IMPLEMENTATION OF HIGH SPEED FIXED AND FLOATING POINT CORDIC TECHNIQUES

A Thesis submitted in partial fulfillment of the requirements

for the award of degree of

MASTER OF TECHNOLOGY

IN

VLSI Design and CAD

Submitted by:

Sukhpreet Kaur

Roll No: 601061024

Under the guidance of:

Dr. Kulbir Singh

Associate Professor, ECED

Department of Electronics and Communication Engineering

THAPAR UNIVERSITY

(Established under the section 3 of UGC Act, 1956)

PATIALA – 147004 (PUNJAB)

JULY 2012

CERTIFICATE

I, Sukhpreet Kaur, hereby certify that the work which is being presented in this thesis entitled **"Implementation of High Speed Fixed and Floating Point CORDIC Techniques"** by me in partial fulfillment of the requirements for the award of degree of Master of Technology in VLSI Design and CAD from Thapar University (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Dr. Kulbir Singh.

The matter presented in this thesis has not been submitted in any other University / Institute for the award of any other degree.

Date: 22 June '2012

Sukhpreet Kaur
Sukhpreet Kaur
Roll No. 601061024

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

Date: 22/06/2012

Dr. Kulbir Singh
Associate Professor, ECED
Thapar University, Patiala

Countersigned by:

Professor and Head ECED
Thapar University, Patiala
Date:

Dean of Academic Affairs
Thapar University, Patiala
Date:

ACKNOWLEDGEMENT

I would like to express my gratitude to **Dr. Kulbir Singh,** Associate Professor, Electronics and Communication Engineering Department, Thapar University, Patiala, for his patience guidance and support throughout this thesis work. I am truly very fortunate to have the opportunity to work with him. He has provided me help in technical writing and presentation style and I found this guidance to be extremely valuable. I would like to thank **Mr. Ankush Kansal** for his kind support during my thesis work.

I convey my sincere thanks to **Head of the Department, Dr. Rajesh Khanna**, entire faculty and staff of Electronics and Communication Engineering Department for their encouragement and cooperation.

My greatest thanks are to all who wished me success especially my parents. Above all I render my gratitude to the Almighty who bestowed self-confidence, ability and strength in me to complete this work for not letting me down at the time of crisis and showing me the silver lining in the dark clouds. I do not find enough words with which I can express my feelings of thanks to my dear friends for their help, inspiration and moral support which went a long way in successful competition of the present study.

Sukhpreet Kaur

(Sukhpreet Kaur)

ABSTRACT

CORDIC is an acronym for COrdinate Rotation Digital Computer. It is a class of shift and add algorithms for rotating vectors in a plane, which is usually used for the calculation of trigonometric functions, multiplication, division and conversion between binary and mixed radix number systems of DSP applications, such as Fourier Transform. A fast and energy-efficient CORDIC for the calculation of elementary function is always needed in electronics systems i.e. DSP processors, image processing and arithmetic units in microprocessors. On VLSI implementation level, the area also becomes quite important as more area means more system cost. The three parameters i.e. power, speed and area are always traded off. For DSP processors area and speed are the main ones. But sometimes, increasing the speed also increases the power consumption, so there is an upper bound of speed for a given power budget.

Since elementary functions calculation dominates the execution time of most DSP algorithms, so there is need for high speed CORDIC algorithm. In this thesis, a very high speed CORDIC algorithm is implemented for fast calculations of trigonometric functions. VHDL is used to implement a technology-independent design. Main drawback of CORDIC algorithm is that to converge to N bit of accuracy, N iterations are required. So, in this thesis, three types of CORDIC algorithm that are Original CORDIC, Control CORDIC and Angle Recoding CORDIC in which number of iterations get reduce are discussed. There are two types of representations for real numbers that is fixed point and floating point. The comparison of Original CORDIC, Control CORDIC and Angle Recoding CORDIC for sine-cosine generation on the basis of their speed, area and number of iterations for 16 bit, 24 bit and 32 bit fixed point number have been discussed. The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values. So, in this thesis, a high speed Original CORDIC for sine cosine generation for 24 bit, 28 bit and 32 bit (single precision IEEE 754-2008) floating point numbers is also synthesized.

The design is simulated on Modelsim SE and synthesized on Xilinx 13.1i. The Thesis pays a significant attention to the analysis of CORDIC algorithm in terms of speed so as to maximize throughput.

TABLE OF CONTENTS

Certificate	i
Acknowledgement	ii
Abstract	iii
Table of Contents	iv-vi
List of Figures	vii-ix
List of Tables	x
List of Abbreviation	xi-xii
CHAPTER 1: INTRODUCTION	**1-6**
1.1 Preamble	1
1.2 CORDIC Algorithm	2
1.2.1 Elementary Functions	2
1.2.2 An Introduction to CORDIC	3
1.3 Number Format	5
CHAPTER 2: LITERATURE REVIEW	**7-14**
2.1 Original CORDIC	7
2.2 Control CORDIC	8
2.3 Angle Recoding CORDIC	10
2.4 Floating point and fixed point CORDIC algorithm	12
2.5 Gaps in the study	13
2.6 Thesis Objective	13
2.7 Organization of Thesis	14
CHAPTER 3: CORDIC ALGORITHM	**15-58**
3.1 Introduction	15
3.2 Basic equations of CORDIC Algorithm	15
3.3 Basic CORDIC iteration	25
3.3.1 Unified CORDIC	31
3.4 CORDIC Hardware	37

3.5 Original CORDIC 38
3.6 Control CORDIC 39
3.7 Angle Recoding CORDIC 42
3.8 Applications of CORDIC 45
3.8.1 Robotics 45
3.8.2 3-D Computer Graphics 45
3.8.3 OFDM 45
3.8.4 DCT Compression 46
3.9 Number Format 46
3.9.1 Fixed Point Numbers 47
3.9.1.1 Unsigned Fixed Point Numbers 47
3.9.1.2 Signed Two's Complement Fixed Point Numbers 47
3.9.2 Floating Point Numbers 49
3.9.2.1 Normalization 50
3.9.2.2 IEEE 754 Standard For Binary Floating-Point Arithmetic 53
3.9.2.3 Formats 53
3.9.2.4 Exceptions 55

CHAPTER 4: FPGA Design Flow 59-66
4.1 Introduction to FPGA 59
4.2 FPGA for Fixed Point and Floating Point Computations 60
4.3 FPGA Technology Trends 60
4.4 FPGA Implementation 60

CHAPTER 5: RESULTS AND DISCUSSIONS 67-88
5.1 ModelSim simulation result 67
5.1.1 For sine-cosine real input and real output of CORDIC algorithm. 67
5.2 Xilinx Simulation result 67
5.2.1 Simulation result of sine cosine CORDIC for 16-bit, 24-bit and 32-bit fixed point number 67

5.2.1.1 Simulation result for Original CORDIC 67
5.2.1.2 Simulation result for Control CORDIC 73
5.2.1.3 Simulation result for Angle Recoding CORDIC 78
5.2.2 Simulation result of sine cosine CORDIC for 24-bit, 28-bit and 32-bit floating point numbers 83
5.2.2.1 Simulation result for Original CORDIC 83
5.3 Discussions 88

CHAPTER 6: CONCLUSION **89-90**
6.1 Conclusion 89
6.2 Future Scope 90

REFERENCES **91-94**

LIST OF FIGURES

Figure 1.1	Fixed point number representation	5
Figure 1.2	Floating point number representation	6
Figure 3.1	Rotation of a vector V by the angle $\emptyset$	16
Figure 3.2	Vector V with magnitude r and phase θ	16
Figure 3.3	Chaining multiple micro-rotations together	19
Figure 3.4	Pseudo rotation	22
Figure 3.5	Rotation and Pseudo rotation	22
Figure 3.6	A balance having θ at one side and small weights (angle) at the other side	24
Figure 3.7	Inclined balance due to difference in weight of two sides	25
Figure 3.8	Rotation mode to compute ($\cos\theta$ and $\sin\theta$)	27
Figure 3.9	Rotation mode Graph of CORDIC algorithm	29
Figure 3.10	Vectoring mode to compute $\tan^{-1} y$	30
Figure 3.11	Circular rotation	33
Figure 3.12	Linear Rotation	34
Figure 3.13	Hyperbolic rotation	36
Figure 3.14	Hardware Implementation of a CORDIC Iteration	37
Figure 3.15	Rotation mode graph of Control CORDIC and Original CORDIC	41
Figure 3.16	Angle selection in the Angle Recoding method	43
Figure 3.17	Hardware implementation of Angle Recoding CORDIC	43
Figure 3.18	Rotation mode graph of Angle Recoding CORDIC and Original CORDIC	44
Figure 3.19	16-bit unsigned Fixed Point Number	47
Figure 3.20	16-bit signed two's complement Fixed Point numbers	48
Figure 3.21	Single Precision Format for Floating Point Numbers	54
Figure 3.22	Bit Double Precision Floating Point Format	54
Figure 4.1	FPGA Design Flow	62
Figure 5.1	Modelsim result for Original CORDIC for real input and real output	67

Figure 5.2	Simulation result for Sine-cosine of 16-bit fixed point Original CORDIC	68
Figure 5.3	Top level schematic for 16-bit fixed point Original CORDIC	68
Figure 5.4	Summary report for 16-bit fixed point Original CORDIC	69
Figure 5.5	Simulation result for Sine-cosine of 24-bit fixed point Original CORDIC	69
Figure 5.6	Top level schematic for 16-bit fixed point Original CORDIC	70
Figure 5.7	Summary report for 24-bit fixed point Original CORDIC	70
Figure 5.8	Simulation result for Sine-cosine of 32-bit fixed point Original CORDIC	71
Figure 5.9	Top level schematic for 32-bit fixed point Original CORDIC	71
Figure 5.10	Summary report for 32-bit fixed point Original CORDIC	72
Figure 5.11	Simulation result for Sine-cosine of 16-bit fixed point Control CORDIC	73
Figure 5.12	Top level schematic for 16-bit fixed point Control CORDIC	73
Figure 5.13	Summary report for 16-bit fixed point Control CORDIC	74
Figure 5.14	Simulation result for Sine-cosine of 24-bit fixed point Control CORDIC	74
Figure 5.15	Top level schematic for 24-bit fixed point Control CORDIC	75
Figure 5.16	Summary report for 24-bit fixed point Control CORDIC	75
Figure 5.17	Simulation result for Sine-cosine of 32-bit fixed point Control CORDIC	76
Figure 5.18	Top level schematic for 32-bit fixed point Control CORDIC	76
Figure 5.19	Summary report for 32-bit fixed point Control CORDIC	77
Figure 5.20	Simulation result for Sine-cosine of 16-bit fixed point Angle Recoding CORDIC	78
Figure 5.21	Top level schematic for 16-bit fixed point Angle Recoding CORDIC	78
Figure 5.22	Summary report for 16-bit fixed point Angle Recoding CORDIC	79
Figure 5.23	Simulation result for Sine-cosine of 24-bit fixed point Angle Recoding CORDIC	79
Figure 5.24	Top level schematic for 24-bit fixed point Angle Recoding	80

	CORDIC	
Figure 5.25	Summary report for 24-bit fixed point Angle Recoding CORDIC	80
Figure 5.26	Simulation result for Sine-cosine of 32-bit fixed point Angle Recoding CORDIC	81
Figure 5.27	Top level schematic for 32-bit fixed point Angle Recoding CORDIC	81
Figure 5.28	Summary report for 32-bit fixed point Angle Recoding CORDIC	82
Figure 5.29	Simulation result for Sine-cosine of 24-bit floating point CORDIC Algorithm	83
Figure 5.30	Top level schematic for 24-bit floating point CORDIC Algorithm	83
Figure 5.31	Summary report for 24-bit floating point CORDIC	84
Figure 5.32	Simulation result for Sine-cosine of 28-bit floating point Angle Recoding CORDIC	84
Figure 5.33	Top level schematic for 28-bit floating point CORDIC Algorithm	85
Figure 5.34	Summary report for 24-bit floating point CORDIC	85
Figure 5.35	Simulation result for Sine-cosine of 32-bit floating point Angle Recoding CORDIC	86
Figure 5.36	Top level schematic for 32-bit floating point CORDIC Algorithm	86
Figure 5.37	Summary report for 32-bit floating point CORDIC	87

LIST OF TABLES

Table 3.1	For 16, 24 and 32 bit CORDIC hardware	20
Table 3.2	CORDIC gain k_i for 16, 24 and 32 bit hardware	21
Table 3.3	Choosing the sign of the rotation angles to force z to zero	27
Table 3.4	Unified CORDIC Operational Modes	32
Table 3.5	Unified CORDIC Rotation Functions	32
Table 3.6	CORDIC mode	36
Table 3.7	Comparison of Original CORDIC and Control CORDIC angle selection	41
Table 3.8	Signed fixed point representation	49
Table 3.9	Adjustments of exponents	51
Table 3.10	Normalization	52
Table 3.11	Examples of Floating Point Numbers	52
Table 3.12	Various basic formats of IEEE 754 standard	53
Table 3.13	Representation of Single Precision floating point numbers	55
Table 3.14	Normalization effect on result's exponent and overflow/underflow detection	58
Table 3.15	Number of sign bit, exponent bit and mantissa bit in 24-bit and 28-bit floating point number	58
Table 5.1	Comparison between 16 bit, 24 bit and 32 bit fixed point number Original CORDIC	72
Table 5.2	Comparison between 16 bit, 24 bit and 32 bit fixed point number Control CORDIC	77
Table 5.3	Comparison between 16 bit, 24 bit and 32 bit fixed point number Angle Recoding CORDIC	82
Table 5.4	Comparison between 24 bit, 28 bit and 32 bit floating point number for sine-cosine CORDIC algorithm	87

LIST OF ABBREVATIONS

3-D	3-Dimensional
ALUs	Arithmetic Logic Unit
ANSI	American National Standard Institute
ASICs	Application-Specific Integrated Circuits
B-58	Bomber's Navigation Computer
BCD	Binary Coded Decimal
CFFT	Complex Fast Fourier Transform
CLBs	Configurable logic blocks
CORDIC	**Co**ordinate **R**otation **Di**gital **C**omputer algorithm
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DSP	Digital signal processing
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GaAs	Gallium Arsenide
HDL	Hardware Description Language
HP	Hewlett Packard
IDCT	Inverse Discrete Cosine Transform
IEEE	Institute of Electrical and Electronics Engineers
IOBs	Input Output Blocks
LTI	Linear Time Invariant
LUTs	Look-Up Table
MAC	Multiply and Accumulate
MSE	Mean Square Error
NaN	Not a number
NCD	Native Circuit Description
NGC	Native Generic Complier
NGD	Native information and Generic Database
OFDM	Orthogonal Frequency Division Multiplexing
PAR	Place and Route
RAM	Random Access Memory

RTL	Register Transfer level
ROM	Read Only Memory
SDR	Software Defined Radio
SVD	Singular Value Decomposition
UCF	User Constraint File
VHDL	Very High Speed Integrated Circuits HDL
VLSI	Very Large Scale Integration

CHAPTER 1
INTRODUCTION

1.1 Preamble

CORDIC is an acronym coined by Jack E. Volder to describe the **Co**ordinate **R**otation **Di**gital **C**omputer algorithm which he developed in 1959 [1]. It was used for the real time navigation system at that time and was further extended by Walther in the year 1971 [2]. It is used for the fast calculation of elementary functions like multiplication, division, trigonometric functions, logarithmic function, and various conversions like conversion of rectangular to polar coordinate and vice-versa. Although CORDIC may not be the fastest technique to perform these operations, it is attractive due to the simplicity of its hardware implementation, since the same iterative algorithm could be used for all these applications using the basic shift-add operations. CORDIC algorithm can be applied in two modes (ex. rotation and vectoring) and three types (ex. linear, circular and hyperbolic mode). The algorithm is very attractive for hardware implementation because it uses only elementary shift-and-add operations to perform the vector rotation. It only needs the use of 2 shifter and 3 adder modules, so its power dissipation is very less and it is also very compact. It is frequently used in an array of processing elements on VLSI chips [3].

Digital signal processing (DSP) algorithms exhibit an increasing need for the efficient implementation of complex arithmetic operations. The computation of trigonometric functions, coordinate transformations or rotations of complex valued phasors is almost naturally involved with modern DSP algorithms. Popular application examples are algorithms used in digital communication technology and in adaptive signal processing. While in digital communications, the straightforward evaluation of the cited functions is important, numerous matrix based adaptive signal processing algorithms require the solution of systems of linear equations, QR factorization or the computation of eigen values, eigenvectors or singular values [4]. All these tasks can be efficiently implemented using processing elements performing vector rotations. Coordinate Rotation Digital Computer (CORDIC) algorithm offers the opportunity to calculate all the desired functions in a rather simple and elegant way.

The algorithm can be coded in firmware as well as 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 its versatility and simplicity that make CORDIC the preferred implementation of math functions on small hand calculators. It calculates the trigonometric functions of sine, cosine, magnitude and phase (arctangent) to any desired precision. It can also calculate hyperbolic functions. CORDIC has been implemented in pocket calculators like Hewlett Packard's HP 35 and in arithmetic coprocessors like the Intel 8087. Some authors proposed to use CORDIC processors for signal processing applications like filtering, SVD [5], for image processing or for solving linear systems. CORDIC algorithm has found its wide application in the computation of Fast Fourier Transform [6]. Today CORDIC algorithm is used in Neural Network VLSI design, high performance vector rotation DSP applications, advanced circuit design, optimized low power design [7].

In this thesis architecture for Original CORDIC, Control CORDIC and Angle Recoding CORDIC for 16-bit, 24-bit and 32-bit fixed point has been proposed. 24-bit, 28-bit and 32-bit single precision IEEE 754-2008 standard floating point CORDIC algorithm is also synthesized using Xilinx.

1.2 CORDIC Algorithm

1.2.1 Elementary Functions

The well-known functions of $\sin\theta$, $\cos\theta$, $\sin^{-1}\theta$, $\cos^{-1}\theta$, $\sinh\theta$, $\cosh\theta$, $\sinh^{-1}\theta$, $\cosh^{-1}\theta$, e^x, $\log x$ etc. are all members of an important class of functions in mathematics, known as elementary functions. Elementary functions are unique in that they cannot be computed exactly in a finite number of arithmetic operations – their exact representation requires the use of an infinite series of algebraic terms. However they can be approximated to a desired precision using a finite number of operations. They have been used in such diverse applications as Robotics, 3-D Computer Graphics, SVD Decomposition, Digital Signal Processing. Computing the trigonometric function is a time consuming operation. In fact of all the arithmetic operations that a chip must perform, trigonometric functions have the worst latency [8]. They require many cycles to evaluate so that instructions dependent upon their evaluation must stall until the result becomes available. Additionally resources such as adders, shifters and multipliers are tied up and are unavailable for use even by other independent instructions, leading to stalls because of structural hazards.

It is therefore imperative that these elementary functions be computed as quickly as possible to avoid degradation in performance. The results must be obtained with high accuracy for any of the angles within the principle domain of the elementary function. Techniques such as range reduction are helpful in mapping the argument to its principal domain, but even so the algorithm must be flexible enough to provide an accurate answer with any input point from within its domain. Power consumption has become an important metric in electronic design today, especially as gadgets and computing devices shrink in size. The heat that is dissipated by the power consumption in the computing chip makes the chip difficult to cool. This chip must either be run at a degraded level of performance to prevent it from burning up, or else expensive and bulky cooling mechanisms such as heat sinks or air flow must be used to keep the temperature down to manageable levels. When power hungry computing elements are used in consumer devices such as digital cameras or MP3 players, they drain the battery quickly, which leads to a poor experience for the user. Either way, excessive power consumption limits the performance of an arithmetic chip.

There are five principal ways in which an elementary function may be computed in hardware – by using table lookup, polynomial approximation, rational approximation, CORDIC and quadratic convergence methods. The CORDIC method is the most versatile of all the algorithms that can be used to evaluate elementary functions. The same hardware can be used to compute trigonometric ratios (sin, cos, tan, etc.), hyperbolic ratios (sinh, cosh, tanh), multiplication, division, inverse trigonometric (arcsine, arccos) and inverse hyperbolic ratios (arcsinh, arccosh). With a slight modification it can also compute logarithms, exponentials, etc.

1.2.2 An Introduction to CORDIC

The Volder's algorithm was derived from the general equations of vector rotation. Mr. Volder was trying to improve the real time navigation systems used in a B-58 bomber [9]. In those days, analog instruments were used to solve the complicated navigational equations which helped to locate the position of the aircraft on the earth. However they possessed limited accuracy. He calculated the Sine and Cosine of an angle, by moving a vector from its initial position (along the X axis) to its final position where it lay inclined at some angle θ to the X axis. The vector was moved in a series of small steps, while simultaneously updating the X and Y coordinates of the vector after every step. The

update operation was simple to perform. Once the vector reached its target angular position, its final X and Y coordinates gave the Cosine and Sine values respectively, of the angle of inclination θ.

The CORDIC algorithm is used to evaluate real time calculation of elementary functions using the iterative rotation of the input vector. The rotation of a given vector is realized by means of a sequence of rotations with fixed angles which results in overall rotation through a given angle or result in a final angular argument of zero. However, the major disadvantage of the CORDIC algorithm is its slow computational speed. For iterative CORDIC structure, the speed performance of CORDIC operation is limited by the large iteration number N which are generally equal to the internal word length, W. At algorithmic level, one trivial solution to overcome such a problem is to reduce the iteration number directly. A number of methods were introduced to reduce iteration count of CORDIC to improve its performance. Online CORDIC was developed by Ercegovac and Lang [10] for applications where input bits became available serially. The Online CORDIC method replaces variable shifters by more area-efficient delays. Their method could also compensate for the value of K online. Duprat and Muller [3] take the tack of reducing the cycle time of a CORDIC iteration by using fast adders, based upon the use of redundant arithmetic to express the operands.

In case of a Control CORDIC, divergent rotations are completely eliminated by eliminating overshoot [11]. For applications that require forward rotation (or vector rotation) only, the Angle Recoding technique provides a relaxed approach to speed up the operation of the CORDIC algorithm [12]. The Coordinate Rotation Digital Computer algorithm offers the opportunity to calculate all the desired functions in a simple and elegant way. Keeping the requirements and constraints of different application environments in view, the development of CORDIC algorithm and architecture has taken place for achieving high throughput rate and reduction of hardware-complexity as well as the latency of implementation [13]. Some of the typical approaches for reduced-complexity implementation are focused on minimization of the complexity of scaling operation and the complexity of barrel-shifter in the CORDIC engine. Latency of implementation is an inherent drawback of the conventional CORDIC algorithm. Angle recoding schemes, mixed-grain rotation and higher radix CORDIC have been developed for reduced latency realization. Parallel and pipelined CORDIC have been suggested for high-throughput computation [14].

1.3 Number Format

A number format in computer is the internal representation of numeric values in digital computer hardware and software. Normally, numeric values are stored as groupings of bits, named for the number of bits that compose them. In real life, we deal with real numbers that is numbers with fractional part. In most modern computer we have have hardware support for fixed point numbers and floating point numbers for representing real numbers.

Fixed point number representation: Fixed point formatting is useful to represents fractions in binary. In fixed point representation every word has the same number of digits and the binary point is always fixed at the same position. By implementing algorithms using fixed point mathematics a significant improvement in execution speed can be observed because of inherent integer math hardware support in a large number of processor as well as the reduced software complexity for emulated integer multiply and divide. This speed improvement does come at the cost of reduced range and accuracy of the algorithm variables.

Qm.n format: m bit for whole part, n bit for fractional part.

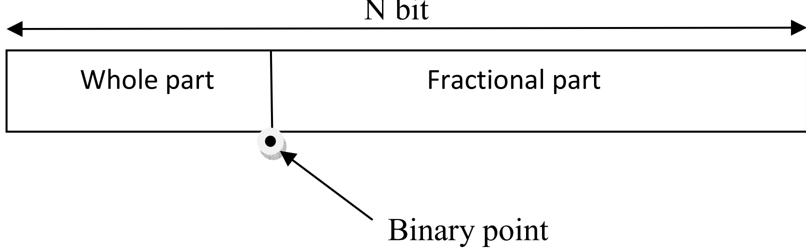


Figure 1.1 Fixed point number represenation

Floating point number representation: In computing, floating point describes a method of representing real numbers in a way that can support a wide range of values. Numbers are, in general, represented approximately to a fixed number of significant digits and scaled using an exponent. The base for the scaling is normally 2, 10 or 16. The typical number that can be represented exactly is of the form:

$$\text{Significant digits} \times \text{base}^{\text{exponent}}$$

The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float" that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a

computer realization of scientific notation. The advantage of floating-point representation is that it can support a much wider range of values and are very accurate.

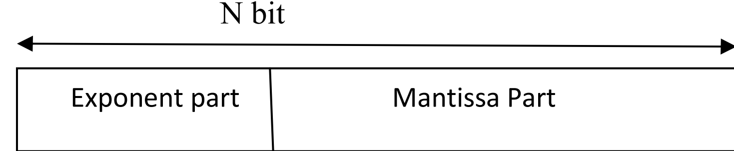


Figure 1.2 Floating point number representation

CHAPTER 2
LITERATURE REVIEW

2.1 Original CORDIC

The modern CORDIC algorithm was first described in 1959 by Jack E. Volder [1]. It was developed at the aeroelectronics department of Convair to replace the analog resolver in the B-58 bomber's navigation computer. The CORDIC airborne navigational computer built for this purpose, outperformed conventional contemporary computers by a factor of 7, mainly due to the revolutionary development of the CORDIC algorithm. Although CORDIC is similar to mathematical techniques published by Henry Briggs as early as 1624, it is optimized for low complexity finite state CPUs. John Stephen Walther at Hewlett-Packard further generalized the algorithm, allowing it to calculate hyperbolic and exponential functions, logarithms, multiplications, divisions and square roots [2]. He told how the unified CORDIC algorithm i.e. combining rotations in the circular, hyperbolic and linear co-ordinate systems and how it was applied in the HP-2116 floating point numerical co-processor. Originally, CORDIC was implemented using the binary numeral system. In the 1970s, decimal CORDIC became widely used in pocket calculators, most of which operate in binary-coded-decimal (BCD) rather than binary.

Sine and Cosine waves have been used in countless applications; in recent research on Software Defined Radio (SDR), digital modalities of sine and cosine waves have received special attention. SDR involves highly reconfigurable resources and uses digital generated waves for modulation and demodulation of signals. Coordinate Rotation Digital Computer (CORDIC) is a well known algorithm used to approximate iteratively some transcendental functions. Arias et.al [15] presented a pipelined CORDIC architecture which is used for designing a flexible and scalable digital sine and cosine waves generator.

A scale factor compensation inherent to the CORDIC algorithm becomes an important drawback when trying to improve its benefits, although some authors have come up with a new scaling-free version, which has been successfully implemented within wireless applications. However, this new CORDIC can still be significantly improved by modifying some of its parts, therefore, Zapata et.al [16] showed an enhanced version of the scaling-free CORDIC. These new enhancements have been obtained some

new architecture which are able to reach a 35% lower latency and a 36% reduction in area and power consumption compared to the original scaling-free architecture.

In conventional CORDIC algorithm, multiplier and a lookup table are needed to achieve calculation of multiple transcendental functions, which will lead to hardware circuit complexity and lower operation speed. Aim at overcoming the shortcomings of traditional CORDIC algorithm, a modified CORDIC algorithm was proposed by Xin et.al [17]. The method does not need the module of correction factor and the lookup table, and just needs a simple shift and add-subtract to achieve the calculation of multiple transcendental function. So it can reduce hardware costs and improve operational performance.

Many hardware efficient algorithms exists for hardware signal processing architecture. Among these algorithm is a set of shift-add algorithms collectively known as CORDIC (COrdinate Rotation for Digital Computers) for computing a wide range of functions including certain trigonometric, hyperbolic, linear and logarithmic functions. Sinith et.al [18] compared the different CORDIC architectures with respect to their area, speed, and data throughput performance especially in three different major styles iterative, parallel and pipelined structures.

Khare et.al [19] presented an area-time efficient CORDIC algorithm that completely eliminates the scale-factor. By suitable selection of the order of approximation of Taylor series the proposed CORDIC circuit meets the accuracy requirement, and attains the desired range of convergence. Besides they have proposed an algorithm to redefine the elementary angles for reducing the number of CORDIC iterations. The proposed CORDIC processor provides the flexibility to manipulate the number of iterations depending on the accuracy, area and latency requirements. A scale factor compensation inherent to the CORDIC algorithm becomes an important drawback when trying to improve its benefits, although some authors have come up with a new scaling-free version, which has been successfully implemented within wireless applications.

2.2 Control CORDIC

Although the CORDIC hardware is very simple, consisting only of 2 shifters and3 adders, it is able to evaluate a wide variety of elementary functions, and consequently it has found its use in many different engineering applications. Prior work by researchers in

this field has concentrated upon improving different aspects of the algorithm, depending upon the characteristics of the application for which it was intended.

Online CORDIC was developed by Ercegovac and Lang [10] for applications where input bits became available serially. Their method could also compensate for the value of K online. For applications that require increased throughput, pipelined CORDIC [14] can be useful. After an initial start-up period, it allows a rotation to be completed every cycle, but involves heavy duplication of hardware in each pipeline stage which is wasteful of power and area. In addition, the iteration count remains unchanged.

Some methods have focussed on reducing the amount of hardware required for CORDIC. One way to reduce the hardware complexity is by combining pairing iterations as shown in [20], which results in smaller shifters having to be used. The Hybrid CORDIC method [21] reduces the amount of ROM space required by approximating the 20 arctan of the angle constant, by the angle constant itself for the last two-thirds of the CORDIC iterations. The Online CORDIC method [22] replaces variable shifters by more area-efficient delays.

Several methods have focussed on the problem of efficiently compensating for the scale factor [23]. The scale factor can be compensated for in parallel, while the CORDIC iterations are being executed. Another method is to perform additional scaling iterations which force the overall scaling factor to unity. Yet another method is repeat some of the CORDIC iterations so as to force K to be power of the machine radix, requiring only a simple shift operation at the end to get the scaled results. There have been several attempts at trying to reduce the latency of CORDIC operations. Some have tried to use a high radix number system to perform the computations [24]. In this case, fewer iterations are required to achieve a given precision at the expense of a more complex selection function as well as the cost of radix conversion.

Control CORDIC [11] uses damping techniques from control theory to reduce the iteration count by about 11% in dynamic situations. The method of Angle Recoding can achieve a 50% or more reduction in the iteration count, but it is confined to static applications such as the chirp-Z transform [25] where the rotation angle is static and known *a* priori. In such cases, the angle constants which can be skipped over can be computed offline in advance.

The CORDIC technique uses a one bit at a time approach to make computation to an arbitrary precision [26]. Typically, these tables only one to two entries per bit of precision. CORDIC algorithm also uses only right shifts and additions, minimizing the computation time. It is hardware efficient algorithm because no multiplier are presenting in CORDIC, to save gate required implementing on FPGA. If multiplier is present, then cost and number of gate increases. The CORDIC has become widely used approach to elementary function evaluation where silicon area is a primary constraint. The implementation of CORDIC algorithm requires less complex hardware. Hardware multiplier is unavailable in CORDIC because the strength of CORDIC algorithm is its ability to solve with vector rotation without using multiplier and to speed up CORDIC algorithm [27]. The only operation is addition, subtraction, bit shift and lookup table. The rotated vector is also scaled making a scale factor necessary. Due to high speed, low cost and greater flexibility offered by FPGA over DSP processors, the FPGA based computing is becoming the heart of all digital signal processing systems of modern era.

2.3 Adaptive CORDIC

Phatak [28] proposed to execute two iterations in the same cycle, using dual CORDIC units. In another method redundant arithmetic is used for CORDIC algorithm which allows fast redundant adders to be used, to reduce the cycle time. There has been a considerable amount of work devoted to solving problems such as sign detection that are associated with using redundant arithmetic . In contrast, there has been very little research performed on investigating the reduction in latency by skipping over some iterations. This is mainly because of the attendant inconvenience of having a variable scaling factor and the need to store these in a ROM. However with there being no shortage of available gates in modern chips, ROM space is much more readily available for use, than it used to be. Accordingly it makes 21 eminent sense to examine methods which may incur a variable K by jumping over iterations, if in doing so, they reduce the number of iterations considerably.

Year 2009 marks the completion of 50 years of the invention of CORDIC. Due to the rapid advances in VLSI technologies have led the way to an entirely different approach to computer-design for real-time application, using special purpose architecture with custom chips. Now a day’s world of information interchange revolves around transmission and viewing real time images. Many of DSP application try to closely stimulate real life images. Many of DSP applications try to closely simulate real life

images. Speed, clarity and resemblance to real time object are some of the issues to be addressed in order to achieve the goel.[4]

The Angle Recoding method proposed by Naganathan et.al [12] used a greedy algorithm to skip over some rotation angles, and can reduce the number of iterations required. The maximum number of iterations required by this method is N/2, with an average value of approximately N/3 iterations.

Rodrigues et.al [29] presented a simpler implementation of the angle selection scheme that does not require an increase in cycle time, thus allowing the Angle Recoding method to be used dynamically for arbitrary angles. The method also has the advantage that all the angle constants are found in parallel, in a single step, by testing only the initial rotation angle, without having to perform successive CORDIC iterations. This dynamic Angle Recoding method can be formulated to use "sections," to limit the number of range comparators needed, to a reasonable value. There is an increase in the number of adaptive CORDIC iterations needed, but this problem can be mitigated by using a buffer in conjunction with the method of section.

