A Thesis Report

On

FPGA IMPLEMENTATION OF DFT USING CORDIC ALGORITHM

Submitted in the partial fulfillment of requirement for the award of the Degree of

MASTER OF TECHNOLOGY

IN

VLSI DESIGN AND CAD

Submitted by

Vikas Kumar

60661004

Under the guidance of

Dr. Kulbir Singh

Assistant Professor

Electronics and Communication Engineering Department

Thapar University

Patiala-147004 (INDIA)

June, 2008

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"*FPGA Implementation of DFT using CORDIC algorithm*"**, in partial fulfillment of the requirements for the award of degree of **Master of Technology in VLSI Design and CAD** at **Thapar University**, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Kulbir Singh** and refers other researcher's works which are duly listed in the reference section.

The matter embodied in this thesis has not been submitted for the award of any other degree of this or any other university.

Vikas Kumar

(VIKAS KUMAR)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Kulbir Singh

Dr. KULBIR SINGH
Assistant Professor
ECED
Thapar University,
Patiala.

Dr. A. K. CHATTERJEE
Professor and Head,
Electronics & Communication Engg. Department,
Thapar University,
Patiala- 147004.

24.7.08.

Dr. R.K. SHARMA
Dean of Academic Affairs,
Thapar University
Patiala -147004

Acknowledgement

At the very outset, I wish to place on record my deep sense of gratitude and indebtedness to my worthy supervisor **Dr. Kulbir Singh (Assistant Professor)**, Electronics and communication Engineering Department at Thapar University, Patiala. His dynamism and diligent enthusiasm have been highly instrumental in keeping my spirits high. His flawless & forthright suggestions blended with an innate intelligent application have crowned my task with success.

I am highly obliged to **Prof. A.K.Chatterjee**, H.O.D Electronics and Communication Engineering Department, Thapar University, Patiala for allowing me to carry out my thesis work in this University.

I would also like to offer my sincere thanks to all faculty and staff, of Electronics & Communication Engg. Deptt. (ECED), and staff of central library, TU, Patiala for their assistance.

I am also thankful to the authors whose works I have consulted and quoted in this work. Last, but not the least, very special thanks to my parents and my friends for their constant encouragement and blessings. Their patience and understanding without which this study would not have been in this present form, is greatly appreciated.

13-07-08

DATE

Vikas Kumar

VIKAS KUMAR

(60661004)

Abstract

CORDIC is an acronym for COrdinate Rotation Digital Computer. It is a class of shift adds 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. The Jack E. Volder's CORDIC algorithm is derived from the general equations for vector rotation. The CORDIC algorithm has become a widely used approach to elementary function evaluation when the silicon area is a primary constraint. The implementation of CORDIC algorithm requires less complex hardware than the conventional method.

In digital communication, the straightforward evaluation of the cited functions is important, numerous matrix based adaptive signal processing algorithms require the solution of systems of linear equations, the computation of eigenvalues, eigenvectors or singular values. All these tasks can be efficiently implemented using processing elements performing vector rotations. The (CORDIC) offers the opportunity to calculate all the desired functions in a rather simple and elegant way. Due to the simplicity of the involved operations the CORDIC algorithm is very well suited for VLSI implementation. The rotated vector is also scaled making a scale factor correction necessary. VHDL coding and simulation of selected CORDIC algorithm for sine and cosine, the comparison of resultant implementations and the specifics of the FPGA implementation has been discussed.

In this thesis, the CORDIC algorithm has been implemented in XILINX Spartan 3E FPGA kit using VHDL and is found to be accurate. It also contains the implementation of Discrete Fourier Transform using radix-2 decimation-in-time algorithm on the same FPGA kit. Due to the high speed, low cost and greater flexibility offered by FPGAs over DSP processors the FPGA based computing is becoming the heart of all digital signal processing systems of modern era. Moreover the generation of test bench by Xilinx ISE 9.2i verifies the results.

LIST OF ACRONYMS

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

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER 1

INTRODUCTION

The CORDIC algorithm was first introduced by Jack E. Volder [1] in the year 1959 for the computation of Trigonometric functions, Multiplication, Division, Data type conversion, Square Root and Logarithms. It is a highly efficient, low-complexity, and robust technique to compute the elementary functions. The basic Algorithm structure is described in [2]. Other information about CORDIC Algorithm and different Issues are in [3]. The CORDIC algorithm has found its way in various applications such as pocket calculators, numerical co-processors, to high performance radar signal processing, supersonic bomber aircraft with a digital counterpart. Bekooij, Huisken, and Nowak research tells about the application of CORDIC in the computation of the (Fast Fourier Transform) FFT, and at the effects on the numerical accuracy.

1.1 Preamble

CORDIC stands for **CO**ordinate **R**otation **DI**gital **C**omputer. It calculates the value of trigonometric functions like sine, cosine, magnitude and phase (arctangent) to any desired precision. It can also calculate hyperbolic functions (such as sinh, cosh, and tanh). The CORDIC algorithm does not use Calculus based methods such as polynomial or rational function approximation. It is used as approximation function values on all popular graphic calculators, including HP-48G as the hardware restriction of calculators require that the elementary functions should be computed using only additions, subtractions, digit shifts, comparisons and stored constants. Today cordic algorithm is used in Neural Network VLSI design [4], high performance vector rotation DSP applications [5], advanced circuit design, optimized low power design. CORDIC algorithm 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 thus it simpler and do not require complex hardware structure as in the case of multiplier. Earlier methods used are Table look up method, Polynomial approximation method etc. for evaluation of trigonometric functions. It is hardware efficient algorithm. No multiplier requirement

as in the case of microcontroller. The drawback in CORDIC is that after completion of each iteration, there is a gain which is added to the magnitude of resulting vector which can be removed by multiplying the resulting magnitude with the inverse of the gain. There are two ways in CORDIC algorithm for calculation of trigonometric and other related functions they are rotation mode and vectoring mode. Both methods initialize the angle accumulator with the desired angle value. The rotation mode, determines the right sequence as the angle accumulator approaches zero while the vectoring mode minimizes the y component of the input vector. CORDIC is generally faster than other approaches when a hardware multiplier is unavailable (e.g. in a microcontroller), or when the number of gates required to implement is to be minimized (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 power series are generally faster than CORDIC. Various CORDIC architectures like bit parallel iterative CORDIC, a bit parallel unrolled CORDIC, a bit-serial iterative CORDIC and the comparison of various CORDIC architecture has been discussed. It can be seen that CORDIC is a feasible way to approximate cosine and sine. CORDIC is useful in designing computing devices. As it was originally designed for hardware applications, there are features that make CORDIC an excellent choice for small computing devices. Since it is an iterative method it has the advantage over the other methods of being able to get better accuracy by doing more iteration, where as the Taylor approximation and the Polynomial interpolation methods need to be rederived to get better results. These properties, in addition to getting a very accurate approximation is perhaps the reason why CORDIC is used in many scientific calculators today. Due to the simplicity of the involved operations the CORDIC algorithm is very well suited for VLSI implementation. However, the CORDIC iteration is not a perfect rotation which would involve multiplications with sine and cosine. The rotated vector is also scaled making a scale factor correction necessary.

1.2 Historical perspective

CORDIC algorithm has found its way in many applications. The CORDIC was introduced in 1956 by Jack Volder as a highly efficient, low-complexity, and robust technique to compute the elementary functions. It is initially intended for navigation technology, the CORDIC algorithm has found its way in a wide range of applications, ranging from pocket calculators, numerical co-processors, to high performance radar

signal processing. After invention CORDIC worked as the replacement for the analog navigation computers aboard the B-58 supersonic bomber aircraft with a digital counterpart. 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. Further Steve Walther [6] continues work on CORDIC, with the application of the CORDIC algorithm in the Hewlett-Packard calculators, such as the HP-9100 and the famous HP-35 in year 1972, the HP-41C in year1980. He told how the unified CORDIC algorithm i.e. combining rotations in the circular, hyperbolic, and linear coordinate systems and how it was applied in the HP-2116 floating-point numerical co-processor. Today's fast rotation techniques are closely related to CORDIC, to perform orthonormal rotation at a very low cost. Although fast rotations exist for certain angles only, they are sufficiently versatile, and have already been widely applied in signal processing. Hekstra found a large range of known, and previously unknown, fast rotation methods. An overall evaluation of the methods exposes the trade-offs that exist between the angle of rotation, the accuracy in scaling and the cost of rotation. Van der Kolk, Deprettere, and Lee [7] formalized the problem of (approximate) vectoring for fast rotations in year 2000. They treated the fast and efficient selection of the appropriate fast rotation, and showed the advantage to be gained when applied to Enhanced Versatile Disc (EVD). The selection technique works equally well for redundant arithmetic and floating-point computations. Antelo, Lang, and Bruguera[8] considers going to a higher radix than the radix-2 for the classical algorithm, so that less iterations are required. The choice of a higher radix implies that the scaling factor is no longer constant. The authors propose an on-line calculation of the logarithm of the scale factor and subsequent compensation. Hsiao, Lau, and Delosme [9] considered multi-dimensional variants of CORDIC, such as the 4-D (dimention) householder CORDIC transform, and their application to singular value deposition (SVD). Rather than building a multi-dimensional transform out of a sequence of 2-D (dimention) CORDIC operations, they proposed to work with multi-dimensional micro-rotations, immediately at the iteration level. Their method is evaluated and benchmarked against solutions by others. Kwak, Choi, and Swartzlander [10] aimed to overcome the critical path in the iteration through sign prediction and addition. They proposed to overlap the sign prediction with the addition, by computing the results for both outcomes of the sign, and to select the proper one at the very end of

the iteration. Novel in their approach is to combine the adder logic for the computation of both results.

1.3 Thesis objective

Based on the above discussion the thesis has following objectives:

- To study and implement CORDIC algorithm using VHDL programming code.
- To implement DXT using CORDIC algorithm in VHDL code.
- To implement CORDIC algorithm on XILINX SPARTAN 3E kit.

1.4 Organization of thesis

Chapter 2 discusses basics of CORDIC algorithm, how it came into picture, its basic equations, different mode of operation i.e. rotation mode and vectoring mode, gain factor, complex form of representation, CORDIC iteration and how it works. Chapter 3 discusses about the calculation of sine-cosine using CORDIC algorithm, different architectures to perform CORDIC iteration and their block diagram, their comparison on the basis of their complexity, speed, area required to implement in chip designing, number of iteration required etc. Chapter 4 discusses use of CORDIC algorithm for calculating DFT, DHT, calculation of DFT using FFT algorithm, basic equations used. Chapter 5 tells about the design flow of XILINX FPGA. This chapter includes FPGA architecture, its logic blocks and different families of FPGA, their specification, technology used, placement and routing, testing and design issues. Chapter 6 contains the results of simulation using ModelSim and XILINX. The thesis concludes in chapter 7 which also discusses future scope of work.

1.5 Methodology

In this thesis, VHDL programming has been used to implement CORDIC algorithm (to calculate Sine and Cosine value for a given angle) and DFT (Digital Fourier Transform) and DHT (Discrete Fourier Transform). Further XILINX

SPARTAN 3E kit is used for FPGA implementation of the generated VHDL code. Programming tools used for the implementations are :

- Operating system WINDOWS XP
- ModelSim SE PLUS 5.5c
- XILINX 9.2i
- FPGA kit SPARTAN 3E

CHAPTER 2

CORDIC ALGORITHM

In 1959 Jack E. Volder [1] described the Coordinate Rotation Digital Computer or CORDIC for the calculation of trigonometric functions, multiplication, division and conversion between binary and mixed radix number systems. The CORDIC-algorithm provides an iterative method of performing vector rotations by arbitrary angles using only shift and add. In this chapter, it is described that how CORDIC algorithm works and how it can be understood more clearly.

2.1 Basic equation of CORDIC algorithm

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

$$X = r\cos\theta,\ Y = r\sin\theta \tag{2.1}$$

$$V' = \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x.\cos(\phi) - y.\sin(\phi) \\ y.\cos(\phi) + x.\sin(\phi) \end{pmatrix} \tag{2.2}$$

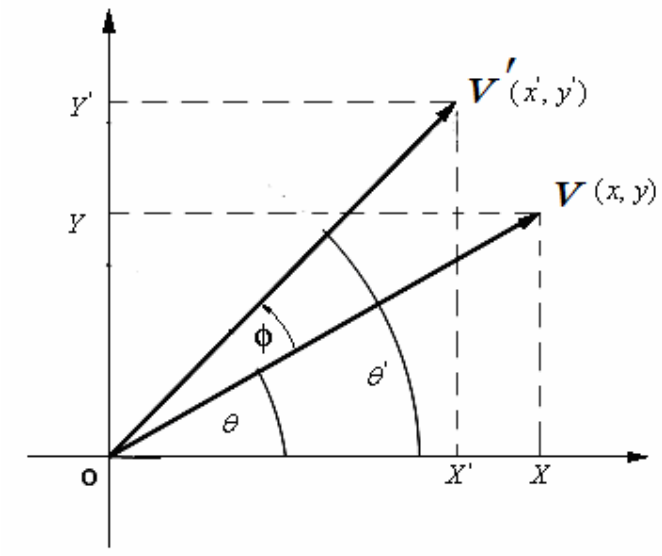


Figure 2.1: Rotation of a vector V by the angle ϕ

Let's find how equation 2.1 and 2.2 came into picture. As shown in the figure 2.1, 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 2.2 illustrates the rotation of a vector $V = \begin{pmatrix} x \\ y \end{pmatrix}$ by the angle ϕ.

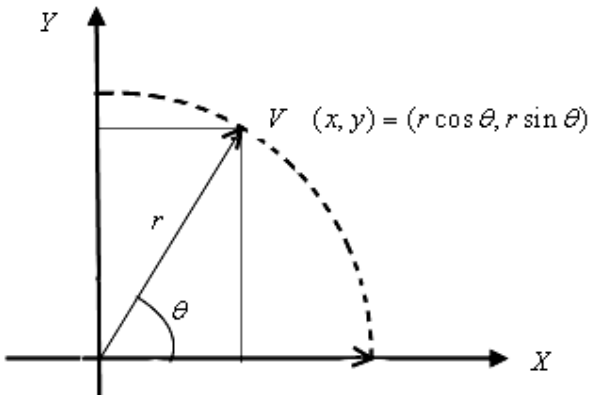


Figure 2.2: Vector V with magnitude r and phase θ

i.e. $$\left.\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned}\right\}. \tag{2.3}$$

Similarly from figure 2.1 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 ϕ. From figure 2.1 it can be observed

$$\theta' - \theta = \phi \tag{2.4}$$

i. e. $$\theta' = \theta + \phi \tag{2.5}$$

$$\begin{aligned} OX' = x' &= r\cos\theta' \\ &= r\cos(\theta + \phi) \\ &= r(\cos\theta.\cos\phi - \sin\theta.\sin\phi) \\ &= (r.\cos\theta)\cos\phi - (r.\sin\theta)\sin\phi \end{aligned} \tag{2.6}$$

Using figure 2.2 and equation 2.3 OX' can be represented as

$$OX' = x' = x\cos\phi - y\sin\phi \tag{2.7}$$

Similarly, OY'

$$OY' = y' = y\cos\phi + x\sin\phi \tag{2.8}$$

Similarly, value for the vector V' in the clockwise direction rotating the vector V by the angle ϕ and the equations obtain in this case be

$$x' = x\cos\phi + y\sin\phi \tag{2.9}$$

$$y' = x\sin\phi - y\cos\phi \tag{2.10}$$

The equations (2.7), (2.8), (2.9), (2.10) can be represented in the matrix form as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \begin{pmatrix} \cos\phi & \mp\sin\phi \\ \pm\sin\phi & \cos\phi \end{pmatrix} \tag{2.11}$$

The individual equations for x' and y' can be rewritten as [11]:

$$x' = x.\cos(\phi) \mp y.\sin(\phi) \tag{2.12}$$

$$y' = y.\cos(\phi) \pm x.\sin(\phi) \tag{2.13}$$

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

$$x' = \cos\phi(x \mp y\tan\phi) \tag{2.14}$$

$$y' = \cos\phi(y \pm x\tan\phi) \tag{2.15}$$

$z' = z \pm \phi$, here ϕ is the angle of rotation ($\pm$ sign is showing the direction of rotation) and z is the argument. For the ease of calculation here only rotation in anticlockwise direction is observed first. Rearranging the equation (2.7) and (2.8).

$$x' = \cos(\phi)[x - y.\tan(\phi)] \tag{2.16}$$

$$y' = \cos(\phi)[y + x.\tan(\phi)] \tag{2.17}$$

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

$$x_{i+1} = k_i[x_i - y_i.d_i.2^{-i}] \tag{2.18}$$

$$y_{i+1} = k_i[y_i + x_i.d_i.2^{-i}] \tag{2.19}$$

where, i denotes the number of rotation required to reach the required angle of the required vector, $K_i = \cos(\arctan(2^{-i}))$ and $d_i = \pm 1$. The product of the K_i's represents the so-called K factor [6]:

$$k = \prod_{i=0}^{n-1} k_i \tag{2.20}$$

where $\prod_{i=o}^{n-1} k_i = \cos\phi_0 \cos\phi_1 \cos\phi_2 \cos\phi_3 \cos\phi_4 ..... \cos\phi_{n-1}$ (ϕ is the angle of rotation here for n times rotation). The above rotations requirement and adding and subtracting of the different ϕ can be understood by the following example of a balance.

Table 2.1: For 8-bit CORDIC hardware.

i	$d^{-i} = 2^{-i} = \tan\phi_i$	$\phi_i = \arctan(2^{-i})$	ϕ_i in radian
0	1	45^o	0.7854
1	0.5	26.565^o	0.4636
2	0.25	14.036^o	0.2450
3	0.125	7.125^o	0.1244
4	0.0625	3.576^o	0.0624
5	0.03125	1.7876^o	0.0312
6	0.015625	0.8938^o	0.0156
7	0.0078125	0.4469^o	0.0078

k_i is the gain and it's value changes as the number of iteration increases. For 8-bit hardware CORDIC approximation method the value of k_i as

$$\begin{aligned} k_i = \prod_{i=0}^{7} \cos\phi_i &= \cos\phi_0 . \cos\phi_1 . \cos\phi_2 . \cos\phi_3 . \cos\phi_4 . \cos\phi_5 . \cos\phi_6 . \cos\phi_7 \\ &= \cos 45^o . \cos 26.565^o ..... \cos 0.4469^o \\ &= 0.6073 \end{aligned} \tag{2.21}$$

From the above table it can be seen that precision up to 0.4469^o is possible for 8-bit CORDIC hardware. These ϕ_i are stored in the ROM of the hard ware of the CORDIC

hardware as the look up table. Now by taking an example of balance it can be understood that how the CORDIC algorithm works.

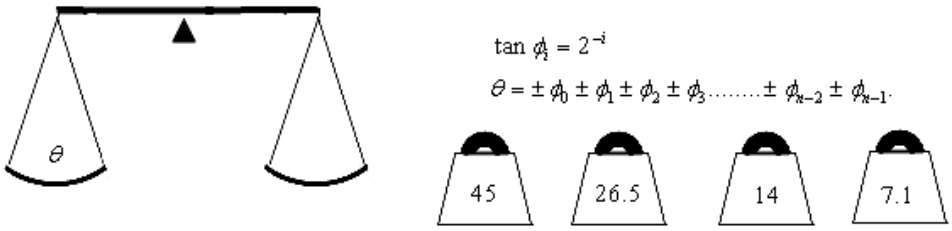


Figure 2.3: A balance having θ at one side and small weights (angle) at the other side.

