Department of Electronics & Communication Engineering,

Government Engineering College, Thrissur

Project Report

on

CORDIC Coprocessor - calculation of transcendental functions

Submitted by:

Ashwin Rajesh	**TCR18EC019**
Harith Manoj	**TCR18EC028**
Abhishek K	**TCR18EC002**
Akin Mary	**TCR18EC007**

Guided By:

Mr. Mohamed Salih K K
Assistant Professor
Dept. of ECE,
Govt. Engg. College, Thrissur

Department of Electronics & Communication Engineering,

Government Engineering College, Thrissur

Project Report

on

CORDIC Coprocessor - calculation of transcendental functions

Submitted by:

Ashwin Rajesh	**TCR18EC019**
Harith Manoj	**TCR18EC028**
Abhishek K	**TCR18EC002**
Akin Mary	**TCR18EC007**

Guided By:

Mr. Mohamed Salih K K
Assistant Professor
Dept. of ECE,
Govt. Engg. College, Thrissur

Department of Electronics & Communication Engineering,

Government Engineering College, Thrissur

CERTIFICATE

Certified that the project report entitled **Cordic Coprocessor - calculation of transcendental function** is a bonafide record of the work done by the team consisting of **Ashwin Rajesh-TCR18EC019, Harith Manoj-TCR18EC028, Akin Mary-TCR18EC007, Abhishek K-TCR18EC002** to the Department of Electronics & Communication Engineering, Govt. Engineering College, Thrissur, in partial fulfilment of the requirement for the award of B.Tech Degree in Electronics & Communication Engineering under A P J Abdul Kalam Technological University during the year 2021-22

Guide:	*Coordinator:*	*Head of Department:*
Mr. Mohamed Salih K K	Mr. Gopi C	Dr.A.R.Jayan
Assistant Professor	Assistant Professor	Professor & HoD
Dept. of ECE, GEC,Thrissur	Dept. of ECE, GEC Thrissur	Dept. of ECE, GEC Thrissur
Thrissur	Mrs. Subhija E N	
Date: 27 June 2022	Assistant Professor	
	Dept. of ECE, GEC Thrissur	

DECLARATION

We undersigned hereby declare that the project report, **‘Cordic Coprocessor- calculation of transcendental functions’**, submitted for partial fulfilment of the requirements for the award of degree of Bachelor of Technology of the APJ Abdul Kalam Technological University, Kerala is a bonafide work done by us under the supervision of **Mr Mohamed Salih K K**. This submission represents our ideas in our own words and where ideas or words of others have been included, we have adequately and accurately cited and referenced the original sources. We also declare that we have adhered to ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in our submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not been previously formed the basis for the award of any degree, diploma or similar title of any other University.

Thrissur

Date : 27 June 2022

Abhishek K

Akin Mary

Ashwin Rajesh

Harith Manoj

ACKNOWLEDGEMENT

We offer our sincere gratitude and appreciation to all the faculty at the Department of Electronics and Communication Engineering, Government Engineering College, Thirssur for their support and guidance throughout this work.

We are thankful to our internal supervisor Prof. Mohamed Salih K. K, Assistant professor, Department of Electronics and Communication engineering, Government engineering college, Thrissur, for his help, advice and guidance. We are thankful to the project coordinators Mr. Gopi C, Assistant Professor, Department of Electronics and Communication Engineering, Government Engineering College, Thrissur, Mrs. Subhija E N, Assistant Professor, Department of Electronics and Communication Engineering, Government Engineering College, Thrissurfor their full fledged support, advice and timely coordination.

We sincerely thank Dr. A. R. Jayan, Head Of Department, Department of Electronics and Communication Engineering, Govt.Engineering College, Thrissur, for his guidance, advice, support and suggestions. We are also indebted to all the teaching and non-teaching staff at the Department of Electronics & Communication Engineering, Govt.Engineering College, Thrissur, for their timely assistance in matters pertaining to this project work.

Finally, we would like to acknowledge my deep sense of gratitude to all our well-wishers , friends and classmates who helped me directly and indirectly to complete this work.

Abhishek K

Akin Mary

Ashwin Rajesh

Harith Manoj

ABSTRACT

Transcendental functions are functions that cannot be represented by finite polynomials. Typically, they are calculated in computers using taylor series approximation, LUTs or other approximations. This is not fast enough for many use cases. With easily accessible reconfigurable platforms, there is a demand for custom hardware accelerators with hardware-software codesign techniques to improve system efficiency and speed. The wide use of transcendental functions warrants exploration of these techniques for computing such functions.

Our aim is to develop a distributable IP (Intellectual Property) that can be used to compute transcendental functions. It uses the CORDIC algorithm, which uses adders and shifters to perform rotations. CORDIC is hardware efficient since it only requires adders and shifters to implement and can be used to generate a large range of sinusoidal and hyperbolic functions.

Our IP is designed using a memory mapped register interface as a controller of the CORDIC algorithm. It has input registers to get the inputs to the algorithm, a control/flag register to configure the algorithm and output registers to retrieve the outputs. The IP has a state machine that implements the CORDIC algorithm. It will be used a slave in a bus system (like AXI4) with a main processor.

We have implemented and tested our IP using the programmable logic on the ZC702 board from Xilinx. We have also characterised the performance of the CORDIC algorithm for circular and hyperbolic modes. This includes their overflow and error characteristics with the number of iterations

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BFM	Bus Functional Model
BRAM	Block Random Access Memory
CORDIC	COordinate Rotation DIgital Computer
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
GPL	General Public License
IDE	Integrated Development Environment
IP	Intellectual Property
LUT	Look Up Table
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SoC	System on a Chip
VIP	Verification IP

1 INTRODUCTION

Our work ultimately aims to provide cheap hardware acceleration for computation of transcendental functions. We use the CORDIC (COordinate Rotation DIgital Computer) algorithm for this.

1.1 CORDIC algorithm

CORDIC can be used to derive all trigonometric functions using vector rotations. Vector rotations can be used for polar to rectangular, and rectangular to polar conversions. CORDIC provides an iterative method of performing vector rotations by arbitrary angles using shifts and adds. The algorithm performs the general rotation transform in a hardware efficient way.

$$x' = x\,cos(\phi) - y\,sin(\phi)$$

$$y' = x\,sin(\phi) + y\,cos(\phi)$$

Eq 1.1 : General rotation transform

1.1.1 Working principle

The general rotation transform equation shown above can be rearranged into the following equation.

$$x' = cos(\phi)\,[x - y\,tan(\phi)]$$

$$y' = cos(\phi)\,[y + x\,tan(\phi)]$$

Eq 1.2 : Rearranged equation for the rotation transform

In CORDIC, we choose values of ϕ such that $tan(\phi) = \pm 2^{-i}$. Multiplication by a power of two is equivalent to a simple shift in binary logic. While multipliers are very expensive in terms of power and resources, shift operations can be very easily implemented. Arbitrary rotations are done by successive rotations of progressively smaller angles, ϕ corresponding to larger values of i. For example, for i = 1, ϕ = 45^{o}. So, eq 2.2 becomes :

$$x' = cos(45)\,[x - y\,/\,2]$$

$$y' = cos(45)\,[y + x\,/\,2]$$

Eq 1.3 : CORDIC iteration for i = 1

Similarly, for i =2 we get ϕ = 14.04^{o}. So, eq 2.2 becomes :

$$x' = cos(14.04)\,[x - y / 4]$$

$$y' = cos(14.04)\,[y + x / 4]$$

Eq 1.4 : CORDIC iteration for i = 2

We can easily get rotations by -45 or -14.04 as well by adding for x calculation and subtracting for y calculation. These operations are called “micro-rotations”. In each iteration, a micro-rotation is performed. This rotates the input by a fixed angle in either the clockwise or counterclockwise direction. This direction is decided at each step to end up with a sum of angles that approaches the required angle of rotation and the number of iterations increases. However, there is a scaling factor which also has to be considered.

1.1.2 Operational modes

The CORDIC rotator can operate in two modes, the first called rotation rotates the input vector by a specified angle. The second called vectoring rotates the input vector to the x axis while recording the angle made to do so, effectively measuring the angle made by the initial vector with the x axis.

In rotation mode, the angle accumulator, z is loaded with the angle to rotate the vector $(x,\ y)$. The rotation decision at each iteration, represented by d_i, is made to reduce z_i to zero. For rotation mode, the CORDIC equations are given by

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

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

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

$$d_i = -1\ if\ z_i < 0;\ +1\ otherwise$$

Eq 1.5 : CORDIC equations in rotation mode

The results after n iterations is :

$$x_n = A_n [x_0 cos\,(z_0) - y_0 sin\,(z_0)]$$
$$y_n = A_n [y_0 cos\,(z_0) + x_0 sin\,(z_0)]$$

$$z_n = 0$$

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

Eq 1.6 : Results of CORDIC in rotation mode

In the vectoring mode, the CORDIC algorithm rotates the vector to bring it to the x axis. The result is the magnitude and angle with the x axis of the initial vector,(x_0, y_0) . It seeks to minimise the y component at each iteration and take it to 0. :

$$x_{i+1} = K_i [x_i - y_i \cdot d_i \cdot 2^{-i}]$$

$$y_{i+1} = K_i [y_i + x_i \cdot d_i \cdot 2^{-i}]$$

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

$$d_i = +1 \, if \, y_i < 0 ; \; -1 \, otherwise$$

Eq 1.7 : CORDIC equations in vectoring mode

The result after n iterations is :

$$x_n = A_n \sqrt{x_0^2 + y_0^2}$$
$$y_n = 0$$

$$z_n = z_0 + tan^{-1}(y_0 / x_0)$$

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

Eq 1.8 : Results of CORDIC in vectoring mode

The vectoring and rotation modes both are limited to angles between -100 and 100 degrees. This is because the tangent is limited to 2^0 in the first iteration. For rotations larger than π/2additional rotations are needed.

$$x' = d \, . \, x$$
$$y' = d \, . \, y$$

$$z' = z \, if \, d = 1 ; \; z - \pi \, if \, d =- 1$$

$$d =- 1 \, if \, x < 0 ; \; + 1 \, otherwise$$

Eq 1.9 : Equations for initial rotation by 180 degrees

The close relationship between the trigonometric and hyperbolic functions suggests the same architecture can be used to compute the hyperbolic functions. While there is early mention of using the CORDIC structure for hyperbolic coordinate transforms the first description of the algorithm is that by Walther [3].

The CORDIC equations for hyperbolic rotations are derived using the same manipulations as those used to derive the rotation in the circular coordinate system. For rotation mode these are :

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

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

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

$$d_i = -1\ if\ z_i < 0;\ +1\ otherwise$$

Eq 1.10 : CORDIC equations in rotation mode of hyperbolic extension

$$x_n = A_n[x_0 cosh(z_0) + y_0 sinh(z_0)]$$
$$y_n = A_n[y_0 cosh(z_0) + x_0 sinh(z_0)]$$

$$z_n = 0$$

$$A_n = \prod_n \sqrt{1 - 2^{-2i}}$$

Eq 1.11 : CORDIC results in rotation mode of hyperbolic extension

In the hyperbolic vectoring mode, the results of the CORDIC algorithm are as follows :

$$x_n = A_n\sqrt{x_0^2 - y_0^2}$$
$$y_n = 0$$

$$z_n = z_0 + tanh^{-1}(y_0 / x_0)$$

$$A_n = \prod_n \sqrt{1 - 2^{-2i}}$$

Equation 1.12 : CORDIC results in vectoring mode of hyperbolic extension

1.1.3 Implementable functions

CORDIC can be used to directly compute the following functions :

● Circular rotation mode	: *sin()* ,	*cos()*
● Hyperbolic rotation mode	: *sinh()* ,	*cosh()*
● Circular vectoring mode	: *tan^{-1}(y/x)* ,	*x^2+y^2*

- Hyperbolic vectoring mode : *$tanh^{-1}(y/x)$,* x^2-y^2

Apart from these functions, we can also extract derived functions from these directly implementable functions. They are :

$$tan(\theta) = sin(\theta) / cos(\theta)$$

$$tanh(\theta) = sinh(\theta) / cosh(\theta)$$

$$e^{\theta} = sinh(\theta) + cosh(\theta)$$

$$ln(\theta) = 2\, tanh^{-1}(y/x);\, x = \theta + 1,\, y = \theta - 1$$

$$\sqrt{\theta} = \sqrt{x^2 - y^2};\, x = \theta + 1/4,\, y = \theta - 1/4$$

Eq 1.13 : Derivation of functions from CORDIC functions

1.1.4 Hardware Implementation

The CORDIC algorithm can be implemented in hardware in 3 ways : sequential/iterative, parallel and pipelined. Each of these has their advantages and drawbacks. One of the first tasks is to find the appropriate implementation style.

The iterative style uses registers to store the temporary value after each iteration and uses the same hardware for each iteration. This means a minimal amount of hardware is required but we can only start one CORDIC operation after the previous one completes.

