ECE 441 Design of Digital and VLSI Systems

Project Final Report Spring 2019

Group Number	#7
Project Title	CORDIC Processor Design & Implementation

Group Members	Kyle Cathers V00851761	Alex Kolodinsky V00831707
Submission Date	March 30th 2019	

Alex Kolodinsky
Kyle Cathers
4143 Cortez Pl.
Victoria, BC
V8N 4R9

Dr. Fayez Gebali, P.Eng
3800 Finnerty Road
Victoria, BC
V8P 5C2

Dear Dr. Gebali,

Please accept the accompanying report, entitled **"CORDIC Processor Design & Implementation"**.

The enclosed report includes research, design specifications, and VHDL code created to implement a CORDIC algorithm on the Basys 3 FPGA board. As well, the report contains final results and a performance evaluation of the final CORDIC implementation.

Throughout the duration of the project, our group has learned the basics of working in the Xilinx Vivado environment, become proficient in VHDL code and syntax, and utilized design simulations to troubleshoot bugs. The Basys 3 FPGA board was studied and used as the hardware source to implement our VHDL CORDIC design. We hope that this report clearly reflects our understanding of the CORDIC algorithm and VLSI Design.

Our team would like to thank Dr. Gebali, as well as Brent Sirna for the support and feedback throughout the term.

Sincerely,

Group 7
ECE 441
Spring 2019

Table of Contents

List of Tables and Figures

Glossary

7-Segment Display - An electronic device which displays hexadecimal numbers by illuminating different combinations of seven LED segments

ALU - Arithmetic Logic Unit: A digital circuit used to perform arithmetic operations

Basys 3 - A digital circuit development platform designed for the Artix-7 series FPGA

BIST - Built-in self-test: A mechanism for a machine to test itself

CORDIC - Coordinate Rotation using Digital Computers: An iterative process to compute functions using addition and shifting operations

CRC - Cyclic Redundancy Check: An error detecting code which detects accidental changes to raw data

DFT - Design for Testability: Integrated circuit design that includes testing features build directly into the design

FPGA - Field Programmable Gate Array: A silicon chip that contains reprogrammable digital circuitry (combinational logic blocks)

LED - Light Emitting Diode

LUT - Lookup Table: A data table which contains pre-calculated values of □ for each CORDIC iteration

VHDL - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language: Hardware language used for the design of FPGAs and other digital or mixed-signal systems

VLSI - Very Large Scale Integration: The creation of an integrated circuit by combining many transistors into one chip

Vivado - Software produced by Xilinx for HDL simulation, synthesis, and design

1.0 Background and Literature Review

Coordinate **R**otation using **Di**gital **C**omputers (CORDIC) was invented by Jack E. Volder in 1959, and further developed by John Walther in 1971 [1]. Published in the International Journal of Engineering Research and Technology, Volume 4, Issue 8 [1], CORDIC is designed to efficiently execute complex computations such as square-roots, logarithms, trigonometric functions, as well as multiplication and division. The CORDIC algorithm computes these functions by rotating a vector along a trace of a circle in the XY-plane, utilizing the equation of said circle. Using this equation, the total rotation of the vector can be solved for x and y in terms of cosine and sine functions [2].

$$x^2 + y^2 = r^2$$

Figure 1: The Equation of a Circle.

$$x = r\cos(\square_0 + \square)$$
$$y = r\sin(\square_0 + \square)$$

Figure 2: Total rotation of a vector in terms of X and Y.

Now the problem with equations 2 & 3 is that rotating a vector this way will involve multiplications steps. By utilizing special angles (□ᵢ) found in a “lookup table”, the CORDIC changes these multiplication steps into simple shift and add operations. The benefit of this is that CORDIC machines do not need a multiplier, which greatly reduces the cost and complexity of the machine. Each CORDIC rotation can be described by three equations:

$$x_{i+1} = x_i - \square_i + y_i\delta_i$$
$$y_{i+1} = y_i + \square_i + x_i\delta_i$$
$$z_{i+1} = z_i - \square_i \square_i$$

Figure 3: Basic CORDIC Equations.

This method simplifies each CORDIC rotation into a procedure involving only one lookup table access, two shifts, and three additions. Variable □ᵢ describes the direction of rotation (+1 for counter-clockwise and -1 for clockwise), □ᵢ corresponds to the special angles referenced in the lookup table, while Z_i denotes the remaining angle of rotation.

CORDIC can also perform it’s calculations based on two different modes: vectoring, and rotation. For vectoring mode, the direction of rotation is controlled such that Y → 0, while in rotation mode the direction is controlled such that Z → 0. The CORDIC algorithm is completed upon the target parameter reaching 0. The equations corresponding to the final outputs can be seen in figures 4 and 5 [2].

$$x_n = \sqrt{{x_0}^2 + {y_0}^2}$$
$$\mathrm{z}_n = z_0 + \arctan\left(\frac{y_0}{x_0}\right)$$

Figure 4: Vectoring Mode Outputs

$$\mathrm{x}_n = x_0 cos(z_0) - y_0 sin(z_0)$$
$$\mathrm{y}_n = y_0 cos(z_0) + x_0 sin(z_0)$$

Figure 5: Rotation Mode Outputs

The CORDIC processor has been implemented in various fields including motor control, wireless navigation and signal processing [2]. According to another International Journal of Engineering Research and Technology article titled, “CORDIC Based DFT on FPGA for DSP”, in digital signal processing, multiple CORDICs can be used to compute discrete fourier transforms and discrete cosine transforms. To achieve this, a pipelined CORDIC algorithm is implemented to handle multiple iterative calculations [3]. As a result of industry applications it has been observed that use of the CORDIC algorithm greatly reduces complexity and cost, at the exchange of a longer computation period due to its iterative nature.

2.0 Project Introduction

The objective of the project is to implement an iterative CORDIC processor in hardware using VHDL code programmed onto the Basys 3's on-board FPGA. Upon completion the Basys 3 will be able to perform rotation and vectoring calculations through simple shifting and addition operations. The Xilinx Vivado environment was used throughout the project for its simplicity and ability to simulate VHDL code.

