STUDIES ON FLAT CORDIC IMPLEMENTATION IN FIELD PROGRAMMABLE GATE ARRAYS (FPGA)

MEERA SUBRAMANIAM

FACULTY OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY

UNIVERSITY OF MALAYA

KUALA LUMPUR

JANUARY 2004

STUDIES ON FLAT CORDIC IMPLEMENTATION

IN FIELD PROGRAMMABLE GATE ARRAYS (FPGA)

MEERA SUBRAMANIAM

THESIS SUBMITTED IN FULFILMENT

OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF COMPUTER SCIENCE

FACULTY OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY

UNIVERSITY OF MALAYA

KUALA LUMPUR

JANUARY 2004

ABSTRACT

The CORDIC algorithm has been widely researched as an efficient hardware algorithm for the computation of trigonometric, hyperbolic and transcendental functions. It is an iterative process of rotations that are carried out through simple shift and addition operations. These operations map well onto hardware, and CORDIC is used in a multitude of signal processing applications. The iterative nature of CORDIC is a drawback, and a technique known as Flat CORDIC was introduced to eliminate the iterations, making the design largely combinatorial. The latest advances in VLSI technology have made it possible to produce series of FPGAs that have large numbers of gates at relatively low costs. This work focuses on FPGA implementation of the Flat CORDIC scheme to efficiently compute trigonometric functions. The successive replacement of the basic CORDIC equations to generate the parallelized Flat CORDIC ones requires that the direction of all the rotations be pre-computed. This work presents a modification to the previous Signed Digit (SD) Generation algorithm and a comparison with the previous method. The second major component is the mapping of the Flat CORDIC equations using these SDs. An architecture is proposed for effective combination of these SDs for sine/cosine generation. Pipelining methods are investigated to increase design speed. The architectures for 9, 12, 15, 18, 21 and 24 bit Flat CORDIC are simulated using XILINX ISE WebPack 5.2i. The functionally simulated designs are synthesized onto SPARTAN FPGAs. Some relevant comparisons are made with other designs in literature. It is found that if properly pipelined, Flat CORDIC on FPGAs can achieve high speeds of up to 43 MHz for 20 bit accuracy. In terms of area, however, the largely combinatorial design is a drawback for FPGA implementation. In summary, the main contribution of this thesis is a study on the effectiveness of mapping Flat CORDIC onto FPGAs

This Work Is Dedicated
To My Beloved Parents

ACKNOWLEDGEMENTS

I would like, first and foremost, to extend my sincere gratitude to my supervisor, Prof. Dato' Ir. Dr. Mashkuri Haji Yaacob, for his endless support throughout the duration of my project.

For the father of Flat CORDIC, Dr. C. T. Clarke, whose boundless enthusiasm in my work has kept me going, I have the deepest regard. My thanks go to him for all the hours spent explaining things to me, and for all the brilliant ideas.

My sincere thanks to Dr Srikanthan who has been a great source of inspiration to me. The dedication and commitment of all the people in CHiPES at NTU has been a huge motivation for me to do my very best.

To the Head of Department of Electrical Engineering, Assoc. Prof Dr. Nasrudin Abd Rahim, for his support and help in obtaining the tutorship during my postgraduate study and the previous Head of Department Dr. Kaharudin Dimyati, for his support, my sincere thanks. Also to all the staff in the department, it has been a pleasure knowing and working with them all.

My dearest parents have given me so much encouragement and have been by my side, ready to help me in any way they could… I would like to take this opportunity to tell them how much I appreciate their love.

To the rest of my family, and to all my wonderful friends, who have stood by me, and have shared many precious moments, I would like to extend my deepest love.

And to Bhagavan Sri Sathya Sai Baba, who has been a beacon, shining brightly always and showing me the way – my humble *pranams* at Your Lotus Feet.

TABLE OF CONTENTS

PUBLICATIONS

1. **Meera Subramaniam** and Ibrahim A. Abdul Razak : 'FPGA Implementation of CORDIC Algorithm', Proceesings of First Technical Postgraduate Symposium, KL, page(s) 387 – 390, October 2002.

LIST OF TABLES

LIST OF FIGURES

Glossary of Acronyms and Terms

A	ASIC	Application Specific Integrated Circuit
C	C&A	Combination & Addition
	CIV	Cumulative Index Value
F	FPGA	Field Programmable Gate Array
L	LSB	Least Significant Bit
	LSC	Least Significant Channel
	LSSD	Least Significant Signed Digit
M	MSB	Most Significant Bit
	MSC	Most Significant Channel
	MSSD	Most Significant Signed Digit
P	P&R	Place & Route
	PWL	Pair-Wise Linear
R	RM	Rotation Mode
	ROM	Read Only Memory
S	SBNR	Signed Binary Number Representation
	SD	Signed Digit
	SDA	Signed Digit Algorithm
	SRAM	Static Random Access Memory
V	VLSI	Very Large Scale Integration
	VM	Vector Mode

CHAPTER 1 : INTRODUCTION

1.1 Motivation

The rapid technological advances in VLSI have made it possible to market high performance logic devices which can accommodate large amounts of logic at low cost. Field Programmable Gate Arrays (FPGAs) are among the class of programmable logic devices that have benefited tremendously from these advances, allowing them to reach gate counts large enough to allow complex applications to be programmed onto them. They can also operate at increasingly higher frequencies. FPGAs in recent times have come to rival ASICs with their short development time and lower costs. The availability of Synthesis as well as Place and Route (P&R) tools has made it easier to create and test designs. In addition, SRAM FPGAs also have unlimited reprogrammability that allow design upgrades without having to replace the hardware (Skahill, K. 1997).

The CORDIC algorithm is well known as a hardware-efficient algorithm for performing trigonometric, hyperbolic and transcendental functions (Walther, 1971). There is an increasing need for huge amounts of rapid calculations in the fields of signal and image processing and robotics, to name a few, and hardware components to execute these calculations have been taking over their software counterparts in recent times (Wang et al, 1997). Research into CORDIC and its applications have been extensive, and many methods have been proposed and implemented to eliminate or minimize its drawbacks. Among them is the Flat CORDIC algorithm (Clarke, 1995) that aims to completely eliminate the iterative nature of CORDIC through a parallelization of the CORDIC equations, and thereby decrease the latency.

1.2 Research Objectives

- This thesis focuses on the FPGA implementation of Flat CORDIC. A method for generating the iteration directions (Signed Digits, or SDs) is studied. In conventional CORDIC, one SD is generated per iteration. In Flat CORDIC, which is a parallelization of the original CORDIC equations through repeated replacement into successive iterative equations, all the SDs need to be known beforehand, because the overall Flat CORDIC equation is completely expressed in term of all these SDs.
- Gisuthan (2000) designed a Signed Digit Algorithm (SDA) using encoders. This work will attempt a detailed examination of the relationship between the input angle values and their corresponding SDs. All possible input values and their corresponding SDs could be examined to see if there is a pattern between them, and identify this pattern. The SD generation could then be synthesized and simulated using VHDL and then compared against results using conventional programming like C++.
- This work will also focus on implementation of the Flat CORDIC equations. The equations for the cosine and sine of an N-bit input angle are made up of many terms, with the number of terms increasing sharply with N. Each term is made up of two parts – the combinational part, which is the product of different sets of SDs, and the positional value part, which is in the form of 2^{-i}, that, in binary, is simply a right shift *i* times. For each specific positional value, there are a number of combinational sets of values that need to be shifted by that positional value.

- A method could be looked into that combines all these terms in a unique manner that eliminates the need for multiple adders to get the sum of all the combinations. This method would involve the design of specific modules that are then repeatedly generated. In addition, the final summation of all the combinational terms with their shifts could be reduced to one single addition. Here again, the detailed input and output data for cosine and sine generation would be generated in C++.
- The full design with all the modules would subsequently be written in VHDL, then functionally simulated and synthesized. The results of these simulations could be compared against the C++ data files mentioned earlier.
- The proposed architecture is organized in a matrix structure of repeated instances. The design is largely combinatorial, and in certain positions, the data has to run through many layers of logic that cause large delays. Studies could then be performed on the possibilities of pipelining the design to reduce the amount of logic per cycle.
- The improvement in circuit frequency with additional pipeline stages would then be compared with the effect of the additional cycles on the overall latency of the design. The results will show whether an improvement is seen if the circuit is correctly pipelined, and whether there are any limits to the benefit of the additional stages (ie. there is an ideal number of extra pipeline stages, beyond which adding them decreases performance).

1.3 Main Contributions

The primary goals of this thesis are to investigate and design architecture for Flat CORDIC on FPGAs. The thesis provides insight into the internal patterns for SD Generation and combinations. Based on the above goals, the key contributions of this thesis include

- A complete analysis on the relationship between input angles and corresponding SDs
- A newly designed combination scheme for piecing together the Flat CORDIC basic equations in a simple and organized manner
- A detailed study on the design issues of implementing the scheme, and the possibilities for improving the design

1.4 Thesis Outline

This work is organized as follows. In ***Chapter 2***, the CORDIC algorithm is presented, and its operation modes are explained. The various modifications of the algorithm are summarized. Implementations of this algorithm on FPGAs are presented.

In ***Chapter 3***, the Flat CORDIC Algorithm is introduced. The design is subdivided into several parts, and these are outlined. The pair-wise linear patterns found between the input angle bits and the Signed Digits are highlighted.

Chapter 4 is dedicated to a detailed description of the Signed Digit Generation methods and algorithm. The method for utilizing the unique pattern is examined, and a comparison is made with the previous generation method.

The combination and channeling of the SDs is shown in ***Chapter 5***. A method for rippling the channel results using combinatorial logic, to avoid the use of many adders is presented. A method to pipeline the design for higher throughput is discussed. Also included is a general description of FPGAs, and specifically the one used in this work.

Chapter 6 presents the synthesis results of Flat CORDIC and its individual components. A comparison is made with other implementation methods.

Chapter 7 concludes the thesis with a summary of the results. Suggestions for future work in this area are also presented.

CHAPTER 2 : CORDIC

2.1 The CORDIC Algorithm

The CORDIC (**CO**ordinate **R**otation **DI**gital **C**omputer) Algorithm is widely used as a powerful and flexible generic architecture to implement many algorithms that involve non-trivial arithmetic functions. It is a versatile algorithm for computing trigonometric, hyperbolic and transcendental functions by performing a rotation of a 2-dimensional vector in linear, circular and hyperbolic coordinate systems. This rotation is the result of an iterative series of simple shift and addition operations, which are easy to incorporate in VLSI technology.

The original trigonometric CORDIC algorithm was developed by Volder (1959) as a digital solution for real-time navigation problems. This version was used to calculate trigonometric functions, multiplication, division and datatype conversion functions, and plane rotations. Extensions to CORDIC theory by Walther (1971) enabled calculation of hyperbolic functions, the results of which could be exploited to generate other transcendental functions.

CORDIC currently forms the integral macro in computer arithmetic. Applications of modern digital signal and image processing, which involve massive computations, exhibit an increasing need for the efficient implementation of complex arithmetic operations, and widely use CORDIC. Other compute-intensive applications include matrix computations, which involve calculation of angles and their use in rotation, for example matrix triangularization, and singular value decomposition

(Symansky *et al.*, 1987). These have benefited tremendously in speed through the incorporation of CORDIC. It is also used widely in the fields of dynamic system modeling, control, robotics, computer graphics, filtering and virtual reality (Wang *et al.*, 1997).

2.2 The CORDIC Equations

The CORDIC Algorithm performs the rotation of a vector (X_i, Y_i) with magnitude M and phase P by means of a sequence of micro-rotations, each one over a fixed elementary angle ϕ_i, as shown below:

$$X_{i+1} = X_i \cos\phi_i - Y_i \sin\phi_i \tag{2.1}$$

$$Y_{i+1} = Y_i \cos\phi_i + X_i \sin\phi_i \tag{2.2}$$

For the i^{th} iteration, as indicated above, the vector prior to rotation is (X_i, Y_i) and the rotated intermediate vector is (X_{i+1}, Y_{i+1})

With the rotation angle restricted such that $\phi_i = \pm \tan^{-1} 2^{-i}$ (the $\pm$ sign indicates that the rotation direction is variable), the micro-rotation can be reduced to simple shift-and-add operations :

$$X_{i+1} = K_i\left(X_i - s_i Y_i 2^{-i}\right) \qquad \text{and} \qquad Y_{i+1} = K_i\left(Y_i + s_i X_i 2^{-i}\right)$$

The direction of the micro-rotation is indicated by s_i (+1 for anticlockwise rotation, and –1 for clockwise rotation). It is to be noted here that each successive rotation is not a pure rotation, but a rotation-extension, where the length of the resultant vector is modified by a factor of $K_i = \dfrac{1}{|\cos\phi_i|}$

In their generalized form, the CORDIC equations are given as :

$$X_{i+1} = X_i - m s_i Y_i 2^{-i} \tag{2.3}$$

$$Y_{i+1} = Y_i + s_i X_i 2^{-i} \tag{2.4}$$

$$Z_{i+1} = Z_i - \frac{1}{\sqrt{m}} s_i \tan^{-1}\left(m\sqrt{m}2^{-i}\right) \tag{2.5}$$

With the selection of an appropriate value for the parameter m, different coordinate systems can be achieved (+1 for circular, -1 for hyperbolic). Z_i is the overall rotation angle accumulator.

The final output vector after N iterations is (X_N, Y_N), and is scaled by a constant scale factor K, which is a combination of the scaling from all the iterations, and is given by $K = \prod_{i=1}^{N} \frac{1}{|\cos\phi_i|}$. A multiplication by 1/K is introduced at the end to correct (X_N, Y_N). To satisfy an N-bit precision CORDIC operation, N iterations are required. In addition, the length of the datapath to compute the X and Y variables has to be ($N + \log_2 N + 2$) bits (Gisuthan *et al.*, 2000).

2.3 CORDIC Operation Modes

There are two basic modes of operation for CORDIC; the Rotation Mode (RM) and the Vector Mode (VM). The rotation mode rotates the input vector by a specified input angle (or argument), and the vector mode rotates the input vector towards the X-axis, while recording the total angle movement. Implementation of these methods is characterized by suitable control of the direction of successive micro-rotations, which force either the Y- or the Z- components to 0.

2.3.1 Rotation Mode CORDIC

The rotation mode is usually used to compute trigonometric functions since/cosine, hyperbolic functions sinh/cosh, and then extended to transform polar coordinates to Cartesian coordinates. The angle accumulator Z is initialized with the desired rotation angle θ. In sine/cosine computations:

$$X_{i+1} = X_i - s_i Y_i 2^{-i} \tag{2.6}$$

$$Y_{i+1} = Y_i + s_i X_i 2^{-i} \tag{2.7}$$

$$Z_{i+1} = Z_i - s_i \tan^{-1}\left(2^{-i}\right) \tag{2.8}$$

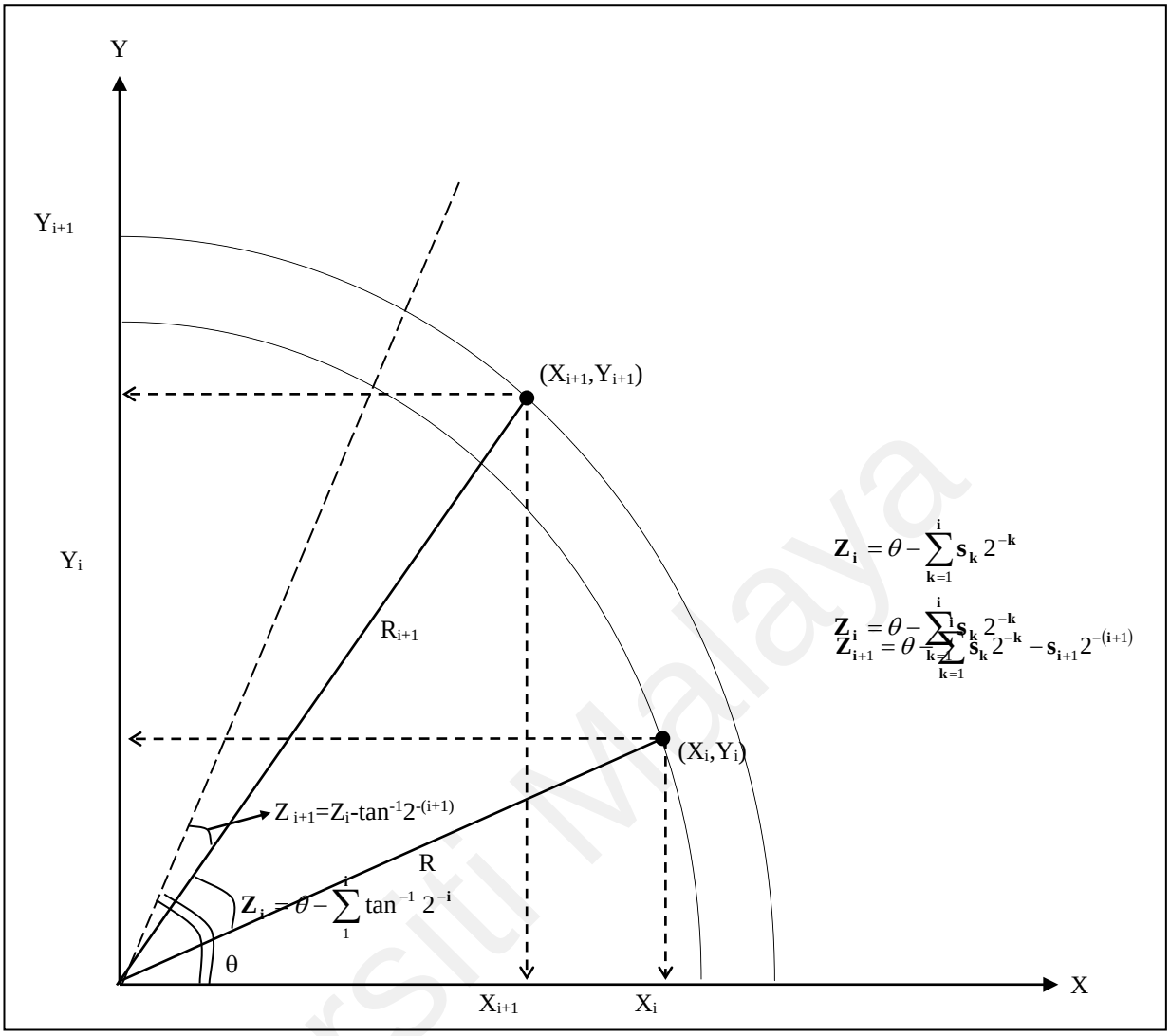

Fig. 2.1 CORDIC Rotation

The initial vector (X_o, Y_o) is aligned along the X-axis, with magnitude 1, ie (1,0)

At the end of the N iterations, the final vector (X_N, Y_N) = ($K_N$cos θ, $K_N$sin θ)

2.3.2 Vector Mode CORDIC

This mode is usually used to calculate magnitudes and angles of given input vectors, and also arctangent values. For a given input vector, Z is initialized to 0, and the vector is rotated until it is aligned along the X-axis. The angle accumulator will now show the total traversed angle, (Z_N), and the magnitude of the original vector is the (X_N)

component. This magnitude, as mentioned earlier, is scaled. For the Vector Mode, the characteristic equations are :

$$X_{i+1} = X_i + sign(Y_i)2^{-i}Y_i \tag{2.9}$$

$$Y_{i+1} = Y_i - sign(Y_i)2^{-i}X_i \tag{2.10}$$

$$Z_{i+1} = Z_i + sign(Y_i)\tan^{-1}(2^{-i}) \tag{2.11}$$

Despite its architectural and algorithmic simplicity, the CORDIC algorithm has several drawbacks:

1) Each successive iteration can only be performed after the previous one, since the sign bit that determines the rotation direction is produced by the previous iteration
2) It is slow because the recurrences involve carry-propagation addition and variable shifting
3) It is area-consuming because of the use of variable shifters and the ROM storing the arctangent values
4) The area complexity and circuit latency are roughly proportional to the accuracy of the desired output. Therefore, the speed of execution becomes restricted with the size

2.4 Literature Review : Modifications of CORDIC

In general, modifications to the CORDIC algorithm aim to either increase the speed of the iterations, or to reduce the total number of iterations. Among the methods used to increase the speed of the iterations are the redundant (or on-line) number representations and redundant adders, which can perform Most Significant Digit (MSD)- first additions. For reduction of the number of iterations, higher radix techniques are used.

2.4.1 Redundant CORDIC

In redundant CORDIC, the X, Y, and Z values are coded using redundant number representation. Introducing this into the iterative computation eliminates the carry-propagate from the addition/subtraction operations, thereby allowing them to be carried out MSD first. The signed digit in this case is selected from the range {-1, 0, 1}, as is explained: In the redundant form of representation, the Most Significant Digit, MSD of the value does not necessarily contain the sign of the value. The signed digit (s_i) has to be estimated from the inspection of a few of the MSDs. However, when all the inspected digits are 0, the proper value of s_i cannot be determined without knowledge of the remaining digits, and here it would seem that the best strategy would be to assign the value 0 to s_i. This move, however, freezes the iteration. Since the final scaling factor, K_N depends on the actually performed iterations, freezing the iteration causes the final scaling factor to become variable.