The architecture and the implementation of a complex fast Fourier transform (CFFT) processor using 0.6 μm gallium arsenide (GaAs) technology was presented by Sarmiento et.al [30]. This processor computes a 1024-point FFT of 16 bit complex data in less than 8 μs, working at a frequency beyond 700 MHz, with a power consumption of 12.5 W. The architecture of the processor is based on the COordinate Rotation DIgital Computer (CORDIC) algorithm, which avoids the use of conventional multiplication-and- accumulation (MAC) units, but evaluates the trigonometric functions using only add and shift operations. Improvements to the basic CORDIC architecture are introduced in order to reduce the area and power of the processor. This together with the use of pipelining and carry save adders produces a very regular and fast processor.

A new memory less CORDIC algorithm for the FFT computation was proposed by Garrido et.al [31]. This approach calculates the direction of the micro-rotations from the control counter of the FFT, so the area of the rotator hardly depends on the number of rotations, which is particularly suitable for the computation of FFTs of a high number of points. Moreover, the new CORDIC presents other advantages such as the simplification of the basic CORDIC processor used to calculate the micro-rotations, or an easy way to compensate the intrinsic gain of the CORDIC algorithm.

2.4 Floating point and fixed point CORDIC algorithm

Fixed-point Fast Fourier Transform (FFT) units are widely used in digital communication systems. The twiddle multipliers required for realizing large FFTs are typically implemented with the Coordinate Rotation Digital Computer (CORDIC) algorithm to restrict memory requirements. Recent approaches aiming to optimize the bit-widths of FFT units while satisfying a given maximum bound on Mean-Square- Error (MSE) mostly focus on the architectures with integer multipliers. They ignore the quantization error of coefficients, disabling them to analyze the exact error defined as the difference between the fixed-point circuit and the reference floating-point model. Radecka et.al. presents an efficient analysis of MSE as well as an optimization algorithm for CORDIC based FFT units, which is applicable to other Linear-Time-Invariant (LTI) circuits as well [32].

Computation of floating-point transcendental functions has a relevant importance in a wide variety of scientific applications, where the area cost, error and latency are important requirements to be attended. Daniel et.al [33] describes a flexible FPGA implementation of a parameterizable floating-point library for computing sine, cosine, arctangent and exponential functions using the CORDIC algorithm. The novelty of the proposed architecture is that by sharing the same resources the CORDIC algorithm can be used in two operation modes, allowing it to compute the sine, cosine or arctangent functions. Additionally, in case of the exponential function, the architectures change automatically between the CORDIC or a Taylor approach, which helps to improve the precision characteristics of the circuit, specifically for small input values after the argument reduction. Synthesis of the circuits and an experimental analysis of the errors have demonstrated the correctness and effectiveness of the implemented cores and allow the designer to choose, for general-purpose applications, a suitable bit-width representation and number of iterations of the CORDIC algorithm.

FPGA chips have become a promising option for accelerating scientific applications, which involve many floating-point transcendental functions, such as sin, log, exp, sqrt and etc. Jie et.al [34] presents a 64-bit ANSI/IEEE floating- point CORDIC coprocessor on FPGA, providing all known CORDIC functions. They propose a hybrid-mode CORDIC algorithm, combining hybrid rotation angle methods with argument reduction algorithm to reduce hardware area usage and meanwhile keep unlimited

convergence domain for any floating-point inputs of the functions. There hybrid-mode CORDIC co-processor is organized into three phases, argument reduction, CORDIC calculation and normalization with 69 pipeline stages for FPGA implementation.

2.5 Gaps in the study

- While there are numerous articles covering various aspects of CORDIC algorithms, most of work related to CORDIC is done on matlab, very few survey more than one or two, and even fewer concentrate on implementation in FPGA.
- Quite high latency was attained while implementing CORDIC algorithm. There has been very little research performed on investigating the reduction in latency by skipping over some iterations.
- Parallel Angle Recoding method which focuses on extending the Static Angle Recoding technique so that the benefits of the reduced iteration count can be available even in dynamic cases, when the angle of rotation is variable.
- Implementation of Control CORDIC and Angle Recoding CORDIC has been done upto 24 bits. Control CORDIC and Angle Recoding CORDIC can be further extended upto 32 bit.
- CORDIC algorithm are mainly been implemented on fixed point numbers. CORDIC can also be implemented for floating point numbers so that it can support a much wider range of values.
- CORDIC has found its wide application as in robotics, 3 D graphics etc. CORDIC can be further modified for its use in digital camera, MP-3 players and other smart computing devices available in the market today.
- The latency of CORDIC can be further reduced by using hardwired shifters with the Adaptive CORDIC method based on Parallel Angle Recoding delay because the complex shifter contributes considerably to the cycle time of the CORDIC cycle.

2.6 Thesis Objective

- To study and implement Original CORDIC, Control CORDIC and Angle Recoding CORDIC using VHDL.
- To implement the above CORDIC techniques for 16 bit, 24 bit and 32 bit fixed point number using VHDL.

- To implement Original CORDIC techniques for 24 bit, 28 bit and 32 bit floating point number using VHDL programming code.
- To implement CORDIC techniques using fixed and floating point numbers on Xilinx 13.1.

2.7 Thesis Organization

Chapter 1: Introduction about the CORDIC algorithm and the objective of this thesis.

Chapter 2: In this chapter the past contributions made by different researchers in this field are documented.

Chapter 3: Starts with the introduction of Original CORDIC which is the basis of all CORDIC algorithms and its block diagram is also studied. Discuss its basic equation and how it is used for the calculation of trigonometric function. Comparison between Original CORDIC, Control CORDIC and Angle Recoding CORDIC is done. Discusses about the floating point numbers and the fixed point numbers, their formats and various exceptions held by floating point numbers.

Chapter 4: This chapter is mainly concerned with FPGA and its design flow.

Chapter 5: This chapter contains the results of simulation ModelSim6.3f and XILINX 13.1.

Chapter 6: Derives the Conclusion and tells about future scope.

CHAPTER 3
CORDIC ALGORITHM

3.1 Introduction

The CORDIC algorithm was first introduced by Volder [1] for the computation of trigonometric functions, multiplication, division and data type conversion, and later on generalized to hyperbolic functions by Walther [2]. The CORDIC algorithm does not use Calculus based methods such as polynomial or rational function approximation. 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, no actual multiplier is needed as compared to other approaches. CORDIC makes it possible to perform rotations and therefore to compute sine, cosine, and arctangent functions and to multiply or divide numbers, using only shift-and-add elementary steps. So CORDIC is a clear winner when a hardware multiplier is unavailable, e.g. in a microcontroller, or when we want to save the gates 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 CORDIC algorithms require only shifts, add and table lookups i.e. 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.

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 implementation.

3.2 Basic equation of CORDIC algorithm

Volder's algorithm [4] is derived from the general equations for a vector rotation. If a vector V with coordinates (x, y) is rotated through an angle □ then a new vector V' can be obtained with coordinates (x', y') where x' and y' can be obtained using x, y and Ø by the following method.

$$X = r\cos\theta, Y = r\sin\theta \tag{3.1}$$

$$V' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x\cos\emptyset - y\sin\emptyset \\ y\cos\emptyset + x\sin\emptyset \end{bmatrix} \tag{3.2}$$

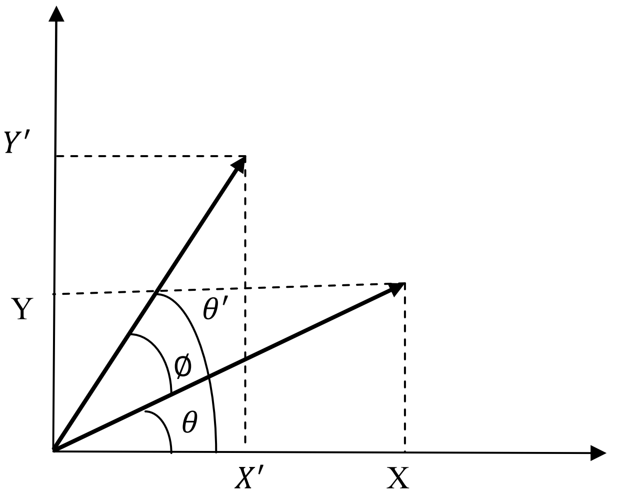


Figure 3.1 Rotation of a vector V by the angle ∅ [4]

Let,s find how equation (3.1) and (3.2) came into picture. As shown in the figure 3.1, a vector V(x,y) can be resolved in two parts along the x-axis and y-axis as $r\cos\theta$ and $r\sin\theta$ respectively. Figure 3.2 illustrates the rotation of a vector $V = \begin{bmatrix} x \\ y \end{bmatrix}$ by the angle θ.

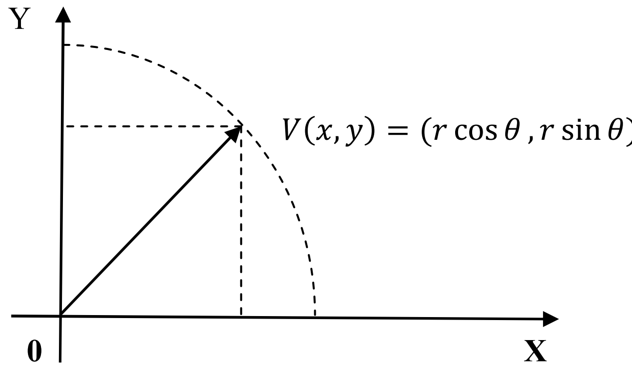


Figure 3.2 Vector V with magnitude r and phase θ

i.e.
$$\left.\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned}\right\} \tag{3.3}$$

Similarly from figure 3.1 it can be seen that vector V and V' can be resolved in two parts. Let V has its magnitude and phase as r and θ respectively and V' has its magnitude and phase as r and θ' where V' came into picture after anticlockwise rotation of vector V by an angle ∅. From figure 3.1 it can be observed

$$\theta' - \theta = \emptyset \tag{3.4}$$

i.e. $$\theta' = \theta + \emptyset \qquad (3.5)$$

$$\begin{aligned} OX' = x' &= r\cos\theta' \\ &= r\cos(\theta + \emptyset) \\ &= r(\cos\theta\cos\emptyset - \sin\theta\sin\emptyset) \\ &= (r\cos\theta)\cos\emptyset - (r\sin\theta)\sin\emptyset \qquad (3.6) \end{aligned}$$

Using figure 3.2 and equation 3.6, OX' can be represented as

$$OX' = x' = x\cos\emptyset - y\sin\emptyset \qquad (3.7)$$

Similarly,

$$\begin{aligned} OY' = y' &= r\sin\theta' \\ &= r\sin(\theta + \emptyset) \\ &= r(\sin\theta\cos\emptyset + \cos\theta\sin\emptyset) \\ &= (r\sin\theta)\cos\emptyset + (r\cos\theta)\sin\emptyset \\ &= y\cos\emptyset + x\sin\emptyset \qquad (3.8) \end{aligned}$$

V' after anticlockwise rotation of vector V by angle $\emptyset$ is

$$x' = x\cos\emptyset - y\sin\emptyset$$
$$y' = y\cos\emptyset + x\sin\emptyset$$

Similarly, value for vector V' in the clockwise direction rotating the vector V by the angle $\emptyset$ is

$$x' = x\cos\emptyset + y\sin\emptyset \qquad (3.9)$$
$$y' = y\cos\emptyset - x\sin\emptyset \qquad (3.10)$$

The equations (3.7), (3.8), (3.9), (3.10) can be represented in the matrix form as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\emptyset & \mp\sin\emptyset \\ \pm\sin\emptyset & \cos\emptyset \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad (3.11)$$

V' after anticlockwise rotation of vector V by angle $\emptyset$ is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\emptyset & -\sin\emptyset \\ \sin\emptyset & \cos\emptyset \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Volder [1] observed that by factoring out a $\cos\emptyset$ from both sides, resulting equation be in terms of the tangent of the angle $\emptyset$, the angle of which we want to find the sine and cosine values. Next if it is assumed that the angle θ is being an aggregate of small angles and composite angles are chosen such that their tangents are all inverse powers of two, then this equation [35] can be rewritten as an iterative formula.

$$x' = \cos\emptyset\,(x - y\tan\emptyset) \qquad (3.12)$$

$$y' = \cos\emptyset\,(y + x\tan\emptyset) \qquad (3.13)$$

$z' = z + \emptyset$, here $\emptyset$ is the angle of rotation (± sign is showing the direction of rotation) and z is the angle accumulator.

The multiplication by the tangent term can be avoided if the rotation angles and therefore $\tan\phi$ are restricted so that $\tan\emptyset = 2^{-i}$. In digital hardware this denotes a simple shift operation. Furthermore, if those rotations are performed iteratively and in both directions every value of $\tan\emptyset$ is representable. With $\emptyset = \tan^{-1}2^{-i}$, the cosine term could also be simplified and since $\cos(\emptyset) = \cos(-\emptyset)$ and it is a constant for a fixed number of iterations.

$$x' = \cos(\emptyset)\,(x - y.\,2^{-i})$$

$$y' = \cos(\emptyset)\,(y + x.\,2^{-i})$$

$\emptyset$ can be positive or negative depending upon whether the rotation is anticlockwise or clockwise. Equation (3.11) can be expressed as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos\emptyset \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad (3.14)$$

Again modifying equation (3.14) for both clockwise and anticlockwise rotation. It can be expressed as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos\emptyset \begin{bmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad (3.15)$$

This iterative equation [35] can now be expressed as:

$$x_{i+1} = k_i(x_i - y_i d_i 2^{-i}) \qquad (3.16)$$

$$y_{i+1} = k_i(y_i + x_i d_i 2^{-i}) \qquad (3.17)$$

where, i denotes the number of rotation required to reach the required angle of the required vector, $k_i = \cos(\tan^{-1}(2^{-i}))$ and $d_i = \pm 1$ the product of k_i represents the so-called K factor.

$$K = \prod_{i=0}^{n-1} k_i$$

where $\prod_{i=0}^{i=n-1} k_i = \cos\emptyset_1 \cos\emptyset_2 \cos\emptyset_3 \cdots\cdots \cos\emptyset_{n-1}$ ($\emptyset$ is the angle of rotation here for n times rotation).

k_i is the CORDIC gain. For 16-bit hardware CORDIC approximation method, the value of k_i as

$k_i = \prod_{i=0}^{15} \cos \emptyset_i$

$= \cos\phi_0 \cos\phi_1 \cos\phi_2 \cos\phi_3 \cos\phi_4 \cos\phi_5 \cos\phi_6 \cos\phi_7$

$\cos\phi_8 \cos\phi_9 \cos\phi_{10} \cos\phi_{11} \cos\phi_{12} \cos\phi_{13} \cos\phi_{14} \cos\phi_{15}$

$= \cos 45^\circ \cos 26.565^\circ \cos 14.036^\circ \dots\dots\dots \cos 0.00699^\circ \cos 0.00349^\circ \cos 0.00175^\circ$

= 0.6073

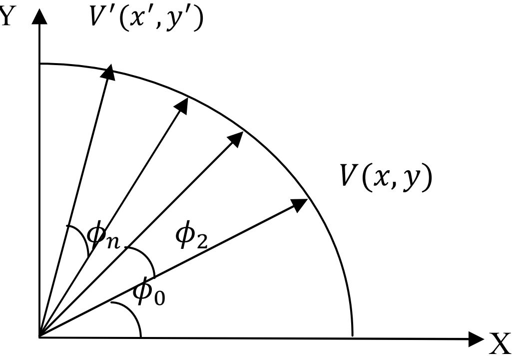


Figure 3.3 Chaining multiple micro-rotations together [29]

As shown in figure 3.3, multiple micro rotations may be chained together as the vector moves in discrete angular steps $(\phi_0, \phi_{1,}\phi_{2,} \dots\dots\dots \phi_n)$ from its initial position of $V(x, y)$ towards its target position of $V'(x', y')$.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos\phi_0 \cos\phi_1 \dots \cos\phi_n \begin{bmatrix} 1 & -\tan\phi_0 \\ \tan\phi_0 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & -\tan\phi_n \\ \tan\phi_n & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (3.17)$$

For 16 bit CORDIC hardware, cosine and sine of angle can be represented in matrix form by using equation (3.17)

$$\begin{bmatrix} \cos\phi \\ \sin\phi \end{bmatrix} = \begin{bmatrix} 1 & -\tan\phi_0 \\ \tan\phi_0 & 1 \end{bmatrix} \dots\dots\dots. \begin{bmatrix} 1 & -\tan\phi_{15} \\ \tan\phi_{15} & 1 \end{bmatrix} \begin{bmatrix} 0.6073 \\ 0 \end{bmatrix} \quad (3.18)$$

For 24 bit CORDIC hardware, the value of k_i is

$$k_i = \prod_{i=0}^{i=23} \cos\phi_i = \cos\phi_0 \cos\phi_1 \dots\dots\dots. \cos\phi_{22} \cos\phi_{23} = 0.6073$$

For 32 bit CORDIC hardware, the value of k_i is

$$k_i = \prod_{i=0}^{i=31} \cos\phi_i = \cos\phi_0 \cos\phi_1 \dots\dots \cos\phi_{30} \cos\phi_{31} = 0.6073$$

CORDIC gain k_i is same for all the CORDIC hardware.

Table 3.1 For 16, 24 and 32 bit CORDIC hardware [29]

i	$2^{-i} = \tan^{-1}\phi_i$	$\phi_i = \tan^{-1}(2^{-i})$	ϕ_i in radians
0	1	45°	0.7854
1	0.5	26.565°	0.4636
2	0.25	14.036°	0.2450
3	0.125	7.124°	0.1244
4	0.0625	3.576°	0.0624
5	0.03125	1.7876°	0.0312
6	0.015625	0.8938°	0.0156
7	0.0078125	0.4469°	0.0078
8	0.00390625	0.2238°	0.0039
9	0.001953125	0.1119°	0.0019
10	0.0009765625	0.05595°	0.00098
11	0.00048828125	0.02798°	0.00049
12	0.000244140625	0.01399°	0.00024
13	0.0001220703125	0.00699°	0.00012
14	0.00006103515625	0.00349°	0.000061
15	0.00003051757813	0.00175°	0.000031
16	0.00001525878906	0.000874°	0.0000152
17	0.000007629394531	0.000437°	0.0000076
18	0.000003814697266	0.0002186°	0.0000038
19	0.000001907348633	0.0001093°	0.0000019
20	0.0000009536743164	0.00005464°	0.00000095
21	0.0000004768371582	0.0000273°	0.00000048
22	0.0000002384185791	0.00001366°	0.00000024
23	0.0000001192092896	0.00000683°	0.00000012
24	0.00000005960464478	0.000003415°	0.0000000595
25	0.00000002980232239	0.000001708°	0.0000000298
26	0.00000001490116119	0.000000854°	0.0000000149
27	0.000000007450580597	0.000000427°	0.00000000745
28	0.000000003725290298	0.000000213°	0.00000000371
29	0.000000001862645149	0.000000107°	0.00000000187
30	0.0000000009313225746	0.0000000534°	0.00000000093
31	0.0000000004656612873	0.0000000267°	0.00000000047

Table 3.2 CORDIC gain k_i for 16, 24 and 32 bit hardware [29]

i	$2^{-i} = \tan^{-1}\phi_i$	$\phi_i = \tan^{-1}(2^{-i})$	$\cos\phi_i$
0	1	45°	0.7071
1	0.5	26.565°	0.8944
2	0.25	14.036°	0.9701
3	0.125	7.124°	0.9922
4	0.0625	3.576°	0.9980
5	0.03125	1.7876°	0.9995
6	0.015625	0.8938°	0.9998
7	0.0078125	0.4469°	0.9999
8	0.00390625	0.2238°	0.9999
9	0.001953125	0.1119°	0.9999
10	0.0009765625	0.05595°	0.9999
11	0.00048828125	0.02798°	0.9999
12	0.000244140625	0.01399°	0.9999
13	0.0001220703125	0.00699°	0.9999
14	0.00006103515625	0.00349°	0.9999
15	0.00003051757813	0.00175°	0.9999
16	0.00001525878906	0.000874°	1
17	0.000007629394531	0.000437°	1
18	0.000003814697266	0.0002186°	1
19	0.000001907348633	0.0001093°	1
20	0.0000009536743164	0.00005464°	1
21	0.0000004768371582	0.0000273°	1
22	0.0000002384185791	0.00001366°	1
23	0.0000001192092896	0.00000683°	1
24	0.00000005960464478	0.000003415°	1
25	0.00000002980232239	0.000001708°	1
26	0.00000001490116119	0.000000854°	1
27	0.000000007450580597	0.000000427°	1
28	0.000000003725290298	0.000000213°	1
29	0.000000001862645149	0.000000107°	1
30	0.0000000009313225746	0.0000000534°	1
31	0.0000000004656612873	0.0000000267°	1

From table 3.2, it can be seen that precision up to 0.00175° is possible for 16-bit CORDIC hardware. These ϕ_i are stored in ROM of the hardware of the CORDIC hardware as the look up table.

Pseudo rotation: The pseudo rotation [4] is introduced by dropping cosine term. The pseudo rotation is introduced by performing givens rotation: the angle of rotation is correct but x and y values are scaled by $\cos(\phi)$. Note that it can make the computation of plane rotations more amenable to simple operations.

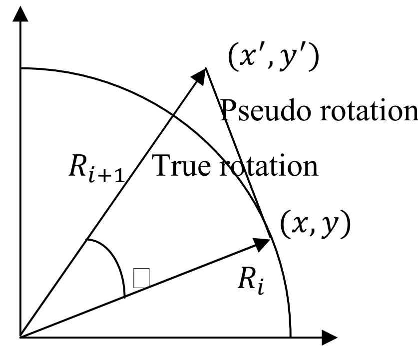


Figure 3.4 Pseudo rotation [4]

Dropping cosine term gives:

$$x' = x - y \tan \phi \tag{3.19}$$

$$y' = y + x \tan \phi \tag{3.20}$$

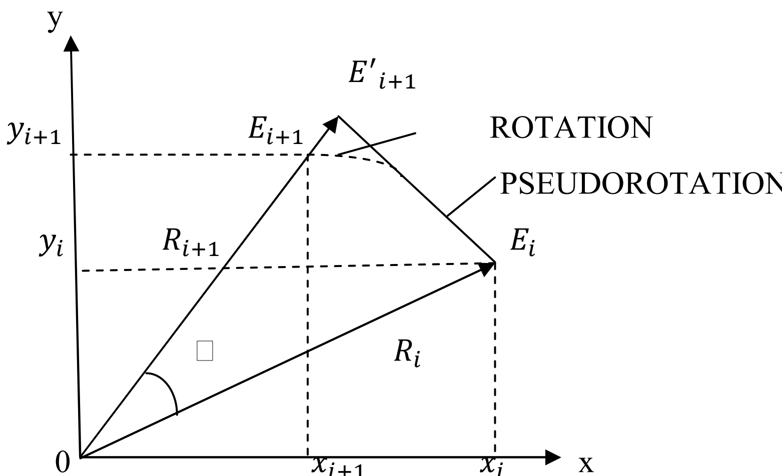


Figure 3.5 Rotation and Pseudo rotation [4]

Rotate the vector OE_i with end point at (x_i, y_i) by ϕ

$$\begin{aligned} x_{i+1} &= x_i \cos \phi_i - y_i \sin \phi_i \\ &= \cos \phi_i \,(x_i - y_i \tan \phi_i) \end{aligned}$$

$$= \frac{(x_i - y_i \tan\phi_i)}{(1 + \tan^2\phi_i)^{1/2}}$$

$$y_{i+1} = y_i \cos\phi_i + x_i \sin\phi_i$$

$$= \cos\phi_i (y_i + x_i \tan\phi_i)$$

$$= \frac{(y_i + x_i \tan\phi_i)}{(1 + \tan^2\phi_i)^{1/2}}$$

Now eliminate the division by $(1 + \tan^2\phi_i)^{1/2}$ and choose ϕ_i so that $\tan\phi_i$ is power of 2. Whereas a real rotation does not changes the length R_i of the vector, a pseudo rotation step increases its length.

From figure 3.5, it is seen that

$$\frac{R_i}{R_{i+1}} = \cos\phi_i$$

$$R_{i+1} = \frac{R_i}{\cos\phi_i}$$

$$R_{i+1} = R_i (1 + \tan^2\phi_i)^{1/2}$$

The coordinates of the new end point E_{i+1} after pseudo rotation is derived by multiplying the coordinates of E_{i+1} by the expansion factor.

$$x_{i+1} = x_i - y_i \tan\phi_i \qquad (3.21)$$

$$y_{i+1} = y_i + x_i \tan\phi_i \qquad (3.22)$$

Original given rotation now reduced to iterative shift add algorithm.

$$x_{i+1} = x_i - d_i y_i 2^{-i} \qquad (3.23)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \qquad (3.24)$$

$$z_{i+1} = z_i - d_i \phi_i \qquad (3.25)$$

$d_i = \pm 1$ (d_i is the direction of angle of rotation)

Each iteration requires :- (1) 2 shifts

(2) 1 look-up table

(3) 3 additions/subtractions

Scaling factor: When simplifying the algorithm [4] [35] to allow pseudo-rotations the cosine term was omitted. Applying to this iterative scheme, the scaling factor K_n can be written as

$$K_n = \prod_{i=0}^{n-1} \left(\frac{1}{\cos\phi_i}\right) = \prod_{i=0}^{n-1} \sqrt{1 + \tan^2\phi_i} = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} = 1.6467$$

As iteration number n is known, K_n can be pre-computed.

Multiplying equation (3.21), (3.22) by $\cos\phi_i$, we get

$$\cos\phi_i \,.\, x_{i+1} = \cos\phi_i\,(x_i - y_i \tan\phi_i)$$
$$\cos\phi_i \,.\, y_{i+1} = \cos\phi_i\,(y_i + x_i \tan\phi_i)$$

or

$$\cos\phi_i \,.\, x_{i+1} = (x_i \cos\phi_i - y_i \sin\phi_i)$$
$$\cos\phi_i \,.\, y_{i+1} = (y_i \cos\phi_i + x_i \sin\phi_i)$$

After n iterations, we have

$$x_n = K_n(x_0 \cos\phi_i - y_0 \sin\phi_i) \qquad (3.26)$$
$$y_n = K_n(y_0 \cos\phi_i + x_0 \sin\phi_i) \qquad (3.27)$$

To find $\cos\phi_i$ and $\sin\phi_i$:- $x_0 = 1, y_0 = 0$

$\frac{x_n}{K_n} = \cos\phi_i$ and $\frac{y_n}{K_n} = \sin\phi_i$

By taking an example of balance it can be understood that how the CORDIC algorithm works.

In the figure (3.6), first of all, keep the angle θ on the left pan balance and if the balance rotates around the anticlockwise direction then add the highest value in the table at the other side.

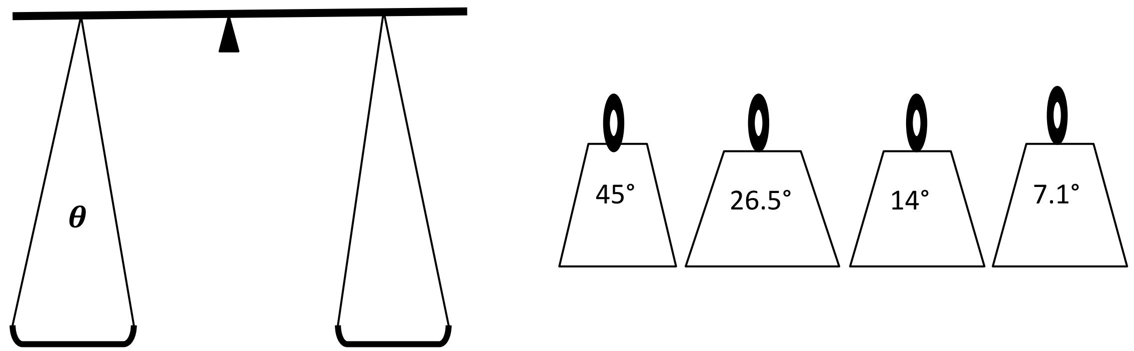


Figure 3.6 A balance having θ at one side and small weights (angle) at the other side

If the balance show a left inclination as in figure 3.7 (a) then other weights are required to add in the right pan or in the term of angle if θ is greater than total ϕ_i then add other weights to reach as near to the θ as possible but in other hand if the balance show a right inclination as in figure 3.7 (b) then a weight required to be removed from the right pan or

in the term of angle if θ is less than total ϕ_i then we subtract other weights this process is repeated to reach as near to the θ as possible.

Figure 3.7 Inclined balance due to difference in weight of two sides

For example: $\theta = 30.0^{\circ}$

Then start with $\emptyset_0 = 45°$ $(45° > 30.0°)$

$45° - 26.565° = 18.44°$ $(18.44° < 30.0°)$

$18.44° + 14.036° = 32.47°$ $(32.47° > 30.0°)$

$32.47° - 7.125° = 25.346°$ $(25.546° < 30.0°)$

$25.346° + 3.576° = 28.913°$ $(28.913° < 30.0°)$

$28.913° + 1.7876° = 30.7°$ $(30.7° > 30.0°)$

$30.7° - 0.89° = 29.8°$ $(29.8° < 30.0°)$

$29.8° + 0.44° = 30.24°$

$\theta = 30°$

$= 45.0° - 26.565° + 14.036° - 7.125° + 3.576° + 1.7876° - 0.89° + 0.44°$

$= 30.2°$

3.3 Basic CORDIC iterations

The basic CORDIC iterations [35] are

$$x_{i+1} = x_i - d_i y_i 2^{-i} \quad (3.28)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \quad (3.29)$$

$$z_{i+1} = z_i - d_i \phi_i \quad (3.30)$$

$d_i = \pm 1$ (d_i is the direction of angle of rotation)

The computation of x_{i+1} and y_{i+1} requires i bit shift and an add/subtract. If the function $\phi_i = \tan^{-1}(2^{-i})$ is pre-computed and stored in table 3.1 for different value of i, a single

add/subtract suffices to compute z_{i+1}. Each CORDIC iteration thus involves two shifts, a table lookup and three additions/subtractions.
If the rotation is done by the same set of angles (with + or - signs), then expansion factor or scaling factor K_n is constant and can be pre-computed.

CORDIC iterations can be used in two operating modes. These two modes differ basically on how the directions of the micro rotations are chosen:

- Rotation mode (RM)
- Vectoring mode (VM)

In the rotation mode [35][36], the angle accumulator is initialized with the desired rotation angle ($z_0 = \theta$). The rotation decision at each iteration is made to diminish the magnitude based on the sign of the residual angle in the angle accumulator. The decision at each iteration is therefore based on the sign of the residual angle after each step.
For rotation mode, the CORDIC equations are

$$x_{i+1} = x_i - d_i y_i 2^{-i} \quad (3.31)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \quad (3.32)$$

$$z_{i+1} = z_i - d_i \phi_i \quad (3.33)$$

$d_i = \pm 1$ (d_i is the direction of angle of rotation)

$$d_i = \begin{cases} +1, if\ z_i \geq 0 \\ -1, if\ z_i < 0 \end{cases} \quad (3.34)$$

After n iteration in CORDIC mode, z_n is sufficiently close to zero, we have

$$z = \sum \phi_i \quad (3.35)$$

The output of the rotation mode $x_n, y_n and\ z_n$ are given by the following expressions, $x_n\ and\ y_n$ being the co-ordinates of the rotated vector.

$$x_n = K_n(x_0 \cos z_0 - y_0 \sin z_0) \quad (3.36)$$

$$y_n = K_n(y_0 \cos z_0 + x_0 \sin z_0) \quad (3.37)$$

$$z_n = 0 \quad (3.38)$$

where $K_n = \prod_{i=0}^{n-1}(1 + 2^{-2i})^{1/2} = 1.6467$

Sine and Cosine Computation using the CORDIC Method

The rotation mode of the CORDIC algorithm could be used to compute sine and cosine of an angle θ. The computation of $\sin\theta$ and $\cos\theta$ is based on the rotation of an initial vector of unit length, that is aligned with the abscissa ($x_0 = 1, y_0 = 0$).

Input values for n iterations:

$$x_0 = 1, \quad y_0 = 0, \quad z_0 = \theta$$

Put the initial condition in equations (3.36), (3.37) to get $\cos\theta$ and $\sin\theta$, we get

$$\frac{x_n}{K_n} = \cos\phi_i \quad \text{and} \quad \frac{y_n}{K_n} = \sin\phi_i \qquad (3.39)$$

Once $\sin z_0$ and $\cos z_0$ are known, $\tan z_0$ can be computed necessary through division.

