MODIFIED CORDIC ALGORITHM BASED DDFS ARCHITECTURE

A project report submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

Suggu Pavan kumar (318126512174)
Perumalla Sai Charan (318126512166)
Ayenala Kapil Vardhan (318126512123)
Shaik Md. Salman (318126512171)

Under the guidance of

Mr. N. Srinivas Naidu B. Tech, M. Tech, (Ph. d)

Assistant Professor

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(UGC AUTONOMOUS)

(*Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'B+' Grade*)

Sangivalasa, Bheemili mandal, Visakhapatnam dist. (A.P)

(2021-2022)

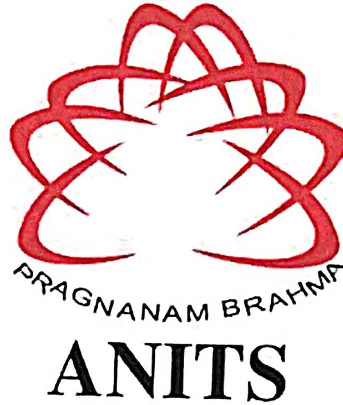
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(UGC AUTONOMOUS)

(*Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'B+' Grade*)

Sangivalasa, Bheemili Mandal, Visakhapatnam dist. (A.P)



ANITS

CERTIFICATE

This is to certify that the project report entitled "**MODIFIED CORDIC ALGORITHM BASED DDFS ARCHITECTURE**" submitted by **S. Pavan Kumar (318126512174), P. Sai Charan (318126512166), A.Kapil vardhan (318126512123), Shaik Mohammad Salman (318126512171)** in partial fulfillment of the requirements for the award of the Degree of **Bachelor of Technology** in **Electronics & Communication Engineering** of Andhra University, Visakhapatnam is a record of bonafide work carried out under my guidance and supervision.

Project Guide

Mr. N. Srinivas Naidu

Assistant Professor

Department of E.C.E

ANITS

Assistant Professor
Department of E.C.E.
Anil Neerukonda
Institute of Technology & Sciences
Sangivalasa, Visakhapatnam-531 162

Head of the Department

Dr. V. Rajya Lakshmi

Professor and HOD

Department of E.C.E

ANITS

Head of the Department
Department of E C E
Anil Neerukonda Institute of Technology & Sciences
Sangivalasa-531 162

// ACKNOWLEDGEMENT

We would like to express our deep gratitude to our project guide **Mr. N. Srinivas Naidu,** Associate Professor, Department of Electronics and Communication Engineering, ANITS, for his guidance with unsurpassed knowledge and immense encouragement. We are grateful to **Dr. V. Rajya Lakshmi**, Head of the Department, Electronics and Communication Engineering, for providing us with the required facilities for the completion of the project work.

We are very much thankful to the **Principal and Management, ANITS, Sangivalasa,** for their encouragement and cooperation to carry out this work.

We express our thanks to all **teaching faculty** of Department of ECE, whose suggestions during reviews helped us in accomplishment of our project. We would like to thank **all non-teaching staff** of the Department of ECE, ANITS for providing great assistance in accomplishment of our project.

We would like to thank our parents, friends, and classmates for their encouragement throughout our project period. Finally, we thank everyone for supporting us directly or indirectly in completing this project successfully.

PROJECT STUDENTS

S. Pavan kumar (318126512174)

P. Sai Charan (318126512166)

A. Kapil Vardhan (318126512123)

Shaik Mohammad Salman (318126512171)

LIST OF CONTENTS: -

LIST OF FIGURES V

LIST OF TABLES VII

LIST OF ABBREVATIONS VIII

ABSTRACT IX

1. INTRODUCTION 01-09

1.1 Generation of sine and cosine waves 01

1.1.1 Wien bridge oscillator 01

1.1.2 Phase shift oscillator 02

1.1.3 Colpitts crystal oscillator 03

1.1.4 Square wave and filter 03

1.1.5 Function generator 04

1.1.6 Phase based sine wave generator 05

1.1.7 Direct digital synthesis 06

1.2 Direct digital frequency synthesizer 07

1.3 CORDIC algorithm 07

1.4 Applications of CORDIC algorithm 08

1.5 Applications of DDFS 08

2. DIRECT DIGITAL FREQUENCY SYNTHESIZER 10-15

2.1 Design of basic DDFS 10

2.1.1 Accumulator 10

2.1.2 ROM or look up table 10

2.1.3 DAC or digital to analog converter 10

2.2 Working of basic DDFS 10

2.3 Phase accumulator 12

2.4 Phase to amplitude convertor (ROM or LUT) 12

2.5 Digital to analog convertor and filter 13

3. IMPROVED ARCHITECTURE OF DDFS 16-18

3.1 Design of a DDFS with reduced ROM 16
3.2 Spectral purity considerations 17
3.3 Cordic based DDFS architecture 17

4. CORDIC ALGORITHM 19-25
4.1 Introduction to CORDIC algorithm 19
4.1 working of CORDIC algorithm 19
4.2 Basic CORDIC iterations 21
4.3 Time shared architecture 23

5. MODIFIED CORDIC ALGORITHM 26-29
5.1 Limitations of basic CORDIC algorithm 26
5.2 Modification to the basic CORDIC algorithm 26
5.3 Recoding of binary representation 26
5.4 Hardware optimization 28

6. INTRODUCTION TO VERILOG 30-46
6.1 Introduction 30
6.2 Features of VERILOG HDL 30
6.3 Module declaration 31
6.3.1 Switch level modelling 32
6.3.2 Gate level modelling 32
6.3.3 Data flow modelling 33
6.4.4 Behavioral modelling 33
6.4 Software design and development 34
6.5 Software tools used 35
6.5.1 Xilinx vivado 35
6.5.2 Language support 35
6.5.3 MATLAB software 35
6.6 Xilinx Vivado ISE Design suite 35
6.7 ISE project navigator 36
6.7.1 Creating a new project 38
6.8 VERILOG design entry 41

6.8.1 Working through the basic project flow 41
6.8.2 Project manager 42
6.8.2.1 Project settings 42
6.8.2.2 Add sources 42
6.8.2.3 Define module 45

7. MATLAB 47-52

7.1 MATLAB introduction 47
7.2 The MATLAB system 47
7.2.1 Development environment 48
7.2.2 The MATLAB mathematical function library 48
7.2.3 The MATLAB language 48
7.2.4 Graphics 48
7.2.5 The MATLAB Application Function Interface 48
7.2.6 Starting MATLAB 48
7.2.7 MATLAB desktop 49
7.3 MATLAB working environment 49
7.3.1 MATLAB working desktop 49
7.3.2 Using MATLAB editor to create M-files 50
7.3.3 Getting help 50
7.4 saving and retrieving work sessions 51
7.4.1 Graph components 52
7.4.2 Plotting tools 52
7.4.3 Editor/debugger 52

8. SIMULATED OUTPUTS 53-66

8.1 Simulated outputs from MATLAB 53
8.1.1 MATLAB code for basic CORDIC algorithm 53
8.1.2 MATLAB code for modified CORDIC algorithm 55
8.2 Simulated outputs from VERILOG 60
8.2.1 VERILOG code for modified CORDIC algorithm 60
8.2.2 VERILOG code for test bench for modified CORDIC algorithm 63

CONCLUSION **67**

REFERENCES **68**

LIST OF FIGURES: -

CHAPTER	PAGE NO.
INTRODUCTION	**01-09**
1.1 Wien bridge oscillator	1
1.2 Phase shift oscillator	2
1.3 Colpitts crystal oscillator	3
1.4 Square wave filter	4
1.5 Function generator	5
1.6 Phase-based sine wave generator	5
1.7 Direct digital synthesis	6
2. DIRECT DIGITAL FREQUENCY SYNTHESIZER	**10-15**
2.1 Design of basic DDFS	11
2.2 Block diagram of DDFS architecture	13
2.3 Phase diagram	14
2.4 Phase detail	14
2.3 Waveform of different blocks of DDFS	15
3. IMPROVED ARCHITECTURE OF DDFS	**16-18**
3.1 Design of DDFS with reduced ROM size	16
3.2 CORDIC based DDFS architecture	18
4. CORDIC ALGORITHM	**19-25**
4.1 Unit vector circle	20
4.2 CORDIC algorithm incremental rotation by i+1	20
4.3 CORDIC architecture using feedback network	22
4.4 Pipelined FDA architecture of CORDIC algorithm	23
4.5 Time-shared with folding factor 16	23
4.6 Four slow folded architecture by a folding factor of 4	24
4.7 Timing diagram of CORDIC architecture which is of folding factor 4	25
5. MODIFIED CORDIC ALGORITHM	**26-29**

5.1 FDA of modified CORDIC algorithm 28

6. INTRODUCTION TO VERILOG 30-46

6.1 Xilinx Vivado project navigator window 37

7.2 Creating new project window 38

7.3 Guiding wizard for the project 38

7.4 Creating a project name 39

7.5 Specifying the RTL project 40

7.6 Choosing a board for project 40

7.7 Project summary 41

7.8 Main window for the project 42

7.9 Project settings window 42

7.10 Adding the source files 43

7.11 Wizard that shows to the design source 43

7.12 Creating a new file name for new design source 44

7.13 Selecting the type of file and location 44

7.14 Module defining with ports 45

7.15 Creating the simulation sources 46

8. SIMULATED OUTPUTS 53-66

8.1 Output showing results of sine and cosine wave using basic CORDIC 54

8.2 Output showing results of sine and cosine using modified CORDIC 57

8.3 Verilog output showing results of sine and cosine using modified CORDIC 64

8.4 Power report of synthesized design 64

8.5 RTL schematic of design 65

8.5.1 Calculation of sign (+, -) for angles(z) 65

8.5.2 Modified cordic block 66

8.6 Memory utilization of synthesized design 66

LIST OF TABLES: -

CHAPTER **PAGE NO.**

2. DIRECT DIGITAL FREQUENCY SYNTHESIZER **10-15**

2.1 Sine values stored in memory 15

3. IMPROVED ARCHITECTURE OF DDFS **16-18**

3.1 Storing of angle values w.r.t the MSB bits 17

4. CORDIC ALGORITHM **19-29**

4.1 Values in the LUT of angles w.r.t to the iterations 22

8. SIMULATED OUTPUTS **53-66**

8.1 Comparison of CORDIC and modified CORDIC cosine values 58

8.2 Comparison of CORDIC and modified CORDIC sine values 59

LIST OF ABBREVATIONS: -

CORDIC	-	Coordinate Rotation Digital Computer
DDFS	-	Direct Digital Frequency Synthesizer
CE	-	Cordic Element
DAC	-	Digital to Analog Converter
ROM	-	Read Only Memory
LUT	-	Look Up Table
DSP	-	Digital Signal Processing
DIP	-	Digital Image Processing
RTL	-	Register Transfer Level
HDL	-	Hardware Description Language
VHDL	-	Verilog Hardware Description Language
MATLAB	-	Matrix Laboratory
GUI	-	Graphical User Interface
ISE	-	Integrated Synthesis Environment
NCO	-	Numerically Controlled Oscillator
PA	-	Phase Accumulator
MSB	-	Most Significant Bits
FCW	-	Frequency Control Word
PLI	-	Programming Language Interface
FDA	-	Fully Dedicated Architecture
FPGA	-	Field Programmable Gate Array

ABSTRACT

This project explores architectures for the digital design of a direct digital frequency synthesizer (DDFS). This generates sine and cosine waveforms. The proposed DDFS is based on a Modified CoORDinate DIgital Computer (CORDIC) algorithm. The algorithm, through successive rotations of a unit vector, computes sine and cosine of an input angle θ. Each rotation is implemented by a CORDIC element (CE).

Coordinate Rotation Digital Computer (CORDIC) algorithm has greatly improved the efficiency of the hardware implementation of digital signal processing algorithms and other mathematical operations. While there exist quite a lot of redundant iterations in the Conventional CORDIC algorithm, this project proposes a novel efficient modified CORDIC algorithm combining the Conventional CORDIC algorithm with the modified CORDIC algorithm.

Key words: - CORDIC, DDFS, Time shared architecture

CHAPTER 1

INTRODUCTION

1.1 Generation of sine and cosine waves: -

The sine wave or cosine wave is a naturally occurring signal shape in communications and other electronic applications. Many electronic products use signals of the sine wave form. Audio, radio, and power equipment usually generates or processes sine and cosine waves. As it turns out, there are literally dozens of ways to generate a sine wave. Some of the popular methods used for generation of sine and cosine waves are:

1. Wien bridge oscillator
2. Phase shift oscillator
3. Colpitts crystal oscillator
4. Square wave and filter
5. Function generator
6. Pulse based sine and cosine wave generators
7. Direct digital synthesis

1.1.1 Wien bridge oscillator: -

A popular low frequency (audio, and up to about 100 kHz or so) sine wave oscillator is the Wien bridge shown in below figure.

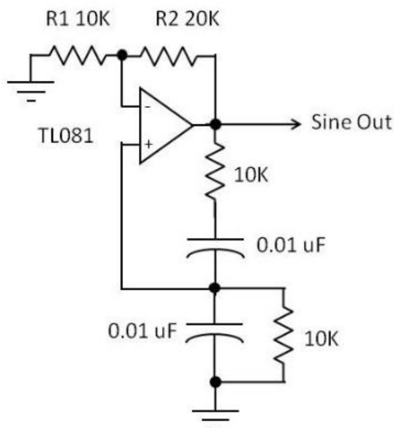


Fig 1.1 Wien bridge oscillator

It uses an RC network that produces a zero-degree phase shift from output back to the input, producing positive feedback that, in turn, produces oscillation. An op-amp is used to produce a gain of three that offsets the attenuation of the RC network. With a net closed loop gain of one, the circuit oscillates at a frequency determined by the values of the RC network:

$$f = 1/2\pi RC$$

This circuit works great and produces a very clean low distortion sine wave. Its problem is that instabilities in the gain and phase can cause the circuit to go out of oscillation completely, or go into saturation producing a clipped sine wave or square wave. Some compensation components are usually added to eliminate this problem.

A simple solution is to replace R1 with a small incandescent bulb whose resistance changes with current. As the output goes up, the bulb current and resistance increases, and reduces the gain to compensate. If the output goes down, the current decreases, lowering the resistance and increasing the gain to keep the output constant. One working example is to make R2 390 ohms and R1 a type 327 bulb. Other more elaborate schemes use an FET as a variable resistor to vary the gain. This circuit works and has a frequency of about 1,592 Hz. Output amplitude depends on the power supply voltages.

1.1.2 Phase shift oscillator: -

A popular way to make a sine or cosine wave oscillator is to use an RC network to produce a 180-degree phase shift to use in the feedback path of an inverting amplifier. Setting the gain of the amplifier to offset the RC network attenuation will produce oscillation. There are multiple variations of phase shifters, including a Twin-T RC network and cascaded RC high pass sections that produce either 45 or 60 degree shifts in each stage. The amplifier can be a single transistor, single op-amp, or multiple op-amps. Below figure shows phase shift oscillator.

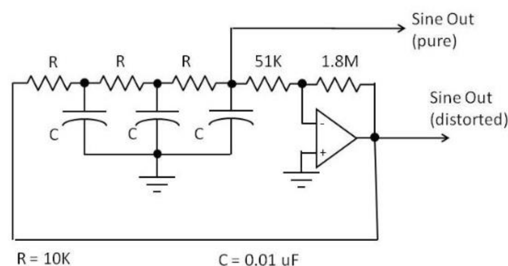


Fig 1.2 Phase shift oscillator

These oscillators produce a very pure low distortion sine wave. However, the frequency is fixed at the point where each RC section produces a 60-degree phase shift. That approximate frequency is:

$$f = 1/2.6RC$$

In the circuit shown above, the frequency should be about 3.85 kHz.

1.1.3 Colpitts crystal oscillator: -

Quartz crystals are often used to set the frequency of an oscillator because of their precise frequency of oscillation and stability. The equivalent circuit of a crystal is a series or parallel LC circuit. Below figure shows sine wave oscillator of the Colpitts type, as identified by the two-capacitor feedback network.

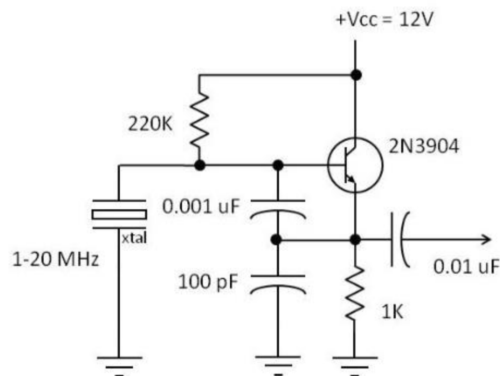


Fig 1.3 Colpitts crystal oscillator

This is another widely used circuit because it’s easy to implement and very stable. Its useful frequency range is approximately 100 kHz to 40 MHz. The output is a sine wave with a slight distortion. By the way, if you need a crystal oscillator with a sine wave out, you can usually buy a commercial circuit. They are widely available for almost any desired frequency. They are packaged in a metal can and are the size of a typical IC. The DC supply is usually five volts.

1.1.4 Square wave and filter: -

An interesting way to produce a sine wave is to select it with a filter. The idea is to generate a square wave first. As it turns out, it’s often easier to generate a square wave or rectangular wave than a sine wave. According to Fourier theory, the square wave is made up of a fundamental sine wave and an infinite number of odd harmonics.

For example, a 10 kHz square wave contains a 10 kHz sine wave, and sine waves at the 3rd, 5th, 7th, etc., harmonics of 30 kHz, 50 kHz, 70 kHz, and so on. The idea is to connect the square wave to a filter that selects the desired frequency.

