POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Development of a High-Throughput Floating-Point CORDIC Architecture for Automotive Applications

Supervisors

Prof. Maurizio MARTINA

Prof. Guido MASERA

Dr. Luigi GIUFFRIDA

Candidate

Luigi TEDONE

April 2024

Abstract

The computation of complex functions such as the trigonometric and hyperbolic ones is usually performed either implying large lookup tables or by long software routines. However, the suitability of both solutions strongly depends on the specific application. For example, the former one, storing the results inside a memory, can be very fast and efficient when accuracy is not paramount, but it can lead to prohibitive area occupations. On the other hand, the latter solution exploits polynomial approximation to provide precise results, but it forces the processor to spend a large amount of time waiting for its completion. As a result, the two approaches cannot be adopted for high-accuracy and low-latency applications. In this context, the integration of a dedicated accelerator can be very effective, since it lets the processor offload the execution of such functions. This work, therefore, presents the design and implementation of a unit that, exploiting CORDIC algorithm, is able to reach a large variety of executable functions, albeit maintaining high levels of accuracy and throughput. In its unified version, CORDIC algorithm can compute trigonometric and hyperbolic functions, multiplications, divisions and plane rotations by only performing, iteratively, shift operations and additions. This makes clear its ability to reach low latencies, but also highlights the need for a quite large number of iterations to ensure the desired accuracy. However, being CORDIC architectures particularly prone to unrolling and pipelining, this does not represent a major issue. The main drawback is, then, its strict bond with fixed-point representation, which makes it unsuitable for high precision applications. To overcome this limitation, the proposed design fuses CORDIC algorithm with floating-point arithmetic, thus allowing it to reach the desired accuracy. This extension can follow two strategies, that is either to adopt floating-point arithmetic blocks or to opportunely convert inputs and outputs to have an internal fixed-point pipeline. For this design, the latter solution has been chosen, since it allows to have much faster stages. However, this choice led to the need for specific arrangements to fuse floating-point-specific concepts (Infinities, NaNs) with fixed-point arithmetic. In this work, two binary IEEE-754 formats have been considered, that is single and half precision, since both of them represent a good trade-off between precision and speed. The architecture description process has been carried out in SystemVerilog starting from each computational stage and then moving to the whole system, having high throughput and accuracy as main focuses. This let it not only satisfy the required working frequency of 100 MHz, but also reach a maximum one above 1 GHz. The design has, then, been tested and synthesized, obtaining promising results in terms of error, area, power and speed. In addition, the unit has been integrated as a peripheral in the open-source X-HEEP platform, providing, as

expected, faster execution times, compared to the software-based routines, without trading accuracy for performance. In conclusion, the proposed design has shown to be a viable solution for low-power, high-performance and high-precision applications and future works may focus on the further reduction of power consumption, for example by means of clock gating, on the integration of other standard or custom floating-point formats or on the execution of more complex functions obtained by exploiting the existing ones.

Summary

Trigonometric, hyperbolic and other complex mathematical functions are crucial for several applications such as digital signal processing (DSP), software-defined radios, machine learning algorithms or navigation systems and wireless communications, but performing such computations in digital hardware has always been challenging. There are two traditional approaches when dealing with these functions, that is either to rely on long software routines or on lookup tables (LUTs), but both of them come with trade-offs, since the former is flexible and accurate, but slow and power-hungry, whereas the latter is fast, but trades area occupation for precision. In this context, a third approach seems to be the most effective, that is to rely on additional hardware that is specialized in performing such computations. Therefore, this thesis project aims at designing a hardware accelerator that, following the unified COordinate Rotation DIgital Computer (CORDIC) algorithm, can compute, with constant latency and a target frequency of 100 MHz, a wide range of functions (sine, cosine and arctangent, as well as their respective hyperbolic versions, multiplication and division) with very low levels of error and latency and with promising results in terms of area occupation and power consumption.
CORDIC algorithm is a fixed-point (FXP) iterative procedure that exploits Given's rotation to compute the desired functions by only implying a sequence of additions, subtractions and shifts. In its unified version, it supports two working modes, that is rotation and vectoring, each of which can use three coordinate systems (i.e. circular, linear and hyperbolic) to perform the desired operations. It takes three inputs, x, y and z, one for each branch, returns three outputs and, at each iteration, it computes the new x, y and z based on their current values and on the selected mode, with the underlying criterion of bringing z to zero in rotation mode and y to zero in vectoring mode. Therefore, it could be now clear how its iterative nature can both be a strength and a weakness, since it implies very straightforward operations, but also requires a large number of iterations to achieve a reasonable accuracy. However, this limitation can be easily overcome by unfolding and pipelining the architecture, which allows to compute the desired functions in a single clock cycle once the initial latency has been paid. In addition, also the drawback represented by the restricted dynamic range of fixed-point arithmetic can be solved by adapting

the algorithm to the floating-point (FLP) one, which is the most common format used in scientific applications and which also ensures a higher precision.
In this work, after a preliminary analysis of the implementations proposed in literature, it has been decided to rely on a traditional unfolded and pipelined architecture, in order to support the whole set of functions and to deliver a high-throughput unit. For what concerns, instead, the extension to floating-point arithmetic, two possible strategies can be followed, that is global and local FLP. The former directly implies floating-point adders and shifters within the internal pipeline, which ensures the maximum accuracy, but it is also very likely to need some fine-grain pipelining due to the large critical path caused by the intrinsic complexity of floating-point blocks. On the other hand, the latter converts the FLP inputs into a custom fixed-point format, which takes care of overflow and precision loss by using additional padding bits, and then converts back the fixed-point outputs into floating-point. As a result, although seeming to be less accurate, this method can rely on much faster internal blocks able to reach higher frequencies without the need for fine-grain pipelining. Therefore, after an error analysis of both approaches, it has been decided to rely on global FLP, since it provided equivalent results in terms of accuracy and it is more likely to reach higher frequencies.
Once the floating-point extension strategy has been chosen, the architecture has been designed and implemented targeting single and half precision as the supported FLP formats because of the popularity of the former in scientific applications and of the latter in low-power and area-constrained systems. After an extensive verification process, where several configurations of the internal fixed-point format have been tested on 10000 random values and compared in terms of average relative error, a number of 3 overflow bits (added to the left of the mantissa and in charge of making the block overflow-proof) for x, y and z, 5 guard bits (added to the right and responsible of minimizing precision loss) for x and y and 3 guard bits for z for single precision, as well as 3 overflow bits for x, y and z, 4 guard bits for x and y and 3 guard bits for z for half precision, seemed to be the best trade-off between accuracy and area occupation. In fact, the obtained average relative error is roughly equal to 0.031% for single precision and 0.223% for half precision, which is very close to the error obtained with the maximum numbers of overflow and guard bits, respectively equal to 0.029% and 0.216% for both precisions, but with an area occupation around 30% smaller. For what concerns, instead, the total number of iterations, namely the pipeline stages in an unfolded architecture, it has been decided to rely on 37 stages for single precision and 21 for half precision, in order to guarantee an optimal accuracy without increasing the area occupation too much.
Overall, the unit can be divided into three main areas, that is the pre-processing block, where the conversion from FLP to FXP takes place, the fixed-point CORDIC pipeline, which includes the core of the algorithm, as well as some scaling stages to compensate for the effects of the algorithm itself, and the post-processing block,

where the conversion from FXP back to FLP is performed. In order to provide full floating-point compatibility, especially for what concerns special cases handling, the pre and post-processing blocks communicate with each other through specific control signals that highlight whenever a result is predictable and equal to zero, infinity or NaN, which would otherwise lead to wrong computations in fixed-point arithmetic. In such cases, since it is not necessary to evaluate the results, a bubble is pushed into the pipeline to minimize switching activity, and, thus, power consumption, by disabling the internal registers.
For what concerns the synthesis process, the unit has first been synthesized at the target frequency of 100 MHz, obtaining an area occupation of roughly 0.11 mm^2 for single precision and 0.04 mm^2 for half precision, as well as a power consumption of around 11 mW and 4 mW, respectively, and has, then, been pushed to the maximum frequency, which turned out to be above 1 GHz for both precisions.
As a final step, in order to try it in a real application, the unit has been integrated as an external peripheral into the X-HEEP microcontroller system, an open-source platform based on a RISC-V core, and tested with a simple program that compares the number of clock cycles required both by the accelerator and the CPU (using the `math.h` library) for each of the 16 operations and for 1024 input combinations. In particular, in order to let the CPU really offload the computation to the peripheral, the CORDIC block has been surrounded by a wrapper that receives the control signals from the CPU, manages read and write transfer requests to the DMA and opportunely feeds the CORDIC unit with the correct input signals based on the chosen operation. Given the presence of a single-port memory and of a 32-bit bus, the implementation of such wrapper has been challenging because not only the designed unit is pipelined, which means that, when active, it will always try to flush its content, but it also requires and produces multiple inputs and outputs, which forces the read and write transfers to be synchronized and performed in a specific order. However, the final implementation, as expected, provided a remarkable speedup, with a ratio between the number of clock cycles needed by the CPU and by the accelerator that is equal to 838.83 for trigonometric functions, 1497.76 for hyperbolic functions and 32.56 for multiplication and division.
Therefore, to sum up, the designed CORDIC unit has shown to be a viable solution for the computation of complex mathematical functions, especially for high-throughput, high-accuracy, low-power and low-area applications, and provided feasible results in terms of speedup in a real application. However, it can be considered as a starting point for future improvements, which could include the implementation of low-power techniques, such as clock gating, the support for other standard or custom floating-point formats, the integration as an internal peripheral in a complex platform, such as a digital signal processor or a machine learning accelerator, to improve the overall performance, as well as the support for more operations that can be obtained combining the ones already supported.

Acknowledgements

I would like first of all to thank Prof. Maurizio Martina and Prof. Guido Masera for the valuable concepts learned during their course, which have been fundamental for the development of the project.
I would also like to thank my family and my friends for their continuous presence, as well as my girlfriend for her endless tolerance, support and encouragement.
Last but not least, I really want to thank Dr. Luigi Giuffrida for encouraging and motivating me to always do my best and to never give up, which is something very precious and laudable that I will not forget.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The computation of complex mathematical functions, such as the trigonometric and hyperbolic ones, has always been challenging in digital hardware and is crucial for several applications such as digital signal processing (DSP) [1], software-defined radios [2], matrix arithmetic [3], machine learning algorithms [4], navigation systems [5] and wireless communications [6]. Generally, these functions are computed either adopting specific software solutions or relying on large lookup tables (LUTs) [7], both of which come with trade-offs. By storing the results of operations in memory and retrieving them based on the input value, the latter approach is the faster one between the two, but the size of such memories can easily explode when high accuracy is required. In fact, small LUTs can lead to a significant loss of precision, whereas large ones can achieve low errors, but they are not efficient in terms of area occupation. On the other hand, the computation of complex mathematical functions through software routines is based on polynomial approximations (mainly Taylor's series), which makes this method more flexible and accurate, but forces the system to perform a large number of operations, thus increasing latency and power consumption.
However, as [1, 2, 3, 4, 5, 6] show, there is a third approach that seems to solve the problems of both the aforementioned solutions. This method is based on the COordinate Rotation DIgital Computer (CORDIC) iterative algorithm, which represents a powerful and flexible tool for the computation of complex mathematical functions. By performing a sequence of additions, subtractions and shifts, CORDIC algorithm can compute a wide range of functions, such as sine, cosine, arctangent, hyperbolic sine and cosine, as well as multiplication and division, with very high precision and low latency, thus making it suitable for high-performance, high-throughput and low-power applications. On the other hand, the main drawbacks of CORDIC algorithm are its being inherently a fixed-point algorithm, which means that it supports smaller dynamic ranges compared to floating-point arithmetic,

and the need for a large number of iterations to achieve reasonable error levels. However, these limitations can be easily overcome by adapting the algorithm to floating-point arithmetic and by applying pipelining to the entire architecture, respectively.
For such reasons, in this work it has been decided to design a CORDIC hardware accelerator able to compute, at a frequency of 100 MHz, the whole set of functions supported by the algorithm itself, while maintaining high levels of accuracy and throughput, as well as low area occupation and power consumption. In addition, the unit supports both single and half precision floating-point, since the former is the most common format used in scientific applications, whereas the latter is becoming more and more popular in embedded systems, thanks to its reduced area occupation and power consumption. Therefore, after a preliminary analysis of the implementations proposed in literature [8], it has been decided to rely on a traditional unfolded and pipelined architecture that can be opportunely configured to compute the desired functions and which showed the ability to reach a maximum frequency above 1 GHz for both floating-point formats. In addition, the final unit has, then, been integrated in the X-HEEP microcontroller system [9], an open-source platform based on a RISC-V core, and tested in a real application, showing that it can be used as an external accelerator capable of providing a significant speedup, especially in the computation of trigonometric and hyperbolic functions.
The rest of the text is organized as follows:

- Chapter 2 provides a brief overview of the IEEE-754 standard for floating-point arithmetic and of CORDIC algorithm, as well as a summary of the most relevant works in literature about CORDIC implementations.
- Chapter 3 describes the proposed architecture for the CORDIC unit, focusing on explaining the strategies adopted within each structural block and the overall design choices.
- Chapter 4 explains the integration process of the accelerator in the X-HEEP microcontroller system.
- Chapter 5 presents the obtained results in terms of accuracy, area occupation, power consumption and maximum achievable frequency, as well as the performance and achievable speedup of the unit in a real application.
- Chapter 6 draws the conclusions and suggests some possible future works.

Chapter 2

Background

2.1 Floating-Point representation

Performing arithmetic operations in digital electronics lays on the conversion of numbers into a binary format. This can be generally pursued in two ways:

- fixed-point representation (FXP).
- floating-point representation (FLP).

Supposing to adopt a sign-magnitude form, in fixed-point representation a decimal number n_{10} is described as

$$n_{10} = -1^S \cdot \sum_{i=m-1}^{i=-n} d_i \cdot 2^{-i} \tag{2.1}$$

where

- S is the sign bit, 0 for positive numbers and 1 for negative ones.
- d_i is the digit in position i.
- m is the number of bits needed to represent the integer part n_i of n_{10}.
- n is the number of bits needed to represent the fractional part n_f of n_{10}.

As a consequence, a number $n_{10} = 10.25$ becomes

$$n_{10} = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \tag{2.2}$$

$$n_2 = 1010.01 \tag{2.3}$$

which implies

- $S = 0$
- $m = 4$
- $n = 2$

The general rule for fixed-point arithmetic is to perform mathematical operations after aligning the operands based on the position of the decimal point, without the need for shifting them. This results in a rather straightforward procedure, given that the steps are quite similar to the ones performed for decimal arithmetic, but it comes at the cost of a limited range of representable values that is restricted by the available number of bits $m + n$ to $2^{-n} \leq |n_{10}| \leq 2^m - 2^{-n}$. In order to cope with this, floating-point representation was introduced.
The IEEE-754 standard [10] established the details about floating-point binary representation and arithmetic. In order to be conform to the standard, each decimal number n_{10} has to be described as

$$n_{10} = -1^S \cdot 2^{E-b} \cdot m \tag{2.4}$$

where

- S is the sign bit, 0 for positive numbers and 1 for negative ones.
- E is the biased exponent and is described by w bits.
- b is the bias that has to be subtracted from the biased exponent to obtain the effective one e. It is equal to $2^{w-1} - 1$.
- m is the mantissa (significand) and is described by p bits. It represents the absolute value of the number in a normalized form with respect to the exponent e and, for this reason, the condition $0 \leq m < 2$ always holds. Among the p bits, only the trailing $p - 1 = t$ ones are actually used in the selected format, representing the trailing significand T in Figure 2.1. The most significant one is called hidden bit and is dropped since always equal to 1. Considering $T = \{d_1, d_2, ..., d_{p-1}\}$, where d_i indicates the digit of T in position i, this results in
$$m = (1 + \sum_{i=1}^{t} t_{t-i} \cdot 2^{-i}) < 2 \tag{2.5}$$

Each floating-point format can represent three types of values:

- **Normalized numbers**: $e_{min} + b \leq E \leq e_{max} + b$, where
  - $e_{min} = 1 - b =$ minimum exponent value $= -(2^{w-1} - 1)$

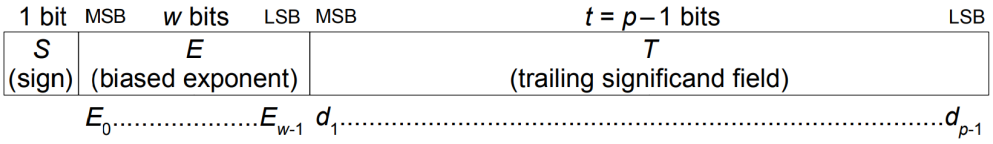


Figure 2.1: IEEE-754 Floating-Point format [10].

- $e_{max} = b =$ maximum exponent value $= 2^{w-1} - 1$

  For such values the hidden bit is 1.

- **Subnormal numbers and zero**: $E = 0$. For such values the hidden bit is zero in order to extend the range of representable values. This causes a loss of precision, since the number of bits available for the fractional part is being reduced.

- **Special values**: $E = 2^w - 1$
  - $T = 0$: $\pm\infty$
  - $T \neq 0$: NaN (Not a Number)
    - signaling NaN (sNaN): the most significant bit of the mantissa is fixed to 0 and at least one of the following is 1.
    - quiet NaN (qNaN): the most significant bit of the mantissa is equal to 1 and the other ones are zeros.

The steps to convert a decimal number n_{10} into its floating-point representation are the following:

1. Compute the sign bit S by looking at the sign of n_{10}.
2. Represent n_{10} in fixed-point format, namely $n_2 = \{n_i, n_f\}$, where n_i is the integer part and n_f the fractional one.
3. Normalize the obtained number so that $n_i = 1$. The unbiased exponent will be equal to the w-bit representation obtained shift amount and the t most significant bits of the final n_f will be the trailing significand T.
4. Compute the biased exponent E by adding the bias b to e.
5. Drop the hidden bit (n_i) and pack the obtained elements as shown in Figure 2.1.

As an example, the procedure to convert $n_{10} = 5.625$ into the binary16 format is:

1. $S = 0$
2. $n_2 = 101.101$, $n_i = 101$, $n_f = 101$
3. $n_2 = 1.01101 \cdot 2^2$, $e = 2$, $T = 01101$
4. $E = 2 + 15 = 17 = 10001_2$
5. $n_{2,\ FLP} = 0\ 10001\ 0110100000$

Instead, to convert a floating-point number n_2 into its decimal representation, the following steps have to be performed:

1. Compute the unbiased exponent e by subtracting the bias b from E.
2. Construct the normalized fixed-point representation by adding the hidden bit to the trailing significand T and convert it into the decimal format.
3. Compute the final value by multiplying the obtained fixed-point number by 2^e.
4. Change the sign of the obtained number if $S = 1$.

As an example, the procedure to convert $n_2 = 0\ 10001\ 0110100000$ into the decimal format is:

1. $e = 17 - 15 = 2$
2. $n_2 = 1 + 0.01101000000 = 1.01101$
3. $n_{10} = 1.01101 \cdot 2^2 = 5.625$

Table 2.1 includes the main binary floating-point formats described in [10].
Floating-point arithmetic follows specific rules, in order to provide meaningful results. Since CORDIC algorithm only relies on additions and shifts, these two operations will be described in the following section.

2.2 Floating-Point addition and shift

2.2.1 Floating-Point addition

The IEEE-754 standard [10] defines the rules to be accomplished when performing floating-point addition. In particular, the following steps have to be followed:

Format	w	t	b	Total width
binary16 (half precision)	5	10	15	16
binary32 (single precision)	8	23	127	32
binary64 (double precision)	11	52	1023	64
binary128 (quad precision)	15	112	16383	128

Table 2.1: IEEE-754 Floating-Point formats parameters.

1. Check the presence of NaNs and infinities. If the result can be predicted depending on Table 2.2, skip the computation.
2. Find the maximum exponent E between the two operands. This will be the final one.
3. Isolate the trailing significand of the two operands and add the hidden bit to obtain the mantissas.
4. Shift right the significand of the operand with the smaller exponent by an amount equal to the difference $E - E_{smaller}$.
5. Find the effective operation (EOP) to be performed depending on the sign bits of the two operands:
   - $S_1 \neq S_2$: subtraction
   - $S_1 = S_2$: addition
6. Add or subtract the significands of the two operands according to the EOP.
7. Eventually normalize the obtained result based on its integer part n_i:
   - $n_i = 1$: no shift operations are needed.
   - $n_i = 0$: shift left the result up to when the integer part is equal to 1 and save this amount to decrease the final exponent by the same value.
   - $n_i > 1$: shift right the result up to when the integer part is equal to 1 and save this amount to increase the final exponent by the same value. Perform rounding according to the rounding mode.

8. Compute the final exponent by adding the saved amount to E and check if $e_{min} + b \leq E \leq e_{max} + b$. If it is outside this range, two cases are possible:
   - $E < e_{min} + b$ (underflow): the result is zero.
   - $E > e_{max} + b$ (overflow): the result is an infinity. The sign will be the same as either of the two operands, given that this occurs if and only if $S_1 = S_2$.
9. If neither underflow nor overflow occurred, drop the hidden bit and pack the obtained elements as shown in Figure 2.1. The sign bit will be the same as the result at point 7.

Operand 1	**Operand 2**	**Result**
$\pm NaN$	$\pm NaN$	$\pm NaN$
$\pm NaN$	$\pm\infty$	$\pm NaN$
$\pm NaN$	$\pm n$	$\pm NaN$
$\pm\infty$	$\pm\infty$	$\pm\infty$
$\pm\infty$	$\mp\infty$	NaN
$\pm\infty$	$\pm n$	$\pm\infty$

Table 2.2: Predictable results for floating-point addition. n is a finite number.

2.2.2 Floating-Point shift

For what concerns the shift operation, thanks to the floating-point representation itself, the procedure is rather straightforward. The shift amount has to be added to or subtracted from the biased exponent, depending on the fact that the shift direction is left or right, respectively. However, also in this case overflow and underflow have to be taken into account:

- $E_{final} < e_{min} + b$ (underflow): the result is zero.
- $E_{final} > e_{max} + b$ (overflow): the result is an infinity.

2.2.3 Floating-Point status flags

The IEEE-754 standard [10] defines five status flags that have to be set depending on the result and the operands of the floating-point arithmetic operations:

1. **Invalid Operation (NV)**: no useful definable results can be produced. This happens for:
   - General computations on NaNs.
   - Multiplications between 0 and ∞.
   - Magnitude subtraction of infinities.
   - Divisions like $\frac{0}{0}$, $\frac{\infty}{\infty}$.
   - Square root of a negative number.
   - Remainder computation of $\frac{x}{y}$ when x is 0 or y is ∞.

   The produced value is a NaN.

2. **Division by Zero (DZ)**: an exact infinite result is produced by an operation on finite numbers. This happens for:
   - Divisions by 0. The produced value is an infinity whose sign is given by the XOR of the sign bits of the two operands.
   - Logarithm of 0. The result is $-\infty$.

3. **Overflow (OF)**: the result is finite but too large to be represented, that is $e > e_{max}$. Depending on the adopted rounding method, the final value will be:
   - $\pm\infty$ if the rounding method is round to the nearest or to the nearest even.
   - The maximum representable value if the rounding method is round towards zero.

4. **Underflow (UF)**: the result is too small to be represented, that is $e < e_{min}$. Depending on the implementation, the final value can be:
   - Zero.
   - A subnormal number.
   - The minimum representable value, that is $\pm 2^{e_{min}}$.

5. **Inexact (NX)**: the result differs from what it would have been if the precision was unbounded. NX is raised whenever overflow or underflow occurs.

As stated in [10], only one flag can be set at a time. The only case in which two of them can be raised is when overflow or underflow occurs, then also NX is being set.

2.3 CORDIC algorithm

2.3.1 Traditional CORDIC algorithm

The COordinate Rotation Digital Computer (CORDIC) algorithm was first presented by J. E. Volder in 1959, when he proposed a special purpose digital computing unit [11] whose aim was to evaluate trigonometric functions by means of plane rotation.
Conventionally, performing the 2D vector rotation by an angle θ in Figure 2.2 lays on the following Givens rotation equations:

$$x_{out} = x_{in} \cdot cos(\theta) - y_{in} \cdot sin(\theta) \quad (2.6)$$
$$y_{out} = y_{in} \cdot cos(\theta) + x_{in} \cdot sin(\theta) \quad (2.7)$$

where (x_{in}, y_{in}) are the initial coordinates and (x_{out}, y_{out}) the final ones.

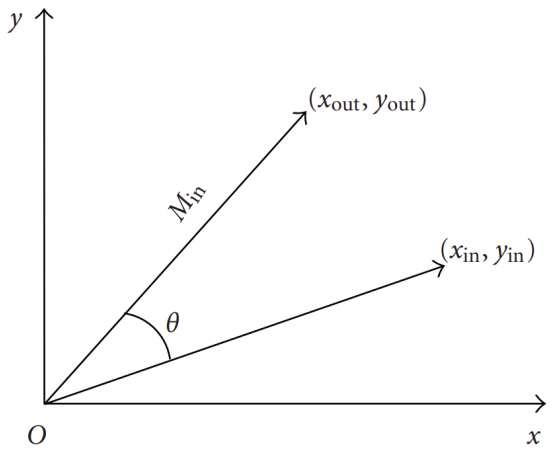


Figure 2.2: Two-dimension vector rotation [8].

However, albeit seeming straightforward, equations 2.6 and 2.7 are difficult to be implemented in hardware, since they would need the allocation of two adders, four multipliers and a lookup table to store all the possible values of the trigonometric functions. CORDIC algorithm, instead, makes this process simpler by decomposing the larger rotation angle θ into several smaller ones, so that, after performing all these micro-rotations, the same outcome can be obtained. For such purpose, the angle θ can be expressed as

$$\theta = \sum_{i=0}^{\infty} \alpha_i \tag{2.8}$$

and equations 2.6 and 2.7 become, at iteration i,

$$\begin{aligned} x_{i+1} &= x_i \cdot cos(\alpha_i) - y_i \cdot sin(\alpha_i) && (2.9)\\ y_{i+1} &= y_i \cdot cos(\alpha_i) + x_i \cdot sin(\alpha_i) && (2.10) \end{aligned}$$

By doing so, the same rotation of an angle θ can be achieved performing a series of micro-rotations, each of them with angle α_i.
According to trigonometric transformations, equations 2.9 and 2.10 can be rewritten as follows:

$$\begin{aligned} x_{i+1} &= \frac{x_i - y_i \cdot tan(\alpha_i)}{\sqrt{1 + tan^2(\alpha_i)}} && (2.11)\\ y_{i+1} &= \frac{y_i + x_i \cdot tan(\alpha_i)}{\sqrt{1 + tan^2(\alpha_i)}} && (2.12) \end{aligned}$$

However, it is still difficult to compute them in hardware due to the presence of the complex denominator. To cope with this, CORDIC algorithm adopts the concept of pseudo-rotations, where the term $\sqrt{1 + tan^2(\alpha_i)}$ is neglected and treated as a scaling contribution to be compensated later. In fact, the amplitude of the final vector will be larger by a quantity equal to the product of the reciprocals of all the denominators. Thanks to this, the new equations become

$$\begin{aligned} x_{i+1} &= x_i - y_i \cdot tan(\alpha_i) && (2.13)\\ y_{i+1} &= y_i + x_i \cdot tan(\alpha_i) && (2.14) \end{aligned}$$

and the inverse of the amount by which the amplitude of the vector is larger than the original one, namely the scaling factor, is

$$K = \prod_{i=0}^{i=\infty} \frac{1}{\sqrt{1 + \sigma_i^2 \cdot tan^2(\alpha_i)}} \tag{2.15}$$

Hence, the final x and y coordinates have to be multiplied by K.
Since also the term $tan(\alpha_i)$ causes some issues when dealing with hardware implementations, CORDIC algorithm further simplifies equations 2.13 and 2.14 by using α_i angles such that

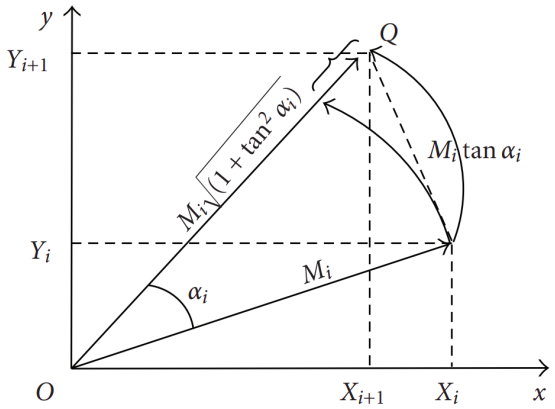


Figure 2.3: Impact of the absence of the denominator [8].

$$tan(\alpha_i) = 2^{-i} \tag{2.16}$$
$$\alpha_i = tan^{-1}(2^{-i}) \tag{2.17}$$
$$\theta = \sigma_0 \cdot \alpha_0 + \sigma_1 \cdot \alpha_1 + ... + \sigma_n \cdot \alpha_n \tag{2.18}$$

where σ_i is the direction of rotation needed to achieve the result in equation 2.18 and is computed in specific ways depending on the CORDIC operational mode. It can be either 1 or -1. At this point, equations 2.9 and 2.10 finally become

$$x_{i+1} = x_i - \sigma_i \cdot y_i \cdot 2^{-i} \tag{2.19}$$
$$y_{i+1} = y_i + \sigma_i \cdot x_i \cdot 2^{-i} \tag{2.20}$$

where

- x_{i+1} and y_{i+1} are the x and y coordinates of the vector found at iteration i and used in iteration $i+1$.
- x_i and y_i are the x and y coordinates of the vector found at iteration $i-1$ and used in iteration i. Their initial value is equal to the input one of x and y.
- α_i is the elementary rotation angle.

and equation 2.15 becomes

$$K = \prod_{i=0}^{i=\infty} \frac{1}{\sqrt{1 + \sigma_i^2 \cdot 2^{-2i}}} \tag{2.21}$$

In addition to the x and y coordinates, another one, z, which tracks the remaining angle at iteration i needed to satisfy equation 2.8, has to be included:

$$z_{i+1} = z_i - \sigma_i \cdot \alpha_i \tag{2.22}$$

CORDIC algorithm provides for the presence of two operational modes, rotation and vectoring, both based on equations 2.19-2.22. The main differences are (assuming that the effects of the scaling coefficient K have been compensated):

1. **Rotation mode**
   - $x_{out} = x_{in} \cdot cos(z_{in}) - y_{in} \cdot sin(z_{in})$
   - $y_{out} = y_{in} \cdot sin(z_{in}) + x_{in} \cdot sin(z_{in})$
   - $z_{out} \to 0$, hence the computation can be stopped as soon as $z_i \approx 0$
   - $\sigma_i = sign(z_i)$ to have $z_{out} \to 0$
2. **Vectoring mode**
   - $x_{out} = sign(x_{in}) \cdot \sqrt{x_{in}^2 + y_{in}^2}$
   - $y_{out} \to 0$, hence the computation can be stopped as soon as $y_i \approx 0$
   - $z_{out} = z_{in} + tan^{-1}(\frac{y_{in}}{x_{in}})$
   - $\sigma_i = -sign(x_i) \cdot sign(y_i)$ to have $y_{out} \to 0$

It can be easily noticed that, neglecting the final multiplication by the scaling factor K, only additions and right-shift operations are required to obtain the final result, with the addition of a small lookup table to store the specific values of the $\alpha_i = tan^{-1}(2^{-i})$ angles. This straightforwardness represents a remarkable result.

2.3.2 Unified CORDIC algorithm

The very turning point for CORDIC algorithm happened in 1971, when J. S. Walther [12] included other coordinates systems in a unified algorithm capable of computing several mathematical functions. Some of these are:

- $sin(\theta)$ and $cos(\theta)$
- $sinh(\theta)$ and $cosh(\theta)$

- $tan^{-1}(\theta)$
- $tanh^{-1}(\theta)$
- Multiplication
- Division
- Two-dimension rotation of a vector
- Amplitude and phase computation of a complex number

It has to be noted that additional mathematical functions can be obtained by executing the listed ones in a specific order [13] or by opportunely choosing the inputs. As an example, the tangent function can be obtained by computing $sin(\theta)$ and $cos(\theta)$ in rotation mode and circular coordinates ($x_{in} = \frac{1}{K_1}$, $y_{in} = 0$) and then performing a division.
In order to achieve this variety of functions, Walther developed a new form for equations 2.19, 2.20 and 2.22, including a new variable, m, which represents the coordinates system to be used

$$x_{i+1} = x_i - m \cdot \sigma_i \cdot y_i \cdot 2^{-S_{m,i}} \tag{2.23}$$
$$y_{i+1} = y_i + \sigma_i \cdot x_i \cdot 2^{-S_{m,i}} \tag{2.24}$$
$$z_{i+1} = z_i - m \cdot \sigma_i \cdot \alpha_{m,i} \tag{2.25}$$

In this case,

1. $m \in \{-1, 0, 1\}$
   - $m = 1$ for a circular coordinates system.
   - $m = 0$ for a hyperbolic coordinates system.
   - $m = -1$ for a linear coordinates system.
2. $S_{m,i}$ is a specific shift sequence that depends on the coordinates system, as shown in Table 2.3.

As a consequence, also equations 2.15 and 2.17 had to be modified

$$\alpha_{m,i} = \frac{1}{\sqrt{m}} \cdot tan^{-1}(\sqrt{m} \cdot 2^{-S_{m,i}}) \tag{2.26}$$
$$K_m = \prod_{i=S_{m,0}}^{\infty} \frac{1}{\sqrt{1 + m \cdot 2^{-2 \cdot S_{m,i}}}} \tag{2.27}$$

Coordinates System m	**Shift Sequence** $S_{m,i}$
1 (Circular)	0, 1, 2, 3, ..., i
0 (Linear)	1, 2, 3, 4, ..., i+1
-1 (Hyperbolic)	1, 2, 3, 4, 4, 5, ... [1]

Table 2.3: Shift sequences.

