0	823	539	92.6
[23]	Posit $< 32, 2 >$	12	5	922	538	129.9

Table 2.15: Comparison of IEEE-754 floating-point Divider implementations versus a posit Divider implementation in FPGA.

Reference [23] has a higher maximum operating frequency. However, it uses much more resources than [15]. This occurs due to the fact that [23] implement a fast division method, that converges faster, contrarily to [15] that implements a slower method.

2.4.4 Fused Multiply-Accumulate unit

Both formats contemplate this kind of operation. However, with major differences. An IEEE-754 complaint FMA corresponds to a multiplier followed by a adder/subtractor, in which a single rounding is performed at the destination format and the format is maintained during the two operations. The posit FMA also corresponds to a multiplier followed by a adder/subtractor but it allows a continuous accumulation in a register (quire), where, the used representation is fixed-point. This allows the realization of a series of operations without any loss of accuracy, in which, only the final result is rounded.

Similarly to the above operators, a comparison of IEEE-754 implementations will be presented, followed by a comparison between posit implementations. Finally, a comparison between IEEE-754 and posit will be also presented. Reference [22] will only be considered in the posit comparison, since it is a much more complete unit that even supports vectorization.

A – IEEE 754 comparison

References [15] and [16] implement single-precision pipelined architectures. Table 2.16 presents the resource utilization and maximum operating frequency, by targeting a Virtex-7 FPGA.

Reference	Latency(cycle)	DSPs	LUTs	Registers	Frequency (MHz)
[15]	17	2	710	1,099	495
[16]	19	3	989	1,210	493

Table 2.16: Comparison of IEEE-754 floating-point FMA implementations.

Both architectures present similar results in terms of frequency. However, [16] uses more resources and has a higher latency. To notice that metrics from [16] might be outdated. In fact, the comparison that is conducted in [16] between their implementation and a previous version of [15] showed that [16] was slightly superior.

B – Posit comparison

The work listed in Table 2.3 for the posit FMA has some differences. In particular, [11] since it is more complete than other previously proposed architectures (such as [27] and [28]) implemented in FPGA and ASIC, respectively. It supports the definition of the exponent size at run-time, being capable of representing all the dynamic range for a given posit precision, as opposite to the other implementations that only support one es value set at synthesis time.

FPGA technologies

Firstly, a comparison between [11] and [27] is presented in Table 2.17 for a Virtex-7 FPGA regarding 8-, 16- and 32-bit configurations. For a -bit configuration, [27] uses less LUTs and has a higher maximum operating frequency, by using one less pipeline stage. However, it only supports one es configuration. When comparing higher precisions, [11] continues to have an inferior operating frequency. However, it uses less resources and has considerably fewer pipeline stages. In fact, Neves et al. [11] claim that if

the architecture was more deeply pipelined, the attained operating frequency would easily match the one from [27].

Reference	Config.	Max. es	Latency(cycle)	DSPs	LUTs	Registers	Frequency (MHz)
[11]	8-bit	5	5	0	667	205	200
	16-bit	13	5	1	1344	407	175
	32-bit	29	6	4	4134	1580	85
[27]	8-bit	0	4	0	400	n.a.	230
	16-bit	1	28	1	1,409	1,763	311
	32-bit	2	40	4	5,068	6,256	112

Table 2.17: Comparison of posit FPGA FMA implementations for 8-, 16-, 32-bit configurations.

ASIC technologies

Table 2.18 presents the same metrics for the ASIC implementations of [11] and [28], regarding 8-, 16- and 32-bit configurations (results obtained from [11]). Implementations from [11] and [28] target different technologies, ($45nm$ and $28nm$, respectively). Therefore, a fair comparison in terms of area is not possible. Nevertheless, [11] has an overhead of $0.2ns$ in every configuration for the same number of pipeline stages, which is explained by its dynamic support of es at run-time. To outperform [28], Neves et al. [11] add an additional pipeline stage that consequently increases the area and achieves the goal of attaining an inferior delay than [28].

Reference	Technology	Config.	Max. es	Latency(cycle)	Area [μm^2]	Delay [ns]
[11]	$45nm$	8-bit	5	5	9,601	1.2
		16-bit	13	5	32,649	1.5
		32-bit	29	5	95,861	1.8
[28]	$28nm$	8-bit	4	5	1,116	1.0
		16-bit	5	5	3,533	1.3
		32-bit	8	5	8,992	1.6

Table 2.18: Comparison of posit ASIC FMA implementations for 8-, 16-, 32-bit configurations.

