IMPLEMENTATION OF DDFS ARCHITECTURE USING CORDIC ALGORITHM

A Project report submitted in partial fulfilment of the requirements for

the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

T. AVINASH (318126512108)   B.V.J. ASHRITHA (318126512067)

S. SREEVALLI (318126512104)   N. SURAJ (318126512091)

Under the guidance of

Mr. N. Srinivasa Naidu, M.Tech, (Ph.D)

(Assistant Professor)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

Autonomous status accorded by UGC and Andhra University

Approved by AICTE, Permanently Affiliated to Andhra University
Accredited by NBA (IT, CSE, EEE, ECE, MECH, CHEM and CIVIL) & accredited by NAAC

Sangivalasa – 531162, Bheemunipatnam (Mandal), Visakhapatnam (District)

(2021-2022)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(UGC AUTONOMOUS)

Autonomous status accorded by UGC and Andhra University

Approved by AICTE, Permanently Affiliated to Andhra University
Accredited by NBA (IT, CSE, EEE, ECE, MECH, CHEM and CIVIL) & accredited by NAAC

Sangivalasa – 531162, Bheemunipatnam (Mandal), Visakhapatnam (District)

PRAGNANAM BRAHMA

ANITS

CERTIFICATE

This is to certify that the project report entitled "**IMPLEMENTATION OF DDFS ARCHITECTURE USING CORDIC ALGORITHM**" submitted by **T. AVINASH (318126512108), B.V.J. ASHRITHA (318126512067), S. SREEVALLI (318126512104), N. SURAJ (318126512091)** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Engineering in Electronics and Communication Engineering** of Andhra University, Visakhapatnam is a record of bonafide work carried out under my guidance and supervision.

Project Guide

Mr. N. Srinivasa 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. Rajyalakshmi

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. Srinivasa Naidu,** Assistant Professor, Department of Electronics and Communication Engineering, ANITS, for his guidance with unsurpassed knowledge and immense encouragement. We are grateful to **Dr. V. Rajyalakshmi**, 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. At last but not the least, we thank everyone for supporting us directly or indirectly in completing this project successfully.

PROJECT STUDENTS

T. AVINASH	(318126512108)
B.V.J. ASHRITHA	(318126512067)
S. SREEVALLI	(318126512104)
N. SURAJ	(318126512091)

ABSTRACT

In this project an efficient approach is proposed in implementing DDFS architecture using CORDIC algorithm. 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 from a fixed clock frequency. It can offer fast switching between output frequencies, fine frequency resolution, and operation over a broad spectrum of frequencies. DDFS architecture can be implemented using ROM/lookup table approach.

CORDIC based DDFS architecture is a process where we design a new kind of architecture using the finest CORDIC algorithm in which we replace the ROM with CORDIC element to save memory. Coordinate rotation digital computer (CORDIC) is an efficient algorithm for computations of trigonometric functions. Scaling-free-CORDIC is one of the famous CORDIC implementations with advantages of speed and area. After describing the algorithm and its implementation in MATLAB, the project covers design techniques that can be applied to implement a DDFS architecture in VIVADO using Verilog Programming language. The output frequency results of the DDFS using LUT approach and DDFS using CORDIC algorithm is compared. The simulation results of the both approaches will be verified.

Keywords: CORDIC Algorithm, DDFS, lookup table, Verilog

CONTENTS

4. VERILOG

5. INTRODUCTION TO SOFTWARE TOOLS

List of Figures Page no.

List of Tables Page no.

List of Abbreviations		Page no.
PLL	- Phase Locked loop	5
DDFS	- Direct Digital Frequency Synthesizer	5
LUT	- Look Up Table	6
DAC	- Digital to Analog Converter	6
FSK	- Frequency Shift Keying	13
CORDIC	-Coordinate Rotation Digital Computer	16
DSP	- Digital Signal Processing	16
DIP	- Digital Image Processing	16
FPGA	- Field Programmable Gate Array	23
RTL	- Register Transfer Level	31
HDL	- Hardware Description Language	31
VHDL	- Verilog Hardware Description Language	36
MATLAB	- Matrix Laboratory	36
GUI	- Graphical User Interface	36
ISE	- Integrated Synthesis Environment	37

CHAPTER -1

INTRODUCTION

1.1 Introduction to Sine and Cosine

Sine is a trigonometric function of an angle. It has a number of properties, such as being periodic and odd. In the context of a triangle, for the specified angle, sine is the ratio of the length of the side that is **OPPOSITE** the angle to the length of the longest side, or **HYPOTENUSE**, of the triangle.

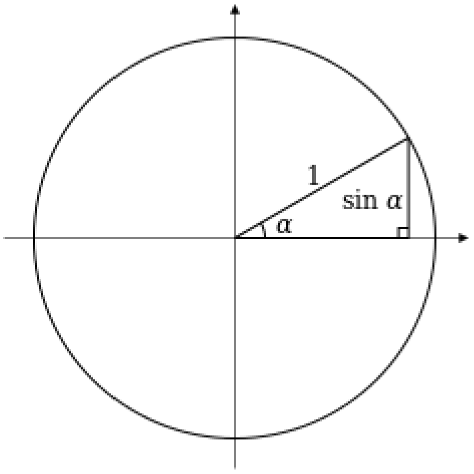


Figure 1.1 Representation of sine angle

sin(α)=opposite/hypotenuse

Like mentioned above, the sine function is **periodic**, which means that it is a function returning to the same value at regular intervals. Sine has a period of 2π, which means we can write it as:

sin (α) = sin(α+2π)

sin (α) = sin(α+2kπ), k ∈ all integers

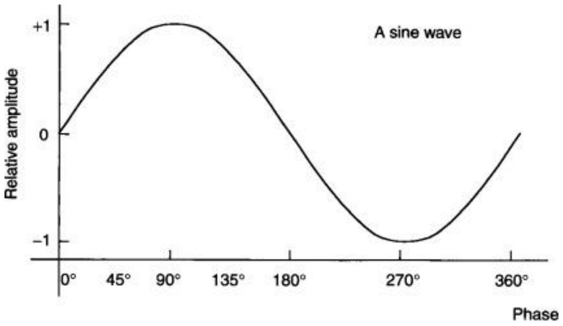


Figure 1.2 Sinusoidal Waveform

The sine of any angle can vary from −1 to +1. For example, the sine of 0° is 0 and the sine of 90° is 1. The sine of 270° is −1 and when we get to 360°, we are back to zero again**.**

Cosine is a trigonometric function of an angle. It has a number of properties, such as being periodic and even. In the context of a triangle, for the specified angle, cosine is the ratio of the length of the side that is **ADJACENT** the angle to the length of the longest side, or **HYPOTENUSE**, of the triangle.

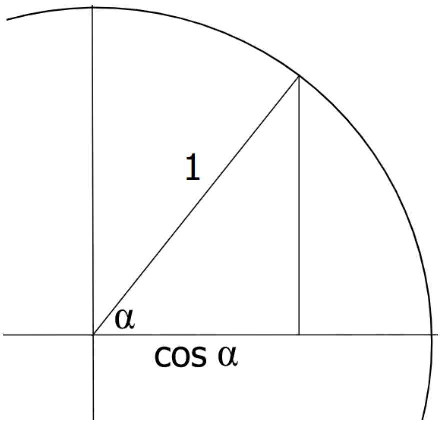


Figure 1.3 Representation of Cosine angle

Once a triangle to analyse is chosen, we can write:

cos(α)=adjacent/hypotenuse

Like mentioned above, the cosine function is **periodic**, which means that it is a function returning to the same value at regular intervals. Cosine has a period of 2π, which means we can write it as:

cos(α)=cos(α+2π)

or, in a more general sense,

cos (α)= cos(α+2kπ), k ∈ all integers

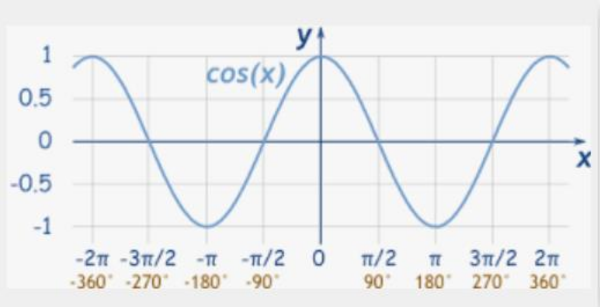


Figure 1.4 Cosine waveform

1.2 Different types of methods for implementing sine and cosine waves

A key requirement across a multitude of industries is to accurately produce, easily manipulate, and quickly change waveforms of various frequencies and types. Whether a wideband transceiver requires an agile low-phase-noise frequency source with excellent spurious-free dynamic performance or an industrial measurement and control system needs a stable frequency stimulus, the ability to quickly, easily, and cost effectively generate an adjustable waveform while maintaining phase continuity is a critical design which is required.

It is not uncommon to need a sine wave but how do you generate it? The "best" or most appropriate method for a particular application depends on several things such as:

- Frequency,
- Purity required,
- Amplitude,
- Possible synchronization with another frequency,
- Variable frequency and/or amplitude.

There are different types of methods for generating sine and cosine waveforms:

1.2.1 Wien Bridge Oscillator

1.2.2 Phase-shift Oscillator

1.2.3 PLL Method

1.2.4 DDFS

1.2.1 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 **Figure 1.5**

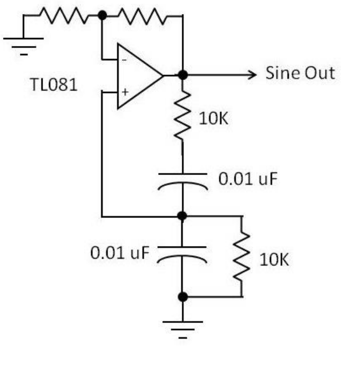


Figure 1.5 Wien bridge oscillator

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:

$$f = \frac{1}{2\pi RC}$$

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.

1.2.2 Phase-shift Oscillator

A popular way to make a sine 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. Figure shows one popular variation.

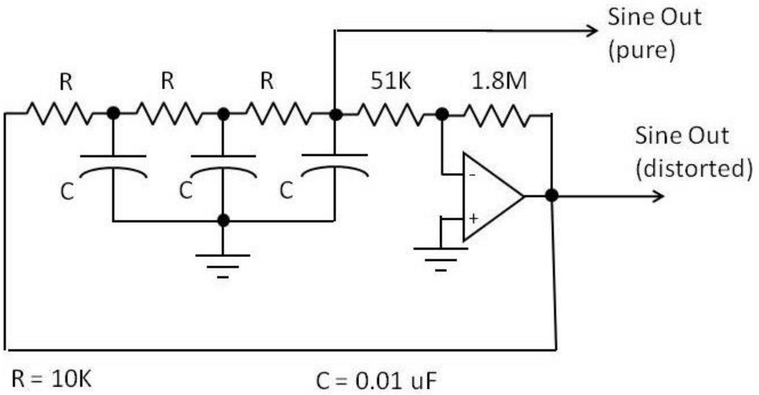


Figure 1.6 Phase-shift Oscillator

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:

$$f = \frac{1}{2.6RC}$$

In the circuit of Figure, the frequency should be about 3.85 kHz.

A fixed frequency is a disadvantage, but for a single frequency is good. The pure output needs to be buffered with an op-amp follower if you are going to drive a load.

1.2.3 Phase Locked Loop

A phase-locked loop is a feedback loop comprising: a phase comparator, a divider, and a voltage-controlled oscillator (VCO). The phase comparator compares a reference frequency with the output frequency (usually divided down by a factor, *N*), The error voltage generated by the phase comparator is applied to the VCO, which generates the output frequency. When the loop has settled, the output will bear an accurate relationship to the reference in frequency and/or phase. PLLs have long been recognized as superior devices for low phase noise and high spurious-free dynamic range (SFDR) applications requiring high fidelity and stable signals in a specific band of interest.

Their inability to accurately and quickly tune the frequency output and waveform and their slow response limits their suitability for applications such as agile frequency hopping and some frequency- and phase-shift keying applications.

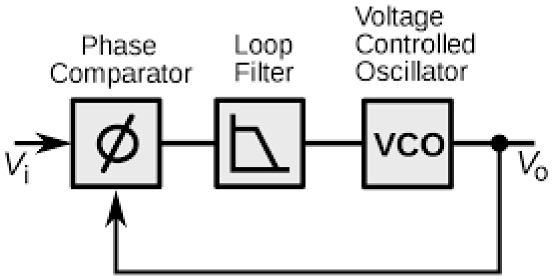


Figure 1.7 Phase locked loop

1.2.4 Direct Digital Frequency Synthesizer

Direct Digital Frequency Synthesis (DDFS) is a multi-step process of generating sinusoidal analogue waveforms. DDFS has a wide application in the modern communication era such as radio receivers, mobile telephones, radio telephones, walkie-talkies, CB radios, satellite receivers and none the less GPS systems. Traditional designs found in literature of high bandwidth frequency synthesizers make use of a Phase Locked-Loop (PLL) approach. The PLL offers very good wide tuning bandwidth due to the use of a programmable divider as compared to DDFS approach. On the other hand, DDFS provides many significant advantages such as fast settling time, sub-Hertz frequency resolution, continuous-phase switching response and low phase noise. One key design parameter of the DDFS is a Look Up Table (LUT). The response time, the power consumption and the size of the DDFS approach are factors that depend on the size of the LUT. In addition to that, the resolution and the size of DDFS are also dependable on the size of the phase accumulator.

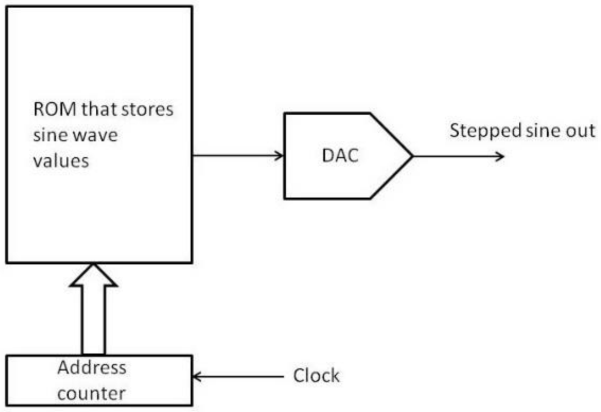


Figure 1.8 Direct digital frequency synthesiser.

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-analogue 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 analogue output signal that is proportional to the binary value from the ROM. What you get is a stepped approximation of a sine wave.

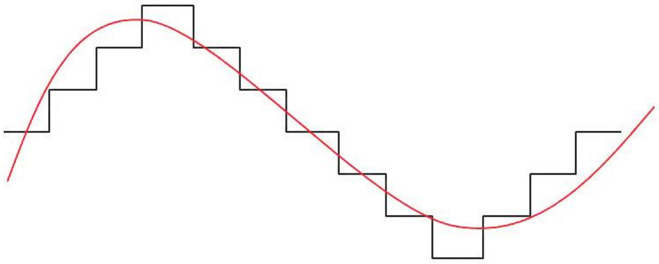


Figure 1.9 A stepped approximation of a sine wave

If you use enough samples and use more bits for the binary value, the steps will be smaller and a more fine-grained sine wave will occur. The frequency of the sine wave depends on the number of samples or values you use for the sine wave and the frequency of the clock signal that reads the values out of the ROM. If the steps are too large, you can pass the stepped signal through a low pass filter to smooth it out.

CHAPTER-2

DIRECT DIGITAL FREQUENCY SYNTHESIZER

2.1 Introduction to DDFS

Direct digital synthesis (DDS) is a method of producing an analogue waveform—usually a sine wave—by generating a time-varying signal in digital form and then performing a digital-to-analogue conversion. They are suitable for portable low battery drain trans receivers. They are also capable of being incorporated with different digital modulation by using different processing methods DDS devices are very compact and draw little power.

DDFS (Direct Digital Frequency Synthesizer) is a novel frequency synthesis technology with a huge relative bandwidth, quick frequency conversion time, high resolution, and outstanding phase consistency. This DDFS architecture is mostly used in modern communication**.**

2.2 Performance of DDFS

Digital synthesis is based on a phase accumulator which generates a series of digital states, the value of which increases linearly, forming a numeric ramp. This signal is made periodic and represents the instantaneous phase of the output waveform, from zero to 2pi radians. This is the digital input to a lookup table which converts the numeric ramp into a sine wave. While the most common DDS output waveform is the sine wave; ramps, triangle waves, and square waves are also easily generated.

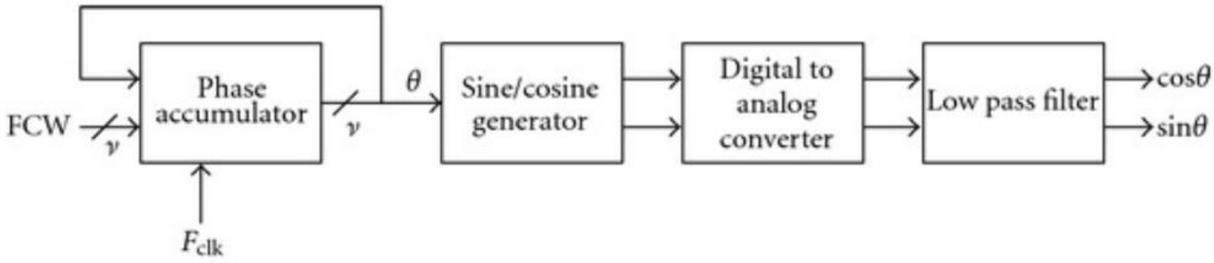


Figure 2.1 DDFS Architecture

The direct digital synthesizer is based on a phase accumulator which generates the instantaneous phase of a waveform. A lookup table provides the phase to amplitude conversion which is applied to a digital-to-analogue converter, producing the desired analogue output after filtering.