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

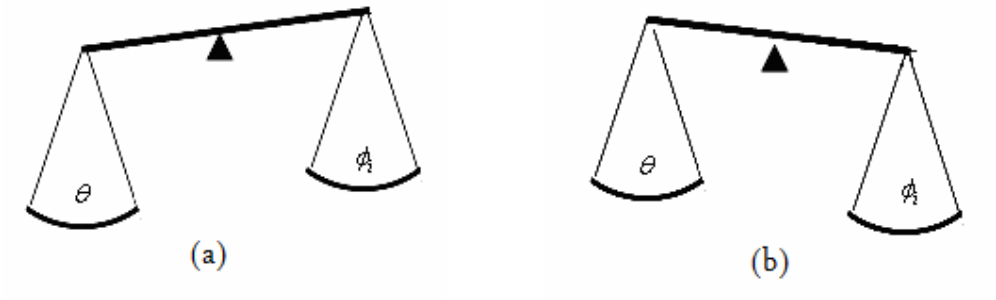


Figure 2.4: Inclined balance due to the difference in weight of two sides.

Then, if balance shows a left inclination as in figure 2.4 (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 shows a right inclination as in figure 2.4 (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.

Matrix representation of the CORDIC algorithm for 8-bit hardware:

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} \cos\phi_i & \mp\sin\phi_i \\ \pm\sin\phi_i & \cos\phi_i \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \tag{2.22}$$

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} \cos\phi_0 & \mp\sin\phi_0 \\ \pm\sin\phi_0 & \cos\phi_0 \end{pmatrix} \begin{pmatrix} \cos\phi_1 & \mp\sin\phi_1 \\ \pm\sin\phi_1 & \cos\phi_1 \end{pmatrix} \ldots\ldots\ldots$$

$$\ldots\ldots\begin{pmatrix} \cos\phi_7 & \mp\sin\phi_7 \\ \pm\sin\phi_7 & \cos\phi_7 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.23}$$

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \cos\phi_0 \cos\phi_1 \ldots\ldots\ldots \cos\phi_7 \begin{pmatrix} 1 & \mp\tan\phi_0 \\ \pm\tan\phi_0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \mp\tan\phi_1 \\ \pm\tan\phi_1 & 1 \end{pmatrix}$$

$$\ldots\ldots\ldots\ldots\ldots\begin{pmatrix} 1 & \mp\tan\phi_7 \\ \pm\tan\phi_7 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

(2.24)

Thus, Scale Factor = $\cos\phi_0 \cos\phi_1 \ldots\ldots\ldots\ldots \cos\phi_7$

$$= 0.6073$$

$$= \frac{1}{1.6466} \tag{2.25}$$

It can be seen from equation (2.22) that cosine and sine of an angle θ can be represented in the matrix form as

$$\begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} = \begin{pmatrix} 1 & \mp\tan\phi_0 \\ \pm\tan\phi_0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \mp\tan\phi_1 \\ \pm\tan\phi_1 & 1 \end{pmatrix} \ldots\ldots\ldots \begin{pmatrix} 1 & \mp\tan\phi_7 \\ \pm\tan\phi_7 & 1 \end{pmatrix} \begin{pmatrix} 0.6073 \\ 0 \end{pmatrix} \tag{2.26}$$

2.2 Complex number representation of CORDIC algorithm

Let a given complex number B having its real and imaginary part as, I_b and Q_b respectively.

Given complex number, $B = I_b + jQ_b$ (2.27)

Let rotated complex number is, $B' = I_{b'} + jQ_{b'}$ (2.28)

It results due to multiplication of a rotation value *r* where *r* is $R = I_r + jQ_r$ when a pair of complex numbers is multiplied, their phases (angles) adds and their magnitudes multiply. Similarly, when one complex number is multiplied by the

conjugate of the other, the phase of the conjugated is subtracted (though the magnitudes still multiply).

Therefore to add R's phase to *B*, $B' = B.R$ i. e.

$$I_{b'} = I_b.I_r - Q_b.Q_r \text{ and } Q_{b'} = Q_b.I_r + I_b.Q_r \quad (2.29)$$

To subtract R's phase from *B*, $B' = B.\bar{R}$ where $\bar{R}$ is the conjugate complex of *R*

$$I_{b'} = I_b.I_r + Q_b.Q_r \quad \text{and} \quad Q_{b'} = Q_b.I_r - I_b.Q_r \quad (2.30)$$

Thus to rotate by +90 degrees, multiply by R = 0 + j1. Similarly, to rotate by -90 degrees, multiply by R = 0 - j1. Going through the above process, the net effect is: To add 90 degrees, multiply by $R = 0 + j1$: results in $I_{b'} = -Q_b$ and $Q_{b'} = I_b$.To subtract 90 degrees, multiply by $R = 0 - j1$: results in $I_{b'} = Q_b$ and $Q_{b'} = -I_b$ .To rotate by phases of less than 90 degrees, given complex number is multiplied by numbers of the form "$R = 1 +/- jK$" where K will be decreasing powers of two, starting with 2^0 = 1.0. Therefore, K = 1.0, 0.5, 0.25, etc. (here the symbol "i" is used to designate the power of two itself: 0, -1, -2, etc.). Since the phase of a complex number "$I + jQ$" is $\arctan(Q/I)$, the phase of "$1 + jK$" is $\arctan(K)$. Likewise, the phase of "$1 - jK$" is $\arctan(-K) = -\arctan(K)$.To *add* phases "$R = 1 + jK$" is used; to subtract phases "$R = 1 - jK$" is used. Since the real part of this, I_r, is equal to 1, we can simplify our table of equations to add and subtract phases for the special case of CORDIC multiplications to:

To add a phase, multiply by $R = 1 + jK$:

$$I_{b'} = I_b - K.Q_b = I_b - (2^{-i}).Q_b \quad (2.31)$$

$$Q_{b'} = Q_b + K.I_b = Q_b + (2^{-i}).I_b \quad (2.32)$$

To subtract a phase, multiply by $R = 1 - jK$:

$$I_{b'} = I_b + K.Q_b = I_b + (2^{-i}).Q_b \quad (2.33)$$

$$Q_{b'} = Q_b - K.I_b = Q_b - (2^{-i}).I_b \quad (2.34)$$

Let's look at the phases and magnitudes of each of these multiplier values to get more of a feel for it. The table below lists values of L, starting with 0, and shows the corresponding values of K, phase, magnitude, and CORDIC Gain (in table 2.2). Since CORDIC uses powers of 2 for the *K* values, multiplication and division can be done only by shifting and adding binary numbers. That's why the CORDIC algorithm

doesn't need any multiplies. Also it can be seen that starting with a phase of 45 degrees, the phase of each successive *R* multiplier is a little over half of the phase of the previous *R*. That's the key to understanding CORDIC: here what is done that a "binary search" on phase by adding or subtracting successively smaller phases to reach some "target" phase.

Table 2.2: Phase, magnitude, and CORDIC Gain for different values of *K*

i	$K = 2^{-i}$	$R = 1 + jK$	$\angle R$ (in degree) $=\tan^{-1}(K)$	$Mag.R$	CORDIC gain
0	1.0	1 + j1.0	45.00000	1.41421356	1.414213562
1	0.5	1 + j0.5	26.56505	1.11803399	1.581138830
2	0.25	1 + j0.25	14.03624	1.03077641	1.629800601
3	0.125	1 + j0.125	7.12502	1.00778222	1.642484066
4	0.0625	1 + j0.0625	3.57633	1.00195122	1.645688916
5	0.03125	1 + j0.031250	1.78991	1.00048816	1.646492279
6	0.015625	1 + j0.015625	0.89517	1.00012206	1.646693254

7	0.007813	1 + j0.007813	0.44761	1.00003052	1.646743507
... .	...	...	...	...	...

The sum of the phases in the table up to L = 3 exceeds 92 degrees, so a complex number can be rotated by +/- 90 degrees by doing four or more "$R = 1 +/- jK$" rotations. Putting that together with the ability to rotate +/-90 degrees using "$R = 0 +/- j1$", and it can be rotated a full +/-180 degrees. Each rotation has a magnitude greater than 1.0. That isn't desirable, but it's the price which is to pay for using rotations of the form "1 + jK ". The "CORDIC Gain" column in the table is simply a "cumulative magnitude" calculated by multiplying the current magnitude by the previous magnitude. Noticing that it converges to about 1.647; however, the actual CORDIC Gain depends on how many iterations have been done. (It doesn't depend on whether the phases are being added or subtracted, because the magnitudes multiply either way.)

2.2.1 Calculation of magnitude of complex number

The magnitude of a complex number $B = I_b + jQ_b$ can be calculated if it is rotated to have a phase of zero; then its new Q_b value would be zero, so the magnitude would be given entirely by the new I_b value.

- It can be determined whether or not the complex number "B" has a positive phase just by looking at the sign of the "Q_b" value: positive Q_b means positive phase. As the very first step, if the phase is positive, rotate it by -90 degrees; if it's negative, rotate it by +90 degrees. To rotate by +90 degrees, just negate Q_b, then swap I_b and Q_b; to rotate by -90 degrees, just negate I_b, then swap. The phase of B is now less than +/- 90 degrees, so the "$1 +/- jK$" rotations to follow can rotate it to zero.
- Next, do a series of iterations with successively smaller values of *K*, starting with *K=1* (45 degrees). For each iteration, simply look at the sign of Q_b to decide

whether to add or subtract phase; if Q_b is negative, add a phase (multiplying by "$1 + jK$"); if Q_b is positive, subtract a phase (multiplying by "$1 - jK$"). The accuracy of the result converges with each iteration, as the more iteration is done, the more accurate its results.

Now, having rotated the complex number to have a phase of zero, it end up with "$B = I_b + j0$". The magnitude of this complex value is just "I_b" (since "Q_b" is zero.) However, in the rotation process, *B* has been multiplied by a CORDIC Gain (cumulative magnitude) of about 1.647. Therefore, to get the *true* value of magnitude it must be multiplied by the reciprocal of 1.647, which is 0.607. (The exact CORDIC Gain is a function of the how much iteration is done.) Unfortunately, gain-adjustment multiplication can't be done using a simple shift/add; however, in many applications this factor can be compensated in some other part of the system. Or, when "relative magnitude" is all that counts (e.g. AM demodulation), it can simply be neglected.

2.2.2 Calculation of phase of complex number

For calculation of phase, complex number is rotated to have zero phase, as done previously to calculate the magnitude.

- For each phase-addition/subtraction step, accumulating the actual number of degrees (or radians) for which it is rotated. The "actual" will come from a table of "$\arctan(K)$" values like the "Phase of *R*" column in the table 2.2. The phase of the complex input value is the negative of the accumulated rotation required to bring it to a phase of zero.

2.2.3 Calculation of sine and cosine of an angle

- Starting with a unity-magnitude value of $B = I_b + jQ_b$. The exact value depends on the given phase. For angles greater than +90, B should be started with $B = 0 + j1$ (that is, +90 degrees); for angles less than -90, B should be started with $B = 0 - j1$ (that is, -90 degrees); for other angles, B should be started with $B = 1 + j0$ (that is, zero degrees). Initializing an "accumulated rotation" variable to +90, -90, or 0 accordingly. (Of course, it should be done in terms of radians).

- Doing a series of iterations. If the desired phase minus the accumulated rotation is less than zero then add the next angle in the table; otherwise, subtract the next angle. Doing this using each value in the table.
- The "cosine" output is in "I_b"; the "sine" output is in "Q_b".

2.3 Basic CORDIC iterations

To simplify each rotation, picking α_i (angle of rotation in *i*th iteration) such that $\alpha_i = d_i . 2^{-i} . d_i$ is such that it has value +1 or -1 depending upon the rotation i. e. $d_i \in \{+1,-1\}$ .Then

$$x_{i+1} = x_i - d_i y_i 2^{-i} \tag{2.35}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \tag{2.36}$$

$$z_{i+1} = z_i - d_i \tan^{-1} 2^{-i} \tag{2.37}$$

The computation of x_{i+1} or y_{i+1} requires an i-bit right shift and an add/subtract. If the function $\tan^{-1} 2^{-i}$ is pre computed and stored in table (Table 3.1) for different values 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.

If the rotation is done by the same set of angles (with + or- signs), then the expansion factor K, is a constant, and can be pre computed. For example to rotate by 30 degrees, the following sequence of angles be followed that add up to $\approx$30 degree.

$$30.0 \approx 45.0 - 26.6 + 14.0 - 7.1 + 3.6 + 1.8 - 0.9 + 0.4 - 0.2 + 0.1$$
$$= 30.1$$

In effect, what actually happens in CORDIC is that z is initialized to 30 degree and then, in each step, the sign of the next rotation angle is selected to try to change

the sign of z; that is, d_i=sign (z_i) is chosen, where the sign function is defined to be -1 or 1 depending on whether the argument is negative or nonnegative. This is reminiscent of no restoring division. Table3.2 shows the process of selecting the signs of the rotation angles for a desired rotation of +30 degree. Figure 3.1 depicts the first few steps in the process of forcing z to zero.

Table 2.3: Approximate value of the function α_i = arctan (2^{-i}), in degree, for $0 \le i \le 9$.

i	α_i
0	45
1	26.6
2	14
3	7.1
4	3.6
5	1.8
6	0.9
7	0.4
8	0.2
9	0.1

In CORDIC terminology the preceding selection rule for d_i, which makes z converge to zero, is known as rotation mode. Rewriting the CORDIC iteration, where $\alpha_i = \tan^{-1} 2^{-i}$:

$$x_{i+1} = x_i - d_i y_i 2^{-i} \tag{2.38}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \tag{2.39}$$

$$z_{i+1} = z_i - d_i \alpha^i \tag{2.40}$$

After m iteration in rotation mode, when z (m) is sufficiently close to zero. we have $\sum \alpha_i = z$, and the CORDIC equations become:

$$x_m = k(x \cos z - y \sin z) \tag{2.40}$$

$$y_m = k(y \cos z + x \sin z) \tag{2.41}$$

$$z_m = 0 \tag{2.42}$$

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

The constant K in the preceding equation is k = 1.646760258121…… Thus, to compute cos z and sin z, one can start with x = 1/K = 0.607252935….. and y = 0. then, as z_m tends to 0 with CORDIC iterations in rotation mode, x_m and y_m converge to cos z and sin z, respectively. Once sin z and cos z are known, tan z can be through necessary division.

Table 2.4: Choosing the signs of the rotation angles to force z to zero

i	$z_i - \alpha_i$	z_{i+1}
0	+ 30.0 – 45.0	-15
1	- 15.0 + 26.6	11.6
2	+ 11.6 - 14.0	-2.4
3	- 2.4 + 7.1	4.7
4	+ 4.7 – 3.6	1.1
5	+ 1.1 – 1.8	-0.7
6	- 0.7 + 0.9	0.2
7	+ 0.2 – 0.4	-0.2
8	- 0.2 + 0.2	0
9	+ 0.0 – 0.1	-0.1

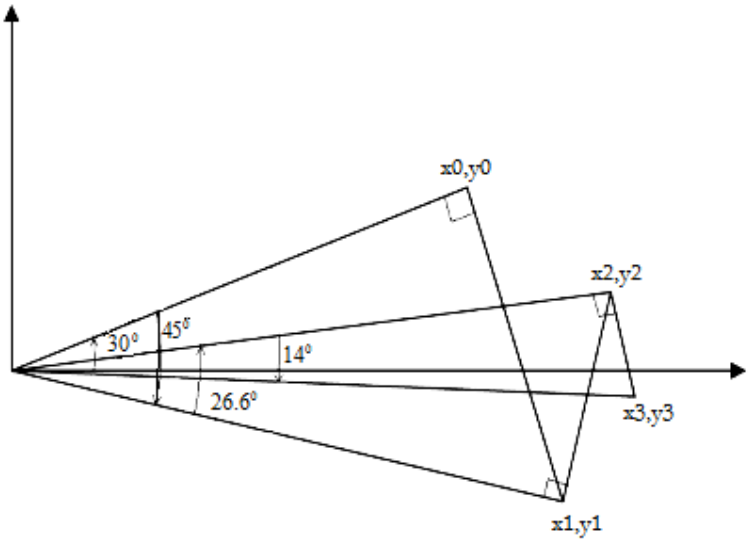


Figure 2.5: First three of 10 iteration leading from $(x_0,\ y_0)$ to $(x_{10}, 0)$ in rotating by $+30^0$, Rotation mode.

For k bits of precision in the resulting trigonometric functions, k CORDIC iterations are needed. The reason is that for large *i* it can be approximated that $\tan^{-i} 2^{-i} \approx 2^{-i}$. Hence, for *i* > *k*, the change in the z will be less than ulp (Unit in the Last Place).

In the rotation mode, convergence of z to zero is possible because each angle in table 3.1 is more than half the previous angle or, equivalently, each angle is less than the sum of the entire angle following it. The domain of convergence is -99.7 < z <99.7, where99.7 is the sum of all the angles in table 3.1. Fortunately, this range includes angle from -90 to +90, or $[-\frac{\pi}{2} to +\frac{\pi}{2}]$ in radians. For outside the preceding range, trigonometric identities can be converted to the problem, to one that is within the domain of convergence:

$$\cos(z \pm 2j\pi) = \cos z \tag{2.43}$$

$$\sin(z \pm 2j\pi) = \sin z \tag{2.44}$$

$$\cos(z - \pi) = -\cos z \tag{2.45}$$

$$\sin(z - \pi) = -\sin z \tag{2.46}$$

Noting that these transformations become particularly convenient if angles are represented and manipulated in multiples of π radians, so that z = 0.2 really means z = 0.2π radian or converted to numbers within the domain quite easily.

In the second way of utilizing CORDIC iterations, known as "vectoring mode," y is made nearer to zero by choosing $d_i = -sign(x_i y_i)$. After m iterations in vectoring mode $\tan(\sum \alpha_i) = -\frac{y}{x}$.

This means that:

$$x_m = k[x\cos(\sum \alpha_i) - y\sin(\sum \alpha_i)] \tag{2.47}$$

$$x_m = k(x - y\tan(\sum \alpha_i))/[1+\tan^2(\sum \alpha_i)]^{1/2} \tag{2.48}$$

$$x_m = k(x + y^2/x)/(1+y^2/x^2) \tag{2.49}$$

$$x_m = k(x^2+y^2)^{1/2} \tag{2.50}$$

The CORDIC equations thus become

$$x_m = k(x^2+y^2)^{1/2} \tag{2.51}$$

$$y_m = 0 \tag{2.52}$$

$$z_m = z + \tan^{-1}(y/x) \tag{2.53}$$

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

$$\tan^{-1} y = -\left(\frac{\pi}{2} - \tan^{-1} y\right) \tag{2.54}$$

to limit the range of fixed point numbers that is encountered. CORDIC method also allows the computation of the other inverse trigonometric functions.

CHAPTER 3

COMPUTATION OF SINE COSINE

Elementary functions, especially trigonometric functions, play important roles in various digital systems, such as graphic systems, automatic control systems, and so on. The CORDIC (Coordinate Rotation DIgital Computer) [11], [12] is known as an efficient method for the computation of these elementary functions. Recent advances in VLSI technologies make it attractive to develop special purpose hardware such as elementary function generators. Several function generators based on the CORDIC have been developed [13]. The CORDIC can also be applied to matrix triangularization, singular value decomposition, and so on [14], [6]. In this chapter, different hardware are dealt for sine and cosine computation using CORDIC. In sine and cosine computation by the CORDIC, iterative rotations of a point around the origin on the X-Y plane are considered. In each rotation, the coordinates of the rotated point and the remaining angle to be rotated are calculated. The calculations in each iteration step are performed by shift, addition and subtraction, and recall of a prepared constant. Since the rotation is not a pure rotation but a rotation-extension, the number of rotations for each angle should be a constant independent of the operand so that the scale factor becomes a constant. When implementing a sine and cosine calculator in digital hardware, the expense of the multiplication needed for many algebraically methods, should be kept in mind. Alternative techniques are based on polynomial approximation, table-lookup [15] etc. as well as shift and add algorithms [15]. Among the various properties that are desirable, we can cite speed, accuracy or the reasonable amount of resource [15]. The architecture of FPGAs specifies suitable techniques or might even change desirable properties. Because the number of sequential cells and amount of storage area, needed for table-lookup algorithms, are limited but combinational logic in terms of LUT (Look Up Table) in the FPGA's (Field Programmable Gate Array) CLBs (Configurable Logic Blocks) is sufficiently available, shift and add algorithms fit perfectly into an FPGA.