Ercegovac and Lang (1988, 1990) developed special purpose CORDIC modules which take an input vector (a, b) as an input, and first determine the angle (vectoring

mode), $\tan^{-1}\left(\frac{b}{a}\right)$ in decomposed form (in the form of the signed bits), and then proceed to calculate the sine and cosine of the input angle (rotation mode, circular coordinates). The highlights of their work are :

1) The conventional CORDIC module is modified so that on-line addition is used. Area-consuming shifters are replaced by area-efficient delays
2) An implementation of the computation of the variable scale factor is developed
3) After the angle θ is obtained, it is transmitted in decomposed form to be used in the rotation mode

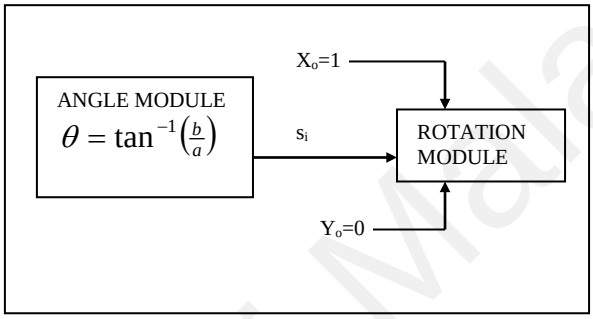

Fig. 2.2 Decomposition of Angle

Part 1 :

The two modifications that improve the CORDIC implementation are :

1) Elimination of the shifter :

   We let $W[j] = 2^j Y_a[j]$ (2.12)

   So, from (1) to (3), we now have

$$X_a[j+1] = X_a[j] + s_j 2^{-2j} W[j] \quad (2.13)$$

$$W[j+1] = 2\left(W[j] - s_j X_a[j]\right) \quad (2.14)$$

$$Z_a[j+1] = Z_a[j] + s_j \tan^{-1}\left(2^{-j}\right) \quad (2.15)$$

With $s_i = \begin{cases} +1 & if\ W[j] \geq 0 \\ -1 & if\ W[j] < 0 \end{cases}$ (2.16)

From this, we see that one shifter has been eliminated, and also, iterations when $j > \frac{N}{2}$ do not affect X[n] for N bit precision

2) Replacing carry propagation by redundant addition :

Here, s_i is allowed to take values from the set {-1, 0, 1} instead of {-1, 1} as follows

With $s_i = \begin{cases} +1 & if\ W[j] \geq \frac{1}{2} \\ 0 & if\ W[j] = 0 \\ -1 & if\ W[j] \leq -\frac{1}{2} \end{cases}$ (2.17)

Where $\hat{W}[j]$ is an estimation of *W[j]* with a precision of 1 fractional bit (usually the 3 MSB of W*[j]*)

Part 2

Following this, to calculate, for example, the sine/cosine values of the original vector (X_o, Y_o), the angle θ, which was produced in decomposed form [from Part 1] is now rotated using the decomposed bits as the signed bits as follows :

$$X_{i+1} = X_i + s_i 2^{-i} Y_i \quad (2.18)$$

$$Y_{i+1} = Y_i - s_i 2^{-i} X_i \quad (2.19)$$

Where $X_o = 1$ and $Y_o = 0$. The final values of X and Y will yield the scaled cosine and sine values of (X_o,Y_o). This CORDIC operation is partial because it uses the angle produced by another CORDIC operation in decomposed form. In addition, the s_i bits are passed in series (most significant bit first) so that the rotation can be overlapped with the angle calculation.

The drawback here is that the scale factor becomes variable due to the iteration where no rotation is performed, and has to be calculated during computation. In addition, the s_i value is an estimate, and should the estimate be inaccurate, the convergence behavior would possibly be disturbed. Finally, the simple adders of conventional CORDIC are traded for the more complex online adders.

Later, Takagi *et al.* (1991) developed two new modifications of the redundant addition scheme which have constant scale factors. These methods are the Double Rotation Method, and the Correcting Rotation Method.

2.4.2 Double Rotation Method CORDIC

In the Double Rotation Method (Takagi *et al.*, 1991), every rotation-extension is carried out by a combination of 2 subrotation-extensions. Here again, the redundant binary representation is used, with the digit set {-1, 0, 1}, and s_i taking values in the same set. A negative rotation, a positive rotation and a non rotation are respectively produced by two negative subrotations, two positive subrotations, and one negative and one positive subrotation.

The corresponding equations are:

$$X_j = X_{j-1} - q_j 2^{-j} Y_{j-1} - p_j 2^{-2j-2} X_{j-1} \tag{2.20}$$

$$Y_j = Y_{j-1} - q_j 2^{-j} X_{j-1} - p_j 2^{-2j-2} Y_{j-1} \tag{2.21}$$

$$Z_j = Z_{j-1} - 2q_j \tan^{-1}\left(2^{-j-1}\right) \tag{2.22}$$

Equations (2.20 – 2.22) are obtained through a combination of two sets of equations which describe the rotation-extensions with the angle $\tan^{-1} 2^{-j-1}$. X_j and Y_j are represented using redundant binary numbers, and Z_j by a redundant binary fraction, the most significant digit of which is located in the j^{th} binary position. The direction

of rotation is determined by evaluating the three most significant digits of Z_{j-1}. (q_j denotes the direction of the j^{th} rotation).

q_j and p_j are obtained using $(q_j, p_j) = \begin{Bmatrix} (\bar{1},1) & if & \left[z_{j-1}^{j-1}, z_{j-1}^{j}, z_j^j\right] < 0 \\ (0,\bar{1}) & if & \left[z_{j-1}^{j-1}, z_{j-1}^{j}, z_j^j\right] = 0 \\ (1,1) & if & \left[z_{j-1}^{j-1}, z_{j-1}^{j}, z_j^j\right] > 0 \end{Bmatrix}$ (2.23)

2.4.3 The Correcting Rotation Method CORDIC

This method has one rotation-extension per iteration. Here, the signed digit, s_i is restricted to ±1. The error introduced by constraining s_i is taken care of with extra correcting iterations every m steps, where m is an arbitrary integer.

For iteration i:

When $i \bmod m \neq 0$

$$X_i = X_{i-1} - q_i 2^{-i} Y_{i-1} \quad (2.24)$$

$$Y_i = Y_{i-1} + q_i 2^{-i} X_{i-1} \quad (2.25)$$

$$Z_i = Z_{i-1} - q_i \tan^{-1}\left(2^{-i}\right) \quad (2.26)$$

where q_i is obtained by evaluating the (m-h+3) MSDs of Z_{i-1} ($h = j \bmod m$)

When $i \bmod m = 0$

$$X_i^{'} = X_{i-1} - q_i 2^{-i} Y_{i-1} \quad (2.27)$$

$$Y_i^{'} = Y_{i-1} + q_i 2^{-i} X_{i-1} \quad (2.28)$$

$$Z_i^{'} = Z_{i-1} - q_i \tan^{-1}\left(2^{-i}\right) \quad (2.29)$$

followed by

$$X_i = X_i^{'} - q_i^{'} 2^{-i} Y_i^{'} \quad (2.30)$$

$$Y_i = Y_i^{'} + q_i^{'} 2^{-i} X_i^{'} \quad (2.31)$$

$$Z_i = Z_i^{'} - q_i^{'} \tan^{-1}\left(2^{-i}\right) \quad (2.32)$$

where $q_i^{'}$ is obtained by evaluating the (m+2) MSDs of Z_{i-1}

The two major advantages of employing this scheme are that, not only is the scale factor incorporated into the iterations, it is also taken care of in such a way that the final value does not need to be scaled.

However, the number of MSDs to be examined here to determine the direction of rotation is higher than in other methods. In addition this method also yields more iterations than normal, although not as many as in the Double Rotation Method.

Later, Timmerman (1992) tried to develop a scheme for determining the SDs and carrying out the iterations in parallel. This scheme maintains the scale factor at a constant value without an increase in the number of iterations. This is done by dividing the entire process into 3 separate parts and carrying them out as follows:

i) The first $\frac{N-3}{4}$ iterations are carried out as normal, with s_i restricted to the set {-1, +1}.

ii) For iterations $\frac{N-3}{4}$ to $\frac{N+1}{2}$, if $s_i = \pm 1$, then again the iterations are carried out as normal. If, however, $s_i = 0$, then no rotation is performed. This, as we know, would result in a non-constant scale factor. To keep the scale factor constant, at this point, instead of a rotation, the vector is simply extended by the same amount as it would have been had the rotation been performed. The modification to the iterations is thus :

$$X_{i+1} = X_i + m2^{-2i-1}X_i \tag{2.33}$$

$$Y_{i+1} = Y_i + m2^{-2i-1}Y_i \tag{2.34}$$

iii) In the final set of iterations, $i > \frac{N+1}{2}$, it is assumed that the change in the scale factor is negligible, and therefore can be neglected in the cases of $s_i = 0$.

2.4.4 Differential CORDIC

Differential CORDIC (Dawid and Meyr, 1996) is made up of a sequence of absolute value computations, in addition to the normal additions, subtractions and shifts typical of CORDIC. The iteration $Z_{i+1} = Z_i - s_i \tan^{-1} 2^{-i}$ can be interpreted in the following way : The rotation direction is chosen which leads to a smaller absolute value of the new Z component. Therefore, the conventional iteration is transformed into an iteration that involves only the absolute value of the new $\hat{Z}_i$ variable.

In the rotation mode, the original recurrence of Z as stated above, is transformed to $\left|\hat{p}_{i+1}\right| = \left|p_{i+1}\right| = \left|\left|\hat{p}_i\right| - \tan^{-1} 2^{-i}\right|$, where $\hat{p}_i$ is the predicted value of p_i. Even though only the absolute value $\left|\hat{p}_{i+1}\right|$ appears in the transformed equations, we still require the sign of p_{i+1}, which is given by $sign(p_{i+1}) = sign(p_i)sign(\hat{p}_{i+1})$. This means, $sign(p_{i+1})$can be recursively calculated given sign(p_o) and $sign(\hat{p}_{i+1})$. This is what is seen as a type of differential decoding, which gives DCORDIC its name. Following this, the carry behavior of the additions of two operands was studied (Timmerman *et al.*, 1998, and Wassatsch *et al.*, 1998), where one operand is increasingly shifted right (made smaller). Special adder cells were then developed and used along with the regular adder cells. The special cells (which are much more compact in size and area) are used for the MSDs, when they are known to be 0's, and the normal adder cells for the rest of the bits. As the iterations progressed to bigger numbers, the total number of special cells increased, thus producing significant savings in terms of the area.

2.5 CORDIC Implementations on FPGA

CORDIC can be mapped onto architecture in a variety of ways. These different methods of implementation provide a variety of options, where the best tradeoff can be chosen among circuit complexity, clock cycle time, latency and throughput, depending on the application requirements (Vladimirova and Tiggeler). The following is an examination of several different architectures as implemented on FPGAs.

2.5.1 Iterative CORDIC

There are two types of Iterative CORDIC implemented. One is the Bir-Parallel Architecture, and the other is the Bit-Serial Architecture (Andraka, 1998).

2.5.1.1 Bit-Parallel Design

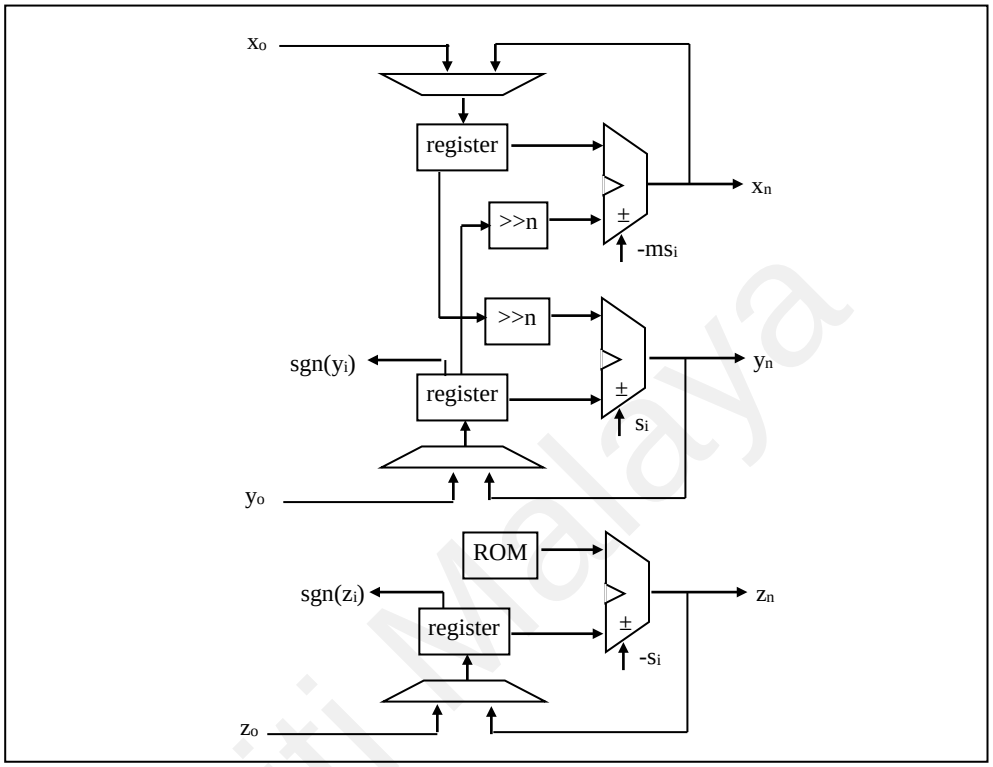

Fig. 2.3 Bit-Parallel Iterative CORDIC

Early FPGAs were not able to implement a parallel CORDIC algorithm due to limited chip size and the impossibility of routing the hard-wired shifters. Consequently, it has been performed in several FPA-based DSP applications (Meyer-Base et al., 1994, Dick, 1996, Meyer-Base et al., 1998, Mayosky et. al., 1998). There are 3 registers, one each for storing the X, Y and Z components (Andraka, 1998). There are 3 adder-subtractors for performing the additions. Depending on the mode of operation, the signed bit for the following iteration is determined. This can be seen in Figure 2.3. In each iteration, the ROM is incremented t provide the appropriate angle for the Z-adder, and the shifters are modified to select the correct degree of shift. When N iterations have been completed, the results can be directly read from the X and Y registers.

On FPGAs, the variable parallel shift registers do not map too well due to the high fan-in. The signal ends up passing through a number of FPGA cells because the shifters require several layers of logic.

2.5.1.2 Bit Serial Design

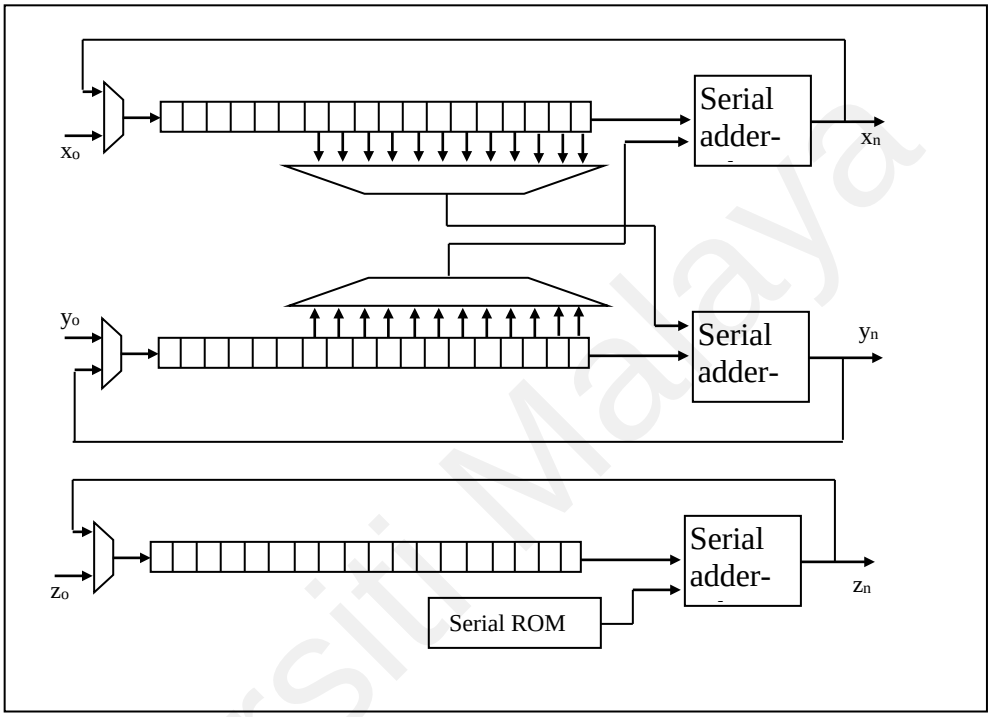

Fig. 2.4 Bit-Serial Iterative CORDIC

A more compact version is implemented with bit-serial arithmetic (Andraka, 1998). This design uses 3-bit serial adder-subtractors, 3 shift registers and a serial ROM. The length of the shift registers is N bits. The design is shown in Fig. 2.4.

Once the data is loaded into the registers (parallel/serial), the data is shifted bit by bit through the adder-subtractors and the resultant values are returned to the registers (this takes N clock cycles). The arctangent constants in the ROM for the Z-component is loaded serially. When the i^{th} iteration is complete, the signed bit for the next iteration is determined. After the N^{th} cycle of the N^{th} iteration, the process is complete and the results can be directly read from the X and Y registers.

Size-wise, this design is more compact than the bit-parallel architecture. The number of clock cycles to complete the entire CORDIC process is large, but it can be compensated for by the fact that extreme bit clock frequencies can be used.

2.4.2 On-Line CORDIC

This is an example of a CORDIC processor that is combinatorial (Andraka, 1998). Instead of using clocked registers, the entire CORDIC processor is unrolled to produce an array of interconnected adder-subtractors. Two distinct advantages of this arrangement are that the shifters are not variable-bit shifters, but hardwired shifters, each one performing a fixed shift. The second is that the look-up values for the angle accumulator are distributed as constants to each adder in the angle accumulator chain. The delay through this huge amount of combinatorial logic is large, but the processing time is reduced compared to the iterative circuit. However, such a large combinatorial design is not suited to FPGAs.

The unrolled process can be easily pipelined by adding registers between the adder-subtractors.

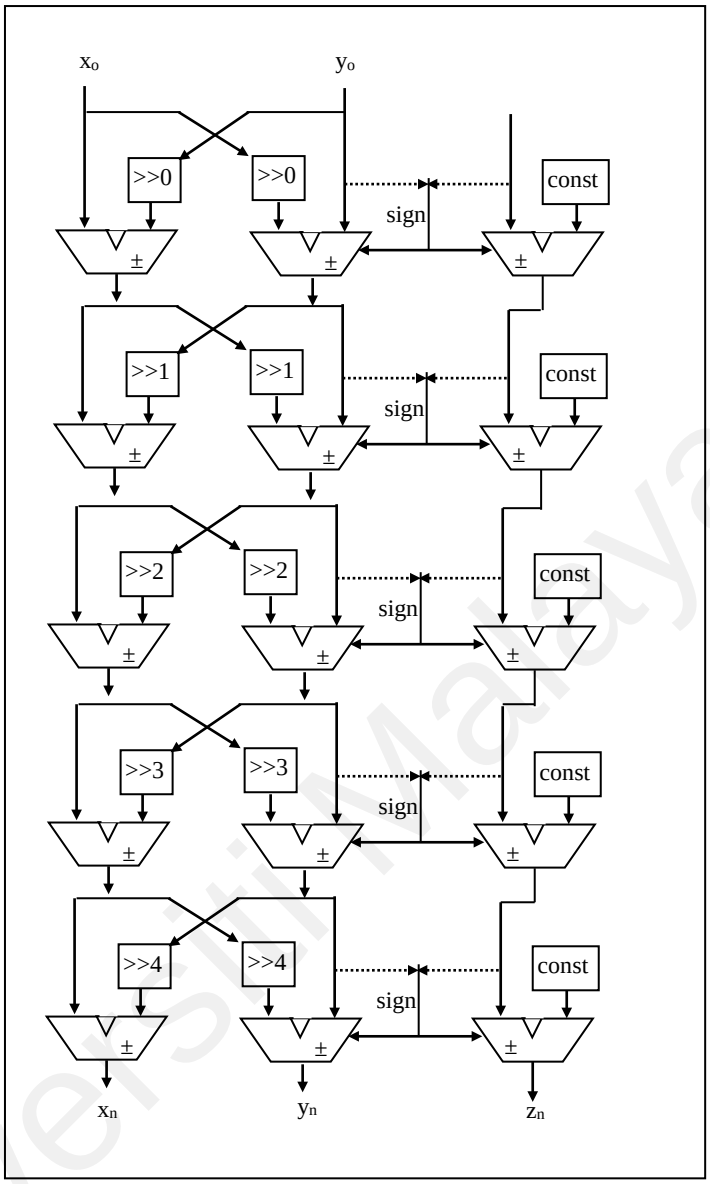

Fig. 2.5 Unrolled CORDIC Processor

In addition, Valls et al. (2002) studied the suitability of redundant arithmetic operators in full custom based CORDIC implementations. It was found that while these methods yielded improvements in speed, the resultant area became close to 4 or 5 times larger than the conventional 2's complement method.

Chapter 3 : FLAT CORDIC

3.1 Introduction to Flat CORDIC

Flat CORDIC is a revolutionary manipulation of the conventional CORDIC algorithm that has produced immense savings in both area and speed. It was put forth by Clarke (1995). Despite all the improvements of the previous modifications of CORDIC, the characteristic iterative process has always remained a bottleneck. Flat CORDIC is the first method to eliminate these iterations to allow for a new level of refinement and optimization. In this method, the results are completely expressed in terms of the original input vector. This is achieved through successive substitution of the original vector in each iterative equation.