The output of the phase to amplitude lookup table is sent to a digital-to-analogue converter (DAC) and is converted into an analogue waveform, which is most commonly sinusoidal. Since the input to the DAC is a series of sampled values, the output has quantization steps. These steps produce spectral images at multiples of the sample rate in the frequency domain which are not desired. A low-pass filter, placed after the DAC, suppresses these unwanted spectral responses.

The phase accumulator:

The phase accumulator is a modulo N counter that has 2^N digital states which are incremented for each system clock input pulse. The size of the increment depends on the value of the tuning word, M, applied to the accumulator adder stage. The tuning word fixes the step size of the counter increment. This will determine the frequency of the output waveform.

The phase accumulator generally has from 24 to 48 bits; at 24 bits there are 2^{24} or 16,777,216 states. This number represents the number of phase values between 0 and 2p radians, or the achievable phase increment. For a 24-bit phase accumulator, the phase resolution is 3.74 E-7 radians. If a larger phase accumulator is used, the phase increment becomes even finer.

One way of visualizing the operation of the phase accumulator is to look at the accumulator operation as a phase wheel

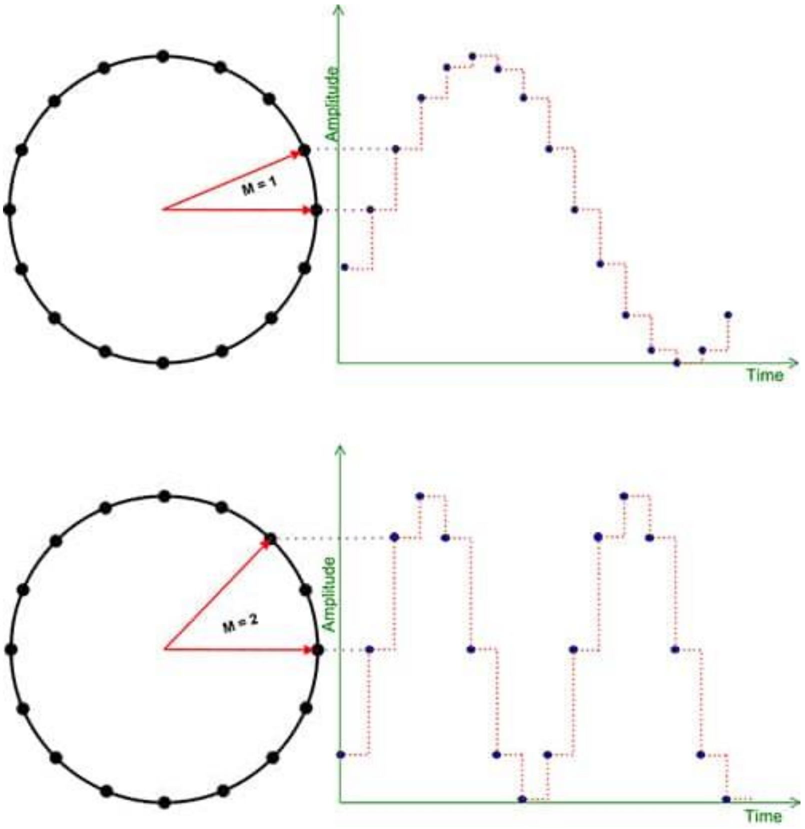


Figure 2.2 Digital Phase Wheel

A simplified view of a 16-state phase accumulator operation using a phase wheel to visualize how the tuning word affects the output frequency of the DDS. (Image source: Digi-Key Electronics)

The accumulator states are periodic and are represented as lying on a circle. Dots on the circle represent all the phase states of the accumulator. In this case, for simplicity, the accumulator has 16 states. If the tuning word is equal to one, as in the top diagram, then the step increment at each clock is one, and all states are selected during the full period.

Projected to the right of the phase wheel is the analogue output for each state. As this is a quantized device, the analogue output holds its current state until the clock advances the phase wheel to its next state. The output waveform consists of a single cycle of the quantized sine wave containing sixteen values.

In the lower diagram the tuning word value is set to two. With this setting, every other state on the phase wheel is selected. The analogue output now consists of two cycles, each with eight amplitudes, giving a total of sixteen states. With the tuning word set to two, the output frequency is now twice the previously obtained value.

The output frequency of the DDS is set by the tuning word value and increases proportionally to the value of the tuning word. The sample rate remains fixed at the system clock rate, and the time between output samples is constant. The output frequency depends on the tuning word increment, so as the tuning word value increases there are fewer steps in each output cycle, thereby increasing the frequency. The tuning word can be increased until there are only two samples per cycle, which brings the DDS output to its Nyquist frequency, or half the system clock rate. Generally, the DDS is limited by design to always have an output frequency that is less than the Nyquist limit.

Along with the system clock frequency, the output frequency of the DDS is also dependent upon the tuning word value, and the length of the accumulator. It is expressed by Equation.

$$f_{out} = \frac{M * f_C}{2^N} \quad \text{……………………………………………………………………} \quad (2.1)$$

Where:

f_{out} is the DDS output frequency

M is the tuning word value

f_c is the system clock frequency

N is the length of the phase accumulator

The output of the phase accumulator, which is the instantaneous phase of the output waveform, is used to drive the phase to amplitude converter. The phase to amplitude converter outputs a digital word, the value of which is the amplitude of the sine waveform for the input phase.

Note that the number of bits used to drive the phase to amplitude converter is less than that used for the phase accumulator. This is referred to as phase truncation and is used to reduce the die area and power consumption of the digital stages after the phase accumulator. While it does cause some spurious spectral components, called truncation spurs, they are minimized by careful design.

E.g. Let our required frequency be f_0= 1 KHz and let N=5 bits and for easy simplification f_{clk}= 32 KHz

Now,

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

$1 \quad = W * 32 / 32$

$W = 1$

Sine Values	N-bit binary numbers
Sin(0) = 0	00000
Sin(5) = 0.0871	00001
Sin(10) = 0.173	00010
Sin(15) = 0.2588	00011
Sin(20) = 0.342	00100

Table 2.1 Sine angle generation table

- Initially let N-bit numbers be N = 00000
- This N is used as an index to ROM, now o/p will be 0
- For next clock pulse, N=00001 then o/p = 0.0871
- For next clock pulse, N=00010 then o/p = 0.173
- For next clock pulse, N=00011 then o/p = 0.2588
- For next clock pulse, N=00100 then o/p = 0.342
- So, to generate a value of 20 degrees, it takes 4 clock cycles.

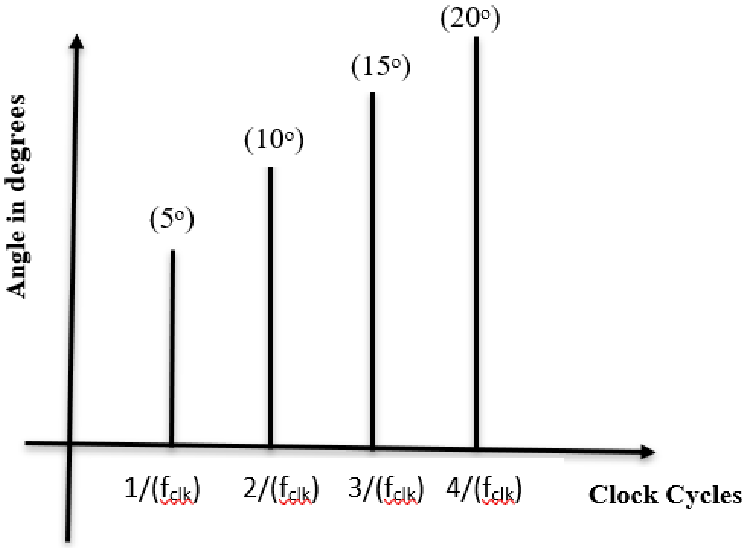


Figure 2.3 Graphical representation of sine angle generation (1)

E.g., Now let the required frequency be f_0 = 2 KHz

$2 = W * 32/ 32$

$W = 2$

- Let N = 00000 be the initial value then the o/p will be 0
- For 1st clock cycle, N = 00010 then the o/p = 0.173
- For 2nd clock cycle, N = 00100 then the o/p = 0.342
- So, here to generate a value of 20 degrees, we just require 2 clock cycles i.e., the wave is compressed when compared to that of f_0 = 1 kHz.

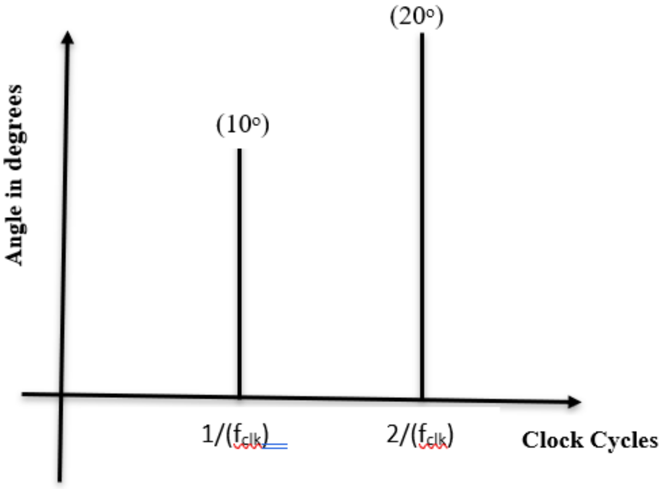


Figure 2.4 Graphical representation of sine angle generation

2.3 Improved DDFS Architecture:

The basic design of DDFS architecture 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.

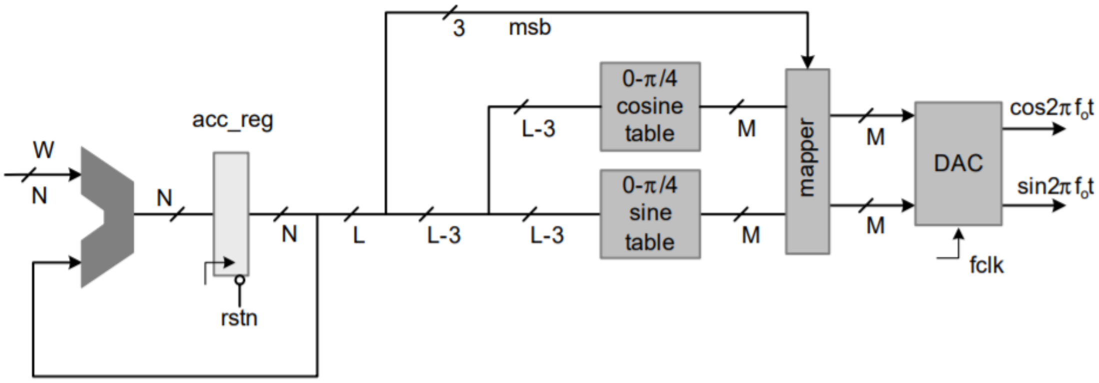


Figure 2.5 Modified DDFS Architecture

A complete period 0 to 2pi of sine and cosine waves can be generated from values of the two signals from 0 to pi/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 two 2^{L-3} memories of width M. The design takes up a large area and dissipates significant power. 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.

Three MSB bits	Remaining angles
000	$0 - \pi/4$
001	$\pi/4 - \pi/2$
010	$\pi/2 - 3\pi/4$
011	$3\pi/4 - \pi$
100	$\pi - 5\pi/4$
101	$5\pi/4 - 3\pi/2$
110	$3\pi/2 - 7\pi/4$
111	$7\pi/4 - 2\pi$

Table 2.2 Angles to the values stored in LUT

2.4 Frequency Shift Keying (FSK) using DDFS:

Binary frequency-shift keying (usually referred to simply as FSK) is one of the simplest forms of data encoding. The data is transmitted by shifting the frequency of a continuous carrier to one of two discrete frequencies (hence binary). One frequency, f_1, (perhaps the higher) is designated as the mark frequency (binary one) and the other, f_0, as the space frequency (binary zero). Figure 2.6 shows an example of the relationship between the mark-space data and the transmitted signal.

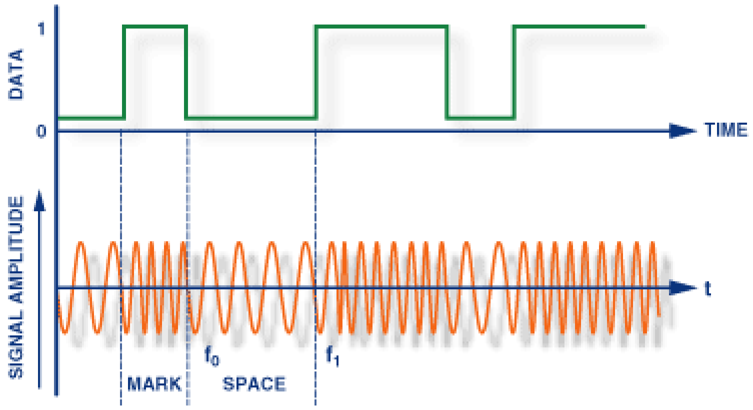


Figure 2.6 FSK Modulation

This encoding scheme is easily implemented using a DDFS. The DDFS frequency tuning word, representing the output frequencies, is set to the appropriate values to generate f_0 and f_1 as they occur in the pattern of 0s and 1s to be transmitted. The user programs the two required tuning words into the device before transmission. In this case, the MUX will be used to select the appropriate frequency word. The 2 x 1 MUX contains two selection lines (s_1, s_0) in which the Modulating signal is given as input which contains the data in binary format which is either 0 or 1. If the Data is bit-0 then first tuning word will be given as input to the DDFS through MUX and the respective frequency output will be obtained. If the Data is bit-1 then second tuning word will be given as input to the DDFS through MUX. The block diagram in Figure 2.7 demonstrates a simple implementation of FSK encoding.

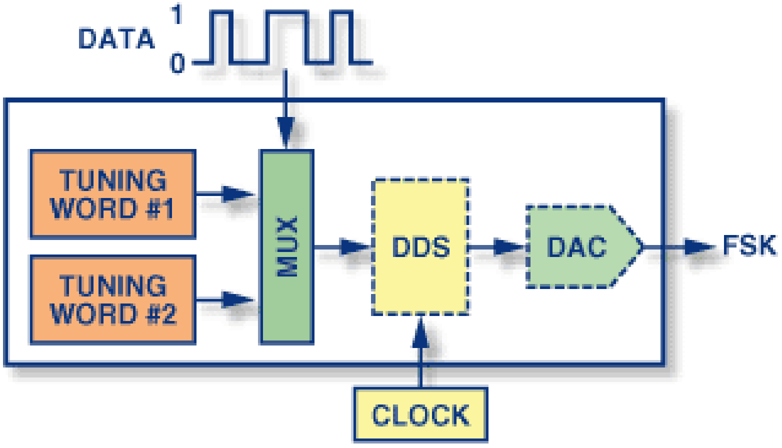


Figure 2.7 A DDFS-based FSK Modulator

2.5 Verilog Program for implementing Basic DDFS

In the Basic DDFS the required inputs are clock input, frequency control word which are given to the phase accumulator for successive increments. The ROM table should be created for storing the values of all samples of sine. The all required sine samples are obtained from MATLAB and then the ROM table of 1024 values of each size 16 bits are created and stored as memory file in VIVADO. The accumulator register is initialized and then after every clock cycle the accumulator register is incremented by frequency control word, then the first 10-bits of the accumulator register indicates the address of the sine values stored in the ROM. The obtained values are assigned to the output which are in digital form and can viewed in Analog form using VIVADO Simulator.

Program:

```
timescale 1ns / 1ps
//module creation
module sine_dds(
      input clk ,
      input [31:0] fcw,
      output [15:0] dds_sin
);
reg signed [15:0] rom_memory [1023:0];  //ROM memory creation
initial begin
   $readmemh("sine.mem", rom_memory);
end
  reg [31:0] accu;
  wire [9:0] lut_index;
initial begin
accu <= 32'd0; // Accumulator is initialized with the zeros
end
always@(posedge clk)
begin
      accu <= accu + fcw;
     // Accumulator is incremented with fcw for every clock cycle
end
```

```
assign lut_index = accu[31:22];

 // The first 10 bits will be the index of ROM

assign dds_sin = rom_memory[lut_index];

// The value at that address will be assigned to the output

endmodule
```

Calculation:

The output frequency of a DDFS device is determined by the given formula for output frequency. The length of the phase accumulator is the length of frequency control word, which determines the degree of frequency control word resolution of the DDFS implementation. Let’s find the frequency control word for an output frequency of 5 KHz where reference clock is 100 MHz and control word length is 32 bits (binary). The Resulting equation would be:

5000 = (fcw x 100MHz)/ (2^32)

fcw = (2^32 x 5000)/(100MHz)

fcw = 214750

Loading this value of fcw into the frequency control register would result in a output frequency of 5KHz, given a reference clock frequency of 100MHz.