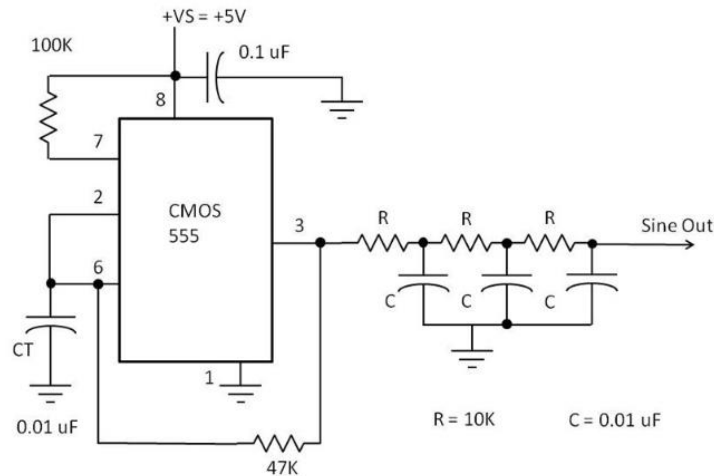


Fig 1.4 Square wave filter

A CMOS 555 timer IC produces a 50% duty cycle square wave. Its output is sent to a low pass RC filter that filters out the harmonics, leaving only the fundamental sine wave. Some distortion is common as it’s difficult to completely eliminate the harmonics. A more selective LC filter can be used to improve sine wave quality. Keep in mind that you can also use a selective band pass filter to pick out one of the harmonic sine waves. This circuit is designed for a frequency of 1,600 Hz

1.1.5 Function generator: -

A function generator is the name for a device that generates sine, square, and triangle waves. It may describe a piece of bench test equipment or an IC. One old but still good function generator IC is the XR-2206. It was first made by Exar in the 1970s, but is still around. If you need a sine wave generator that can be set to any frequency in the 0.01 Hz to 1 MHz or more, take a look at the XR-2206. Figure below shows the XR-2206 connected as a sine wave generator.

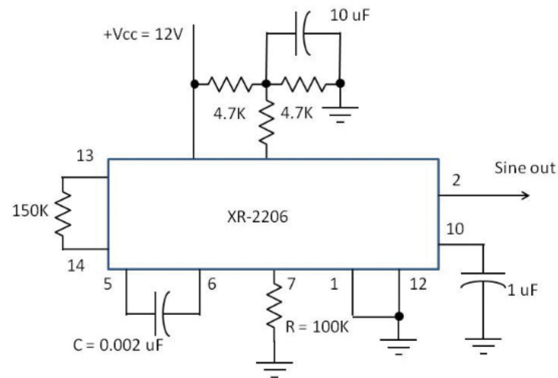


Fig 1.5 Function generator

The frequency is set by R and C and is calculated with the expression:

$$f = 1/RC$$

The internal oscillator generates a square wave and a triangle wave. The sine shaper circuit takes the triangle wave and modifies it into a sine wave. This is still a great chip. Besides the three common waveforms it generates, it can amplitude or frequency modulate them as well.

1.1.6 Phase-based sine wave generators: -

There are several other clever ways to make an approximate sine wave from pulses and filters. One way is to simply add together two square waves of the same amplitude where one is shifted 90 degrees from the other. A pair of JK flip-flops driven from opposite phase clock pulses can produce the two square waves to be added.

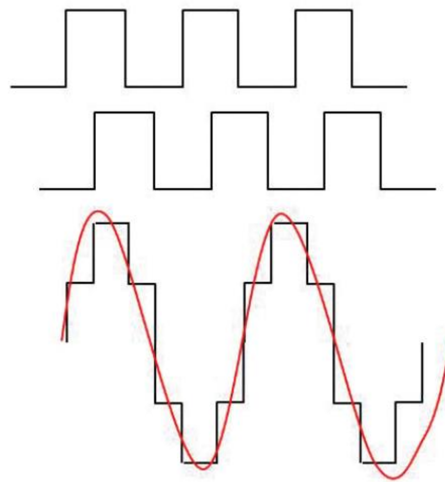


Fig 1.6 Phase-based sine wave generator

The result is a signal that can be used in some applications to replace a sine wave. Some crude DC-to-AC inverters use this method. The effect is an average power similar to what a sine wave would deliver to a load. Some RC or LC filtering can smooth the wave into a more continuous sine-like shape.

An interesting technique uses a sequence of varying width pulses that are filtered into a sine wave. If you apply a square wave with equal on and off times to a low pass filter, the output will be an average of pulse voltage over the on-off period. With a five-volt pulse, the average output over the full cycle of the wave would be 2.5 volts. By varying the pulse duration or width, different average voltages can be produced.

1.1.7 Direct digital synthesis: -

An interesting way to produce a sine wave is to do it digitally. Direct digital synthesis is one such technique. It begins with a read-only memory (ROM) that stores a series of binary values that represent values that follow the trigonometry equation for a sine wave. These values are then read out of the ROM one at a time and applied to a digital-to-analog converter (DAC). A clock signal steps an address counter that then accesses the sine values in ROM sequentially, and sends them to the DAC. The DAC generates an analog output signal that is proportional to the binary value from the ROM. What you get is a stepped approximation of a sine wave.

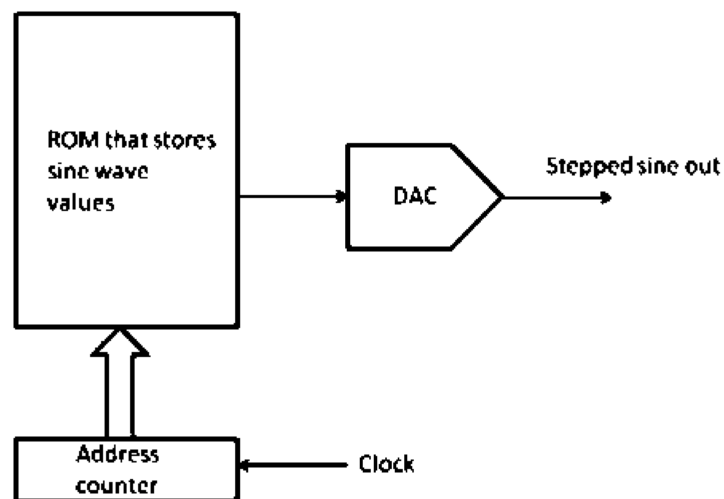


Fig 1.7 Direct Digital Synthesis

Direct digital frequency synthesizer is a direct digital synthesis method which is used to generate finest sine and cosine waves. This project mainly concentrates on direct digital frequency synthesizer (DDFS) and its working. Methods used in its architectures and refinements to the DDFS and CORDIC algorithm and at last a CORDIC based DDFS architecture.

1.2 Direct digital frequency synthesizer (DDFS): -

Direct Digital Frequency synthesizer: A DDFS an integral component of high-performance communication system. A DDFS generates a spectrally pure sine and cosine for quadrature mixing and frequency and phase correction in a digital receiver. A DDFS is characterized by its spectral purity. A measure of spectral purity is the spurious free dynamic Range (SFDR).

It is defined as the ratio of amplitude of the desired frequency to the highest frequency component of undesired frequency. The concept of DDFS was first project by J. Tierney in 1971.A DDFS can provide fast switching and high Frequency resolution, over a wide band of frequency. A major advantage of a direct digital synthesizer is that its output frequency, phase and amplitude can be precisely and rapidly manipulated under digital processor control. The DDFS addresses a variety of applications including Cable moderns, measurement equipment, arbitrary waveform. Direct digital Frequency synthesizer also known as Numerically controlled Oscillator (NCO) There are different methods of sine/cosine generation are reported with different merit and their limitations. These are with memory, reduced memory and memory less architecture. Few are DDFS with LUT, DDFS with sine and cosine function, interpolation based and parabola-based Taylor series based DDFS

1.3 CORDIC algorithm

This algorithm was developed by Volder in 1959 for computing the rotation of a vector in the Cartesian coordinate system. Initially DDFS architecture is implemented using Look Up Tables (LUT's) and also several algorithms and techniques have been proposed that reduce or complexity eliminate look up tables in memories. An efficient algorithm is CORDIC, (Coordinate Rotational Digital Computer) which uses rotation of vectors in Cartesian coordinates to generate angles of sine and cosine. This method also extended for computation of hyperbolic functions, exponentials and algorithms. Operations required in cordic algorithm are addition, subtraction, bit shift and it also uses very few look up tables.

1.4 Applications of CORDIC: -

Cordic has a variety of applications and widely used in our day-to-day life some of the applications of cordic algorithm are mentioned below:

- Signal And Image Processing
- Communication Systems
- Robotics
- 3D Graphs
- 8087 math co-processor
- HP 35 calculator
- Aerospace Application
- Different DSP And DIP Filters
- Network Security
- Biometric
- RADAR signal processor

1.5 Applications of DDFS: -

DDFS has a wide range of applications. It is used for communication, instrumentation, lab-on-chip, electronic measuring device etc.… it has a huge advantages of low power consumption, tunable frequency with sub hertz resolution, fast frequency switching and simple design.

The ability to generate arbitrary frequencies with accuracy and stability, limited only by the oscillator used to clock the phase accumulator. Crystal oscillators, depending on their specifications, can deliver tolerances of 50 parts per million to ~0.1 part per billion, making DDFS extremely accurate. Analog signal generators can only deliver accuracy and stability of a few tenths of a percent unless using a high-end device.

The frequencies provided by DDFS are repeatable. Loading the tuning word register with the value corresponding to frequency F1 generates a signal at frequency F1. If the tuning register is then loaded with the value for frequency F2, the output signal is quickly changed to frequency F2. When the tuning register is reloaded with the value for F1, the

exact the same frequency F1 is provided as was generated before. Analog generators can't guarantee this precision.

High frequency resolution can be achieved with the digital techniques used in DDFS. Increasing the resolution is as simple as adding more bits to the least significant end of the phase accumulator and tuning register. Analog waveform generators, which depend on mechanical components like potentiometers and variable capacitors to tune the oscillator, are limited in the resolution they can provide.

This ability to quickly change the output frequency with precision is also essential in communication techniques like spread-spectrum frequency hopping where radio signals are transmitted by rapidly switching a carrier among many frequency channels. Being able to reproduce exact frequencies and deliver frequency changes quickly forms the basis of the modulation technique.

CHAPTER 2

DIRECT DIGITAL FREQUENCY SYNTHESIZER

2.1 Design of basic DDFS

Direct digital frequency synthesis (DDFS) is a method of producing an analog waveform—usually a sine wave— by generating a time-varying signal in digital form and then performing a digital-to-analog conversion. The operations within a DDFS device are primarily digital, therefore, it can offer fast switching between output frequencies, fine frequency resolution, and operation over a broad spectrum of frequencies. The digital frequency synthesis approach employs a stable source frequency i.e., reference clock to define times at which digital sinusoidal sample values are produced. These samples are converted from digital to analog format and smoothed by reconstruction filter to produce analog frequency signals A Standard DDFS architecture consists of a phase accumulator, a ROM / lookup table, a DAC and some reconstruction filters. The phase accumulator combines the reference frequency and the value in the tuning word register. The output from the DAC is usually applied to filters to smooth the waveform and remove any extraneous output.

2.1.1 Accumulator: -

An accumulator in the DDFS keeps computing the next angle for the CORDIC to compute the sine and cosine values.

2.1.2 ROM or look up table: -

ROM serves as a lookup table, converting its index (phase) input to sine or cosine amplitude samples

2.1.3 DAC on Digital to analog convertor: -

It converts the digital value to its corresponding analog voltage output.

2.2 working of DDFS architecture: -

The tuning word is used to change the output frequencies during operation. The tuning word is a binary value held in the tuning register. The value of the tuning word is added to the phase accumulator with every clock update. For example, if the tuning word is set to 1, every clock interval increments the phase accumulator by 1. Setting the tuning word to 2, every clock cycle increments the phase accumulator by 2. Since the phase accumulator provides the phase value for the phase-amplitude lookup, the tuning word controls the number of values

retrieved from the phase-amplitude table for a cycle. With a tuning word of 1, every value in the table is retrieved. A tuning word of 2 reads every other value and also causes the accumulator to clock through to zero twice as fast, with the result that the output frequency has been doubled.

As an example, consider a DDFS designed with a phase-amplitude table of 360 entries, holding the amplitude (voltage) values for each one of the 360 degrees of a sine wave. The accumulator resets after 360 clock cycles. The reference frequency will be pulled from the system clock, so everything is clocked and updated at the same rate. With a tuning word of 1, the phase accumulator is incremented by 1 for every clock, and the table values are retrieved in order. Every 360 reference clocks, the accumulator resets and another waveform are created. Setting the tuning word to 2 has the result of reading every other value from the phase-amplitude table; the accumulator clocks through twice as fast and the output frequency is doubled. Of course, using only 360 values would produce a choppy output and the jitter would make it unusable. DDFS systems typically have phase-amplitude tables with thousands of data points and 16-bit registers for the tuning register and phase accumulators. A frequency control word W in every clock cycle of frequency f_{clk} is added in an N bit phase accumulator. If W=1, it takes the clock 2^N cycles to make the accumulator overflow and starts again. The DDFS can generate any frequency f_0 by an appropriate selection of W using

$$f_0 = W^* f_{clk}/2^N$$

The above equation is called as DDFS "tuning frequency". The digital signals cos ($\omega_0 n$) and sin ($\omega_0 n$) can be input to a D/A converter at sampling rate T = 1/ f_{clk} for generating analog sinusoids of frequency f_0. The maximum frequency from the DDFS is constrained by the Nyquist sampling criterion equal to $f_{clk}/2$

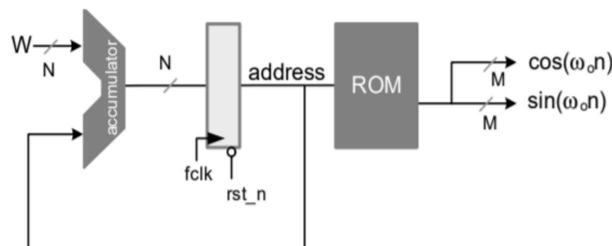


Fig 2.1 Design of basic DDFS

2.3 Phase accumulator

A binary phase accumulator consists of an N bit binary adder and register and each block produces a new N bit output consisting of the previous output obtained from the register sum, with the frequency control word (FCW). The resulting output wave form is a staircase with some stem size The phase accumulator (PA) is basically a counter that increments its digital output value each time it receives a clock pulse the magnitude of the increment to determine by the binary coded input (W). This word forms the phase step size between reference clock updates; it effectively sets how many points to skip around the phase wheel. The larger the jump size, the faster the phase accumulator overflows and completes the equivalent of a sine-wave cycle. The number of discrete phase points contained in the wheel is determined by the resolution of PA (n bits) which determines the tuning resolution of the DDFS.

for example: -for an n = 28-bit phase accumulator will have a value of 0000…0001, which would cause the phase accumulator to overflow after $\mathbf{2^{28}}$ reference clock cycles increments. With the value of w is changed to 0111…1111, phase accumulator will overflow after only 2 reference - clock cycles. A change to the value of w results in immediate and phase continuous changes in the output Frequency. In a DDFS as the output frequency increased, the number of samples per cycle decreases. Since, sampling theory, dictates that at least two Samples per cycle are required to reconstruct the output waveform, the maximum fundamental output frequency of a DDFS is $\boldsymbol{f_{clk}/2}$ However, for practical applications, the output frequency is limited to somewhat less than that, improving the quality of the reconstructed waveform and permitting filtering on the output when generating a constant frequency, the output of PA increases linearly

2.4 Phase to amplitude converter: - (ROM or LUT)

The DDFS's Rom is a sine Lookup converts digital phase input from the accumulator to output amplitude. The accumulator output represents the phase of the wave as well as an address to a word which is the corresponding amplitude of the phase in the LUT. This phase amplitude from the ROM LUT drive the PAC to provide an analog output. It is also called a digital phase-to-Amplitude convertor (PAC), or polar to rectangular transformation. (or)

sine waveform mapping device a memory. The lookup memory contains one cycle of the waveform to be generated. The size of LUT is $\mathbf{2^n}$ words LUT translates truncated phase information being in digital form, into quantized numerical waveform samples. Some DDFS systems can be implemented with ROM (or) without ROM. The advantages of ROM less architecture can be seen when high bit accuracy is desired. The ROM LUT stores the values of phase amplitudes while ROM less amplitude architecture computes phase amplitudes.

2.5 Digital to Analog convertor and filter:

The phase accumulator computes a phase (angle) address for the look up table, which outputs the digital value of amplitude corresponding to the time of that phase angle to the DAC. The DAC, in turn converts the number to a corresponding value of analog voltage (or) current. The DAC adds the rest of the system run at the same reference clock for synchronisation The DAC adds quantization error at the output to the sine wave. It removes the extra frequency components added to the sine wave and hence produces a smooth sine wave.

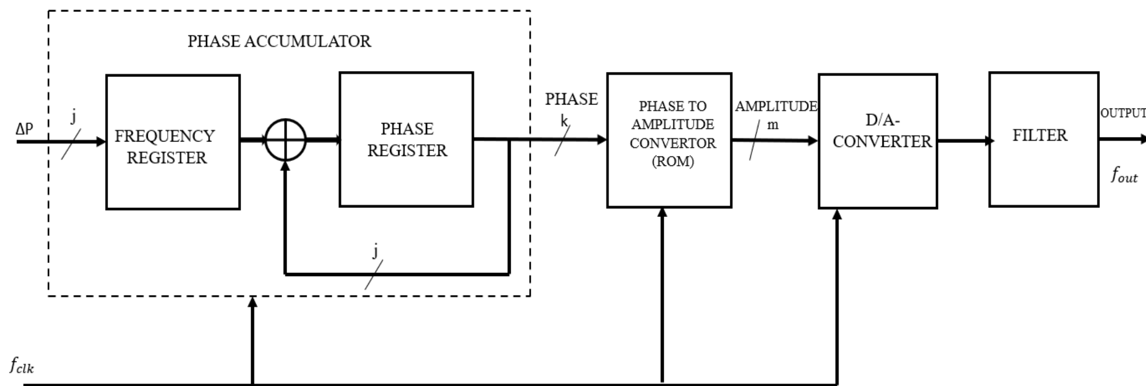


Fig.2.2 Block diagram of DDFS architectures

Figure 2.3 shows a basic trigonometric phase diagram where a sine wave is shown as a projection from a circle representing the phase of the waveform. The maximum voltage amplitude for the sine wave is the radius of the circle. For this discussion, we'll take the maximum voltage to be one for simplicity. As the phase angle Θ advances counter clockwise, there is a corresponding value of voltage. One complete rotation is 2*Pi radians. No matter how many times around, the same voltage corresponds a specific angle Θ. The frequency of

the sine wave produced depends on how quickly rotations through 2*Pi are completed (the angular velocity, ω).