3.2 The Flat CORDIC Equations

From Chapter 2, the main CORDIC equations are as stated below :

$$X_{i+1} = X_i - ms_i Y_i 2^{-i} \tag{3.1}$$

$$Y_{i+1} = Y_i + s_i X_i 2^{-i} \tag{3.2}$$

where (X_i, Y_i) is the original vector, and (X_{i+1}, Y_{i+1}) is the rotated vector. In the next iteration,

$$X_{i+2} = X_{i+1} - ms_{i+1} Y_{i+1} 2^{-(i+1)} \tag{3.3}$$

$$Y_{i+2} = Y_{i+1} + s_{i+1} X_{i+1} 2^{-(i+1)} \tag{3.4}$$

Substituting (3.1) and (3.2) in (3.3) and (3.4), we get

$$X_{i+2} = X_i\left(1 - ms_i s_{i+1} 2^{-i} 2^{-(i+1)}\right) - mY_i\left(s_i 2_{-i} + s_{i+1} 2^{-(i+1)}\right) \tag{3.5}$$

$$Y_{i+2} = Y_i\left(1 - ms_i s_{i+1} 2^{-i} 2^{-(i+1)}\right) + X_i\left(s_i 2_{-i} + s_{i+1} 2^{-(i+1)}\right) \tag{3.6}$$

The results of this second rotation are given directly in terms of the original vector (X_i, Y_i). Proceeding in this manner, successive substitution gives us a generalized Flat CORDIC equation that is completely flattened and parallelized :

$$X_N = X_o\left[1 - \sum_{i=1}^{N-1}\sum_{j=i+1}^{N} s_i s_j 2^{-i} 2^{-j} + \sum_{i=1}^{N-3}\sum_{j=i+1}^{N-2}\sum_{k=j+1}^{N-1}\sum_{l=k+1}^{N} s_i s_j s_k s_l 2^{-i} 2^{-j} 2^{-k} 2^{-l} - \ldots\right]$$
$$- mY_o\left[\sum_{i=1}^{N} s_i 2^{-i} - \sum_{i=1}^{N-2}\sum_{j=i+1}^{N-1}\sum_{k=j+2}^{N} s_i s_j s_k 2^{-i} 2^{-j} 2^{-k} + \sum_{i=1}^{N-4}\sum_{j=i+1}^{N-3}\sum_{k=j+1}^{N-2}\sum_{l=k+1}^{N-1}\sum_{m=l+1}^{N} s_i s_j s_k s_l s_m 2^{-i} 2^{-j} 2^{-k} 2^{-l} 2^{-m} - \ldots\right] \tag{3.7}$$

$$Y_N = Y_o\left[1 - \sum_{i=1}^{N-1}\sum_{j=i+1}^{N} s_i s_j 2^{-i} 2^{-j} + \sum_{i=1}^{N-3}\sum_{j=i+1}^{N-2}\sum_{k=j+1}^{N-1}\sum_{l=k+1}^{N} s_i s_j s_k s_l 2^{-i} 2^{-j} 2^{-k} 2^{-l} - \ldots\right]$$
$$+ X_o\left[\sum_{i=1}^{N} s_i 2^{-i} - \sum_{i=1}^{N-2}\sum_{j=i+1}^{N-1}\sum_{k=j+2}^{N} s_i s_j s_k 2^{-i} 2^{-j} 2^{-k} + \sum_{i=1}^{N-4}\sum_{j=i+1}^{N-3}\sum_{k=j+1}^{N-2}\sum_{l=k+1}^{N-1}\sum_{m=l+1}^{N} s_i s_j s_k s_l s_m 2^{-i} 2^{-j} 2^{-k} 2^{-l} 2^{-m} - \ldots\right] \tag{3.8}$$

To calculate sine/cosine functions, rotation mode CORDIC is used in circular coordinates. The initial vector is (1,0). The Flat CORDIC Equations now become

$$X_N = 1 - \sum_{i=1}^{N-1}\sum_{j=i+1}^{N} s_i s_j 2^{-i} 2^{-j} + \sum_{i=1}^{N-3}\sum_{j=i+1}^{N-2}\sum_{k=j+1}^{N-1}\sum_{l=k+1}^{N} s_i s_j s_k s_l 2^{-i} 2^{-j} 2^{-k} 2^{-l} \tag{3.9}$$

$$Y_N = \sum_{i=1}^{N} s_i 2^{-i} \sum_{i=1}^{N-2}\sum_{j=i+1}^{N-1}\sum_{k=j+2}^{N} s_i s_j s_k 2^{-i} 2^{-j} 2^{-k} + \sum_{i=1}^{N-4}\sum_{j=i+1}^{N-3}\sum_{k=j+1}^{N-2}\sum_{l=k+1}^{N-1}\sum_{m=l+1}^{N} s_i s_j s_k s_l s_m 2^{-i} 2^{-j} 2^{-k} 2^{-l} 2^{-m} \tag{3.10}$$

$$\left(X_N, Y_N\right) = \left(K_N \cos\theta, K_N \sin\theta\right) \tag{3.11}$$

A quick glance shows that firstly, an efficient implementation would require prior knowledge of the polarity of the micro-rotation (the signed digits, or SDs). Secondly, the number of combinations of SDs would very quickly become huge as the size N increases.

3.3 Generation of SDs

Conventionally, each CORDIC iteration generates one SD from the equation

$$Z_{i+1} = Z_i - s_i \tan^{-1} 2^{-i}$$

(3.12)

The sign of the following iteration, s_{i+1} comes from the most significant bit (MSB) of $\mathbf{z}_{\mathbf{i+1}}$. Now, suppose we could make the approximation $\tan^{-1} 2^{-i} \approx 2^{-i}$. Then, $Z_{i+1} = Z_i - s_i 2^{-i}$, and we would be able to directly generate all the SDs from the original input angle. For small values of i, this approximation is not valid. Alternatively, we assume $\tan^{-1} 2^{-\mathbf{i}} \approx 2^{-\mathbf{i}}$ for i > M (Gisuthan, 2000). If we can identify a value for M, then we need only carry out M iterations. Then, all the SDs can be obtained directly from the remaining angle after M iterations.

Theorem 1 shows that the value of M that validates our assumption for N-bit size is N/3. Theorem 2 shows that the remaining SDs can be obtained directly from Z_M.

3.3.1 Theorem 1

Theorem 1 : For N-bit CORDIC, $\tan^{-1} 2^{-\mathbf{i}} \neq 2^{-\mathbf{i}}$ for $\mathbf{i} = 1 \textbf{ to } \left\lceil \frac{\mathbf{N} - \log_2 3}{3} \right\rceil$

Proof : The expansion of $\tan^{-1} x$ for $|x| \leq 1$ is

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots \tag{3.13}$$

For $x = 2^{-i}$, $\quad \tan^{-1}\left(2^{-i}\right) = 2^{-i} - \frac{2^{-3i}}{3} + \frac{2^{-5i}}{5} - \frac{2^{-7i}}{7} + \frac{2^{-9i}}{9} - \ldots$ (3.14)

For N-bit accuracy, the necessary conditions for obtaining the minimum value of i for which the expression $\tan^{-1} 2^{-i} = 2^{-i}$ is valid is that the error due to the i[th] bit must be $< 2^{-N}$. That is: $\quad e_i = 2^{-i} - \tan^{-1} 2^{-i} < 2^{-N}$ (3.15)

But $e_i = \frac{2^{-3i}}{3} - \frac{2^{-5i}}{5} + \frac{2^{-7i}}{7} - \frac{2^{-9i}}{9} + \ldots < \frac{2^{-3i}}{3} - \frac{2^{-5i}}{3} + \frac{2^{-7i}}{3} - \frac{2^{-9i}}{3} + \ldots$ (3.16)

The next condition to obtain the minimum value of i is : $\frac{2^{-3i}}{3} < 2^{-N}$ (3.17)

$$-3i - \log_2 3 < -N \quad \therefore \quad i > \frac{N - \log_2 3}{3} \tag{3.18}$$

i is an integer, therefore i is the smallest integer that is equal to or larger than $\frac{N - \log_2 3}{3}$. If N is a multiple of 3, then Equation (3.18) is true for $i = \frac{N}{3}$

Now, we need to show that the accumulated error, $\left| \sum_{i=N/3}^{N} e_i \right| < 2^{-N}$

From (3.16) $\quad \sum_{i=N/3}^{N} e_i = \sum_{i=N/3}^{N} \frac{2^{-3i}/3}{1 + 2^{-2i}} < \sum_{i=N/3}^{N} 2^{-3i}/3 < 2^{-N}$ (3.19)

It has been verified that for the first $\frac{N}{3} - 1$ bits, the arctangent value of each bit $\left(\tan^{-1}\left(2^{-i}\right)\right)$ is not equal to the positional value of the corresponding bit in the normal

binary representation $\left(2^{-i}\right)$, whereas for bits $\frac{N}{3}$ to N, they are equal for N-bit accuracy. Due to this discrepancy, it is not possible to directly precompute the SDs corresponding to the first $\frac{N}{3}-1$ bits.

3.3.2 Theorem 2

Theorem 2 : In N-bit CORDIC, the first $\frac{N}{3}-1$ signed digits (SDs) cannot be pre-computed

Proof : Let $\mathbf{s_i}$ be the SD of the i[th] iteration. It carries a weight of $\tan^{-1} 2^{-\mathbf{i}}$ which represents an angle measured in radians. Let a_k be the k[th] bit of the angle remaining after i iterations and A(i) be the angle remaining after the i[th] iteration.

$A(i) = A(0) - \sum_{j=1}^{i} s_j \tan^{-1} 2^{-j}$ where $A(0)$ is the input angle for rotation. (3.20)

It is assumed that angles are represented in $(1,-1)$ format, so after i iterations have been performed, $s_{i+1} = a_{i+1}$. After i iterations, the Signed Binary Number Representation (SBNR) of $A(i)$ must have the first i bits as zeros and the remaining bits strictly non-zeros.

In all these cases, it is assumed only one SD per iteration is considered. If m (1 < m < N) SDs are considered at the $(\mathbf{i}+1)^{\mathbf{th}}$ iteration, then $A(i+1)$ holds the following inequality (Walther, 1971).

$$\left|A(i+m)\right| = \left|A(i) - \sum_{j=i+1}^{i+m} s_j \tan^{-1} 2^{-j}\right| < \tan^{-1}\left(2^{-(i+m)}\right) \quad (3.21)$$

It must now be shown that this inequality fails if there exists a situation in which the computation of more than 1 SD results in an error.

Consider a case where $a_{i+1} = 1$ and $a_{i+2} = -1$

Then, $s_{i+1} = 1$ and $s_{i+2} = -1$

And $$|A(i+1)| = \left| A(i) - \sum_{j=i+1}^{i+2} s_j \tan^{-1} 2^{-j} \right| < \tan^{-1}\left(2^{-(i+2)}\right) \quad (3.22)$$

But $$A(i) = \sum_{k=i+1}^{N} 2^{-k} a_k \quad (3.23)$$

ie.,

$$\left| \sum_{k=i+3}^{N} \left(2^{-k} a_k\right) + \left(a_{i+1} 2^{-(i+1)} + a_{i+2} 2^{-(i+2)}\right) - \left(s_{i+1} \tan^{-1}\left(2^{-(i+1)}\right) + s_{i+2} \tan^{-1}\left(2^{-(i+2)}\right)\right) \right| < \tan^{-1} 2^{-(i+1)}$$

(3.24)

Maximising A(i) by setting all a_k values to 1 for $(i+3) \le k \le N$,

$$A(i)_{\max} = \left| \left(2^{-(i+2)} - 2^{-N}\right) + \left(2^{-(i+1)} - 2^{-(i+2)}\right) \right| = \left| 2^{-(i+1)} - 2^{-N} \right| < 2^{-(i+1)} \quad (3.25)$$

ie. $$A(i+1) = \left\| \left(2^{-(i+1)} - 2^{-N}\right) - \left(\tan^{-1} 2^{-(i+1)} - \tan^{-1} 2^{-(i+2)}\right) \right\| < 2^{-(i+2)} \quad (3.26)$$

and so, $$2^{-(i+1)} - \tan^{-1} 2^{-(i+1)} < 2^{-N} \quad (3.27)$$

but this condition is only satisfied when $i \ge N/3$

Therefore, the inequality (3.21) is not valid up to $N/3 - 1$ iterations, and it is not possible to predict more than 1 SD per iteration until the $(N/3 - 1)^{\text{th}}$ iteration is performed. However this does not prove that it is possible to predict more than 1 SD per iteration beyond $N/3 - 1$ iterations.

In order to ascertain that all SDs beyond $N/3 - 1$ iterations can be pre-computed in parallel, we must first prove that the bits of $A(N/3 - 1)$ which has the value

$\sum_{N/3}^{N} a_j \tan^{-1} 2^{-j}$ {where the first bits of $A(N/3 - 1)$ are 0's} are the SDs.

For this to be true, the following must be valid:

$$\sum_{N/3}^{N} a_j 2^{-j} - \sum_{N/3}^{N} s_j \tan^{-1} 2^{-j} < 2^{-N} \quad (3.28)$$

Theorem 1 proves that for N-bit accuracy, $\tan^{-1} 2^{-i} = 2^{-i}$ for $i = N/3$ to **N** (3.29)

Therefore, (3.15) is valid, which implies that after $N/3$ iterations, the remaining SDs can be pre-computed in parallel

3.4 Combination of SDs and Addition

After the signed digits have been pre-computed, the evaluation of the Flat CORDIC equation involves the summation of the positional valued products of the signed digits in different combinations. Each term consists of two parts, namely the signed digit combination part, and the positional value part, as seen in Fig. 3.1.

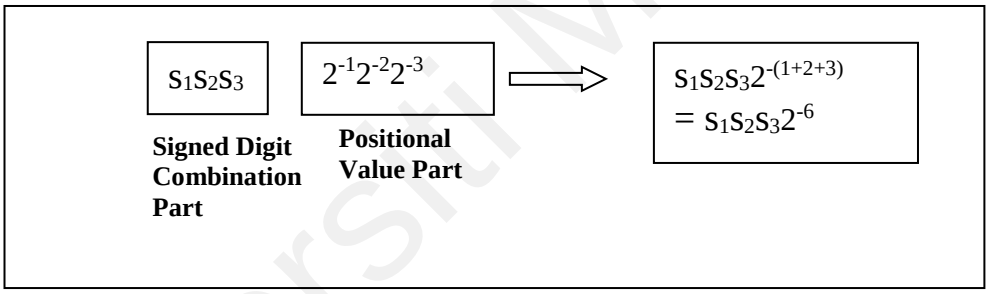

Fig. 3.1 Signed Digit Combinations and Positional Value

For the example shown in Fig. 3.1, the cumulative index value (CIV) is the sum of the negative indices (the total binary shift). As previously mentioned, the number of combinations increases sharply with the size N. To reduce this, exhaustive error analysis has shown that any term with a cumulative index value (CIV) greater than the value E_N, does not affect the results of X_N and Y_N, and can be left out. For N-bit CORDIC, $E_N = N + \log_2 N + 2$. For Flat CORDIC, $E_N = N + \log_2 N$.

Graph 3.2 depicts the rise in the number of combinations for each size N, using $E_N = N + \log_2 N$. These terms are taken for X_N calculation. The difference in the number of terms between X_N and Y_N calculation is not very significant.

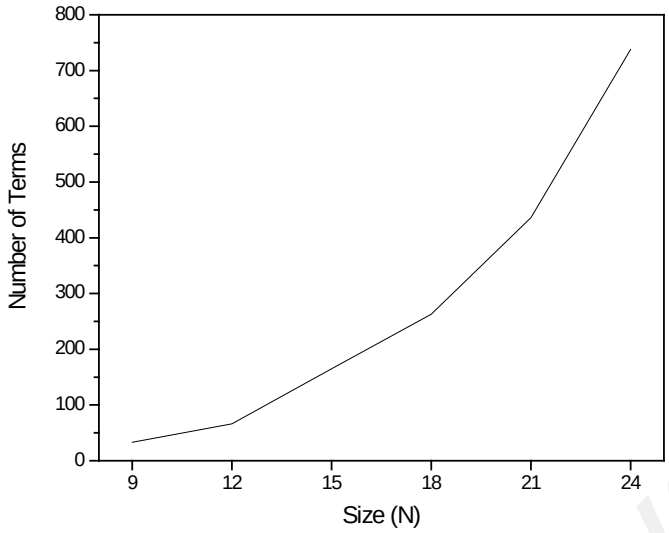

Figure 3.2 Graph Number of Combinational Terms Versus Size (N)

The number of terms increases sharply with size (N). Since both the X_N as well as the Y_N equations need to be calculated, a complete set has almost double the number of combinations/terms indicated in Fig. 3.2.

For a hardware implementation of Flat CORDIC, equations 3.9 and 3.10 are rewritten as

$$X_N = \left[1 - \begin{Bmatrix} 2^{-3}\left(\sum s_i \cdot s_j\big|_{i+j=3} - \sum s_i \cdot s_j \cdot s_k \cdot s_l\big|_{i+j+k+l=3} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n\big|_{i+j+k+l+m+n=3}\right) \\ + 2^{-4}\left(\sum s_i \cdot s_j\big|_{i+j=4} - \sum s_i \cdot s_j \cdot s_k \cdot s_l\big|_{i+j+k+l=4} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n\big|_{i+j+k+l+m+n=4}\right) \\ + \ldots \\ + 2^{-E_N}\left(\sum s_i \cdot s_j\big|_{i+j=E_N} - \sum s_i \cdot s_j \cdot s_k \cdot s_l\big|_{i+j+k+l=E_N} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n\big|_{i+j+k+l+m+n=E_N}\right) \end{Bmatrix}\right] \tag{3.30}$$

and

$$Y_N = \begin{bmatrix} 2^{-1}\left(s_1 - \sum s_i \cdot s_j \cdot s_k\big|_{i+j+k=1} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m=1}\right) \\ + 2^{-2}\left(s_2 - \sum s_i \cdot s_j \cdot s_k\big|_{i+j+k=2} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m=2}\right) \\ + 2^{-3}\left(s_{24} - \sum s_i \cdot s_j \cdot s_k\big|_{i+j+k=3} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m=34}\right) \\ + \ldots \\ + 2^{-E_N}\left(- \sum s_i \cdot s_j \cdot s_k\big|_{i+j+k=E_N} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m=E_N}\right) \end{bmatrix} \tag{3.31}$$

This divides all the combinations into a set of channels. The terms are assigned to their respective channels by their CIV. The sum of all the combinations is then shifted accordingly. After all the channel sums have been collected and shifted, these values are added together to get the final unscaled X_N and Y_N values.

The final simplification comes from limiting the input angle so $\theta > 0$ radians. Then, the first rotation is always anti-clockwise, and so the first SD (s_1) is fixed as +1. Now, all the terms with s_1 can be simplified to produce Equations 3.32 and 3.33.

$$X_N = \left[1 - \left\{ \begin{array}{l} 2^{-3}\left(\begin{array}{l} \sum s_j \big|_{j+1=3} + \sum s_i \cdot s_j \big|_{i+j=3} - \sum s_j \cdot s_k \cdot s_l \big|_{i+j+k+1=3} - \sum s_i \cdot s_j \cdot s_k \cdot s_l \big|_{i+j+k+l=3} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \big|_{i+j+k+l+m+1=3} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n \big|_{i+j+k+l+m+n=3} \end{array} \right) \\ + 2^{-4}\left(\begin{array}{l} \sum s_j \big|_{j+1=4} + \sum s_i \cdot s_j \big|_{i+j=4} - \sum s_j \cdot s_k \cdot s_l \big|_{i+j+k+1=4} - \sum s_i \cdot s_j \cdot s_k \cdot s_l \big|_{i+j+k+l=4} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \big|_{i+j+k+l+m+1=4} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n \big|_{i+j+k+l+m+n=4} \end{array} \right) \\ ..... \\ + 2^{-E_N}\left(\begin{array}{l} \sum s_j \big|_{j+1=3} + \sum s_i \cdot s_j \big|_{i+j=E_N} - \sum s_j \cdot s_k \cdot s_l \big|_{i+j+k+1=E_N} - \sum s_i \cdot s_j \cdot s_k \cdot s_l \big|_{i+j+k+l=E_N} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \big|_{i+j+k+l+m+1=E_N} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n \big|_{i+j+k+l+m+n=E_N} \end{array} \right) \end{array} \right\} \right] \tag{3.32}$$

$$Y_N = \left[\begin{array}{l} 2^{-1}\left(\begin{array}{l} 1 - \sum s_j \cdot s_k \big|_{j+k+1=1} - \sum s_i \cdot s_j \cdot s_k \big|_{i+j+k=1} + \sum s_j \cdot s_k \cdot s_l \cdot s_m \big|_{j+k+l+m+1=1} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \big|_{i+j+k+l+m=1} - ... \end{array} \right) \\ + 2^{-2}\left(\begin{array}{l} 1 - \sum s_j \cdot s_k \big|_{j+k+1=2} - \sum s_i \cdot s_j \cdot s_k \big|_{i+j+k=2} + \sum s_j \cdot s_k \cdot s_l \cdot s_m \big|_{j+k+l+m+1=2} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \big|_{i+j+k+l+m=2} - ... \end{array} \right) \\ ..... \\ + 2^{-E_N}\left(\begin{array}{l} 1 - \sum s_j \cdot s_k \big|_{j+k+1=E_N} - \sum s_i \cdot s_j \cdot s_k \big|_{i+j+k=E_N} + \sum s_j \cdot s_k \cdot s_l \cdot s_m \big|_{j+k+l+m+1=E_N} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \big|_{i+j+k+l+m=E_N} - ... \end{array} \right) \end{array} \right] \tag{3.33}$$

Although this seems more complex, the range of i is: $\mathbf{i} \geq 2$, which reduces the number of gates required for the combinations.

CHAPTER 4 : SIGNED DIGIT GENERATION

4.1 The Signed Digit Algorithm (SDA)

Due to the discrepancy between the values of $\tan^{-1} 2^{-i}$ and 2^{-i}, as seen earlier, the polarity of all the micro-rotations or Signed Digits (SDs), cannot be pre-computed. Theorem 2 however showed that it is only the first N/3 SDs (for N-bit Flat CORDIC) that cannot be pre-computed; the following 2N/3 SDs can be generated in parallel. Then, Bimal (2000) discovered the Signed Digit Algorithm (SDA), a ROM-less algorithm to generate all the SDs.

Here, the input angle (represented in radians) is restricted to the range $|\theta| < 45^\circ$ (so that the angle value is always smaller than 1 radian). The input angles outside this range are suitably manipulated. The N-bit input angle, Zo, is then divided into two parts : the first N/3 bits (the Most Significant Bits, MSBs) and the remaining 2N/3 bits (Least Significant Bits, LSBs). The MSBs are then channeled into a unit that generates the first N/3 SDs (Most Significant Signed Digits, MSSDs). The MSSDs are also used to generate an angle, $\mathbf{Z'_{rem}}$ which is explained later.