CHAPTER – 3

CORDIC METHODOLOGY

3.1 Introduction to CORDIC

CORDIC (for Coordinate Rotation Digital Computer), also known as Volder's algorithm, is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions, typically converging with one digit (or bit) per iteration. CORDIC is used to calculate trigonometric, hyperbolic functions, square roots, multiplications, divisions etc. These can be achieved by arbitrary base, typically converging with bit per iteration. CORDIC is therefore also an example of digit-by-digit algorithms. CORDIC and closely related methods known as pseudo multiplication and pseudo-division or factor combining are commonly used when no hardware multiplier is available (e.g., in simple microcontrollers and FPGAs), as the only operations it requires are addition, subtraction, bit shift and table lookup. As such, they belong to the class of shift-and-add algorithms.

3.1.1 Importance of CORDIC

CORDIC Algorithm is applicable for square root, logarithmic, exponential function and for digital computer. Unit trigonometric functions are crucial functions like sine cosine functions can be computed easily.

Present technology and limitations on power, operating frequency and energy consumption, on generating trigonometric functions using multiplier divider adder takes more time and complex. To reduce CORDIC algorithm converted to hardware known as CORDIC processor finally it reduces the use of hardware multiplier.

3.1.2 CORDIC Applications

- Signal And Image Processing
- Communication Systems
- Robotics
- 3D Graphs
- Aerospace Application
- Different DSP And DIP Filters
- Network Security
- Biometric

3.2 Modes of Operation

3.2.1 Rotation Mode

CORDIC can be used to calculate a number of different functions. This explanation shows how to use CORDIC in rotation mode to calculate the sine and cosine of an angle, assuming that the desired angle is given in radians and represented in a fixed-point format. To determine the sine or cosine for an angle, the *y* or *x* coordinate of a point on the unit circle corresponding to the desired angle must be found. Using CORDIC, one would start with the vector v_0.

$$v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

In the first iteration, this vector is rotated 45° counter clockwise to get the vector . Successive iterations rotate the vector in one or the other direction by size-decreasing steps, until the desired angle has been achieved.

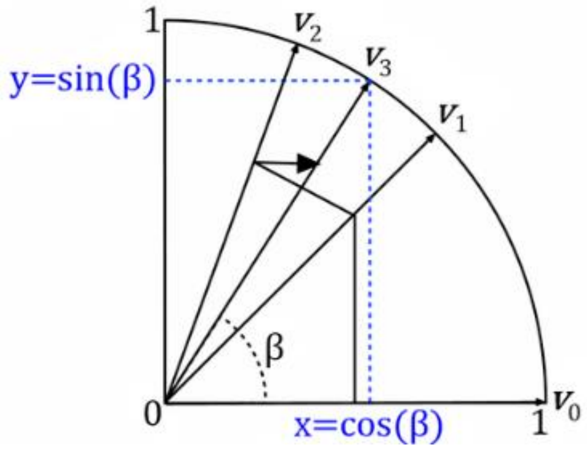


Figure 3.1 Illustration of the CORDIC Algorithm

$$v_{i+1} = R_i v_i . \quad \text{....……………………………………………………………….. (3.1)}$$

The rotation matrix is given by

$$R_i = \begin{bmatrix} \cos(\gamma_i) & -\sin(\gamma_i) \\ \sin(\gamma_i) & \cos(\gamma_i) \end{bmatrix}$$

Using the following two trigonometric identities:

$$\cos(\gamma_i) = \frac{1}{\sqrt{1+\tan^2(\gamma_i)}},$$
$$\sin(\gamma_i) = \frac{\tan(\gamma_i)}{\sqrt{1+\tan^2(\gamma_i)}},$$

The rotation matrix becomes

$$R_i = \frac{1}{\sqrt{1+\tan^2(\gamma_i)}}\begin{bmatrix} 1 & -\tan(\gamma_i) \\ \tan(\gamma_i) & 1 \end{bmatrix}$$

The expression for the rotated vector is given by

$$v_{i+1} = R_i v_i$$

$$v_{i+1} = \frac{1}{\sqrt{1+\tan^2(\gamma_i)}}\begin{bmatrix} 1 & -\tan(\gamma_i) \\ \tan(\gamma_i) & 1 \end{bmatrix}\begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

…………………… (3.2)

Where x_i and yi are the components of v_i . Restricting the angles y_i such that $\tan(\text{ɣ}_i) = \pm 2^{-i}$, the multiplication with the tangent can be replaced by a division by a power of two, which is efficiently done in digital computer hardware using a bit shift. The expression then becomes

$$v_{i+1} = K_i\begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix}\begin{bmatrix} x_i \\ y_i \end{bmatrix},$$

Where,

$$K_i = \frac{1}{\sqrt{1+2^{-2i}}},$$

And, σ_i is used to determine the direction of the rotation: if the angle ɣ_i is positive, then σ_i is +1, otherwise it is −1.

K_i can be ignored in the iterative process and then applied afterward with a scaling factor

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}},$$

…………………………………….…… (3.3)

which is calculated in advance and stored in a table or as a single constant, if the number of iterations is fixed. This correction could also be made in advance, by scaling V_0 and hence saving a multiplication. Additionally, it can be noted that

$$K = \lim_{n\to\infty} K(n) \approx 0.6072529350088812561694$$

to allow further reduction of the algorithm's complexity. Some applications may avoid correcting for K altogether, resulting in a processing gain A :

$$A = \frac{1}{K} = \lim_{n\to\infty} \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}} \approx 1.64676025812107.$$

After a sufficient number of iterations, the vector's angle will be close to the wanted angleβ. For most ordinary purposes, 40 iterations (n = 40) is sufficient to obtain the correct result to the 10th decimal place.

The only task left is to determine whether the rotation should be clockwise or counter clockwise at each iteration (choosing the value of β). This is done by keeping track of how much the angle was rotated at each iteration and subtracting that from the wanted angle; then in order to get closer to the wanted angle β, β_{n+1} if is positive, the rotation is clockwise, otherwise it is negative and the rotation is counter clockwise:

$$\beta_0 = \beta$$

$$\beta_{i+1} = \beta_i - \sigma_i \gamma_i, \quad \gamma_i = \arctan(2^{-i}). \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3.4)$$

The values of γ_n must also be precomputed and stored. But for small angles, arc tan (γ_n) = γ_n in fixed-point representation, reducing table size.

As can be seen in the illustration above, the sine of the angle β is the *y* coordinate of the final vector V_n while the *x* coordinate is the cosine value.

3.2.2 Vectoring Mode

In this type of mode, the y-axis of the input vector is required to be zero. So, this mode calculates the phase and magnitude of the input vector. The rotation-mode algorithm described above can rotate any vector (not only a unit vector aligned along the *x* axis) by an angle between −90° and +90°. Decisions on the direction of the rotation depend on being positive or negative.

The vectoring-mode of operation requires a slight modification of the algorithm. It starts with a vector the *x* coordinate of which is positive and the *y* coordinate is arbitrary. Successive rotations have the goal of rotating the vector to the *x* axis (and therefore reducing the *y* coordinate to zero).

At each step, the value of *y* determines the direction of the rotation. The final value of contains the total angle of rotation. The final value of *x* will be the magnitude of the original vector scaled by *K*. So, an obvious use of the vectoring mode is the transformation from rectangular to polar coordinates.

3.3 Methodology Used

A convergence method for evaluating trigonometric functions

- If a unit-length vector with end point at (X, Y) = (1,0) is rotated by an angle Z, its new end point will be at (X, Y) = (cos z, sin z).
- Simple hardware – shifters, adders, lookup table.
- Family of algorithms: Rotation, Vector mode
    1. Circular rotations
    2. Linear rotations
    3. Hyperbolic rotations

3.4 Real CORDIC Rotations

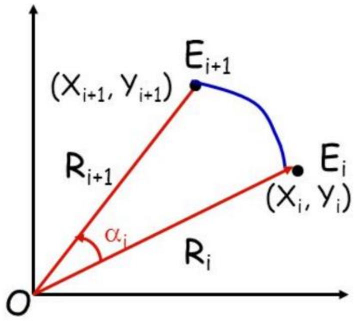


Figure 3.2 Real CORDIC Rotations

If vector OE_i is rotated about the origin by an angle I, the new vector OE_{i+1} will have the coordinates

Real rotation: E_{i+1}

$X_{i+1} = X_i \cos \alpha_i - Y_i \sin \alpha_i$……………………………………………….………… (3.5)

$Y_{i+1} = Y_i \cos \alpha_i + X_i \sin \alpha_i$……………………………………………………... (3.6)

$Z_{i+1} = Z_i - \alpha_i$……………………………………………………….…….……..…. (3.7)

The variable Z allows us to keep track of the total rotation over several steps. If Zo is the initial rotation goal and if the α_i angles are selected at each step such that after n iterations Za tends to 0, then E_1 will be the end point after rotation by angle Zo.

3.5 Pseudo CORDIC Rotations

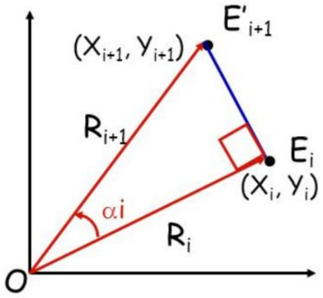


Figure 3.3 Pseudo CORDIC rotations

Pseudo rotations increase the vector length to

$R_{i+1} = R_i\,(1 + \tan^2 \alpha_i)^{1/2}$……………………………………………..……..… (3.8)

Pseudo rotation: E'_{i+1}

$X_{i+1} = X_i - Y_i \tan \alpha_i$………………………………………………….…..…..... (3.9)

$Y_{i+1} = Y_i + X_i \tan \alpha_i$……………………………………………..………... (3.10)

$Z_{i+1} = Z_i - \alpha_i$…………………………………………..………………... (3.11)

3.6 Basic CORDIC Rotations:

To simplify pseudo rotations, pick α_i such that $\tan \alpha_i = d_i 2^{-i}$ where $d_i \in \{-1,1\}$. Then

$X_{i+1} = X_i - Y_i\, d_i 2^{-i}$……………………………………………………….….. (3.12)

$Y_{i+1} = Y_i + X_i\, d_i 2^{-i}$…………………………………………...…………... (3.13)

$Z_{i+1} = Z_i - d_i \tan^{-1} 2^{-i}$…………………………………………. ……….....(3.14)

Computation of X_{i+1} and Y_{i+1} requires an i-bit right shift and an add/subtract; Z_{i+1} only requires an add/subtract and one table lookup. Precompute and store the function $\tan^{-1} 2^{-i}$.

Choosing angles:

The angles to be taken for every ith iteration is precomputed and stored for easy access of all the values in computing sine and cosine at every iteration. The below table 3.1 shows the values of angles to chosen.

i	**$\alpha_i = 2^{-i}$**	**$E_i = \tan^{-1}2^{-i}$**	**d_i**	**$Z_i - d_iE_i = z_{i+1}$**
0	1.000000	45.0000000	1	30.00-45.00 = -15.00
1	0.500000	26.565051	-1	-15.00 +26.57 = 11.57
2	0.250000	14.036243	1	11.57 – 14.04 = -2.47
3	0.125000	7.125016	-1	-2.47 + 7.13 =4.66
4	0.062500	3.576334	1	4.66 – 3.58 =1.08
5	0.031250	1.789910	1	1.08 -1.79 = -0.71
6	0.015625	0.895174	-1	-0.71+0.90 = 0.19
7	0.007813	0.447614	1	0.19 -0.45 = -0.26
8	0.003906	0.223811	-1	-0.26 +0.22 =-0.04
9	0.001953	0.111906	-1	-0.04 + 0.11 =0.07

Table 3.1 Choosing the predefined angles

For example ,

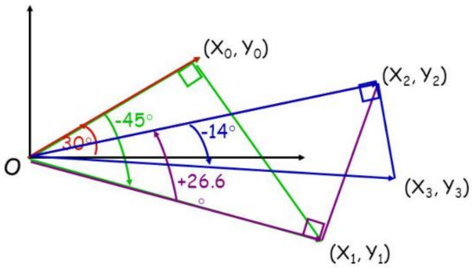


Figure 3.4 Illustration of the first three rotations for a Z of 30^0

If we want to calculate sine and cosine of a required angle, we should rotate the vector from the initial position by few successive angles which are predefined in the ROM. The angles required for every rotation which are stored in ROM are tabulated in Table 3.1.

For example, if required angle is 30^0 then make Z = 30^0 and then compute the further iterations as shown in the Figure 3.4.

3.7 Hardware Mapping

CORDIC is generally faster than other approaches when a hardware multiplier is unavailable (e.g., in a microcontroller) or when the number of gates required to implement the function is to be minimized (e.g., in an FPGA). On the other hand, when a hardware multiplier is available (e.g., in a DSP microprocessor), table lookup methods and power series are generally faster than CORDIC.

A straight forward hardware implementation for CORDIC arithmetic is shown below. It requires three registers for x, y and z, a look up table to store the values of $\alpha i=\tan^{-1} 2^{-i}$ two shifter to supply the terms 2^{-i} x and 2^{-i} y to the adder/subtractor units.

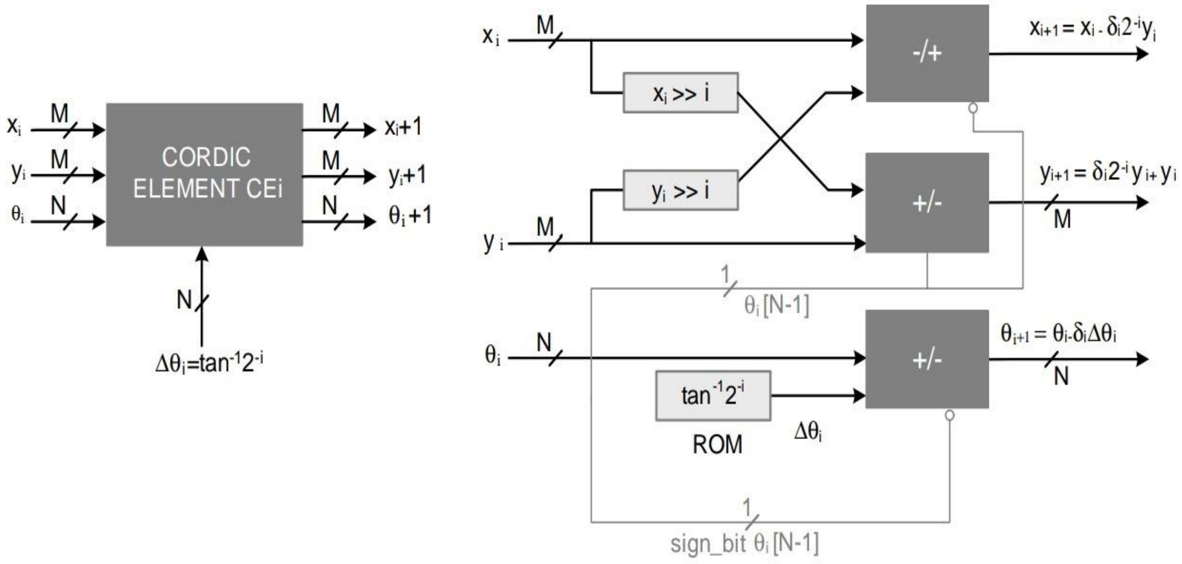

x_i
y_i
θ_i
M
M
N
CORDIC ELEMENT CEi
M
M
N
x_i+1
y_i+1
θ_i+1
N
$\Delta\theta_i=\tan^{-1}2^{-i}$
x_i
M
$x_i >> i$
$y_i >> i$
y_i
M
-/+
+/-
+/-
$x_{i+1} = x_i - \delta_i 2^{-i} y_i$
$y_{i+1} = \delta_i 2^{-i} y_i + y_i$
M
1
θ_i [N-1]
θ_i
N
$\theta_{i+1} = \theta_i - \delta_i \Delta\theta_i$
N
$\tan^{-1}2^{-i}$
ROM
$\Delta\theta_i$
1
sign_bit θ_i [N-1]

Figure 3.5 Hardware Mapping

These CEs are cascaded together for a fully parallel implementation. That the above CORDIC algorithm computes sin and cosine of a particular angle which should be in the range of -54.88 degrees to 54.88 degrees.

When we want to calculate the sin and cosine of 70 degrees the θ_0 is taken as 70 degrees and the CORDIC tries to make the resultant angle equal to 0 it applies a negative rotation $\Delta\theta=\tan^{-1} 2^{0}$ to bring the angle $\theta 1=16.44$. Two more negative rotations take the angle to the negative side with $\theta 3 = -4.73$. The algorithm now gives positive rotation$\Delta\theta 3 = \tan^{-1} 2^{-3}$ and keeps working to make the final angle equal to 0, and in the final iteration the angle $\theta_{16} = 0.0008$ degrees.

