A FIXED-POINT IMPLEMENTATION OF THE EXPANDED HYPERBOLIC CORDIC ALGORITHM

Daniel R. Llamocca-Obregón
llamocca.dr@pucp.edu.pe

Carla P. Agurto-Ríos
agurto.cp@pucp.edu.pe

Grupo de Procesamiento Digital de Señales e Imágenes - Pontificia Universidad Católica del Perú
Av. Universitaria s/n Cuadra 18 - Lima 32, Perú
Telf.: +511-6262000 Anexo 4681

ABSTRACT

The original hyperbolic CORDIC (Coordinate Rotation Digital Computer) algorithm [1] imposes a limitation to the inputs' domain which renders the algorithm useless for certain applications in which a greater range of the function is needed. To address this problem, Hu et al [2] have proposed an interesting scheme which increments the iterations of the original hyperbolic CORDIC algorithm and allows an efficient mapping of the algorithm onto hardware.

A fixed-point implementation of the hyperbolic CORDIC algorithm with the expansion scheme proposed by Hu is presented. Three architectures are proposed: a low cost iterative version, a fully pipelined version, and a bit serial iterative version. The architectures were described in VHDL, and to test the architecture, it was targeted to a Stratix FPGA. Various standard numerical formats for the inputs are analyzed for each hyperbolic function directly obtained: *Sinh, Cosh, Tanh*$^{-1}$ and *exp*. For each numerical format and for each hyperbolic function an error analysis is performed.

1. INTRODUCTION

The hyperbolic CORDIC algorithm as originally proposed by Walther[1] allows the computation of hyperbolic functions in an efficient fashion. However, the domain of the inputs is limited in order to guarantee that outputs converge and yield correct values, and this limitation will not satisfy the applications in which nearly the full range of the hyperbolic functions is needed.

Various strategies have been proposed to address the problem of limited convergence of the hyperbolic CORDIC algorithm. One strategy is to use mathematical identities to preprocess the CORDIC input quantities[1]. While such mathematical identities work, there is no single identity that will remove or reduce the limitations of all the functions in the hyperbolic mode. In addition, the mathematical identities are cumbersome to use in hardware applications because their implementation requires a significant increase in processing time and hardware[2]. Another approach, proposed by Hu et al [2], involves a modification to the basic CORDIC algorithm (inclusion of additional iterations) that can be readily implemented in a VLSI architecture or in a FPGA without excessively increasing the processing time.

Three architectures for the fixed-point implementation of the hyperbolic CORDIC algorithm with the expansion scheme proposed by Hu[2] are presented: a low cost iterative version, a fully pipelined version, and a bit serial iterative version. Results in terms of resource count and speed were obtained by targeting the architectures, described in VHDL, to a Stratix FPGA of ALTERA®.

Four different numerical formats are proposed for the inputs. For each hyperbolic function, an analysis of each numerical format is performed and the optimal number of iterations along with the optimal format for the angle are obtained. Finally, an error analysis is performed for each hyperbolic function with each numerical format. The data obtained with the fixed-point architectures are contrasted with the ideal values obtained with MATLAB®.

The rest of this paper is organized as follows: In Section 2, the expansion scheme for the hyperbolic CORDIC algorithm is presented. Section 3 describes the three architectures implemented. Section 4 presents an analysis of each input numerical format for each hyperbolic function, so that an optimum output numerical format and architecture can be obtained. In addition, the results of the FPGA implementation for selected numerical formats and functions are shown. Section 5 presents an error analysis. Finally, conclusions and recommendations are given.

2. EXPANSION SCHEME FOR THE HYPERBOLIC CORDIC ALGORITHM

2.1 Original Hyperbolic CORDIC algorithm

The original hyperbolic CORDIC algorithm, first described by Walther [1], states the following iterative equations:

$$\begin{aligned} X_{i+1} &= X_i + \delta_i Y_i 2^{-i} \\ Y_{i+1} &= Y_i + \delta_i X_i 2^{-i} \\ Z_{i+1} &= Z_i - \delta_i \theta_i \end{aligned} \quad (1)$$

$$\text{Where: } \theta_i = Tanh^{-1}\left(2^{-i}\right) \quad (2)$$

And i is the index of the iteration (i = 1, 2, 3, … N). The following iterations must be repeated in order to guarantee the convergence: *4, 13, 40,… k, 3k + 1*. The value of δ_i is either +1 or –1 depending on the mode of operation:

$$\begin{aligned} &Rotation: \quad \delta_i = -1 \quad if \ z_i < 0, +1, otherwise \\ &Vectoring: \quad \delta_i = -1 \quad if \ x_i y_i \geq 0, +1, otherwise \end{aligned} \quad (3)$$

In the rotation mode, the quantities X, Y and Z tend to the following results, for sufficiently large N:

$$X_n \leftarrow A_n\left[X_0 CoshZ_0 + Y_0 SinhZ_0\right]$$
$$Y_n \leftarrow A_n\left[Y_0 CoshZ_0 + X_0 SinhZ_0\right] \quad (4)$$
$$Z_n \leftarrow 0$$

And, in the vectoring mode, the quantities X, Y and Z tend to the following results, for sufficiently large *N*:

$$X_n \leftarrow A_n\sqrt{{X_0}^2 - {Y_0}^2}$$
$$Y_n \leftarrow 0 \quad (5)$$
$$Z_n \leftarrow Z_0 + Tanh^{-1}\left(Y_0 / X_0\right)$$

Where 'A_n' is: $A_n \leftarrow \prod_{i=1}^{N}\sqrt{1-2^{-2i}}$ (6)

With a proper choice of the initial values X_0, Y_0, Z_0 and the operation mode, the following functions can be directly obtained: *Sinh, Cosh, Tanh*$^{-1}$, and *exp*. Additional functions (e.g. *ln, sqrt, Tanh*) may be generated by applying mathematical identities, performing extra operations and/or using the circular or linear CORDIC algorithms [3].

2.2 Basic Range of Convergence

The basic range of convergence, obtained by a method developed by Hu[2] states the following:

Rotation Mode: $|Z_0| \le \theta_N + \sum_{i=1}^{N}\theta_i$ (7)

➔ $|Z_0| \le Tanh^{-1}\left(2^{-N}\right) + \sum_{i=1}^{N} Tanh^{-1}\left(2^{-i}\right)$ (8)

➔ $|Z_0|_{\max} \approx 1.182,\ for\ N \to \infty$ (9)

This is the restriction imposed to the domain of the input argument of the hyperbolic functions in the rotation mode. Note that the domain of the functions *Sinh* and *Cosh* is $\langle -\infty, +\infty \rangle$.

Vectoring Mode: $\left|Tanh^{-1}\left(Y_0 / X_0\right)\right| \le \theta_N + \sum_{i=1}^{N}\theta_i$ (10)

➔ $\left|Tanh^{-1}\left(Y_0 / X_0\right)\right| \le 1.1182,\ for\ N \to \infty$ (11)

➔ $\left|Y_0 / X_0\right|_{max} \approx 0.80694,\ for\ N \to \infty$ (12)

This is the limitation imposed to the domain of the quotient of the input arguments of the hyperbolic functions in the vectoring mode. Note that the domain of *Tanh*$^{-1}$ is $\langle -1, +1 \rangle$, and thus this function remains greatly limited in its domain.

2.3 Expansion of the Range of Convergence

The convergence range described by (9) and (12) is unsuitable to satisfy all applications of the hyperbolic CORDIC algorithm.

One strategy to address the problem of limited convergence is the use of mathematical identities to preprocess the CORDIC input quantities[1]. However, a different preprocessing scheme is necessary for each function, making it very difficult to have a unified hyperbolic CORDIC hardware. Moreover, the preprocessing leads to a significant increase in processing time and hardware.

