FPGA IMPLEMENTATION OF EEAS CORDIC BASED SINE AND COSINE GENERATOR

Thesis report submitted towards the partial fulfilment of requirements for the award of the degree of

Master of Technology (VLSI Design & CAD)

Submitted by

Vikas Sharma
Roll No. 60761025

Under the Guidance of

Mrs. Manu Bansal
Senior Lecturer, ECED

Department of Electronics and Communication Engineering
THAPAR UNIVERSITY
PATIALA-147004, INDIA
JULY 2009

DECLARATION

I hereby declare that the thesis report entitled "FPGA implementation of EEAS CORDIC based Sine and Cosine generator" is an authentic record of my own work carried out as requirements for the award of degree of M. Tech. (VLSI Design & CAD) at Thapar University, Patiala, under the guidance of Mrs. Manu Bansal, Senior Lecturer, ECED during January to June, 2009.

Vikas Sharma

Vikas Sharma
Roll No.60761025

Date: 14/07/09

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Manu Bansal

Mrs. Manu Bansal
Sr. Lecturer, ECED

Counter signed by:

Dr. A.K. Chatterjee
Head,
Electronics & Communication.
Engineering Department.

Mr. R.K.Sharma 21/7
Dean,
Academic Affairs,

i

ACKNOWLEDGEMENT

To discover, analyze and to present something new is to venture on an untrodden path towards an unexplored destination is an arduous adventure unless one gets a true torchbearer to show the way. This enlightening guidance, I found in my revered guide Mrs. Manu Bansal, Senior Lecturer, Electronics & Communication Engineering Department, Thapar Institute of Engineering & Technology (Deemed University), Patiala, without whose patronization it was never possible to give final shape to this thesis. I express my heartfelt gratitude towards her for her valuable guidance, encouragement, constant involvement, inspiration and the enthusiasm with which she solved my difficulties.

I shall be failing in my duties if I do not express my deep sense of gratitude towards Dr. A.K. Chatterjee Professor & Head of the Department, Electronics & Communication Engineering Department and Mrs Alpana Aggarwal, P.G. Coordinator, Electronics and Communication Engineering Department.

I would also like to thank all the staff members and my co-students who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of my thesis. I am also thankful to the authors whose works I have consulted and quoted in this work. Last but not the least I would like to thank God for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Vikas Sharma

ABSTRACT

With increasing on-chip complexities the on-chip area is a major concern. Today users desire every gadget to be smaller in size, mainly the handheld systems. So researchers keep on exploring for the methods to minimize the on chip area. They have to constantly muddle through speed area trade-off. CORDIC is one such kind of algorithm. The CORDIC algorithm has become a widely used approach to elementary function evaluation when silicon area is a primary constraint. The implementation of CORDIC algorithm requires less complex hardware than the conventional DSP methods. CORDIC is far more economical compared to DSP algorithms in terms of area and power consumption. The main benefit of this algorithm is its flexibility. Even if the speed of the system implemented using basic CORDIC degrades, but it can be easily adjusted by introducing few modifications as discussed in this thesis.

The scope of this thesis includes study of VHDL for the design of EEAS CORDIC algorithm to generate basic trigonometric functions along with validations using implementation on FPGA with the design interfaced to the LCD to view the results.

ABBREVIATIONS

CLBs	Configurable Logic Blocks
CORDIC	Cordic Rotation Digital Computer
DFT	Digital Fourier Transform
DHT	Digital Hartley Transform
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
LUT	Look Up Table
RAM	Random Access Memory
ASICs	Application-Specific Integrated Circuits
RTL	Register Transfer Level
SRAM	Static RAM
SVD	Singular Value Deposition
ULP	Unit in the Last Place
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Due to rapid advances made in VLSI technologies have led the way to an entirely different approach to computer design for real-time applications, using special-purpose architectures with custom chips. Special-purpose architectures efficiently map the algorithmic needs of the problem to hardware. Use of special arithmetic techniques for special applications and introduction of pipelining and parallelism lead to designs very different from basic computers.

Nowadays world of information interchange revolves around transmission and viewing real-time images. Many of the Digital Signal Processing (DSP) applications try to closely simulate real life images. Speed, clarity, and resemblance to real time objects are some of the many issues to be addressed in order to achieve this goal. Trigonometric function calculation is one of the primary tasks performed in DSP applications. For a long time microprocessor-based systems have been used to perform this task. Software algorithms used by the processors do not meet the highly demanding needs of all DSP tasks.

Using hardware systems to perform these DSP tasks is a competent solution to this problem. FPGAs are often used as coprocessors to perform all the high-speed tasks that cannot be achieved by microprocessors. FPGAs are chosen because they are on-site programmable and are highly suitable for hardware implementations. The software solutions adapted by the microprocessors to implement trigonometric functions are computer intensive. They do not suit hardware platforms because they need complex circuits to perform the mathematical operations. Hence hardware algorithms are adopted for the calculation of trigonometric functions.

To calculate trigonometric functions one such algorithm is the CORDIC algorithm. The Word CORDIC stands for **CO**rdinate **R**otation **DI**gital **C**omputer .The CORDIC algorithm was developed by Jack E. Volder in 1959 .His objective was to build a real-

time navigational computer for use on aircrafts, so he was primary interested in computing trigonometric functions. We give an introduction to the CORDIC method used my most handheld calculators (such as the ones by Texas Instruments and Hewlett-Packard) to approximate the standard transcendental functions. The CORDIC algorithm does not use Calculus based methods such as polynomial or rational function approximation. The CORDIC algorithms require only shifts, adders and table lookups, simple integer math. So, it is possible to implement a specialized CORDIC machine, small and fast enough for real time calculations, dedicated to that one purpose. The algorithm can be coded in firmware on even small microcontrollers. With slight modifications in initial conditions and data tables, the core algorithm can multiply, divide, modulate, and calculate square roots, hyperbolic functions, exponentials and logs. It is this versatility and simplicity that make CORDIC the preferred implementation of math functions on small hand calculators. All these tasks can be efficiently implemented using processing elements performing vector rotations. The COordinate Rotation DIgital Computer algorithm (CORDIC) offers the opportunity to calculate all the desired functions in a rather simple and elegant way. Due to the simplicity of the involved operations the CORDIC algorithm is very well suited for VLSI implementations.

Compared to other approaches, CORDIC is a clear winner when a hardware multiplier is unavailable, e.g. in a microcontroller, or when it is desired to save the gates required to implement one, e.g. in an FPGA. On the other hand, when a hardware multiplier is available, e.g. in a DSP microprocessor, table-lookup methods and good old-fashioned power series are generally faster than CORDIC.

The shortcoming of reduced speed of the system due to the introduction of CORDIC can be compensated by making the angles set of the CORDIC bigger by the introduction of one more signed power of two.

1.2 OBJECTIVE OF THESIS

In this thesis CORDIC algorithm with Extended Elementary Angles Set is explored and it is used to implement basic trigonometric functions i.e. Sin and Cos in VHDL. The design is then implemented on Spartan 3E FPGA board and validated using interfacing design to the LCD on the board itself. The algorithm is hardware efficient compared to

respective DSP applications as it eliminates the multipliers needed during DSP implementations. Due to relaxed constraints on the angle recording mechanism in CORDIC i.e. with the introduction of one more signed power of two the system became more complex but the speed of computation increases. XILINX ISE 9.1i and ModelSimXE III6.3c are used to perform all the simulations and synthesis.

1.3 ORGANIZATION OF THESIS

In this thesis the Chapter 2 explains the details of basic CORDIC algorithm along with the applications of CORDIC.

In Chapter 3 the improvements to enhance the speed of the algorithm i.e. the EEAS scheme along with the introduction of constraint on maximum number of iterations is explained.

In Chapter 4 FPGA implementation of CORDIC is discussed Xilinx Integrated Software Environment (ISE 9.2i) software. Language used for coding is VHDL.

In Chapter 5 the final results of synthesis and implementation along with conclusion and future scope.

CHAPTER 2
LITERATURE REVIEW

2.1 Introduction

Two basic CORDIC modes are known leading to the computation of different functions, the rotation mode and the vectoring mode. For both modes the algorithm can be realized as an iterative sequence of additions/subtractions and shift operations, which are rotations by a fixed rotation angle (sometimes called micro-rotations) but with variable rotation direction. Due to the simplicity of the involved operations the CORDIC algorithm is very well suited for VLSI implementations. However, the CORDIC iteration is not a perfect rotation which would involve multiplications with sine and cosine. The rotated vector is also scaled making a scale factor correction necessary. CORDIC revolves around the idea of rotating the phase of a complex number, by multiplying it by a succession of constant values. However, the multiplies can all be powers of 2, so in binary arithmetic they can be done using just shifts and adds and no actual multiplier circuit is needed.

2.2 ROTATIONS AND PSEUDO ROTATIONS

A simple form of CORDIC is based on the observation that if a unit-length vector with end

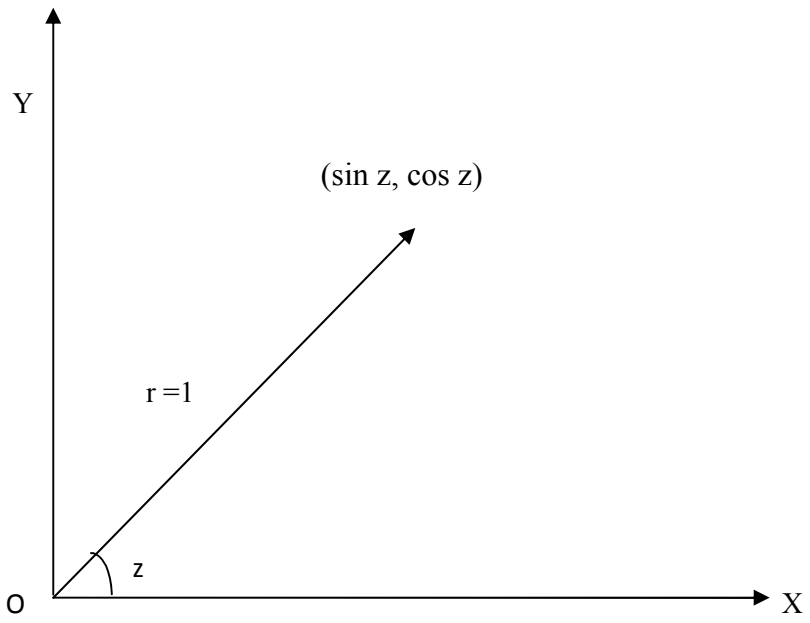


Figure: 2.1- Vector in Cartesian coordinates

point at (x,y)=(1,0) is rotated by an angle z, its new end point will be (x, y) = (cos z, sin z). Thus cos z and sin z can be computed by finding the coordinates of the new end point of the vector after rotation by z[2]. When a vector is rotated by an angle α from (x_1, y_1) to new point with coordinates (x_2,y_2). The rotation of a vector from one point in rectangular coordinate system can be explained by multiplication by a complex number with magnitude unity. Let initially the vector depicts the coordinate location

$$P = X + j\,Y \tag{1.1}$$

This point can be shifted to a new location by simply multiplying this complex number by another complex number with unity magnitude

$$Q = \cos\theta + j \sin\theta \tag{1.2}$$

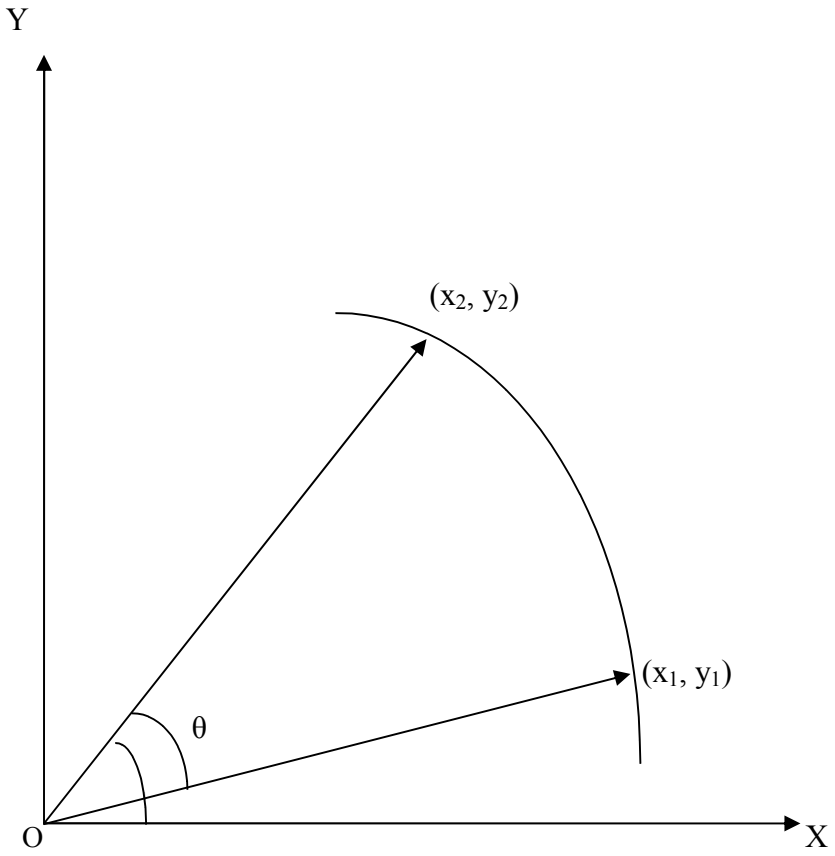


Figure: 2.2-Rotation of a vector

Vector Q having the magnitude unity. The new vector now is

$$P' = PQ \tag{1.3}$$

With P' being

$$P' = R + jI \tag{1.4}$$

$$R = X\cos\theta - Y\sin\theta \tag{1.5}$$

$$I = Y \cos\theta + X \sin\theta \quad (1.6)$$

When two complex numbers are multiplied their phases simply add up algebraically and their magnitudes get multiplied .So by just multiplying with a complex number with unity magnitude we are not going to change the magnitude but only rotate the vector by a desired angle. This is called the real rotation.

Let us suppose a vector in cartesian coordinates rotated as the equations below

$$x_{(i+1)} = x_{(i)} \cos\theta - y_{(i)} \sin\theta \quad (1.7)$$

$$y_{(i+1)} = y_{(i)} \cos\theta + x_{(i)} \sin\theta \quad (1.8)$$

Take out cos θ common from the above equations

$$x_{(i+1)} = \cos\theta \, [x_{(i)} - y_{(i)} \tan\theta] \quad (1.9)$$

$$y_{(i+1)} = \cos\theta \, [y_{(i)} + x_{(i)} \tan\theta] \quad (1.10)$$

$$[\text{We know } \tan\theta = \sin\theta/\cos\theta] \quad (1.11)$$

In the set of equations above we can easily see that the term cosθ provides scaling i.e. it reduces the magnitude of the vector as always $\cos\theta \leq 1$ and the term tanθ provides the rotating operation.

So now if the cosθ term is neglected from the original equations the magnitude of the resultant shifted vector increases by $\cos^{-1}\theta$ as $\cos^{-1}\theta > 1$. The equations now become

$$x_{(i+1)} = x_{(i)} - y_{(i)} \tan\theta \quad (1.12)$$

$$y_{(i+1)} = y_{(i)} + x_{(i)} \tan\theta \quad (1.13)$$

The resultant rotation shown by the vector as shown in figure below is called the pseudo rotations.