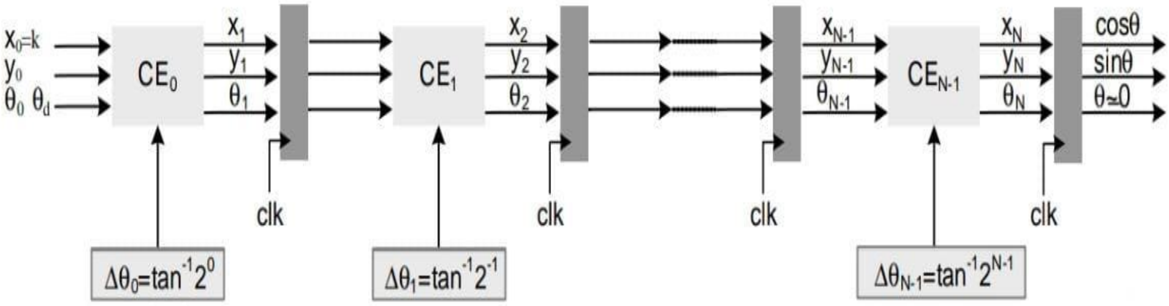


Figure 3.6 Pipelined FDA architecture of CORDIC algorithm

At first, initial coordinate values and required angle will be given to the first CORDIC element. At every CORDIC element, the pre-computed angle will be given as input to it. The corresponding result of the first CORDIC element will be the first iteration values which will be given to the next CORDIC element for the further iterations. For further iterations, the CORDIC elements will be cascaded serially one after the another. The number of CORDIC elements that we require depends on accuracy of the sine and cosine values. At final iteration, coordinates obtained will be sine and cosine of required angle.

3.8 Time Shared Architecture

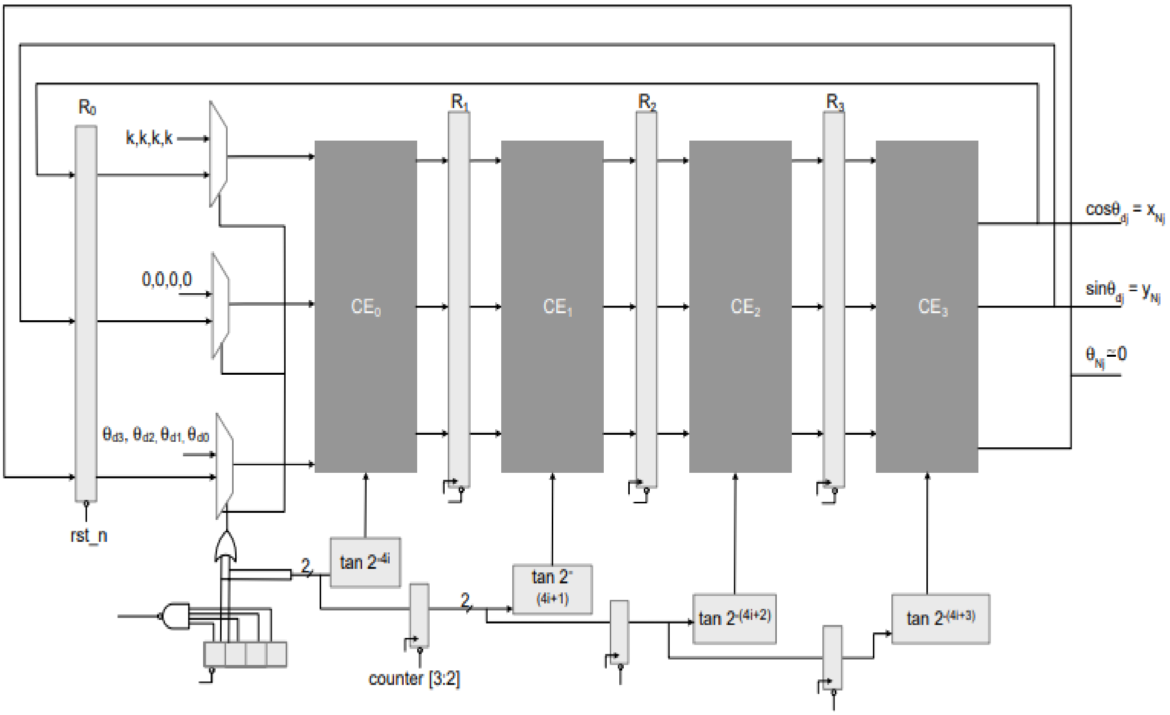
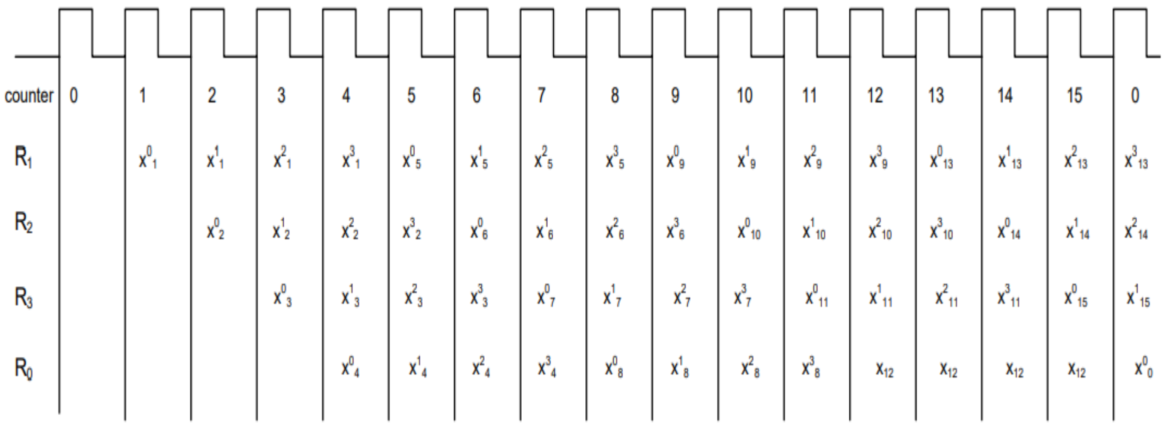


Figure 3.7 Four slow folded architecture by a folding factor of 4

Here in the before diagram, we have seen that the four CORDIC elements have been arranged in cascaded form and the output of last CORDIC element has been given to the first CORDIC element. Before giving to the first CORDIC element it is stored in register R0.

The main idea of this time-shared architecture is to achieve a greater number of outputs in less time interval or else we can say that in a smaller number of clock cycles. For achieving that we have arranged this CORDIC elements in this format.

The working mainly starts from taking four inputs and giving one after other inputs for every clock. After first clock the outputs of first CORDIC element will be stored in register R1. And for next cycle the output of first CORDIC element is given to next CORDIC element and the new input is given to the first CORDIC element. In the same way after four clock cycles all the four inputs will be given to the architecture. And after fixed number of cycles/iterations the outputs will be obtained.



counter	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
R_1		x^0_1	x^1_1	x^2_1	x^3_1	x^0_5	x^1_5	x^2_5	x^3_5	x^0_9	x^1_9	x^2_9	x^3_9	x^0_{13}	x^1_{13}	x^2_{13}	x^3_{13}
R_2			x^0_2	x^1_2	x^2_2	x^3_2	x^0_6	x^1_6	x^2_6	x^3_6	x^0_{10}	x^1_{10}	x^2_{10}	x^3_{10}	x^0_{14}	x^1_{14}	x^2_{14}
R_3				x^0_3	x^1_3	x^2_3	x^3_3	x^0_7	x^1_7	x^2_7	x^3_7	x^0_{11}	x^1_{11}	x^2_{11}	x^3_{11}	x^0_{15}	x^1_{15}
R_0					x^0_4	x^1_4	x^2_4	x^3_4	x^0_8	x^1_8	x^2_8	x^3_8	x_{12}	x_{12}	x_{12}	x_{12}	x^0_0

Figure 3.8 Timing diagram of 4 slow folded CORDIC architecture

In the above-mentioned Figure 3.8 we can observe the Timing diagram of 4 slow folded CORDIC architecture. In the first clock cycle the first input first iteration value stored in R_1 and after in second clock cycle the second input first iteration value stored in R_1 and first input second iteration value stored in R_2. And after further clock cycles all the outputs of the given inputs will calculated in fixed number of iterations because of this folder architecture, which is one of the biggest advantage.

3.9 CORDIC Based DDFS Architecture

Direct Digital Frequency Synthesis (DDFS) is a technique for creating an analogue waveform—usually a sine wave—by creating a time-varying signal in digital form and then converting it to analogue. It was written in Verilog and then emulated with XILINX VIVADO. DDFS is implemented in a variety of ways. The frequency control word is delivered to the phase accumulator in a conventional DDFS, which determines the DDFS' output frequency. The frequency control word is added to the accumulator register for each clock pulse. The phase of the output waveform is represented by the value in the accumulator register.

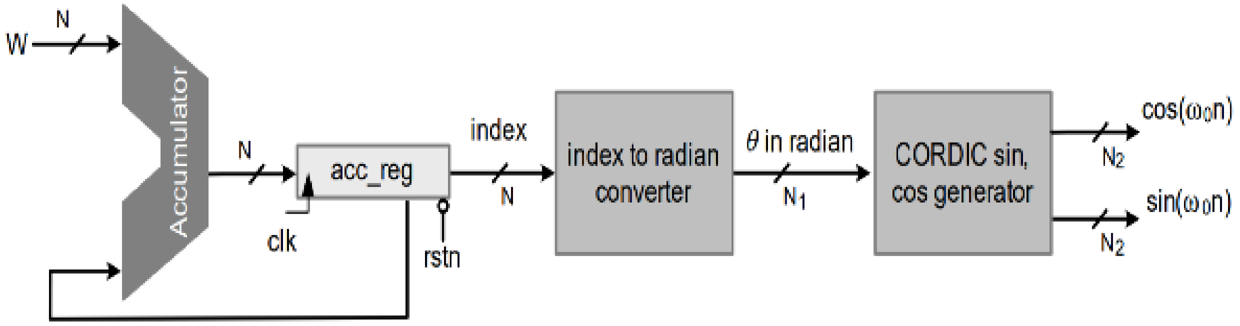


Figure 3.9 CORDIC based DDFS Architecture

When the frequency control word is large, fewer phase angles are taken, and the time period is reduced. When the frequency control word is tiny, a larger number of phase angles are taken, resulting in a longer waveform time period. The CORDIC architecture is used in this application. The CORDIC algorithm is used instead of ROM to compute the sin and cosine of the desired angle by rotating the original vector. Each clock pulse alters the phase angle in the accumulator register, and the CORDIC algorithm computes the sine and cosine of each phase angle in a fixed number of iterations to generate a digital sinusoidal waveform.

3.9.1 Calculations:

The output frequency of a DDFS device is determined by the above given formula for output frequency. The length of the phase accumulator is the length of frequency control word, which determines the degree of frequency control word resolution of the DDFS implementation. Let’s find the frequency control word for an output frequency of 1 KHz where reference clock is 100 MHz and control word length is 32 bits (binary). The Resulting equation would be:

1000 = (fcw x 100MHz)/ (2^32)

fcw = (2^32 x 1000)/(100MHz)

fcw = 42950

Loading this value of fcw into the frequency control register would result in a output frequency of 1KHz, given a reference clock frequency of 100MHz.

3.10 Verilog Implementation of CORDIC based DDFS architecture

In the CORDIC based DDFS architecture the required inputs are X_{in}, Y_{in} and angle. The 32 bit angle value is the accumulator register which is initialized and then after every clock cycle, it is incremented by frequency control word, then the 32 bit angle value will be given to the CORDIC element that computes the Sine and Cosine values of required angle. The obtained values are assigned to the output which are in digital form and can viewed in Analog form using VIVADO Simulator.

Program:

```
`timescale 1 ns/100 ps
 module procor (clock, angle, Xin, Yin, Cosine, Sine);
   parameter c = 16;   // bit width of input and output data
   localparam STG = c ; // similar bit width of vectors X and Y
   input  clock;
   input  signed  [31:0] angle;
   input  signed  [c-1:0] Xin;
   input  signed  [c-1:0] Yin;
   output signed  [c :0] Cosine;
   output signed  [c :0] Sine;
   wire signed [31:0] tan_inverse [0:30];
   // Assigning tan_inverse table
   assign tan_inverse[00] = 32'b00100000000000000000000000000000;
   // 45.000 degrees -> atan(2^0)
   assign tan_inverse[01] = 32'b00010010111001000000010100011101;
   // 26.565 degrees -> atan(2^-1)
   assign tan_inverse[02] = 32'b00001001111110110011100001011011;
   assign tan_inverse[03] = 32'b00000101000100010001000111010100;
   assign tan_inverse[04] = 32'b00000010100010110000110101000011;
   assign tan_inverse[05] = 32'b00000001010001011101011111100001;
   assign tan_inverse[06] = 32'b00000000101000101111011000011110;
   assign tan_inverse[07] = 32'b00000000010100010111110001010101;
   assign tan_inverse[08] = 32'b00000000001010001011111001010011;
   assign tan_inverse[09] = 32'b00000000000101000101111100101110;
   assign tan_inverse[10] = 32'b00000000000010100010111110011000;
   assign tan_inverse[11] = 32'b00000000000001010001011111001100;
   assign tan_inverse[12] = 32'b00000000000000101000101111100110;
   assign tan_inverse[13] = 32'b00000000000000010100010111110011;
   assign tan_inverse[14] = 32'b00000000000000001010001011111001;
   assign tan_inverse[15] = 32'b00000000000000000101000101111101;
   assign tan_inverse[16] = 32'b00000000000000000010100010111110;
   assign tan_inverse[17] = 32'b00000000000000000001010001011111;
   assign tan_inverse[18] = 32'b00000000000000000000101000101111;
   assign tan_inverse[19] = 32'b00000000000000000000010100011000;
   assign tan_inverse[20] = 32'b00000000000000000000001010001100;
   assign tan_inverse[21] = 32'b00000000000000000000000101000110;
   assign tan_inverse[22] = 32'b00000000000000000000000010100011;
   assign tan_inverse[23] = 32'b00000000000000000000000001010001;
   assign tan_inverse[24] = 32'b00000000000000000000000000101000;
   assign tan_inverse[25] = 32'b00000000000000000000000000010100;
   assign tan_inverse[26] = 32'b00000000000000000000000000001010;
   assign tan_inverse[27] = 32'b00000000000000000000000000000101;
   assign tan_inverse[28] = 32'b00000000000000000000000000000010;
   assign tan_inverse[29] = 32'b00000000000000000000000000000001;
   assign tan_inverse[30] = 32'b00000000000000000000000000000000;
   //stage outputs
   reg signed [c :0] Co [0:STG-1];
   reg signed [c :0] Si [0:STG-1];
   reg signed [31:0] s [0:STG-1];
```

```
// upper 2 bits = 2'b00 which represents 0 - π/2 range
// upper 2 bits = 2'b01 which represents π /2 to π range
// upper 2 bits = 2'b10 which represents π to 3* π /2 range (i.e. - π /2 to - π)
// upper 2 bits = 2'b11 which represents 3* π /2 to 2* π range (i.e. 0 to - π /2)
wire   [1:0] quad;
assign   quad = angle[31:30];
always @(posedge clock)
begin
// first 2 MSB bits will determine the quadrant
  case (quad)
    2'b00,
    2'b11:   // no pre-rotation needed for these quadrants
    begin
      Co[0] <= Xin;
      Si[0] <= Yin;
      s[0] <= angle;
    end
    2'b01:
    begin
      Co[0] <= -Yin;
      Si[0] <= Xin;
      s[0] <= {2'b00,angle[29:0]}; // subtract pi/2 from angle for this quadrant
    end
    2'b10:
    begin
      Co[0] <= Yin;
      Si[0] <= -Xin;
      s[0] <= {2'b11,angle[29:0]}; // add pi/2 to angle for this quadrant
    end
```

```
   endcase
  end
  genvar i;
  generate
  for (i=0; i < (STG-1); i=i+1)
  begin: XYZ
    wire  sign;
    wire signed  [c :0] X_shift, Y_shift;
    assign X_shift = Co[i] >>> i; // shifting right
    assign Y_shift = Si[i] >>> i;
    assign sign = s[i][31]; // Z_sign = 1 if Z[i] < 0
    always @(posedge clock)
    begin
     // rotation of vectors
      Co[i+1] <= sign ? Co[i] + Y_shift     :  Co[i] - Y_shift;
      Si[i+1] <= sign ? Si[i] - X_shift         :  Si[i] + X_shift;
      s[i+1] <= sign ? s[i] + tan_inverse[i] :  s[i] - tan_inverse[i];
    end
  end
   endgenerate
   // assigning output
   assign Cosine = Co[STG-1];
   assign Sine   = Si[STG-1];
endmodule
```

CHAPTER-4

VERILOG

4.1 Introduction to Verilog

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

4.1.1 Verilog Capabilities:

- Verilog is a case sensitive language.
- It is vendor independent which means a program can be executed in any simulator.
- It is human and machine readable. Thus, it can be used as an exchange language between tools and designers.
- Verilog allows different levels of abstraction to be mixed in same model.
- Thus, a designer can define a hardware model in terms of switches, gates, RTL or behavioural code using tools like synthesis tools and his netlist is used for gate level simulation and for backend.

4.1.2 Data types in Verilog

4.1.2.1 Value Set

Verilog consists of, mainly, four basic values. All Verilog data types, which are used in Verilog, store these values −

0 (logic zero, or false condition)

1 (logic one, or true condition)

x (unknown logic value)

z (high impedance state)

Use of x and z is very limited for synthesis.

4.1.2.2 Wire

A wire is used to represent a physical wire in a circuit and it is used for connection of gates or modules. The value of a wire can only be read and not assigned in a function or block. A wire cannot store value but is always driven by a continuous assignment statement or by connecting wire to output of a gate/module. Other specific types of wires are −

Wand (wired-AND) − here value of Wand is dependent on logical AND of all the device drivers connected to it.