Hu et al [2] have proposed another scheme to address the problem of the range of convergence. The approach consists in the inclusion of additional iterations to the basic CORDIC algorithm. As it will be shown in Section 3, the hardware and processing time increase is bearable and suitable for VLSI and FPGA implementation.

The method proposed by Hu consists in the inclusion of additional iterations for negative indexes *i*:

$$\theta_i = Tanh^{-1}\left(1 - 2^{i-2}\right),\ for\ i \le 0 \quad (13)$$

Therefore, the modified algorithm results:

$$For\ i \le 0$$
$$X_{i+1} = X_i + \delta_i\left(1 - 2^{i-2}\right)Y_i$$
$$Y_{i+1} = Y_i + \delta_i\left(1 - 2^{i-2}\right)X_i \quad (14)$$
$$Z_{i+1} = Z_i - \delta_i Tanh^{-1}\left(1 - 2^{i-2}\right)$$

$$For\ i > 0$$
$$X_{i+1} = X_i + \delta_i Y_i 2^{-i}$$
$$Y_{i+1} = Y_i + \delta_i X_i 2^{-i} \quad (15)$$
$$Z_{i+1} = Z_i - \delta_i Tanh^{-1}\left(2^{-i}\right)$$

The trend of the results for the rotation and vectoring mode is the same as that stated in (4) and (5). The value of δ_i is the same as indicated in (3). But the quantity A_n, described in (6), must be redefined as follows:

$$A_n \leftarrow \left[\prod_{i=-M}^{0}\sqrt{1-\left(1-2^{(i-2)}\right)^2}\right]\left[\prod_{i=1}^{N}\sqrt{1-2^{-2i}}\right] \quad (16)$$

The range of convergence, stated in (7) and (10) for the basic hyperbolic CORDIC algorithm, now becomes:

Rotation Mode: $|Z_0| \le \theta_{\max}$ (17)

Vectoring Mode: $\left|Tanh^{-1}\left(Y_0 / X_0\right)\right| \le \theta_{\max}$ (18)

Where: $\theta_{\max} = \sum_{i=-M}^{0} Tanh^{-1}\left(1 - 2^{i-2}\right) +$

$$+\left[Tanh^{-1}\left(2^{-N}\right) + \sum_{i=1}^{N} Tanh^{-1}\left(2^{-i}\right)\right] \quad (19)$$

Although (17) and (18) look nearly the same, they are interpreted differently: (17) states the maximum input angle the user can enter to obtain a valid result, whereas (18) states the maximum value attainable for the *Tanh*$^{-1}$ function to which Z-Z_0 tends (according to (5)). If Z_0=0, (18) states the maximum value attainable at Z, and therefore imposes a limitation to the inputs X_0 and Y_0.

The values for θ_{max} have been tabulated for M between 0 an 10 and are shown in Table 1.

M	θ_{max} from (19)	M	θ_{max} from (19)
0	2.09113	5	12.42644
1	3.44515	6	15.54462
2	5.16215	7	19.00987
3	7.23371	8	22.82194
4	9.65581	9	26.98070
		10	31.48609

Table 1. θ_{max} versus M for the Modified Hyperbolic CORDIC algorithm (after Hu[2])

For example, if M = 5 is chosen (six additional iterations), then $\theta_{max}=12.42644$, and the domain of the functions *Cosh* and *Sinh* is greatly expanded to [-12.42644,+12.42644] compared with the domain in (9). Similarly, the range of the function $Tanh^{-1}$ is increased to [-12.42644,+12.42644], which means that the domain of the quotient Y_0/X_0 becomes nearly $\langle -1,+1 \rangle$, which is the entire domain of $Tanh^{-1}$.

From the last example, it is clear that the expansion scheme does work. The more domain of the functions is needed, the more the iterations *(M+1)* that must be executed.

3. ARCHITECTURES PROPOSED FOR THE EXPANDED HYPERBOLIC CORDIC ALGORITHM

The architectures presented here implement the expanded hyperbolic CORDIC algorithm described in (14) and (15). The architectures are such that the inputs and outputs have an identical bit width. The intermediate registers and operators can be of higher bit width due to particular details of the algorithm and precision considerations which will be explored later in this paper. As it will be shown in Section 4, the bit width of the intermediate registers, the fixed-point format of the inputs and outputs, and the number of iterations vary considerably with the input/output bit width and the particular function desired. That is, to obtain an optimum architecture which yields a particular hyperbolic function (e.g. $Tanh^{-1}$, *Sinh/Cosh, exp*), the architecture has to be changed for each function and for each input/output bit width. In Section 4, we explore the particularities of each architecture for $Tanh^{-1}$, *Sinh, Cosh,* and *exp*.

It is worth to note, however, that a unified hyperbolic CORDIC hardware, capable of obtaining all the functions within the same architecture, is desirable for certain applications, as has been shown in [5]. The same principle which will be applied to the analysis of $Tanh^{-1}$, *Sinh, Cosh* and *exp* can be applied to this case and thus the optimum architecture can be attained.

In addition, there exists a precision consideration which extends the bit width: it is a 'rule of thumb' found in [5]: "If *n* bits is the desired output precision, the internal registers should have $log_2(n)$ additional guard bits at the LSB position". This consideration, although arbitrary, have proved to work very well. With these considerations in mind, three fixed-point architectures are presented: a low cost iterative version, a fully pipelined version, and a bit serial iterative version. But first, it is necessary to define some nomenclature used:

- *n*: input/output bit width
- *nr*: bit width of the internal registers and operators
- *ng*: additional guard bits. $ng = log_2(n)$
- *N*: number of basic iterations
- *M*: number of additional iterations minus one.

Note that $nr \geq ng + n$. We define the quantity $na = nr - (ng+n)$ as the additional bits that are added to the MSB part, which will be necessary as we will demonstrate in Section 4.

3.1 Low-Cost Iterative Architecture

Fig. 1 depicts the architecture that implements the equations (14) and (15) in an iterative fashion. The two LUTs (look-up tables) are needed to store the two sets of elementary angles defined in equations (2) and (13).

The process begins when a start signal is asserted. After '$M+1+N+v$' clock cycles ('v' is the number of repeated iterations stated in Section 2.1), the result is obtained in the registers *X, Y* and *Z*, and a new process can be started.

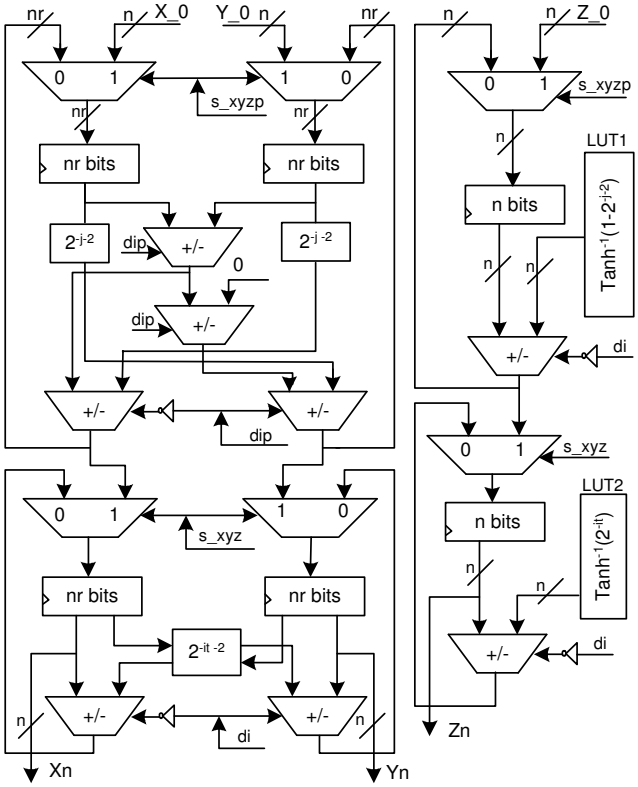

Figure 1

Inputs: X_0, Y_0, and Z_0
Outputs: X_N, Y_N, and Z_N
j = M → 0 it = 1 → N

There are two stages: One that implements the iterations for $i \leq 0$ and is depicted in the upper part, it needs two multiplexers, two registers, four adders and two barrel shifters. This is the most critical part of the design, and introduces considerable delay, thus reducing the frequency of operation. The lower part of Figure 1 implements the iterations for $i > 0$, this is a classical hardware found in many textbooks and papers.