3.1 CORDIC Hardware

A straight forward hardware implementation for CORDIC arithmetic is shown below in figure 3.2. It requires three registers for x, y and z, a look up table to store the values of $\alpha_i = \tan^{-1} 2^{-i}$, and two shifter to supply the terms $2^{-i}x$ and $2^{-i}y$ to the adder/subs tractor units. The d_i factor (-1 and 1) is accommodated by selecting the (shift) operand or its complement.

Of, course a single adder and one shifter can be shared by three computations if a reduction in speed by a factor of 3 is acceptable. In the extreme, CORDIC iterations can be implemented in firmware (micro program) or even software using the ALU and general purpose registers of a standard microprocessor. In this case, the look up table supplying the term α_i can be stored in the control ROM or in main memory.

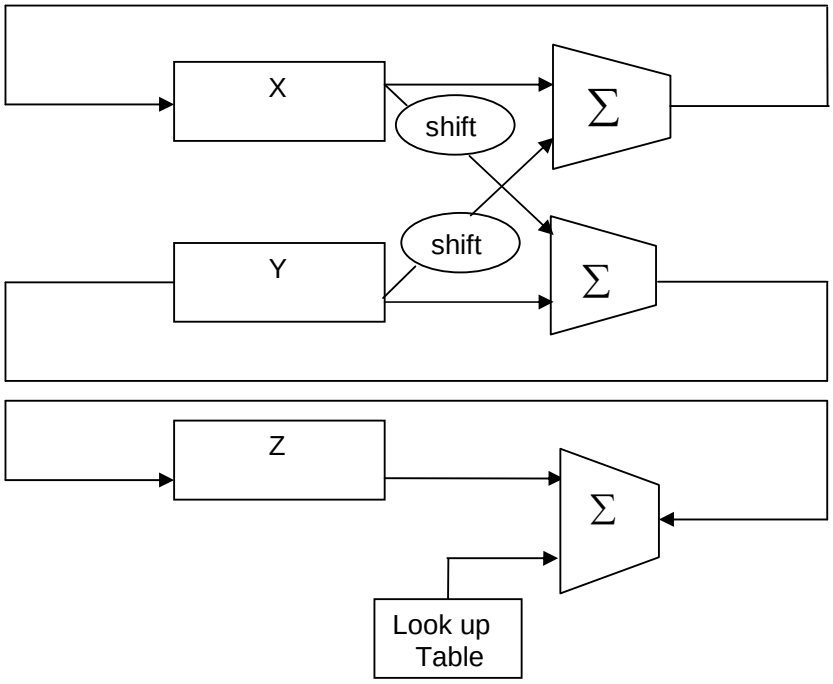


Figure. 3.1: Hardware elements needed for the CORDIC method

where high speed is not required and minimizing the hard ware cost is important (as in calculator), the adder in fig 3.1 can be bit serial. Then with k bit operands, O(k^2) clock cycle would be required to complete the k CORDIC iterations. This is acceptable for hand handled calculators, since even a delay of tens of thousands of clock cycles constitutes a small fraction of a second and thus is hardly noticeable to a

human user. Intermediate between the fully parallel and fully bit-serial realizations are a wide array of digit serial (for example decimal or radix-16) implementation that provide trade off speed versus cost.

3.2 Generalized CORDIC

The basic CORDIC method can be generalized to provide the more powerful tool for function evaluation. Generalized CORDIC is defined as follows:

$$\left.\begin{aligned} x_{i+1} &= x_i - \mu d_i y_i 2^{-i} \\ y_{i+1} &= y_i + d_i x_i 2^{-i} \\ z_{i+1} &= z_i - d_i \alpha^i \end{aligned}\right\} \qquad (3.1)$$

[Generalized CORDIC iteration]

Noting that the only difference with basic CORDIC is the introduction of the parameter μ in the equation for x and redefinition of α_i. The parameter μ can assume one of the three values:

$\mu = 1$ Circular rotations (Basic CORDIC) $\alpha_i = \tan^{-1} 2^{-i}$

$\mu = 0$ linear rotation $\alpha_i = 2^{-i}$

$\mu = -1$ Hyperbolic rotation $\alpha_i = \tanh^{-1} 2^{-i}$

Figure3.2 illustrates the three type of rotation in generalized CORDIC. For the circular case with $\mu = 1$, we introduce rotations that led to expansion of vector length by a factor $(1+\tan\alpha_i)^{1/2} = i/\cos\alpha_i$ in each step and by K = 1.646760258121…overall, where the vector length is the familiar $r_i = \sqrt{x^2 + y^2}$. With reference to figure3.2, the rotation angle AOB can be defined in terms of the area of sector AOB as follows:

$$angleAOB = \frac{2(areaAOB)}{(OU)^2}$$

The following equations, repeated here for ready comparison, characterize the results of circular CORDIC rotations:

$$\left.\begin{aligned} x_m &= k(x\cos z - y\sin z) \\ y_m &= k(y\cos z + x\sin z) \\ z_m &= 0 \end{aligned}\right\} \qquad (3.2)$$

(Circular rotation mode, Rule: choose $d_i \in \{-1,1\}$ such that $z \to 0$)

$$\left.\begin{aligned} x_m &= k(x^2 + y^2)^{1/2} \\ y_m &= 0 \\ z_m &= z + \tan^{-1}(y/x) \end{aligned}\right\} \tag{3.3}$$

(Circular vectoring mode, Rule: Choose $d_i \in \{-1,1\}$ such that $y \to 0$)

In linear rotations corresponding to $\mu = 0$, the end point of the vector is kept on the line $x = x_0$ and the vector length is defined by $r_i = x_i$. Hence, the length of the vector is always its true length OV and the scaling factor is 1. The following equations characterized the results of linear CORDIC rotations:

$$\left.\begin{aligned} x_m &= x \\ y_m &= y + xz \\ z_m &= 0 \end{aligned}\right\} \tag{3.4}$$

(Linear rotation mode, Rule: Choose $d_i \in \{-1,1\}$ such that $z \to 0$)

$$\left.\begin{aligned} x_m &= x \\ y_m &= 0 \\ z_m &= z + y/x \end{aligned}\right\} \tag{3.5}$$

(Linear vectoring mode, Rule: Choose $d_i \in \{-1,1\}$ such that $y \to 0$)

hence, linear CORDIC rotations can be used to perform multiplication (rotation mode, y = 0), multiply-add (rotation mode), division (vectoring mode, z = 0), or divide-add (vectoring mode).

In hyperbolic rotations corresponding to μ = -1, the rotation angle EOF can be defined in terms of the area of the hyperbolic sector EOF as follows:

$$angleEOF = \frac{2(areaEOB)}{(OW)^2}$$

The vector length is defined as $r_i = \sqrt{x^2 + y^2}$, with the length expansion due to rotation being $(1 + \tanh\alpha_i)^{1/2} = i/\cosh\alpha_i$. Because $\cosh\alpha_i$ > 1, the vector length actually shrinks, leading to an overall shrinkage factor K' = 0.8281593609602…… after all the iterations. The following equations characterize the results of hyperbolic CORDIC rotations:

$$\left.\begin{aligned} x_m &= k'(x\cosh z + y\sinh z) \\ y_m &= k(y\cosh z + x\sinh z) \\ z_m &= 0 \end{aligned}\right\} \tag{3.6}$$

(Hyperbolic rotation mode, Rule: choose $d_i \in \{-1,1\}$ such that $z \to 0$)

$$\left.\begin{aligned} x_m &= k(x^2 - y^2)^{1/2} \\ y_m &= 0 \\ z_m &= z + \tanh^{-1}(y/x) \end{aligned}\right\} \tag{3.7}$$

(Hyperbolic vectoring mode, Rule: Choose $d_i \in \{-1,1\}$ such that $y \to 0$.

hence, hyperbolic CORDIC rotations can be used to compute the hyperbolic sine and cosine functions (rotation mode, x = 1/k', y = 0) or the $\tanh^{-1}$ function (vectoring mode, x = 1, z = 0). Other functions can be computed indirectly [16].

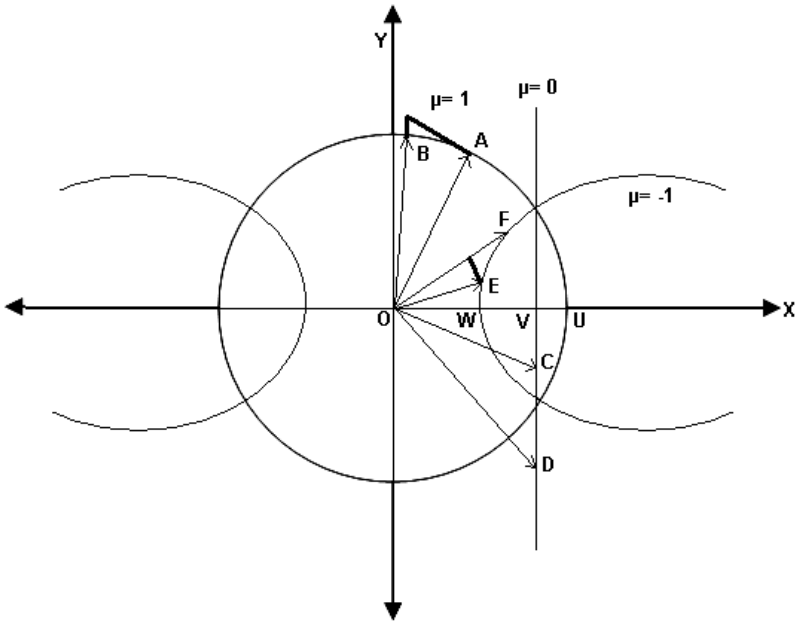


Figure. 3.2: Circular, Linear and Hyperbolic CORDIC [16]

3.3 The CORDIC-algorithm for Computing a Sine and Cosine

Jack E. Volder [1] described the Coordinate Rotation Digital Computer or CORDIC for the calculation of trigonometric functions, multiplication, division and conversion between binary and mixed radix number systems. The CORDIC-algorithm provides an iterative method of performing vector rotations by arbitrary angles using only shifts and adds. Volder's algorithm is derived from the general equations for

vector rotation. If a vector v with components (x, y) is to be rotated through an angle ϕ a new vector v' with components (x', y') is formed by (as in equation 2.1 and 2.2):

$$X = r\cos\theta, Y = r\sin\theta \tag{3.8}$$

$$V' = \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x.\cos(\phi) - y.\sin(\phi) \\ y.\cos(\phi) + x.\sin(\phi) \end{pmatrix} \tag{3.9}$$

Figure 4.2 illustrates the rotation of a vector $V = \begin{pmatrix} x \\ y \end{pmatrix}$ by the angle ϕ.

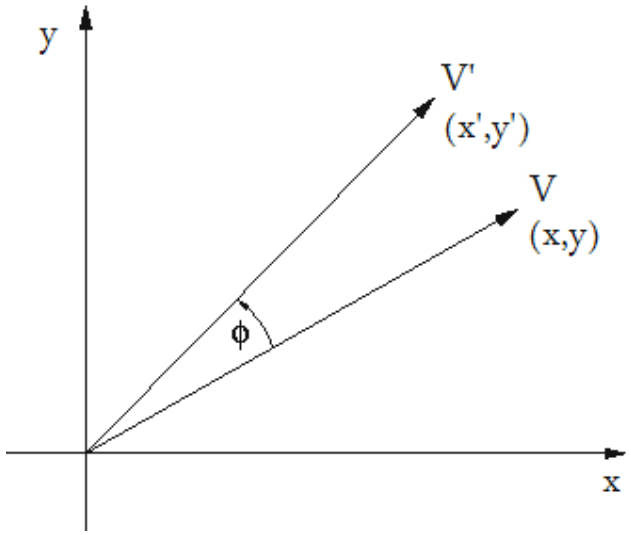


Figure 3.3: Rotation of a vector V by the angle ϕ

The individual equations for x' and y' can be rewritten as [17]:

$$x' = x.\cos(\phi) - y.\sin(\phi) \tag{3.10}$$

$$y' = y.\cos(\phi) + x.\sin(\phi) \tag{3.11}$$

and rearranged so that:

$$x' = \cos(\phi)[x - y.\tan(\phi)] \tag{3.12}$$

$$y' = \cos(\phi)[y + x.\tan(\phi)] \tag{3.13}$$

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

$$x_{i+1} = k_i[x_i - y_i.d_i.2^{-i}] \tag{3.14}$$

$$y_{i+1} = k_i[y_i + x_i.d_i.2^{-i}] \tag{3.15}$$

where $K_i = \cos(\arctan(2^{-i}))$ and $d_i = \pm 1$. The product of the K_i's represents the so-called K factor [6]:

$$k = \prod_{i=0}^{n-1} k_i \tag{3.16}$$

This *k* factor can be calculated in advance and applied elsewhere in the system. A good way to implement the *k* factor is to initialize the iterative rotation with a vector of length $|k|$ which compensates the gain inherent in the CORDIC algorithm. The resulting vector v' is the unit vector as shown in Figure 4.3.

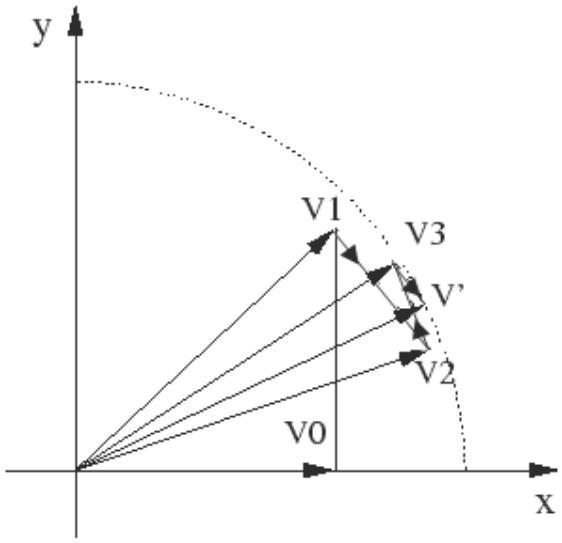


Figure 3.4: Iterative vector rotation, initialized with V0

Equations 4.7 and 4.8 can now be simplified to the basic CORDIC-equations:

$$x_{i+1} = [x_i - y_i.d_i.2^{-i}] \tag{3.17}$$

$$y_{i+1} = [y_i + x_i.d_i.2^{-i}] \tag{3.18}$$

The direction of each rotation is defined by d_i and the sequence of all d_i's determines the final vector. This yields to a third equation which acts like an angle accumulator and keeps track of the angle already rotated. Each vector *v* can be

described by both the vector length and angle or by its coordinates x and y. Following this incident, the CORDIC algorithm knows two ways of determining the direction of rotation: the *rotation mode* and the *vectoring mode*. Both methods initialize the angle accumulator with the desired angle z_0. The *rotation mode*, determines the right sequence as the angle accumulator approaches 0 while the *vectoring mode* minimizes the y component of the input vector.

The angle accumulator is defined by:

$$z_{i+1} = z_i - d_i.arctan(2^{-i}) \tag{3.19}$$

where the sum of an infinite number of iterative rotation angles equals the input angle ϕ [14]:

$$\phi = \sum_{i=0}^{\infty} d_i.arctan(2^{-i}) \tag{3.20}$$

Those values $arctan(2^{-i})$ can be stored in a small lookup table or hardwired depending on the way of implementation. Since the decision is which direction to rotate instead of whether to rotate or not, d_i is sensitive to the sign of z_i Therefore d_i can be described as:

$$d_i = \begin{cases} -1; if & z_i < 0 \\ +1; if & z_i \geq 0 \end{cases} \tag{3.21}$$

With equation 4.14 the CORDIC algorithm in rotation mode is described completely. Note, that the CORDIC method as described performs rotations only within $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. This limitation comes from the use of 2^0 for the tangent in the first iteration. However, since a sine wave is symmetric from quadrant to quadrant, every sine value from 0 to 2π can be represented by reflecting and/or inverting the first quadrant appropriately.

3.4 Implementation of various CORDIC Architectures

As intended by Volder, the CORDIC algorithm only performs shift and add operations and is therefore easy to implement and resource-friendly. However, when implementing the CORDIC algorithm one can choose between various design methodologies and must balance circuit complexity with respect to performance. The

most obvious methods of implementing a CORDIC, bit-serial, bit-parallel, unrolled and iterative, are described and compared in the following sections.

3.4.1 A Bit-Parallel Iterative CORDIC

The CORDIC structure as described in equations 4.10, 4.11, 4.12 and 4.14 is represented by the schematics in Figure 4.4 when directly translated into hardware. Each branch consists of an adder-sub tractor combination, a shift unit and a register for buffering the output. At the beginning of a calculation initial values are fed into the register by the multiplexer where the MSB of the stored value in the z-branch determines the operation mode for the adder-sub tractor. Signals in the x and y branch pass the shift units and are then added to or subtracted from the unshifted signal in the opposite path.

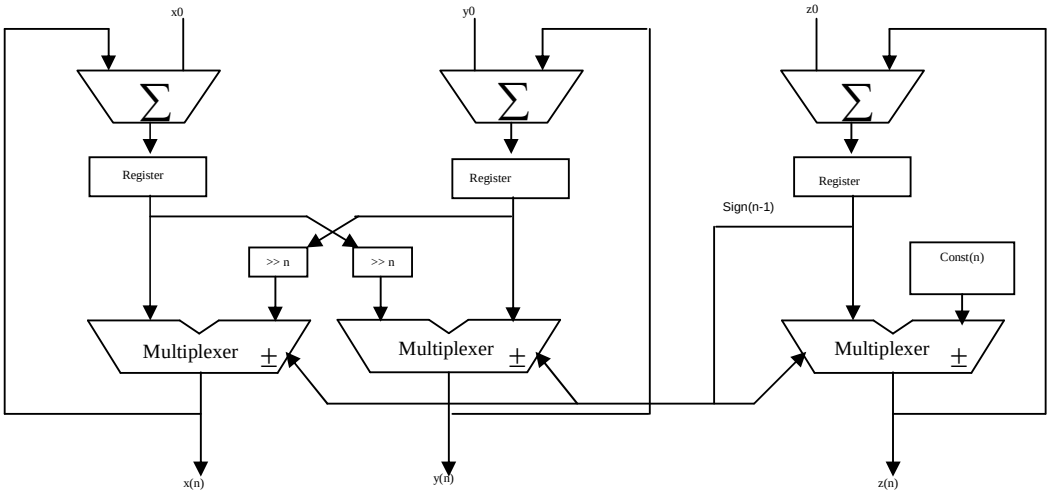


Figure 3.5: Iterative CORDIC

The z branch arithmetically combines the registers values with the values taken from a lookup table (LUT) whose address is changed accordingly to the number of iteration. For n iterations the output is mapped back to the registers before initial values are fed in again and the final sine value can be accessed at the output. A simple finite-state machine is needed to control the multiplexers, the shift distance and the addressing of the constant values. When implemented in an FPGA the initial values for the vector coordinates as well as the constant values in the LUT can be hardwired in a word wide manner. The adder and the sub tractor component are carried out separately and a multiplexer controlled by the sign of the angle accumulator distinguishes between addition and subtraction by routing the signals as required. The

shift operations as implemented change the shift distance with the number of iterations but those require a high fan in and reduce the maximum speed for the application [18]. In addition the output rate is also limited by the fact that operations are performed iteratively and therefore the maximum output rate equals 1/n times the clock rate.

3.4.2 A Bit-Parallel Unrolled CORDIC

Instead of buffering the output of one iteration and using the same resources again, one could simply cascade the iterative CORDIC, which means rebuilding the basic CORDIC structure for each iteration. Consequently, the output of one stage is the input of the next one, as shown in Figure 4.5, and in the face of separate stages two simplifications become possible. First, the shift operations for each step can be performed by wiring the connections between stages appropriately. Second, there is no need for changing constant values and those can therefore be hardwired as well.
The purely unrolled design only consists of combinatorial components and computes one sine value per clock cycle. Input values find their path through the architecture on their own and do not need to be controlled.