The LSBs are combined with Z'_{rem} and the result is used to generate the rest of the SDs (Least Significant Signed Digits, LSSDs).

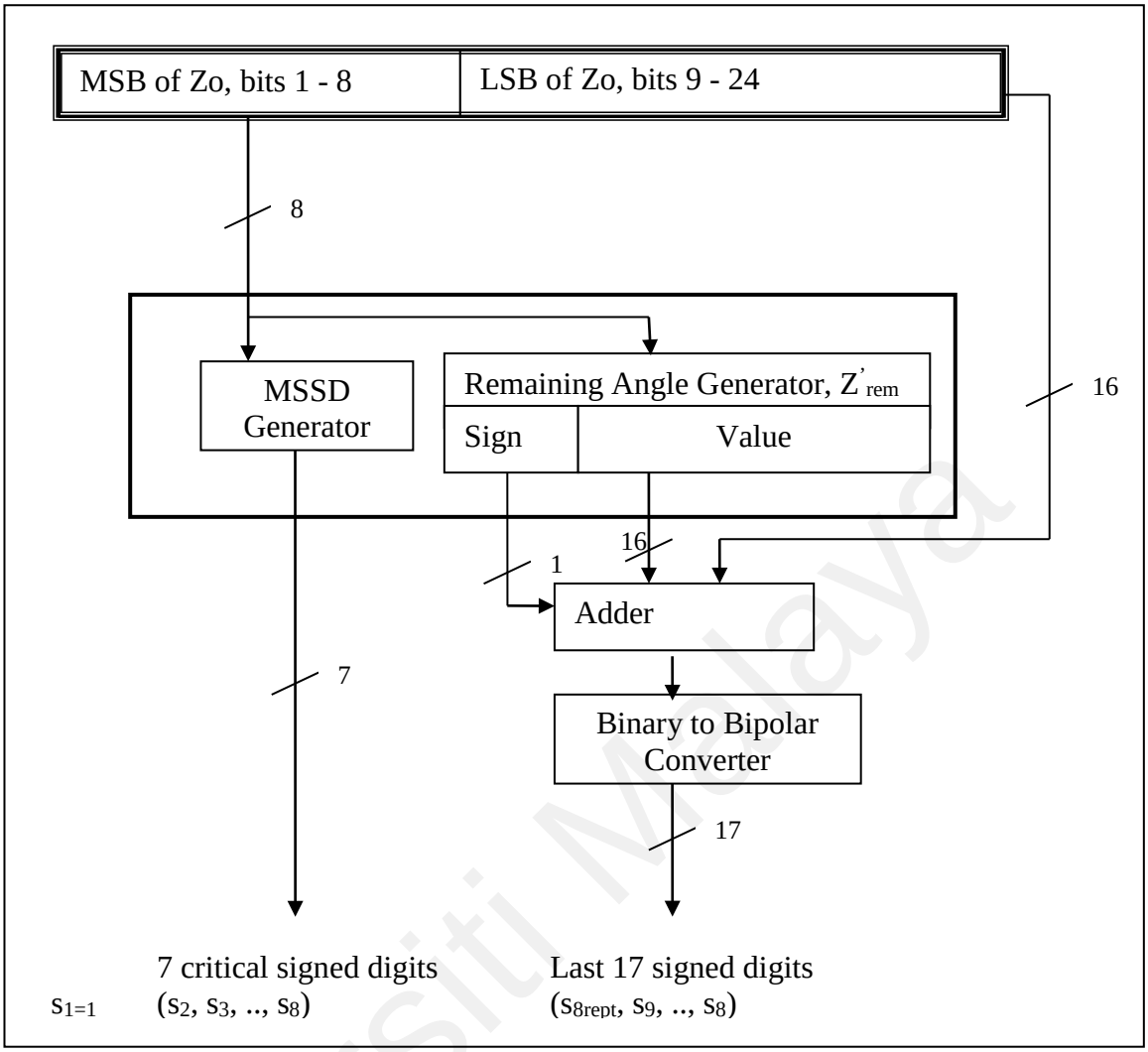

Figure 4.1 Signed Digit Generation for Size N = 24

4.1.1 Most Significant Signed Digit (MSSD) Generation

For N-bit Flat CORDIC, for the $2^{N/3}$ combinations of MSBs of Z_o, the resulting MSSDs were observed. A striking pattern, known as the pair-wise linearity (PWL) relationship was seen. Each consecutive pair of MSBs share a common set of MSSDs. For the smaller sizes of N, this pattern was unbroken. As N got bigger, however, there were some breakpoints that divided the entire set into several regions, in each of which the pair-wise linear pattern was seen. This relationship is illustrated in Table 4.1.

Table 4.1 Pair-Wise Equality of Signed Digits for Size N = 24

Region	Z_o (radians)	MSB of Z_o	MSSDs (s_1 to s_8)	MSSDs in decimal
ONE	0.00000000	00000000	01111100	124
	0.00390625	00000001	01111011	123
	0.00781250	00000010	01111011	123
	0.01171875	00000011	01111010	122
	0.01562500	00000100	01111010	122
	0.01953125	00000101	01111001	121
	0.02343750	00000110	01111001	121
	…..			
	0.20703125	00110101	01100001	97
	0.21093750	00110110	01100001	97
	0.21484375	00110111	01100000	96
TWO	0.21875000	00111000	01011111	95
	0.22265625	00111001	01011111	95
	…..			
	0.45312500	01110100	01000001	65
	0.45703125	01110101	01000001	65
	0.46093750	01110110	01000000	64
THREE	0.46484375	01110111	00111111	63
	0.46875000	01111000	00111110	62
	0.47265625	01111001	00111110	62
	…..			
	0.77343750	11000110	00010111	23
	0.77734375	11000111	00010111	23
	0.78125000	11001000	00010110	22
	0.78515625	11001001	00010110	22

In the case of N = 24, there are 2 breakpoints, which divide the range of values into 3 regions. The SD value ‘1’ represents anticlockwise rotation, and the ‘0’ represents

clockwise rotation. The PWL mentioned can be seen in the table above. Within the regions, each pair of Z_o values share a common set of MSSDs.

4.1.2 Least Significant Signed Digit (LSSD) Generation

If the first N/3 iterations were really performed, the remaining angle after these iterations would be:

$$Z_{rem} = Z_o - \left(\overbrace{s_1 \tan^{-1} 2^{-1}}^{\text{first iteration}} + \overbrace{s_2 \tan^{-1} 2^{-2}}^{\text{second iteration}} + s_3 \tan^{-1} 2^{-3} + \ldots + \overbrace{s_{N/3} \tan^{-1} 2^{-N/3}}^{(N/3)^{th}\text{ iteration}}\right) \quad (4.1)$$

Suppose we separate Z_o into two parts, $Z_{o,MSB}$ and $Z_{o,LSB}$, each one with the size N bits.

$Z_{o,MSB}$ is made up of the MSBs of the original Z_o, followed by a string of 0's

$Z_{o,LSB}$ is made up of $N/3$ 0's, followed by the LSB of the original Z_o

Then,

$$Z_{rem} = \left[Z_{o,MSB} - \left(s_1 \tan^{-1} 2^{-1} + s_2 \tan^{-1} 2^{-2} + s_3 \tan^{-1} 2^{-3} + \ldots + s_{N/3} \tan^{-1} 2^{-N/3}\right)\right] + Z_{o,LSB}$$

$$Z_{rem} = Z'_{rem} + Z_{o,LSB} \quad (4.2)$$

The first $N/3$ iterations, as we know, are carried out to obtain the MSSDs. Section 4.1.1 has shown a definite pattern for obtaining the MSSDs without the need for these iterations by its association with $Z_{o,MSB}$ (as will be explained in detail in Section 4.2.1). However, if these iterations are not carried out, we will not have the Z'_{rem} value.

This led to an inspection of the Z'_{rem} values with the corresponding MSBs of Z_o as shown in Table 4.2. Here too a pattern was found :

1) The sign of Z'_{rem} alternated between +1 and –1 for each consecutive $Z_{o,MSB}$ value (represented by 0 and 1)

2) the value of Z'_{rem} was found to exhibit the pair-wise linearity (PWL) relationship much the same as for the MSSDs.

3) This pattern is seen to break at the same places where the MSSD PWL is broken

The alternating sign of Z'_{rem} is explained by considering the following two angles, $Z_{o,MSB}$ and $\left(Z_{o,MSB} + 2^{-N/3}\right)$. Since $Z_{o,MSB}$ is an N-bit angle defined only by the first $N/3$ bits, to $N/3$-bit accuracy, the two angles are considered adjacent.

As defined by the convergence property of CORDIC, $\left|Z'_{rem}\right| < 2^{-N/3}$. For the first angle,

$$\left| Zo_{,MSB} - \left(\sum_{i=1}^{N/3-2} s_i \tan^{-1} 2^{-i} + s_{N/3-1} \tan^{-1} 2^{-(N/3-1)} + s_{N/3} \tan^{-1} 2^{-(N/3)} \right) \right| < 2^{-N/3} \quad (4.3)$$

If the value within these absolute limits is negative, then the convergence property will be followed for the same SDs as:

$\left(Zo_{,MSB} + 2^{-N/3}\right) - \left(\sum_{i=1}^{N/3-2} s_i \tan^{-1} 2^{-i} + s_{N/3-1} \tan^{-1} 2^{-(N/3-1)} + s_{N/3} \tan^{-1} 2^{-(N/3)} \right) < 2^{-N/3}$, which will be positive

Therefore, it can be expected that the sign of Z'_{rem} would alternate from one value of $Z_{o,MSB}$ to the next. The values of the remaining bits of Z'_{rem} were observed with relation to the MSBs of Z_o. A PWL relationship was seen in exactly the same regions, with the same breakpoints. Generating Z'_{rem} therefore can be done with an encoder.

Table 4.2 shows the values of Z'_{rem} and the MSBs of Z_o

Table 4.2 Z'_{rem} for MSBs of Z_o for size N = 24

Region	Z_o (radians)	MSB of Z_o	Z'_{rem}	
			Sign	**Value**
ONE	0.00000000	00**00000**0	0	1101010011111011
	0.00390625	00**00000**1	1	1101010100100000
	0.00781250	00**00001**0	0	1101010100100000
	0.01171875	00**00001**1	1	1101010100100000
	0.01562500	00**00010**0	0	1101010100100000
	0.01953125	00**00010**1	1	1101010100100101
	0.02343750	00**00011**0	0	1101010100100101
	…..			
	0.20703125	00**11010**1	1	1110000100011110
	0.21093750	00**11011**0	0	1110000100011110
	0.21484375	00**11011**1	1	1110000100011111
TWO	0.21875000	00**11100**0	1	0010100110000000
	0.22265625	00**11100**1	0	0010100110000000
	…..			
	0.45312500	01**11010**0	1	0011010110101001
	0.45703125	01**11010**1	0	0011010110101001
	0.46093750	01**11011**0	1	0011010110101010
THREE	0.46484375	01**11011**1	0	0110011100011101
	0.46875000	01**11100**0	1	0110011100011110
	0.47265625	01**11100**1	0	0110011100011110
	…..			
	0.77343750	11**00011**0	1	1011110011111101
	0.77734375	11**00011**1	0	1011110011111101
	0.78125000	11**00100**0	1	1011110011111110
	0.78515625	11**00100**1	0	1011110011111110

The value of Z'_{rem} is added to the LSB of the original input angle to generate the Z_{rem} value of Equation 4.1. The sum goes through a binary to bipolar converter to generate all the LSSDs in parallel. This is further explained in Section 4.2.2.

4.2 Implementation of the SDA

4.2.1 Algorithm for MSSD

Table 4.1 shows that the breakpoints that bound the 3 regions occur at “00110111” and “01110111”. A simple algorithm is used to classify the regions, and generate the MSSDs, as shown below :

inp_msb[0:7] denotes the MSBs of Z_o, and mssd_out represents the SDs($s_2 \rightarrow s_8$)

```
If inp_msb [1 to 8]     >= “01110111”
                            mssd_out = “0111010” – inp_msb[1 to 7];
                            region <= “11”;                 (Region 3)
else if inp_msb[1 to 8] >= “00110111”
                            mssd_out = “0111010” – inp_msb[1 to 7];
                            region <= “10”;                 (Region 2)
else if inp_msb[8] = ‘0’
                            mssd_out = “01111100” – inp_msb[1 to 7];
                            Region <= “00”;                 (Region 1)
else                        mssd_out ‘ “01111011” – inp_msb[1 to 7];
                            region <= “00”;                 (Region 1)
```

Figure 4.2 MSSD Generation Algorithm for Size N = 24

4.2.2 Generation of Z'_{rem}

i) In Region One, the sign bit follows inp_msb[0]; in the other regions, it is the opposite

ii) In all regions, with the exception of the sign bit, the Z'_{rem} values occur in pairs. In Region One, the $\mathbf{Z'_{rem}}$ pairs are odd-even, whereas in Regions Two and Three, they are even-odd.

iii) For Region One, the odd values of the $Z_{o,MSB}$ are incremented before feeding the encoder. This is due to the odd-even nature of the pair. A 5-16 bit encoder is used to encode the remaining angle Z'_{rem}

iv) For Regions Two and Three, 5-16 bit and 6-16 bit encoders are used respectively

v) We know that $Z'_{rem} < 2^{-\left(\frac{N}{3}-1\right)}$, and that $\mathbf{Z_{o,LSB}} < 2^{-\left(\frac{N}{3}-1\right)} - 2^{-N}$ Since both of these values are $< 2^{-\left(\frac{N}{3}-1\right)}$, the result of their addition becomes constrained by $2^{-\left(\frac{N}{3}-2\right)}$

Therefore, $\left|\mathbf{Z'_{rem}} + \mathbf{Z_{o,LSB}}\right| < 2^{-\frac{N}{3}+2}$ This means, the result requires an extra bit to represent it, which makes a total of $\left(\frac{2}{3}N+1\right)$bits

4.2.3 Binary - Bipolar Conversion and LSSD Generation

Z_{rem} can either be a positive or a negative value, depending on its sign bit. Just as the MSSDs are represented in Signed Binary Number Representation (SBNR), Z_{rem} also needs to be converted to SBNR. The conversion is as follows:

i) if Z_{rem} is positive, the first signed digit $s_{\frac{N}{3},repeat}$ is '0'. The next signed digit, $s_{\frac{N}{3}+1}$ is '1' if the next bit of Z_{rem} is 0, and '0' otherwise. This is carried on until all the rest of the signed digits are obtained (ie. until s_N)

ii) *if* Z_{rem} is negative, the first signed digit $s_{\frac{N}{3},repeat}$ is '1'. The next signed digit, $s_{\frac{N}{3}+1}$ is '1' if the next bit of Z_{rem} is 0, and '0' otherwise. This is carried on until all the rest of the signed digits are obtained (ie. until s_N)

Fig. 4.3 depicts the LSSD Generation as it is implemented in VHDL.

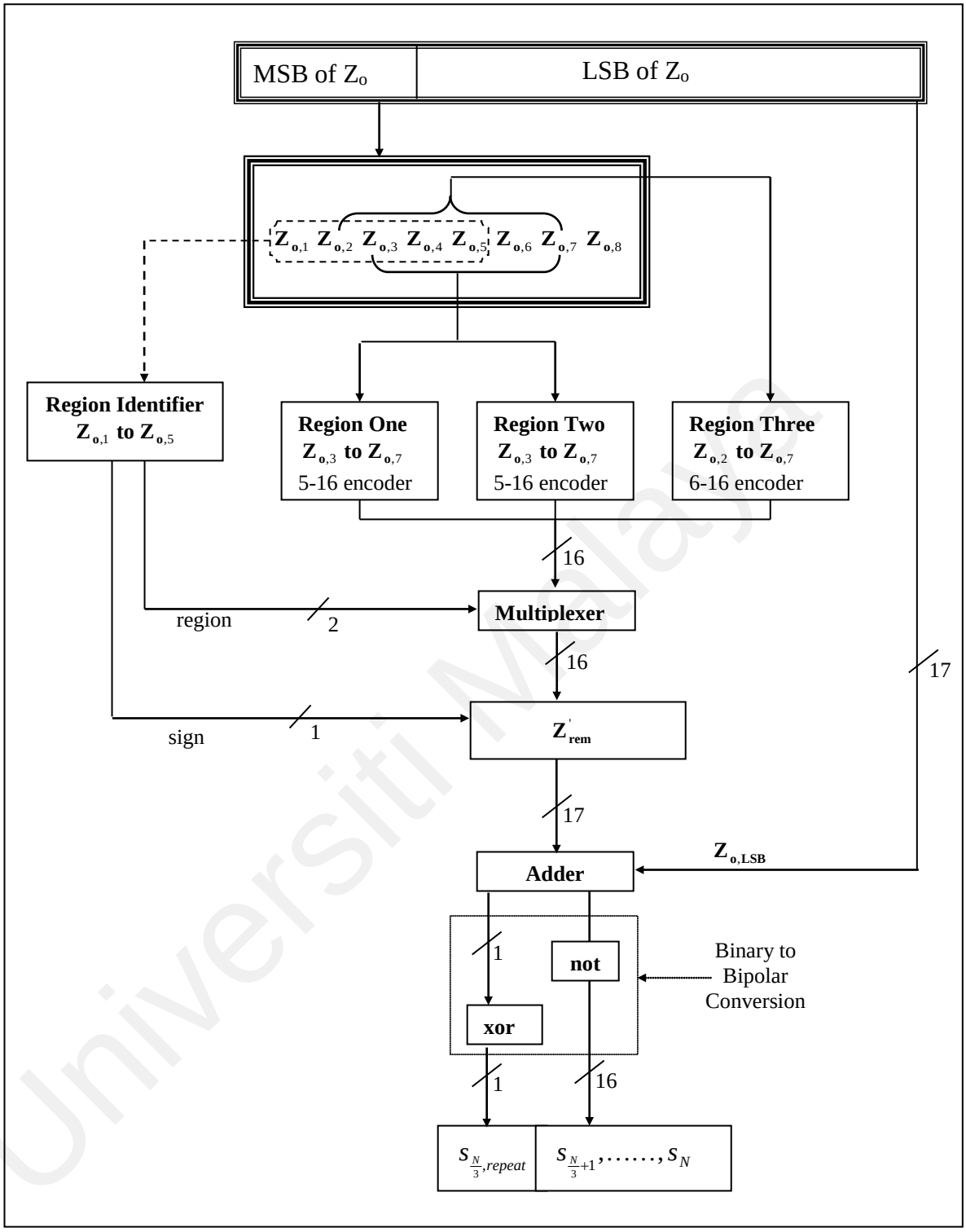

Fig. 4.3 : LSSD Generation for Size N = 24

4.3 Modification Of The SDA

A detailed examination of the SDs for all the possible MSSDs and LSSDs for N-bit input angle however, yielded some interesting results. Compared to the method outlined earlier, which examines $2^{N/3}$ variants of Z_o and the corresponding SDs, a set of 2^N possible values of Z_o were observed, which showed that while the PWL relationship of the MSSDs and $\mathbf{Z}'_{\mathbf{rem}}$ do exist, their changing points do not fall exactly at the places where the MSBs of Z_o transition.

The example of 9-bit Flat CORDIC is considered. For this size, there are no breakpoints. The PWL of the MSSDs and Z'_{rem} are seen in Table 4.3.

Table 4.3: Z'_{rem} for MSBs of Z_o, size N = 9

MSB of Z_o	MSSD	Z'_{rem}	
		Sign	Value
000	011	1	001111
001	011	0	001111
010	010	1	010000
011	010	0	010000
100	001	1	010100
101	001	0	010100
110	000	1	010101

A more detailed examination considers the first N/3 iterations performed on values of Z_o. The MSSDs and the encoder angle $\left(Z_{rem} - Z_{o,LSB}\right)$ are shown in Table 4.4.

Table 4.4: $(Z_{rem} - Z_{o,LSB})$ for MSBs of Z_o for size N = 9

#	Z_o	MSSD	$(Z_{rem} - Z_{o,LSB})$ value		Expected Transition	Actual Transition
			Sign	Value		
0	000_000000	011	1	001111		
63	000_111111	011	1	001111		
64	001_000000	011	0	001111		
111	001_101111	011	0	001111		
112	001_110000	010	0	010000		☼
127	001_111111	010	0	010000		
128	010_000000	010	1	010000	☼	
191	010_111111	010	1	010000		
192	011_000000	010	0	010000		
237	011_101101	010	0	010000		
238	011_101110	001	0	010100		☼
255	011_111111	001	0	010100		
256	100_000000	001	1	010100	☼	
319	100_111111	001	1	010100		
320	101_000000	001	0	010100		
362	101_101010	001	0	010100		
363	101_101011	000	0	010101		☼
383	101_111111	000	0	010101		
384	110_000000	000	1	010101	☼	

Investigations of the SDs of different sizes showed similar results. Both the MSSDs and $(Z_{rem} - Z_{o,LSB})$ transition simultaneously, but not at the places where the MSBs of Z_o change. This implies that the SD generation could not be implemented as simply as with the previously mentioned algorithm.

An alternative method for generating the SDs was still possible, that can also exploit the unique PWL properties of the MSSDs and LSSDs.Using C++, the exact transition points for sizes N = 9, 12, 15, 18, 21 and 24 were identified. These points were used to design comparators for each size, which use the actual transition values to assign the corresponding MSSD and $(Z_{rem} - Z_{o,LSB})$ values. However, use of the comparator increases the size of the design. The results of the comparison between these two methods are given in Chapter 6. Included in the Appendix is the full set of values of SDs at the transition points for N = 24.