A state machine controls the load of the registers, the data that passes onto the multiplexers, the add/substract decision of the adder/substracters, and the count given to the barrel shifters.

3.2 Fully Pipelined Architecture

To develop the architecture, the algorithm described in (14) and (15) is unfolded. In addition, the stages that implement the expansion (the upper part of Figure 2) need to be partitioned in order to avoid large delays. Therefore '2*(M+1) + N + v' stages

will appear ('*v*' is the number of repeated iterations as stated in Section 2.1). The architecture is depicted in Figure 2.

Such architecture can obtain a new result each cycle. The initial latency is '2*(M+1) + N + v' cycles.

At each stage, *X* and *Y* have a fixed shift that can be implemented in the wiring, thus removing the barrel shifters of Section 3.1. In addition, the look-up values for the θ_i are distributed as constants across the stages, which are hardwired, hence removing the look-up table. The entire hardware is reduced to an array of interconnected adder/substractors and registers. A little additional hardware in needed to obtain the '*dix*' signals, which are obtained as indicated in (3).

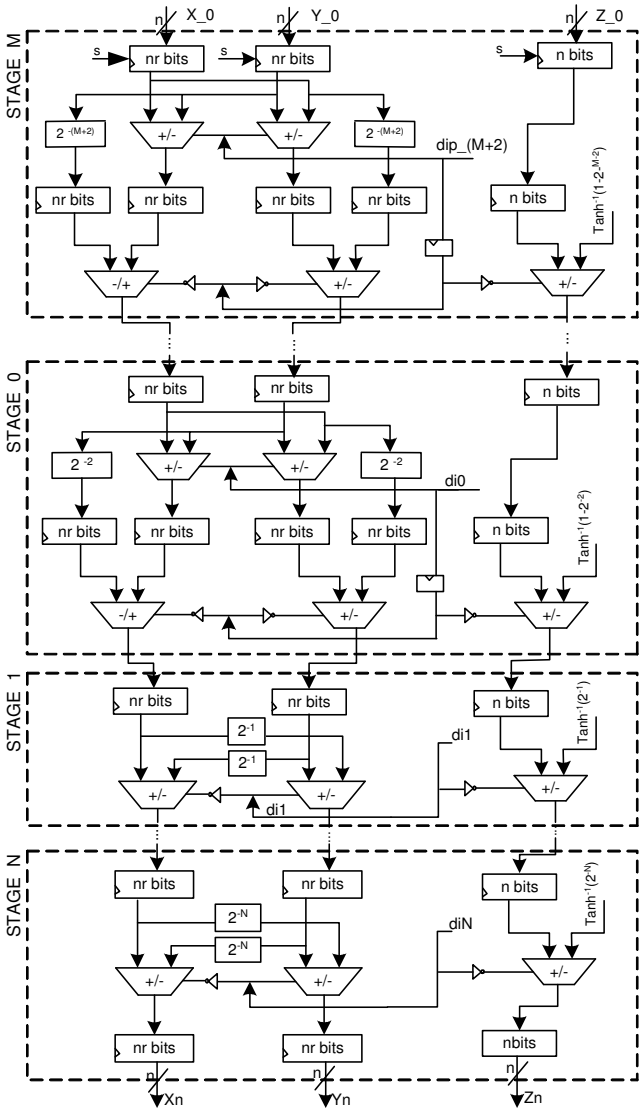

Figure 2. Pipelined-CORDIC

3.3 Bit serial iterative architecture

The simplified interconnect and logic in a bit serial design should allow it to work at a much higher frequency than other architectures. However, the design needs to be clocked *'n'* times for each iteration (*'n'* is the width of the data). This architecture maps well in FPGA [4] and is depicted in Figure 3.

The input data (*X_0, Y_0* and *Z_0)* is loaded into the register bit per bit. Then the calculation starts. The array of serial adders/substractors, multiplexers, flip flops are arranged in a special fashion and controlled by a state machine, so that one output bit is computed every cycle, and after '*n* + *n*(M+1+N+v)*' cycles a new result is obtained in the registers. * '*v*' is the number of repeated iterations as stated in Section 2.1.

4. ANALYSIS OF NUMERICAL FORMATS FOR EACH HYPERBOLIC FUNCTION

For our fixed-point hardware, four standard bit widths for the inputs/ outputs are explored: 12, 16, 24, and 32.

Our aim is to obtain an optimum architecture that implements just one hyperbolic function. As a result, we will restrict our analysis to each architecture that implements one of the following functions: $Tanh^{-1}$, *Cosh, Sinh,* and *exp,* which can be directly obtained from the hyperbolic CORDIC equations in (14) and (15). In case a unified hyperbolic CORDIC hardware is desired, the same analysis can be applied to obtain the optimum architecture. We will show that the intermediate registers and operators need to be augmented in the MSB part and that the format for *X, Y, Z,* and the number of iterations varies for each architecture that implements a particular hyperbolic function. The least significant positions are always extended for $log_2(n)$ bits, where *n* is the input/output bit width. In the following sub-sections we will calculate the internal datapath, **but this value will not consider the guard bits ($log_2(n)$)**, because it is always present and to avoid complicating the explanation.

The numerical format is defined as: [T D]

Where T: total number of bits
D: total number of fractional bits

4.1 Inverse Hyperbolic Tangent ($Tanh^{-1}$)

To obtain this function in *Z*, we have to set $Z_0=0$ and $X_0=1$ in the vectoring mode. Then, $Z_N \leftarrow Tanh^{-1}(Y_0)$.

As the domain of $Tanh^{-1}$ is $\langle -1,+1 \rangle$, the input Y_0 is restricted to 1 integer bit in the 2's complement fractional fixed-point representation ($|Y_0|<1$). But, as the input $X_0=1$ requires 2 integer bits for correct representation, and the format for *X* and *Y* must be the same, then *X* and *Y* must have 2 integer bits.

The critical case occurs when Y_0 is at its maximum value, from which the maximum value of Z_N is obtained. Then we use Table 1 to find the number of additional iterations (*M*) needed to correctly represent Z_N (by locating the nearest θ_{max}) .

It is needless to add more bits to *X* and *Y*, because *X* and *Y* tend to decrease as shown in (5). At each different format (12, 16, 24 and 32) we will obtain an adequate format for the bits of *Z.*

4.1.1 Input/Output bit width: 12. Format for X and Y:[12 10]

$|Y_0|_{max} = 3FFh = 0.999023475$

$\rightarrow Z_{N\ max} = Tanh^{-1}(3FFh) = 3.812065$

Given $Z_{N\ max}$, we need 3 integer bits to represent *Z.* And Table 1 specifies that 3 additional iterations are needed (M=2, $\theta_{max}=5.162$).