Wor (wired-OR) − here value of a Wor is dependent on logical OR of all the device drivers connected to it.

Tri (three-state) − here all drivers connected to a tri must be z, except only one (which determines value of tri).

4.1.2.3 Register:

A reg (register) is a data object, which is holding the value from one procedural assignment to next one and are used only in different functions and procedural blocks. A reg is a simple Verilog, variable-type register and can't simply a physical register. In multi-bit registers, the data is stored in the form of unsigned numbers and sign extension is not used.

Example −

```
reg c; // single 1-bit register variable

reg [5:0] gem; // a 6-bit vector;

reg [6:0] d, e; // two 7-bit variables
```

4.1.2.4 Input, Output, Inout

These keywords are used to declare input, output and bidirectional ports of a task or module. Here input and inout ports, which are of wire type and output port is configured to be of wire, reg, wand, wor or tri type. Always, default is wire type.

4.2 Program structure in Verilog:

- Module is a basic building block in Verilog.
- It provides necessary information about input and output ports but hides the internal implementation.

Syntax:

```
module<module name>(input, output):
……….
<logic of program>
………
………
end module
```

4.2.1 Declaration of input and output:

- After declaration of module in the next step is to define the input and output ports.

  e.g.: input a, b;
- If input and output are more than one bit i.e., either two or more bits then we can define as below

  input [3:0] a, b;   //four bit input (A3-A0&B3-B0)

```
output [3:0] c;       //four bit output (C3-C0)
```

4.3 Module declaration in Verilog

4.3.1 Verilog Module

A module is a block of Verilog code that implements certain functionality. Modules can be embedded within other modules, and a higher-level module can communicate with its lower-level modules using their input and output ports.

A module should be enclosed within a module and end module keywords. The name of the module should be given right after the module keyword, and an optional list of ports may be declared as well.

Syntax:

```
module < name>([port_list]);
//contents of module
endmodule
// A module can have an empty port_list
module name;
// Contents of the module
endmodule
```

All variable declarations, functions, tasks, dataflow statements, and lower module instances must be defined within the module and endmodule keywords.

4.3.2 Levels of Abstraction

Verilog supports a design at many levels of abstraction. The major three are –

- The switch Level Modelling.
- Gate – level Modelling.
- The Data – Flow Level.
- The Behavioural or Procedural Level.

4.3.2.1 Switch level Modelling

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

```
module inverter (out, in):
supply1 vdd;
nmosx1(out, in, gnd);
pmosx2(out, in, vdd);
endmodule
```

4.3.2.2. 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.

```
module fulladder (a, b, sum, cout);
input a, b;
output sum, cout;
xor x1(a, b, y);
xor x2(a, b, y);
endmodule
```

4.3.2.3 Data-flow Modelling

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

```
module fulladder (a, b, sum, Cout);
input a, b;
output sum, cout;
assign sum = a^b;
assign cout = a^b;
endmodule
```

4.3.2.4 Behavioural modelling

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.

```
module fulladder (a, b, clk, sum);
input a, b, clk;
output sum;
always@ (posedgeclk)
 begin
sum= a+b;
endmodule
```

4.4 Implementation of Basic Programs Using Verilog:

4.4.1 Multiplexer:

```
module mux99(a ,b ,c ,d, sel, out);
input wire a,b,c,d;
input wire [1:0] sel;
output reg out;
always @ (a or b or c or d or sel)
begin
case (sel)
2'b00 : out <= a;
2'b01 : out <= b;
2'b10 : out <= c;
2'b11 : out <= d;
endcase
endmodule
```

4.4.2 Full Adder:

```
module fulladder99(a, b, cin, sum, cout);
input a, b, cin;
output  sum,cout;
wire x, y, z
half_adder  h1(.a(a), .b(b), .sum(x), .cout(y));
half_adder  h2(.a(x), .b(cin), .sum(sum), .cout(z));
or or_1(cout, z, y);
endmodule
module half_adder( a, b, sum, cout );
input a, b;
output sum,  cout;
xor xor_1 (sum, a, b);
and and_1 (cout, a, b);
endmodule
```

CHAPTER – 5

INTRODUCTION TO SOFTWARE TOOLS

5.1 Software Tools Used

5.1.1 MATLAB

MATLAB (Matrix Laboratory) is a programming platform developed by Math Works, 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 analyse data, develop algorithms, and create models and applications.

Usage of MATLAB Software In This Project

- MATLAB is used here to implement the CORDIC algorithm from the scratch since CORDIC has different modes available, we used MATLAB software to check those techniques.
- It is used to do the time comparisons between the in-built codes available in MATLAB and the CORDIC algorithm
- It is used to compare the accuracy of values between the values generated with the CORDIC algorithm and the direct functions available in MATLAB.
- It is used to calculate the CPU-time required to implement the CORDIC algorithm.

5.1.2 MODELSIM

MODELSIM is a multi-language HDL simulation environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and System C, and includes a built-in C debugger MODELSIM can be used independently, or in conjunction with Intel Quartus Prime, Xilinx ISE or XILINX VIVADO. Simulation is performed using the graphical user interface (GUI), or automatically using scripts.

ModelSim uses a unified kernel for simulation of all supported languages, and the method of debugging embedded C code is the same as VHDL or Verilog.

ModelSim and QuestaSim products enable simulation, verification and debugging for the following languages

- VHDL
- Verilog
- Verilog 2001
- System Verilog
- PSL

Usage of ModelSim in this project:

- We used ModelSim to implement the logic in Verilog.
- It is used to check the simulation results and graphs.
- Minimize the errors in the Verilog code.

5.1.3 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.

Language support

The VIVADO High-Level Synthesis compiler enables C, C++ and SystemC 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

5.2 XILINX VIVADO ISE Design Suite (16.1 Version)

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) Behavioural - 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 begin... -- 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 simulation, 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.

5.3 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.

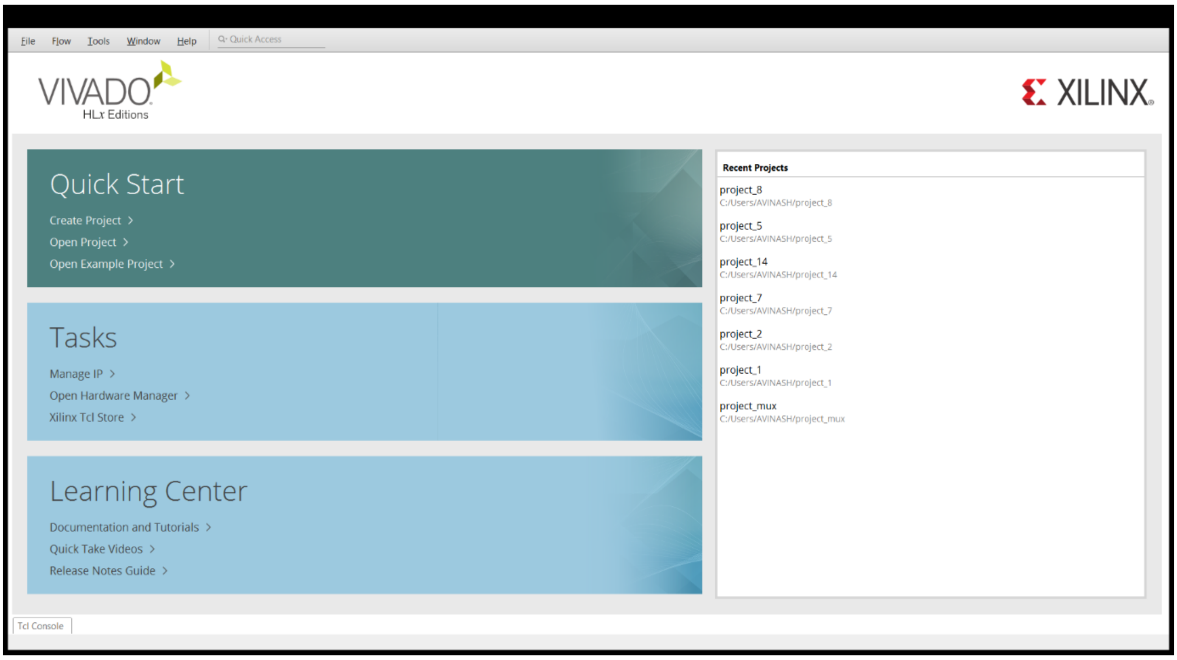


Figure 5.1 Xilinx Vivado Project Navigator window

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 Centre

In the Quick start block, we have create a new project, open project and open example project.

In the Tasks, We have Manage IP, open hardware manager, Xilinx Td store.

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 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.

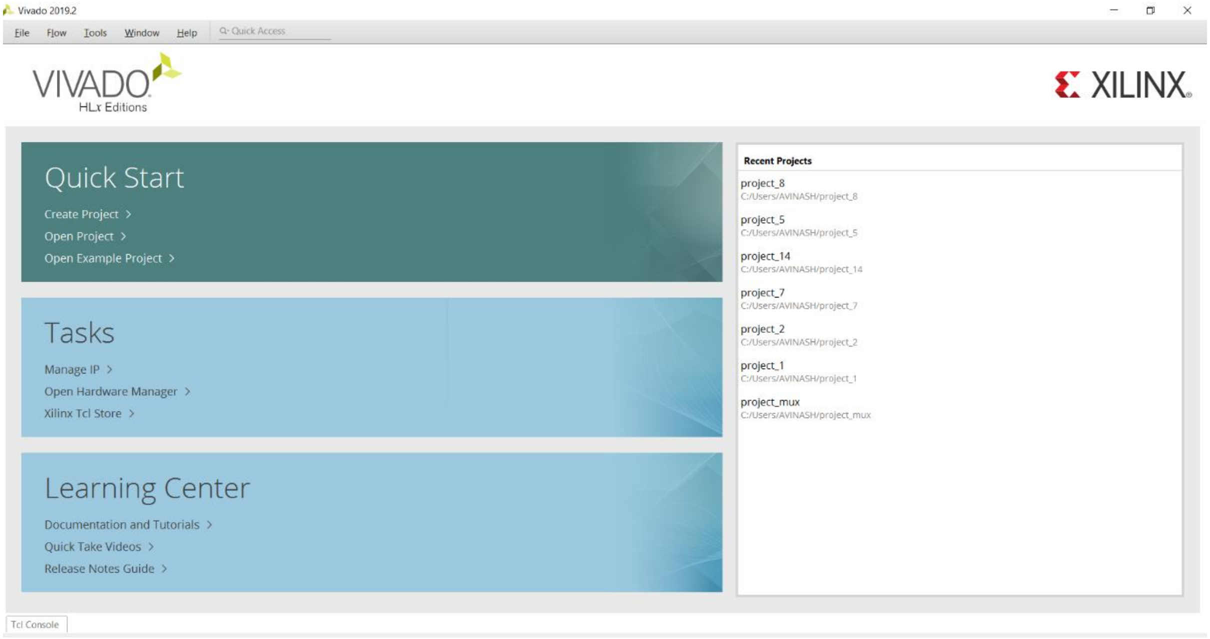
Step 4 --- Simulating a Design

This specifies the design unit you want to simulate and opens a structure tab in the workspace panel. Your specify will be used to create a working library subdirectory within the Project

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

Creating a New Project

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



project_8
C:/Users/AVINASH/project_8

project_5
C:/Users/AVINASH/project_5

project_14
C:/Users/AVINASH/project_14

project_7
C:/Users/AVINASH/project_7

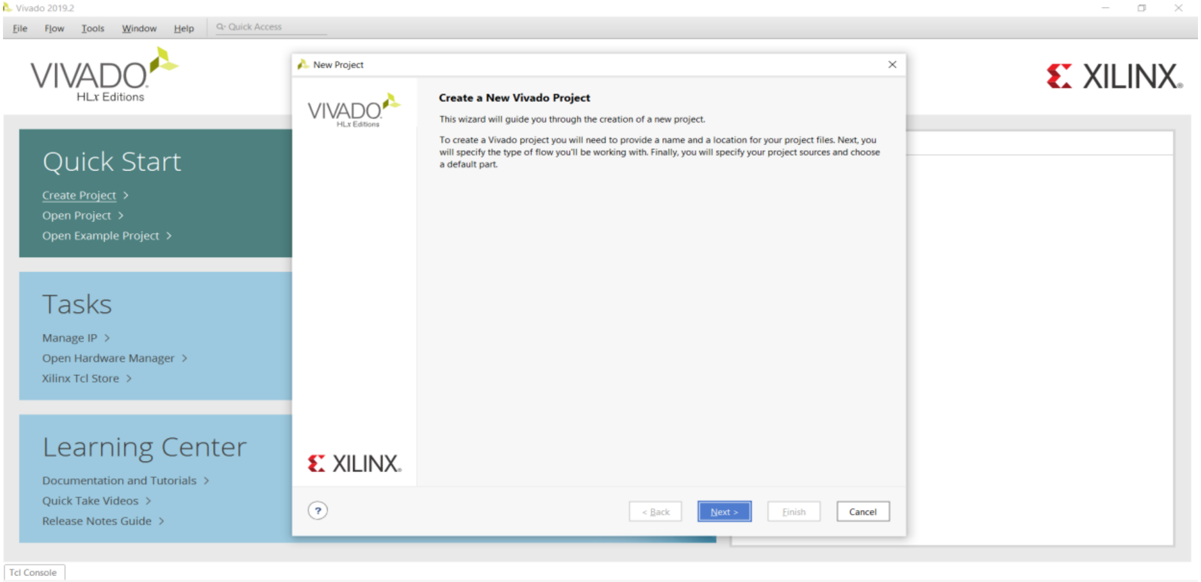
project_2
C:/Users/AVINASH/project_2

project_1
C:/Users/AVINASH/project_1

project_mux
C:/Users/AVINASH/project_mux

Figure 5.2 Creating new project window

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



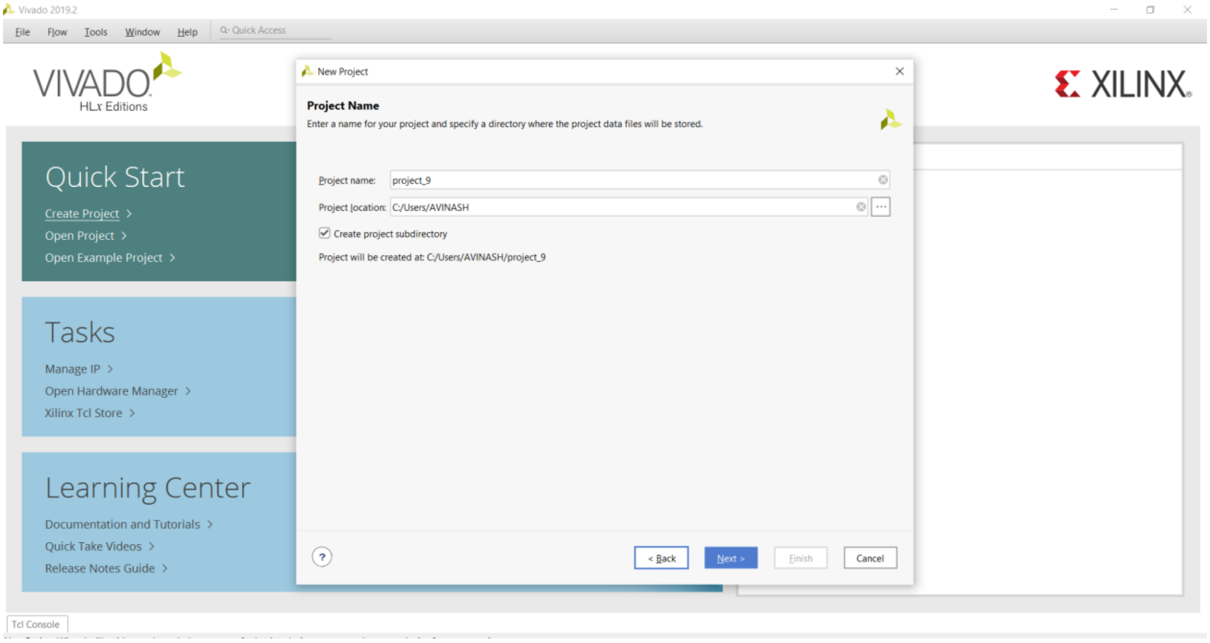
Create a New Vivado Project

This wizard will guide you through the creation of a new project.

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

Figure 5.3 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. 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.



Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: project_9

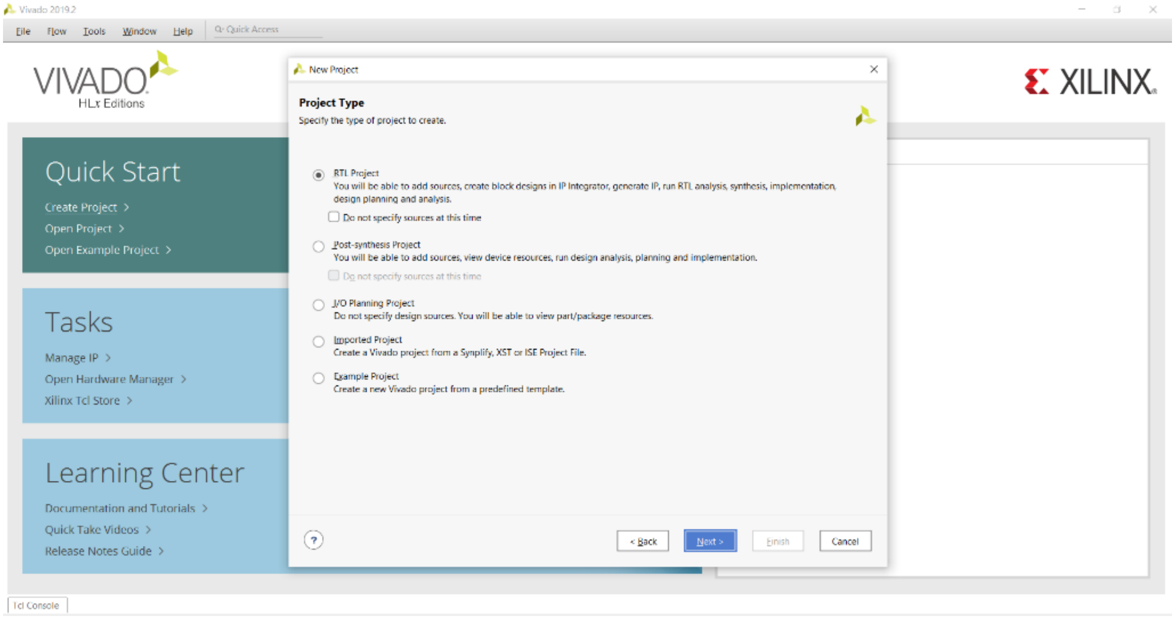
Project location: C:/Users/AVINASH

☑ Create project subdirectory

Project will be created at: C:/Users/AVINASH/project_9

Figure 5.4 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 pre-existing 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.



Project Type

Specify the type of project to create.

● RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
□ Do not specify sources at this time

○ Post-synthesis Project
You will be able to add sources, view device resources, run design analysis, planning and implementation.
□ Do not specify sources at this time

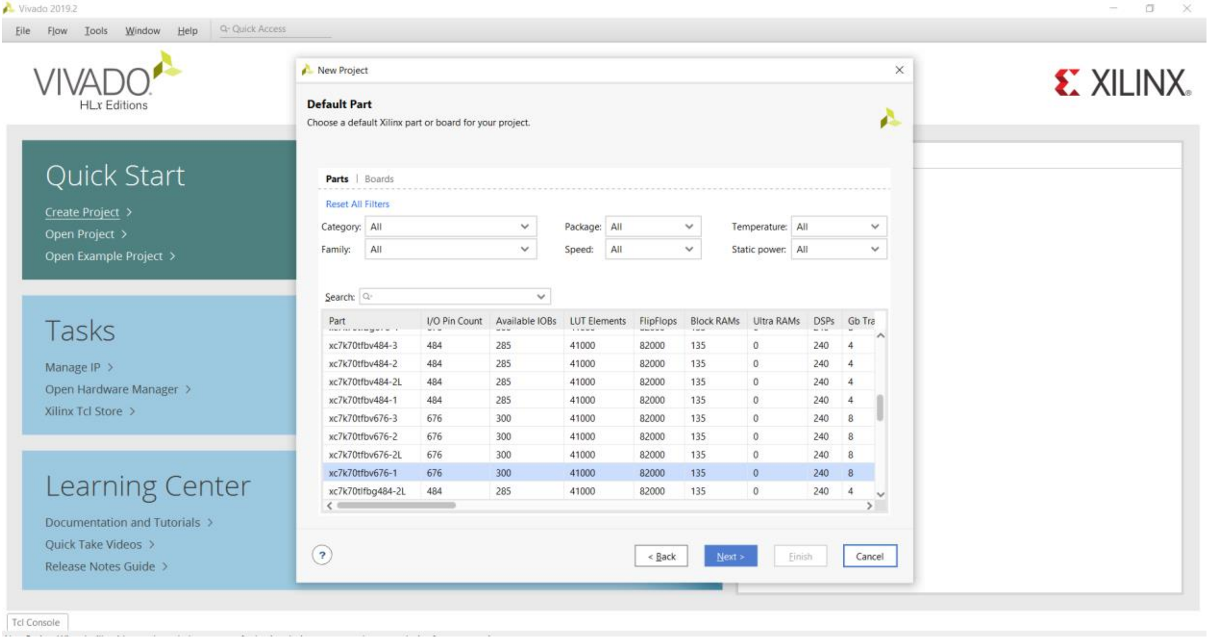
○ I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

○ Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

○ Example Project
Create a new Vivado project from a predefined template.

Figure 5.5 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.



Default Part

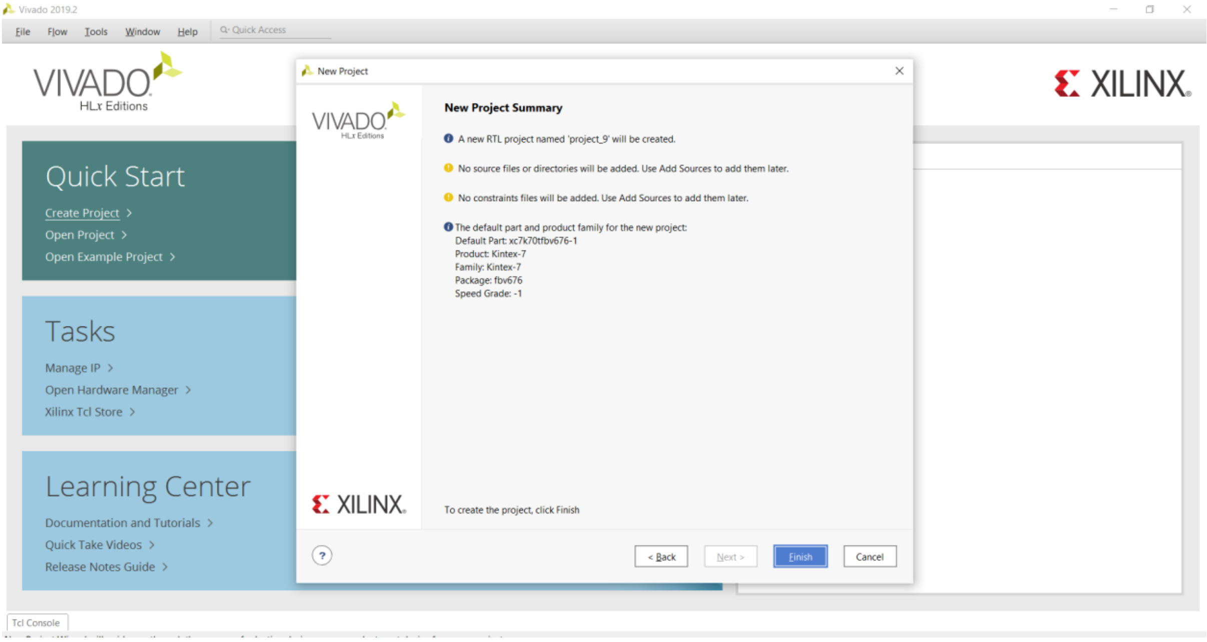
Choose a default Xilinx part or board for your project.

Category: All
Family: All
Package: All
Speed: All
Temperature: All
Static power: All
Search:

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Tra
xc7k70tfbv484-3	484	285	41000	82000	135	0	240	4
xc7k70tfbv484-2	484	285	41000	82000	135	0	240	4
xc7k70tfbv484-2L	484	285	41000	82000	135	0	240	4
xc7k70tfbv484-1	484	285	41000	82000	135	0	240	4
xc7k70tfbv676-3	676	300	41000	82000	135	0	240	8
xc7k70tfbv676-2	676	300	41000	82000	135	0	240	8
xc7k70tfbv676-2L	676	300	41000	82000	135	0	240	8
xc7k70tfbv676-1	676	300	41000	82000	135	0	240	8
xc7k70tlfbg484-2L	484	285	41000	82000	135	0	240	4

Figure 5.6 Choosing a board for project

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



New Project Summary

A new RTL project named 'project_9' will be created.

No source files or directories will be added. Use Add Sources to add them later.

No constraints files will be added. Use Add Sources to add them later.

The default part and product family for the new project:
Default Part: xc7k70tfbv676-1
Product: Kintex-7
Family: Kintex-7
Package: fbv676
Speed Grade: -1

To create the project, click Finish

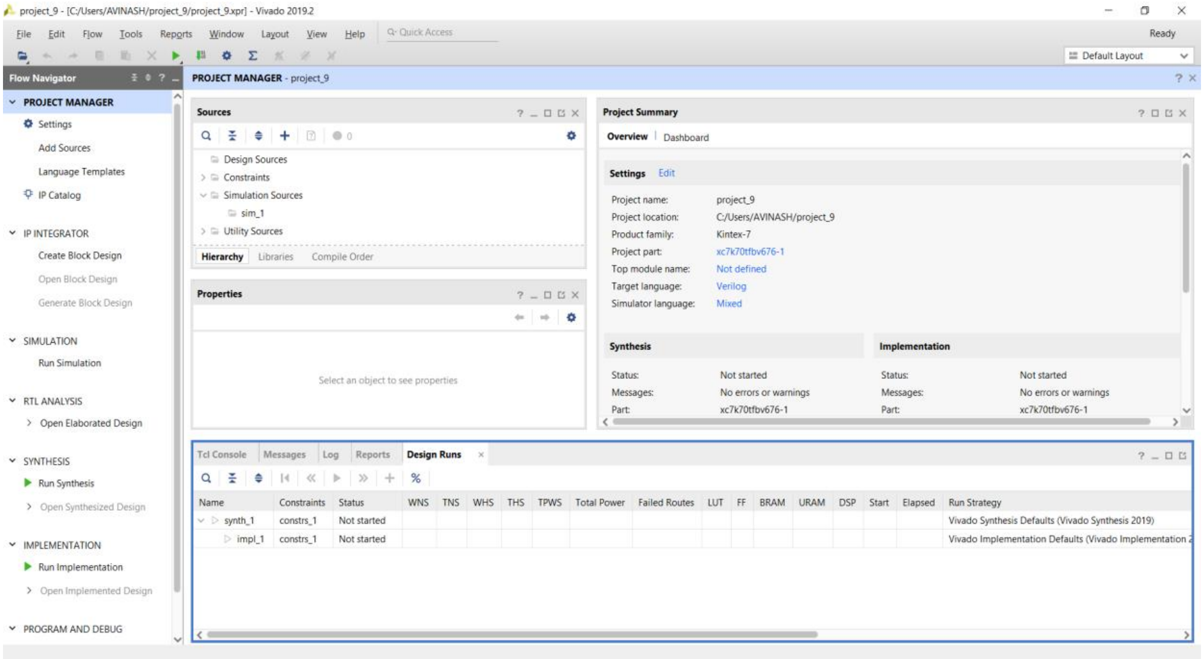
Figure 5.7 Project Summary

5.4 Steps for Design Entry:

5.4.1 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.



project_9 - [C:/Users/AVINASH/project_9/project_9.xpr] - Vivado 2019.2

PROJECT MANAGER - project_9

Settings Edit

Project name:	project_9
Project location:	C:/Users/AVINASH/project_9
Product family:	Kintex-7
Project part:	xc7k70tfbv676-1
Top module name:	Not defined
Target language:	Verilog
Simulator language:	Mixed

Synthesis		Implementation	
Status:	Not started	Status:	Not started
Messages:	No errors or warnings	Messages:	No errors or warnings
Part:	xc7k70tfbv676-1	Part:	xc7k70tfbv676-1

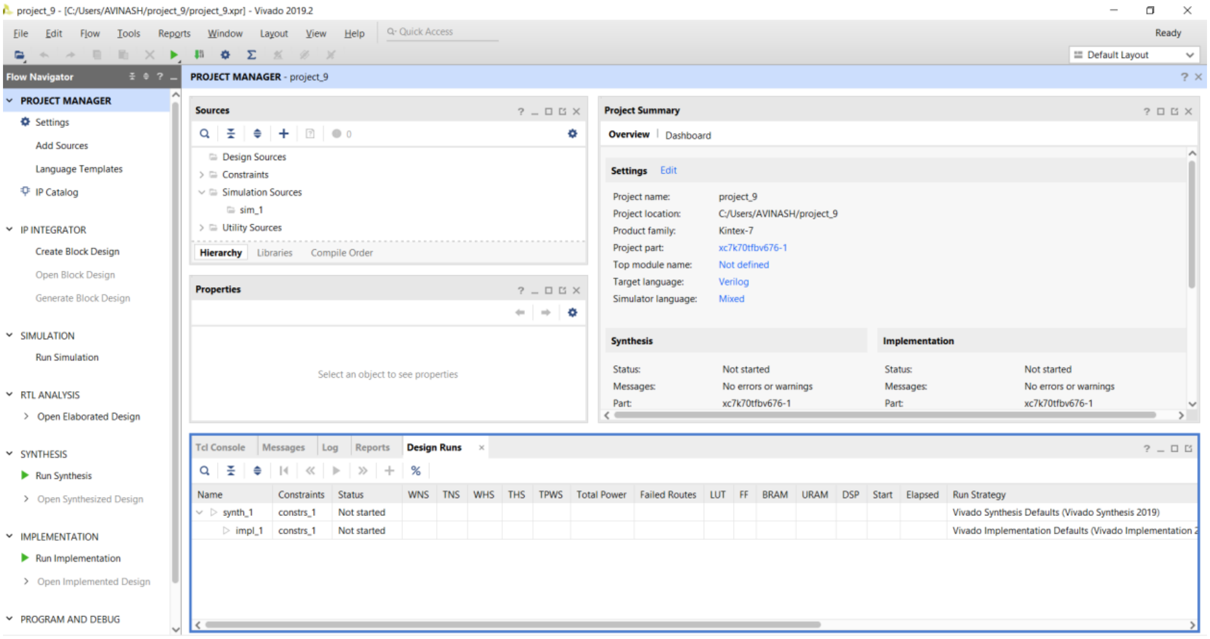
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2019)
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implementation 2

Figure 5.8 Main window for the project

5.4.2 Project Manager

5.4.2.1 Project Settings

Begin by clicking on "Project Settings" under the Project Manager phase of the Flow Navigator.



project_9 - [C:/Users/AVINASH/project_9/project_9.xpr] - Vivado 2019.2

PROJECT MANAGER - project_9

Settings Edit

Project name:	project_9
Project location:	C:/Users/AVINASH/project_9
Product family:	Kintex-7
Project part:	xc7k70tfbv676-1
Top module name:	Not defined
Target language:	Verilog
Simulator language:	Mixed

Synthesis		Implementation	
Status:	Not started	Status:	Not started
Messages:	No errors or warnings	Messages:	No errors or warnings
Part:	xc7k70tfbv676-1	Part:	xc7k70tfbv676-1

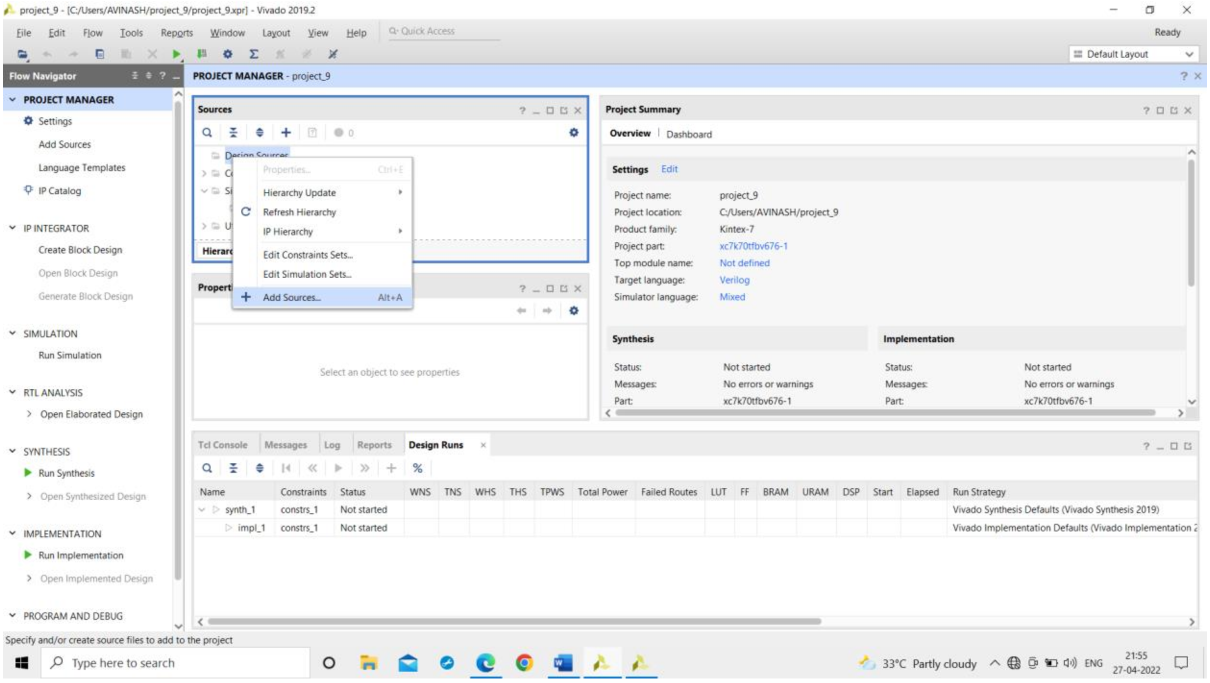
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2019)
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implementation 2

Figure 5.9 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.