Obviously the resources in an FPGA are not very suitable for this kind of architecture. As we talk about a bit-parallel unrolled design with 16 bit word length, each stage contains 48 inputs and outputs plus a great number of cross-connections between single stages. Those cross-connections from the x-path through the shift components to the y-path and vice versa make the design difficult to route in an FPGA and cause additional delay times. From table 4.1 it can be seen how performance and resource usage change with the number of iterations if implemented in an XILINX FPGA. Naturally, the area and therefore the maximum path delay increase as stages are added to the design where the path delay is an equivalent to the speed which the application could run at.

Table 3.1: Performance and CLB usage in an XC4010E [19]

No. of Iterations	8	9	10	11	12	13
Complexity [CLB]	184	208	232	256	280	304
Max path delay[ns]	163.75	177.17	206.9	225.72	263.86	256.87

As described earlier, the area in FPGAs can be measured in CLBs, each of which consist of two lookup tables as well as storage cells with additional control components [20], [21]. For the purely combinatorial design the CLB's function generators perform the add and shift operations and no storage cells are used. This means registers could be inserted easily without significantly increasing the area.

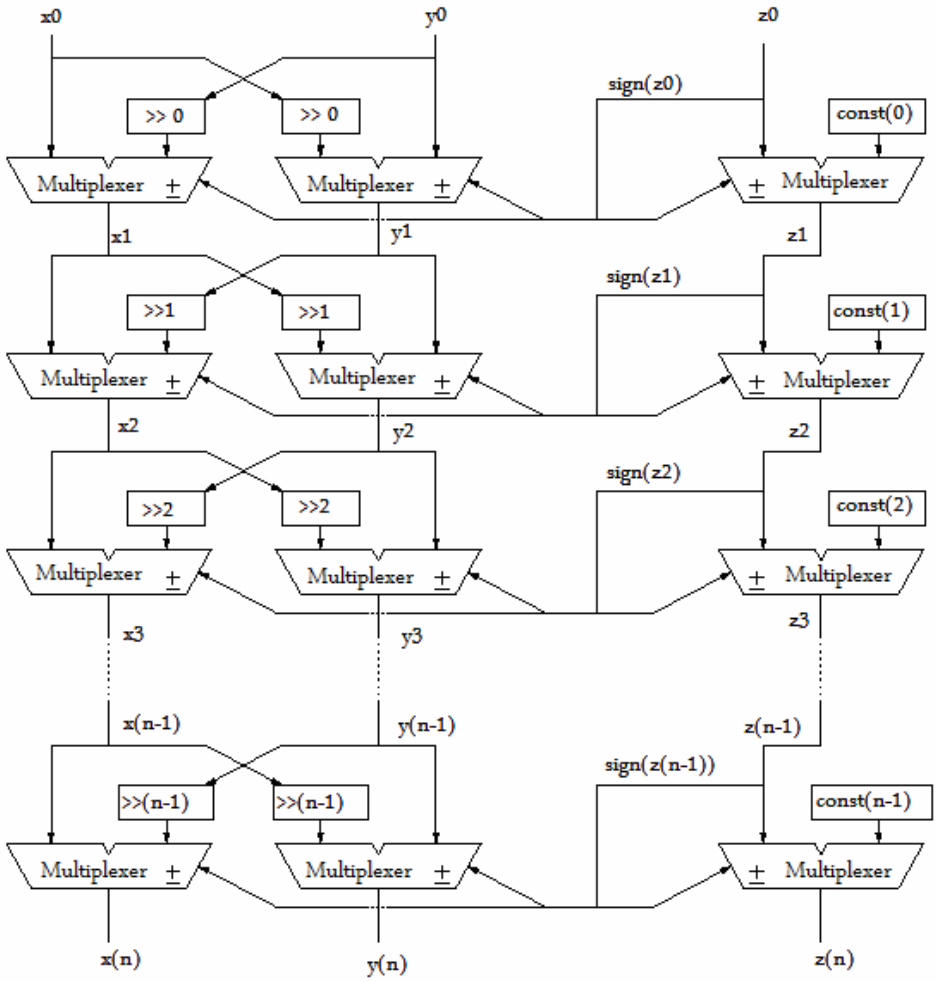


Figure 3.6: Unrolled CORDIC

However,inserting registers between stages would also reduce the maximum path delays and correspondingly a higher maximum speed can be achieved. It can be seen, that the number of CLBs stays almost the same while the maximum frequency increases as registers are inserted. The reason for that is the decreasing amount of combinatorial logic between sequential cells. Obviously, the gain of speed when

inserting registers exceeds the cost of area and makes therefore the fully pipelined CORDIC a suitable solution for generating a sine wave in FPGAs. Especially if a sufficient number of CLBs is at one's disposal, as is the case in high density devices like XILINX's Virtex or ALTERA's FLEX families, this type of architecture becomes more and more attractive.

3.4.3 A Bit-Serial Iterative CORDIC

Problems which involve repeated evaluation of a fixed set of nonlinear, algebraic equations appear frequently in scientific and engineering applications. Examples of such problems can be found in the robotics, engineering graphics, and signal processing areas. Evaluating complicated equation sets can be very time consuming in software, even when co-processors are used, especially when these equations contain a large number of nonlinear and transcendental functions as well as many multiplication and division operations. Both, the unrolled and the iterative bit-parallel designs, show disadvantages in terms of complexity and path delays going along with the large number of cross connections between single stages. To reduce this complexity we can change the design into a completely bit-serial iterative architecture. Bit-serial means only one bit is processed at a time and hence the cross connections become one bit-wide data paths. Clearly, the throughput becomes a function of

$$\frac{clock\ \ rate}{number\ \ of\ \ iterations\ \times\ \ word\ \ width}$$

In spite of this the output rate can be almost as high as achieved with the unrolled design. The reason is the structural simplicity of a bit-serial design and the correspondingly high clock rate achievable. Figure 4.6 shows the basic architecture of the bit serial CORDIC processor as implemented in a XILINX Spartan.

In this architecture the bit-serial adder-subtractor component is implemented as a full adder where the subtraction is performed by adding the 2's complement of the actual subtrahend [22]. The subtraction is again indicated by the sign bit of the angle accumulator as described in section 4.2.1. A single bit of state is stored at the adder to realize the carry chain [23] which at the same time requires the LSB to be fed in first. The shift-by-i operation can be realized by reading the bit $i-1$ from it's right end in

the serial shift registers. A multiplexer can be used to change position according to the current iteration. The initial values x_0, y_0 and z_0 are fed into the array at the left end of the serial-in serial-out register and as the data enters the adder component the multiplexer at the input switch and map back the results of the bit-serial adder into the registers. The constant LUT for this design is implemented as a multiplexer with hardwired choices. Finally, when all iterations are passed the input multiplexers switch again and initial values enter the bit-serial CORDIC processor as the computed sine values exit.

The design as implemented runs at a much higher speed than the bit-parallel architectures described earlier and fits easily in a XILINX SPARTAN device. The reason is the high ratio of sequential components to combinatorial components. The performance is constrained by the use of multiplexers for the shift operation and even more for the constant LUT. The latter could be replaced by a RAM or serial ROM where values are read by simply incrementing the memory's address. This would clearly accelerate the performance.

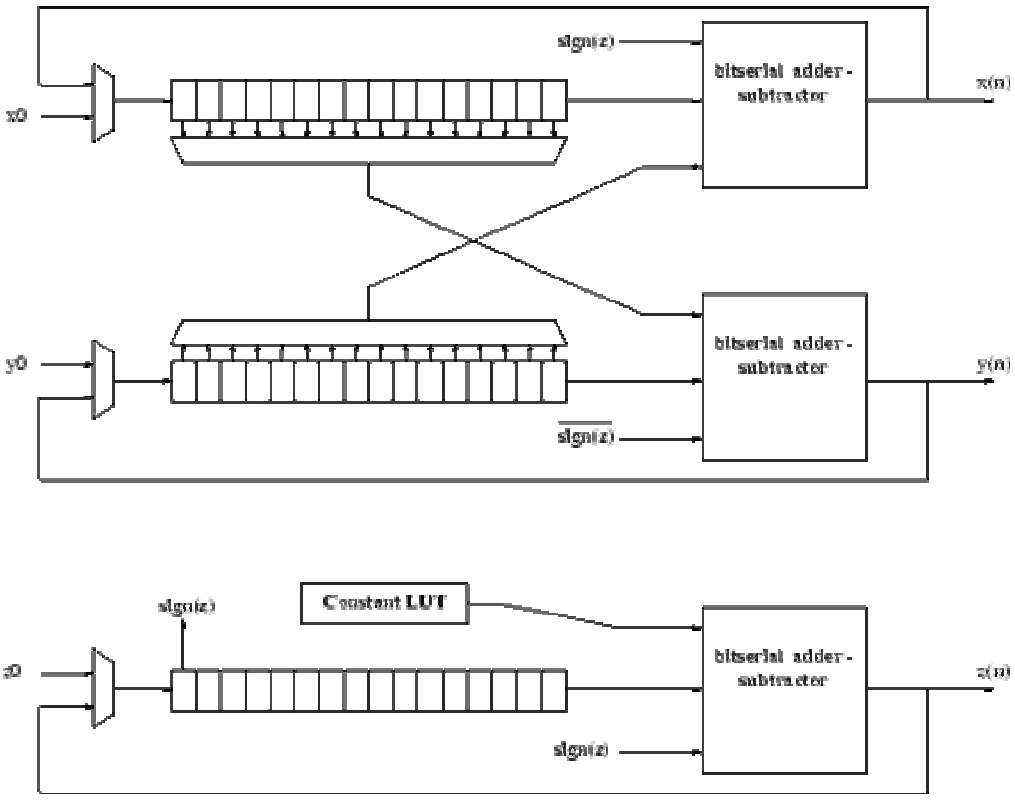


Figure 3.7: Bit-serial CORDIC

3.5 Comparison of the Various CORDIC Architectures

In the previous sections, we described various methods of implementing the CORDIC algorithm using an FPGA. The resulting structures show differences in the way of using resources available in the target FPGA device. Table 4.2 illustrates how the architectures for the iterative bit-serial and iterative bit-parallel designs for 16 bit resolution vary in terms of speed and area. The bit-serial design stands out due to it's low area usage and high achievable speed. Whereas the latency and hence the maximum throughput rate is much lower compared to the bit-parallel designs. The bit-parallel unrolled and fully pipelined design uses the resources extensively but shows the best latency per sample and maximum throughput rate. The prototyping environment limited the implementation of the unrolled design to 13 iterations. The iterative bit-parallel design provides a balance between unrolled and bit-serial design and shows an optimum usage of the resources.

3.6 Numerical Comparison

By running a test program on our Taylor, Polynomial Interpolation, and CORDIC approximations for cosine one can obtained the output attached. As one can see all three give fairly reasonable approximations to cosine. We can see from the absolute error that our Taylor approximation does just what is expected. As the values of x get further away from our centers 0, θ /6, θ /3, and θ /2, the error increases. The error then decreases as our angle again nears the next center. The polynomial interpolation turns out to be the worst approximation. By looking at the graph again, it appears that the approximation should be best when x is near 0, θ /4, and θ /2, looking at the absolute errors it appear that this is the case. However, the values computed in the test case show that at most angles the polynomial doesn't accurately correspond with the cos function. The best values, those near our chosen points, are still off by at the most 1/50. The best approximation looking at the absolute error is definitely CORDIC. In fact it turns out to be exact on nearly every angle (at least in terms of MATLAB's cos ( ) function). Clearly by numerical standards, the CORDIC method is the winner. However, note that the Taylor approximation did very well and with more centers would do even better. As for the polynomial interpolation, it does not seem to fit the sinusoid very well and this will apparently give a poor approximation.

3.7 Other Considerations

By the numerical comparison there is an obvious loser – polynomial interpolation, however there may be certain conditions that require different properties other than just accuracy. For instance, polynomial interpolation, while crude, is very fast to calculate and in terms of complexity it is the simplest of all three.

The Taylor approximation while it is slower to calculate than the polynomial approximation is a function and can be calculated quickly as well. However, the complexity of the method is much greater than the simple quadratic and for reasonable accuracy needs multiple expansions. Also, for the x values that fall in the middle of the centers accuracy is still an issue, albeit a small one.

Finally, the CORDIC method, which by far finds the most accurate approximations, has the most direct solution to the problem of evaluating trigonometric functions. By rotating a unit vector in a coordinate system, it is essentially finding (with precision limitations) the actual values of sin and cosine. However, this method is not a function that can be easily evaluated, but rather an iterative formula. This means that how fast it is depends on how fast it converges to the actual answer. While the convergence isn't great it is fairly fast, giving an error bound of $1/2^n \geq |\cos \theta - x_{n+1}|$, where x_{n+1} is the current step in the CORDIC algorithm. This means that the algorithm gets at about twice as close to the real solution every iteration. The complexity is great in that CORDIC has to be done n times to get a solution for an n-bit computer. However, this is combated by the fact that in n iterations both the cosine and sine are found. Something that the other methods can't do. Thus CORDIC is the best way of fast calculation by using subtraction and addition only.

In actual fact it would be more accurate to look at the resources available in the specific target devices rather than the specific needs in order to determine what architecture to use. The bit-serial structure is definitely the best choice for relatively small devices, but for FPGAs where sufficient CLBs are available one might choose the bit-parallel and fully pipelined architecture [24] since latency is minimal and no control unit is needed.

Table 3.2: Performance and CLB usage for the bit-parallel and bit-serial iterative designs. [19]

	CLB	LUT	FF	Speed	Latency	Max. Throughput
	[1]	[1]	[1]	[MHz]	[μs]	[Mio. Samples /sec]
bit-serial	111	153	108	48	5.33	0.1875
bit-parallel	138	252	52	36	0.44	2.25

3.8 Hardware Implementation

As demonstrated the amplitude control can be carried out within the CORDIC structure. Instead of hard-wiring the initial values as proposed in section 4.2.2, the values are now fed into the CORDIC structure through a separate input. Figure 4.9 illustrates the resulting structure of the complete oscillator

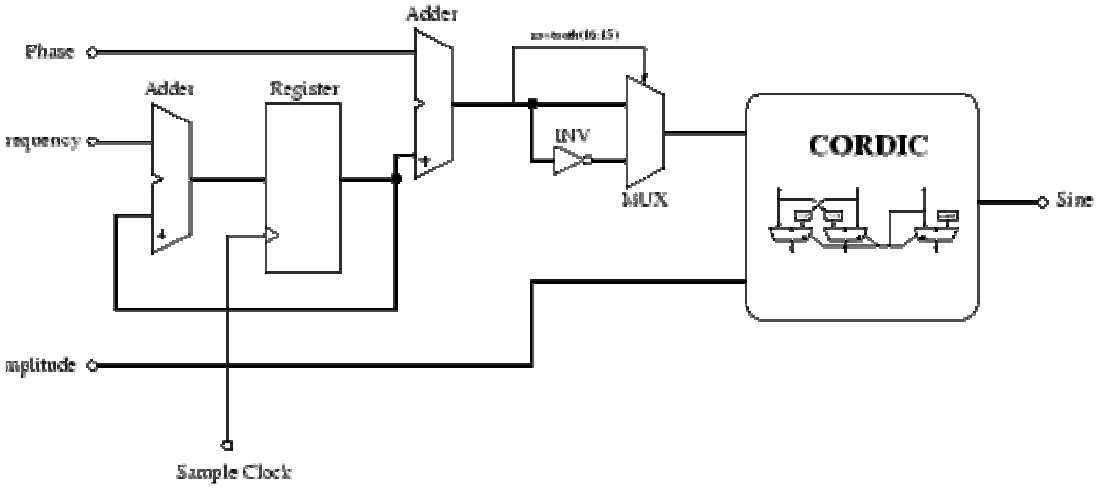


Figure 3.8: A CORDIC-based Oscillator for sine generation

The oscillator has been implemented and tested in a XILINX XC4010E. The architecture of this device provides specific resources in terms of CLBs (configurable logic blocks), LUTs, storage cells and maximum speed.

CHAPTER 4

CORDIC FOR DFT CALCULATION

In chapter 3 it has discussed that how sine and cosine can be calculated using CORDIC algorithm and now using this algorithm how Digital Fourier Transform (DFT) can be calculated has been discussed in this chapter. A Fourier transform is a special case of a wavelet transform with basis vectors defined by trigonometric functions sine and cosine. It is concerned with the representation of the signals by a sequence of numbers or symbols and the processing of these signals. Digital signal processing and analog signal processing are subfields of signal processing. DSP (Digital Signal Processing) includes subfields like: audio and speech signal processing, sonar and radar signal processing, sensor array processing, spectral estimation, statistical signal processing, digital image processing, signal processing for communications, biomedical signal processing, seismic data processing, etc. Since the goal of DSP is usually to measure or filter continuous real-world analog signals, the first step is usually to convert the signal from an analog to a digital form, by using an analog to digital converter. Often, the required output signal is another analog output signal, which requires a digital to analog converter. DSP algorithms have long been run on standard computers, on specialized processors called digital signal processors (DSPs), or on purpose-built hardware such as application-specific integrated circuit (ASICs). Today there are additional technologies used for digital signal processing including more powerful general purpose microprocessors, field-programmable gate arrays (FPGAs), digital signal controllers (mostly for industrial apparatus such as motor control), and stream processors, among others. There are several ways to calculate the Discrete Fourier Transform (DFT), such as solving simultaneous linear equations or the correlation method. The Fast Fourier Transform (FFT) (Figure 4.1 Eight point decimation-in-time FFT algorithm) is another method for calculating the DFT. While it produces the same result as the other approaches, it is incredibly more efficient, often reducing the computation time by hundreds. J.W. Cooley and J.W. Tukey are the founder credit for bringing the FFT (also known as divide and conquer algorithm). A Fast Fourier transform (FFT) is an efficient algorithm to compute the Discrete Fourier transform (DFT) and it’s inverse. FFTs are of great importance to a wide variety of applications, from digital signal processing

and solving partial differential equations to algorithms for quick multiplication of large integers. This article describes the algorithms, of which there are many; see discrete Fourier transform for properties and applications of the transform. The DFT is defined by the formula

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi kn}{N}} \quad for \quad k = 0,1,2....., N-1.$$

Evaluating these sums directly would take (N^2) arithmetical operations. An FFT is an algorithm to compute the same result in only (N log N) operations. In general, such algorithms depend upon the factorization of N, but (contrary to popular misconception) there are FFTs with (N log N) complexity for all N, even for prime N.

Many FFT algorithms only depend on the fact that $e^{-\frac{j2\pi}{N}}$ is an N*th* primitive root of unity, and thus can be applied to analogous transforms over any finite field, such as number-theoretic transforms

Decimation is the process of breaking down something into it's constituent parts. Decimation in *time* involves breaking down a signal in the time domain into smaller signals, each of which is easier to handle.

4.1 Calculation of DFT using CORDIC

If the input (time domain) signal, of N points, is x(n) then the frequency response *X(k)* can be calculated by using the DFT.

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad for \quad k = 0,1,2....., N-1.$$

(4.1)

where $W_N^{kn} = e^{-j2\pi kn/N}$

For a real sample sequence f (n), where n is {0, 1, …….. , (N-1)} DFT and the DHT, can be defined as

DFT:

$$F(k) = \sum_{n=0}^{N-1} f(N)[\cos(2\pi/N)kn - j.\sin(2\pi/N)kn]$$

$$= F_x(k) + F_y(k) \qquad (4.2)$$

DHT:

$$H(k) = \sum_{n=0}^{N-1} f(N)[\cos(2\pi / N)kn + \sin(2\pi / N)kn] \tag{4.3}$$

As it is evident from the expressions, above transforms involve trigonometric operations on the input sample sequences. These transforms can be expressed in terms of plane rotations. In other words, all the input samples are given a vector rotation by the defined angle in each of the transforms. The CORDIC unit can iteratively rotate an input vector $A = [A_X \quad A_Y]$ by a target angle θ through small steps of elementary angles θ_i (so that $\sum_i \theta_i$) to generate an output vector $B = [B_X \quad B_Y]$. The operation can be represented mathematically as:

$$[B_X \quad B_Y] = [A_X \quad A_Y]\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{4.4}$$

The rotation by a certain angle can be achieved by the summation of some elementary small rotations given by: $\theta = \sum_{i=0}^{15} \theta_i$ for a 16-bit machine. Now the rotation by elementary angles can be expressed in terms of sine of the angle as:

$$\sin\theta = \theta_i = 2^{-i} \tag{4.5}$$

where *i* is a positive integer. Since the elementary rotational angles θ_i have been assumed to be sufficiently small, the higher-order terms in the expansion of the sine and the cosine can be neglected. This assumption imposes some restriction on the allowable values of *i*.