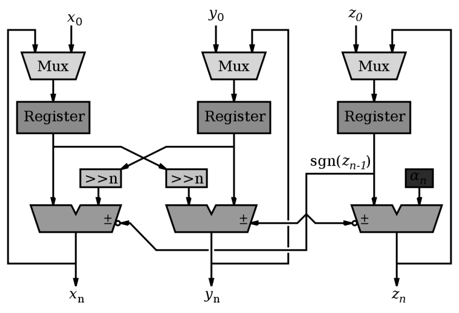


Fig 1.1 : CORDIC Iterative implementation

The pipelined style is the unrolled version of the iterative implementation. Each iteration has a separate shift and add unit and the intermediate values after each iteration are stored in a register. The pipelined style can perform one CORDIC rotation in every cycle with the same latency as iterative style. The parallel style uses a sequence of shifter and adder units with no registers in between to store intermediate values. This means all the iterations should happen in a single clock cycle. This decreases the maximum frequency but the hardware requirement is lower than pipelined because of the absence of registers for intermediate values.

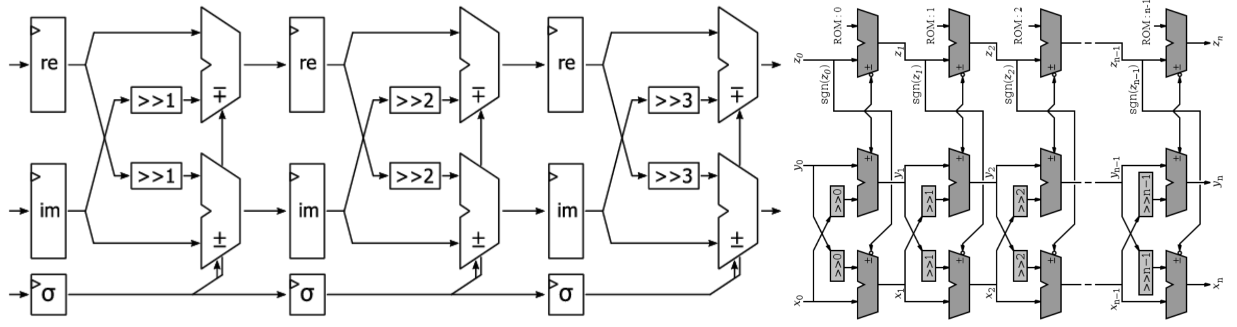


Fig 1.2 : CORDIC pipelined (left) and parallel (right) implementations

2 Literature Review

Jack E Volder et al.[1] proposed a computing technique which is suitable for solving trigonometric relationships involving plane coordinates rotation and conversion from rectangular to polar coordinates. There are basically 2 modes of operation: rotation and vectoring. In essence the basic computing use in both the rotation and vectoring modes in CORDIC is a step by step sequence of pseudo rotations which result in an overall rotation through a given angle or result in a final angular argument of zero.

Ray Andraka et al.[2] proposed implementation of functions using CORDIC architecture using FPGAs.There are a number of ways to implement a CORDIC processor. In this paper, it discusses iterative CORDIC processors.

JS Walther et al.[3]It outlines the implementation of the math ”builtin” functions using CORDIC methods and details the processes that will be taken to benchmark the resource usage, maximum frequency, and latency of each function on Xilinx 7 Series FPGAs.In FPGAs all functions must be implemented using the combination of DSP, LUT, and BRAM elements. As each of these elements are limited within a given FPGA, a general purpose implementation of the elementary functions should attempt to reduce resource usage to a minimum.Latency combined with a delay specification is the metric which determines the speed with which an implementation can produce a result. Area is defined as the number of FPGA elements used in the implementation of the elementary function. This is measured in the number of BRAM, DSP, and LUT elements.

Ramesh Bhakthavatchalu et al.[4] The paper compares the different CORDIC architectures with respect to their area, speed, and data throughput performance especially in three different major styles iterative, parallel and pipelined structures.There exist 2 modes of CORDIC algorithm vector translation mode and vector rotation mode.this paper discuss about vector rotation mode. Different implementations are compared based on power consumption ,speed/tradeoff area ,time analysis ,frequency analysis and device utilisation.

Jason Todd Arbaugh et al.[5]proposed methods to improve the efficiency of the CORDIC algorithm using lookup tables(LUT). It also mentions hybrid CORDIC processors and how rotations are done using LUTs. Different versions of the classic CORDIC algorithm have been developed to enhance the performance of calculating elementary functions. It also explains how LUTs are implemented in different CORDIC algorithms and their performance is analysed.

STMicroelectronics[6] have described how to use their CORDIC coprocessor block, STM32H7. It covers the main features of this block,which is used to accelerate trigonometric functions. The CORDIC coprocessor provides hardware acceleration of certain mathematical functions, notably trigonometric, commonly used in motor control, metering, signal processing and many other applications. It speeds up the calculation of these functions compared to a software implementation, allowing a lower operating frequency, or freeing up processor cycles in order to perform other tasks. Here it describes the bus interfaces ,controllers and number system in the STM32H7.

Accellera[7] specifies the Accellera extensions for a higher level of abstraction for modelling and verification with the Verilog Hardware Description Language.SystemVerilog is built on top of Verilog 2001. SystemVerilog improves the productivity, readability, and reusability of Verilog based code. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing tools into current hardware implementation flows. The enhancements also provide extensive support for directed and constrained-random testbench development, coverage driven verification, and assertion based verification.

Free Software Foundation, Inc[8] The GNU General Public License is a free, copyleft licence for software and other kinds of works.The licences for most software and other practical works are designed to take away your freedom to share and change the works.Developers who write software can release it under the terms of the GNU GPL. When they do, it will be free software and stay free software, no matter who changes or distributes the program. We call this copyleft: the software is copyrighted, but instead of using those rights to restrict users like proprietary software does, we use them to ensure that every user has freedom.

Li et al.[9]The WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores is a flexible design methodology for use with semiconductor IP cores. Its purpose is to foster design reuse by alleviating System-on-Chip integration problems. This is accomplished by creating a common interface between IP cores. This improves the portability and reliability of the system, and results in faster time-to-market for the end user.

Li et al.[10]The AMBA AXI protocol supports high-performance, high-frequency system designs.The AXI protocol is suitable for high-bandwidth and low-latency designs , provides high-frequency operation without using complex bridges , meets the interface requirements of a wide range of components ,is suitable for memory controllers with high initial access latency , provides flexibility in the implementation of interconnect architectures and is backward-compatible with existing AHB and APB interfaces. The paper is a complete guide of AXI protocol and its versions .

2.1 Inference

The literature survey has clearly given us the inference that CORDIC is an efficient and proven method to generate a wide range of transcendental functions and so is the best suited for our requirements. We were also able to conclude that the iterative implementation is the best suited for our application between the three alternatives, iterative pipelined and parallel. A register mapped IP with data registers for input and output and a control register was found to be the best way to implement the interface for our coprocessor. The GPL3 licence was found to be the best suited to licence our work for open access for the community. Further, the Zynq platform was chosen to test our IP with system level integration because it uses the popular AXI bus and has good support and is widely used in academia and industry.

3 Problem Statement

The problem we are trying to address is the lack of open source distributable IP for generation of transcendental functions for low power devices. This has demand in applications like power electronics, robotics, communication, etc and with the rise in reconfigurable platforms like FPGAs and SoCs with programmable logic, there is good demand for such application specific hardware acceleration, yet this has not been addressed yet.

Our work aims to produce a solution to be used by system designers for integrating transcendental function computation with low hardware and energy costs. Computation of such functions is crucial in many domains including

- Digital Signal Processing
- Power Electronics
- Robotics
- Communication
- Machine Learning

Our solution will be able to improve the performance of an SoC for such use-cases without consuming significant power or resources. This is relevant in the current market situation, where adaptive computing platforms using FPGAs are becoming popular, and custom computing platforms can be designed to solve issues with hardware acceleration modules for performance improvements for specialised use-cases.

3.1 Objective

The objective of the project is to design, develop and test a coprocessor for computing transcendental functions using the CORDIC algorithm. This involves the research and development of a CORDIC computing core and its interfacing and control logic. This core should be able to operate in both rotation and vectoring modes and be able to generate using the circular and hyperbolic extensions of CORDIC.

4 System Description

4.1 System Architecture

Our system is an Intellectual Property (IP) that can be used by designers of ASIC and FPGA based SoCs. It interfaces with the main processor using a memory mapped bus interface and an interrupt. The processor will access the IP by reading and writing to memory mapped registers using the bus interface. The IP raises the interrupt after computing the result or encountering an error.

The IP has 4 components. The interfaces between each are defined using system verilog interfaces. Splitting the IP into these submodules was done to make maintenance and customization easier. For example, adding support for another bus protocol only requires modifying the bus manager and changing the number system used for the angle only requires updating the LUTs.

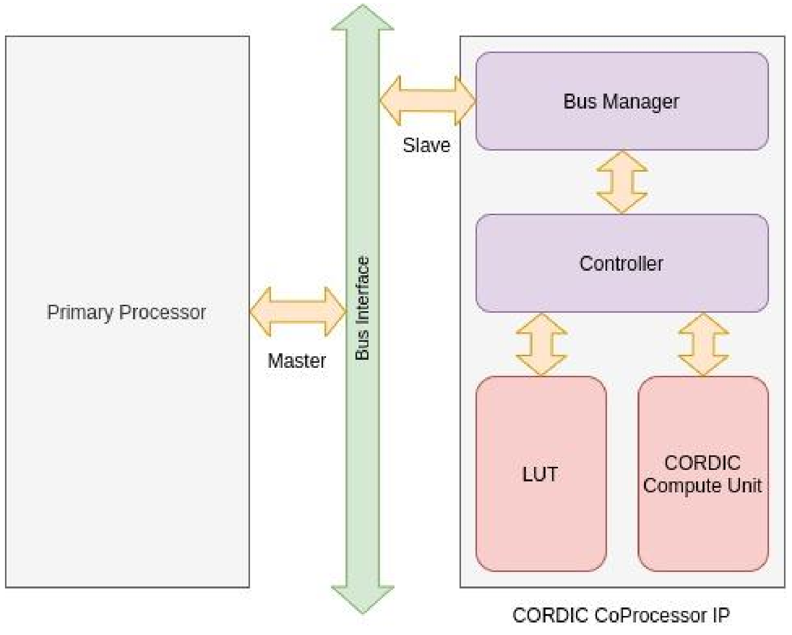


Fig 4.1 : System Block Diagram

We have tested the system with the Zynq-7000 SoC with an ARM Cortex-A9 processor as the primary processor and AXI4 as the bus standard.

4.1.1 Register format

The primary processor accesses the IP by reading and writing to 7 32 bit registers. There are 3 input registers to get the input values for the CORDIC algorithm, 3 output registers to get the output values from the algorithm and a control register that stores 16 read only flag bits and 16 read/write control bits.

Register Index	Register Name	Type	Description
0	xInput	Write only	Input x value
1	yInput	Write only	Input y value
2	zInput	Write only	Input angle value
3	xResult	Read only	Output x value
4	yResult	Read only	Output y value
5	zResult	Read only	Output angle value
6	controlReg	Read/Write	Control (start, stop, mode, system) and flag (error, overflow, ready)

Table 4.1 : Bus accessible registers and their memory mapping

The X, Y and Z inputs are in fixed point formats. X and Y can be in any 2's complement Qn.m format given n+m = 31 and both have the same format. The output X and Y have the same number system as the input. The angle is always in Q0.31 format. For circular mode, the value represents degrees/180 or radians/pi. A value of 1 represents 180 degrees or pi radians. For hyperbolic mode, the value directly represents radian. A value of 1 represents 1 radian.

The control register stores control bits to control the algorithm and flag bits to get relevant information of the last CORDIC run.