The finished design must be able to convert from predetermined fixed point 16-bit input data, perform iterative calculations based on said input, and display the result to a 7-segment HEX display on the Basys 3 board. The input data is formatted to represent specific data ranges ($0 \leq x \leq 1$, $0 \leq y \leq 1$, $0 \leq z \leq 2$ radians).

The project will use start and reset inputs (debounced pushbuttons) for state control, and a number of switches for input and display data selection. The project includes a number of sub components within the CORDIC processor component: an ALU, a storage module, and a lookup table. The controller for the CORDIC is built into the processor component, running a finite state machine to control processor operation (e.g. iteration control and signal assignment to sub components).

3.0 Project Deliverables

In accordance to the University of Victoria's Design of Digital and VLSI Systems (ECE 441), this report was created as a portion of deliverables for the CORDIC Processor Design & Implementation project. A full list of deliverables include:

- Mid-Term Progress Report (0%)
- Method for BIST (15%)
- In-Person Demo (20%)
- Final Report (65%)

4.0 Design Responsibilities

The design process was divided into milestones and organized over the course of the project timeline. These milestones were tasked to group members based on personal strengths, with the secondary group member acting as a supporting role in its development. Once the project milestone was fully specified, a hierarchical approach was then used to begin design in accordance to the project timeline. A complete design overview for each task can be referenced in Section 6.0: *Design Implementation.*

Top.vhd

Responsibility: Kyle Cathers

Deliverables

- Integrate the CORDIC processor with the debounce, hex driver, and constraint file (board setup) to create communication between internal systems and hardware operation
- Design logic to allow for initial data selection (initial data based on project requirements and the ECE 441 course website)
- Design logic to allow for selection of output data variable (x_i, y_i, or z_i) to send to hex display

CORDIC Processor

Responsibility: Kyle Cathers

Deliverables

- Implement CORDIC finite state machine (see section 5)
- Design processor to control CORDIC subunits (ALU, storage, LUT) according to appropriate FSM behavior

Lookup Table

Responsibility: Alex Kolodinsky

Deliverables

- Implement a lookup table to store specific values of □ (special angles) in radians, in binary form for each iteration (according to format shown by Figure 13)
- Corresponding □ output accessed through iteration input signal

Hex Driver

Responsibility: Alex Kolodinsky

Deliverables

- Analyze the given Hex_Driver.vhd from the course website to be used with the design
- Ensure that binary data is transformed into hexadecimal format correctly, verify proper positioning for digits on the 7-segment display
- Implement structural code for top level block to use hex driver

Debouncer

Responsibility: Alex Kolodinsky

Deliverables

- Analyze the given debounce.vhd from the course website to be used with the design
- Implement debounce for both “Start” and “Reset” buttons into top level block

Troubleshooting & Testing

Responsibility: Alex Kolodinsky & Kyle Cathers

Deliverables

- Ensure a working final implementation
    - Run simulations through Xilinx and analyze waveform diagrams to understand and fix problems
    - Modify and re-simulate VHDL code as needed when problems arise
- Successfully program the finished CORDIC implementation onto the Basys 3
- Exhaust all test scenarios via both design simulations and physical testing of the board

5.0 Design Specifications and Approach

The CORDIC processor operates on 16 bit data, and outputs 16 bit data to the display in a hexadecimal format. In order to represent the data inputs, a fixed point format is used. The formatting (fixed-point to binary conversion) can be seen in Figure 6 below. Data is implemented primarily in the form of STD_LOGIC_VECTORS in the project, and is converted to SIGNED data when used by the ALU.

X Data

Index:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

Y Data

Index:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

Z Data

Index:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}

Figure 6: Data Formatting for X, Y, Z

The finite state machine for the CORDIC processor block can be seen below in Figure 7. The processor start in the wait mode by default, and is set back to this mode whenever the RESET button is pressed. During the wait mode the display defaults to ‘0000’, and the input switches (mode and reset switches) are read in as inputs. Once the START button is pressed, the processor begins it’s calculations based on said inputs by first entering the start mode. During

start mode, the initial data (x_0, y_0, z_0) is entered into the ALU input and storage (position 0). After the next clock rising edge is detected, the processor enters into iteration mode. Each clock cycle the iteration mode then increments the i signal, sends the result of the last iteration back into the ALU input, and feeds the result into the storage (position i). After 15 iterations the processor is done, and enters into its display mode. During display mode, the iteration_select and data_select switches are read, and the corresponding data (x_i, y_i, or z_i) is sent to the display. The wait mode may be set at any point in the operation by pressing the RESET button, the START button is only read during the wait mode.

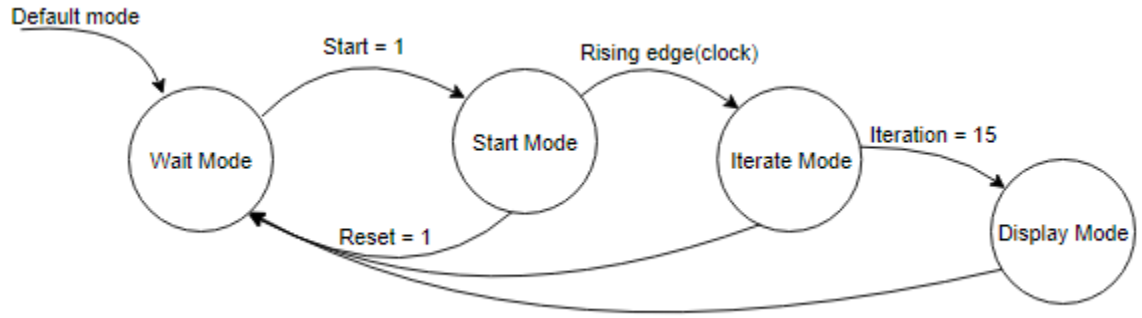


Figure 7: CORDIC Processor Finite State Machine

The connection between primary internal and external signals of the CORDIC processor block can be seen in Figure 8 below. The inputs to the storage block depend on the mode of the processor, during start mode the block takes in the initial data x_0, y_0, z_0, and then takes in the outputs of the ALU during iteration mode. During the wait and display modes the storage block does not take in data. The input of the ALU also depends on the state of the processor in the same way as the storage block, also taking in initial data during start mode, and taking in the result of the previous iteration during iteration mode.

