KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Error bound analysis of sum product networks using posit representation number over FPGA

Antoine GENNART

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

Thesis supervisor:
Prof. dr. ir. Marian VERHELST

Assessor:
Ir. TO FILL

Mentor:
Ir. Nimish SHAH

Academic year 2019 – 2020

Preface

TODODODOD I would like to thank everybody who kept me busy the last year, especially my promoter and my assistants. I would also like to thank the jury for reading the text. My sincere gratitude also goes to my wive and the rest of my family.

Antoine Gennart

Contents

Abstract

abstract..

The `abstract` environment contains a more extensive overview of the work. But it should be limited to one page.

Chapter 1

Introduction

Sum product networks (SPNs) are a brand new deep network architecture for graphical inference [4]. This architecture is designed to make the partition function tractable. In this work, I study the error bound of the SPN architecture and observe how it grows in each layer.

Two representations are studied: posit [7] and floating point [1]. These two number representations are used to represent floating point numbers in computers. The main difference between these representations is that the exponent field is encoded using a variable length for posit. Therefore there would be cases where posit is better than floating point because exponent fields would be saved, and cases where floating point is better than posit because exponent fields would be wasted. the code to compute the error bound is available at https://github.com/gennartan/spn_sw.

A hardware implementation of SPNs in field programable gate array (FPGA) is also provided to prove the efficiency of posit representation in a real application. The entire design is made available at https://github.com/gennartan/Arithmetic_circuit_Posit_FPGA/.

Introduction should be longer and explain a bit more the problematic...

Chapter 2

State of the art

2.1 Sum product networks

Definition 1 *SPN A SPN is a rooted acyclic graph with variables as leaves, sum and product as internal nodes, and weighted edges [4].*

SPNs allows to represent a probability function by keeping the general conditions that makes this function tractable. Two graphical representations from [4] 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 $P(X_0, \bar{X}_0, ..., X_N, \bar{X}_N)$.

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 both be 0 at the same time.

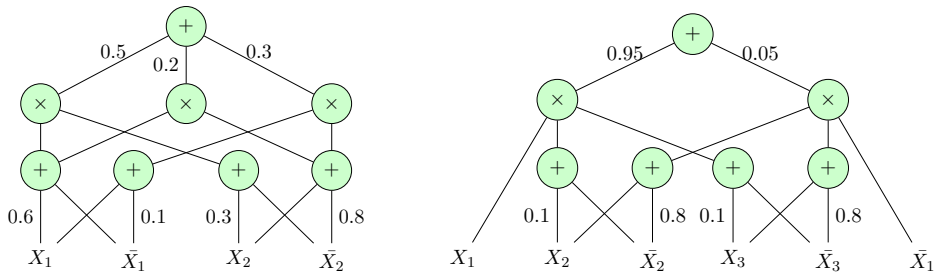


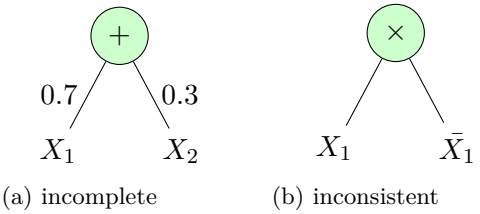
Figure 2.1: Examples of SPNs from [4]

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.*



(a) incomplete (b) inconsistent

Maybe add a real application of SPN as an example ? To be discussed with Nimish

2.2 Floating point representation

Floating point representation is the most widely used binary representation of numbers for digital applications. The floating point standard IEEE Std 754-2008 is defined in [1]. In this work, we do not use floating point number compatible with this standard. The number of bits for the exponent and the fraction may be optimized for a given application. In addition to this, the sign bit is not used since SPNs are used to represent probabilities, which are only positive numbers.

2.2.1 Floating point bit-fields

The general representation of the fields of the floating point representation are shown in Figure 2.2. There is a sign bit (which is removed in this work), a certain number of exponent bits, and a certain number of fraction bits with an implicit leading one. In a given application, the number of exponent and fraction bits are fixed and cannot be modified.