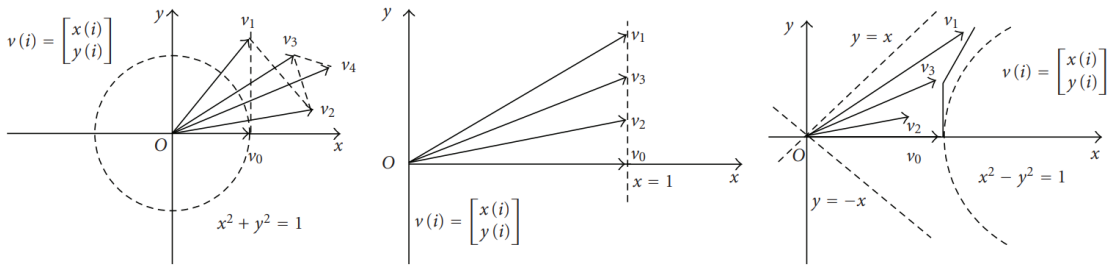
m	$\alpha_{m,i}$	$\frac{1}{K_m}$ [2]
1 (Circular)	$tan^{-1}(2^{-S_{1,i}})$	$\prod\sqrt{1+\sigma_i^2 \cdot 2^{-S_{1,i}}} \approx 1.65$
0 (Linear)	$2^{-S_{0,i}}$	1
-1 (Hyperbolic)	$tanh^{-1}(2^{-S_{-1,i}})$	$\prod\sqrt{1-\sigma_i^2 \cdot 2^{-S_{-1,i}}} \approx 0.83$

Table 2.4: Angles and scaling coefficients for unified CORDIC.

thus resulting in what is described in Table 2.4.
At this point, it should be clear how powerful CORDIC architectures are, since the allocation of one unit can provide for the computation of several functions with fewer resources compared to standalone solutions for each of them. However, this comes at a cost, since the range of values that can produce a valid result (Range Of Convergence, ROC) and, then, that can be accepted as inputs is rather restricted. Table 2.5 summarizes the resulting values of each working mode and the corresponding ROC for CORDIC algorithm.

[1] For $m = -1$ iterations with number $i = 3\cdot$k+1, with $k = 1, 2, ...$ have to be repeated twice.

[2] $\frac{1}{K_m}$ has been used to make the table cleaner.



(a) Circular system (b) Linear system (c) Hyperbolic system

Figure 2.4: CORDIC algorithm among the various coordinates systems [8].

2.3.3 Traditional hardware implementations of unified CORDIC

The traditional hardware implementations of CORDIC algorithm lay on fixed-point arithmetic to perform all the required operations. This implies that, since the operands' description is limited to n bits, a finite number of iterations can be used. As declared in [12], in order to obtain a n-bit precision, a final shift value $S_{m,i}$ equal to n is needed.
There are two types of architectures, folded and unfolded, each of which is focused on optimizing a specific factor between area and timing. In this section the scaling circuitry is not described, as it only consists of a multiplier.

Folded architecture

The typical folded architecture is shown in Figure 2.5.
The main pro of this implementation is the allocation of only three adders, two Barrel shifters and a lookup table, which makes it very area-efficient. However, this comes at a cost, given that each result will be available after n iterations, thus making this architecture the slower one in terms of latency. The clock frequency, instead, can be reasonably high, considering a relatively short critical path.

Unfolded and pipelined architecture

The previous solution can be improved in terms of timing performance by unfolding it into n stages and pipelining them [1]. These operations surely increase the required

[1]Unfolding without pipelining only increases the required area, without effectively reducing the overall latency, since in this case a clock frequency f_{clk}/n is needed

Mode	**m**	**Result**	**ROC** [14]
Rotation	Circular	$x_n = x_{in} \cdot cos(z_{in}) - y_{in} \cdot sin(z_{in})$ $y_n = y_{in} \cdot cos(z_{in}) + x_{in} \cdot sin(z_{in})$ $z_n = 0$	$\|z_{in}\| < 1.74$
	Hyperbolic	$x_n = x_{in} \cdot cosh(z_{in}) + y_{in} \cdot sinh(z_{in})$ $y_n = y_{in} \cdot sinh(z_{in}) + x_{in} \cdot cosh(z_{in})$ $z_n = 0$	$\|z_{in}\| < 1.11$
	Linear	$x_n = x_{in}$ $y_n = y_{in} + x_{in} \cdot z_{in}$ $z_n = 0$	$\|z_{in}\| < 1$
Vectoring	Circular	$x_n = sign(x_{in}) \cdot \sqrt{x_{in}^2 + y_{in}^2}$ $y_n = 0$ $z_n = z_{in} + tan^{-1}(\frac{y_{in}}{x_{in}})$	$\|tan^{-1}(\frac{y_{in}}{x_{in}})\| < 1.74$
	Hyperbolic	$x_n = sign(x_{in}) \cdot \sqrt{x_{in}^2 - y_{in}^2}$ $y_n = 0$ $z_n = z_{in} + tanh^{-1}(\frac{y_{in}}{x_{in}})$	$\|tanh^{-1}(\frac{y_{in}}{x_{in}})\| < 1.11$
	Linear	$x_n = x_{in}$ $y_n = 0$ $z_n = z_{in} + \frac{y_{in}}{x_{in}}$	$\|\frac{y_{in}}{x_{in}}\| < 1$

Table 2.5: CORDIC working modes summary.

area by a factor equal to the number of iterations n, but also raise the throughput by the same amount when the pipeline is completely full.
The typical unfolded architecture is shown in Figure 2.6.
As it can be easily noticed, the area occupation is now n times larger, but, after waiting for the pipeline to be filled, a new result is available at each clock cycle. It is also remarkable that shift operations can be hardwired here and, thus, no Barrel

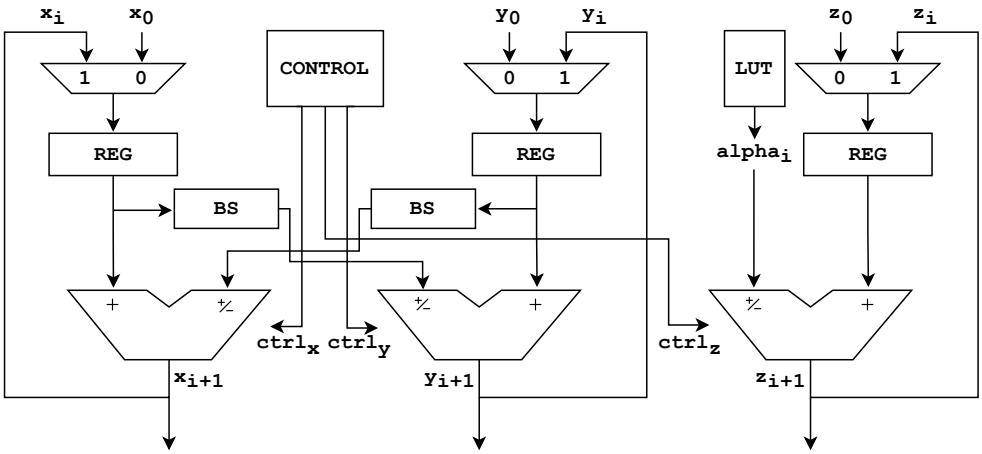


Figure 2.5: Folded CORDIC architecture.

shifters are required for this type of circuit, since the iteration number is fixed and known in advance for each stage i. The clock frequency can be considered equal to the one of the folded solution.

2.3.4 Possible improvements

Apart from bare unfolding and pipelining, additional techniques can be used to further decrease latency: redundant arithmetic and higher radices.

Redundant arithmetic

Thanks to carry propagation-free adders, so Carry Save (CS) Adders or Signed Binary Digits (SBD) Adders, redundant arithmetic makes the delay of each stage lower and independent on the size of the datapath. However, this causes the scale factor of equation 2.21 to be variable, since $\sigma_i = 0$ becomes a feasible alternative depending on intermediate results. This is a major drawback, since the computation of K causes the presence of additional latency, whereas in non-redundant arithmetic the scaling factor is known in advance. In addition, in redundant representations it becomes difficult to find the sign of the coordinates within each iteration, causing the need for considering not only one bit, generally the sign bit, but several ones to compute σ_i.

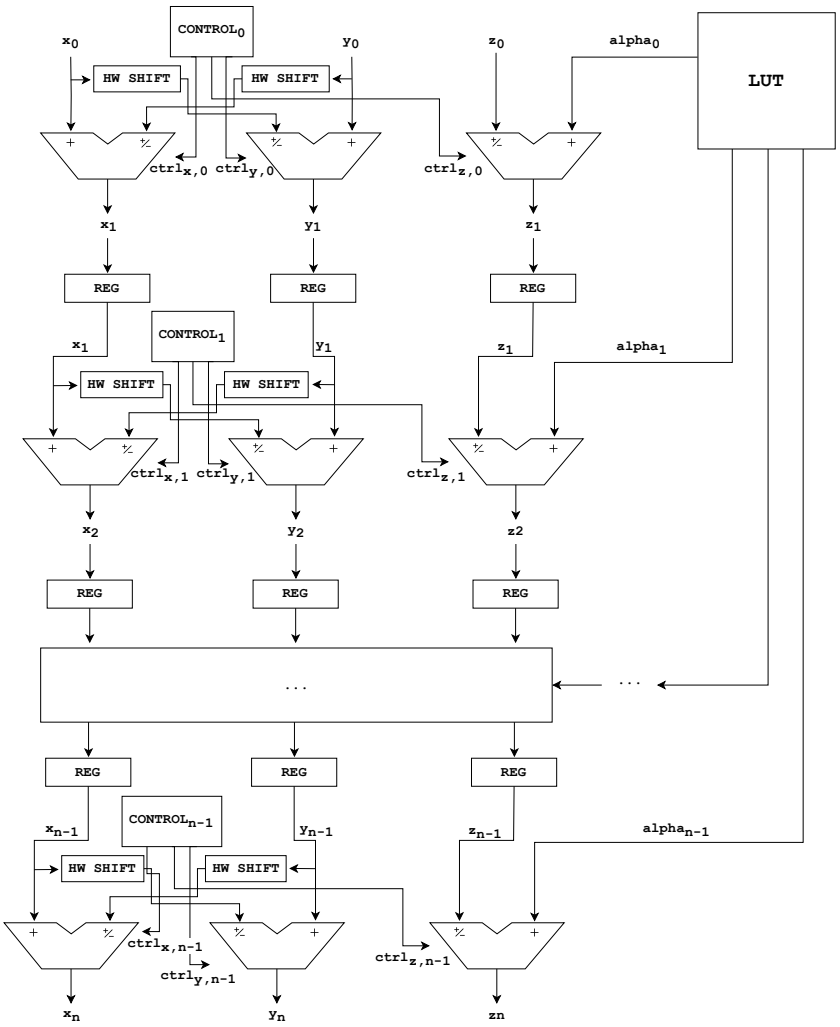


Figure 2.6: Unfolded and pipelined CORDIC architecture.

Higher radices

Adopting a higher radix ρ helps reducing the number of iterations, thus requiring fewer CORDIC stages, but causes the scaling factor K not to be constant anymore, given that $|\sigma_i| \in [0, \rho - 1]$ and $\rho - 1$ can be different from 1. In addition, it generally introduces a noticeable overhead in terms of complexity.

2.4 Floating-Point extension for CORDIC algorithm

CORDIC algorithm was first developed to work with fixed-point arithmetic. However, this way of representing numbers results in a rather small range of values that can be used once the number of integer and fractional bits has been chosen. In order to cope with this, floating-point arithmetic can be adopted. To perform this extension, two possible strategies can be used [15]:

- Local FLP: each micro-rotation performs floating-point operations, hence adjusting the exponents and normalizing the result at each iteration.
- Global FLP: the floating-point inputs are converted into specific fixed-point formats that include additional overflow and guard bits to take care of accuracy loss throughout the iterations.

Although the local solution could seem to be the most promising one, in reality it is very costly in terms of hardware and speed, since FLP adders and shifters are needed for each stage. In addition, it leads to useless operations, because, between two iterations, numbers are unnecessarily denormalized and normalized again. With the latter approach, instead, CORDIC algorithm remains fixed-point and FLP-specific operations are done only at the beginning and at the end of the unit. This makes the second solution the most speed-effective.

2.5 Related works

2.5.1 Fixed-point CORDIC

Nonredundant Low Latency CORDIC [16]

The algorithm proposed in [16] improves conventional CORDIC by employing linear approximation to rotation when the residual angle is small. This remaining angle, θ_r, is chosen so that

$$sin(\theta_r) \approx \theta_r \tag{2.28}$$
$$cos(\theta_r) \approx 1 \tag{2.29}$$

For the first $n/2+1$ micro-rotations equations 2.19, 2.20 and 2.22 are employed. The σ_i values for the first $n/3$ rotations are determined iteratively, whereas, starting from iteration $n/3+1$, they can be computed in parallel, since the relation

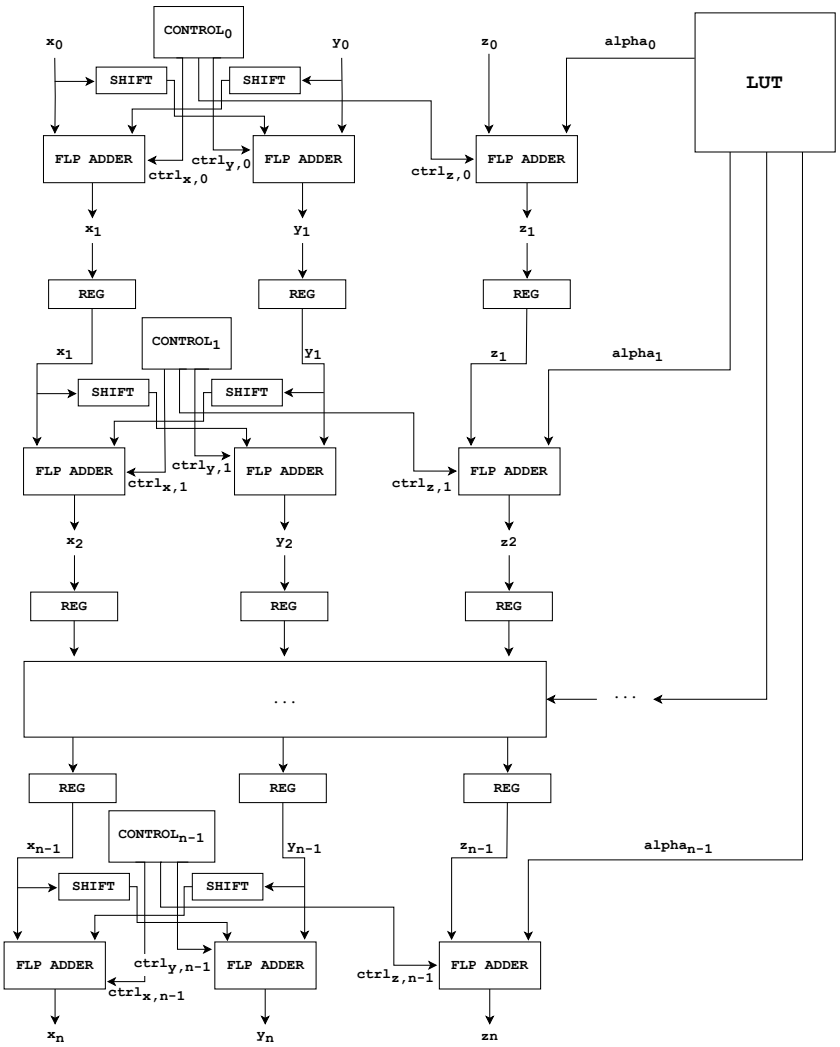


Figure 2.7: Unfolded and pipelined CORDIC architecture with local FLP.

$$\lim_{k\to\infty} \frac{\tan 2^{-k}}{2^{-k}} = 1 \tag{2.30}$$

holds. Thus, for $(n/3+1) \leq i \leq (n/2+1)$, all the σ_i values are obtained from the remaining angle $z_{n/3+1}$ and are used to generate $z_{n/2+1}$ from $z_{n/3+1}$. For $i > (n/2+1)$, only a single rotation of an angle equal to the remaining one $z_{n/2+1}$ is performed. Therefore, at the end,

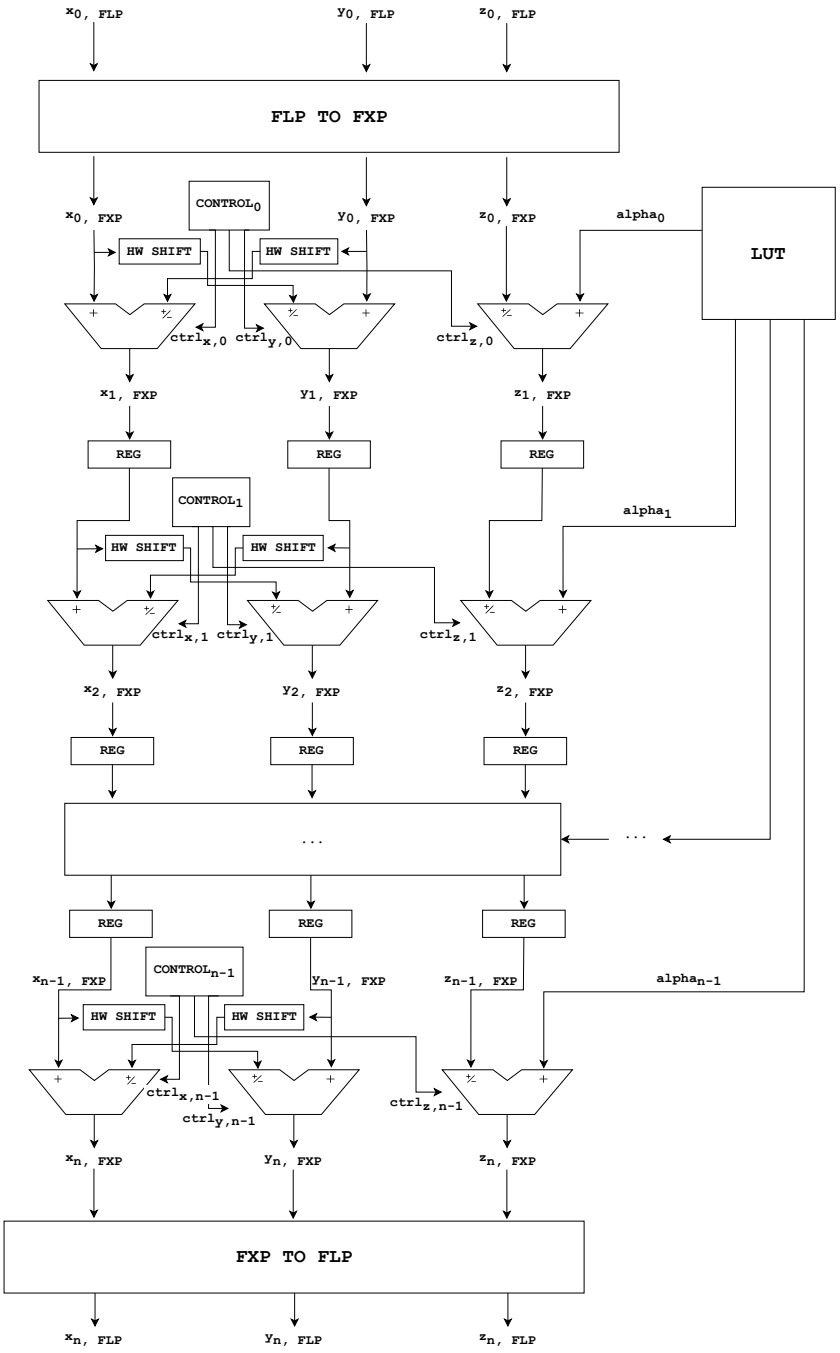


Figure 2.8: Unfolded and pipelined CORDIC architecture with global FLP.

$$x_f = x_{n/2+2} = k_{n/2+1}(x_{n/2+1} - \theta_r y_{n/2+1}) \quad (2.31)$$

$$y_f = y_{n/2+2} = k_{n/2+1}(\theta_r x_{n/2+1} + y_{n/2+1}) \quad (2.32)$$

where $\theta_r = z_{n/2+1}$, $k_{n/2+1}$ is the scale factor in iteration $(n/2+1)$ (which is constant, since $\sigma_i \in \{-1,1\}$, and whose compensation is done concurrently with the computation of the x and y coordinates using two multipliers in parallel) and x_f and y_f are the scaled final coordinates.
In [17] a unified architecture is proposed, but it supports both CORDIC rotation and vectoring modes only in circular coordinates.

Pipeline FPGA Implementation of Unified CORDIC [18]

In [18] a recent and straightforward pipelined FPGA implementation of CORDIC algorithm able to work in both rotation and vectoring mode and in circular and hyperbolic coordinates is presented. The architecture does not rely on complex algorithms to skip iterations or to predict the rotation directions, but simply puts several pipeline stages (named STEPs) one after the the other, each of which is in charge of computing equations 2.19, 2.20 and 2.22. In order to have a unified implementation, proper control signals and multiplexers are used to choose the working mode and coordinate system. The presented architecture is for 16 bits, but can be easily extended to larger precisions.

Double Rotation Method CORDIC [19]

Albeit relying on redundant arithmetic, the double rotation mode performs two rotations (each of them called sub-rotation) for each elementary angle during the first $n/2$ iterations, in order to achieve a constant scale factor independently on the operands. During the remaining $n/2$ iterations, instead, only one rotation is being done. In order to satisfy equations 2.19, 2.20 and 2.22, the following workarounds have been taken into account:

- A positive rotation ($\sigma_i = 1$) is made by two positive sub-rotations.
- A negative rotation ($\sigma_i = -1$) is made by two negative sub-rotations.
- A non-rotation ($\sigma_i = 0$) is made by a positive and a negative sub-rotations.

Therefore, compared to a basic redundant CORDIC algorithm where K is data-dependent and has to be computed alongside with the iterations, 50% additional iterations are required to make it constant.
Since [19] uses this method to compute sine and cosine functions, it has been assumed that the algorithm is only suitable for CORDIC in rotation mode and circular coordinates.

Correcting Rotation Method CORDIC [19]

The correcting rotation method also relies on redundant arithmetic and achieves a constant K. This is pursued by avoiding rotations corresponding to $\sigma_i = 0$ and by performing extra rotations, at fixed intervals, to correct the errors introduced by this choice. Assuming to use b bits to estimate the sign of z_i, this correction interval has to be lower or equal than $(b-2)$. If $b = 3$ or $b = 4$, this means that also this method requires 50% additional iterations.
Since [19] uses this method to compute sine and cosine functions, it has been assumed that the algorithm is only suitable for CORDIC in rotation mode and circular coordinates.

Pipelined Low Latency Redundant CORDIC [20]

This algorithm reduces the latency of redundant CORDIC by following these rules for each range of iterations:

- $0 \leq i \leq (n-3)/4$: $\sigma_i = 0$ is avoided and the correction rotation method [19] is used for circular coordinates in rotation mode. The algorithm adopted for hyperbolic coordinates has not been found (the paper is too old), whereas linear coordinates have not been considered.
- $(n-3)/4 < i \leq (n+1)/2$: $\sigma_i = 0$ is accepted and specific operations are executed in this case:
  - $\sigma_i \neq 0$: execute the normal CORDIC equations 2.19, 2.20 and 2.22.
  - $\sigma_i = 0$:

$$x_{i+1} = x_i + m \cdot 2^{-2i-1} \cdot x_i \tag{2.33}$$
$$y_{i+1} = y_i + m \cdot 2^{-2i-1} \cdot y_i \tag{2.34}$$
$$z_{i+1} = z_i \tag{2.35}$$

- $i > (n+1)/2$: $\sigma_i = 0$ is accepted. If $\sigma_i \neq 0$, equations 2.19, 2.20 and 2.22 are executed, otherwise the coordinates are kept unchanged, since the scale factor approaches 1.

Since, for the second group, the scale factor of each iteration k_i can be assumed to be equal to $\sqrt{1+2^{-2i}} = 1 + 2^{-2i}$ within n-bit precision, when $\sigma_i = 0$ the vector is not being rotated, even if its amplitude is increased by the scale factor k_i of that iteration. This is taken into account later, since the final coordinates are scaled assuming a constant scale factor. For $i > (n+2)/2$, no correcting factor is required, as the scale factor becomes unity.

[20] describes two architectures, a pipelined one and an unfolded one that parallelizes the generation of the σ_i coefficients by using prediction. The first case can be used for both rotation and vectoring CORDIC modes in circular and hyperbolic coordinates.

Branching Method CORDIC [21]

Exploiting two modules (z^+ and z^-) that perform two conventional CORDIC iterations in parallel, this method restricts σ_i to {-1,1} without the need for extra rotations. Depending on the sign of z_i, there can be two possibilities:

- If the sign of z_i can be correctly determined, the two modules perform a rotation in the same direction.
- If the sign of z_i cannot be correctly found, a branching operation is performed, where the z^+ module executes a positive rotation ($\sigma_i = 1$), whereas z^- performs a negative one ($\sigma_i = -1$).

In both cases, only one result is chosen between the two paths and the direction of the rotation in the next iteration is decided by analyzing the three most significant bits of the smaller result between z_i^+ and z_i^-. Since this method does not require additional iterations to achieve a constant scale factor, it is faster with respect to the double or correcting rotation ones [19]. However, the two branching paths that work in parallel cause its its complexity to be higher.
This algorithm is only suitable to CORDIC algorithm in rotation mode and circular coordinates.

DCORDIC [22]

In all the previous methods, half of the effort in the x, y and z datapaths is required to correct possible errors deriving from the fact that the sign estimation is not perfect (due to redundant arithmetic). To avoid this, the differential CORDIC algorithm (DCORDIC) transforms the conventional CORDIC equations into partially fixed ones:

$$|\widehat{z_{i+1}}| = ||\widehat{z}_i| - \alpha_i| \tag{2.36}$$

$$x_{i+1} = x_i - sign(z_i)2^{-i}y_i \tag{2.37}$$

$$y_{i+1} = sign(z_i)2^{-i}x_i + y_i \tag{2.38}$$

As it can be noticed, the sign of z_i is only needed for the computation of x and y, whereas the angle accumulator only requires the absolute value of $\widehat{z}_i$. The sign of z_i (and so σ_i in rotation mode) can be determined by tracking the sign changes

with respect to the initial sign of z_0 during the absolute value computation of $\widehat{z}_i$. In DCORDIC, this technique is implemented through SBD arithmetic.
This algorithm can potentially support all the six modes of CORDIC algorithm. However, rotation and vectoring modes rely on different architectures.

Pipelined Radix-4 CORDIC [23]

This algorithm performs the following iteration equations

$$x_{i+1} = x_i + m \cdot (\sigma_{i,1} + \sigma_{i,2}) \cdot 4^{-i} \cdot y_i - m \cdot \sigma_{i,1} \cdot \sigma_{i,2} \cdot 4^{-2i} \cdot x_i \quad (2.39)$$

$$y_{i+1} = y_i - (\sigma_{i,1} + \sigma_{i,2}) \cdot 4^{-i} \cdot x_i - m \cdot \sigma_{i,1} \cdot \sigma_{i,2} \cdot 4^{-2i} \cdot y_i \quad (2.40)$$

$$z_{i+1} = z_i + (\sigma_{i,1} + \sigma_{i,2}) \cdot \alpha_{i,m} \quad (2.41)$$

where two successive radix-2 micro-rotations with the same angle are combined into a single radix-4 one. $\sigma_{i,1}$ and $\sigma_{i,2}$ are redundant radix-2 coefficients used to build the radix-4 one $\sigma_i \in \{-2, -1,0,1,2\}$, according to the relation $\sigma_i = \sigma_{i,1} + \sigma_{i,2}$. Table 2.6 shows this relation. The value of α_i is computed so that, for $1 \leq i \leq n-1$, $\alpha_0 = 2^{-1}$ and $\alpha_i = 4^{-i}$. Since the architecture relies on redundant SBD arithmetic, the selection function for σ_i uses the five most significant digits of the z coordinate, which means that the scale factor K is data-dependent and needs to be computed in each iteration with the expression

$$K = \prod_{i=0}^{n/2-1} k_i = \prod_{i=0}^{n/2-1} \sqrt{1 + |\sigma_{i,1}|4^{-2i}} \cdot \sqrt{1 + |\sigma_{i,2}|4^{-2i}} \quad (2.42)$$

$\boldsymbol{\sigma_i}$	$\boldsymbol{\sigma_{i,1}}$	$\boldsymbol{\sigma_{i,2}}$
0	1	-1
± 1	± 1	0
± 2	± 1	± 1

Table 2.6: Relation among the σ_i coefficients.

This algorithm is suitable for all the six CORDIC modes, whereas the proposed architecture can only be applied to the three coordinate systems in rotation mode, since for vectoring mode more complex modules are needed.

Redundant Radix 2-4 CORDIC [17]

[17] proposes a unified architecture able to work in all CORDIC modes, apart from linear coordinates, and to reduce the overall latency of about 25% with respect to a redundant radix-2 CORDIC rotation unit. The algorithm exploits CS arithmetic to solve different equations depending on the considered subset of iterations. In particular:

- For $1 \leq i < n/4$ nonredundant radix-2 CORDIC algorithm with $\sigma_i = \{-1,1\}$ is employed.
- For $n/4 \leq i < (n/2+1)$ the correcting iteration method [19] is used.
- For $i > n/2+1$ redundant radix-4 CORDIC algorithm is adopted, halving the number of iterations.

Since $\sigma_i = 0$ is avoided for $i \leq (n/2+1)$ and that, for $i > (n/2+1)$, $k_i = \sqrt{1+4^{-2i}} \approx 1$, the scale factor remains constant.

Radix-4 CORDIC [24]

In [24] a redundant radix-4 CORDIC algorithm based on CS arithmetic is proposed to reduce the latency compared to redundant radix-2. It uses the σ_i coefficients using two different techniques:

- For $0 \leq i < (n/6)$ σ_i is determined sequentially. Here micro-rotations are pipelined to increase the throughput.
- For $i > (n/6)$ the σ_i values are predicted using the remaining angle after the first $n/6$ iterations.

The scale factors are computed in advance and stored inside a ROM. For $\sigma_i^2 \in \{0,1,4\}$, the number of possible scale factors is $3^{n/4+1}$, causing the size and access time of the memory to increase with n. Therefore, only some scale factors are saved in the ROM and the remaining ones are computed from the stored values. This algorithm is only suitable for CORDIC in rotation mode and circular coordinates.

Unfolded Low Latency Radix-2 CORDIC [20]

The algorithm described in [20] has also an unfolded version that parallelizes the generation of the σ_i coefficients by using prediction, thus eliminating the sequential dependency of the z path. The directions are computed only for a group of iterations at a time, so that prediction error can be reduced. The convergence range is less

than $(\pi/2, -\pi/2)$ for this architecture, given that it does not allow rotations for index $i = 0$,
The main bottleneck of this implementation are the redundant to binary conversions of intermediate results within the z path, which limits the possible improvements that could be brought by pipelining it. Therefore, in order to reduce the overall latency, termination algorithm or Booth encoding method have been proposed in [16].
This algorithm is only suitable for CORDIC in rotation mode and circular coordinates.

P-CORDIC [25]

The P-CORDIC algorithm tries both to eliminate the sequential computation of the directions of rotation in CORDIC algorithm and to maintain a constant scale factor. To do so, it finds in advance the overall direction of micro-rotations before the actual beginning of CORDIC iterations in the x and y paths. This is achieved by relating the binary representation of the direction of rotations σ to the input rotation angle θ, given by

$$\sigma = 0.5 \cdot \theta + 0.5 \cdot c_1 + sign(\theta) \cdot \epsilon_0 + \delta \tag{2.43}$$

where $c_1 = 2 - \sum_{i=0}^{\infty}(2^{-i} - \tan^{-1}(2^{-i}))$, $\delta = \sum_{i=1}^{n/3}(\sigma_i \epsilon_i)$, $\epsilon_0 = 1 - \tan^{-1}(1)$ and $\epsilon_i = 1 - \tan^{-1}(2^{-i})$. δ is computed using the partial offset ϵ_i and the corresponding direction bit σ_i for the first $n/3$ iterations, since the value of ϵ_i decreases by a factor of 8 for $i > n/3$. By taking δ from a ROM memory, the directions of rotations for any input angle θ in binary form are obtained solving the aforementioned equation, thus canceling the z path, reducing the occupied area and obtaining better latency and hardware results than the radix-2 architecture proposed in [20].
This algorithm is only suitable for CORDIC in rotation mode and circular coordinates.

Flat CORDIC [26]

The flat CORDIC algorithm cancels the iterative nature of the x and y paths by transforming their recurrences into a parallelized version obtained by performing successive substitution, so that the final vectors can be expressed in terms of the initial ones, thus resulting only in a single equation. For example, for a precision of 16 bits the final coordinates become

$$
\begin{aligned}
x_{16} = [1 - \{&(\sigma_1\sigma_2 2^{-1}2^{-2} - ... - \sigma_1\sigma_{23}2^{-1}2^{-23} \\
&- \sigma_2\sigma_3 2^{-2}2^{-3} - ... - \sigma_9\sigma_{10}2^{-9}2^{-10}) \\
&+ (\sigma_1\sigma_2\sigma_3\sigma_4 2^{-1}2^{-2}2^{-3}2^{-4} + ... \\
&+ \sigma_2\sigma_3\sigma_4\sigma_5 2^{-2}2^{-3}2^{-4}2^{-5} + ... \\
&+ \sigma_3\sigma_4\sigma_6\sigma_7 2^{-3}2^{-4}2^{-6}2^{-7}) + E_{C-X}\}] \\
y_{16} = [&\sigma_1 2^{-1} + \sigma_2 2^{-2} + ... + \sigma_{16}2^{-16} \\
&- (\sigma_1\sigma_2\sigma_3 2^{-1}2^{-2}2^{-3} - ... - \sigma_5\sigma_7\sigma_8 2^{-5}2^{-7}2^{-8}) \\
&+ (\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5 2^{-1}2^{-2}2^{-3}2^{-4}2^{-5} + ... \\
&+ \sigma_2\sigma_3\sigma_4\sigma_5\sigma_6 2^{-2}2^{-3}2^{-4}2^{-5}2^{-6}) + E_{C-Y}]
\end{aligned}
\tag{2.44}
$$

where E_{C-X} and E_{C-Y} are the final error compensation factors and x_{in} and y_{in} are initialized with 1/K and 0 respectively. The 16 sigma values $(\sigma_1, \sigma_2, ..., \sigma_{16})$, for 16-bit precision, can be either 1 or -1, thus keeping the scale factor constant, and represent the direction of each micro-rotation needed to achieve the target angle. For the first $n/3$ iterations, they are obtained using the Split Decomposition Angle (SDA) technique, which limits the input angle range to $(0, \pi/4)$, whereas the last $2n/3$ σ_i are predicted from the remaining angle after $n/3$ iterations.
This algorithm is only suitable for CORDIC in rotation mode and circular and hyperbolic coordinates.