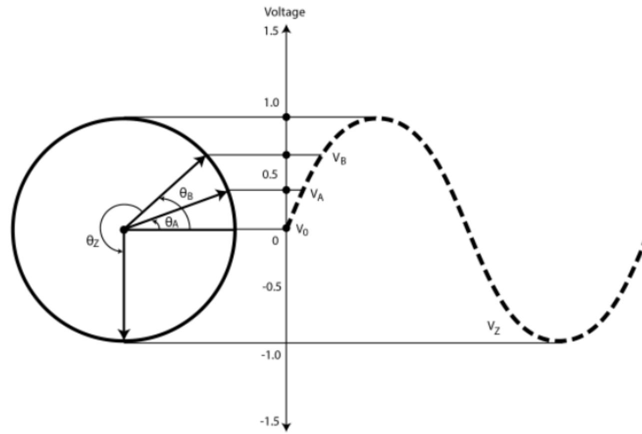


Fig 2.3 phase diagram

Figure 2.4 shows how for each phase, the specific voltage is sampled. The more points provided for the waveform by the sampling techniques used, the more definition the waveform has. The phase-amplitude table holds the phase/voltage points for each waveform and functions as phase to voltage converter.

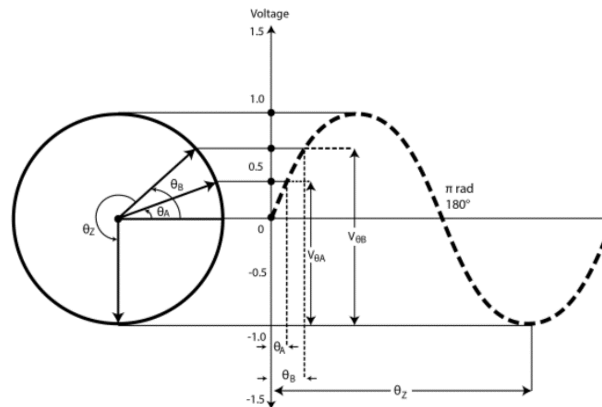


Fig 2.4 phase detail

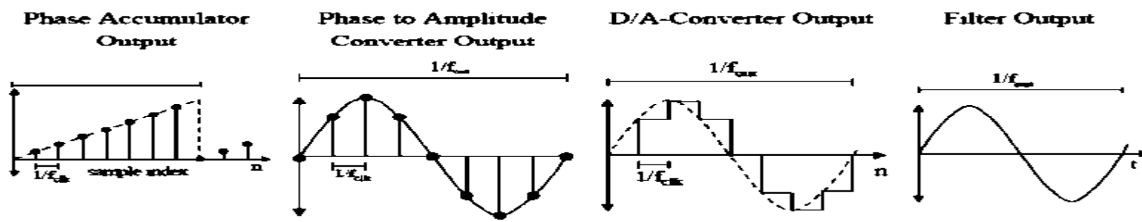


Fig.2.3 waveform of different blocks of DDFS

The above figure describes the different set of outputs obtained by the various blocks present in the DDFS architecture. The first waveform is the output obtained from phase accumulator. The next waveform is the phase amplitude converter output. The D/A converter is used to convert the digital signal to analog signal. The final output obtained is the sine wave came out from the filter output.

EXAMPLE: -

Let our required frequency be $f_o = 1khz$ and let N=5 bits and for easy calculation $f_{clk} = 32khz$

Now $$f_0 = \mathrm{W}*\frac{f_{clk}}{2^N}$$

with the given values therefore,

W=1 If initially, let N bit number be N=0000 This ‘N’ is used as an index to ROM, now o/p will be 0
For next clk pulse, N = 00001 now o/p = 0.0871
For next clk pulse, N = 00010 now o/p = 0.173
For next clk pulse, N =00011 now o/p = 0.2588
For next clk pulse, N = 00100 now o/p = 0.342. So, to generate a value of 20^0,it takes 4 cycles

Sin (θ)	address
Sin (0) =0	0000
Sin (5) = 0.0871	0001
Sin (10) = 0.173	0010
Sin (15) = 0.2588	0011
Sin (20) = 0.342	0100

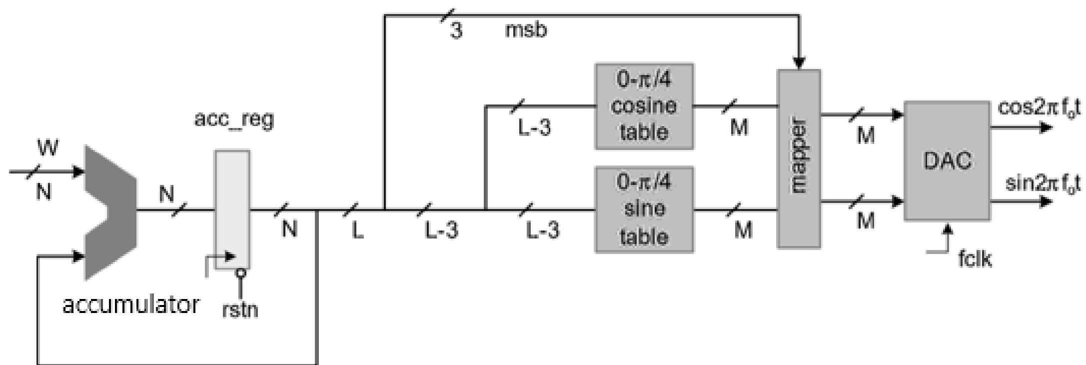
Table 2.1 Sine values stored in memory

CHAPTER 3

Improved architecture of DDFS

3.1 Design of DDFS with reduced ROM size

The basic design of DDFS is improved by exploiting the symmetry of sine and cosine waves. The output of the accumulator is truncated from N to L bits to reduce the memory requirement. A complete period of 0 to 2π of sine and cosine waves can be generated from values of the two signals from 0 to π/4. The sizes of the two memories are reduced by one eighth by only storing the values of sine and cosine from 0 to π/4 The L-3 bits are used to address the memories and then three most significant bits (MSBs) of the address are used to map the values to generate complete periods of cosine and sine. A ROM/RAM, based DDFS requires 2^{L-3} deep memories of width M. The design takes up a large area and dissipates significant power



3.1 Design of DDFS with reduced ROM size

In reduced memory concept, L-3 bits are used to store the values of cosine and sine values from (0 to **π**/4) and '3' most significant bits are used to map the values of remaining angles to the values stored in LUT's i.e.

3 MSB bits	values
000	0-π/4
001	π/4-π/2
010	π/2-3π/4
011	3π/4-π
100	π-5π/4
101	5π/4-3π/2
110	3π/2-7π/4
111	7π/4-2π

Table 3.1: - Storing of angle values w.r.t the MSB bits

3.2 Spectral purity considerations

The fidelity of a signal formed by recalling. samples of a sinusoid from a LUTs are affected by both the phase and amplitude quantization of the process. The length and width of the look-up table affect the signal's phase angle resolution and the signal's amplitude respectively. In conjuction with the system clock frequency. PA width determines the frequency resolution of the DDFS. The PA must have a sufficient field width to span the desired frequency resolution. For most practical application a large number of bits are allocated to the phase accumulator in order to satisfy the system frequency resolution requirements.

3.3 CORDIC based DDFS architecture: -

Several algorithms and techniques have been proposed that reduce or completely eliminate look up tables in memories an efficient algorithm is CORDIC which uses rotation of vectors in cartesian coordinates to generate value of sine and cosine. The CORDIC algorithm takes angle θ in radians, whereas the DDFS accumulator specifies the angle as an index. value. To use a CORDIC block in DDFS a CSD multiplier is required to convert index N to angle θ in radians

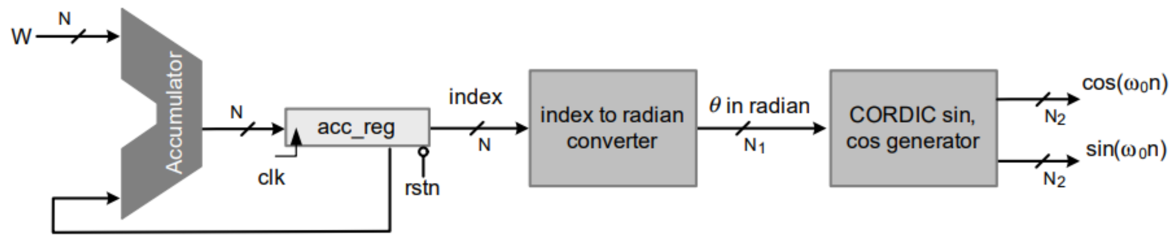


Fig 3.2 CORDIC based DDFS architecture

In the above figure cordic block needs an angle as an input and this angle is provided by the index to radian converter. Here the accumulator initially stores a certain value and W is a frequency control word which adds to the accumulator data and the result is stored in another accumulator register as the index (converted to required angle). So, at every clock pulse a new index is generated and this index is converted to an angle using the formula

$$\theta = \frac{\text{index(N)}}{2^N} * 2\pi$$

The cordic algorithm is one of the efficient algorithms and it generates the cosine and sine of angle in digital form and it is converted in analog form using digital to analog converter (DAC).

CHAPTER 4

CORDIC Algorithm

4.1 Introduction to CORDIC algorithm: -

The digital signal processing landscape has long been dominated by the microprocessors with enhancements such as single cycle multiply-accumulate instructions and special addressing modes. While these processors are low cost and offer extreme flexibility, they are often not fast enough for truly demanding DSP tasks. The advent of reconfigurable logic computers permits the higher speeds of dedicated hardware solutions at costs that are competitive with the traditional software approach. Unfortunately, algorithms optimized for these microprocessors-based systems do not map well into hardware. While hardware efficient solutions often exist, the dominance of the software systems has kept these solutions out of the spotlight. Among these hardware-efficient algorithms is a class of iterative solutions for trigonometric and other transcendental functions that use only shifts and adds to perform. The trigonometric functions are based on vector rotations, while other functions such as square root are implemented using an incremental expression of the desired function.

The trigonometric algorithm is called CORDIC an acronym for Coordinate Rotation Digital Computer. The incremental functions are performed with a very simple extension to the hardware architecture and while not CORDIC in the strict sense, are often included because of the close similarity. The CORDIC algorithms generally produce one additional bit of accuracy for each iteration. The trigonometric CORDIC algorithms were originally developed as a digital solution for real time navigation problems.

The original work is credited to Jack Volder. The CORDIC algorithm has found its way into diverse applications including the 8087-math coprocessor, the HP-35 calculator, radar signal processors and robotics. CORDIC rotation has also been proposed for computing Discrete Fourier, Discrete Cosine, Singular Value Decomposition and solving linear systems.

4.2 working of CORDIC algorithm: -

The main idea of cordic algorithm is to rotate a unit vector continuously for a fixed number of times. And when that unit vector reaches a particular position in coordinate system its projection on x-axis(x-coordinate) gives cosine value of that angle and its projection on y-

axis(y-coordinate) gives sine value of that angle. To bring a unit vector to the desired angle, the basic cordic algorithm undergoes certain recursive rotations.

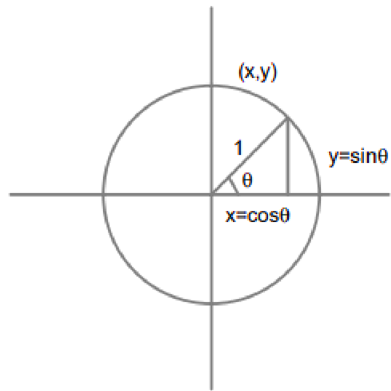


Fig.4.1 Unit vector circle

Let initially the coordinates of vector be x(i), y(i) and after rotating the vector by an angle be the new coordinates be x(i+1), y(i+1).

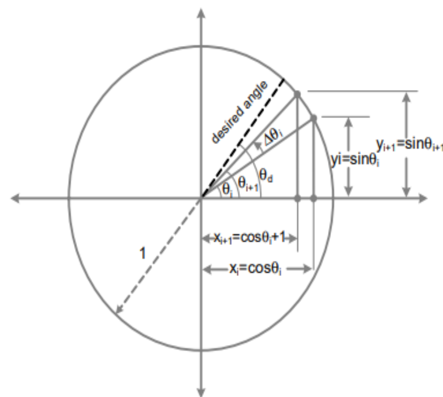


Fig. 4.2 CORDIC algorithm incremental rotation by i+1

Let x(i) = cosθ, y(i) = sinθ

x(i+1) = cos (θ - α)

therefore, $x(i+1) = x(i)\cos\alpha + y(i)\sin\alpha$…... (1)

similarly,

y(i+1) = sin (θ - α)

therefore, y (i + 1) = $y(i)\cos\alpha - x(i)\sin\alpha$ …… (2)

In the above example, our vector is rotational clockwise(downwards), but sometimes vectors may be rotated anticlockwise(upwards) then equations are

$$x(i+1) = \cos\alpha.(x(i) - y(i)\tan\alpha) \ldots (3)$$

$$y(i+1) = \cos\alpha.(x(i) + y(i)\tan\alpha) \ldots (4)$$

But we can choose the rotation angles $\alpha_1, \alpha_2, \ldots, \alpha_m$

So, choose $\alpha_i = tan^{-1}2^{-i}$

So that $\tan\alpha_i = 2^{-i}$

Now the ith step calculating (x_{i+1}, y_{i+1}) from (x_i, y_i)

Equations 3, 4 Can be written as

$$x_{i+1} = k_i(x_i - y_i d_i 2^{-i}) \ldots (5)$$

$$y_{i+1} = k_i(y_i + x_i d_i 2^{-i}) \ldots (6)$$

where $K_i = \cos\alpha_i = (\cos(tan^{-1}2^{-i}) = \frac{1}{\sqrt{(1+2^{-2i})}}$

and $d_i = \pm 1$

which is determined by the direction of the necessary rotation

Now the product, for n = 10 rotations

$$\pi\, k_i = 1/\pi(\sqrt{(1+2^{-2i})} = 0.6073$$

For every rotation of a vector forms a new angle i.e.

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

4.3 Basic CORDIC iterations: -

Pick α_i such that $\tan\alpha_i = d_i 2^{-i}, d_i = \{-1,1\}$

So finally basic CORDIC iterations are

$$x_{i+1} = x_i - d_i y_i 2^{-i} \ldots (7)$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \ldots (8)$$

$$z_{i+1} = z_i + d_i tan^{-1} 2^{-i} \ldots (9)$$

If we always pseudo rotate by the same set of angles (with + or –signs), then the expansion factor k is a constant that can be precomputed. So, for N=10, we need to store '10' values in Look up tables (LUT's). It contains the angles that the vector needs to rotated to reach the final desired angle. They are

i	$tan^{-1}2^{-i}$
0	45.0
1	26.6
2	14.0
3	7.1
4	3.6
5	1.8
6	0.9
7	0.4
8	0.2
9	0.1

Table 4.1: - Values in the LUT of angles w.r.t to the iterations

Example: - Let our required angle be 30 degrees. It can be achieved as

- 30 ≈ 45 - 26.6 + 14 - 7.1 + 3.6 + 1.8 - 0.9 + 0.4 - 0.2 + 0.1 = 30.1

Each iteration of the algorithm can be implemented as a CORDIC element (CE) This CE implements i^{th} iteration of the algorithm given by below figure

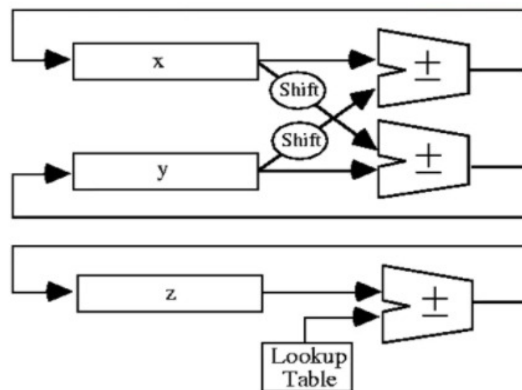


Fig. 4.3 CORDIC architecture using feedback network

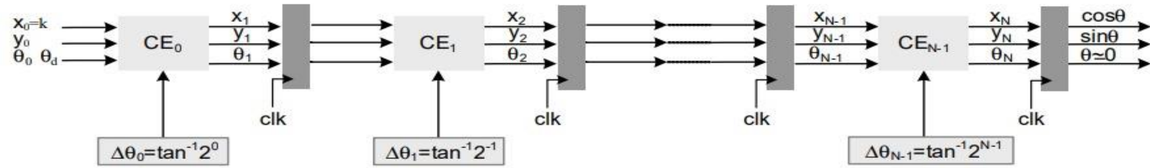


Fig.4.4 Pipelined FDA architecture of CORDIC algorithm

Here, instead of rotating a unit vector (1,0) from x-axis, we rotate a vector of magnitude k. i.e., we start from (k,0). This is because, if we rotate a unit vector, we get a multiplication factor of k at the end of all rotations.so to avoid use of multiplier, we rotate (k,0) vector

4.4 Time shared architecture: -

So far, we have seen that pipeline Fully Dedicated Architecture (FDA) and feedback network architecture. In feedback network architecture, previous values are sent back to the input of CORDIC element for calculating next values. However, it may require only one CORDIC element, but it increases latency. In pipelined FDA, all CORDIC elements are connected in cascade and it may involve large circuitry. So, A folded and time-shared architecture is more preferred. Folding factor here defines the number of times that the feedback is supplied to the input.

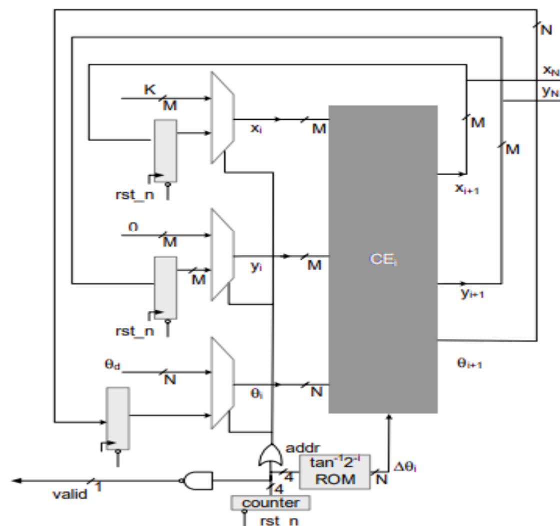


Fig.4.5 Time-shared architecture with Folding factors 16

C-slowed time-shared architecture means that it can be used for computing ‘C’ values(angles) and it can be better implemented using pipelining i.e., adding registers to each and every cordic elements. The architecture for ‘four slow folded architecture by a folding factor of ‘4’’ is shown below.

