KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Low precision artihmetic using posit in sum product networks over FPGA and error bound analysis

Antoine GENNART

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Electronics and Integrated Circuits

Thesis supervisor:
Prof. Marian VERHELST

Assessor:
Prof. Luc DE RAEDT
Prof. Nele MENTENS

Mentor:
Ir. Nimish SHAH

Academic year 2019 – 2020

Preface

I would like to thank everybody who kept me busy the last year, especially my promoter and my daily advisor. I would also like to thank the jury for reading the text.

todo

Antoine Gennart

Contents

Preface **i**

Abstract **iii**

1 Introduction **1**

2 State of the art **3**

2.1 Sum product networks 3

2.2 Low precision arithmetic 5

2.3 Posit representation 6

3 Error bound **11**

3.1 Error bound for Float 11

3.2 Error bound for Posit 12

3.3 Error bound for SPNs 13

4 Hardware implementation **17**

4.1 Hardware operators 17

4.2 Sum product networks 18

4.3 Global system 20

5 Results and experiments **23**

5.1 Error bound 23

5.2 Hardware implementation 27

6 Conclusion **31**

Bibliography **35**

Abstract

todo

Chapter 1

Introduction

This work focus on the implementation of sum product networks (SPNs) in field programable gate array (FPGA) using low precision numbers. Two numbers representation are studied: Posit representation, also known as type III universal number (UNum) [17] and floating point representation [1]. Posit is a quite new standard (2017) and has a lot of advantages compared to floating point such has an increased range of numbers and an increased precision for a large range of values. However, there are some disadvantages to posit, such as the increased hardware size, that justify the fact that it is not a good idea to completely replace floating point by posit for general operations. In this work, we focus on the use of posit for specific use case of SPNs and try to find out in what cases posit would be preferable to floating point.

Chapter 2 begins with theory about SPN. SPNs are a new architecture [11] of deep networks for graphical probabilistic inference. This architecture is designed to make the partition function tractable. Tractable means that it is possible to retrieve the impact of each input feature of the network on the output. Then, low precision arithmetic, namely posit [17] and floating point [1] standard are described. Posit and floating point are two number representations as binary string for computer based arithmetic. While floating point is widely used on all electronic devices up to now, posit is a new standard which has a lot of advantages compared to floating point and may be more efficient in a lot of applications. Therefore, this work studies the use of posit in SPN.

In Chapter 3, a model is developed to compute an error bound for posit representation based on the error bound for floating point in [14]. Number representation used in this work does not conforms to standard of posit and floating point respectively defined in [17] and [1]. Each number representation uses a custom definition of the fields length to minimize the error bound. This error bound represents an upper bound for any set of inputs of the SPN. As in [14], relative errors are used since values inside a SPN may be very small and not be efficiently represented by standard number representation such as double floating point.

Then, the development of an hardware implementation of SPN using custom posit representation is done in Chapter 4. It uses a trained SPN as input and shows how to convert it up to hardware description language (HDL). This process is automatized, only some meta parameters should be set to generate a new SPN. An architecture integrating the SPN alongside with a control unit and memory is built. Thanks to this architecture, the SPN can be controlled.

Finally, in Chapter 5 , the results from both Chapter 3 and 4 are shown. First the error bound for a large set of SPN is analyzed. Then, the hardware implementation of a specific SPN is analyzed and demonstrate the operation of the full design.

Chapter 2

State of the art

This chapter gives an theoretical overview of different topics covered in this for low precision arithmetic over FPGA. At first, SPNs are described. Then, we explain how floaint point and posit representation works and how they are used in this work.

A SPN is a deep network architecture for probabilistic inference. This Chapter explains what is the general architecture of a SPN and what are the different properties that it can have and how it will be used in this work.

Low precision arithmetic regroup different binary number encoding sheme to represent floating point numbers with a low number of bits. Two representations are studied: Floating point and posit.

2.1 Sum product networks

Definition 1 *SPN In [11], a SPN is defined as a rooted acyclic graph with variables as leaves, sum and product as internal nodes, and weighted edges.*

SPNs are a new deep network architecture that allow to perform graphical probabilistic inference. This architecture is differentiated to others by the fact that the function it represents is tractable. That is: the output of the network can be analysed given the inputs.

As an example, two graphical representations from [11] are shown in Figure 2.1. The leaves of this network are the indicators X_i and $\bar{X_i}$. A SPN represent the probability $P(X_0, \bar{X_0}, ..., X_N, \bar{X_N})$. If the input of the SPN is an image, the indicators whould be the pixel values (1 for black and 0 for white).

In order to compute a specific set of probability with the evidence that $X_i = 0$, the input of the SPN X_i must be set to 0 and the indicator $\bar{X_i}$ must be set to 1. In order to compute a specific set of probability and marginalize over the variable X_i, the indicators X_i and $\bar{X_i}$ must both be set to 1. In any case, the values of the indicator X_i and $\bar{X_i}$ will never both be 0 at the same time.

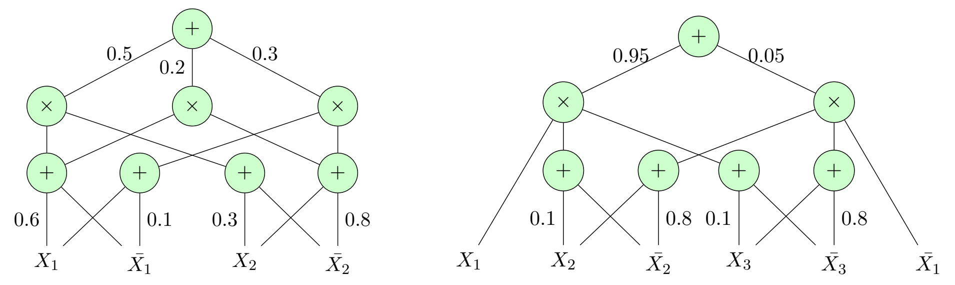


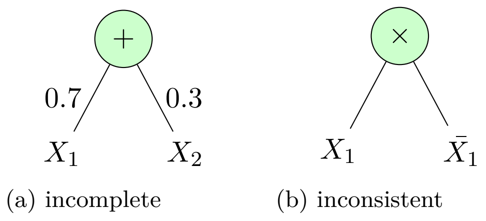
Figure 2.1: Examples of SPNs from [11]

2.1.1 Properties

Two properties are interesting for SPNs: it can be complete and consistent. If an SPN is complete and consistent, it represent a partition function and can be used to compute all the marginals of this partition function.

Definition 2 *Complete A SPN is complete iff all children of the same sum node have the same scope. A counter example is shown in Figure 2.2a.*

Definition 3 *Consistent A SPN is consistent iff no variable appears negated in one child of a product node and non-negated in another. A counter example is shown in Figure 2.2b.*



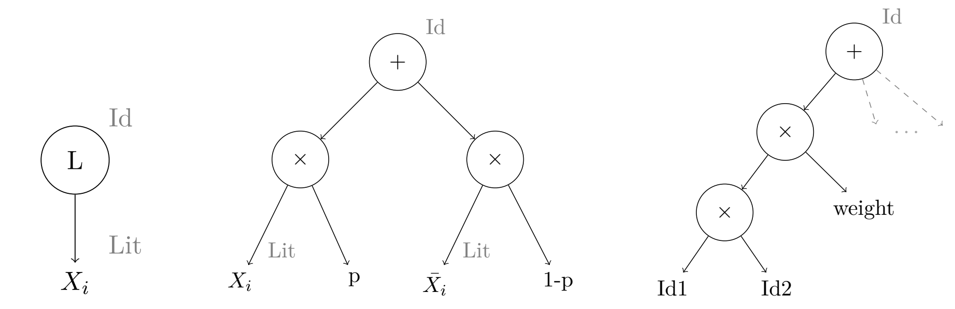
2.1.2 Applications

SPNs have a wide range of applications. A GitHub repository is dedicated to list and all ressources about SPNs and related work [2]. A wide range of machine learning problems can be solved with SPN such as regression or classification. As an example, there is [13] which classify images using SPNs for autonomous flight.