However, in this case, the maximum intermediate value for *Z* is *4.04*, then 1 bit must be extended to the MSB. This change will be implemented in the internal architecture. Thus, the format for *Z* remains [12 9] and the internal datapath is 13 bits. With the *Z* format [12 9] the LUT's angles are:

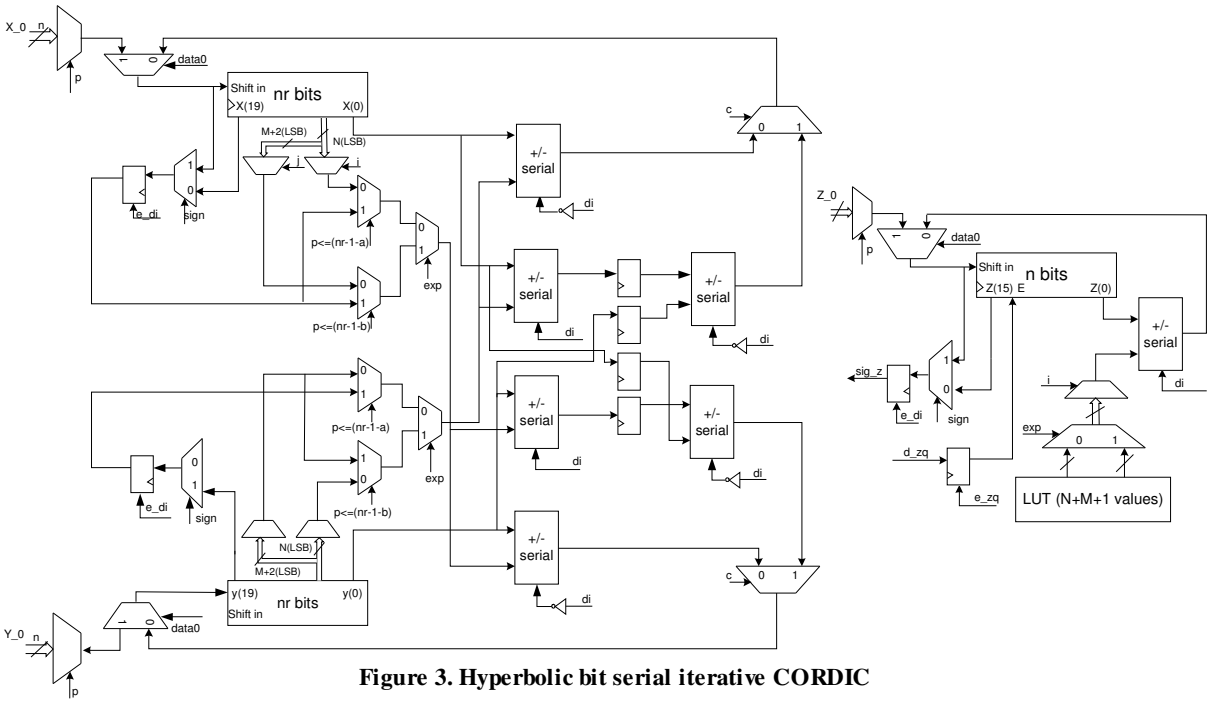

Figure 3. Hyperbolic bit serial iterative CORDIC

M	Value	N	Value	N	Value
-2	36F	1	119	6	008
-1	2B5	2	083	7	004
0	1F2	3	040	8	002
		4	020	9	001
		5	010		

Table 2

Table 2 shows that the number of iterations needed is 9. Any further iteration will yield a value less or equal than 001h for the fixed angle rotation, which is useless.

In conclusion, M=2 and N=9. *Z* format is [12 9], and the internal datapath for *Z* is 13.

4.1.2 Input/Output bit width: 16. Format for X and Y: [16 14]

$|Y_0|_{max} = 3FFFh = 0.99993896484375$

$\rightarrow Z_{N\,max} = Tanh^{-1}(3FFFh) = 5.1985885952$

Table 1 indicates that 4 additional iterations are needed (M=3, $\theta_{max}=7.23$). Given the maximum Z_N, we found that 4 integer bits are needed to represent *Z*.

And, as $\theta_{max}=7.23$ needs 4 integer bits, no bit will be extended. Thus, the format for *Z* remains [16 12] and the internal datapath is 16 bits. With the *Z* format [16 12] the LUT's angles are:

M	Value	N	Value	N	Value
-3	2125	1	08CA	7	0020
-2	1B79	2	0416	8	0010
-1	15AA	3	0203	9	0008
0	0F91	4	0100	10	0004
		5	0080	11	0002
		6	0040	12	0001

Table 3

Table 3 shows that the number of iterations needed is 12. Any further iteration will yield a value minor or equal than 001h for the fixed angle rotation, which is useless.

In conclusion, M = 3 and N = 12. *Z* format is [16 12], and the internal datapath for *Z* is 16.

4.1.3 Input/Output bit width: 24. Format for X and Y: [24 22]

$|Y_0|_{max} = 3FFFFFh$

$\rightarrow Z_{N\,max} = Tanh^{-1}(3FFFFFh) = 7.9711925$

Table 1 specifies that 5 additional iterations are needed (M=4, $\theta_{max}=9.655$). Given the maximum Z_N, we found that 4 integer bits are needed to represent *Z*.

However, in this case, the maximum intermediate value for *Z* is *8.5346*, so we have to extend 1 bit to the MSB. This change will be implemented in the internal architecture. Thus, the format for *Z* remains [24 20] and the internal datapath is 25 bits.

With the *Z* format [24 20] the LUT's angles are:

M	Value	N	Value	N	Value
-4	26C0E5	1	0F9139	9	000800
-3	212524	2	08C9F5	10	000400
-2	1B78CE	3	04162C	11	000200
-1	15AA16	4	0202B1	12	000100
0	0F9139	5	010056	13	000080
		6	00800B	14	000040
		7	002000	15	000020
		8	001000	16	000010

Table 4

We have chosen 16 as the number of iterations. While 20 iterations can be executed, it would increase the amount of hardware excessively.

In conclusion, M=4 and N=16. *Z* format is [24 20], and the internal datapath for *Z* is 25.

4.1.4 Input/Output bit width: 32. Format for X and Y: [32 30]

$|Y_0|_{max} = 3FFFFFFFh$

$\rightarrow Z_{N\,max} = Tanh^{-1}(3FFFFFFFh) = 10.743781$

Table 1 specifies that 6 additional iterations are needed (M=5, $\theta_{max}=12.42644$). Given the maximum Z_N, 5 integer bits are needed to represent *Z*.

And, as $\theta_{max}=12.42644$ requires 5 bits, no bit is needed to be extended. Thus, the format for *Z* remains [32 27] and the internal

datapath is 32 bits. With the *Z* format [32 27] the LUT's angles are:

M	Value	N	Value	N	Value
-5	162A40FE	1	0464FA9F	9	00040000
-4	13607294	2	020B15DF	10	00020000
-3	109291E9	3	01015892	11	00010000
-2	0DBC6724	4	00802AC4	12	00008000
-1	0AD50B1D	5	00400556	13	00004000
0	07C89CAC	6	002000AB	14	00002000
		7	00100015	15	00001000
		8	00080003	16	00000800

Table 5

We have chosen the number of iterations to be 16. While 27 iterations can be executed, it would increase the amount of hardware excessively.

In conclusion, M=5 and N=16. *Z* format is [32 27], and the internal datapath for *Z* is 32.

4.2 Hyperbolic Sine and Hyperbolic Cosine

To obtain these functions in *X* and *Y*, it is necessary to set $Y_0=0$ and $X_0=1/A_n$ in the rotation mode. Then, $Y_N \leftarrow Sinh(Z_0)$ and $X_N \leftarrow Cosh(Z_0)$. As the domain of *Sinh* and *Cosh* is $\langle -\infty,+\infty \rangle$, there is no input restriction. Our strategy will consist in fixing to [12 10] the input *Z* for the bit width of 12, and increment 1 integer bit for a larger bit width, so that by augmenting the bit width, the range of the functions *Sinh and Cosh* is incremented.