Field Index	Field Name	Type	Description
0	Start	Read/Write	Write 1 for starting the CORDIC algorithm. Automatically reset to 0
1	Stop	Read/Write	Write 1 to stop the algorithm. Automatically reset to 0
2	Rotation Mode	Write only	1 for rotation mode, 0 for vectoring mode
3	Rotation System	Write only	1 for circular system, 0 for hyperbolic system
4	Error Interrupt Enable	Write only	1 to enable output interrupt, 0 to disable all interrupts
5	Result Interrupt Enable	Write only	1 to enable interrupt on getting result or overflow

Field Index	Field Name	Type	Description
6	Overflow Stop En	Write only	1 to enable interrupt on overflow
7	Z Overflow Stop En	Write only	1 to enable interrupt on Z overflow, 0 to disable interrupt on Z overflow
12-8	Iteration count	Write only	Number of iterations to run the algorithm
13	Z Overflow Report Enable	Write only	1 to report Z overflow as an overflow error
15-14	Unused	Nil	Not used
16	Ready	Read only	1 if the controller is in idle state and is not computing anything
17	Input Error	Read only	1 if there was an invalid input (only for hyperbolic vectoring)
18	Overflow Error	Read only	1 if there was an overflow
19	X Overflow	Read only	1 if there was an overflow in the X computation
20	Y Overflow	Read only	1 if there was an overflow in the Y computation
21	Z Overflow	Read only	1 if there was an overflow in the Z computation
26-22	Iterations Elapsed	Read only	Number of iterations elapsed currently
31-27	Overflow iterations	Read only	Number of iterations at which the first overflow took place

Table 4.2 : Control Register fields

The X and Y registers use the signed Qn.m fixed point number system where n+m = 31. This could be Q0.31, Q3.28, etc. The specific number system to be used for the X and Y inputs can be decided by the user and depends on their requirements. The same number system has to be used for both X and Y and the same will be used by the output X and Y. The number system used for the angles (Z) is Q0.31. So, it can take values between -1 and 1. For circular mode, -1 corresponds to -180 degrees and 1 corresponds to 180 degrees. This was done so that overflows in the angle computation do not cause any issues and since an overflow from -1 to 1 only represents an overflow from -180 to 180 (which are equivalent), the angle value after an overflow is still valid, only that it is wrapped around. For hyperbolic mode, the 1 corresponds to 1 radian and -1 corresponds to -1 radian.

4.1.2 CORDIC Compute Unit

The Compute Unit is a purely combinational unit that does the calculations for each CORDIC iteration. It has inputs for the X and Y coordinates, the angle and LUT value. It has outputs for the output

X and Y coordinates and the angle after one iteration. It also has several control inputs for the current iteration number, rotation mode (rotation or vectoring), rotation system (hyperbolic or circular), rotation direction (clockwise or counter-clockwise), etc. It also has outputs indicating if overflow occurred in the previous computation.

Port name	Direction	Width	Description
xPrev	Input	32	Previous (input) x value
yPrev	Input	32	Previous (input) y value
zPrev	Input	32	Previous (input) angle value
rotationDir	Input	1	0 for anti-clockwise, 1 for clockwise rotation
rotationSystem	Input	1	0 for hyperbolic, 1 for circular
rotationAngle	Input	32	Angle we are rotating with (from LUT)
shiftAmount	Input	5	Shift amount in current iteration
xResult	Output	32	Output x value
yResult	Output	32	Output y value
zResult	Output	32	Output angle
xOverflow	Output	1	Did x addition lead to overflow?
yOverflow	Output	1	Did y addition lead to overflow?
zOverflow	Output	1	Did z addition lead to overflow?
xResult	Output	32	Output x value

Table 4.3 : CORDIC Compute Unit - Controller Interface

4.1.3 Look Up Table

The look up table stores the $\tan^{-1}$ and $\tanh^{-1}$ values corresponding to each iteration. It is simply a 32-bit wide ROM (Read-Only-Memory). The values are generated using a python program and are coded as a system verilog array. The input to the LUT is the iteration count and the rotation system and the output is the corresponding angle. Its interface with the controller is as follows :

Port name	Direction	Width	Description
lutAngle	Output	32	Angle value from the look up table
lutOffset	Input	5	The address to search for in the LUT
lutSystem	Input	1	1 for circular, 0 for hyperbolic

Table 4.4 : LUT - Controller Interface

4.1.4 Controller

The controller implements the state machine that performs the CORDIC iterations. It drives the compute unit and stores the intermediate results after each iteration. It is the core component of the IP and has interfaces with all the other components. The state machine has 4 states.

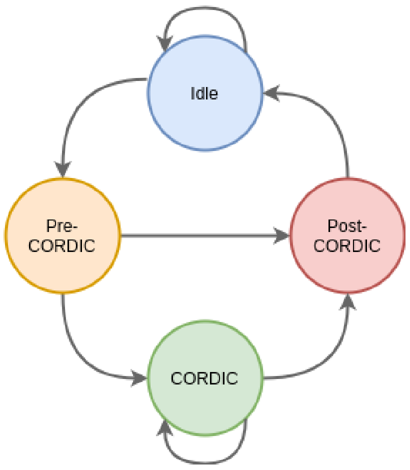


Fig 4.2 : Controller state diagram

In the idle state, the controller waits for the ready signal from the bus by writing the ready bit to the control register. In the idle state, it waits for changes for a write of 1 to the start bit of the control register. In the pre-CORDIC state, the controller checks the input and does pre-rotation if required, and checks if the input is valid. It raises an error and moves to the post-CORDIC state if the input is invalid, else it moves to the CORDIC state. The CORDIC state does the iteration for the algorithm and drives to compute unit and saves the intermediate results. If an overflow occurs, it raises the appropriate flags and moves to the post CORDIC stage. The post CORDIC stage writes the 3 output registers and flag bits of the control register. It also raises the interrupt if required.

4.1.5 Bus Manager

The bus manager acts as the interface between the bus and the controller. It stores the registers that the main processor reads from and writes to. It offers an interface to the controller to access these registers. It also handles interrupt to the processor, reset of the IP and get the clock signal from the bus. It is the interface of the IP to the rest of the system.

Separating this functionality into another module improves reusability since we only need to redesign the bus manager to add support for a different bus protocol or integrate into a different system with different requirements for interrupt, clocking or reset. The interface between the bus manager and the controller is as follows :

Port name	Direction	Width	Description
xInput	Output	32	Initial X value for CORDIC
yInput	Output	32	Initial Y value for CORDIC
zInput	Output	32	Initial angle (Z) value for CORDIC
controlRegisterInput	Output	32	Current value of the control register
xResult	Input	32	Final (Output) X value for CORDIC
YResult	Input	32	Final (Output) Y value for CORDIC
ZResult	Input	32	Final (Output) angle value for CORDIC
controlRegisterOutput	Input	32	Value to write to control register from controller
controlRegisterWriteEnable	Input	32	1 if control register should be written
rst	Output	1	1 to reset the IP (from the bus)
clk	Output	1	Clock signal from the bus
interrupt	Input	1	Make high for 1 cycle to interrupt the processor

Table 4.5 : Bus Manager - Controller Interface

We have developed and tested the bus interface for the AXI4-Lite bus and connect to the AXI4 bus on board the Zynq7000 SoC.

5 System Design

5.1 Hardware

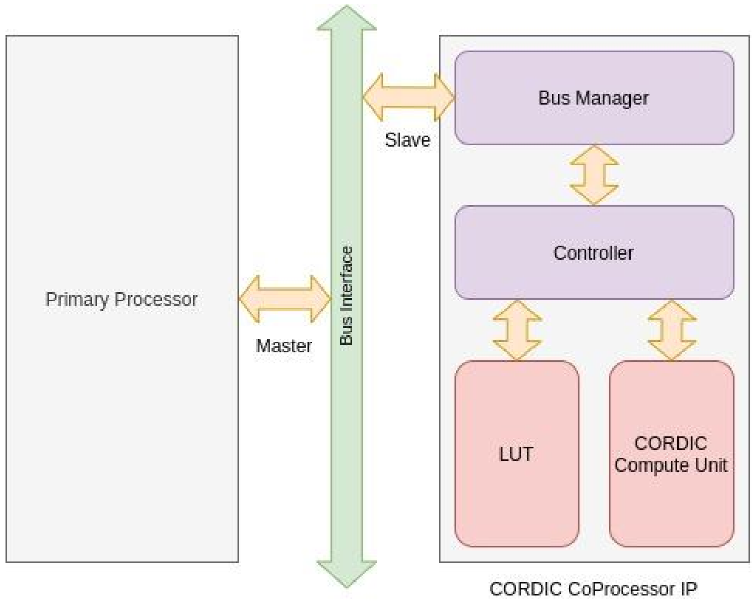


Fig 5.1 System Blocks

5.1.1 Compute Unit

The CORDIC compute unit is a combinational block created using system verilog (behavioural modelling) that executes each micro-rotation in the CORDIC algorithm. The micro-rotation is implemented as a matrix multiplication which is reduced to simple shift - add circuit by utilising only specific rotation angles.

The circuits outputs can be described by the equation:

$$x_{n+1} = x_n \pm (y_n >> i)$$
$$y_{n+1} = y_n \pm (x_n >> i)$$
$$z_{n+1} = z_n \pm angle$$

Where $angle$ is the angle corresponding to the micro-rotation, i is the iteration count (also $-\ log_2$ of the tan or tanh of the angle). The rotation direction and rotation system decide the operation to be either addition or subtraction. The circuit additionally outputs whether the addition or subtraction operation results in an overflow which is used by the controller to suspend operations and report error if necessary.

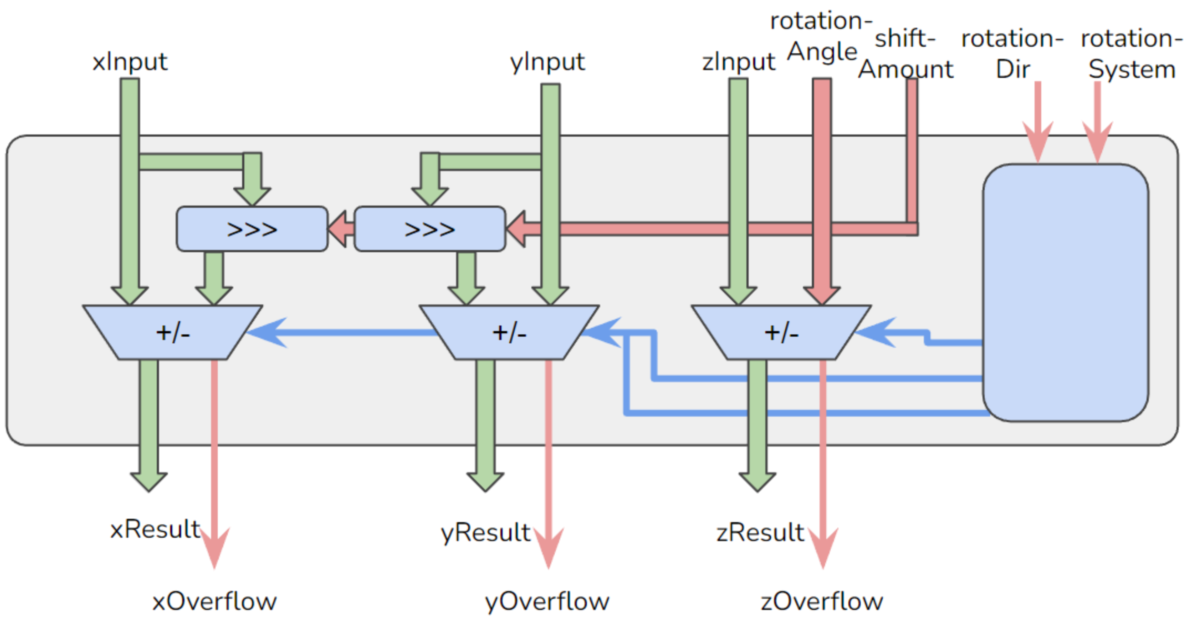


Fig 5.2 CORDIC Compute Unit Logic Diagram

5.1.2 Controller

The controller is a finite state machine comprising 4 states. It is created using system verilog (behavioural modelling) to provide necessary inputs to the compute unit, handle errors and report results to the bus controller.

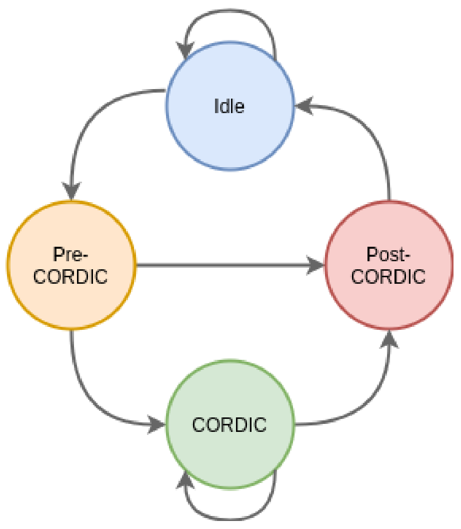


Fig 5.3 State diagram for Controller

The idle state loops without any change until the start bit in the control register is set to high. The inputs, X, Y, Z, Control register flags etc are set and the state transitions to pre-CORDIC .