Para-CORDIC [27]

Employing the binary to bipolar representation (BBR) and micro-rotation angle recoding (MAR) techniques, the Para-CORDIC algorithm parallelizes the generation of the direction of rotations σ from the binary value of the input angle θ. By doing so, only the x and y coordinates datapaths remain iterative. In addition, the input angle θ is divided into two parts, θ_H and θ_L. The binary representation of the input angle θ is

$$
\theta = (-d_0) + \sum_{i=1}^{l-1} d_i 2^{-i} + \sum_{i=l}^{n} d_i 2^{-i} \tag{2.45}
$$

where $d_i \in \{0,1\}$ and $l = (n - log_2 3)/3$. The first $(l-1)$ bits of the input angle are converted into BBR and the MAR technique is used to find the σ coefficients σ_1 to σ_{l-1}. The values from σ_l to σ_{n+1} are obtained from BBR of the corrected θ_L, that is the resulting angle obtained by adding to θ_L the remaining one after the first $(l-1)$ rotations. By doing so, no ROMs are needed for storing the predetermined direction of rotations. However, it requires an array of adders to compute the corrected θ_L and some additional stages to perform correcting rotations.

This algorithm is only suitable for CORDIC in rotation mode and circular and hyperbolic coordinates.

Semi-Flat CORDIC [28]

The Semi-Flat CORDIC algorithm partially parallelizes the x, y and z paths to improve latency without increasing the area requirements. For the first λ bits of σ_i, the x and y recurrences are computed iteratively using the double rotation method, resulting in $x_{\lambda-1}$ and $y_{\lambda-1}$. Then x_{n-1} and y_{n-1} can be expressed in terms of these $x_{\lambda-1}$ and $y_{\lambda-1}$, if all the σ_i have been predicted. The σ_i for the $n_{int}/3 - \lambda$ bits of the input angle, where n_{int} is the internal precision chosen higher than the external one to cope with quantization errors, are precomputed and stored in a ROM, which is addressed by $n_{int}/3 - \lambda$ bits of the input angle. The remaining $2n_{int}/3$ number of σ_i are predicted from the rotation angle. After λ iterations, all the terms of x_n and y_n are then added using the Wallace tree, flattening the x and y paths. As a natural consequence, computation time and area are affected by the choice of λ. In [28] it is shown that the best trade-off is obtained by $\lambda = 6$ for a 16-bit precision ($n_{int} = 22$) and $\lambda = 8$ for a 32-bit one ($n_{int} = 39$). This algorithm reaches constant scale factor, as the σ_i are chosen between {-1,1}.
[28] does not provide any description or reference on how to precompute the $n_{int}/3 - \lambda$ number of σ_i.
This algorithm is only suitable for CORDIC in rotation mode and circular, linear and hyperbolic coordinates.

2.5.2 Floating-point CORDIC

Floating-point Vector-Arithmetic Unit [29]

[29] describes a floating-point arithmetic unit based on CORDIC algorithm. In particular, it supports all its six available modes and this makes the proposed implementation the most complete one. This can be done thanks to the availability of control and selection logic circuits that decide which mode and which coordinate system have to be employed. The proposed architecture uses global FLP normalization, with an internal extended FXP format that takes care both of overflow and loss of accuracy. This implies different formats for each datapath:

- x/y 32-bit datapath:
  - 1 sign bit
  - 3 overflow bits
  - 23 mantissa bits
  - 5 guard bits

- z 29-bit datapath:
  - 1 sign bit
  - 2 overflow bits
  - 23 mantissa bits
  - 3 guard bits

The presented architecture is a 44-stage pipeline that has been created to work with single-precision numbers and is structured as follows:

- **Front stage**: it converts the inputs from FLP to FXP, adapts them to the operation to be executed and computes the reference exponent.
- **CORDIC pipeline**: it takes care of 29 iterations by performing standard CORDIC equations. To cope with convergence issues when working with hyperbolic coordinates, it uses a specific shifting sequence throughout the execution.
- **Scaling pipeline**: it compensates the scaling factor K by performing, in 8 stages, successive multiplications by (1 ± 2^{-j}).
- **End stage**: it performs the conversion from FXP to FLP by adjusting exponent and mantissa and packing everything together.

It has to be noted that the scaling stages do not use a multiplier, but simply reuse the same micro-rotation stages architecture, opportunely modified with multiplexers to suit the variable shifts that depend on a variable $a(m, i)$ described in the paper.

Unified reconfigurable CORDIC processor [30]

In [30] a CORDIC processor for floating-point arithmetic is proposed, able to a variety of operations. In fact, among the six available modes for CORDIC algorithm, only the rotation one in hyperbolic coordinates is not supported. This means that the processor is capable of computing

- Sine/cosine (rotation mode and circular coordinates)
- Arctangent (vectoring mode and circular coordinates)
- Multiplication (rotation mode and linear coordinates)
- Division (vectoring mode and linear coordinates)
- Square-root (vectoring mode and hyperbolic coordinates).

The proposed architecture uses global FLP normalization, with an internal 25-bit FXP format (1-bit sign, 1-bit integer part, 23-bit mantissa), and exploits parallel paths in the second half of iterations. To do so, the pipeline is divided as follows:

- **Pre-processing module**: it converts the inputs from FLP to FXP, adapts them to the operation to be executed and computes the reference exponent. In case of trigonometric functions in circular coordinates, it also extends the range of convergence of θ, in order to cover $\{0, 2\pi\}$.
- **Rotation Unit A**: it takes care of half of the iterations by performing standard CORDIC equations.
- **Scaling circuit**: it completes the compensation of the scaling factor K for square-root operations.
- **Rotation Unit B**: exploiting the binary-to-bipolar recoding technique, it computes the directions of the second half of rotations ([27]), so that parallel computation of x and y can be made.
- **Post-processing module**: it performs the conversion from FXP to FLP by adjusting exponent and mantissa and packing everything together.

It has to be noted that, apart from the Scaling circuit module, no scaling factor compensation has been included for operations that are not square-root. This is because the architecture has been structured to work with specific values of x_0 and y_0 where K is already taken into account. Therefore, in order to generalize it, a scaling circuit can also be included at the end of the iteration path. In addition, since it does not compute hyperbolic functions, no iterations need to be repeated to make the algorithm converge. As a result, the shifts in Rotation Unit A are hardwired.

2.5.3 General comparison

Tables 2.7 and 2.8 briefly compare the most important characteristics of all the algorithms analyzed up to now, focusing on aspects related to the algorithms themselves and to the differences among the architectural implementations respectively. In Table 2.7, SBD stands for Signed-Bit Digit arithmetic, CS for Carry-Save arithmetic and 2's comp. for nonredundant 2's complement arithmetic (nonredundant).

Algorithm	Radix	Arithmetic	Range of θ	Scale factor	Iterative x/y path	Iterative z path
Double rotation/Correcting [19]	2	SBD	$0, \pi/4$	Constant	✓	✓
Low Latency [20]	2	CS	$-\pi/2, \pi/2$ [2]	Constant	✓	✓
Branching [21]	2	SBD	$-\pi/2, \pi/2$ [2]	Constant	✓	✓
DCORDIC [22]	2	SBD/CS	$-\pi/2, \pi/2$ [2]	Constant	✓	✓
Radix-4 [23]	4	SBD	$-\pi/2, \pi/2$ [2]	Variable	✓	✓
Radix2-4 [17]	2-4	CS	$-\pi/2, \pi/2$ [3]	Constant	✓	✓
Radix-4 [24]	4	CS	$-\pi/2, \pi/2$	Variable	✓	n/6
PCORDIC [25]	2	SBD	$-\pi/2, \pi/2$ [2]	Constant	✓	×
Flat CORDIC [26]	2	SBD	$0, \pi/4$	Constant	Combinational	×
Para-CORDIC [27]	2	CS	$-\pi/4, \pi/4$	Constant	✓	×
Semi-flat [28]	2	SBD	$-\pi/2, \pi/2$ [2]	Constant	λ/Combinational	λ
Nonredundant low-latency [16]	2	2's comp.	$-\pi/2, \pi/2$	Constant	(n/2 + 1)/Multiplier	n/3
Pipelined unified [18]	2	2's comp.	$-\pi/2, \pi/2$	Constant	✓	✓
Unified reconfigurable FLP [30]	2	2's comp.	$0, 2\pi$	Constant	✓	✓
FLP Vector-Arithmetic unit [29]	2	2's comp.	$-\pi/2, \pi/2$ [2]	Constant	✓	✓

Table 2.7: Algorithms comparison.

The following assumptions have been made:

1. Since most of the papers express latency and area in terms of full adders, this has been used as comparison unit
2. [19]:
   - The latency does not include the numbers conversion and the scale factor compensation, as described in [22].
   - Since the area details are not expressed, they have been taken from [28],

[2]The paper of the algorithm does not provide any information on the range of θ, so the standard range of conventional CORDIC algorithm has been considered

[3]To reach this range, the micro-rotation with $i = 1$ has to be repeated

Algorithm	**Latency (t_{FA})**	**Area (A_{FA})**	**Error**	**Supported modes with the same architecture**					
				Rotation			**Vectoring**		
				Circ	**Hyp**	**Lin**	**Circ**	**Hyp**	**Lin**
Double rotation [19] Correcting [19]	$3.75n$ [22]	$16.5n(n+log_2n)+2n$ [28]	2^{-n}	✓					
Low Latency Nonpipelined [20]	$n+log_3(n)+3log_2(n)-2$	$6(n+log_2n)(n+log_3(n))+24n$ [28]	2^{-n}	✓	✓	✓			
Low Latency Pipelined [20]	$(9n-3)/8$		2^{-n}	✓	✓		✓	✓	
Branching [21]	$2.5n$	$2*[16.5n(n+log_2n)+2n]$	2^{-n}	✓					
DCORDIC [22]	$3.5n+1$ (rot) $4n+log_2n+5$ (vec)	$n[10(n+log_2n)+2n]$	2^{-n}						
Radix-4 [23]	$4(3/4n+1)$	$2n(3n+3)+n^2(log_2n+1)/2+1.7n^2$	2^{-n}	✓	✓	✓			
Radix2-4 [17]	$(n/4-1)3.8+$ $+(n/4+2)4.3+$ $+(n/4)4.7$	$(n/4-1)(7.5n+16.5)+$ $+(n/4+2)(8.5n+14)+(n/4)(7n+10.5)$	2^{-n}	✓	✓		✓	✓	
Radix-4 [24]	$(2n/3+3)5$	$2(n+log_2n)(2n+14/6n+6)$	2^{-n}	✓					
PCORDIC [25]	$1.7n+1.25+log_2n$	$2^{\lceil n/5-2\rceil}(0.83+n/50)+$ $+(7n/15+5)(2n/5+6)+$ $+n(n/2+3)$	2^{-n}	✓					
Flat CORDIC [26]	34 for 16-bit 39 for 24-bit 50 for 32-bit	1850 for 16-bit 3156 for 24-bit 5499 for 32-bit	2^{-n}	✓	✓				
Para-CORDIC [27]	$2(s(n)+n/2-m+2)+$ $+log_{3/2}(n+2)$	$(n-m-1)(m-1)+$ $+[4n(s(n)+n/2-m+2)+$ $+2n(n+1)]$	2^{-n}	✓	✓				
Semi-flat [28]	33 for 16-bit, 49 for 32-bit	1622 for 16-bit, 4619 for 32-bit	2^{-n}	✓	✓	✓			
Nonredundant Low Latency [16]	$(n/2+3)[(log_2b)/2+1]t_{FA}+$ $+t_{FA}+t_{Wallace}$	$(n/2+6)[(blog_2b-b+1)/2+b]A_{FA}+$ $+\lceil(n/2+log_2n)/2\rceil[(n/2+log_2n)+1]A_{FA}+$ $+A_{Wallace}+A_{CSD\ mult}$	2^{-n}	✓			✓		
Pipelined unified [18]	$n(log_2n+1)$	$n[2n^2/7+n(2log_2n-1/7)+2]$	10^{-2} for 16-bit	✓	✓		✓	✓	
Unified Reconfigurable FLP [30]	$log_2n(3/2+n/2)+3/2n+2$	$[log_2n(2+2n^2+n)+4n^2+6n+77]A_{FA}+A_{scale}$	10^{-4} for 16-bit 10^{-6} for 24-bit	✓		✓	✓	✓	✓
FLP Vector-Arithmetic Unit [29]	$20log_2n+76+n/2$	$40nlog_2n+65n+1.3n^2+76$	2^{-n}	✓	✓	✓	✓	✓	✓

Table 2.8: Architectures comparison.

where

$$N_{ext} = \text{external precision} = n \tag{2.46}$$
$$N_{int} = \text{internal precision} = n + log_2 n \tag{2.47}$$
$$A_{CPA} = \text{area of a carry propagating adder} = nA_{FA} \tag{2.48}$$
$$A_{MUX} = \text{area of a 2-1 MUX} \approx A_{XOR} \approx A_{FA}/2 \tag{2.49}$$

A_{SS} is the area for the sign estimation, which has been assumed to be the same as a 4-1 MUX, hence $A_{SS} \approx 1.5A_{FA}$.

3. [20] Nonpipelined:

- The version with termination algorithm has been considered. The standard one has a latency of $(2n + log_3(n) - 1 + log_2(n))t_{FA}$.
- Since the area details are not expressed, they have been taken from [28]. The same considerations made for [19] hold.

4. [20] Pipelined:

   - No area details have been found this implementation. Most of the papers focus on its unfolded version.
   - Only an algorithm to be used with redundant arithmetic that relies on different implementations for different coordinate systems is proposed in [20]. Hence, to have a unified architecture that implements this algorithm, custom architectures of redundant adders and rotation direction selection circuits need to be made.

5. [21]:

   - It has only been found from [8] that the latency is nt_{stage}, with t_{stage} unknown, since [21] only provides a latency in terms of an On-Line delay of 6. Therefore, since [21] uses the same arithmetic as [20], it has been assumed an iteration delay $t_{stage} = 2.5t_{FA}$. This similarity can also be seen in [24].
   - No area details have been found, but, due to its duplicated architecture and thanks to the graphs in [24], it has been assumed that its area occupation is roughly twice as [19] within 32 bits.

6. [22]:

   - It supports all the modes, but it needs different architectures for rotation and vectoring.
   - Since the area details were not sufficient, they have been taken from [26]. For [22], no details were given about the area for the absolute value computation, which has then been assumed to be the one of two 4-1 MUXs for each bit. Therefore $A_{abs} \approx 2nA_{4-1\ MUX} \approx 4nA_{XOR} \approx 2nA_{FA}$.

7. [23]:

   - Since the area details were not generalized, they have been interpolated from the available data, assuming $A_{red.\ adder} = nA_{4-2\ comp} = 2nA_{FA}$, $A_{2-1\ MUX} = A_{FA}/2$ and $A_{3-1\ MUX} = A_{FA}$.

8. [17]:

- In [17] the delays of each stage are expressed in terms of NAND gates. Hence, it has been used $t_{FA} = 6T_{NAND}$. In addition, the delay due to the path through registers ($T_{reg} = 8T_{NAND}$) has been ignored to consider the combinational part only.
- Since the area details in [17] were not sufficient, they have been computed looking at all the adopted stages and assuming

$$A_{4-bit\ CLA} \approx 6A_{FA} \quad (2.50)$$
$$A_{6-bit\ CLA} \approx 10.5A_{FA} \quad (2.51)$$
$$A_{7-bit\ CLA} \approx 14A_{FA} \quad (2.52)$$
$$A_{2-1\ MUX} \approx A_{FA}/2 \quad (2.53)$$
$$A_{4-1\ MUX} \approx A_{FA} \quad (2.54)$$

9. [24]:
   - In [24] the delay of each stage is expressed in terms of unitary gate delay t_g and is equal to $21t_g$. Given that standard logic implementations for the different hardware blocks have been considered in this paper, it has been assumed $t_{FA} = 2t_{XOR} = 4t_g$, where a 2-input XOR gate has a delay of $2t_g$. Therefore $t_{stage} \approx 5t_{FA}$.
10. [27]:
    - $s(n)$ and m can be derived from [27]. In particular, $s(16) = 5$, $s(24) = 10$ and $s(32) = 18$, whereas $m = \lceil (n - log_2 3)/3 \rceil$.
11. [16]:
    - In [16] the total delay is $(n/2 + 2)t_{adder} + t_{multiplier}$, where t_{adder} is the delay of a b-bit Kogge-Stone prefix adder, with $b = 3 + n - 1 + log_2 n$ (b is the internal datapath width), and $t_{multiplier}$ the one of a $n/2 + log_2 n$ multiplier based on radix-4 recoding and 4-2 CSAs. For the KS adder, the delay is $log_2 b + T_{PG\ block}$, with $T_{PG\ block} = T_{AND} + T_{OR} = T_{XOR} = t_{FA}/2$, for the carry generation network and t_{FA} for the final sum. Therefore, $t_{adder} = log_2 b/2$. For what concerns the multiplier, instead, depending on the external precision and on the adder tree, its latency will be $t_{Wallace} + t_{adder} + t_{ENC+MUX}$. Assuming to use a KS adder for the redundant to nonredundant conversion, t_{adder} will be the same as the aforementioned KS adder, whereas $t_{ENC+MUX}$ can be assumed to be

$$t_{ENC+MUX} = t_{ENC} + t_{MUX} = 2t_{AND} + 2t_{OR} + t_{XNOR} \approx t_{FA} \quad (2.55)$$

      For $n = 16$ and $n = 24$, $t_{Wallace} = 2t_{4-2} = 3t_{FA}$ whereas for $n = 32$ $t_{Wallace} = 3t_{4-2\ comp} = 4.5t_{FA}$.

- The same considerations of previous point hold here. In particular, there are a total of $n/2 + 6$ KS adders (including the ones at the end of the multipliers) with

$$\begin{aligned} A_{adder} &= (blog_2b - b + 1)A_{PG\ block} + bA_{FA} \\ &\approx [(blog_2b - b + 1)/2 + b]A_{FA} \end{aligned} \tag{2.56}$$

  assuming $A_{PG\ block} \approx A_{XOR}$, two multiplier based on radix-4 recoding and 4-2 CSAs with $A_{multiplier} = A_{ENC+MUX} + A_{Wallace} + A_{adder}$ and three CSD constant multipliers with $A_{CSD\ mult}$. Assuming the number of Booth encoders to be $\lceil (n/2 + log_2n)/2 \rceil$, where $(n/2 + log_2n)$ is the size of the multiplier, with $A_{ENC} = 2A_{AND} + A_{OR} + A_{XOR} \approx A_{FA}$ [31] and the number of MUXs to be $(\lceil (n/2 + log_2n)/2 \rceil)((n/2 + log_2n))$, with $A_{MUX} = 2A_{AND} + A_{NOR} + A_{XNOR} \approx A_{FA}$ [31], $A_{ENC+MUX} = (\lceil (n/2 + log_2n)/2 \rceil)((n/2 + log_2n) + 1)A_{FA}$. $A_{Wallace}$, instead, can be approximated to be between $150A_{FA}$ and $300A_{FA}$. $A_{CSD\ mult}$ depends on the specific constant to be multiplied.

12. [30]: no details are given about latency or area occupation, apart the ones related to the implementation in FPGA. As a consequence, latency and area have been computed according to the architecture block diagram and the following assumptions:

    - Latency (the architecture is assumed to be unpipelined to compute the total latency. Pipelining can be applied):
      - The pre-processing stage has a critical path

$$T_{PRE} = 2T_{adder} + T_{shifter} + 2T_{4-1\ MUX} \tag{2.57}$$

        Assuming a KS $(n + 2)$-bit adder ($n + 2$ is the length of the used FXP format) to emulate [16], $T_{adder} = [(log_2n)/2 + 1]t_{FA}$. $T_{shifter}$ and $T_{4-1\ MUX}$ have been computed according to the model expressed in [24], therefore $T_{shifter} = (log_2n)/2t_{FA}$ and $T_{4-1\ MUX} = t_{FA}$. So, $T_{PRE} = (log_2n + 2)t_{FA}$.
      - The Rotation Unit A has a latency $T_{RUA} = T_{adder} + T_{4-1\ MUX} = [(log_2)n/2 + 2]t_{FA}$.
      - The Rotation Unit B has a critical path $T_{RUB} = \lceil log_2(n/2) \rceil T_{adder} + T_{2-1\ MUX} = \lceil log_2(n/2) \rceil [(log_2n)/2 + 1]t_{FA}$.
      - The scaling unit can be created ad hoc by using a constant multiplier that depends in K and is not in the critical path. Hence, it has not been included in the computation.

  - The post-processing unit has a latency $T_{POST} = T_{4-1\ MUX} + T_{LOD} + 2T_{shifter} = (n + log_2 n)/2t_{FA}$, assuming $T_{LOD} = (n-2)T_{2-1\ MUX} + T_{AND} \approx (n-2)/2t_{FA}$.
  - Since $(n/2)T_{RUA} > T_{RUB}$ has been assumed for the second half of iterations, the critical path lays in $T_{CP} = T_{PRE} + nT_{RUA} + T_{POST}$.

- Area:
  - The pre-processing unit has an area

$$\begin{aligned} A_{PRE} = 4A_{n-bit\ adder} + 3A_{9-bit\ adder} + 3A_{shifter} + \\ + 8A_{4-1\ MUX} + A_{COMP} \end{aligned} \tag{2.58}$$

    Assuming again KS adders, $A_{n-bit\ adder} = (nlog_2 n - n/2 + 1)A_{FA}$ and $A_{9-bit\ adder} = 25A_{FA}$. In addition, $A_{shifter} = 1/7n^2 A_{FA}$ ([24]) and $A_{4-1\ MUX} = 6/7A_{FA}$ ([24]). Therefore,

$$A_{PRE} = [4(nlog_2 n - n/2 + 1) + 75 + 3/7n^2 + 48/7n]A_{FA} \tag{2.59}$$

  - The Rotation Unit A occupies $A_{RUA} = 3A_{n-bit\ adder} + 2A_{4-1\ MUX} + A_{2-1\ MUX} = (nlog_2 n - n/2 + 15/7n)A_{FA}$.
  - The Rotation Unit B has an area $A_{RUB} = 2(n/2-1)(A_{n-bit\ adder} + 2A_{2-1\ MUX}) = (n-2)(nlog_2 n - n/2 + 1 + 12/7n)A_{FA}$
  - The scaling unit has not been included for the same reasons as for latency.
  - The post-processing unit occupies

$$A_{POST} = 2A_{4-1\ MUX} + A_{n-bit\ adder} + 3A_{LOD} + 6A_{shifter} \tag{2.60}$$

    where

$$\begin{aligned} A_{LOD} &= (n-2)A_{2-1\ MUX} + (n-1)A_{AND} \\ &= [(n-2)3/7 + (n-1)/6]A_{FA} \end{aligned} \tag{2.61}$$

  - The total area is $A_{TOT} = A_{PRE} + nA_{RUA} + A_{RUB} + A_{scale} + A_{POST}$.

13. [18]: no details are given about latency or area occupation, apart the ones related to the implementation in FPGA. As a consequence, latency and area have been computed according to the architecture block diagram and the following assumptions:

    - Latency (the architecture is assumed to be unpipelined to compute the total latency. Pipelining is applied in [18]):

  - Each STEP block has a latency $T_{STEP} = T_{shifter} + T_{adder}$, where $T_{shifter}$ and T_{adder} have been computed in the same way as [30]. Therefore, $T_{STEP} = (log_2 n + 1)t_{FA}$ and $T_{CP} = nT_{STEP} = n(log_2 n + 1)t_{FA}$.
- Area:
  - Each STEP block has an area

$$A_{STEP} = 2A_{shifter} + 2A_{adder} + 2A_{2-1\ MUX} \tag{2.62}$$

    where $A_{shifter}$, A_{adder} and $A_{2-1\ MUX}$ have been computed in the same way as [30]. Therefore,

$$A_{STEP} = [2/7n^2 + n(2log_2 n - 1/7) + 2]A_{FA} \tag{2.63}$$

$$\begin{aligned} A_{TOT} &= nA_{STEP} \\ &= n[2/7n^2 + n(2log_2 n - 1/7) + 2]A_{FA} \end{aligned} \tag{2.64}$$

14. [29]: the architecture (assumed unpipelined to compute the total latency. Pipelining is applied in [29]) is similar to the one [30], hence the latency and area formulas have been computed with the following assumptions:
    - The front-stage has the same area and latency as the pre-processing block of [30] but without the part that extends the range of conversion of θ (which is not in the critical path). Therefore, $T_{FS} = (log_2 n + 2)t_{FA}$ and $A_{FS} = [3(nlog_2 n - n/2 + 1) + 75 + 3/7n^2 + 36/7n]A_{FA}$.
    - The each stage of the CORDIC pipeline has the same area and delay as Rotation Unit A, so $T_{C-STAGE} = [(log_2 n)/2 + 2]t_{FA}$ and $A_{C-STAGE} = (n_l og 2n - n/2 + 15/7n)A_{FA}$. As stated in [29], the scaling stages are pretty similar to the CORDIC ones and, so, have been considered equivalent to them.
    - The end-stage has the same area and latency as the pre-processing block of [30] but without the part that takes care of the range of θ. Therefore, $T_{ES} = (n + log_2 n)t_{FA}$ and $A_{ES} \approx (2.6n - 1.8 + 6/7n^2)A_{FA}$.
    - The total latency and area are, then,

$$\begin{aligned} T_{CP} &= T_{FS} + 37T_{C-STAGE} + T_{ES} \\ &= (20log_2 n + 76 + n/2)t_{FA} \end{aligned} \tag{2.65}$$

$$A_{TOT} = (40nlog_2 n + 65n + 1.3n^2 + 76)A_{FA} \tag{2.66}$$

    for single-precision ($n = 32$),

$$\begin{aligned} T_{CP} &= T_{FS} + 25T_{C-STAGE} + T_{ES} \\ &= (12.5log_2 n + 46 + n/2)t_{FA} \end{aligned} \tag{2.67}$$

Algorithm	Latency (t_{FA})			Area (A_{FA})		
	16 bits	24 bits	32 bits	16 bits	24 bits	32 bits
Double rotation/Correcting [19]	60	90	120	5312	11367	19600
Low Latency Nonpipelined [20]	26	39	48	2606	5188	8572
Low Latency Pipelined [20]	18	27	36			
Branching [21]	40	60	80	10624	22734	39200
DCORDIC [22]	57 (rot) 73 (vec)	85 (rot) 126 (vec)	113 (rot) 158 (vec)	3712	8013	13888
Radix-4 [23]	52	76	100	2708	6188	11149
Radix2-4 [17]	56	82	107	1800	3798	6532
Radix-4 [24]	68	95	122	3014	6289	10705
PCORDIC [25]	33	47	61	334	622	1013
Flat CORDIC [26]	34	39	50	1850	3156	5499
Para-CORDIC [27]	28 [1]	32 [2]	31 [3]	1224	2841	5512
Semi-flat [28]	33		49	1622		4619
Nonredundant low-latency [16]	39	57	75	$978A_{FA}$+ $+ A_{CSD\ mult}$	$1781A_{FA}$+ $+ A_{CSD\ mult}$	$2833A_{FA}$+ $+ A_{CSD\ mult}$
Pipelined unified [18]	80	134	192	3213	9197	19520
Unified reconfigurable FLP [30]	27	39	51	$3317 + A_{scale}$	$7926 + A_{scale}$	$14775 + A_{scale}$
FLP Vector-Arithmetic unit [29]	103	304		2680	6786	

Table 2.9: Architectures comparison for 16, 24 and 32 bits.

$$A_{TOT} = (25nlog_2n + 42n + 1.3n^2 + 76)A_{FA} \tag{2.68}$$

for half-precision ($n = 16$).

[1] For 16 bits, $s(16) = 5$ and $m = 5$.

[2] For 24 bits, $s(24) = 10$ and $m = 8$.

[3] For 16 bits, $s(32) = 18$ and $m = 11$.

Chapter 3

Proposed architecture

3.1 Preliminary choices

In order to select the most suitable architecture for the final implementation, several design choices have been made, based on the requirements and on the analysis of the state of the art. In particular, the following aspects have been taken into account.

3.1.1 Supported floating-point formats

Based on requirements, the architecture must support both single and half precision floating-point. In particular, given the presence of a 32-bit datapath, the main focus is on allowing the execution of operations either on a single set of 32-bit inputs or on two sets of 16-bit ones in parallel. Therefore, even though the designed unit can be easily adapted to other formats thanks to the use of parametric modules, the main focus has been put on these two configurations.

3.1.2 Algorithm selection

In Section 2.5 several algorithms have been explained, each of which has its own pros and cons. For this work, one major requirement is the possibility, for the final architecture, to accomplish all the six CORDIC modes, in order to reach the largest variety of executable functions. As a result, most of the approaches, as much as efficient, have been discarded, since they are able only to perform a subset of the available modes. In addition, also the idea of using two complementary blocks that, at the end, reach the complete set of functions has been abandoned, since it would lead to twice the area occupation. Therefore, it has been chosen to follow the traditional algorithm implementation together with the unfolded and

pipelined architectural approach explained in Section 2.3.3, in order to combine area-efficiency and high throughput with a large number of available functions. As a result, the scaling factor is constant and known in advance, since nonredundant arithmetic is being adopted, and there are no data dependencies among the various stages, which means that, at each clock cycle, not only operations on new inputs can start, but also valid results are available.

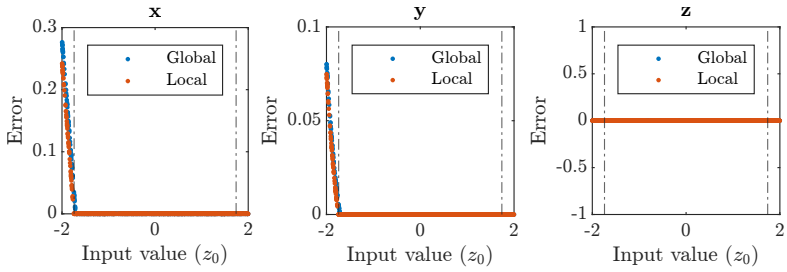
3.1.3 Floating-point extension strategy

As explained in Section 2.4, there are two ways to adapt CORDIC algorithm to floating-point arithmetic: global FLP and local FLP. In order to compare the strategies and select the most suitable one, a C code for both implementations has been created and tested over 1000 random values. For what concerns global floating-point, the approach proposed in [29] has been followed to build the model. Figures 3.1-3.4 summarize the obtained results for both precisions and for all the CORDIC working modes. In particular, Figures 3.1 and 3.3 refer to rotation mode and, thus, plot the obtained absolute error over the ROC of the z input value[1]. Figures 3.2 and 3.4, instead, are related to vectoring mode and, thus, represent the variation of the absolute error over the range of the y input value. At first glance, local FLP could seem to be the most suitable solution, since it adopts floating-point arithmetic directly within the computational blocks, thus avoiding precision loss due to fixed-point limitations. However, as clearly shown in Figures 3.1-3.4, both strategies provided similar results in terms of error, with global floating-point showing an even better behavior outside the range of convergence. As a result, thanks to its simplicity and possibly higher speed, global FLP has been chosen for the final architecture. In particular, [29] has been followed as first reference for single precision, even though other configurations for the internal fixed-point format have been explored. For what concerns half precision, instead, the number of padding bits for the FXP datapath have been first interpolated from [12, 29] and then evaluated through simulations.

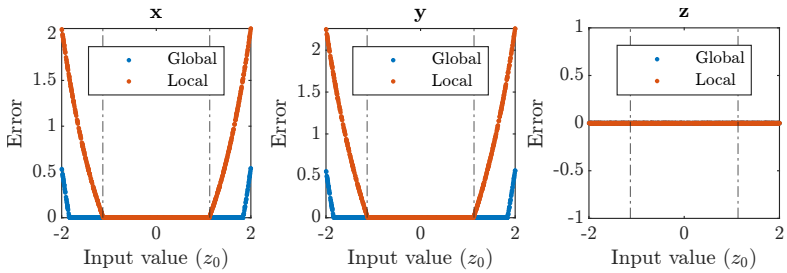
3.1.4 Scaling factor compensation technique

Traditional CORDIC architectures lay on two main techniques to compensate for the amplitude variation of the input vector during the iterations: pre-scaling and post-scaling. The former consists of assigning to the input vector pre-scaled x and y coordinates, whereas the latter multiplies the output vector by the inverse of the scaling factor K.