2.1.3 PSDDs

Probabilistic sequential decision diagrams (PSDDs) are a specific kind of SPN. It was first reported in [6]. PSDDs are composed of three kind of nodes described in Figure 2.2: Literal nodes (leafs), true nodes and decomposition nodes.

This work uses pre-trained PSDD networks to generate SPNs in HDL. The networks used in work are trained using a method described in [18].



(a) Literal node (b) True node (c) Decomposition node

Figure 2.2: Decomposition of PSDD file into simple arithmetic expression for SPN format.

2.2 Low precision arithmetic

As we target a FPGA to implement a SPN, size and speed are important concerns that sould be optimized. Therefore we use low precision arithmetic to represent numbers inside the FPGA with the smallest hardware as possible. As opposed to central processing unit (CPU) and application-specific itegrated circuit (ASIC), FPGAs have the advantage to be quickly reconfigurable which allow the use of custom number representation for any network implemented.

Two number representations are studied. Floating point and posit. Both these representations are built to encode floating point numbers in a binary string. Floating point is the most widely used number representations in every electronic devices and is defined in the standard IEEE std 754-2008 [1]. Posit is a new standard developped in 2017 by Gustafson [7] which allow to represent a wider range of numbers than floating point, and to perform a higher precision than floating point for a wide range of numbers.

2.2.1 Floating point

In this work, we use floating point as a custom number representation. That means that it is not compatible with IEEE std 754-2008 and use custom field lenght which

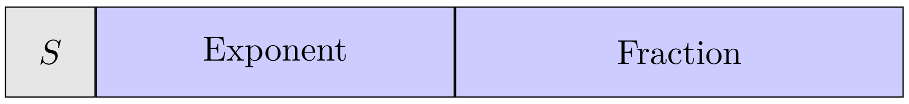


Figure 2.3: Floating point format with sign bit. There are three main field: A sign bit (S), a fixed size exponent, and a fixed size fraction.

are optimized for a given application. In addition to this, the sign bit is not used since SPNs are used to make proabilistic inference which are composed exclusively of positive numbers.

2.2.2 Floating point bit-fields

The general representation of the fields of the floating point representation are shown in Figure 2.3. It is composed of three fields: A sign bit (which is removed in this work), a certain number of exponent bits, and a certain number of fraction bits. The lengths of each of these fields must be fixed for a given application and can be optimized to increase the range or to increase the precision of numbers to be reperesented. Increasing the number of exponent bits increases the range, but it also reduces the number of fraction bits. A trade-off should be make to maximize the efficiency.

In order to compute the value represented by a floating point number, the Equation 2.1 must be used where exp represent the value of the exponent field and b_i represent the i^{th} bits of the fraction. The value "1" before the sum corresponds to the implicit one of the fraction which is not explicitly encoded because it is redundant.

$$x = \xcancel{(-1)^s} \cdot 2^{\mathrm{exp}} \cdot \left(1 + \sum_{i=1}^{N} \mathbf{b}_{N-i} \cdot 2^{-i}\right) \tag{2.1}$$

2.2.3 Floating point operations

Addition and multiplication operations are also defined for floating point. They require simple operation such as comparison, shifting and integer addition and multiplication. Addition and multiplication are described in Algorithm 1 and 2 respectively.

2.3 Posit representation

Posit is also know as the type III UNum [4]. The main difference with floating point representation is that the exponent which is applied to the fraction is encoded using a field of variable lenght. Thanks to this properties, posit benefits of an increased range of representable numbers and an increased accuracy for a large range of numbers.

Algorithm 1: Floating addition

```
Result: r = a + b
Compare exponent;
if $a.exp > b.exp$ then
    r.exp = a.exp;
    f1 = a.frac;
    f2 = b.frac » (a.exp - b.exp);
else
    r.exp = b.exp;
    f1 = a.frac » (b.exp - a.exp);
    f2 = b.frac;
end
r.frac = f1 + f2;
if $overflow(r)$ then
    r.frac » 1;
    r.exp = r.exp + 1;
end
Return : r;
```

Algorithm 2: Floating multiplication

```
Result: r = a · b
r.exp = a.exp + b.exp;
r.frac = a.frac · b.frac;
Return : r;
```

2.3.1 Posit fields

Posit format possesses four fields as shown in Figure 2.4. As for floating point, this work does not use the sign bit. A general posit number has four fields: A sign bit (again, not used in this work), a regime of variable size, an exponent of fixed size and a fraction. The fraction fill in the remaining bits that are not used for the regime and the exponent.

The regime field makes the specificity of the posit representation. It is a variable size field using unary encoding. That is: the value of this field can be found by computing the number of consecutive identical binary units. It ends with one bit which break the sequence. Therefore, the minimum length for this field is two: either “b10” or “b01”. The maximum length for this field is equal to the number of bits of the posit representation minus one (for the sign bit). A sequence starting with a 1 represent a positive number, and a sequence starting with a 0 represent a negative number. A regime of zero is represented by “b10”.

The exponent field has a fixed maximum size. That means that if there are not enough bits left to encode the exponent field due to the regime size, every unknown bit of this field has a value of “b0”. For example, if an encoding scheme use 8 bits

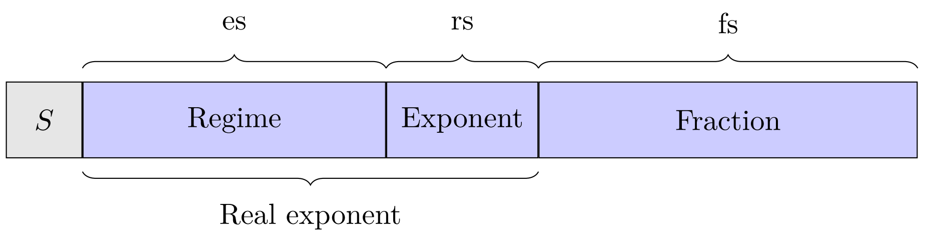


Figure 2.4: POSIT format with sign bit. There are four main fields: A sign bit (S), a regime (k), an exponent (exp) and a fraction (frac).

to represent a number and 1 bit of exponent. A regime of -7 would be encoded by "b00000001" and would use all the 8 bits available. Therefore, the exponent bits is not encoded and the fraction is also not encoded. The exponent value would be 0 and the fraction value would be 0 as well .

The fraction field has the same meaning as the fraction field of the floating point representation. There is also a implicit leading one in this field. The term significand used to described the fraction to which we add the leading one. In the previous example, the significand had then a value of 1.

Given these four fields, one can compute the value using Equation 2.2. In Figure 2.4, it is mentioned that the combination of the regime and the exponent field form the real exponent. Indeed, we must use both these fields to compute the power of two that must be applied to the significand to compute the value of the encoded number as it can be understood with Equation 2.2.

$$x = \xcancel{(-1)^s} \cdot 2^{2 \cdot es \cdot \mathbf{k}} \cdot 2^{\mathbf{exp}} \cdot \left(1 + \sum_{i=1}^{N-1} \mathbf{b}_{N-i} \cdot 2^{-i}\right) \tag{2.2}$$

2.3.2 Posit operations

As for floating point, basic operators are defined for posit. However, as posit has a more complicated structure, it should be decoded before performing any operations. This decoding should extract the value of the real exponent (using the regime and exponent fields) and the significand. It is a complex operation since fields can have variable size and start at unknown location in the bit number. The pseudo code for the encode and decoder are described in Algorithm 4 and 3 respectively.