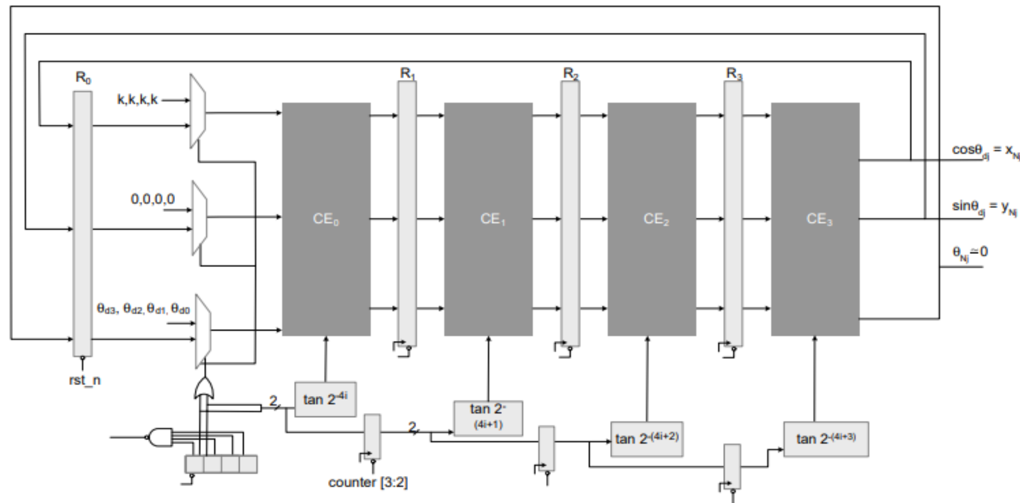


Fig.4.6 Four slow folded architecture by a folding factor of 4

The feedback register is replicated initially for ‘4’ times and after all the registers are retimed (set a different time) to reduce critical path. A simple counter-based controller is used to appropriately select the input to each cordic element (CE). Two MSB’s of counter are used as selection lines to three Multipliers at initial stage i.e., In the first four cycles, four desired angles (θ_{d0}, θ_{d1}, θ_{d2}, θ_{d3}) and values of x_0 and y_0 are given as input to CE_0. All the subsequent cycles feed the values from CE_3 to R_0 to CE_0. The working of algorithm for initial few cycles is shown in the below timing diagram.

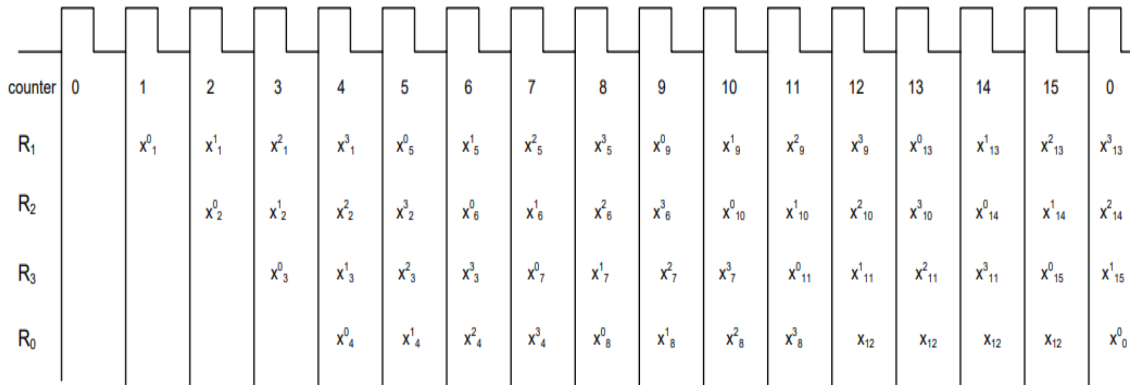


Fig.4.7 Timing diagram of CORDIC architecture which is of a folding factor 4

Initially Counter is loaded with 0000, and for 1st clock pulse R_1 register is loaded with values of first iteration for θ_{d0} angle. For 2nd clock pulse, R_2 register is loaded with values of second iteration for θ_{d0} angle and at the same clock pulse, R_1 is loaded with values of 1st iteration for θ_{d1} angle. Similarly for 3rd clock pulse, R_3 is loaded with values of third iteration for θ_{d0}, R_2 with second iteration for θ_{d1} and R_1 with values of first iteration for θ_{d2}. When 4th clock pulse is given, R_0 gets the values of 4th iteration for θ_{d0}, R_1With values of first iteration for θ_{d3}, R_2 with second iteration for θ_{d2} and R_3 with values of third iteration for θ_{d1}. After this, all four angles are, selection line of mux will be change. This process repeats for 16 clock cycles, and then counter overflows.

CHAPTER 5

Modified CORDIC algorithm

5.1 Limitations of basic CORDIC algorithm: -

In the basic cordic algorithm, it requires computation of d_i and only then it conditionally adds (or) subtracts one of the operands. That means for each iteration, d_i needs to be assigned with +1 (or) -1 which is time consuming. So, in order to avoid that limitation or simple modification is used which eliminates that computation of d_i at that particular instant and efficient parallel architectures can be realized.

Here, instead of calculating d_i at that particular instant, if we can calculate all the d_i values of a particular angle and store them previously, then these values (+1 or -1) can be directly used at that instant.

5.2 Modification to the basic CORDIC algorithm: -

A binary representation of a positive value of (considering θ in radian) for micro rotations can be considered as

$$\theta = \sum_{i=0}^{N-1} b_i 2^{-i} \, for \;\; b_i \epsilon \{0,1\}$$

That 'b' stores the sign (+ or -) of the operation to be performed at that instant. But this representation denotes either a positive rotation = 2^{-i} or no rotation depending on the value of bit b_i at location 'i' which makes the value of 'k' data dependent ['k' is computed based on the values of $\tan^{-1}(2^{-i})$]. So, to avoid that, it is necessary to recode the expression to use only +1 or -1.

5.3 Recoding of binary representation: -

The bits b_i in the expression can be recoded to r_i ϵ {+1, -1} as:

$$\theta = \sum_{i=0}^{N-1} b_i 2^{-i} = \sum_{i=0}^{N-1} r_i 2^{-(i+1)} + 2^0 - 2^{-N}$$

$$r_i = 2b_i - 1 \, where \;\; r_i \in \{+1, -1\}$$

This recoding requires first giving an initial fixed rotation θ_{init} to cater for the constant factor $(2^0 - 2^{-N})$ along with computing constant K as is done in the basic CORDIC algorithm. The recoding of $b_i s$ as ±1 helps in formulating K as constant and is equal to:

$$K = \prod_{i=0}^{N-1} \cos(2^{-(i)})$$

The rotation for θ_{init} can then be first applied, where:

$$\theta_{init} = 2^0 - 2^{-N}$$

$$x_0 = K\, cos\, (\theta_{init})$$

$$y_0 = K \sin(\theta_{init})$$

Therefore, the new iterations are

$$x_i = x_{i-1} - r_i tan 2^{-i} y_{i-1} \ldots (10)$$

$$y_i = y_{i-1} + r_i tan 2^{-i} x_{i-1} \ldots (11)$$

Here, unlike d_i,the values of r_i are predetermined, and these iterations do not include any computations of the α_i as are done in the basic CORDIC algorithm. But here in modified CORDIC, a multiplication factor $\tan^{-1}(2^{-i})$ is occurring in every iteration, but this is not advised. So, this multiplication factor can be avoided for stage i>4. i.e., $\tan^{-1}(2^{-i})$ can be approximated to 2^{-i} (from i>4)

$$tan 2^{-i} \approx 2^{-i}$$

So, it is necessary for us to pre compute all the initial four values and store them in the actual hardware implementation, the initial '4' iterations are skipped and the output value from the 4th iteration is directly indexed from memory. Therefore, the above approximation requires $\tan^{-1}(2^{-i})$ with 2^{-i}. The equations implementing simplified iteration for i=M+1, M+2…. N are:

$$x_i = x_{i-1} - r_i 2^{-i} y_{i-1} \ldots (12)$$

$$y_i = y_{i-1} + r_i 2^{-i} x_{i-1} \ldots (13)$$

And this modification results in simple fully parallel and time-shared hardware implementation as shown in the figure below

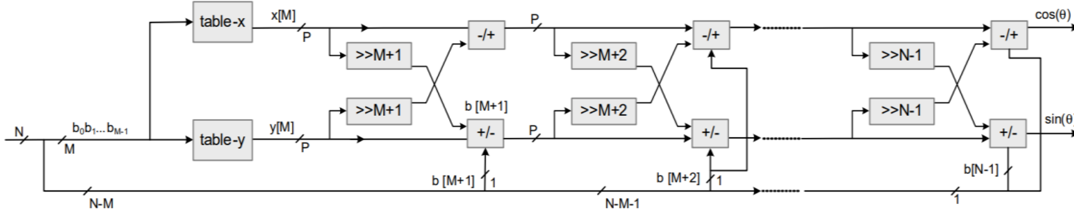


Fig.5.1 FDA of modified CORDIC algorithm

5.4 Hardware optimization: -

As the iterations now do not depend on the values of α_i the values of previous iterations can be directly substituted in to the current iteration. As x_4 and y_4 are known

For i = 5

$$x_5 = x_4 - r_5 2^{-5} y_4 \dots (14)$$

$$y_5 = y_4 + r_5 2^{-5} x_4 \dots (15)$$

For i = 6

$$x_6 = x_5 - r_6 2^{-6} y_5 \dots (16)$$

$$y_6 = y_5 + r_6 2^{-6} x_5 \dots (17)$$

Now by substituting equations (14), (15) values in i=6th iteration, we modify the equations (16), (17) as:

$$x_6 = (1 - r_5 r_6 2^{-11}) x_4 - (r_5 2^{-5} + r_6 2^{-6}) y_4 \dots (18)$$

$$y_6 = (1 - r_5 r_6 2^{-11}) y_4 + (r_5 2^{-5} + r_6 2^{-6}) x_4 \dots (19)$$

similarly, i=7th iteration can be calculated as

$$x_7 = (1 - r_5 r_6 2^{-11} - r_5 r_7 2^{-12} + r_7 r_6 2^{-13}) x_4 - (r_5 2^{-5} + r_6 2^{-6} + r_7 2^{-7} - r_5 r_7 r_6 2^{-18}) y_4$$

$$y_7 = (1 - r_5 r_6 2^{-11} - r_5 r_7 2^{-12} + r_7 r_6 2^{-13}) y_4 + (r_5 2^{-5} + r_6 2^{-6} + r_7 2^{-7} - r_5 r_7 r_6 2^{-18}) x_4$$

However, the terms including 2^{-x} with x > N will shift the entire value outside the range and they can be simply ignored. Therefore, the final iteration is

$$cos\theta = \left(1 - \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} r_i r_j 2^{-(i+j)}\right) x_4 - \left(\sum_{i=5}^{N-1} r_i 2^{-i}\right) y_4$$

$$sin\theta = \left(1 - \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} r_i r_j 2^{-(i+j)}\right) y_4 + \left(\sum_{i=5}^{N-1} r_i 2^{-i}\right) x_4$$

here $(i + j) \leq N$

So, if all the iterations of the cordic algorithm are merged into one expression and the final values can be effectively computed in a single cycle

CHAPTER 6

INTRODUCTION TO VERILOG

6.1 Introduction

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in design and verification of digital circuits at the regular -transfer level of abstraction. It is also used in verification of analog circuits and mixed signal circuits HDLs allows the design to be simulated earlier in the design circuits in order to correct errors or experiments with different architectures.

Designs described in HDL are technology independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits. Verilog can be used to describe designs at four levels if abstractions:

1) Algorithmic level (much like as code if, case and loop statements).
2) Register transfer level (RTL uses registers connected by Boolean equations)
3) Gate level (interconnected AND, NOR etc.).
4) Switch level (the switches are MOS transistors inside gates).

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output and bidirectional ports.

Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies).

6.2 Features of Verilog HDL

Verilog HDL offers many useful features for hardware design. Some of them are given below:

- Verilog (verify logic) HDL is general purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or

behavior code. Also, a designer needs to learn only one language for stimulus and hierarchical designs.

- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- All fabrication vendors provide Verilog HDL libraries for post logic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
- The Programming language interface (PLI) is a powerful feature that allows the user to custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their need with the Programming language interface (PLI).

6.3 Module Declaration

A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and port list (arguments). The next few lines specify the I/O type (**input**, **output** or **inout**) and width of each port. The default port with is 1 bit. Then the port variables must be declared **wire**, **reg**. The default is **wire**.

Typically, inputs are **wire** since their data is latched outside the module. Outputs are type **reg** if their signals were stored inside always or **initial** block

Syntax:

module model_ name(port_list); //module name, outputs and inputs are specified

input [msb: lsb] input_port_list; //inputs are taken here

output [msb: lsb] output_port_list; //outputs are mentioned here

inout [msb: lsb] inout_port_list;// inout ports are specified here

……. statements…………

endmodule

Example:

module add_sub (add, in1, in2, out); //module add_sub inputs and outputs are taken

input add; //defaults to wire

input [7:0] in1, in2**, wire** in1, in2; // inputs default to wire are taken here

output [7:0] out;

```
reg out;                          // output default to register is declared here

         .........statements.........
      Endmodule                 // end of the program
```

Verilog has four levels of modelling:

1) The switch level Modeling.
2) Gate- level Modeling.
3) The Data-Flow level.
4) The behavioral or procedural level.

6.3.1 Switch level Modeling

A circuit is defined by explicitly showing how to construct it using transistors like pmos and nmos, predefined modules.

Example:

```
module inverter (out, in); // module invertor, output and input ports are declared
output out;                // output port is declared
input in;                  // input port is declared
Supply0 gnd;               // supply of ground port is declared
Supply1 vdd;               // supply voltage port is declared
nmosx1 (out, in, gnd);     // nmos transistor logic with inputs and outputs is declared
pmosx2 (out, in, vdd);     // pmos transistor logic with outputs and inputs is declared
endmodule                  // end of the program
```

6.3.2 Gate level modelling: -

A circuit is defined by explicitly showing how to correct it using logic gates. Predefined modules, and the connections between them. In this first we think of our circuit as a box or module which is encapsulated from its outer environment, in such a way that its only communication with the outer environment, is through input and output ports. We then set out to describe structure within the module by explicitly describing its gates and sub modules, and how they connect with one another as well as to the module ports.

In other words, structural modelling is used to draw a schematic diagram for the circuit. As an example, consider the full-adder below.

Example:

```
module fulladder (a, b, sum, Cout); // module full adder is declared
input a, b;                          // input ports are declared
output sum, Cout;                    // output ports are declared
xor x1(a, b, y);                     // xor gate is initiated
xor x2(a, b, y);                     // xor gate is initiated
endmodule                            // end of the program
```

6.3.3 Data-flow modelling

Dataflow modeling uses Boolean expressions and operators. In this we use assign statement.

Example:

```
module fulladder (a, b, sum, Cout); // module full adder is declared
input a, b;                          // input ports are declared
output sum, Cout;                    // output ports are declared
assign sum=a^b;                      // sum is assigned to the necessary operation
assign Cout=a^b;                     // Cout is assigned to the necessary operation
endmodule                            // end of the program
```

6.3.4 Behavioral modeling

It is higher level of modeling where behavior of logic is modelled. Verilog behavioral Code is inside procedure blocks, but there is an exception: some behavioral code also exists outside procedure blocks.

There are two types of procedural blocks in Verilog

Initial: initial blocks execute only once at time zero (start execution at time zero)

Always: always blocks loop to execute over and over again; in other words, as other words as the name suggests, it executes always.

An always statement executes repeatedly, it starts and its execution at other 0 ns

Syntax: always@ (sensitivity list)

```
Begin
Procedural statements
end
```

Example:

```
module fulladder (a, b, clk, sum); // module full adder is declared
input a, b, clk;                   // input ports declared
output sum;                        // output ports declared
always@ (posedgeclk)               // always block is declared
begin                              // always block is begun
sum= a+b;                          // sum statement
endmodule                          // end of the program
```

6.4 SOFTWARE DESIGN AND DEVELOPMENT

A description of the hardware's structure and behavior is written in a high-level hardware description language (usually VHDL or Verilog and those codes is then compiled and downloaded prior to execution. Although schematic capture can be used for design entry but due to more complex designs and the improvement of the language-based tools it has become less popular.

The most distinct difference between hardware and software design is the way a developer must think about the problem. Software developers tend to think sequentially, even when they are tasked to program a multithread application. Most of the time, the source code is always executed in that order. At the design entry phase, hardware designers must think and program in parallel.

All of the input signals are processed in parallel: inside each ore is a series of macro cells and interconnections routed toward their destination output signals. Therefore, the statement of a hardware description language creates structures, all of which are process at the very same time. (Normally the link between each macro cell to another macro cell usually -synchronized to some other signal, like a common clock).

In a typical design, after each design entry is completed, the next step is to perform periods of functional simulation. This is where a simulator comes in place. It is used to execute the design and confirm that the correct/required outputs are produced for a given set of test inputs.

This step is to ensure the designer that his/her logic is functionally correct before going on to the next stage development. This is a good practice as compared to simulating a full-scale de

sign entry. As the design entry gets more complex, the troubleshooting will be much difficult and time consuming.

6.5 SOFTWARE TOOLS USED

6.5.1 Xilinx Vivado

Vivado enables developers to synthesize their designs, perform timing analysis examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Vivado is a design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors.

6.5.2 Language support

The Vivado High-Level Synthesis compiler enables C, C++ and System C programs to be directly targeted into Xilinx devices without the need to manually create RTL. Vivado HLS is widely reviewed to increase developer productivity, and is confirmed to support C++ classes, templates, functions and operator overloading.

Xilinx vivado enables simulation, verification and synthesis for the following languages

- VHDL
- Verilog
- System Verilog

6.5.3 MATLAB software:

MATLAB (Matrix Laboratory) is a programming platform developed by MathWorks, which uses its proprietary MATLAB programming language. The MATLAB programming language is a matrix-based language which allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python. It is used in a wide range of application domains from Embedded Systems to AI, mainly to analyze data, develop algorithms, and create models and applications.