CHAPTER 5 : SIGNED DIGIT COMBINATION & PIPELINING

5.1 Signed Digit Combination

Once the SDs have been pre-computed, the next step is to combine them according to the equations (5.1) and (5.2):

$$X_N = \left[1 - \left\{ \begin{array}{l} 2^{-3}\left(\begin{array}{l} \sum s_j\big|_{j+1=3} + \sum s_i \cdot s_j\big|_{i+j=3} - \sum s_j \cdot s_k \cdot s_l\big|_{i+j+k+1=3} - \sum s_i \cdot s_j \cdot s_k \cdot s_l\big|_{i+j+k+l=3} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m+1=3} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n\big|_{i+j+k+l+m+n=3} - \ldots \end{array} \right) \\ + 2^{-4}\left(\begin{array}{l} \sum s_j\big|_{j+1=4} + \sum s_i \cdot s_j\big|_{i+j=4} - \sum s_j \cdot s_k \cdot s_l\big|_{i+j+k+1=4} - \sum s_i \cdot s_j \cdot s_k \cdot s_l\big|_{i+j+k+l=4} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m+1=4} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n\big|_{i+j+k+l+m+n=4} - \ldots \end{array} \right) \\ \ldots\ldots \\ + 2^{-E_N}\left(\begin{array}{l} \sum s_j\big|_{j+1=3} + \sum s_i \cdot s_j\big|_{i+j=E_N} - \sum s_j \cdot s_k \cdot s_l\big|_{i+j+k+1=E_N} - \sum s_i \cdot s_j \cdot s_k \cdot s_l\big|_{i+j+k+l=E_N} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m+1=E_N} + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m \cdot s_n\big|_{i+j+k+l+m+n=E_N} - \ldots \end{array} \right) \end{array} \right\} \right] \tag{5.1}$$

and

$$Y_N = \left[\begin{array}{l} 2^{-1}\left(\begin{array}{l} 1 - \sum s_j \cdot s_k\big|_{j+k+1=1} - \sum s_i \cdot s_j \cdot s_k\big|_{i+j+k=1} + \sum s_j \cdot s_k \cdot s_l \cdot s_m\big|_{j+k+l+m+1=1} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m=1} - \ldots \end{array} \right) \\ + 2^{-2}\left(\begin{array}{l} 1 - \sum s_j \cdot s_k\big|_{j+k+1=2} - \sum s_i \cdot s_j \cdot s_k\big|_{i+j+k=2} + \sum s_j \cdot s_k \cdot s_l \cdot s_m\big|_{j+k+l+m+1=2} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m=2} - \ldots \end{array} \right) \\ \ldots\ldots \\ + 2^{-E_N}\left(\begin{array}{l} 1 - \sum s_j \cdot s_k\big|_{j+k+1=E_N} - \sum s_i \cdot s_j \cdot s_k\big|_{i+j+k=E_N} + \sum s_j \cdot s_k \cdot s_l \cdot s_m\big|_{j+k+l+m+1=E_N} \\ + \sum s_i \cdot s_j \cdot s_k \cdot s_l \cdot s_m\big|_{i+j+k+l+m=E_N} - \ldots \end{array} \right) \end{array} \right] \tag{5.2}$$

Here, the SD combinations are segregated into the channels in which they belong. A channel refers to the positional value of the term (how much it has to shift).

Suppose we consider a particular channel, ch, for X_N. The sum for that channel (*sumx[ch]*) would be the total of the combinational parts of all the terms in the channel. We denote the number of terms in channel *ch* of X_N as *comx[ch]*.

Then, *sumx[ch]* is bounded by the range $|sumx[ch]| <= comx[ch]$. (5.3)

$$X_N = 1 - \sum_{ch=3}^{E_N} sumx[ch] \cdot 2^{-ch} \tag{5.4}$$

After obtaining the sums of each of the channels, these sumx[ch] values are shifted to the right (*ch* times) to obtain $sumx[ch] \cdot 2^{-ch}$. Then, all these values are added together.

1) This method requires adders to get the sum for each channel. The size of these adders have to be able to accommodate the range $-comx[ch] < sumx[ch] < comx[ch]$
2) This method requires shifters to position the sums of the respective channels
3) Since the desired accuracy is E_N, the size of the registers containing the value of $sumx[ch] \cdot 2^{-ch}$ has to be E_N bits

5.2 Ripple Method

This shifting and addition process could be made much simpler if *sumx[ch]* could be restricted such that $|sumx[ch]| \leq 1$ for channels 1 through E_N. (ie. *sumx[ch]* = 0,±1, which is represented with 2 bits).

This can be achieved by rippling the values of sumx[ch] from the channel with the higher negative indices to the channels with the lower negative indices. The concept of this can be seen in Eqs.. (5.5) and (5.6).

$$term1 \cdot 2^{-ch} + term2 \cdot 2^{-ch} = (term1 + term2) \cdot 2^{-ch} \tag{5.5}$$

$$= \pm 2 \cdot 2^{-ch} = \pm 1 \cdot 2^{-(ch-1)} \qquad where\ term1 = term2 = \pm 1 \quad \text{Case 1}$$

or, $= 0 \cdot 2^{-ch} = 0$ *where* $term1 = -(term2)$ Case 2

For Case 1, the ripple into channel (*ch-1*) is ±1, and in Case 2, the ripple is 0. A VLSI-efficient implementation of the combination and addition process has been realized based on this concept. Basically, the entire equation is represented in a grid of cells. The **columns** represent the channel sums sumx[ch], and the **rows** are the layers. The cells are filled in the following manner:

1) The first row values are filled first, their ranges bound by $|comx[ch]|$
2) The cells are then filled vertically, from top to bottom, and from the Least Significant Channel (LSC, or channel with value 2^{-E_N}) to the Most Significant Channel (MSC)

For the first row (Layer 1), the values for each channel, *ch*, are simply bound by the range of *comx[ch]*. The values of *comx[ch]* are divided by 2, and the result of the division repples into the next channel, **cell**[*ch-1, Layer 2*]. The remainder drops down the same channel to **cell**[*ch, Layer 2*]. This is illustrated in Fig. 5.1.

For each column, the aim is to terminate the layer as quickly as possible, the moment when the range of sumx[ch] is restricted by $|sumx[ch]| \leq 1$. For any particular cell, for example **cell**[*ch, Layer L*], the two inputs into it are the previous sum from **cell**[*ch, Layer L-1*], and the ripple from **cell**[*ch+1, Layer L-1*]. The total of these two inputs is divided by 2 as before, the ripple traveling to the next channel, and the remainder to the next layer.

If the total input into that given cell is (0, ±1), and there are no ripples in at any lower layer, the channel is terminated. The termination layer is TL_{ch}.

Example :

Take the case of N = 9, and $E_N = 12$

Following the Signed Digit Generation as outlined in Chapter 4 produces 10 SDs (s_1 through s_{10}).

$$X_9 = 1 + 2^{-3}(-s_2) + 2^{-4}(-s_3) + 2^{-5}(-s_4 - s_2 s_3) + 2^{-6}(-s_5 - s_2 s_4) + 2^{-7}(-s_6 - s_2 s_5 - s_3 s_4) + 2^{-8}(-s_7 - s_2 s_6 - s_3 s_5) + 2^{-9}(-s_8 - s_2 s_7 - s_3 s_6 - s_4 s_5) + 2^{-10}(-s_9 - s_2 s_8 - s_3 s_7 - s_4 s_6 + s_2 s_3 s_4) + 2^{-10}(-s_{10} - s_2 s_9 - s_3 s_8 - s_4 s_7 - s_5 s_6 + s_2 s_3 s_5) + 2^{-10}(-s_2 s_{10} - s_3 s_9 - s_4 s_8 - s_5 s_7 + s_2 s_3 s_6 + s_2 s_4 s_5) \quad (5.6)$$

For the case of θ = 30°, Zo = 100001100 (0.5234375 radians)

The SDs are : 1 1 -1 -1 1 -1 -1 -1 -1 1

Table 5.1 Example of Ripple for N = 9

Ch	0	1	2	3	4	5	6	7	8	9	10	11	12
Comx[ch]	1	0	0	1	1	2	2	3	3	4	5	6	6
Sumx[ch]·2^{-ch} LAYER 1	2^{-0}	0	0	-2^{-3}	2^{-4}	$2\cdot2^{-5}$ $=2^{-5}$	$0\cdot2^{-6}$	-2^{-7}	$3\cdot2^{-8}$ $=2^{-8}$ $+2^{-7}$	$2\cdot2^{-9}$ $=2^{-8}$	$1\cdot2^{-10}$	$2\cdot2^{-11}$ $=2^{-10}$	$-2\cdot2^{-12}$ $=-2^{-11}+$ $0\cdot2^{-12}$
LAYER 2 previous sum + ripple	$1\cdot2^{-0}$ TL_0	$0\cdot2^{-1}$ TL_1	$0\cdot2^{-2}$ TL_2	$-1\cdot2^{-3}$ TL_3	$1\cdot2^{-4}$ TL_4	$1\cdot2^{-5}$ TL_5	$0\cdot2^{-6}$ TL_6	$-1\cdot2^{-7}$	2^{-8} $+2^{-8}$ $=2^{-7}$	$0\cdot2^{-9}$	2^{-10} $+2^{-10}$ $=2^{-9}$	$0\cdot2^{-11}$ -2^{-11} $=-2^{-11}$	$0\cdot2^{-12}$ TL_{12}
LAYER 3 previous sum + ripple								$-1\cdot2^{-7}$ $+2^{-7}$ $=0$	$0\cdot2^{-8}$ TL_8	$1\cdot2^{-9}$ TL_9	$0\cdot2^{-10}$ TL_{10}	-2^{-11} TL_{11}	
LAYER 4 previous sum + ripple								$0\cdot2^{-7}$ TL_7					

5.3 Implementation of The Ripple Method

For the implementation of this method, modules are designed to fit into the cells. For any given cell, the module is chosen based on the number of inputs into the cell. An extra simplification to the combination of terms described above is given in Section 5.3.2.

5.3.1 Layer One

For the first layer, the inputs are the combinational terms of the SDs, which are ±1. For each pair of inputs, one instance of module PAIR is created, that follows Equation 5.5.

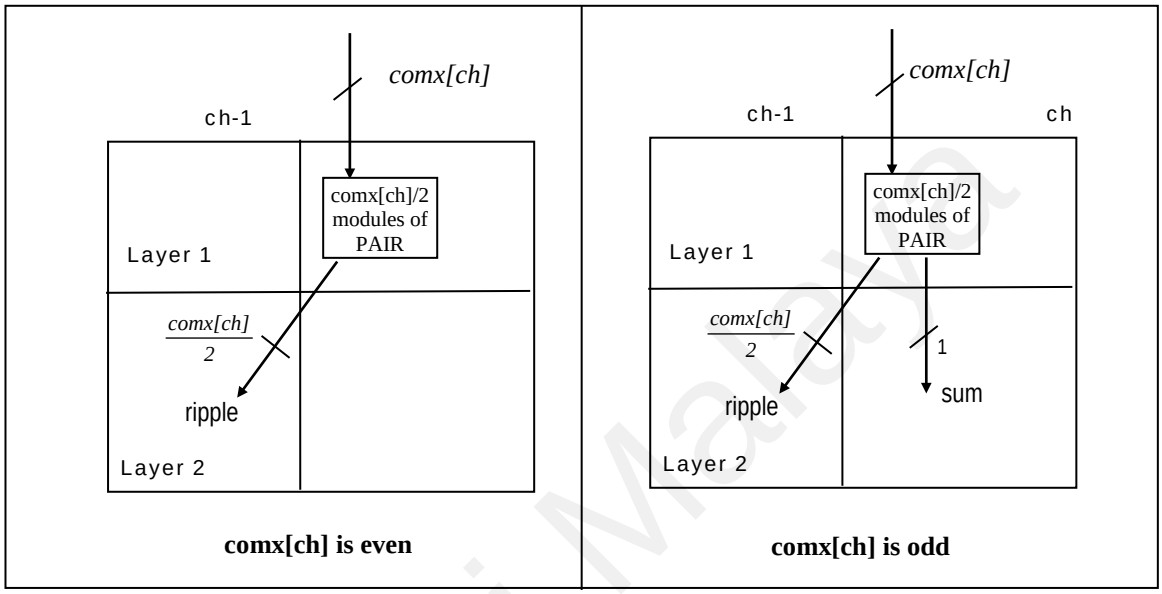

Figure 5.1 : Module PAIR for Layer One

Each instance of Module PAIR generates the ripple as shown in Table 5.2.

Table 5.2: Inputs and Output of Module PAIR

in 1	in 2	Ripple
+1	+1	**+1**
+1	-1	**0**
-1	-1	**-1**
-1	+1	**0**

5.3.2 Remaining Layer Modules

For all the rest of the cells, they are filled by considering the total number of inputs into the cells. These are divided into the following categories :

Case 1: Number of Inputs ≥ 3

An extra simplification of the combination of terms is incorporated here, by combining the terms in groups of three.

$$term1 \cdot 2^{-ch} + term2 \cdot 2^{-ch} + term3 \cdot 2^{-ch} = (term1 + term2 + term3) \cdot 2^{-ch}$$

$$= (0,\pm 1)2^{-(ch-1)} + (0,\pm 1)2^{-ch} \qquad (5.7)$$

Assuming the number of inputs into the cell is 3g + h, where g ≥ 1, and h = 1,2. Here, (g-1) sets of module blue_0 are generated, and one of either blue_1 or blue_2, depending on the value of h.

blue_0 takes in 3 inputs and produces one ripple value for cell[*ch-1,layer+1*] and one sum value for cell[*ch,layer+1*]

blue_1 and **blue_2** take in 4 and 5 inputs respectively. 3 of them are used to produce one ripple value and one sum value. The extra inputs bypass the cell, and go into the cell directly below : **cell**[*ch,layer+1*].

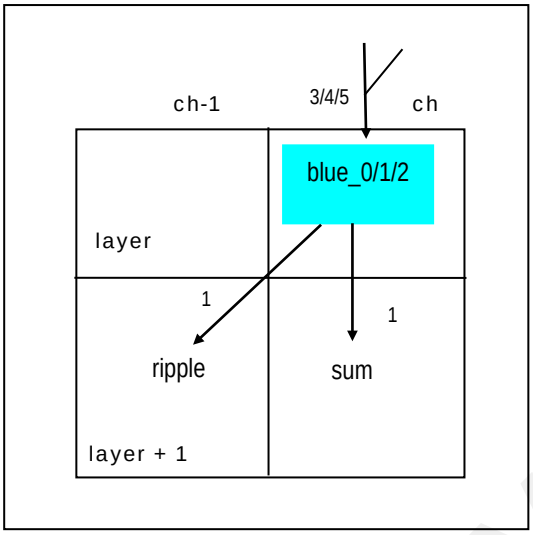

Figure 5.2 Module BLUE

This design is implemented onto the FPGA using case statements.

Table 5.3 : Inputs and output of Module BLUE

in 1	in 2	in 3	total	ripple	sum
-1	-1	-1	-3	**-1**	**-1**
-1	-1	0	-2	**-1**	**0**
-1	0	-1	-2	**-1**	**0**
-1	0	0	-1	**0**	**-1**
0	-1	-1	-2	**-1**	**0**
0	-1	0	-1	**0**	**-1**
0	0	-1	-1	**0**	**-1**
0	0	0	0	**0**	**0**
0	0	1	1	**0**	**1**
0	1	0	1	**0**	**1**
0	1	1	2	**1**	**0**
1	0	0	1	**0**	**1**
1	0	1	2	**1**	**0**
1	1	0	2	**1**	**0**
1	1	1	3	**1**	**1**
1	1	-1	1	**0**	**1**
1	-1	1	1	**0**	**1**
1	-1	-1	-1	**0**	**-1**
-1	1	1	1	**0**	**1**
-1	1	-1	-1	**0**	**-1**
-1	-1	1	-1	**0**	**-1**

+1 is coded in binary as “01”, -1 as “10” and 0 as “00”

Case 2: Number of Inputs = 2

When the number of inputs is 2, the lower layers are checked to see if there are any more ripples coming into the channel. If there are, then the inputs are just passed through to the subsequent layer (GREEN module). If not, the inputs are added, to get a ripple value for **cell**[*ch-1*,*layer+1*] and, and a sum value for **cell**[*ch*,*layer+1*] (ORANGE module).

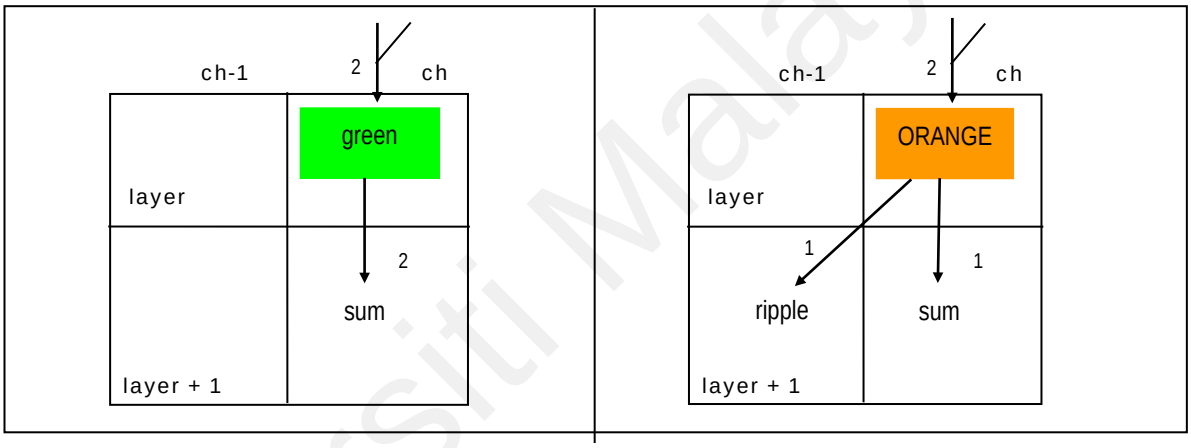

Figure 5.3 : Modules GREEN and ORANGE

Table 5.4 : Inputs and output of Module ORANGE

in 1	in 2	total	ripple	sum
-1	-1	-2	**-1**	**0**
-1	1	0	**0**	**0**
-1	0	-1	**0**	**-1**
0	0	0	**0**	**-1**
0	-1	-1	**0**	**-1**
0	1	1	**0**	**1**
1	-1	0	**0**	**0**
1	1	2	**1**	**0**
1	0	1	**0**	**1**

+1 is coded in binary as “01”, -1 as “10” and 0 as “00”

Case 3 : Number of Inputs =1

When the number of inputs is 1, a check is performed on the lower layers to see if there are any more ripples coming into the channel. If there are, then the inputs are just passed through to the subsequent layer. If not, the channel is terminated.