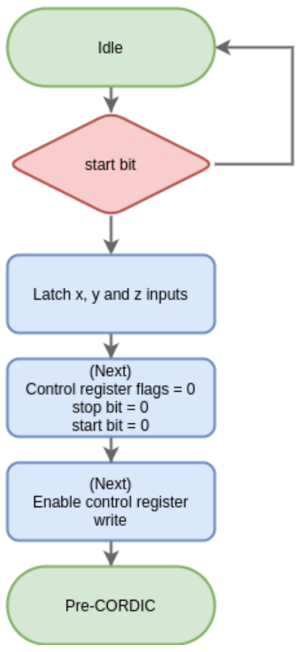


Fig. 5.4 Idle State operations

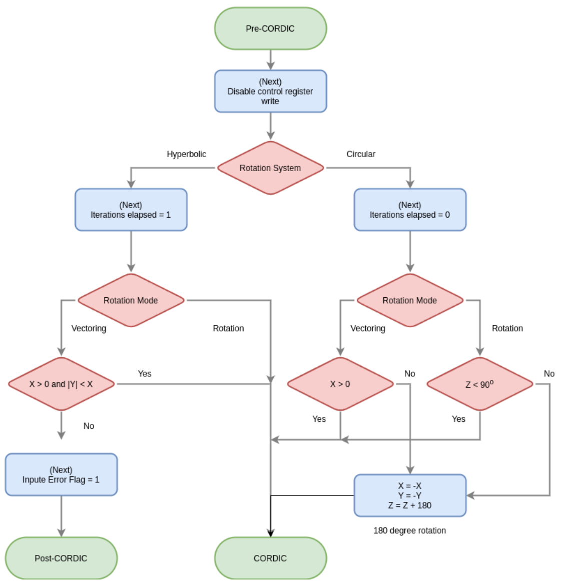


Fig. 5.5 Pre-CORDIC state operations.

Here the pre-CORDIC state is designed to catch erroneous inputs and correct them if possible. Circular rotation mode imposes certain conditions on the X, Z values; if these conditions fail, then the inputs are pre-rotated by -180. This allows the controller to extend its rotation range above limitations.

In Hyperbolic mode, the rotation limitation is automatically imposed on the input by the input number format hence removing need for correction. In vectoring mode the failing the limitation imposed on the inputs forces transition to post-CORDIC state and reporting of input-error.

The state transitions to CORDIC state after corrections if necessary.

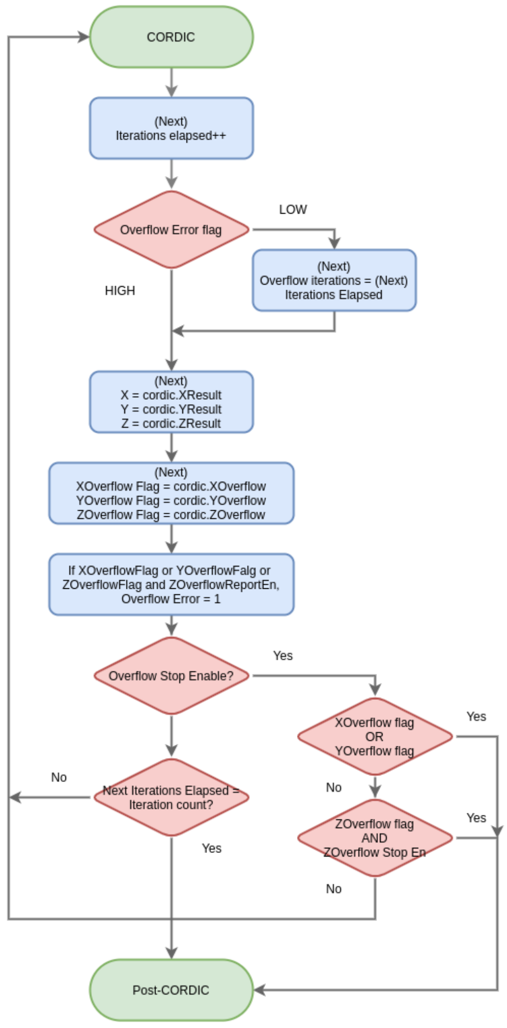


Fig 5.6 CORDIC operations.

In the CORDIC state the controller feeds, the input values X, Y, Z, the rotation system described by the control register, and rotation direction calculated to the compute unit. The CORDIC state continues to loop and perform micro-rotations until :

- Iteration elapsed exceeds iteration count specified in control register
- X or Y overflow occurs and Overflow stop Enable is set to high in the control register.
- Z overflow occurs and Overflow stop Enable, Z Overflow stop Enable is set to high in the control register.

When any of the stop conditions are met, the state transitions to post-CORDIC .

The post-CORDIC state is responsible for setting error flags, and raising interrupts if necessary.

The post-CORDIC state sets the ready flag then, depending on if any error occurred, sets the input error flag and overflow error flag.

Interrupt is raised if any of the following conditions are true:

- Result interrupt enable is set to high in control register
- Overflow error occurs and overflow stop enable is set in the control register and error interrupt enable is set.
- Input error occurs and error interrupt enable is set.

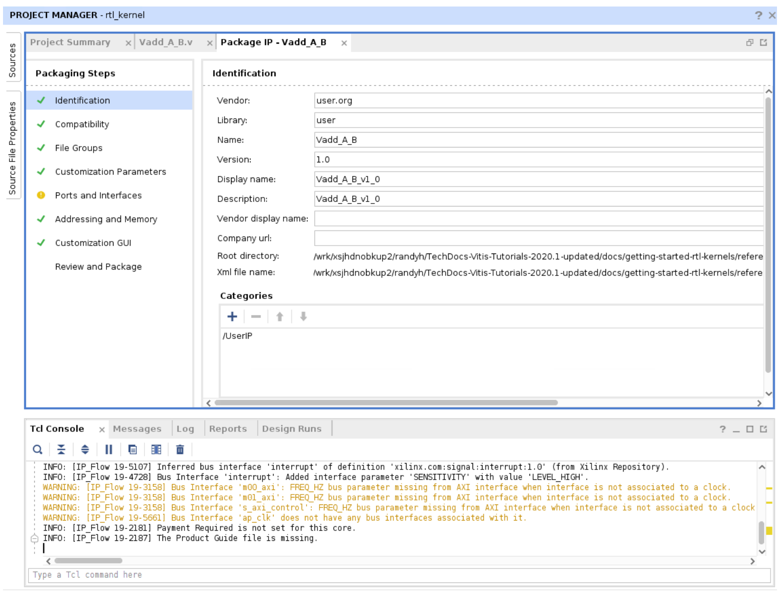
The post-CORDIC state then transitions to idle state.

5.1.3 Look Up Table

The look Up Table values were generated using a python script. The script would calculate the $\tan^{-1}$ and $\tanh^{-1}$ values corresponding to different values of i and would convert them into the required fixed point format and output a list of values in 32-bit hexadecimal. These values were then added into a system verilog array inside the LUT module which acted like a ROM.

5.1.4 Bus Manager

The Bus manager was designed using the Xilinx IP Packager tool. This tool takes in information like the number of registers, register width, etc about the IP we need to create and creates an IP with the required number of memory mapped registers. The IP is generated as verilog code. Modification of this code was required to make the automatically generated registers which were write-only into read only, write only and read/write registers as required.



Vendor:	user.org
Library:	user
Name:	Vadd_A_B
Version:	1.0
Display name:	Vadd_A_B_v1_0
Description:	Vadd_A_B_v1_0
Vendor display name:	
Company url:	
Root directory:	/wrk/xsjhdnobkup2/randyh/TechDocs-Vitis-Tutorials-2020.1-updated/docs/getting-started-rtl-kernels/refere
Xml file name:	/wrk/xsjhdnobkup2/randyh/TechDocs-Vitis-Tutorials-2020.1-updated/docs/getting-started-rtl-kernels/refere

/UserIP

```
INFO: [IP_Flow 19-5107] Inferred bus interface 'interrupt' of definition 'xilinx.com:signal:interrupt:1.0' (from Xilinx Repository).
INFO: [IP_Flow 19-4728] Bus Interface 'interrupt': Added interface parameter 'SENSITIVITY' with value 'LEVEL_HIGH'.
WARNING: [IP_Flow 19-3158] Bus Interface 'm00_axi': FREQ_HZ bus parameter missing from AXI interface when interface is not associated to a clock.
WARNING: [IP_Flow 19-3158] Bus Interface 'm01_axi': FREQ_HZ bus parameter missing from AXI interface when interface is not associated to a clock.
WARNING: [IP_Flow 19-3158] Bus Interface 's_axi_control': FREQ_HZ bus parameter missing from AXI interface when interface is not associated to a clock
WARNING: [IP_Flow 19-5661] Bus Interface 'ap_clk' does not have any bus interfaces associated with it.
INFO: [IP_Flow 19-2181] Payment Required is not set for this core.
INFO: [IP_Flow 19-2187] The Product Guide file is missing.
```

Type a Tcl command here

Fig 5.7 : Screenshot of Xilinx IP Packager

5.2 Software and simulation

We have developed the RTL code in system verilog and have exhaustively verified and characterised the submodules individually and have also verified the IP as a whole. This ensures that the users of our hardware will not encounter any errors after using our IP and the behaviour of our IP is well defined. Verification is very important in hardware projects because it is very difficult and in the case of ASICs (Application Specific Integrated Circuits), impossible to fix hardware bugs in deployed devices.

5.2.1 Tools Used

- System verilog for development of RTL code and testbenches
- Jupyter notebook for visualisation and analysis of data
- numpy python library for processing logfile data
- matplotlib python library for plotting the data
- Vivado simulator for behavioural simulation
- AXI VIP (Verification IP) from Xilinx for IP verification

5.2.2 Verification

Behavioural verification was done for each submodule individually using testbenches and then for the whole IP with the bus and memory mapped interface. Constrained random verification methodology was used, where random inputs are generated and the outputs are checked against outputs from a reference model.

First, the CORDIC compute unit was verified. We used a testbench that generates random input values and drives the input interface using a system class called a “sequencer” that acts like the controller and checks if the output matches that from a reference model. A log file is created that stores the inputs, outputs and errors of each test case from the ideal value using trigonometric function implementations built into system verilog. The log file is then analysed using a jupyter notebook that runs python.

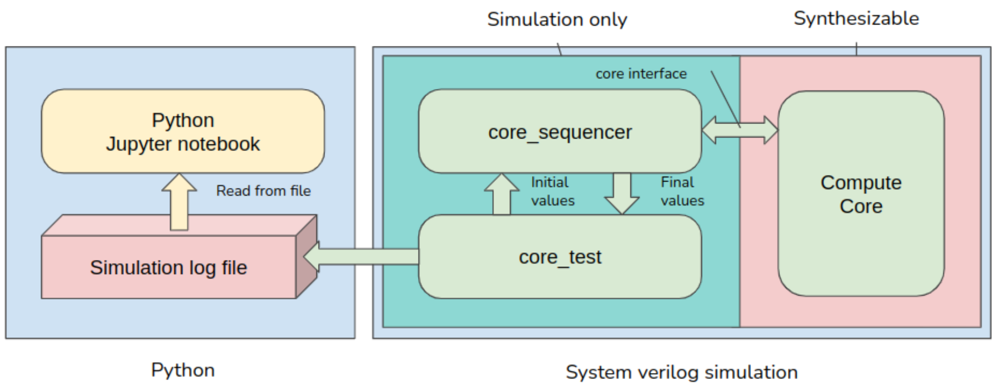


Fig 5.8 Block diagram of CORDIC compute unit verification environment

The controller was also verified in a similar manner. The controller and compute unit were connected together and the controller was driven by a testbench. The testbench also provided the functionality of the LUT. This testbench generates random inputs and logs the outputs and errors from expected output values from a reference model. The log file generated has the same format as that from the one used in the compute unit so the jupyter notebook used is identical.

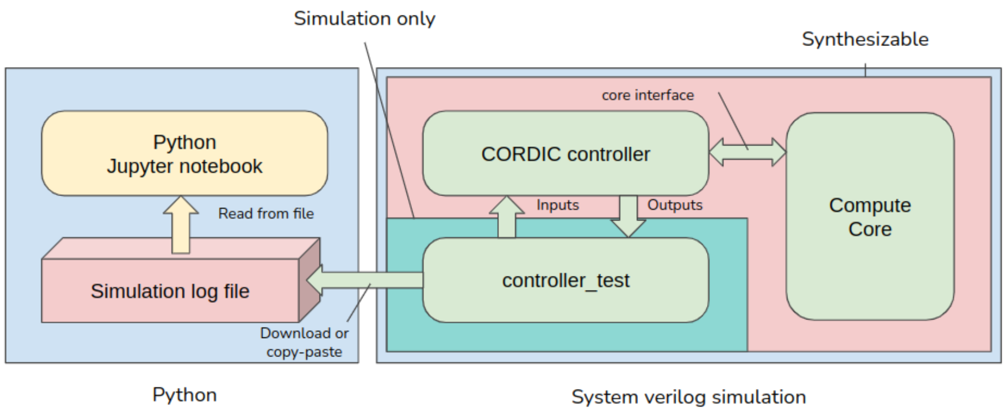


Fig 5.9 : Block diagram of CORDIC controller and LUT verification environment

The next step is to add the bus manager and verify the packaged IP. as a whole We have used the AXI VIP for this. The VIP (Verification IP) was developed by Xilinx and presents a BFM (Bus Functional Model) with system verilog functions to read and write to addresses via an AXI interface. The VIP and IP are connected, with the VIP as the master in a vivado block design. The testbench generates random inputs, sends them using the VIP to the IP and then checks the output with the result from another reference model.