6.6 XILINX VIVADO ISE DESIGN SUITE (16.1version)

Xilinx is a powerful software tool that is used to design, synthesize, simulate, test and verify digital circuit designs. The designer can describe the digital design by either using the schematic entry tool or a hardware description language. In this software we will create

VHDL design input files – the hardware description of the logic circuit, compile VHDL source files, create a test bench and simulate the design to make sure of the correct operation of the design (functional simulation). The purpose of this is to give new users an exposure to the basic and necessary steps to implement and examine your own designs using ISE environment. In this, we will design one simple module (OR gate); however, in the future, you will be designing such modules and completing the overall circuit design from these existing files. A VHDL input file in the Xilinx environment consists of: Entity Declarations: module name and interface specifications (I/O) – list of input and output ports; their mode, which is direction of data flow; and data type. Architecture: defines a component's logic operation.

There are different styles for the architecture body: (i)Behavioral – set of sequential assignment statements (ii) Data Flow – set of concurrent assignments o Structural – set of interconnected components A combination of these could be used, but in this tutorial, we will use Dataflow. In its simplest form, the architectural body will take the following format, regardless of the style: architecture architecture_name of entity_name is begun … -- statement end architecture_name;

ISE (Integrated Software Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different simuli, and configure the target device with the programmer.

Xilinx is an American technology company, primarily a supplier of programmable logic devices. It is known for inventing FPGA.The Xilinx ISE is primarily used for circuit synthesis and design, while the Modelsim logic simulator is used for system-level testing.

6.7 ISE Project Navigator:

In this section, we introduce the reader to the main components of an "ISE Project Navigator" window, which allows us to manage our design files and move our design process from creation to synthesis and to simulation phase.

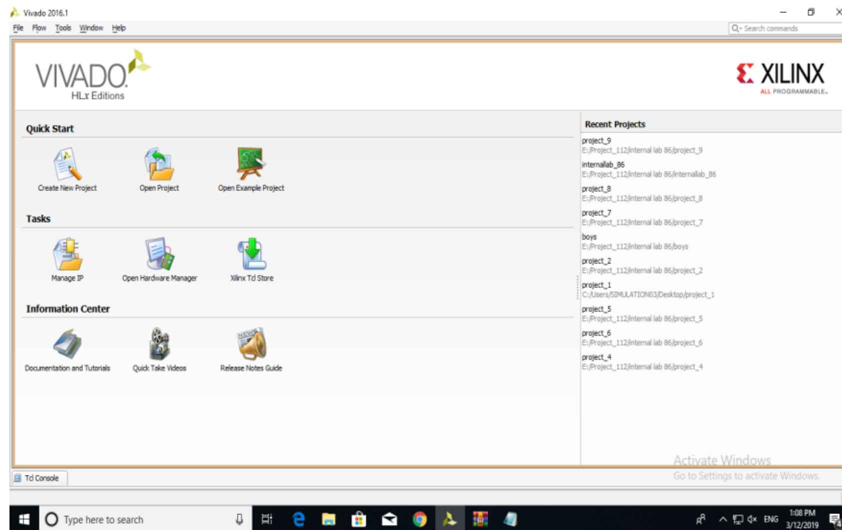


Fig 6.1 Xilinx Vivado Project Navigator window

By opening the Xilinx vivado ISE suite, we will come to see the 3 main points. They are

1) Quick start
2) Tasks
3) Information Center

In the Quick start block, we have created a new project, open project and open example project. In the Tasks, we have Manage IP, open hardware manager, Xilinx Td store In the Information center, we have documentation and tutorials, quick take videos and release notes guide.

This section describes the four basic steps to working with a project.

Step 1—— Creating a New Project

This creates .xpr file and a working library.

Step 2—— Adding Items to the project

Projects can reference or include source files, folders for organizations, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.

Step 3—— Compiling the Files

This checks syntax and semantics and creates the pseudo machine code that Vivado uses for simulation.

Step 4—— Simulating a Design

This specifies the design unit you want to simulate and opens a structure tab in the workspace panel.

you specify will be used to create a working library subdirectory within the Project

In order to start ISE double, click the desktop icon:

6.7.1 Creating a New Project

After launching Vivado, from the startup page click the "Create New Project" icon. Alternatively, you can select **File -> New Project**

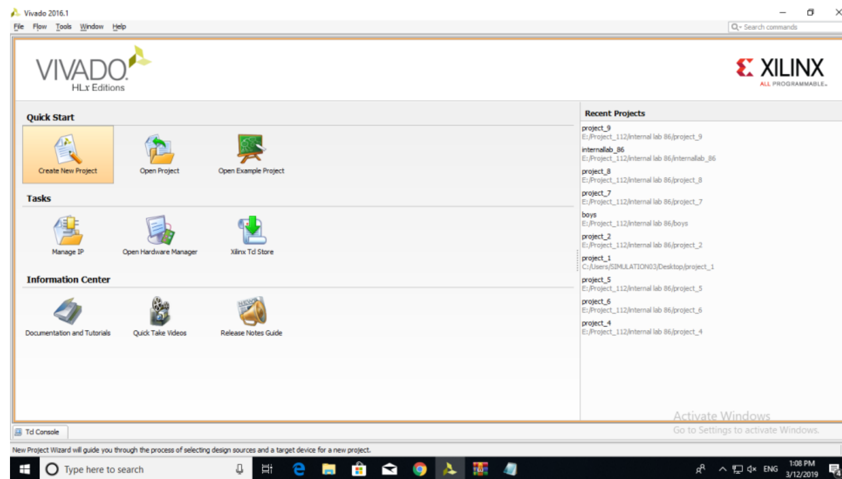


Fig 6.2. Creating new project window

The New Project wizard will launch, click the "Next >" button to proceed

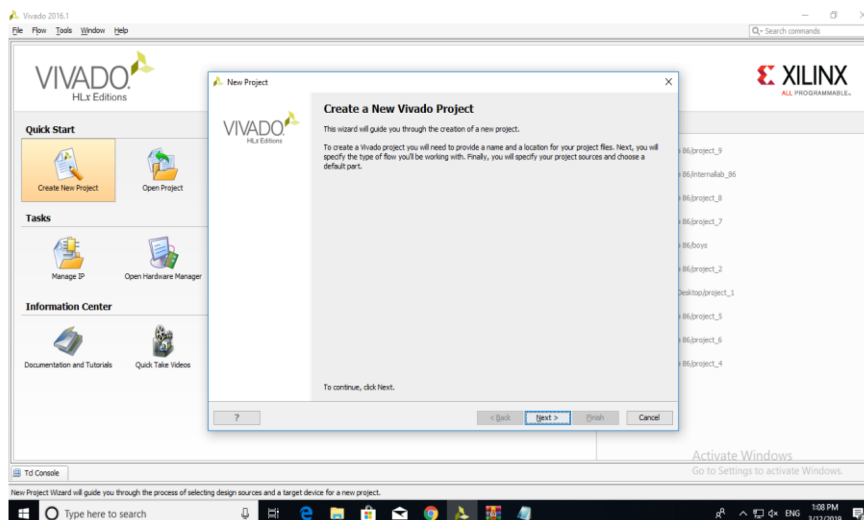


Fig 6.3. Guiding wizard for the project

Enter a project name and select a project location. **Make certain there are NO SPACES in either!** It's not a bad idea to only use letters, numbers, and underscores as well. If necessary, simply create a new directory for your Xilinx Vivado projects in your root drive (e.g., C:\Vivado). You will likely always want to select the "Create project sub-directory" check-box as well. This keeps things neatly organized with a directory for each project and helps avoid problems. Click the "Next >" button to proceed.

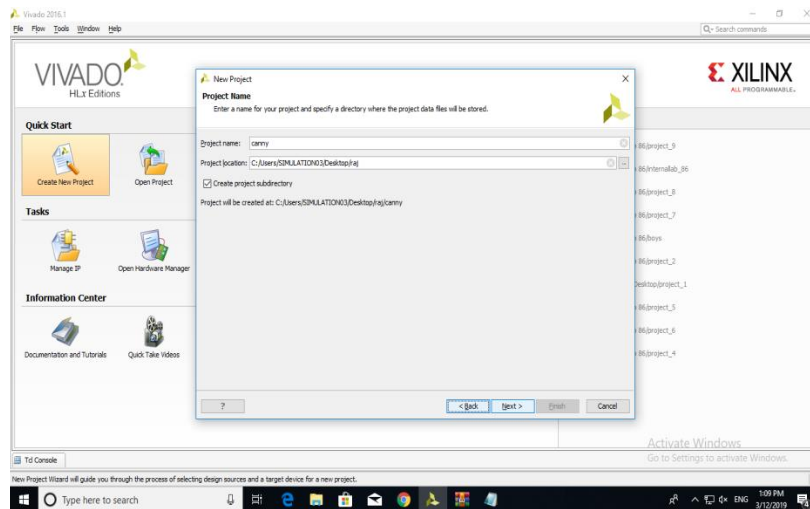


Fig 6.4. Creating a project name

Select the "RTL Project" radial and select the "Do not specify sources at this time" check-box. If you don't select the check-box the wizard will take you through some additional steps to optionally add preexisting items such as VHDL or Verilog source files, Vivado IP blocks, and. XDC constraint files for device pin and timing configuration. For this first project you will add the necessary items later. Click the "Next >" button to proceed.

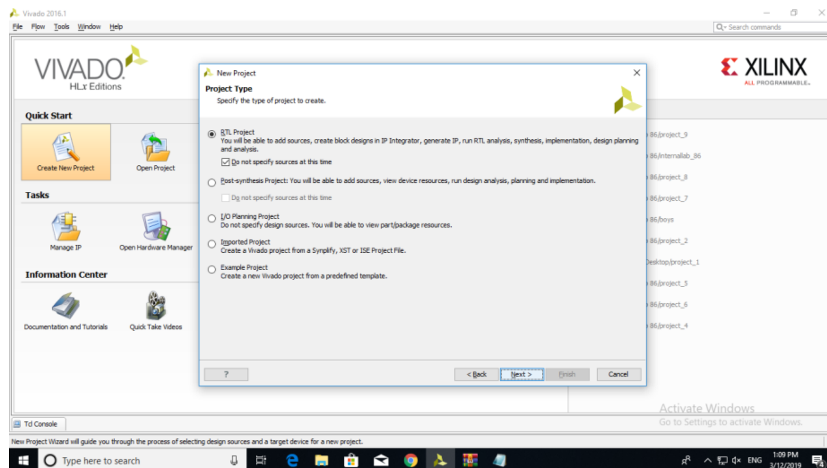


Fig 6.5. Specifying the RTL project

You need to filter down to and select the specific part number for your project. You can physically read the markings on your chip or refer to your board's documentation to find its part number. In the case of the Basys 3 it's the Artix-7 chip that's on the board, and the filters shown will help you get to the correct device that's highlighted. Once you select the correct device click the "Next >" button to proceed.

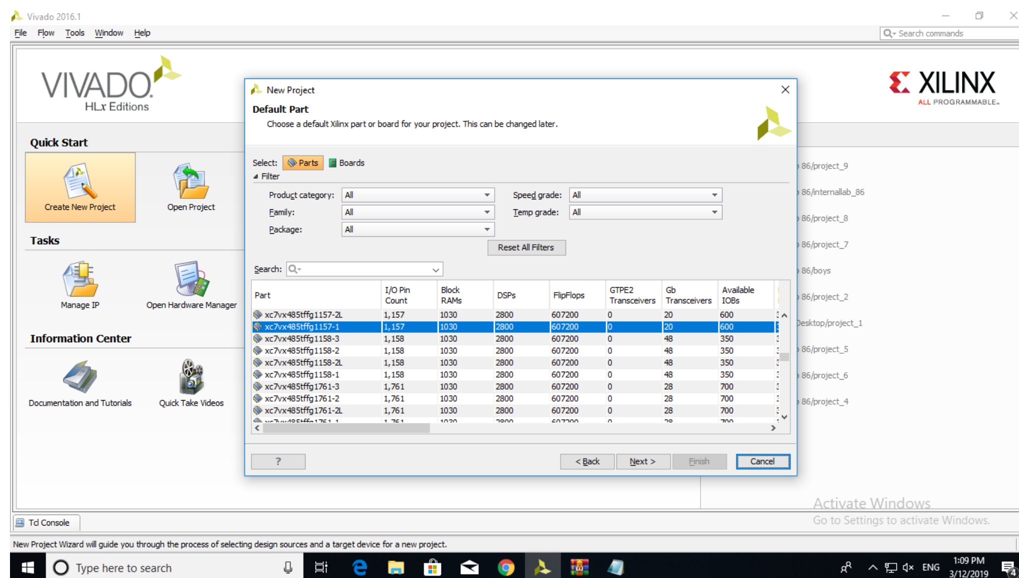


Fig 6.6. Choosing a board for project

Click the "Finish" button and Vivado will proceed to create your project as specified.

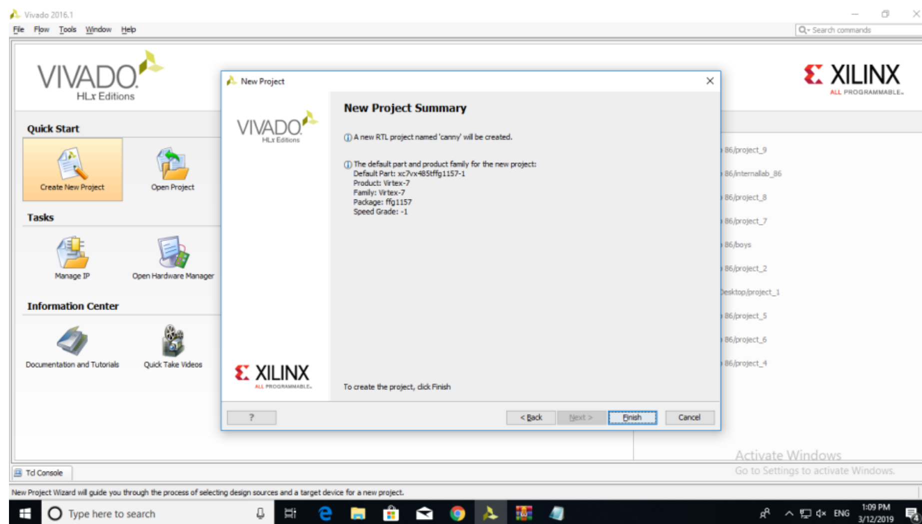


Fig 6.7. Project summary

6.8 STEPS FOR DESIGN ENTRY:

6.8.1 Working through the Basic Project Flow:

The Vivado project window contains a lot of information, and the information displayed can change depending on what part of the design you currently have open as you work through the steps of your project. Keep this in mind as you work through this guide, because if you don't see a specific sub-window or sub-window tab it's possible you aren't in the correct part of the design

The "Flow Navigator" on the left side of the screen has all the major project phases organized from top to bottom in their natural chronological order. You begin in the "Project Manager" portion of the flow and the header at the top of the screen next to the Flow Navigator reflects this. This header and the corresponding highlighted section in the Flow Navigator will tell you which phase of the design you have open.

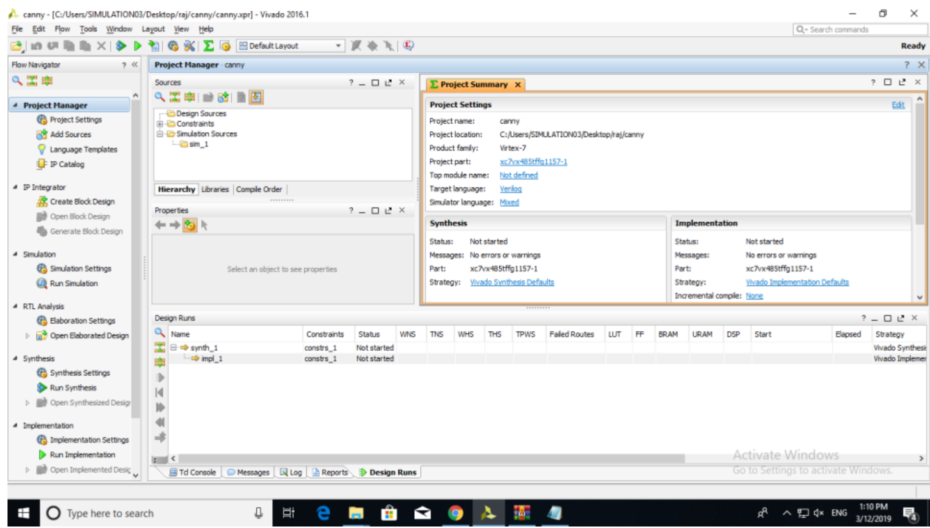


Fig 6.8. Main window for the project

6.8.2 Project Manager

6.8.2.1 Project Settings

Begin by clicking on “Project Settings” under the Project Manager phase of the Flow Navigator

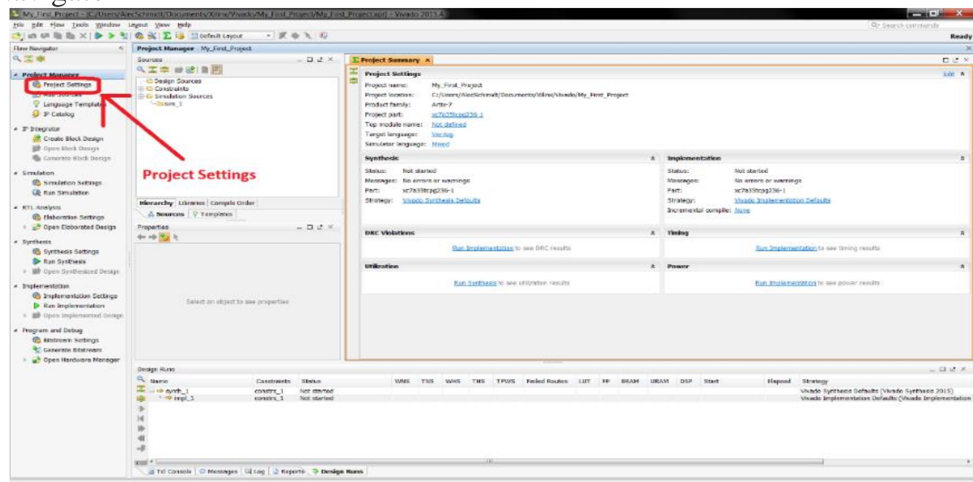


Fig 6.9. Project settings window

There are a lot of settings available here for all phases of the project flow, but for now just select “System Verilog” from the drop-down for the “Target language” in the “General” project settings and click the “OK” button.