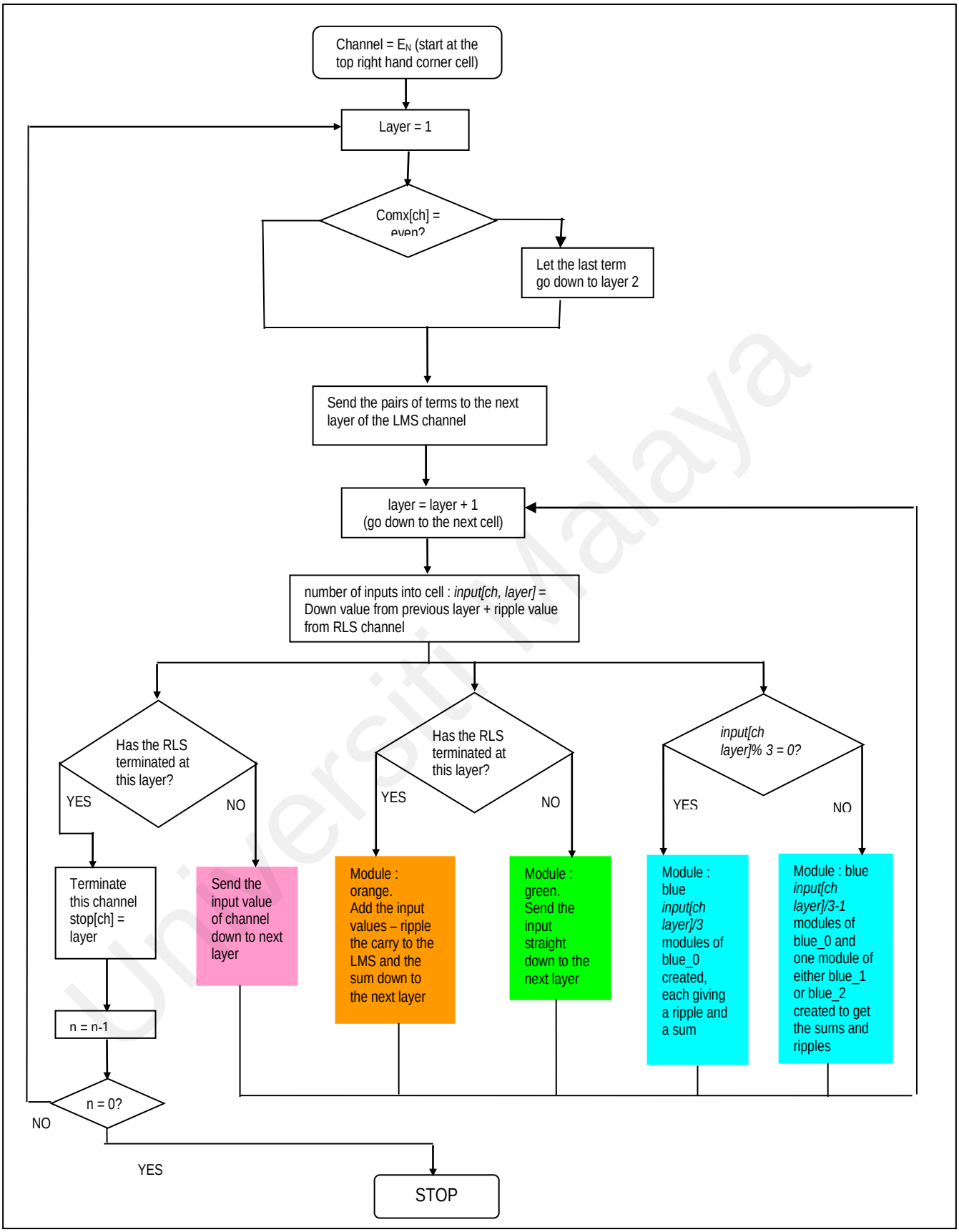

Figure 5.4 Flowchart for Ripple Method, size N

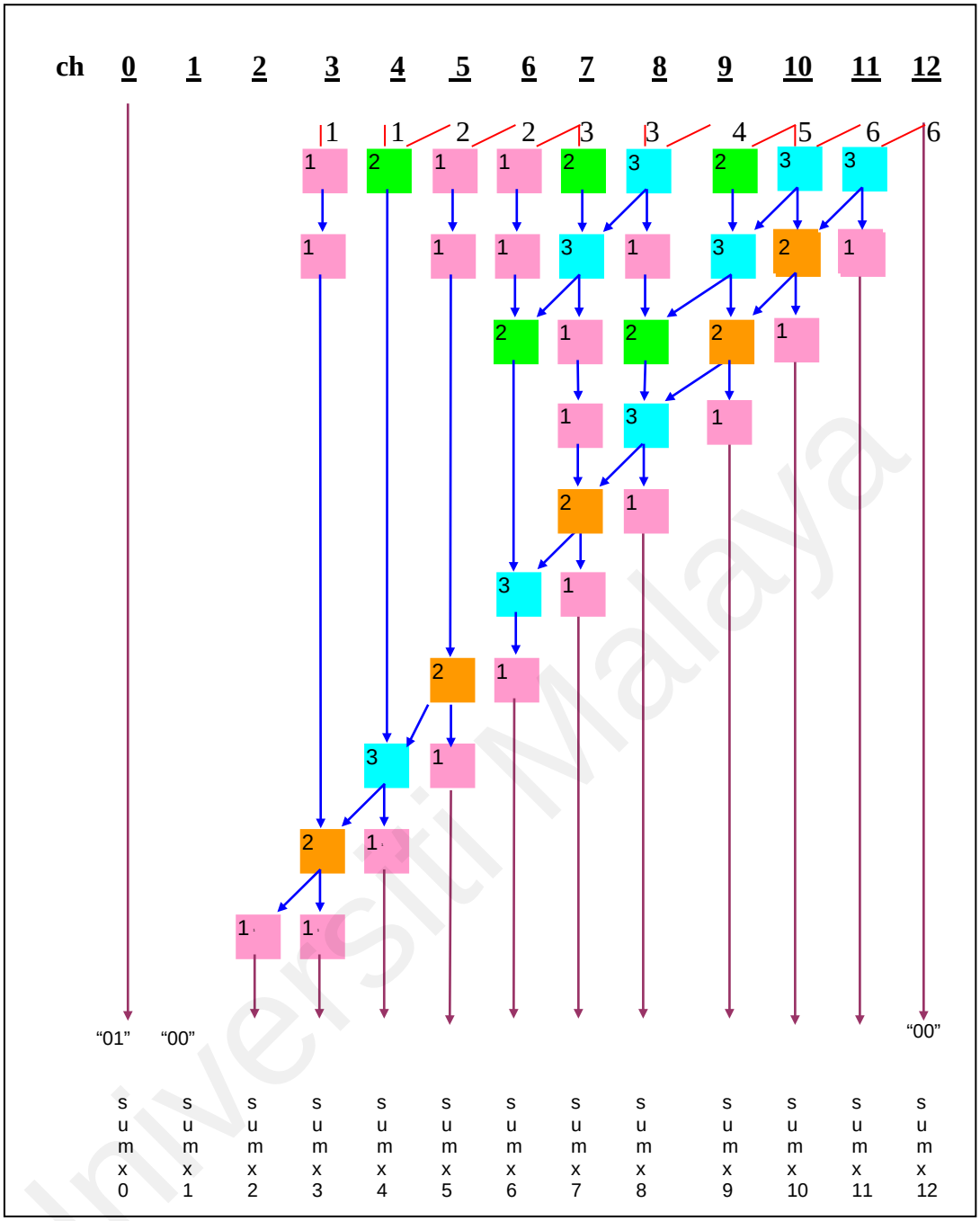

Figure 5.5 Implementation of Ripple Method for N = 9

We now have 13 sets of 2-bit output values. If we designate 13-bit variables : answer_0 which is [sumx0(0) to sumx12(0)], and answer_1 which is [sumx0(1) to sumx12(1)], then $X_N = answer\_0 - answer\_1$

5.4 Pipelining Flat CORDIC

As the size N gets bigger, the total number of combinations increases sharply. This is shown in Figure 3.2. The result is a huge increase in the number of modules generated. This increase in combinational circuitry causes the combinatorial path delay to increase, thereby reducing the maximum operating frequency of the circuit.

A method that can be used to boost the frequency is to pipeline the design. The circuit is divided into two or more sections (depending on the number of pipeline stages), and each part is performed on a separate clock cycle. The output of each stage is sent to the input of the next stage through a register. The matrix structure of the design lends itself particularly well to pipelining. The pipeline stages are drawn horizontally across the diagram.

At the transition point from one stage to the next, a new module/VHDL file is created. The vertical lines that cut across from one stage to the next become the input lines feeding into the next stage. These inputs are tied to the clock signal.

For the 2-stage pipeline, the maximum increase in frequency is double the original value. This is assuming the division of the stages equally divides the entire design into two equal halves. Additional pipeline stages can potentially increase the frequency many times more, but at the cost of the latency of the circuit. With the exception of the first additional stage, which increases the latency by 2 cycles, each additional stage increases the latency

by one clock cycle. Pipelining is only a worthwhile measure if the critical path of the circuit is affected by this combinational delay.

A 2-stage pipeline implementation for size N = 9 is shown in Fig. 5.6.

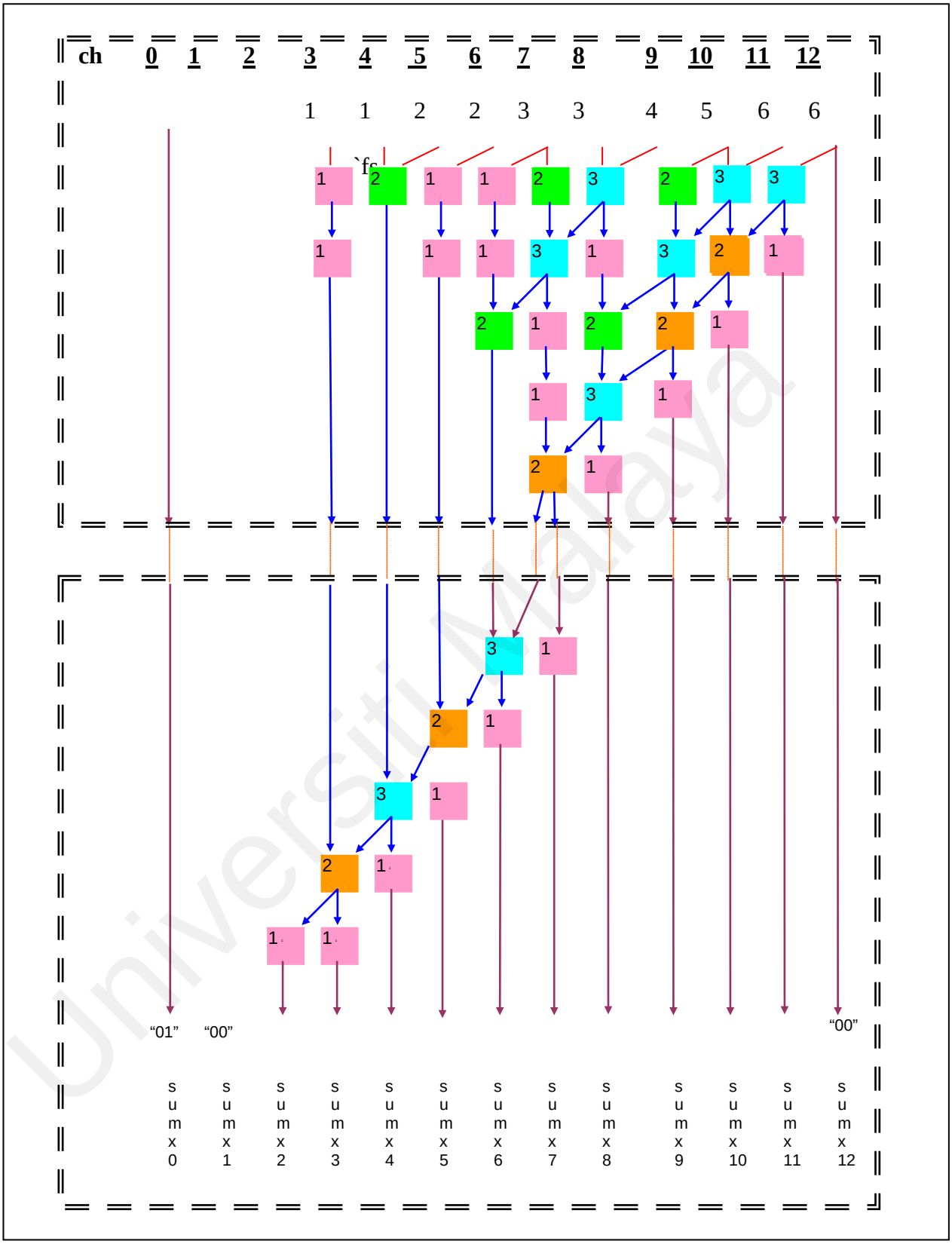

Figure 5.6 Two Stage Pipeline for Ripple Method

5.5 Scaling of Final Values

Each rotation in CORDIC, and by extension in Flat CORDIC as well, is not a pure-rotation, but a rotation-extension. The scaling factor is roughly the same. For each bit-size that was tested, a thorough analysis was performed using C++ to determine the precise scaling factor values. These values were then incorporated into the Flat CORDIC architectures with fixed multipliers, of length $E_N = N + \log_2 N$ for bit size N.

5.6 Field Programmable Gate Arrays (FPGAs)

The advent of VLSI technology has made it possible to produce high-density programmable logic devices, and resulted in the increased popularity of Field Programmable Gate Arrays (FPGAs). Huge amounts of logic, up to hundreds of thousands of gates can be fitted onto single devices. FPGAs are now even taking over ASICs in high performance applications due to the additional flexibility in design time and design upgrades that can be performed without hardware replacement.

An FPGA is generally made up of a matrix of cells arranged in rows and columns. The cells are interconnected via programmable elements. These elements also connect the cells to the Input/Output blocks (IOs). The logic cells and exact routing designs differ from one system to another.

There are 2 main technologies used for FPGA – SRAM and antifuse. SRAM (SRAM FPGAs are generally less dense, because the physical dimensions are an order of magnitude larger than the antifuse ones. However, due to larger chip sizes, SRAM

FPGAs hold more gates. These FPGAs also have unlimitted reprogrammability unlike antifuse FPGAs, since the interconnect elements are not physically altered during the programming. For this reason, SRAM FPGAs are volatile, and need to have the design loaded into them via external PROMs on startup. Antifuse FPGAs on the other hand, are One Time Programmable (OTP) devices.

The SPARTAN II, 2.5V FPGAs used in this work, are a low cost family of devices that contain abundant logic resources, ranging from 15,000 gates in the XC2S15 to 200,000 gates in the XC2S200. The matrix elements are Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable IOBs. There are 4 Delay Lock Loops (DLLs), one on each corner of the die, and 2 columns of block RAM on opposite sides of the die. Webs of versatile routing channels run through the matrix connecting these elements.

5.6.1 Spartan II CLB

CLBs are the basic building blocks of the FPGA. The Spartan II CLB contains 4 Logic Cells (LCs) arrainged in pairs on 2 slices. Each LC contains a 4-input function generator carry logic and a storage element.

Function Generator

The function generator in the LC is basically a 4-input Look Up Table (LUT) that can double up to provide 16-bit synchronous RAM. The 2 LUTs in a slice can be used to create

a 16x2-bit or a 32x1-bit synchronous RAM. The LUT can also be used as a 16-bit shift register, or alternatively, to store data.

Storage Element

The storage element in each LC can either be used as a level sensitive latch or an edge-triggered D flip-flop. When used as a D flip-flop, the input to the flip-flop can either be driven by the fuction generator or directly from the inputs into the slices.

Additional Logic

Each pair of LCs additionally has an extra multiplexer that can be used in combination with both the function generator outputs of one slice to produce a 5-input function generator, or with both the slice outputs to generate a 9-input function. The Spartan II CLB also has dedicated carry logic that enables high-speed arithmetic functions to be carried out. Fig. 5.7 illustrates a Spartan II CLB Slice.

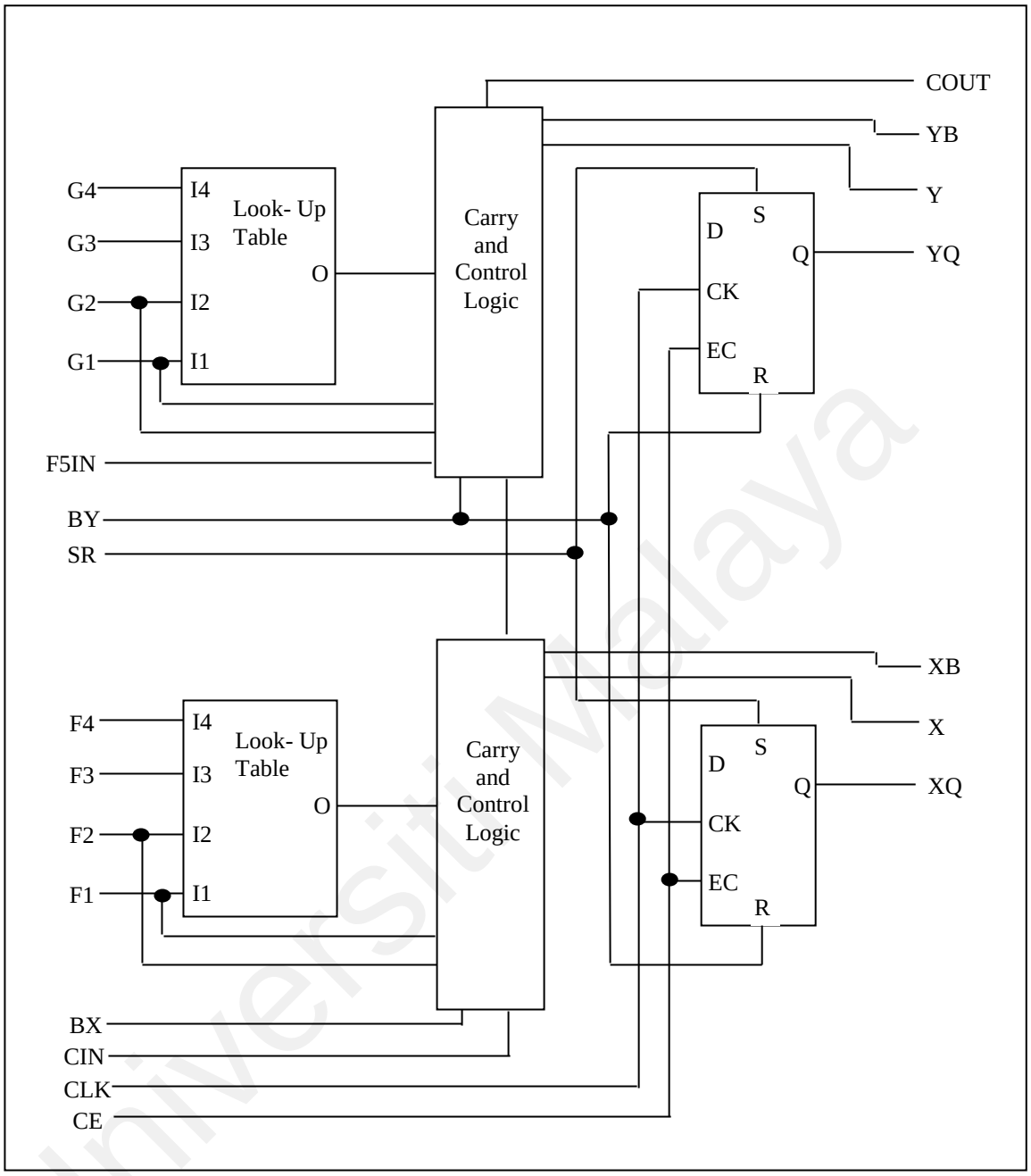

Fig. 5.7 Spartan II CLB Slice

Chapter 6 : Results & Discussion

6.1 Introduction

This chapter focuses on the results of Flat CORDIC simulation and synthesis onto Spartan FPGAs. The simulations were performed using ModelSim XE v5.6e and the resultant binary data files compared against expected results from corresponding data files generated using C++.

The first part concentrates on Flat CORDIC size and speed results. The increase in circuit size and drop in frequency for increasing values of N is observed. The next part focuses on the SD Generation. A comparison is made between the SD Algorithm as proposed by Bimal (2000) which uses an encoder, with the newly designed comparator method. The impact of these two designs as part of the entire Flat CORDIC structure is also considered.

The combination of all the generated SDs and their subsequent addition is examined in detail. The specially designed instances that make up the backbone of the design are shown. The implementation of the Combination and Addition (C&A) section is seen in terms of the number of gates used, and the speed of operation. Ideal pipeline results for additional number of stages are examined, and the implementation results of one extra pipeline stage are shown.

The C&A part produces output values that are scaled using multipliers. The size of the multipliers as part of the entire architecture is shown. Finally, a comparison is

made between the Flat CORDIC design and two other sine/cosine generation implementations on FPGA. The computer processor internal sine/cosine calculation speed as carried out using C++ is also compared with the Flat CORDIC speed.

6.2 Flat CORDIC on FPGA

The Flat CORDIC module takes 4 clock cycles to produce the sine/cosine values of a given input angle.

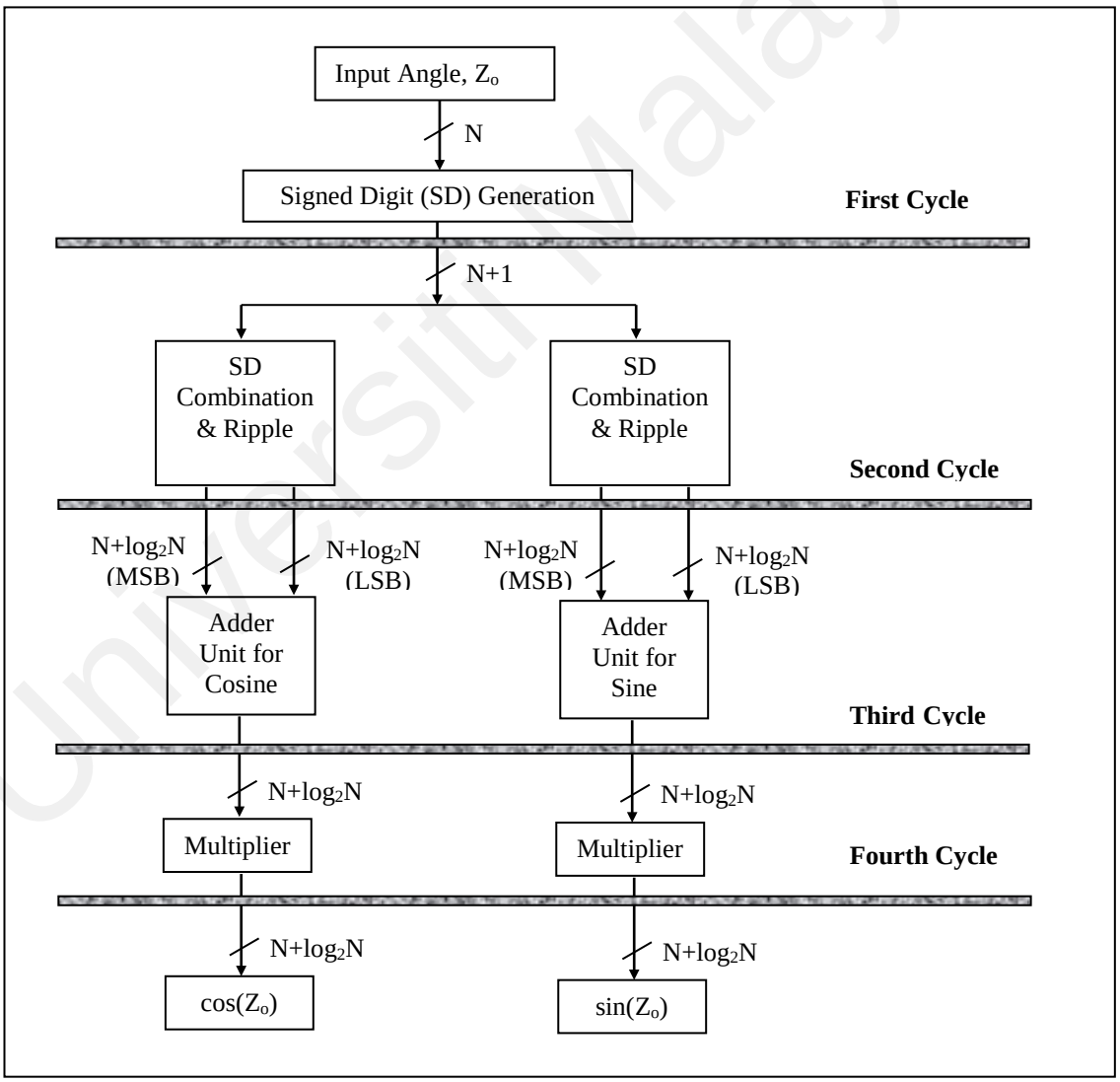

Figure 6.1 Flat CORDIC Pipeline Stages

Table 6.1 and Figs. 6.2 and 6.3 show the overall circuit size and speed.