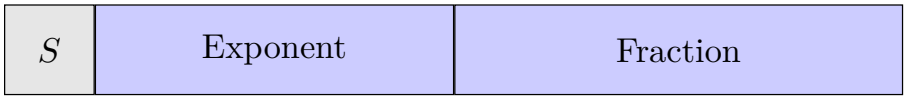


Figure 2.2: Floating point format with sign bit. There are three main field: A sign bit (S), a fixed size exponent, and a fraction.

In order to compute the value represented by a floating point number, the Equation 2.1 must be used where *exp* represent the signed value of the exponent field and b_i represent the i^{th} bits of the fraction. Depending on the standard used, the exponent can also be encoded using a unsigned integer and be shifted afterward.

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

2.2.2 Floating point operation

Addition and multiplication operations are also possible using the floating point operation. They require simple operation such as comparison, and shifting. Addition and multiplication are described in Algorithm 1 and 2 respectively.

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 Posit representation

Just like floating point representation, posit representation is a method to represent a floating point number using a fixed point number of binary units. It is also know as the type III universal number (Unum) [2].

2.3.1 Posit fields

Posit format possesses four fields as shown in Figure 2.3. In this work, the sign bit is not used since posit numbers are used to compute SPNs, which computes probabilities. Therefore, there are only positive numbers to be represented.

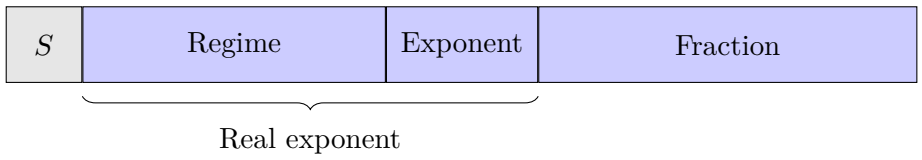


Figure 2.3: POSIT format with sign bit. There are four main fields: A sign bit (S), a regime (k), an exponent (exp) and a fraction (frac).

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 bits. 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 size. If there are not enough number of bits left in the posit representation due to the regime variable size, every unknown bit of this field has a value of "b0".

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 significand is the term used to described the fraction to which we add the leading one.

Given these four fields, one can compute the value using Equation 2.2. In Figure 2.3, it is mentioned that the combination of the regime and the exponent field form the real exponent. It is the case because we must use both these numbers to compute the exponent which is applied to the significand.

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

2.3.2 Posit operations

Once decoded, posit numbers can be represented as an exponent and a significand. Therefore, a posit addition or multiplication can be performed using Algorithm 1 and 2. However, a posit encoder and decoder must be used. The pseudo code for the encode and decoder are described in Algorithm 4 and 3 respectively.

There are two negative impact for posit numbers. First, because it requires two decoders and one encoders, computation time may be longer. Second, the floating

point operation that is performed after decoding the input numbers is also slower than a usual floating point operations since the size of each field (exponent and mantissa) would be bigger than for floating point (given the same number of bits for each representation).