Now what Volder did is, he just proposed to break this rotation angle θ into a series of small successively shrinking angles α_i defined by the equation

$$\tan(\alpha_i) = 2^{-i} \quad (1.14)$$

so the equations become

$$x_{(i+1)} = x_{(i)} - (y_{(i)} 2^{-i})$$

$$y_{(i+1)} = y_{(i)} + (x_{(i)} 2^{-i}) \quad (1.15)$$

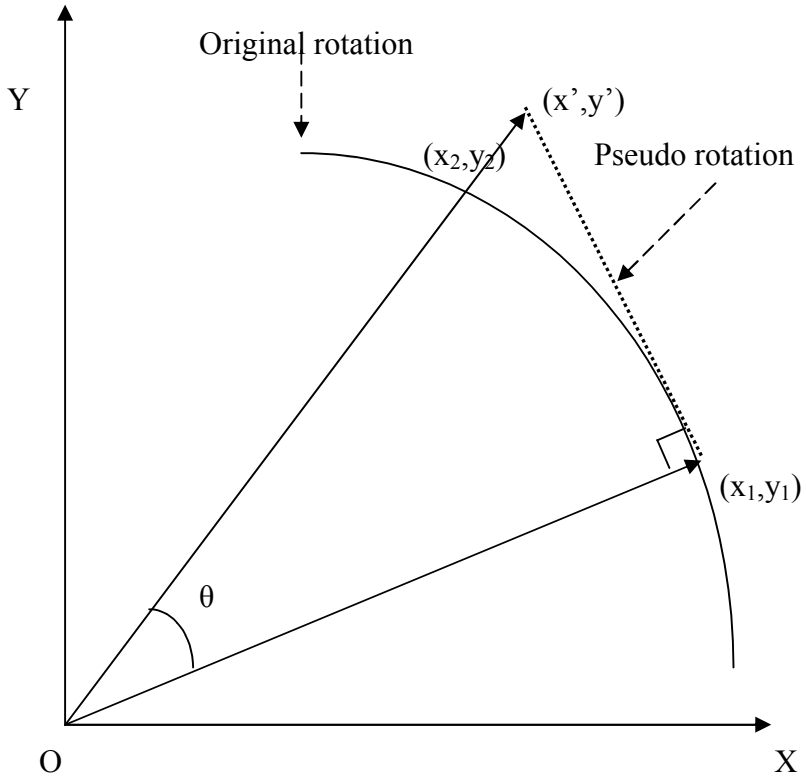


Figure: 2.3- Pseudo- rotations

Note that with increasing i values tan α_i goes on decreasing and so is this angle α_i. After each iteration this angle gets added (or subtracted depending on value of d_i explained later) to the angle accumulator.

$$\theta = \pm\theta_1 \pm\theta_2 \pm \theta_3 \pm \theta_4 \pm \theta_5 \pm \theta_6 \pm \theta_7 \pm \theta_8 \pm.....$$
$$= \pm\tan^{-1}(2^{-0}) \pm\tan^{-1}(2^{-1}) \pm\tan^{-1}(2^{-2}) \pm\tan^{-1}(2^{-3}) \pm\tan^{-1}(2^{-4}) \pm\tan^{-1}(2^{-5}).....$$ (1.17)

$$[\ \alpha_i = \tan^{-1}(2^{-i})]$$

And let z denotes the angle accumulator

$$z_{(i+1)} = z_{(i)} - (d_i\, e_{(i)})$$ (1.18)

and the equations are now

$$x_{(i+1)} = x_{(i)} - d_i\,(y_{(i)}\, 2^{-i})$$
$$y_{(i+1)} = y_{(i)} + d_i(x_{(i)}\, 2^{-i})$$ (1.19)

We now introduce a deciding factor d_i which is generated after comparison of the value to be reached with the value of the angle or the coordinate itself, depending upon the application. CORDIC algorithm for different applications depends just on the way of generation of this

decision factor d_i and the factors on which depend. It may be the angle z or the value of coordinate y.

Table 2.1– Angle set using one SPT

I	Tan (α_i)	α_i (degrees)
0	1	45
1	0.5	26.5
2	0.25	14.03
3	0.125	7.125
4	0.0625	3.576
5	0.03125	1.7899
6	0.015625	0.895
7	0.00781	0.4476
8	0.00390	0.2238
9	0.001953125	0.1

For generalized CORDIC for circular, linear and hyperbolic systems the equations can be modified as below

$$x_{(i+1)} = x_{(i)} - \mu d_i (y_{(i)} 2^{-i})$$

$$y_{(i+1)} = y_{(i)} + d_i(x_{(i)} 2^{-i}) \quad (1.20)$$

$$z_{(i+1)} = z_{(i)} - (d_i e_{(i)}) \quad (1.21)$$

Circular rotations $\mu = 1, e_{(i)} = \tan^{-1}2^{-i}$ (1.22)

Linear rotations $\mu = 0, e_{(i)} = 2^{-I}$ (1.23)

Hyperbolic rotation $\mu = -1, e_{(i)} = \tanh^{-1}2^{-I}$ (1.24)

A few observations

1. Since we are using powers of two for the K values, we can just shift and add our binary numbers. That's why the CORDIC algorithm doesn't need any multiplies!
2. You can see that starting with a phase of 45 degrees, the phase of each successive R multiplier is a little over half of the phase of the previous R. That's the key to understanding CORDIC: we will be doing a "binary search" on phase by adding or subtracting successively smaller phases to reach some "target" phase.
3. The sum of the phases in the table up to L = 3 exceeds 92 degrees, so we can rotate a complex number by ± 90 degrees as long as we do four or more "R = 1 ± jK" rotations. Put that together with the ability to rotate ±90 degrees using "R = 0 ± j1", and you can rotate a full ±180 degrees.
4. Each rotation has a magnitude greater than 1.0. That isn't desirable, but it's the price we pay for using rotations of the form "1 + jK". The "CORDIC Gain" column in the table is simply a "cumulative magnitude" calculated by multiplying the current magnitude by the previous magnitude. Notice that it converges to about 1.647; however, the actual CORDIC Gain depends on how many iterations we do. (It doesn't depend on whether we add or subtract phases, because the magnitudes multiply either way.)[3]

Examples: In conventional CORDIC we use these angles to form all other angles 45, 26.6, 14, 7.1, 3.6, 1.8, .9, 0.4 ……..

30 = 45 -26.6+14 -7.1+3.6+1.8 -0.9+0.4

90 = 45+26.6+14+7.1-3.6+1.8 -0.9+0.4

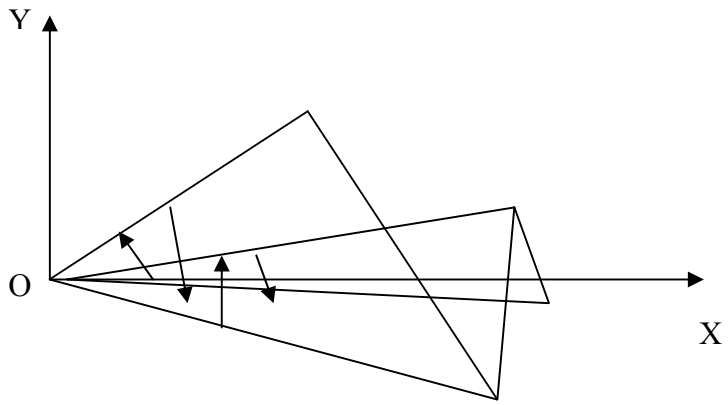


Figure: 2.4- The CORDIC iterations

The angles add up to ±91.7° and angles outside this range are dealt using angle wrapper which uses standard trigonometric identities.

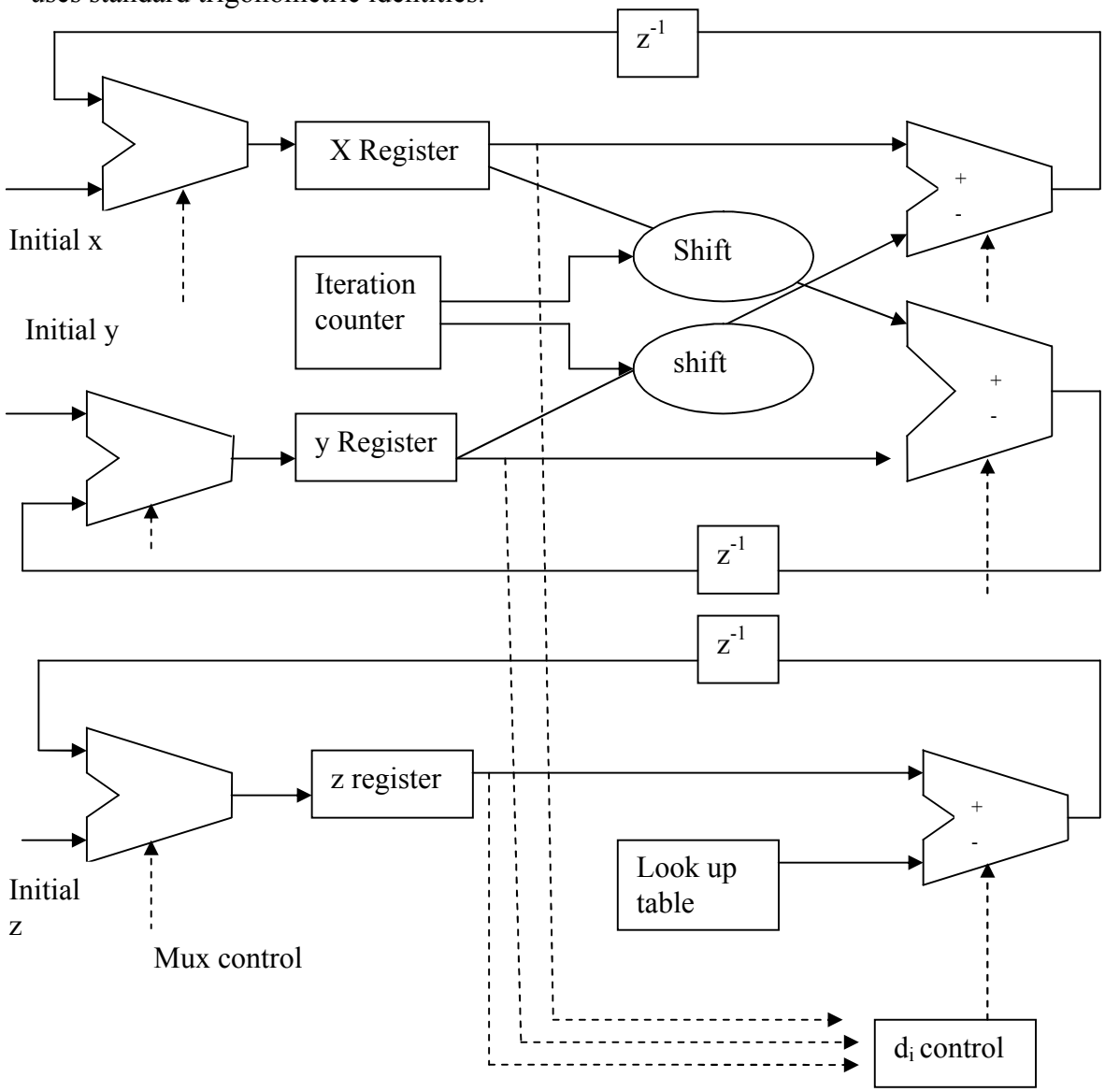


Figure: 2.5-CORDIC block diagram

2.3 GIVENS ROTATIONS

The rotation of the vectors can be easily explained using the Givens rotations. Givens rotation actually forms the basis of CORDIC algorithm. The equations for the rotation of a vector in cartesian coordinates can be easily depicted from this figure only using some geometric manipulations. These equations can be derived also by using conversion equations from cartesian to polar and polar to cartesian coordinates. The diagram depicting Givens rotations is shown in Figure 2.6.

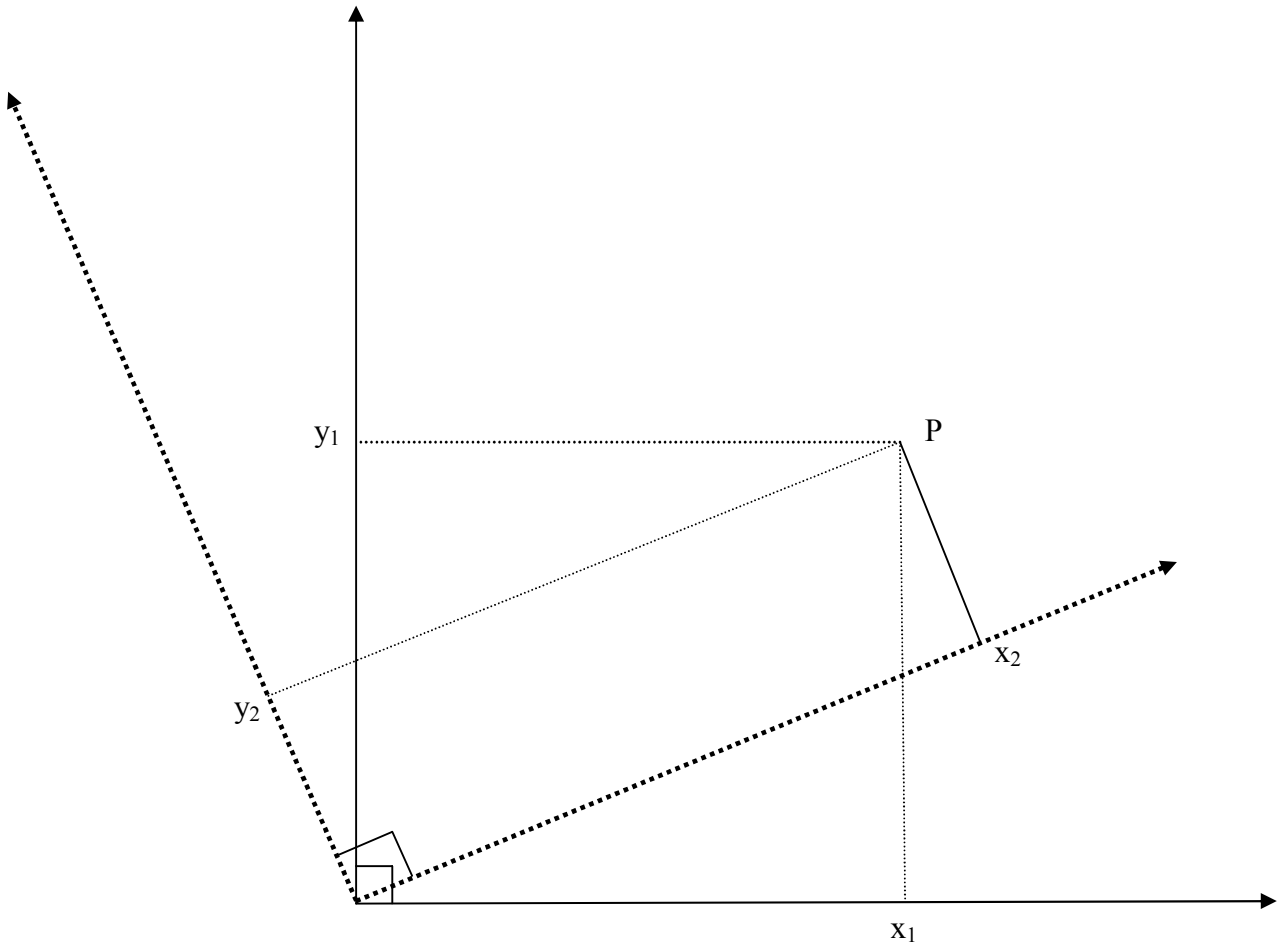


Figure: 2.6 - Givens Rotation.

2.4. THE SCALING FACTOR

As can be seen from the figure above the exclusion of cos term results in increase of the resultant vector after rotation. So a scaling factor needs to be multiplied. The value of cos H does not depend on the direction of rotation. That is a property of cosine, that

$$\cos(H) = \cos(-H) \qquad (1.25)$$

That fact is very convenient, because the cosine terms are a product that does not depend on d_i.

$\cos(\tan^{-1}(1)) \times \cos(\tan^{-1}(1/2)) \times \cos(\tan^{-1}(1/4)) \times \cos(\tan^{-1}(1/8)) \times \cos(\tan^{-1}(1/16)) \quad ...$

$= 0.707107 \times 0.894427 \times 0.970143 \times 0.992278 \times 0.998052 \times 0.999512 \times 0.999878 \quad .....$

The value of cos(tan(H)) approaches unity as H approaches zero, and the limit of the product of terms is a constant number, 0.607252935..... It is that many significant digits even after 16 steps. The exact value of the product is easily computed in advance and stored as a

constant. This factor is called the CORDIC gain. Some computations (like the length of the vector, or the values of cosine and sine, have to be compensated at some point in the calculation by this gain. That can be done either in the initial conditions or at the end result. Multiplying by 0.607252935 is done as 39797, because 39797/65536 ≈ 0.607252935.