Once decoded, posit numbers can be represented as an exponent and a significand. Therefore, a floating point addition or multiplication can be performed using Algorithm 1 and 2. However, since posit representation has a wider range than floating point, the floating point operations performed after the decoding would represent the numbers in an extended form.

Both the fact that any operation require an encoder and a decoder, and that the number are represented using an extended form inside an operator impact the size and speed of any hardware implementations as shown in [12] and [10].

Algorithm 3: Posit decoder

```
Result: exp, frac ← P
regime = leading_zero_counter(P);
exp = (P « abs(regime))[0:es];
real_exp = exp + (regime « es);
frac = {1, P[es+1:]};
return real_exp, frac;
```

Algorithm 4: Posit encoder

```
Result: P ← exponent, significand
regime, exp ← exponent;
P ← encode_unary(regime);
P ← encode_binary(exp);
P ← encode_binary(significand);
return P;
```

2.3.3 Comparison of floating point numbers and posit representation

Since floating point notation was discovered first, it has been widely used in every electronic devices and is now the norm for number encoding in almost every computers. Posit was discovered in 2017 [17]. Even with the increased range, and its increased accuracy for a wide range of numbers, posit and has not enough advantages to replace completely floating point. There are two disadvantages compared to floating point. First, its increased accuracy is only true for a range of numbers and there exists cases where floating point is better than posit. Second, it is less hardware efficient and requires more power for a single computation.

Figure 2.5 shows the decimal accuracy of floating point and posit representations as a function of the encoded value. As we can see, posit has a better decimal accuracy for number close to 0. These numbers are the number for which the regime is small and that the combined size of the regime and the exponent of the posit representation is smaller than the exponent size of floating point representation. Then floating point representation has a better for a large range of big numbers. This graphics also show the extented range of posit in comparison with floating point.

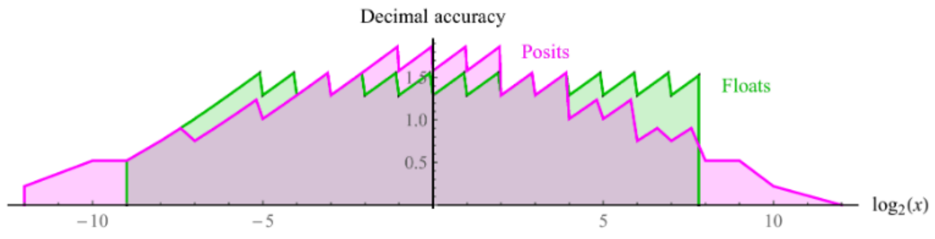


Figure 2.5: Image from [7] showing the decimal accuracy of floating point and posit representatoin as a function of the encoded value

Chapter 3

Error bound

In this chapter we described the models of error bound for both posit and floating point representation. An error bound is an estimation of the worst case scenario. That means that there are no errors acutally reported in real cases that are bigger than the computed error bound. All errors are computed in absolute value.

3.1 Error bound for Float

As float is a widely used format, there already exists error bounds for floating point representation in the context of SPNs [14]. It uses a relative error bound instead of an absolute error because absolute error for floating point can be very small and cannot be represented by a standard floating point number while a relative error can be represented using floatng point numbers.

In [14], relative error is defined as follow: Given a number f, the encoded using a floating point representation is $\hat{f}$ with an absolute error of Δf and a relative error of ϵ as described in Equation 3.1.

$$\tilde{f} = f \pm \Delta f = f \cdot \left(1 \pm \frac{\Delta f}{f}\right) = f \cdot (1 \pm \epsilon) \tag{3.1}$$

For SPNs applications, errors can occurs in three different cases: While encoding a new number for any litterals or any results of an addition or multiplication, while performing an addition and while performing a multiplication. Note that there is a difference between the error made while performing and encoding an operation.

3.1.1 Encoding error

The relative error for a floating point number can be computed based on the fraction size (fs) as it is done is Equation 3.2. As the size of floating point fields are constant, this error is a constant for any value encoded. The only exception we take into account is zero which can be encoded with a relative error equals to zero.

$$|\epsilon| = \left|\frac{\Delta f}{f}\right| \leq 2^{-(fs+1)} = 2^{-(N-es+1)} \tag{3.2}$$

3.1.2 Addition error

Given ϵ, the error of encoding a floating point number, a number a with an accumulated relative error of ϵ_a and a number b with an accumulated relative error ϵ_b. A new error bound for the result of the addition of a and b can be computed through Equation 3.3.

$$\begin{aligned}
\hat{f} &= (\hat{a} + \hat{b}) \cdot (1 \pm \epsilon) \\
&= (a \cdot (1 \pm \epsilon_a) + b \cdot (1 \pm \epsilon_b)) \cdot (1 \pm \epsilon) \\
&= (a + b + a\epsilon_a + b\epsilon_b) \cdot (1 \pm \epsilon) \\
&= (a + b) \cdot \left(1 \pm \frac{a}{a+b} \cdot \epsilon_a \pm \frac{b}{a+b} \cdot \epsilon_b\right) \cdot (1 \pm \epsilon) \\
&\leq (a + b) \cdot (1 \pm max(\epsilon_a, \epsilon_b)) \cdot (1 \pm \epsilon) \\
(1 + \epsilon_f) &= (1 \pm max(\epsilon_a, \epsilon_b)) \cdot (1 \pm \epsilon)
\end{aligned} \tag{3.3}$$

3.1.3 Multiplication error

Given ϵ, the error of encoding a floating point number, a number a with an accumulated relative error of ϵ_a and a number b with an accumulated relative error ϵ_b. A new error bound for the result of the addition of a and b can be computed through Equation 3.4.

$$\begin{aligned}
\hat{f} &= \hat{a} \cdot \hat{b} \cdot (1 \pm \epsilon) \\
&= a \cdot (1 \pm \epsilon_a) \cdot b \cdot (1 \pm \epsilon_b) \\
&= a \cdot b \cdot (1 \pm \epsilon_a) \cdot (1 \pm \epsilon_b) \cdot (1 \pm \epsilon) \\
(1 \pm \epsilon_f) &= (1 \pm \epsilon_a) \cdot (1 \pm \epsilon_b) \cdot (1 \pm \epsilon)
\end{aligned} \tag{3.4}$$

3.2 Error bound for Posit

In this section, it is considered that a posit number has N bits (without sign bit), rs is the regime size (variable), es is the number of exponent bits (fixed) and ms is the number of mantissa bits.

As shown in Equation 3.5, the relative error for a posit representation can be expressed using the same method as for floating point (Eq. 3.2). Again, it is more interesting to compute the relative error since posit representation has a wider range than floating point representation. Therefore an absolute error would not be encodable as a floating point numbers.

$$\hat{p} = p \pm \Delta p = p \cdot \left(1 \pm \frac{\Delta p}{p}\right) = p \cdot (1 \pm \epsilon) \tag{3.5}$$

Note that there also exists some reasearch about absolute error for posit representation as it is done in [7] or [9]. These require to use a model with a high precision for very small errors.

3.2.1 Encoding error

In the same way as it is done for floating point in Equation 3.2, the encoding error can be computed on the basis of the fraction size fs. For posit representation, the fraction size is not constant and depends on the value of the encoded number.

Given a number which is in the bound of the posit representation, the maximum relative error that could occur is expressed by Equation 3.6. This error represent the inability to encode one more fraction bit. The fraction size is represented by fs, rs represents the regime size and es the exponent size. N is the number of bits of the posit representation.

$$|\epsilon| = \left|\frac{\Delta p}{p}\right| \leq 2^{-(fs+1)} = 2^{-(N-min(rs+es,N)+1)} \tag{3.6}$$

In contrast to floating point, this error highly depends on the number to be represented. It would be minimal for numbers around 1, and it would grows for high numbers and very small numbers (closer to 0) as it is shown in Figure 3.1. The only exception is zero, which can be encoded with a relative error equals to zero.