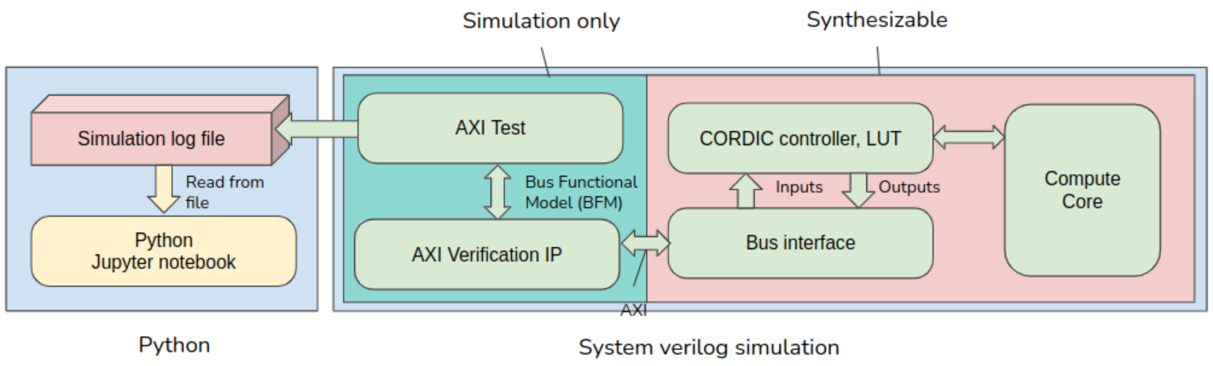
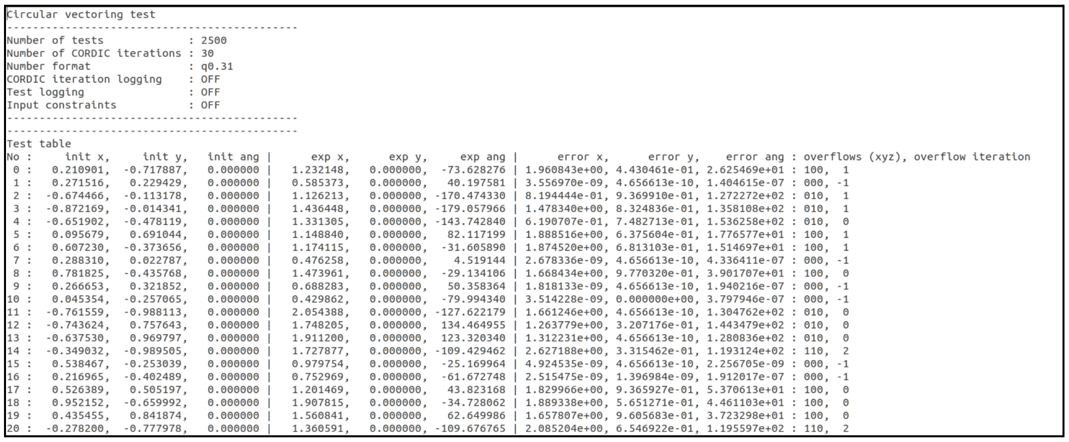


Fig 5.10 : Block diagram of IP verification using AXI VIP

The logfile for all 3 of the above tests is standardised and looks like below. It has one row for each test case and has input x, y and z, expected x, y and z and the error. It also has the information on overflow, if it occurred on x, y or z computation and the iteration count when overflow happened.

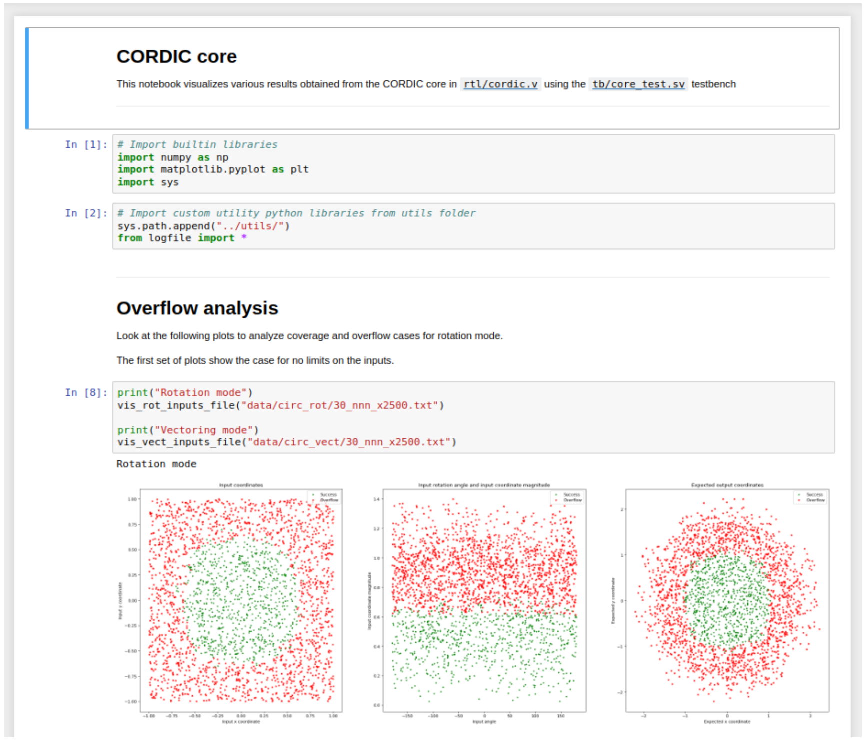


```
Circular vectoring test
---------------------------------------------
Number of tests              : 2500
Number of CORDIC iterations : 30
Number format                : q0.31
CORDIC iteration logging     : OFF
Test logging                 : OFF
Input constraints            : OFF
---------------------------------------------
---------------------------------------------
Test table
No :     init x,     init y,   init ang |      exp x,      exp y,     exp ang |      error x,      error y,    error ang : overflows (xyz), overflow iteration
 0 :   0.210901,  -0.717887,   0.000000 |   1.232148,   0.000000,  -73.628276 | 1.960843e+00, 4.430461e-01, 2.625469e+01 : 100,  1
 1 :   0.271516,   0.229429,   0.000000 |   0.585373,   0.000000,   40.197581 | 3.556970e-09, 4.656613e-10, 1.404615e-07 : 000, -1
 2 :  -0.674466,  -0.113178,   0.000000 |   1.126213,   0.000000, -170.474330 | 8.194444e-01, 9.369910e-01, 1.272272e+02 : 010,  1
 3 :  -0.872169,  -0.014341,   0.000000 |   1.436448,   0.000000, -179.057966 | 1.478340e+00, 8.324836e-01, 1.358108e+02 : 010,  1
 4 :  -0.651902,  -0.478119,   0.000000 |   1.331305,   0.000000, -143.742840 | 6.190707e-01, 7.482713e-01, 1.536258e+02 : 010,  0
 5 :   0.095679,   0.691044,   0.000000 |   1.148840,   0.000000,   82.117199 | 1.888516e+00, 6.375604e-01, 1.776577e+01 : 100,  1
 6 :   0.607230,  -0.373656,   0.000000 |   1.174115,   0.000000,  -31.605890 | 1.874520e+00, 6.813103e-01, 1.514697e+01 : 100,  1
 7 :   0.288310,   0.022787,   0.000000 |   0.476258,   0.000000,    4.519144 | 2.678336e-09, 4.656613e-10, 4.336411e-07 : 000, -1
 8 :   0.781825,  -0.435768,   0.000000 |   1.473961,   0.000000,  -29.134106 | 1.668434e+00, 9.770320e-01, 3.901707e+01 : 100,  0
 9 :   0.266653,   0.321852,   0.000000 |   0.688283,   0.000000,   50.358364 | 1.818133e-09, 4.656613e-10, 1.940216e-07 : 000, -1
10 :   0.045354,  -0.257065,   0.000000 |   0.429862,   0.000000,  -79.994340 | 3.514228e-09, 0.000000e+00, 3.797946e-07 : 000, -1
11 :  -0.761559,  -0.988113,   0.000000 |   2.054388,   0.000000, -127.622179 | 1.661246e+00, 4.656613e-10, 1.304762e+02 : 010,  0
12 :  -0.743624,   0.757643,   0.000000 |   1.748205,   0.000000,  134.464955 | 1.263779e+00, 3.207176e-01, 1.443479e+02 : 010,  0
13 :  -0.637530,   0.969797,   0.000000 |   1.911200,   0.000000,  123.320340 | 1.312231e+00, 4.656613e-10, 1.280836e+02 : 010,  0
14 :  -0.349032,  -0.989505,   0.000000 |   1.727877,   0.000000, -109.429462 | 2.627188e+00, 3.315462e-01, 1.193124e+02 : 110,  2
15 :   0.538467,  -0.253039,   0.000000 |   0.979754,   0.000000,  -25.169964 | 4.924535e-09, 4.656613e-10, 2.256705e-09 : 000, -1
16 :   0.216965,  -0.402489,   0.000000 |   0.752969,   0.000000,  -61.672748 | 2.515475e-09, 1.396984e-09, 1.912017e-07 : 000, -1
17 :   0.526389,   0.505197,   0.000000 |   1.201469,   0.000000,   43.823168 | 1.829966e+00, 9.365927e-01, 5.370613e+01 : 100,  0
18 :   0.952152,  -0.659992,   0.000000 |   1.907815,   0.000000,  -34.728062 | 1.889338e+00, 5.651271e-01, 4.461103e+01 : 100,  0
19 :   0.435455,   0.841874,   0.000000 |   1.560841,   0.000000,   62.649986 | 1.657807e+00, 9.605683e-01, 3.723298e+01 : 100,  0
20 :  -0.278200,  -0.777978,   0.000000 |   1.360591,   0.000000, -109.676765 | 2.085204e+00, 6.546922e-01, 1.195597e+02 : 110,  2
```

Fig 5.11 : Screenshot of logfile

5.2.3 Characterization

In characterization, we check the error and overflow characteristics of hardware from the log files. We do this using jupyter notebooks that run python code. The plots in the results section were taken from the jupyter notebook. The numpy library was used for handling the data and the matplotlib library was used to create the plots.



CORDIC core

This notebook visualizes various results obtained from the CORDIC core in `rtl/cordic.v` using the `tb/core_test.sv` testbench

```
In [1]: # Import builtin libraries
        import numpy as np
        import matplotlib.pyplot as plt
        import sys
```

```
In [2]: # Import custom utility python libraries from utils folder
        sys.path.append("../utils/")
        from logfile import *
```

Overflow analysis

Look at the following plots to analyze coverage and overflow cases for rotation mode.

The first set of plots show the case for no limits on the inputs.

```
In [8]: print("Rotation mode")
        vis_rot_inputs_file("data/circ_rot/30_nnn_x2500.txt")

        print("Vectoring mode")
        vis_vect_inputs_file("data/circ_vect/30_nnn_x2500.txt")
```

```
Rotation mode
```

Fig 5.12 : Screenshot of jupyter notebook used for characterization

The results of characterization are useful for the users to understand the behavior of the algorithm. Various patterns were identified relating to errors and overflow.

5.3 System integration

After developing and verifying the IP in simulation, we also need to test it as part of a complete system. We chose the Zynq-7000 based ZC702 evaluation board for this purpose. We had to add the IP and other components like the AXI interconnect and reset controller and synthesise the design.

5.3.1 Tools and hardware used

- ZC702 evaluation board with the Zynq-7000pSoC
- Vivado IDE for synthesis and bitstream generation
- Vitis IDE for programming the Zynq-7000 SoC

5.3.2 Block design

The design was done in vivado’s block design editor. There was no need to write or modify any RTL code at this stage. Our IP was added from its packaged location and we also added the processing system, the AXI interconnect and the reset controller. The block design looks as shown below. The connection automation tool connected all the required ports automatically.

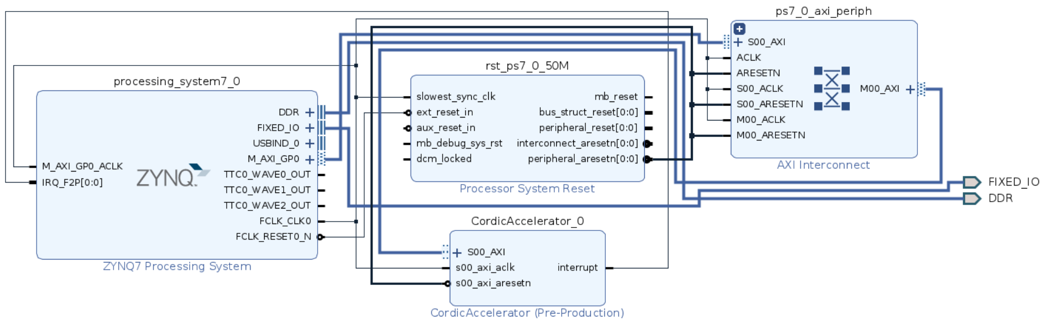


Fig 5.13 : Block design for system integration testing

The AXI interconnect decides the mapping of the memory space of each IP connected as a slave. It translates the addresses from the processors space to the address space of the slave IP. Here, our IP has 7 registers. A base address is taken which will then be translated to the address of the first register, 0 in the memory space of our IP. The address editor can be used to change this. In our case, our IP has a 64KB address space (even though it has only 7*4 = 28 bytes of valid addresses), which can be addresses from the processor starting from index 0x4fc0_0000. This was the address allocated by the system by default.