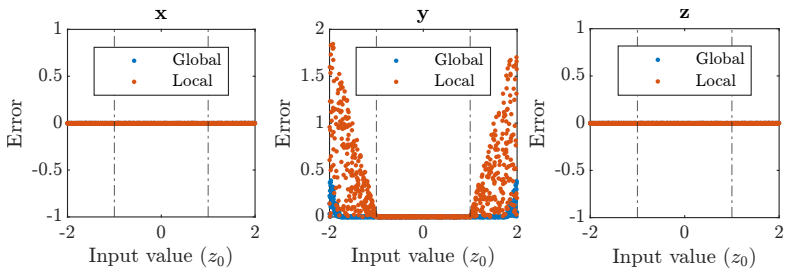
[1] In order to make this preliminary comparison faster, it has been decided to make the input that defines the ROC vary, that is z for rotation mode and y for vectoring mode



(a) Circular coordinates



(b) Hyperbolic coordinates

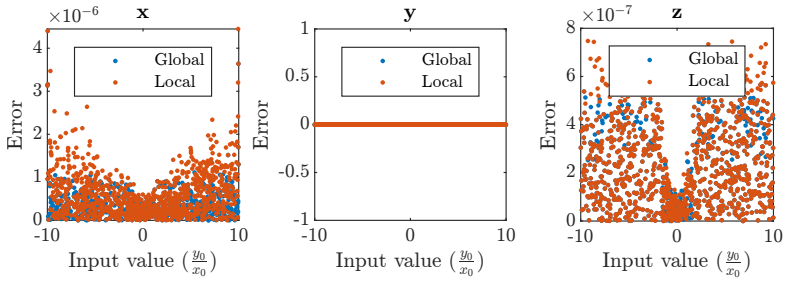


(c) Linear coordinates

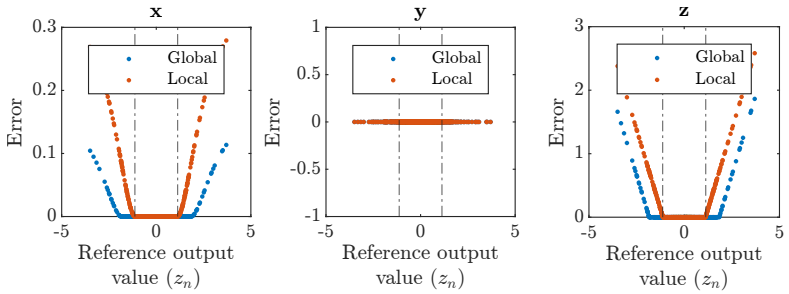
Figure 3.1: Error comparison in rotation mode between global (blue) and local (red) floating-point for single precision. The black vertical lines show the range of convergence according to Table 2.5.

Thanks to the absence of a scaling circuitry, the first method provides a clear advantage in terms of area occupation and latency, but it offers a lower adaptability, since it forces the user to select already pre-scaled input values to obtain the expected result. The second technique, instead, is far more flexible and, for this reason, has been chosen for the final architecture.
According to [8], post-scaling can be implemented in two ways:

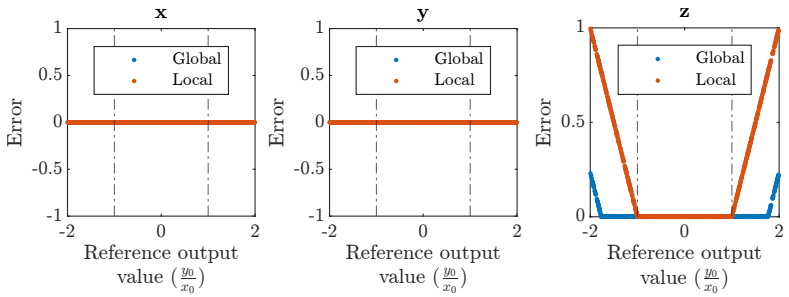
1. **Multiplier-based**: the output vector is scaled by $1/K$ through a dedicated



(a) Circular coordinates



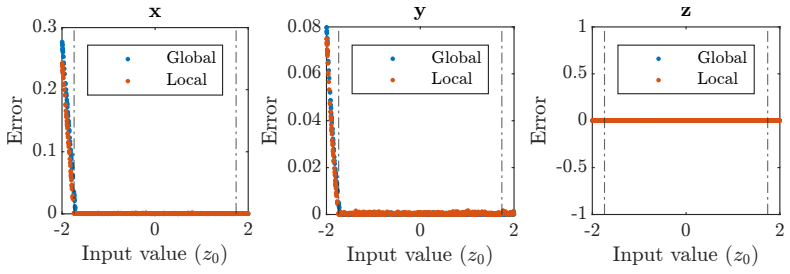
(b) Hyperbolic coordinates



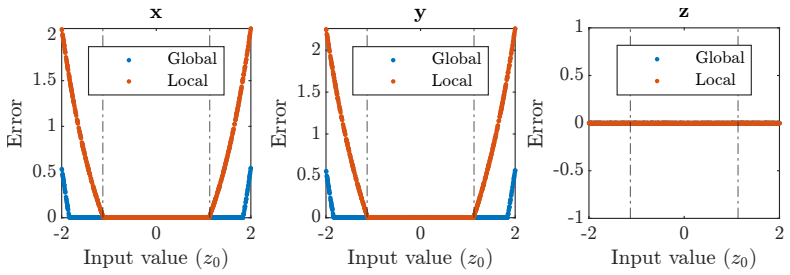
(c) Linear coordinates

Figure 3.2: Error comparison in vectoring mode between global (blue) and local (red) floating-point for single precision. The black vertical lines show the range of convergence according to Table 2.5.

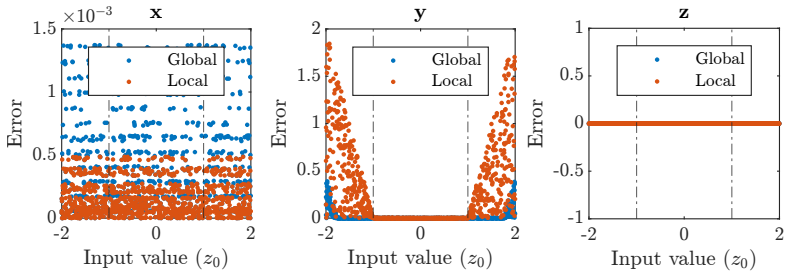
multiplier, which, as suggested by [8] to improve latency, can be either based on canonical signed digit (CSD) representation or on a Wallace tree. This approach is the most straightforward one, but the required multiplier represents a complex and area-consuming circuitry. In addition, since, in order to improve the throughput, the final architecture will be an unfolded and pipelined one, the multiplier constitutes a slow stage that limits the maximum achievable



(a) Circular coordinates



(b) Hyperbolic coordinates

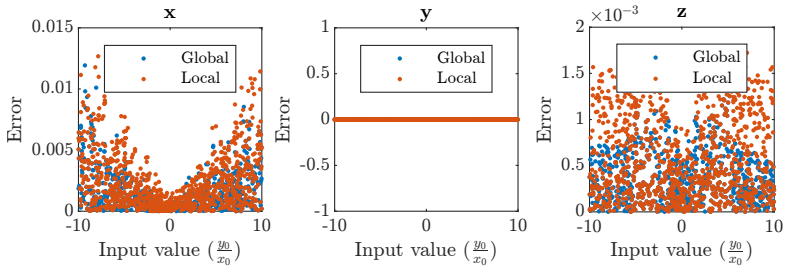


(c) Linear coordinates

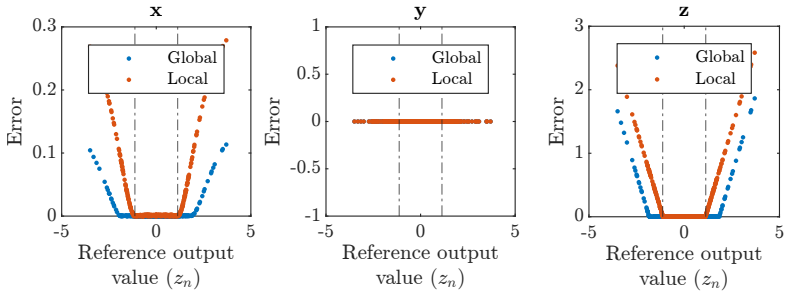
Figure 3.3: Error comparison in rotation mode between global (blue) and local (red) floating-point for half precision. The black vertical lines show the range of convergence according to Table 2.5.

frequency and makes the pipeline unbalanced, thus requiring to be internally pipelined as well.

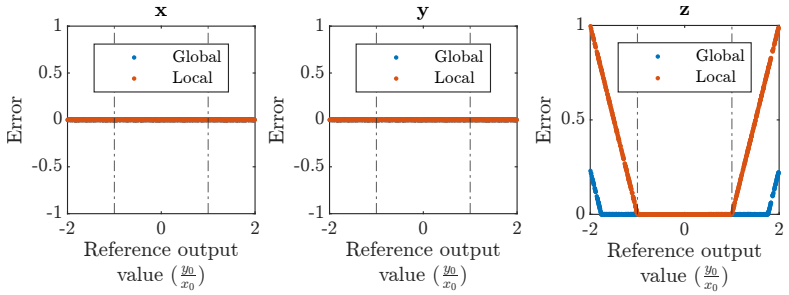
2. **Performing additional iterations**: the output vector is scaled by $1/K$ by performing additional iterations, which represents a more complex approach than the previous one, since it requires a specific sequence of iterations that accomplishes this task. On the other hand, based on what has been explained in [29], each scaling stage can be almost identical to the traditional CORDIC



(a) Circular coordinates



(b) Hyperbolic coordinates



(c) Linear coordinates

Figure 3.4: Error comparison in vectoring mode between global (blue) and local (red) floating-point for half precision. The black vertical lines show the range of convergence according to Table 2.5.

.

ones, thus allowing not only to increase the maximum achievable frequency, but also to balance the pipeline.

In this work both strategies have been analyzed during simulation. However, since the resulting average relative error for both of them was almost identical, the second approach has been chosen for the final architecture, as it allows to improve

the maximum achievable frequency and to balance the pipeline. In particular, [29] has been followed for the design of the scaling circuitry.

3.2 Architecture overview

As previously declared, the following choices have been taken:

- Single and half precision floating-point have been selected as supported formats.
- Global FLP has been chosen over the local one for the floating-point extension of CORDIC algorithm.
- The unfolded and pipelined approach has been preferred over the folded one, in order to maximize the throughput. Therefore, there are no data dependencies among the various stages, which means that, at each clock cycle, not only operations on new inputs can start, but also valid results is available.
- Post-scaling through additional iterations has been selected as scaling factor compensation technique, since it is a good trade-off among pipeline balance, maximum achievable frequency and flexibility of the architecture.

As a result, it has been decided to follow the architecture proposed in [29], since it satisfies all these points. This leads to the following architectural choices:

- The entire design can be seen as a black box that receives and returns data in IEEE-754 single and half precision formats.
- The internal pipeline adopts a specific fixed-point format to cope with precision loss due the use of FXP arithmetic. This means that the original mantissa is padded with additional overflow (left) and guard (right) bits. Therefore, no floating-point blocks are being implied.
- The shift amount that is required during each CORDIC or scaling iteration is known in advance, hence Barrel shifters can be replaced by hardwired shifts.
- The LUT stores the $\alpha_{m,i}$ angles directly in the internal fixed-point format.
- When converting floating-point inputs to fixed-point numbers, they need to be aligned to be correctly added or subtracted. For x and y this is dynamically done depending on their respective exponents, whereas z is aligned to a fixed exponent that is the maximum one among all the $\alpha_{m,i}$ angles.

As it can be easily noticed from Figure 3.5, which shows the block diagram of the entire floating-point CORDIC architecture, the unit can be divided in three macro-blocks:

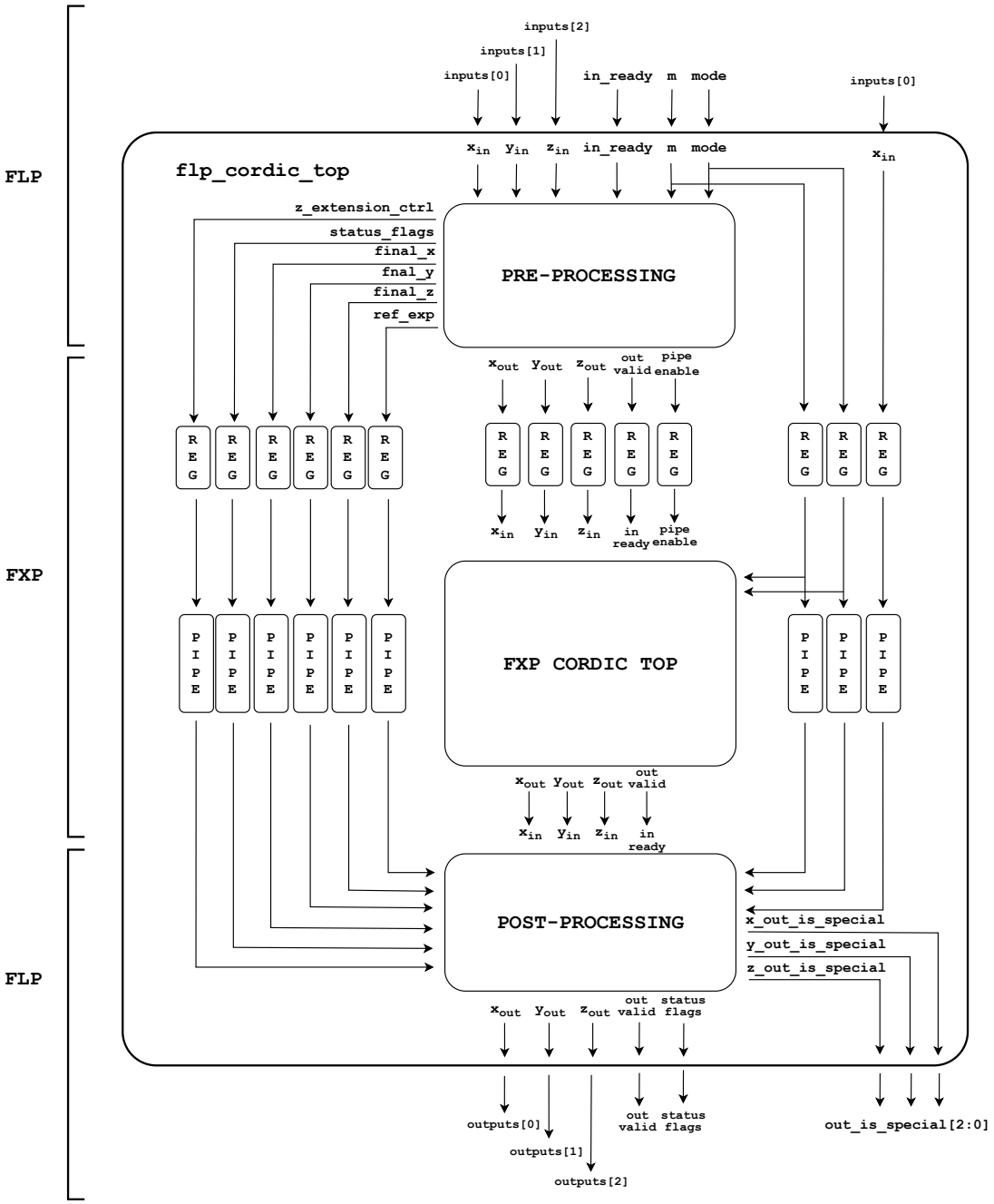


Figure 3.5: Block diagram of the entire floating-point CORDIC unit.

- **Pre-processing block**: it is responsible for converting the input floating-point numbers to the internal fixed-point format, as well as for aligning them

to the correct exponent and setting the status flags depending on the input values.

- **Fixed-point CORDIC top module**: it is the core of the architecture, since it performs the actual CORDIC iterations, as well as the scaling factor compensation procedure. This block works on fixed-point numbers.
- **Post-processing block**: it is responsible for converting the output fixed-point numbers to the IEEE-754 single or half precision format, as well as for setting the status flags depending on the input ones and on specific conditions on the output values.

In the following sections each of these blocks will be described in details.
Whenever the unit has to perform a new operation, it complies with these steps:

1. **Pre-processing**:
   (a) Set the status flags according to specific input combinations.
   (b) Unpack the floating-point inputs and isolate sign, exponent and mantissa.
   (c) Find the reference exponent for the fixed-point pipeline.
   (d) Build the internal fixed-point format by padding the mantissa with the hidden bit and the additional guard and overflow bits.
   (e) Align each operand depending on its initial exponent.
   (f) Compute the 2's complement of each number if its sign is negative.
   (g) If the unit is working in rotation mode with circular coordinates, map the original z angle from $[0, 2\pi]$ to $\left[0, \frac{\pi}{2}\right]$ to satisfy the ROC for such mode (Table 2.5).
2. **Fixed-point CORDIC algorithm execution**: perform the CORDIC iterations by executing equations 2.23-2.25 within each stage.
3. **Scaling factor compensation**: perform the scaling factor compensation procedure according to [29].
4. **Post-processing**:
   (a) Convert the operands into sign-magnitude form from 2's complement.
   (b) Detect if any of the numbers is zero.
   (c) Find the position of leading one and use this amount to shift the values in order to have a unitary integer part. Update the respective exponents accordingly.

(d) Shift right each operand to remove the additional guard bits.

(e) Round the values according to the selected rounding mode.

(f) Eventually normalize the operands if there is a non-unitary integer part and update the respective exponents accordingly.

(g) Pack the operands into the IEEE-754 single or half precision format.

(h) Select the final outputs based on the status flags coming from the input section and on specific conditions on the output values.

In addition, throughout the execution a handshake system through a pair of signals has been adopted for each of the macro-blocks. This is based on an `in_ready` input signal and on an `out_valid` output one. The former is raised whenever each module receives meaningful data, the latter when valid results are being produced. For unpipelined blocks, this means that `in_ready` is simply forwarded to `out_valid`, since computation takes zero time. For pipelined modules, instead, `in_ready` is forwarded to `out_valid` through a delay pipeline whose depth is equal to the module latency. In this way, the `out_valid` signal is raised only when the module has completed its computation. As an obvious consequence, the `in_ready`/`out_valid` pair of each macro-block is being cascaded, which means that the `out_valid` coming from the pre-processing block is connected to the `in_ready` of the fixed-point CORDIC top module, and so on.
As there is no need to have two separated elements to enable the computation and to signal the availability of meaningful data, the `in_ready` signal also works as an enable one. This means that, whenever it is low, all the internal registers are not updating their content.
Finally, since, as explained later, only the fixed-point CORDIC top module and the scaling factor compensation block are pipelined, the overall latency of the unit is `N_ITERATIONS` + `SCALING_STAGES`, so:

- 37 clock cycles for single precision
- 21 clock cycles for half precision

which are large numbers for sure, but have to be compared to a throughput of 2 output values per clock cycle.

3.3 Pre-processing block

The pre-processing block is responsible for converting the floating-point inputs into the internal fixed-point format, as well as for aligning them based on the respective exponents and for setting the status flags depending on the input values.

In addition, in case of rotation mode with circular coordinates, it also maps the original z angle from $[0, 2\pi]$ to $\left[0, \frac{\pi}{2}\right]$ to satisfy the ROC for such mode (Table 2.5). The block diagram of this module is shown in Figure 3.6.

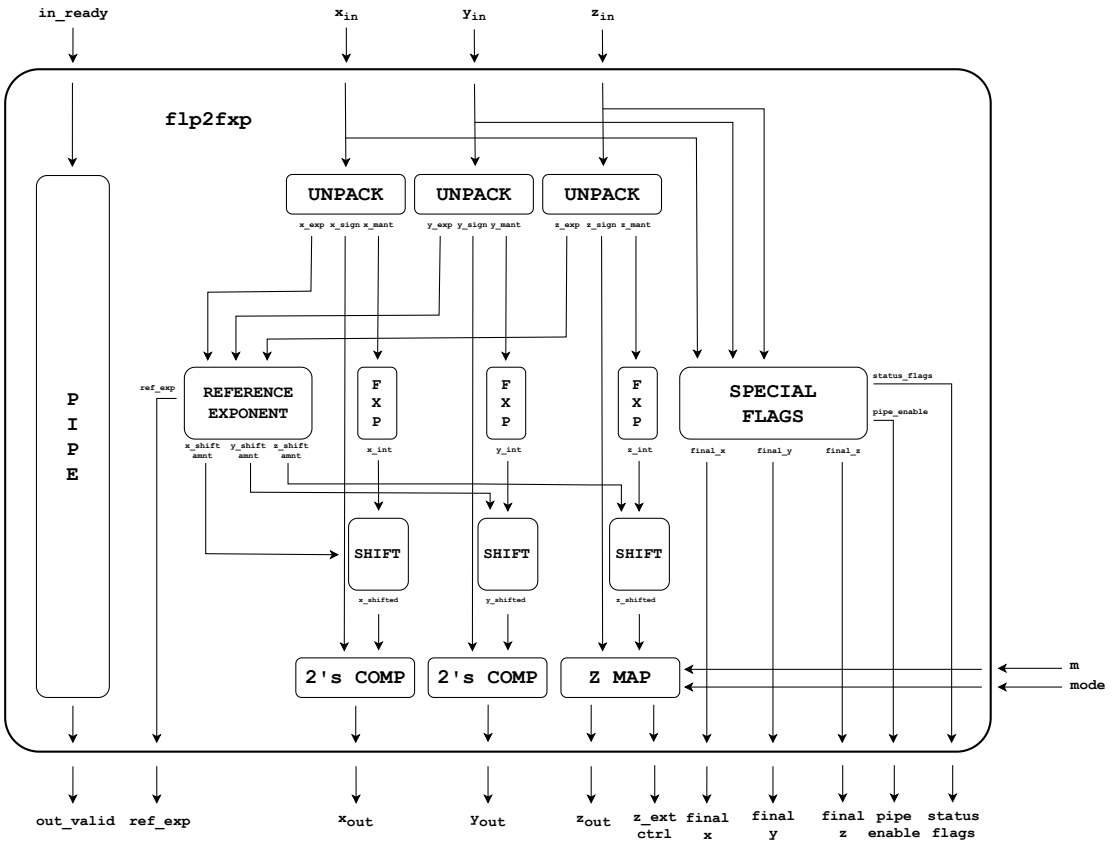


Figure 3.6: Block diagram of the pre-processing block.

Four main operations are performed by this block:

1. **FLP unpacking and FXP packing**: unpack the floating-point inputs and isolate sign, exponent and mantissa. Then, build the internal fixed-point format by padding the mantissa with the hidden bit and the additional guard and overflow bits.

2. **Reference exponent computation, alignment and 2's complement conversion**: find the reference exponent for the fixed-point pipeline. Then, align each operand depending on its initial exponent and compute the 2's complement of each operand if its sign is negative.

3. ***z* mapping**: if the unit is working in rotation mode with circular coordinates, map the original z angle from $[0, 2\pi]$ to $\left[0, \frac{\pi}{2}\right]$ to satisfy the ROC for such mode (Table 2.5).

4. **Status flags generation**: set the status flags according to specific input combinations.

The following subsections describe in details each of these steps.

3.3.1 FLP unpacking and FXP packing

The first operation to be performed whenever the CORDIC unit receives floating-point inputs is to unpack them and isolate sign, exponent and mantissa. This is done through the FLP unpack block shown in Figure 3.6, whose tasks are:

1. Forward the corresponding fields of the FLP format to specific internal signals Considering the generic FLP input `in_FLP` (it could be any of x, y or z) with:

   - `WIDTH` = width of `in_FLP`
   - `N_EXP` = number of exponent bits
   - `N_MANTISSA` = number of mantissa bits

   this means that the following bits will be isolated:

   - **Sign** = s = `in_FLP[WIDTH-1]`
   - **Biased exponent** = E = `in_FLP[N_MANTISSA+N_EXP-1:N_MANTISSA]`
   - **Mantissa** = m = `in_FLP[N_MANTISSA-1:0]`

2. Subtract the bias b from the exponent to obtain the unbiased one $e = E - b = E - 2^{(\texttt{N\_EXP}-1)}$.

3. Check if the input number is subnormal ($e < e_{min} = 1 - b$) and set the hidden bit accordingly:

   - If $e < e_{min}$, the input number is subnormal and the hidden bit is set to 0.
   - If $e \geq e_{min}$, the input number is normal and the hidden bit is set to 1.

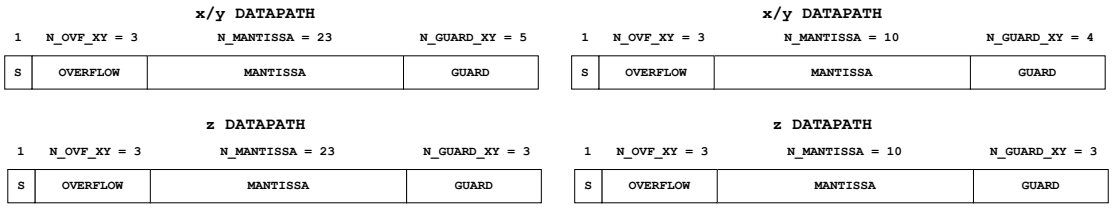
It has to be underlined that both the biased and unbiased exponents are described using `N_EXP`+2 bits, in order to take into account both negative and large (overflow) values and to perform meaningful comparisons.
When all the inputs have been unpacked, the FXP pack block is responsible for building the internal fixed-point format by padding the mantissa with the hidden bit and the additional guard and overflow bits. Since fixed-point arithmetic suffers

from a more limited range of representable values with respect to floating-point, additional bits are required both to avoid overflow and to cope with precision loss during CORDIC iterations. Starting from [29], the number of these bits has been considered as follows for single precision:

- `N_OVF_XY` = number of overflow bits in the x and y datapaths = 3
- `N_GUARD_XY` = number of guard bits in the x and y datapaths = 5
- `N_OVF_Z` = number of overflow bits in the z datapath = 3
- `N_GUARD_Z` = number of guard bits in the z datapath = 3

The only parameter that differs from [29] is `N_OVF_Z`, since one additional overflow bit has been considered to have a wider range of acceptable values for z. For what concerns half precision, instead, the amount of padding bits has been found starting from [12] and interpolating the values that have been previously chosen for single precision:

- `N_OVF_XY` = 3
- `N_GUARD_XY` = 4
- `N_OVF_Z` = 3
- `N_GUARD_Z` = 3



(a) Single precision (b) Half precision

Figure 3.7: Fixed-point format for single (a) and half (b) precision.

Assuming to allocate anyway three overflow bits also for half precision[1], [12] highlights the need for a total number of $log_2(w)$ (w is the fixed-point width) additional ones to make precision loss negligible. Since for half precision

[1]As explained later, this especially helps to improve the range of acceptable inputs for z.

$$w = \texttt{N\_MANTISSA} + \texttt{N\_OVF} + 1 = 10 + 3 + 1 = 14$$

this means that $log_2(14) \approx 3.8 \approx 4$ guard bits are required. This number has, then, been reduced by one for z to follow what has been done in [29] for single precision. The final fixed-point format number will be (following SystemVerilog syntax):

```
in_FXP = {0, {(N_OVF-1){1'b0}}, hb, m, {N_GUARD{1'b0}}}
```

with `hb` = hidden bit and `m` = mantissa. The sign bit is forced to zero because the conversion from sign-magnitude form to 2's complement is performed later for negative numbers. The final width of `in_FXP` is, then,

$$w = \texttt{N\_MANTISSA} + \texttt{N\_OVF} + \texttt{N\_GUARD} + 1 \tag{3.1}$$

which leads to:

- $w_{single,\ x/y} = 23 + 3 + 5 + 1 = 32$
- $w_{single,\ z} = 23 + 3 + 3 + 1 = 30$
- $w_{half,\ x/y} = 10 + 3 + 4 + 1 = 18$
- $w_{half,\ z} = 10 + 3 + 3 + 1 = 17$

It has to be underlined that, since $\texttt{N\_OVF\_Z} < 4$, `N_OVF_Z` is temporarily forced to 4 and restored after the mapping operation, in order to allow the fixed-point representation of z to span over the entire extended ROC for rotation mode with circular coordinates. This will be more clear in the following section.

3.3.2 Reference exponent computation, alignment and 2's complement conversion

As explained in Section 2.2.1, a floating-point addition can be considered as a fixed-point one that uses opportunely shifted operands, so that each number is aligned to the larger exponent between the two. Since the proposed CORDIC unit lays on global FLP, the operations that are being performed internally to the fixed-point pipeline have to be compliant with floating-point arithmetic. As a consequence, before entering the pipeline itself, the FXP representations of the input operands have to be aligned to the same larger exponent, which is, then, called reference exponent. This reference exponent is, therefore, the larger between the exponents of the two operands to be added or subtracted together. This leads to:

- **Reference exponent for x and y**:

$$e_{REF,\ xy} = max(e_x, e_y) \tag{3.2}$$

  where e_x and e_y are the exponents of x and y, respectively. This is dynamically done depending on each new set of inputs x and y.

- **Reference exponent for z and the $\alpha_{m,i}$ angles**:

$$e_{REF,\ z} = max(e_z, e_\alpha) \tag{3.3}$$

  where e_z is the exponent of z and e_α is the exponent of the $\alpha_{m,i}$ angle. Since, throughout the execution of the algorithm, each stage i refers to a specific angle $\alpha_{m,i}$, aligning z and $\alpha_{m,i}$ at each iteration would lead to the loss of all the advantages of global FLP. Hence, it has been decided to choose the maximum exponent among all the $\alpha_{m,i}$ angles of all working modes, namely -1, as the reference one for both z and each $\alpha_{m,i}$.

Once e_{REF} has been computed, the amount by which each operand has to be shifted is:

$$a_x = e_x - e_{REF} \tag{3.4}$$
$$a_y = e_y - e_{REF} \tag{3.5}$$
$$a_z = e_z - (-1) \tag{3.6}$$

Depending on the sign of a and on the considered pipeline, each FXP representation of the inputs will be shifted left or right by $|a|$ bits as follows:

$$x \text{ and } y \ \rightarrow \ \texttt{in\_shifted} = \begin{cases} \texttt{in\_FXP} & \text{if } a = 0 \\ \texttt{in\_FXP} >> |a| & \text{if } a < 0 \end{cases} \tag{3.7}$$

$$z \ \rightarrow \ \texttt{in\_shifted} = \begin{cases} \texttt{in\_FXP} << a & \text{if } a \geq 0 \\ \texttt{in\_FXP} >> |a| & \text{if } a < 0 \end{cases} \tag{3.8}$$

As previously mentioned, z will always be aligned to exponent -1, which means that, with the availability of `N_OVF_Z` bits for its integer part, only input numbers up to $2^{(\texttt{N\_OVF\_Z}-2)}$ can be correctly converted. Assuming three overflow bits, the maximum acceptable input value for z is $2^{(3-2)} = 2$, which is enough to cover the entire range of convergence of most working modes apart from rotation mode with circular coordinates, whose larger limit is $2\pi \approx 6.28 > 2$. To cope with this, it has been decided to temporarily force `N_OVF_Z` to 4 and restore it after the mapping operation, which brings back z to be at most $\frac{\pi}{2}$.

The last step to be performed before letting the inputs exit the pre-processing block is the conversion from sign-magnitude form to 2's complement for negative numbers, in order to permit the execution of additions and subtraction in the standard way.

3.3.3 *z* mapping

The z-map unit is responsible for:

- Mapping the original z angle from $[0, 2\pi]$ to $\left[0, \frac{\pi}{2}\right]$, in order to combine the ROC limits with a wider range of acceptable angles in rotation mode with circular coordinates.
- Compute the 2's complement of z, if its sign is negative, for the other working modes.

Figure 3.8 shows the block diagram of this module.

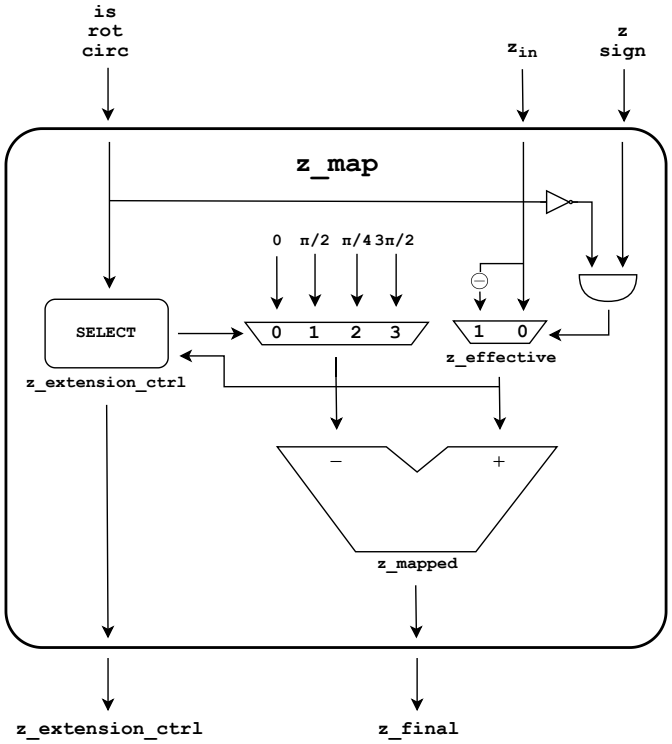


Figure 3.8: Block diagram of the z-map unit.

Whenever the pre-processing block detects that the unit has to perform an operation in rotation mode with circular coordinates, it checks which is the angular range that includes z and subtracts a specific value according to equation 3.9. The values $\frac{\pi}{2}$, π, $\frac{3\pi}{2}$ and 2π are being directly considered in the fixed-point representation shown in Figure 3.7, with the number of overflow bits set to four.

$$z_{final} = \begin{cases} z & \text{if } 0 \leq z < \frac{\pi}{2} \\ z - \frac{\pi}{2} & \text{if } \frac{\pi}{2} \leq z < \pi \\ z - \pi & \text{if } \pi \leq z < \frac{3\pi}{2} \\ z - \frac{3\pi}{2} & \text{if } \frac{3\pi}{2} \leq z < 2\pi \end{cases} \tag{3.9}$$

If, instead, the unit is working in any of the other modes, the 2's complement of z is computed if its sign is negative. In either of the two cases, the final value is brought back to the original fixed-point format with `N_OVF_Z` overflow bits.
In addition to this, the unit also sets a specific control signal, `z_extension_ctrl`, which travels through the pipeline and suggests the post-processing block which output it has to select. Its encodings are:

- `NO_MAP`: the unit is not working in rotation mode with circular coordinates or the z angle is already within the ROC.
- `FIRST_RANGE`: the unit is working in rotation mode with circular coordinates and the z angle is in the range $\left[\frac{\pi}{2}, \pi\right]$.
- `SECOND_RANGE`: the unit is working in rotation mode with circular coordinates and the z angle is in the range $\left[\pi, \frac{3\pi}{2}\right]$.
- `THIRD_RANGE`: the unit is working in rotation mode with circular coordinates and the z angle is in the range $\left[\frac{3\pi}{2}, 2\pi\right]$.