{We know that $\cos(\tan^{-1}(1/2^i)) =$ ———}

2.5 CORDIC OPERATION MODES

Using the CORDIC algorithm and the shift sequences stated above, a number of different functions can be calculated in rotation mode and vectoring mode as shown below:

CORDIC ROTATION MODE

In CORDIC terminology, the preceding selection rule for d_i,

which makes z converge to 0, is known as "rotation mode".

$$x(i+1) = x(i) - d_i\,(2^{-i}y(i))$$

$$y(i+1) = y(i) + d_i\,(2^{-i}x(i)) \qquad (1.26)$$

$$z(i+1) = z(i) - d_i\,e(i) \qquad (1.27)$$

$$\text{where } e(i) = \tan^{-1} 2^{-I} \qquad (1.28)$$

After m iterations in rotation mode, when z(m) =0, we have $\Sigma\alpha(i) = z$ and the CORDIC equations become:

$$x(m) = K(x\cos z - y\sin z)$$

[Rotation mode] $$y(m) = K(y\cos z + x\sin z) \qquad (1.29)$$

$$z(m) = 0 \qquad (1.30)$$

Rule: Choose $d_i \in \{-1, 1\}$ such that $z \rightarrow 0$

The constant K is

$$K = 1.646\ 760\ 258\ 121...$$

Start with $x = 1/K$

$$= 0.607\ 252\ 935...$$

and $y = 0$

as z(m) tends to 0 with CORDIC in rotation mode, x(m) and y(m) converge to cos z and sin z

For k bits of precision in the results, k CORDIC iterations are needed, because $\tan^{-1} 2^{-i} = 2^{-i}$. Convergence of z to 0 is possible since each of our angles is more than half of the previous angle or, equivalently, each is less than the sum of all the angles following it.[4]

Domain of convergence is

$$-99.7^o \le z \le 99.7^o,$$

where 99.7^o is the sum of all the angles (contains $[-\pi/2, \pi/2]$ radians)

CORDIC VECTORING MODE

Let us now make y tend to 0 by choosing

$$d_i = -\text{sign}(x(i)y(i)) \quad (1.31)$$

After m steps in vectoring mode

$$\tan(\Sigma\alpha(i)) = -y/x \quad (1.32)$$

$$x(m) = K(x\cos(\Sigma\alpha(i)) - y\sin(\Sigma\alpha(i))) \quad (1.33)$$

$$= K(x - y\tan(\Sigma\alpha(i)))/(1 + \tan^2(\Sigma\alpha(i)))^{1/2}$$

$$= K(x + y^2/x)/(1 + y^2/x^2)^{1/2}$$

$$= K(x^2 + y^2)^{1/2} \quad (1.34)$$

The CORDIC equations thus become:

$$x(m) = K(x^2 + y^2)^{1/2} \quad (1.35)$$

[Vectoring mode] $$y(m) = 0 \quad (1.36)$$

$$z(m) = z + \tan^{-1}(y/x) \quad (1.37)$$

Choose $d_i \in \{-1, 1\}$ such that $y \to 0$

Compute $\tan^{-1}y$ by starting with x = 1 and z = 0

This computation always converges. However, one can take advantage of

$$\tan^{-1}(1/y) = \pi/2 - \tan^{-1}y \quad (1.38)$$

to limit the range of fixed-point numbers encountered.

2.6 GENERALIZED CORDIC

The basic CORDIC method discussed above can be generalized to provide a more powerful tool for function evaluation.

Generalized CORDIC is now defined as follows:

$$x(i+1) = x(i) - \mu d_i\, y(i)2^{-i}$$

[Gen. CORDIC iteration] $$y(i+1) = y(i) + d_i\, x(i)2^{-i} \quad (1.39)$$

$$z(i+1) = z(i) - d_i\, e(i) \quad (1.40)$$

μ is the decision factor which decides in which direction to rotate.

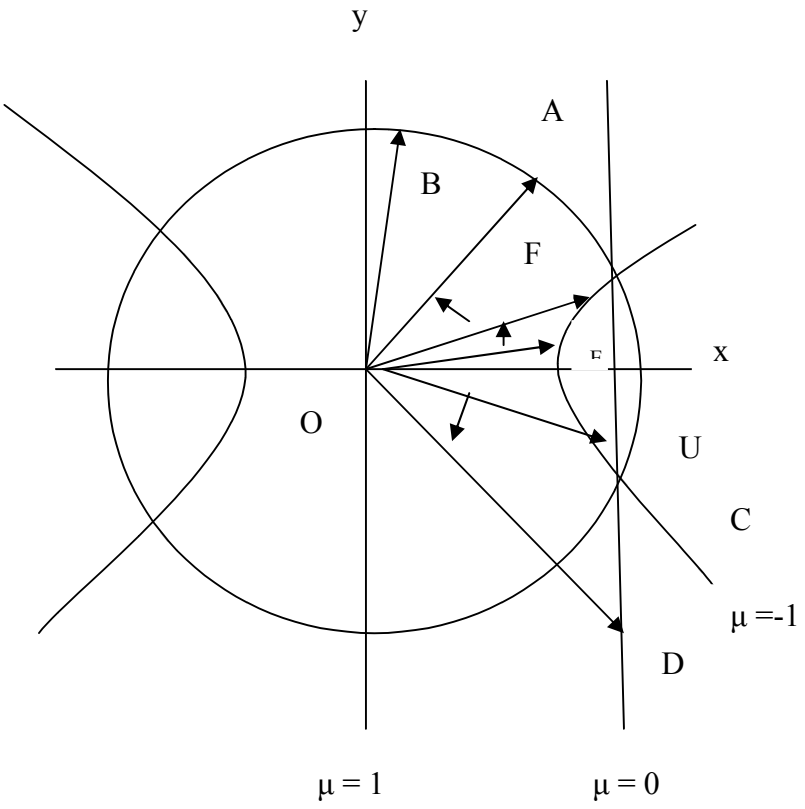


Figure: 2.7 - Circular, Linear, and Hyperbolic CORDIC

μ = 1 Circular rotations (basic CORDIC)

$$e(i) = \tan^{-1}2^{-i}$$

μ = 0 Linear rotations

$$e(i) = 2^{-i}$$

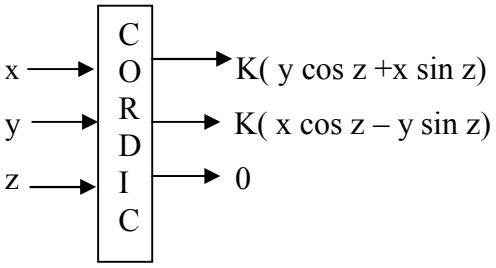
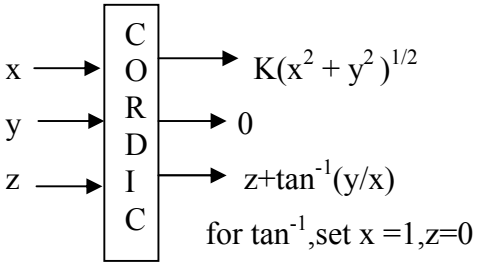
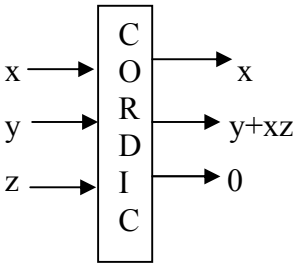
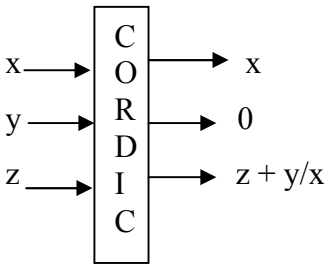
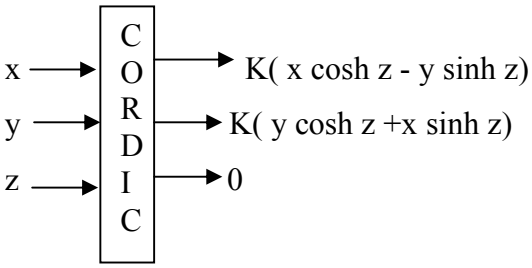
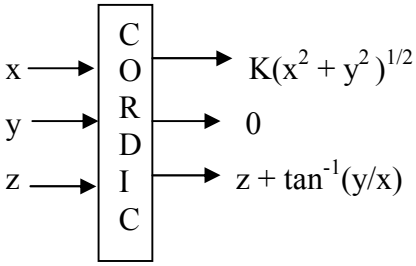
μ = –1 Hyperbolic rotations

$$e(i) = \tanh^{-1}2^{-i}$$

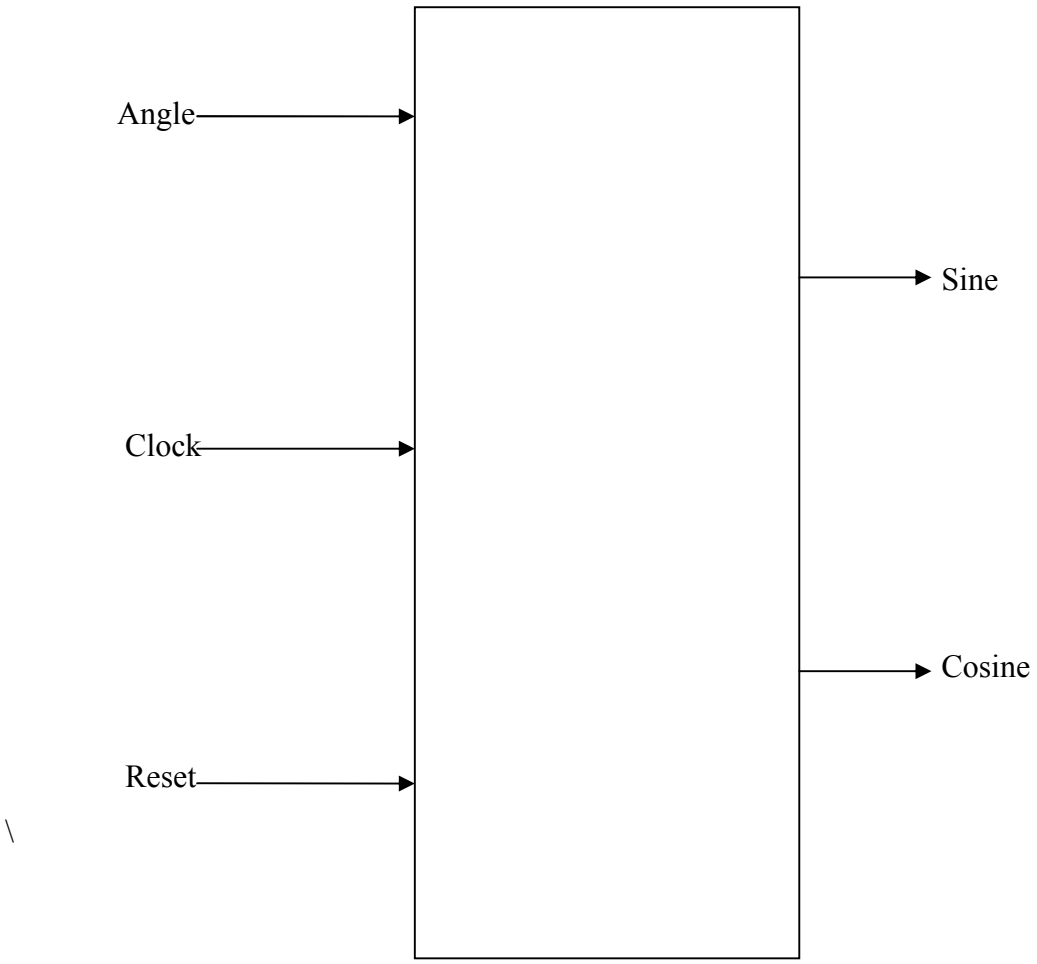
Table-2 below clearly shows the operation of CORDIC vector in both rotation and vectoring mode, for each

i. Circular rotations
ii. Linear rotations
iii. Hyperbolic rotations.

Table 2.2 – CORDIC Modes

	Rotation mode: d_i=sign(z(i)) z(i) → 0	Vectoring mode: d_i= - sign(x(i)y(i)) y(i) → 0
μ=1 circular $e(i) = \tan^{-1}2^{-i}$	 for cos & sin, set x=1/K, y=0	 for $\tan^{-1}$,set x =1,z=0 $\cos\text{-}1w=\tan^{-1}[((1-w^2)^{1/2})/w]$ $\sin\text{-}1w=\tan^{-1}[w/(1-w^2)^{1/2}]$
μ= 0 linear $e(i) =2^{-i}$	 for multiplication, set y=0	 for division, set z = 0
μ= -1 hyperbolic $e(i)=\tanh^{-1}2^{-i}$	 for cosh & sinh, set x=1/K', y=0 tanh z = sinh z /cosh z e^z = sinh z + cosh z $w^t=e^{t \ln w}$	 for $\tanh^{-1}$,set x=1,z = 0 ln w = 2 tanh-1[(w-1)/(w+1)] $w^{1/2} = [(w+1/4)^2-(w-1/4)^2]^{1/2}$ $\cosh^{-1}w= \ln[w + (1- w^2)^{1/2}]$ $\sinh^{-1}w=\ln[w + (1+ w^2)^{1/2}]$

TOP LEVEL PROCESSOR DIAGRAM OF A CORDIC



\

Figure: 2.8- Pin diagram of CORDIC Algorithm

2.7 APPLICATIONS OF CORDIC

CORDIC is an acronym for COrdinate Rotation DIgital Computer and was developed by J.E. Volder in 1959. The CORDIC algorithm is an example of an approach that is quite different from traditional math and which is very efficient. The concept of CORDIC is to take an angle θ and rotate a vector over this angle towards zero in a series of steps such that the sum of all the steps taken equals θ and we can accumulate corresponding X and Y increments for each step such that when we complete the process, Y/X = tan (θ). Variations of this basic concept can result in easy calculation of all the forward and inverse trigonometric functions as well as square root, hyperbolic trigonometric functions, and the exponential and logarithmic functions.

2.7.1 SIN & COS Function

CORDIC can be used to compute Sin of any angle θ with little variation. The angle is given as input. A vector of length 1.647 (CORDIC gain) along the x-axis is taken. The vector is then rotated in steps so as to reach the desired input angle θ. The x and y values are accumulated. After fixed number of iterations the final coordinates of the vector i.e. the x and y values give the values of Cos and Sin respectively of the given angle θ. This is what is done in this thesis.

2.7.2 TANGENT

Tangent of an input angle can be easily calculated using CORDIC by dividing the accumulated values of Y and X after rotation by the desired angle θ.

$$Y/X = \tan\theta \qquad (1.41)$$

2.7.3 INVERSE TRIGONOMETRIC FUNCTIONS

Give the values of x and y the inverse trigonometric functions can also be calculated. To get the inverse tangent following initial values of constants can be taken.

Set K=0
θ = 0
X=1

Value of Y is given as input. The vector is rotated using CORDIC iterations in steps until the final value of Y is reached equal to zero. The angle traced during this rotation is the inverse tangent of the value given to Y.

2.7.4 PHASE

To calculate phase, just rotate the vector to have zero phase, as done to calculate magnitude. The process is different by just a small bit.

1. For each phase-addition/subtraction step, accumulate the actual number of degrees (or radians) that are rotated. The values are taken from table of atan K. The phase of the

complex input value is the negative of the accumulated rotation required to bring it to a phase of zero.

2. The CORDIC gain can be skipped if only the phase is to be calculated.

2.7.5 MAGNITUDE

Magnitude can be calculated using CORDIC algorithm. The magnitude of a complex number $C = I_c + jQ_c$ can be calculated if it is rotated to have a phase equal to zero; then its new Qc value would be zero, so the magnitude would be given entirely by the new I_c value. The procedure to rotate the vector is given below:

1. It can be determined whether or not the complex number C has a positive phase just by looking at the sign of the Q_c value. If Q_c is positive it means phase is positive. As the very first step, if the phase is positive, rotate it by -90 degrees; if it is negative, rotate it by +90 degrees. To rotate by +90 degrees, just negate Q_c, then swap I_c and Q_c; to rotate by -90 degrees, just negate I_c, then swap. The phase of C is now less than +/- 90 degrees, so the 1 +/- jK rotations to follow can rotate it to zero.
2. Next a series of iterations is performed with successively smaller values of K, starting with K=1 (45 degrees). For each iteration, simply look at the sign of Q_c to decide whether to add or subtract phase; if Q_c is negative, add a phase (by multiplying by 1 + jK); if Q_c is positive, subtract a phase (by multiplying by 1 - jK). The accuracy of the result converges with each iteration. The more the iterations are performed, the more accurate results are obtained.