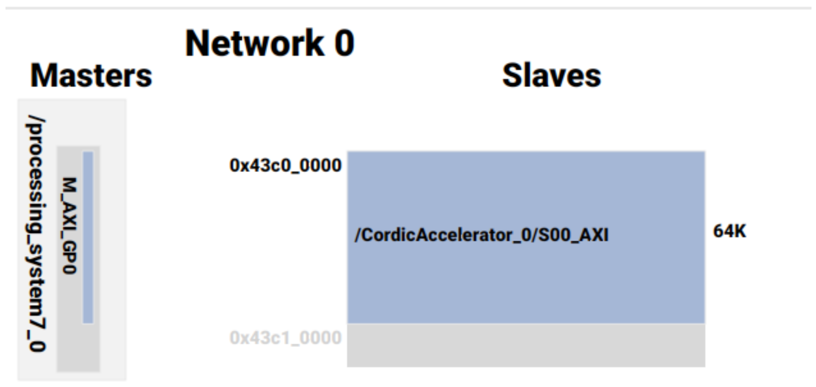


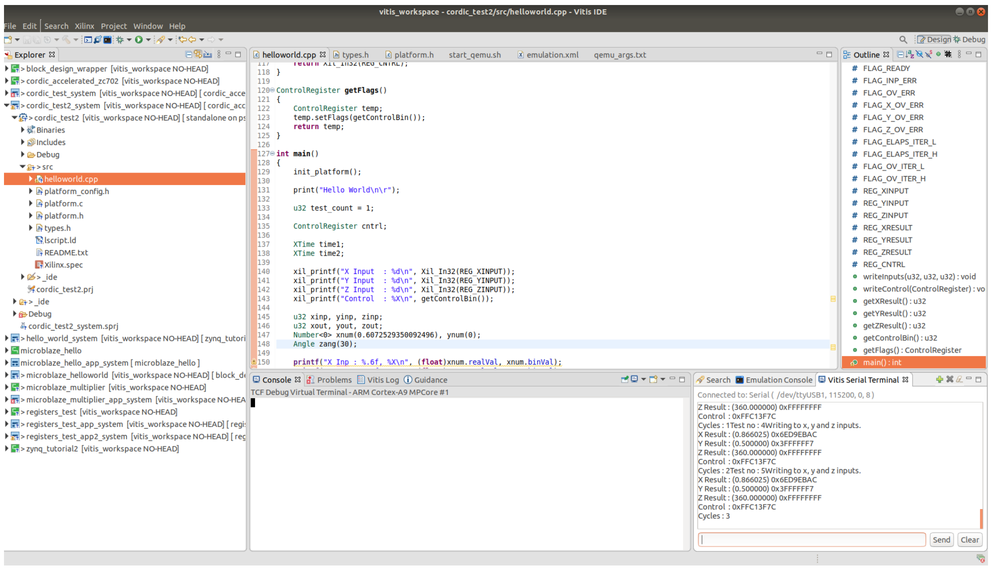
Fig 5.14 : Memory map for interconnect

5.3.3 Synthesis and implementation

After the block design, we perform synthesis and implementation. In the synthesis stage, the RTL code is converted to the resources on board the programmable logic of the SoC, that is registers, block memories and LUTs. In the implementation stage, the positions of these resources are decided and the routing between them is made. The ZC702 board uses a 50MHz oscillator.

5.3.4 Embedded software and testing

To test if the IP is working and to interact with it, we need to write software that runs on the ARM processor on board the Zynq-7000 SoC. We use the vitis embedded ide from Xilinx for this. The result of implementation from vivado is imported from vitis and we write the code that is compiled and flashed on the Zynq SoC. Here, we have written some C++ code that computes the sine of some angle. The result and latency measured for the IP to compute the result is sent to a computer via serial communication.



```
}

ControlRegister getFlags()
{
    ControlRegister temp;
    temp.setFlags(getControlBin());
    return temp;
}

int main()
{
    init_platform();

    print("Hello World\n\r");

    u32 test_count = 1;

    ControlRegister cntrl;

    XTime time1;
    XTime time2;

    xil_printf("X Input  : %d\n", Xil_In32(REG_XINPUT));
    xil_printf("Y Input  : %d\n", Xil_In32(REG_YINPUT));
    xil_printf("Z Input  : %d\n", Xil_In32(REG_ZINPUT));
    xil_printf("Control  : %X\n", getControlBin());

    u32 xinp, yinp, zinp;
    u32 xout, yout, zout;
    Number<0> xnum(0.6072529350092496), ynum(0);
    Angle zang(30);

    printf("X Inp : %.6f, %X\n", (float)xnum.realVal, xnum.binVal);
```

```
Connected to: Serial ( /dev/ttyUSB1, 115200, 0, 8 )
Z Result : (360.000000) 0xFFFFFFFF
Control  : 0xFFC13F7C
Cycles : 1Test no : 4Writing to x, y and z inputs.
X Result : (0.866025) 0x6ED9EBAC
Y Result : (0.500000) 0x3FFFFFF7
Z Result : (360.000000) 0xFFFFFFFF
Control  : 0xFFC13F7C
Cycles : 2Test no : 5Writing to x, y and z inputs.
X Result : (0.866025) 0x6ED9EBAC
Y Result : (0.500000) 0x3FFFFFF7
Z Result : (360.000000) 0xFFFFFFFF
Control  : 0xFFC13F7C
Cycles : 3
```

Fig 5.15 : Screenshot of Vitis IDE

5.3.5 Using the CORDIC coprocessor

To use the CORDIC coprocessor to find, for example the sine or cosine of an angle, convert the input angle to the number system as described in 4.1.1 and set the input X, Y and Z registers and the control register according to the following table.

Input X	Input Y	Input Z	Rotation System	Rotation Mode
0.6072529350..	0	Input angle	1 (circular)	1 (rotation)

Table 5.1 : Inputs to find sine and cosine

Then let the CORDIC coprocessor complete computation.

After completion of computation is notified the resulting X value provides the cosine of the input angle and the resulting Y value provides sine of input angle, Z value provides the angle residue.

6 Results

In our work, we have designed, tested and characterised a CORDIC coprocessor IP in simulation and on the Zynq-7000 SoC. We have studied the performance and overflow characteristics of the CORDIC algorithm.

6.1 CORDIC Characterization

Our characterization efforts were on two fronts. The first was to characterise the performance of the algorithm and its variation with various changes in the inputs like iteration count and identify patterns that would be useful for users. The second was to study the overflow characteristics of the algorithm and identify patterns to provide recommendations to the user on the range of inputs they should give.

Both types of analysis were done on all 4 modes of operation, circular rotation, circular vectoring, hyperbolic rotation and hyperbolic vectoring. We used the Q0.31 fixed point system for X and Y values in the circular mode and Q3.28 system for inputs in the hyperbolic mode. All tests are done with 30 iterations unless specified.

6.1.1 Overflow characteristics for circular system

If the input coordinates are plotted as a scatter plot with separate colours for successful CORDIC operation and for instances that overflowed, a pattern emerges. The plot is identical for rotation and vectoring. The green dots are the successful inputs and red dots are the inputs that caused overflow.

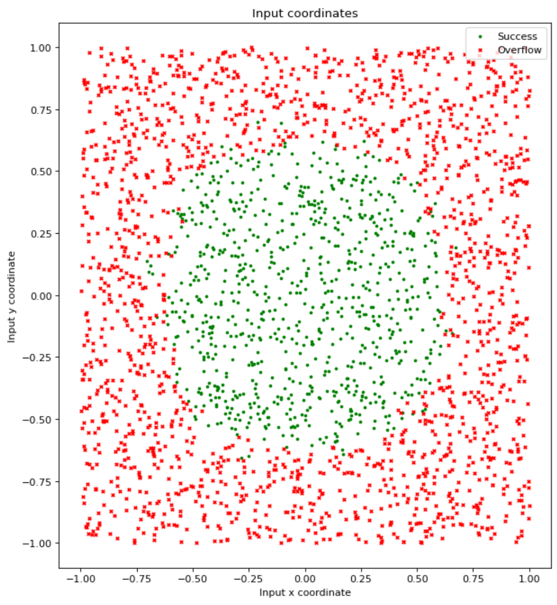


Fig 6.1 : Scatter plot of input coordinates to observe overflow characteristics in circular system

Only input points that are more than a particular distance away from the centre tend to overflow. This value was found to be near the inverse of the scaling factor, 1/1.646… = 0.607… . This is expected since only the coordinates above 0.607… in magnitude can reach X or Y values greater than 0.607… * 1.646… = 1 after including the scaling factor and hence exceeding the limit of our number system.

The users are recommended to keep input coordinates to have magnitude less than 0.6 times the maximum value to guarantee that no overflows will occur. Additionally, the same plot was made for the output coordinates for rotation mode and output X value v/s output angle for vectoring mode.

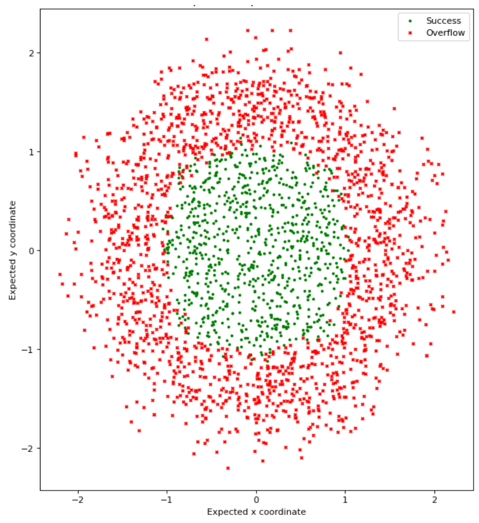


Fig 6.2 : Scatter plot of output coordinates in circular rotation

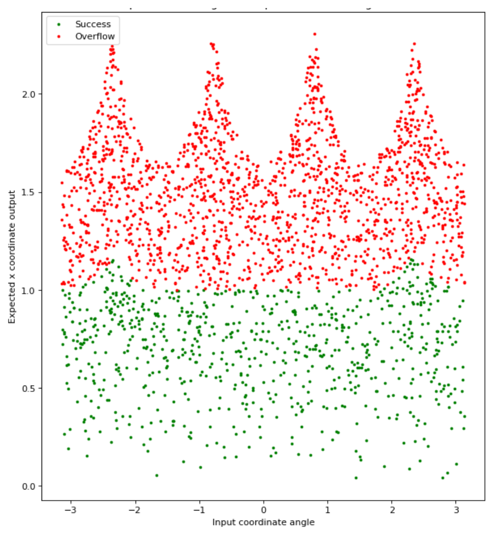


Fig 6.3 : Scatter plot of output X value v/s expected output angle in circular vectoring

Here, we can notice that the rotation mode overflows when either of the output exceeds 1 and the vectoring mode overflows when the X output (input amplitude * 1.646…) exceeds 1. There is no dependence on the expected angle. Only the magnitude of the X, Y input vector impacts overflow probability.

6.1.2 Overflow characteristics for hyperbolic system

If the input coordinates are plotted as a scatter plot for hyperbolic rotation mode, the following pattern emerges between points that overflow and points that successfully complete.

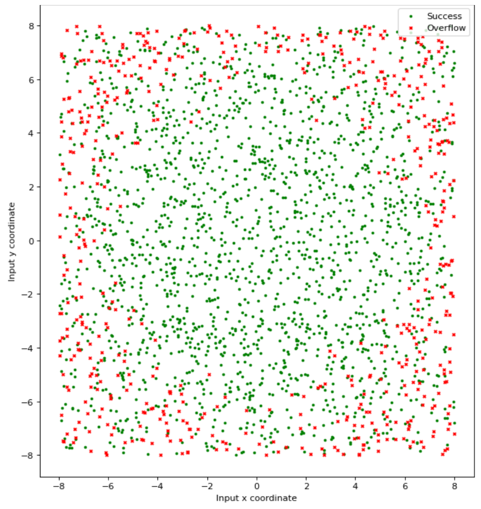


Fig 6.4 : Scatter plot of input coordinates to observe overflow characteristics in hyperbolic rotation

Again, only points with magnitude greater than a particular value tends to overflow. Similar to the circular rotation system, this is because of the scaling factor. However, here since we are using the Q3.28 number system, the maximum number that can be represented is 8. So, the boundary becomes 8/1.205… = 6.638… So, the recommendation is to only give inputs that are within 0.82 times the maximum value of the X and Y number system.