3.3.4 Status flags generation

The last operation to be performed by the pre-processing module is the generation, for each operand, of the status flags described in Section 2.2.3, which are:

- **Invalid operation (NV)**
- **Division by zero (DZ)**
- **Overflow (OF)**
- **Underflow (UF)**
- **Inexact (NX)**

The purpose of this unit is not only to let the post-processing block know which output it has to select to provide a meaningful result, but also to disable the pipeline whenever the outcome of an operation can be predicted, so that a lower power consumption can be achieved. Each of the flags is set to 1 or to 0 according both to the combinations explained in Section 2.2.3 and to algorithm-specific conditions. In particular,

1. **Invalid operation (NV)**: in addition to the cases in Section 2.2.3, this flag is set to 1 also if the range of convergence is not being respected. This can be easily predicted for all modes but the vectoring one with hyperbolic coordinates, where the ROC depends on the computation of the hyperbolic arctangent. In this case, the flag is raised if the absolute value of the ratio between y and x is greater than or equal to 1, which means that $atanh\left(\frac{y}{x}\right)$ is not defined, thus not valid. In particular, the conditions that cause NV to be set are:

   (a) **Rotation mode**:

      i. **Circular coordinates**:

      - $\boldsymbol{x_n = x_{in} \cdot cos(z_{in}) - y_{in} \cdot sin(z_{in})}$
        - Any of the inputs is a NaN.
        - $|x_{in}| = \infty$ and $cos(z_{in}) = 0$
        - $|y_{in}| = \infty$ and $sin(z_{in}) = 0$
        - $|x_{in}| = |y_{in}| = \infty$ and $sign(x_{in} \cdot cos(z_{in})) = sign(y_{in} \cdot sin(z_{in}))$
        - z_{in} outside the ROC
      - $\boldsymbol{y_n = x_{in} \cdot sin(z_{in}) + y_{in} \cdot cos(z_{in})}$
        - Any of the inputs is a NaN.
        - $|x_{in}| = \infty$ and $sin(z_{in}) = 0$
        - $|y_{in}| = \infty$ and $cos(z_{in}) = 0$
        - $|x_{in}| = |y_{in}| = \infty$ and $sign(x_{in} \cdot sin(z_{in})) \neq sign(y_{in} \cdot cos(z_{in}))$
        - z_{in} outside the ROC
      - $\boldsymbol{z_n = 0}$
        - z_{in} is a NaN
        - z_{in} outside the ROC

      ii. **Hyperbolic coordinates**:

      - $\boldsymbol{x_n = x_{in} \cdot cosh(z_{in}) + y_{in} \cdot sinh(z_{in})}$
        - Any of the inputs is a NaN.
        - $|y_{in}| = \infty$ and $sinh(z_{in}) = 0$
        - $|x_{in}| = |y_{in}| = \infty$ and $sign(x_{in} \cdot cosh(z_{in})) \neq sign(y_{in} \cdot sinh(z_{in}))$

  - z_{in} outside the ROC
- $\boldsymbol{y_n = x_{in} \cdot sinh(z_{in}) + y_{in} \cdot cosh(z_{in})}$
  - Any of the inputs is a NaN.
  - $|x_{in}| = \infty$ and $sinh(z_{in}) = 0$
  - $|x_{in}| = |y_{in}| = \infty$ and $sign(x_{in} \cdot sinh(z_{in})) \neq sign(y_{in} \cdot cosh(z_{in}))$
  - z_{in} outside the ROC
- $\boldsymbol{z_n = 0}$
  - z_{in} is a NaN
  - z_{in} outside the ROC

iii. **Linear coordinates**:

- $\boldsymbol{x_n = x_{in}}$
  - Any of the inputs is a NaN.
  - z_{in} outside the ROC
- $\boldsymbol{y_n = y_{in} + x_{in} \cdot z_{in}}$
  - Any of the inputs is a NaN.
  - $|x_{in}| = \infty$ and $z_{in} = 0$
  - $|x_{in} \cdot z_{in}| = \infty$, $|y_{in}| = \infty$ and $sign(x_{in} \cdot z_{in}) \neq sign(y_{in})$
  - $|x_{in}| = |y_{in}| = \infty$ and $sign(x_{in}) \neq sign(y_{in})$
  - z_{in} outside the ROC
- $\boldsymbol{z_n = 0}$
  - z_{in} is a NaN
  - z_{in} outside the ROC

(b) **Vectoring mode**:

i. **Circular coordinates**:

- $\boldsymbol{x_n = sign(x_{in}) \cdot \sqrt{x_{in}^2 + y_{in}^2}}$
  - Either x_{in} or y_{in} is a NaN.
- $\boldsymbol{y_n = 0}$
  - Either x_{in} or y_{in} is a NaN.
- $\boldsymbol{z_n = z_{in} + tan^{-1}\left(\frac{y_{in}}{x_{in}}\right)}$
  - Any of the inputs is a NaN.
  - $|x_{in}| = \infty$ and $|y_{in}| = \infty$
  - $x_{in} = 0$ and $y_{in} = 0$

ii. **Hyperbolic coordinates**:

- $\boldsymbol{x_n = \sqrt{x_{in}^2 - y_{in}^2}}$
  - Either x_{in} or y_{in} is a NaN.

  - $|x_{in}| < |y_{in}|$
  - $|x_{in}| = \infty$ and $|y_{in}| = \infty$
  - $tanh^{-1}\left(\frac{y_{in}}{x_{in}}\right)$ is not defined ($|x_{in}| \leq |y_{in}|$)
- $\boldsymbol{y_n = 0}$
  - Either x_{in} or y_{in} is a NaN.
  - $tanh^{-1}\left(\frac{y_{in}}{x_{in}}\right)$ is not defined ($|x_{in}| \leq |y_{in}|$)
- $\boldsymbol{z_n = z_{in} + tanh^{-1}\left(\frac{y_{in}}{x_{in}}\right)}$
  - Any of the inputs is a NaN.
  - $|x_{in}| = \infty$ and $|y_{in}| = \infty$
  - $x_{in} = 0$ and $y_{in} = 0$
  - $|z_{in}| = tanh^{-1}\left(\frac{y_{in}}{x_{in}}\right) = \infty$ and $sign(z_{in}) \neq sign\left(tanh^{-1}\left(\frac{y_{in}}{x_{in}}\right)\right)$
  - $tanh^{-1}\left(\frac{y_{in}}{x_{in}}\right)$ is not defined ($|x_{in}| \leq |y_{in}|$)

iii. **Linear coordinates**:

- $\boldsymbol{x_n = x_{in}}$
  - x_{in} is a NaN.
  - $\frac{y_{in}}{x_{in}}$ is outside the ROC
- $\boldsymbol{y_n = 0}$
  - Either x_{in} or y_{in} is a NaN.
  - $\frac{y_{in}}{x_{in}}$ is outside the ROC
- $\boldsymbol{z_n = z_{in} + \frac{y_{in}}{x_{in}}}$
  - Any of the inputs is a NaN.
  - $x_{in} = 0$ and $y_{in} = 0$
  - $|x_{in}| = \infty$ and $|y_{in}| = \infty$
  - $|z_{in}| = \infty$, $\frac{y_{in}}{x_{in}} = \infty$ and $sign(z_{in}) \neq sign\left(\frac{y_{in}}{x_{in}}\right)$
  - $\frac{y_{in}}{x_{in}}$ is outside the ROC

2. **Division by zero (DZ)**: this flag is being set to 1 only when x is zero in vectoring mode with linear coordinates, since it is the only case in which a division by zero that leads to an infinite result can occur.

3. **Overflow (OF), Underflow (UF) and Inexact (NX)**: these flags cannot be usefully predicted at the input side, since operands combinations that make a result overflow or underflow could be valid only for one of the three values exiting the unit. Therefore, since the pipeline cannot be deactivated to save power, there is no advantage in trying to set them here. Further checks are left to the output side.

The status flags encodings are:

- `NO_EXCEPTIONS`: all the flags are 0.
- `INVALID`: NV is set.
- `DIVISION_BY_ZERO`: DZ is set.
- `OVERFLOW`: OF and NX are set.
- `UNDERFLOW`: UF and NX are set.

The disabling mechanism to decrease power consumption is based on a control signal, `pipe_enable`, which enables the pipeline registers. Whenever the final result can be predicted (NaN, Inf or 0), `pipe_enable` is set to 0, thus preventing the data registers content from being updated and avoiding further and useless switching activity. The other control signals, however, are still propagated through the pipeline, since they are required by the post-processing block. In addition to `pipe_enable`, another signal is used, for each operand, to communicate to the post-processing block which value it has to select for the corresponding output. These signals are called `final_x`, `final_y` and `final_z` and travel through the pipeline alongside the operands. Whenever $NV_x = NV_y = NV_z = 1$ holds, the final outputs are forced to be NaNs, otherwise their value is set depending on specific combinations of the NV status flag:

1. **Rotation mode**:
   (a) **Circular coordinates**:
   - $\boldsymbol{NV_x = NV_y = 0}$
     - `final_x`:
       * If $|x_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of $x_{in} \cdot cos(z_{in})$.
       * If $|y_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of $y_{in} \cdot sin(z_{in})$.
       * If $x_{in} = y_{in} = 0$, `final_x = ZERO_FLAG`.
       * `final_x = NORMAL_FLAG` otherwise.
     - `final_y`:
       * If $|x_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of $x_{in} \cdot sin(z_{in})$.
       * If $|y_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of $y_{in} \cdot cos(z_{in})$.
       * If $x_{in} = y_{in} = 0$, `final_y = ZERO_FLAG`.
       * `final_y = NORMAL_FLAG` otherwise.

  - **`final_z`** is always equal to `ZERO_FLAG`, given that, in rotation mode, $z_{in} = 0$.
  - **`pipe_enable`** is 1 only when both `final_x` and `final_y` are equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 0}$ and $\boldsymbol{NV_y = 1}$:
  - **`final_x`**:
    * If $|x_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of $x_{in} \cdot cos(z_{in})$.
    * If $|y_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of $y_{in} \cdot sin(z_{in})$.
    * `final_x = NORMAL_FLAG` otherwise.
  - **`final_y`** is always equal to `POS_NAN_FLAG`, given that $NV_y = 1$, which means that the y result is not valid.
  - **`final_z`** is always equal to `ZERO_FLAG`, given that, in rotation mode, $z_{in} = 0$.
  - **`pipe_enable`** is 1 when `final_x` is equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 1}$ and $\boldsymbol{NV_y = 0}$:
  - **`final_x`** is always equal to `POS_NAN_FLAG`, given that $NV_x = 1$, which means that the x result is not valid.
  - **`final_y`**:
    * If $|x_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of $x_{in} \cdot sin(z_{in})$.
    * If $|y_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of $y_{in} \cdot cos(z_{in})$.
    * `final_y = NORMAL_FLAG` otherwise.
  - **`final_z`** is always equal to `ZERO_FLAG`, given that, in rotation mode, $z_{in} = 0$.
  - **`pipe_enable`** is 1 when `final_y` is equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.

(b) **Hyperbolic coordinates**:

- $\boldsymbol{NV_x = NV_y = 0}$
  - **`final_x`**:
    * If $|x_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of x_{in}.
    * If $|y_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of $y_{in} \cdot sinh(z_{in})$.

    - If $x_{in} = y_{in} = 0$, `final_x = ZERO_FLAG`.
    - `final_x = NORMAL_FLAG` otherwise.
  - **`final_y`**:
    - If $|x_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of $x_{in} \cdot sinh(z_{in})$.
    - If $|y_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of y_{in}.
    - If $x_{in} = y_{in} = 0$, `final_y = ZERO_FLAG`.
    - `final_y = NORMAL_FLAG` otherwise.
  - **`final_z`** is always equal to `ZERO_FLAG`, given that, in rotation mode, $z_{in} = 0$.
  - **`pipe_enable`** is 1 when both `final_x` and `final_y` are equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 0}$ and $\boldsymbol{NV_y = 1}$:
  - **`final_x`**:
    - If $|x_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of x_{in}.
    - If $|y_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of $y_{in} \cdot sinh(z_{in})$.
    - `final_x = NORMAL_FLAG` otherwise.
  - **`final_y`** is always equal to `POS_NAN_FLAG`, given that $NV_y = 1$, which means that the y result is not valid.
  - **`final_z`** is always equal to `ZERO_FLAG`, given that, in rotation mode, $z_{in} = 0$.
  - **`pipe_enable`** is 1 when `final_x` is equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 1}$ and $\boldsymbol{NV_y = 0}$:
  - **`final_x`** is always equal to `POS_NAN_FLAG`, given that $NV_x = 1$, which means that the x result is not valid.
  - **`final_y`**:
    - If $|x_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of $x_{in} \cdot sinh(z_{in})$.
    - If $|y_{in}| = \infty$, `final_y = PLUS_INF_FLAG` or `final_y = MINUS_INF_FLAG` depending on the sign of y_{in}.
    - `final_y = NORMAL_FLAG` otherwise.
  - **`final_z`** is always equal to `ZERO_FLAG`, given that, in rotation mode, $z_{in} = 0$.

- **pipe_enable** is 1 when final_y is equal to NORMAL_FLAG, otherwise it is 0, since the results can be predicted.

(c) **Linear coordinates**:

- $\boldsymbol{NV_x = NV_y = 0}$
  - **final_x** is always equal to NORMAL_FLAG, given that, in linear coordinates, the post-processing block directly sets x_{out} to x_{in}.
  - **final_y**:
    - If $|x_{in} \cdot z_{in}| = \infty$, final_y = PLUS_INF_FLAG or final_y = MINUS_INF_FLAG depending on the sign of $x_{in} \cdot z_{in}$.
    - If $|y_{in}| = \infty$, final_y = PLUS_INF_FLAG or final_y = MINUS_INF_FLAG depending on the sign of y_{in}.
    - final_y = NORMAL_FLAG otherwise.
  - **final_z** is always equal to ZERO_FLAG, given that, in rotation mode, $z_{in} = 0$.
  - **pipe_enable** is 1 when final_y is equal to NORMAL_FLAG, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 0}$ and $\boldsymbol{NV_y = 1}$:
  - **final_x** is always equal to NORMAL_FLAG, given that, in linear coordinates, the post-processing block directly sets x_{out} to x_{in}.
  - **final_y** is always equal to POS_NAN_FLAG, given that $NV_y = 1$, which means that the y result is not valid.
  - **final_z** is always equal to ZERO_FLAG, given that, in rotation mode, $z_{in} = 0$.
  - **pipe_enable** is always equal to 0, given that the results can be predicted.
- $\boldsymbol{NV_x = 1}$ and $\boldsymbol{NV_y = 0}$: this combination cannot take place.

2. **Vectoring mode**:

(a) **Circular coordinates**:

- $\boldsymbol{NV_x = NV_z = 0}$
  - **final_x**:
    - If $|x_{in}| = \infty$ or $|y_{in}| = \infty$, final_x = PLUS_INF_FLAG or final_x = MINUS_INF_FLAG depending on the sign of x_{in}.
    - final_x = NORMAL_FLAG otherwise.
  - **final_y** is always equal to ZERO_FLAG, given that, in vectoring mode, $y_{in} = 0$.
  - **final_z**:

    - If $|x_{in}| = \infty$ or $|y_{in}| = \infty$ or $|z_{in}| = \infty$, `final_z = PLUS_INF_FLAG` or `final_z = MINUS_INF_FLAG` depending on the sign of z_{in}.
    - `final_z = NORMAL_FLAG` otherwise.
  - **`pipe_enable`** is 1 when both `final_x` and `final_z` are equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 0}$ and $\boldsymbol{NV_z = 1}$:
  - **`final_x`**:
    - If $|x_{in}| = \infty$ or $|y_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of x_{in}.
    - `final_x = NORMAL_FLAG` otherwise.
  - **`final_y`** is always equal to `ZERO_FLAG`, given that, in vectoring mode, $y_{in} = 0$.
  - **`final_z`** is always equal to `POS_NAN_FLAG`, given that $NV_z = 1$, which means that the z result is not valid.
  - **`pipe_enable`** is 1 when `final_x` is equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 1}$ and $\boldsymbol{NV_z = 0}$: this combination cannot take place.

(b) **Hyperbolic coordinates**:

- $\boldsymbol{NV_x = NV_z = 0}$
  - **`final_x`**:
    - If $|x_{in}| = \infty$ and $|z_{in}| = \infty$, `final_x = PLUS_INF_FLAG` or `final_x = MINUS_INF_FLAG` depending on the sign of x_{in}.
    - If $|x_{in}| = |y_{in}|$, `final_x = ZERO_FLAG`.
    - `final_x = NORMAL_FLAG` otherwise.
  - **`final_y`** is always equal to `ZERO_FLAG`, given that, in vectoring mode, $y_{in} = 0$.
  - **`final_z`**:
    - If $|x_{in}| = \infty$ and $|z_{in}| = \infty$, `final_z = PLUS_INF_FLAG` or `final_z = MINUS_INF_FLAG` depending on the sign of z_{in}.
    - If $|z_{in}| = \infty$, `final_z = PLUS_INF_FLAG` or `final_z = MINUS_INF_FLAG` depending on the sign of z_{in}.
    - If $|x_{in}| = |y_{in}|$, `final_z = PLUS_INF_FLAG` or `final_z = MINUS_INF_FLAG` depending on the sign of $\frac{y_{in}}{x_{in}}$.
    - `final_z = NORMAL_FLAG` otherwise.
  - **`pipe_enable`** is 1 when both `final_x` and `final_z` are equal to `NORMAL_FLAG`, otherwise it is 0, since the results can be predicted.

- $\boldsymbol{NV_x = 0}$ and $\boldsymbol{NV_z = 1}$:
  - **final_x**:
    - If $|x_{in}| = |y_{in}|$, final_x = ZERO_FLAG.
    - final_x = NORMAL_FLAG otherwise.
  - **final_y** is always equal to ZERO_FLAG, given that, in vectoring mode, $y_{in} = 0$.
  - **final_z** is always equal to POS_NAN_FLAG, given that $NV_z = 1$, which means that the z result is not valid.
  - **pipe_enable** is 1 when final_x is equal to NORMAL_FLAG, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 1}$ and $\boldsymbol{NV_z = 0}$: this combination cannot take place.

(c) **Linear coordinates**:

- $\boldsymbol{NV_x = NV_z = 0}$
  - **final_x** is always equal to NORMAL_FLAG, given that, in linear coordinates, the post-processing block directly sets x_{out} to x_{in}.
  - **final_y** is always equal to ZERO_FLAG, given that, in vectoring mode, $y_{in} = 0$.
  - **final_z**:
    - If $|z_{in}| = \infty$, final_z = PLUS_INF_FLAG or final_z = MINUS_INF_FLAG depending on the sign of z_{in}.
    - If $|x_{in}| = 0$ or $|y_{in}| = \infty$, final_z = PLUS_INF_FLAG or final_z = MINUS_INF_FLAG depending on the sign of $\frac{y_{in}}{x_{in}}$.
    - final_z = NORMAL_FLAG otherwise.
  - **pipe_enable** is 1 when final_z is equal to NORMAL_FLAG, otherwise it is 0, since the results can be predicted.
- $\boldsymbol{NV_x = 0}$ and $\boldsymbol{NV_z = 1}$:
  - **final_x** is always equal to NORMAL_FLAG, given that, in linear coordinates, the post-processing block directly sets x_{out} to x_{in}.
  - **final_y** is always equal to ZERO_FLAG, given that, in vectoring mode, $y_{in} = 0$.
  - **final_z** is always equal to POS_NAN_FLAG, given that $NV_z = 1$, which means that the z result is not valid.
  - **pipe_enable** is always equal to 0, given that the results can be predicted.
- $\boldsymbol{NV_x = 1}$ and $\boldsymbol{NV_z = 0}$: this combination cannot take place.

As an obvious consequence, whenever $NV_x = NV_y = 1$ in rotation mode or $NV_x = NV_z = 1$ in vectoring mode, the pipeline is disabled and the final results are directly set to NaNs.

The adopted notation for the `final_x`, `final_y` and `final_z` signals is the following:

- `PLUS_INF_FLAG`: the final result is $+\infty$.
- `MINUS_INF_FLAG`: the final result is $-\infty$.
- `POS_NAN_FLAG`: the final result is a positive NaN.
- `ZERO_FLAG`: the final result is zero.
- `NORMAL_FLAG`: the final result is equal to the value produced the CORDIC unit.

3.3.5 Other arrangements

Since the IEEE-754 standard includes the representation of infinities, specific arrangements have been developed to correctly handle them in fixed-point arithmetic. In particular, depending on the couple of operands to be added or subtracted together, namely x/y and $z/\alpha_{m,i}$, the following cases have been considered:

- Whenever only one operand between x_{in} and y_{in} is infinite, the other one is directly set to zero before entering the pipeline, since the result of the sum would always lead to the same infinite value. However, this solution causes some issues when y_{in} is an infinity and x_{in} is not in linear coordinates, given that the resulting x_n has to be equal to the input one (set to zero). To cope with this, the post-processing module can directly see the value of x_{in} and forward it to the output in such cases. If, instead, both operands are infinities, nothing is changed and their FXP representation is used to feed the CORDIC pipeline with new values.
- If z is infinite, the `z_is_inf` control signal is raised and pushed inside the pipeline, so that no $\alpha_{m,i}$ is being added to or subtracted from z.

3.4 Fixed-point CORDIC top module

The fixed-point CORDIC top module, which is shown in Figure 3.9, is in charge of providing the final and scaled results of CORDIC algorithm. It is pipeline-structured and composed of three main blocks:

- **Fixed-point CORDIC pipeline**, where the actual execution of CORDIC algorithm, namely equations 2.23-2.25, takes place. It includes `N_ITERATIONS` computational stages, each followed by a register to break the critical path.

- **Scaling pipeline**, where the scaling factor compensation is performed. It includes `SCALING_STAGES` execution stages, each followed by a register to break the critical path.
- **Lookup table**, which is used to provide all the stages with the values of the $\alpha_{m,i}$ angles and of the a_i coefficients used by the scaling pipeline [1]. The stored data follows the internal custom FXP format.

3.4.1 Fixed-point CORDIC pipeline

The fixed-point CORDIC pipeline shown in Figure 3.10 is composed by `N_ITERATIONS` computational stages. According to [12], in order to ensure a n-bit precision, the last CORDIC iteration has to perform a n-bit right-shift when executing equations 2.23-2.25, which means that, assuming the availability of `N_MANTISSA` mantissa bits, at least a shift amount equal to

- 23 for single precision
- 10 for half precision

has to be reached. Therefore, based on [29], these values have been increased by one, leading to 24 and 11 respectively.
As shown in Table 3.1, in order to allow the actuation of scaling factor compensation through additional iterations, [29] developed a specific shift sequence for the entire execution of CORDIC algorithm, which takes the actual number of iterations, and therefore of pipeline stages in an unfolded implementation, to

- `N_ITERATIONS` = 29 for single precision
- `N_ITERATIONS` = 15 for half precision

Since the architecture in [29] has been designed for single precision only, the value for half precision has been obtained by simply cutting the shift sequence to the required value 11. A remarkable result is that a unique shift sequence can be used for all coordinates systems, instead of dealing with different ones for each of them.

CORDIC stage

Apart from registers, the fixed-point CORDIC pipeline is composed by `N_ITERATIONS` computational stages which execute equations 2.23-2.25. The block diagram of a generic stage is shown in Figure 3.11.

[1] As anticipated, scaling factor compensation is performed by including additional iterations, instead of allocating a multiplier [29].

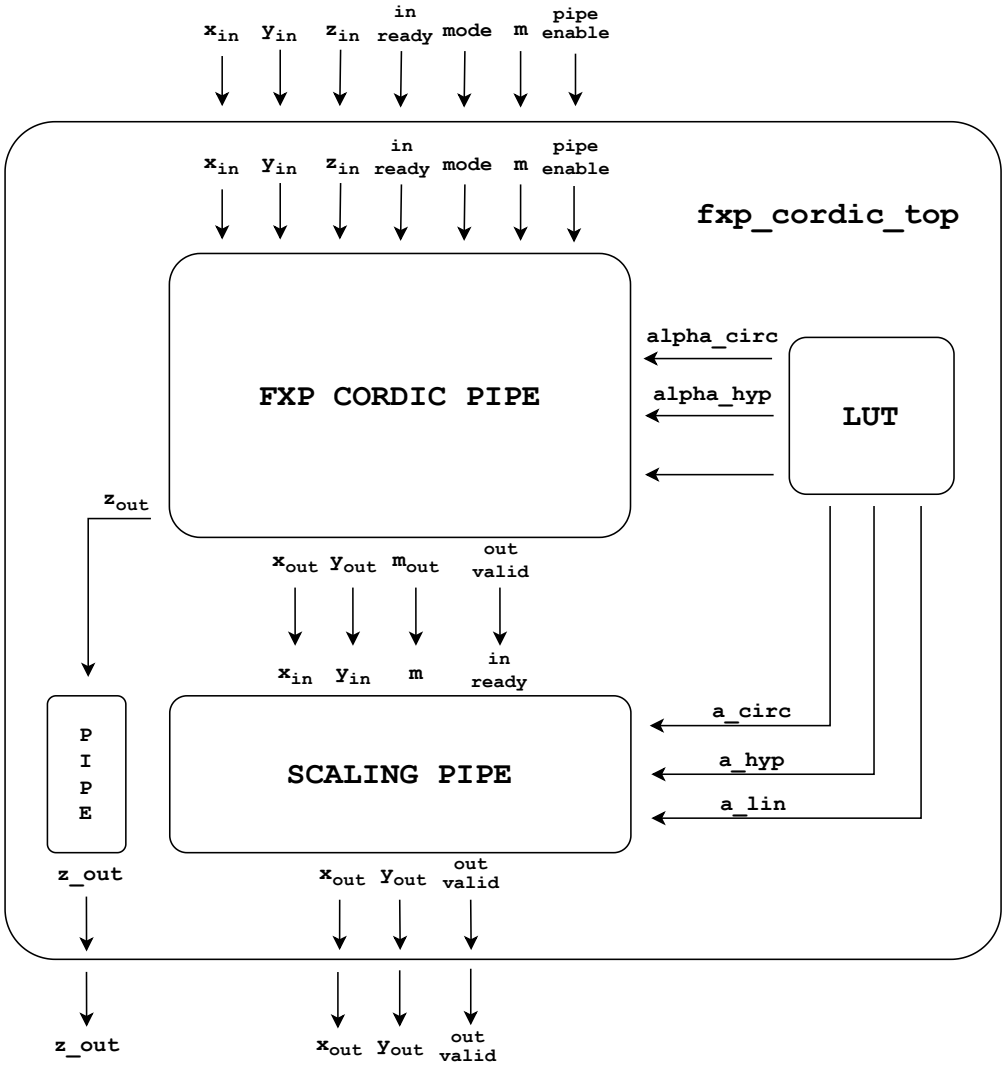


Figure 3.9: Block diagram of the fixed-point CORDIC top module.

The main components are:

- Three fixed-point adders, one for each variable x, y and z.
- Several multiplexers to choose the correct operands
- A control unit to drive the selection signals:

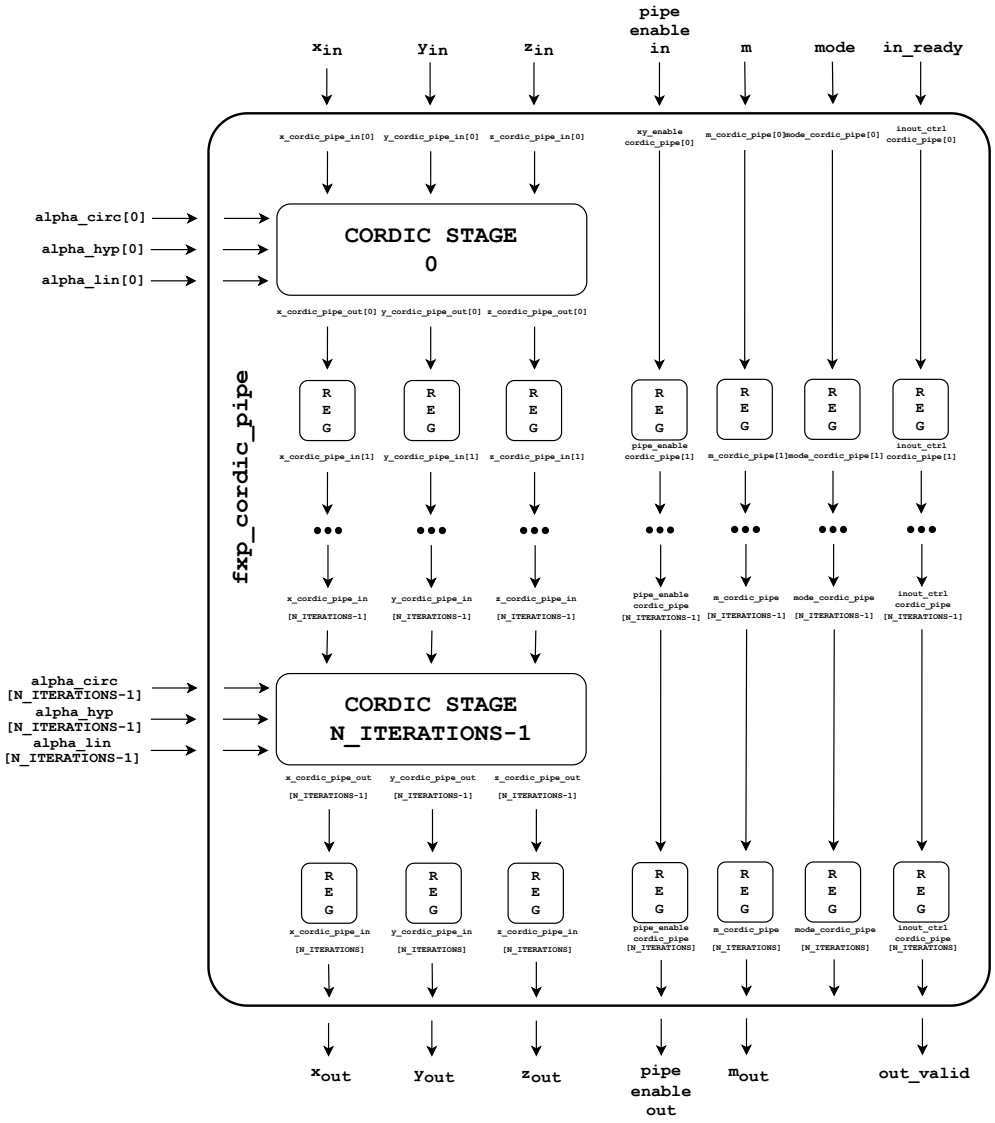


Figure 3.10: Block diagram of the fixed-point CORDIC pipeline.

– `add_sub_x`, `add_sub_y` and `add_sub_z` select the correct operation to be performed by the adders, depending on the sign of the input operands and on the working mode. A low value means addition, while a high one subtraction.

– In linear coordinates `is_linear` keeps x_{i+1} unchanged, so that $x_n = x_{in}$. In such case, `is_linear` is set to 1, otherwise it is set to 0.

Iteration (stage number)	Shift amount $S_{m,i}$
1	1
2	2
3	2
4	2
5	2
6	3
7	4
8	5
9	6
10	6

Iteration (stage number)	Shift amount $S_{m,i}$
11	7
12	8
13	9
14	10
15	11
16	12
17	13
18	13
19	14
20	15

Iteration (stage number)	Shift amount $S_{m,i}$
21	16
22	17
23	18
24	19
25	20
26	21
27	22
28	23
29	24

Table 3.1: Shift sequence for the fixed-point CORDIC pipeline [29].

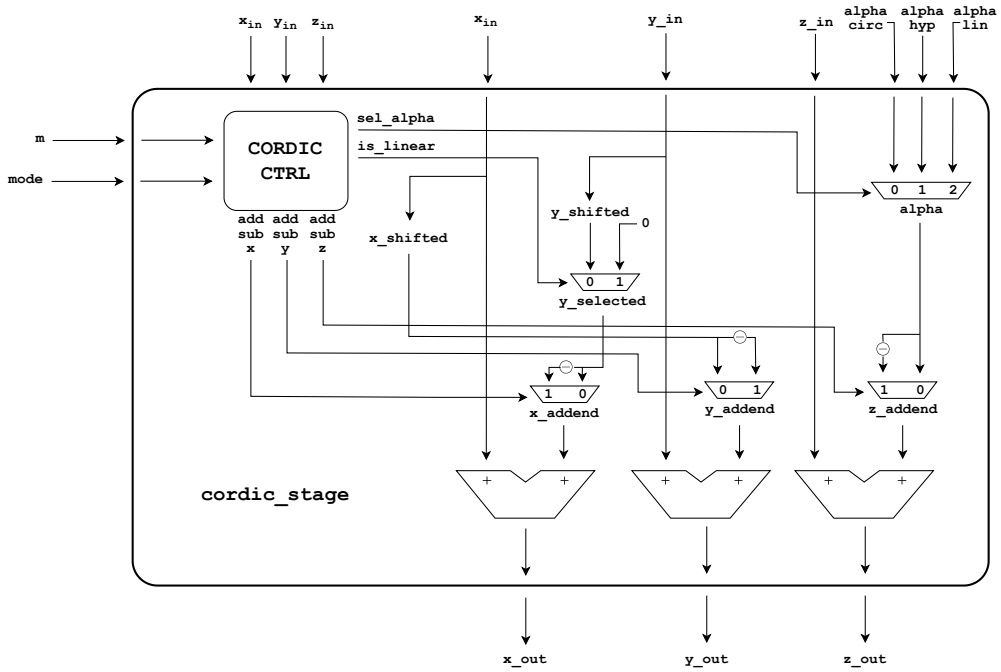


Figure 3.11: Block diagram of a CORDIC stage.