For example: rotation mode ($\theta = 30^\circ$)

$$z_0 = \theta = 30^\circ$$

Table 3.3 Choosing the sign of the rotation angles to force z to zero [35]

i	$z_i - d_i\phi_i$	z_{i+1}	d_i	$\theta = 30^\circ$
0	$+30.0^\circ - 45.0^\circ$	-15° $(30.0^\circ > 0)$	+1	$0 + 45.0^\circ = 45.0^\circ$
1	$-15^\circ + 26.565^\circ$	$+11.6^\circ$ $(-15^\circ < 0)$	-1	$45.0^\circ - 26.565^\circ = 18.44^\circ$
2	$+11.6^\circ - 14.0^\circ$	-2.4° $(11.6^\circ > 0)$	+1	$18.44^\circ + 14.036^\circ = 32.47^\circ$
3	$-2.4^\circ + 7.1^\circ$	$+4.7^\circ$ $(-2.4^\circ < 0)$	-1	$32.47^\circ - 7.125^\circ = 25.346^\circ$
4	$+4.7^\circ - 3.6^\circ$	$+1.1^\circ$ $(4.7^\circ > 0)$	+1	$25.346^\circ + 3.567^\circ = 28.913^\circ$
5	$+1.1^\circ - 1.8^\circ$	-0.7° $(1.1^\circ > 0)$	+1	$28.914^\circ + 1.7876^\circ = 30.70^\circ$
6	$-0.7^\circ + 0.9^\circ$	$+0.2^\circ$ $(-0.7^\circ < 0)$	-1	$30.70^\circ - 0.89^\circ = 29.8^\circ$
7	$+0.2^\circ - 0.4^\circ$	-0.2° $(0.2^\circ > 0)$	+1	$29.8^\circ + 0.44^\circ = 30.24^\circ$
8	$-0.2^\circ + 0.2^\circ$	0.0° $(-0.2^\circ < 0)$	-1	$30.24^\circ - 0.2238^\circ = 30.1^\circ$

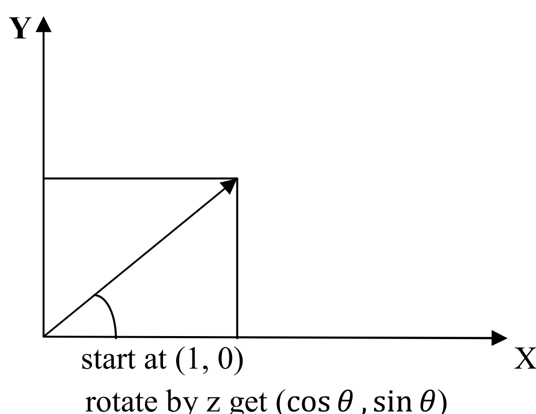


Figure 3.8 Rotation mode to compute ($\cos\theta$ *and* $\sin\theta$)

Outputs after n micro-rotations:

$$x_n = K_n(x_0\cos\theta - y_0\sin\theta) \qquad (3.40)$$

$$y_n = K_n(y_0\cos\theta + x_0\sin\theta) \qquad (3.41)$$

$$z_n = 0 \quad (3.42)$$

Rule: choose $d_i \epsilon \{-1, +1\}$ such that $z \to 0$

where $K_n = \prod_{i=0}^{n-1} (1 + 2^{-2i})^{1/2} = 1.6467$

Put $x_0 = 1$, $y_0 = 0, z_0 = \theta$ (using equation (3.40), (3.41))

$\frac{x_n}{K_n} = \cos \phi_i$ and $\frac{y_n}{K_n} = \sin \phi_i$

Using equations (3.31), (3.32), (3.33), we have to compute x and y

I	$z_{i+1} = z_i - d_i \phi_i$	d_i	Target ($\theta = 30^\circ$)	To compute x and y
0	-15° $(30.0^\circ > 0)$	$+1$	$0 + 45.0^\circ = 45.0^\circ$	$x_1 = 1, y_1 = 0$
1	$+11.6^\circ$ $(-15^\circ < 0)$	-1	$45.0^\circ - 26.565^\circ = 18.44^\circ$	$x_2 = 1.5, y_2 = 0.5$
2	-2.4° $(11.6^\circ > 0)$	$+1$	$18.44^\circ + 14.036^\circ = 32.47^\circ$	$x_3 = 1.375, y_3 = 0.875$
3	$+4.7^\circ$ $(-2.4^\circ < 0)$	-1	$32.47^\circ - 7.125^\circ = 25.346^\circ$	$x_4 = 1.4843, y_4 = 0.7032$
4	$+1.1^\circ$ $(4.7^\circ > 0)$	$+1$	$25.346^\circ + 3.567^\circ = 28.913^\circ$	$x_5 = 1.44035, y_5 = 0.7959$
5	-0.7° $(1.1^\circ > 0)$	$+1$	$28.914^\circ + 1.7876^\circ = 30.70^\circ$	$x_6 = 1.4154, y_6 = 0.8409$
6	$+0.2^\circ$ $(-0.7^\circ < 0)$	-1	$30.70^\circ - 0.89^\circ = 29.8^\circ$	$x_7 = 1.4285, y_7 = 0.8187$
7	-0.2° $(0.2^\circ > 0)$	$+1$	$29.8^\circ + 0.44^\circ = 30.24^\circ$	$x_8 = 1.42210, y_8 = 0.8298$
8	0.0° $(-0.2^\circ < 0)$	-1	$30.24^\circ - 0.2238^\circ = 30.1^\circ$	

$$x_{last} = 1.42210, y_{last} = 0.8298$$

$$\cos\theta = \frac{1.42210}{1.6467} \Rightarrow \cos\theta = 0.8636$$

$$\sin\theta = \frac{0.8298}{1.6467} \Rightarrow \sin\theta = 0.50$$

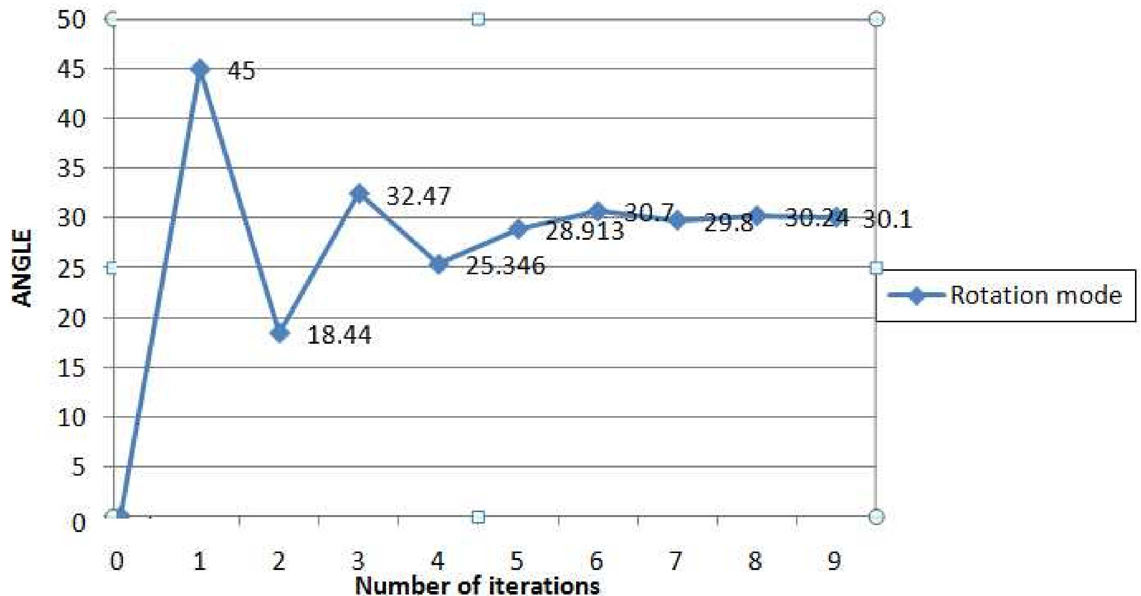


Figure 3.9 Rotation mode Graph of CORDIC algorithm

Vectoring mode: In vectoring mode [35][36], the co-ordinates components of a vector are given and the magnitude and angular argument of the original vector are computed. In the CORDIC, the selection rule for d_i, which make y converge to zero, is known as vectoring mode. It can be used to compute $\tan^{-1} y, \sin^{-1} y$ *and* $\cos^{-1} y$.

In the vectoring mode, the CORDIC rotator rotates the input vector through whatever angle is necessary to align the result vector with the x-axis. The result of the vectoring operation is a rotation angle and its scaled magnitude of the original vector (the x component of the result). The vectoring function works by seeking to minimize the y component of the residual vector at each rotation. The sign of the residual y component is used to determine which direction to rotate next. If the angle accumulator is initialized with zero, it will contain the transverse angle at the end of the iteration.

In vectoring mode, the CORDIC equations are

$$x_{i+1} = x_i - d_i y_i 2^{-i} \tag{3.43}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \tag{3.44}$$

$$z_{i+1} = z_i - d_i \phi_i \tag{3.45}$$

$d_i = \pm 1$ (d_i is the direction of angle of rotation)

$$d_i = \begin{cases} +1, if\ y_i < 0 \\ -1, if\ y_i \geq 0 \end{cases} \tag{3.46}$$

In the vectoring mode, y converges to zero.

$$0 = y + x.2^{-i}$$

$$-y = x.2^{-i}$$

$$2^{-i} = -\frac{y}{x}$$

$$\tan \sum \phi_i = -\frac{y}{x} \quad (3.47)$$

After n iteration, we have

$$x_n = K_n\left(x \cos\left(\sum \phi_i\right) - y \cos\left(\sum \phi_i\right)\right)$$

$$x_n = K_n \cos\left(\sum \phi_i\right)\left(x - y \tan\left(\sum \phi_i\right)\right)$$

$$x_n = \frac{K_n(x - y \tan(\sum \phi_i))}{1 + tan^2(\sum \phi_i)^{1/2}}$$

$$x_n = \frac{K_n(x + \frac{y^2}{x})}{(1 + \frac{y^2}{x^2})^{1/2}}$$

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

$$z_n = z - \tan^{-1}\left(-\frac{y}{x}\right)$$

or

$$z_n = z + \tan^{-1}\left(-\frac{y}{x}\right)$$

$$y_n = 0$$

The CORDIC equations has become

$$x_n = K_n(x^2 + y^2)^{1/2} \quad (3.48)$$

$$z_n = z + \tan^{-1}(-\frac{y}{x}) \quad (3.49)$$

$$y_n = 0 \quad (3.50)$$

Rule: choose $d_i \in \{-1, +1\}$ such that $y \to 0$

One can compute $\tan^{-1} y$ in vectoring mode by starting with x=1 and z=0.

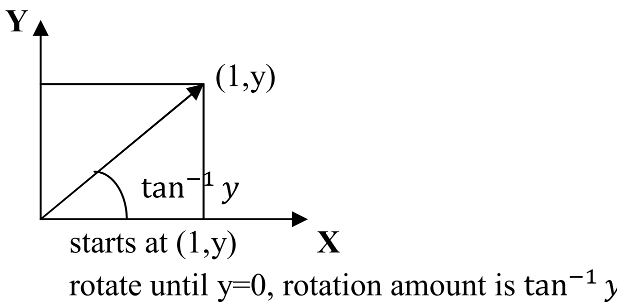


Figure 3.10 Vectoring mode to compute $\tan^{-1} y$

How to compute $\tan^{-1} y$: angle accumulator is initialized to zero $z_0 = 0$.

For example: To find angle, $x_n = 1.732$ and $y_n = 1$ are given

i	y_i	d_i	x_i	z_{i+1}
	1		1.732	0.0°
0	-0.732	-1 $(1 > 0)$	2.732	$0.0^{\circ} + 45^{\circ} = 45^{\circ}$
1	0.634	$+1$ $(-0.732 < 0)$	3.098	$45^{\circ} - 26.565^{\circ} = 18.345^{\circ}$
2	-0.1405	-1 $(0.634 > 0)$	3.2565	$18.345^{\circ} + 14.036^{\circ} = 32.471^{\circ}$
3	0.2665	$+1$ $(-0.1405 < 0)$	3.27406	$32.471^{\circ} - 7.125^{\circ} = 25.346^{\circ}$
4	0.0635	-1 $(0.2665 > 0)$	3.2907	$25.346^{\circ} + 3.125^{\circ} = 28.922^{\circ}$
5	-0.0393	-1 $(0.0635 > 0)$	3.2926	$28.922^{\circ} + 1.7876^{\circ} = 30.70^{\circ}$
6	0.0121	$+1$ $(-0.0393 < 0)$	3.2932	$30.70^{\circ} - 0.8938^{\circ} = 29.80^{\circ}$
7	-0.013	-1 $(0.0121 > 0)$	3.2932	$29.8^{\circ} + 0.4469^{\circ} = 30.249^{\circ}$

Magnitude: $x_R = \sqrt{x^2 + y^2}$

$$x_R = \sqrt{(1.732)^2 + (1)^2} = 1.9999$$

$$y_R = 0$$

From equation (3.40), we get

$$\mathrm{x_n = K_n(x_0 \cos z_0 - y_0 \sin z_0)}$$

$$\text{Put } x_0 = 1, y_0 = 0, z_0 = \theta$$

$$\frac{x_n}{K_n} = \cos\theta$$

$$\cos\theta = \frac{3.2932}{1.6467} = 1.999$$

Trigonometric functions as well as exponential are computed in the Rotation mode. Vectoring mode is used to compute logarithm and arctangent Functions [35].

3.3.1 Unified CORDIC

The CORDIC algorithm iteration equations are shown in equation (3.51), (3.52) and (3.53). There is minimal difference between the equations of a Classic CORDIC and Unified CORDIC. The first difference is in the insertion of α parameter into the x_{i+1} equation. The m parameter determines whether the hardware will perform circular, linear and hyperbolic function. The allowable value of m and their corresponding operational modes are shown in Table 3.3

$$x_{i+1} = x_i - md_iy_i2^{-i} \quad (3.51)$$
$$y_{i+1} = y_i + d_ix_i2^{-i} \quad (3.52)$$
$$z_{i+1} = z_i - d_i\phi_i \quad (3.53)$$

The second difference found in the z_{i+1} equation. Instead of rotating the vector by $\tan^{-1}(2^{-i})$, the generalized function ϕ_i is used. The function ϕ_i is used because rotation function is different for each of the Unified CORDIC algorithm's modes of operation. A list of the operational modes and its rotation function, ϕ_i is shown in table 3.4. The selection of sign bit, d_i, remains the same as the Classic CORDIC.

Table 3.4 Unified CORDIC Operational Modes

m	**Rotation Type**
1	Circular rotation (sin, cos, etc.)
0	Linear rotation (multiplication, division.)
-1	Hyperbolic rotation (sinh, cosh, etc.)

Table 3.5 Unified CORDIC Rotation Functions

Rotation Type	ϕ_i
Circular Rotations	$\tan^{-1}(2^{-i})$
Linear Rotations	2^{-i}
Hyperbolic Rotations	$\tanh^{-1}(2^{-i})$

Circular function: m=1, $\phi_i = \tan^{-1}(2^{-i})$

Put these value in equation (3.51), (3.52), (3.52)

The basic CORDIC iteration are

$$x_{i+1} = x_i - 1d_iy_i2^{-i}$$
$$y_{i+1} = y_i + d_ix_i2^{-i}$$
$$z_{i+1} = z_i - d_i\tan^{-1}(2^{-i})$$
$$d_i = \pm 1$$ (d_i is the direction of angle of rotation)

In rotation mode, z converges to zero

$$z = \sum \phi_i \quad (3.54)$$

The output of the rotation mode x_n, y_n and z_n are given as:

$$\mathrm{x_n} = \mathrm{K_n}(\mathrm{x_0}\cos\theta - \mathrm{y_0}\sin\theta) \quad (3.55)$$

$$y_n = K_n(y_0 \cos\theta + x_0 \sin\theta) \quad (3.56)$$

$$z_n = 0 \quad (3.57)$$

Rule: choose $d_i \epsilon \{-1, +1\}$ such that $z \to 0$

where $K_n = \prod_{i=0}^{n-1}(1 + 2^{-2i})^{1/2} = 1.6467$

In vectoring mode, y converges to zero.

$$0 = y + x.2^{-i}$$

$$-y = x.2^{-i}$$

$$2^{-i} = -\frac{y}{x}$$

$$\tan(\sum \phi_i) = -\frac{y}{x} \quad (3.58)$$

After n iterations, we have

$$x_n = K_n(x^2 + y^2)^{1/2} \quad (3.59)$$

$$z_n = z + \tan^{-1}(-\frac{y}{x}) \quad (3.60)$$

$$y_n = 0 \quad (3.61)$$

Rule: choose $d_i \in \{-1, +1\}$ such that $y \to 0$

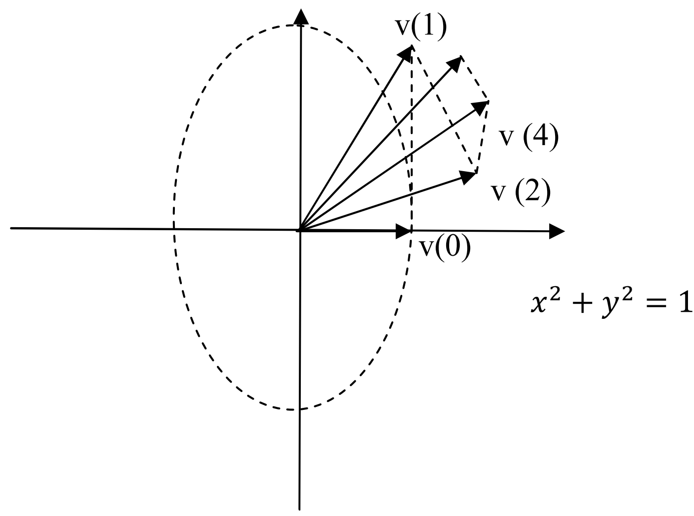


Figure 3.11 Circular Rotation [37]

Linear function: m = 0, $\phi_i = 2^{-i}$

Put these value in equation (3.51), (3.52), (3.52)

The basic CORDIC iteration are

$$x_{i+1} = x_i - 0.y_i 2^{-i}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i}$$

$$z_{i+1} = z_i - d_i 2^{-i}$$

$$d_i = \pm 1 \qquad (d_i \text{ is the direction of angle of rotation})$$

In rotation mode, z converges to zero

$$z = 2^{-i} \qquad (3.62)$$

The output of the rotation mode x_n, y_n and z_n are given as:

$$x_n = x_0 \qquad (3.63)$$

$$y_n = y_0 + x_0 z_0 \qquad (3.64)$$

$$z_n = 0 \qquad (3.65)$$

Rule: choose $d_i \epsilon \{-1, +1\}$ such that $z \to 0$

In vectoring mode, y converges to zero.

$$0 = y + x.2^{-i}$$

$$-y = x.2^{-i}$$

$$2^{-i} = -\frac{y}{x}$$

After n iterations, we have

$$x_n = x_0 \qquad (3.66)$$

$$z_n = z_0 + \frac{y_0}{x_0} \qquad (3.66)$$

$$y_n = 0 \qquad (3.67)$$

Rule: choose $d_i \epsilon \{-1, +1\}$ such that $y \to 0$

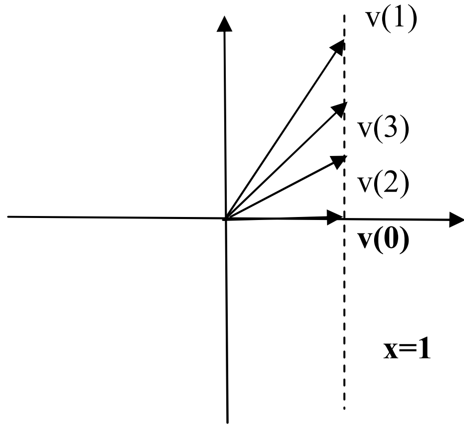


Figure 3.12 Linear Rotation [37]

Hyperbolic function: $m = -1, \phi_i = \tanh^{-1}(2^{-i})$

Put these value in equation (3.51), (3.52), (3.52)

The basic CORDIC iterations are

$$x_{i+1} = x_i + 1.y_i 2^{-i} \qquad (3.68)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \qquad (3.69)$$

$$z_{i+1} = z_i - d_i \tanh^{-1}(2^{-i}) \qquad (3.70)$$

$$d_i = \pm 1 \qquad (d_i \text{ is the direction of angle of rotation})$$

In rotation mode, z converges to zero

$$z = \sum \phi_i \qquad (3.71)$$

The output of the rotation mode x_n, y_n and z_n are given as:

$$\mathrm{x_n = K_n(x_0 \cosh z_0 - y_0 \sinh z_0)} \qquad (3.72)$$

$$\mathrm{y_n = K_n(y_0 \cosh z_0 + x_0 \sinh z_0)} \qquad (3.73)$$

$$\mathrm{z_n = 0} \qquad (3.74)$$

Rule: choose $d_i \epsilon \{-1, +1\}$ such that $z \to 0$

where $K_n = \prod_{i=0}^{n-1} (1 - 2^{-2i})^{1/2} = 0.82816$

We know that $\tanh^{-1}(2^0) = \infty$ $(and\ K_n = 0)$. So begin from iteration i=1

Hyperbolic rotation does not converge with sequence of angles $\tanh^{-1}(2^{-i})$ since

$$\sum_{i=j+1}^{\infty} \tanh^{-1}(2^{-i}) < \tanh^{-1}(2^{-j})$$

A solution is that to repeat some iterations [6]

$$\sum_{j=i+1}^{\infty} \tanh^{-1}(2^{-i}) < \tanh^{-1}(2^{-i}) < \sum_{j=i+1}^{\infty} \tanh^{-1}(2^{-j}) < \tanh^{-1}(2^{-(i3+1)})$$

Repeating iterations 4, 13, 40, … k, 3k+1,…..results in a convergent algorithm.

In vectoring mode, y converges to zero.

$$0 = y + x.2^{-i}$$

$$-y = x.2^{-i}$$

$$2^{-i} = -\frac{y}{x}$$

$$\tan(\sum \phi_i) = -\frac{y}{x} \qquad (3.75)$$

After n iterations, we have

$$x_n = K_n(x^2 - y^2)^{1/2} \qquad (3.76)$$

$$z_n = z + \tanh^{-1}(\frac{y}{x}) \qquad (3.77)$$

$$y_n = 0 \qquad (3.78)$$

Rule: choose $d_i \ \epsilon\ \{-1, +1\}$ such that $y \to 0$

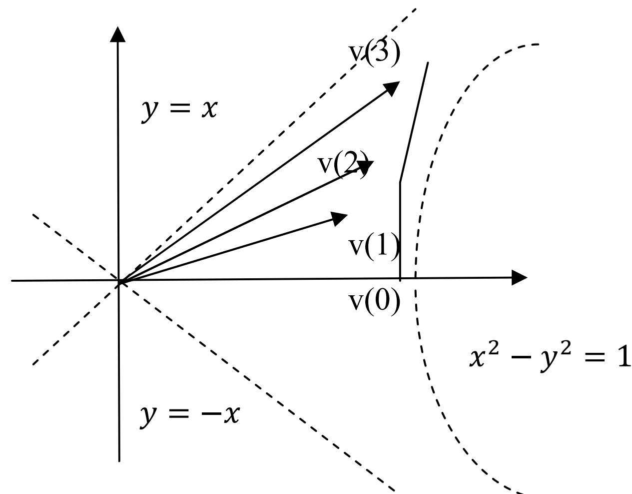


Figure 3.13: Hyperbolic Rotation [37]

Table 3.6 CORDIC mode [3]

	Rotation mode $d_i = \begin{cases} +1, if\ z_i \ \geq 0 \\ -1, if\ z_i \ < 0 \end{cases}$ $z(i) \rightarrow 0$	Vectoring mode $d_i = \begin{cases} +1, if\ y_i \ < 0 \\ -1, if\ y_i \ \geq 0 \end{cases}$ $y(i) \rightarrow 0$
m=1 Circular $\phi_i = \tan^{-1}(2^{-i})$	x → → $K_n(x \cos z - y \sin z)$ y → → $K_n(y \cos z + x \sin z)$ z → → 0 For cos and sin: set $x = \frac{1}{K_n},\ y = 0$	x → → $K_n(x^2 + y^2)^{1/2}$ y → → 0 z → → $z + \tan^{-1}(-\frac{y}{x})$ For tan^{-1}: set x=1, z=0
m=0 Linear $\phi_i = 2^{-i}$	x → → x y → → $y + x.z$ z → → 0 For multiplication: set y=0	x → → x y → → 0 z → → $z + \frac{y}{x}$ For division: set z=0
$m = -1$ Hyperbolic $\phi_i = \tanh^{-1}(2^{-i})$	x → → $K_n(x \cosh z + y \sinh z)$ y → → $K_n(y \cosh z + x \sinh z)$ z → → 0	x → → $K_n(x^2 - y^2)^{1/2}$ y → → 0 z → → $z + \tanh^{-1}(-\frac{y}{x})$

	For cosh and sinh: set $x = \frac{1}{K_n}$, $y = 0$ $\tanh z = \frac{\sinh z}{\cosh z}$, $e^z = \sinh z + \cosh z$	For tan^{-1}: set x=1, z=0

3.4 CORDIC Hardware

The basic CORDIC iterations are

$$x_{i+1} = x_i - d_i y_i 2^{-i} \quad (3.79)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \quad (3.80)$$

$$z_{i+1} = z_i - d_i \phi_i \quad (3.81)$$

$d_i = \pm 1$ (d_i is the direction of angle of rotation)

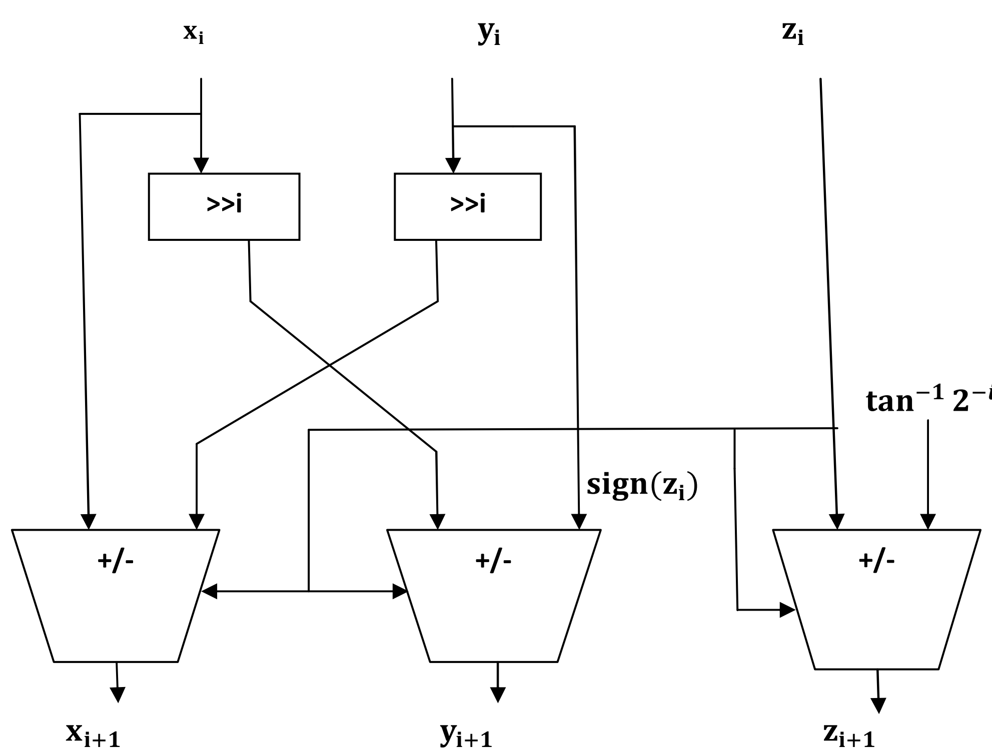


Figure 3.14 Hardware Implementation of a CORDIC Iteration

A hardware implementation for CORDIC arithmetic is shown in figure 3.14. It requires three registers for x, y, z, a look up table to store values of ϕ_i and two shifter to supply the terms $2^{-i}x$ and $2^{-i}y$ to the adder/subtractor units. The d_i factor (-1, +1) is accommodated by selecting the shift operand or its complement. In the extreme, CORDIC iterations can be implemented in firmware (micro program) or software using the ALU

and general purpose registers of a standard microprocessor. In this case, the look up table supplying the term ϕ_i can be stored in the control ROM or in main memory, where high speed is not required and minimizing the hardware cost is important (as in calculator).

3.5 Original CORDIC

The CORDIC algorithm is based upon the idea of rotating a vector in a plane, starting from its initial position, until it coincides with the desired target position. Two operating modes are possible with CORDIC, rotation and vectoring. In the rotation mode the initial position of the vector and the angle through which to rotate it by are known and the final coordinates of the target vector are to be determined. In the vectoring mode, the coordinates of the initial and final (target) vector positions are known, and the angle between the two positions is to be determined.

The rotation of the vector is accomplished using a *fixed* number of angular steps which are executed in sequence. These are known as micro-rotations, although the term angle constants or Arc Tangent Radices (ATR's) are also used interchangeably to refer to them. Each micro-rotation is smaller than the previous one, and the process is carried out until the vector has arrived within an arbitrarily small angular distance of its final resting position. The algebraic sum of the angle constants then approximates the desired rotation angle within a given precision. The number of micro-rotations to be carried out depends upon the level of precision desired in the results – N micro-rotations result in N bits of precision, thus making it a linear convergence algorithm. The angle constants which are used are specially chosen so that they simplify the process of calculating the new coordinates of the vector after every micro-rotation operation.

Original CORDIC—Rotation through 30 Degrees

Consider the rotation of a vector from the x-axis through an angle of, e.g. 30 degrees. Assuming that the rotation is to be accomplished in 16 number of iterations ($N = 0,1,2,3 \ldots 14,15$), the set Q of predetermined angle constants that are used is as follows:

$$\mathrm{Q} = \{45°, 26.565°, 14.036°, 7.125°, 3.576°, 1.7876°, 0.8938°, 0.4469°, 0.2238°, 0.1119, 0.05595°, 0.02798°, 0.01399°, 0.00699°, 0.00349°, 0.00175°\}$$

In the Original CORDIC method, the rotation through 30 degrees is carried out by the following sequence of angular steps or pseudo-rotations, that adds up to approximately 30 degrees, as shown in equation (3.82):

$$30° \approx (45° - 26.564° + 14.036° - 7.125° + 3.576° + 1.7876° - 0.8938° + 0.4469° - 0.2238° - 0.1119° + 0.05595° + 0.02798° - 0.01399° + 0.00699° - 0.00349° - 0.00175°) \quad (3.82)$$

Even though the algorithm as a whole converges, individual intermediate pseudo-rotations may diverge, and must be corrected by succeeding pseudo-rotations. For example, the iteration from $i = 2 \; to \; i = 3$ is a divergent pseudo-rotation.

In the vectoring mode, the vector V is rotated towards the x-axis so that the y-component approaches zero. The sum of all angles of micro rotations (output angle) is equal to the angle of rotation of vector V, while output corresponds to its magnitude. In this operating mode, the decision about the direction of the micro rotation depends on the sign of y_i, if it is positive then $d_i = -1$ and $d_i = 1$ otherwise. If the angle accumulator is initialized with zero, it will contain the transverse angle at the end of the iteration.

3.6 Control CORDIC

The Control CORDIC method [11] was one method that took advantage of the availability of ROM space to implement a technique which reduced the number of iterations, but required a ROM to store the different scaling factors. The technique was based upon the observation that in the Original CORDIC algorithm, the iteration variable z_i does not always converge monotonically to 0 – some of the iterations may actually result in divergent micro-rotations, which do nothing to improve the convergence towards the target vector. As shown above in section 3.5, these divergent micro-rotations (seen at i = 3) in the Original CORDIC algorithm, as the accumulated angle approaches a target angle of 30°.

The angle trajectory looks very similar to the classic under-damped response of a second order control system, with no overshoot. The Control CORDIC method modifies the angle trajectory so that it now resembles a critically damped system, with no overshoot, resulting in faster convergence. Since divergent rotations can only occur when there is an overshoot, this method eliminates divergent micro-rotations by completely eliminating the overshoot itself. Unfortunately this also means that convergent micro-rotations that overshoot the target are eliminated at the same time. The angle trajectory is modified by imposing a limit on the rotation directions.

For positive angles the rotation direction (σ_i) is restricted to $\{-1,0\}$ as given below:

$$\sigma_i = \begin{cases} +1 \text{ when } z \geq \alpha_i \\ 0 \text{ otherwise} \end{cases}$$

Similarly for negative angles the rotation direction is restricted to $\{-1,0\}$. This simple angle selection function thus prevents any overshoot of the target position by the moving vector.

The advantage of this method is that its angle selection function is quite simple, and is easy to implement with only a minimal effect on the cycle time, thus allowing its use in dynamic situations where the angle of rotation can take any value. However in return for the reduction in iteration count, this method requires the use of a ROM to store the different scaling factors.

Control CORDIC—Rotation through 30 Degrees

Consider the rotation of a vector from the x-axis through an angle of, e.g. 30 degrees. Assuming that the rotation is to be accomplished in 16 number of iterations $(N = 0,1,2,3 \ldots\ldots\ldots\ldots 14,15)$, the set Q of predetermined angle constants that are used is as follows

$$Q = \{45°, 26.565°, 14.036°, 7.125°, 3.576°, 1.7876°, 0.8938°, 0.4469°, 0.2238°, 0.1119, 0.05595°, 0.02798°, 0.01399°, 0.00699°, 0.00349°, 0.00175°\}$$

In the Control CORDIC method, the rotation through 30 degrees is carried out by the following sequence of angular steps or pseudo-rotations, that add up to approximately 30 degrees, as shown in equation (3.83):

$$30° \approx (0 + 26.564° + 0 + 0 + 0 + 1.7876° + 0.8938° + 0.4469° + 0.2238° + 0.05595° + 0 + 0.01399° + 0.00699° + 0.00349° + 0.00175°) \quad (3.83)$$

As we seen in equation (3.83), in case of Control CORDIC, divergent rotations are completely eliminated by eliminating overshoot.

Table 3.7 shows the comparison of Original CORDIC and Control CORDIC angle selection for 16 numbers of iterations and figure 3.15 shows the comparison of number of iterations in Original CORDIC and Control CORDIC for 30 degree angle.