The critical case occurs when $|Z_0|$ is at the maximum value attainable at each format, from which the maximum values of X_N and Y_N are obtained. Then we use Table 1 to find the number of additional iterations (*M*) needed to correctly represent Z_0 (by locating the nearest θ_{max}).

Note that it is unnecessary to add more bits to *Z*, because *Z* tends to 0 as shown in (4). At each different format (12, 16, 24 and 32) we will obtain an adequate format for the bits of *X* and *Y*.

4.2.1 Input/Output bit width: 12. Input format for Z: [12 10]

$|Z_0|_{max} = |400h| = |-2|$

Table 1 shows that 1 additional iteration is needed (M=0). With the *Z* format [12 10] the LUT's angles are:

M	Value	N	Value	N	Value
0	3E4	1	232	6	010
		2	106	7	008
		3	081	8	004
		4	040	9	002
		5	020	10	001

Table 6

Table 6 shows that the number of iterations needed is 10. Any further iteration will yield a value less or equal than 001h for the fixed angle rotation, which is useless. Then, $A_n=0.54777601990563$ and $X_0=1/A_n=1.82556366774193$.

Also, the maximum values of X_N and Y_N, given $|Z_0|_{max}$ are:

$X_N = Sinh(400h) = -3.62686040784702$

$Y_N = Cosh(400h) = +3.76219569108363$

Given these maximum values, we found that 3 integer bits are needed to represent *X* and *Y*. In addition, in this case, the maximum intermediate values for *X* is *2.510150043145* and for *Y* is *2.281954584677*. So, no bit need to be extended. Thus, format for X and Y remains [12 9] and the internal datapath is 12 bits.

Note that the format for *Z*([12 10]) has been chosen arbitrarily. However, this is a good format, because with [12 9] for *Z* we would have obtained [12 6] for *X* and *Y*, where many fractional bits would have been lost.

In conclusion, M=0 and N=10. X, Y format is [12 9], and the internal datapath for *X, Y* is 12.

4.2.2 Input/Output bit width: 16. Input format for Z: [16 13]

$|Z_0|_{max} = |8000h| = |-4|$

Table 1 indicates that 3 additional iterations are needed (M=2). With the Z format [16 13] the LUT's angles are:

M	Value	N	Value	N	Value
-2	36F2	1	1194	7	0040
-1	2B54	2	082C	8	0020
0	1F22	3	0405	9	0010
		4	0201	10	0008
		5	0100	11	0004
		6	0080	12	0002
				13	0001

Table 7

Table 7 shows that 13 iterations are needed. Any further iteration will yield a value less or equal than 001h for the fixed angle rotation, which is useless. Then, $A_n = 0.09228252133203$ *and* $X_0 = 1/A_n = 10.83628823276322$.

Also, the maximum values of X_N and Y_N, given $|Z_0|_{max}$ are:

$X_N = Sinh(8000h) = -27.289917197$

$Y_N = Cosh(8000h) = +27.3082328361$

These maximum values indicate that 6 integer bits are needed to represent *X* and *Y*. In addition, in this case, the maximum intermediate values for *X* is *34.45601024011* and for *Y* is *-34.4348456146*, so we have to extend 1 bit to the MSB. This change will be implemented in the internal architecture. Thus, the format for *X* and *Y* remains [16 10] and the internal datapath is 17 bits.

Note that the format for *Z*([16 13]) has been chosen arbitrarily. However, this is a good format, because with [16 12] for *Z* we would have obtained [16 4] for X and Y, where many fractional bits would have been lost.

In conclusion, M=2 and N=13. X, Y format is [16 10], and the internal datapath for X, Y is 17.

4.2.3 Input/Output bit width: 24. Input format for Z: [24 20]

$|Z_0|_{max}=|800000h|=|-8|$

From Table 1, we found that 5 additional iterations are needed (M=4). With the *Z* format [24 20] the elementary angles defined for the LUT (look-up table) are equal as those in Table 4.

From Table 4, the number of iterations selected is 16. While 20 iterations can be executed, it would increase the amount of hardware excessively. Then, $A_n=4.0305251 \times 10^{-3}$ and $X_0=1/A_n=248.1066$.

Also, the maximum values of X_N and Y_N, given $|Z_0|_{max}$ are:

$X_N = Sinh(800000h) = -1490.47882$

$Y_N = Cosh(800000h) = +1490.47916$

Given these maximum values, we found that 12 integer bits are needed to represent correctly X and Y. In addition, in this case, the maximum intermediate values for *X* is *3081.0854* and for *Y* is *3081.085173*, so we have to extend 1 bit to the MSB. This change will be implemented in the internal architecture. Thus, the format for X and Y remains [24 12] and the internal datapath is 25 bits.

Note that the format for *Z*([24 20]) has been chosen arbitrarily. However, this is a good format, because with [24 19] for Z we

would have obtained [24 0] for X and Y, and the fractional bits would have disappeared.

In conclusion, M=4 and N=16. X, Y format is [24 12], and the internal datapath for *X, Y* is 25.

4.2.4 Input/Output bit width: 32. Input format for Z: [32 27]

$$|Z_0|_{max} = |80000000h| = |-16|$$

Table 1 indicates that 8 additional iterations are needed (M=7). With the Z format [32 27] the elementary angles defined for the LUT (look-up table) are equal as those in Table 5.

From Table 5, the number of iterations selected is 16. While 27 iterations can be executed, it would increase the amount of hardware excessively. Then, $A_n = 2.7737\text{x}10^{-6}$ and $X_0 = 1/A_n = 3.605287519x10^{-5}$.

Also, the maximum values of X_N and Y_N, given $|Z_0|_{max}$ are:

$$X_N = Sinh(80000000h) = 4.44305526x10^{6}$$
$$Y_N = Cosh(80000000h) = 4.44305526x10^{6}$$

Given these maximum values, we found that 24 integer bits are needed to represent *X* and *Y*. In addition, in this case, the maximum intermediate value for *X* is $20.32x10^{6}$ and for *Y* is $20.32x10^{6}$, so we have to extend 2 bits to the MSB. This change will be implemented in the internal architecture. Thus, the format for *X* and *Y* remains [32 8] and the internal datapath is 34 bits.

Note that the format for *Z*([32 27]) has been chosen arbitrarily. But this is a good format, because with [32 26], we would have needed more than 32 integer bits for X and Y, that is impossible to implement.

In conclusion, M=7 and N=16. *X, Y* format is [32 8], and the internal datapath for *X, Y* is 34.

4.3 Exponential (e^x)

To obtain this function, we have to set X_0=Y_0=$1/A_n$ in the rotation mode. Then, $Y_N \leftarrow Sinh(Z_0) + Cosh(Z_0)$ and $X_N \leftarrow Cosh(Z_0) + Sinh(Z_0)$. And, as $e^w = Sinh(w) + Cosh(w)$, we can rewrite: $Y_N \leftarrow e^{Z_0}$ and $X_N \leftarrow e^{Z_0}$.

The domain of e^w is $\langle -\infty, +\infty \rangle$, hence there is no input restriction. Our strategy will consist in fixing to [12 10] the input *Z* for the bit width of 12, and incrementing 1 integer bit for each larger bit width, so that by augmenting the bit width, the range of the function e^w is incremented. It is worth to mention that the hardware will be the same of the hardware that computes *Sinh* and *Cosh*.

The critical case occurs when Z_0 is at the maximum value attainable at each format, from which the maximum values of X_N and Y_N are obtained. Then we use Table 1 to find the number of additional iterations (*M*) needed to correctly represent Z_0 (by locating the nearest θ_{max}).

As *Z* tends to 0 (as shown in (4)), it is needless to add more bits to *Z*. At each different format (12, 16, 24 and 32) we will obtain an adequate format for the bits of *X* and *Y*.