Using sine cosine value generated by CORDIC algorithm DFT can be calculated by arranging them in matrix. It can be done by arranging sine cosine value in two different matrix one as real part and another as imaginary part then multiplying by input sampled data $f(n)$ which results in $F(k)$. Result can be stored in matrix as real and imaginary part. In the case of DHT computation, since there is no imaginary part the two matrixes can be shown in a single matrix by adding the two resulting matrix of sine and matrix of cosine in the case of 8×8 matrix the two results in the case of DFT and DHT can be arranged as below. In the case of DFT

$$F_R(k)=\begin{bmatrix} W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 \\ W_8^0 & W_8^1 & W_8^2 & W_8^3 & W_8^4 & W_8^5 & W_8^6 & W_8^7 \\ W_8^0 & W_8^2 & W_8^4 & W_8^6 & W_8^8 & W_8^{10} & W_8^{12} & W_8^{14} \\ W_8^0 & W_8^3 & W_8^6 & W_8^9 & W_8^{12} & W_8^{15} & W_8^{18} & W_8^{21} \\ W_8^0 & W_8^4 & W_8^8 & W_8^{12} & W_8^{16} & W_8^{20} & W_8^{24} & W_8^{28} \\ W_8^0 & W_8^5 & W_8^{10} & W_8^{15} & W_8^{20} & W_8^{25} & W_8^{30} & W_8^{35} \\ W_8^0 & W_8^6 & W_8^{12} & W_8^{18} & W_8^{24} & W_8^{30} & W_8^{36} & W_8^{42} \\ W_8^0 & W_8^7 & W_8^{14} & W_8^{21} & W_8^{28} & W_8^{35} & W_8^{42} & W_8^{49} \end{bmatrix}_R \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} \tag{4.6}$$

suffix R shows real part, in the above 8×8 matrix only cosine value is stored at corresponding positions.

$$F_I(k)=\begin{bmatrix} W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 \\ W_8^0 & W_8^1 & W_8^2 & W_8^3 & W_8^4 & W_8^5 & W_8^6 & W_8^7 \\ W_8^0 & W_8^2 & W_8^4 & W_8^6 & W_8^8 & W_8^{10} & W_8^{12} & W_8^{14} \\ W_8^0 & W_8^3 & W_8^6 & W_8^9 & W_8^{12} & W_8^{15} & W_8^{18} & W_8^{21} \\ W_8^0 & W_8^4 & W_8^8 & W_8^{12} & W_8^{16} & W_8^{20} & W_8^{24} & W_8^{28} \\ W_8^0 & W_8^5 & W_8^{10} & W_8^{15} & W_8^{20} & W_8^{25} & W_8^{30} & W_8^{35} \\ W_8^0 & W_8^6 & W_8^{12} & W_8^{18} & W_8^{24} & W_8^{30} & W_8^{36} & W_8^{42} \\ W_8^0 & W_8^7 & W_8^{14} & W_8^{21} & W_8^{28} & W_8^{35} & W_8^{42} & W_8^{49} \end{bmatrix}_I \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} \tag{4.7}$$

suffix I shows real part, in the above 8×8 matrix only sine value is stored at corresponding positions.

Final value is given as: $F(k)=F_Z(k)+j.F_I(k)$. The two result i.e. real and imaginary part can be stored in RAM location for further use. But in the case of DHT since there is no imaginary part the two values generated in the matrices be added together and used further for different applications. The resulting value of $H(k)$ by using equation (4.3) .

$$H(k)=\begin{bmatrix} W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 \\ W_8^0 & W_8^1 & W_8^2 & W_8^3 & W_8^4 & W_8^5 & W_8^6 & W_8^7 \\ W_8^0 & W_8^2 & W_8^4 & W_8^6 & W_8^8 & W_8^{10} & W_8^{12} & W_8^{14} \\ W_8^0 & W_8^3 & W_8^6 & W_8^9 & W_8^{12} & W_8^{15} & W_8^{18} & W_8^{21} \\ W_8^0 & W_8^4 & W_8^8 & W_8^{12} & W_8^{16} & W_8^{20} & W_8^{24} & W_8^{28} \\ W_8^0 & W_8^5 & W_8^{10} & W_8^{15} & W_8^{20} & W_8^{25} & W_8^{30} & W_8^{35} \\ W_8^0 & W_8^6 & W_8^{12} & W_8^{18} & W_8^{24} & W_8^{30} & W_8^{36} & W_8^{42} \\ W_8^0 & W_8^7 & W_8^{14} & W_8^{21} & W_8^{28} & W_8^{35} & W_8^{42} & W_8^{49} \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} \tag{4.8}$$

In the right side of equation (4.9) each term of 8×8 matrix can be seen as $\cos(2\pi kn/N)+\sin(2\pi kn/N)$ and $H(k)$ is given as: $H(k)=F_Z(k)+F_I(k)$.

4.2 FFT method for DFT calculation

Fast Fourier transform (FFT) is an efficient algorithm for computing the discrete Fourier transform. The discovery of the FFT algorithm paved the way for widespread use of digital methods of spectrum estimation which influenced the research in almost every field of engineering and science. DFT is a tool to estimate the samples of the CFT at uniformly spaced frequencies. DFT of an input sampled signal as discussed earlier in this chapter. It requires less multiplication than a simple approach of calculating DFT [25]. The basic computation using FFT is called butterfly computation which s shown in figure 4.2

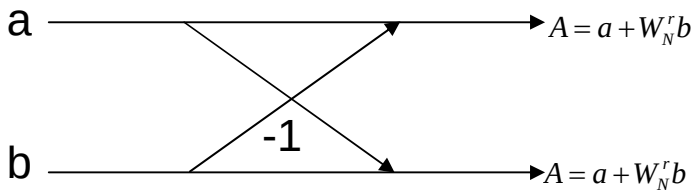


Figure 4.1: Basic butterfly computation in the decimation-in-time.

By using the above butterfly computation technique an 8×8 DFT can be calculated as shown figure 4.2 .

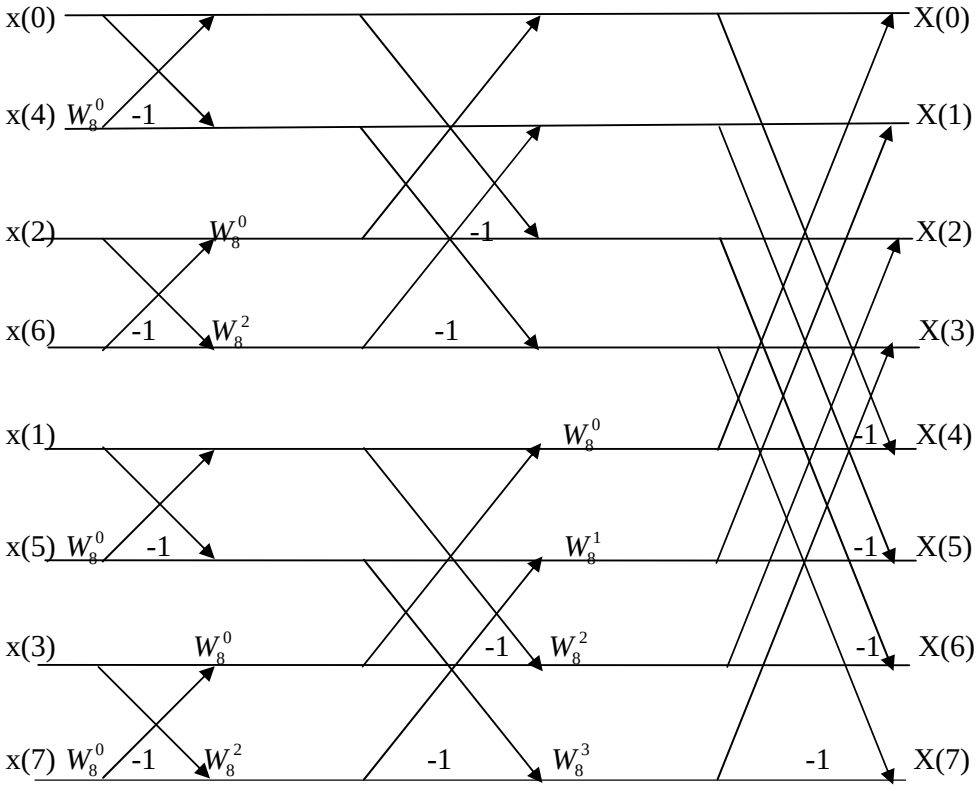


Figure 4.2: Eight point decimation-in-time FFT algorithm.

The steps involved in FFT method can be understood by the following figure 4.3

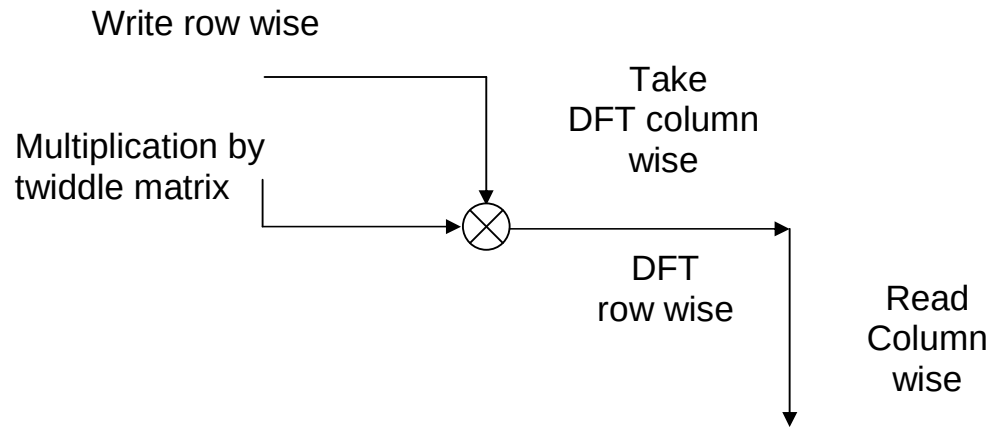


Figure 4.3: FFT divide and conquer method

We take this *N* point DFT, & break it down into two *N/2* point DFTs by splitting the input signal into odd & even numbered samples to get:

$$X(k) = \frac{1}{N}\sum_{m=0}^{N/2-1} x(2n)W_N^{2mk} + \frac{1}{N}\sum_{m=0}^{N/2-1} x(2n+1)W_N^{(2m+1)k} \qquad (4.9)$$

i.e. $X(k)$ = *even number samples* + *odd number samples*

$$X(k) = \frac{1}{N}\sum_{m=0}^{N/2-1} x(2m)(W_N^2)^{mk} + \frac{1}{N}W_N^k\sum_{m=0}^{N/2-1} x(2m+1)(W_N^2)^{rk} \qquad (4.10)$$

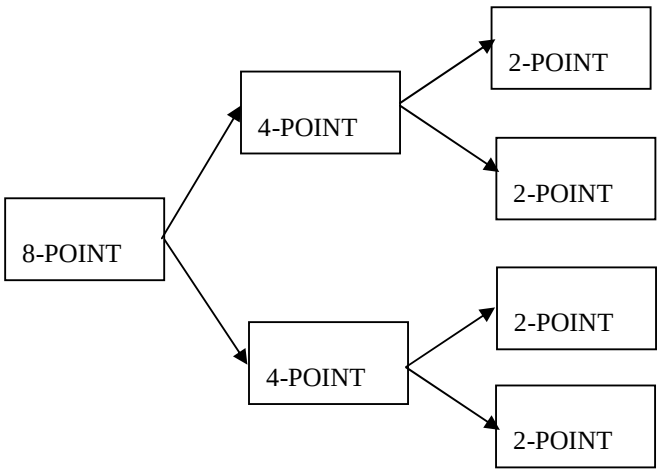


Figure 4.4: Fast Fourier Transform

Block diagram for FFT is in figure 4.4 which tells about the different stages to calculate DFT of a sampled signal.

CHAPTER 5

FPGA DESIGN FLOW

5.1 Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays are called this because rather than having a structure similar to a PAL or other programmable device, they are structured very much like a gate array ASIC. This makes FPGAs very nice for use in prototyping ASICs, or in places where and ASIC will eventually be used. For example, an FPGA maybe used in a design that need to get to market quickly regardless of cost. Later an ASIC can be used in place of the FPGA when the production volume increases, in order to reduce cost.

5.2 FPGA Architectures

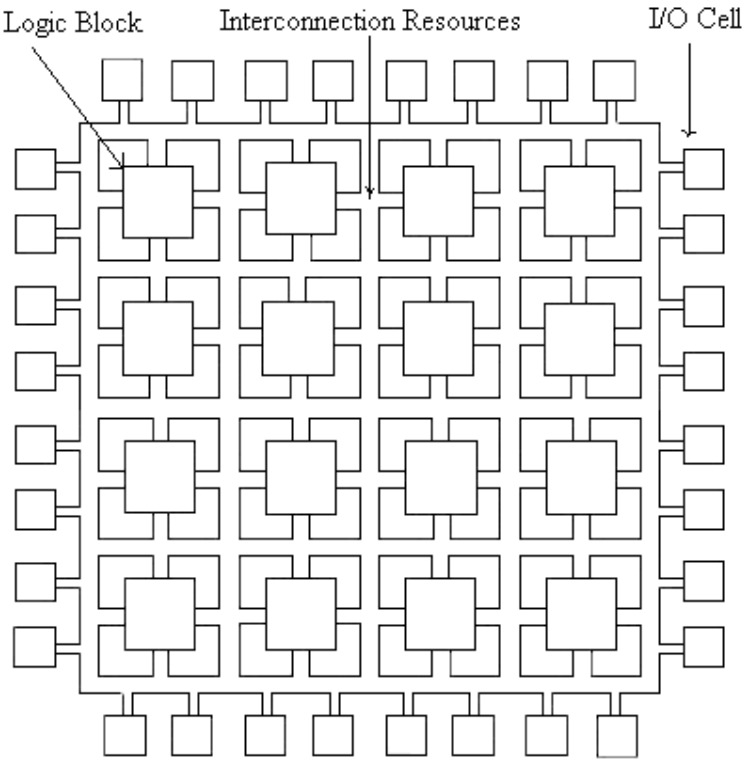


Figure5.1: FPGA Architecture

Each FPGA vendor has its own FPGA architecture, but in general terms they are all a variation of that shown in Figure 5.1. The architecture consists of configurable

logic blocks, configurable I/O blocks, and programmable interconnect. Also, there will be clock circuitry for driving the clock signals to each logic block, and additional logic resources such as ALUs, memory, and decoders may be available. The two basic types of programmable elements for an FPGA are Static RAM and anti-fuses.

5.2.1 Configurable Logic Blocks

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

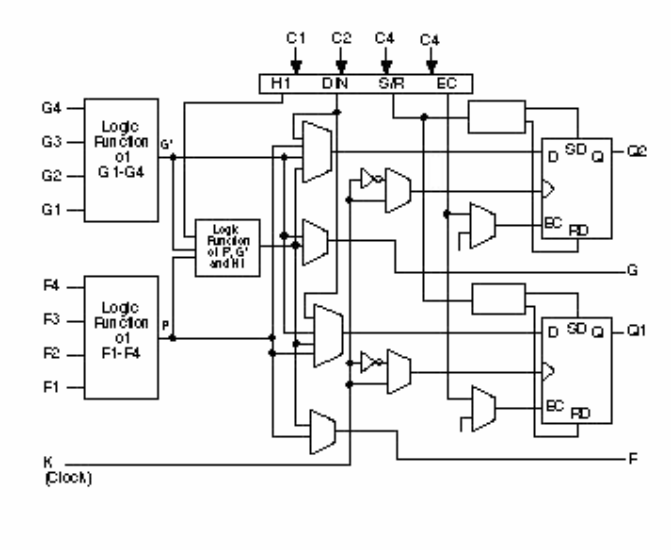


Figure 5.2: FPGA Configurable Logic Block

5.2.2 Configurable I/O Blocks

A Configurable I/O Block, shown in Figure 5.3, is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three state and open collector output controls. Typically there are pull up resistors on the outputs and sometimes pull down resistors. The polarity of the output

can usually be programmed for active high or active low output and often the slew rate of the output can be programmed for fast or slow rise and fall times. In addition, there is often a flip-flop on outputs so that clocked signals can be output directly to the pins without encountering significant delay. It is done for inputs so that there is not much delay on a signal before reaching a flip-flop which would increase the device hold time requirement.

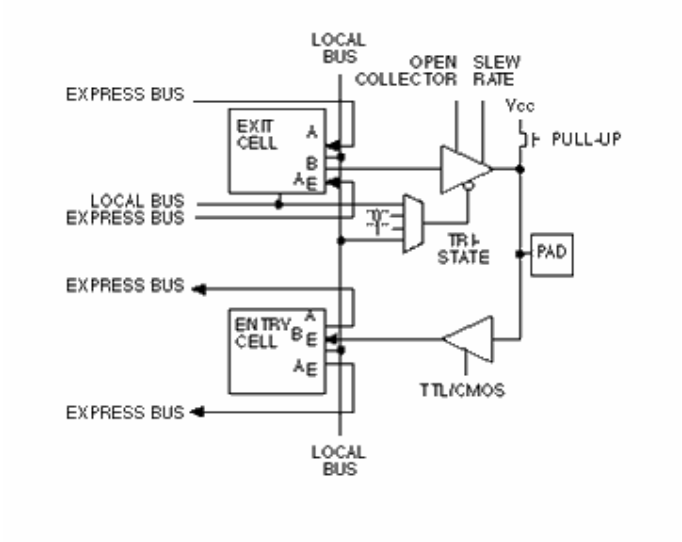


Figure 5.3: FPGA Configurable I/O Block

5.2.3 Programmable Interconnect

The interconnect of an FPGA is very different than that of a CPLD, but is rather similar to that of a gate array ASIC. In Figure 5.4, a hierarchy of interconnect resources can be seen. There are long lines which can be used to connect critical CLBs that are physically far from each other on the chip without inducing much delay. They can also be used as buses within the chip. There are also short lines which are used to connect individual CLBs which are located physically close to each other. There is often one or several switch matrices, like that in a CPLD, to connect these long and short lines together in specific ways. Programmable switches inside the chip allow the connection of CLBs to interconnect lines and interconnect lines to each other and to the switch matrix. Three-state buffers are used to connect many CLBs to a long line, creating a bus. Special long lines, called global clock lines, are specially designed for low impedance and thus fast propagation times. These are connected to

the clock buffers and to each clocked element in each CLB. This is how the clocks are distributed throughout the FPGA.

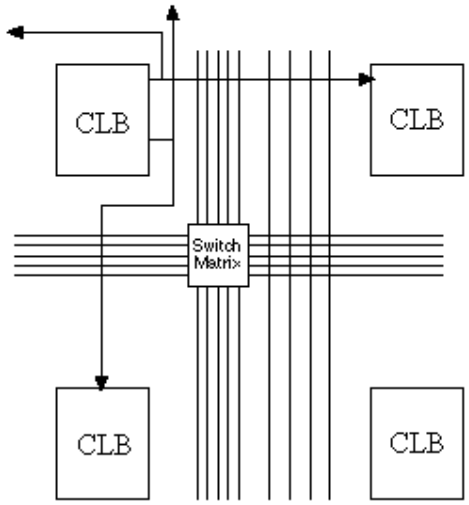


Figure 5.4: FPGA Programmable Interconnect

5.2.4 Clock Circuitry

Special I/O blocks with special high drive clock buffers, known as clock drivers, are distributed around the chip. These buffers are connected to clock input pads and drive the clock signals onto the global clock lines described above. These clock lines are designed for low skew times and fast propagation times. As we will discuss later, synchronous design is a must with FPGAs, since absolute skew and delay cannot be guaranteed. Only when using clock signals from clock buffers can the relative delays and skew times are guaranteed.