Since each phase is a little more than half the previous phase, this algorithm is slightly underdamped. It could be made slightly more accurate on average, for a given number of iterations, by using ideal K values which would add/subtract phases of 45.0, 22.5, 11.25 degrees, etc. However, then the K values wouldn't be of the form 2^{-i}, they are supposed to be 1.0, 0.414, 0.199, etc., and multiplication using just shift/add operations is then not possible (which would eliminate the major benefit of the algorithm). In practice, the difference in accuracy between the ideal Ks and these binary Ks is generally negligible, therefore, for a multiplier-less CORDIC, use the binary Ks, and if more accuracy is needed, just increase the number of iterations or increase the number of signed powers of two to get better results. So after the said rotation is performed to reduce the phase of the vector to zero, the resulting

complex number obtained after this entire operation is $C = I_c + j0$. The magnitude of this complex value is just Ic (since Q_c is zero.) However, in the rotation process, C has been multiplied by a CORDIC Gain (the cumulative magnitude) of about 1.647. Therefore, to get the true value of magnitude we must multiply by the reciprocal of 1.647, which is 0.607. (the value of exact CORDIC Gain is a function of the how many iterations are performed) Unfortunately, we can't do this gain-adjustment multiply using a simple shift/add; however, in many applications this factor can be compensated for in some other part of the system. When relative magnitude is all that counts (e.g. AM demodulation), it can simply be neglected.

NOTE: Since magnitude and phase are both calculated by rotating to a phase of zero, both operations can be done simultaneously.

2.7.6 POLAR TO RECTANGULAR TRANSFORMATION

A logical extension to the sine and cosine computer is a polar to Cartesian coordinate transformer. The transformation from polar to Cartesian space is defined by

$$x = r \cos\theta$$

$$y = r \sin\theta$$

As pointed out above, the multiplication by the magnitude comes for free using the CORDIC rotator. The transformation is accomplished by selecting the rotation mode with x_0 = polar magnitude, z_0 = polar phase, and $y_0 = 0$. The vector result represents the polar input transformed to Cartesian space. The transform has a gain equal to the rotator gain, which needs to be accounted for somewhere in the system. If the gain is unacceptable, the polar magnitude may be multiplied by the reciprocal of the rotator gain before it is presented to the CORDIC rotator.

2.77 EXTENSION TO HYPERBOLIC FUNCTIONS

The close relationship between the trigonometric and hyperbolic functions suggests the same architecture can be used to compute the hyperbolic functions. While, there is early mention of using the CORDIC structure for hyperbolic coordinate transforms, the first description of the algorithm is that by Walther [1]. The CORDIC equations for hyperbolic rotations are derived using the same manipulations as those used to derive the rotation in the circular coordinate system. For rotation mode these are y and z registers (these registers could also be parallel loaded to initialize). Once loaded, the data is shifted right through the serial adder-subtractors

and returned to the left end of the register. At the beginning of each iteration, the control state machine reads the sign of the y.[6]

2.78 APPLICATION TO DSP ALGORITHMS

Linear transformation: DFT, Chirp-Z transform, DHT and FFT.

FFT application:

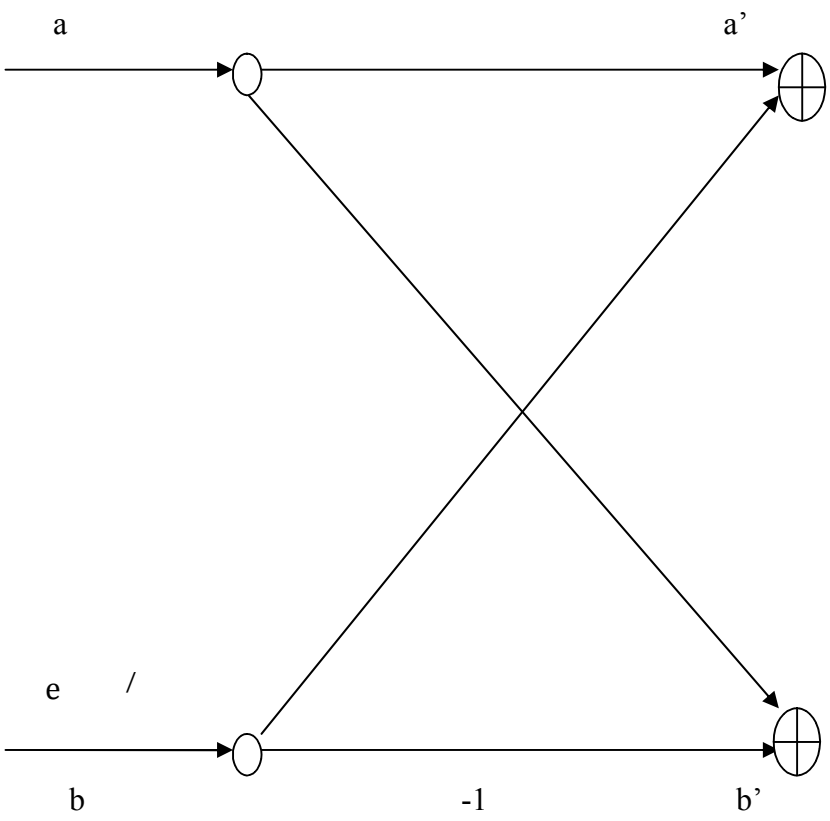


Figure: 2.9 - FFT

Digital filters

Orthogonal digital filters, and adaptive lattice filters.

2.8 IMPLEMENTATION OF CORDIC PROCESSOR

There are no of ways to implement CORDIC processor. The ideal architecture depends upon the speed versus area tradeoffs in the intended application. First the iterative architecture that

is direct translation from the CORDIC equations is to be examined. From there, minimum hardware solution and maximum performance solution can be easily found..

2.8.1 ITERATIVE CORDIC PROCESSOR

An iterative CORDIC architecture can be obtained by duplicating each of three difference equations in hardware as shown in figure 2.10 .The decision function d_i is driven by the sign of y an z registers depending on whether it is operating in rotation or vectoring mode. In operation, the initial values are loaded via multiplexer in x, y, z registers. Then each of the n clock cycle, the values from the registers are passed through the shifters and adders–subtractor and result backed in to the registers. The shifters are modified on each iteration so that to cause the desired shift for the iteration. Likewise, the ROM address is increment on each iteration so that appropriate elementary angle value is presented to the z add-sutractor. On the last iteration the result are read from adder-subtractor. A simple state machine is required to keep the track of the current iteration and to select the degree of shift and ROM address for each iteration.

The design depicted in Figure 2.10 uses word wide data paths called bit – parallel design. The bit-parallel variable shift shifter do not map well to FPGA architecture because of high fan in required. If implemented those shifters will required several layers of logic. (i.e. the signal need to pass through a number of FPGA cells.) The result is a slow design that uses a large number of logic cells.

A considerable more compact design is possible using bit serial arithmetic. The simplified interconnect and logic in a bit serial design allows it to work at a much higher clock rate than the equivalent bit parallel design. The design need to clocked w times for each iteration where w is the width of the data word. The bit–serial design consists of a three bit serial adder subtractor, three shift registers and a serial rom. Each shift register has a length equal to word width. There is also some gating or multiplexer to select taps off the shift registers for the right shifted cross terms (shifting is accomplished using bit delays in bit serial systems). The bit serial CORDIC architecture is shown in Figure 2.10. In this design, w clocks are required for each of the n iterations, where w is precision of the adders. In operation, the load multiplexers on the left are opened for w clock periods to initialize the x, (or z) register and sets the add/subtract controls accordingly. The appropriate tap off the register for the cross terms is also selected at the beginning of each iteration. During the nth iteration, the results can be read from the outputs of the serial adders while the next initialization data is shifted into the registers. The simplicity of the bit serial design is

apparent from figure 2.10. Even in this case, the wiring of the shift tap multiplexers can present problems in some FPGAs (this is one place where tri-state long lines can come in handy). Even so, the interconnect is minimal and the logic between registers is simple. This combination permits bit clock rates near the maximum toggle frequency of the FPGA. The possibility of using extreme bit clock frequencies makes up for the large number of clock cycles required to complete each rotation.

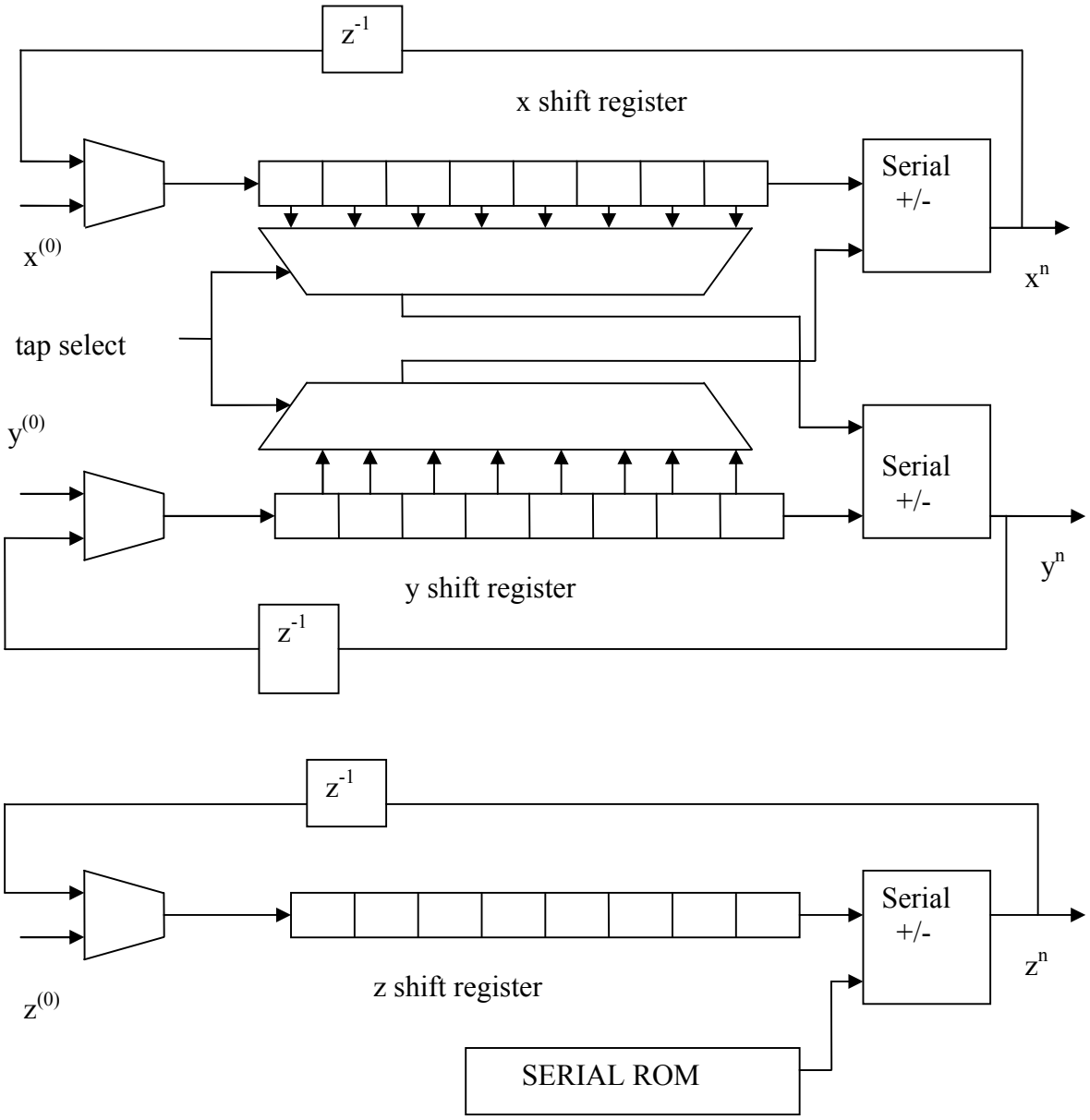


Figure: 2.10 - Bit Serial Iterative CORDIC

Now, if the design is in a Xilinx 4000E series part, the shift registers can be implemented in the CLB RAM. The RAM emulates a shift register by incrementing the read/write address

after each access. The dual port capability of the CLB RAM provides the capability to read two locations in the 16x1 RAM simultaneously. By properly sequencing the second address, the effect of the shift tap multiplexer is achieved without a physical multiplexer. The result is the shift register and multiplexer for word lengths up to 16 bits are implemented in a single CLB (plus 8 CLBs for the 2 address sequencers and iteration counter, which are shared by the three shifters). The serial ROM also uses the CLB for data storage. One CLB is required for every two iterations. The 16 bit, 8 iteration CORDIC processor uses only 21 CLBs, and will run at bit rates up to about 90 MHz (mainly limited by the RAM write cycle).[7]

2.8.2 ON-LINE CORDIC PROCESSORS

The CORDIC processors discussed so far are iterative, which means the processor has to perform iterations at n times the data rate. The iteration process can unroll so that each of n processing elements always performs the same iteration. An unrolled CORDIC processor is shown in Figure below. Unrolling the processor results in two significant simplifications. First the shifters are each a fixed shift, which means that they can be implemented in the wiring. Second, the lookup values for the angle accumulator are distributed as constants to each adder in the angle accumulator chain. Those constants can be hardwired instead of requiring storage space. The entire CORDIC processor is reduced to an array of interconnected adder-subtractors. The need for registers is also eliminated, making the unrolled processor strictly combinatorial. The delay through the resulting circuit would be substantial, but the processing time is reduced from that required by the iterative circuit (if by nothing else than the set-up and hold times of the register). Most times, especially in an FPGA, it does not make sense to use such a large combinatorial circuit. The unrolled processor is easily pipelined by inserting registers between the adder-subtractors. In the case of most FPGA architectures there are already registers present in each logic cell, so the addition of the pipeline registers has no hardware cost. The unrolled processor can also be converted to a bit serial design. Each adder subtractor is replaced by a serial adder subtractor, separated by w bit shift registers. The shift registers are necessary to extract the sign of the y or z element before the first bits (LSBs) reach the next adder-subtractors. The rights shifted cross terms are taken from fixed taps in the shift registers. Some method of sign extension for the shifted terms is required too. Figure at the end shows two iterations of a bit serial CORDIC processor implemented in an Atmel 6005 or NSC Clay31 FPGA. Notice the cross term is taken from different taps off the shift register at each iteration. This particular processor is used to compute vector magnitude. Since this is a vector mode process and the

result angle is not required, there is no need for an angle accumulator. Figure 2.10 shows the detail of the adder-subtractor for that design.

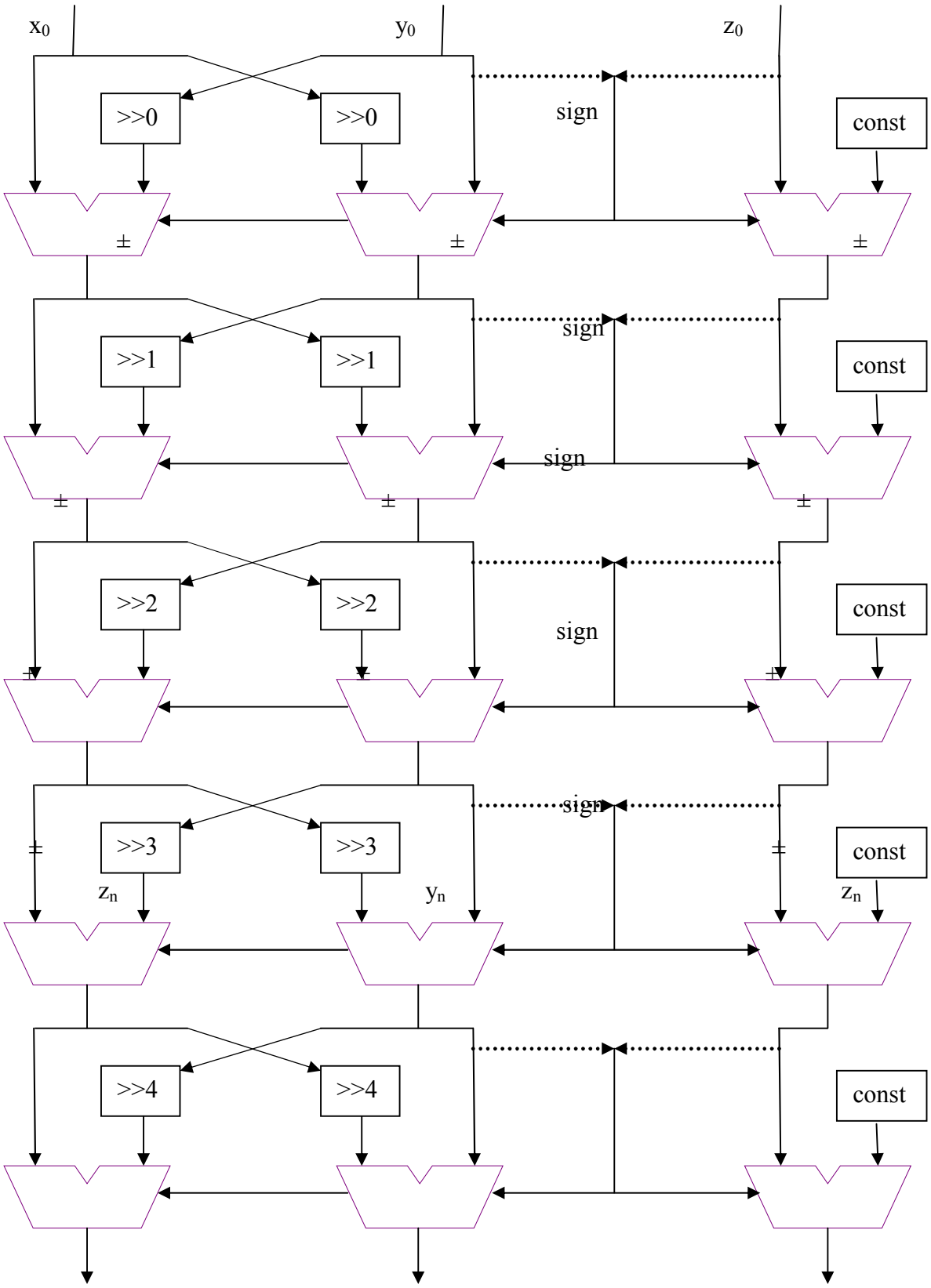


Figure 2.11-5 iteration pipelined CORDIC processor

The adder subtractor in this case includes logic to extend the sign of the shifted cross term and to reset the adder subtractor between words. The entire 7 iteration design occupies approximately 20% of the FPGA and runs at bit rates up to 125 MHz. Higher performance requires either multiple bit serial processors running in parallel, or an unrolled parallel pipeline. Until recently, FPGAs did not have the required combination of logic and routing resource to build a parallel processor. This barrier is mostly due to the large amount of cross routing required between the x and y registers at each pipeline stage. Additionally, the performance diminishes as the word width is increased because of the carry propagation times across the adders. The Xilinx 4000E series has sufficient routing to realize a reasonably compact parallel CORDIC pipeline. Its dedicated carry logic provides acceptable performance for the adders. Figure 2.11 shows a 14 bit, 5 iteration pipelined CORDIC processor that fits comfortably in half of a 4013E. That design, used for polar to Cartesian coordinate transformations in a radar target generator, runs at 52 MHz (clock rate and data rate) in an XC4013E-2.