IEEE-754 vs posit

As referred before, the standards differ regarding this fused operation. However, a comparison between both standards implementations is still relevant. Reference [11] will be the reference for posit units, both for FPGA and ASIC implementations. Regarding IEEE-754, [15] will be the reference for FPGA and [22] for ASIC.

FPGA technologies

Firstly, a comparison between [15] and [11] is presented in Table 2.19 for a Virtex-7 FPGA. For the same bit width, posit implementation uses more resources with a lower maximum operating frequency (with less pipeline stages). Nonetheless, posit quire makes the computation exact [2] for continuous accumulation. In fact, in [27], a sum of 1000 products were compared against a regular floating-point hardware and it was concluded that the overall latency is reduced by 10 times. This difference is related

to the standards approach to fused operations. To notice that lower bit width posit quire configurations easily achieve higher precision compared to IEEE-754. For example, 16-bit quire has 56 bits for the quire fraction field.

Reference	Format	Config.	Latency(cycle)	DSPs	LUTs	Registers	Frequency (MHz)
[15]	IEEE-754	32-bit	17	2	710	1,099	495
[11]	Posit	8-bit	5	0	667	205	200
		16-bit	5	1	1,344	407	175
		32-bit	6	4	4,134	1,580	85

Table 2.19: Comparison of IEEE-754 floating-point FMA implementations versus a posit FMA implementation in FPGA.

ASIC technologies

The metrics for ASIC implementations are presented in Table 2.20. Different technologies were used; therefore, a fair comparison of area and delay is not possible. Nevertheless, [22] has much more functionalities (support multiple operations at the same time, depending on the configuration) than [11] and presents half the latency. However, posit quire has a much higher accuracy.

Reference	Technology	Configuration	Latency (cycle)	Area [μm^2]	Delay [ns]
[22]	90 nm	Multiple	3	794,790	2
[11]	45 nm	32-bit	6	112,350	1.5

Table 2.20: Comparison of IEEE-754 floating-point FMA implementation versus posit FMA implementation in ASIC.

2.5 Summary

As observed in the previous section, for the same precision, the posit arithmetic units tend to use more hardware resources than the IEEE-754 units. However, posit architectures can compete in terms of performance (in some operators) with the IEEE-754 units. In general, posit arithmetic is more costly due to its variable size fields, which implies an overhead in terms of time and resources for the encode and decode steps. The quire is also extremely costly is terms of resources, being its use prohibitive with more than 32-bit due to its resulting size.

With this disadvantages, the reader might wonder, what really are the advantages of using such numerical representation that uses more resources and, for some operations, is slower. For the same precision, posits have more accuracy. However, this does not solve the problems in DNNs applications. The great advantage of posits is the fact that they can be used on the training phase of DNNs with fewer bits. The results from reference [5] suggest that 8-bit posits can replace 32-bit floats in a mixed low-precision posit configuration for the training phase, with no negative impact on the resulting accuracy. This impressive results not only suggest an energy save for this kind of costly applications, but also a performance boost.

Chapter 3

Proposed Architecture

This chapter presents a high-level description of the proposed AALU architecture. The envisioned architecture will adopt the structure represented in Figure 3.1. The great challenge of this work is to dynamically reconfigure the structure in order to satisfy, in run-time, the performance an/or energy requisites of the underlying application. Due to the several technologies characteristics, different approaches will be considered to dynamically support both formats and configurations.

The parameterization control structure will have some similarities in both technologies. In particular, the configuration must be stored in registers and logic to detect and trigger the architecture re-parameterization. Hence, this transition will be different depending on the technology.

Further work improvements will be considered, in particular, a vectorization of the units (similar to [22]).