Table 3.7 Comparison of Original CORDIC and Control CORDIC angle selection

i	Original CORDIC	Control CORDIC
0	0°	0°
1	45°	0°
2	18.435°	26.565°
3	32.471°	26.565°
4	25.346°	26.565°
5	28.922°	26.565°
6	30.7096°	28.355°
7	29.8159°	29.2488°
8	30.2627°	29.6957°
9	30.0389°	29.9195°
10	29.927°	29.9195°
11	29.982°	29.975°
12	30.010°	29.975°
13	29.996°	29.989°
14	30.003°	29.996°
15	30.000°	29.999°

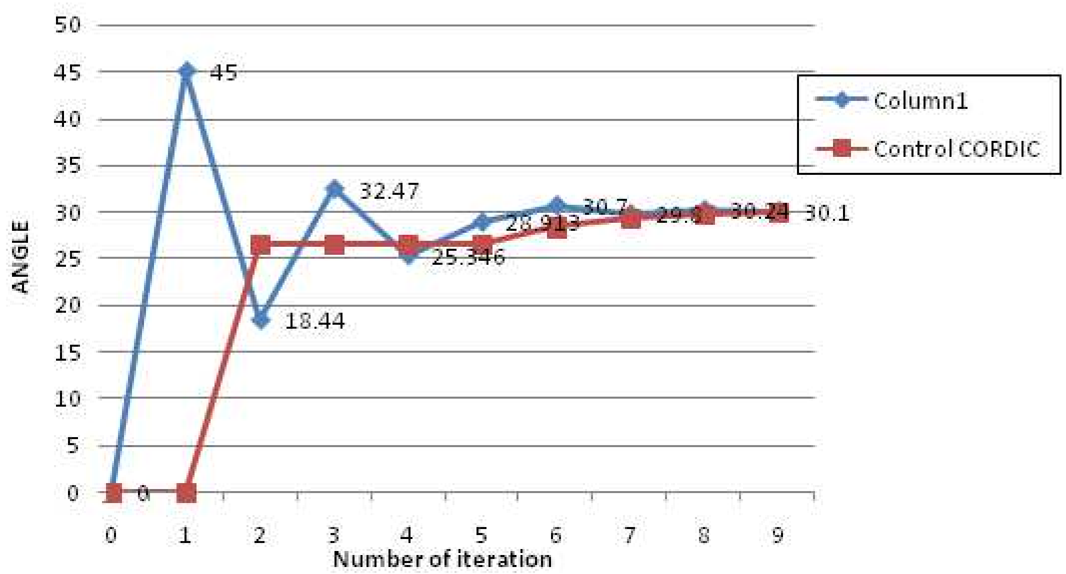


Figure 3.15 Rotation mode graph of Control CORDIC and Original CORDIC

3.7 Angle Recoding CORDIC

The Angle Recoding (AR) technique is very suitable for applications that use CORDIC algorithm in only forward rotation mode (also known as vector rotation mode) [12]. Basically, the superior performance (improved angle precision and reduced iteration number) of the AR technique comes from the relaxation on the form of rotation sequence in the CORDIC algorithm. It encodes the desired rotation angle as a linear combination of very few elementary rotation angles. Each of these elementary rotation angles takes one CORDIC iteration to compute. The fewer the number of elementary rotation angles, the fewer the number of iterations are required. Angle Recoding uses a greedy algorithm to skip over some rotation angles, and can reduce the number of iterations required. The maximum number of iterations required by this method is N/2, with an average value of approximately N/3 iterations. It is studied that this algorithm is able to reduce the total number of required elementary rotation angles by at least 50% without affecting the computational accuracy.

The AR algorithm proposed by Hu and Naganathan [12] is essentially a software algorithm, and it must be modified to get it into a form that can be implemented in hardware, so as to make it dynamic and capable of handling any rotation angle. This is done by replacing the serial testing of the angle constants in the original algorithm by a parallel test to be carried out in hardware in case of Adaptive CORDIC [29]. It was done by completely eliminate the angle selection step from every iteration altogether. Instead of using the current residual angle to determine the angle constant for that iteration, it would be much more efficient if all the angle constants could be identified in a single step, using only the initial rotation angle as input.

In case of Angle Recoding, instead of choosing all the angle constants stored in ROM, we make use of a technique in which residual angle z_i is compared with all the angle constants α_i stored in ROM and the angle constant with minimum difference $(z_i - \sigma_i \alpha_i)$ is chosen. The corresponding angle constant with minimum difference is chosen and the index i related to that angle constant is used for the shift operations. After shifting operation, the process of calculating the new X and Y coordinates of the vector is done. The extra logic needed for the calculation of angle constant with minimum difference will increase the cycle time. Due which the reduction in iteration count will not affect the latency of CORDIC algorithm and even we get more latency for large value of N.

Let θ be the desired angle of rotation

$$\alpha_{min} = \alpha_N$$

$$Z = \theta$$

while $(abs(Z) > \alpha_{min}/2)$

$\{ \quad \sigma = (Z \geq 0)?(+1):(-1);$

$$\alpha_{min} = \alpha_0$$

foreach α_i $(\alpha_0, \alpha_1 \dots \dots \dots .. \alpha_N)$

$\{ \quad if\ (|abs(Z) - \alpha_i| < |abs(Z) - \alpha_{max}|)$

then $\alpha_{max} = \alpha_i \quad \}$

Store α_{max} on adaptive_angle_list

$Z = Z - \sigma * \alpha_{max} \quad \}$

Figure 3.16 Angle selection in the Angle Recoding method [29]

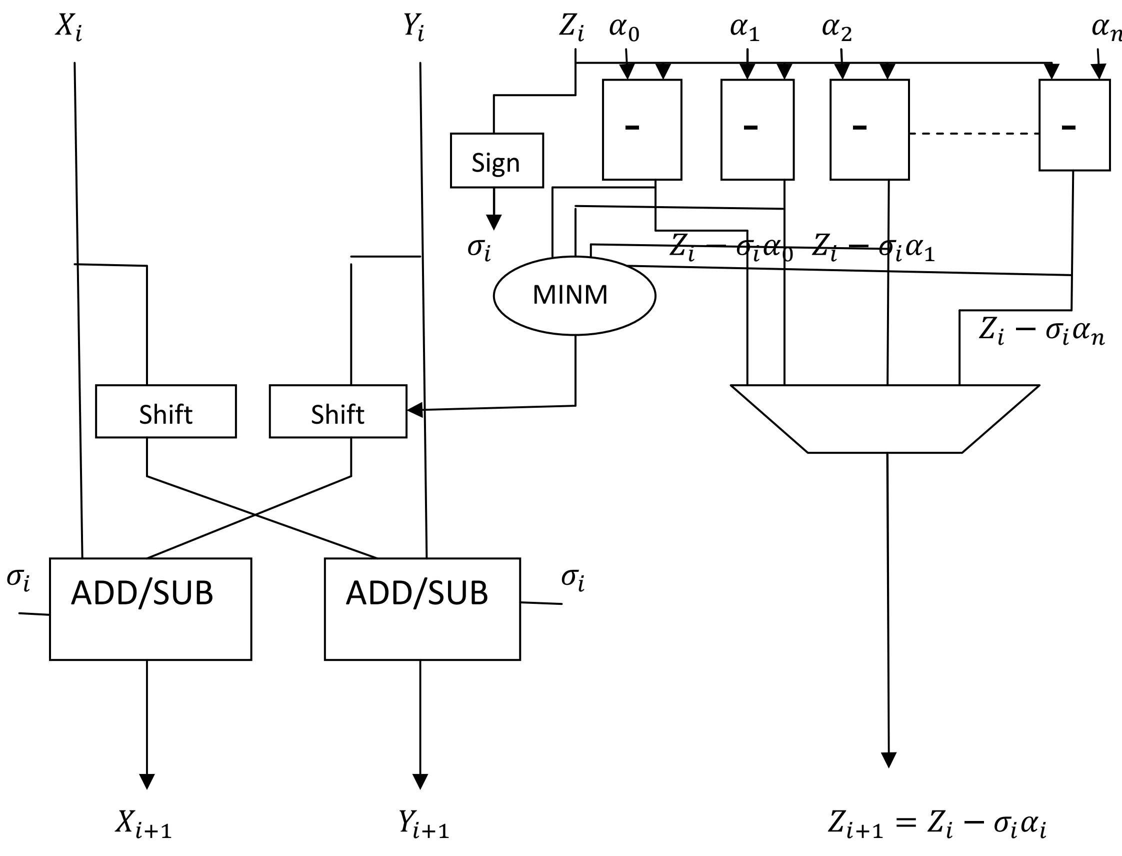


Figure 3.17 Hardware implementation of Angle Recoding CORDIC [29]

Angle Recoding CORDIC—Rotation through 30 Degrees

Consider the rotation of a vector from the x-axis through an angle of 30 degrees. Assuming that the rotation is to be accomplished for 16 number of bits ($N = 0,1,2,3 \ldots 14,15$), the set Q of predetermined angle constants that are used is as follows:

$$Q = \{45°, 26.565°, 14.036°, 7.125°, 3.576°, 1.7876°, 0.8938°, 0.4469°, 0.2238°, 0.1119, 0.05595°, 0.02798°, 0.01399°, 0.00699°, 0.00349°, 0.00175°\}$$

In AR CORDIC, the rotation through 30 degrees is carried out by the following sequence of angular steps or pseudo-rotations, that add up to approximately 30 degrees, as shown in equation (3.84):

$$30 \approx (26.565 + 3.576 - 0.1119 - 0.02798 - 0.00175) \quad (3.84)$$

So we see in equation (3.84) that in case of AR CORDIC, we require only 5 iterations instead of 16 iterations for 16 bit data. So, number of iteration count is reduced and speed get increase.

Figure 3.18 shows the rotation graph of Angle recoding method for rotation angle 30°. The graph also shows the comparison between Original CORDIC and Angle Recoding CORDIC.

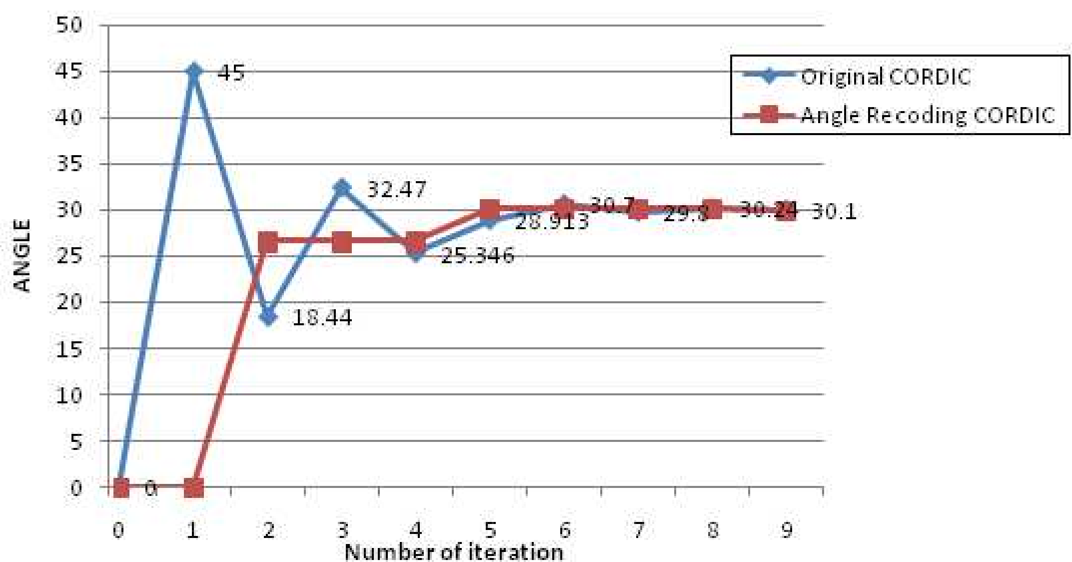


Figure 3.18 Rotation mode graph of Angle Recoding CORDIC and Original CORDIC

3.8 Applications of CORDIC algorithm

The CORDIC method has achieved widespread acceptance and has found use in a number of disparate applications and a short summary of some example applications will be detailed here.

3.8.1 Robotics

In Robotics, if an end-effector is required to be positioned at a particular point in space, with a certain orientation, the angles of the joints that can accomplish this is found by solving the inverse-kinematic equation set for that robot [10]. The solution of these equations requires the use of modules which can perform trigonometric and hyperbolic operations as well as their inverse, additions, multiplication, division and square root, all of which map very well to a systolic array of CORDIC processing elements. As such there are several different sets of joint angles which will satisfy the inverse kinematic equations and the host processor can choose the set which is closest to the desired trajectory path [38].

3.8.2 3-D Computer Graphics

3-D computer graphics [39] is used to render detailed views of mechanical components from different orientations, and is extensively used in the automobile and aircraft manufacturing industries. The computer gaming industry as well as the movie industry also used 3-D techniques to provide a realistic rendering of animated objects. The specialized graphics processors which are used in these applications employ CORDIC processing elements within them to perform the diverse set of mathematical operations on the input data streams. They perform vector interpolation in 3-D shading algorithms and are also used in lighting. Operations such as rotating an object or moving it, or viewing it from a different angle are all performed by applying a transformation of a coordinate system to each of the vertices of the body. The transformation matrix used is composed of trigonometric ratios which are evaluated using CORDIC elements.

3.8.3 OFDM

CORDIC elements have even found use in modern technology such as OFDM (Orthogonal Frequency Division Multiplexing) which is used in wireless radio protocols like IEEE 802.11a and 802.11g to transmit large amounts of digital data up to 54 Mbps. OFDM technology provides a reduction in interference, distortion and multi-path delay distortion. However OFDM systems are susceptible to inter-carrier interference arising

from frequency mismatch between the transmitter and receiver. In such cases, the IEEE 820.11a standard transmits a preamble at the beginning of each information packet, which allows the frequency offset to be determined. The frequency offset is compensated by using a CORDIC module to perform a rotation of each incoming data sample by an amount related to the frequency offset [39].

3.8.4 DCT Compression

The Discrete Cosine Transform (DCT) [40] is a widely used block-coding technique for digital data compression. The compressed data requires smaller storage space, and also consumes less bandwidth during transmission. The 1D-DC has been used in the Dolby AC-2 and AC-3 standards while the 2D-DCT is used in the JPEG standard for image compression, as well as MPEG-1 and MPEG-2 standards for video compression. It is also used in the H.261 and H.263 standards to provide moving image compression, as well as in the MP-3 codec for audio compression. The calculation of the DCT/IDCT is performed in a hardware block using multiplier elements. It has been shown that the butterfly operation is equivalent to a CORDIC rotation. If the calculation of the DCT is done using CORDIC processing elements instead of multipliers, the computational complexity is reduced, since the multiplies and adds are replaced by adds and shift operations. This reduces the power consumption as well as the area required for the block.

3.9 Number Format

A computer number format is the internal representation of numeric values in digital computer and calculator hardware and software. Normally, numeric values are stored as groupings of bits, named for the number of bits that compose them. Because of the use of transistor-transistor logic, computers represent data with binary numbers. This data is composed of bits, which in turn are grouped into larger sets such as bytes. A bit is a binary digit that represents one of two states. The concept of a bit can be understood as a value of either 1 or 0, on or off, yes or no, true or false, or encoded by a switch or toggle of some kind. While a single bit, on its own, is able to represent only two values, a string of bits may be used to represent larger values. As the number of bits composing a string increases, the number of possible 0 and 1combinations increases exponentially. While a single bit allows only two value-combinations and two bits combined can make four separate values and so on. In real life, we deal with real numbers that is numbers with fractional part. In most modern computer we have have

hardware support for fixed point numbers and floating point numbers for representing real numbers.

3.9.1 Fixed Point Numbers

A fixed-point number representation has a fixed number of digits after (and sometimes also before) the radix point. A fixed point representation of values allows the user to select a different decimal point location depending on the dynamic range and precision required.

3.9.1.1 Unsigned Fixed Point Numbers

Rational numbers are represented using the form U (a,b), where a is the number of bits used for integers and b is the number of bits used to represent fractions. A 16-bit unsigned fixed number is represented in figure 3.19.

10 bits for integer part   6 bits for fractional part

U (10,6) =   b b b b b b b b b b   b b b b b b

Figure 3.19 16-bit unsigned Fixed Point Number

This format allows a range of representation from 0 to $(2^a - 2^{-b})$ and the value of U(a,b) can be obtained from:

$$x = (1/2^b) \sum_{n=0}^{N-1} 2^n X_n$$

3.9.1.2 Signed Two's Complement Fixed Point Numbers

The same format U (a, b) as in section (3.9.1.1) is used to represent signed rational numbers by adding two important characteristics:

- The most significant bit is referred to as the sign bit with a value of zero for positive numbers and one for negative numbers.
- The N bit word is represented in its two's complement form.

A 16-bit signed two's complement Fixed Point number is shown in figure 3.20

1 bit for sign 10 bit for integer   5-bits for fractional part

U(10,5) = b b b b b b b b b b b b b b b b

Figure 3.20 16-bit signed two's complement Fixed Point number

The range for signed two´s complement number is:

$$-2^{N-1-b} \leq x \leq 2^{N-1-b-1}/2^{b}$$

The rational number represented by U(a, b) is given by:

$$x = (1/2^{b})\left[-2^{N-1}X^{N-1} + \sum_{n=0}^{N-2} 2^{n}X_{n}\right]$$

Finite Precision Concepts

- Precision: Number of non-zero bits. Ex. Signed A(13,2) has a precision of 16 bits.
- Resolution: Smallest non-zero magnitude that can be represented.
  Ex. Signed A(10,5) has a resolution of 1/25 = 0.03125
- Range: Difference between maximum positive and negative values
  Ex. Signed A(13,2) has a range of 8191.75 to 8192 = 16,383.75
- Accuracy: Maximum difference between represented and real value
  Accuracy = Resolution/2. Ex. Signed A(13,2) has accuracy of 1/8
- Word length: Minimum number of bits required to represent a number.
  Unsigned U(a,b) = a + b
  Signed U(a,b) = a + b + 1

Fixed Point Math

Notation: For an N bit DSP, the fixed-point format for numbers is usually referred as Q.b, where b is obtained from U(a,b). For example, for a 16-bit processor, a signed A(2,13) number would be referred to as Q.13

Addition

Fixed-point numbers have to be scaled before being added so that both operands have the same word length. The result for the addition operation will have the same word length as both operands.

Q.b + Q.b = Q.b

Multiplication

- The result for the addition operation will have the added wordlength of both operands.

  Q.b X Q.c = Q.(b+c)

- Duplicated sign bit: The result of the multiplication of 2 signed fixed point rationals, will have two sign bits in the first 2 MSBs. One of these sign bits has to be eliminated by shifting. Some DSPs have modes of operation that perform this shift automatically so be sure to turn this mode on or off depending on the operation being performed.
- Since a 16 bit X 16 bit multiplication will yield a 32-bit word, the result must be shifted by 16 bits in order to consider only the highest 16-bit word.

Shifting

- Logic Shift: The word including the decimal point is shifted. Empty bits are padded with zeroes.

  x(a,b) >> n = x(a+n, b-n)

  x(a,b) << n = x(a-n, b+n)

- Arithmetic Shift: The decimal point is shifted to the opposite direction of the shift.

  x(a,b) >> n = x(a-n, b+n)

  x(a,b) << n = x(a+n, b-n)

In this thesis, 16-bit, 24-bit and 32-bit signed fixed point number has been used for the computation sine/cosine functions. The MSB represents the sign bit and the remaining bits are divided into integer and fractional part.

Table 3.8 Signed fixed point representation

Total number of bits	16-bit	24-bit	32-bit
Sign bit	1 bit	1 bit	1 bit
Number of integer bits	8 bits	8 bits	8 bits
Number of fractional bits	7 bits	15 bits	23 bits

3.9.2 Floating Point Numbers

The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point, that is, the decimal point can float. In general, floating point representations are slower and less accurate than fixed-point representations, but

they can handle a larger range of numbers. Floating Point Numbers are numbers that can contain a fractional part. For e.g. following numbers are the floating point numbers: 3.0, -111.5, ½, 3E-5 etc. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow.

The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values. For example, a fixed point representation that has seven decimal digits, with the decimal point assumed to be positioned after the fifth digit, can represent the numbers 12345.67, 8765.43, 123.00, and so on, whereas a floating-point representation (such as the IEEE 754 decimal32 format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, 1234567000000000, and so on. The floating-point format needs slightly more storage (to encode the position of the radix point), so when stored in the same space, floating-point numbers achieve their greater range at the expense of slightly less precision. Over the years, several different floating-point representations have been used in computers; however, for the last ten years the most commonly encountered representation is that defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754). It is a technical standard established by the Institute of Electrical and Electronics Engineers (IEEE) and the most widely used standard for floating-point computation, followed by many hardware and software implementations. Single precision representation occupies 32 bits: a sign bit, 8 bits for exponent and 23 for the mantissa. Floating-point representation, in particular the standard IEEE format, is by far the most common way of representing an approximation to real numbers in computers because it is efficiently handled in most large computer processors.

3.9.2.1 Normalization

Before a floating-point binary number can be stored correctly, its mantissa must be normalized. The process is basically the same as when normalizing a floating-point decimal number.

- **Sign**

The sign of a binary floating-point number is represented by a single bit. A 1 bit indicates a negative number, and a 0 bit indicates a positive number.

- **Mantissa**

Using -3.154 x 10^5 as an example, the **sign** is negative, the **mantissa** is 3.154, and the **exponent** is 5. The fractional portion of the mantissa is the sum of each digit multiplied by a power of 10:

.154 = 1/10 + 5/100 + 4/1000

In the number +11.1011 x 2^3**,** the sign is positive, the mantissa is 11.1011, and the exponent is 3. The fractional portion of the mantissa is the sum of successive powers of 2. In our example, it is expressed as: .1011 = 1/2 + 0/4 + 1/8 + 1/16

- **Exponent**

IEEE Short Real exponents are stored as 8-bit unsigned integers with a bias of 127. Let's use the number 1.101 x 2^5 as an example. The exponent (5) is added to 127 and the sum (132) is binary 10100010. Here are some examples of exponents, first shown in decimal, then adjusted, and finally in unsigned binary.

The binary exponent is unsigned, and therefore cannot be negative. The largest possible exponent is 128 when added to 127, it produces 255, the largest unsigned value represented by 8 bits. The approximate range is from 1.0 x 2^{-127} to 1.0 x 2^{+128}. Table 3.9 shows the adjustment of exponent by adding the bias. For example, exponent of -10 is added with a bias of 127 to give the adjusted exponent 117.

Table 3.9 Adjustments of exponents

Exponent (E)	**Adjusted (E + 127)**	**Binary**
+5	132	10000100
0	127	01111111
-10	117	01110101
+128	255	11111111
-127	0	00000000

Before a floating-point binary number can be stored correctly, its mantissa must be normalized. The process is basically the same as when normalizing a floating-point decimal number. For example, decimal 1234.567 is normalized as 1.234567 x 10^3 by

moving the decimal point so that only one digit appears before the decimal. The exponent expresses the number of positions the decimal point was moved left (positive exponent) or moved right (negative exponent).Similarly, the floating-point binary value 1101.101 is normalized as 1.101101 x 2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3. Table 3.10 shows some examples of normalizations.

Table 3.10 Normalization

Binary Value	**Normalized As**	**Exponent**
1101.101	1.101101	3
.00101	1.01	-3
1.0001	1.0001	0
10000011.0	1.0000011	7

In a normalized mantissa, the digit 1 always appears to the left of the decimal point. In fact, the leading 1 is omitted from the mantissa in the IEEE storage format because it is redundant.

Sign, exponent, and normalized mantissa are combined into the binary IEEE short real representation. The value 1.101 x 2^0 is stored as sign = 0 (positive), mantissa = 101, and exponent = 01111111 (the exponent value is added to 127). The "1" to the left of the decimal point is dropped from mantissa. Some more examples are shown in table 3.11.

Table 3.11 Examples of Floating Point Numbers

Binary Value	**Biased Exponent**	**Sign, Exponent, Mantissa**
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

3.9.2.2 IEEE 754 Standard For Binary Floating-Point Arithmetic

The IEEE (Institute of Electrical and Electronics Engineers) has produced a Standard to define floating-point representation and arithmetic. The standard brought out by the IEEE come to be known as IEEE 754. The IEEE 754 Standard for Floating-Point Arithmetic is the most widely-used standard for floating-point computation, and is followed by many hardware (CPU and FPU) and software implementations [41]. Many computer languages allow or require that some or all arithmetic be carried out using IEEE 754 formats and operations.

The standard specifies :

- Basic and extended floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversions between integer and floating-point formats
- Conversions between different floating-point formats
- Conversions between basic format floating-point numbers and decimal strings
- Floating-point exceptions and their handling, including non numbers

3.9.2.3 Formats

The standard defines five basic formats, named using their base and the number of bits used to encode them. There are three binary floating-point formats (which can be encoded using 32, 64, or 128 bits) and two decimal floating-point formats (which can be encoded using 64 or 128 bits). The first two binary formats are the 'Single Precision' and 'Double Precision' formats of IEEE 754-1985, and the third is often called 'quad'; the decimal formats are similarly often called 'double' and 'quad'.

Table 3.12 Various basic formats of IEEE 754 standard

parameter → **format name**	***B*** **Base**	***P*** **(bits or digits)**	***Emax***
Binary32	2	23+1 bits	+127
Binary64	2	52+1 bits	+1023
Binary128	2	112+1 bits	+16383
Decimal64	10	16 digits	+384
Decimal128	10	34 digits	+6144

a) Single Precision

The most significant bit starts from the left. The three basic components are the sign, exponent, and mantissa. The storage layout for single-precision is shown in figure 3.21.

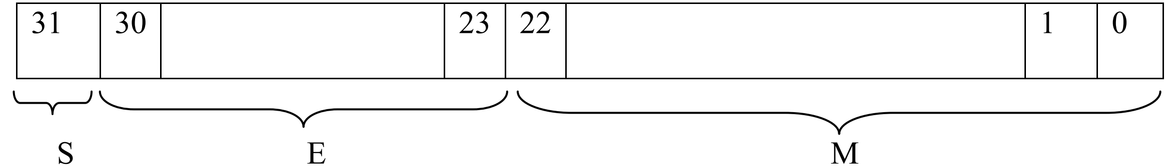


Figure 3.21 Single Precision Format for Floating Point Numbers [42]

The number represented by the single-precision format is:

Value = $(-1)^s\, 2^{E-127} * 1.M$(normalized) when E>0

Where $M = m_{22}\, 2^{-1} + m_{21}\, 2^{-2} + m_{20}\, 2^{-3} + \ldots + m_1\, 2^{-22} + m_0\, 2^{-23}$;

Bias = 127.

s= sign (0 is positive, 1 is negative)

E =biased exponent; E_{max} = 255; E_{min}= 0. E=255 and E=0 are used to represent special values.

e =unbiased exponent; e = E-127(bias)

A bias of 127 is added to the actual exponent to make negative exponents possible without using a sign bit. So for example if the value 100 is stored in the exponent placeholder, the exponent is actually -27(100 – 127). Not the whole range of E is used to represent numbers. The leading fraction bit before the decimal point is actually implicit (not given) and can be 1 or 0 depending on the exponent and therefore saving one bit.

a) Double Precision

The double precision format helps overcome the problems of single precision floating point. Using twice the space, the double precision format has an 11-bit excess-1023 exponent and a 53 bit mantissa plus a sign bit. Double precision floating point values take the form shown in figure 3.22.

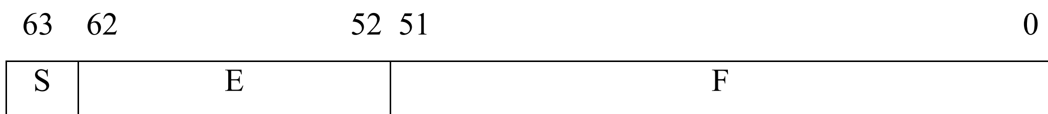


Figure 3.22 Bit Double Precision Floating Point Format

In order to ensure accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The

extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa; four of the additional bits are appended to the end of the exponent. Although the 32 bit ("single") and 64 bit ("double") formats are by far the most common, the standard actually allows for many different precision levels. Computer hardware (for example, the Intel Pentium series and the Motorola 68000 series) often provides an 80 bit extended precision format, with a 15 bit exponent, a 64 bit significand, and no hidden bit.

3.9.2.4 Exceptions

The IEEE standard defines five types of exceptions that should be signalled through a one bit status flag when encountered.

a. Invalid Operation

Some arithmetic operations are invalid, such as a division by zero or square root of a negative number. The result of an invalid operation shall be a NaN (Not a number). There are two types of NaN, quiet NaN (QNaN) and signaling NaN (SNaN). They have the following format, where s is the sign bit:

QNaN = s 11111111 10000000000000000000000

SNaN = s 11111111 00000000000000000000001

Table 3.13 Representation of Single Precision floating point numbers

Sign(s)	Exponent(e)	Fraction	Value
0	00000000	00000000000000000000000	+0 (positive zero)
1	00000000	00000000000000000000000	-0 (negative zero)
1	00000000	10000000000000000000000	-2^{0-127}x0.(2^{-1})= -2^{0-127} x 0.5
0	00000000	00000000000000000000001	$+2^{0-127}$x0.(2^{-23}) (smallest value)
0	00000001	01000000000000000000000	$+2^{1-127}$x1.(2^{-2})= $+2^{1-127}$ x 1.25
0	10000001	00000000000000000000000	$+2^{129-127}$ x1.0= 4
0	11111111	00000000000000000000000	+ infinity
1	11111111	00000000000000000000000	- infinity
0	11111111	10000000000000000000000	Not a Number (NaN)
1	11111111	10000100010000000001100	Not a Number(NaN)

The result of every invalid operation shall be a NaN string with a QNaN or SNaN exception. The SNaN string can never be the result of any operation, only the SNaN exception can be signalled and this happens whenever one of the input operand is a SNaN string otherwise the QNaN exception will be signalled. The SNaN exception can for example be used to signal operations with uninitialized operands, if we set the uninitialized operands to SNaN. Under default exception handling, any operation signaling an invalid operation exception and for which a floating-point result is to be delivered shall deliver a quiet NaN. Signaling NaNs shall be reserved operands that, under default exception handling, signal the invalid operation exception for every general-computational and signaling-computational operation.

The following are some arithmetic operations which are invalid operations and that give as a result a QNaN string and that signal a QNaN exception:

- Any operation on a NaN
- Addition or subtraction: $\infty + (-\infty)$
- Multiplication: $\pm 0 \times \pm \infty$
- Division: $\pm 0 / \pm 0$ or $\pm \infty / \pm \infty$
- Square root: if the operand is less than zero

b. Division by Zero

In mathematics, a division is called a division by zero if the divisor is zero. Such a division can be formally expressed as a/0 where a is the dividend. Whether this expression can be assigned a well-defined value depends upon the mathematical setting. In ordinary (real number) arithmetic, the expression has no meaning. In computer programming, integer division by zero may cause a program to terminate or, as in the case of floating point numbers, may result in a special not-a-number value. The division of any number by zero other than zero itself gives infinity as a result. The addition or multiplication of two numbers may also give infinity as a result. So to differentiate between the two cases, a divide-by-zero exception was implemented.

c. Underflow/ Overflow

Two events cause the underflow exception to be signalled, tininess and loss of accuracy. Tininess is detected after or before rounding when a result lies between ±2Emin. Loss of accuracy is detected when the result is simply inexact or only when a renormalizations

loss occurs. The implementer has the choice to choose how these events are detected. They should be the same for all operations. The implemented FPU core signals an underflow exception whenever tininess is detected after rounding and at the same time the result is inexact.

The overflow exception is signalled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signalled when one of the operands is infinity, because infinity arithmetic is always exact. Division by zero also doesn't trigger this exception.

- **Underflow/Overflow detection:**

Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one. An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition may be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it's an underflow that can never be compensated; if the intermediate exponent $= 0$ then it's an underflow that may be compensated during normalization by adding 1 to it.

When an overflow occurs an overflow flag signal goes high and the result turns to ±Infinity (sign determined according to the sign of the floating point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs). Denormalized numbers are signaled to Zero with the appropriate sign calculated from the inputs and an underflow flag is raised. Assume that E1 and E2 are the exponents of the two numbers A and B respectively; the result's exponent is calculated by

Eresult = E1 + E2 - 127

E1 and E2 can have the values from 1 to 254; resulting in Eresult having values from -125 (2-127) to 381 (508-127); but for normalized numbers, Eresult can only have the values from 1 to 254. Table 3.14 summarizes the Eresult different values and the effect of normalization on it [41].

Table 3.14 Normalization effect on result's exponent and overflow/underflow detection

Eresult	Category	Comments
$-125 \leq$ Eresult < 0	Underflow	Can't be compensated during Normalization
Eresult = 0	Zero	May turn to normalized number during normalization (by adding 1 to it)
$1 <$ Eresult < 254	Normalized Number	May result in overflow during Normalization
$255 \leq$ Eresult	Overflow	Can't be compensated

In this thesis, 24-bit, 28-bit and 32-bit floating point numbers have been used in which MSB represents the sign bit and the remaining bits are divided into exponent and mantissa part as shown in table 3.15.

Table 3.15 Number of sign bit, exponent bit and mantissa bit in 24-bit and 28-bit floating point number

	Sign bit	Exponent bits	Mantissa bits
24-bit Floating Point Number	1 bit	6 bits	17 bits
28-bit Floating Point Number	1 bit	7 bits	20 bits

CHAPTER 4

FIELD PROGRAMMABLE DESIGN FLOW

This chapter introduces about the FPGA concepts and FPGA Synthesis Flow. An FPGA is a device that consists of thousands or even millions of transistors connected to perform logic functions. They perform functions from simple addition and subtraction to complex digital filtering and error detection and correction.