5.4.2.2 Add sources

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



project_9 - [C:/Users/AVINASH/project_9/project_9.xpr] - Vivado 2019.2

Project name: project_9
Project location: C:/Users/AVINASH/project_9
Product family: Kintex-7
Project part: xc7k70tfbv676-1
Top module name: Not defined
Target language: Verilog
Simulator language: Mixed

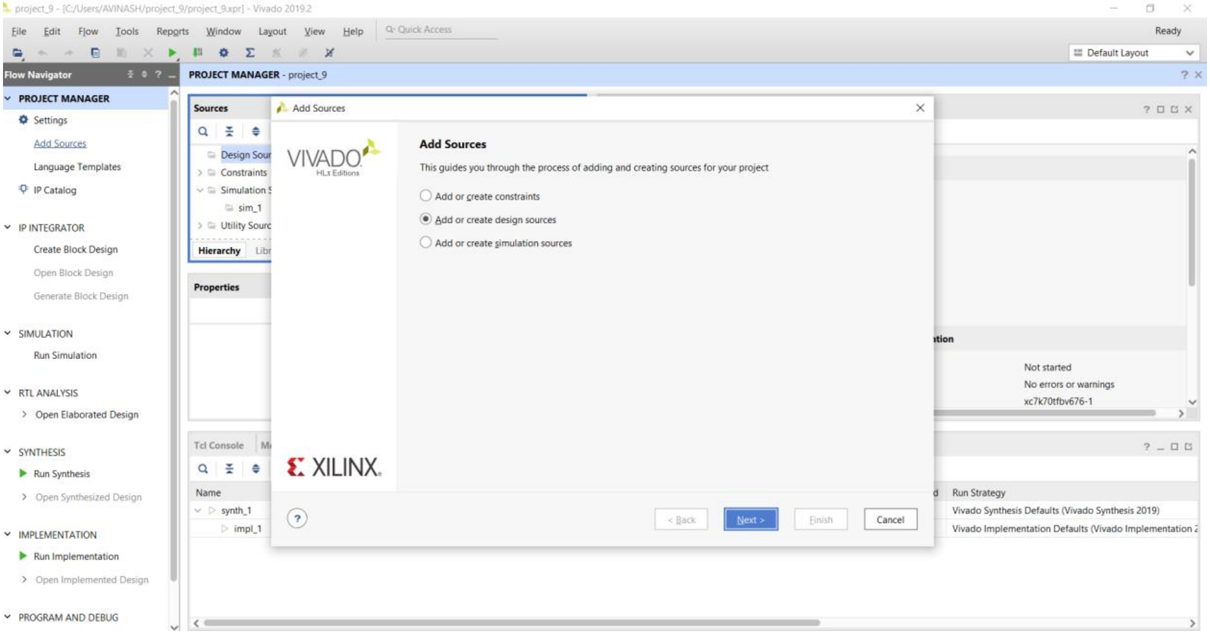
Synthesis		Implementation	
Status:	Not started	Status:	Not started
Messages:	No errors or warnings	Messages:	No errors or warnings
Part:	xc7k70tfbv676-1	Part:	xc7k70tfbv676-1

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2019)
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implementation 2

Specify and/or create source files to add to the project

Figure 5.10 Adding the source files

Select the "Add or create Design sources" radial and then click the "Next>" button.



project_9 - [C:/Users/AVINASH/project_9/project_9.xpr] - Vivado 2019.2

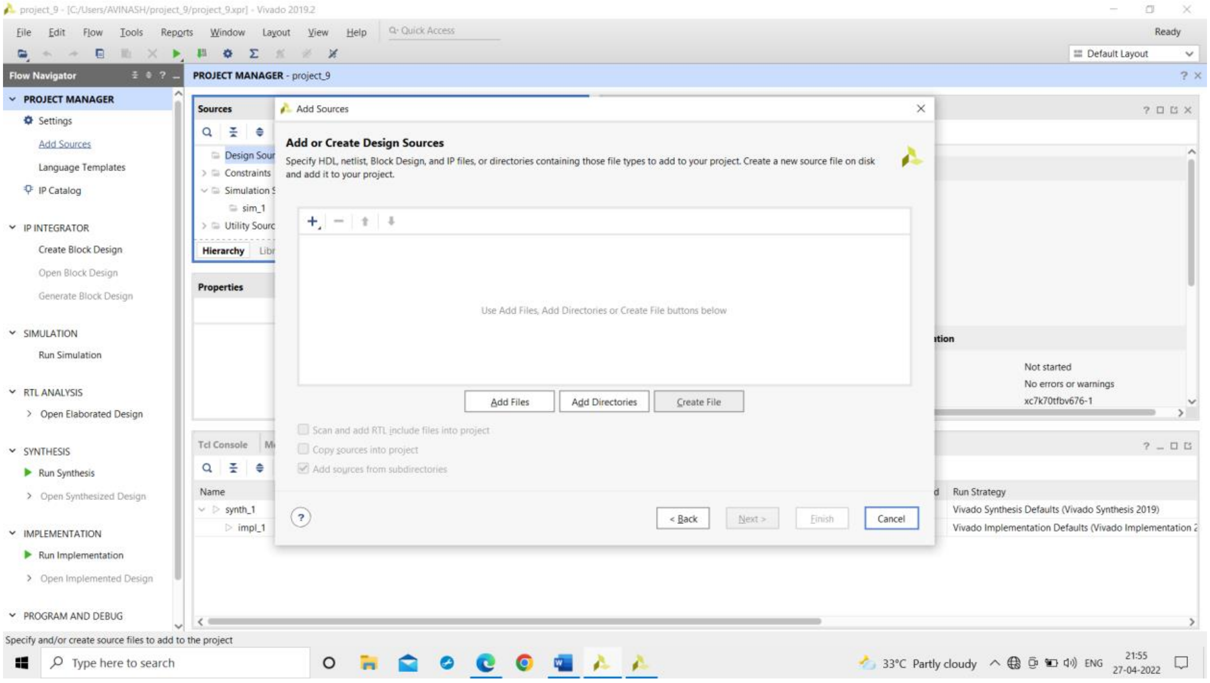
Add Sources

This guides you through the process of adding and creating sources for your project

○ Add or create constraints
● Add or create design sources
○ Add or create simulation sources

Figure 5.11 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.



Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Use Add Files, Add Directories or Create File buttons below

☐ Scan and add RTL include files into project

☐ Copy sources into project

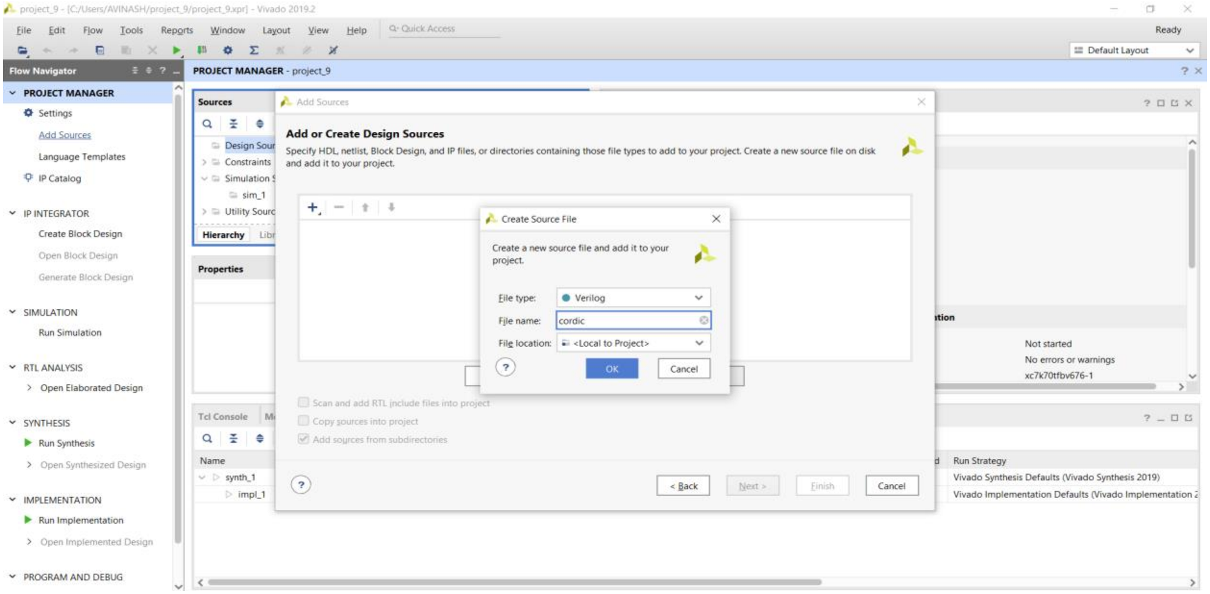
☑ Add sources from subdirectories

Specify and/or create source files to add to the project

Figure 5.12 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.



Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Create a new source file and add it to your project.

File type: Verilog

File name: cordic

File location: <Local to Project>

☐ Scan and add RTL include files into project

☐ Copy sources into project

☑ Add sources from subdirectories

Figure 5.13 Selecting the type of file and location

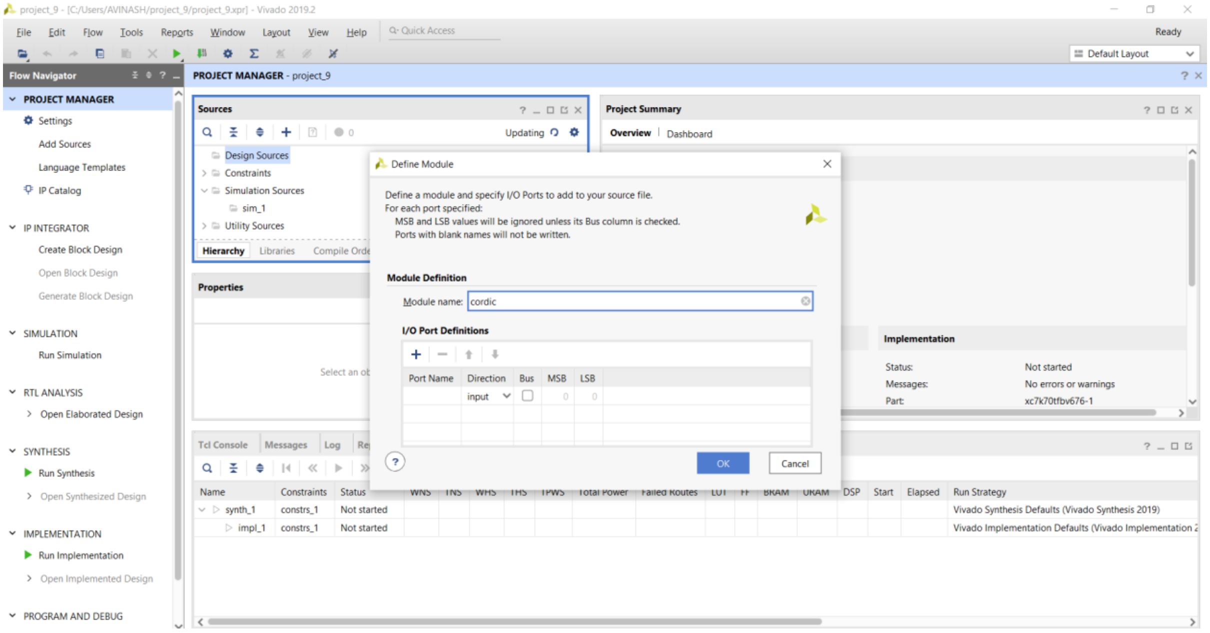
Click the “Finish” button and VIVADO will then bring up the “Define Module” window.

5.4.2.3 Define a 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.



Define Module

Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

Module name: cordic

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
	input	□	0	0

OK Cancel

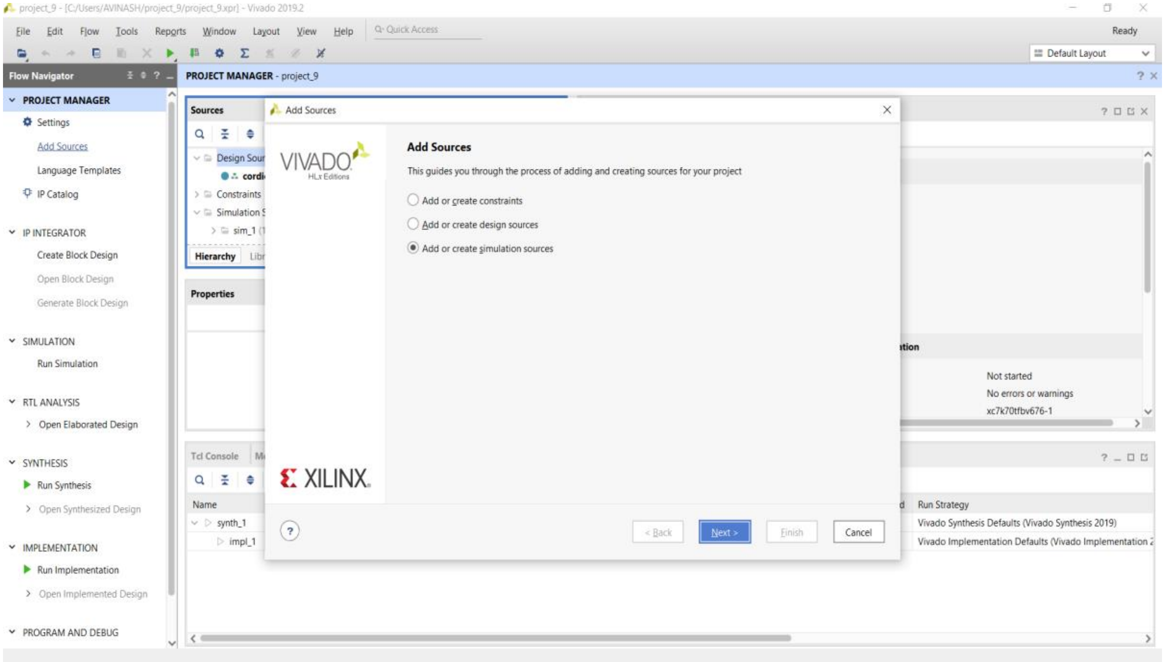
Status: Not started
Messages: No errors or warnings
Part: xc7k70tfbv676-1

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2019)
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implementation 2

Figure 5.14 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 night clicking the add source block in the panel.



Add Sources

This guides you through the process of adding and creating sources for your project

○ Add or create constraints

○ Add or create design sources

● Add or create simulation sources

Figure 5.15 Creating the simulation sources

CHAPTER 6

REPORTS AND SIMULATION RESULTS

6.1 Time Comparison

In this section we took different angles and calculated the sine and cosine of that particular angle and the time taken to execute them using inbuilt functions in MATLAB and the CORDIC algorithm and tabulated the results as shown in the figure below. From the table we can conclude that the time taken to implement the CORDIC algorithm is much less than inbuilt function available in MATLAB. Therefore, the efficiency is more in CORDIC.

Angle(degrees)	Sine	Cosine	Time (inbuilt)	Sine	Cosine	Time (CORDIC)
30	0.5	0.8660	0.012468	0.4989	0.8666	0.002841
45	0.7071	0.7071	0.012061	0.7080	0.7062	0.002365
60	0.8660	0.5	0.013507	0.8666	0.4989	0.002356
90	1	0	0.013502	1.0000	0.0012	0.002276

Table 6.1 Time Comparisons

6.2 Values comparison in MATLAB

The below figure shows the Error between the values of sine and cosine Calculated using CORDIC algorithm and the actual value.

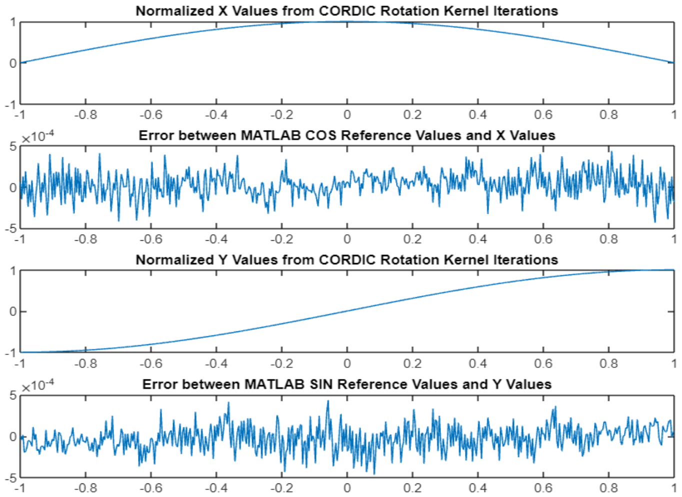


Figure 6.1 Values comparison in MATLAB

6.3 Values Comparison in Xilinx

In this section we calculated cosine and sine values in both MATLAB and XILINX and tabulated them. Then we noticed that the values are almost similar to each other. So, we can conclude that the values of sine and cosine using CORDIC algorithm are not compromised when compared to sine and cosine values using inbuilt function and through LUT approach. And we can also conclude that CORDIC algorithm implemented in both MATLAB and XILINX have similar values with less precision.