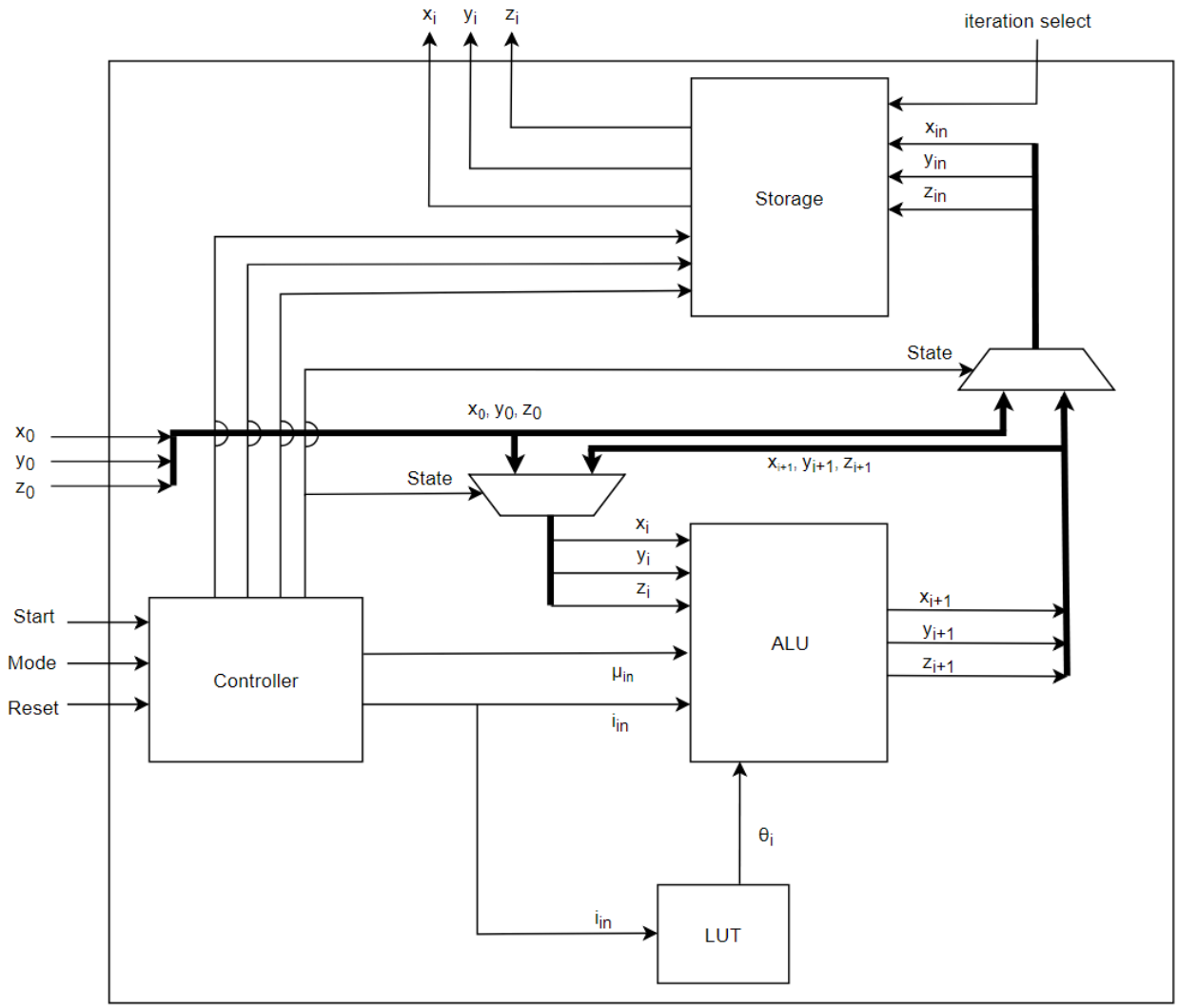


Figure 8: CORDIC Process System Diagram

6.0 Design Implementation

6.1. VHDL Implementation

Block diagrams and descriptions covering overall functionality of each VHDL component (with a focus on IO signals) can be seen in the subsections below. Code for each component can be seen in the 'VHDL Code' section of the appendix.

6.1.1 Top

The top module acts as a connection block between the CORDIC processor, the hex display driver, and the debounce circuits. The I/O signals specified in the constraints file (that connect to the board) are also connected to the project at this level.