Table 6.1: Flat CORDIC Speed and Gate Count

Size (bits)	Max Frequency (MHz)	Total eq. gate count
9	30.58	6,238
12	20.58	11,325
15	15.29	21,389
18	13.32	37,281
21	11.34	61,669
24	9.94	104,690

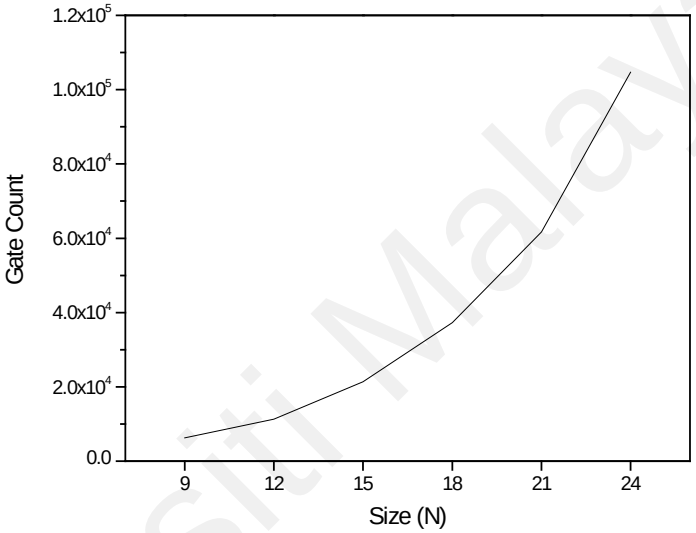

Figure 6.2 Graph Flat CORDIC Gate Count Versus Size (N)

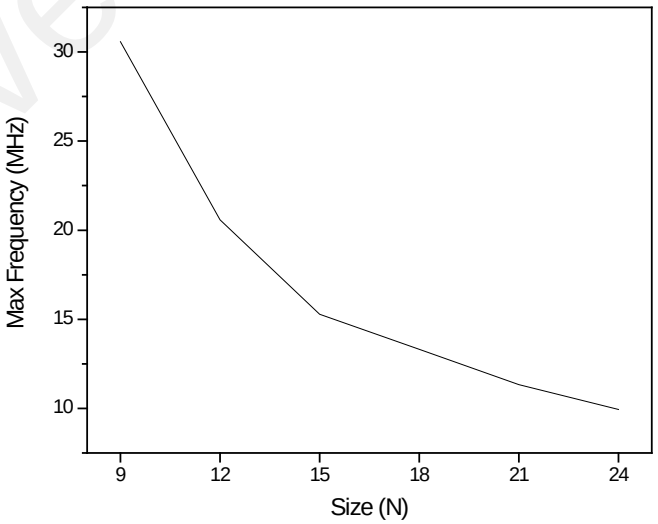

Figure 6.3 Graph Flat CORDIC Maximum Operating Frequency Versus Size (N)

Due to the increased complexity, the maximum operating frequency reduces with N. The size of the design increases in a non-linear manner with N. This is because the number of terms in the Flat CORDIC equation, the number of SDs to generate, and the size of the adders and multipliers increase considerably with N.

6.3 Signed Digit Generation

This section deals with a comparison between the two methods of generating the Signed Digits (SDs).

6.3.1 Signed Digit Algorithm (SDA) – Encoder Method (Bimal)

In the SDA, a simple algorithm is used to generate the MSSDs and encoders for the remaining angle generation. As outlined in Chapter 4, the MSSD transition points are taken as corresponding to the input MSB changes. For each of the regions identified by the break in the PWL pattern, the relationship between the MSBs and the MSSDs is used to incorporate simple shift-addition operation to directly obtain the MSSDs. Depending on the total number of variations in each region, special encoders are used to generte the remaining angle, which is then processed to obtain the LSSDs.

6.3.2 Signed Digit Algorithm – Comparator Method

Using C++, the exact transition points of the MSSDs and remaining angles for the entire range of N-bit values are examined. It was verified that both the MSSDs as well as the remaining angles transition at exactly the same points. The transition points were used

to create the comparators that simultaneously generate both the MSSDs and the remaining angle values.

Included in the Appendix is the detailed data of the input angles and the transition points, along with the corresponding MSSDs and remaining angles for 24-bit Flat CORDIC. The highlighted portions indicate the transition points.

Table 6.2: Changes in Gate Count and Speed for SD Generation with Size N)

Size	Frequency (MHz)		Gate Count	
	Encoder	**Comparator**	**Encoder**	**Comparator**
9	62.6	83.7	180	309
12	60.4	69.7	228	537
15	32.1	52.8	562	1,249
18	26.1	51.0	789	4,714
21	26.7	47.0	1,048	10,208
24	22.4	40.9	1,736	23,418

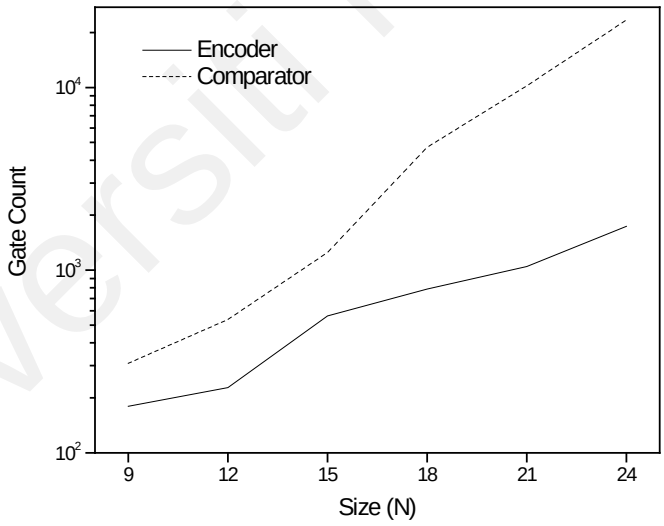

Figure 6.4 Graph SD Generation Gate Count Versus Size (N)

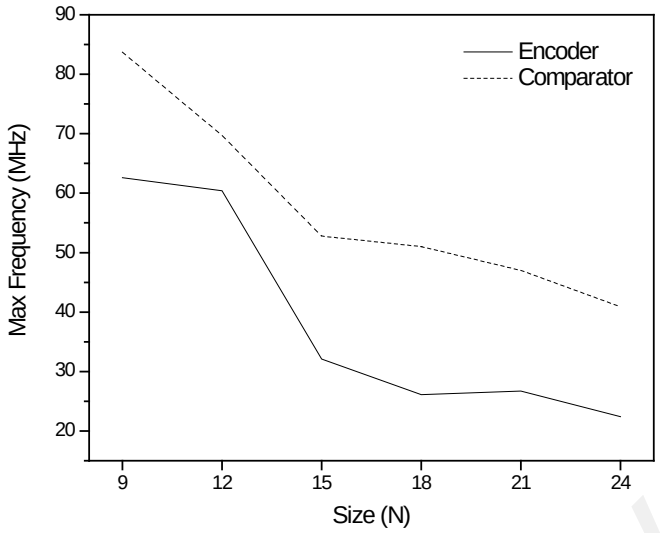

Figure 6.6 Graph SD Generation Latency Versus Size (N)

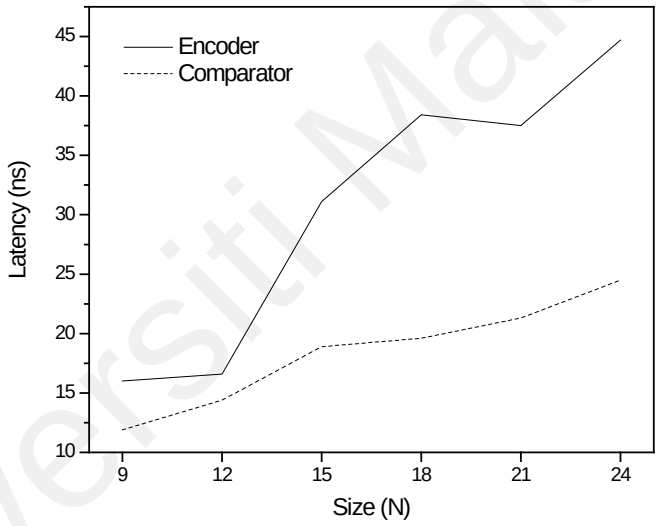

Figure 6.5 Graph SD Generation Maximum Operating Frequency Versus Size (N)

Figure 6.4 gives an estimate of the size of SD generation using the two methods. Even for smaller values of N, the comparator method takes up almost twice the area of the encoder method. As the values of N increase, this difference increases dramatically, with the comparator taking almost 12 times as many gates as the encoder for size N = 24 bits.

The results of Figure 6.5 and 6.6 show that the SD generation using the comparator method is faster compared with the encoder, using roughly 2/3 the total amount of time

to generate all the SDs. The benefits of this improvement in time will be seen in the final design only if the SD generation lies in the critical path of the design.

6.3.3 Comparison of Design

The two different SD Generation modules were incorporated into the full Flat CORDIC design and implemented. Table 6.3 shows the maximum frequencies obtainable for the designs, and the total number of gates.

Table 6.3: Changes in Gate Count and Speed with Size (N)

Size (bits)	Max Frequency (MHz)		Gate Count	
	Encoder	Comparator	Encoder	Comparator
9	30.4	30.6	6,217	6,238
12	24.0	20.6	10,010	11,325
15	15.3	15.3	21,388	21,839
18	13.2	13.3	33,739	37,281
21	11.3	11.3	57,235	61,669
24	10.1	10.0	85,182	104,690

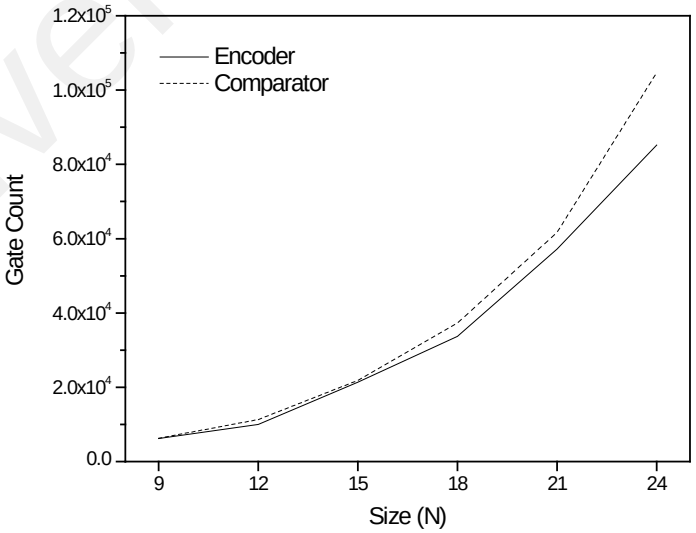

Figure 6.7 Graph Flat CORDIC Gate Count Versus Size (N) For Different SD Generation Methods

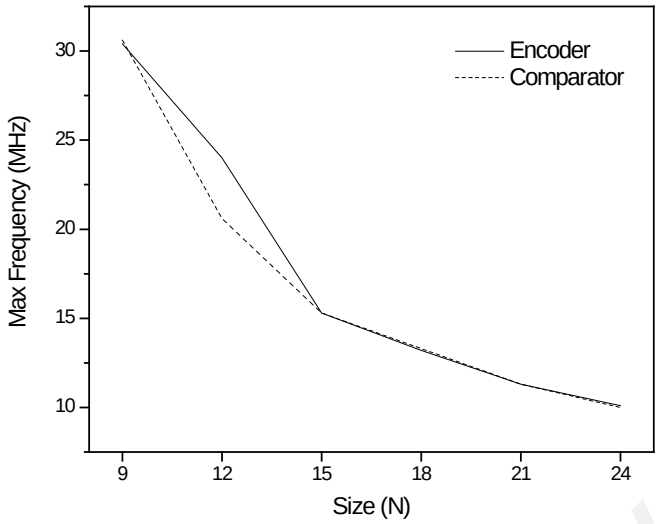

Figure 6.8 Graph Flat CORDIC Maximum Frequency Versus Size (N) For Different SD Generation Methods

Section 6.3.1 showed that the gate count of the Comparator Method is significantly bigger than that of the Encoder Method. Fig. 6.7, however, indicates that once incorporated into the Flat CORDIC architecture, the difference ceased to be so significant.

Fig. 6.8 shows that for the most part, once incorporated into the fuill Flat CORDIC architecture, the speed benefits of the Comparator Method are no longer seen. This is due to the fact that the SD Generation Module is not in the critical path.

6.4 Combination of SDs and Addition

The combination of all the SDs and their routing into the respective channels, followed by the implementation of the Ripple Method makes up the bulk of the Flat CORDIC module. This design is almost entirely combinatorial. The signals have to go through many instances of three specially designed modules, as explained in detail in Chapter 5. These three modules are referred to as PAIR, BLUE and ORANGE

Module PAIR takes in pairs of SDs (each being a 1-bit value representing ±1), and returns a carry to the next channel. It requires 2 sets of 4-input LUTs, and occupies 1 slice (0.5 CLB). The number of gates required is 12.

Module BLUE takes in three SDs (each being a 1-bit value representing ±1), and returns a carry to the next channel, and a sum. It requires 11 sets of 4-input LUTs, and occupies 6 slices (3 CLBs). The number of gates required is 66.

Module ORANGE takes in two SDs (each being a 1-bit value representing ±1), and returns a carry to the next channel, and a sum. It requires 4 sets of 4-input LUTs , and occupies 2 slices (1 CLB). The number of gates required is 24.

The regular design of the Combination and Addition (C&A) module takes two clock cycles. The first cycle combines the SDs, goes through the channels (and all the instances of the above-mentioned modules) and provides the input to the final adder ($N+\log_2 N$ bits). In the next cycle, the addition is performed. From the Flat CORDIC Equation, the result of the X-channels gives the unscaled cosine of the input angle, and the result of the Y-channels gives the unscaled sine value.

Table 6.4 and Fig. 6.9 show the increase in the size of the design and also the maximum frequency for the X- and Y- parts respectively:

Table 6.4: Maximum Frequency and Gate Count for C&A Module

Size (bits)	Speed (MHz)		Number of gates	
	Cosine (X)	Sine (Y)	Cosine (X)	Sine (Y)
9	31.6	30.1	1,648	1,411
12	22.0	21.4	3,706	4,181
15	16.0	17.8	9,368	8,877
18	13.9	13.6	12,597	12,523
21	12.2	11.8	22,789	23,225
24	10.8	10.4	38,745	39,359

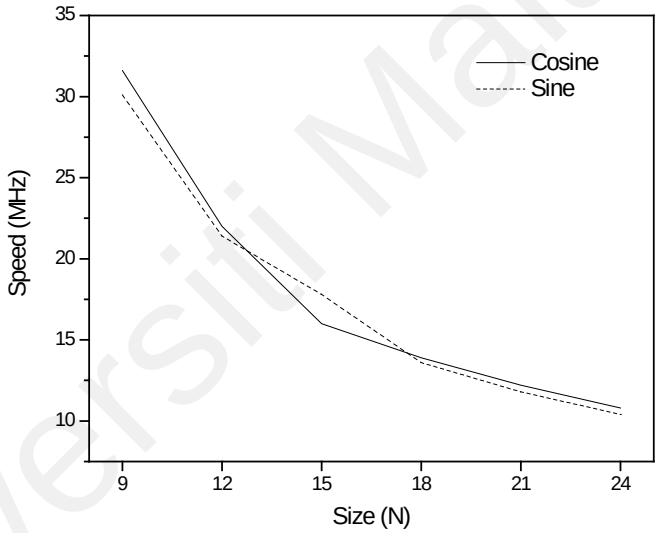

Figure 6.9 Graph Maximum Frequency Versus Size (N) for C&A Module

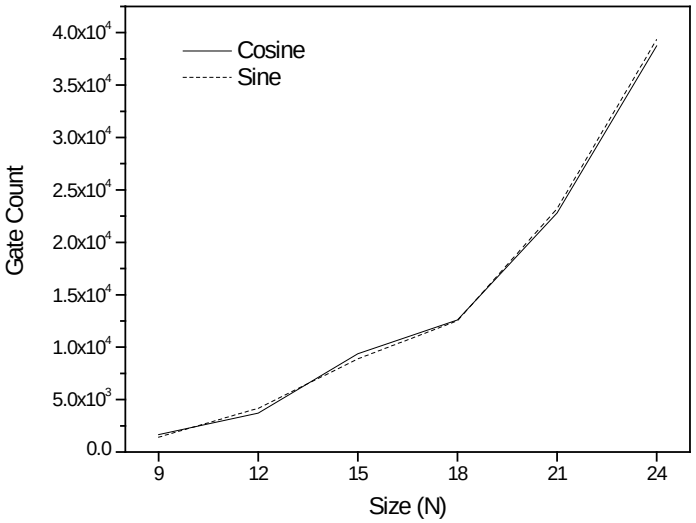

Figure 6.10 Graph Gate Count Versus Size (N) for C&A Module

Fig. 6.9 indicates that the speed of the sine/cosine calculation drops significantly as the size (N) increases. Fig. 6.10 shows that the gate count for both the sine as well as the cosine generation increases significantly with N. These results coincide with the non-linear increase in the number of terms for the existing as well as additional channels when N increases. Fig. 6.10 also indicates that the number of gates required for sine and cosine calculation is almost the same.

The C&A Module to generate the pre-scaled sine/cosine values is seen as a percentage of the overall Flat CORDIC architecture in the table below:

Table 6.5: C&A Module as Part of Flat CORDIC Architecture

Size (bits)	C&A Gate Count	Flat CORDIC Gate Count	Percentage of whole (%)
9	3,059	6,238	49
12	7,887	11,325	70
15	18,245	21,839	84
18	25,120	37,281	67
21	46,014	61,669	75
24	78,104	104,690	75

Even at its smallest, this part still takes up almost half the entire design, for bit size N = 9. This is when the multipliers for the final scaling are at a comparable size. For the rest of the designs, the C&A Module takes up a higher portion of the architecture.

6.4.1 Pipelining The C&A Section

It was indicated in Section 6.1 and later shown that the maximum operating frequency and therefore the minimum time to obtain results is determined during the SD Combination cycle. The second longest delay path is in the SD Generation cycle. This uneven distribution of logic per cycle can be balanced out through the addition of

pipeline stages to split up the critical path logic to be carried out over extra cycles. With a smaller amount of logic per cycle, the critical path delay can be considerably reduced. The organized structure of the C&A section lends itself particularly well to pipelining. The structure can be divided into sections which each run on separate clock cycles. As illustrated in Chapter 5, horizontal lines are drawn across the matrix structure to separate the pipeline sections.

Assume that the time taken for the entire combination and ripple process is T_{CR}. Taken into consideration are the setup and hold times of the FPGA. The setup time, T_S, is the time relative to a clock event during which the data input to a latch or flip-flop must remain stable in order to guarantee that the latched data is correct. The hold time, T_H, is the time following a clock event during which the data input remains stable for the same reason. SPARTAN FPGAs have a T_S value of 5 ns, and T_H value of 0 ns.

Each additional pipeline stage increases the circuit latency by 1 clock cycle (except for the first stage, as explained later). Therefore, in a given module, each extra stage can potentially cut the delay by close to half (exact 50% reduction is not possible due to non-zero hold time). The optimum place to set up the pipeline is at a point that divides the entire circuit into equal parts, and each extra stage is added until the module no longer lies in the critical path of the entire design.

From the data, the second biggest delay comes from the SD Generation comparator section, T_{SDG}. The following table shows the potential improvement in circuit latency for additional pipeline stages. These are ideal figures, assuming the divisions occur exactly at the right places. The number of stages are added until $T_{CR,P}$ is less than or equal to T_{SDG}, where $T_{CR,P}$ is the new delay for each clock.

Table 6.6: Improvement in Latency With Additional Pipeline Stages, N = 9

Size (bits)	# stages	Delay per Cycle, $T_{CR,P}$ (ns)	Max Frequency (MHz)	Time to Match, T_{SDG} (ns)	Total # Cycles	Circuit Latency (ns)	Savings in Time
9	1	$T_s + T_{CR} = 32.70$ $T_{CR} = 27.70$	30.58	11.94	4	131	-
	2	$\frac{T_{CR}}{2} + T_S = 18.85$	53.05	11.94	6	113	14%
	3	$\frac{T_{CR}}{3} + T_S = 14.23$	70.26	11.94	7	100	23%
	4	$\frac{T_{CR}}{4} + T_S = 11.93$					

Table 6.7: Improvement in Latency With Additional Pipeline Stages, N = 12

Size (bits)	# stages	Delay per Cycle, $T_{CR,P}$ (ns)	Max Frequency (MHz)	Time to Match, T_{SDG} (ns)	Total # Cycles	Circuit Latency (ns)	Savings in Time
12	1	$T_s + T_{CR} = 48.60$ $T_{CR} = 43.60$	20.58	14.36	4	194	-
	2	$\frac{T_{CR}}{2} + T_S = 26.80$	37.30	14.36	6	161	17%
	3	$\frac{T_{CR}}{3} + T_S = 19.53$	51.20	14.36	7	137	29%
	4	$\frac{T_{CR}}{4} + T_S = 15.90$	62.89	14.36	8	127	35%
	5	$\frac{T_{CR}}{5} + T_S = 13.72$					

Table 6.8: Improvement in Latency With Additional Pipeline Stages, N = 15

Size (bits)	# stages	Delay per Cycle, $T_{CR,P}$ (ns)	Max Frequency (MHz)	Time to Match, T_{SDG} (ns)	Total # Cycles	Circuit Latency (ns)	Savings in Time
15	1	$T_s + T_{CR} = 65.42$ $T_{CR} = 60.42$	16.56	18.94	4	262	-
	2	$\frac{T_{CR}}{2} + T_S = 35.21$	28.40	18.94	6	211	19%
	3	$\frac{T_{CR}}{3} + T_S = 25.14$	39.78	18.94	7	176	33%
	4	$\frac{T_{CR}}{4} + T_S = 20.11$					

Table 6.9: Improvement in Latency With Additional Pipeline Stages, N = 18

Size (bits)	# stages	Delay per Cycle, $T_{CR,P}$ (ns)	Max Frequency (MHz)	Time to Match, T_{SDG} (ns)	Total # Cycles	Circuit Latency (ns)	Savings in Time (%)
18	1	$T_s + T_{CR} = 75.09$	13.32	19.63	4	280	-
24	1	$T_s + T_{CR} = 100.59$	9.94	24.47	4	402	-
	2	$T_s + T_{CR} = 70.09$	24.97	19.63	6	240	14%
	2	$\frac{T_{CR}}{2} + T_S = 40.05$	18.94	24.47	6	317	21%
	3	$\frac{T_{CR}}{2} + T_S = 52.80$	35.26	19.63	7	199	29%
	3	$\frac{T_{CR}}{3} + T_S = 28.36$	27.13	24.47	7	258	36%
	4	$\frac{T_{CR}}{3} + T_S = 36.86$	44.40	19.63	8	180	36%
	4	$\frac{T_{CR}}{4} + T_S = 22.52$	24.61	24.47	8	231	43%
	5	$\frac{T_{CR}}{4} + T_S = 28.90$					
	5	$\frac{T_{CR}}{5} + T_S = 19.02$					
		$\frac{T_{CR}}{5} + T_S = 24.12$					

Table 6.10: Improvement in Latency With Additional Pipeline Stages, N = 21

Size (bits)	# stages	Delay per Cycle, $T_{CR,P}$ (ns)	Max Frequency (MHz)	Time to Match, T_{SDG} (ns)	Total # Cycles	Circuit Latency (ns)	Savings in Time
21	1	$T_s + T_{CR} = 88.21$	11.34	21.29	4	353	-
		$T_{CR} = 83.21$					
	2	$\frac{T_{CR}}{2} + T_S = 46.61$	21.46	21.29	6	280	21%
	3	$\frac{T_{CR}}{3} + T_S = 32.74$	26.73	21.29	7	262	26%
	4	$\frac{T_{CR}}{4} + T_S = 25.80$	38.76	21.29	8	206	42%
	5	$\frac{T_{CR}}{5} + T_S = 21.64$	46.21	21.29	9	195	45%
	6	$\frac{T_{CR}}{6} + T_S = 18.87$					

Table 6.11: Improvement in Latency With Additional Pipeline Stages, N = 24

For every size (N), ideal additional pipeline stages can be seen to improve the overall latency. The number of extra pipeline stages is limited by the SD Generation delay, and also by the setup time, T_s.