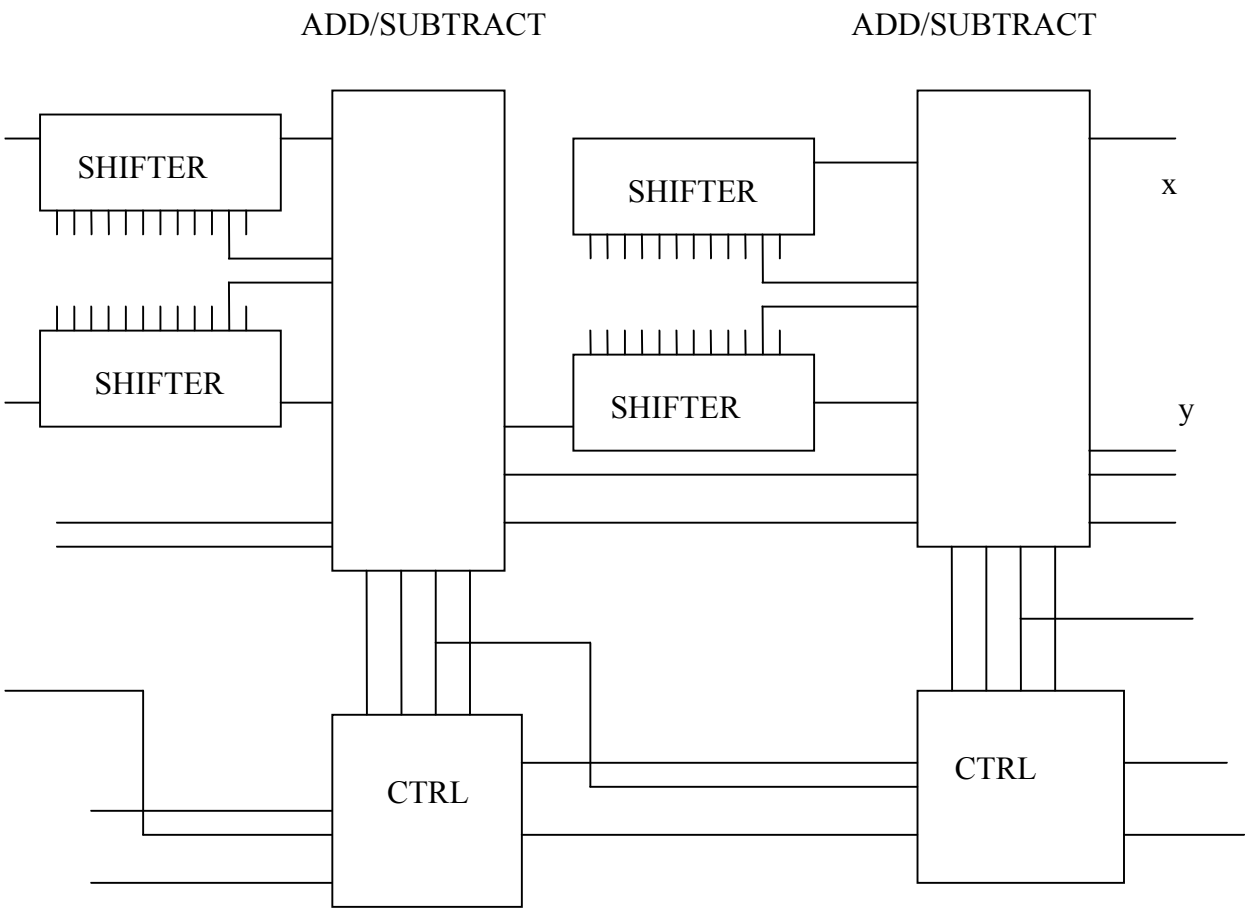


Figure: 2.12 - Two Iterations of Bit Serial CORDIC Pipeline

CHAPTER 3
CORDIC WITH EEAS SCHEME

3.1 HOW TO IMPROVE CORDIC

1. Use Pipelined Architecture Improve the Performance of the Adders (redundant arithmetic). Unfolding also enhances speed of operation.
2. Reduce Iteration Number.
3. High radix CORDIC.
4. Find a optimized shift sequence.
5. Improve the Scaling Operation.

3.2 ANGLE RECODING TECHNIQUE

The CORDIC algorithm decomposes the rotation angle, θ, into a combination of pre-defined elementary angles [l], i.e.

$$\theta = \sum \mu(i)a(i) + e \qquad (3.1)$$

where N is the number of elementary angles, μ (i) ϵ (-1, 1) is the rotation sequence which determines the direction of the i^{th} elementary angle of $a(i) = \tan^{-1}(2^{-i})$, and e denotes the residue angle. Based on Eq. (I), the recurrence equations of the CORDIC algorithm can be written as below

$$x(i+1) = x(i) - \mu(i)\, y(i)2^{-i}$$
$$y(i+1) = y(i) + \mu(i)\, x(i)\, 2^{-i} \qquad (3.2)$$
$$\text{for } i = 0,1,\ldots\ldots N-1.$$

Also, the final values, z (N) and y (N) need to be scaled by scaling factor,

$$P = (\prod \overline{1 + 2 \quad}) \qquad (3.3)$$

to retain the norm of the initial vector [x(0), y(0)]. From Equation (3.2), it is known that in the conventional CORDIC algorithm, each elementary angle needs to be performed sequentially so as to complete the micro-rotation phase. However, in the applications where the rotation angles are known in advance, it would be advantageous to relax the sequential constraint on the micro-rotation phase. The Angle Recoding (AR) technique is done by

extending the set of μ(i) from (1, -1) to (1, -l, 0) [2]. With the relaxation on, μ(i), for certain angles, we can obtain better approximation of θ (i.e. smaller residue angle, e) but with reduced iteration number. For example, consider the target angle $\theta = \pi/4$. The conventional CORDIC approach has to go through all the N micro-rotations with μ(i) sequence = [11,-1, 1, 1, 1,..]. The residue angle, e, is 7.2×10^{-3} for N = 8. However, with the AR technique, we can rotate $\theta = \pi/4$ by performing only one micro-rotation of elementary angle $\theta(0) = \tan^{-1}(2^0)$, and skipping all remaining micro-rotations. The resulting μ(i) sequence = [l,0,0,0,0,0, . . …..], and it takes only one iteration with the residue angle e = 0.[8]

3.3 IMPROVED CORDIC ALGORITHM (EEAS)

First, the AR technique presented in the basic CORDIC imposes no restriction on the iteration number. As one can expect that rotation angles of different values may need unequal number of iterations which may lead to bus timing alignment problems in VLSl circuits. In this thesis, one additional parameter, the maximum iteration number R_m to limit the number of iterations is introduced[9]. By doing so, the total iterations number in the micro-rotation phase can be held fixed for various rotation angles θ. Now, the AR problem with fixed iterations number can be summarized as

Given a target angle θ and the maximum iteration number R_m, & find the rotation sequence

$$\mu(i) \in \{1, -1, 0\} \text{ for } 0 \leq i \leq N \qquad (3.4)$$

such that the residue angle error.

$$\xi_m = \theta_i - \sum \mu(i).\tan^{-1}(2^{-i}) \qquad (3.5)$$

is minimized subject to the constraint that the total iterations number

$$\sum |\mu(i)|. \leq R_m \qquad (3.6)$$

3.4 REFORMULATION OF THE ANGLE RECODING PROBLEM

Moreover, to facilitate the derivation of the proposed EEAS scheme(X), the AR problem described above can be written in an alternative form as

$$\xi_m = \theta_i - \sum . \tan^{-1} (\alpha(j).2^{-s(j)}) \qquad (3.7)$$

where

1. j, $0 \leq j \leq R_m - 1$, denotes the iteration index,
2. $s(j) \in \{ 0 , 1 , . . . , N - 1\}$ is the rotational sequence that determines the micro-rotation angle in the j^{th} iteration,

3. $\alpha(j) \in \{-1, 0, 1\}$ is the directional sequence that controls the direction of the j^{th} micro-rotation of $\alpha(s(j))$,

Equation 3.7 shows that the reformulated angle recording scheme is to find the combination of elements from a set, which consists of all possible value of $\tan^{-1}(\alpha(j).2^{-s(j)})$, so that ξ_m can be minimized. This set of angles is called the elementary angles set (EAS) S_1, defined as

$$S_1 = \{tan^{-1}(\alpha.2^{-s}): \quad \alpha \in \{-1, 0, 1\}, s \in \{0, 1, \ldots\ldots, N-I\}\} \tag{3.8}$$

The subscript with S is used to denote the number of SPT terms. By doing so, the AR problem becomes: Given θ and R_m, find the combination of elementary angles from elementary angles set S_1, such that the residue angle error ξ_n is minimized.[10]

3.5 EXTENDED ELEMENTARY-ANGLE SET (EEAS) SCHEME

The constraint on the number of elementary angles is relaxed so as to extend the elementary angles set S_1. By doing this, more choices (elementary angles) can be achieved in approximating the target angle θ. Hence, it is expectable that the residue angle error ξ_m, can be reduced correspondingly.

First, by observing Eq. (3.8), we can see that the EAS S_1 are comprised of inverse tangent of single signed power of two (SPT) term $[\tan^{-1}(\alpha(j).2^{-s(j)}]$. To achieve increased speeds using CORDIC algorithm the angle set is extended by introducing another signed power of two (SPT) in the equation (3.9).

$$S_2 = \{tan^{-1}(\alpha_0.2^{\quad} + \alpha_1 2^{\quad}) \quad \alpha_0, \alpha_1 \in \{-1, 0, 1\}, \quad s_0, s_1 \in \{0, 1, \ldots\ldots, N-I\}\} \tag{3.9}$$

S_2 is called the extended elementary angles set (EEAS)

Effect of introduction of another signed power of two can be easily explained.

To understand the effect only positive powers of 2 are taken. As shown in Figure 3.1 below, the Figure 3.1 (a) shows the points that can be realized on the number line by single positive power of two, and just for the sake of explanation the 2^i values are taken up to 32 only. So we see that using single positive power of two only 6 points on the number line can be realized. But on the other hand using two positive powers of two Figure 3.1 (b) large number of more points can be realized on the line. For Example number 5 cannot be realized using single power of two. But $5 = 2^2 + 2^0$, $12 = 2^3 + 2^2$, ……etc. In this paper, motivated by the SPT representation of elementary angles, we proposed two novel schemes.[11] The EEAS scheme improves the error performance in the micro-rotation phase.

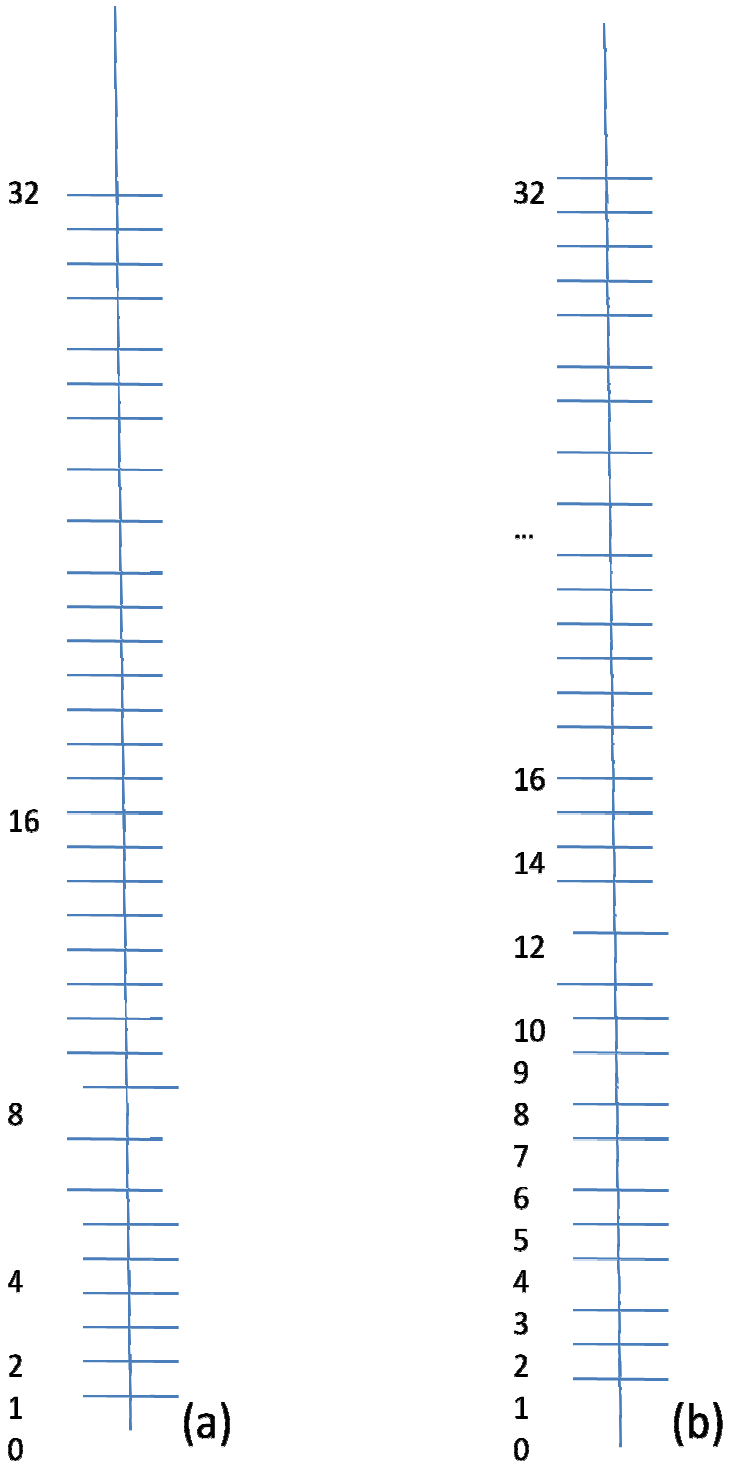


Figure 3.1-Effect of introduction of one more SPT

Vector rotation can be easily accomplished with only a few shift-and-add operations without sacrificing error performance. The significant improvement makes applications, which call for high complexity, feasible, such as high-point high-speed discrete transformations (FFT, DCT) and high-order digital lattice filters.