4.3.1 Input/Output bit width: 12. Input format for Z: [12 10]

$$Z_{0max} = 7FFh = 1.9990234375.$$

Table 1 indicates that 1 additional iteration is needed (M=0). With the Z format [12 10] the LUT's angles are those shown in Table 6, which specifies that 10 iterations are needed. Any further iteration will yield a value less or equal than 001h for the fixed angle rotation, which is useless. Then, $A_n=0.54777601990563$ and $X_0=Y_0=1/A_n=1.82556366774193$.

Also, the maximum values of X_N and Y_N, given $Z_{0\,max}$ are:

$$X_N = Y_N = e^{7FFh} = 7.38184374606390$$

Given this maximum value, we found that 4 integer bits are needed to correctly represent *X* and *Y*.

In addition, in this case, the maximum intermediate value for *X and Y* is *7.607583091*. So, no bit need to be extended. Thus, *X* and *Y*'s format remains [12 8] and the internal datapath is 12 bits.

Note that the format for *Z*([12 10]) has been chosen arbitrarily. But this is a good format, since with [12 9] for *Z* we would have obtained [12 5] for *X* and *Y*, and many fractional bits would have been lost.

In conclusion, M=0 and N=10. *X, Y* format is [12 8], and the internal datapath for *X, Y* is 12.

4.3.2 Input/Output bit width: 16. Input format for Z: [16 13]

$$Z_{0\,max} = 7FFFh = 3.9998779296875.$$

Table 1 specifies that 3 additional iterations are needed (M=2). With the *Z* format [16 13] the LUT's angles are those shown in Table 7, which specifies that 13 iterations are needed. Any further iteration will yield a value less or equal than *001h* for the fixed angle rotation, which is useless. Then, $A_n = 0.09228252133203$ *and* $X_0 = 1/A_n = 10.83628823276322$.

Also, the maximum values of X_N and Y_N, given $|Z_0|_{max}$ are:

$$X_N = Y_N = e^{7FFFh} = 54.59148562667914$$

Given this maximum value, we found that 7 integer bits are needed to correctly represent *X* and *Y*.

Besides, in this extreme case, the maximum intermediate value for *X* and *Y* is *68.89085585*, so we have to extend 1 bit to the MSB. This change will be implemented in the internal architecture. Thus, the format for *X* and *Y* remains [16 9] and the internal datapath is 17 bits.

Note that the format for *Z*([16 13]) has been chosen arbitrarily. However, this is a good format, because with [16 12] for Z we would have obtained [16 3] for *X* and *Y*, and many fractional bits would have been lost.

In conclusion, M=2 and N=13. X, Y format is [16 9], and the internal datapath for *X, Y* is 17.

4.3.3 Input/Output bit width: 24. Input format for Z: [24 20]

$$Z_{0max} = 7FFFFFh = 7.99999904632568.$$

Table 1 indicates that 5 additional iterations are needed (M=4). With the *Z* format [24 20] the elementary angles defined for the LUT (look-up table) are equal as those in Table 4.

From Table 4, the number of iterations selected is 16. While 20 iterations can be executed, it would increase the amount of hardware excessively. Then, A_n=4.0305251x10^{-3} and $X_0=Y_0=1/A_n=248.1066$.

Also, the maximum values of X_N and Y_N, given $|Z_0|_{max}$ are:

$$X_N = Y_N = e^{7FFFFFh} = 2980.955144180013$$

Given these maximum values, we found that 13 integer bits are needed to correctly represent *X* and *Y*.

In addition, in this case, the maximum intermediate value for *X* and *Y* is *6162.17058*, so we have to extend 1 bit to the MSB. This will be implemented in the internal architecture. Thus, the format for X and Y remains [24 11] and the internal datapath is 25 bits.

Note that the format for *Z*([24 20]) has been chosen arbitrarily. However, this is a good format, because with [24 19] for *Z* we would have needed 25 integer bits for *X* and *Y*, which would be impossible to implement.

In conclusion, M=4 and N=16. X, Y format is [24 11], and the internal datapath for *X, Y* is 25.

4.3.4 Input/Output bit width: 32. Input format for Z: [32 27]

$Z_{0\ max}$ = 7FFFFFFFh = 15.99999999254942.

Table 1 specifies that 8 additional iterations are needed (M=7). With the Z format [32 27] the elementary angles defined for the LUT (look-up table) are equal as those in Table 5. From Table 5, the number of iterations selected is 16. While 27 iterations can be executed, it would increase the amount of hardware excessively. Then, $A_n = 2.7737 \times 10^{-6}$ and $X_0 = Y_0 = 1/A_n = 3.605287519 \times 10^{-5}$.

Also, the maximum values of X_N and Y_N, given $|Z_0|_{max}$ are:

$$X_N = Y_N = e^{7FFFFFFF} = 8.886110454 \times 10^{6}$$

Given these maximum values, 25 integer bits are needed to represent X and Y.

In addition, in this case, the maximum intermediate value for *X* and *Y* is 4.06×10^{7}, so we have to extend 2 bits to the MSB. This will be implemented in the internal architecture. Thus, the format for *X* and *Y* remains [32 7] and the internal datapath is 34 bits.

Note that the format for *Z*([32 27]) has been chosen arbitrarily. But this is a good format, since with [32 26], we would have needed more than 32 integer bits for X and Y, that is impossible to implement.

In conclusion, M=7 and N=16. *X, Y* format is [32 7], and the internal datapath for *X, Y* is 34.

4.4 Results of FPGA implementation

Note that the hardware for obtaining *exp and Sinh/Cosh* is exactly the same, though the results are interpreted differently.

Type	N	Function	LEs	f_{max}
Iterative (Folded Recursive)	12	*Sinh/Cosh, exp*	402	97.53
		Tanh⁻¹	433	108.75
	16	*Sinh/Cosh, exp*	544	85.73
		Tanh⁻¹	516	99.70
	24	*Sinh/Cosh, exp*	866	77.50
		Tanh⁻¹	869	84.74
	32	*Sinh/Cosh, exp*	1170	83.03
		Tanh⁻¹	1119	87.92
Fully Pipelined (Unfolded Pipelined)	12	*Sinh/Cosh, exp*	662	182.22
		Tanh⁻¹	675	187.18
	16	*Sinh/Cosh, exp*	1308	168.75
		Tanh⁻¹	1396	177.34
	24	*Sinh/Cosh, exp*	1512	160.35
		Tanh⁻¹	1527	171.15
	32	*Sinh/Cosh, exp*	1686	152.35
		Tanh⁻¹	1718	165.40
Bit Serial Iterative (Folded recursive)	12	*Sinh/Cosh, exp*	225	206.10
		Tanh⁻¹	234	208.77
	16	*Sinh/Cosh, exp*	345	192.74
		Tanh⁻¹	348	193.25
	24	*Sinh/Cosh, exp*	469	182.50
		Tanh⁻¹	478	188.62
	32	*Sinh/Cosh, exp*	703	187.40
		Tanh⁻¹	710	191.50

Table 8. Final Results
Device: Stratix EP1S10F484C5

The results, obtained with *Quartus II 5.0*, show that the hyperbolic CORDIC implementation is amenable to FPGA. The clock rates are relatively high and the resource effort is bearable for high-density FPGAs.

5. ERROR ANALYSIS

For the cases analyzed in 4.1, 4.2 and 4.3, an error analysis is performed. The results are contrasted with the ideal values obtained in MATLAB®. The error measure will be:

$$Relative\ Error = \left| \frac{ideal\ value - CORDIC\ value}{ideal\ value} \right| \qquad (20)$$

The three cases will be tested. We have taken 1024 values equally spaced along the maximum domain of functions obtained for each bit width analyzed. In the case of $Tanh^{-1}$, it has been necessary to add more values, for the $Tanh^{-1}$ grows dramatically as its argument nears ± 1.