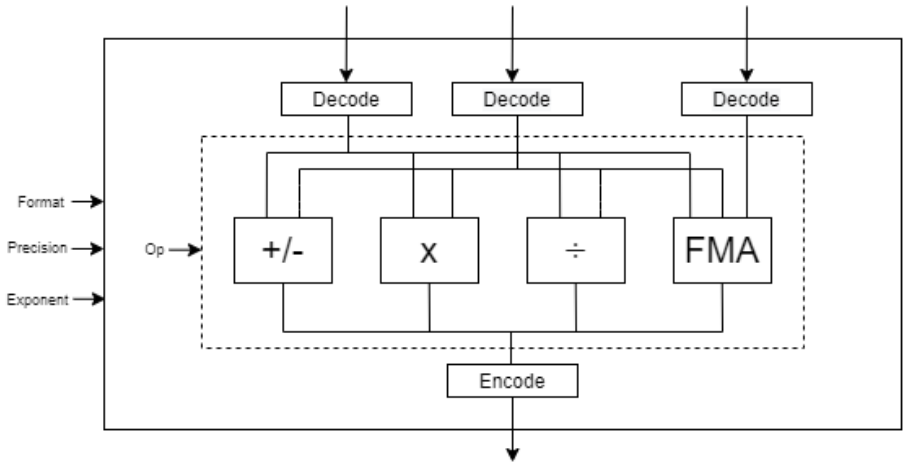

Figure 3.1: Proposed ALU structure.

3.1 Run-time ALU parameterization

In FPGA technology, it is possible to dynamically reconfigure the structure. With this in mind, the ALU shall be reconfigured in run-time depending on the format, precision and exponent of the aimed number-

ing system. While this reconfiguration implies a time overhead, it allows a efficient way of parameterizing the structure, since only the necessary hardware is synthesized.

For ASIC technologies, reconfiguration of the structure is not an option. In this kind of technology, after manufactured, its hardware structure cannot be changed. Due to this, the synthesised hardware must support all configurations of format, precision and exponent. The clock or power gating techniques shall be applied to turn off unused structure parts during the execution.

In Figure 3.2, it is represented a simple example of application of power gating for the addition operation using a Ripple Carry Adder. The full adder state is controlled by the *sleep* signal (red – turned-off, green – turned-on). This way, only the necessary bits are computed. To control the various *sleep* signals (in the case of power gating) the parameters stored in the configuration registers must be processed and used in the control logic.

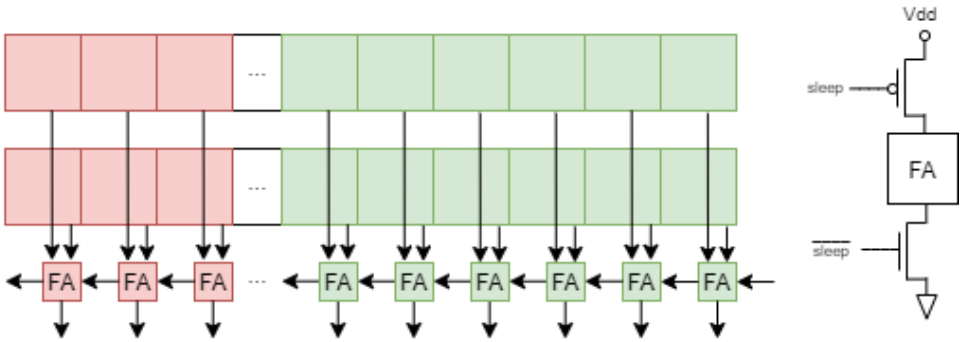

Figure 3.2: The computation of the red bits is turned off and the green bits are computed normally.

Another simple example of the application of clock gating is the case of the unit using posits. The logic to decode and encode of the IEEE-754 will not be used. By removing the clock signal for this stages, the power dissipation can be reduced.

3.2 ALU Architecture

The ALU structures will share the decode and encode across all operators, differing only in the core arithmetic part (see Figure 3.1). The supported operations will be: addition/subtraction, multiplication, division and fused multiply-accumulate. Previous available work will be the base for the proposed structures, namely, [23], [27] and [11] for posit. For IEEE-754, the structures presented in [17] and [18] will be considered.