With the potential frequency/time improvements in mind, an extra pipeline stage was implemented.

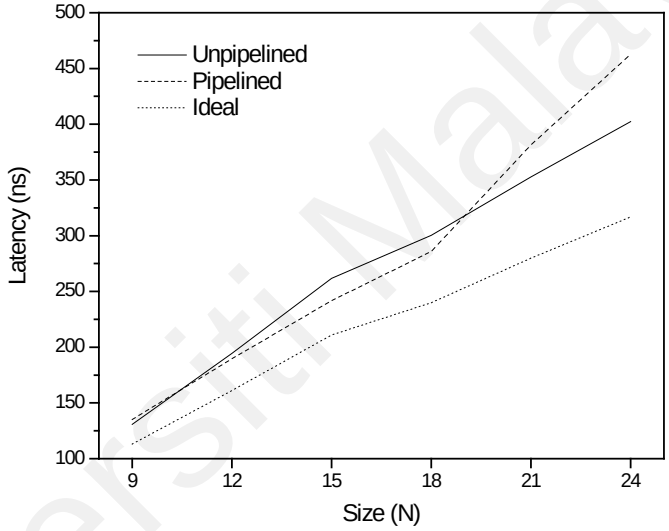

Figure 6.11 Graph Flat CORDIC Latency for Unpipelined, Pipelined & Ideal Designs Versus Size (N)

The results in Fig. 6.11 show that the implemented pipeline stage improves the latency for some values of N, while increasing the latency for others. The two factors that could cause the deterioration for some sizes with this addition of an extra pipeline stage are :

- The uneven distribution of combinational logic in the extra cycles result in very small savings in time per cycle
- The fact that the first additional pipeline stage increases the number of cycles by 2, combined with the factor above

6.5 Comparison of Flat CORDIC against Other Methods

The performance of the Flat CORDIC module was compared against other implementations of sine/cosine generating functions in both VHDL as well as internal computer processing speeds in C++.

6.5.1 Implementation of Iterative CORDIC Architecture

The iterative architecture for CORDIC can be implemented directly from the CORDIC equations as given in Chapter 2. The design uses 3 registers, one each for X, Y and Z values. The micro-rotation direction is driven by the sign of either the Y or the Z register, depending on whether vector mode or rotation mode CORDIC is used.

First, the initial values are loaded into the X, Y and Z registers. (X_o : 1, Y_o : 0, Z_o : input angle). Then, on each of the N clock cycles, the values from the registers are passed through the shifters and adder-subtractors, and the results placed back in the registers. The ROM address is incremented so that the appropriate elementary angle is presented to the Z-adder-subtractor. The shifters are modified to cause the desired shift for the iteration.

To standardize the comparison, the number of iterations for conventional CORDIC was chosen in such a way that the error range matched that of the Flat CORDIC results. For the most part, the latency is N+1 cycles (one extra for the final scaling). For sizes N = 15 and 18 however, the latency is N+2 cycles. It uses word-wide datapaths ($N+\log_2 N$ bits). The speed and size results are shown in the Table 6.12.

Table 6.12: Max. Frequency and Gate Count of Flat CORDIC and Conventional CORDIC Architectures

Size (bits)	Accuracy	Max Frequency (MHz)		Latency (ns)		Gate Count	
		Flat CORDIC	CORDIC	Flat CORDIC	CORDIC	Flat CORDIC	CORDIC
9	2^{-9}	30.58	53.66	130.8	186.4	6,238	6,103
12	2^{-12}	20.58	50.04	194.4	255.7	11,325	7,253
15	2^{-15}	15.29	41.99	261.7	404.9	21,389	13,553
18	2^{-18}	13.32	43.72	300.4	457.5	37,281	13,733
21	2^{-19}	11.34	43.12	352.8	510.2	61,669	15,561
24	2^{-20}	9.94	43.45	402.4	575.3	104,690	17,289

6.5.2 Implementation of Direct Sine/cosine Generation

The basic definitions for sine and cosine are as given below :

$$\cos(z) = 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - \frac{z^6}{6!} + \ldots = 1 + \sum_{k=1}^{\infty} (-1)^k \frac{z^{2k}}{(2k)!} \tag{6.1}$$

$$\sin(z) = z - \frac{z^3}{3!} + \frac{z^5}{5!} - \frac{z^7}{7!} + \ldots = \sum_{k=0}^{\infty} (-1)^k \frac{z^{2k+1}}{(2k+1)!} \tag{6.2}$$

where z is an angle in radians

This was implemented in VHDL. The design utilizes a ROM to store the inverse factorial values and multipliers to generate the values of z^i, as well as multiply these values by the ones stored in the ROM. The designs were implemented for sizes N = 9, 12, 15 and 18 using the standardized E_N bits ($E_N = N + \log_2 N$) to match the Flat CORDIC module. The number of terms for each size N was determined by the maximum error (this also standardized to match the Flat CORDIC error results).

Table 6.13: Number of Terms using Direct Sine/cosine Generation

Size (bits)	Cos	# terms	Sin	# terms
9	$1 - \frac{\mathbf{z}^2}{2!} + \frac{\mathbf{z}^4}{4!}$	3	$\mathbf{z} - \frac{\mathbf{z}^3}{3!} + \frac{\mathbf{z}^5}{5!}$	3

12	$1-\frac{\mathbf{z}^2}{2!}+\frac{\mathbf{z}^4}{4!}$	3	$\mathbf{z}-\frac{\mathbf{z}^3}{3!}+\frac{\mathbf{z}^5}{5!}$	3
15	$1-\frac{\mathbf{z}^2}{2!}+\frac{\mathbf{z}^4}{4!}-\frac{\mathbf{z}^6}{6!}$	4	$\mathbf{z}-\frac{\mathbf{z}^3}{3!}+\frac{\mathbf{z}^5}{5!}-\frac{\mathbf{z}^7}{7!}$	4
18	$1-\frac{\mathbf{z}^2}{2!}+\frac{\mathbf{z}^4}{4!}-\frac{\mathbf{z}^6}{6!}$	4	$\mathbf{z}-\frac{\mathbf{z}^3}{3!}+\frac{\mathbf{z}^5}{5!}-\frac{\mathbf{z}^7}{7!}$	4

Table 6.14: Max. Frequency, Latency and Gate Count of Flat CORDIC and Conventional CORDIC Architectures

Size (bits)	Accuracy	Max Frequency (MHz)		Latency (ns)		Gate Count	
		Flat CORDIC	Power Series	Flat CORDIC	Power Series	Flat CORDIC	Power Series
9	2^{-9}	30.58	32.07	130.8	187.1	6,238	8,410
12	2^{-12}	20.58	26.53	194.4	226.1	11,325	15,451
15	2^{-15}	15.29	24.76	261.7	282.7	21,389	37,978
18	2^{-18}	13.32	24.06	300.4	290.9	37,281	52,500

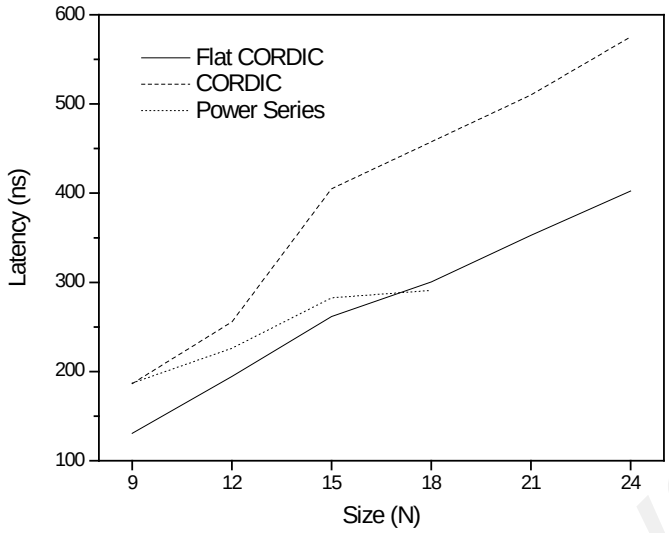

Figure 6.12: Latency of Flat CORDIC, Conventional CORDIC And Power Series Architectures

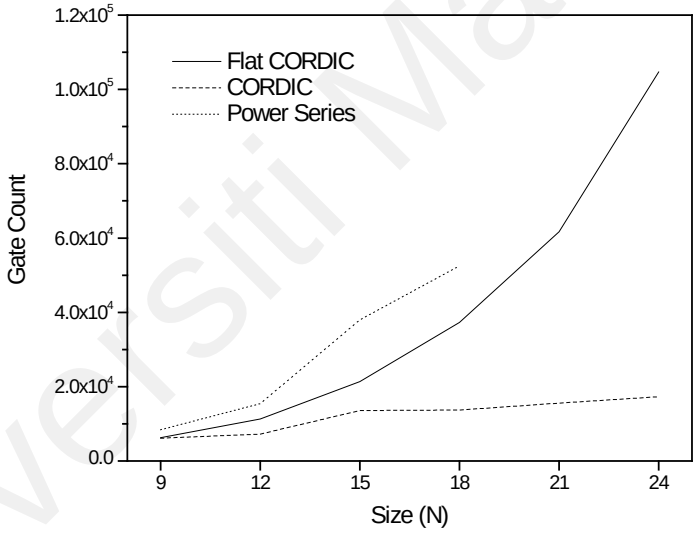

Figure 6.13: Gate Count for Flat CORDIC, Conventional CORDIC And Power Series Architectures

Fig. 6.12 shows that even the unpipelined Flat CORDIC can generate the sine/cosine values a lot faster than conventional CORDIC for size N = 9, it is roughly 100 ns faster. For sizes N = 15 and above, there is approximately a 200 ns difference between the two methods. There is roughly a 30% savings in time using the Flat CORDIC architecture. Fig. 6.12 also shows that the Flat CORDIC executes faster than the Power Series for small values of N (9, 12, 15). However, as N increases, the Power Series method of

calculation appears to yield results faster as can be seen in the intersection of the lines in Fig. 6.12.

Fig. 6.13 however indicates that while the gate count for conventional CORDIC increases very gradually, that for Flat CORDIC increases a lot more. The increase in gate count for the Power Series Method is even more drastic, due to the many increasingly large multipliers that are required.

6.5.3 C++ Implementation of CORDIC Sine/cosine Generation

The simple addition and shifting process of CORDIC [with a final multiplication for scaling factor compensations] was written into C++ code to test the computer processor speed. The processor is an Intel Pentium ® 4 chip, operating at 2.40 GHz, with 256 MB RAM. An inbuilt clock function was incorporated in the program to find the average speed of sine/cosine value generation.

6.5.4 C++ Implementation of MATH.H Sine/cosine Generation

The sine/cosine values generated using the math.h inbuilt function was also examined to get the average sine/cosine computation of the computer processor. For both these methods, the accuracy was standardized with the Flat CORDIC results.

Table 6.15: Maximum Operating Frequencies For Sine/cosine Computation (Hardware Versus Software)

Size (bits)	Max Frequency (MHz)		
	Flat CORDIC	**C++ CORDIC**	**C++ Internal Calculation**
9	30.58	0.252	1.186
12	20.58	0.181	1.123
15	15.29	0.119	0.704
18	13.32	0.104	0.736
21	11.34	0.097	0.615
24	9.94	0.101	0.587

It was mentioned earlier that in high-speed operations, hardware solutions are being favored to their software counterpart. The results displayed in Table 6.15 are an indication of the vast difference in calculation speeds when making use of software solutions compared with hardware solutions.

CHAPTER 7 : CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This thesis has concentrated on the implementation of Flat CORDIC on FPGAs. Specifically, Flat CORDIC has been utilized to perform the specific trigonometric functions of sine/cosine generation. The proposed implementation has also been compared with existing sine/cosine generation methods in literature in terms of speed of operation and circuit size.

The first part of this research focuses on the derivation of the Flat CORDIC equations and the subsequent simplifications to make it suitable for implementation. The existing algorithm for generating the SDs required for the Flat CORDIC equations has been tested and its accuracy checked. An alternative to the existing encoder method was proposed that yielded more accurate results was synthesized and analyzed. The two methods were implemented in the full Flat CORDIC design, and the synthesis results of these were also examined.

From the results obtained, the proposed comparator method produces more accurate results than its encoder counterpart. The speed of SD Generation is also higher. However, the comparator method utilizes a much larger number of gates. Once integrated into the full Flat CORDIC design, the improvements in speed of the comparator method could no longer be seen. This is due to the SD Generation Module not being the bottleneck in the overall design. The increase in size using the comparator method, however, ceased to be very significant when integrated into the main

architecture. This is because the SD Generation portion by itself constitutes a small percentage of the full design.

The next part of the thesis focuses on designing a new method to combine the SDs to correspond with the Flat CORDIC equations and generate the pre-scaled sine/cosine values (the C&A Module). The proposed architecture is a matrix of cells, each seating one of a selection of specially designed modules. The choice of the module filling each cell depends on the number of inputs into the cell. The outputs of the cell were designed to ripple through to the next column and layer to effectively produce left/right shifts.

Simulations were performed to check that the C&A Module produced the expected results. Synthesis results indicated that this portion took up a big part of the Flat CORDIC architecture and also that the amount of combinational logic was quite large, resulting in delays that limited the maximum operating frequency.

The following part explored the possibilities of speeding up the design by pipelining the C&A Module. Calculations were performed to estimate the savings in time. The maximum possible number of extra pipeline stages for increased performance was extrapolated assuming ideal pipeline implementation. An extra pipeline stage was also implemented to check the synthesis results.

It was seen that the maximum savings in time could reach a high of 46%. The number of additional stages for bit sizes 9 through 24 ranged between 3 and 5. The implementation results, however, indicated that an extra stage could potentially cause a decrease in performance if not implemented in the right place. This was due to a

combination of the uneven logic distribution per cycle and also the fact that the first extra pipeline stage increased the circuit latency by 2 cycles.

Finally, a comparison was made with other methods in literature. Flat CORDIC was found to be much faster than the conventional bit-parallel CORDIC as well as the power series sine/cosine generation method. The Flat CORDIC size (gate count) was also smaller than that of the power series generation method. However, while the conventional CORDIC gate count increase was seen to be very gradual with the increase in N, the Flat CORDIC gate counts were not only larger, but they increased at a higher rate as well. It was also seen in a comparison against software sine/cosine generation, that the Flat CORDIC speed was several orders of magnitude larger. This supports the increasing trend towards hardware methods for high-speed and massive compute-intensive processes.

7.2 Suggestions for Future Work

This work has presented a detailed study of Flat CORDIC Sine/Cosine in Rotation Mode. Among the other topics of interest as an extension would be an extension into the hyperbolic coordinate system to generate sinh/cosh values. The main difference in the characteristic equations for these two coordinate systems is the value of m, which is either +1 or -1. The sine/cosine generating module can be modified with sign-changing units like xor gates to include this difference.

It is seen that the major bottleneck in the design, and also the largest part of the design is the C&A Section. It would be worthwhile to investigate this ripple architecture further, to find ways to simplify it by combining more units to make the design more

compact. It is also seen that the BLUE module in this architecture is instantiated many times, and forms a core of the ripple architecture. As the bit-size (N) increases, the number of combinational terms increases and the number of BLUE modules also increases. An optimization of this unit in terms of delay could improve the area/time measurements of the Flat CORDIC design.

Another potential area of interest would be to extend Flat CORDIC to Vectoring Mode CORDIC. The simplification of $X_o = 1$ and $Y_o = 1$ would no longer be applicable in this case, due to the fact that the initial vector is specified, and will not coincide with the X-axis. Suitable modifications to the equations to accommodate this mode of operation could be studied.

REFERENCES

1. Andraka, R. “A Survey of CORDIC Algorithms for FPGAs,” Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA’98), Monterey, CA, Feb 22 – 24, pp. 191 – 200.
2. Clarke, C. T. (1995). “Unravelling CORDIC”, Proc. ISIC-95, Singapore, Sept. 1996.
3. Dawid, H., and Meyr, H. (1996). “The Differential CORDIC Algorithm: Constant Scale Factor Redundant Implementation without Correcting Iterations,” IEEE Transactions on Computers, Vol. 45, No. 3, March 1996, pp. 307 – 318.
4. Dick, C. (1996). “Computing the Discrete Fourier Transform on FPGA Based Systolic Arrays,” ACM/SIGDA Int. Symp. On Field Programmable Gate Arrays, Feb 1996, pp. 129 – 135.
5. Ercegovac, M. D. and Lang, T. (1988). “Implementation of Fast Angle Calculation and Rotation Using On-line CORDIC,” ISCAS’88, pp. 2703 – 2706.
6. Ercegovac, M. D. and Lang, T. (1990). “Redundant and On-Line CORDIC : Application to Matrix Triangularization and SVD,” IEEE Trans. Computers, vol. 39, No.6, pp. 725 – 740, June 1990.
7. Gisuthan, B., Srikanthan, T., Asari, K. V. (2000). “A High Speed Flat CORDIC Based Neuron with Muli-Level Activation Function for Robust Pattern Recognition,” Proc. Of Fifth IEEE International Workshop on Computer Architectures for Machine Perception (CAMP’00).
8. Gisuthan, B. (2000). “A Unified Architecture for Flat CORDIC,” M. A. Sc. Thesis, School of Computer Engineering, Nanyang Technological University, Singapore.
9. Mayosky, M.A., Battaiotto, P. E. and Toccaceli, G. M. (1998). “A CORDIC Architecture for Vector Control,” Proc. Of the Int. Conf. on Signal Processing Applications and Technology.

10. Meyer-Base, U., Meyer-Base, A. and Hilberg, W. (1994). “Coordinate Rotation DIgital Computer (CORDIC) Synthesis for FPGA,” 4th International Workshop on Field Programmable Logic and Applications (FPL’94), Prag, Czech Republic.

11. Meyer-Base, U., Meyer-Base, Mellott, J., and Taylor, F. (1998). “A Fast Modified CORDIC- Implementation of Radial Basis Neural Networks,” Journal of VLSI Signal Processing, vol. 20, pp. 211 – 218.

12. Symanski, J. J., Henderson, T., Shirasago, J., Celto, J., and Drake, B. (1987). “SLAPP : A special purpose multiprocessor array for signal processing and linear algebra,” SPIE : Advanced Algorithms and Architectures for Signal Processing II, vol. 826, pp. 232 – 239.

13. Takagi, N., Asada, T., and Yajima, S. (1991). “Redundant CORDIC Methods with a Constant Scale Factor for Sine and Cosine Computation,” IEEE Trans. Computers, vol. 40, no. 9, pp. 989-995, Sept .1991.

14. Timmermann D., Hahn H., Hosticka B. (1992). “A Low Latency Time CORDIC Algorithm with Increased Parallelism,” IEEE Transactions on Computers, Vol. 41, No. 8, pp. 1010 – 1015, 1992

15. Timmermann D., Dolling S. (1997). “Unfolded Redundant CORDIC VLSI Architectures With Reduced Area and Power Consumption,” VLSI ’97, Gramado, Brasilien, Aug. 1997.

16. Valls, J., Kuhlmann, M. and Parhi, K. K. (2002). “Evaluation of CORDIC Algorithms for FPGA Design,” Journal of VLSI Signal Processing, vol. 32, pp. 207 – 222.

17. Vladimirova, T. and Tiggeler, H. (1999). “FPGA Implementation of Sine and Cosine Generators Using the CORDIC Algorithm,” MAPLDCon’99.

18. Walther, John S. (1971). “A Unified Algorithm For Elementary Functions,” Spring Joint Computer Conference Proceedings, Vol. 38, 1971, pp. 379 – 385.

19. Wang, S., Piuri, V., Swartzlander, Earl E. Jr. (1997). “Merged Scaling Multiplication CORDIC Algorithm,” IEEE International Symposium on Circuits and Systems, Hong Kong, June 9 – 12.

20. Wassatsch A., Dolling S., Timmermann D. (1998). “Area Minimization of Redundant CORDIC Pipeline Architectures,” ICCD’98, Intl. Conference on Computer Design.