4.1 Introduction to FPGA

A field programmable gate array (FPGA) is a semiconductor device that can be configured by the customer or the designer after manufacturing hence the name "field-programmable". Field Programmable gate arrays (FPGAs) are truly revolutionary devices that blend the benefits of both hardware and software. FPGAs are programmed using a logic circuit diagram or a source code in Hardware Description Language (HDL) to specify how the chip will work. They can be used to implement any logical function that an Application Specific Integrated Circuit (ASIC) could perform but the ability to update the functionality after shipping offers advantages for many applications. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" somewhat like a one chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions or merely simple logic gates like AND and XOR. In most FPGAs, the logic block also includes memory elements, which may be simple flip flops or more complete blocks of memory.

FPGAs blend the benefits of both hardware and software. They implement circuits just like hardware performing huge power, area and performance benefits over softwares, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Just like computer hardware, FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor-based designs. However unlike in ASICs, these computations are programmed into a chip, not permanently frozen by the manufacturing process. This means that an FPGA based system can be programmed and reprogrammed many times. FPGAs are being incorporated as central processing elements in many applications such as consumer

electronics, automotive, image/video processing military/aerospace, base stations, networking/communications, super computing and wireless applications.

4.2 FPGA for Fixed Point and Floating Point Computations

Fixed point numbers are small in size as compared to floating point number and therefore fixed point computations are suitable for FPGA. With gate counts approaching ten million gates, FPGA's are quickly becoming suitable for major floating point computations also. However, to date, few comprehensive tools that allow for floating point unit trade offs have been developed. Most commercial and academic floating point libraries provide only a small number of floating point modules with fixed parameters of bit-width, area and speed [43]. Due to these limitations, user designs must be modified to accommodate the available units. The balance between FPGA floating point unit resources and performance is influenced by subtle context and design requirements. Generally, implementation requirements are characterized by throughput, latency and area.

- FPGAs are often used in place of software to take advantage of inherent parallelism and specialization. For data intensive applications, data throughput is critical.
- If floating point computation is in a dependent loop, computation latency could be an overall performance bottleneck.

4.3 FPGA Technology Trends

- General trend is bigger and faster.
- This is being achieved by increases in device density through even smaller fabrication process technology.
- New generations of FPGAs are geared towards implementing entire systems on a single device.
- Features such as RAM, dedicated arithmetic hardware, clock management and transceivers are available in addition to the main programmable logic.
- FPGAs are also available with the embedded processors (embedded in silicon or as cores within the programmable logic fabric).

4.4 FPGA Implementation

The FPGA that is used for the implementation of the circuit is the Xilinx Virtex 5 (Family) and Xilinx Virtex 6 (Family). The working environment/tool for the design is

the Xilinx ISE 13.1i is used for FPGA Design flow of VHDL code.

As the FPGA architecture evolves and its complexity increases. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit stream to program FPGA chips. A typical FPGA design flow includes the steps and components shown in Fig. 1. Inputs to the design flow typically include the HDL specification of the design, design constraints, and specification of target FPGA devices . We further elaborate on these components of the design input in the following: Design constraints typically include the expected operating frequencies of different clocks, the delay bounds of the signal path delays from input pads to output pads (I/O delay), from the input pads to registers (setup time), and from registers to output pads (clock-to-output delay). In some cases, delays between some specific pairs of registers may be constrained.

The second design input component is the choice of FPGA device. Each FPGA vendor typically provides a wide range of FPGA devices, with different performance, cost, and power tradeoffs. The selection of target device may be an iterative process. The designer may start with a small (low capacity) device with a nominal speed-grade. But, if synthesis effort fails to map the design into the target device, the designer has to upgrade to a high-capacity device. Similarly, if the synthesis result fails to meet the operating frequency, he has to upgrade to a device with higher speed-grade. In both the cases, the cost of the FPGA device will increase in some cases by 50% or even by 100%. This clearly underscores the need to have better synthesis tools since their quality directly impacts the performance and cost of FPGA designs [44].

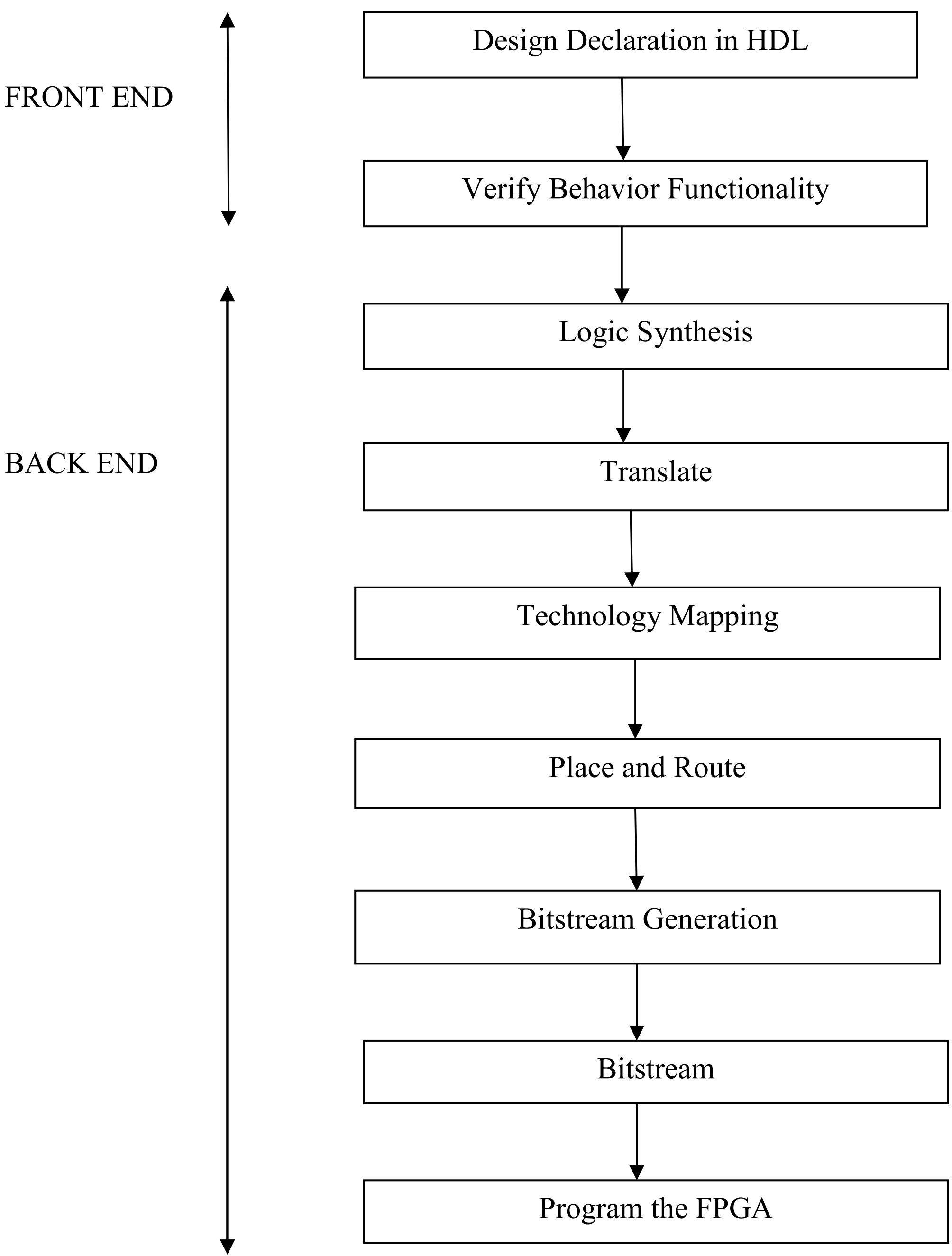


Figure 4.1 FPGA Design Flow

Design Entity

The basic architecture of the system is designed in this step which is coded in a Hardware description Language like Verilog or VHDL. A design module is split into two parts, each of which is called a design unit in VHDL. The module declaration represents the external interface to the design module. The module internals represents the internal description of the design module-its behavior, its structure, or a mixture of both.

Behavioral Simulation

After the design phase, create a test bench waveform containing input stimulus to verify the functionality of the verilog code module using a simulation software i.e. Modelsim SE for different inputs to generate outputs and if it verifies then proceed further, otherwise modifications and necessary corrections will be done in the HDL code. This is called as the behavioral simulation.

Design Synthesis

After the correct simulations results, the design is then synthesized. During synthesis, the Xilinx ISE tool does the following operations:

a) HDL Compilation: The tool compiles all the sub-modules in the main module if any and then checks the syntax of the code written for the design.

b) Design Hierarchy Analysis: Analysis the hierarchy of the design.

c) HDL Synthesis: The process which translates VHDL or Verilog code into a device netlist formate, i.e. a complete circuit with logical elements such as Multiplexer, Adder/subtractors, counters, registers, flip flops Latches, Comparators, XORs, tristate buffers, decoders, etc. for the design. If the design contains more than one sub designs, ex. to implement a processor, we need a CPU as one design element and RAM as another and so on, and then the synthesis process generates netlist for each design element. Synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture, the designer has selected. The resulting netlist is saved to an NGC (Native Generic Circuit) file (for Xilinx® Synthesis Technology (XST)).

d) Advanced HDL Synthesis: 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, lookup 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 optimizes it. The final netlist output file has an extension of .ngc. This NGC file contains both the design data and the constraints. The optimization goal can be pre-specified to be the faster speed of operation or the minimum area of implementation before running this process. The level optimization effort can also be specified. The higher the effort, the more optimized is the design but higher effort can

also be specified. The higher the effort, the more optimized is the design but higher effort requires larger CPU time (i.e. the design time) because multiple optimization algorithms are tried to get the best result for the target architecture.

Design Implementation

The design implementation process consists of the following sub processes:

1. Translation: The Translate process combines all the input netlists and constraints to a logic design file. This information is saved a NGD (Native Generic Database) file. This can be done using NGD Build program and .ngd file describes the logical design reduced to the Xilinx device primitive cells. 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. 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.

2. Mapping: The Map process is run after the Translate process is complete. Map process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output Blocks (lOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. MAP program is used for this purpose.

3. Place and Route: 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.

4. Bitstream Generation: The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a "bitstream," 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.

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

6. Static timing analysis: Three types of static timing analysis can be performed that are:

(i) 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 interactively select timing paths in design for tracing.

(ii) Post-Map Static Timing Analysis: Analyze the timing results of the Map process. Post Map timing reports can be very useful in evaluating timing performance. Although route delays are not accounted for the logic delays can provide valuable information about the design. If logic delays 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 delays typically account for 45% to 65% of the total path delays. By identifying problem paths, 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-and-route software can use very low placement effort levels. In these cases, reducing effort levels allow to decrease runtimes while still meeting performance requirements.

(iii) Post Place and Route Static Timing Analysis: Analyze the timing results of the Post- Place and Route process. Post-PAR timing reports incorporate all delays to provide a comprehensive timing summary. If a placed and routed design has met all of timing constraints, then proceed by creating configuration data and downloading a device. On the other hand, identify problems and the timing reports, try fixing the problems by

increasing the placer effort level, using re-entrant routing, or using multi-pass place and route. 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 paths.

(iv) Timing Simulation: Perform Post-Place and Route simulation after the design has been and routed. After the design has been through all of the Xilinx implementation tools, a timing simulation netlist can be created. This simulation process allows to see how the design will behave in the circuit. Before performing this simulation it will benefit to create a test bench or test fixture to apply stimulus to the design. After this a .ncd file will be created that is used for the generation of power results of the design.

CHAPTER 5
RESULTS AND DISCUSSIONS

CORDIC algorithm is used to compute $\sin\theta$ and $\cos\theta$ by rotation method. Figure 5.1 consists of Modelsim simulation result for real input and real outputs angle $\sin\theta$ and $\cos\theta$ in the form of waveform for original CORDIC.

5.1 Modelsim Simulation Results:

5.1.1 For sine-cosine real input and real output of CORDIC algorithm

$$\theta = 30^{\circ}$$
$$\sin 30^{\circ} = 0.5$$
$$\cos 30^{\circ} = 0.8637$$

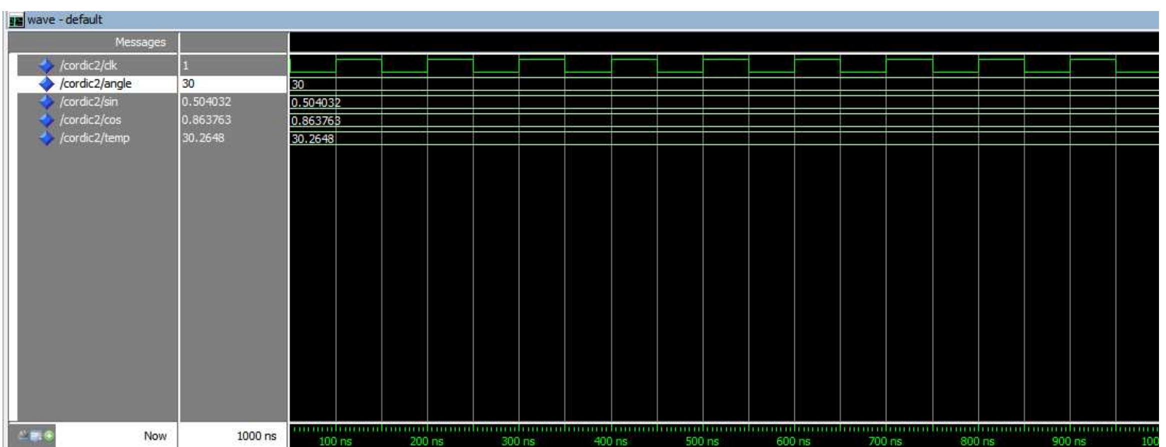


Figure 5.1: Modelsim result for Original CORDIC for real input and real output

5.2 Xilinx Simulation results:

5.2.1 Simulation result of sine cosine CORDIC for 16-bit, 24-bit and 32-bit fixed point numbers

In this thesis, three types of signed fixed point number representations that are 16-bit, 24-bit and 32-bit have been used.

5.2.1.1 Simulation result for Original CORDIC

Block diagram generated by Xilinx 13.1i for sine-cosine using Original CORDIC for 16-bit, 24-bit and 32-bit fixed point numbers are shown in figure (5.3), (5.6), (5.9) respectively. Here inputs are angle (binary input), clk (clock) and outputs are sine (binary output) and cosine (binary output).

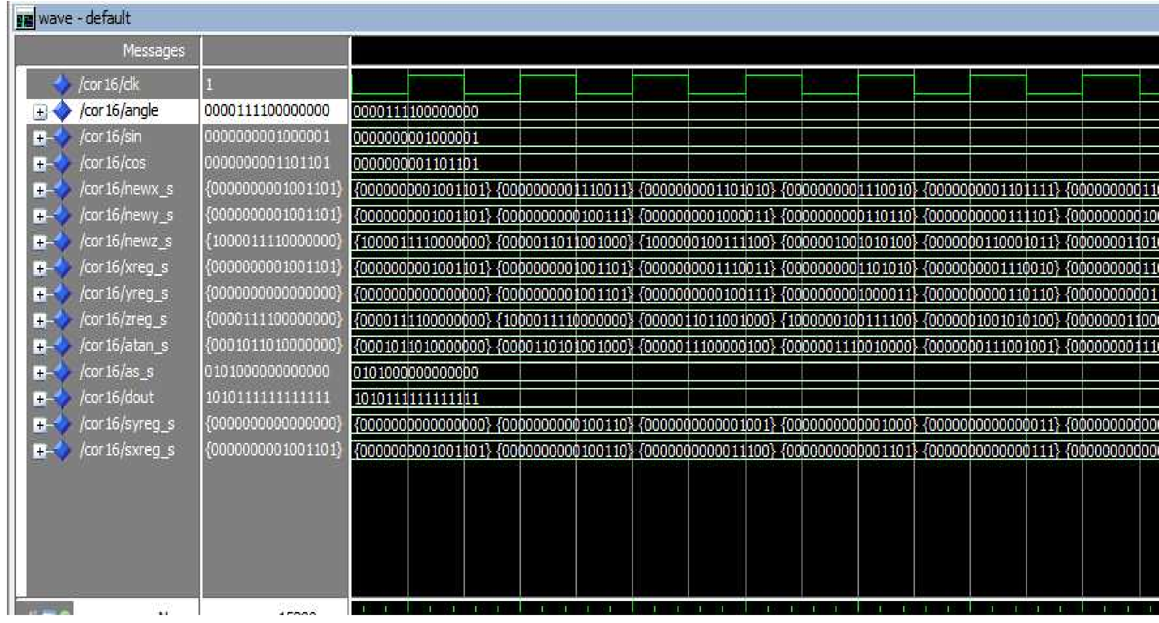


Figure 5.2 Simulation result for Sine-cosine of 16-bit fixed point Original CORDIC

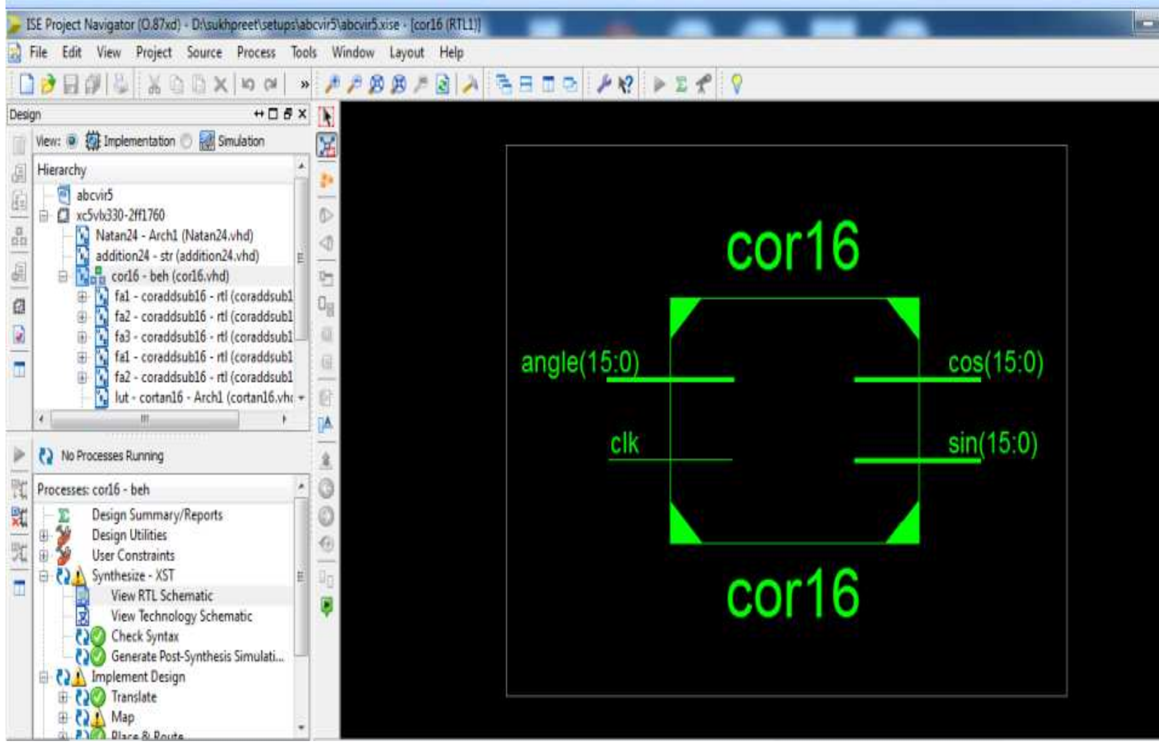


Figure 5.3 Top level schematic for 16-bit fixed point Original CORDIC

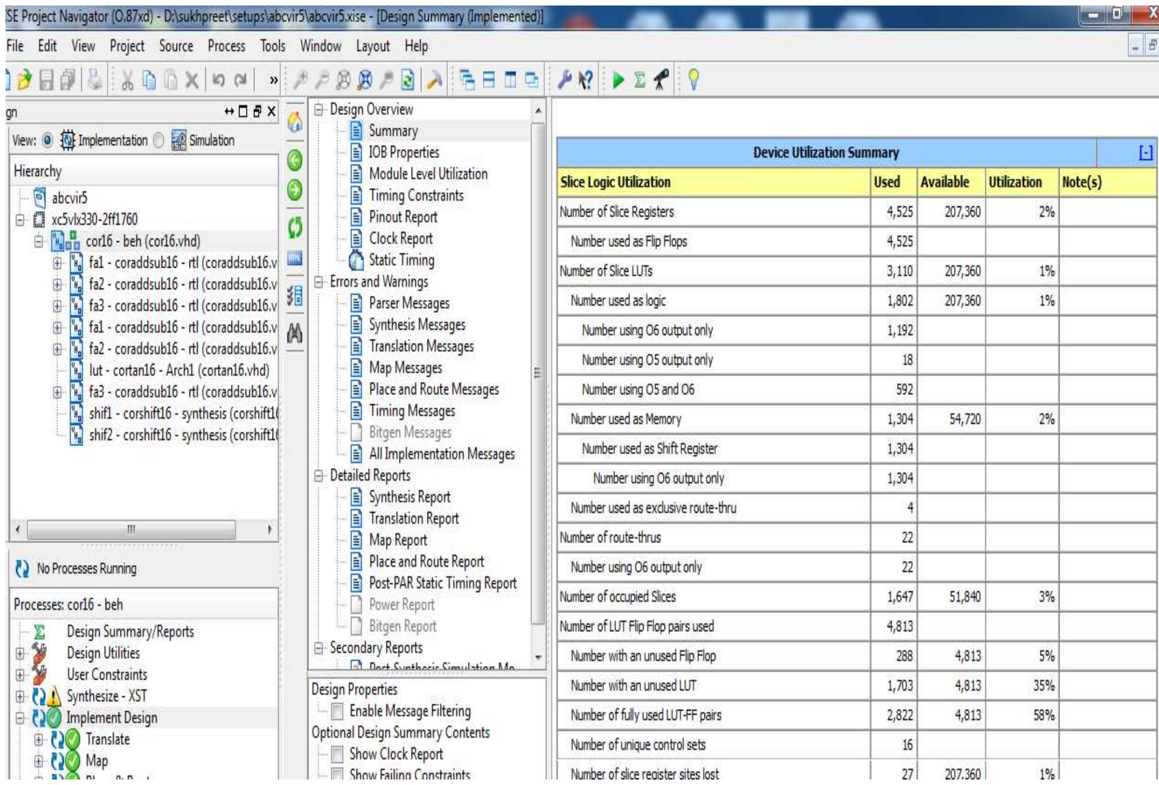


Figure 5.4 Summary report for 16-bit fixed point Original CORDIC

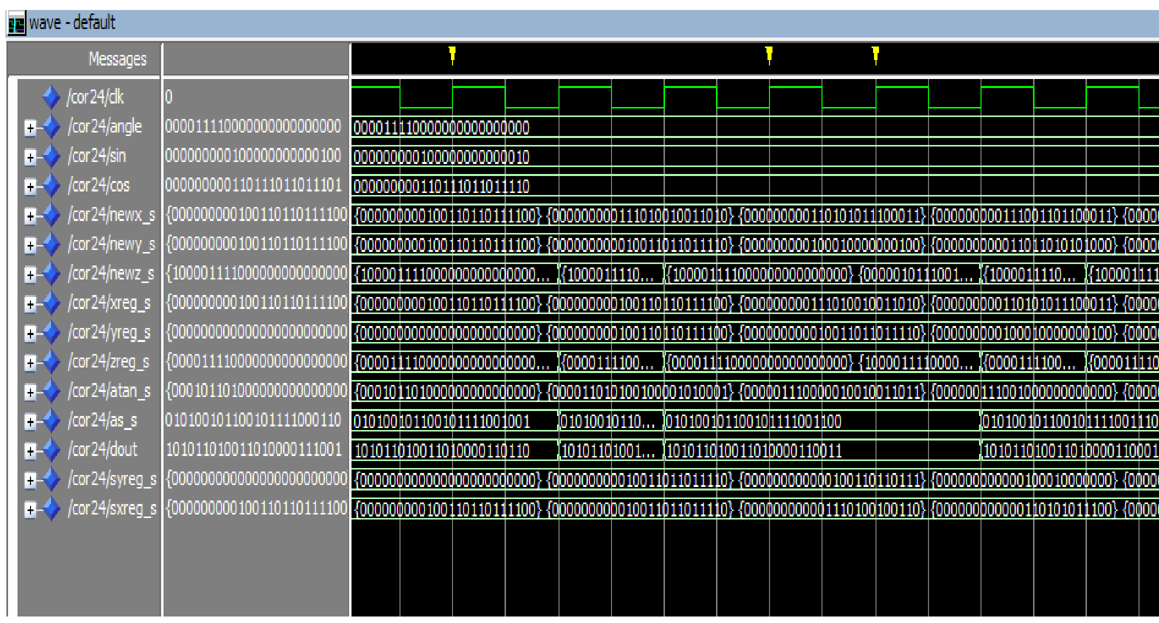


Figure 5.5 Simulation result for Sine-cosine of 24-bit fixed point Original CORDIC

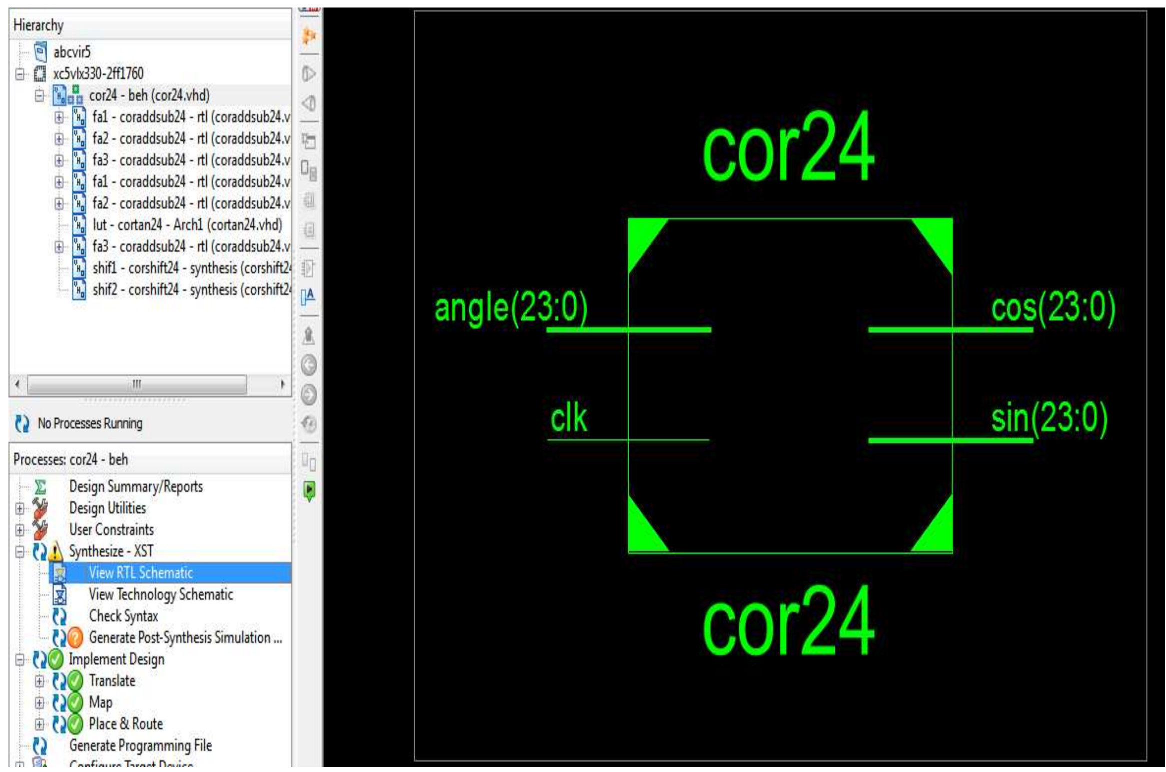
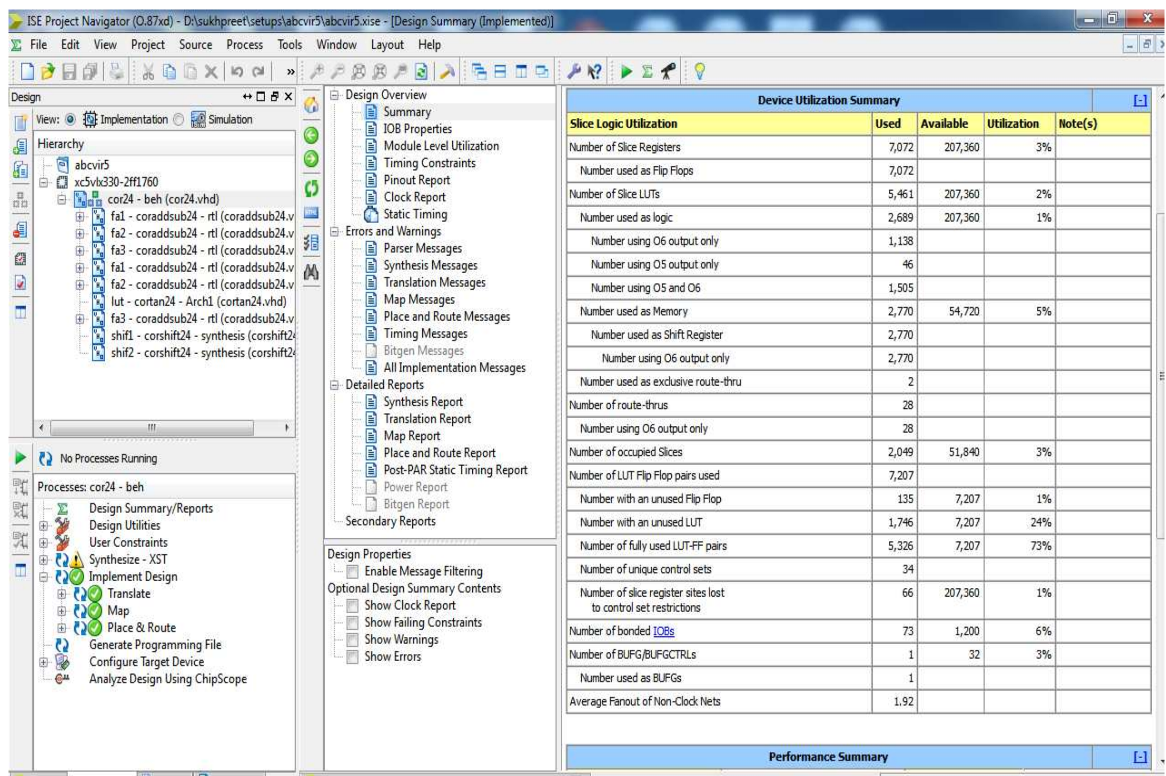


Figure 5.6 Top level schematic for 16-bit fixed point Original CORDIC



Device Utilization Summary

Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	7,072	207,360	3%	
Number used as Flip Flops	7,072			
Number of Slice LUTs	5,461	207,360	2%	
Number used as logic	2,689	207,360	1%	
Number using O6 output only	1,138			
Number using O5 output only	46			
Number using O5 and O6	1,505			
Number used as Memory	2,770	54,720	5%	
Number used as Shift Register	2,770			
Number using O6 output only	2,770			
Number used as exclusive route-thru	2			
Number of route-thrus	28			
Number using O6 output only	28			
Number of occupied Slices	2,049	51,840	3%	
Number of LUT Flip Flop pairs used	7,207			
Number with an unused Flip Flop	135	7,207	1%	
Number with an unused LUT	1,746	7,207	24%	
Number of fully used LUT-FF pairs	5,326	7,207	73%	
Number of unique control sets	34			
Number of slice register sites lost to control set restrictions	66	207,360	1%	
Number of bonded IOBs	73	1,200	6%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Average Fanout of Non-Clock Nets	1.92			

Performance Summary

Figure 5.7 Summary report for 24-bit fixed point Original CORDIC

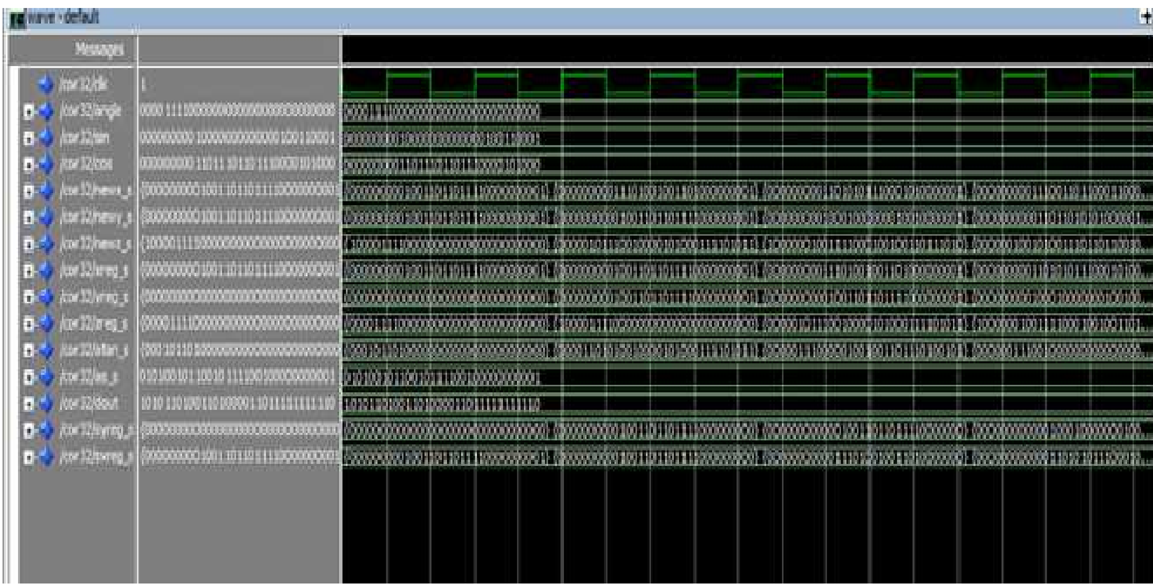


Figure 5.8 Simulation result for Sine-cosine of 32-bit fixed point Original CORDIC