6.8.2.2 Add Sources

Now click on “Add Sources” under the Project Manager phase of the Flow Navigator

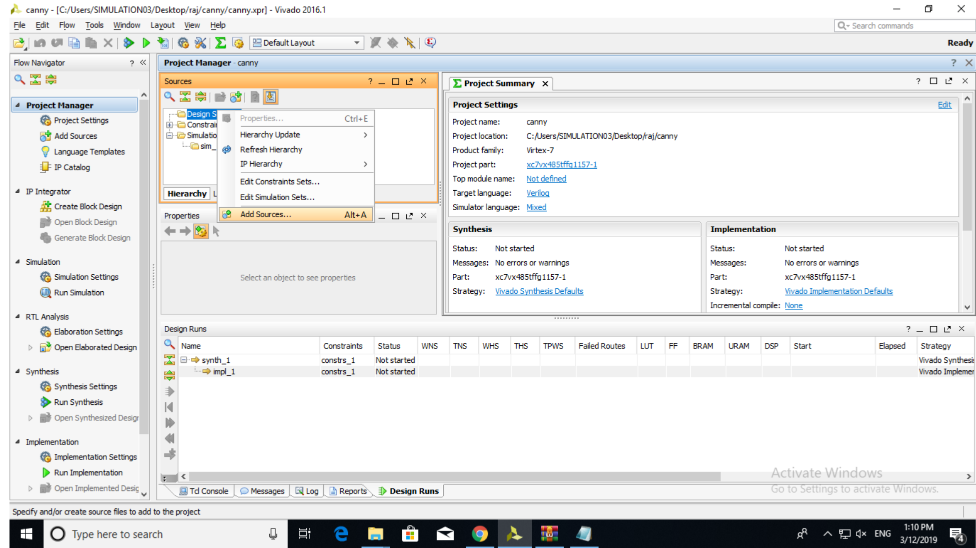


Fig 6.10. Adding the source files

Select the “Add or create design sources” radial and then click the “Next >” button.

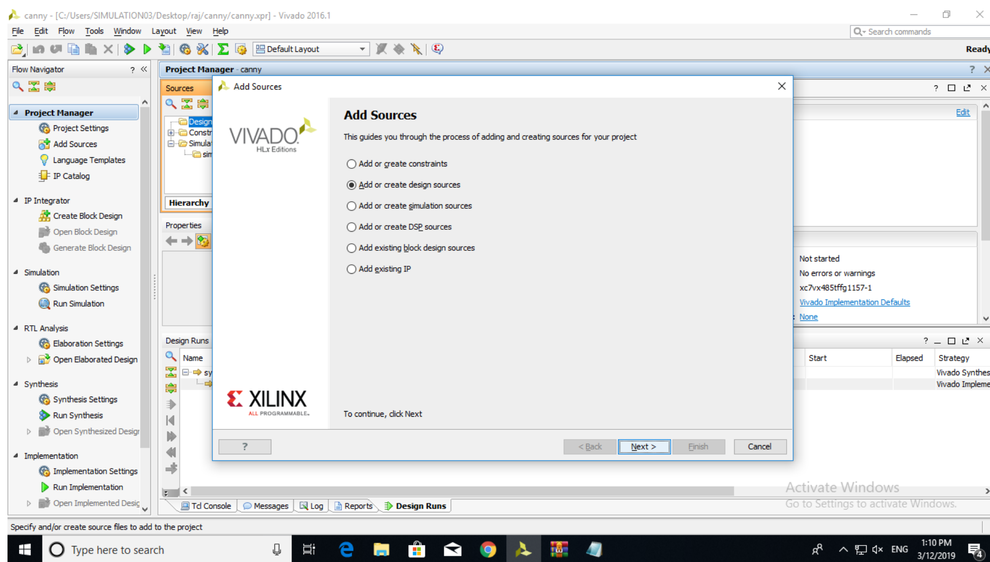


Fig 6.11. Wizard that shows to the design source

Click the “Create File” button or click the green “+” symbol in the upper left corner and select the “Create File…” option.

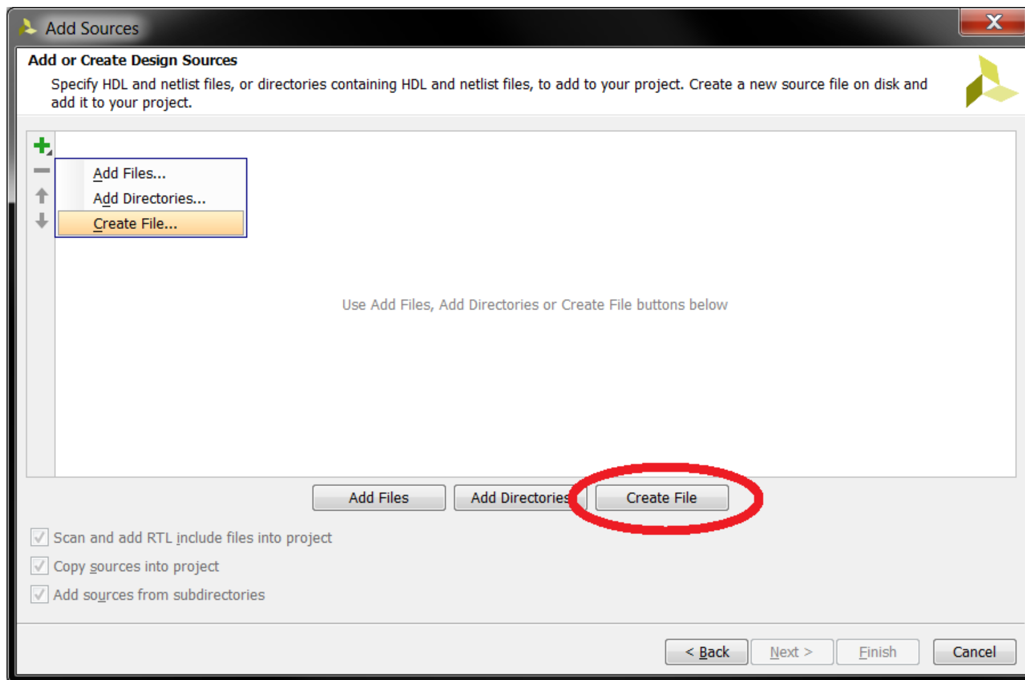


Fig 6.12. Creating a new file name for new design source

Make sure the options shown are selected in the "Create Source File" popup, and for the sake of following along enter "convolution (Gaussian filter)" for the "File name". Click the "OK" button when finished. You can normally enter anything you like for the "File name" as long as it's valid, but **always make certain there are NO SPACES!**

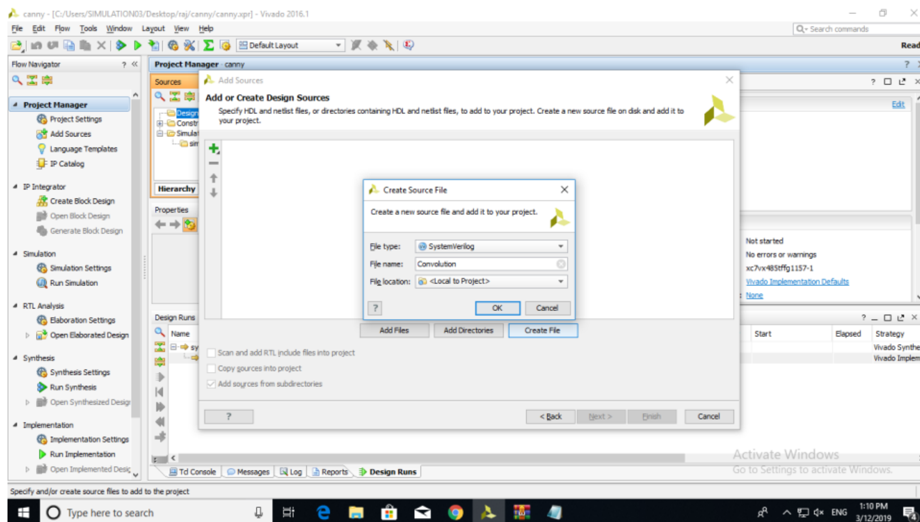


Fig 6.13. Selecting the type of file and location

Click the "Finish" button and Vivado will then bring up the "Define Module" window.

6.8.2.3 Define Module

You can use the “Define Module” window to automatically write some of the VHDL code for you. Additional “I/O Port Definitions” can be added by either clicking the green “+” symbol in the upper left or by simply clicking on the next empty line. The “Entity name” and “Architecture name” will be the corresponding Verilog HDL identifiers used in the code, as will whatever is typed in for each “Port Name”. Any valid Verilog HDL identifier can be used for any of these, but for the sake of following along enter the information as shown. Make sure the proper “Direction” is set for each. Click the “OK” button when finished.

Note that if you would rather write your own code from scratch, you can simply click the “Cancel” button and Vivado will create a completely blank System Verilog VHDL source file inside your project. If you click the “OK” button without defining any “I/O Port Definitions” Vivado will still write the basic Verilog HDL code structure but the port definition will be empty and commented out for you to uncomment and fill later.

Also note that the port names here match the silkscreen reference designators of the switches and LEDs on the Basys 3 board that will be utilized for the example. This is for the convenience of those following along with the Basys 3, but should not be inferred as a requirement by beginners; each name is simply an arbitrary identifier

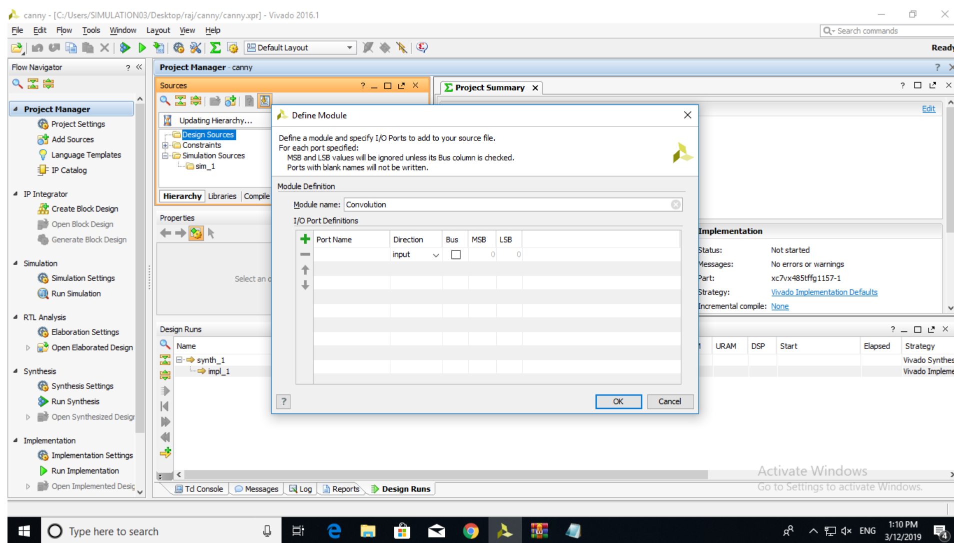


Fig 6.14. Module defining with ports

The System Verilog HDL source file generated will be added to your project in the "Design Sources" folder as shown. Double click it and it will open up in a new tab for you to view/edit. All the code here was generated by the previous "Define Module" window, and for this example you only need to manually enter the three highlighted lines between the "begin" and "end" keywords

If we want to create a simulation source, we have to select a new simulation source by right clicking the add source block in the panel

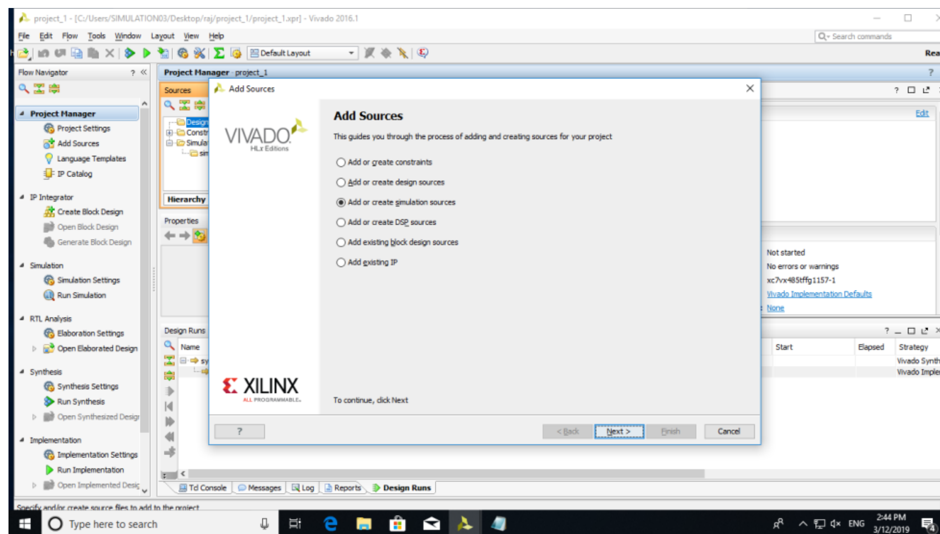


Fig 6.15. Creating the simulation sources

CHAPTER-7

MATLAB

7.1 MATLAB Introduction

MATLAB is a high-performance language for technical computing. It integrates computation visualization and programming in an easy-to-use environment. MATLAB stands for matrix laboratory. It was written originally to provide easy access to matrix software developed by LINPACK (linear system package) and EISPACK (Eigen system package) projects. MATLAB is therefore built on a foundation of sophisticated matrix software in which the basic element is matrix that does not require pre dimensioning.

Typical uses of MATLAB

1. Math and computation
2. Algorithm development
3. Data acquisition
4. Data analysis, exploration and visualization
5. Scientific and engineering graphics

The main features of MATLAB

1. Advanced algorithm for high performance numerical computation, especially in the Field matrix algebra
2. A large collection of predefined mathematical functions and the ability to define one's own functions.
3. Two-and three-dimensional graphics for plotting and displaying data
4. A complete online help system
5. Powerful matrix or vector oriented high level programming language for individual applications.
6. Toolboxes available for solving advanced problems in several application areas.

7.2 The MATLAB System

The MATLAB System consists of five main parts

7.2.1 Development Environment:

This is the set of tools and facilities that help you use MATLAB functions and files. Many of these tools are graphical user interfaces. It includes the MATLAB desktop and Command Window, command history an editor and debugger, and browsers for viewing help the workspace, files, and the search path.

7.2.2 The MATLAB Mathematical Function Library:

This is a vast collection of computational algorithms ranging from elementary functions, like sum sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix Eigen values, Bessel functions, and fast Fourier transforms.

7.2.3 The MATLAB Language:

This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both programming in the small to rapidly create quickly programs, and "programming in the large" to create large and complex application programs.

7.2.4 Graphics:

MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. It includes high-level functions for two-dimensional and three-dimensional data visualization, video processing, animation, and presentation graphics. It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build complete graphical user interfaces on your MATLAB applications

7.2.5 The MATLAB Application Program Interface (API):

This is a library that allows you to write C and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files

7.2.6 Starting MATLAB:

On Windows platforms, start MATLAB by double-clicking the MATLAB shortcut icon on your Windows desktop. On UNIX platforms, start MATLAB by typing mat lab at the operating system prompt. You can customize MATLAB start-up. For example, you can

change the directory in which MATLAB starts or automatically execute MATLAB statements in a script file named start-ups.

7.2.7 MATLAB Desktop:

When you start MATLAB, the MATLAB desktop appears, containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB. The following illustration shows the default desktop. You can customize the arrangement of tools and documents to suit your needs.

7.3 MATLAB Working Environment

7.3.1 MATLAB Desktop:

MATLAB Desktop is the main MATLAB application window. The desktop contains five sub windows the command window, the workspace browser the current directory window, the command history window, and one or more figure windows, which are shown only when the user displays a graphic.

The command window is where the user types MATLAB commands and expressions at the prompt (>>) and where the output of those commands is displayed. MATLAB defines as the workspace as the set of variables that the user creates in a work session. The workspace browser shows these variables and some information about them. Double clicking on a variable in the workspace browser launches the Array Editor, which can be used to obtain information and income instances edit certain properties of the variable.

The current Directory tab above the workspace tab shows the contents of the current directory, whose path is shown in the current directory window. For example, in the windows operating system the path might be as follows: C-MATLAB Work, indicating that directory "work" is a subdirectory of the main directory MATLAB WHICH ISINSTALLED IN DRIVE C. clicking on the arrow in the current directory window shows a list of recently used paths. Clicking on the button to the right of the window allows the user to change the current directory.

MATLAB uses a search path to find M-files and other MATLAB related files, which are organize in directories in the computer file system. Any file run in MATLAB must reside in the current directory or in a directory that is on search path. By default, the files supplied with MATLAB and math works toolboxes are included in the search path. The easiest way to see which directories is on the search path. The easiest way to see which directories are soon

the search paths, or to add or modify a search path, is to select set path from the File menu the desktop, and then use the set path dialog box. It is good practice to add any commonly used directories to the search path to avoid repeatedly having the change the current directory.

The Command History Window contains a record of the command window, including both current and previous MATLAB sessions. Previously entered MATLAB commands can be selected and re-executed from the command History window by right clicking on a command or sequence of commands. This action launches a menu from which to select various options in addition to executing the commands. This is a use to select 34 various options in addition to executing the commands. This is useful feature when experimenting with various commands in a work session.

7.3.2 Using the MATLAB Editor to create M-Files:

The MATLAB editor is both a text editor specialized for creating M-files and graphical MATLAB debugger. The editor can appear in a window by itself, or it can be a sub window in the desktop, M-files are denoted by the extension m The MATLAB editor window has numerous pull-down menus for tasks such as saving viewing, and debugging files. Because it performs some simple checks and also uses color to differentiate between various elements of code, this text editor is recommended as the tool of choice for writing and editing M functions. To open the editor, type edit at the prompt opens the M-file filenames in an editor window ready for editing. As noted earlier the file must be in the current directory, or in a directory in the search path

7.3.3 Getting Help:

The principle way to get help online is lo use the MATLAB help browser, opened as a separate window either by clicking on the question mark symbol (?) on the desktop toolbar, or by typing help browser at the prompt in the command window. The help Browser is a web browser integrated into the MATLAB desktop that displays a Hypertext Markup Language (HTML) document. The Help Browser consists of two panes, the help navigator pane, used to find information, and the display pane, used to view the information. Self- explanatory tabs other navigator pane is used to perform a scratch. For example, help on a specific function is obtained by selecting the search tab, selecting Function Name as the Search Type, and then typing in the function name in the Search for field. It is good practice to open the Help Browser

at the beginning of a MATLAB session to have help readily available during code development or other MATLAB task.