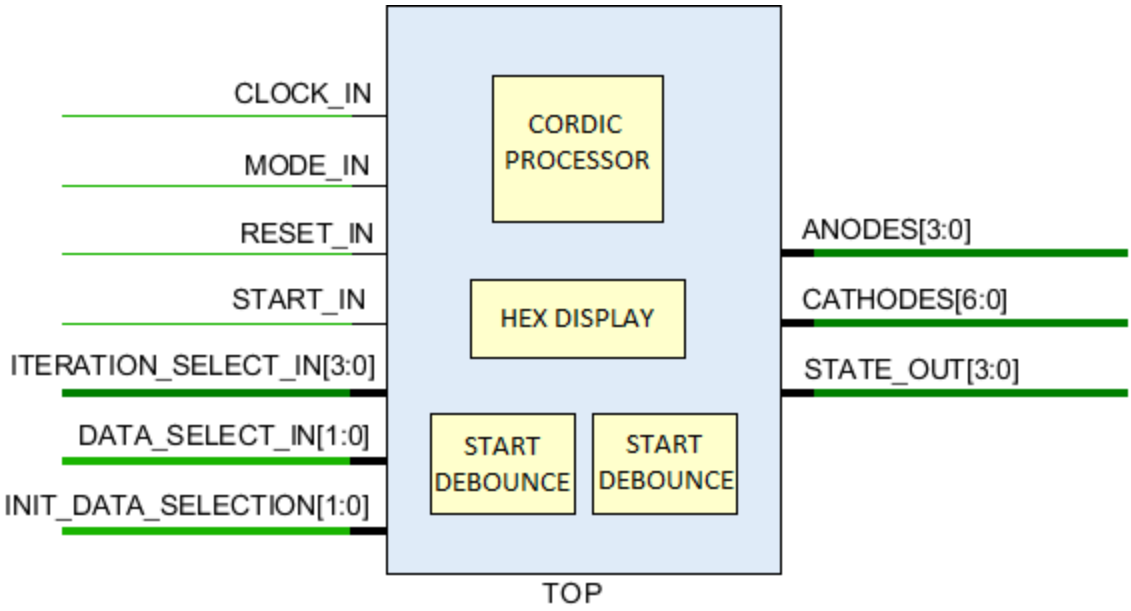


Figure 9: Top Level Component Block Diagram

Signal Name	Mode	Data Type	Functionality
CLOCK_IN	In	STD_LOGIC	Clock signal, on board generated
MODE_IN	In	STD_LOGIC	Switch input, Denotes CORDIC operation mode: Vectoring = '1' Rotation = '0'
RESET_IN	In	STD_LOGIC	Button input, sets CORDIC back to wait state
START_IN	In	STD_LOGIC	Button input, jumps CORDIC to begin calculations
ITERATION_SELECT_IN	In	STD_LOGIC_VECTOR 4 bits	Switch input, selects output data iteration (0-15) to display
DATA_SELECT_IN	In	STD_LOGIC_VECTOR 2 bits	Switch input, selects X, Y, or Z data to display 00 = X, 01 = Y, 10 = Z, 11 = outputs 0's
INIT_DATA_SELECTION	In	STD_LOGIC_VECTOR 2 bits	Switch inputs, works in conjunction with MODE_IN. Used to select input data to CORDIC (x_0, y_0, z_0)
ANODES	Out	STD_LOGIC_VECTOR 4 bits	Outputs to HEX display, selects which digit to display (e.g. 1110 displays rightmost digit)
CATHODES	Out	STD_LOGIC_VECTOR 7 bits	Outputs to HEX display, selects which segments to enable to create digits (e.g. 0110000 = digit 3)
STATE_OUT	Out	STD_LOGIC_VECTOR 4 bits	Outputs to board LEDs, signifies current mode of CORDIC processor (0001 = wait, 0010 = start, 0100 = iterate, 1000 = display)

Table 1: Top Level Component I/O Signal Descriptions

6.1.2 CORDIC Processor

The CORDIC block is comprised of a controller, an ALU, a LUT, and a storage block. Note the controller is integrated into the processor, and thus is not shown in the block diagram below.

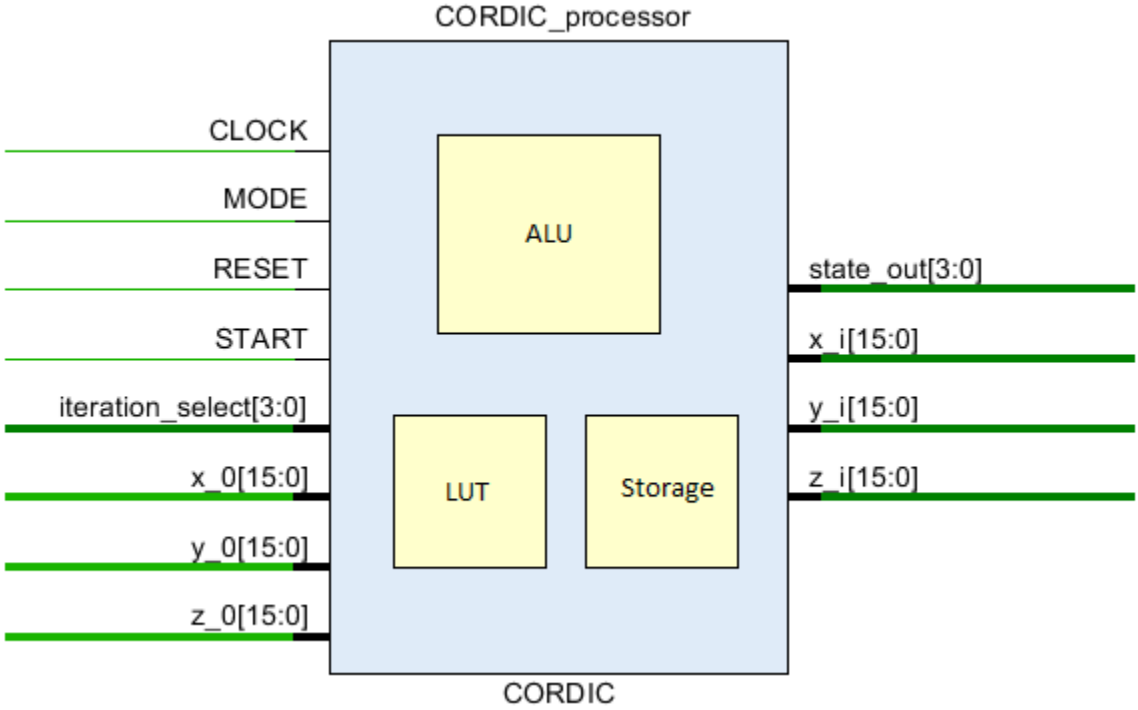


Figure 10: CORDIC Processor Component Block Diagram

Signal Name	Mode	Data Type	Functionality
CLOCK	In	STD_LOGIC	Clock signal
MODE	In	STD_LOGIC	Denotes CORDIC operation mode: Vectoring = ‘1 Rotation = ‘0’
RESET	In	STD_LOGIC	Sets CORDIC back to wait state
START	In	STD_LOGIC	Sets CORDIC to start state (begins iterations)
iteration_select	In	STD_LOGIC_VECTOR 4 bits	Switch input, selects output data iteration (0-15) to display
x_0, y_0, z_0	In	STD_LOGIC_VECTOR 16 bits	Initial input data, used for first iteration ALU input Stored into position 0 of storage arrays
state_out	Out	STD_LOGIC_VECTOR 4 bits	Outputs current CORDIC state to LEDs (same as top level)
x_i, y_i, z_i	Out	STD_LOGIC_VECTOR 16 bits	Outputs current iteration data to be selected at top level once CORDIC is done, controlled by iteration_select input