5.2.5 Small vs. Large Granularity

Small grain FPGAs resemble ASIC gate arrays in that the CLBs contain only small, very basic elements such as NAND gates, NOR gates, etc. The philosophies that small elements can be connected to make larger functions without wasting too much logic. In a large grain FPGA, where the CLB can contain two or more flip-flops, a design which does not need many flip-flops will leave many of them unused. Unfortunately, small grain architectures require much more routing resources, which

take up space and insert a large amount of delay which can more than compensate for the better utilization.

Small Granularity	**Large Granularity**
Better utilization	Fewer levels of logic
Direct conversion to ASIC	Less interconnect delay

A comparison of advantages of each type of architecture is shown in Table. The choice of which architecture to use is dependent on your specific application.

5.2.6 SRAM vs. Anti-fuse Programming

There are two competing methods of programming FPGAs. The first, SRAM programming, involves small Static RAM bits for each programming element. Writing the bit with a zero turns off a switch, while writing with a one turns on a switch. The other method involves anti-fuses which consist of microscopic structures which, unlike a regular fuse, normally make no connection. A certain amount of current during programming of the device causes the two sides of the anti-fuse to connect. The advantages of SRAM based FPGAs is that they use a standard fabrication process that chip fabrication plants are familiar with and are always optimizing for better performance. Since the SRAMs are reprogrammable, the FPGAs can be reprogrammed any number of times, even while they are in the system, just like writing to a normal SRAM. The disadvantages are that they are volatile, which means a power glitch could potentially change it. Also, SRAM based devices have large routing delays. The advantages of Anti-fuse based FPGAs are that they are non-volatile and the delays due to routing are very small, so they tend to be faster. The disadvantages are that they require a complex fabrication process, they require an external programmer to program them, and once they are programmed, they cannot be changed.

FPGA Families

Examples of SRAM based FPGA families include the following:

- Altera FLEX family
- Atmel AT6000 and AT40K families

- Lucent Technologies ORCA family
- Xilinx XC4000 and Virtex families

Examples of Anti-fuse based FPGA families include the following:

- Actel SX and MX families
- Quick logic pASIC family

5.3 The Design Flow

This section examines the design flow for any device, whether it is an ASIC, an FPGA, or a CPLD. This is the entire process for designing a device that guarantees that you will not overlook any steps and that you will have the best chance of getting backs a working prototype that functions correctly in your system. The design flow consists of the steps in

5.3.1 Writing a Specification

The importance of a specification cannot be overstated. This is an absolute must, especially as a guide for choosing the right technology and for making your needs known to the vendor. As specification allows each engineer to understand the entire design and his or her piece of it. It allows the engineer to design the correct interface to the rest of the pieces of the chip. It also saves time and misunderstanding. There is no excuse for not having a specification.

A specification should include the following information:

- An external block diagram showing how the chip fits into the system.
- An internal block diagram showing each major functional section.
- A description of the I/O pins including
- Output drive capability
- Input threshold level
- Timing estimates including

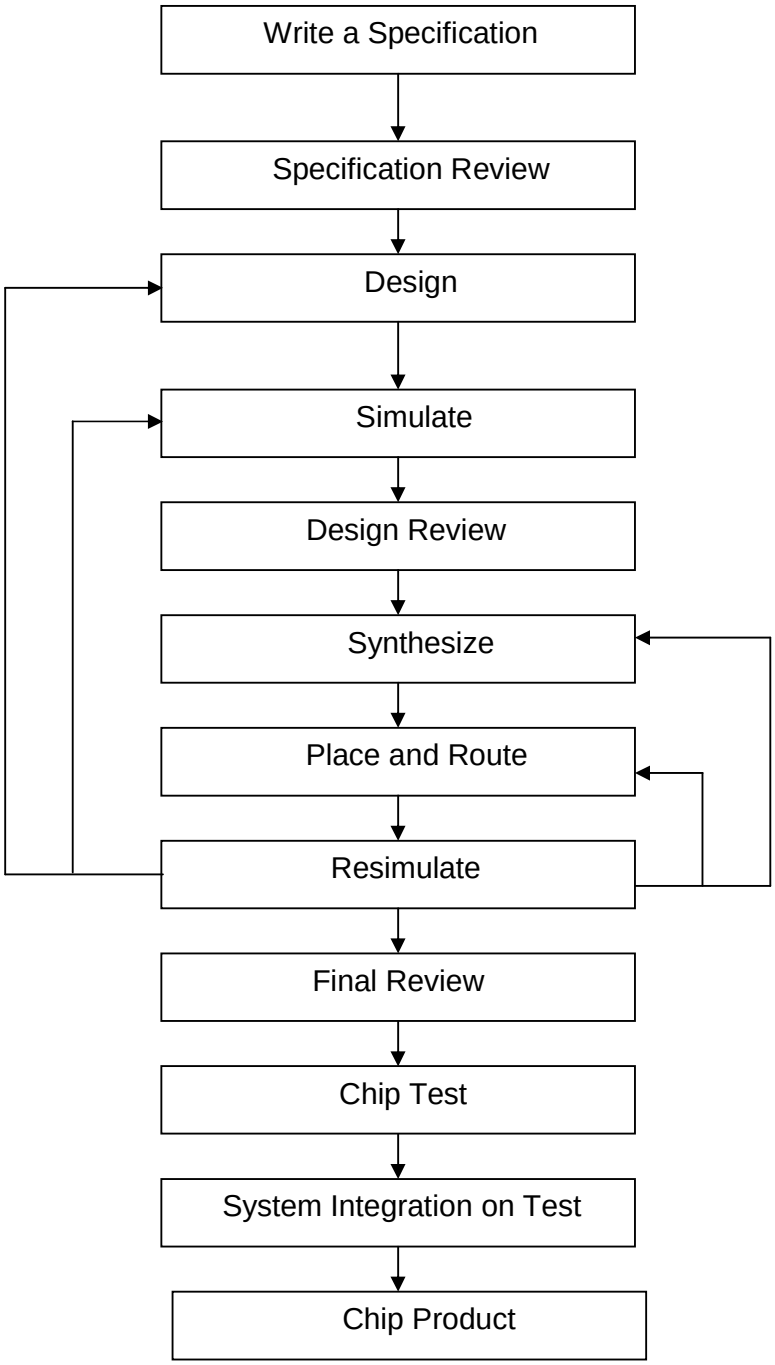


Figure 5.5: Design Flow of FPGA

- Setup and hold times for input pins
- Propagation times for output pins
- Clock cycle time
- Estimated gate count
- Package type
- Target power consumption
- Target price
- Test procedures

It is also very important to understand that this is a living document. Many sections will have best guesses in them, but these will change as the chip is being designed.

5.3.2 Choosing a Technology

Once a specification has been written, it can be used to find the best vendor with a technology and price structure that best meets your requirements.

5.3.3 Choosing a Design Entry Method

You must decide at this point which design entry method you prefer. For smaller chips, schematic entry is often the method of choice, especially if the design engineer is already familiar with the tools. For larger designs, however, a hardware description language (HDL) such as Verilog or VHDL is used because of its portability, flexibility, and readability. When using a high level language, synthesis software will be required to "synthesize" the design. This means that the software creates low level gates from the high level description.

5.3.4 Choosing a Synthesis Tool

You must decide at this point which synthesis software you will be using if you plan to design the FPGA with an HDL. This is important since each synthesis tool has recommended or mandatory methods of designing hardware so that it can correctly perform synthesis. It will be necessary to know these methods up front so that sections of the chip will not need to be redesigned later on. At the end of this phase it

is very important to have a design review. All appropriate personnel should review the decisions to be certain that the specification is correct, and that the correct technology and design entry method have been chosen.

5.3.5 Designing the chip

It is very important to follow good design practices. This means taking into account the following design issues that we discuss in detail later in this chapter.

- Top-down design
- Use logic that fits well with the architecture of the device you have chosen
- Macros
- Synchronous design
- Protect against metastability
- Avoid floating nodes
- Avoid bus contention

5.3.6 Simulating - design review

Simulation is an ongoing process while the design is being done. Small sections of the design should be simulated separately before hooking them up to larger sections. There will be many iterations of design and simulation in order to get the correct functionality. Once design and simulation are finished, another design review must take place so that the design can be checked. It is important to get others to look over the simulations and make sure that nothing was missed and that no improper assumption was made. This is one of the most important reviews because it is only with correct and complete simulation that you will know that your chip will work correctly in your system.

5.3.7 Synthesis

If the design was entered using an HDL, the next step is to synthesize the chip. This involves using synthesis software to optimally translate your register transfer level (RTL) design into a gate level design that can be mapped to logic blocks in the FPGA. This may involve specifying switches and optimization criteria in the HDL

code, or playing with parameters of the synthesis software in order to insure good timing and utilization.

5.3.8 Place and Route

The next step is to lay out the chip, resulting in a real physical design for a real chip. This involves using the vendor's software tools to optimize the programming of the chip to implement the design. Then the design is programmed into the chip.

5.3.9 Resimulating - final review

After layout, the chip must be resimulated with the new timing numbers produced by the actual layout. If everything has gone well up to this point, the new simulation results will agree with the predicted results. Otherwise, there are three possible paths to go in the design flow. If the problems encountered here are significant, sections of the FPGA may need to be redesigned. If there are simply some marginal timing paths or the design is slightly larger than the FPGA, it may be necessary to perform another synthesis with better constraints or simply another place and route with better constraints. At this point, a final review is necessary to confirm that nothing has been overlooked.

5.3.10 Testing

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

will only fail because of electrical or mechanical problems that will usually show up with this kind of stress testing.

5.4 Design Issues

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

5.4.1 Top-Down Design

Top-down design is the design method whereby high level functions are defined first, and the lower level implementation details are filled in later. A schematic can be viewed as a hierarchical tree as shown in Figure 14. The top-level block represents the entire chip. Each lower level block represents major functions of the chip. Intermediate level blocks may contain smaller functionality blocks combined with gate-level logic. The bottom level contains only gates and macro functions which are vendor-supplied high level functions. Fortunately, schematic capture software and hardware description languages used for chip design easily allows use of the top-down design methodology.

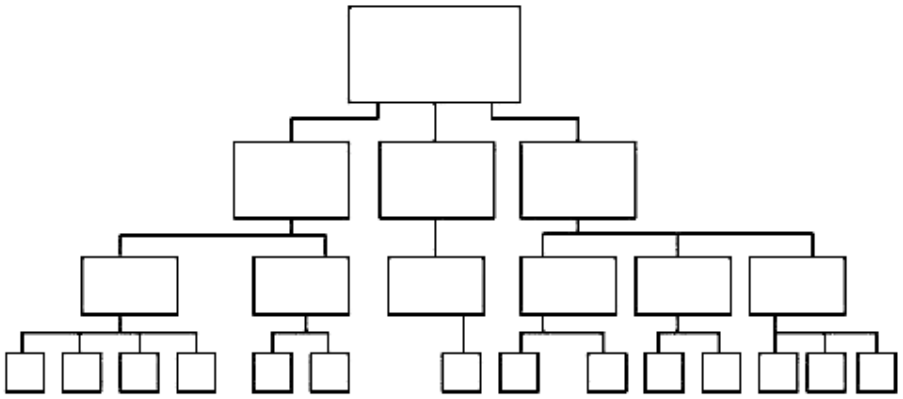


Figure 5.6: Top-Down Design

Top-down design is the preferred methodology for chip design for several reasons. First, chips often incorporate a large number of gates and a very high level of

functionality. This methodology simplifies the design task and allows more than one engineer, when necessary, to design the chip. Second, it allows flexibility in the design. Sections can be removed and replaced with a higher-performance or optimized designs without affecting other sections of the chip.

Also important is the fact that simulation is much simplified using this design methodology. Simulation is an extremely important consideration in chip design since a chip cannot be blue-wired after production. For this reason, simulation must be done extensively before the chip is sent for fabrication. A top-down design approach allows each module to be simulated independently from the rest of the design. This is important for complex designs where an entire design can take weeks to simulate and days to debug. Simulation is discussed in more detail later in this chapter.

5.4.2 Keep the Architecture in Mind

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

5.4.3 Synchronous Design

One of the most important concepts in chip design, and one of the hardest to enforce on novice chip designers, is that of synchronous design. Once an chip designer uncovers a problem due to asynchronous design and attempts to fix it, he or she usually becomes an evangelical convert to synchronous design. This is because asynchronous design problems are due to marginal timing problems that may appear intermittently, or may appear only when the vendor changes its semiconductor process. Asynchronous designs that work for years in one process may suddenly fail when the chip is manufactured using a newer process. Synchronous design simply means that all data is passed through combinatorial logic and flip-flops that are synchronized to a single clock. Delay is always controlled by flip-flops, not combinatorial logic. No signal that is generated by combinatorial logic can be fed back to the same group of combinatorial logic without first going through a synchronizing flip-flop. Clocks cannot be gated - in other words, clocks must go

directly to the clock inputs of the flip-flops without going through any combinatorial logic. The following sections cover common asynchronous design problems and how to fix them using synchronous logic.

5.4.4 Race conditions

Figure 5.7 shows an asynchronous race condition where a clock signal is used to reset a flip-flop. When SIG2 is low, the flip-flop is reset to a low state. On the rising edge of SIG2, the designer wants the output to change to the high state of SIG1. Unfortunately, since we don't know the exact internal timing of the flip-flop or the routing delay of the signal to the clock versus the reset input, we cannot know which signal will arrive first - the clock or the reset. This is a race condition. If the clock rising edge appears first, the output will remain low. If the reset signal appears first, the output will go high. A slight change in temperature, voltage, or process may cause a chip that works correctly to suddenly work incorrectly. A more reliable synchronous solution is shown in Figure 16. Here a faster clock is used, and the flip-flop is reset on the rising edge of the clock. This circuit performs the same function, but as long as SIG1 and SIG2 are produced synchronously - they change only after the rising edge of CLK - there is no race condition.

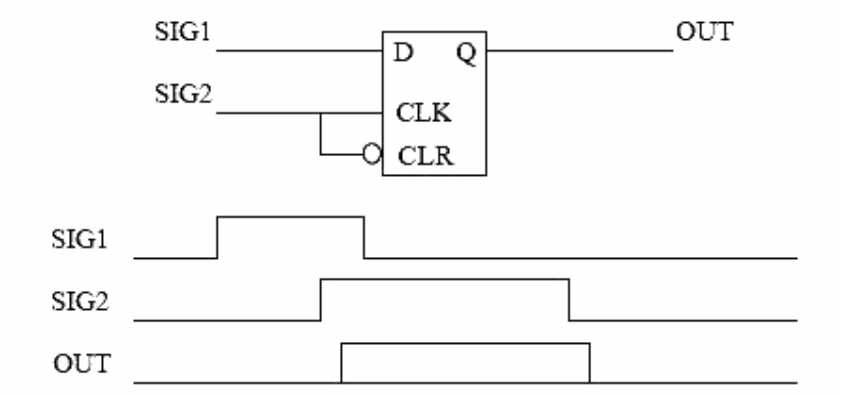


Figure 5.7: Asynchronous: Race Condition

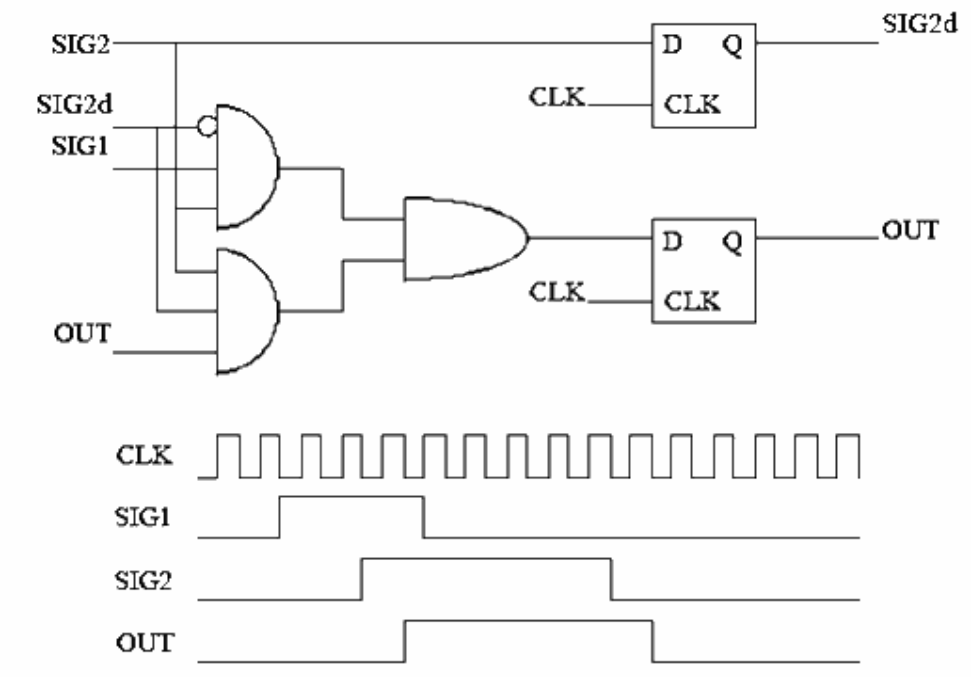


Figure 5.8: Synchronous: No Race Condition

5.4.5 Metastability

One of the great buzzwords, and often misunderstood concepts, of synchronous design is metastability. Metastability refers to a condition which arises when an asynchronous signal is clocked into a synchronous flip-flop. While chip designers would prefer a completely synchronous world, the unfortunate fact is that signals coming into a chip will depend on a user pushing a button or an interrupt from a processor, or will be generated by a clock which is different from the one used by the chip. In these cases, the asynchronous signal must be synchronized to the chip clock so that it can be used by the internal circuitry. The designer must be careful how to do this in order to avoid metastability problems as shown in Figure 5.9. If the ASYNC_IN signal goes high around the same time as the clock, we have an unavoidable race condition. The output of the flip-flop can actually go to an undefined voltage level that is somewhere between a logic 0 and logic 1. This is because an internal transistor did not have enough time to fully charge to the correct level. This meta level may remain until the transistor voltage leaks off or “decays”, or until the next clock cycle. During the clock cycle, the gates that are connected to the output of the flip-flop may interpret this level differently. In the figure, the upper gate sees the level as logic 1 whereas the lower gate sees it as logic 0. In normal operation, OUT1

and OUT2 should always be the same value. In this case, they are not and this could send the logic into an unexpected state from which it may never return. This metastability can permanently lock up the chip.