Chapter 4

FPGA IMPLEMENTATION

4.1 INTRODUCTION

FPGA stands for field programmable gate arrays that can be configured by the customer or designer after manufacturing. Field programmable gate arrays are called this because rather than having a structure similar to a PAL or other programmable device, they are structured very much like a gate array ASIC. This makes FPGAs very nice for use in prototyping ASICs, or in places where and ASIC will eventually be used[13].For example, an FPGA maybe used in a design that need to get to market quickly regardless of cost. Later an ASIC can be used in place of the FPGA when the production volume increases, in order to reduce cost. FPGAs are programmed using a logic circuit diagram or a source code in a hardware description language (HDL) to specify how the chip will work. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together". The programmable logic blocks are called configurable logic blocks and reconfigurable interconnects are called switch boxes. Logic blocks (CLBs) can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory[14]

4.2 FPGA Architecture

Each FPGA vendor has its own FPGA architecture, but in general terms they are all a variation of that shown in Figure 4.1. The architecture consists of configurable logic blocks, configurable I/O blocks, and programmable interconnect. Also, there will be clock circuitry for driving the clock signals to each logic block, and additional logic resources such as ALUs, memory, and decoders may be available. The two basic types of programmable elements for an FPGA are Static RAM and anti-fuses. An application circuit must be mapped into an FPGA with adequate resources. While the number of

CLBs and I/Os required is easily determined from the design, the number of 3routing tracks needed may vary considerably even among designs with the same amount of logic.[15]

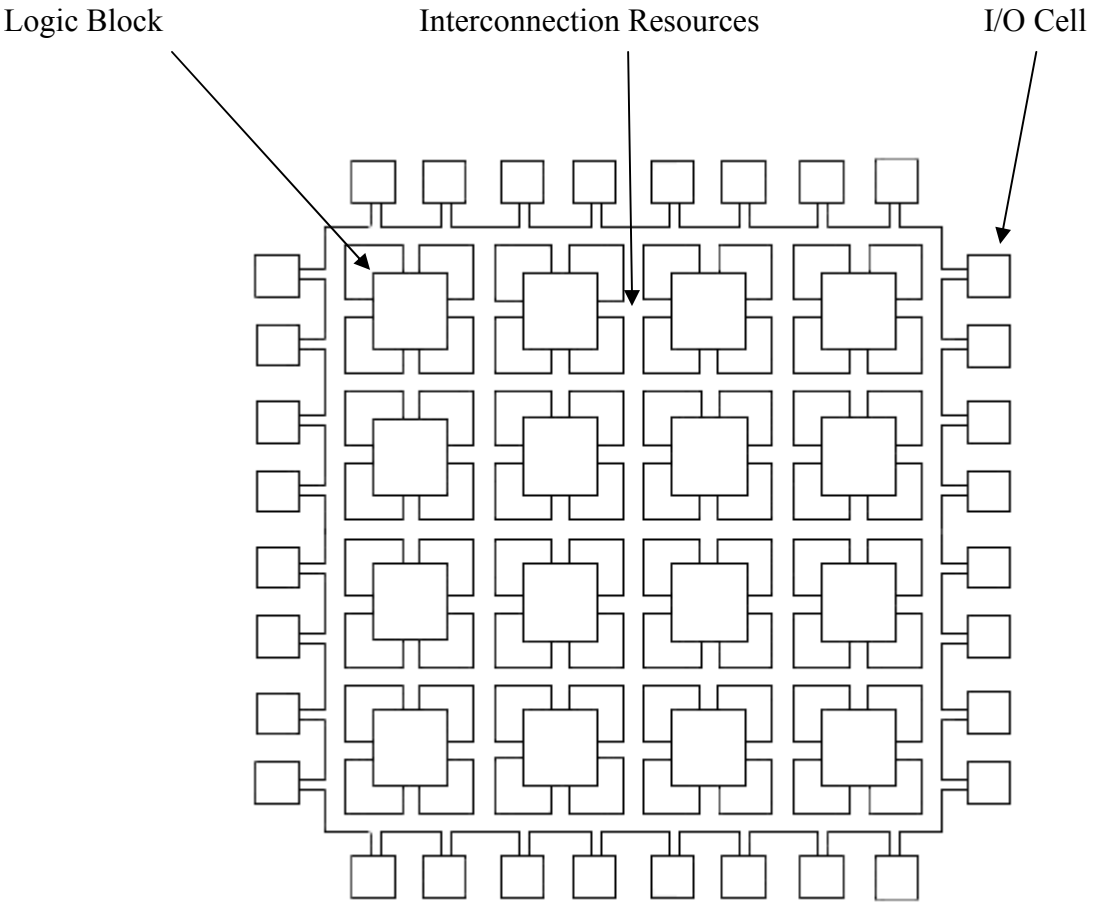


Figure: 4.1- FPGA Architecture

4.3 Configurable Logic Blocks

Configurable Logic Blocks contain the logic for the FPGA. In large grain architecture, these CLBs will contain enough logic to create a small state machine. In fine grain architecture, more like a true gate array ASIC, the CLB will contain only very basic logic[26]. The diagram in Figure 5.2 would be considered a large grain block. It contains RAM for creating arbitrary combinatorial logic functions. It also contains flip-flops for clocked storage elements, and multiplexers in order to route the logic within the block and to and from external resources. The multiplexers also allow polarity selection and reset and clear input selection.

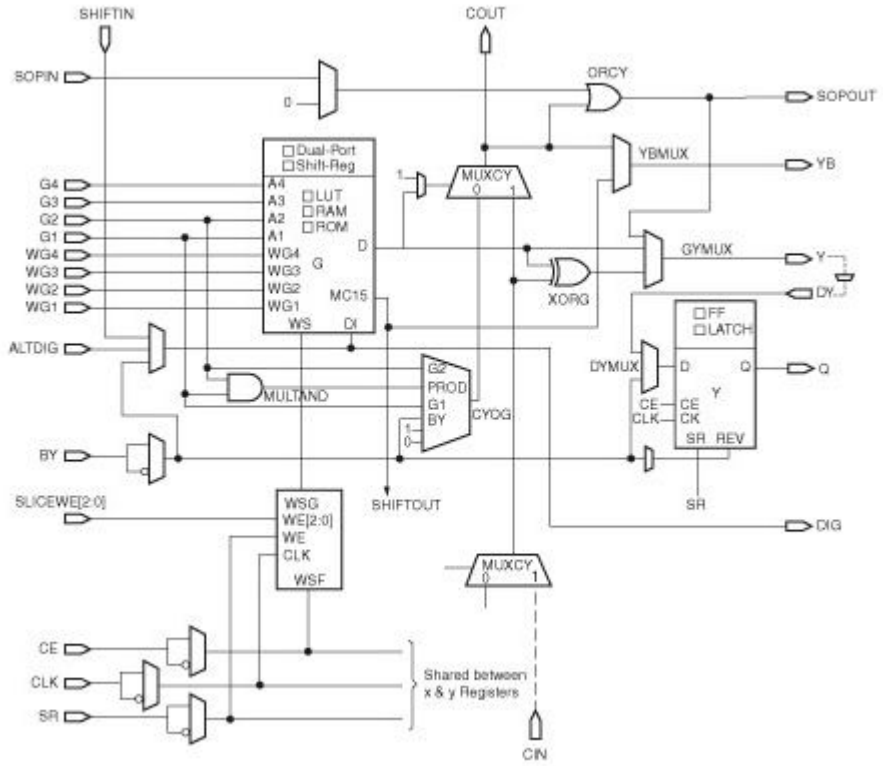


Figure: 4.2 - The Configurable Logic Block[15]

4.4 Configurable I/O Blocks

A Configurable I/O Block is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three state and open collector output controls. Typically there are pull up resistors on the outputs and sometimes pull down resistors. The polarity of the output can usually be programmed for active high or active low output and often the slew rate of the output can be programmed for fast or slow rise and fall times. In addition, there is often a flip-flop on outputs so that clocked signals can be output directly to the pins without encountering significant delay. It is done for inputs so that there is not much delay on a signal before reaching a flip-flop which would increase the device hold time requirement.

4.5 Programmable Interconnect

The interconnect of an FPGA is very different than that of a CPLD, but is rather similar to that of a gate array ASIC. In Figure 5.4, a hierarchy of interconnect resources can be seen. There are long lines which can be used to connect critical CLBs that are physically far from each other on the chip without inducing much delay. They can also be used as buses within the chip. There are also short lines which are used to connect

individual CLBs which are located physically close to each other. There is often one or several switch matrices, like that in a CPLD, to connect these long and short lines together in specific ways. Programmable switches inside the chip allow the connection of CLBs to interconnect lines and interconnect lines to each other and to the switch matrix. Three-state buffers are used to connect many CLBs to a long line, creating a bus. Special long lines, called global clock lines, are specially designed for low impedance and thus fast propagation times. These are connected to the clock buffers and to each clocked element in each CLB. This is how the clocks are distributed throughout the FPGA.

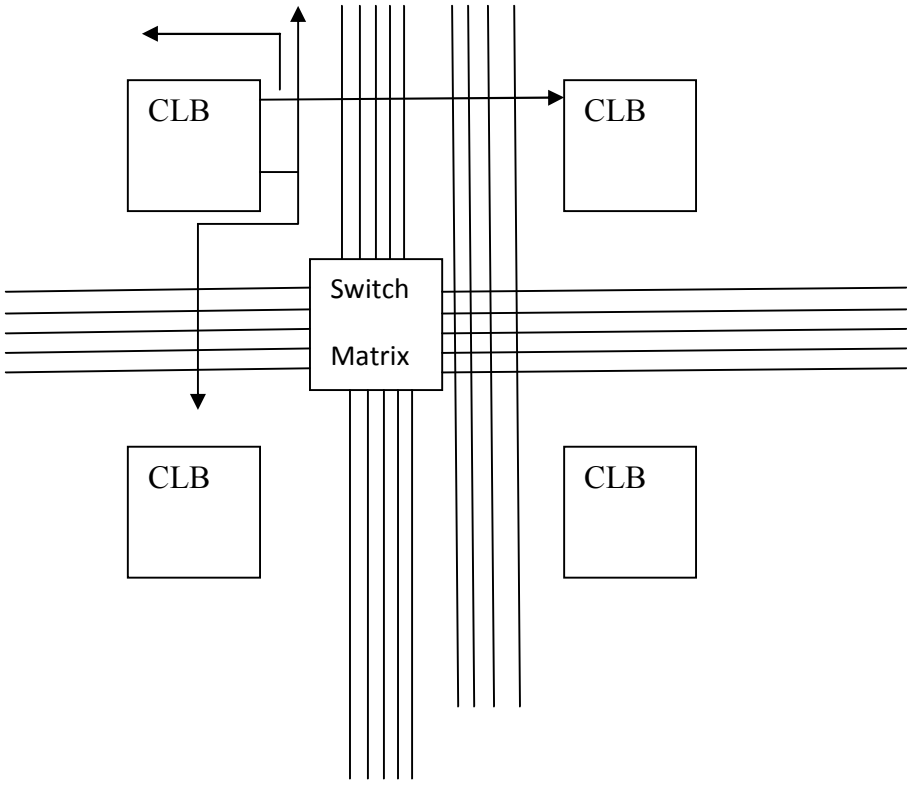


Figure: 4.3 - FPGA Programmable Interconnect

4.6 Clock Circuitry

Special I/O blocks with special high drive clock buffers, known as clock drivers, are distributed around the chip. These buffers are connected to clock input pads and drive the clock signals onto the global clock lines described above. These clock lines are designed for low skew times and fast propagation times. Synchronous design is a must with FPGAs, since absolute skew and delay cannot be guaranteed[16]. Only when

using clock signals from clock buffers can the relative delays and skew times are guaranteed.

4.7 The Design Flow

This section examines the flow for design using FPGA. This is the entire process for designing a device that guarantees that you will not overlook any steps and that you will have the best chance of getting backs a working prototype that functions correctly in your system[17]. The design flow consists of the steps in

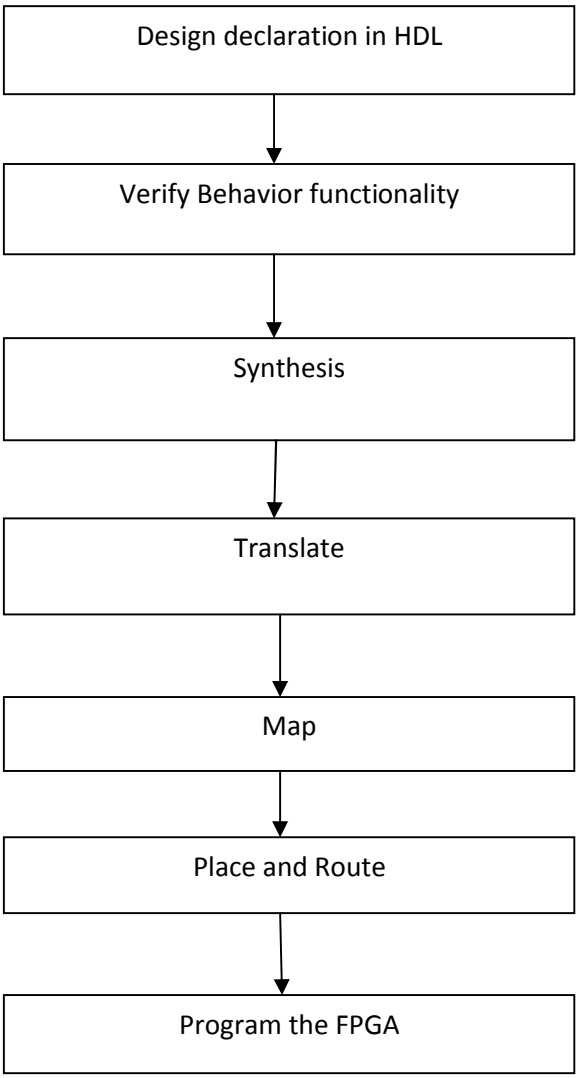


Figure: 4.4 - FPGA Design Flow

4.7.1 Design Entity

The basic architecture of the system is designed in this step which is coded in Hardware Description Language like VHDL or Verilog.

4.7.2 Behavioral Simulation

After the design phase, the code is verified using simulation software i.e. ModelSim or Xilinx ISE Simulator for different inputs to generate outputs and if it verifies then we will proceed further otherwise modification and necessary correction will be done in the HDL code. This is called as the behavioral simulation. Simulation is an ongoing process while the design is being done. Small sections of the design should be simulated separately before hooking them up to larger sections. There will be many iterations of design and simulation in order to get the correct functionality[18]. Once design and simulation are finished, another design review must take place so that the design can be checked. It is important to get others to look over the simulations and make sure that nothing was missed and that no improper assumption was made. This is one of the most important reviews because it is only with correct and complete simulation that you will know that your chip will work correctly in your system

4.7.3 Design Synthesis

After the correct simulation results the design is then synthesized. During synthesis the Xilinx ISE tool does the following operation.

(i) **HDL Compilation:** The tool compiles all the sub-modules in the main module if having any problem regarding module then check the syntax of the code written for the design.

(ii) **Design Hierarchy Analysis:** Analysis the hierarchy of the design.

(iii) **HDL synthesis:** Synthesizes the HDL code in terms of the components such as Multiplexer, Adder/Sub tractors, Counter, Register, Latches, Comparators, XORs tri state buffers, decoders etc.

(iv) **Advanced HDL Synthesis:** In low level synthesis, the blocks synthesized in the HDL synthesis and the Advanced HDL synthesis are further defined in terms of the low level blocks such as buffers, look up tables. It also optimizes the logic entities in the design by eliminating the redundant logic, if any. The tool then generates a netlist file (NGC file) and then optimize it. The final netlist output file has an extension of .ngc. This NGC file contains both the design data and constraints. The optimization goal can be pre defined to be the faster speed of operation or the minimum area of implementation before running this process. The level optimization effort requires larger CPU times (i.e. the design time) because multiple optimization algorithms are tried to get the best result for the target architecture.[19]

4.7.4 Design Implementation

The design implementation process consists of the following sub process:

(i) **Translation:** The translate process merges all of the input netlist and design constraint outputs a Xilinx NGD (Native information and Generic Database) file. The .ngd file describes the logical design reduced to the Xilinx device primitive cells. The output in the floor planner tool supplied with Xilinx ISE software. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design[20]. This information is stored in a file named UCF (User Constraints File). Tools used to create or modify the UCF are PACE, Constraint Editor Etc.

(ii) **Mapping:** The map process is run after the translate process is complete. Mapping maps the logical design described in the NGD file to the components/primitives (Slices/CLBs) present on the NCD file created by the Map process to place and route the design on the target FPGA design. In this process the whole circuit is divided into sub blocks so that they can fit into FPGA logic blocks. The logic defined in the input NGD file is mapped into targeted FPGA elements i.e. CLBs and IOBs and an output NCD

(native circuit description) file is generated which depicts the design mapped into the FPGA.