For the hyperbolic vectoring mode, a different pattern emerges. Only inputs with a negative X coordinate or inputs with X coordinate less than the Y coordinate overflowed. The hyperbolic vectoring mode is not designed to take inputs with X less than 0. For values absolute of Y greater than X, $\tanh^{-1}$ and $\sqrt{x^2 - y^2}$ do not have defined real values.

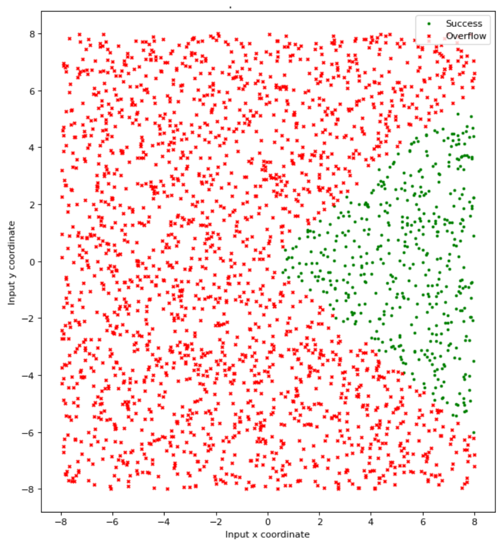


Fig 6.5 : Scatter plot of input coordinates for hyperbolic vectoring

When we plot the output coordinates of the hyperbolic rotation mode, an inference similar to the circular mode can be made. Overflows happen when the either the X or Y expected output is greater than 8, which is the maximum value that can be represented in Q3.28

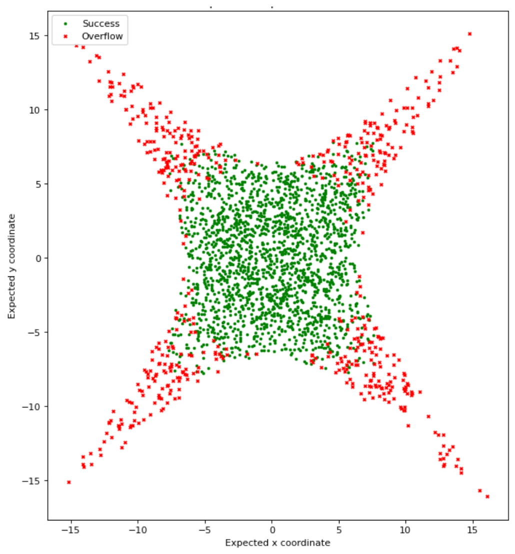


Fig 6.6 : Scatter plot of output coordinates for hyperbolic rotation

Since overflow could be predicted very accurately for the hyperbolic vectoring mode from input coordinates alone, plotting the outputs reveals no other patterns.

6.1.3 Error characteristics for circular system

If we plot the magnitude of error in the output X and Y coordinates from the expected value on a logarithmic scale (dB), we get the following plot.

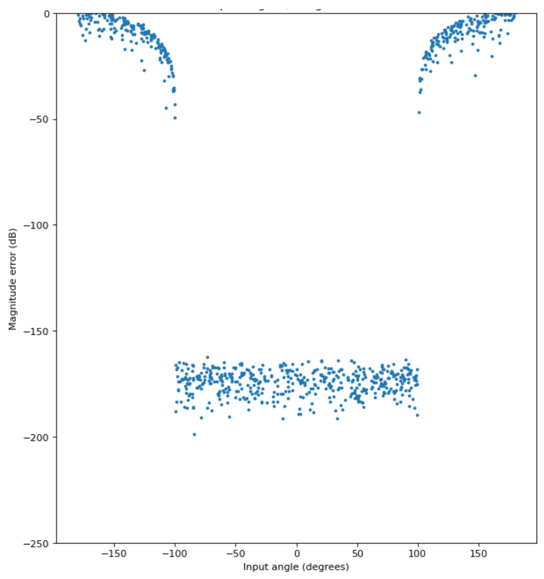


Fig 6.7 : Output coordinate error magnitude v/s input rotation angle in circular rotation

We can observe how the error is very low for values between -100 and 100 degrees but it increases as we move away from this range. This is because of the convergence limit of the CORDIC algorithm. To solve this, we do a “pre-rotation” by 180 degrees in the controller if required, to bring the input into this range. The same plot for the IP as a whole would not have this issue. The output for the circular vectoring mode shows a similar trend for both output angle and X coordinate. The plot below shows error when pre-rotation is done.

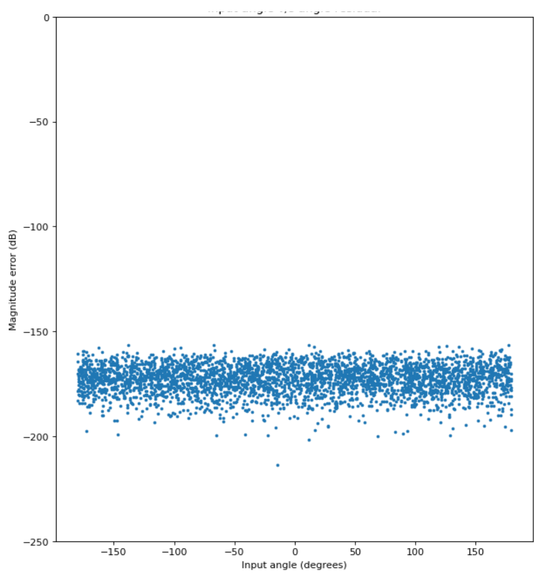


Fig 6.8 : Output coordinate error magnitude v/s input rotation angle in circular rotation after adding pre-rotation

Next, if we plot the histogram of the output errors, it looks as shown below

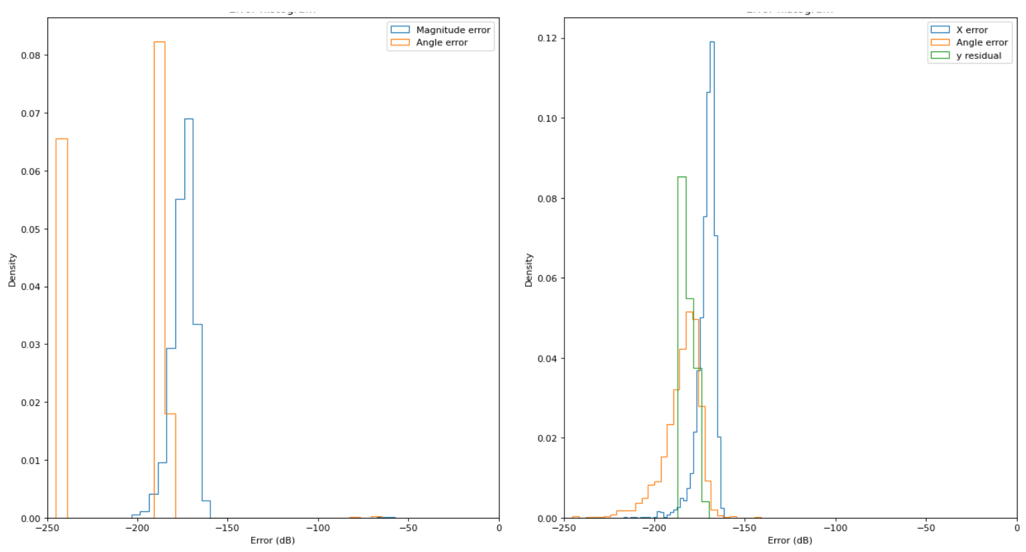


Fig 6.9 : Output error histograms in circular rotation (left) and vectoring (right)

6.1.4 Error characteristics for hyperbolic system

If we plot the magnitude of error in the output X and Y coordinates from the expected value on a logarithmic scale (dB) for 10,000 inputs, we get the following plot.

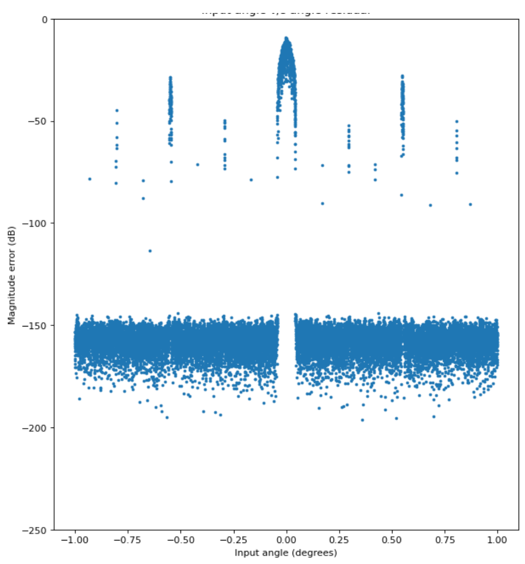


Fig 6.10 : Output coordinate error magnitude v/s input rotation angle in hyperbolic rotation

We can see a distinct pattern where the error is very high if the input rotation angle is equal to some specific values. This is an artefact of the CORDIC algorithm itself. The angles near which this happens are the first few angles in the $\tanh^{-1}$ table. An identical trend can be seen in hyperbolic vectoring mode.

The histograms of the errors looks as shown below

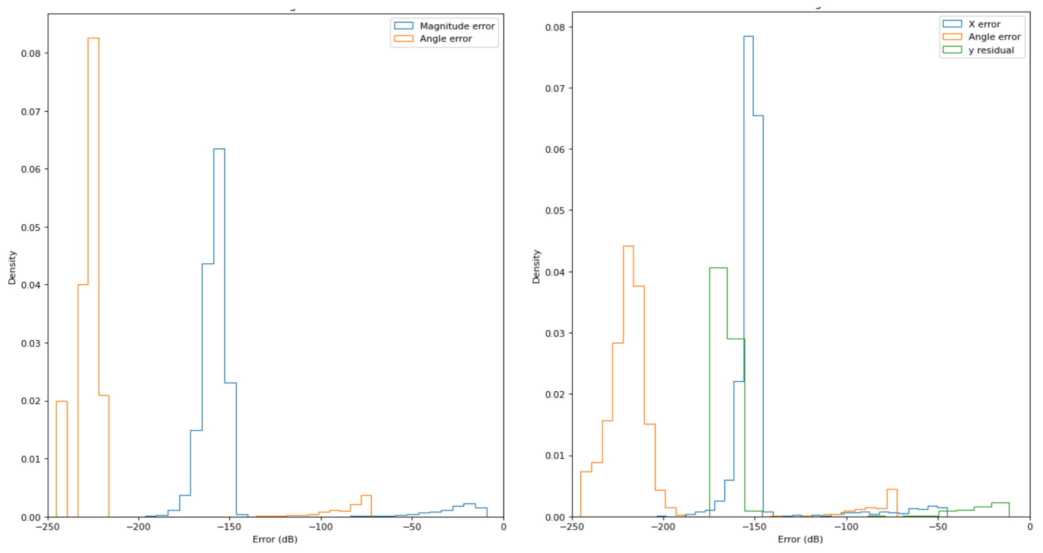


Fig 6.11 : Output error histograms in hyperbolic rotation (left) and vectoring (right)

6.1.5 Error variation with iteration count

The log files were generated for multiple values of maximum iteration count and the variation in output error with the iteration count was plotted using box-and-whiskers plots. The orange lines represent the median value. 50% of the test cases have errors in the range given by the boxes. 90% of test cases have errors within the whiskers. The rest of the points are plotted outside.