Decoding and Encoding structures

In what concerns the decode and encode modules, there exists major differences between both formats (see section 2.2). The decode module of the IEEE-754 format is a simple field extraction, with some exception checking. In contrast, the decode module of the posit format involves some costly hardware (such as LZD and shifter), with some exception checking. Some state of the art implementations directly

decode the posit in sign, scale factor and fraction. However, there are some architecture differences to extract these fields, mainly in what concerns the scale factor. Regarding the encoding, both standards perform rounding and packing. For IEEE-754 this is particularly simple, since the position to round is known. However, the posit is different since the position to which the rounding is performed is only known at this step and depend on the regime and exponent value of the result (which are appended to the fraction in this step). This said, the decode and encode modules have major differences in the logic structure that the two formats do not share. Hence, the IEEE-754 encode and decode modules will be based in [17] and [18], while the posit will be based in [27] and [11]. In Figure 3.3 it is represented the high level structure to adopt in the decode and encode modules for the posit. To avoid reconfiguration in the case of different es values for the same precision, some logic used in [11] will be considered.

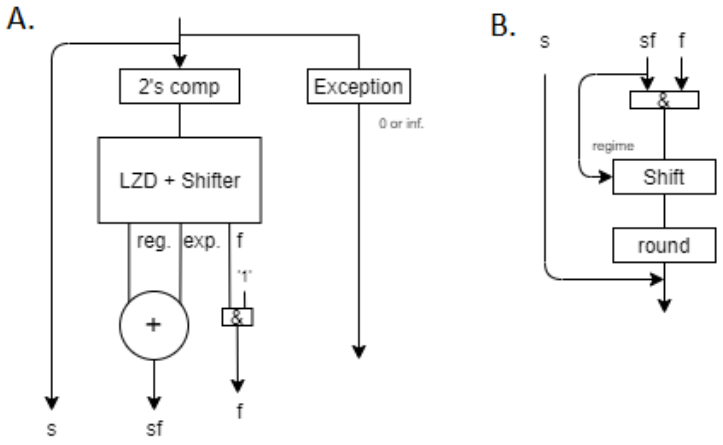

Figure 3.3: Posit (A) decoding and (B) encoding modules.

Arithmetic structures

Regarding the arithmetic core of the operators, the adder/subtractor and multiplier modules will be based on the standard algorithms that are used in most state of the art implementations. The Divider will be based on the Newton-Raphson division, since a fast divider implementation was introduced by [23]. The fused multiply-accumulate will be based on the architecture presented in [11].

The addition, multiplication and division operations have very similarities between the two formats, mainly in what concerns the processing of the exponent/scale factor and the significand/fraction. This will allow a shared core arithmetic very similar between the two formats, with a slight difference in the exponent length.

The fused multiply-accumulate of [11] will be used and adjusted to support IEEE-754 floating-point. This is possible because most of its core arithmetic is similar to IEEE-754 operations. The quire accumulation is not compatible to the IEEE-754 format. Therefore, the result will be directly propagated to the normalization and encoding stages. Despite this, a new feature will be added, corresponding to the possibility of performing quire accumulation, using the quire register. This feature is non-compliant but will increase the arithmetic accuracy and allow continuous accumulation reducing the overall latency.

Similar to the previous operators, the unpacking, rounding and packing will have some major differences in the logic parts that the two formats do not share.

3.3 ALU Vectorization

This architectural optimization is not yet defined, however, it will be based on the typical vectorization units (for example [22]). In this units, it can be typically performed 1 quadruple-precision, 2 double-precision, 4 single-precision, or 8 half-precision operations (Figure 3.4). For this work, the ideal minimum precision would be 8-bit length, since 8-bit posits can be used in DNN applications. As a matter of fact, the ideal minimum precision would be also parameterizable, however this might not be viable.

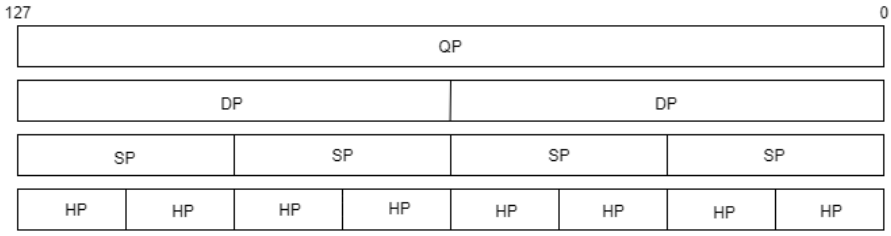

Figure 3.4: Unified operands format for different precisions.