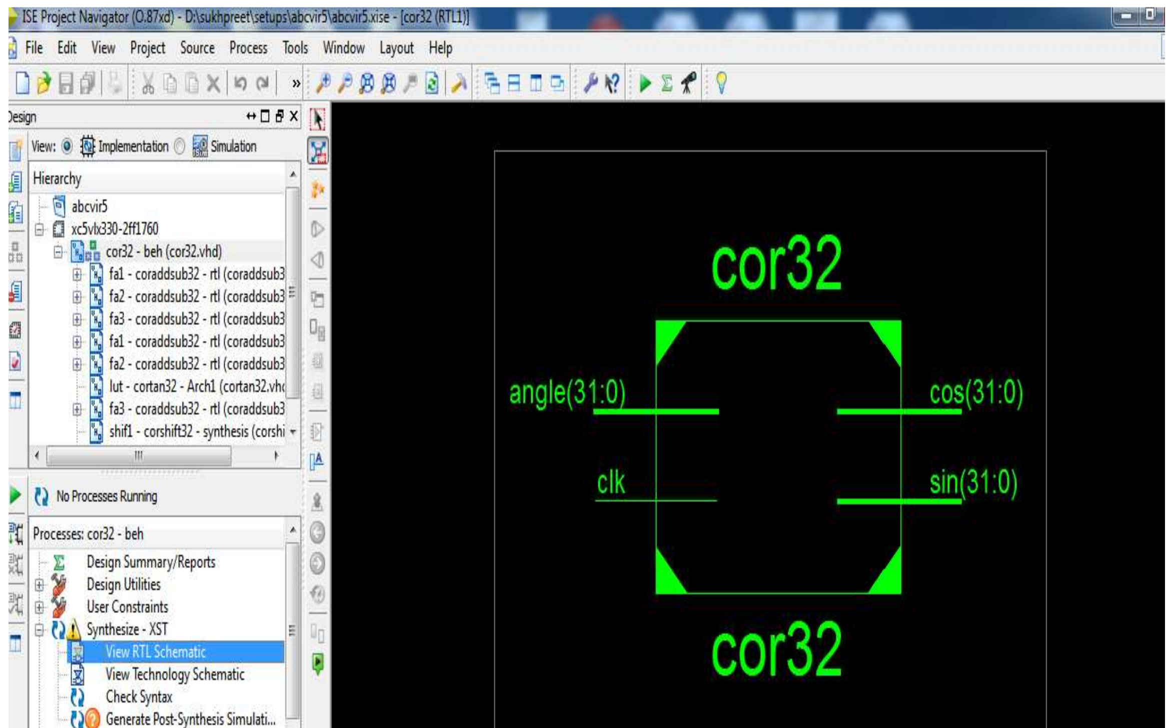


Figure 5.9 Top level schematic for 32-bit fixed point Original CORDIC

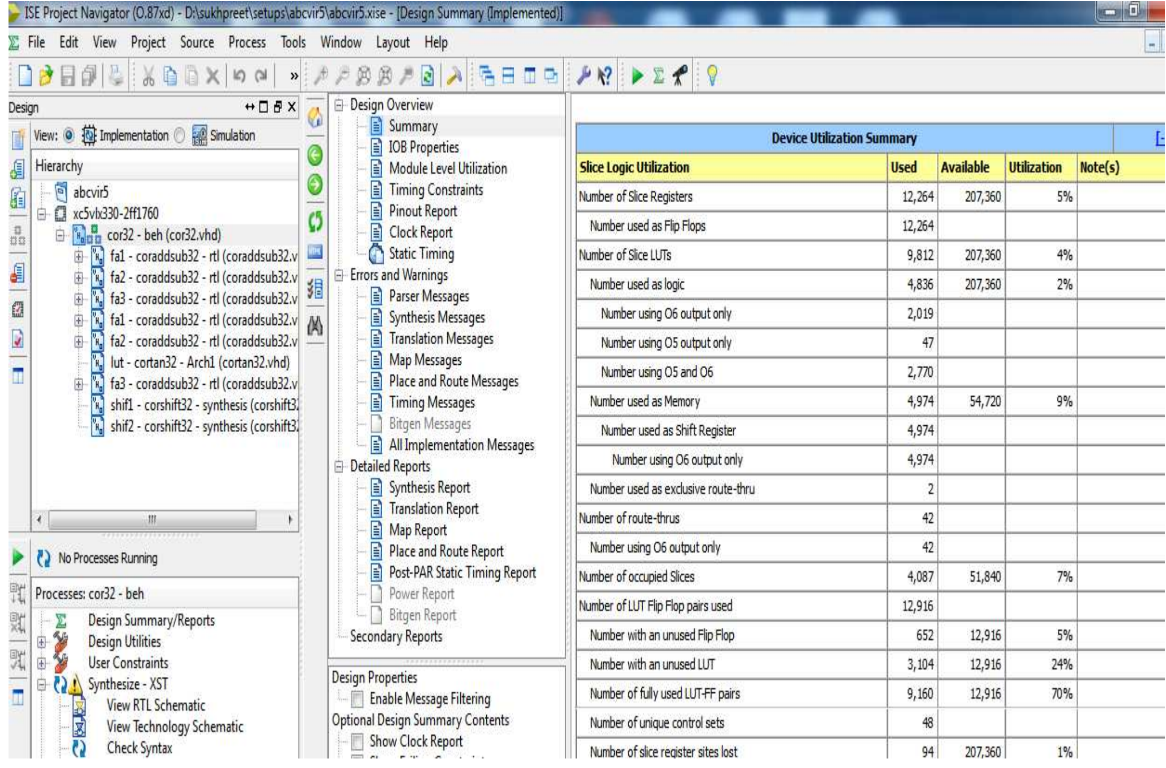


Figure 5.10 Summary report for 32-bit fixed point Original CORDIC

Table 5.1 Comparison between 16 bit, 24 bit and 32 bit fixed point Original CORDIC

	16 bit		24 bit		32 bit	
	Total	utilization	Total	utilization	Total	utilization
Number of slice Registers	4525/ 207360	2%	7072/ 207360	3%	12264/ 207360	5%
Number of slice LUTs	3110/ 207360	1%	5461/ 207360	2%	9812/ 207360	4%
Number of bonded IOBs	49/1200	4%	73/1200	6%	97/1200	8%
Delay	1.88ns		2.22ns		2.41ns	
Frequency	525.91 MHz		448.69 MHz		414.81 MHz	

5.2.1.2 Simulation result for Control CORDIC

Block diagram generated by Xilinx 13.1i for sine-cosine using Control CORDIC for 16-bit, 24-bit and 32-bit fixed point numbers are shown in figure (5.12), (5.15), (5.18) respectively. Here inputs are angle (binary input), clk (clock) and outputs are sine (binary output) and cosine (binary output).

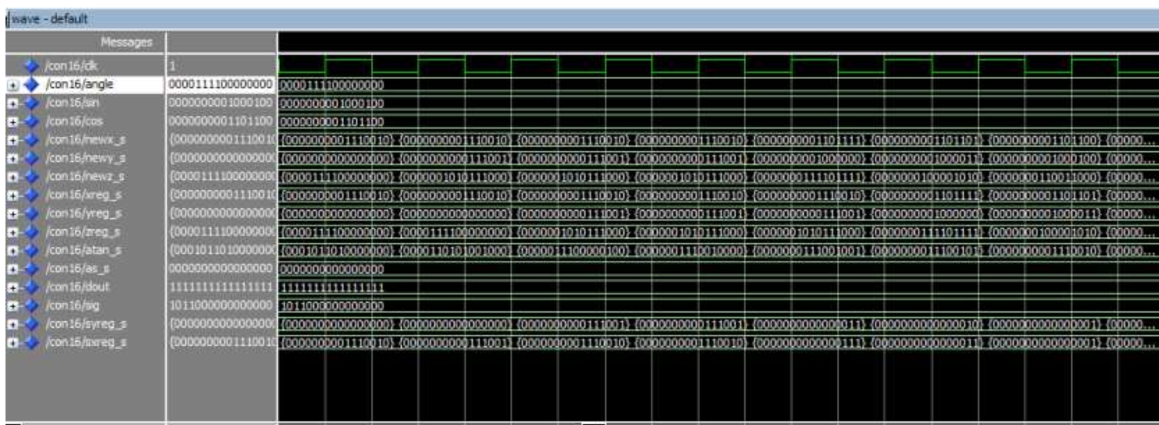


Figure 5.11 Simulation result for Sine-cosine of 16-bit fixed point Control CORDIC

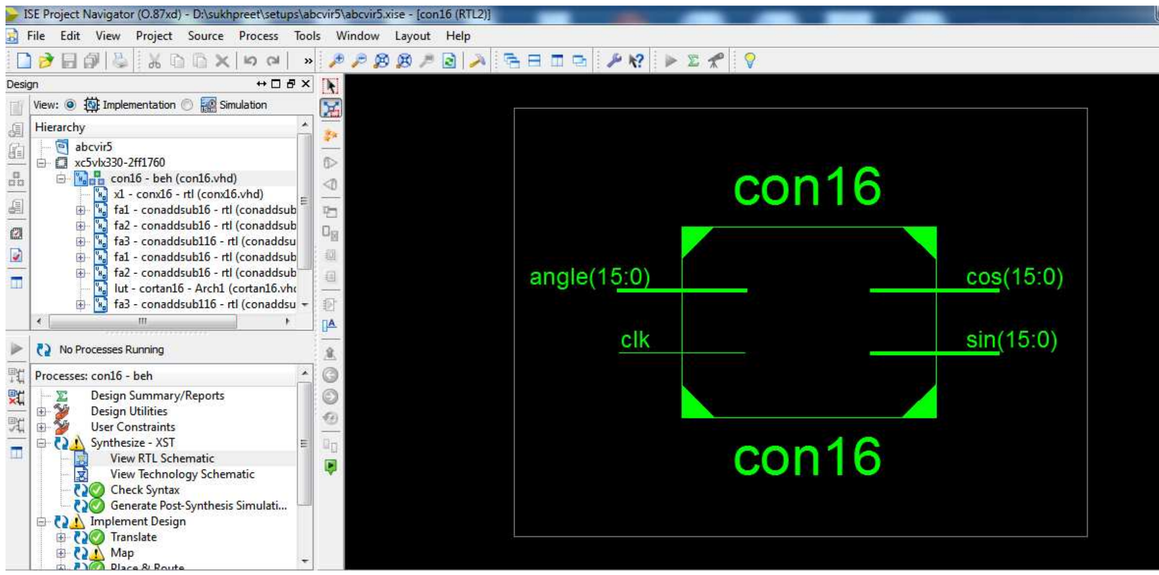
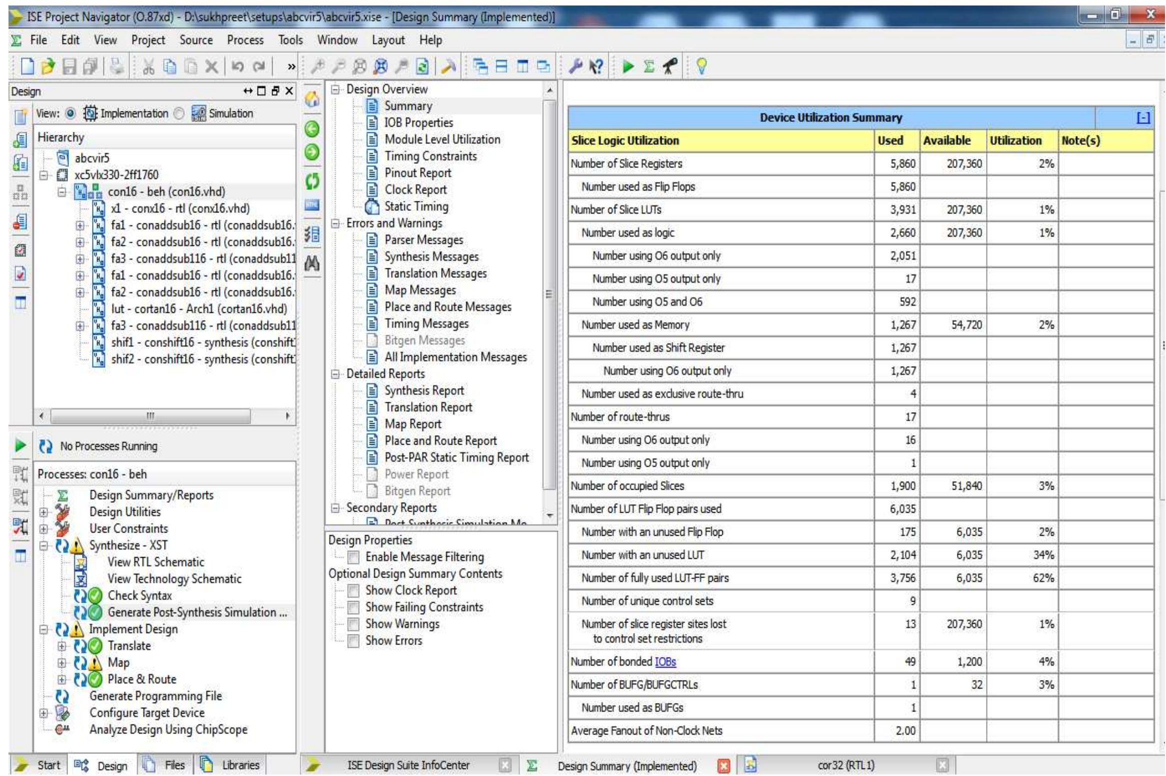


Figure 5.12 Top level schematic for 16-bit fixed point Control CORDIC



Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	5,860	207,360	2%	
Number used as Flip Flops	5,860			
Number of Slice LUTs	3,931	207,360	1%	
Number used as logic	2,660	207,360	1%	
Number using O6 output only	2,051			
Number using O5 output only	17			
Number using O5 and O6	592			
Number used as Memory	1,267	54,720	2%	
Number used as Shift Register	1,267			
Number using O6 output only	1,267			
Number used as exclusive route-thru	4			
Number of route-thrus	17			
Number using O6 output only	16			
Number using O5 output only	1			
Number of occupied Slices	1,900	51,840	3%	
Number of LUT Flip Flop pairs used	6,035			
Number with an unused Flip Flop	175	6,035	2%	
Number with an unused LUT	2,104	6,035	34%	
Number of fully used LUT-FF pairs	3,756	6,035	62%	
Number of unique control sets	9			
Number of slice register sites lost to control set restrictions	13	207,360	1%	
Number of bonded IOBs	49	1,200	4%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Average Fanout of Non-Clock Nets	2.00			

Figure 5.13 Summary report for 16-bit fixed point Control CORDIC

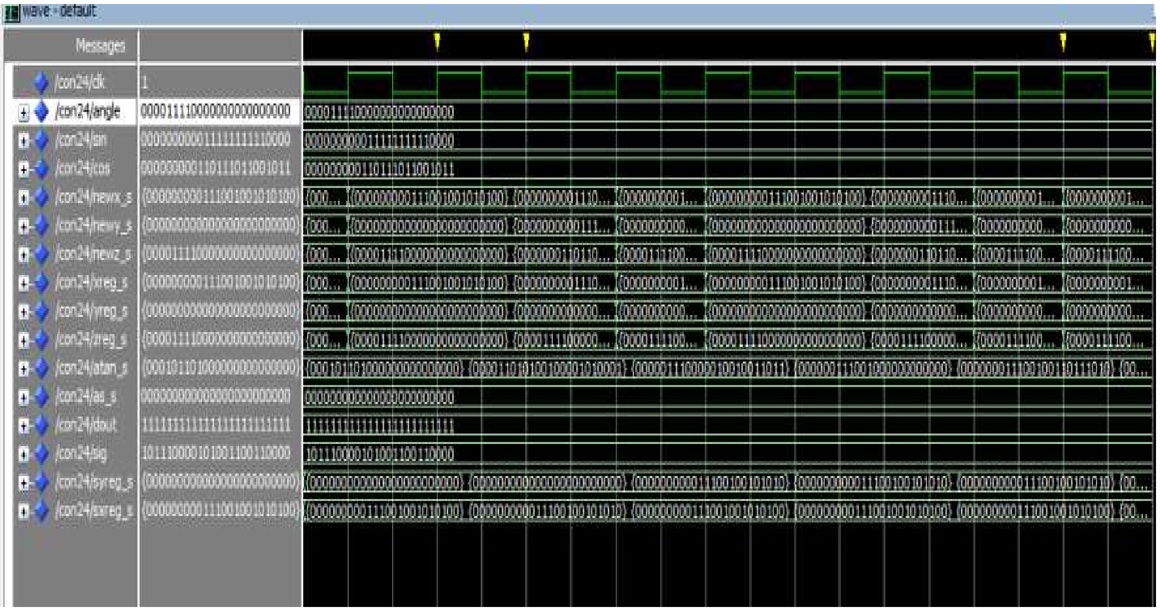


Figure 5.14 Simulation result for Sine-cosine of 24-bit fixed point Control CORDIC

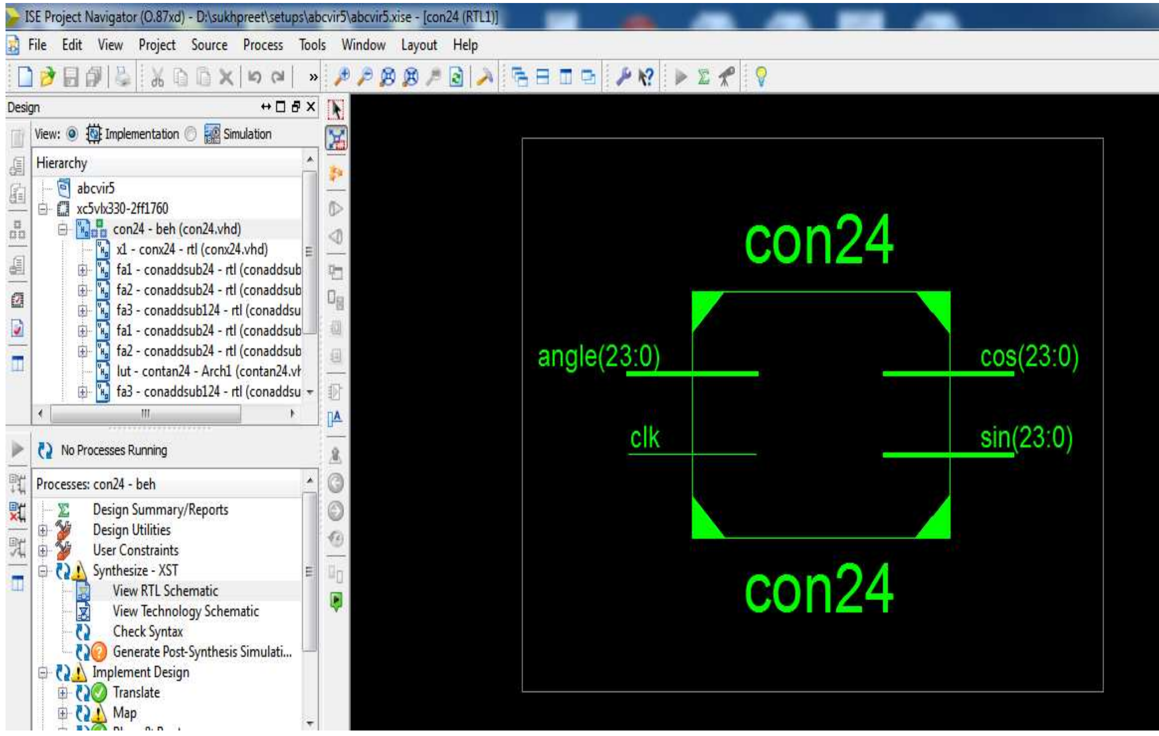
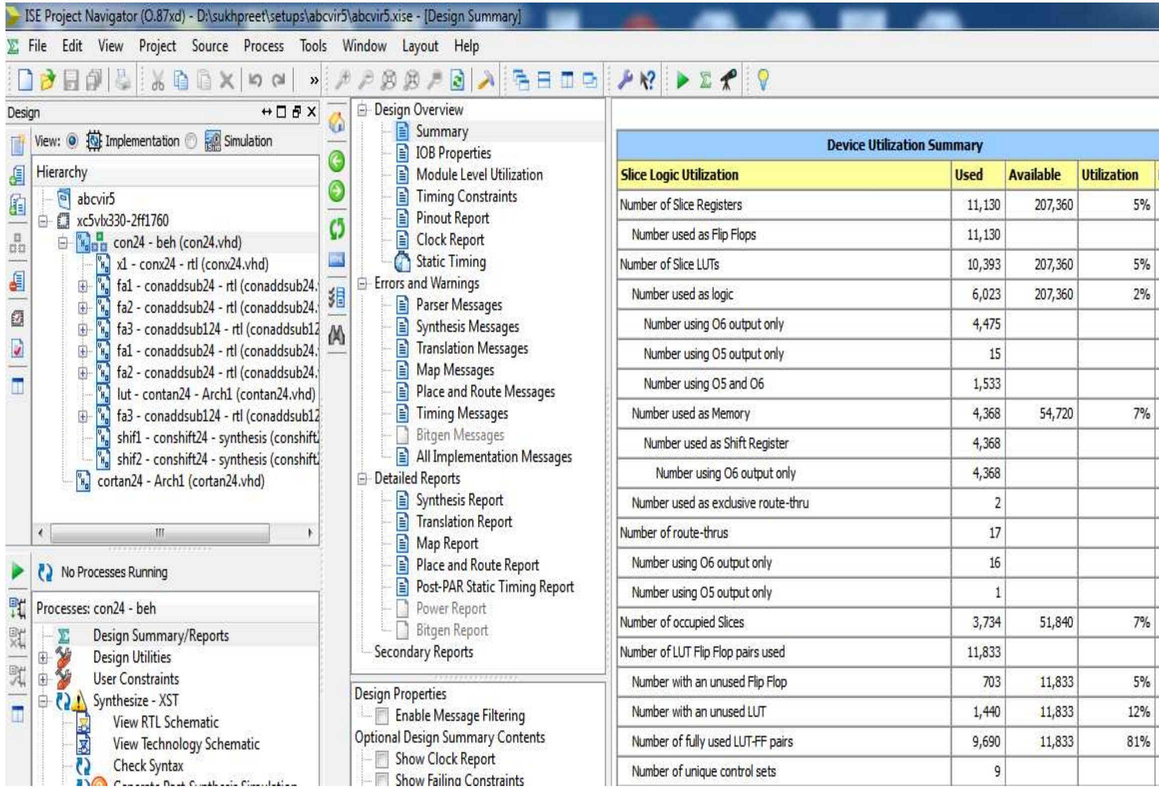


Figure 5.15 Top level schematic for 24-bit fixed point Control CORDIC



Device Utilization Summary

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	11,130	207,360	5%
Number used as Flip Flops	11,130		
Number of Slice LUTs	10,393	207,360	5%
Number used as logic	6,023	207,360	2%
Number using O6 output only	4,475		
Number using O5 output only	15		
Number using O5 and O6	1,533		
Number used as Memory	4,368	54,720	7%
Number used as Shift Register	4,368		
Number using O6 output only	4,368		
Number used as exclusive route-thru	2		
Number of route-thrus	17		
Number using O6 output only	16		
Number using O5 output only	1		
Number of occupied Slices	3,734	51,840	7%
Number of LUT Flip Flop pairs used	11,833		
Number with an unused Flip Flop	703	11,833	5%
Number with an unused LUT	1,440	11,833	12%
Number of fully used LUT-FF pairs	9,690	11,833	81%
Number of unique control sets	9		

Figure 5.16 Summary report for 24-bit fixed point Control CORDIC

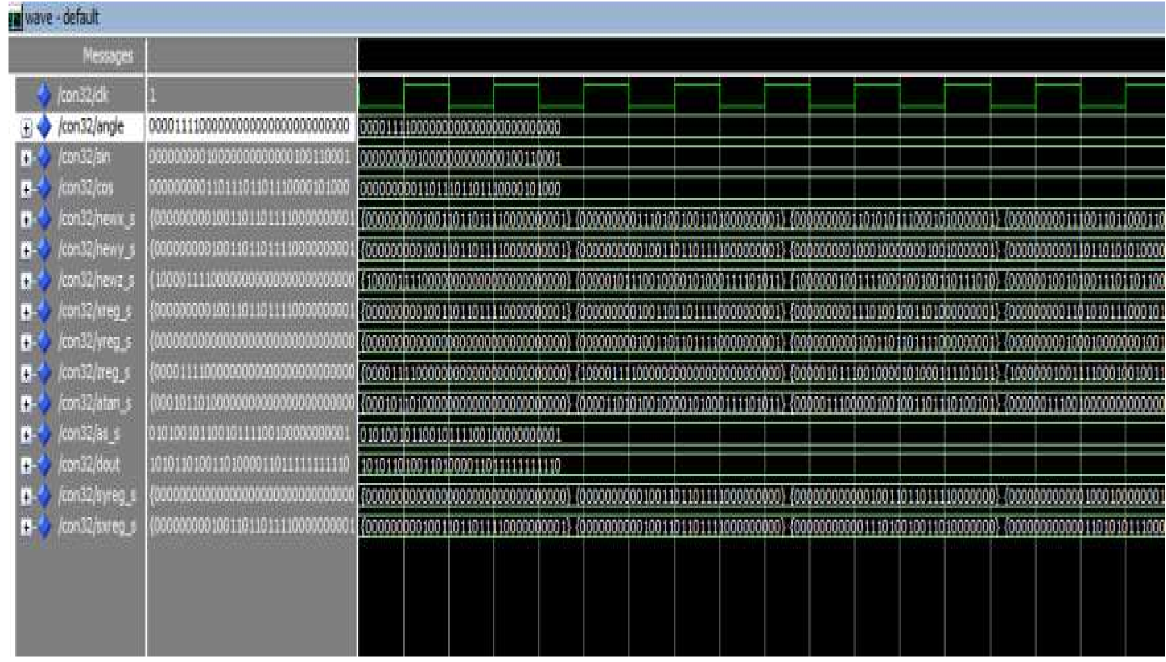


Figure 5.17 Simulation result for Sine-cosine of 32-bit fixed point Control CORDIC

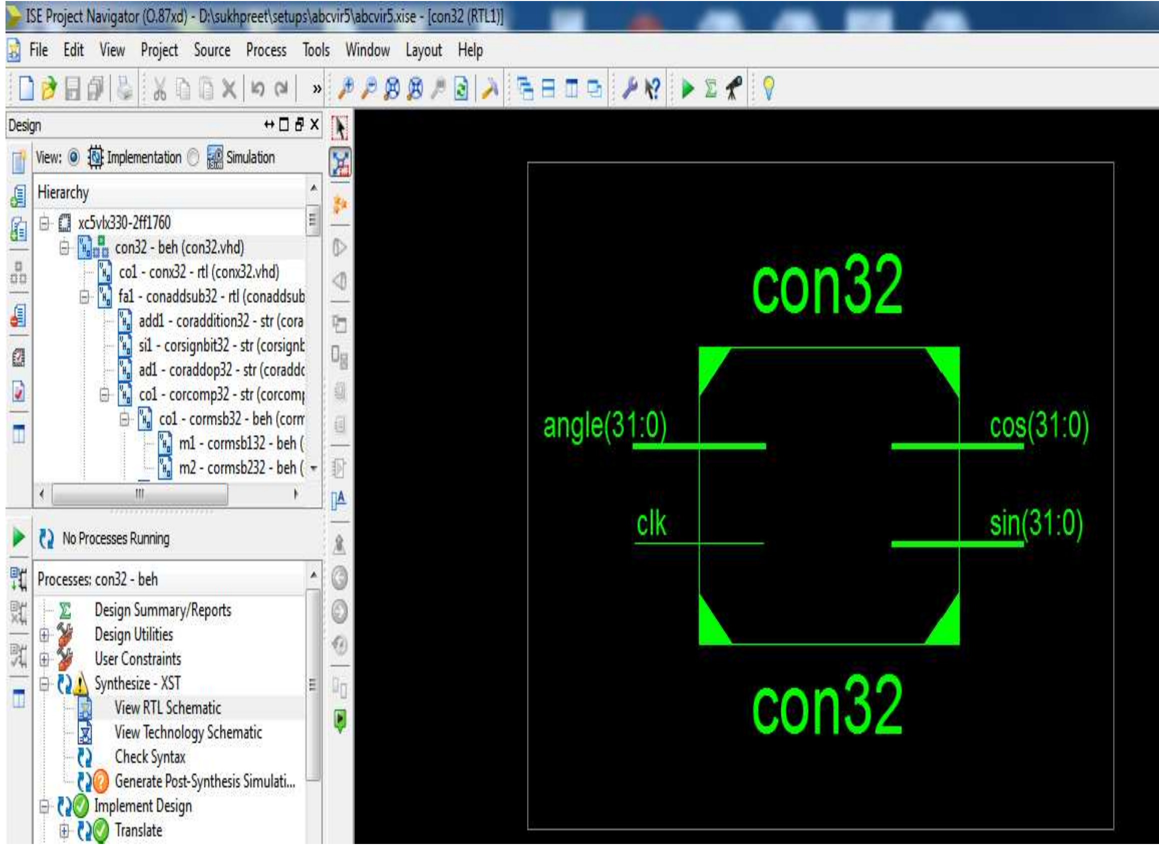


Figure 5.18 Top level schematic for 32-bit fixed point Control CORDIC

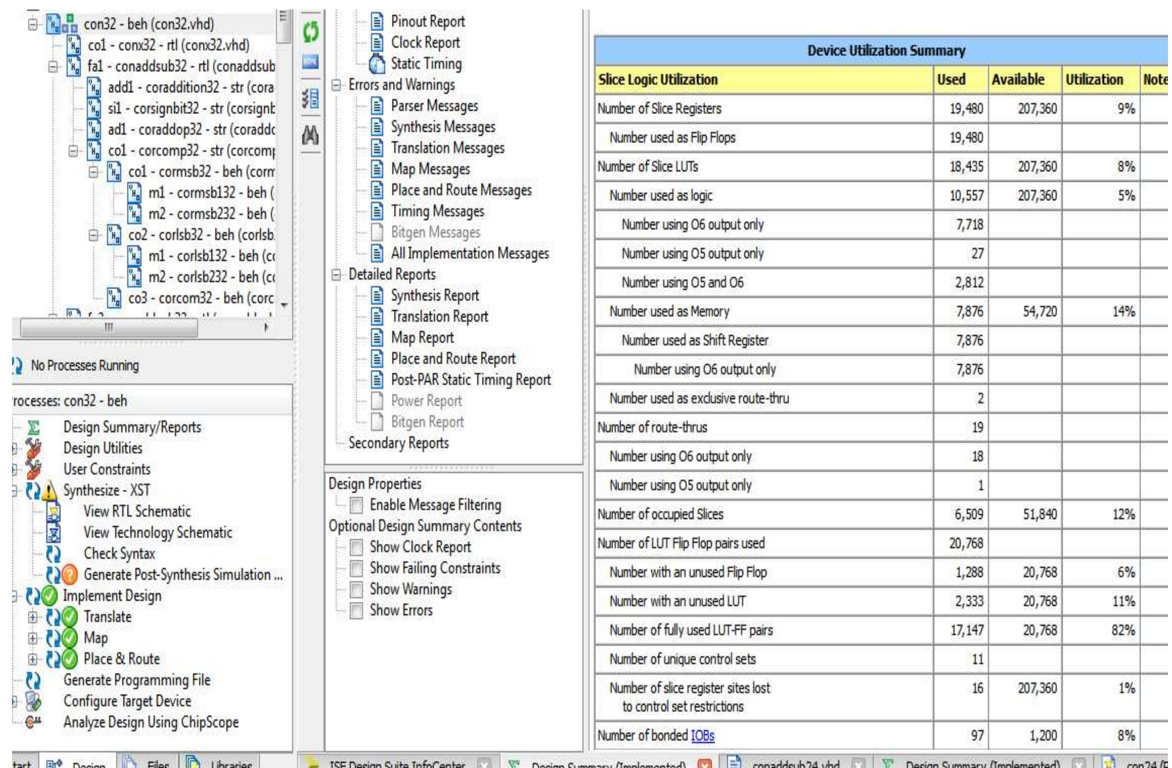


Figure 5.16 Summary report for 32-bit fixed point Control CORDIC

Table 5.2 Comparison between 16 bit, 24 bit and 32 bit fixed point Control CORDIC

	16 bit		**24 bit**		**32 bit**	
	Total	utilization	Total	utilization	Total	Utilization
Number of slice Registers	5860/ 207360	2%	11130/ 207360	5%	19490/ 207360	9%
Number of slice LUTs	3931/ 207360	1%	10393/ 207360	5%	18435/ 207360	8%
Number of bonded IOBs	49/1200	4%	73/1200	6%	97/1200	8%
Delay	1.88ns		2.22ns		2.41ns	
Frequency	525.91MHz		448.69MHz		414.81MHz	

5.2.1.3 Simulation result for Angle Recoding CORDIC

Block diagram generated by Xilinx 13.1i for sine-cosine using Angle Recoding CORDIC for 16-bit, 24-bit and 32-bit fixed point numbers are shown in figure (5.21), (5.24), (5.27) respectively. Here inputs are angle (binary input), clk (clock) and outputs are sine (binary output) and cosine (binary output).