3.2.2 Addition and multiplication error

given that a relative error has the same definition for posit representation (Eq. 3.5) than floating point representation (Eq. 3.2), addition and multiplication error can be computed using the same formulas than those described for floating point in Equations 3.3 and 3.4.

The difference with floating point is that the actual value of the relative error ϵ is different given the number that is encoded. A summary of floating point and posit error bound model is shown in Figure 3.1. This shows that posit representation has a wider range that floating point numbers and that the error bound for posit may vary as a function of the numbers to be represented.

3.3 Error bound for SPNs

3.3.1 Floating point

In order to compute the error bound for a SPN, the error must be computed for each node and be propagated up to the output. The error bound that is computed must be the worst case scenario. Therefore, there should not be any zero values since that is the only exception that can be encoded with a zero error bound.

By setting all the literals to one, and propagating the error up to the output of the SPN, we can then compute the error bound of floating point representation for a given network.

In addition to this, an additional computation must be done to ensure that floating point can be used to represent every numbers in the SPN (even the smallest). Therefore the minimum output of the network must also be computed.

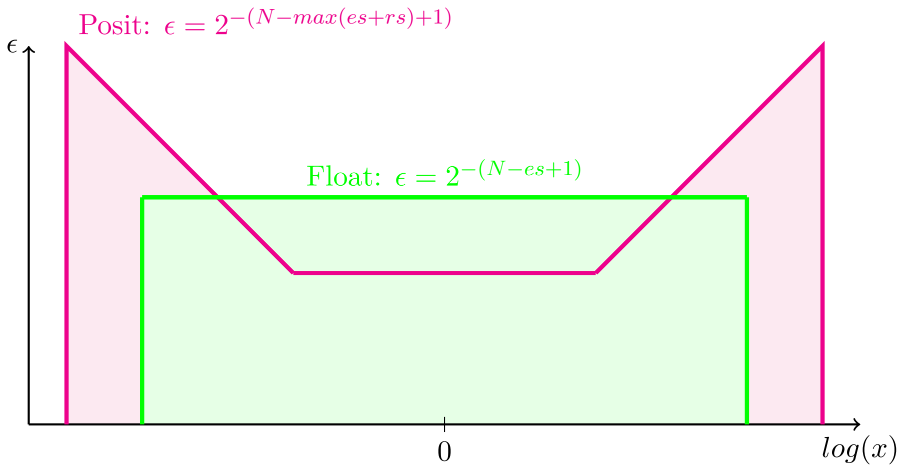


Figure 3.1: Summary of the error bound model for positive posit and floating point numbers. The flat area for posit corresponds to numbers that can be encoded using a regime size of 2. For larger (or smaller) numbers, the regime size increases and the error increases as well because there are less bits to encode the fraction. Floating point has a constant relative error bound due to its fixed size fraction field.

3.3.2 Posit

Error bound for posit require more work since the encoding error bound depends on the value which is encoded. Two cases must be taken into account: the maximum value of the SPN, and the minimum non-zero value of the SPN. Both of these can be the worst case scenario for a SPN since that are the values that require the most number of regime bits. Therefore there would be less bits allocated to the fraction and the relative error will be high.

Error for maximum output value This error can be computed by setting every input literal to one and propagate the error bounds up to the output as it was done for floating point. However, this error is rarely the biggest error since it is common that the maximum output value of the SPN is equal to one since it represents a probability[1]

Error for minimum output value The smallest value is more difficult to compute since there are a lot of possible combination of inputs. Therefore, a trick is used to get an approximation of the minimum possible value of the SPN. This trick consist in setting every literal to one (as for the maximum value), and replacing every SUM node by MIN nodes. In this case, the output of the SPN may not be a valid set of literals. But the output is smaller or equal to the minimum output for a valid set of literals.

[1]It is not a mandatory condition. If the maximum output of the SPN is not equal to one, the we should divide any results by the maximum value to get a probability.

In the end, the posit error bound for an SPN is the maximum value between the relative error bound computed for maximum output value and minimum output value.

Chapter 4

Hardware implementation

A hardware implementation consist in a set of file written in HDL with constraints associated to these files. First, I describe the hardware implementation of operations on posit numbers, namely addition and multiplication which are the basic building blocks of SPNs. Then I explain how SPNs hardware files are generated from a trained SPN file. Finally, I describe the implementation of the SPN inside the FPGA with the complete control scheme. The design is intended to work on a Zed Board.

4.1 Hardware operators

In this section, I describe how posit operations are performed in hardware. The general schematic is described in Figure 4.1. The floating point operation mentioned here represents the operation described in Algorithm 1 and 2. For a posit operation, the size of each field must be properly computed and is different from a classical floating point operation.

4.1.1 Encoder and Decoder

As posit representation use variable size fields, an encoder and decoder are required to extract useful information from a raw binary sequence. The useful information consists in a significand and an exponent (the exponent mentionned here corresponds to the real exponent of 2.4 which combines the exponent fields and the regime).

The decoder schematized in Figure 4.2a must extract the different quantities of a posit number to transform it into a useful format for computations. First it must retrieve the value of the real exponent. To do so, a leading zero counter can extract the value of the regime. Once the regime is found, the position of the exponent is known and can also be extracted. This extraction is done using shifting operators. These operators have the advantage to ignore the fact that the exponent bits may not be encoded due to the regime size. Then it must extract the value of the significand. This is similar to the extraction of the exponent.

The encoder schematized in Figure 4.2b is conceptually the opposite of the decoder. It takes three inputs: A regime value (in 2's complement), an exponent

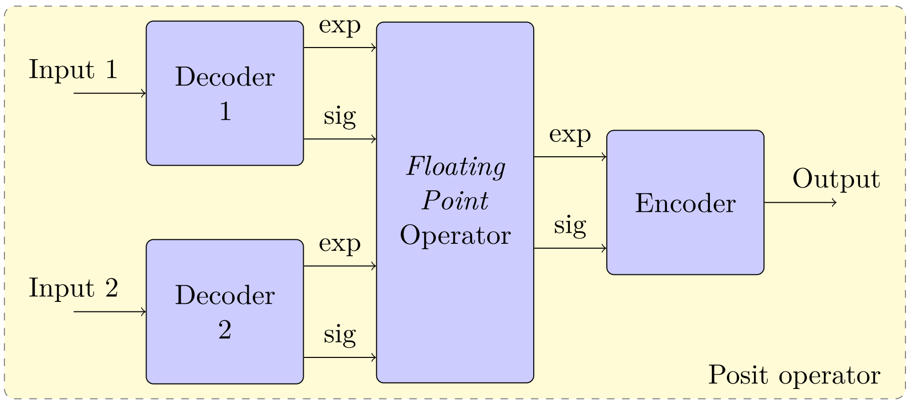


Figure 4.1: Illustration for posit operations. With a posit number of N bits and es exponent bits, the size of the significand is equal to $(N - 2 - es + 1)$ and the size of the exponent is equal to $(\log_2(N) + es + 1)$

value (usingned int) and a significand (unsigned int). First it extract the regime size as well as a regime writen in unary notation. Then it uses the regime size to shift the exponent and significand to place them at the correct place in the posit number.