Considering minimum precision of 8-bit length. The decode module would extract each field from the corresponding format and organized it in the corresponding vector. The sign vector (s) would have 16 bits. For 8-bit operation, each bit represents the sign of one sub-operand. For 16-bit operation, s[1], s[3], s[5], s[7], s[9], s[11], s[13] and s[15] are respectively used for each sub-operand while the other bits are set to zero. For other precision the similar process would be applied.

A total of sixteen regime and exponent (for posit) processing units would be necessary. Some would be shared and others only used in the 8-bit operands. For example, the first unit would only be dedicated to 8-bit operations (which implies a smaller unit) but the last would process up to 128-bit operands.

The fraction/significand would also be placed in a vector similar to the represented in Figure 3.5 (it corresponds to a floating-point implementation). In Figure 3.5 is also represented the example of a multiplication for half precision operands.

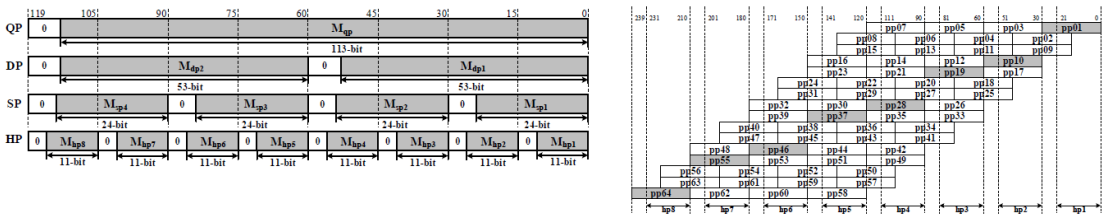

Figure 3.5: Unified significand format for different precisions and half-precision multiplication [22].

Chapter 4

Evaluation methodology

This chapter will briefly explain how the proposed architecture will be verified. The architecture will be evaluated by integrating it in a RISC-V processor and the impact of using the posit format will be evaluated against the IEEE-754 format by extracting a Power-Performance-Area model (PPA).

4.1 ALU correctness

The ALU will be developed in VHDL and tested using Vivado Design Suite. To verify the correctness of the arithmetic units, several random test cases will be used and compared against TestFloat[1] and SoftPosit[2] or SigmoidNumbers[3]. The first correspond to a set of programs (to run in the command-line) to test whether an implementation of IEEE-754 arithmetic conforms to the standard. Only two programs of this package are planned to be used. One to generate test cases for each operator and the other to verify if the results obtained are correct. SoftPosit and SigmoidNumbers correspond to software implementations of the posit number system endorsed and developed by the posit developers, respectively. These libraries will be used to test whether each operator implementation conforms to the standard.

4.2 RISC-V integration

To directly evaluate deep learning applications, the ALU will be integrated in a processor architecture based on the RISC-V ISA [34]. This is a prominent open source ISA with an ample support by the research community and natively comprises a floating-point extension. In particular, the floating-point unit from the processor (see Figure 4.1) will be substituted by the developed unit and the native floating-point extension for RISC-V will be used since the operations are analogous.

To support the reconfiguration of the unit, no solution is yet defined. Some options are: developing a new special purpose instruction (in which the ALU parameters are encoded); perform a memory access

[1] http://www.jhauser.us/arithmetic/TestFloat.html
[2] https://github.com/milankl/SoftPosit.jl
[3] https://github.com/interplanetary-robot/SigmoidNumbers

that maps to the reconfiguration unit (similar to Memory-mapped I/O, where the same address space address both memory and I/O devices, which in this case corresponds to the reconfiguration unit).