Another way to obtain for a specific function is by typing doc followed by the function name at the command prompt. For example, typing doc format displays documentation for the function called format in the display pane of the Help Browser. This command opens the browser if it is not already open.

M-functions have two types of information that can be displayed by the user The first is called the HI line, which contains the function name and alone line description the second is a block of explanation called the Help text block. Typing help at the prompt followed by a function name displays both the HI line and the Help text for that function in there command window. Typically look for followed by a keyword displays all the HI lines that contain that keyword. This function is useful when looking for a particular topic without knowing the names of applicable functions. For example, typing look for edge at the prompt displays the HI lines containing that keyword. Because the HI line contains the function name, it then becomes possible to look at specific functions using the other help methods. Typing look for edge-all at the prompt displays the HI line of all functions that contain the word edge in cither the HI line or the Help text block. Words that contain the characters edge also are detected. For example, the HI line of a function containing the word poly edge in the H1 linear Help text would also be displayed.

7.4 Saving and Retrieving a Work Session

There are several ways to save and load an entire work session or selected workspace variables in MATLAB. The simplest is as follows. To save the entire workspace, simply right-click on any blank space in the workspace Browser window and select Save Workspace as from the menu that appears. This opens a directory window that allows naming the file and selecting any folder in the system in which to save it. Then simply click Save To save a selected variable from the workspace, select the variable with a left click and then right-click on the highlighted area. Then select Save Selection As from the menu that appears. This again opens a window from which a folder can be selected to save the variable.

To select multiple variables, use shift click or control click in the familiar manner, and then use the procedure just described for a single variable. All files are saved in the doubleprecision, binary format with the extension mat. These saved files commonly are

referred to as MAT-files. For example, a session named, says mywork_2012-02-10, and would appear as the MAT-file mywork_2012_02_10.mat when saved. Similarly, a saved video called final video will appear when saved as final_ video. Mat.

To load saved workspaces and/or variables, left-click on the folder icon on the toolbar of the workspace Browser window. This causes a window to open from which a folder containing MAT-file or selecting open causes the contents of the file to be restored in the workspace Browser window. It is possible to achieve the same results described in the preceding paragraphs by typing save and load at the prompt with the appropriate file names and path information. This approach is not as convenient, but it is used when formats other than those available in the menu method are required.

7.4.1 Graph Components:

MATLAB displays graphs in a special window known as a figure. To create a graph, you need to define a coordinate system. Therefore, every graph is placed within Axes, which are contained by the figure. The actual visual representation of the data is achieved with graphics objects like lines and surfaces. These objects are drawn within the coordinate system defined by the axes, which MATLAB automatically creates specifically to accommodate the range of the data. The actual data is stored as properties of the graphics objects.

7.4.2 Plotting Tools

Plotting tools are attached to figures and create an environment for creating Graphs. These tools enable you to do the following:

- Select from a wide variety of graph types
- Change the type of graph that represents a variable
- See and set the properties of graphics objects
- Annotate graphs with text, arrows, etc.
- Drag and drop data into graphs

Display the plotting tools from the View menu or by clicking the plotting tools icon in the figure toolbar, as shown in the following picture.

7.4.3 Editor/Debugger

Use the Editor/Debugger to create and debug M-files, which are programs you write to run MATLAB functions. The Editor/Debugger provides a graphical user interface for text editing, as well as for M-file debugging.

CHAPTER 8

SIMULATED OUTPUTS

8.1 simulated outputs from MATLAB:

Basic CORDIC algorithm and modified CORDIC algorithm are simulated and results are obtained using MATLAB software. The results are given below followed by their codes respectively.

8.1.1 MATLAB code for basic cordic algorithm:

```
clc
clear all
close all
atheta = [0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 99];
k=1
sin = [];
cos = [];
tan = [];
for i = 0:9
    r=atan(2^(-1*i));
    tan(i+1) = (180*r)/pi;
end
for i= 0:9
    k=k* sqrt(1+(2^(-2*i)));
end
k=1/k;
for p=1:21
   rtheta=atheta(p);
x = [k];
y = [0];
wtheta = 0;
theta = [0];
```

```
for i=1:10
   if wtheta<rtheta
      sigma=-1;
   else
      sigma=1;
   end
   x(i+1) = x(i)+(sigma*(y(i)*(2^(-(i-1)))));
   y(i+1) = y(i)-(sigma*(x(i)*(2^(-(i-1)))));
   wtheta = wtheta-(sigma*tan(i));
   theta(i+1) = wtheta;
end
sin(p) = y(11)
cos(p) = x (11)
end
stem (atheta, sin)
grid on
figure
stem (atheta, cos)
grid on
```

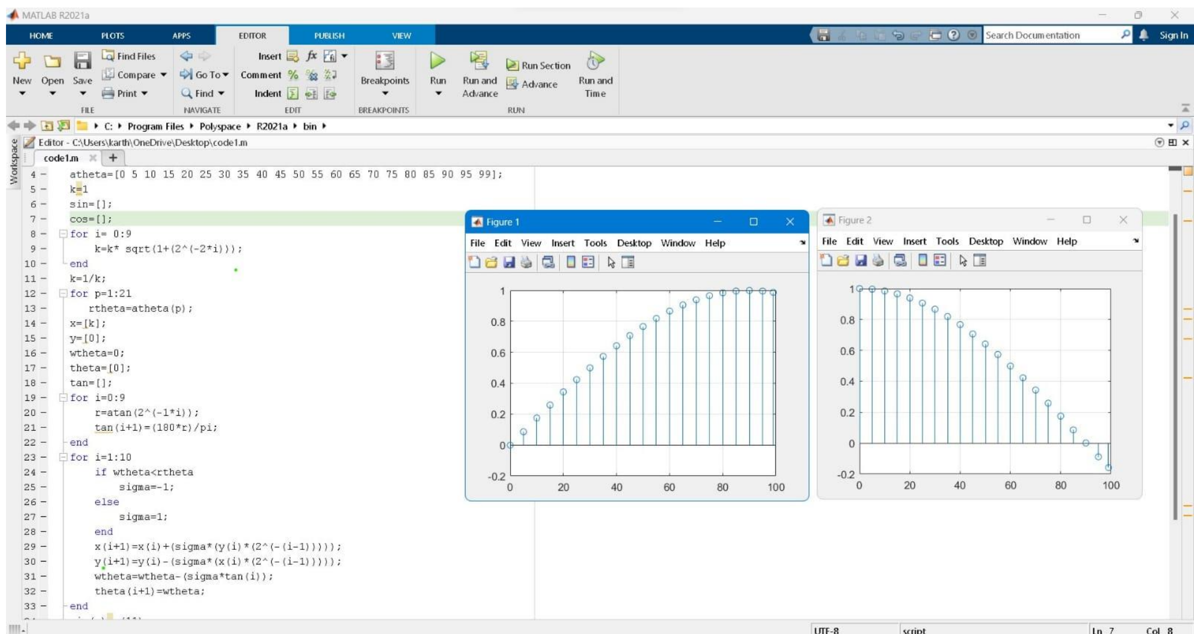


Fig 8.1 output showing results of sine and cosine waves using basic CORDIC algorithm

8.1.2 MATLAB code for modified cordic algorithm:

```
clc
clear all
close all
atheta = [-99 -95 -90 -85 -80 -75 -70 -65 -60 -55 -50 -45 -40 -35 -30 -25 -20
        -15 -10 -5 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 99];
k=1;
l=length(atheta);
sin = [];
cos = [];
for i= 0:9
k=k* sqrt(1+(2^(-2*i)));
end
k=1/k;
tan = [];
for i=0:9
r=atan(2^(-1*i));
tan(i+1) = (180*r)/pi;
end
for p = 1: l
rtheta=atheta(p);
x = [k];
y = [0];
wtheta=0;
```

```
theta = [0];

b = [];

for i=1:10

if wtheta>rtheta

b(i)=0;

wtheta=wtheta-tan(i);

else

b(i)=1;

wtheta = wtheta+tan(i);

end

end

for i=1:10

r(i)=2*b(i)-1;

end

for i=1:10

x(i+1) = x(i)-(r(i)*(y(i)*(2^(-(i-1)))));

y(i+1) = y(i)+(r(i)*(x(i)*(2^(-(i-1)))));

end

sin(p) = y (11);

cos(p) = x (11);

end

stem (atheta, sin)

title ("sine wave form")

xlabel ("angle in degrees")

ylabel ("sine of angle")
```

grid on

figure

stem (atheta, cos)

title ("cosine wave form")

xlabel ("angle in degrees")

ylabel ("cosine of angle")

grid on

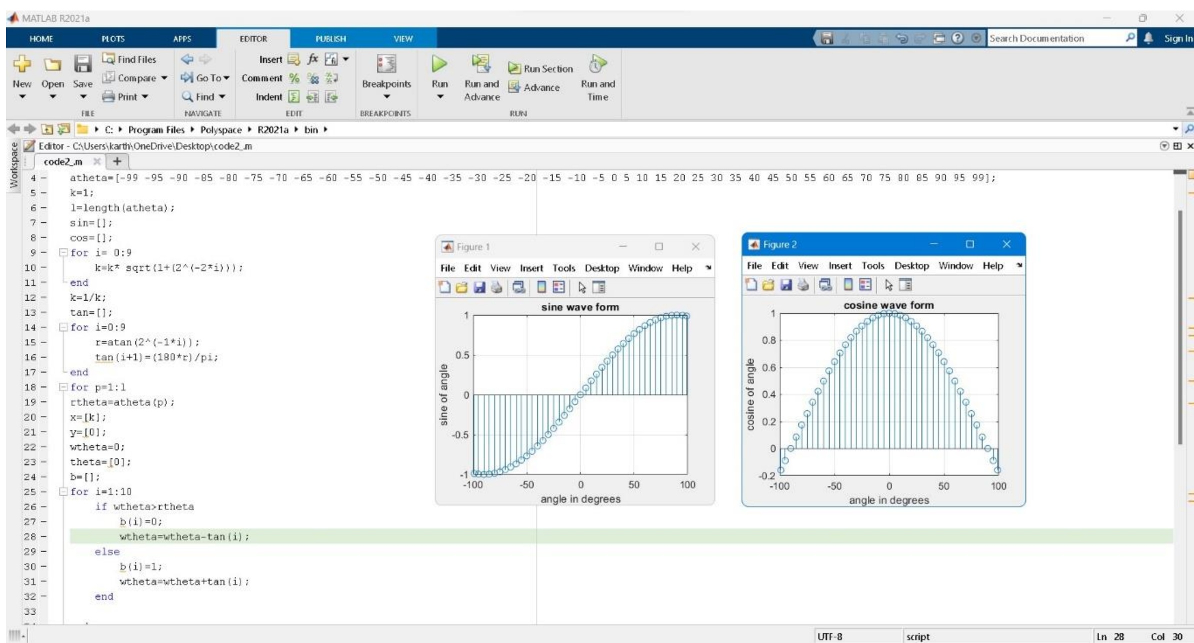


Fig 8.2 output showing results of sine and cosine waves using modified CORDIC algorithm

The below table shows the values of cosine and sine with respective to the angles (θ). The values of sine and cosine values using basic CORDIC and modified CORDIC are calculated. The differences are also calculated and are shown in the table below

θ in degrees	Value of cos θ	Basic CORDIC (cosθ)	Modified CORDIC (cosθ)	difference
-90	0.0000	0.0012	0.0012	0.0000
-85	0.0871	0.0859	0.0859	0.0000
-80	0.1736	0.1749	0.1749	0.0000
-75	0.2588	0.2586	0.2586	0.0000
-70	0.3420	0.3435	0.3435	0.0000
-65	0.4226	0.4228	0.4228	0.0000
-60	0.5000	0.4989	0.4989	0.0000
-55	0.5735	0.5738	0.5738	0.0000
-50	0.6427	0.6418	0.6418	0.0000
-45	0.7071	0.7080	0.7062	0.0018
-40	0.7660	0.7669	0.7669	0.0000
-35	0.8191	0.8190	0.8190	0.0000
-30	0.8660	0.8666	0.8666	0.0000
-25	0.9063	0.9062	0.9062	0.0000
-20	0.9396	0.9391	0.9391	0.0000
-15	0.9659	0.9660	0.9660	0.0000
-10	0.9848	0.9846	0.9846	0.0000
-05	0.9961	0.9963	0.9963	0.0000
00	1.0000	1.0000	1.0000	0.0000
05	0.9961	0.9963	0.9963	0.0000
10	0.9848	0.9846	0.9846	0.0000
15	0.9659	0.9660	0.9660	0.0000
20	0.9396	0.9391	0.9391	0.0000
25	0.9063	0.9062	0.9062	0.0000
30	0.8660	0.8666	0.8666	0.0000
35	0.8191	0.8190	0.8190	0.0000
40	0.7660	0.7669	0.7669	0.0000
45	0.7071	0.7062	0.7080	0.0018
50	0.6427	0.6418	0.6418	0.0000
55	0.5735	0.5738	0.5738	0.0000
60	0.5000	0.4989	0.4989	0.0000
65	0.4226	0.4228	0.4228	0.0000
70	0.3420	0.3435	0.3435	0.0000
75	0.2588	0.2586	0.2586	0.0000
80	0.1736	0.1749	0.1749	0.0000
85	0.0871	0.0859	0.0859	0.0000
90	0.0000	0.0012	0.0012	0.0000

Table 8.1 comparison of CORDIC and modified CORDIC cosine values

θ in degrees	Value of sin θ	Basic CORDIC (sinθ)	Modified CORDIC (sinθ)	difference
-90	-1.0000	-1.0000	-1.0000	0.0000
-85	-0.9961	-0.9963	-0.9963	0.0000
-80	-0.9848	-0.9846	-0.9846	0.0000
-75	-0.9659	-0.9660	-0.9660	0.0000
-70	-0.9396	-0.9391	-0.9391	0.0000
-65	-0.9063	-0.9062	-0.9062	0.0000
-60	-0.8660	-0.8666	-0.8666	0.0000
-55	-0.8191	-0.8190	-0.8190	0.0000
-50	-0.7660	-0.7669	-0.7669	0.0000
-45	-0.7071	-0.7062	-0.7080	0.0018
-40	-0.6427	-0.6418	-0.6418	0.0000
-35	-0.5735	-0.5738	-0.5738	0.0000
-30	-0.4999	-0.4989	-0.4989	0.0000
-25	-0.4226	-0.4228	-0.4228	0.0000
-20	-0.3420	-0.3435	-0.3435	0.0000
-15	-0.2588	-0.2586	-0.2586	0.0000
-10	-0.1736	-0.1749	-0.1749	0.0000
-05	-0.0871	-0.0859	-0.0859	0.0000
00	0.0000	-0.0012	0.0012	0.0000
05	0.0871	0.0859	0.0859	0.0000
10	0.1736	0.1749	0.1749	0.0000
15	0.2588	0.2586	0.2586	0.0000
20	0.3420	0.3435	0.3435	0.0000
25	0.4226	0.4228	0.4228	0.0000
30	0.4999	0.4989	0.4989	0.0000
35	0.5735	0.5738	0.5738	0.0000
40	0.6427	0.6418	0.6418	0.0000
45	0.7071	0.7080	0.7062	0.0018
50	0.7660	0.7669	0.7669	0.0000
55	0.8191	0.8190	0.8190	0.0000
60	0.8660	0.8666	0.8666	0.0000
65	0.9063	0.9062	0.9062	0.0000
70	0.9396	0.9391	0.9391	0.0000
75	0.9659	0.9660	0.9660	0.0000
80	0.9848	0.9846	0.9846	0.0000
85	0.9961	0.9963	0.9963	0.0000
90	1.0000	1.0000	1.0000	0.0000