– `sel_alpha` chooses the correct $\alpha_{m,i}$ angle based on the coordinates system.

As anticipated, no Barrel shifters are required, since shift operations are hardwired. Even if not represented in Figure 3.11, each CORDIC stage also receives the `z_is_inf` control signal described in Section 3.3.5. If it is set to 1, the $\alpha_{m,i}$ angle is not added to or subtracted from z, which is directly forwarded to the output. This helps to properly handle infinities in fixed-point arithmetic.

3.4.2 Scaling pipeline

The scaling pipeline is shown in Figure 3.12.
As anticipated in the previous sections, scaling factor compensation is performed by including additional iterations at the end of the CORDIC pipeline. According to [29], this procedure pursues its task by executing, iteratively, the following equations:

$$x_{i+1} = x_i + x_i \cdot sign(a_{m,i}) \cdot 2^{|a_{m,i}|} \tag{3.10}$$
$$y_{i+1} = y_i + y_i \cdot sign(a_{m,i}) \cdot 2^{|a_{m,i}|} \tag{3.11}$$

where $a_{m,i}$ is the coefficient, taken from Table 3.2, for iteration i and coordinates system m. At the end, the final results will be scaled by a factor

$$\frac{1}{K_1} = 0.784039965 \text{ for circular coordinates} \tag{3.12}$$
$$\frac{1}{K_{-1}} = 1.327798882 \text{ for hyperbolic coordinates} \tag{3.13}$$

As expressed in equations 2.11 and 2.12, only the x and y have to be scaled, whereas the z one can be accepted as is.
In order to be compliant with the fixed-point CORDIC pipeline implementation, also the scaling one is unfolded and pipelined, which means that equations 3.10 and 3.11 refer to a generic stage i rather than to single iterations. This leads to the presence of a total number of `SCALING_STAGES` blocks, with

- `SCALING_STAGES` = 8 for single precision
- `SCALING_STAGES` = 6 for half precision

Also in this case, [29] only provides an architecture that is suitable for single precision. Therefore, the value of `SCALING_STAGES` for half precision has been obtained by simply cutting the sequence in Table 3.2 to the maximum shift amount

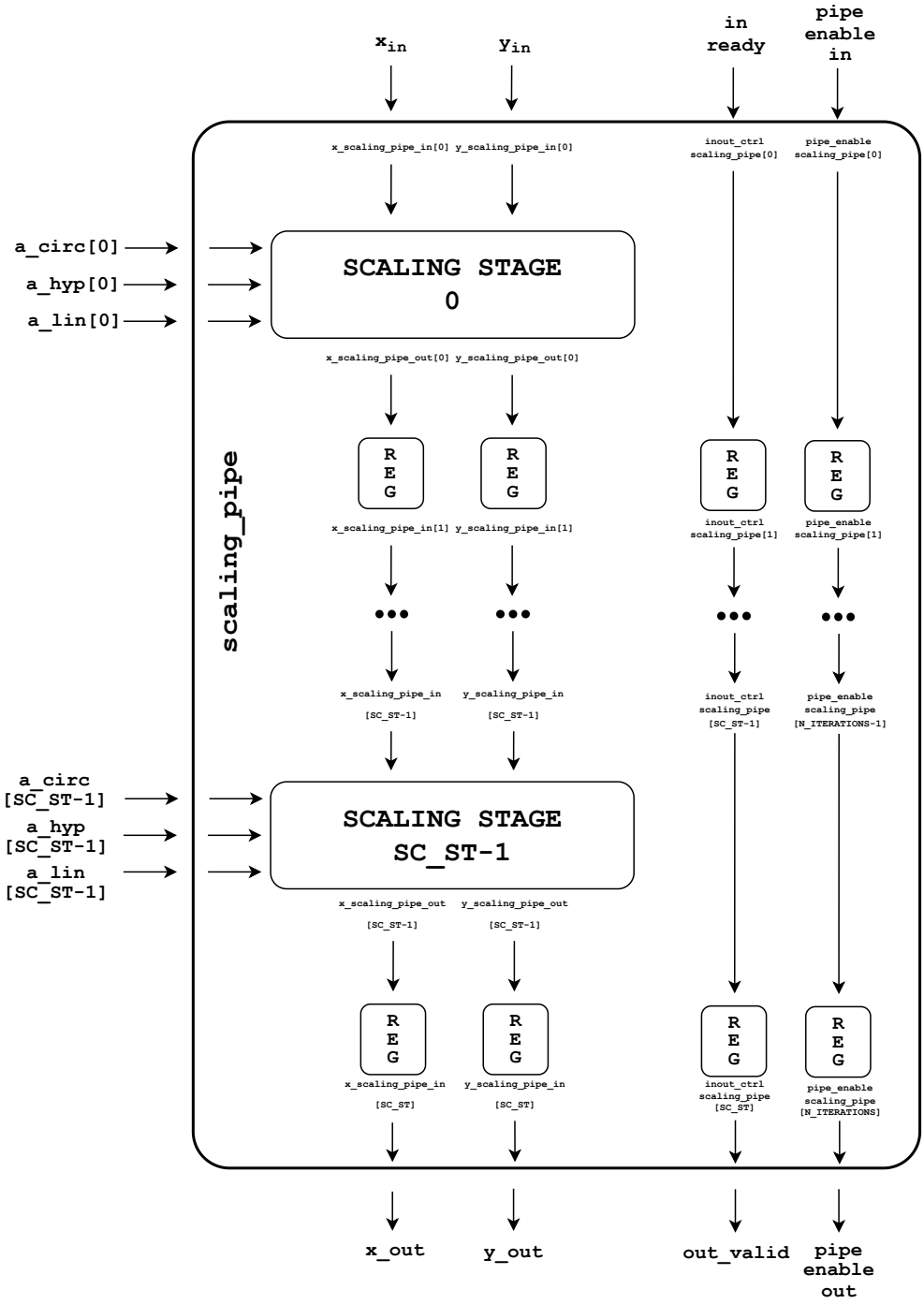


Figure 3.12: Scaling pipeline. `SC_ST` stands for `SCALING_STAGES`.

that can fit the internal fixed-point representation for such format. Being it equal to 18 for x and y in half precision (Section 3.3.1), the maximum acceptable shift amount is 17, leading to `SCALING_STAGES` = 6.
For what concerns z, instead, no scaling factor compensation is required and, hence, it is only delayed by `SCALING_STAGES` clock cycles before arriving at the output section.

Iteration (stage number)	**Circular coordinates $a_{1,i}$**	**Hyperbolic coordinates $a_{-1,i}$**
1	-2	2
2	4	4
3	-5	0
4	6	6
5	0	-6
6	17	0
7	-20	-20
8	0	-21

Table 3.2: Scaling coefficients for the scaling pipeline [29]. No coefficients are described for linear coordinates, since no scaling factor compensation is required.

Scaling stage

The block diagram of a generic scaling stage is shown in Figure 3.13. The main components are:

- Two fixed-point adders, one for each variable x and y.
- Several multiplexers to choose the correct operands
- A control unit to drive the selection signals:

– `add_sub` selects the correct operation to be performed by the adders, depending on the sign of the $a_{m,i}$ coefficient and on the working mode. A low value means addition, while a high one subtraction.
– Based on the coordinates system, `sel` chooses the correct value of x and y shifted right by $|a_{m,i}|$ bits. Once again, since all the $a_{m,i}$ coefficients are known in advance, shift operations are hardwired. Hence, no Barrel shifters are required.
– `is_zero` is set to 1 whenever $a_{m,i}$ is equal to zero, which means that no addition or subtraction has to be performed by the internal adders. In such case, the incoming operand is added to zero.

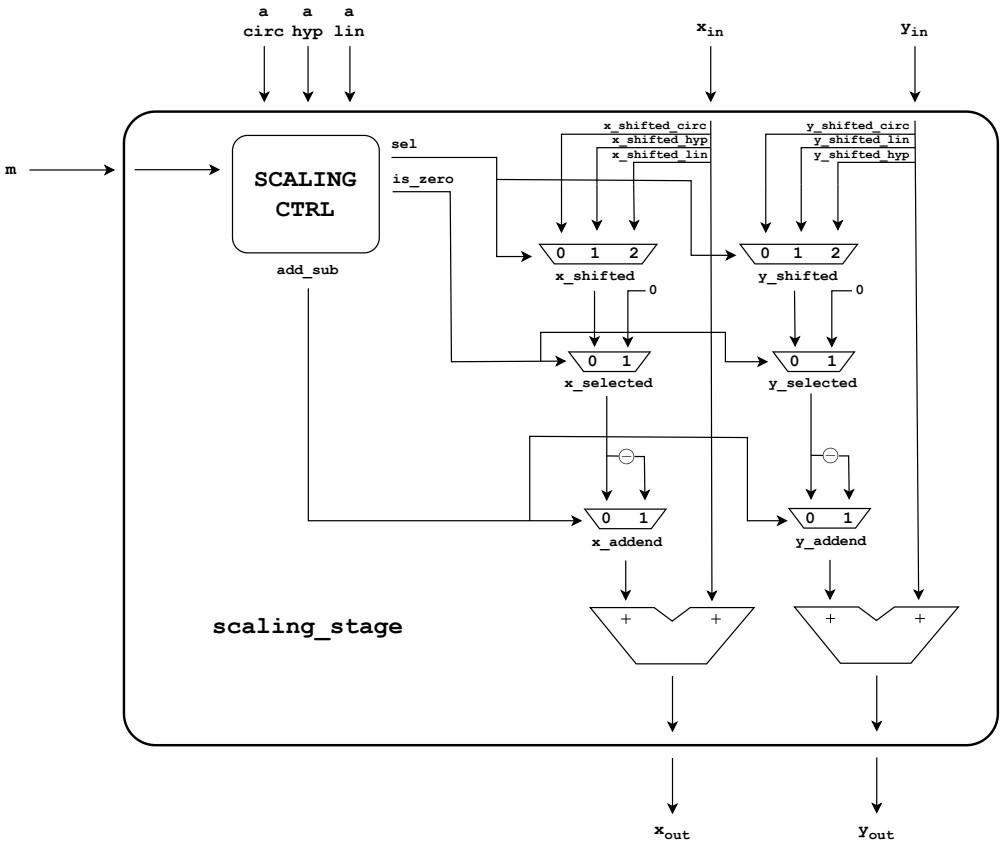


Figure 3.13: Block diagram of a scaling stage.

At this point, it can now be easily noticed how adopting this scaling strategy instead of the multiplier-based one leads to pipeline balance, since Figures 3.11 and 3.13 clearly show how similar the two types of stages are in terms of delay.

3.4.3 Lookup table

The last block that completes the fixed-point CORDIC top module is the lookup table, whose purpose is to store the values of the $\alpha_{m,i}$ angles and of the $a_{m,i}$ coefficients and to provide each stage with the required constants. In order to be consistent with the operations performed by the CORDIC pipeline, the stored angles follow the internal custom fixed-point format. This means that

$$w_{single,\ z} \cdot \texttt{N\_ITERATIONS} \cdot 3 = 30 \cdot 29 \cdot 3 = 2610 \text{ bits}$$
$$w_{half,\ z} \cdot \texttt{N\_ITERATIONS} \cdot 3 = 18 \cdot 15 \cdot 3 = 810 \text{ bits}$$

are required to store the $\alpha_{m,i}$ angles for single and half precision respectively.
For what concerns, instead, the $a_{m,i}$ coefficients, they are described by six bits, because their largest absolute value is 21[1]. As a consequence,

$$\texttt{SCALING\_STAGES} \cdot 6 \cdot 2 = 8 \cdot 6 \cdot 2 = 96 \text{ bits}$$
$$\texttt{SCALING\_STAGES} \cdot 6 \cdot 2 = 6 \cdot 6 \cdot 2 = 72 \text{ bits}$$

are required to store the $a_{m,i}$ coefficients for single and half precision respectively.

3.5 Post-processing block

The post-processing block, which is shown in Figure 3.14, is in charge of converting back the fixed-point results into floating-point ones, as well as managing special cases and producing the output status flags.
The main operations performed by this block are:

1. **Zero detection**: check if the CORDIC algorithm execution led to a zero result.
2. **Leading one detection, alignment and rounding**: after converting each number in sign-magnitude form, find the position of the leading one and align the mantissas, in order to have a unitary integer part.
3. **Exponent update and FLP packing**: update the reference exponent based on the alignment outcome and pack the floating-point result.
4. **Outputs selection and status flags generation**: select the correct output to be provided and generate the output status flags based on both the input ones and on specific conditions on the results.

In the following sections, each of these operations will be described in detail.

[1] One additional bit is required to represent signed values.

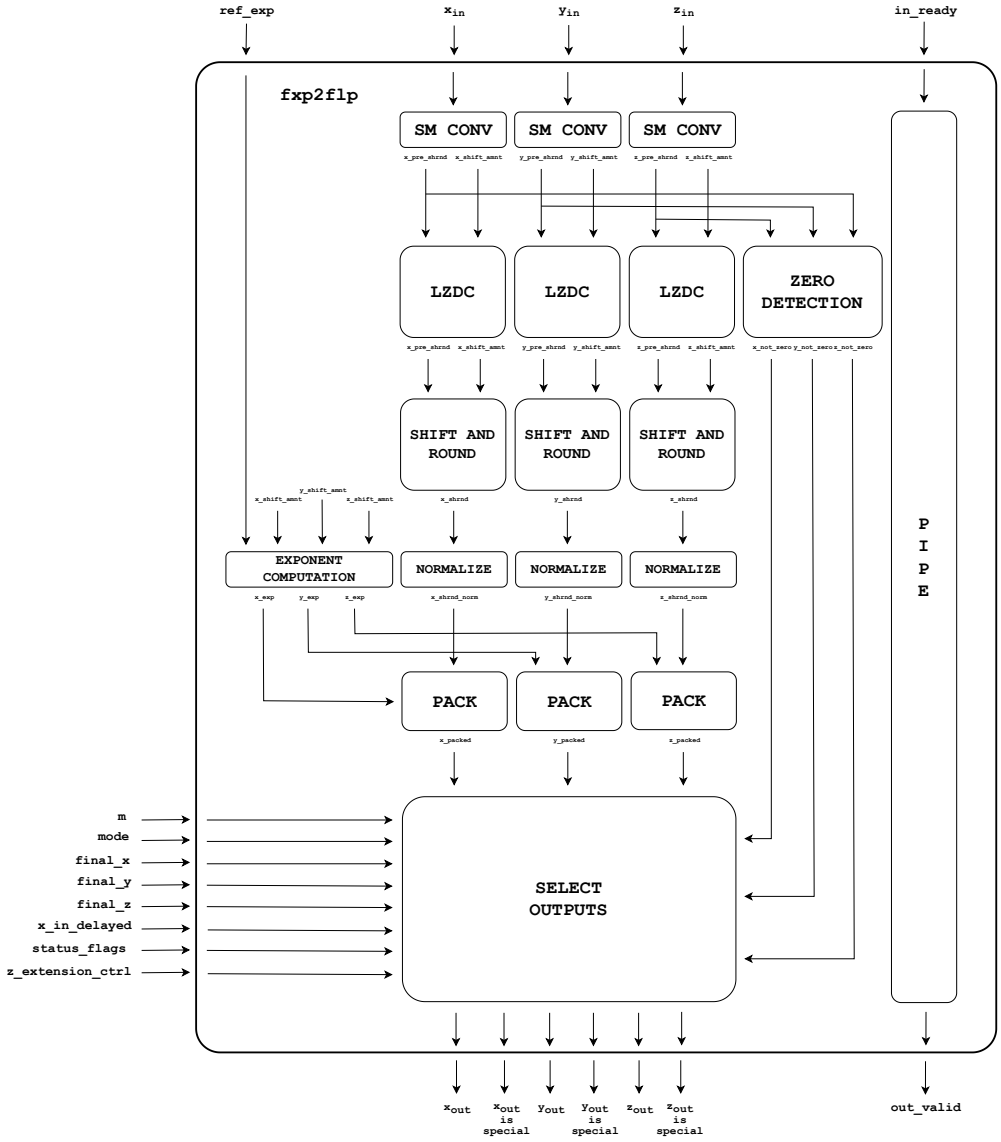


Figure 3.14: Block diagram of the post-processing block.

3.5.1 Zero detection

The first step to be performed is the conversion of each number from 2's complement to sign-magnitude form, in order to properly analyze the results. The sign bit, however, is saved and used later to correctly fill the sign field of the final floating-point value. Once this has been done, zero detection is conducted to all the incoming

operands to tell the output selection block if any of them has to be considered as zero. In fact, assuming to receive a number whose fixed-point representation is formed by all zeros, if no zero detection is carried out, the post-processing block would consider it as $2^{(e_{ref}-\texttt{N\_MANTISSA}-\texttt{N\_GUARD})}$[1], since, assuming an infinite accuracy, it would seem that the leading one is to the right of the least significant bit, thus outside the available precision. This can be easily avoided by forcing the output to zero whenever certain conditions are met.

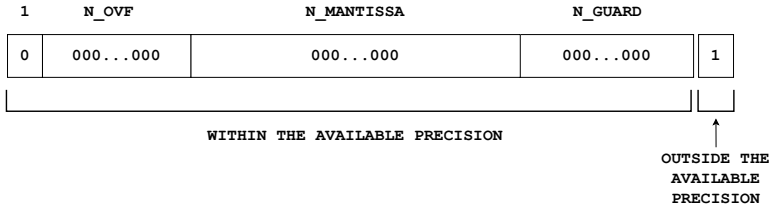


Figure 3.15: Example of what the post-processing block sees when an incoming operand is formed by all zeros.

Two strategies have been analyzed:

1. **Zero detection without zero-forcing**: each operand is considered as is, checking if all the bits are equal to zero. This means that the threshold is set to 1 LSB, because the least significant bit is the one that decides if the number is zero or not. However, albeit seeming effective, this approach led to a slightly larger average error when considering results that, independently on the input values, have to be zero at the end of the algorithm, namely z in rotation mode and y in vectoring mode. To cope with this, it has been found that moving the decision threshold to bit in position $\texttt{N\_GUARD} + 1$ brought to a lower error.

2. **Zero detection with zero-forcing**: the operands that have to be zero anyway, namely z in rotation mode and y in vectoring mode, are directly set to zero, whereas for the other ones a threshold equal to 1 LSB is adopted. This approach led to a lower average error, since the results that have to be zero are forced to be so, thus canceling their error contribution.

Considering that z in rotation mode and y in vectoring mode are not actually useful results, the second approach has been adopted, because it permits to increase the

[1]Further details on why this would be the result will be more clear in the following alignment-dedicated section.

accuracy when very small numbers, which would directly go to zero in the first strategy, are involved.
The zero detection block communicates with the output selection one by means of three signals, `x_not_zero`, `y_not_zero` and `z_not_zero`, which are raised whenever the corresponding operand has to be considered nonzero.

3.5.2 Leading one detection, alignment and rounding

When dealing with floating-point representation, the fixed-point number to be converted has to be aligned, in order to have a unitary integer part. This means that:

- If the integer part is greater than 1, the number has to be right-shifted until it become so.
- If the integer part is equal to zero, the number has to be left-shifted to have a unitary hidden bit.

This process is performed by finding the position of the leading one, which will, at the end, represent the hidden bit, and by shifting the number accordingly. The LZDC block[1] is in charge of taking care of this task. It receives the operands in sign-magnitude form, which permits to neglect sign bits for negative numbers, and returns the amount by which the number has to be adjusted, that is `x_shift_amnt`, `y_shift_amnt` and `z_shift_amnt`. In particular, the following criteria have been adopted to make later operations easier:

$$\begin{aligned} \texttt{shift\_amnt} &< 0 \implies \text{left-shift} \\ \texttt{shift\_amnt} &> 0 \implies \text{right-shift} \end{aligned}$$

Given the availability of `N_OVF` bits for the integer part and of `N_MANTISSA` + `N_GUARD` bits for the fractional one, the maximum amounts of shift that can be performed are

- `N_OVF` - 1 for the largest number
- - (`N_MANTISSA` + `N_GUARD`) for the smallest one.

[1]The name LZDC stands for Leading Zero Detection and Counting, even if the module basically looks for the leading one. In order to follow [29], it has been decided to leave the same name.

Being N_MANTISSA + N_GUARD very likely to be larger than N_OVF, the output signals of the LZDC block can be correctly described by a number of bits equal to $log_2(\texttt{N\_MANTISSA} + \texttt{N\_GUARD}) + 1$, where the additional bit takes into account negative amounts.
Once x_shift_amnt, y_shift_amnt and z_shift_amnt have been computed, they are read by the block in charge of adjusting and rounding the operands accordingly. Given the presence of N_GUARD bits, this operation has to be mixed with a fixed right-shift by N_GUARD position, so that, at the end, the fractional part is only N_MANTISSA bits wide. Hence, the incoming adjustment amounts are increased by such number to obtain the effective shift ones shift_amnt_eff, which complies with the same rules as before, namely a left adjustment for negative values and a right one for positive values. Once this quantity is known, the actual shift operation can take place and the exponent of each operand can be updated by adding shift_amnt_eff to the corresponding reference one.
For what concerns rounding, it is actually carried out only when a number has been right-shifted, since some bits are being lost when going outside the available width. Therefore, in order to achieve the maximum possible accuracy, additional padding bits are temporarily added to the right of each operand to store the outgoing ones. Being the maximum right-shift equal to N_GUARD + (N_OVF - 1), such value has been used to determine the number of padding bits.

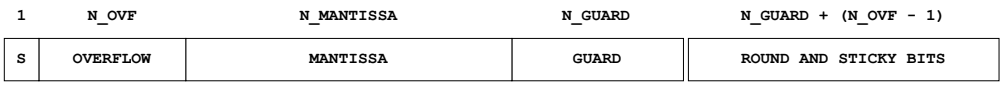


Figure 3.16: Internal fixed-point format for the rounding process.

In order to obtain better performance, the rounding to the nearest even method has been implied, since it promises to cancel the rounding bias. This approach exploits two bits, the round and sticky ones, to take decisions on the rounding direction. Assuming to have the ideal number

$$x = 1 \ . \ x_{-1}x_{-2}...x_{-n}x_{-(n+1)}...x_{-\infty}$$

and only a n-bit precision, it means that all the digits after x_{-n} have to be dropped. Among these lost bits, the most significant one is called round bit and the others are ORed together to form the sticky bit. Then, the following rules are applied:

- If the sticky bit is 1, the round bit is added to the truncated number.
- If the sticky bit is 0, two cases are possible:
  - If the round bit is 0, the truncated number is left unchanged.

– If the round bit is 1, the LSB of the truncated number is added to it, in order to obtain an even final result.

Sticky bit S	**Round bit R**	**Bit to be added**
0	0	0
0	1	$x_{-(n+1)}$
1	X	R

Table 3.3: Rounding to the nearest even method.

Being the signal extended with the additional padding bits `in_ext_shifted`, the round and sticky bits are:

- R = `in_ext_shifted[N_GUARD+N_OVF-2]`
- S = `|in_ext_shifted[(N_GUARD+N_OVF-3):0]`[1]

After rounding, the last operation to be carried out before having the final mantissa is normalization, whose need could rise whenever rounding led to a value with an integer part greater than one. In such case, the number has to be right-shifted by one position and the exponent has to be further increase by one.
At the end of all these steps, all the operands have `N_MANTISSA` fractional bits and a unitary integer part, which means that they are ready to be packed together to form the final result.

3.5.3 Exponent update and FLP packing

The last operations to be performed before obtaining the final results in floating-point are the exponent update by the shift quantities and the FLP packing of the three fields.
Thanks to the criteria that have been explained in the previous section, the first step basically adds the signed effective shift amount of each operand to the respective

[1] In SystemVerilog, the reducing OR is performed by putting `|` before the involved signal.

reference exponent. In fact, a left direction (negative amount) means that the received number is smaller than $2^{e_{ref}}$, causing e_{ref} to be reduced, whereas a right direction (positive amount) is associated to a number greater than $2^{e_{ref}}$, bringing an increase of e_{ref}. In addition to this, normalization potentially leads to an additional increment of the exponent by one, whenever the operand has been right-shifted by one position. Finally, the corresponding bias of the FLP format is added to each exponent to obtain the final values.
At this point the three floating-point fields are ready to be packed together to form the final result. The sign field is directly taken from the sign bit of the fixed-point representation, the mantissa one is obtained by dropping the hidden bit and by taking the `N_MANTISSA` fractional ones and the exponent bits are taken from the outcome of the previous operation.

3.5.4 Outputs selection and status flags generation

Before providing meaningful results, some additional checks have to be carried out to ensure that the output values are correct and to set the output status flags accordingly. These are done by opportunely looking at the received `final_x`, `final_y` and `final_z` signals that, created by the pre-processing block, traveled through the entire pipeline and reached the post-processing one. However, prior to this, the incoming operands have first to be remapped based on the decisions taken by the z mapping block. Being `x_in` and `y_in` the received numbers and `x_int` and `y_int` the internally remapped ones, this leads to:

- `z_extension_ctrl` = `NO_MAP`:
  - `x_int` = `x_in`
  - `x_not_zero` = `x_not_zero`
  - `y_int` = `y_in`
  - `y_not_zero` = `y_not_zero`
- `z_extension_ctrl` = `FIRST_RANGE`:
  - `x_int` = `-y_in`
  - `x_not_zero` = `y_not_zero`
  - `y_int` = `x_in`
  - `y_not_zero` = `x_not_zero`
- `z_extension_ctrl` = `SECOND_RANGE`:
  - `x_int` = `-x_in`

  - x_not_zero = x_not_zero
  - y_int = -y_in
  - y_not_zero = y_not_zero

- **z_extension_ctrl = THIRD_RANGE**:
  - x_int = y_in
  - x_not_zero = y_not_zero
  - y_int = -x_in
  - y_not_zero = x_not_zero

Assuming to consider the generic signals final_sig, status_flags and sig_is_special (the sig word stands for any among x, y and z), the following conditions have to be met to provide valid outputs:

1. **final_sig = NORMAL_FLAG**:
   - If the reference exponent is equal to the one related to infinities, it means that the result can either be $\pm\infty$ or zero (NaNs can be excluded because of the presence of NORMAL_FLAG).
     - If the zero detection block has found a zero operand, the corresponding output is forced to zero and the sig_is_special signal is set to NORMAL_FLAG, given that the result is not a special value.
     - If the zero detection block has not found a zero operand, the corresponding output is set to $\pm\infty$ and the sig_is_special signal is set to PLUS_INF_FLAG or MINUS_INF_FLAG depending on the sign of the result.

     In either of the two cases, the status_flags signal is set to NO_EXCEPTIONS, since the presence of NORMAL_FLAG assumes that no status flags have been raised at the input section for the considered operand.
   - If the final biased exponent is larger than the maximum admitted one, namely $e_{max}+b$, it means that overflow occurred. Hence, the corresponding output is forced to $\pm\infty$ based on its sign, the sig_is_special signal is set to PLUS_INF_FLAG or MINUS_INF_FLAG and the status_flags signal is equal to OVERFLOW.
   - If the final biased exponent is smaller than the minimum admitted one, namely $e_{min} + b$, it means that underflow occurred. Hence, the corresponding output is forced to zero, the sig_is_special signal is set to NORMAL_FLAG and the status_flags signal is equal to UNDERFLOW.

- In any other case, the incoming operand is forwarded to the output section, the `sig_is_special` signal is set to `NORMAL_FLAG` and the `status_flags` signal is set to `NO_EXCEPTIONS`.

2. **`final_sig` = `POS_NAN_FLAG`**: the corresponding output is forced to be a NaN and `sig_is_special` is set to `POS_NAN_FLAG`. The `status_flags` signal is set to `INVALID`.

3. **`final_sig` = `PLUS_INF_FLAG`**: the corresponding output is forced to be $+\infty$ and `sig_is_special` is set to `PLUS_INF_FLAG`. The `status_flags` signal is set to `NO_EXCEPTIONS`.

4. **`final_sig` = `MINUS_INF_FLAG`**: the corresponding output is forced to be $-\infty$ and `sig_is_special` is set to `MINUS_INF_FLAG`. The `status_flags` signal is set to `NO_EXCEPTIONS`.

5. **`final_sig` = `ZERO_FLAG`**: the corresponding output is forced to be zero and `sig_is_special` is set to `NORMAL_FLAG`. The `status_flags` signal is set to `NO_EXCEPTIONS`.

It has to be noted that, as anticipated in Section 3.3.5, in order to properly handle floating-point infinities in fixed-point computations, the x input travels through the pipeline and reaches the post-processing block. Therefore, the arrangements that have just been described are valid for x only if the unit is not working in linear coordinates. In such case, whenever `final_x` is equal to `NORMAL_FLAG`, the original input value is forwarded to the output section, the `status_flags` signal is set to `NO_EXCEPTIONS` and the `x_is_special` signal can be:

- `NORMAL_FLAG`: the input value is not an infinity.
- `PLUS_INF_FLAG`: the input value is $+\infty$.
- `MINUS_INF_FLAG`: the input value is $-\infty$.

Chapter 4

X-HEEP platform

As a final work, the CORDIC architecture has been integrated into X-HEEP [9], an open-source 32-bit RISC-V microcontroller system whose block diagram is shown in Figure 4.1. The considered typical usage implies it as an autonomous external peripheral that, once configured, directly transfers data to and from the memory.

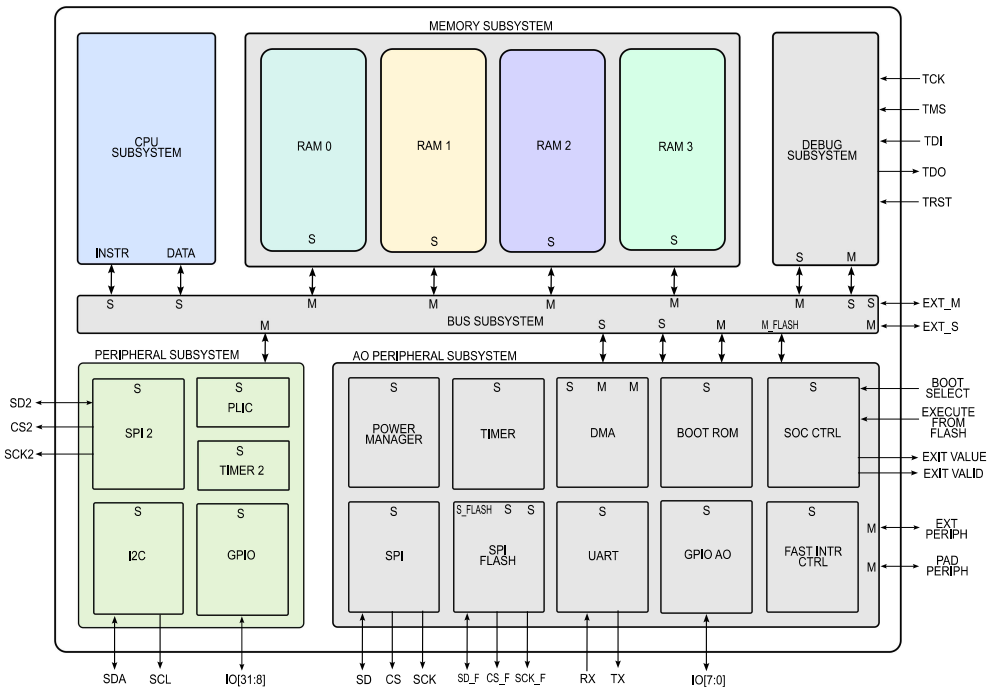


Figure 4.1: Block diagram of the X-HEEP microcontroller system [9].

4.1 RTL files creation

The X-HEEP platform is highly configurable and lets the users select the system parameters that best fit their needs. This choice includes:

1. CPU type, all of which are based on the RISC-V CPU and have a 32-bit datapath:
   - **CV32E20**, a 2-stage pipeline and low cost-oriented fork of the Ibex core. It supports a reduced set of instructions, which include the Integer (I), Integer Multiplication and Division (M) and Compressed (C) extensions.
   - **CV32E40P**, a 4-stage pipeline which supports the Integer (I), Integer Multiplication and Division (M), Single-Precision Floating-Point (F) and Compressed (C) extensions. It was previously known as RI5CY.
   - **CV32E40X**, a fork of the CV32E40P core that includes a general purpose extension interface by which the support for custom instructions can be added.
2. BUS type:
   - **onetoM**: only one master at a time can access the bus.
   - **NtoM**: multiple masters can access the bus at the same time.
3. Number of interleaved memory banks, from 0 to 8.
4. Number of memory banks, from 2 (one for code and one for data) to 16 minus the number of interleaved ones. Each of them is 32 kB wide.

In order to efficiently integrate and test the CORDIC accelerator, it has been decided to rely on:

- The CV32E40P CPU, since it supports single precision floating-point.
- The NtoM BUS type, since, working in pipeline, the accelerator has to issue simultaneous read and write requests, if possible.
- 3 memory banks to fit the input and output data, as well as the reference values. In fact, assuming 1024 32-bit elements for each of the three inputs and outputs, the total amount of needed bytes is:

$$\begin{aligned} n_{bytes} &= 4 \cdot (n_{inputs} + n_{outputs} + n_{ref}) \\ &= 4 \cdot (3 \cdot 1024 + 3 \cdot 1024 + 3 \cdot 1024) \\ &= 36 \cdot 1024 = 36\ kB \end{aligned} \tag{4.1}$$

which means that 2 banks of 32 kB are enough for data. Even if no check is being done, thus not storing the reference values, two banks are still not sufficient, since the test application compilation gives memory space errors.