Table 2: CORDIC Processor Component I/O Signal Descriptions

6.1.3 CORDIC ALU

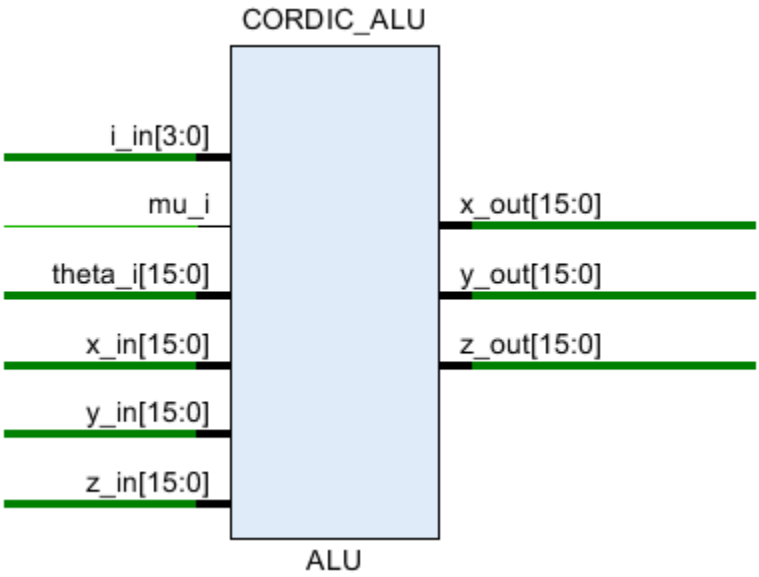


Figure 11: ALU Component Block Diagram

Signal Name	Mode	Data Type	Functionality
i_in	In	STD_LOGIC_VECTOR 4 bits	Current iteration (used for δ_i parameter in calculation)
mu_i	In	STD_LOGIC	Indicates direction of rotation: 1 = counterclockwise rotation 0 = clockwise rotation
theta_i	In	STD_LOGIC_VECTOR 16 bits	Angle of rotation for current iteration, input from LUT, used for z value calculation
x_in, y_in, z_in	In	STD_LOGIC_VECTOR 16 bits	Input values for calculation
x_out, y_out, z_out	Out	STD_LOGIC_VECTOR 16 bits	Output values after calculation is finished, corresponds to x_{i+1}, y_{i+1}, z_{i+1}

Table 3: ALU Component I/O Signal Descriptions

6.1.4 CORDIC Lookup Table (LUT)

The lookup table is a simple data table within the CORDIC processor that stores constant angles to be used by the ALU.

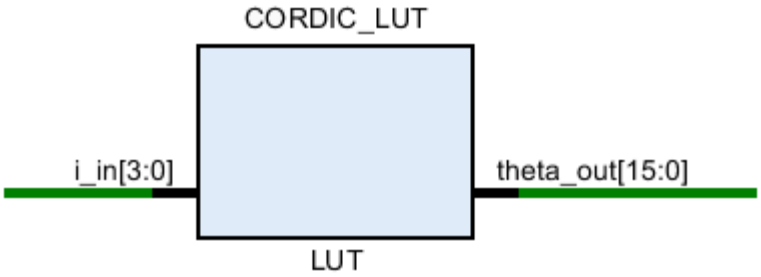


Figure 12: LUT Component Block Diagram

Signal Name	Mode	Data Type	Functionality
i_in	In	STD_LOGIC_VECTOR 4 bits	Current iteration (same as ALU)
theta_out	Out	STD_LOGIC_VECTOR 16 bits	Angle data corresponding to current iteration, fed to ALU input

Table 4: LUT Component I/O Signal Descriptions

The format for the values can be seen in Figure 13, along with the actual data held in the table shown in Figure 14 below.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}

Figure 13: LUT Data Formatting for Theta

Iteration	θ (radians)	θ (binary)	θ (Hexadecimal)
0	0.7854	0011 0010 0100 0100	3244
1	0.4636	0001 1101 1010 1100	1DAC
2	0.245	0000 1111 1010 1110	0FAE
3	0.1244	0000 0111 1111 0110	07F6
4	0.0624	0000 0011 1111 1110	03FE
5	0.0312	0000 0001 1111 1111	0200
6	0.0156	0000 0000 1111 1111	0100
7	0.0078	0000 0000 0111 1111	0080
8	0.0039	0000 0000 0100 0000	0040
9	0.002	0000 0000 0010 0000	0020
10	0.00098	0000 0000 0001 0000	0010
11	0.00049	0000 0000 0000 1000	0008
12	0.00024	0000 0000 0000 0100	0004
13	0.00012	0000 0000 0000 0010	0002
14	0.000061	0000 0000 0000 0001	0001
15	0.000031	0000 0000 0000 0001	0001

Figure 14: LUT Data

6.1.5 CORDIC Storage

The storage block is a component within the CORDIC processor used to store the data of each iteration output by the processors ALU. The storage unit inputs data from the ALU (write mode) while the CORDIC is in start/iterate mode (done = 0), and outputs data (read mode) to the hex display (through the CORDIC processor) when the CORDIC is in display mode (done = 1). Note that the storage block writes data during the negative clock edge to avoid timing issues.