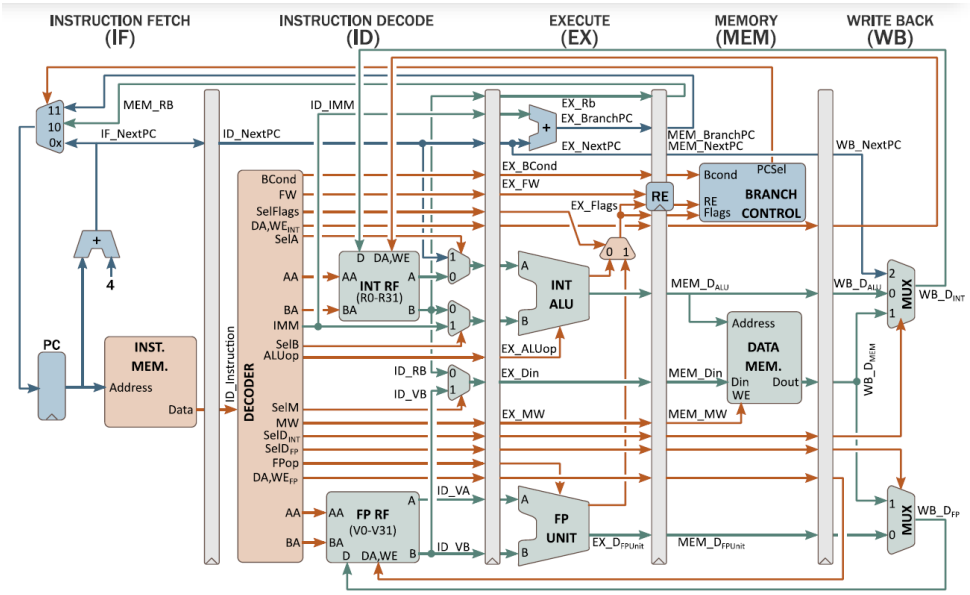

Figure 4.1: Simple in-order pipeline processor architecture.

Then, the deep learning framework develop in [5] will be used to analyze DNNs applications, for example, classical classification or/and regression problems. These applications will be compiled using the RISC-V compiler and directly evaluated in a FPGA.

4.3 PPA model

The ultimate objective of using low-precision posits in DNNs is to improve the energy efficiency. In this kind of applications, energy consumption is becoming a major problem. With this in mind, by deriving a PPA model, it will be possible to evaluate the impact of adopting posit architectures in other processor architectures. In particular, this model will be derived for technologies available in INESC-ID. The FPGA implementation will be performed with a Xilinx Virtex® UltraScale+™ VU37P HBM FPGA. An ASIC synthesis will also be performed for a 45nm technology, by considering the Nangate 45nm PDK.

The extracted PPA model will be based on the models used in McPAT simulator [35]. The McPAT simulator will be used together with the processor simulator gem5 [36] in order to obtain energy and performance estimates. This way, it will be possible to evaluate the impact on applications that use the posit format. Especially for [5], where the framework to use posits in DNNs is develop but convenient models for evaluation are not available.

Chapter 5

Conclusion and Thesis Planning

This Chapter presents a conclusion of the report and the prospective workplan until the conclusion of this thesis (see Figure 5.1). This plan comprehends the work that was already performed for this report (for IIEEC), which mainly corresponds to research and the present report.

In this report, a state of the art study was conducted for floating-point arithmetic implementations. This study includes the standard representation: IEEE-754; and a new representation: posit. It was concluded that posit arithmetic is more costly in terms of time and resources. However, this new representation is particularly useful in DNN applications, since low-precision posits with as few as 8 bits can be used to obtain similar results to 32-bit IEEE-754 floats. Therefore, this thesis proposes the development of a ALU unit that supports both representations with different configurations at run-time. The architecture to develop was explained and how it will be evaluated.

The development of the architecture is the core task and it is planned to take about 5 months. This task involves the implementation in FPGA and, if possible, in ASIC. The vectorization of the unit is also included. Since the ALU includes several operators and logic blocks, a comprehensive set of tests will be conducted during the development.

The integration of the new ALU in the processor and the PPA model extraction will be conducted in parallel, since the model only involves the acquisition of metrics. The integration is predicted to take about 1.5 months.

Research work will be conducted during the whole timeframe and the writing of the thesis will start as soon as architecture design is ready. The conclusion is expected to October 15th, 2021.

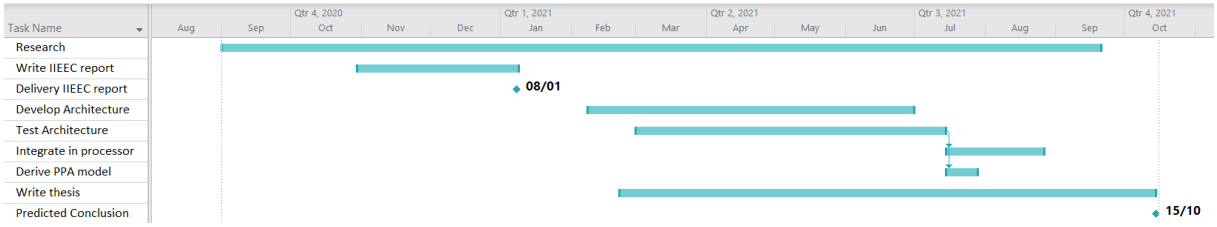

Figure 5.1: Gantt Chart

Bibliography

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

[2] J. L. Gustafson and I. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2), 2017. doi: 10.14529/jsfi170206.