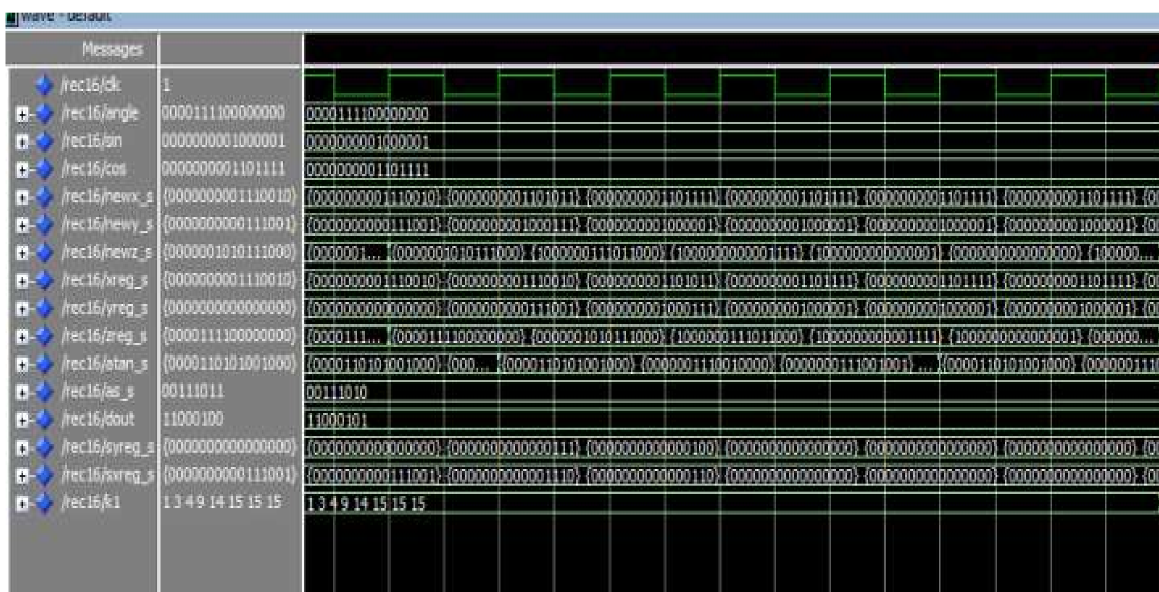


Figure 5.20 Simulation result for Sine-cosine of 16-bit fixed point Angle Recoding CORDIC

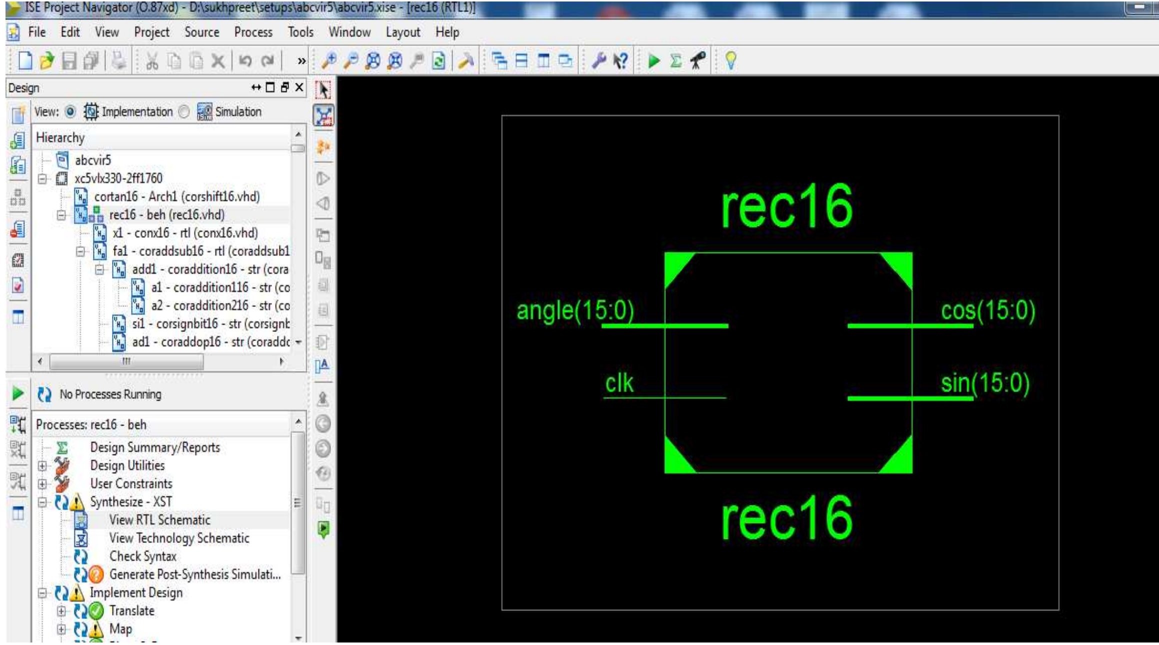
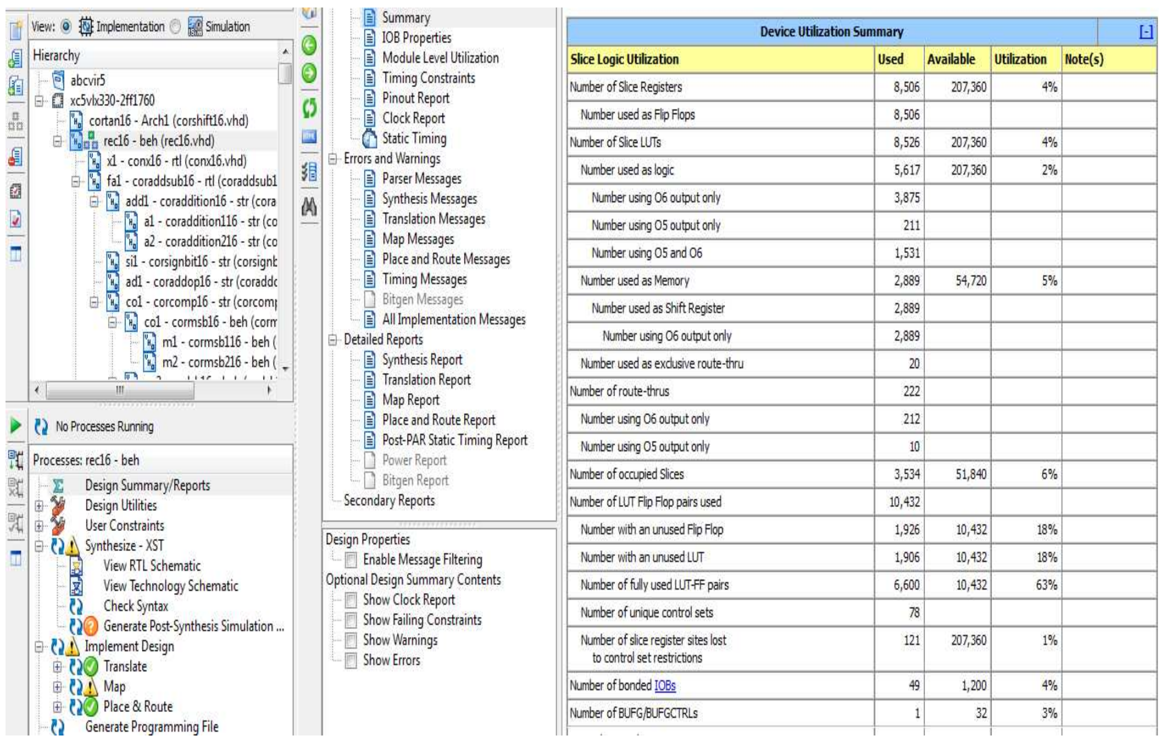


Figure 5.21 Top level schematic for 16-bit fixed point Angle Recoding CORDIC



Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	8,506	207,360	4%	
Number used as Flip Flops	8,506			
Number of Slice LUTs	8,526	207,360	4%	
Number used as logic	5,617	207,360	2%	
Number using O6 output only	3,875			
Number using O5 output only	211			
Number using O5 and O6	1,531			
Number used as Memory	2,889	54,720	5%	
Number used as Shift Register	2,889			
Number using O6 output only	2,889			
Number used as exclusive route-thru	20			
Number of route-thrus	222			
Number using O6 output only	212			
Number using O5 output only	10			
Number of occupied Slices	3,534	51,840	6%	
Number of LUT Flip Flop pairs used	10,432			
Number with an unused Flip Flop	1,926	10,432	18%	
Number with an unused LUT	1,906	10,432	18%	
Number of fully used LUT-FF pairs	6,600	10,432	63%	
Number of unique control sets	78			
Number of slice register sites lost to control set restrictions	121	207,360	1%	
Number of bonded IOBs	49	1,200	4%	
Number of BUFG/BUFGCTRLs	1	32	3%	

Figure 5.22 Summary report for 16-bit fixed point Angle Recoding CORDIC

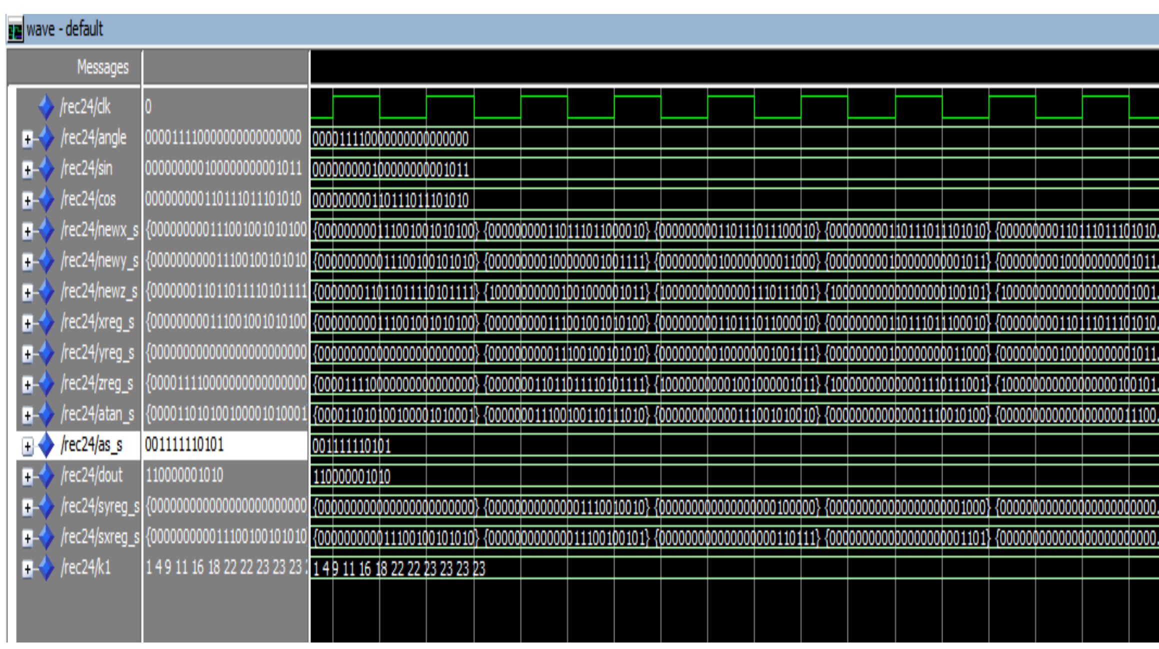


Figure 5.23 Simulation result for Sine-cosine of 24-bit fixed point Angle Recoding CORDIC

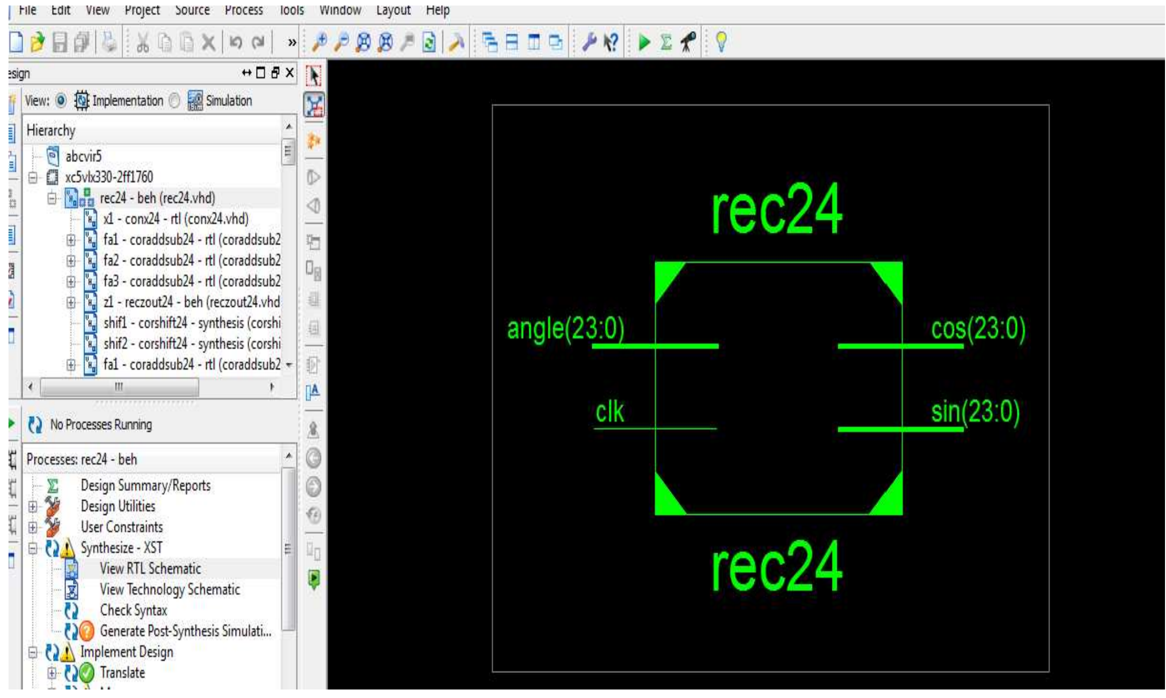
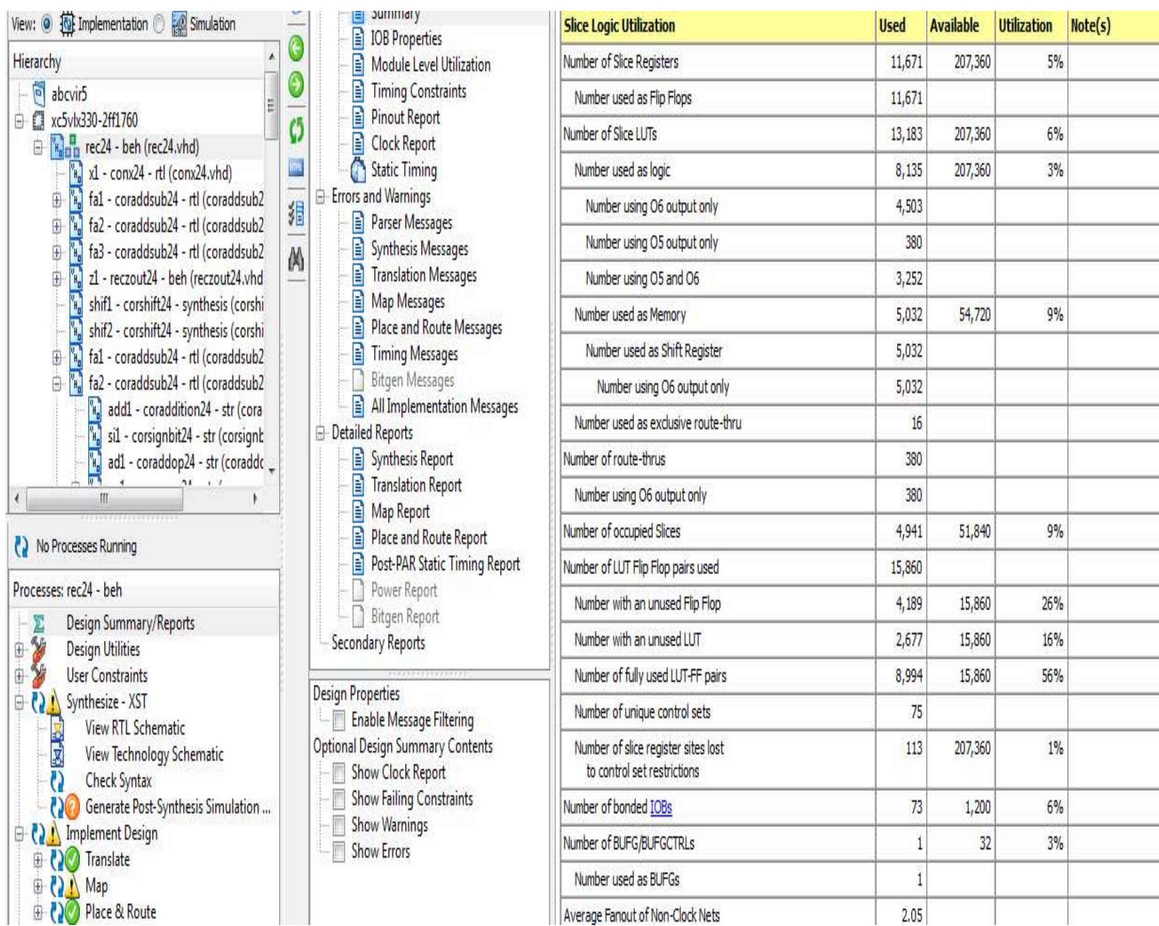


Figure 5.24 Top level schematic for 24-bit fixed point Angle Recoding CORDIC



Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	11,671	207,360	5%	
Number used as Flip Flops	11,671			
Number of Slice LUTs	13,183	207,360	6%	
Number used as logic	8,135	207,360	3%	
Number using O6 output only	4,503			
Number using O5 output only	380			
Number using O5 and O6	3,252			
Number used as Memory	5,032	54,720	9%	
Number used as Shift Register	5,032			
Number using O6 output only	5,032			
Number used as exclusive route-thru	16			
Number of route-thrus	380			
Number using O6 output only	380			
Number of occupied Slices	4,941	51,840	9%	
Number of LUT Flip Flop pairs used	15,860			
Number with an unused Flip Flop	4,189	15,860	26%	
Number with an unused LUT	2,677	15,860	16%	
Number of fully used LUT-FF pairs	8,994	15,860	56%	
Number of unique control sets	75			
Number of slice register sites lost to control set restrictions	113	207,360	1%	
Number of bonded IOBs	73	1,200	6%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Average Fanout of Non-Clock Nets	2.05			

Figure 5.25 Summary report for 24-bit fixed point Angle Recoding CORDIC

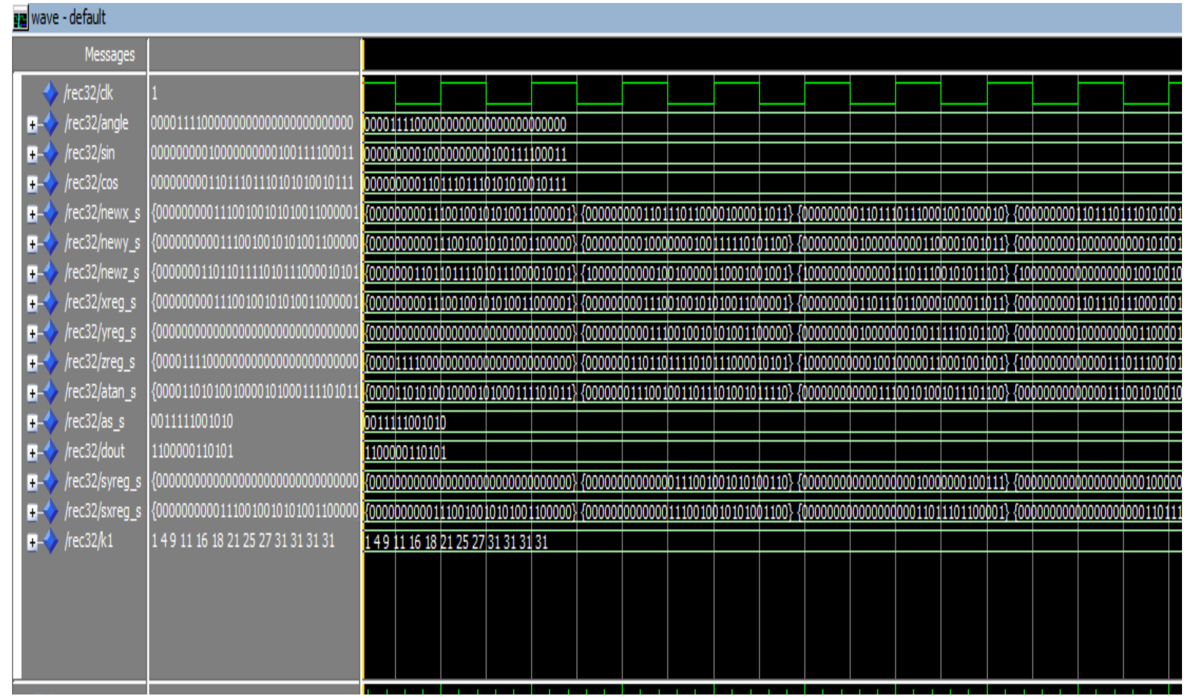


Figure 5.26 Simulation result for Sine-cosine of 32-bit fixed point Angle Recoding CORDIC

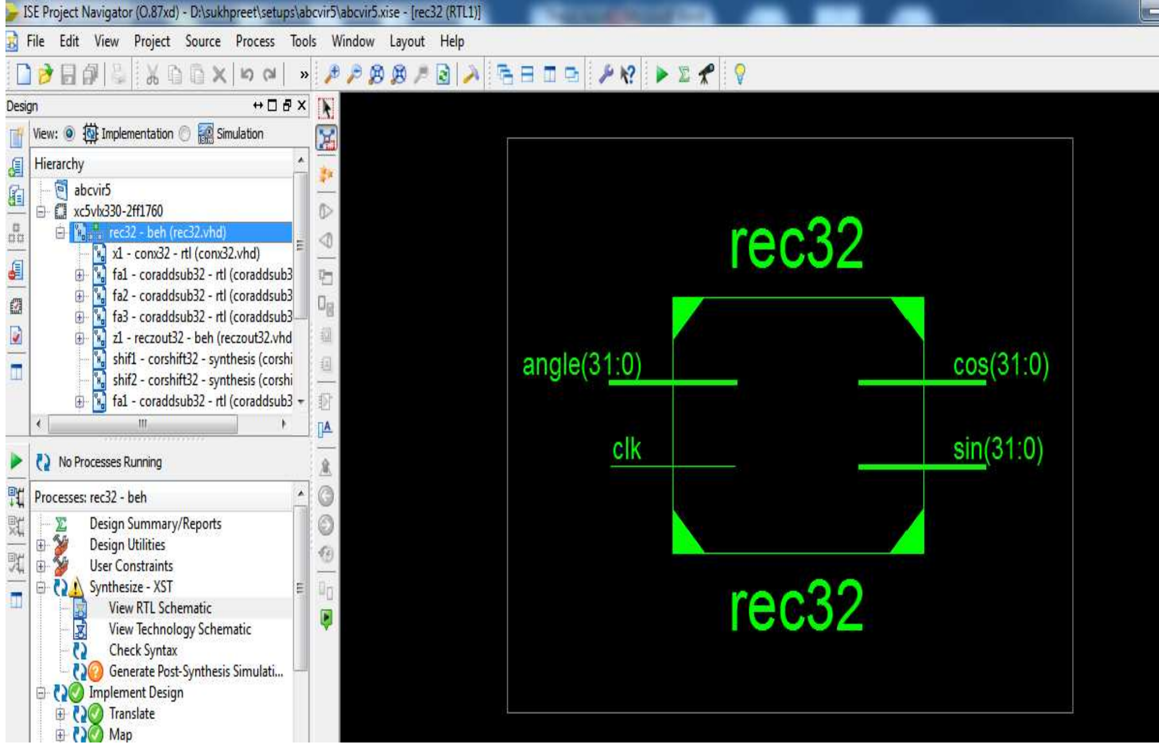
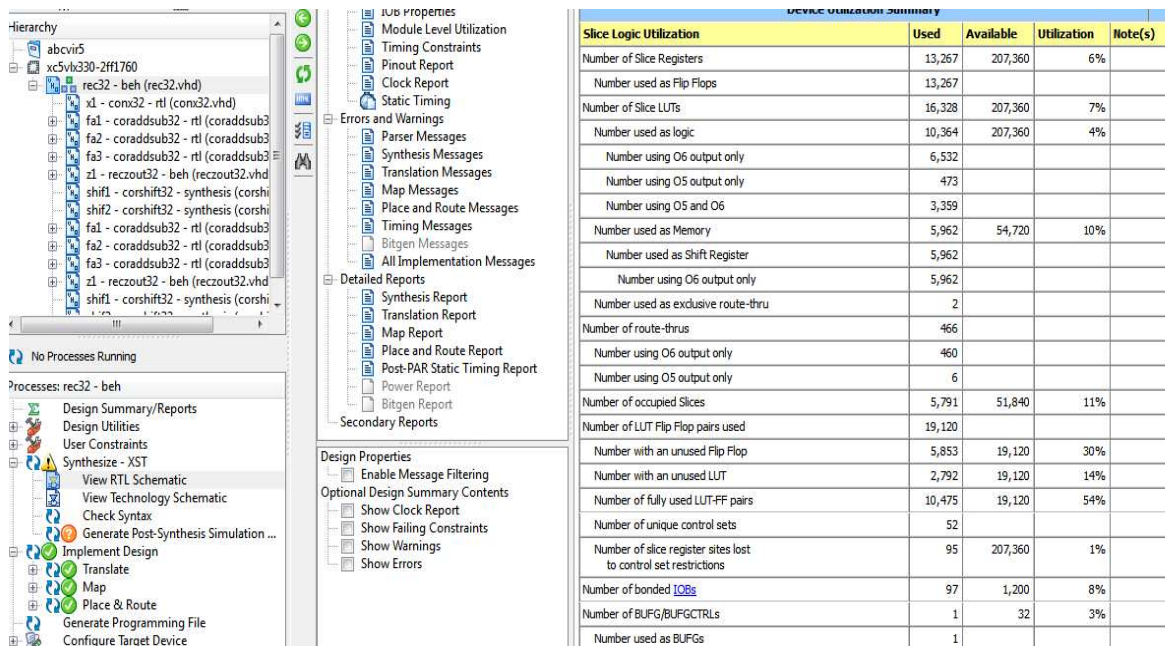


Figure 5.27 Top level schematic for 32-bit fixed point Angle Recoding CORDIC



Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	13,267	207,360	6%	
Number used as Flip Flops	13,267			
Number of Slice LUTs	16,328	207,360	7%	
Number used as logic	10,364	207,360	4%	
Number using O6 output only	6,532			
Number using O5 output only	473			
Number using O5 and O6	3,359			
Number used as Memory	5,962	54,720	10%	
Number used as Shift Register	5,962			
Number using O6 output only	5,962			
Number used as exclusive route-thru	2			
Number of route-thrus	466			
Number using O6 output only	460			
Number using O5 output only	6			
Number of occupied Slices	5,791	51,840	11%	
Number of LUT Flip Flop pairs used	19,120			
Number with an unused Flip Flop	5,853	19,120	30%	
Number with an unused LUT	2,792	19,120	14%	
Number of fully used LUT-FF pairs	10,475	19,120	54%	
Number of unique control sets	52			
Number of slice register sites lost to control set restrictions	95	207,360	1%	
Number of bonded IOBs	97	1,200	8%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			

Figure 5.28 Summary report for 32-bit fixed point Angle Recoding CORDIC

Table 5.3 Comparison between 16 bit, 24 bit and 32 bit fixed point Angle Recoding CORDIC

	16 bit		**24 bit**		**32 bit**	
	Total	utilization	Total	utilization	Total	Utilization
Number of slice Registers	8506/ 207360	4%	11671/ 207360	5%	13367/ 207360	6%
Number of slice LUTs	8526/ 207360	4%	13183/ 207360	6%	16328/ 207360	7%
Number of bonded IOBs	49/1200	4%	73/1200	6%	97/1200	8%
Delay	1.88ns		2.22ns		2.41ns	
Frequency	525.91MHz		448.69MHz		414.81MHz	

5.2.2 Simulation result of sine cosine CORDIC for 24-bit, 28-bit and 32-bit floating point numbers

In this thesis, three types of floating point number representation i.e. 24 bit, 28 bit and 32 bit (single precision IEEE 754-2008 standard) have been used.

5.2.2.1 Simulation result for CORDIC Algorithm

Block diagram generated by Xilinx 13.1i for sine-cosine using CORDIC for 24-bit, 28-bit and 32-bit floating point numbers are shown in figure (5.30), (5.33), (5.36) respectively. Here inputs are angle (binary input), clk (clock), start and outputs are sine (binary output) and cosine (binary output).