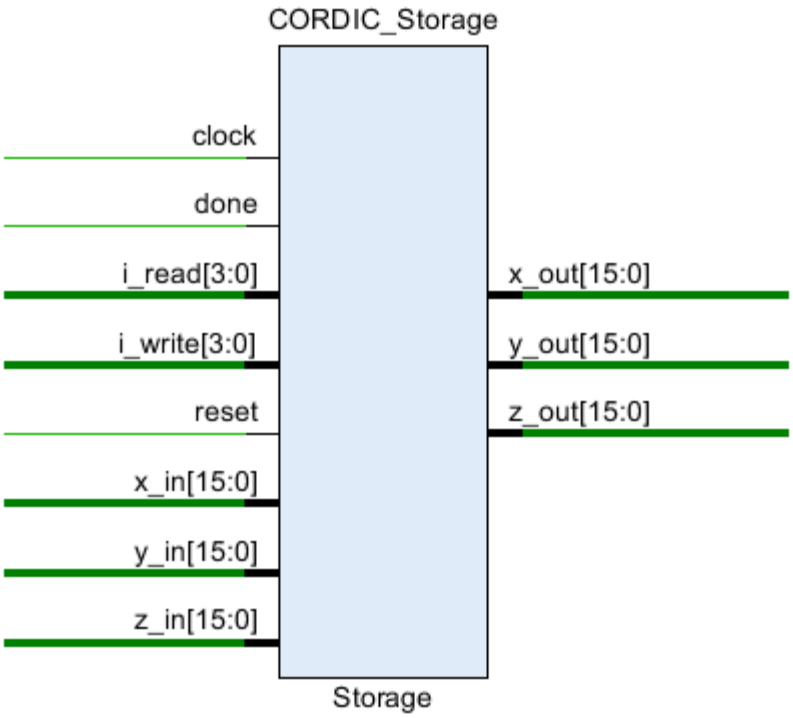


Figure 15: Storage Component Block Diagram

Signal Name	Mode	Data Type	Functionality
clock	In	STD_LOGIC	Clock signal
done	In	STD_LOGIC	Indicates if CORDIC processor is done, if not done then storage is write only, if done then storage is read only
i_read	In	STD_LOGIC_VECTOR 4 bits	Iteration of arrays to read from (0-15)
i_write	In	STD_LOGIC_VECTOR 4 bits	Iteration of arrays to write to (0-15)
reset	In	STD_LOGIC	Used to clear data in arrays (set to 0), activated when RESET button is pushed on board
x_in, y_in, z_in	In	STD_LOGIC_VECTOR 16 bits	Input data to write to array, index location controlled by i_write
x_out, y_out, z_out	Out	STD_LOGIC_VECTOR 16 bits	Output data read from array, index location controlled by i_read

Table 5: Storage Component I/O Signal Descriptions

6.1.6 HEX Display

The hex display driver is a pre supplied unit that converts input data (hex signals) into the required signals needed to drive the 7-segment display (an, sseg).

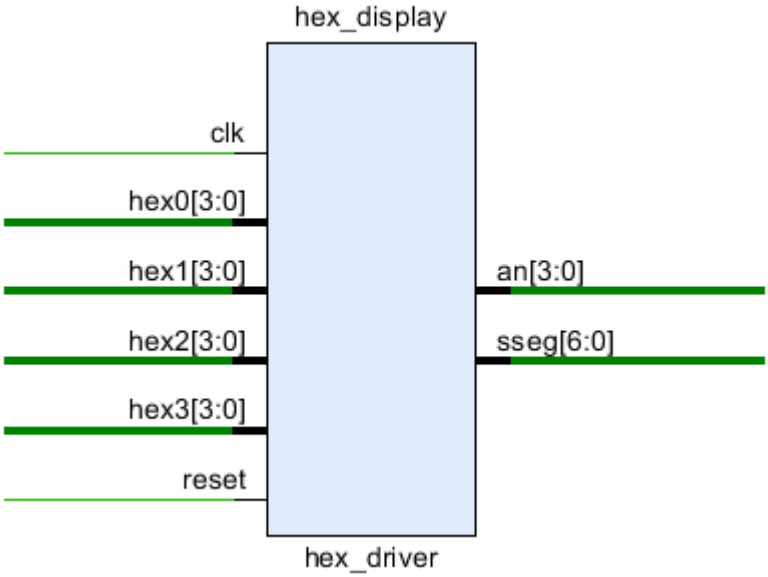


Figure 16: HEX Display Driver Component Block Diagram

Signal Name	Mode	Data Type	Functionality
clock	In	STD_LOGIC	Clock signal
done	In	STD_LOGIC	Indicates if CORDIC processor is done, if not done then storage is write only, if done then storage is read only
i_read	In	STD_LOGIC_VECTOR 4 bits	Iteration of arrays to read from (0-15)
i_write	In	STD_LOGIC_VECTOR 4 bits	Iteration of arrays to write to (0-15)
reset	In	STD_LOGIC	Used to clear data in arrays (set to 0), activated when RESET button is pushed on board
x_in, y_in, z_in	In	STD_LOGIC_VECTOR 16 bits	Input data to write to array, index location controlled by i_write
x_out, y_out, z_out	Out	STD_LOGIC_VECTOR 16 bits	Output data read from array, index location controlled by i_read

Table 6: Hex Display Driver Component I/O Signal Descriptions

6.1.7 Start Debounce & Reset Debounce

The project includes 2 debounce components in order to ensure noiseless operation of the start and reset pushbuttons. Like the hex driver block, the code for this component was supplied. The reset signal is not needed for this design and is therefore simply grounded to ‘0’.