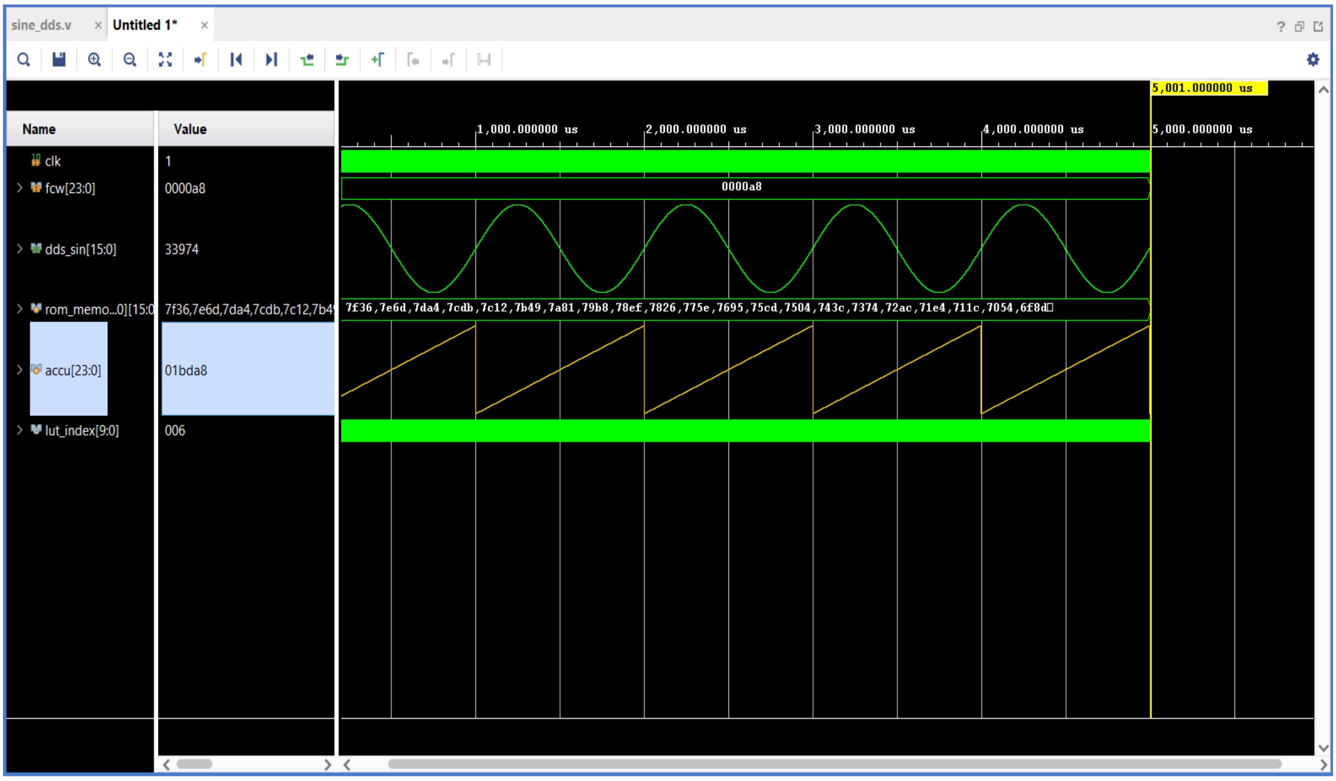
ANGLE (degrees)	Cosine (inbuilt)	Cosine (CORDIC IN MATLAB)	Cosine CORDIC in verilog)	Cosine (DDFS)	Sine (inbuilt)	Sine (CORDIC IN MATLAB)	Sine (CORDIC IN VERILOG)	Sine (DDFS)
0	1	1	0.9998	1	0	-0.0012	0	0
10	0.9848	0.9846	0.9846	0.9848	0.1736	0.1749	0.1736	0.1736
20	0.9396	0.9391	0.9394	0.9396	0.3420	0.3435	0.3420	0.3420
30	0.8660	0.8666	0.8658	0.8660	0.4999	0.4989	0.4999	0.4999
40	0.7660	0.7669	0.7658	0.7660	0.6427	0.6418	0.6426	0.6427
50	0.6427	0.6418	0.6426	0.6427	0.7660	0.7669	0.7658	0.7660
60	0.4999	0.4989	0.4999	0.4999	0.8660	0.8666	0.8658	0.8660
70	0.3420	0.3435	0.3420	0.3420	0.9396	0.9391	0.9394	0.9396
80	0.1736	0.1749	0.1736	0.1736	0.9848	0.9846	0.9846	0.9848
90	0	0	0	0	1	1	0.9999	1

Table 6.2 Sine and Cosine value comparison in Xilinx

6.4 Simulation Results

6.4.1 DDFS Architecture using LUT:

The below figure shows the sine and cosine wave form generated in XILINX using lookup table approach.



Name	Value
clk	1
fcw[23:0]	0000a8
dds_sin[15:0]	33974
rom_memo...0][15:0	7f36,7e6d,7da4,7cdb,7c12,7b4
accu[23:0]	01bda8
lut_index[9:0]	006

Figure 6.2 DDFS architecture using LUT

Calculations:

$F_0 = (f_{clk} \text{ x } fcw)/2^{24}$

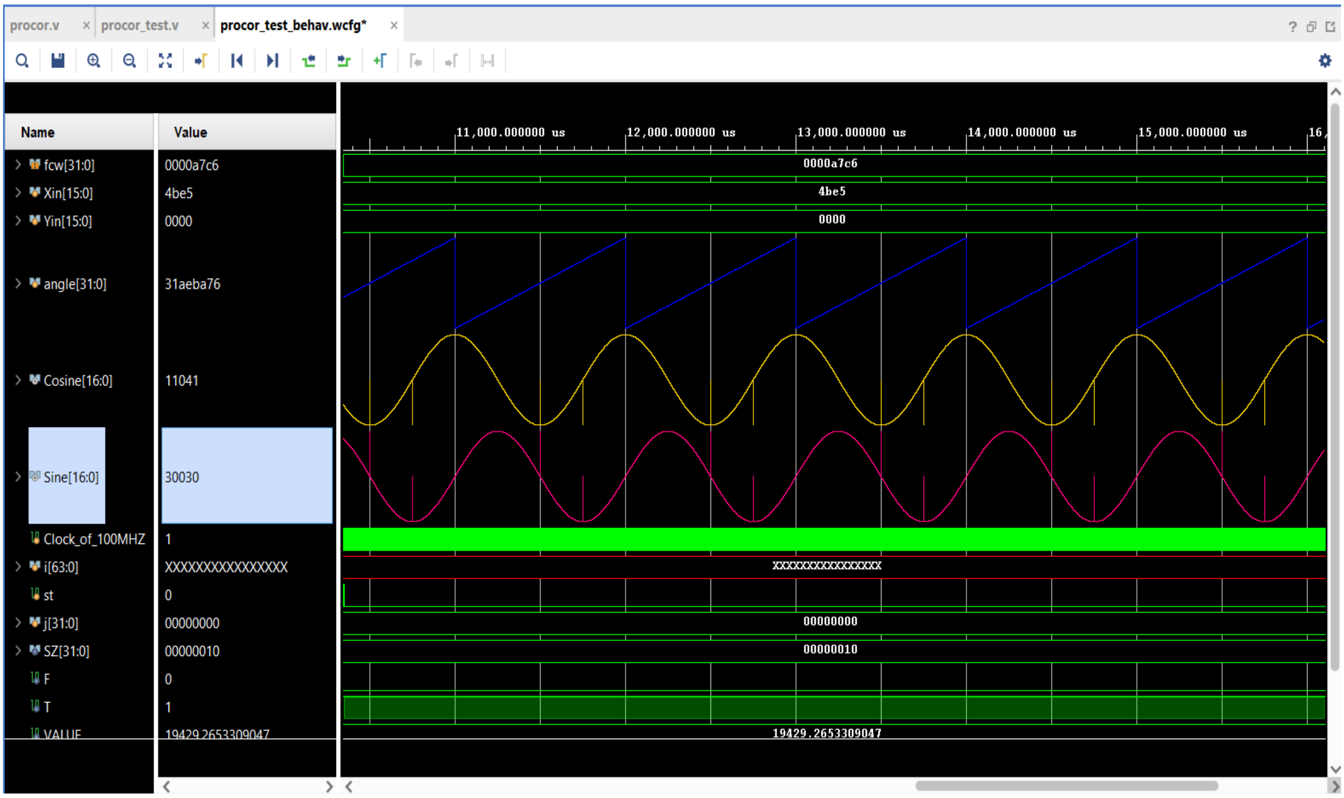
$F_0 = (100MHz \text{ x } 168)/2^{24}$

$F_0 = 1000Hz$

Therefore, the fcw = 168 is given as input to the phase accumulator to produce the sine waveform with output frequency of 1000Hz. The practical value is calculated and verified with the Theoretical value.

6.4.2 DDFS architecture using CORDIC Approach:

The below figure shows the sine and cosine wave form generated in XILINX using CORDIC approach.



Name	Value
fcw[31:0]	0000a7c6
Xin[15:0]	4be5
Yin[15:0]	0000
angle[31:0]	31aeba76
Cosine[16:0]	11041
Sine[16:0]	30030
Clock_of_100MHZ	1
i[63:0]	XXXXXXXXXXXXXXXX
st	0
j[31:0]	00000000
SZ[31:0]	00000010
F	0
T	1
VALUE	19429.2653309047

Figure 6.3 DDFS architecture using CORDIC Approach

Calculations:

$F_0 = (f_{clk} \text{ x } fcw)/2^{32}$

$F_{0} = (100\text{MHz x } 42950)/2^{32}$

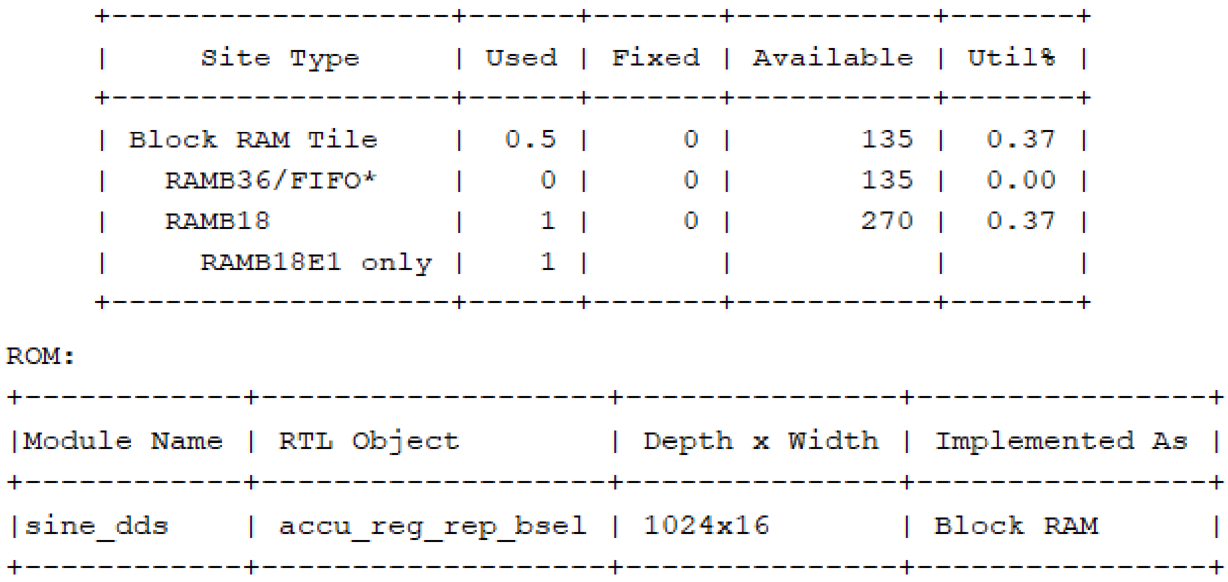
$F_0 = 1000\text{Hz}$

Therefore, the fcw = 42950 is given as input to the phase accumulator to produce the sine waveform with output frequency of 1000Hz. The practical value is calculated and verified with the Theoretical value.

6.5 Memory Utilization

6.5.1 Memory Utilization in LUT approach:

RAM and ROM are used in LUT approach, where all the 1024 sampled values of sine of 16 bit wide are stored. This is said to be the disadvantage as it occupies more memory and this has to be minimized.



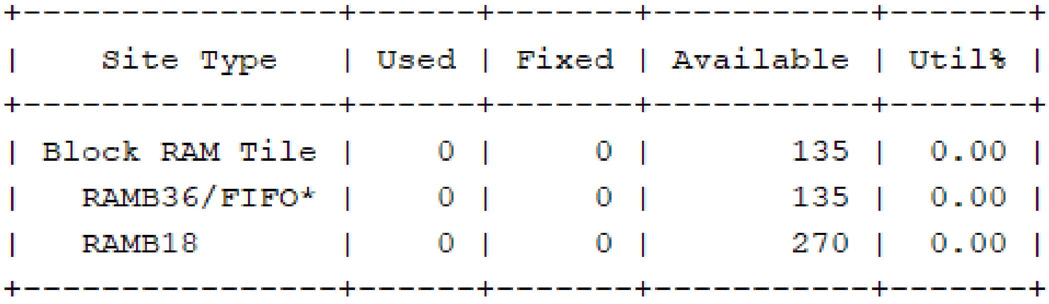
```
+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |  0.5 |     0 |       135 |  0.37 |
|   RAMB36/FIFO*    |    0 |     0 |       135 |  0.00 |
|   RAMB18          |    1 |     0 |       270 |  0.37 |
|     RAMB18E1 only |    1 |       |           |       |
+-------------------+------+-------+-----------+-------+
```

```
ROM:
+------------+-------------------+---------------+----------------+
|Module Name | RTL Object        | Depth x Width | Implemented As |
+------------+-------------------+---------------+----------------+
|sine_dds    | accu_reg_rep_bsel | 1024x16       | Block RAM      |
+------------+-------------------+---------------+----------------+
```

Table 6.3 ROM and RAM utilization in LUT approach

6.5.2 Memory Utilization in CORDIC approach:

In CORDIC approach RAM and ROM are not used which overcomes the limitation in LUT approach of occupying more memory.



```
+----------------+------+-------+-----------+-------+
|    Site Type   | Used | Fixed | Available | Util% |
+----------------+------+-------+-----------+-------+
| Block RAM Tile |    0 |     0 |       135 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |       135 |  0.00 |
|   RAMB18       |    0 |     0 |       270 |  0.00 |
+----------------+------+-------+-----------+-------+
```

Table 6.4 ROM and RAM utilization in CORDIC approach

6.6 Frequency Comparison

The below table 6.5 shows the comparison of frequency, time period and number of ROMs used in the DDFS using LUT approach and the CORDIC approach.

Frequency Control Word	**Frequency**	**Time Period**	**LUT approach**			**CORDIC approach**		
			Frequency	**Time period**	**No. of ROM's used**	**Frequency**	**Time period**	**No. of ROM's used**
0000A7C6	1KHz	1000µsec	1KHz	1000µsec	1	1KHz	1000µsec	0
00068DB9	10KHz	100µsec	10KHz	100µsec	1	10KHz	100µsec	0
0020C94C	50KHz	20µsec	50KHz	20µsec	1	50KHz	20µsec	0

Table 6.5 Frequency Comparison

After comparing the results it is observed that there is no compromise in the frequency and time period of the sinusoidal wave obtained in both LUT and CORDIC approach, but CORDIC approach overcomes the disadvantage of using a ROM in LUT approach. As it is observed that there is no ROM used in CORDIC approach.

CONCLUSION

Direct Digital frequency Synthesizer has been successfully implemented using LUT approach and CORDIC method. From the results obtained we can conclude that CORDIC method of implementation gives an identical output to the lookup table with a slightly increased distortion but gives advantage of removing the usage of ROM. Using CORDIC algorithm in implementing Direct Digital Frequency Synthesizer Architecture makes the calculation efficient and saves memory when compared to DDFS implementation using LUT approach. In this project we have implemented the CORDIC algorithm and implemented DDFS architecture and analysed the power and synthesis results. We did time comparisons to show that CORDIC saves time and is efficient. And also compared the results to show that the accuracy using CORDIC is not compromised.

REFERENCES

[1]- P. K. Meher, J. Valls, T. B. Juang, K. Sridharan and K. Maharatna, “50 years of CORDIC: algorithms, architectures, and applications,” IEEE Transactions on Circuits and Systems I, 2009, vol. 56, pp. 1893 1907.

[2] - J. Volder, “The CORDIC computing technique,” IRE Transactions on Computing, 1959, pp. 330 334.

[3] - 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.

[4] - T. Zaidi, Q. Chaudry and S. A. Khan, “An area and time efficient collapsed modified CORDIC DDFS architecture for high rate digital receivers,” inProceedings of INMIC, 2004, pp. 677 681. S. A. Khan, ‘‘A fixed point single stage CORDIC architecture’’, Technical report CASE, June 2010.

[5] - 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.

[6] - J. S. Walther, “A unified algorithm for elementary functions,” in Proceedings of AFIPS Spring Joint Computer Conference, 1971, pp. 379 385.

[7] - Sharma, S., Kulkarni, S., & Lakshminarasimhan, P. (2009). Implementation and application of CORDIC algorithm in satellite communication. In *15th National Conference on Communication, January 2009.* Guwahati, India.

[8] - Khan, S. A. (2011). Digital design of signal processing systems: a practical approach. Wiley (2011).