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;
```

Chapter 3

Error bound

3.1 Error bound for Float

As posit is a widely used format, there already exists error bounds for floating point representation in the context of SPNs [6]. Using an absolute error bound would lead to very small quantity, therefore a relative error is used instead. This section report the method to compute error bound for floating point representation as described in [6].

Is it clear enough that floating point comes from Nimish's paper ? To be discussed with Nimish. Value that are out of range for float are just discarded.

Given that the goal is to use limit the number of bits used for a given representation, it may happen that a number of bits is not sufficient to represent all the values in a SPN. In this case, the number of bits is marked as *out of range* and will be discarded.

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.

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

3.1.1 Encoding error

The relative error for a floating point number can be computed based on the mantissa size (ms) as it is done is Equation 3.2

$$\left|\frac{\Delta f}{f}\right| \leq 2^{-(ms+1)} \tag{3.2}$$

The relative error ϵ is then bounded by a value that does not depend on the value to be encoded, but depends on the number of mantissa bits (which is fixed). The only exception is zero, which can be encoded with a relative error equals to zero.

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 can encode very small and very big numbers.

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

3.2.1 Encoding error

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.

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

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.

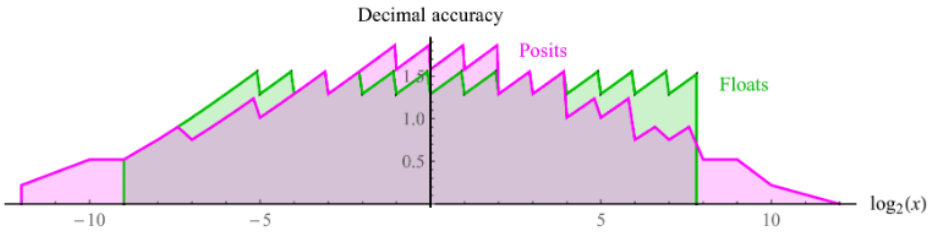


Figure 3.1: Image from [3]. It shows how posit as a better accuracy for representing small numbers, and is able to represent very high numbers while floating point cannot.

3.2.2 Addition error

The error for every addition can be computed as it is done in Equation 3.7. In this case, ϵ represent the error performed while encoding the result of the operation. Therefore, it depends on the value of the number.

$$\begin{aligned}
\hat{p} &= (\hat{a} + \hat{b}) \cdot (1 \pm \epsilon) \\
&= (a \cdot (1 \pm \epsilon_a) + b \cdot (1 \pm \epsilon_b)) \cdot (1 \pm \epsilon) \\
&= (a + b + a\epsilon_a + b\epsilon_b) \cdot (1 \pm \epsilon) \\
&= (a + b) \cdot \left(1 \pm \frac{a}{a+b} \cdot \epsilon_a \pm \frac{b}{a+b} \cdot \epsilon_b\right) \cdot (1 \pm \epsilon) \\
&\leq (a + b) \cdot (1 \pm max(\epsilon_a, \epsilon_b)) \cdot (1 \pm \epsilon) \\
(1 + \epsilon_f) &= (1 \pm max(\epsilon_a, \epsilon_b)) \cdot (1 \pm \epsilon)
\end{aligned} \tag{3.7}$$

3.2.3 Multiplication error

The error for every multiplication can be computed as it is done in Equation 3.8. As for posit addition, ϵ depends on the result of the operation.

$$\begin{aligned} \hat{p} &= \hat{a} \cdot \hat{b} \cdot (1 \pm \epsilon) \\ &= a \cdot (1 \pm \epsilon_a) \cdot b \cdot (1 \pm \epsilon_b) \\ &= a \cdot b \cdot (1 \pm \epsilon_a) \cdot (1 \pm \epsilon_b) \cdot (1 \pm \epsilon) \\ (1 \pm \epsilon_p) &= (1 \pm \epsilon_a) \cdot (1 \pm \epsilon_b) \cdot (1 \pm \epsilon) \end{aligned} \tag{3.8}$$