4.1.2 Adder and multiplier

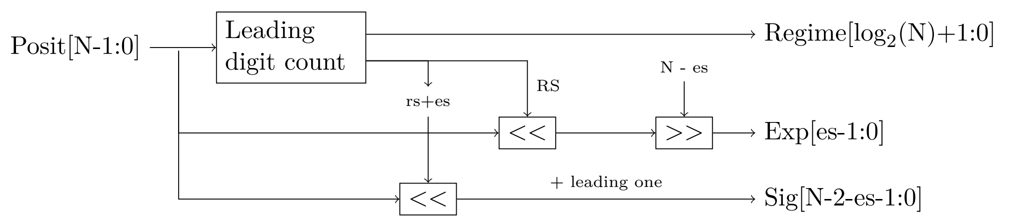
Once posit numbers are decoded into a single exponent and a significand, performing an addition or a multiplication is the same as the application of an addition or a multiplication in the context of floating point numbers as shown in Figure 4.1.

From 4.1, it is easy to conceive that a posit operation consume more hardware resources than a floating point operation since posit operations requires two encoders and one encoder. Note that even the floating point operation performed in the posit operator is bigger than a standard posit operation because the exponent and the significant must be big enough to represents the entire range of posit. The significand size is then of $(N - 2 - es + 1)$, and the exponent size of $(\log_2(N) + es + 1)$.

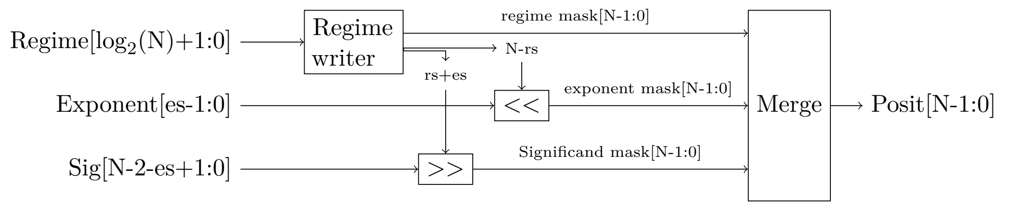
As a reminder, the benefit of using posit is that the range of representable number is increased and that a large set of number can be encoded using more fraction bits than floating point representation.

4.2 Sum product networks

SPNs are built using only adders and multipliers. The challenge of this section is that there exists many different architecture of sum product networks, and that there can be thousands of nodes in a single SPN. Therefore, the process of generating a SPN in HDL from a trained one is automatized.



(a) Decoder



(b) Encoder

Figure 4.2: Logical blocks for the encoder and decoder using shifts. Outputs of the decoder and input of the encoders are the regime (in 2's complement), the exponent (unsigned integer) and the significand (usigned integer). In order to get the value of the real exponent mentionned in Figure 2.4, the regime and the exponent field have to be concatenated. The leading digit count compute the value of the regime and the size of the regime. The regime writer build a regime in unary notation and output the regime size. N and es are constant and represent the number of bits and the exponent size of the posit number. rs is the regime size and is dynamically computed.

As inputs, it takes a file containing a trained SPN in the form of a PSDD. It outputs a HDL file representing the SPN with all the weights in posit representation. The inputs of the SPN are binary units because there are only zeros or ones. The first layer of the SPN consist in converting these inputs into posit numbers.

4.2.1 Pipelining

A SPN can be built using only basic building block such as adder and multiplier, but it would be very slow since the depth of the SPN can be high. Every time a new input would be sent, the output would be available after the longest datapath in the network is crossed. Furthermore, it would induce a high consumption of switching power. Therefore it is a good idea to pipeline the path from input to output in order to increase the maximum frequency and reduce the power consumption.

One can not simply add a register at every input or output of every nodes since some values can skip multiple layers at once. In order to pipeline properly the entire SPN, the depth of each node must be computed and an appropriate number of register must be set on each path. As shown in Figure 4.3, some paths may cross different pipeline stage and require multiple registers.

In addition to this, a valid signal flow through the pipeline. This signal is useful to synchronize the output of the SPN with the rest of the architecture. Whenever a new input is given to the SPN, valid is set to one, and flow up to the output. And whenever a valid equals to one is read at the output of the SPN, it means that the output was generated from a valid input.

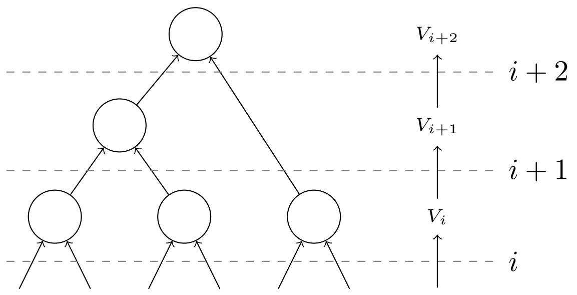


Figure 4.3: Pipelining of SPN. A new register is added every time a path cut a depth level. Valid signals flow through the pipeline to assert that values in this layers are generated from a valid set of inputs. Here valid means that it is a set of inputs sent by the user, it may not be valid in the definition of a SPN.

4.3 Global system

The global system shown in Figure 4.4 show the three main part of the structure: The FPGA, the control unit and the memory.

4.3.1 FPGA

Once the SPN is built, it must be integrated into a structure to be able to properly push inputs and get outputs. Two major problems are faced. Firstly, the number of inputs of a SPN may be high and we are limited to a 32 bits per clock cycle using the AXI interface. Secondly, the SPN works at a lower frequency than the rest of the hardware, so some clock domain crossing would be necessary to integrate the SPN inside a system.

In order to solve the problems of the number of inputs, a module whose function is to parallelize the inputs must be added in the FPGA. It is called here *input module*. The input width of the input module is limited to 32 bits (AXI interface) and works at a high frequency. The output width of the input module is the number of different literals of the SPN.

The different frequencies in the global systems are shown by the colors in Figure 4.4. In order to maximize throughput and perform a clock domain crossing, a FIFO is placed at the output of the SPN. The input of this FIFO works at a low frequency (SPN frequency) and the output works at a high frequency (AXI frequency).

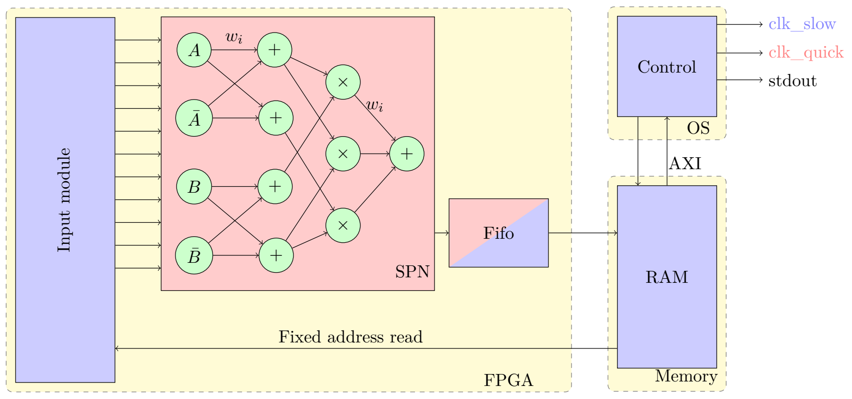


Figure 4.4: Architecture of Zed Board. The FPGA contains the SPN, an input module to parallelize the inputs and a first in first out (FIFO) to synchronize the outputs. The memory contains the software to be exectued by the CPU and a list of inputs to be sent to the FPGA. The CPU controls the entire structure through advanced extensible interface (AXI) communication.

The FPGA is size limited. Not all SPN will fit inside it. The key limiting factors are the number of slice look up tables (LUTs) and the number of digital signal processing (DSPs). A Zed Board possesses 53200 slice LUTs and 220 DSPs.

In answer to Nimish's comment: The biggest size of the SPN that fits on the Zed Board is mentioned in the results. Should it be also written here ?

4.3.2 Memory

The memory initially contains the software to be executed as well as the set of inputs for which we want to compute the probability.