Table 8.2 comparison of CORDIC and modified CORDIC sine values

8.2 Simulated outputs from VERILOG:

8.2.1 VERILOG code for modified CORDIC algorithm:

```
`timescale 1ns/100 ps
  module m_cordic (clock, angle, Xin, Yin, Xout, Yout);
   parameter c_parameter = 16;   // bit width of input and output data
   localparam STG = c_parameter ; // similar bit width of vectors X and Y
   input                     clock;
   input  signed       [31:0] angle;
   input  signed  [c_parameter-1:0] Xin;
   input  signed  [c_parameter-1:0] Yin;
   output signed    [c_parameter :0] Xout;
   output signed    [c_parameter :0] Yout;
   //arctan_table
   // Note: The atan_table was chosen to be 31 bits wide giving resolution up to atan(2^-30)
   wire signed [31:0] atan_table [0:30];
   // upper 2 bits = 2'b00 which represents 0 - PI/2 range
   // upper 2 bits = 2'b01 which represents PI/2 to PI range
   // upper 2 bits = 2'b10 which represents PI to 3*PI/2 range (i.e. -PI/2 to -PI)
   // upper 2 bits = 2'b11 which represents 3*PI/2 to 2*PI range (i.e. 0 to -PI/2)
   // The upper 2 bits therefore tell us which quadrant we are in.
   assign atan_table[00] = 32'b00100000000000000000000000000000; // 45.000 degrees ->
atan(2^0)
   assign atan_table[01] = 32'b00010010111001000000010100011101; // 26.565 degrees ->
atan(2^-1)
   assign atan_table[02] = 32'b00001001111110110011100001011011; // 14.036 degrees ->
atan(2^-2)
   assign atan_table[03] = 32'b00000101000100010001000111010100; // atan(2^-3)
   assign atan_table[04] = 32'b00000010100010110000110101000011;
   assign atan_table[05] = 32'b00000001010001011101011111100001;
   assign atan_table[06] = 32'b00000000101000101111011000011110;
   assign atan_table[07] = 32'b00000000010100010111110001010101;
   assign atan_table[08] = 32'b00000000001010001011111001010011;
   assign atan_table[09] = 32'b00000000000101000101111100101110;
   assign atan_table[10] = 32'b00000000000010100010111110011000;
   assign atan_table[11] = 32'b00000000000001010001011111001100;
   assign atan_table[12] = 32'b00000000000000101000101111100110;
   assign atan_table[13] = 32'b00000000000000010100010111110011;
   assign atan_table[14] = 32'b00000000000000001010001011111001;
   assign atan_table[15] = 32'b00000000000000000101000101111101;
```

```
  assign atan_table[16] = 32'b00000000000000000010100010111110;
  //registers
  //stage outputs
  reg signed [c_parameter :0] X [0:STG-1];
  reg signed [c_parameter :0] Y [0:STG-1];
 // reg signed    [31:0] Z [0:STG-1]; // 32bit
 reg signed [31:0]Z;
  //------------------------------------------------------------------------------
  //                              stage 0
  //------------------------------------------------------------------------------
  wire                 [1:0] quadrant;
  assign   quadrant = angle[31:30];
  always @(posedge clock)
  begin //rotation angle is in the -pi/2 to pi/2 range.  If not then pre-rotate
    case (quadrant)
      2'b00,
      2'b11:   // no pre-rotation needed for these quadrants
      begin    // X[n], Y[n] is 1 bit larger than Xin, Yin, but Verilog handles the assignments
properly
        X[0] <= Xin;
        Y[0] <= Yin;
        Z <= angle;
      end
      2'b01:
      begin
        X[0] <= -Yin;
        Y[0] <= Xin;
        Z <= {2'b00,angle[29:0]}; // subtract pi/2 from angle for this quadrant
      end
      2'b10:
      begin
        X[0] <= Yin;
        Y[0] <= -Xin;
        Z<= {2'b11,angle[29:0]}; // add pi/2 to angle for this quadrant
      end
    endcase
  end
 reg b[0:STG-1];
integer j;
  always @(Z)
```

```
  begin
  for (j=0; j <=(STG-1); j=j+1)
  begin
  if (Z[31]==1'b1)
  begin
  b[j] = 1'b1;
  Z =Z+atan_table[j];
  end
  else
  begin
  b[j]=1'b0;
  Z=Z-atan_table[j];
  end
  end
  end
  //--------------------------------------------------------------------------
  //                    generate stages 1 to STG-1
  //--------------------------------------------------------------------------
  genvar i;
  generate
  for (i=0; i <= (STG-1); i=i+1)
  begin: XYZ
    wire signed  [c_parameter :0] X_shr, Y_shr;

    assign X_shr = X[i] >>> i; // signed shift right
    assign Y_shr = Y[i] >>> i;
      always @(posedge clock)
    begin
      // add/subtract shifted data
      X[i+1] <= b[i] ? X[i] +Y_shr        : X[i] - Y_shr;
      Y[i+1] <= b[i] ? Y[i] - X_shr        : Y[i] +X_shr;
    end
  end
  endgenerate
  //--------------------------------------------------------------------------
  //                         output
  //--------------------------------------------------------------------------
  assign Xout = X[STG-1];
  assign Yout = Y[STG-1];
endmodule
```

8.2.2 VERILOG code for test bench for modified CORDIC algorithm:

```
`timescale 1ns/100 ps
module test();
localparam  SZ = 16;
reg  [SZ-1:0] Xin, Yin;
reg  [31:0] angle;
wire [SZ:0] Xout, Yout;
reg         clk;
//localparam FALSE = 1'b0;
//localparam TRUE = 1'b1;
localparam VALUE = 32000/1.647; // reduce by a factor of 1.647 since thats the gain of the
system
reg signed [63:0] i;
m_cordic sin_cos (clk, angle, Xin, Yin, Xout, Yout);
initial
begin
  $write("Starting sim");
  clk = 1'b0;
  angle = 0;
  Xin = VALUE;                    // Xout = 32000*cos(angle)
  Yin = 1'd0;                   // Yout = 32000*sin(angle)
end
always #5 clk=~clk;
always @(posedge clk)
begin
#2;
for (i = 0; i < 360; i = i + 1)     // from 0 to 359 degrees in 1 degree increments
 begin
  @(posedge clk);
    angle = ((1 << 32) *i)/360;   // example: 45 deg = 45/360 * 2^32 =
32'b00100000000000000000000000000000 = 45.000 degrees -> atan (2^0)
     $monitor ("time =%d angle = %d, %h", $time, i, angle);
     end
end
endmodule
```

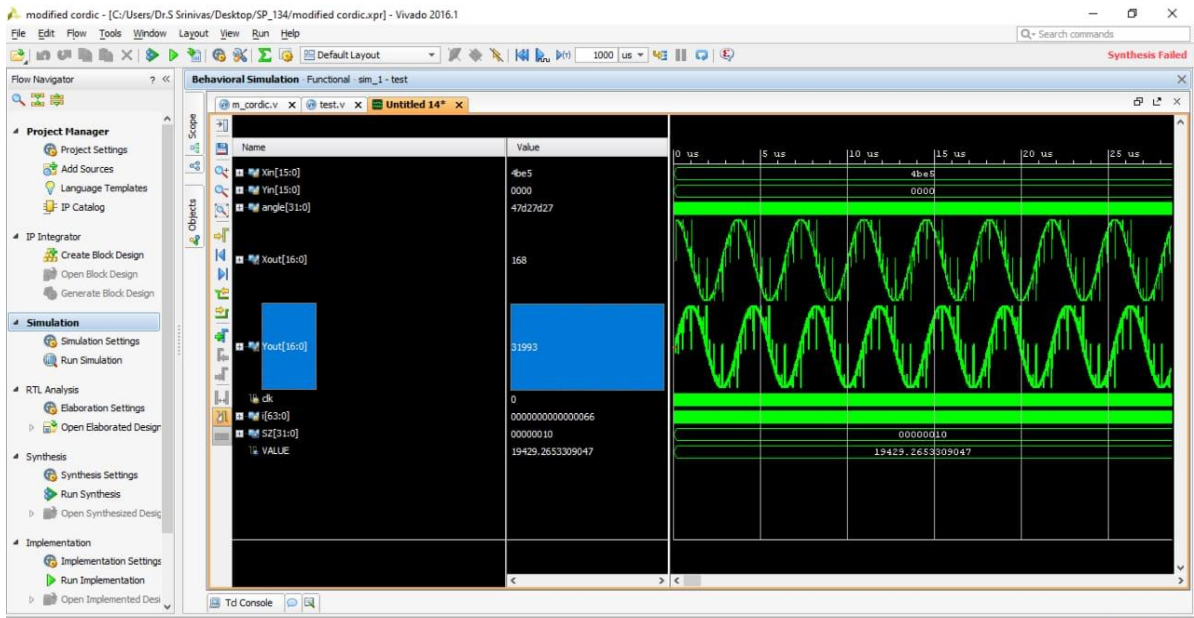


Fig 8.3 Verilog output showing results of sine and cosine waves using modified CORDIC

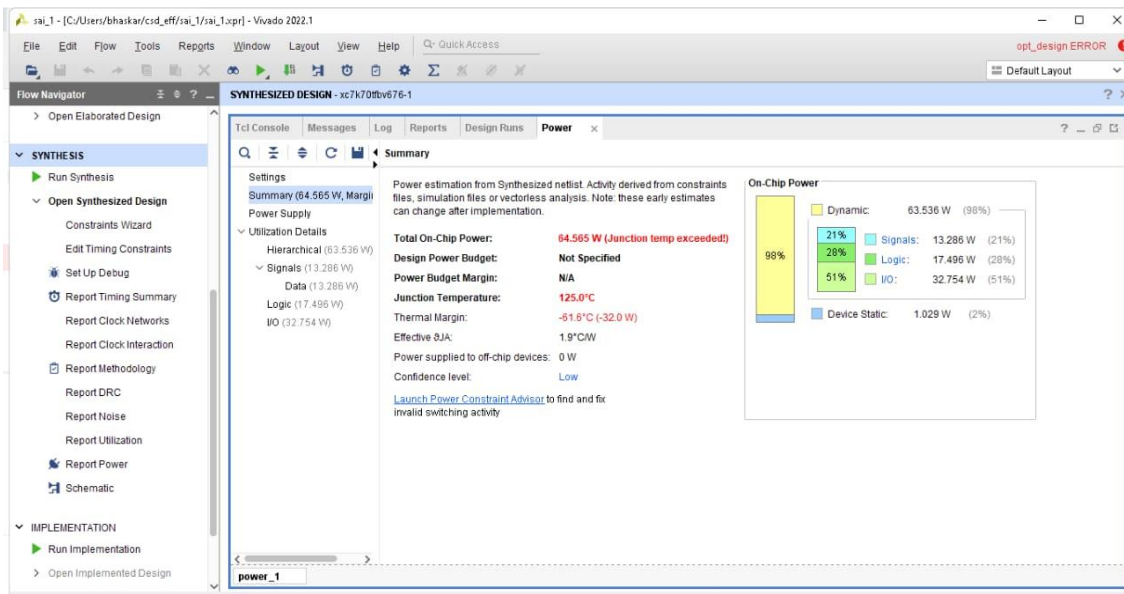


Fig 8.4 Power report of synthesized design

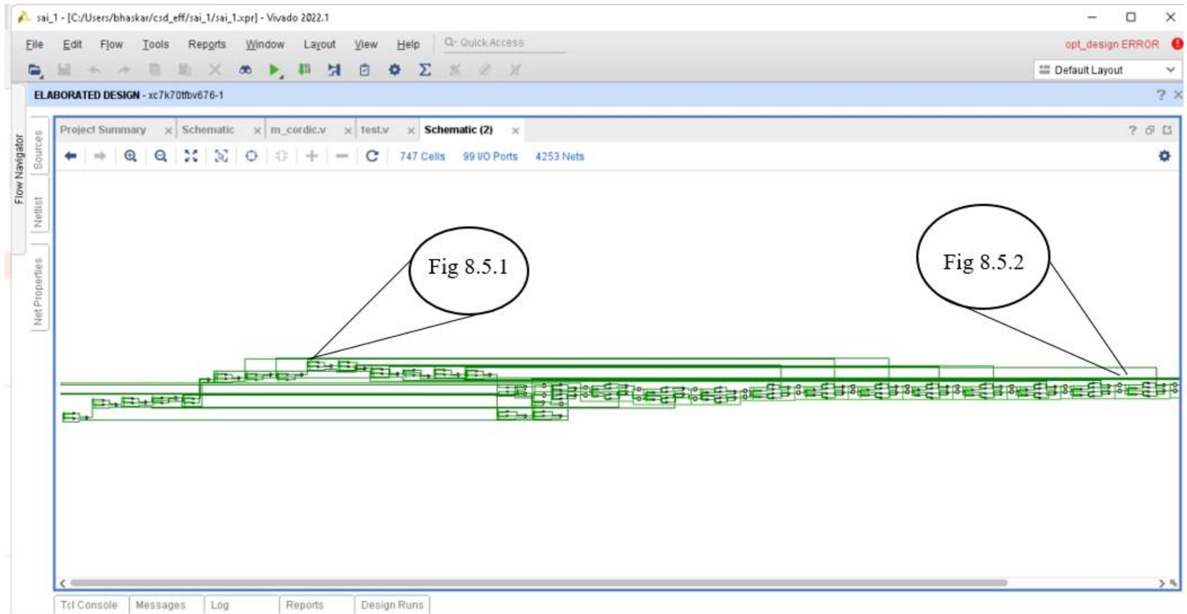


Fig 8.5 RTL schematic of design

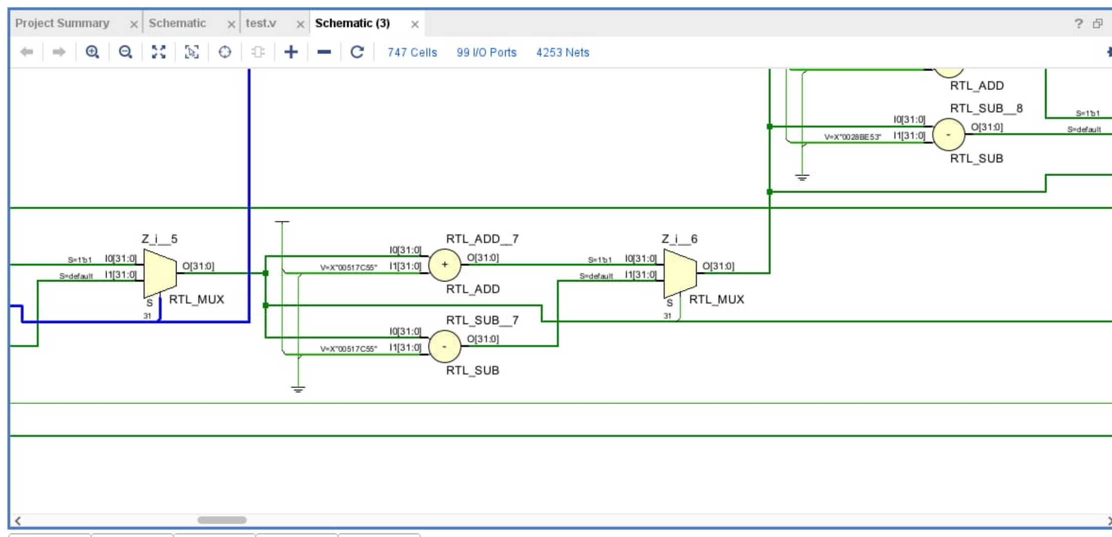


Fig 8.5.1 calculation of sign (+, -) for angles(z)

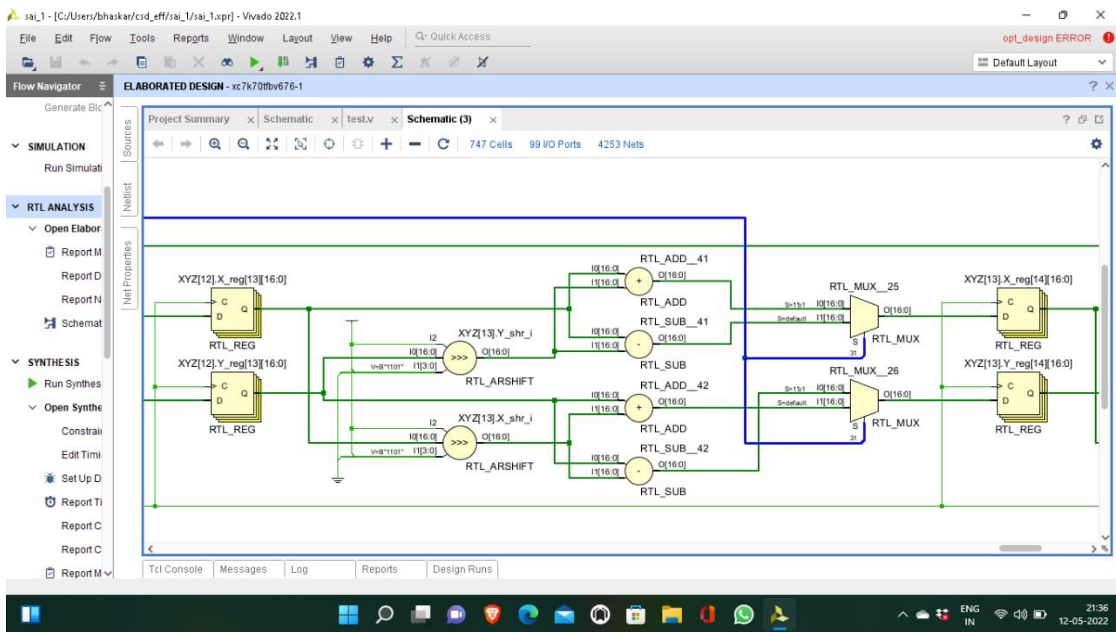


Fig 8.5.2 Modified cordic block

sai_1 - [C:/Users/bhaskar/csd_eff/sai_1/sai_1.xpr] - Vivado 2022.1

SYNTHESIZED DESIGN - xc7k70tfbv676-1

Utilization — Hierarchy

Name	Slice LUTs (41000)	Slice Registers (82000)	Bonded IOB (300)	BUFGCTRL (32)
m_cordic	1106	576	99	1

Fig 8.6 memory utilization of synthesized design

CONCLUSION

DDFS architecture generates sine and cosine waveforms which are used in communication systems. In basic design of DDFS architecture, it uses large amount of LUTs as it stores all values of angles. So, for implementation of DDFS architecture, a better and new algorithm, Modified Cordic algorithm is studied. It uses only shifting, add and subtract operations and also it uses very few LUTs. The modified cordic algorithm decreases hardware complexity when compared to conventional cordic algorithm. Therefore, the modified cordic algorithm is executed in MATLAB to test the working of the algorithm and the algorithm is also designed in Verilog, verified for individual angles and the RTL schematic of design, power, memory utilization reports of synthesized design are observed.

REFERENCES: -

1. A Digital Frequency Synthesizer- J. Tierney, C.M. Radre, and B. Gold IEEE Transactions on Audio and Electroacoustics, March 1971
2. Digital Design of Signal Processing Systems: A Practical Approach, First Edition. Shoab Ahmed Khan. 2011 John Wiley & Sons, Ltd. Published 2011 by John Wiley & Sons, Ltd
3. J. Valls, T. Sansaloni, A. P. Pascual, V. Torres and V. Almenar, “The use of CORDIC in software defined radios: a tutorial,” IEEE Communications Magazine, 2006, vol. 44, no. 9, pp. 46 50.
4. K. Murota, K. Kinoshita and K. Hirade, “GMSK modulation for digital mobile telephony,” IEEE Transactions on Communications, 1981, vol. 29, pp. 1044 1050
5. J. Volder, “The CORDIC computing technique,” IRE Transactions on Computing, 1959, pp. 330 334.
6. J. S. Walther, “A unified algorithm for elementary functions,” in Proceedings of AFIPS Spring Joint Computer Conference, 1971, pp. 379 385.
7. S. Wang, V. Piuri and E. E. Swartzlander, “Hybrid CORDIC algorithms,” IEEE Transactions on Computing, 1997, vol. 46, pp. 1202 1207.
8. D. De Caro, N. Petra and G. M. Strollo, “A 380 MHz direct digital synthesizer/mixer with hybrid CORDIC architecture in 0.25-micron CMOS,” IEEE Journal of Solid-State Circuits, 2007, vol. 42, pp. 151 160.
9. T. Rodrigues and J. E. Swartzlander, “Adaptive CORDIC: using parallel angle recoding to accelerate rotations,” IEEE Transactions on Computers, 2010, vol. 59, pp. 522 531.