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.

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.

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.

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. However, this error is rarely the biggest error since it is common to normalize the SPN at the end of the training (set maximum output to 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 than the minimum output for a valid set of literals.

Chapter 4

Hardware implementation

A hardware implementation consist in a set of file written in hardware description language (HDL) with constraints associated to these files. First, I describe the hardware implementation of operations on posit numbers, 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 Posit representation

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.

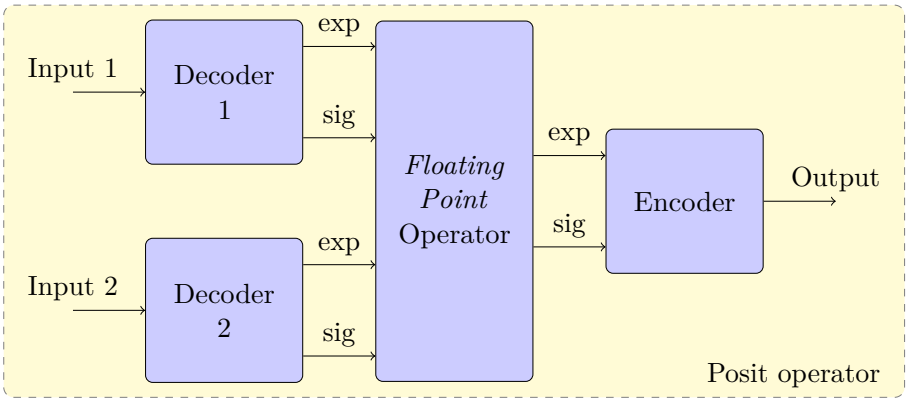


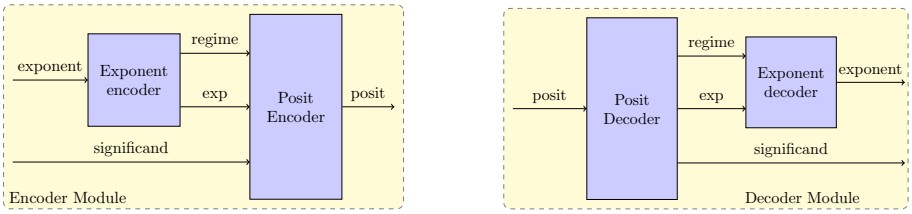
Figure 4.1: Illustration for posit operations

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 encoder schematized in Figure 4.2a is composed of two smaller encoder. A first one which convert an exponent into a regime value and an exponent value. And a second one which take as input the regime, the exponent and the significand to output a posit number. It is in this block that the number are truncated in case of rounding and that information is lost.

The decoder schematized in Figure 4.2b is basically the inverse of the encoder except that there is no rounding implemented.



(a) Encoder (b) Decoder

Figure 4.2: Schematic of encoder and decoder modules

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. 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 are of variable size.

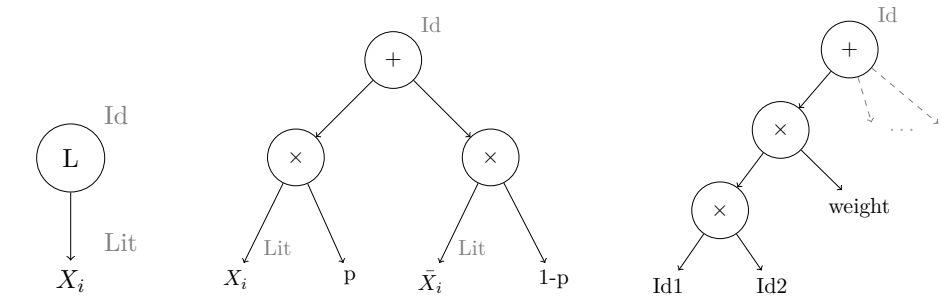
4.2 Sum product networks

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, we automatized the process of generating a SPN in HDL from a trained one.

4.2.1 PSDD file format

In order to generate hardware representation of an SPN, We choose to use PSDD file format since it was already used by the research group of my daily advisor.

A PSDD file contains three types of nodes: leaf nodes, true nodes and decomposition nodes. Each of these node can be decomposed into a set of wire, sum or product as it is shown in Figure 4.3.



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

Figure 4.3: Decomposition of PSDD file into simple arithmetic expression for SPN format.

4.2.2 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. Therefore it is a good idea to pipeline the path from input to output in order to increase the maximum SPN frequency.

One can not simply add a register at every input or output of every node 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.

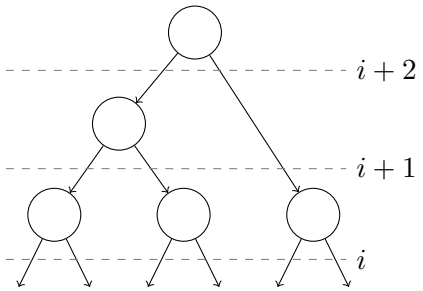


Figure 4.4: Pipelining of SPN. A new register is added every time a path cut a depth level.

4.3 Global system