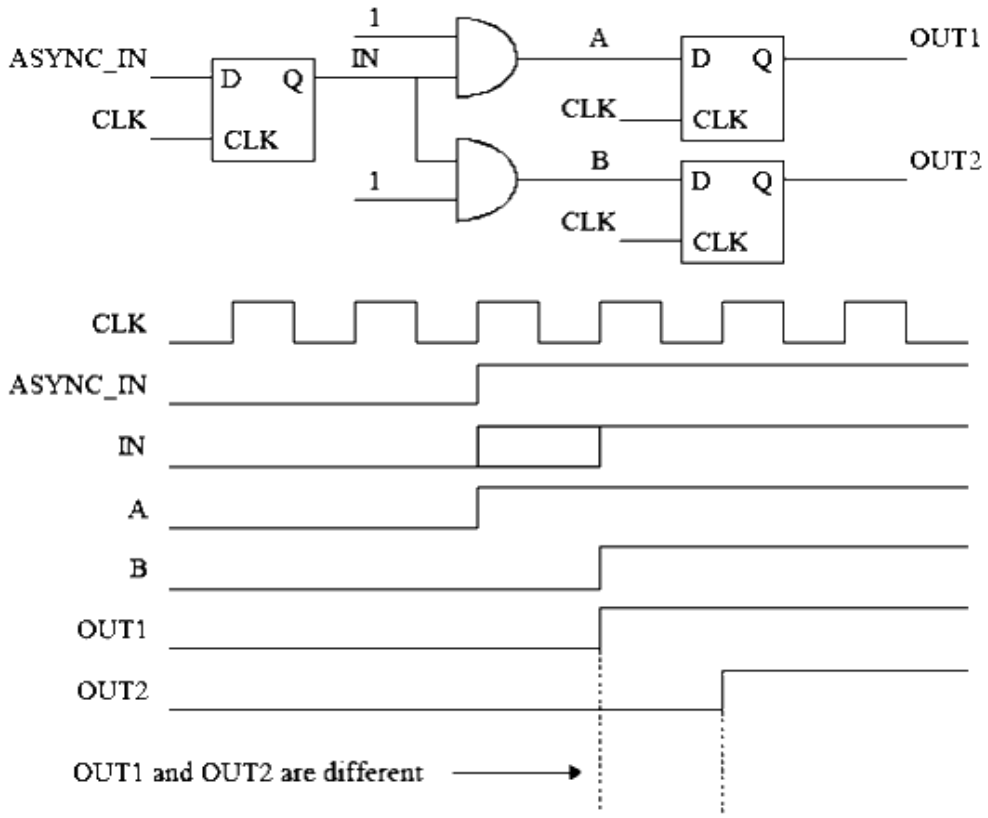


Figure 5.9: Metastability - The Problem

The “solution” to this metastability problem by placing a synchronizer flip-flop in front of the logic, the synchronized input will be sampled by only one device, the second flip-flop, and be interpreted only as logic 0 or 1. The upper and lower gates will both sample the same logic level, and the metastability problem is avoided. Or is it? The word solution is in quotation marks for a very good reason. There is a very small but non-zero probability that the output of the synchronizer flip-flop will not decay to a valid logic level within one clock period. In this case, the next flip-flop will sample an indeterminate value, and there is again a possibility that the output of that flip-flop will be indeterminate. At higher frequencies, this possibility is greater. Unfortunately, there is no certain solution to this problem. Some vendors provide special synchronizer flip-flops whose output transistors decay very quickly. Also,

inserting more synchronizer flip-flops reduces the probability of metastability but it will never reduce it to zero. The correct action involves discussing metastability problems with the vendor, and including enough synchronizing flip-flops to reduce the probability so that it is unlikely to occur within the lifetime of the product.

5.4.6 Timing Simulation

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

.

CHAPTER 6

RESULTS AND DISCUSSIONS

6.1 ModelSim Simulation Results

Figure 6.1 and table 6.1 consists the ModelSim simulation result for binary input angle z0 and binary outputs xn1(sin(z0)), yn(cos(z0)) in the form of waveform and their corresponding magnitude respectively. Figure 6.2 and table 6.2 consists the ModelSim simulation result for real input angle z0 and real outputs xn1(sin(z0)), yn(cos(z0)) in the form of waveform and their corresponding magnitude respectively.

6.1.1 For binary input and binary output

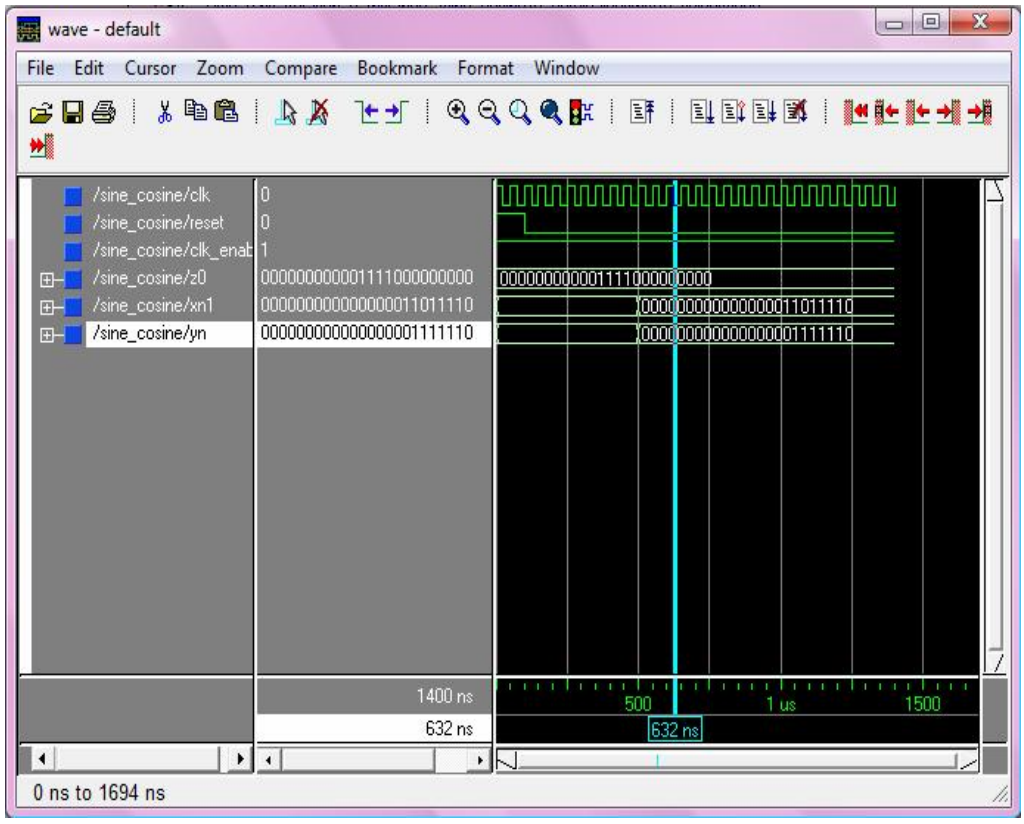


Figure 6.1: Sine-Cosine value generated for input angle z0 (binary value)

Table 6.1: Sine-Cosine value for input angle z0

Reset	1	0
Clk_enable	1	1
Input(z0)	00000000000001111000000000	00000000000001111000000000
Sine value	00000000000000000000000000	00000000000000000001111110
Cosine value	00000000000000000000000000	00000000000000000011011110

6.1.2 For sine-cosine real input and real output

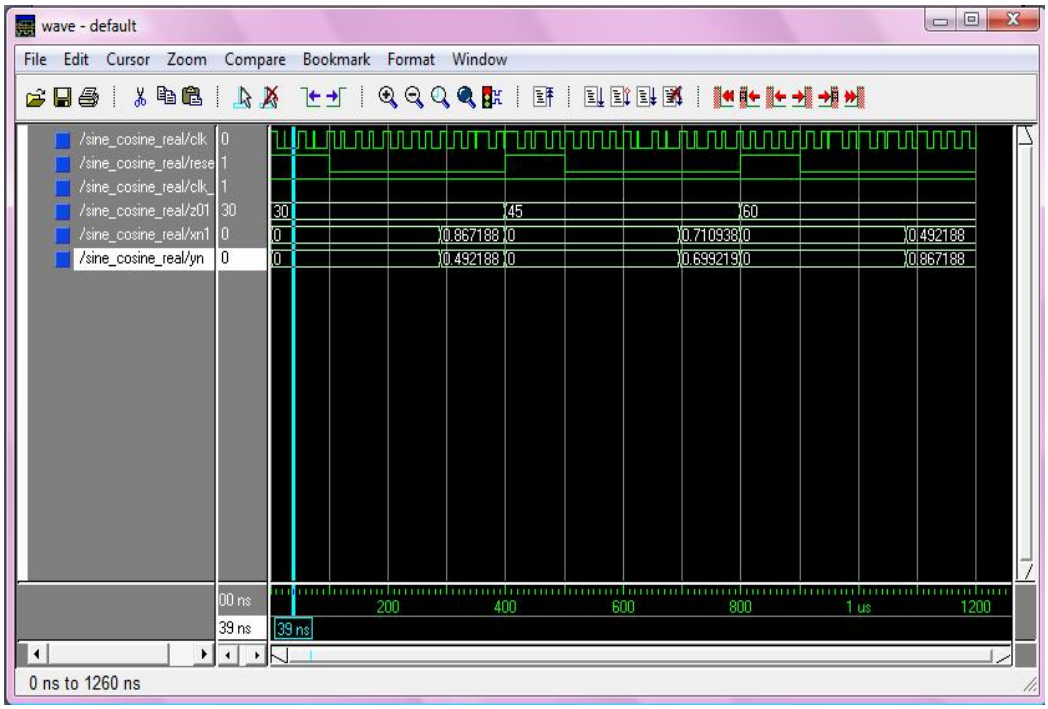


Figure 6.2: Sine-Cosine value generated for input angle z0 (integer value)

Table 6.2: Sine Cosine value for input angle z0

S. No.	Reset	Clk_enable	Input angle(z0)	Sine value	Cosine value
1	1	×	×	0	0
2	0	1	30	0.49	0.867
3	0	1	45	0.711	0.699
4	0	1	60	0.867	0.49

6.1.3 For DFT using FFT Algorithm

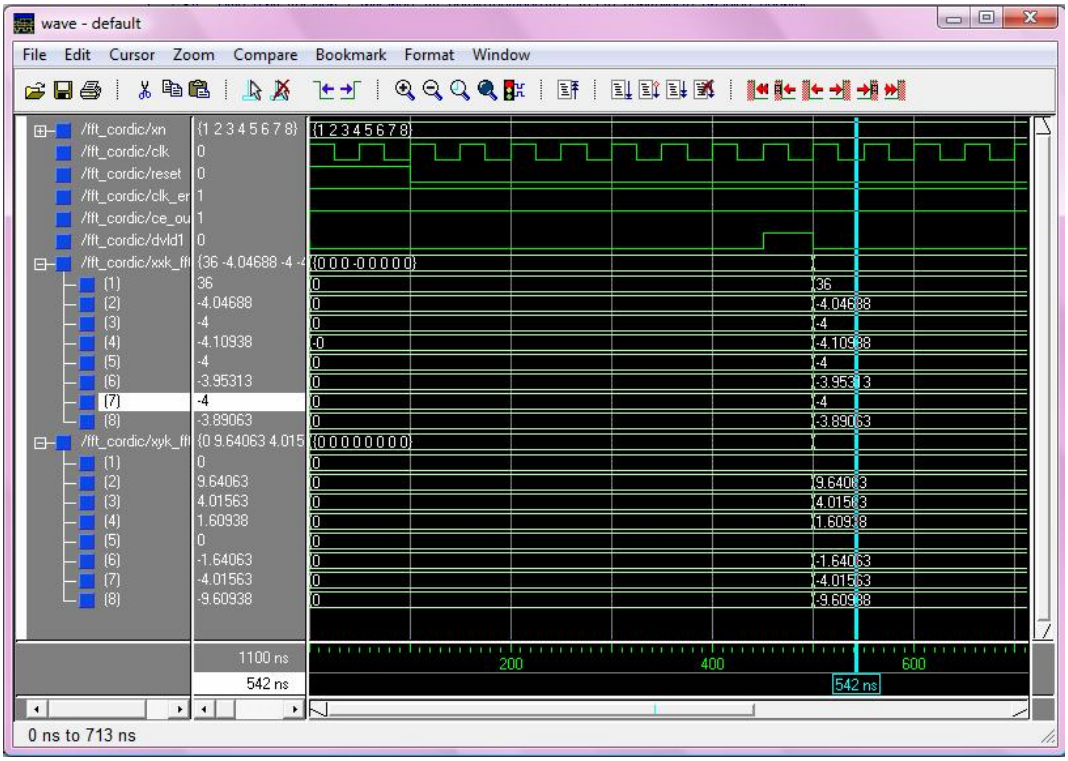


Figure 6.3: Real input/output waveforms of DFT using FFT algorithm

Table 6.3: Real input/output values of DFT using FFT algorithm

S.N.	Reset	Clk_enable	Input	Output real	Output imaginary
1.	0	×	×	0	0
2.	1	1	1	36	0
3.	1	1	2	-4.04	9.64
4.	1	1	3	-4	4.0
5.	1	1	4	-4.1	1.61
6.	1	1	5	-4	0
7.	1	1	6	-3.95	-1.64
8.	1	1	7	-4	-4.01
9.	1	1	8	-3.98	-9.61

6.2 XILINX simulation results

Block diagram generated by XILINX 9.2i for sine-cosine using CORDIC is shown in figure 6.4. Here inputs are z0 (input angle), clk (clock), clk_enable (clock enable), reset and outputs are xn1 (magnitude of cosine of input angle), yn (magnitude of sine of input angle), ce_out and dvld are chip enable signal for the next stage blocks. Figure 6.5 shows the RTL schematic of sine-cosine generator and its internal block diagram.

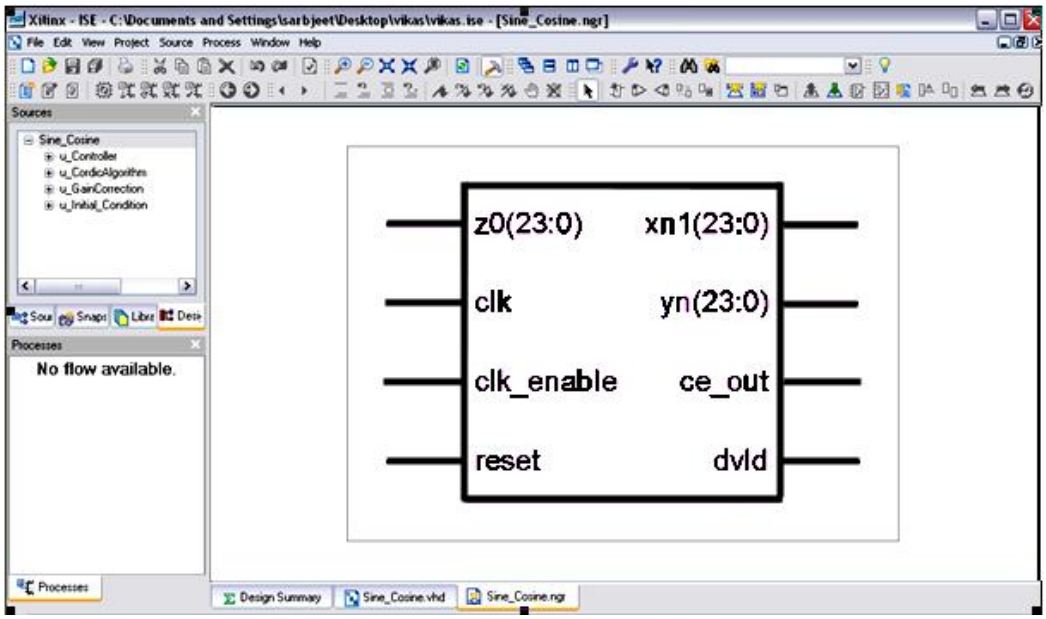


Figure 6.4: Top level RTL schematic for Sine Cosine

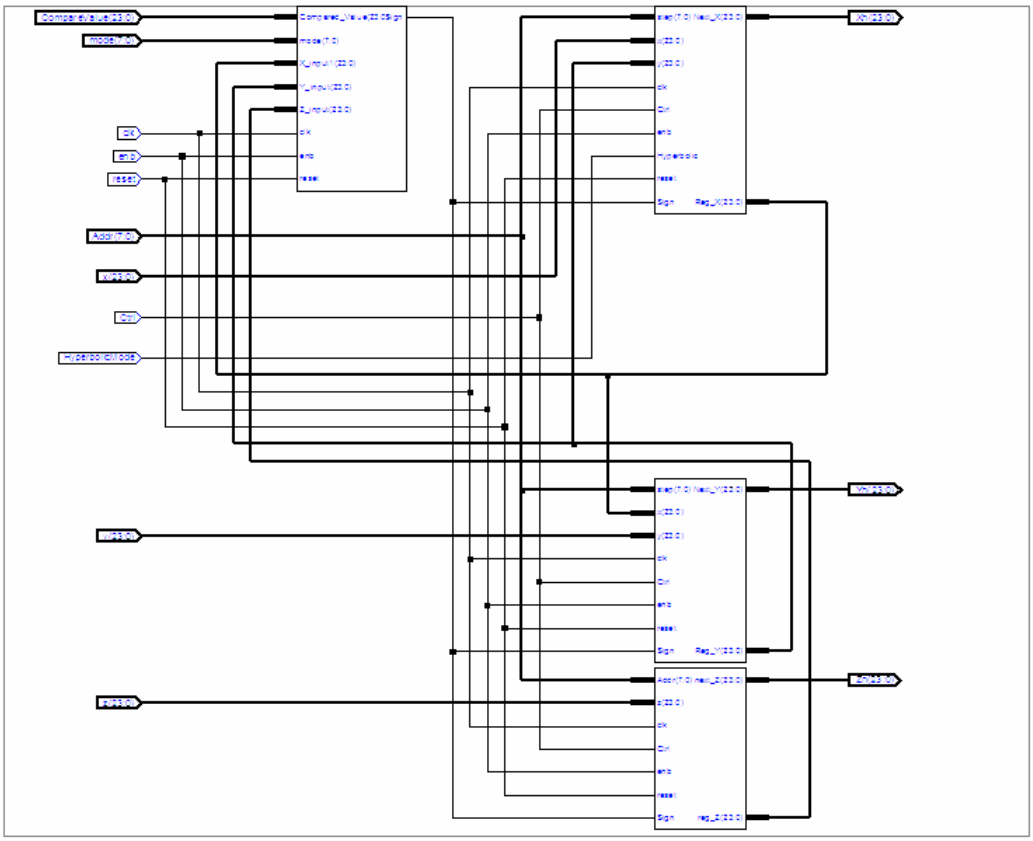


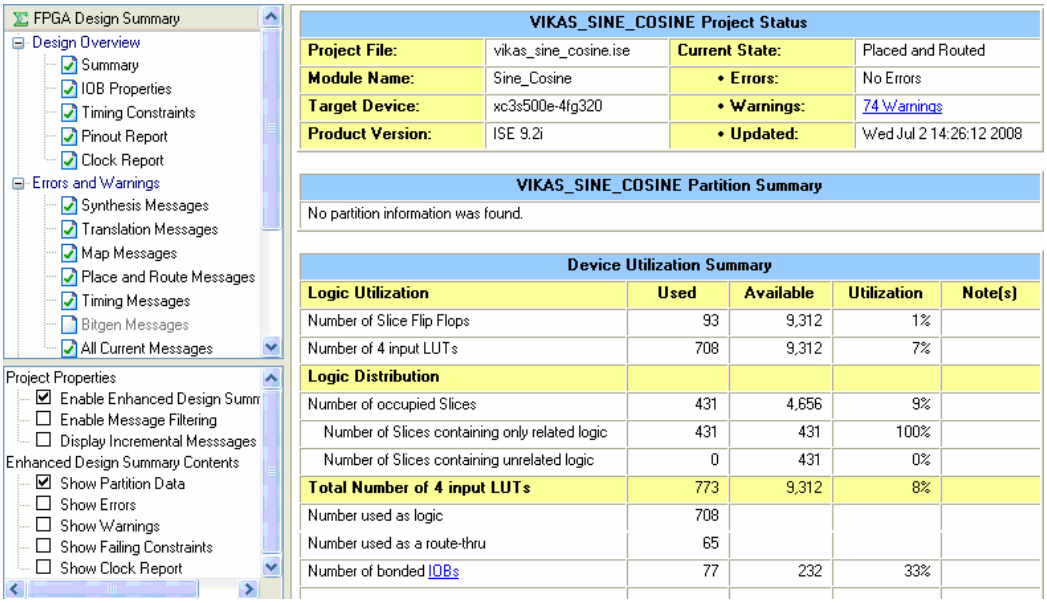
Figure 6.5: RTL schematic for Sine-Cosine

Table 6.4 and table 6.5 (a),(b) shows the power summary and design summary respectively produced by XILINX tool. Table 6.6, table 6.7 and table 6.8 have synthesis report generated by XILINX showing number of multiplexers, number of adder, number of flip-flops used, timing and thermal summary of the chip generated respectively.