(iii) **Place and Route:** After map the design is placed and routed. Place and Route **(**PAR) program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. Example if a sub block is placed in a logic block which is very near to IO pin, then it may save the time but it may affect some other constraint. So tradeoff between all the constraints is taken account by the place and route process .The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consist the routing information. During the place process the sub blocks are placed according to the logic but these blocks do not have physical routing among them and with I/O pads also but there is only a logical connection between them which can be clearly seen using the FPGA Editor just after the place process ends. These logical connections are shown by rats nets in FPGA Editor. Then the Route process is run which makes physical connections between the sub blocks placed on FPGA. The connections are made using the switch matrices.

(iv) **Bitstream Generation:** The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a bit stream, although this is somewhat misleading because the data are no more bit oriented than that of an instruction set processor and there is generally no "streaming". While in an instruction set processor the configuration data are in fact continuously streamed into the internal units, they are typically loaded into the reconfigurable logic device only once during an initial setup phase.

(v) A programming file is generated by running the Generate Programming File process. This process can be run after the FPGA design has been completely routed. The Generate Programming File process runs BitGen, the Xilinx bitstream generation program, to produce a bitstream (.BIT) or (.ISC) file for Xilinx device configuration. The FPGA device is then configured with the .bit file using the JTAG boundary scan method. After the Spartan device

is configured for the intended design, then its working is verified by applying different inputs.

(vi) **Functional Simulation:** Post Translate (functional) simulation can be performed prior to mapping of the design. This simulation process allows the user to verify that your design has been synthesized correctly and any differences due to the lower level of abstraction can be identified.

(vii) **Static timing Analysis:** Three types of static timing analysis can be performed that are:

(a) **Post fit Static timing analysis:** The timing results of the Post fit process can be analyzed. The analyze Post fit Static Timing process opens the Timing Analyzer window, which lets you interactively select timing paths in your design for tracing.

(b) **Post map Static Timing Analyze:** You can analyze the timing results of the Map process. Post Map timing reports can be very useful in evaluating timing performance[20]. Although route delays are not accounted for the logic delay can provide valuable information about the design. If logic delay account for a significant portion (> 50 %) of the total allowable delay of a path, the path may not be able to meet your timing requirements when routing delays are added. Routing delay typically account for 45% to 65% of the total path delays. By identifying problem path, you can mitigate potential before investing time in place and route. You can redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the path. If logic only delays account for much less (>35%) than the total allowable delay for a path or timing constraint, then the place-route software can use very low placement effort levels. In these cases, reducing effort levels allow you to decrease runtimes while still meeting performance requirements.[21]

(viii) Testing

For a programmable device, you simply program the device and immediately have your prototypes. You then have the responsibility to place these prototypes in your system and determine that the entire system actually works correctly. If you have followed the procedure up to this point, chances are very good that your system will perform correctly with only minor problems. These problems can often be worked around by modifying the system or changing the system software. These problems need to be tested and documented so that they can be fixed on the next revision of the chip. System integration and system testing is necessary at this point to insure that all parts of the system work correctly together. When the chips are put into production, it is necessary to have some sort of burn-in test of your system that continually tests your system over some long amount of time. If a chip has been designed correctly, it will only fail because of electrical or mechanical problems that will usually show up with this kind of stress testing. [22]

4.7.5 Implementing the design on FPGA

The FPGA that is used for the implementation of the circuit is the Xilinx Spartan3E XC3S500 family FG320 (package) FPGA device. The working environment/tool for the design is the Xilinx ISE 9.1.Xilinx ISE simulator is used for the Behavioral, Post translate, Post map and Post place & route simulation of VHDL model

Spartan-3E FPGA specific features

- Parallel NOR Flash configuration
- MultiBoot FPGA configuration from Parallel NOR Flash PROM
- SPI serial Flash configuration

Embedded development

- MicroBlaze 32-bit embedded RISC processor
- PicoBlaze 8-bit embedded controller
- DDR memory interfaces

Implement the design and verify that it meets the timing constraints after check syntax and synthesis.

4.8 The Flow for EEAS CORDIC

1. Select the cordic_lcd source file in the source window.
2. Synthesize the design by double clicking the synthesize option. The design report can be seen in the project directory with the .syr extension (SYR file).
3. Open the User Constraints menu and double click Assign package pins, XILINX pace opens which shows the top level pictorial view of the FPGA in the architecture view. The ports are be assigned by dragging and dropping the ports. This can also be done using Edit Constraints (text) option in the User Constraints menu. Other constraints are also provided here. XILINX PACE generates the User Constraints File (.ucf). Select File to Save. You are prompted to select the bus delimiter type based on the Synthesis tool you are using. Select XST Default and click OK. Close PACE. Notice that the Implement Design processes have an orange question mark next to them, indicating they are out-of-date with one or more of the design files. This is because the UCF File has been modified
4. Then open the pull down menu Implement Design and the design is mapped placed and routed automatically bye ISE. If any changes are desired in the physical constraints or the user constraints FPGA Editor can be used to see the placed and routed design and can be accordingly edited.
5. Download Design to the Spartan-3E Demo Board. This is the last step in which the design is downloaded into the Spartan 3E FPGA Starter kit and the output is viewed using LCD and XILINX Chipscope.
   a. Connect the 5V DC power cable to the power input on the demo board (J4).
   b. Connect the download cable between the PC and demo board (J7).
   c. Select Implementation from the drop-down list in the Sources window.
   d. Select lcd_fp_sqrt in the Sources window.
   e. In the Process window, double-click the Configure Target Device process.
   f. The Xilinx web talk Dialog box may open during this process. Click Decline.
   g. In the Welcome dialog box, select Configure devices using Boundary-Scan (JTAG).

h. Verify that automatically connect to a cable and identify Boundary-Scan chain is selected.

i. Click Finish.

j. If you get a message saying that there are two devices found, click OK to continue. The devices connected to the JTAG chain on the board will be detected and displayed in the impact window.

k. The Assign New Configuration File dialog box appears. To assign a configuration file to the XC3S500E device in the JTAG chain, select the counter.bit file and click Open.

l. If you get a Warning message, click OK.

m. Select Bypass to skip any remaining devices.

n. Right-click on the XC3S500E device image, and select Program. The Programming Properties dialog box opens.

o. Click OK to program the device and the program succeeded message appears on the screen the output can be viewed on the LCD screen.

4.9 Design Issues

In the next sections of this chapter, we will discuss those areas that are unique to FPGA design or that are particularly critical to these devices.

4.9.1 Top-Down Design

Top-down design is the design method whereby high level functions are defined first, and the lower level implementation details are filled in later. A schematic can be viewed as a hierarchical tree as shown in Figure 4.5. The top-level block represents the entire chip. Each lower level block represents major functions of the chip. Intermediate level blocks may contain smaller functionality blocks combined with gate-level logic. The bottom level contains only gates and macro functions which are vendor-supplied high level functions. Fortunately, schematic capture software and hardware description languages used for chip design easily allows use of the top down design methodology.

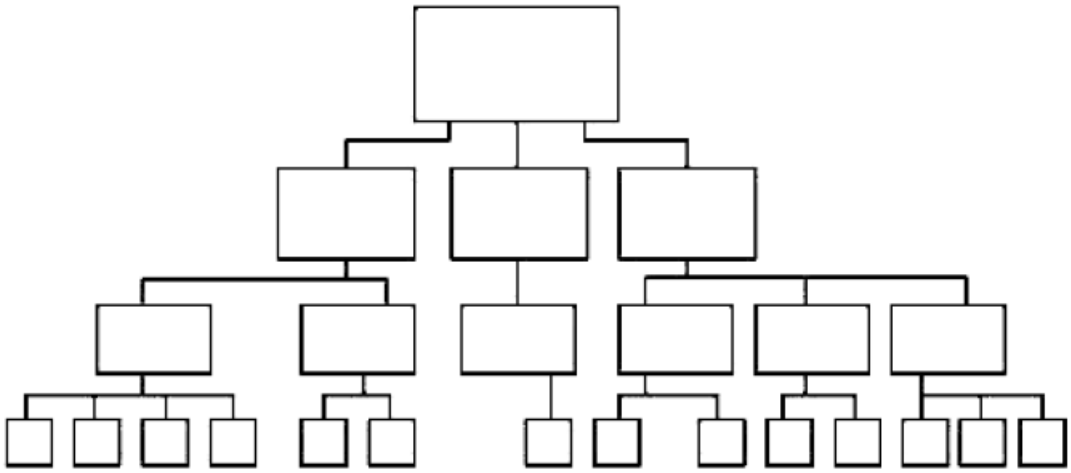


Figure: 4.5 - TOP Down Design

Top-down design is the preferred methodology for chip design for several reasons. First, chips often incorporate a large number of gates and a very high level of functionality. This methodology simplifies the design task and allows more than one engineer, when necessary, to design the chip. Second, it allows flexibility in the design. Sections can be removed and replaced with a higher-performance or optimized designs without affecting other sections of the chip. Also important is the fact that simulation is much simplified using this design methodology. Simulation is an extremely important consideration in chip design since a chip cannot be blue-wired after production. For this reason, simulation must be done extensively before the chip is sent for fabrication. A top-down design approach allows each module to be simulated independently from the rest of the design. This is important for complex designs where an entire design can take weeks to simulate and days to debug. Simulation is discussed in more detail later in this chapter.

4.9.2 Keep the Architecture in Mind

Look at the particular architecture to determine which logic devices fit best into it. The vendor may be able to offer advice about this. Many synthesis packages can target their results to a specific FPGA or CPLD family from a specific vendor, taking advantage of the architecture to provide you with faster, more optimal designs.

4.9.3 Synchronous Design

One of the most important concepts in chip design, and one of the hardest to enforce on novice chip designers, is that of synchronous design. Once a chip designer uncovers a problem due to asynchronous design and attempts to fix it, he or she usually becomes

an evangelical convert to synchronous design. This is because asynchronous design problems are due to marginal timing problems that may appear intermittently, or may appear only when the vendor changes its semiconductor process. Asynchronous designs that work for years in one process may suddenly fail when the chip is manufactured using a newer process. Synchronous design simply means that all data is passed through combinatorial logic and flip-flops that are synchronized to a single clock. Delay is always controlled by flip-flops, not combinatorial logic. No signal that is generated by combinatorial logic can be fed back to the same group of combinatorial logic without first going through a synchronizing flip-flop. Clocks cannot be gated - in other words, clocks must go directly to the clock inputs of the flip-flops without going through any combinatorial logic. The following sections cover common asynchronous design problems and how to fix them using synchronous logic.

4.9.4 Race conditions

Figure 4.6 shows an asynchronous race condition where a clock signal is used to reset a flip-flop. When SIG2 is low, the flip-flop is reset to a low state. On the rising edge of SIG2, the designer wants the output to change to the high state of SIG1.Unfortunately, since we don't know the exact internal timing of the flip-flop or the routing delay of the signal to the clock versus the reset input, we cannot know which signal will arrive first - the clock or the reset. This is a race condition. If the clock rising edge appears first, the output will remain low. If the reset signal appears first, the output will go high. A slight change in temperature, voltage, or process may cause a chip that works correctly to suddenly work incorrectly. Here a faster clock is used, and the flip-flop is reset on the rising edge of the clock. This circuit performs the same function, but as long as SIG1 and SIG2 are produced synchronously - they change only after the rising edge of CLK - there is no race condition.

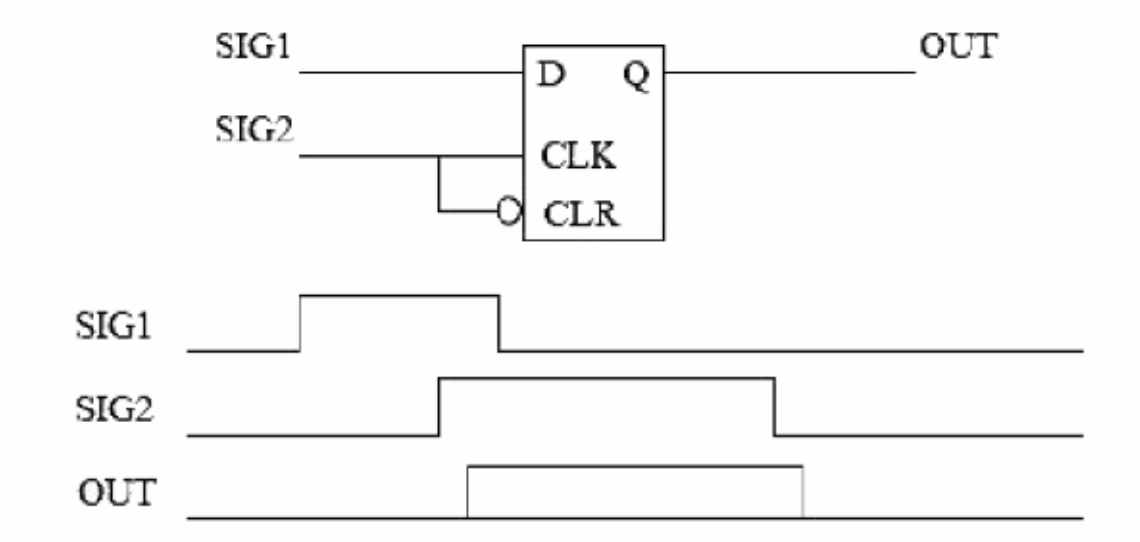


Figure: 4.6 - Asynchronous Race Condition

4.9.5 Metastability

One of the great buzzwords, and often misunderstood concepts, of synchronous design is metastability. Metastability refers to a condition which arises when an asynchronous signal is clocked into a synchronous flip-flop. While chip designers would prefer a completely synchronous world, the unfortunate fact is that signals coming into a chip will depend on a user pushing a button or an interrupt from a processor, or will be generated by a clock which is different from the one used by the chip. In these cases, the asynchronous signal must be synchronized to the chip clock so that it can be used by the internal circuitry. The designer must be careful how to do this in order to avoid metastability problems. If the ASYNC_IN signal goes high around the same time as the clock, we have an unavoidable race condition. The output of the flip-flop can actually go to an undefined voltage level that is somewhere between a logic 0 and logic 1. This is because an internal transistor did not have enough time to fully charge to the correct level. This meta level may remain until the transistor voltage leaks off or .decays., or until the next clock cycle. During the clock cycle, the gates that are connected to the output of the flip-flop may interpret this level differently. In the figure, the upper gate sees the level as logic 1 whereas the lower gate sees it as logic 0. In normal operation, OUT1 and OUT2 should always be the same value. In this case, they are not and this could send the logic into an unexpected state from which it may never return. This metastability can permanently lock up the chip. The solution to this metastability

problem by placing a synchronizer flip-flop in front of the logic, the synchronized input will be sampled by only one device, the second flip-flop, and be interpreted only as logic 0 or 1. The upper and lower gates will both sample the same logic level, and the metastability problem is avoided. Or is it? The word solution is in quotation marks for a very good reason. There is a very small but non-zero probability that the output of the synchronizer flip-flop will not decay to a valid logic level within one clock period. In this case, the next flip-flop will sample an indeterminate value, and there is again a possibility that the output of that flip-flop will be indeterminate. At higher frequencies, this possibility is greater. Unfortunately, there is no certain solution to this problem. Some vendors provide special synchronizer flip-flops whose output transistors decay very quickly. Also, inserting more synchronizer flip-flops reduces the probability of metastability but it will never reduce it to zero. The correct action involves discussing metastability problems with the vendor, and including enough synchronizing flip-flops to reduce the probability so that it is unlikely to occur within the lifetime of the product.

4.9.6 Timing Simulation

This method of timing analysis is growing less and less popular. It involves including timing information in a functional simulation so that the real behavior of the chip is simulated. The advantage of this kind of simulation is that timing and functional problems can be examined and corrected. Also, asynchronous designs must use this type of analysis because static timing analysis only works for synchronous designs. This is another reason for designing synchronous chips only. As chips become larger, though, this type of compute intensive simulation takes longer and longer to run. Also, simulations can miss particular transitions that result in worst case results. This means that certain long delay paths never get evaluated and a chip with timing problems can pass timing simulation. If you do need to perform timing simulation, it is important to do both worst case simulation and best case simulation. The term best case can be misleading. It refers to a chip that, due to voltage, temperature, and process variations, is operating faster than the typical chip. However, hold time problems become apparent only during the best case conditions.