Having the hardware code for a SPN is not sufficient. It is also necessary to be able to control it and request for a specific output given a set of input. In this case, the SPN is intended to work on a Zed Board FPGA. In this board, we can have access to a FPGA, a random access memory (RAM) and a central processing unit (CPU) as shown in Figure 4.5.

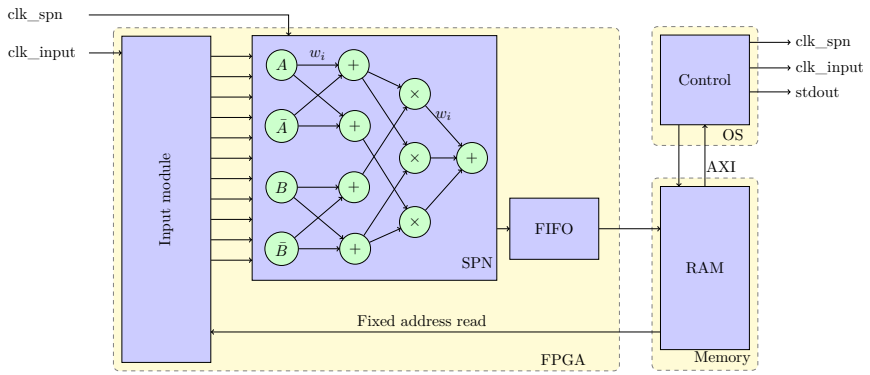


Figure 4.5: Content of zed Board. The FPGA contains the SPN and an input module to paralellize the inputs of the SPN. The embedded operating system launch a software which control the flow of data from the memory to the SPN and from the SPN to the memory. Data transfer are performed through advanced extensible interface (AXI) bus

4.3.1 FPGA

The FPGA contains the SPN as well as an input module whose function is to convert a serial input to a parallel output. SPNs may have a high number of inputs and the connection from the RAM to the FPGA is limited to 32 bits. Therefore this module is required.

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.

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

5.1 Error bound

In this section, we are computing error bound for different SPNs 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.

we start 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 can be encoded. Then we explore some networks for which only posit representation can be used.

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 and shows a SPN with 4 different literals (1, $\bar{1}$, 2, $\bar{2}$) which represents to 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.

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).

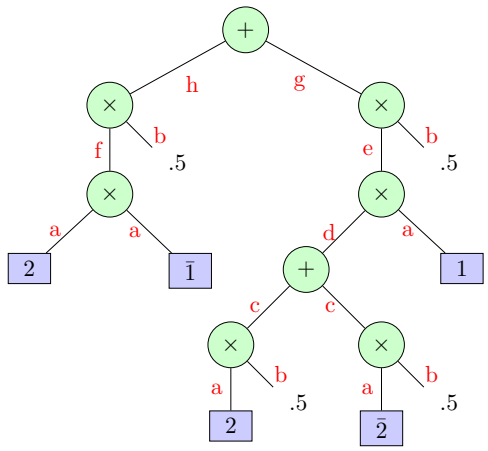


Figure 5.1: SPN example with 4 literals (2 variables), 2 sum nodes and 5 product nodes.

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.

Label	min value	rel error	max value	rel error	rel error float
a	1	2^{-7}	1	2^{-7}	0.2^{-6}
b	0.5	2^{-6}	0.5	2^{-6}	2^{-6}
c	0.5	0.0396	0.5	0.0396	0.0476
d	0.5	0.0558	1	0.0477	0.0639
e	0.5	0.0807	1	0.0641	0.0975
f	1	0.0236	1	0.0236	0.0476
g	0.25	0.1319	0.5	0.0976	0.1321
h	0.5	0.0559	0.5	0.0559	0.0806
res	**0.25**	**0.1672**	**1**	**0.1062**	**0.150**

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 float 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	min rel error	max rel error	exp size	rel error
20	6	7e-2	6e-2	10	3e-1
24	6	4e-3	4e-3	10	2e-2
28	6	3e-4	2e-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.

Table 5.3: Relative error for posit and float 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	min rel error	max rel error	exp size	rel error
16	4	1e-1	4e-2	8	2e-1
20	4	7e-3	3e-3	8	1e-2
24	4	4e-4	2e-4	8	6e-4
28	4	3e-5	1e-5	8	4e-5

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

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 float 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 when floating point numbers can not be used in a specific SPN, 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.

	Posit		
# bits	exp size	min rel error	max rel error
20	8	5e-1	5e-1
24	8	3e-2	2e-2
28	8	2e-3	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 and that it could be used in every networks.