[3] 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*, CoNGA'19. Association for Computing Machinery, 2019. doi: 10.1145/3316279.3316285.

[4] U. Kulisch. *Computer Arithmetic and Validity: Theory, Implementation, and Applications*, volume 33 of *De Gruyter Studies in Mathematics*. De Gruyter, 2013.

[5] G. Raposo. Deep learning with approximate computing: an energy efficient approach. Master's thesis, Instituto Superior Técnico, 2021.

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

[7] J. L. Gustafson. A radical approach to computation with real numbers. *Supercomputing Frontiers and Innovations*, 3(2):38–53, 2016. doi: 10.14529/jsfi160203.

[8] P. W. Group et al. Posit standard documentation. *Posit Standard Documentation*, 2018.

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

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

[11] 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)*, pages 1–6, 2020. doi: 10.1109/SiPS50750.2020.9195256.

[12] 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)*, pages 408–412, 2017. doi: 10.1109/ICPCSI.2017.8392326.

[13] 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)*, pages 278–283, 2019. doi: 10.1109/ISVLSI.2019.00058.

[14] 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)*, volume 2160-052X, pages 227–234, 2019. doi: 10.1109/ASAP.2019.00011.

[15] Xilinx. *LogiCORE IP Floating-Point Operator v7.1*, 2019. URL `https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf`.

[16] 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)*, pages 1–4, 2014. doi: 10.1109/FPL.2014.6927391.

[17] G. Marcus. fpuvhdl. URL `https://opencores.org/projects/fpuvhdl`.

[18] J. Al-Eryani. fpu100. URL `https://opencores.org/projects/fpu100`.

[19] 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)*, pages 1–6, 2017. doi: 10.1109/ICVD.2017.7913322.

[20] V. K. R, A. R. S, and N. D. R. A comparative study on the performance of FPGA implementations of high-speed single-precision binary floating-point multipliers. In *2019 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pages 1041–1045, 2019. doi: 10.1109/ICSSIT46314.2019.8987800.

[21] P. Malík. High Throughput Floating-Point Dividers Implemented in FPGA. In *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pages 291–294, 2015. doi: 10.1109/DDECS.2015.66.

[22] H. Zhang, D. Chen, and S. Ko. Efficient Multiple-Precision Floating-Point Fused Multiply-Add with Mixed-Precision Support. *IEEE Transactions on Computers*, 68(7):1035–1048, 2019. doi: 10.1109/TC.2019.2895031.

[23] M. K. Jaiswal and H. K. . So. PACoGen: A Hardware Posit Arithmetic Core Generator. *IEEE Access*, 7:74586–74601, 2019. doi: 10.1109/ACCESS.2019.2920936.

[24] 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)*, pages 334–341, 2018. doi: 10.1109/ICCD.2018.00057.

[25] 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.

[26] 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*, 9(10):1622, 2020. doi: 10.3390/electronics9101622.

[27] 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*, pages 1–7, Anglet, France, June 2019.

[28] 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)*, pages 1–5, 2019. doi: 10.1109/ISCAS.2019.8702349.

[29] M. K. Jaiswal. *PACoGen: Posit Arithmetic Core Generator*. `https://github.com/manish-kj/PACoGen`.

[30] M. K. Jaiswal and H. K. . So. Universal number posit arithmetic generator on FPGA. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1159–1162, 2018. doi: 10.23919/DATE.2018.8342187.

[31] M. K. Jaiswal and H. K. . So. Architecture Generator for Type-3 Unum Posit Adder/Subtractor. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018. doi: 10.1109/ISCAS.2018.8351142.

[32] A. Malik and S. Ko. A Study on the Floating-Point Adder in FPGAS. In *2006 Canadian Conference on Electrical and Computer Engineering*, pages 86–89, 2006. doi: 10.1109/CCECE.2006.277498.

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

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

[35] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009.

[36] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.