5.1 Inverse Hyperbolic Tangent

We will show the relative error for the hardware that implements the hyperbolic tangent in its entire domain for 12, 16, 24 and 32 bits. Figures 7, 8, and 9 show the relative error performance for the function $Tanh^{-1}(w)$ for 12, 16, 24 and 32 bits. Although the domain of $Tanh^{-1}$ is $\langle -1,+1 \rangle$, we have just plotted for $w \in [0,+1\rangle$ since $Tanh^{-1}$ is an odd function.

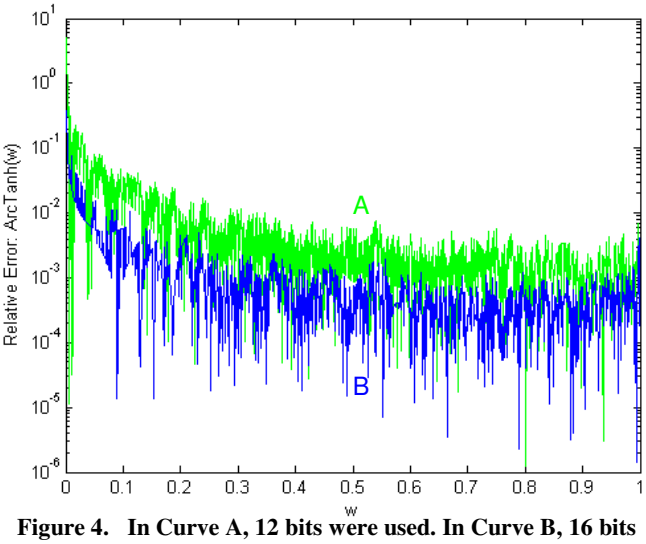

Figure 4. In Curve A, 12 bits were used. In Curve B, 16 bits were used.

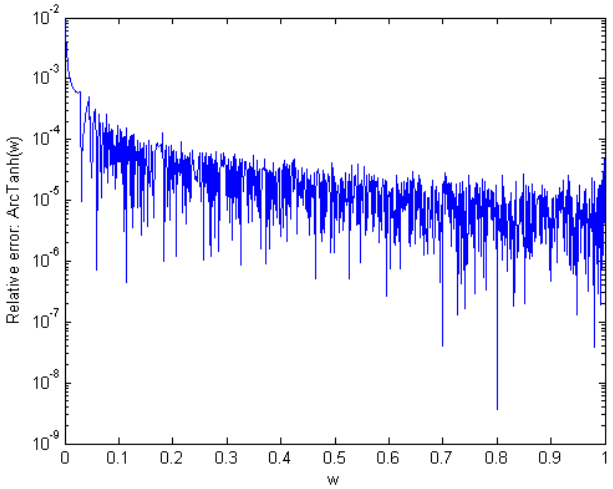

Figure 5. In the curve, 24 bits were used.

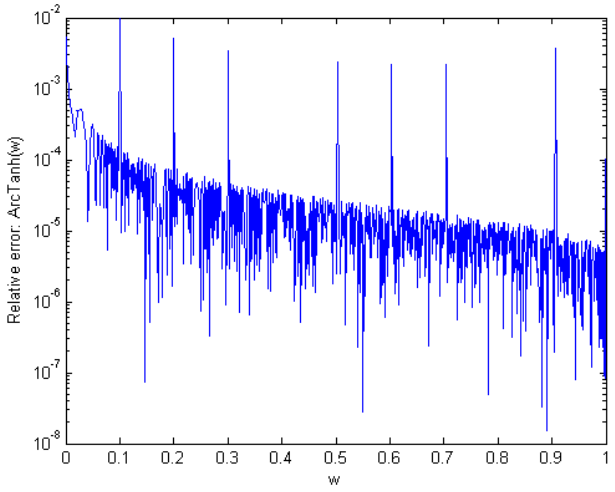

Figure 6. In the curve, 32 bits were used.

For *w* near *0*, all the curves exhibit high relative error values, because $Tanh^{-1}(w)$ yields the smallest values for *w* near *0*, and the fixed-point hardware fails representing those small values.

The more the bit width, the less the relative error. For example, for 12 bits (figure 4) nearly all the relative error values are below 10^{-2} (an error below 1%), and for 24 bits (Figure 5), the relative error values are below 10^{-4} (an error below 0.1%).

The curve for 32 bits (Figure 6) exhibits some irregularities due to the reduced basic iterations (16); but in general it provides the least relative error. However, it is unusual to have a $Tanh^{-1}$ hardware with an bit input data width of 32 bits.

5.2 Hyperbolic Sine and Hyperbolic Cosine

We will show the relative error for the hardware that implements *Sinh* and *Cosh* in the maximum domain obtained in 4.2 for 12, 16, 24 and 32 bits.

Figures 7 and 8 show the relative error for *Cosh(w)* for 12, 16, 24 and 32 bits. We have just plotted the positive domain. The negative domain is not plotted, since *Cosh* is an even function and will yield the same values.

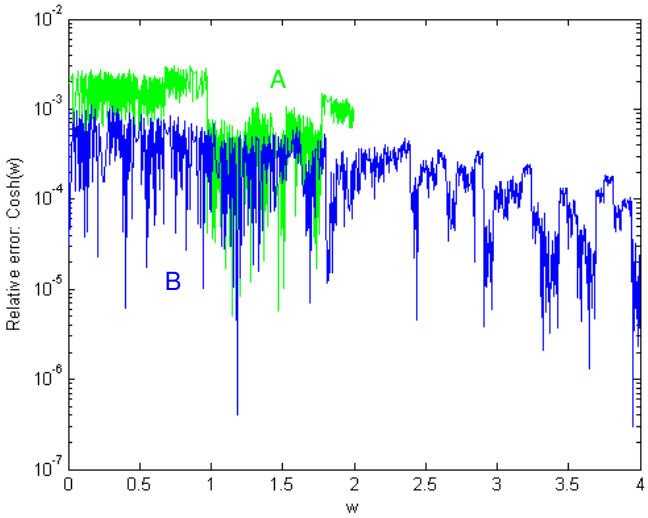

Figure 7. In Curve A, 12 bits were used ($w \in [0,2\rangle$). In Curve B, 16 bits were used ($w \in [0,4\rangle$).

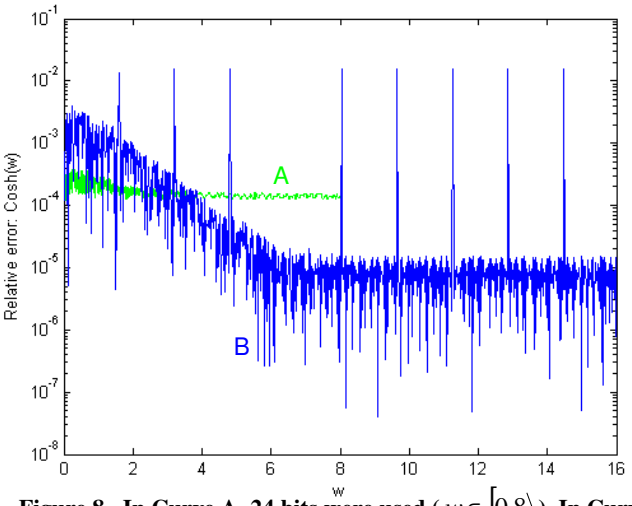

Figure 8. In Curve A, 24 bits were used ($w \in [0,8\rangle$). In Curve B, 32 bits were used ($w \in [0,16\rangle$).

The curve A (Figure 8) for 24 bits is very regular because in this format we use a larger quantity of fractional bits than with the other formats.

The curve B (Figure 8) for 32 bits exhibits some irregularities due to the reduced fractional bits (8) and the reduced number of basic iterations (16). However, it provides the greatest domain for the *Cosh(w)* function ($w \in [-16,16\rangle$).