Network 4 This fourth network contains 7769 nodes (6134 products and 1635 sums). The results are shown in Table 5.5. In the worst case it is better than posit. This is only by a small margin.

5.1.5 Analysis of different parameters on a large set of SPNs

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

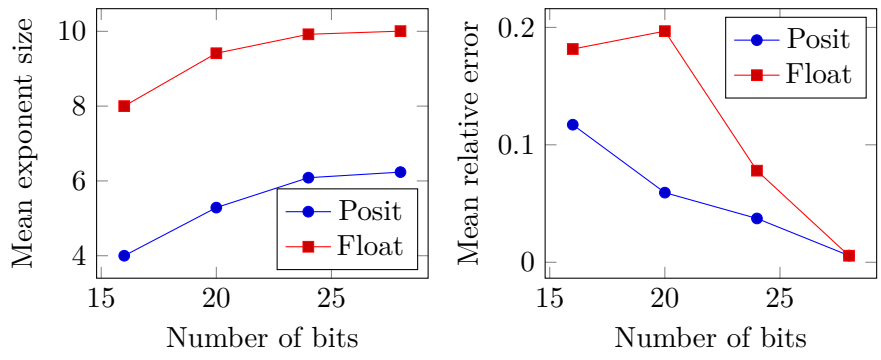
A first analysis consists in observing the exponent size as a function of then number of bits. Results are shown in Figure 5.2a. It can be observed that the optimal

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.

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

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 Figure 5.2b. 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.



(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.

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 4.5.

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.

5.2.1 Posit

Here I present the properties of the posit adder and multiplier that I designed. It consist in describing the size of these blocks (in terms of LUT) and the maximum frequency at which these blocks can run. Then I compare these characteristics with another working implementation of posit on FPGA.

Table 5.6: Size and maximum frequency of posit operators reported by Vivado. This table also introduce the results from [5] which also implements posit operations under a zed Board FPGA.

	New implementation			Literature [5]		
N, es	#LUTAdd	#LUTMult	f(MHz)	#LUT Add	#LUTMult	f(MHz)
8, 0	141	170	67	NA	NA	NA
8, 1	159	187				
8, 2	193	173				
8, 3	174	154				
16, 0	430	321	45	≃400	≃ 220 +1DSP	32
16, 1	457	394				
16, 2	500	381				
16, 3	494	391				
32, 0	1197	866	35	≃950	≃570 +4DSP	25
32, 1	1092	811				
32, 2	1110	914				
32, 3	1061	1056				

5.2.2 SPN

SPNs results.. Results are ready, I already have the output from the FPGA, but I just need to compare it with the output of the software

Add reference of the data-set used in hardware

Add reference of test set for hardware also

Chapter 6

Conclusion

Acronyms

AXI advanced extensible interface. 16

CPU central processing unit. 16

DSP digital signal processing. 16, 24

FPGA field programable gate array. 1, 13, 16, 23, 24

HDL hardware description language. 13, 14

JTAG joint test action group. 17

LUT look up table. 16, 24

RAM random access memory. 16

SPN sum product network. 1, 3, 4, 6, 9, 12–17, 19–23

Unum universal number. 5

Bibliography

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

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

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

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

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

[6] 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.

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

Master's thesis filing card

Student: Antoine Gennart

Title: Error bound analysis of sum product networks using posit representation number over FPGA

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. dr. ir. Marian Verhelst

Assessor: Ir. TO FILL

Mentor: Ir. Nimish Shah