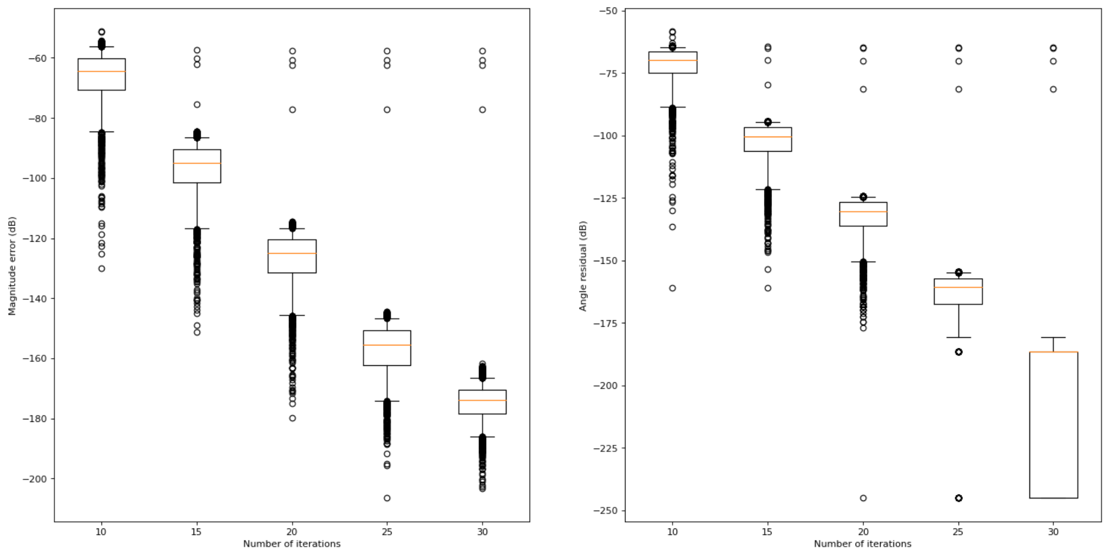


Fig 6.12 : Variation of output error with iteration count for circular rotation (left) and vectoring (right)

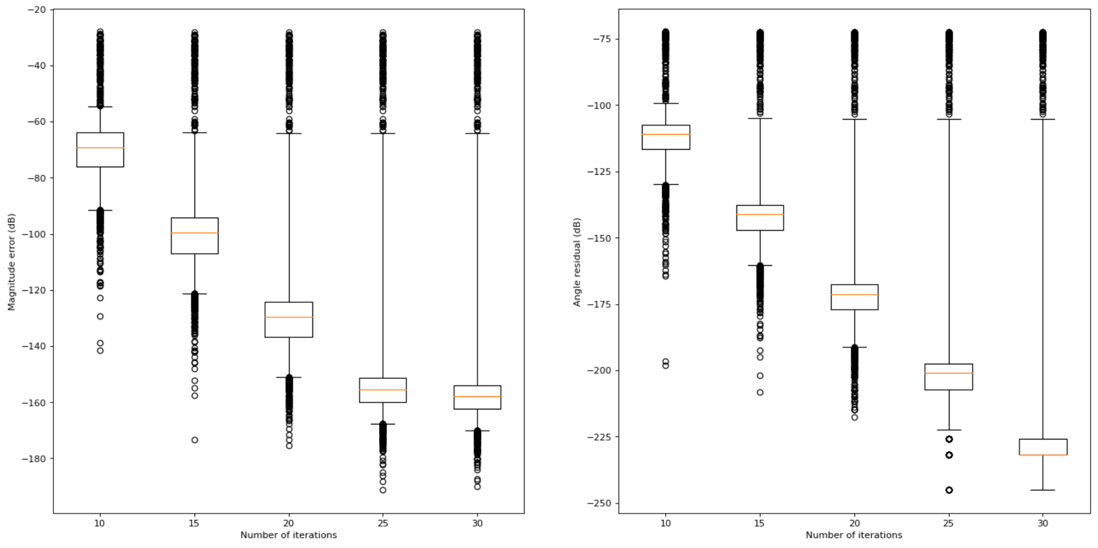


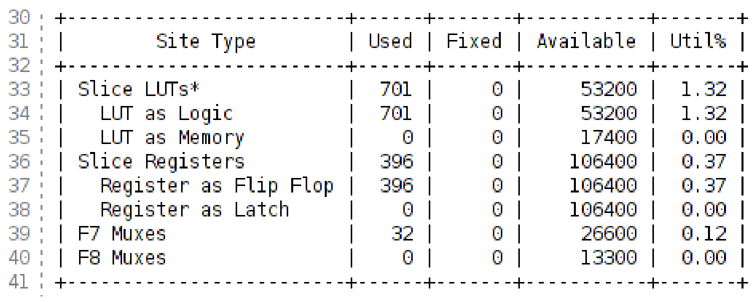
Fig 6.13 : Variation of output error with iteration count for hyperbolic rotation (left) and vectoring (right)

6.2 IP Performance

The IP was synthesised targeting the Zynq-7000 SoC and consumed less than 1% of the on-board resources. The total time taken for the algorithm to run was also measured using code running on the ARM processor.

6.2.1 Resource utilisation

Only about 1% of the total resources on the programmable part of the board was used for synthesising the IP

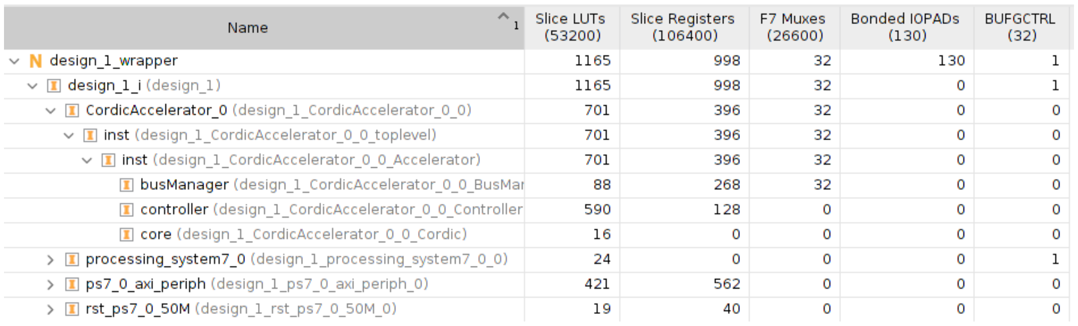


```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  701 |     0 |     53200 |  1.32 |
|   LUT as Logic          |  701 |     0 |     53200 |  1.32 |
|   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
| Slice Registers         |  396 |     0 |    106400 |  0.37 |
|   Register as Flip Flop |  396 |     0 |    106400 |  0.37 |
|   Register as Latch     |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                |   32 |     0 |     26600 |  0.12 |
| F8 Muxes                |    0 |     0 |     13300 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Fig 6.14 : Screenshot of the resource utilisation report for the IP after synthesis

The detailed utilisation report reveals the utilisation of the submodules of the IP as well as the utilisation of the other components of the system like the processing system, the AXI interconnect and the reset controller. We can see that the controller takes up the majority of the logic resources (Slice

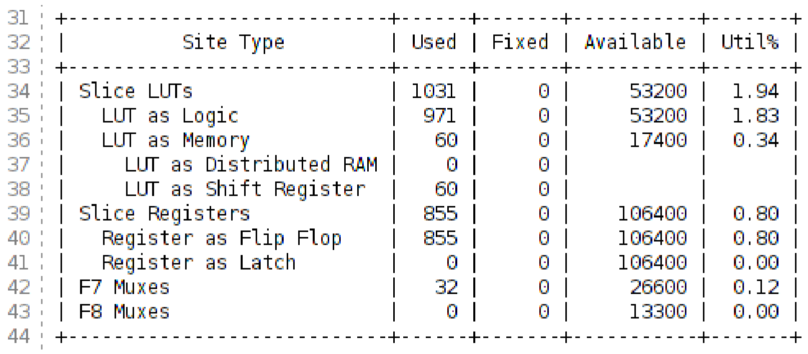
LUTs) whereas the bus manager takes up the memory resources (Slice Registers). The look up table module was absorbed into the controller module as part of some optimization stage.



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Bonded IOPADs (130)	BUFGCTRL (32)
design_1_wrapper	1165	998	32	130	1
design_1_i (design_1)	1165	998	32	0	1
CordicAccelerator_0 (design_1_CordicAccelerator_0_0)	701	396	32	0	0
inst (design_1_CordicAccelerator_0_0_toplevel)	701	396	32	0	0
inst (design_1_CordicAccelerator_0_0_Accelerator)	701	396	32	0	0
busManager (design_1_CordicAccelerator_0_0_BusMar	88	268	32	0	0
controller (design_1_CordicAccelerator_0_0_Controller	590	128	0	0	0
core (design_1_CordicAccelerator_0_0_Cordic)	16	0	0	0	0
processing_system7_0 (design_1_processing_system7_0_0)	24	0	0	0	1
ps7_0_axi_periph (design_1_ps7_0_axi_periph_0)	421	562	0	0	0
rst_ps7_0_50M (design_1_rst_ps7_0_50M_0)	19	40	0	0	0

Fig 6.15 : Detailed utilisation report after synthesis

The total utilisation of the system after implementation stage was also studied as can be seen in the following screenshot

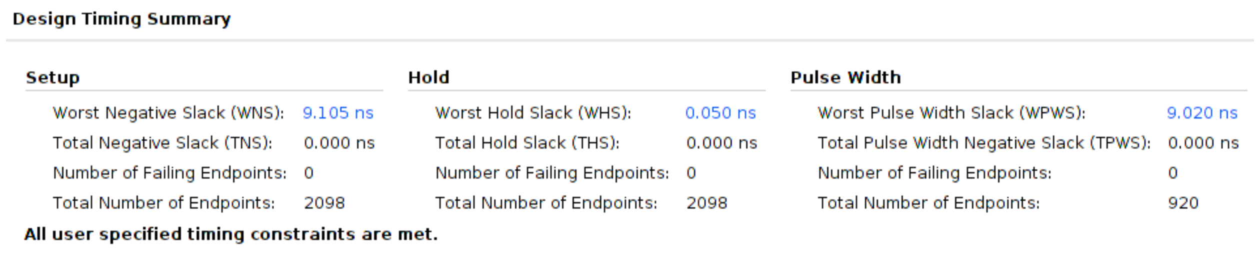


```
+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 1031 |     0 |     53200 |  1.94 |
|   LUT as Logic             |  971 |     0 |     53200 |  1.83 |
|   LUT as Memory            |   60 |     0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |           |       |
|     LUT as Shift Register  |   60 |     0 |           |       |
| Slice Registers            |  855 |     0 |    106400 |  0.80 |
|   Register as Flip Flop    |  855 |     0 |    106400 |  0.80 |
|   Register as Latch        |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   32 |     0 |     26600 |  0.12 |
| F8 Muxes                   |    0 |     0 |     13300 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Fig 6.16 : Screenshot of the resource utilisation report for the system after implementation

6.2.2 Timing Analysis

Analysing the timing summary, we can see that the worst negative slack is 9.105ns. The timing analysis and implementation was done using the 50MHz clock of the ZC702 board. Our design was able to pass the timing analysis without need for any modifications.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	9.105 ns	Worst Hold Slack (WHS):	0.050 ns	Worst Pulse Width Slack (WPWS):	9.020 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2098	Total Number of Endpoints:	2098	Total Number of Endpoints:	920

All user specified timing constraints are met.

Fig 6.17 : Design Timing Summary after implementation

6.2.3 Total latency for the user

The total latency taken for writing to 4 registers (X, Y, Z inputs and control register) and reading from 3 registers (X, Y, Z outputs) was measured to be 2290 cycles. Of this, 1058 cycles were consumed for writing, 642 cycles polling for the result and 590 cycles for the reads.

7 Conclusion & Future Scope

7.1 Conclusion

We have developed, tested and characterised a CORDIC coprocessor IP. It was able to achieve the expected performance in terms of error, resource utilisation and timing. We were able to demonstrate the working of the IP on the Zynq-7000 SoC on board the ZC702 board.

7.2 Future Scope

Further work can be done to document the usage of the IP to compute specific functions for system designers. We have tested the most important subset of functions that can be generated and have exhaustively verified the correctness of the hardware. Though this is enough to ensure that any of the functions that can be derived from CORDIC can indeed be computed using our IP, it will be difficult for system designers to use our IP to do the same. So, we need to provide better documentation on not just the interface to our IP but also on how it can be used to compute said functions and various recommendations on how to use it. We can also develop software to interface with the coprocessor to abstract away lower level details like the positions of the fields of the control register.

Our IP has only been tested for the Zynq-7000 SoC and currently only has support for the AXI4-Lite protocol. In the future, more bus protocol support must be added to improve compatibility with systems using other bus protocols like Wishbone. We can also test our IP with other SoCs and FPGAs to improve support.

8 Bibliography

[1.] **Jack E Volder**, "*The CORDIC Trigonometric Computing Technique*", IRE TRANSACTIONS ON ELECTRONIC COMPUTERS, Vol EC-8, pp330-334 Sept 1959

[2.] **Ray Andraka**, "*A survey of CORDIC algorithms for FPGA based computers*", Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, March 1998

[3.] **JS Walther**, "*A unified algorithm for elementary functions*" Proceedings of the May 18-20, 1971, spring joint computer conference, May 1971

[4.] **Ramesh Bhakthavatchalu; M.S. Sinith; Parvathi Nair; K. Jismi,**"*A comparison of pipelined parallel and iterative CORDIC design on FPGA",* 5th International Conference on Industrial and Information Systems, 2010

[5.] **Jason Todd Arbaugh**, "*Table Look-up CORDIC : Effective Rotations Through Angle Partitioning*" PhD dissertation, The University of Texas at Austing, December 2004

[6.] **STMicroelectronics**, "*STM32H7- CORDIC coprocessor*", Revision 1.0 Product training presentation

[7.] **Accellera**, "SystemVerilog 3.1a Language Reference Manual Accellera's Extensions to Verilog"

[8.] **Free Software Foundation, Inc**, "*GNU GENERAL PUBLIC LICENSE*", Version 3, 29 June 2007

[9.] **OpenCores.org**, "*WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*", Revision: B.3, September 7, 2002

[10.] **Arm Limited**, "*AMBA AXI and ACE Protocol Specification Version E*", Online documentation/manual