Figures 9 and 10 show the relative error performance for *Sinh(w)* for 12, 16, 24 and 32 bits. We have just plotted the positive domain. The negative domain is not plotted, since *Sinh* is an odd function and will yield the negative values of those obtained in the positive domain.

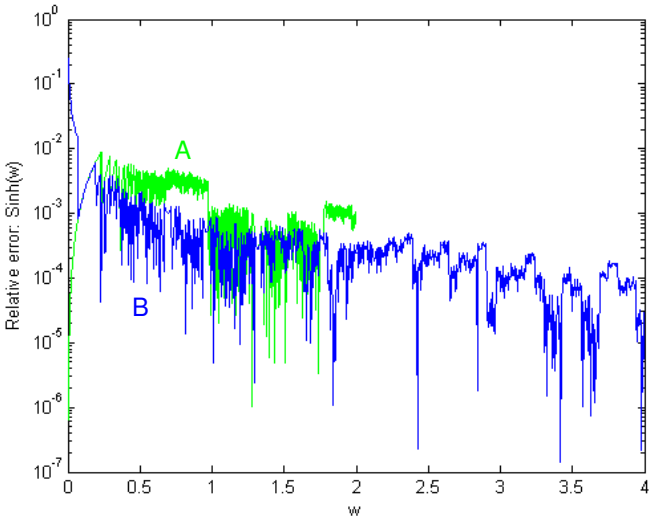

Figure 9. In Curve A, 12 bits were used($w \in [0,2\rangle$). In Curve B, 16 bits were used($w \in [0,4\rangle$).

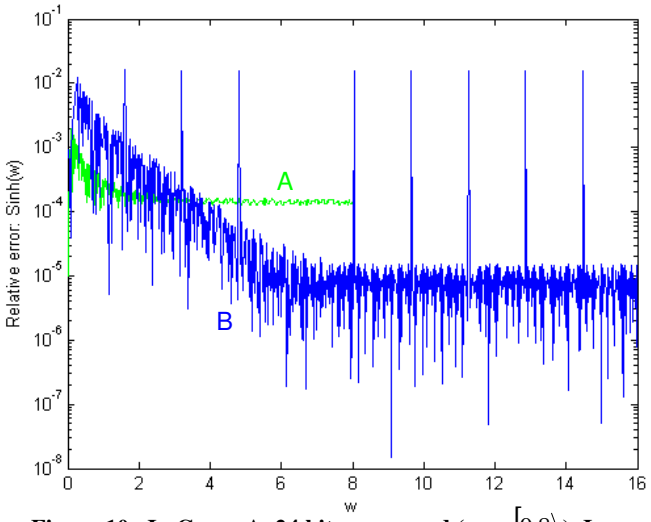

Figure 10. In Curve A, 24 bits were used ($w \in [0,8\rangle$). In Curve B, 32 bits were used ($w \in [0,16\rangle$).

The curve A (Figure 10) for 24 bits is very regular because in this format we use a larger quantity of fractional bits than with the other formats.

The curve B (Figure 10) for 32 bits exhibits some irregularities due to the reduced fractional bits (8) and the reduced number of basic iterations (16). However, it provides the greatest domain for the *Sinh(w)* function ($w \in [-16,16\rangle$).

5.3 Exponential

We will show the relative error for the hardware that implements e^x in the domain obtained in 4.3 for 12, 16, 24 and 32 bits.

Figures 11 and 12 show the relative error for e^w for 12, 16, 24 and 32 bits.

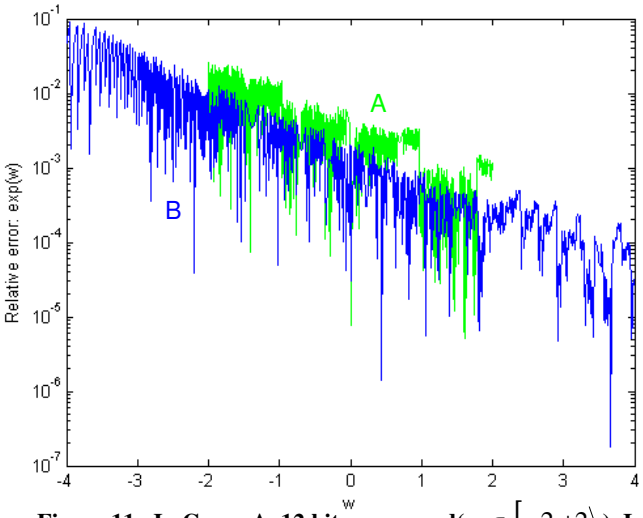

Figure 11. In Curve A, 12 bits were used($w \in [-2,+2\rangle$). In Curve B, 16 bits were used($w \in [-4,+4\rangle$).

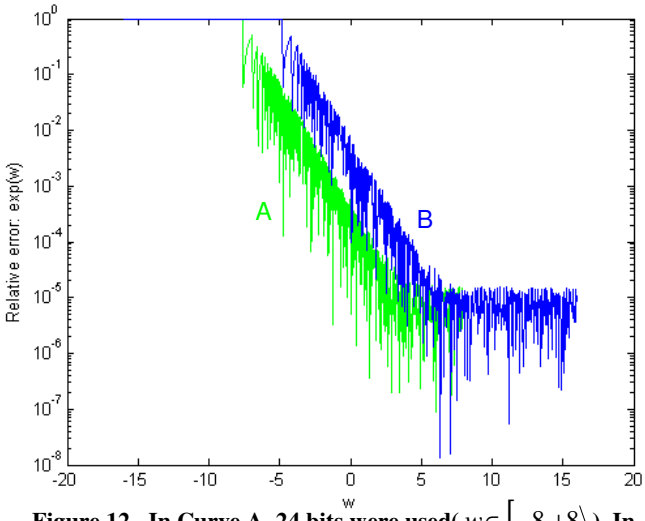

Figure 12. In Curve A, 24 bits were used($w \in [-8,+8\rangle$). In Curve B, 32 bits were used($w \in [-16,+16\rangle$).

Note that, as *w* is more negative, the error increases and even becomes constant (as in Figure 12). The reason is that e^w is very small for large negative values of *w*, and the fixed-point hardware fails representing those small values.

6 CONCLUSIONS

- The expansion scheme proposed by Hu[2], despite the additional hardware needed, has proved to be amenable for our FPGA implementation, as the clock rate and resource effort indicates. The function *Tanh-1* gets expanded in all its domain, and the functions *Cosh* and *Sinh* have a greater domain as the bit width increases.
- The analysis for a unified CORDIC algorithm has not been performed in order to not to lengthen this paper. But the analysis for this case is very similar to that of Section 4.
- The error analysis shows certain irregularities in the relative error performance. This irregularities are due to the truncation of the fractional bits and the ever-limited number of basic and additional iterations. We have tested the CORDIC algorithm in MATLAB® and have found that the error performance is uniform.

7 REFERENCES

[1] J.S. Walther, "A unified algorithm for elementary functions", in Proc. Spring Joint Comput. Conf., 1971, pp. 379-385.

[2] X. Hu, R. Huber, S. Bass, "Expanding the Range of Convergence of the CORDIC Algorithm", IEEE Transactions on Computers. Vol. 40, Nº 1, pp. 13-21, Jan. 1991.

[3] U. Meyer – Baese, Digital Signal Processing with Field Programmable Gate Arrays: Springer-Verlag Berlin Heidelberg, May 2001.

[4] Ray Andraka, "A survey of CORDIC algorithm for FPGA based computers",

[5] Y. Hu: "CORDIC-Based VLSI Architectures for Digital Signal Processing", IEEE Signal Processing Magazine pp. 16-35 (1992)