Once all the parameters have been set, the RTL modules can be generated by running the `make mcu-gen` command, which will exploit the provided templates to create the necessary files.

4.2 Wrapper design

The accelerator interface and its intrinsic behavior made necessary the implementation of a wrapper, which is responsible for the management of the CORDIC unit configuration and data transfers. Its block diagram is shown in Figure 4.2 for single precision, 4.3 for half precision and 4.4 for transprecision. The behavior of the wrappers supporting only one floating-point format is practically the same, except for the fact that the half precision one implies two parallel CORDIC units, one for the lower and one for the upper part of the 32-bit input and output data. For what concerns the third one, instead, it follows the transprecision concept [32], which implies the possibility to work with either of the two supported precisions. In fact, depending on the `tp_bit` signal, the wrapper will send the right data to the respective CORDIC unit and will opportunely take care of the produced results. When `tp_bit` is high, the behavior of the wrapper is the same as Figure 4.3, thus splitting the inputs and outputs into two 16-bit parts and forwarding them to the half precision CORDIC units, whereas, when it is low, the behavior is the same as Figure 4.2, thus sending the data to the single precision CORDIC unit. Since, apart from this, the other blocks (FSMs, OBI interface managers) are identical, the following sections, which describe each part of the wrapper, will not take care of specifying which one is being referred to.

4.2.1 Interface signals

As anticipated earlier, the considered test case implies the CORDIC accelerator as an external peripheral that, once configured, can work autonomously. To achieve this, it was necessary to include in the wrapper:

- A slave port for configuration purposes through the register interface. This will be connected to the `ext_peripheral_slave` port of X-HEEP.
- Two master ports to issue read and write requests to the internal DMA. They will be packed into a 2-element array and connected to the `ext_xbar_master` port of X-HEEP.

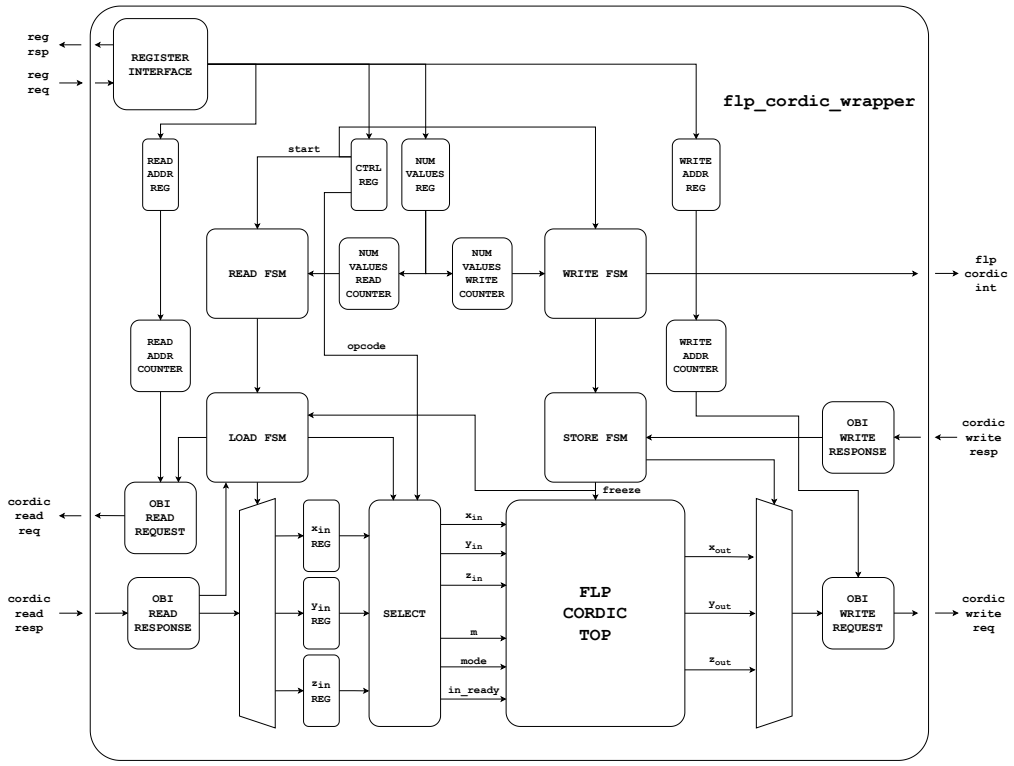


Figure 4.2: Block diagram of the wrapper for single precision.

Since the accelerator interface differs from the exposed ones of X-HEEP, some arrangements have been made to ensure a proper integration. In order to make this efficiently, thus letting the CPU really offload the execution of the CORDIC unit supported functions, it has been decided to rely on the register interface only to set the control and start signals. As a result, data is transferred autonomously by issuing read and write requests directly to the internal DMA through the exposed master ports. This choice led to the following interface signals:

- **`clk_i`**: the system clock, to be directly forwarded to the CORDIC unit.
- **`rst_n_i`**: the reset signal, to be directly forwarded to the CORDIC unit.
- **`reg_req_i`**: the register interface request signals. It uses the Register Interface [33] and is of type `reg_req_t`.
- **`reg_rsp_o`**: the register interface response signals. It uses the Register Interface and is of type `reg_rsp_t`.

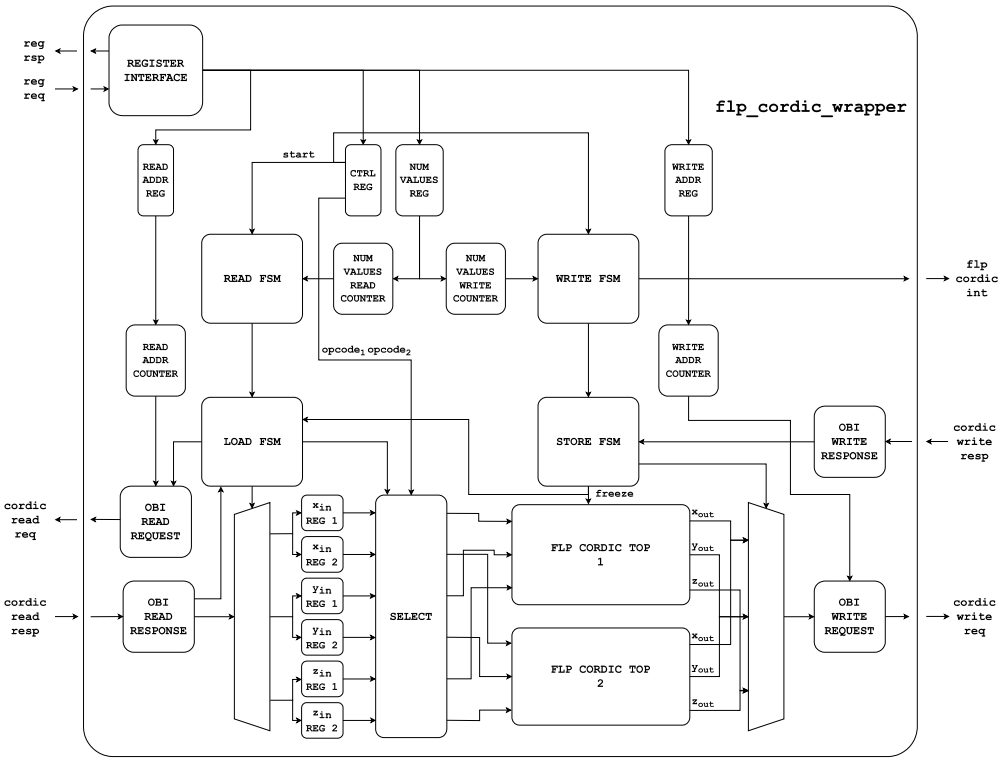


Figure 4.3: Block diagram of the wrapper for half precision.

- **`cordic_read_ch0_req_o`**: the read request signals for the DMA channel 0. It uses the Open Bus Interface (OBI) [34] and is of type `obi_req_t`.
- **`cordic_read_ch0_resp_i`**: the read response signals for the DMA channel 0. It is OBI-compliant and of type `obi_resp_t`.
- **`cordic_write_ch0_req_o`**: the write request signals for the DMA channel 0. It is OBI-compliant and of type `obi_req_t`.
- **`cordic_write_ch0_resp_i`**: the write response signals for the DMA channel 0. It is OBI-compliant and of type `obi_resp_t`.
- **`flp_cordic_int_o`**: the interrupt signal, raised when the CORDIC unit has completed the computation.

The `reg_req_t` and `reg_rsp_t` types include the following signals:

1. **`reg_req_t`**:

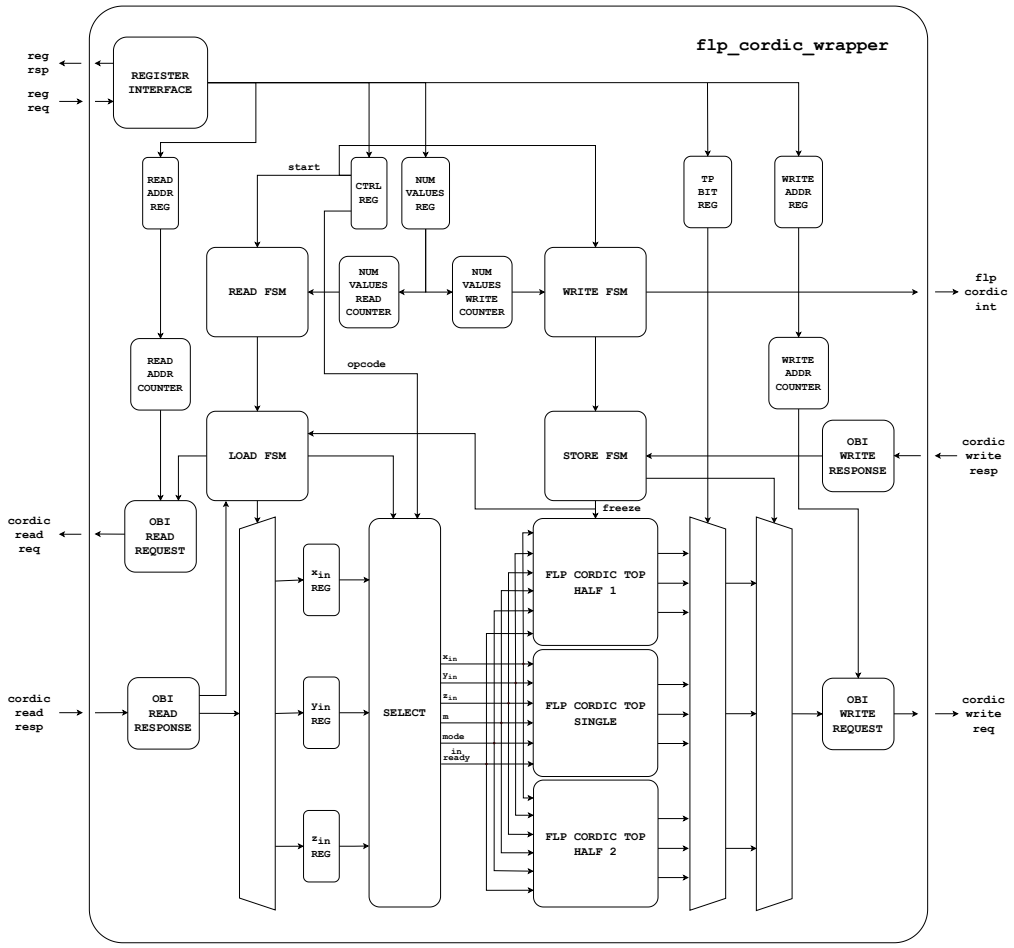


Figure 4.4: Block diagram of the wrapper for transprecision.

- **`logic valid`**: valid request signal.
- **`logic write`**: write enable signal, high for write transactions and low for read ones.
- **`logic [3:0] wstrb`**: write strobe signal, to select the number of bytes to be written.
- **`logic [31:0] addr`**: address of the transfer.
- **`logic [31:0] wdata`**: data to be written.

2. **`reg_rsp_t`**:

- **`logic error`**: error signal, high if the transaction has failed.

- **logic ready**: ready signal, high if the transaction has been completed.
- **logic [31:0] rdata**: data read from the address.

As described in [33], in order to efficiently implement the register interface, its creation process is based on the compilation of a .hjson file with the description of the registers and of their internal fields. Then, the `regtool.py` Python script will take care of generating the necessary RTL files, as well as the C headers and SystemVerilog packages that collect the internal offsets of the registers. As a result, no address decoder is needed, since the register interface itself will directly forward the requests to the right element. It has been chosen to allocate the following control registers:

- **ctrl_reg**: it sets the opcode, the start signal and the transprecision bit.
- **read_address_reg**: it sets the initial address for the read requests.
- **write_address_reg**: it sets the initial address for the write requests.
- **n_values_reg**: it sets the number of values to be processed.
- **int_en_reg**: it enables the interrupt signal.

For what concerns status registers, it has been decided not to include them, since they would be mainly related to the floating-point status flags of the outputs. Given that the produced results are directly written to memory and are not transferred through the register interface, it would cause the CPU to continuously read and check the status flags before that the output values can actually be stored in memory. This would lead to a waste of time and, therefore, has been avoided.
For a single-input/single-output use, instead, thus where data is only transferred through the register interface, it would be meaningful to include status registers storing the produced outputs and the status flags.
The `obi_req_t` and `obi_resp_t` types include the following signals:

1. `obi_req_t`:
   - **logic req**: transfer request signal.
   - **logic we**: write enable signal, high for write transactions and low for read ones.
   - **logic [3:0] be**: byte enable signal, to select the number of bytes to be transferred.
   - **logic [31:0] addr**: address of the transfer.
   - **logic [31:0] wdata**: data to be written.

2. **obi_resp_t**:

    - **logic gnt**: transfer grant signal.
    - **logic rvalid**: valid response signal, to signal the availability of meaningful signals in the response phase.
    - **logic [31:0] rdata**: data read from the address.

Given that no framework is available to automatically manage the OBI signals, this has to be done manually. As a result, whenever the read FSM [1] is in the `RUNNING` state and the load one is not in the `SEND_INPUTS` state, the following signals have to be set for the read request:

- `cordic_read_ch0_req_o.req` = `1'b1`
- `cordic_read_ch0_req_o.we` = `1'b0`
- `cordic_read_ch0_req_o.be` = `4'b1111`
- `cordic_read_ch0_req_o.addr` = `read_address`
- `cordic_read_ch0_req_o.wdata` = `32'b0`

where `read_address` comes from a counter that updates the read address by 4 each time that the read request received a positive grant and that the next state of the load FSM is not `SEND_INPUTS`. For what concerns write operations, instead, the following signals have to be set when the write FSM is in the `RUNNING` state and the store one is not in the `NO_STORE` state:

- `cordic_write_ch0_req_o.req` = `1'b1`
- `cordic_write_ch0_req_o.we` = `1'b1`
- `cordic_write_ch0_req_o.be` = `4'b1111`
- `cordic_write_ch0_req_o.addr` = `write_address`
- `cordic_write_ch0_req_o.wdata` will be equal to any among `x_out`, `y_out` and `z_out` depending on the state of the store FSM [1].

[1] Further explanations about the internal Finite State Machines are provided in the following section.

Once again, `write_address` comes from a counter that updates the write address by 4 each time that the write request received a positive grant.
For what concerns, instead, the interrupt line, this is raised whenever the `int_en_reg` enables it and the accelerator has completed all the write operations. This is done by checking that the write FSM is in the `RUNNING` state and that the total number of remaining values to be stored in memory reached zero.

4.2.2 Internal FSMs

The wrapper includes four Finite State Machines (FSMs) that take care of solving the main issue that rises when dealing with data transfers. In fact, at each clock cycle, the CORDIC pipeline, when full, requires three inputs and produces three outputs, but the integrated DMA has only one channel for read operations and one for write operations. This means that only up to two elements can be contemporarily moved from and to the memory, which would cause the other four values to be lost. Therefore, four Finite State Machines (FSMs) have been created to take care of this:

- **Read FSM**: it takes care of starting and finishing the inputs read operation.
- **Load FSM**: it selects which register has to sample the incoming data.
- **Write FSM**: it takes care of starting and finishing the outputs write operation.
- **Store FSM**: it selects which output has to be forwarded for writing.

The read and write FSMs are very simple ones and include three states:

- `READY`: the accelerator is either ready to start issuing transfer requests or has completed its work. Both FSMs start from this state at reset and go back to it when the computation has been completed. They can go to next state only when the start signal field of the control register becomes high.
- `STARTED`: the unit uploads the initial address for both read and write operations and is ready to issue transfer requests. It directly goes to next state in the following clock cycle.
- `RUNNING`: the accelerator is actually transferring data by issuing or not requests depending on the actual state of the load and store FSMs.

The load FSM is, instead, a bit more complex and includes five states:

- **NO_LOAD**: the unit is not asking for data to be loaded. This is the initial state and the one to which the FSM goes back when the computation has been completed. It goes to next state when the read FSM is in the RUNNING state, since this means that the first input is being read and can be sampled during the next clock cycle.

- **LOAD_X**: the unit is loading the first input into the corresponding register. It goes to next state only when the read transaction has been completed successfully, which only happens when the rvalid signal of the read response is high.

- **LOAD_Y**: the unit is loading the second input into the corresponding register. It goes to next state only when the read transaction has been completed successfully.

- **LOAD_Z**: the unit is loading the third input into the corresponding register. It goes to next state only when the read transaction has been completed successfully.

- **SEND_INPUTS**: all the inputs have been correctly loaded, so the computation can start by setting the in_ready signal to 1. In order to synchronize read and write operations, thus avoiding to produce further outputs when it is not possible to write them, in_ready goes high only if the load FSM is in the SEND_INPUTS state and write requests are not being stalled, which happens when both the req signal of the write request and the gnt one of the write response are high. It goes again to LOAD_X when in_ready has been successfully set or to NO_LOAD if the number of inputs to be read reaches zero.

The store FSM is includes four states:

- **NO_STORE**: the unit is not asking for data to be stored. This is the initial state and the one to which the FSM goes back when the computation has been completed or when it is waiting for a new set of outputs. It goes to next state when the out_valid signal is high, which means that new results are available.

- **STORE_X**: the unit is storing the first output. It goes to next state only when the write transaction has been completed successfully, which happens when the gnt signal of the write response is high.

- **STORE_Y**: the unit is storing the second output. It goes to next state only when the write transaction has been completed successfully.

- `STORE_Z`: the unit is storing the third output. It goes again to `STORE_X` only if, in this cycle, both `gnt` and `out_valid` are high. Otherwise, it goes to `NO_STORE` to wait for the next set of outputs.

It has to be noted that synchronization between read and write operations is obtained by exploiting the `in_ready`, `out_valid` and `freeze` signals. In fact, whenever read operations are stalled, `in_ready` cannot go high because the load FSM will not be able to reach the `SEND_INPUTS` state. This causes `out_valid` to be kept low, which, in turn, prevents write requests from being issued. If, instead, the unit cannot write to memory, the `freeze` signal is raised, thus causing:

- The load FSM to be kept in the `SEND_INPUTS` state, thus preventing new inputs from being sampled.
- `in_ready` to be kept low, therefore not allowing the computation to start.
- The entire CORDIC pipeline to be frozen and, therefore, not producing new outputs.

To accomplish this last point, an input signal, called `ext_freeze`, has been added to the CORDIC unit to disable all its internal registers.

4.2.3 CORDIC unit management

In order to transform the raw setting of the `mode` and `m` signals of the CORDIC unit into something more user-friendly, it has been decided to rely on a set of predefined working modes taken from [29] and accessible through an opcode. To maximize variety, a total number of 17 configurations have been made available, including both generic and specific ones. For the latter, an input selection stage has been added to obtain the desired functions. Table 4.1 shows the available configurations, their corresponding opcodes and the chosen inputs to achieve the final results.

Configuration	Opcode	Inputs	Outputs
`NOP`	00000	$x = 0$ $y = 0$ $z = 0$ `mode` $= 0$ `m` $= 0$	$x = 0$ $y = 0$ $z = 0$

Continued on next page

Configuration	Opcode	Inputs	Outputs
SIN_COS	00001	$x = 1$ $y = 0$ $z = z_{in}$ mode = ROTATION m = CIRCULAR	$x = cos(z_{in})$ $y = sin(z_{in})$ $z = 0$
ROT_CIRC	00010	$x = x_{in}$ $y = y_{in}$ $z = z_{in}$ mode = ROTATION m = CIRCULAR	$x = x_{in} \cdot cos(z_{in}) - y_{in} \cdot sin(z_{in})$ $y = x_{in} \cdot sin(z_{in}) + y_{in} \cdot cos(z_{in})$ $z = 0$
SINH_COSH	00011	$x = 1$ $y = 0$ $z = z_{in}$ mode = ROTATION m = HYPERBOLIC	$x = cosh(z_{in})$ $y = sinh(z_{in})$ $z = 0$
ROT_HYP	00100	$x = x_{in}$ $y = y_{in}$ $z = z_{in}$ mode = ROTATION m = HYPERBOLIC	$x = x_{in} \cdot cosh(z_{in}) + y_{in} \cdot sinh(z_{in})$ $y = x_{in} \cdot sinh(z_{in}) + y_{in} \cdot cosh(z_{in})$ $z = 0$
ROT_HYP_M	00101	$x = x_{in}$ $y = y_{in}$ $z = -z_{in}$ mode = ROTATION m = HYPERBOLIC	$x = x_{in} \cdot cosh(z_{in}) - y_{in} \cdot sinh(z_{in})$ $y = y_{in} \cdot cosh(z_{in}) - x_{in} \cdot sinh(z_{in})$ $z = 0$
ROT_LIN	00110	$x = x_{in}$ $y = y_{in}$ $z = z_{in}$ mode = ROTATION m = LINEAR	$x = x_{in}$ $y = y_{in} + z_{in} \cdot x_{in}$ $z = 0$

Continued on next page

Configuration	Opcode	Inputs	Outputs
ROT_LIN_M	00111	$x = x_{in}$ $y = y_{in}$ $z = -z_{in}$ mode = ROTATION m = LINEAR	$x = x_{in}$ $y = y_{in} - z_{in} \cdot x_{in}$ $z = 0$
VEC_CIRC	01000	$x = x_{in}$ $y = y_{in}$ $z = z_{in}$ mode = VECTORING m = CIRCULAR	$x = sign(x_{in}) \cdot \sqrt{x_{in}^2 + y_{in}^2}$ $y = 0$ $z = z_{in} + tan^{-1}(y_{in}/x_{in})$
VEC_CIRC_M	01001	$x = x_{in}$ $y = -y_{in}$ $z = z_{in}$ mode = VECTORING m = CIRCULAR	$x = sign(x_{in}) \cdot \sqrt{x_{in}^2 + y_{in}^2}$ $y = 0$ $z = z_{in} - tan^{-1}(y_{in}/x_{in})$
VEC_HYP	01010	$x = x_{in}$ $y = y_{in}$ $z = z_{in}$ mode = VECTORING m = HYPERBOLIC	$x = sign(x_{in}) \cdot \sqrt{x_{in}^2 - y_{in}^2}$ $y = 0$ $z = z_{in} + tanh^{-1}(y_{in}/x_{in})$
VEC_HYP_M	01011	$x = x_{in}$ $y = -y_{in}$ $z = z_{in}$ mode = VECTORING m = HYPERBOLIC	$x = sign(x_{in}) \cdot \sqrt{x_{in}^2 - y_{in}^2}$ $y = 0$ $z = z_{in} - tanh^{-1}(y_{in}/x_{in})$
VEC_LIN	01100	$x = x_{in}$ $y = y_{in}$ $z = z_{in}$ mode = VECTORING m = LINEAR	$x = x_{in}$ $y = 0$ $z = z_{in} + y_{in}/x_{in}$

Continued on next page

Configuration	Opcode	Inputs	Outputs
VEC_LIN_M	01101	$x = x_{in}$ $y = -y_{in}$ $z = z_{in}$ mode = VECTORING m = LINEAR	$x = x_{in}$ $y = 0$ $z = z_{in} - y_{in}/x_{in}$
EXPONENTIAL	01110	$x = x_{in}$ $y = x_{in}$ $z = z_{in}$ mode = ROTATION m = HYPERBOLIC	$x = x_{in} \cdot e^{z_{in}}$ $y = x_{in} \cdot e^{z_{in}}$ $z = 0$
ARCCOTANH	01111	$x = x_{in}$ $y = 1$ $z = 0$ mode = VECTORING m = HYPERBOLIC	$x = \sqrt{x^2 - 1}$ $y = 0$ $z = cotanh^{-1}(x_{in})$
ARCCOTAN	10000	$x = 1$ $y = y_{in}$ $z = \frac{\pi}{2}$ mode = VECTORING m = CIRCULAR	$x = \sqrt{1 + y_{in}^2}$ $y = 0$ $z = cotan^{-1}(y_{in})$

Table 4.1: Available configurations and their corresponding opcodes.

As it can be noticed from Table 4.1, based on the received opcode, the wrapper will take care of setting the actual inputs to be forwarded to the CORDIC module, without any need from the user to do so.
The last arrangement that has been made came from the aforementioned issue related to stalled write operations. In fact, whenever read requests cannot be satisfied, this can be easily handled by exploiting the handshake signals of the CORDIC unit and, thus, by keeping `in_ready` low, which will prevent the computation from starting. However, the same solution cannot be adopted for write operations, since the CORDIC module will continue to fetch new data and produce new results at each clock cycle as long as the pipeline is kept active. Furthermore, even blocking read requests is not sufficient, because the CORDIC unit will try to flush its pipeline anyway. In this context two solutions were possible, that is

either to temporarily store the results in a FIFO or to freeze the whole CORDIC pipeline. For this integration process, the latter method has been chosen as the most effective, since it both does not require any additional resources and does not cause any delay. To do so, the `ext_freeze` input signal has been added to the CORDIC module to disable, when high, all its internal registers, thus preventing them from being updated. This has, then, been connected to the `freeze` one of the wrapper, which is raised whenever the write FSM is in the `RUNNING` state and a write request received a negative grant.

Chapter 5

Obtained results

This chapter highlights the obtained results in terms of precision, timing, area occupation, power consumption and speedup.

5.1 Simulation results

The whole CORDIC architecture has been tested using QuestaSim and implying 10000 random values, which made it possible to obtain variegated outputs to perform comparisons, in terms of relative error, among the different internal fixed-point formats.

5.1.1 Preliminary choices

Before starting the extensive tests, it has been noted that, whenever the exponents difference between the x_{in} and y_{in} floating-point input operands is too large, the overall error starts to increase significantly. However, nothing related to this issue has been found in literature and, thus, a hypothesis has been made. In fact, assuming to be in single precision floating-point and to have, for example:

- $x_{in} = 1 =$ `32'h3F800000`, with exponent $e_x = 0$
- $y_{in} = 2^{-23} =$ `32'h34000000`, with exponent $e_y = -23$

the alignment and fixed-point conversion processes lead to the following results:

- $x =$ `32'h10000000`
- $y =$ `32'h00000010`

which means that, throughout the pipeline, the initial y operand will be practically considered as zero and the final result will be wrong. In addition, in order to double-check that this issue is not related to the global floating-point strategy of Section 3.1.3, also the C model that uses local FLP has been tested with the same values and provided similar results. As a consequence, a threshold has been set to the difference between the exponents of x_{in} and y_{in}, in order to overcome this problem.
The selection process of such threshold laid on the creation of a C code that, keeping the absolute value of x fixed to 1.9999999[1] and varying the one of y from $2^{e_{min}} \cdot 1.9999999$ to $2^{e_{max}} \cdot 1.9999999$, checks the obtained relative error for both precisions. An example of the obtained results is shown in Figures 5.1 and 5.2, where the relative error is plotted against the exponent difference for single and half precision in rotation mode and hyperbolic coordinates.

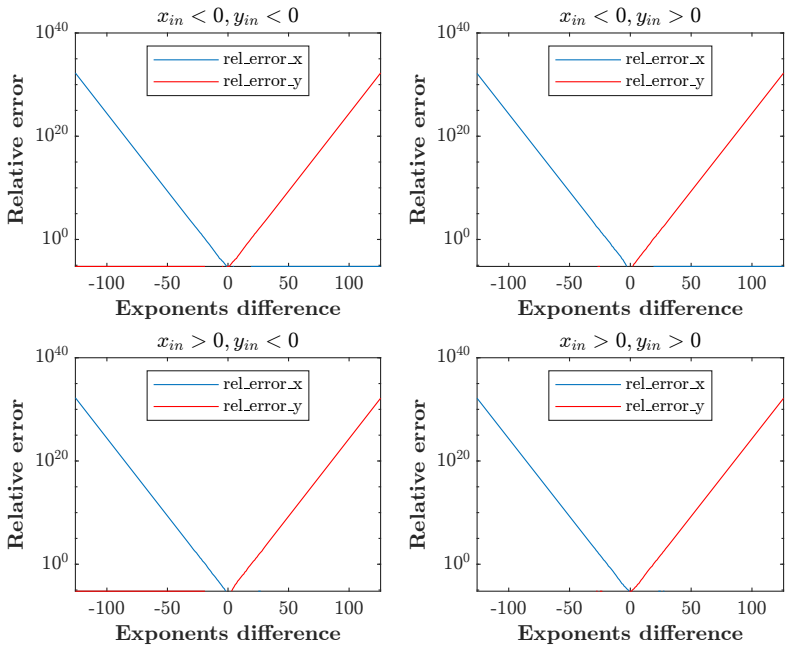


Figure 5.1: Relative error comparison for single precision in rotation and hyperbolic coordinates. The y axis is in logarithmic scale.

Table 5.1 shows the obtained minimum thresholds for both precisions to guarantee

[1] 1.9999999 makes the mantissa equal to 1.111...1, which should emphasize the loss of data during the alignment.

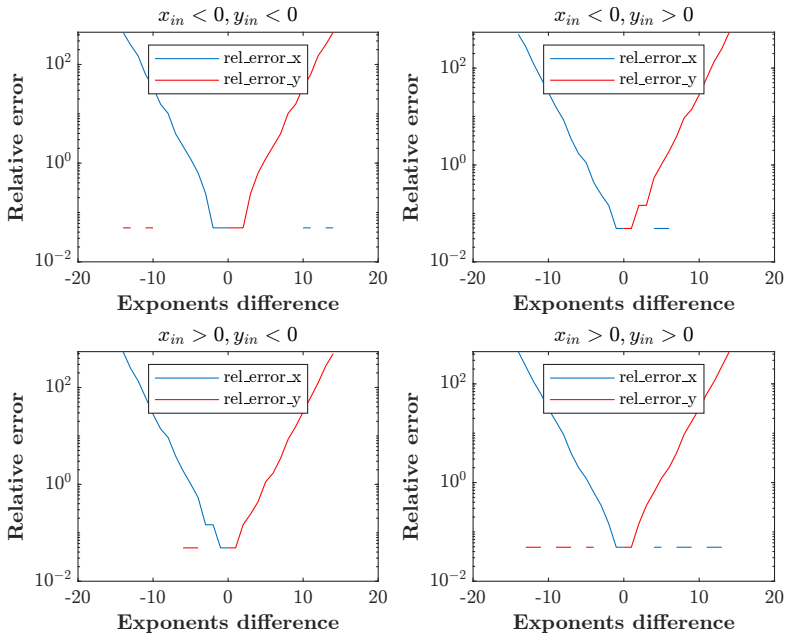


Figure 5.2: Relative error comparison for half precision in rotation and hyperbolic coordinates. The y axis is in logarithmic scale.

a relative error lower than 10% and 20%. As it can be noticed, the obtained values are close to the number of mantissa bits, 23 for single precision and 10 for half precision, which is a non-negligible constraint. However, simulations have shown that values equal to 18 for single precision and 7 for half precision have to be considered to guarantee a very low error.

FLP format	**10% threshold**	**20% threshold**
Single precision	20	21
Half precision	7	8

Table 5.1: Minimum thresholds for the exponent difference.

It has to be noted that this procedure has not to be applied to z, since the restricted amount of overflow bits guarantees an exponents difference with respect to the α angles that is already lower than the considered thresholds.

5.1.2 Random numbers generation

To efficiently test the design, it has been decided to mix fixed and random values. In particular, the fixed inputs have been chosen to cover specific cases, such as the maximum and minimum representable numbers, as well as infinities and NaNs, whereas the random ones have been used to cover the largest possible number of values within the range of convergence. For such purpose, it has been decided to rely on the `randomize()` method provided by SystemVerilog and to fix some constraints through the dedicated class `flp_rand_class`. In particular, these constraints guarantee that:

- Only normalized values are used, thus forcing the inputs exponent to be between e_{min} and e_{max}.
- The exponents difference is at most equal to the threshold.
- The inputs are within the range of convergence, which is dynamically set depending on the considered working mode.