Table 6.4: Power summary

	I(mA)	P(mW)
Total estimated power consumption		81
Vccint 1.20V	26	31
Vccaux 2.50V	18	45
Vcco25 2.50V	2	5
Clocks	0	0
Inputs	0	0
Logic	0	0
Vcco25	0	0
Signals	0	0
Quiescent Vccint 1.20V	26	31
Quiescent Vccaux 2.50V	18	45
Quiescent Vcco25 2.50V	2	5

Table 6.5: (a) Design summary of Sine-Cosine

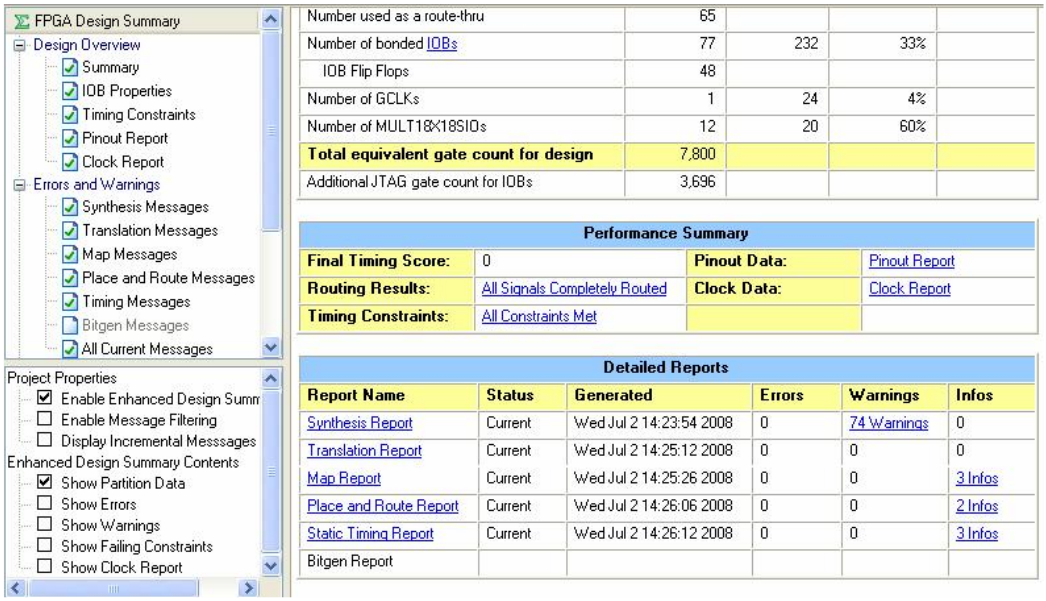


VIKAS_SINE_COSINE Project Status			
Project File:	vikas_sine_cosine.ise	**Current State:**	Placed and Routed
Module Name:	Sine_Cosine	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	74 Warnings
Product Version:	ISE 9.2i	**• Updated:**	Wed Jul 2 14:26:12 2008

VIKAS_SINE_COSINE Partition Summary
No partition information was found.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	93	9,312	1%	
Number of 4 input LUTs	708	9,312	7%	
Logic Distribution				
Number of occupied Slices	431	4,656	9%	
Number of Slices containing only related logic	431	431	100%	
Number of Slices containing unrelated logic	0	431	0%	
Total Number of 4 input LUTs	773	9,312	8%	
Number used as logic	708			
Number used as a route-thru	65			
Number of bonded IOBs	77	232	33%	

Table 6.5: (b) Design summary of Sine-Cosine



Number used as a route-thru	65			
Number of bonded IOBs	77	232	33%	
IOB Flip Flops	48			
Number of GCLKs	1	24	4%	
Number of MULT18X18SIOs	12	20	60%	
Total equivalent gate count for design	7,800			
Additional JTAG gate count for IOBs	3,696			

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Wed Jul 2 14:23:54 2008	0	74 Warnings	0
Translation Report	Current	Wed Jul 2 14:25:12 2008	0	0	0
Map Report	Current	Wed Jul 2 14:25:26 2008	0	0	3 Infos
Place and Route Report	Current	Wed Jul 2 14:26:06 2008	0	0	2 Infos
Static Timing Report	Current	Wed Jul 2 14:26:12 2008	0	0	3 Infos
Bitgen Report					

Table 6.6: Advanced HDL Synthesis Report for sine cosine

Macro Statistics:	
32x12-bit multiplier	: 24
32x13-bit multiplier	: 15
32x14-bit multiplier	: 8
32x15-bit multiplier	: 6
Total Multipliers	**: 73**
2-bit adder	: 1
3-bit adder	: 1
32-bit adder	: 72
4-bit adder	: 1
5-bit adder	: 1
6-bit adder	: 1
7-bit adder	: 1
8-bit adder	: 1
Total Adders/Subtractors	**: 79**
Flip-Flops	: 616
Total Registers(Flip-Flops)	**: 616**

Table 6.7: Timing summary

Minimum period	: 93.191ns (Maximum Frequency: 10.731MHz)
Minimum input arrival time before clock	: 5.176ns
Maximum output required time after clock	: 4.283ns
Maximum combinational path delay	: No path found

Table 6.8: Thermal summary

Estimated junction temperature:	27^0C
Ambient temp:	25^0C
Case temp:	26^0C
Theta J-A range:	26 – 26^0C/W

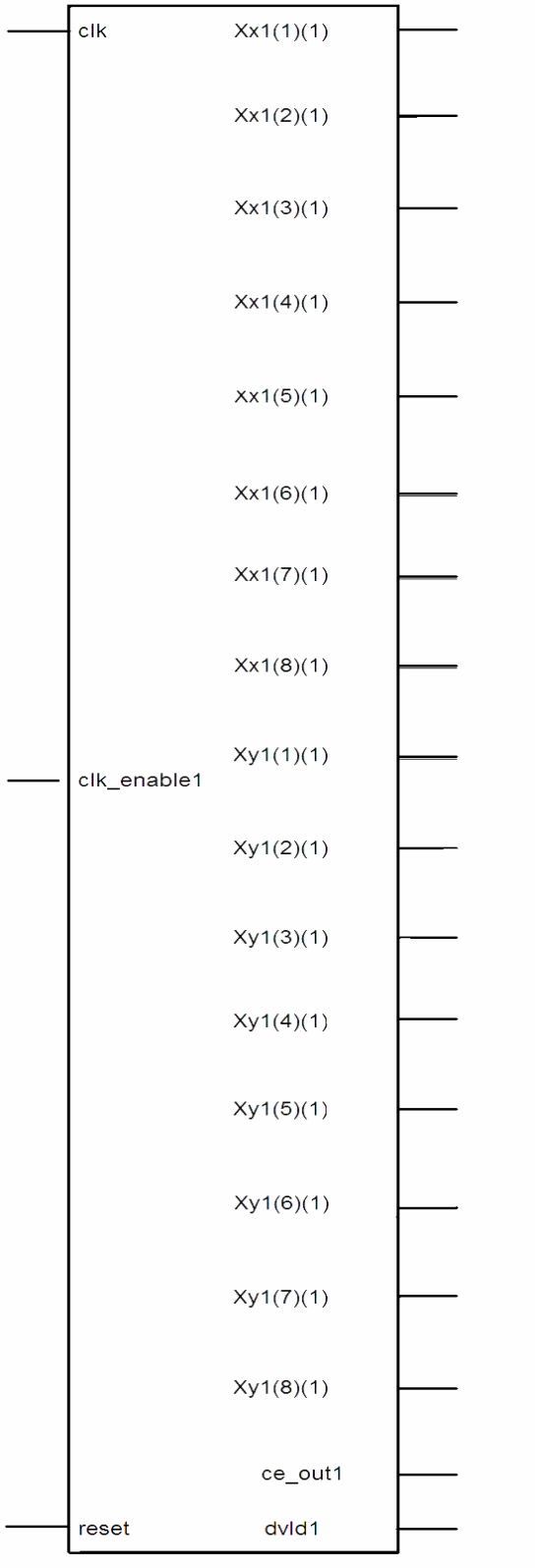


Figure 6.6: Top level RTL schematic of DFT

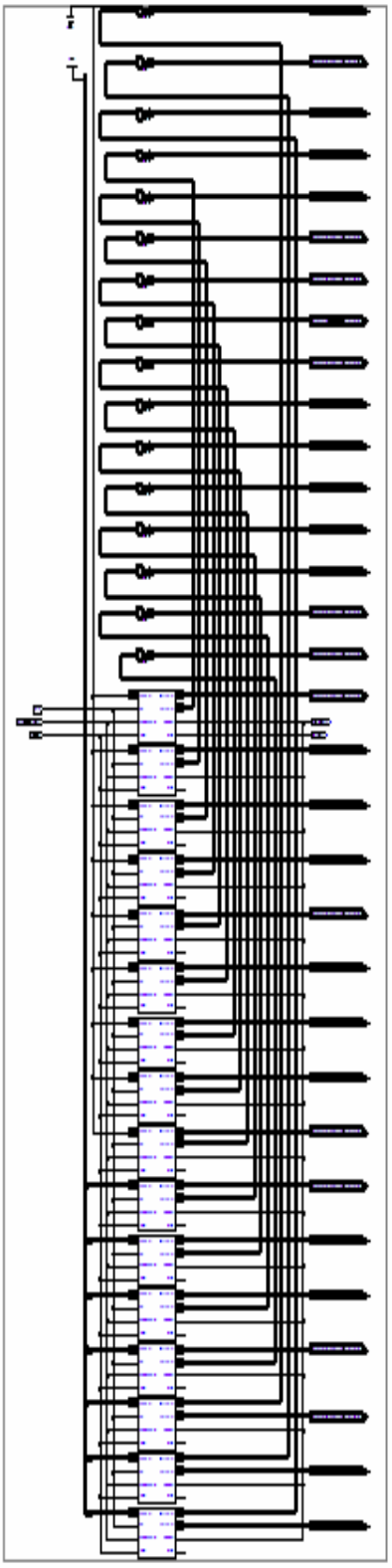
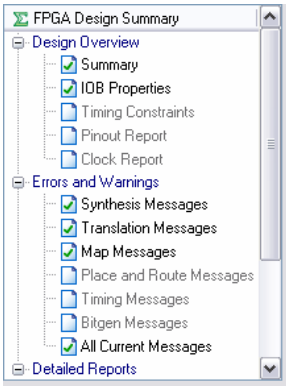


Figure 6.7: RTL schematic of DFT

Table 6.9: (a) Design summary for DFT

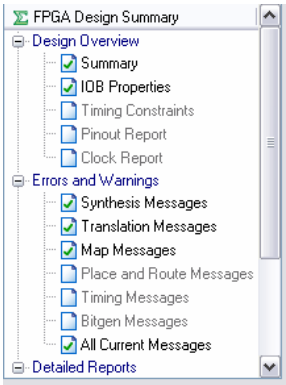


FFT Project Status			
Project File:	vikasfft.ise	**Current State:**	Mapped
Module Name:	dft_based_on_cordic_n_n	**• Errors:**	No Errors
Target Device:	xc3s500e-4fg320	**• Warnings:**	467 Warnings
Product Version:	ISE 9.2i	**• Updated:**	Mon Jul 14 02:25:12 2008

VIKASFFT Partition Summary
No partition information was found.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	1,945	9,312	20%	
Number of 4 input LUTs	20,547	9,312	220%	OVERMAPPED
Logic Distribution				
Number of occupied Slices	11,319	4,656	243%	
Number of Slices containing only related logic	10,775	11,319	95%	
Number of Slices containing unrelated logic	544	11,319	4%	
Total Number of 4 input LUTs	22,275	9,312	239%	OVERMAPPED
Number used as logic	20,547			
Number used as a route-thru	1,728			
Number of bonded IOBs	3,077	232	1326%	OVERMAPPED
IOB Flip Flops	192			
Number of bonded Out/Bidir IOBs	3,074	176	1746%	OVERMAPPED

Table 6.9: (b) Design summary for DFT



Number of Slices containing only related logic	10,775	11,319	95%	
Number of Slices containing unrelated logic	544	11,319	4%	
Total Number of 4 input LUTs	22,275	9,312	239%	OVERMAPPED
Number used as logic	20,547			
Number used as a route-thru	1,728			
Number of bonded IOBs	3,077	232	1326%	OVERMAPPED
IOB Flip Flops	192			
Number of bonded Out/Bidir IOBs	3,074	176	1746%	OVERMAPPED
Number of bonded Input IBUFs	3	56	9%	
Number of GCLKs	1	24	4%	
Number of MULT18X18SIOs	20	20	100%	
Total equivalent gate count for design	242,654			
Additional JTAG gate count for IOBs	147,696			

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Mon Jul 14 02:08:02 2008	0	467 Warnings	0
Translation Report	Current	Mon Jul 14 02:22:48 2008	0	0	0
Map Report	Current	Mon Jul 14 02:25:12 2008	No Errors	0	3 Infos
Place and Route Report					
Static Timing Report					
Bitgen Report					

Table 6.10: Advanced HDL Synthesis Report for DFT

Macro Statistics

10x24-bit multiplier	: 16
24x10-bit multiplier	: 16
24x24-bit multiplier	: 32
32x32-bit multiplier	: 32
# Multipliers	**: 96**
10-bit adder	: 32
2-bit adder	: 64
24-bit adder	: 80
24-bit subtractor	: 64
3-bit adder	: 64
32-bit adder	: 46
32-bit subtractor	: 28
4-bit adder	: 64
5-bit adder	: 64
6-bit adder	: 64
7-bit adder	: 64
8-bit adder	: 80
9-bit adder	: 64
# Adders/Subtractors	**: 778**
Flip-Flops	: 2576
# Registers	**: 2576**
24-bit comparator greatequal	: 32
24-bit comparator lessequal	: 48
8-bit comparator greatequal	: 16
# Comparators	**: 96**
32-bit 4-to-1 multiplexer	: 32
# Multiplexers	**: 32**

6.3 Discussions

Multipliers used for implementation of CORDIC algorithm for sine cosine generation are 73 in numbers. Number of adders/subtractors and registers are 79 and 616 respectively. In case of DFT implementation number of multiplier, adders/subtractors, registers, comparators, multiplexers are 96, 778, 2576, 96, 32 respectively.

Minimum period for sine cosine generation is 93.191 ns (maximum frequency 10.73 MHz) power consumed by the sine cosine generator and DFT generator is 81 mW each with the junction temperature of 27^0C. Total number of 4 input LUTs (Look Up Tables) used 708 and 20,547 for sine cosine generator and DFT calculator respectively. Total number of gates used 7,800 for sine cosine generator and 242,654 gates for 8×1 DFT generator.

CHAPTER 7

CONCLUSION

The CORDIC algorithm is a powerful and widely used tool for digital signal processing applications and can be implemented using PDPs (Programmable Digital Processors). But a large amount of data processing is required because of complex computations. This affects the cost, speed and flexibility of the DSP systems. So, the implementation of DFT using CORDIC algorithm on FPGA is the need of the day as the FPGAs can give enhanced speed at low cost with a lot of flexibility. This is due to the fact that the hardware implementation of a lot of multipliers can be done on FPGA which are limited in case of PDPs.

In this thesis the sine cosine CORDIC based generator is simulated using ModelSim which is then used for simulation of Discrete Fourier Transform. Then the implementation of sine cosine CORDIC based generators is done on XILINX Spartan 3E FPGA which is further used to implement eight point Discrete Fourier Transform using radix-2 decimation-in-time algorithm on FPGA. The results are verified by test bench generated by the FPGA. This thesis shows that CORDIC is available for use in FPGA based computing machines, which are the likely basis for the next generation DSP systems. It can be concluded that the designed RTL model for sine cosine and DFT function is accurate and can work for real time applications.

Future Scope of work

The future scope should include the following

- Implementation of dif algorithm
- DFT computation and simulation for more number of points
- Implementation and simulation for DHT, DCT and DST calculations

.

REFERENCES

[1] Volder J. E., "The CORDIC trigonometric computing technique", IRE Trans. Electronic Computing, Volume EC-8, pp 330 - 334, 1959.

[2] Lindlbauer N., www.cnmat.berkeley.edu/~norbert/CORDIC/node3.htmlhtt.

[3] Avion J.C., http://www.argo.es/~jcea/artic/CORDIC.htm

[4] Qian M., "Application of CORDIC Algorithm to Neural Networks VLSI Design", IMACS Multiconference on "Computational Engineering in Systems Applications (CESA)", Beijing, China, October 4-6, 2006.

[5] Lin C. H. and Wu A. Y., "Mixed-Scaling-Rotation CORDIC (MSR-CORDIC) Algorithm and Architecture for High-PerformanceVector Rotational DSP Applications", Volume 52, pp 2385- 2398, November 2005

[6] Walther J.S.A, "Unified algorithm for elementary functions", Spring Joint Computer Conference, pp 379 - 385, Alantic city, 1971.

[7] Kolk K. J. V., Deprettere E.F.A. and Lee J. A., " A Floating Point Vectoring Algorithm Based on Fast Rotations", Journal of VLSI Signal Processing, Volume25, pp 125–139, Kluwer Academic Publishers, Netherlands, 2000.

[8] Antelo E., LangT. and Bruguera J. D., "Very-High Radix CORDIC Rotation Based on Selection by Rounding", Journal of VLSI Signal Processing, Vol.25, 141–153, Kluwer Academic Publishers, Netherlands, 2000.

[9] Delosme M. J.,Lau C. Y. and Hsiao S. F., "Redundant Constant-Factor Implementation of Multi-Dimensional CORDIC and Its Application to Complex SVD", Journal of VLSI Signal Processing, Volume 25, pp 155–166, Kluwer Academic Publishers, Netherlands, 2000.

[10] Choi J. H., Kwak J. H. and Swartzlander, Journal of VLSI Signal Processing, Volume 25, Kluwer Academic Publishers, Netherlands, 2000.

[11] Roads C., "The Computer Music Tutorial", MIT Press, Cambridge, 1995.

[12] Rhea T., "The Evolution of electronic musical instruments", PhD thesis, Nashville: George Peabody College for Teachers, 1972.

[13] Goodwin M., "Frequency-domain analysis-synthesis of musical sounds", Master's thesis, CNMAT and Department of Electrical Engineering and Computer Science, UCB, 1994.

[14] Muller J. M., “Elementary Functions - Algorithms and Implementation”, Birkhauser Boston, New York, 1997.

[15] H. M. Ahmed, J. M. Delosme, and M. Morf, “Highly concurrent com- Comput. Mag., Volume 15, no. 1, pp. 65-82, Jan. 1982.

[16] Parhami B., “Computer Arithmetic – Algorithms and hardware designs,” Oxford University Press, New York, 2000.

[17] Considine V., “CORDIC trigonometric function generator for DSP”, IEEE-89th, International Conference on Acoustics, Speech and Signal Processing, pp 2381 - 2384, Glasgow, Scotland, 1989.

[18] Andraka R.A., “Survey of CORDIC algorithms for FPGA based computers”, Proceedings of the 1998 ACM/SIGDA sixth international symposium on FPGAs, pp 191-200, Monterey, California, Feb.22-24, 1998.

[19] http://www.dspguru.com/info/faqs/CORDIC.htm

[20] www.xilinx.com/partinfo/#4000.pdf

[21] Troya A., Maharatna K., Krstic M., Crass E., Kraemer R.,“OFDM synchronizer Implementation for an 1EEE 802.1 la Compliant Modem", Proc. IASTED International Conference on Wireless and Optical Communications, Banff, Canada,July 2002.

[22] Andraka R., “Building a high performance bit serial processor in an FPGA”, On-Chip System Design Conference, North Kingstown, 1996.

[23] http://comparch.doc.ic.ac.uk/publications/files/osk00jvlsisp.ps.

[24] Krsti M., Troyu A., Muharutnu K. and Grass E., “Optimized low-power synchronizer design for the IEEE 802.11a standard”, Frankfurt (Oder), Germany, 2003.

[25] Proakis J. G.,Manolakis D. G., “Digital signal processing principles, algorithms and applications”, Prentice Hall, Delhi, 2008.

[26] www.51protel. com/tech /Introduction.pdf