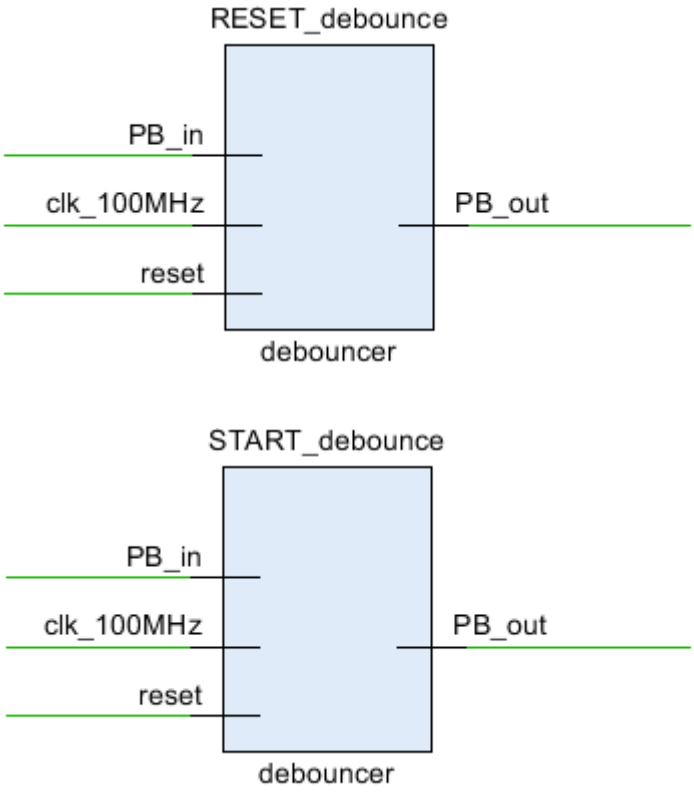


Figure 17: Debounce Components Block Diagram

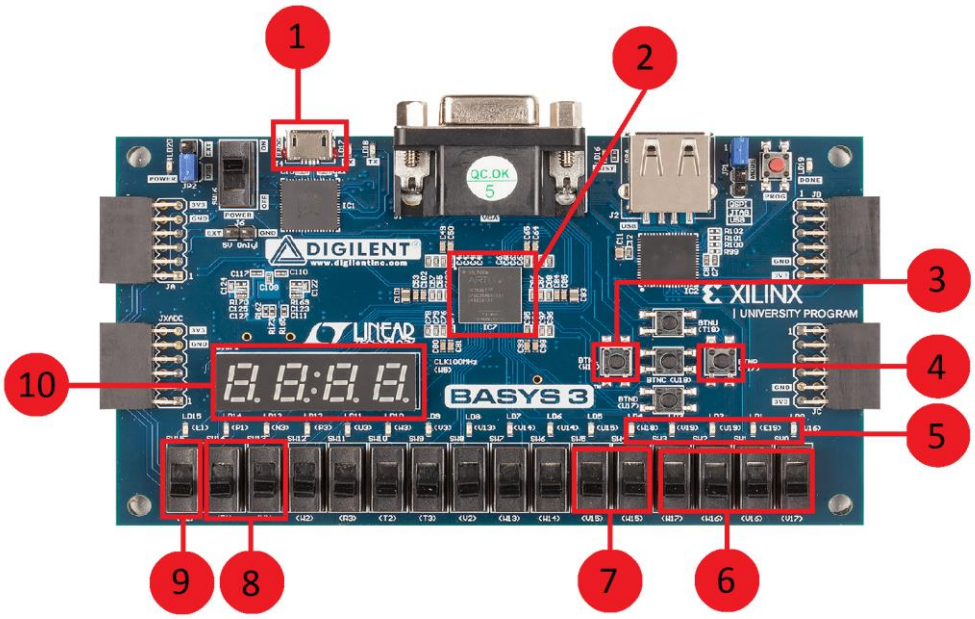
Signal Name	Mode	Data Type	Functionality
clk_100MHz	In	STD_LOGIC	Clock signal
PB_in	In	STD_LOGIC	Non-debounced input bit, read from pushbuttons (START or RESET)
reset	In	STD_LOGIC	Sets debouncer to initial state: wait_for_push Unused (set to '0')
PB_out	In	STD_LOGIC	Debounced output bit, sent to CORDIC processor

Table 7: Debounce Components I/O Signal Descriptions

6.2 Hardware Implementation

The Basys 3 board, with the Artix-7 35T FPGA built in, was chosen as the hardware platform for the project. The board comes equipped with 16 programmable DIP switches, 5 pushbuttons, a 4 digit 7-segment hex display, and various LED's. The finished project code was downloaded onto the Basys 3 board through the Digilent USB-JTAG port. The VHDL is converted into usable

code for the FPGA through the Xilinx Vivado software synthesis, implementation, bitstream generation, and programming to the target board. The final implemented design utilizes two pushbuttons, four LED's, nine switches, and the 7-segment display.



Tag:	Component Description	Tag:	Component Description
1:	Shared UART/JTAG USB Port	6:	Switches SW0-SW3: Iteration select
2:	Artix-7 35T FPGA	7:	Switches SW4-SW5: Output variable select
3:	"RESET" Pushbutton (W19)	8:	Switches SW13-SW14: Input Data Selection
4:	"START" Pushbutton (T17)	9:	Switch SW15: Vectoring/Rotation mode select
5:	LEDs LD0-LD3: "Wait" "Start" "Iterate" "Display" Indication	10:	Four digit 7-segment display

Figure 18: Relevant Basys 3 Board Architecture

Pushbutton T17 is used to commence the iteration process, acting as a start button, while button W19 serves as a reset to return the CORDIC to it's initial wait mode. LED's LD0, LD1, LD2 and LD3 is used for indications of states "wait", "start", "iterate" and "display" respectively. The "wait" state indicates to the user that the program is ready to begin operation, while the "display" state indicates that the CORDIC operation is complete.

Switches located on the left of the Basys 3 board serve as input data selection (as seen on the project section of the ECE 441 course website). Switch SW15 is used to select CORDIC rotation or vectoring mode, with a 1 setting corresponding to vectoring mode, and a 0 setting corresponding to rotation mode. Switches SW13 and SW14 correspond to the selection data set, and work in conjunction with SW15 to select sets of predetermined variables x_0, y_0, z_0.

Using both the mode selected by SW15 and value of switches SW13 and SW14, the CORDIC may select each of the eight combinations of data required to test the CORDIC algorithm [6].