4.3.3 Control unit

The control unit execute the software. It should send data to the FPGA and receive the output from the SPN. The maximum frequency of the FPGA is known by design. Therefore we can just push the data at the best frequency that will not break the SPN and get the data. Once the data is received, it is sent over stdout which can be accessible through joint test action group (JTAG) to get the data out of the Zed Board.

Chapter 5

Results and experiments

Two main subjects are introduced in Chapter 3 and 4. In this chapter, results and experiments for these two subjects are shown.

5.1 Error bound

In this section, the error bound for different SPNs is computed as it was described in 3. These error bounds over-estimate the maximum possible relative that is possible to observe in a given SPN. It may happen that a relatively high error bound is theoretically computed, and that in reality, the error is much smaller or even zero.

It starts by computing the error bound for a relatively small SPN in order to show exactly how it is computed and how the results can be interpreted. After that, we explore networks for which both floating point and posit representation could be used since the maximum and minimum value in these networks are in range of number format. Then we explore some networks for which only posit representation can be used thanks to its wider range.

The simulation is done using a C++ software available at `https://github.com/gennartan/spn_sw`. This software uses double floating point precision as ground truth. It is considered that this format is sufficient to represent error bounds of SPN. As a reminder, double floating point precision numbers uses 64 bits and 11 bits of exponent. Since double floating point format uses 11 bits of exponents, no more than 11 bits of exponents can be used in the simulation.

5.1.1 Simple example

Lets start by showing results for an simple example to give an intuition on the results. This examples is shown in Figure 5.1 represents a SPN with 4 different literals (1, $\bar{1}$, 2, $\bar{2}$) which represents two variables. Each branch has a label for which the error and the possible values are computed in Table 5.1.

In this example, we use 8-bits posit with 0 bits of exponent, and 8-bits float with 3 bits of exponents. From Table 5.1, it can be observed that the result has a smaller error bound using floating point representation than using posit representation. This behavior is mainly observed on small network because the values inside the SPN have a relatively short range. Therefore, encoding the exponent in a fixed size field lead to better results. In general, networks are more likely to have a large range of values and posit will have better results.

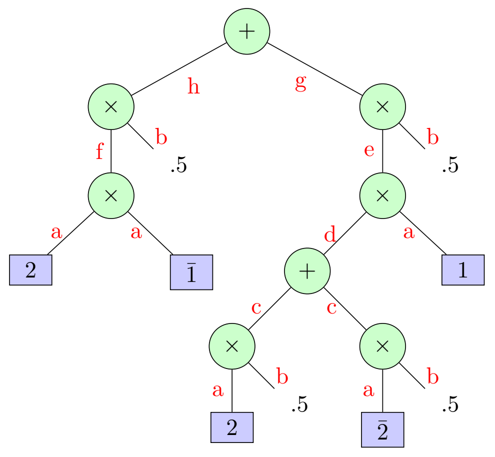


Figure 5.1: SPN example with 4 literals (2 variables), 2 sum nodes and 5 product nodes.

Table 5.1: Error analysis for 8 bits posit number with 0 exponent bits and 8 bits float with 3 bits of exponent for the SPN in Figure 5.1. Minimum and maximum value represents the minimum (non zero) and maximum value that the node can have. The relative error shows the relative error for posit representation of the minimum value and maximum value. These are the two candidates for worst case of SPN. The floating point representation only has one column of relative error since it does not depends on the value encoded. The final worst case relative error bound for posit is 0.1672 and for float it is 0.15.

Label	Min value	Max value	rel error Posit	rel error Float
a	1	1	2^{-7}	2^{-6}
b	0.5	0.5	2^{-6}	2^{-6}
c	0.5	0.5	0.0396	0.0476
d	0.5	1	0.0558	0.0639
e	0.5	1	0.0807	0.0975
f	1	1	0.0236	0.0476
g	0.25	0.5	0.1319	0.1321
h	0.5	0.5	0.0559	0.0806
res	0.25	1	**0.1672**	**0.150**

5.1.2 Error bound when both posit and float representation are in range

Given a number of bits to represent a number, we compute the relative error of floating point and posit representation. Each network was tested with a different number of bits for each representation from the following list : (8, 12, 16, 20, 24, 28). For each number of bits, and each representation, we optimize the exponent size by choosing the value that minimize the relative error from the following list: (0, 2, 3, 4, 6, 8, 10, 11).

In this section, we show only the results when both posit and floating point are in range for the minimum and maximum value of the SPN. The maximum exponent size which is tested is 11 since it represents the number of exponent bits for double precision floating point numbers which are used by the computer which performed the simulation.

Network 1 The first network that will be analyzed is the network containing 37460 nodes (28770 products and 8690 sums). For this network, it can be observed that posit representation is better than floating point representation. It can be interpreted by the fact that posit representation with 6 bits of exponent can represent most of the numbers in the SPN with a minimum regime size[1]. A higher regime size will be used only in some rare cases.

This show the major limitation of floating point numbers. In order to be able to represent the lower and maximum value of the SPN, it must have enough exponents bits. Therefore, some exponents bits are useful only in some specific set of literals, and reduce the mean precision of the SPN.

Table 5.2: Relative error for posit and floating point representation given a SPN containing 37460 nodes (28770 products, and 8690 sums). The exponent size was optimized to minimize the relative error.

	Posit		Float	
# bits	exp size	rel error	exp size	rel error
20	6	7e-2	10	3e-1
24	6	4e-3	10	2e-2
28	6	3e-4	10	1e-3

Network 2 The second network contains 3199 nodes (2501 products and 698 sums). It is about 10 times smaller than the previous one. In this case, it is expected that floating point numbers requires less exponent bits are are more competitive.

The results for this network are shown in Table 5.3. It can be observed that the worst case of the relative error for posit representation is closer to the relative error of floating point. However, posit is still better than floating point for every number representation.

[1]The minimum regime size is two bits. In this case it can represent a 0 ($b10$) or a -1 ($b10$).

Table 5.3: Relative error for posit and floating point representation given a SPN containing 3199 nodes (2501 products and 698 sums). The exponent size was optimized to minimize the relative error.

	Posit		Float	
# bits	exp size	rel error	exp size	rel error
16	4	1e-1	8	2e-1
20	4	7e-3	8	1e-2
24	4	4e-4	8	6e-4
28	4	3e-5	8	4e-5

From the two examples above, it can be returned that posit representation is better than floating point numbers when both representation are in range of their numbers.

5.1.3 Error bound when posit is in range and floating point representation is not

In this section we describe the results when the smallest value of a SPN can be represented by posit and cannot be represented by a floating point number due to the limited exponent size. The goal is to show that because floating point numbers can not be used in a specific SPN due to a lack of range, posit can be used and still have an error which is relatively small.

Network 3 The third network contains 827 nodes (622 products and 205 sums). The results are shown in Table 5.4. It can be observed that the error for posit representation is in a reasonable range. For this network, we should use at least 24 bits since the relative error of 0.5 for 20 bits can be considered as high value.

Table 5.4: Relative error for posit and float representation given a SPN containing 827 nodes (622 products and 205 sums). The exponent size was optimized to minimize the relative error.

# bits	exp size	rel error
20	8	5e-1
24	8	3e-2
28	8	2e-3

5.1.4 Error bound when floating point is more suitable than posit

In some rare case, floating point representation is more suitable than posit as it was shown in the simple example in Section 5.1.1. The goal here is to prove that even in the worst case SPN, posit representation remains competitive.

Network 4 This fourth network contains 7769 nodes (6134 products and 1635 sums). The results are shown in Table 5.5. In any case, floating point is more suitable than posit since the relative error is smaller for floating point than for posit. However, they remains very close. In the set of networks that were tested, only a few were more suitable for floating point than posit.

Table 5.5: Relative error for posit and float representation given a SPN containing 7769 nodes (6134 products and 1635 sums). The exponent size was optimized to minimize the relative error.