Chapter 5

RESULTS AND CONCLUSION

5.1 CORDIC SIMULATION

The code was synthesized using XILINX ISE 9.2i and implemented on Spartan 3E FPGA board. The simulation results are shown below:

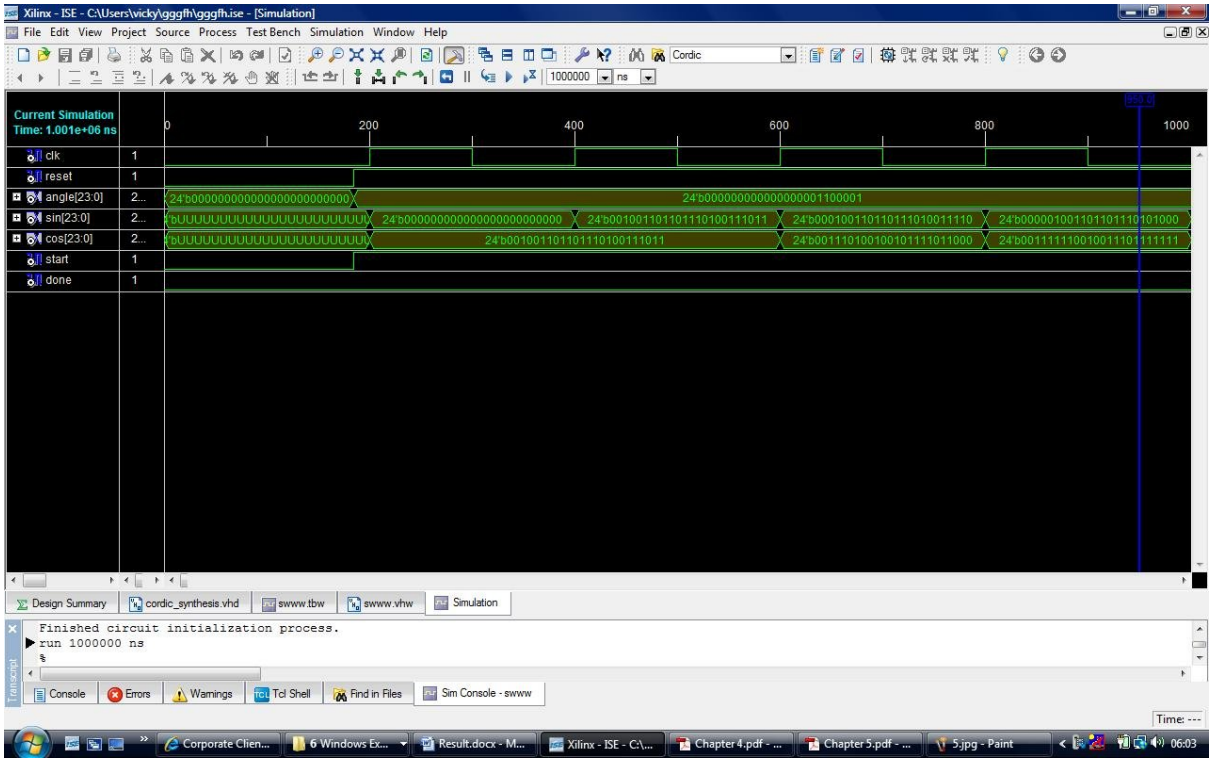


Figure: 5.1- Calculation of Sin and Cos

The input given in this case is angle 30^0 and the respective Sine and Cosine function values are generated and as clearly shown in the figure N above the values of both functions change with each iteration and after the 20 iterations the final values are obtained.

The angle input is given in radians in binary format as given below

30^0 is represented in radians in binary as = 000000000000000001100001

And the output obtained is Sin 30^0 = 0.5

$\approx 00000000000000000001100000$

And the approximated $Cos30^0$ value = 0.0866

$\approx 00111111111111111111111$

The final results were available after 5000ns simulation time. The following figure shows the final results obtained after simulation.

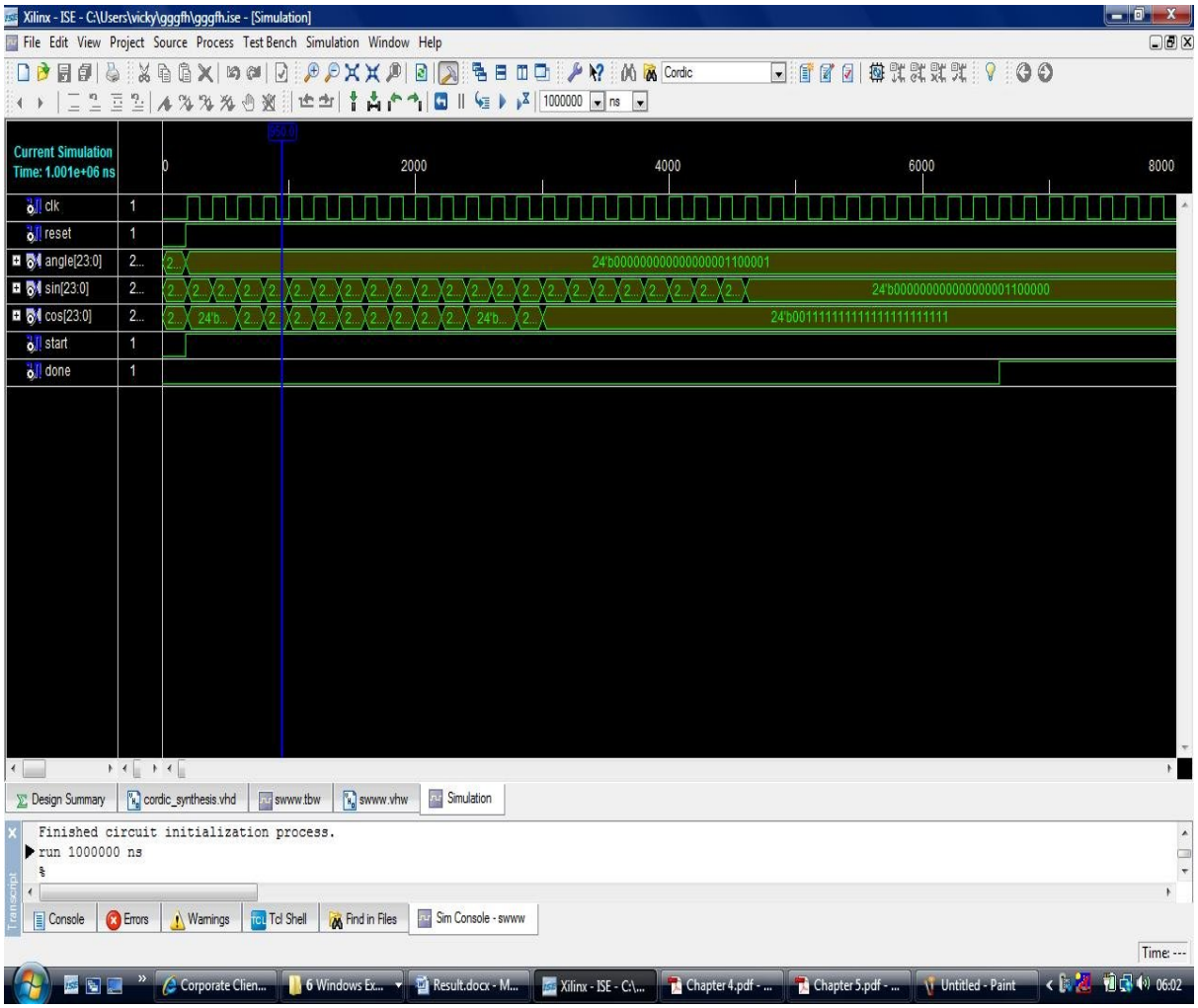


Figure: 5.2 – A closer view**.**

The design is interfaced with onboard LCD display. The code was then successfully implemented on Spartan3E board after successful completion of translate, map, place and route processes. And the outputs can be viewed on LCD display. The outputs for the above taken example for angle 30^0 are shown displayed on FPGA. The display shows the final value of Cos 30^0 to validate the implementation.

Figure 5.3-output for cosine on LCD

5.2 ANALYSIS OF THE SYNTHESIS REPORT

```
=========================================================================
*                    Synthesis Options Summary                    *
=========================================================================
---- Source Parameters
Input File Name                    : "cordic.prj"
Input Format                       : mixed
Ignore Synthesis Constraint File   : NO
---- Target Parameters
Output File Name                   : "cordic"
Output Format                      : NGC
Target Device                      : xc3s500e-4-fg320

---- Source Options
```

Top Module Name : cordic
Automatic FSM Extraction : YES
FSM Encoding Algorithm : Auto
Safe Implementation : No
FSM Style : lut
RAM Extraction : Yes
RAM Style : Auto
ROM Extraction : Yes
Mux Style : Auto
Decoder Extraction : YES
Priority Encoder Extraction : YES
Shift Register Extraction : YES
Logical Shifter Extraction : YES
XOR Collapsing : YES
ROM Style : Auto
Mux Extraction : YES
Resource Sharing : YES
Asynchronous To Synchronous : NO
Multiplier Style : auto
Automatic Register Balancing : No

---- Target Options
Add IO Buffers : YES
Global Maximum Fanout : 500
Add Generic Clock Buffer(BUFG) : 24
Register Duplication : YES
Slice Packing : YES
Optimize Instantiated Primitives : NO
Use Clock Enable : Yes
Use Synchronous Set : Yes
Use Synchronous Reset : Yes
Pack IO Registers into IOBs : auto
Equivalent register Removal : YES

---- General Options

Optimization Goal : Speed

Optimization Effort : 1

Library Search Order : cordic.lso

Keep Hierarchy : NO

RTL Output : Yes

Global Optimization : AllClockNets

Read Cores : YES

Write Timing Constraints : NO

Cross Clock Analysis : NO

Hierarchy Separator : /

Bus Delimiter : <>

Case Specifier : maintain

Slice Utilization Ratio : 100

BRAM Utilization Ratio : 100

Verilog 2001 : YES

Auto BRAM Packing : NO

Slice Utilization Ratio Delta : 5

Final Report

==========

Final Results

RTL Top Level Output File Name : cordic.ngr

Top Level Output File Name : cordic

Output Format : NGC

Optimization Goal : Speed

Keep Hierarchy : NO

Design Statistics

IOs : 76

Cell Usage :

BELS : 579

```
#      INV                         : 4
#      LUT2                        : 28
#      LUT2_D                      : 3
#      LUT2_L                      : 2
#      LUT3                        : 166
#      LUT3_D                      : 6
#      LUT3_L                      : 4
#      LUT4                        : 148
#      LUT4_D                      : 7
#      LUT4_L                      : 10
#      MUXCY                       : 69
#      MUXF5                       : 59
#      VCC                         : 1
#      XORCY                       : 72
# FlipFlops/Latches                : 85
#      FDC                         : 2
#      FDE                         : 24
#      FDRE                        : 45
#      FDSE                        : 14
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 75
#      IBUF                        : 26
#      OBUF                        : 49
```

Device utilization summary:

Selected Device : 3s500efg320-4

Number of Slices:	196 out of 4656	4%
Number of Slice Flip Flops:	85 out of 9312	0%
Number of 4 input LUTs:	378 out of 9312	4%
Number of IOs:	76	

Number of bonded IOBs:	76 out of	232	32%
Number of GCLKs:	1 out of	24	4%

The Delay Summary Report

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

The AVERAGE CONNECTION DELAY for this design is: 1.160

The MAXIMUM PIN DELAY IS: 3.561

The AVERAGE CONNECTION DELAY on the 10 WORST NETS is: 2.980

5.3 CALCULATION OF QUANTIZATION ERROR

The angle accumulated after 15 iterations is 29.886^0.

The Quantization error is given by the expression:

$$\xi_m = \theta_i - \sum \quad . \tan^{-1} \quad (\alpha(j).2^{-s(j)}) \tag{5.1}$$

$$\xi_m = 30^0 - 29.986^0 \tag{5.2}$$

$$= 0.014^0$$

Error in decibles SQNR in dB= 10 log —

$$\xi_m = - 50.077\text{dB} \tag{5.3}$$

Quantization error for same number of iterations (15 iterations).

Table 5.1 Comparision

ξ_m(EEAS CORDIC)	50 .077 dB
ξ_m(CORDIC)	30.55 dB

This shows significant improvement in SQNR for same number of iterations.

5.4 CONCLUSION

In this thesis few improvements are included in the CORDIC algorithm for calculating some common trigonometric functions. The method is better than CORDIC in the following aspects which are:

- More accurate.
- More faster.
- Reduced Quantization error.

There are some drawbacks also when compared to CORDIC. The inclusion of extra SPTs to the algorithm increases the hardware complexity and hence area. The algorithm can be used to compute trigonometric functions where area on chip is not a big issue. It has its applications in biomedical such as speech recognition, coding, sound processing and medical imaging.

5.5 FUTURE SCOPE

Speed and area trade-off is one of the major challenges that are being faced by designers. As the introduction of numerous SPTs to the design increases the area on-chip, more work can be done to optimize speed and area, and introduce parallelism in the design. Parallelism can be introduced by using pipelined architectures.

.

REFERNCES

[1]. J. S. Walther, "A unified algorithm for elementary functions,"Spring Joint Computer Conference, pp. 379-385, 1971.

[2]. Y. H. Hu and S. Naganathan, "An angle recoding method for CORDIC algorithm implementation," IEEE Transaction on Computers, volume 42, pp. 99-102, January 1993.

[3]. B. Sklar, Digital Communications: Fundamentals and Applications Prentice Hall, 1988.

[4]. Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing," IEEE Signal processing Magazine, pp. 16-35, July 1992.

[5]. Y. C. Lim, R. Yang, D. Li, and J. Song, "Signed power-of-two term allocation scheme for the design of digital filters," IEEE Transaction Circuits Syst. II, volume 46, pp. 577-584, May 1999.

[6]. Cheng Shing Wu, Au Yen Wu "EEAS based CORDIC algorithm, " IEEE May 2001.

[7]. Chi Hsui Lin , An Yen Wu " Mixed – scaling rotation CORDIC algorithm and architecture for high – speed performance vector rotation DSP application" , Volume 52 , November 2005.

[8]. Sungwook Yu , Swaret Slander EE. Jr "A scaled DCT architecture with CORDIC algorithm, "Volume 50, January 2002.

[9]. Naik, R. Stojcevski, V. Singh "Implementation of magnitude estimation algorithm for hearing aids" IEEE Dec 2004.

[10]. Cheng Shing Wu, An Yen Wu "A novel rotational VLSI architecture based on Extended Elementary Angle Set CORDIC Algorithm, "IEEE August 2000.

[11]. Cheng Shing Wu, Chih Hsing Lin " A high performance, low latency vector rotational CORDIC architecture based on extended elementary angle set and Trellis based searching scheme", IEEE September 2003.

[12]. Shen Fu Hsiao "Parallel processing of complex data using quaternion and pseudo quaternion CORDIC algorithm "IEEE , August 1994.

[13]. Summanasena M.G.B "A scale factor correction scheme for CORDIC algorithm", IEEE , August 2008.

[14]. Hamill R. Mccanny "Online CORDIC algorithm and VLSI architecture for implementating OR–array processor", IEEE February 2000.

[15]. Rao, P.R. Chakrabarti "High performance compensation technique for radix-4 CORDIC algorithm", IEEE September 2002.

[16]. Boudabous A, Ghozzi, M. W. Asmondi "Implementation of hyperbolic function using CORDIC algorithm", IEEE December 2004.

[17]. Meng Qian "Application of CORDIC algorithm to neutral networks VLSI design", IEEE October 2006.

[18]. Ching Shing Wu , An Ye Wu "Modified vector rotation CORDIC algorithm and its application to FFT" , IEEE may 2000.

[19]. Ching Shing Wu, An Yen Wu "Modified vector rotational CORDIC algorithm and architecture J. S. Walther, "A unified algorithm for elementary functions," Spring Joint Computer Conference, pp. 379-385, 1971.

[20]. Y. H. Hu and S. Naganathan, "An angle recoding method for CORDIC algorithm implementation," IEEE Trans. on Computers, vol. 42, pp. 99-102, Jan. 1993.

[21]. Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing" ,IEEE Signal processing Magazine, pp. 16-35, July 1992.

[22]. Y. C. Lim, R. Yang, D. Li, and J. Song, "Signed power-of-two term allocation scheme for the design of digital filters," IEEE Trans. Circuits Syst. II, vol. 46, pp. 577-584, May 1999 Lecture , IEEE, june 2001.