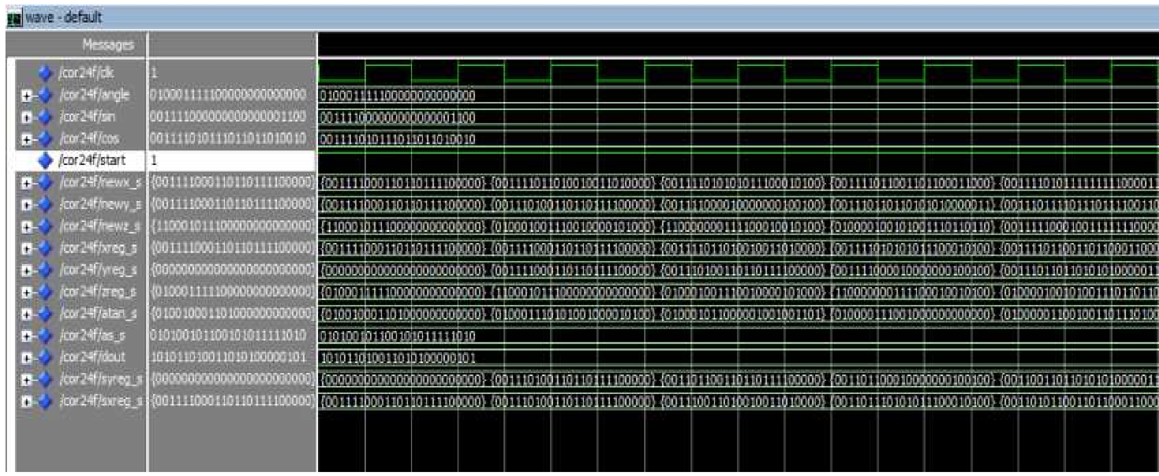


Figure 5.29 Simulation result for Sine-cosine of 24-bit floating point CORDIC Algorithm

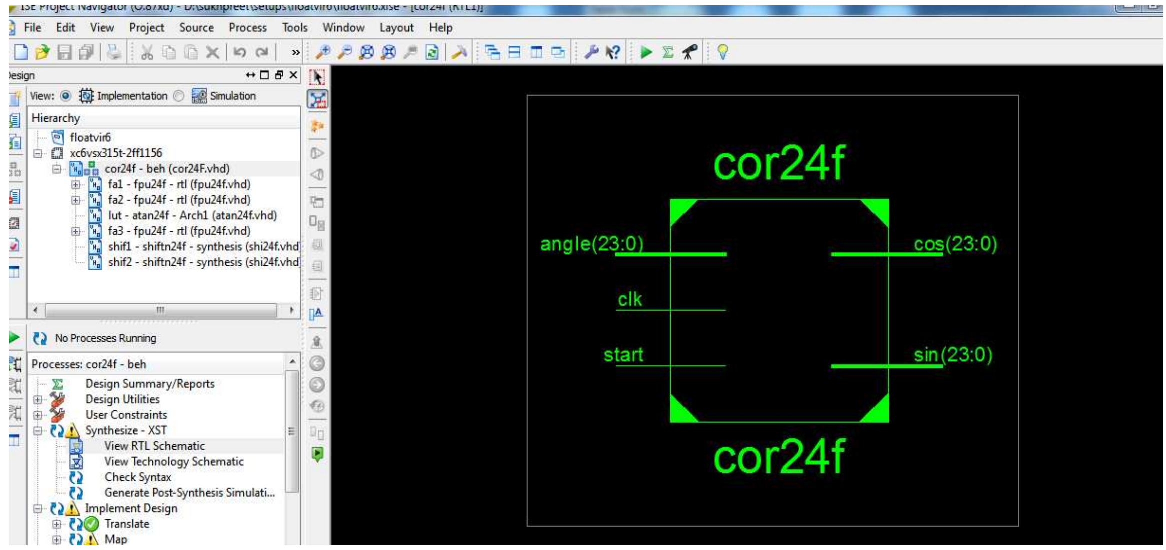
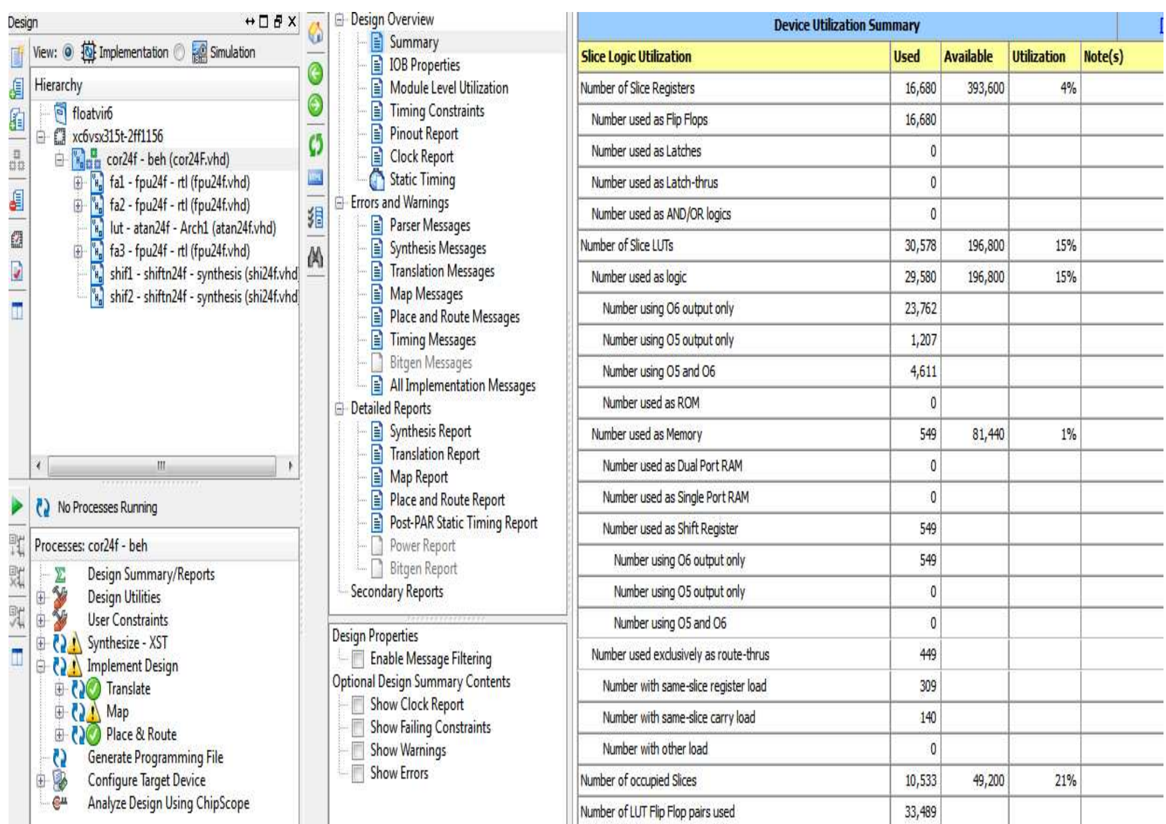


Figure 5.30 Top level schematic for 24-bit floating point CORDIC Algorithm



Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	16,680	393,600	4%	
Number used as Flip Flops	16,680			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	30,578	196,800	15%	
Number used as logic	29,580	196,800	15%	
Number using O6 output only	23,762			
Number using O5 output only	1,207			
Number using O5 and O6	4,611			
Number used as ROM	0			
Number used as Memory	549	81,440	1%	
Number used as Dual Port RAM	0			
Number used as Single Port RAM	0			
Number used as Shift Register	549			
Number using O6 output only	549			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used exclusively as route-thrus	449			
Number with same-slice register load	309			
Number with same-slice carry load	140			
Number with other load	0			
Number of occupied Slices	10,533	49,200	21%	
Number of LUT Flip Flop pairs used	33,489			

Figure 5.31 Summary report for 24-bit floating point CORDIC

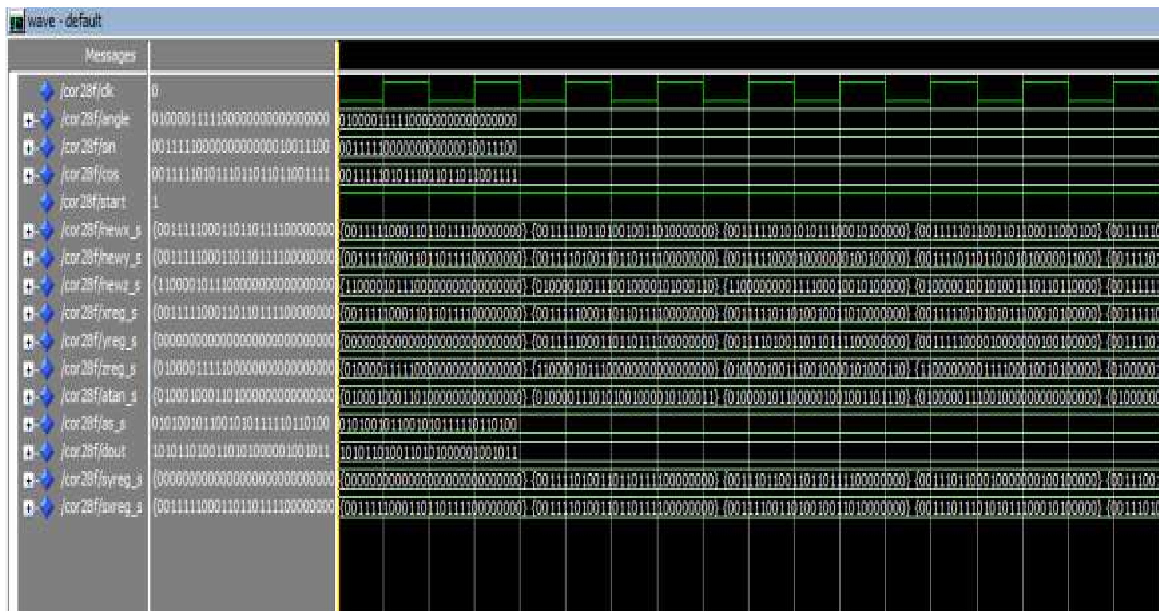


Figure 5.32 Simulation result for Sine-cosine of 28-bit floating point Angle Recoding CORDIC

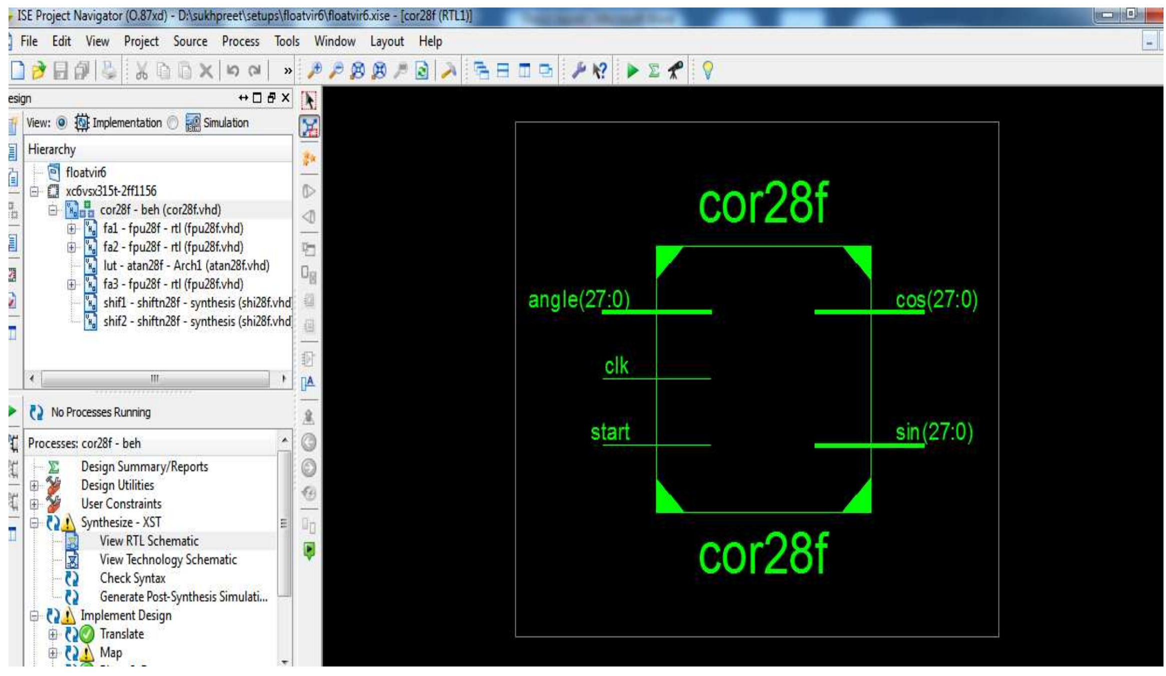
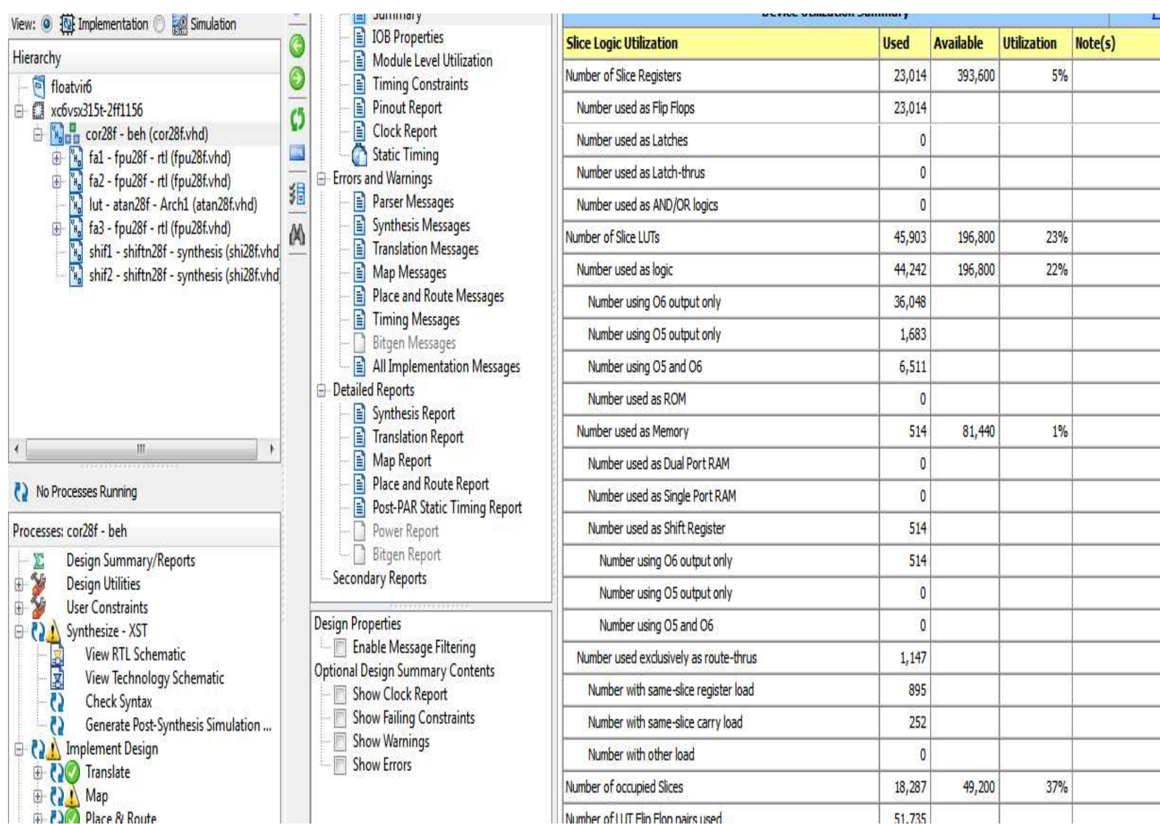


Figure 5.33 Top level schematic for 28-bit floating point CORDIC Algorithm



Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	23,014	393,600	5%	
Number used as Flip Flops	23,014			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	45,903	196,800	23%	
Number used as logic	44,242	196,800	22%	
Number using O6 output only	36,048			
Number using O5 output only	1,683			
Number using O5 and O6	6,511			
Number used as ROM	0			
Number used as Memory	514	81,440	1%	
Number used as Dual Port RAM	0			
Number used as Single Port RAM	0			
Number used as Shift Register	514			
Number using O6 output only	514			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used exclusively as route-thrus	1,147			
Number with same-slice register load	895			
Number with same-slice carry load	252			
Number with other load	0			
Number of occupied Slices	18,287	49,200	37%	
Number of LUT Flip Flop pairs used	51,735			

Figure 5.34 Summary report for 28-bit floating point CORDIC

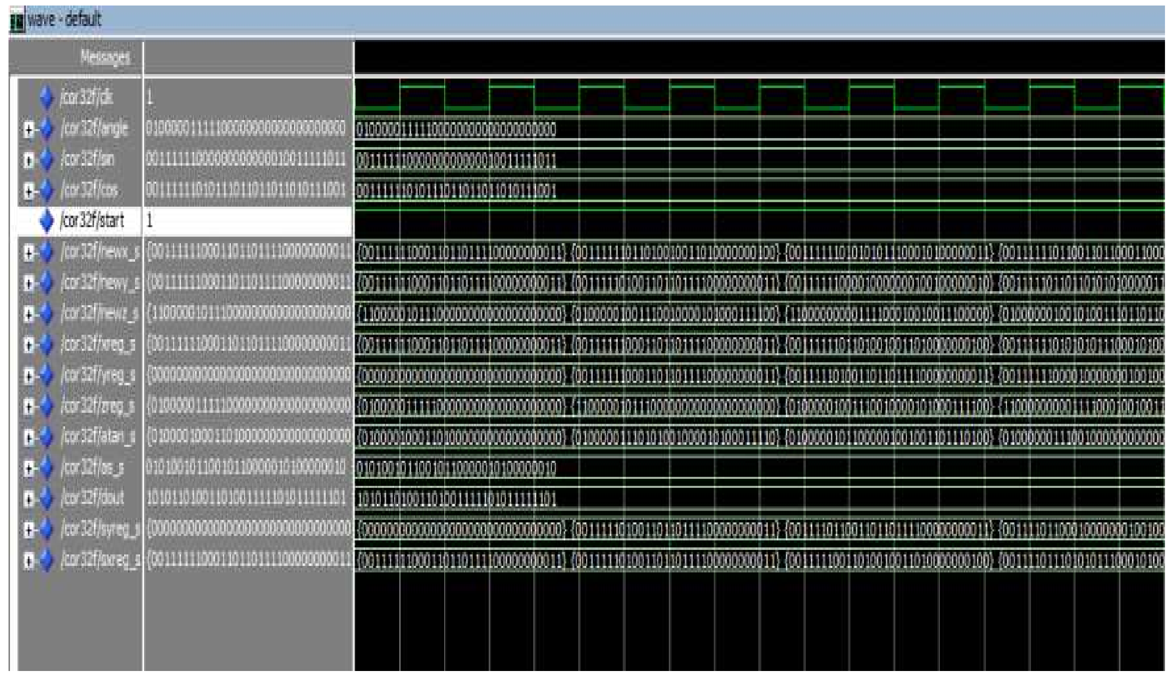


Figure 5.35 Simulation result for Sine-cosine of 32-bit floating point Angle Recoding CORDIC

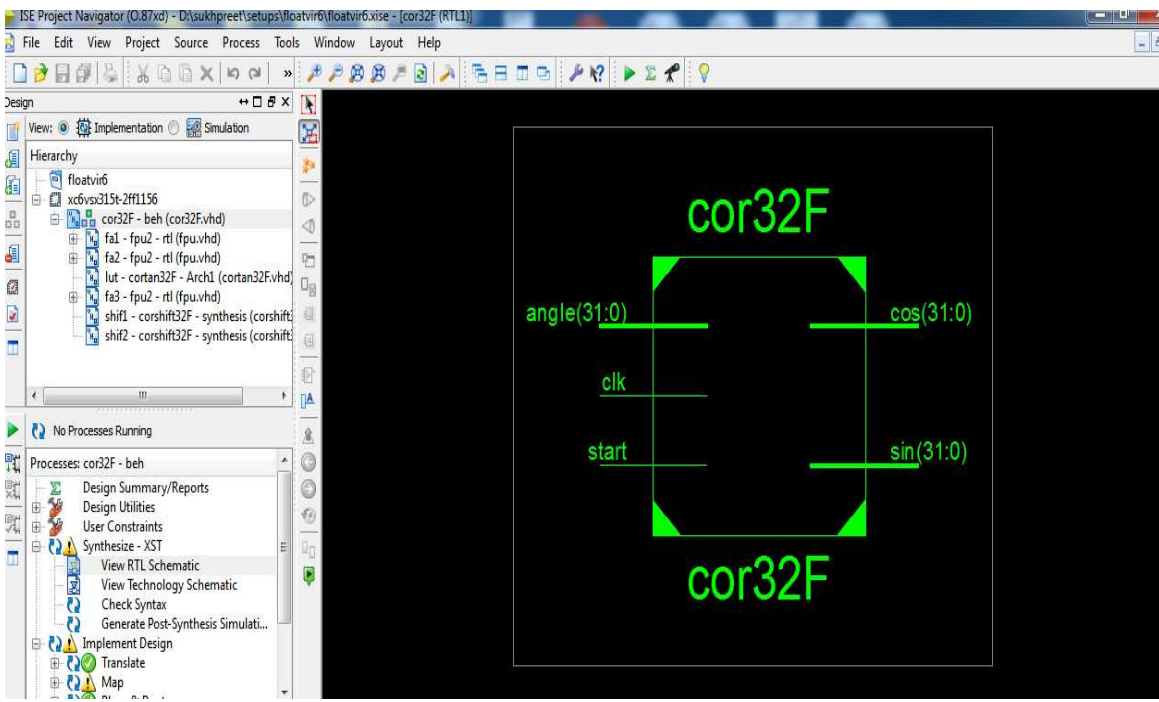


Figure 5.36 Top level schematic for 32-bit floating point CORDIC Algorithm

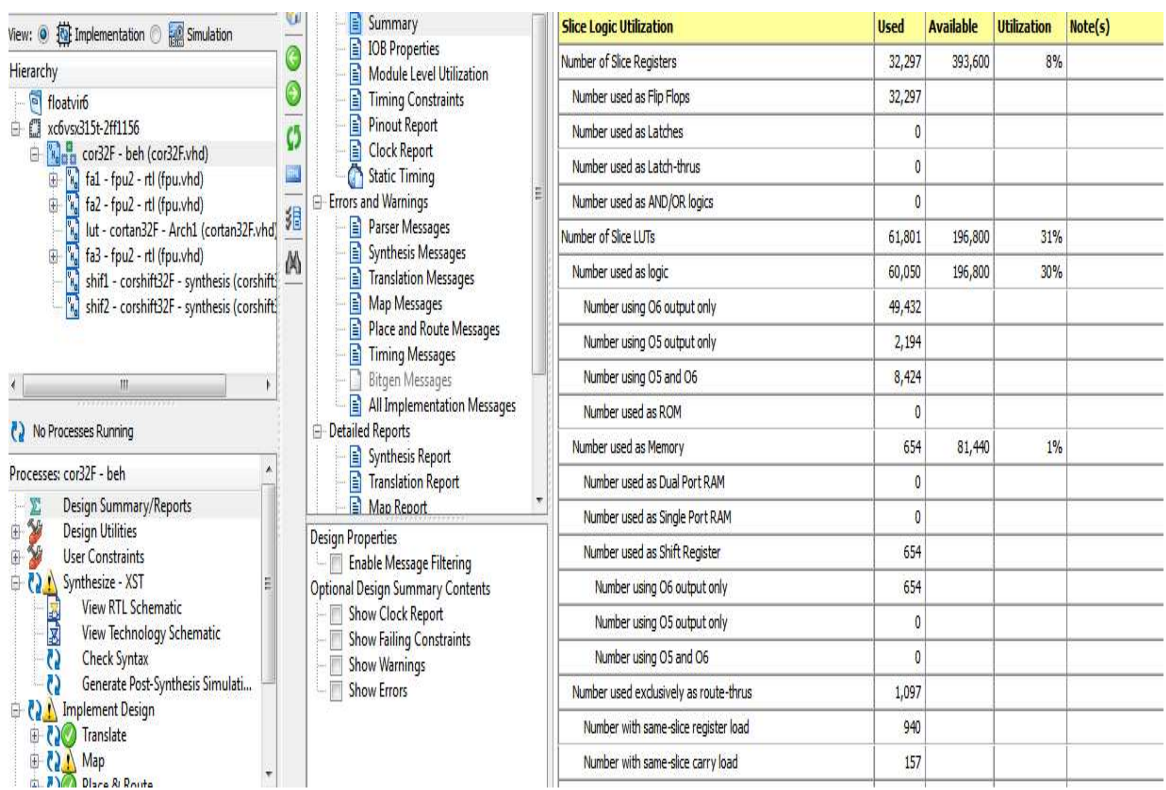


Figure 5.37 Summary report for 32-bit floating point CORDIC

Table 5.4 Comparison between 24 bit, 28 bit and 32 bit floating point CORDIC algorithm for sine-cosine computation

	16 bit		24 bit		32 bit	
	Total	utilization	Total	utilization	Total	Utilization
Number of slice Registers	16680/ 393600	4%	23014/ 393600	5%	29655/ 393600	7%
Number of bonded IOBs	60/600	10%	80/600	14%	97/600	16%
Delay	4.39ns		4.85ns		4.95ns	
Frequency	227.30MHz		206.07MHz		202.03MHz	

5.3 Discussions

No multipliers used for implementation of CORDIC algorithm for sine cosine generation. Number of adder/subtractors and registers used in the implementation of CORDIC algorithm are 3 and 3 respectively. In addition to this, 2 shifters are also used. So, CORDIC algorithm is multipierless approach and it serves a lot of hardware. But the main problem related with CORDIC algorithm is long latency. But in this thesis, various techniques for reducing the latency have been discussed and a very speed CORDIC algorithm for sine cosine generation for different numbers of bits is achieved.

CHAPTER 6

CONCLUSION & FUTURE SCOPE

6.1 Conclusion

In the present work, an idea about the performance of CORDIC algorithm for the generation of sine-cosine for fixed point numbers as well as floating point numbers is presented. The Original CORDIC, Control CORDIC and Angle Recoding CORDIC for 16-bit, 24-bit and 32-bit fixed point number have been simulated using Modelsim. The synthesis of Original CORDIC, Control CORDIC and Angle Recoding CORDIC for 16-bit, 24-bit and 32-bit fixed point number have been done on Xilinx Virtex 5. The area required has been measured in terms of slice register, slice LUTs and IOBs. These are basically synthesized to get high speed in term of frequency. Delay comes out to be 1.88ns, 2.22ns and 2.41ns in case of 16-bit, 24-bit and 32-bit fixed point CORDIC respectively. Frequency is 525.91MHz, 448.69MHz and 414.81MHz in case of 16-bit, 24-bit and 32-bit fixed point CORDIC respectively. Delay remains same for Original CORDIC, Control CORDIC and Angle Recoding CORDIC for same numbers of bits. But the iteration count is nearly half in case of Angle Recoding CORDIC as compared to Original and Control CORDIC. The percentage utilization for number of slice registers used have been found to be equal to 2%, 3% and 5% for 16-bit, 24-bit and 32-bit fixed point Original CORDIC. For Control CORDIC, percentage utilization for number of slice registers used have been found to be equal to 2%, 5% and 9% for 16-bit, 24-bit and 32-bit fixed point. For Angle Recoding CORDIC, percentage utilization for number of slice registers used have been found to be equal to 2%, 5% and 9% for 16-bit, 24-bit and 32-bit fixed point.

The synthesis of Original CORDIC for 24-bit, 28-bit and 32-bit floating point number has been done on Xilinx Virtex 6. The area required has been measured in terms of slice register, slice LUTs and IOBs. These are basically synthesized to get high speed in term of frequency. Delay comes out to be 4.39ns, 4.85ns and 4.95ns in case of 24-bit, 28-bit and 32-bit floating point CORDIC respectively. Frequency is 227.30MHz, 206.30MHz and 202.03MHz in case of 24-bit, 28-bit and 32-bit floating point CORDIC respectively. Look up tables are by the fastest way to make the computation, however the precision of the result is directly related to size of the look up table. Resolution of CORDIC algorithm

is best 32 bit single precision IEEE 754-2008 standard floating point data rates however it offers high complexity as compared to lower data rate CORDIC algorithm. It has found that 3 adders/subtractors, 3 registers and 2 shifters are required and no multiplier is used in CORDIC algorithm. Also Control CORDIC shows no overshoot as compared to Original CORDIC. Number of iteration is reduced to half in case of Angle Recoding CORDIC, but delay remains same due to increase in cycle time needed for angle selection process. Percentage improvement in speed is 2.5% for 16-bit fixed point CORDIC and 12.95% for 24-bit fixed point CORDIC as compared to reference [29]. In case of floating point, speed improvement is 57% for 24-bit, 56.6 % for 28-bit and 57.3% for 32 bit sine-cosine CORDIC algorithm as compared to reference [33]. So in thesis, a very high speed CORDIC algorithm has been synthesized.

6.2 Future Scope

Further this work is extended to increase the speed by using parallel angle recoding technique in which all angle constants are found in parallel. To get a very high precision, double precision IEEE Floating Point CORDIC algorithm can be designed. The latency of CORDIC can be further reduced by using hardwired shifters with the Adaptive CORDIC method based on Parallel Angle Recoding delay because the complex shifter contributes considerably to the cycle time of the CORDIC cycle.

REFERENCES

[1] J. Volder, "The CORDIC Trigonometric Computing Technique," IRE Transactions on Electronic Computers, vol. EC-8, no. 3, pp. 330-334, 1959.

[2] J. Walther, "A Unified Algorithm for Elementary Functions," Proceedings of Spring Joint Computer Conference, vol. 38, pp. 379-385, 1971.

[3] J. Duprat and J. Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implementation," IEEE Transactions on Computers, vol. 42, no. 2, pp. 168-178, 1993.

[4] P. Meher, J. Valls, T. Juang, K. Sridharan and K. Maharatna, "50 Years of CORDIC: Algorithms, Architectures, and Applications," IEEE Transactions on Circuits and Systems, vol. 56, no. 9, pp. 1893-1907, 2009.

[5] J. Cavallaro and F. Luk, "Architectures for a CORDIC SVD Processor," Proceedings of International Society for Optical Engineering, vol. 698, pp. 45-53, 1987.

[6] S. Abdulla, H. Nam, M. Dermot and J. Abraham, "A High Throughput FFT processor with no Multipliers," IEEE International Conference on Computer Design, no. 10, pp. 485-490, 2009.

[7] B. Parhami, Computer arithmetic, algorithms and hardware design, Oxford University Press, ed. 1st, 2000.

[8] J. Harrison, T. Kubaska, S. Story and P. Tang, "The Computation of Transcendental Functions on the IA-64 Architecture," Intel Technology Journal, Q4, 1999.

[9] J. Volder, "The Birth of CORDIC," Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology, vol. 25, pp. 101-105, June 2000.

[10] M. Ercegovac and L. Tomas, "Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD," IEEE Transactions on Computers, vol. 39, pp. 725-740, 1990.

[11] S. Wang and E. Swartzlander, "Critically Damped CORDIC Algorithm," Proceedings of the 37th Midwest Symposium on Circuits and Systems, vol. 1, pp. 236-239, 1994.

[12] Y. Hu and S. Naganathan, "An Angle Recoding Method for CORDIC Algorithm Implementation," IEEE Transactions on Computers, vol. 42, no. 1, pp. 99-102,

1993.

[13] DSP Guru, http://www.dspguru.com/info/faqs/CORDIC.htm.

[14] C. Lee and P. R. Chang, "A Maximum Pipelined CORDIC Architecture for Robot Inverse Kinematics Computation," Proceedings of the Japan-USA Symposium on Flexible Automation, Osaka, Japan, pp. 45-51, 1986.

[15] E. O. Garcia, R. Cumplido, M. Arias, "Pipelined CORDIC Design on FPGA for a Digital Sine and Cosine Waves Generator," 3rd IEEE Transactions on Computers, vol. 59, no.4, pp. 522-531, 2010.

[16] F. Jaime, M. Sánchez, J. Hormigo, J. Villalba, and L. Zapata, "Enhanced Scaling-Free CORDIC," IEEE Transactions on Circuits and System, vol. 57, no.7, pp. 1654-1662, 2010.

[17] H. Li and Y. Xin, "Modified CORDIC Algorithm and Its Implementation Based on FPGA," International Conference on Intelligent Networks and Intelligent Systems, pp. 618-621, 2010.

[18] R. Bhakthavatchalu, M. Sinith, P. Nair and K. Jismi, "A Comparison of Pipelined Parallel and Iterative CORDIC Design on FPGA," International Conference on Industrial and Information Systems, pp. 239-243, 2010.

[19] S. Aggarwal, P. Meher and K. Khare, "Area-Time Efficient Scaling-Free CORDIC Using Generalized Micro-Rotation Selection," IEEE transactions on Very Large Scale Interation Systems, vol. 20, no. 8, pp. 1542-1546, 2010.

[20] S. Wang and E. Swartzlander, "Merged CORDIC Algorithm," IEEE International Symposium on Circuits and Systems, vol. 3, pp. 1988-1991, 1995.

[21] S. Wang, V. Piuri and E. Swartzlander, "Hybrid CORDIC Algorithms," IEEE Transactions on Computers, vol. 46, pp. 1202-1207, 1997.

[22] Lin, H. Xiang and H. Sips, "On-Line CORDIC Algorithms," IEEE Transactions On Computers, vol. 39, pp. 1038-1052, 1990

[23] C. Yu, S. Chen and J. Chih, "Efficient CORDIC Designs for Multi-Mode OFDM FFT," Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, vol. 3, pp 1036-1039, 2006.

[24] E. Antelo, T. Lang and J. Bruguera, "Very-High Radix Circular CORDIC: Vectoring and Unified Rotation/Vectoring," IEEE Transactions on Computers, vol. 49, no. 7, pp 727-739, 2000.

[25] Y. Hu and S. Naganathan, "A Novel Implementation of Chirp Z-Transformation

Using a CORDIC Processor," IEEE Transactions on ASSP, vol. 38, pp. 352-354, 1990.

[26] M. Pascale, "Using CORDIC methods for Computations in micro-controllers," Dr. Dobbs The World of Software Development, [Online]. Available: www.mikrocontroller.net/attachment/31117/cordic1.pdf.

[27] W. Han, Z. Yousi, L. Xiaokang, "A Parallel Double-Step CORDIC Algorithm for Digital down Converter", 7th Annual Communications Network and Services Research Conference, no. 5, pp. 257-261, 2009.

[28] D. Phatak, "Double Step Branching CORDIC: A New Algorithm for Fast Sine and Cosine Generation," IEEE Transactions on Computers, vol. 47, no. 5, pp. 587-602, 1998.

[29] T. Rodrigues and E. Swartzlander, "Adaptive CORDIC: Using Parallel Angle Recoding to Accelerate Rotations," IEEE Transactions on Computers, vol. 59, no. 4, pp. 522-531, 2010.

[30] R. Sarmiento, F. Tobajas, V. Armas, R. Esperchain, J. Lopez, J. Montiel-Nelson and A. Nunez, "A CORDIC Processor for FFT Computation and Its Implementation Using Gallium Arsenide Technology," IEEE Transactions On Very Large Scale Integration Systems, vol. 6, no. 1, pp. 18-30, 1998.

[31] M. Garrido and J. Grjal, "Efficient memory-less CORDIC for FFT Computation," IEEE International Conference on Acoustic, Speech and Signal Processing, vol 2, no. 2, pp 113-116, 2007.

[32] A. Despain, "Fourier Transform Computers Using CORDIC Iterations," IEEE Transactions on Computers, vol. 23, pp. 993-1001, 1974.

[33] D. Munoz, D. Sanchez, C. Llanos and M. Ayala-Rincon, "FPGA based Floating Point Library for CORDIC Algorithm," Southern Conference on Programmable logic, pp. 55-60, 2010.

[34] J. Zhou, Y. Dou, Y. Lei, J. Xu and Y. Dong, "Double Precision Hybrid-Mode Floating-Point FPGA CORDIC Co-processor," International Conference on High Performance Computing and Communications, pp. 182-189, 2008

[35] R. Andraka, "A survey of CORDIC algorithm for FPGA based computers," International Symposium on Field Programmable Gate Arrays, no. 2, pp. 191-200, 1998.

[36] T. Vladimirova and H. Tiggler, "FPGA Implementation of sine and cosine

generators Using the CORDIC Algorithm," Surrey Space Center, University of Surrey, Guildford, Surrey. [Online]. Available: http://klabs.org/richcontent/MAPLDCon99/Papers/A2_Vladimirova_P.pdf.

[37] H. Hahn, B. Timmermann, and B. Rix, "A unified and division-free CORDIC argument reduction method with unlimited convergence domain including inverse hyperbolic functions," IEEE Transactions on Computers, vol. 43, pp.1339-1344, 1994.

[38] Y. Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing," IEEE Signal Processing Magazine, vol. 9, Issue 3, pp. 16-35, 1992.

[39] J. Cavallaro and F. Luk "CORDIC Arithmetic for a SVD Processor," Proceedings of 8th Symposium On Computer Archchitecture , pp. 271-290, 1987.

[40] J. G. Proakis and D. G. Manolakis, Digital signal processing principles, algorithms and applications, Delhi: Prentice Hall, ed. 2nd, 2008.

[41] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.

[42] Al-Ashrafy, M. Salem and A. Anis, "An efficient implementation of floating point multiplier," Saudi International Conference on Electronics, Communications and Photonics, pp. 1-5, 2011.

[43] B. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPGA," Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, 2002.

[44] D. Chen, J. Cong and P. Pan, "FPGA Design Automation: A Survey," Foundations and Trends in Electronic Design Automation, 2006.