# bits	exp size	rel error Posit	exp size	rel error Float
24	8	8e-2	11	5e-2
28	8	5e-3	11	3e-3

5.1.5 Analysis of different parameters on a large set of SPNs

In this section, only SPNs which are in range of posit and floating point representation are taken into account.

A first analysis consists in observing the exponent size as a function of then number of bits. Results are shown in Figures 5.2a and 5.3a. It can be observed that the optimal exponent size for posit representation is 4 bits less that the optimal exponent size for floating point.

A second analysis consists in observing the mean error as a function of the number of bits. Results are shown in Figures 5.2b and 5.3b. The mean relative error is always better for posit than for floating point. However, when using a higher number of bits (28), the error is really close to zero and the actual difference of precision would be very very small.

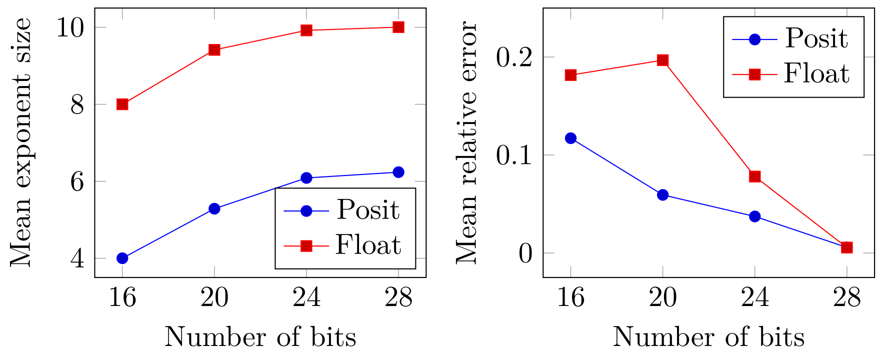
5.2 Hardware implementation

A complete hardware implementation can be found at `https://github.com/gennartan/Arithmetic_circuit_Posit_FPGA`. It contains all the necessary code to implement a SPN on a FPGA according to Figure 4.4.

The parameters to be set are the number of bits, the size of the exponent, the number of bits at the input of the FPGA (limited to 32), and the number of pipeline stage to be used. The key target of this implementation are the size (the SPN must fit into the FPGA) and the speed (it must run as quick as possible). Energy consumption is not optimized in this work.

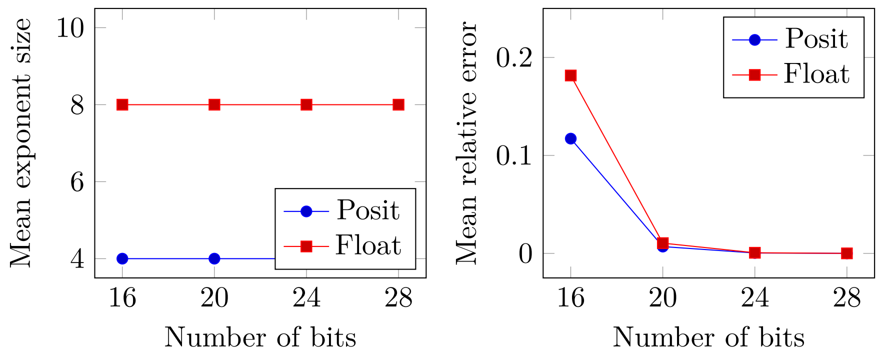
5.2.1 Posit

The properties of posit operators are shown in Table 5.6. It describes the size of these blocks (in terms of LUT) and the maximum frequency at which these units can run. Results of this sections are also compared to other working FPGA implementation of



(a) Mean exponent size (b) Mean error

Figure 5.2: Mean exponent size (left) and mean relative error (right) as a function of the number of bits of a large set of SPNs. This graphic is generated by generating each point with all the SPNs that are able for this number of bits. This induce a small bias in the results and explain the fact that error bound for 16 bits is smaller than 32 bits (the networks that works for 16 bits usually have a smaller error).



(a) Mean exponent size (b) Mean error

Figure 5.3: Mean exponent size (left) and mean relative error (right) as a function of the number of bits of a large set of SPNs. This grahic is generated using all the SPNs that are in range for 16 bits posit. Therefore the error becomes very small for larger size.

posit. This comparison does not compare two identical things since this work does not take into account the sign bit. Therefore there is more space allocated to the different fields of posit, which have a higher cost in hardware than a sign bit.

Table 5.6: Size and maximum frequency of posit operators reported by Vivado. This table also introduce the results from [12] which also implements posit operations under a Zed Board FPGA.

	New implementation			Literature [12]		
N	#LUTAdd	#LUTMult	f(MHz)	#LUT Add	#LUTMult	f(MHz)
8	141	170	67			
16	430	310+1DSP	45	400	220+1DSP	32
32	1100	850+4DSP	35	950	570+4DSP	25

Table 5.7: Relative error for posit and float representation given a SPN containing 67 nodes (50 products and 17 sums). The exponent size was optimized to minimize the relative error.

# bits	exp size	rel error Posit
16	4	5e-2
20	4	3e-3
24	4	2e-4
28	4	1e-5

5.2.2 SPN

In this section, we will show that the hardware implementation can be instanciated for a specific SPN, and that it can be extented to any network which conforms PSDDs properties. The chosen SPN has 67 nodes (50 products and 17 sums). The error bound for this network are shown in Table 5.7. This network was chosen because it fits on the Zed Board and because 16 bits posit numbers have enough precision to represent all the numbers in the SPN. 16 bits posit is the smallest number of bits that could work with a SPN of the dataset.

Hardware properties After an optimization using Vivado tool, this network uses 11000 LUTs and 50 DSPs which represents approximately one fourth of the available hardware on this Zed Board. This size cannot be directly computed from Table 5.6 because the tool optimizes the network are remove / reduce the size of nodes which are not completely used. This highly depends on the SPN architecture and on its weights. Without the tool optimization, the network size would be in the range of 20000 LUTs and 50 DSPs.

The SPN was generated with a frequency of 100 MHz for the input module and 35MHz for the SPN. Using these frequences, the global system is limited to a maximum thoughput of 35MHz. However, the actual thoughput of the SPN is equal 13MHz. This result is much lower that was expected. There are two reasons that explains this lack of efficiency. The first reason is that the clock domain syncer at the output of the input module takes two clocks cycle to transfer a data. Only because of this, the maximum expected throughput fall from 35MHz to 17.5MHz. The second reason is that the software that must send the data to the FPGA must also get the output data. Therefore it will loose some time when it is not sending inputs to the input module.

Error bound verification In order to proove that our design works properly and that the error bound is also correctly computed, we test the SPN with a bunch of inputs for this network that comes from [15] and which were introduced in [5], [8], [3] and [16]. If none of these inputs prove to have a relative error bigger than the error bound it can be considered that both the hardware implementation and the error bound are correcly set.

The maximum relative error computed is $0.747111 \cdot 2^{-5} == 2.33e-2$ which is smaller than the computed error bound ($2.33e-2 \leq 5e-2$). Therefore we can conclude that this error bound is correctly computed. We also computed the mean relative error through the network in order to know if the error observed on an output is seldom or not. The mean relative error computed is equalt to $0.513 \cdot 2^{-5}$ which is close to the maximum relative error.

5.2.3 Futher work

The hardware implementation works properly and can be easily regenerated for any SPN. However, some particular cases may require additional work in order to run. In this section I describe the limitations of the current implementation and explain what could be done in order to address these problems.

The current hardware posit implementation is able to work only with three different number of bits: 8, 16 and 32. This is due to the use of a custom module designed to count the number of leading zero (or one). To remedy this problem, this module should be upgraded to be able to count the number of leading zero for any number of bits. This could be done by adding padding to the inputs or by creating a new module for this function.