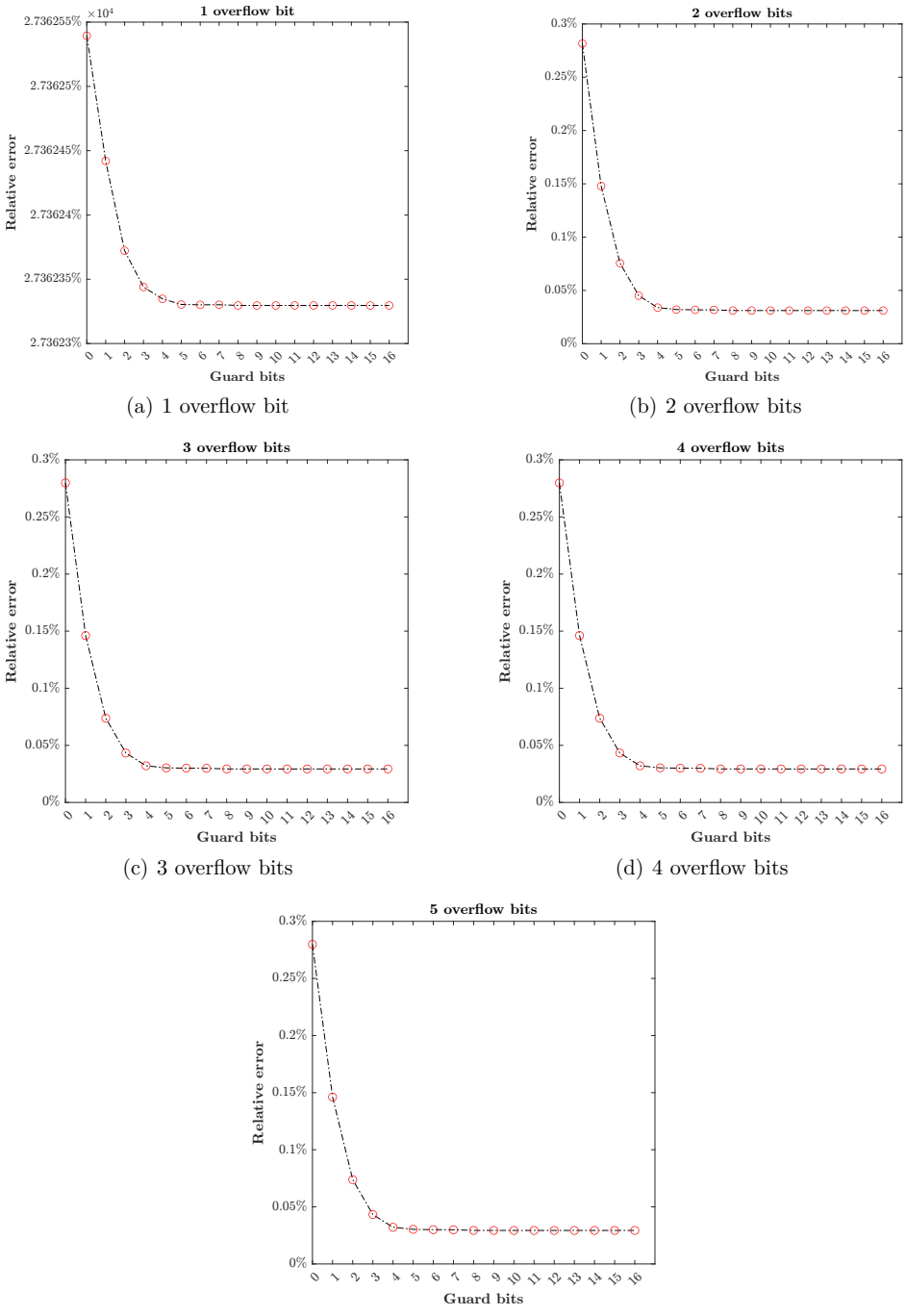
In order to use the same module for both precision, the `flp_rand_class` will, each time, generate random values for both formats.
At this point, the extensive test procedure can start and, each time, the inputs generator performs the following steps:

1. Select the working mode by setting `mode` and `m`. This will tell the `flp_rand_class` which range of convergence has to be used.
2. Launch the `randomize()` method.
3. Select which values have to be forwarded to the CORDIC unit based on the adopted floating-point precision.

5.1.3 Error analysis

Figures 5.3 and 5.4 compare the results obtained from various configurations of overflow and guard bits. In particular, the former range from 1 to 5, while the latter from 0 to 16[1] for single precision and from 0 to 8 for half precision. The error metric is computed as the average among the relative errors obtained for each output variable x, y and z, where the relative error is defined as the ratio between the absolute error and the absolute value of the reference output.

[1]Values of guard bits that are larger than the chosen ones for both precision did not provide noticeable improvements and, hence, have been avoided to make the number of configurations to be tested lower.



(a) 1 overflow bit

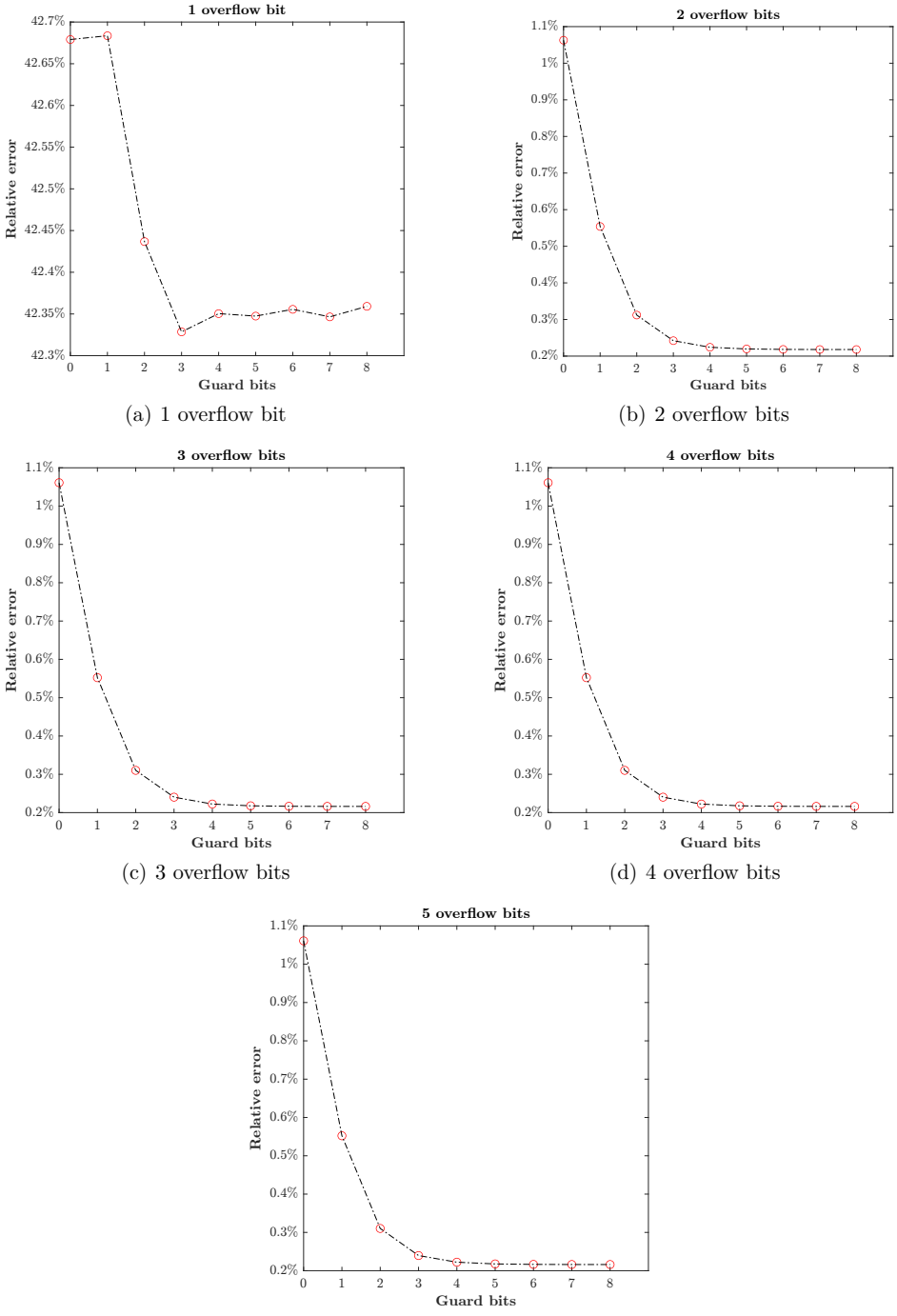
(b) 2 overflow bits

(c) 3 overflow bits

(d) 4 overflow bits

(e) 5 overflow bits

Figure 5.3: Error comparison among different configurations of the internal fixed-point format for single precision.



(a) 1 overflow bit

(b) 2 overflow bits

(c) 3 overflow bits

(d) 4 overflow bits

(e) 5 overflow bits

Figure 5.4: Error comparison among different configurations of the internal fixed-point format for half precision.

From the figures, it can be easily noticed that the graphs of both floating-point formats share the same behavior. The main contribution is given by the number of overflow bits, whereas the guard bits have a minor impact. For both precisions, the error becomes reasonable and almost constant starting from a number of overflow bits equal to 2. This is due to the fact that the lower their number, the higher the probability to have overflow, and thus wrong outputs, throughout the pipeline. Therefore, in order to be on the safe side, the best choice is to set the number of overflow bits to 3 for both floating-point formats. For what concerns, instead, the guard bits, in the graphs related to a number of overflow bits equal to 3, the error becomes almost constant as soon as their number approaches 4 for half precision and 5 for single precision, which validates the choice of such values in Section 3.3.1. Table 5.2 compares the obtained average error for both precisions with the lowest reached one, that is with maximum number of both overflow and guard bits. As it can be noticed, not only the adopted format provided positive results, but also highlighted the fact that this design is suitable for high-accuracy applications, given that the obtained error is spread over a range of 10000 random values.

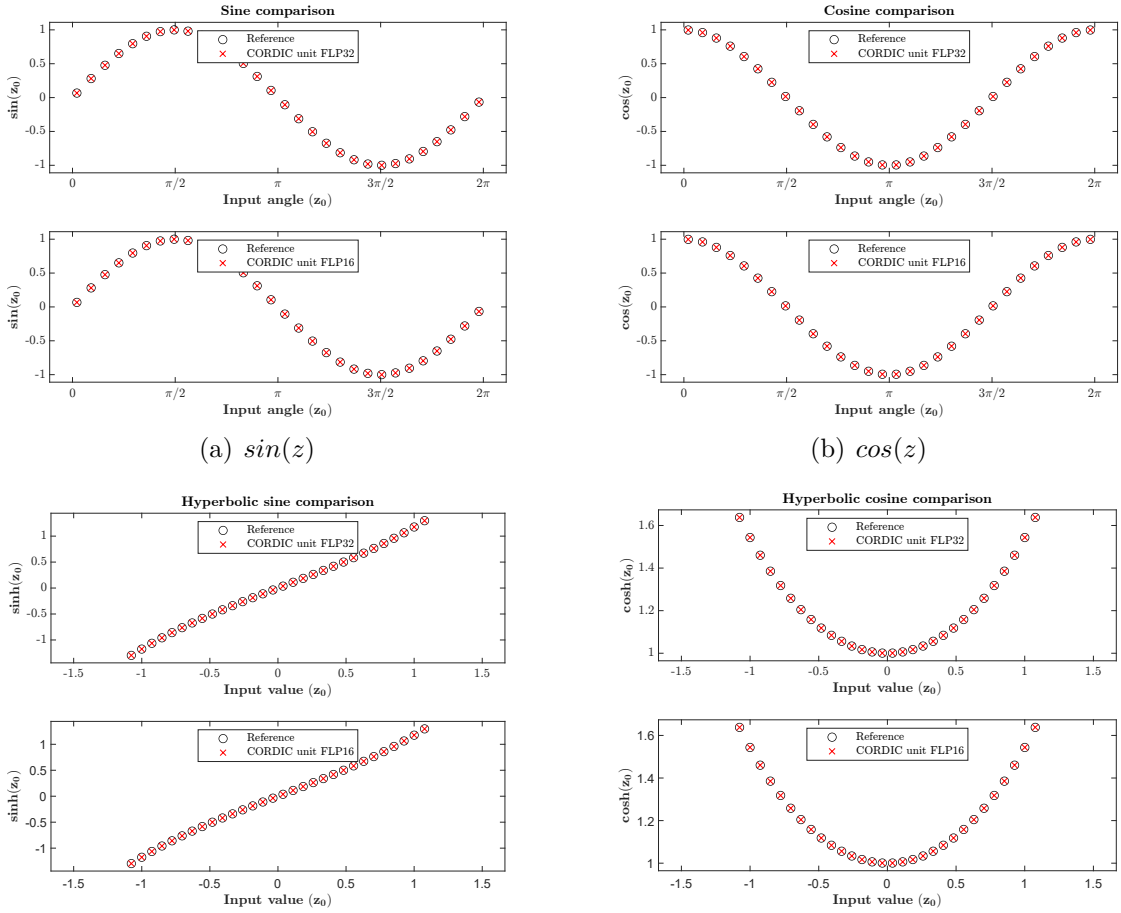
FLP format	**Obtained error**	**Best error**
Single precision	0.030364%	0.029330%
Half precision	0.223205%	0.216155%

Table 5.2: Average relative error comparison.

A proof of this can be seen in Figures 5.5 and 5.6, which show the obtained waveforms considering the reference values and the outputs of the CORDIC unit for both precisions.

5.2 Synthesis results

Once the unit has been tested, simulation left the place to synthesis. The main goal was to satisfy a target frequency of 100 MHz, which is a reasonable value since, given that two useful outputs are available at each clock cycle, the final theoretical throughput would be 200 Msamples/s. However, it has also been tried to find the maximum achievable frequency, which went above 1 GHz for both precisions. The logical synthesis procedure has been performed using Synopsys Design Compiler and issuing the `compile_ultra` command, which forces the tool

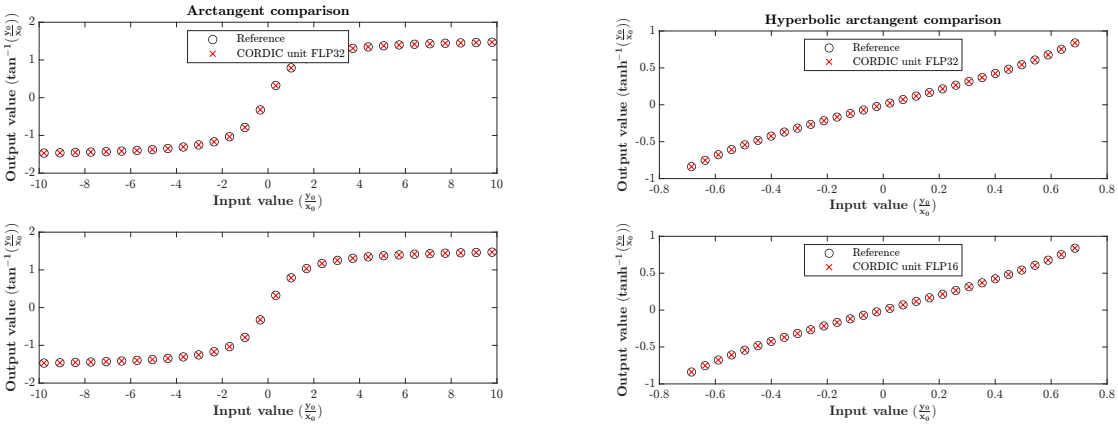


(a) $sin(z)$ (b) $cos(z)$

(c) $sinh(z)$ (d) $cosh(z)$

Figure 5.5: Example waveforms of the reference values (black) and the outputs of the CORDIC unit (red) for both precisions and rotation mode.

to perform aggressive optimizations and gives overall better results compared to the basic `compile` command. The obtained results are shown in Table 5.3.
As it can be noticed, at the target frequency the synthesis procedure provided promising results in terms of area occupation and power consumption. In particular, it is curious to notice that the design for half precision, which, as Table 5.2 shows, is already quite accurate, provided area and power results that are almost one third of the ones obtained for single precision. This is a clear proof of the fact that this floating-point format can be a very good choice for low-power and low-area applications when the values range does not need to be too large. On the other hand, the maximum achievable frequency is quite high for both precisions, which



(a) $tan^{-1}(\frac{y}{x})$ (b) $tanh^{-1}(\frac{y}{x})$

Figure 5.6: Example waveforms of the reference values (black) and the outputs of the CORDIC unit (red) for both precisions and vectoring mode.

FLP format	**Period** [ns]	**Frequency** [MHz]	**Area** [μm^2]	**Power** [mW]
Single precision	10	100	109485.36	11.12
Single precision	0.85	1176.47 (max)	185828.76	63.24
Half precision	10	100	38977.56	3.90
Half precision	0.75	1333.33 (max)	62600.40	27.44

Table 5.3: Synthesis results for both precisions.

is a good result since it means that the design is quite fast and can be used in high-speed applications. However, it has to be noted that area occupation and power consumption are quite high at such frequency, which is a common trade-off in digital design.
In addition to the chosen internal fixed-point format, it has been decided to also compare the values in Table 5.3 and 5.2 with the ones obtained by synthesizing

the design using a minimum [1] and maximum number of overflow and guard bits. The results are shown in Table 5.4 and in Figure 5.7.

FLP format	Overflow bits	Guard bits	Area [μm^2]	Power [mW]	Relative error
Single precision	3	5	109485.36	11.12	0.031%
Single precision	2	0	94343.04	9.66	0.281%
Single precision	5	16	144047.88	14.49	0.029%
Half precision	3	4	38977.56	3.90	0.223%
Half precision	2	0	30121.92	3.04	1.062%
Half precision	5	8	52339.68	5.10	0.216%

Table 5.4: Synthesis results for both precisions with different configurations of overflow and guard bits. The frequency is 100 MHz.

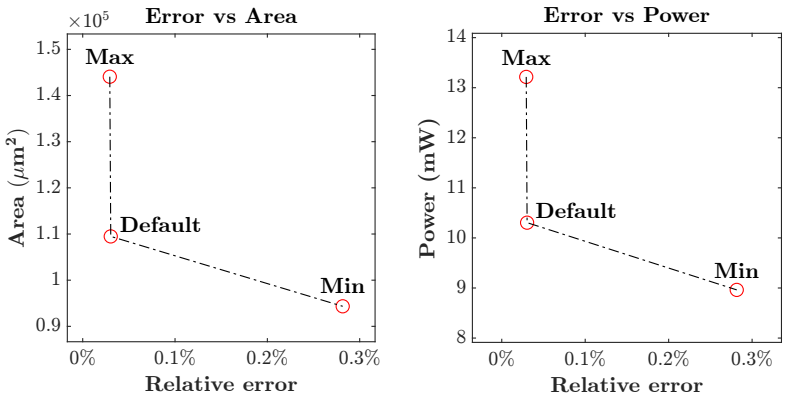
As expected, the chosen format is the best compromise between area occupation, power consumption and accuracy. In fact, the results in Table 5.4 show that the design with the minimum number of overflow and guard bits provides the best area and power results, but the worst accuracy. On the other hand, the largest format gives the best accuracy, which is, however, very close to the one obtained with the chosen format, while the area and power results are the worst.

5.3 X-HEEP results

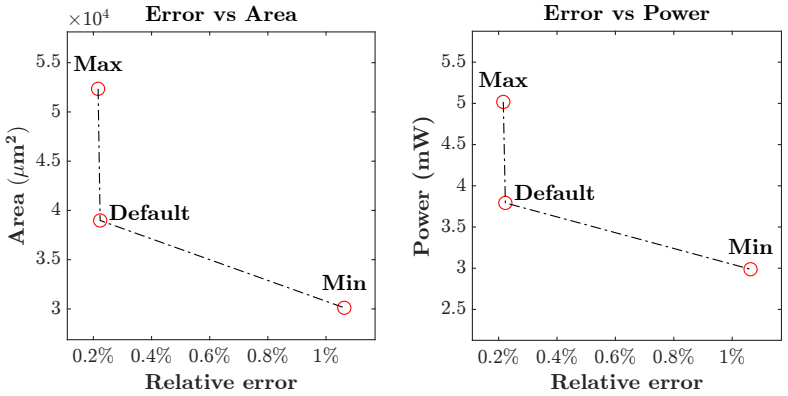
As anticipated, the X-HEEP platform has been used to obtain feasible results in terms of speed improvements on a typical application. In particular, the design has been integrated as a custom accelerator and tested through a simple C code that performs the following operations:

1. Set the CORDIC accelerator control registers, such as the number of inputs to be processed, the starting address of the input and output buffers and the opcode.

[1] A minimum number of two overflow bits has been considered, in order to have error values that can be compared.



(a) Single precision



(b) Half precision

Figure 5.7: Comparison of the area and power results for both precisions with different configurations of overflow and guard bits.

2. Start the accelerator.
3. Wait for the accelerator to finish, which will be signaled by an interrupt.
4. Compare the obtained results with the reference ones obtained by the standalone CORDIC model.

Once the test for each opcode ensured a correct behavior, a performance comparison between the integrated CORDIC accelerator and the corresponding operations performed by the CPU with the `math.h` library has been carried out. However, given the absence of a native support for half precision floating-point arithmetic in the CPU, the comparison has been performed only for single precision. This

has been done by opportunely initializing the performance counter and measuring the time taken by the accelerator and the CPU to perform the same operations. The obtained results for 1024 input combinations (so 3072 values to be read from memory and written to it) and single precision are shown in Table 5.5. It has to be underlined that, in order to use the same inputs for both the accelerator and the CPU, they have to be converted from hexadecimal integers to floating-point numbers. After several trials, it has been noticed that performing this step before actually starting the CPU computation produced wrong results in terms of number of cycles, whereas carrying it out just before issuing each operation provided results that seem to be correct. Therefore, the number of cycles that are needed to perform such conversion has been subtracted from the obtained values.

Configuration	**Accelerator time**	**CPU time**	**Time ratio**
SIN_COS	7.292	6.475.034	887.96
ROT_CIRC	7.292	7.703.830	1056.48
SINH_COSH	7.950	11.268.731	1417.45
ROT_HYP	7.951	23.432.989	2947.18
ROT_HYP_M	7.951	23.444.066	2948.57
ROT_LIN	7.926	219.902	27.74
ROT_LIN_M	7.927	228.625	28.84
VEC_CIRC	7.967	6.947.438	872.03
VEC_CIRC_M	7.967	6.954.583	872.92
VEC_HYP	7.952	7.851.883	987.41
VEC_HYP_M	7.952	7.856.970	988.05
VEC_LIN	7.946	292.603	36.82

Continued on next page

Configuration	Accelerator time	CPU time	Time ratio
VEC_LIN_M	7.946	292.603	36.82
EXPONENTIAL	7.950	7.236.055	910.19
ARCCOTANH	7.951	2.269.557	285.45
ARCCOTAN	7.173	3.620.719	504.77

Table 5.5: Performance comparison between the integrated CORDIC accelerator and the CPU. The latency values are expressed in number of cycles and the time ratio is the ratio between the CPU time and the accelerator time.

As it can be noticed, on average, compared to the CPU the accelerator is:

- 838.83 times faster for trigonometric functions,
- 1497.76 times faster for hyperbolic functions,
- 32.56 times faster for multiplications and divisions.

In addition, thanks to the fact that each computation takes the same time, the number of cycles needed by the CORDIC unit remains almost constant for each function, which is a remarkable result given that latency will be always constant and predictable, no matter what type of operation is being performed. The average number of needed cycles is 7.6 for each input set, which leads to a throughput of 26.3 Msamples/s. Actually, this number is strongly affected by the presence of a single-port memory, which forces read and write accesses to be interleaved, and of a 32-bit data bus, which makes the number of cycles needed to read and write a single set of inputs and outputs equal to 4. As a consequence, the throughput is far from the theoretical value of 200 Msamples/s, which can be obtained by adopting a dual-port memory, a larger data bus or other arrangements (e.g. input and output FIFOs).
For what concerns half precision, instead, the functional correctness test part provided an overall number of cycles equal to 5455, which means that, on average, 5.3 cycles are needed for each 32-bit input set. Even if the results are quite

close to the ones obtained for single precision, it has to be taken into account both that the unit is not completely exploiting pipeline parallelism due to the aforementioned memory limitations and that the throughput is now doubled because of the presence of two units working in parallel and producing four useful outputs at a time. Therefore, the throughput becomes now equal to 75.5 Msamples/s, which is a good result given the fact that the design is quite small and low-power, but is far from the theoretical value of 400 Msamples/s.

Chapter 6

Conclusions

In this work, the CORDIC algorithm has been first described and then implemented in a hardware accelerator able both to be used in all its six working modes and to support the half and single precision FLP formats. In particular, two strategies have been at first analyzed to adapt its fixed-point nature to the floating-point arithmetic, that is global and local FLP, and their overall accuracy has been compared. However, being high-throughput the main goal of the project, local FLP has been chosen for its capability to reach higher frequencies without the need for fine-grain pipelining. In order to minimize latency, an unfolded and pipelined architecture has been designed and the final implementation has been synthesized at the target frequency of 100 MHz, providing promising results in terms of accuracy, area occupation and power consumption. In addition, it has also been tried to find the maximum achievable frequency, which turned out to be above 1 GHz for both floating-point formats. Finally, the unit has been integrated in the X-HEEP microcontroller system and tested in a real application, showing that not only it can be used as an external accelerator capable of providing a significant speedup, especially in the computation of trigonometric and hyperbolic functions, but also that it offers constant latency independently from the operation to be computed.

6.1 Future works

Even if this work provided remarkable results and can be considered complete, there are still some aspects that could be improved or further investigated. First of all, in addition to the integrated register disabling mechanism that prevents the unit from increasing the switching activity when it is not in use or the final result can be predicted, other low-power techniques, such as clock gating, could be implemented to further reduce the power consumption. Furthermore, since the

CORDIC unit is fully parametrized, other standard or custom floating-point formats can be supported with very little effort. The unit can also be integrated as an internal peripheral in a microcontroller system, together with specific arrangements to improve data transfers that could significantly increase the throughput. For example, the unit could be connected either to a dedicated dual-port memory through a larger bus or to input and output FIFOs, so that all the inputs are available at once and the outputs can be stored in one clock cycle, thus achieving the maximum throughput. Finally, the unit could be used in a more complex platform, such as a digital signal processor, a software-defined radio or a machine learning accelerator, to improve the overall performance of the system, or the support for more operations that can be obtained combining the ones already supported could be added.

Bibliography

[1] Chih-Hsiu Lin and An-Yeu Wu. «Mixed-scaling-rotation CORDIC (MSR-CORDIC) algorithm and architecture for high-performance vector rotational DSP applications». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 52.11 (2005), pp. 2385–2396 (cit. on p. 1).

[2] J. Valls, T. Sansaloni, A. Perez-Pascual, V. Torres, and V. Almenar. «The use of CORDIC in software defined radios: a tutorial». In: *IEEE Communications Magazine* 44.9 (2006), pp. 46–50 (cit. on p. 1).

[3] N.D. Hemkumar and J.R. Cavallaro. «Efficient complex matrix transformations with CORDIC». In: *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. 1993, pp. 122–129 (cit. on p. 1).

[4] Vipin Tiwari and Ashish Mishra. «Neural network-based hardware classifier using CORDIC algorithm». In: *Modern Physics Letters B* 34.15 (2020), pp. 2050161-1–2050161-11 (cit. on p. 1).

[5] Shoaib Bhuria and P. Muralidhar. «FPGA implementation of sine and cosine value generators using Cordic Algorithm for Satellite Attitude Determination and calculators». In: *2010 International Conference on Power, Control and Embedded Systems*. 2010, pp. 1–5 (cit. on p. 1).

[6] Debaprasad De, Archisman Ghosh, K Gaurav Kumar, Anurup Saha, and Mrinal Kanti Naskar. «Multiplier-less Hardware Realization of Trigonometric Functions for High Speed Applications». In: *2018 IEEE Applied Signal Processing Conference (ASPCON)*. 2018, pp. 149–152 (cit. on p. 1).

[7] Raimund Kirner, Markus Grössing, and Peter Puschner. «Comparing WCET and Resource Demands of Trigonometric Functions Implemented as Iterative Calculations vs. Table-Lookup». In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Open Access Series in Informatics (OASIcs). 2006, pp. 1–6 (cit. on p. 1).

[8] B. Lakshmi and A. S. Dhar. «CORDIC Architectures: A Survey». In: *VLSI Design* 2010 (2010) (cit. on pp. 2, 10, 12, 16, 35, 43, 44).

[9] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. *X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators*. 2024. arXiv: `2401.05548 [cs.AR]` (cit. on pp. 2, 85).

[10] «IEEE Standard for Floating-Point Arithmetic». In: *IEEE Std 754-2008* (2008), pp. 1–70 (cit. on pp. 4–6, 9).

[11] J. E. Volder. «The CORDIC trigonometric computing technique». In: *IRE Transactions on Electronic Computers* 8.3 (1959), pp. 330–334 (cit. on p. 10).

[12] J. S.Walther. «A Unified Algorithm for Elementary Functions». In: *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*. AFIPS '71 (Spring). 1971, pp. 379–385 (cit. on pp. 13, 16, 42, 53, 68).

[13] J. Mack, S. Bellestri, and D. Llamocca. «Floating Point CORDIC-based Architecture For Powering Computation». In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2015, pp. 1–6 (cit. on p. 14).

[14] X. Hu, R. G. Harber, and S. C. Bass. «Expanding the range of convergence of the CORDIC algorithm». In: *IEEE Transactions on Computers* 40.1 (1991), pp. 13–21 (cit. on p. 17).

[15] A.A.J. de Lang and E.F. Deprettere. «Design and implementation of a floating-point quasi-systolic general purpose CORDIC rotator for high-rate parallel data and signal processing». In: *Proceedings 10th IEEE Symposium on Computer Arithmetic*. 1991, pp. 272–281 (cit. on p. 20).

[16] E. Antelo, J. Villalba, and E. L. Zapata. «A low-latency pipelined 2D and 3D CORDIC processors». In: *IEEE Transactions on Computers* 57.3 (2008), pp. 404–407 (cit. on pp. 20, 28, 33, 34, 36, 37, 40).

[17] J. D. Bruguera, E. Antelo, and E. L. Zapata. «Unified mixed Radix 2–4 redundant CORDIC processor». In: *IEEE Transactions on Computers* 45.9 (1996), pp. 1068–1073 (cit. on pp. 23, 27, 33–36, 40).

[18] W. J. Peréz-Holguín and R. A. Limas-Sierra. «Pipeline Implementation of the Unified CORDIC Algorithm in FPGA». In: *Revista Facultad de Ingeniería Universidad de Antioquia* 107 (2022), pp. 66–79 (cit. on pp. 23, 33, 34, 38, 40).

[19] N. Takagi, T. Asada, and S. Yajima. «Redundant CORDIC methods with a constant scale factor for sine and cosine computation». In: *IEEE Transactions on Computers* 40.9 (1991), pp. 989–995 (cit. on pp. 23–25, 27, 33–35, 40).

[20] D. Timmermann, H. Hahn, and B. J. Hosticka. «Low latency time CORDIC algorithms». In: *IEEE Transactions on Computers* 41.8 (1992), pp. 1010–1015 (cit. on pp. 24, 25, 27, 28, 33–35, 40).

[21] J. Duprat and J.M. Muller. «The CORDIC algorithm: new results for fast VLSI implementation». In: *IEEE Transactions on Computers* 42.2 (1993), pp. 168–178 (cit. on pp. 25, 33–35, 40).

[22] H. Dawid and H. Meyr. «The differential CORDIC algorithm: constant scale factor redundant implementation without correcting iterations». In: *IEEE Transactions on Computers* 45.3 (1996), pp. 307–318 (cit. on pp. 25, 33–35, 40).

[23] J. D. Bruguera, E. Antelo, and E. L. Zapata. «Design of a pipelined Radix 4 CORDIC processor». In: *Parallel Computing* 19.7 (1993), pp. 729–744 (cit. on pp. 26, 33–35, 40).

[24] E. Antelo, J. Villalba, J. D. Bruguera, and E. L. Zapata. «High performance rotation architectures based on the Radix-4 CORDIC algorithm». In: *IEEE Transactions on Computers* 46.8 (1997), pp. 855–870 (cit. on pp. 27, 33–38, 40).

[25] M. Kuhlmann and K. K. Parhi. «P-CORDIC: a precomputation based rotation CORDIC algorithm». In: *EURASIP Journal on Applied Signal Processing* 2002.9 (2002), pp. 936–943 (cit. on pp. 28, 33, 34, 40).

[26] B. Gisuthan and T. Srikanthan. «Pipelining flat CORDIC based trigonometric function generators». In: *Microelectronics Journal* 33.1–2 (2002), pp. 77–89 (cit. on pp. 28, 33–35, 40).

[27] T.-B. Juang, S.-F. Hsiao, and M.-Y. Tsai. «Para-CORDIC: parallel CORDIC rotation algorithm». In: *IEEE Transactions on Computers* 51.8 (2004), pp. 1515–1524 (cit. on pp. 29, 32–34, 36, 40).

[28] H. S. Kebbati, J. Ph. Blonde, and F. Braun. «A new semi-flat architecture for high speed and reduced area CORDIC chip». In: *Microelectronics Journal* 37.2 (2006), pp. 181–187 (cit. on pp. 30, 33–35, 40).

[29] D. Timmermann, B. Rix, H. Hahn, and B. J. Hosticka. «A CMOS Floating-Point Vector-Arithmetic Unit». In: *IEEE Journal of Solid-State Circuits* 29.5 (1994), pp. 634–639 (cit. on pp. 30, 33, 34, 39, 40, 42, 45, 47, 49, 53, 54, 68, 71, 72, 74, 79, 95).

[30] L. Fang, B. Li, Y. Xie, H. Chen, and L. Pang. «A Unified Reconfigurable CORDIC Processor for Floating-Point Arithmetic». In: *International Journal of Electronics* 107 (2018), pp. 1436–1450 (cit. on pp. 31, 33, 34, 37, 39, 40).

[31] O.A. Pfander ad R. Nopper and al. «Comparison of reconfigurable structures for flexible word-length multiplication». In: *Advances in Radio Science* 6 (2008), pp. 113–118 (cit. on p. 37).

[32] A. Cristiano I. Malossi et al. «The transprecision computing paradigm: Concept, design, and applications». In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 1105–1110 (cit. on p. 87).

[33] OpenHW Group. *Register Interface*. `https://github.com/pulp-platform/register_interface` (cit. on pp. 88, 91).

[34] OpenHW Group. *Open Bus Interface*. `https://github.com/pulp-platform/obi` (cit. on p. 89).