Posit standard describes a specific rounding method. However, due to the fact that the regime is unary coded, the rouding of a number can lead to a specific case where the regime would gain (or lose) one bit and require speciifc hardware module to handle this. It would reduce speed and increases ressources consumption.

The operations designed for posit are only two inputs operations. In a SPN, it is common to observe nodes with more than two inputs. A possible upgrade to this could be to create specific n-inputs operators which could perform multiple operations at the same time. In order to implement this, the code that generate the SPN should be update to handle this.

The output of the input module uses a clock domain syncer to send the data to the SPN. This clock domain syncer is not efficient and require two SPN clock cycles. Therefore the effective frequency of the SPN is divided by two. This could be increased by placing a FIFO at the output of the input module in the same way that there is a FIFO at the output of the FPGA.

The complete design is controlled by a software that runs on the Zed Board. The generation of this software is not automated. Therefore it should be written again for any new SPN. A process to automatize this task could be performed.

Chapter 6

Conclusion

SPNs are a new architecture [11] of deep networks for graphical probabilistic inference. This master's thesis studied the use of custom posit representation for SPNs implementation over FPGA. A dual approach is taken. First, a comparison of error bounds of floating point and posit is made. This analysis compare the efficiency in terms of accuracy of different number representation. Second, a hardware implementation is built to evaluate the hardware cost of such a design in a real environment.

From the analysis of relative error bounds in Chapter 3, it emerged that posit representation had better accuracy for most of the SPNs of the dataset. The analysis performed limits the number of bits for each representation to 32 and the size of the exponent fields to 11. Four different networks have been analyzed corresponding different use cases.

A first network was analyzed for which both floating point and posit are in range. It was chosen because it has a huge number of nodes (28770 nodes). For this network the error bound for posit was 3 to 5 times lower than for floating point.

A second network was analyzed for which both representations are also in range. This one was chosen because it was much smaller than the previous one (3199 nodes) and confirmed the results of the first network.

The third network corresponds to the case where posit is in range while floating point is not. These SPNs have a wide range of number for which 11 bits of exponents is not sufficient. Therefore the use of posit becomes mandatory. This analysis shown that the computed error bound for posit was relatively small and that the fact that floating point is not in range does not mean that the error for posit explodes.

A last network shows an example for which floating point has better accuracy than posit. There exists a few networks for which floating point is better than posit. Even if posit remains very competitive to floating point in such networks, posit should not completely replace them. Floating point numbers still have a lot of hardware advantages compared to posit that justify the preferred use of floating point.

The last analysis concerning the comparison of error bound for posit and float averages the results from the entire set of SPNs. It emerged from this experiment that posit had a big advantages for lower number of bits (16, 20) and that this advantages is getting lower for larger number of bits (24, 28).

In Chapter 4, a complete hardware implementation of SPNs using posit representation is built. It is done in three steps: First, the basic building blocks of posit arithmetic are designed. These are the main blocs to be used for SPNs generation. Second, SPNs are generated from a file. Third, this generated SPN is integrated into a global structure to link the network with a control unit. The control unit runs a software that would send the inputs to the FPGA and get the outputs from it.

The results for the design of posit arithmetic show that the implementations described in this work has comparable size and speed to other implementations. Using 8, 16 and 32 bits, the maximum frequency of goes to 67, 45 and 35MHz respectively.

An example of SPN generation is done using a network with 67 nodes. Such a network use approximately one quarter of the available hardware resources. This shows the main limitation of the Zed Board. Large networks cannot fit inside it. Using 16 bits posit, the maximum frequency of this network is then 45MHz.

The global system is built and allows the transfer of data from the control using to the FPGA. Using this implementation, a maximum throughput of 13MHz can be achieved. This throughput, much lower than the expected SPN throughput of 35MHz, is mainly due to a clock domain crossing consuming two clock cycle to transfer one data.

Finally, this global system also given the opportunity to run a complete test by comparing the output of the hardware posit implementation with the simulation software of SPNs. This test validate both the error model and the hardware implementation and showed that the error bound computed is actually two times lower than the error bound.

Acronyms

ASIC application-specific itegrated circuit. 5

AXI advanced extensible interface. 21

CPU central processing unit. 5, 21

DSP digital signal processing. 21, 29

FIFO first in first out. 20, 21, 30

FPGA field programable gate array. 1, 3, 5, 17, 20, 21, 27, 29–32

HDL hardware description language. 1, 5, 17, 18

JTAG joint test action group. 21

LUT look up table. 21, 27, 29

PSDD probabilistic sequential decision diagram. 5, 19, 29

SPN sum product network. 1–6, 11, 13–15, 17–21, 23–32

UNum universal number. 1, 6

Bibliography

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[2] arranger1044. awesome-spn. https://github.com/arranger1044/awesome-spn, Jun 2019. [Online; accessed 28. May 2020].

[3] J. Bekker, J. Davis, A. Choi, A. Darwiche, and G. Broeck. Tractable Learning for Complex Probability Queries. *Advances in neural information processing systems*, 28, Dec 2015.

[4] Contributors to Wikimedia projects. Unum (number format) - Wikipedia, Apr 2020. [Online; accessed 12. May 2020].

[5] J. D. Daniel Lowd. Learning Markov Network Structure with Decision Trees. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.188.7026, May 2020. [Online; accessed 27. May 2020].

[6] K. et. al. Probabilistic sequential decision diagrams: Learning with massive logical constraints. 2014.

[7] J. L. Gustafson. Posit arithmetic. https://www.posithub.org/docs/Posits4.pdf, 2017.

[8] J. D. Jan Van Haaren. Markov network structure learning: A randomized feature generation approach. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.704.5655, May 2020. [Online; accessed 27. May 2020].

[9] I. Y. John L. Gustafson. Beating floating point at its own game: Posit arithmetic. http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf.

[10] A. Podobas and S. Matsuoka. Hardware Implementation of POSITs and Their Application in FPGAs. *ResearchGate*, pages 138–145, May 2018.

[11] H. Poon and P. Domingos. Sum-Product Networks: A New Deep Architecture. *arXiv*, Feb 2012.

[12] R. S. J. N. S. N. K. N. F. M. R. L. Rohit Chaurasiya, John Gustafson. Parameterized posit arithmetic hardware generator.

[13] B. M. Sguerra and F. G. Cozman. Image classification using sum-product networks for autonomous flight of micro aerial vehicles. In *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 139–144, 2016.

[14] N. Shah, L. I. G. Olascoaga, W. Meert, and M. Verhelst. ProbLP: A framework for low-precision probabilistic inference. *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2020.

[15] Ucla-Starai. Density-Estimation-Datasets, Dec 2019. [Online; accessed 27. May 2020].

[16] B. Uria, M.-A. Côté, K. Gregor, I. Murray, and H. Larochelle. Neural Autoregressive Distribution Estimation. *arXiv*, May 2016.

[17] P. working group. Posit standard documentation. `https://posithub.org/docs/posit_standard.pdf`.

[18] G. V. d. B. Yitao Liang, Jessa Bekker. Learnig the structure of probabilistic sequential decision diagram.

Master's thesis filing card

Student: Antoine GENNART

Title: Low precision artihmetic using posit in sum product networks over FPGA and error bound analysis

UDC: 621.3

Abstract:

Here comes a very short abstract, containing no more than 500 words. LaTeX commands can be used here. Blank lines (or the command `\par`) are not allowed!

Thesis submitted for the degree of Master of Science in Electrical Engineering, option Electronics and Integrated Circuits

Thesis supervisor: Prof. Marian VERHELST

Assessor: Prof. Luc DE RAEDT
Prof. Nele MENTENS

Mentor: Ir. Nimish SHAH