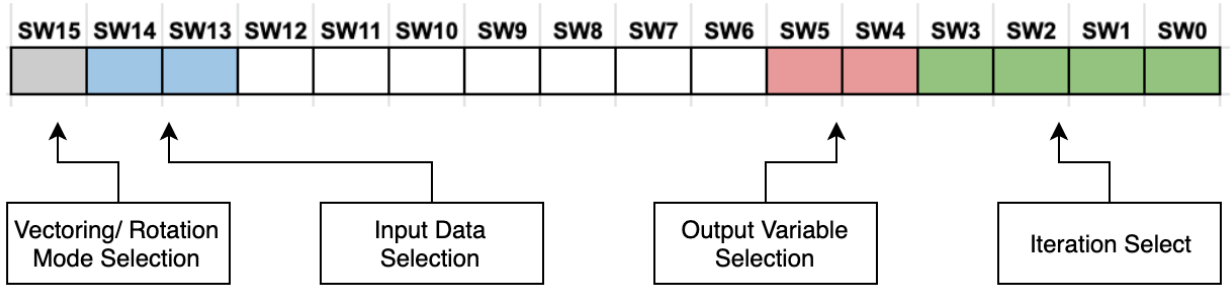


Figure 19: Basys 3 Switch Design

Located on the right side of the Basys 3 switch bank are functions that allow for selection of output data. SW4 and SW5 create a 2-bit binary vector that determines which output, x_i, y_i, or z_i will be shown on the 7-segment display (based on binary values “00”, “01” and “10”). Switches SW0-3 create a 4-bit vector in order to display each iteration from start (iteration 0) to finish (iteration 15). The 7-segment display will display values in hexadecimal format based on data stored in the storage block, either of variable x_i, y_i, z_i and of iteration 0-15.

In the initial wait state, “mode_wait”, the 7-segment display will default to output zeros on each of the 4 positions of the display. The rightmost LED (LD0) will be illuminated to verify to the user that the memory contains no previous data and that the system is ready to perform its iterative process. The system will remain in state “mode_wait” until the start button is pressed. Modifying any switches on the Basys 3 board will not have an effect at this point on the output of the 7-segment display.

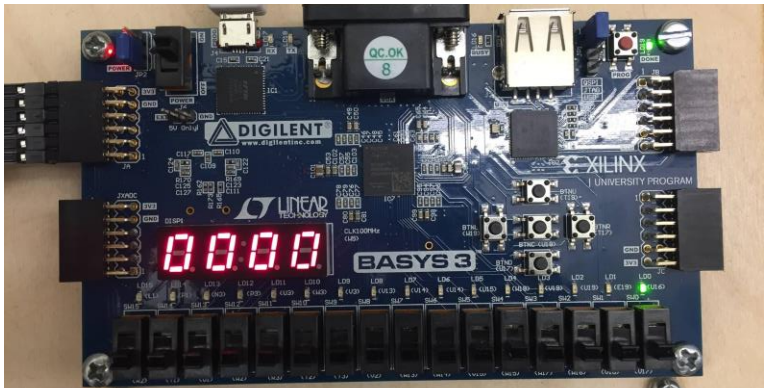


Figure 20: Basys 3 Board in Wait State, LD0 Illuminated

When the system receives a signal “START” from the start button, the CORDIC algorithm will move through states “mode_start”, “mode_iterate” and “mode display”, based on logic in the controller. LED’s LD1 and LD2 will illuminate for states “mode_start” and “mode_iterate” respectively, although this will be done at a rate too fast to be seen by the human eye under

normal operation. Once iterations are complete, LD3 will illuminate to notify the user that the CORDIC iterations are done. If desired, the clock cycle can be manually controlled to view the illumination of LD1 and LD2.

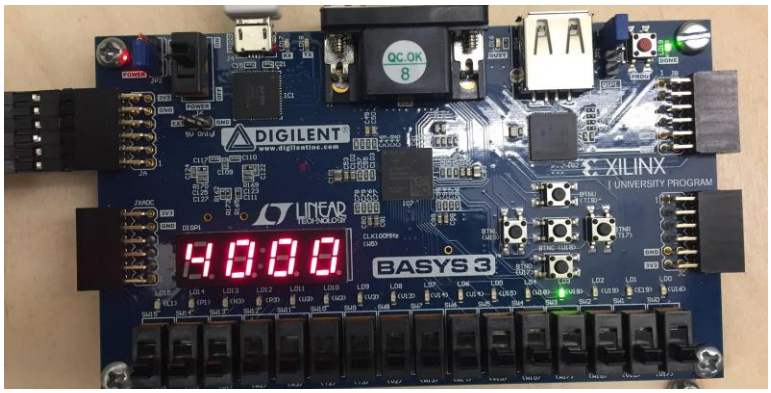


Figure 21: Basys 3 Board in Finished State, Iteration 0, LD3 Illuminated

Once in “mode_display” the controller will read in from switches SW0-SW5 (user input) in order to select data (x_i, y_i, or z_i) to send to the display. As seen in Figures 21 and 22, while in “mode_display” the outputted value of the x-parameter has been selected to display iterations 0 and 15 respectively.

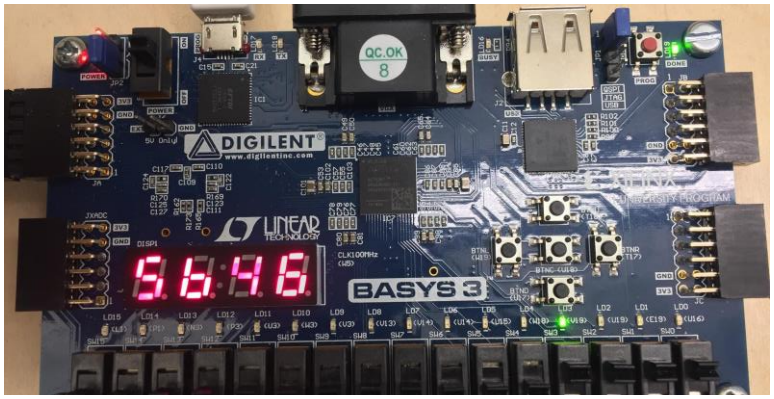


Figure 22: Basys 3 Board in Finished State, Iteration 15, LD3 illuminated

7.0 Hardware Performance

For some iterations, a small discrepancy was noticed between the provided test data [7] and simulation results. This was normally in the magnitude of a 1-2 bit difference. This discrepancy was noticed while testing iteration 15 of selection 2 data while in vectoring mode.

Vectoring Mode, Selection 2	
Provided Test Data [7]	**Simulation Data**
x: 4A86	x: 4A87
y: 0000	y: FFFF
z: 3244	z: 3244

Figure 23: Discrepancy between Provided Test Data and Simulation Results, Iteration 15.

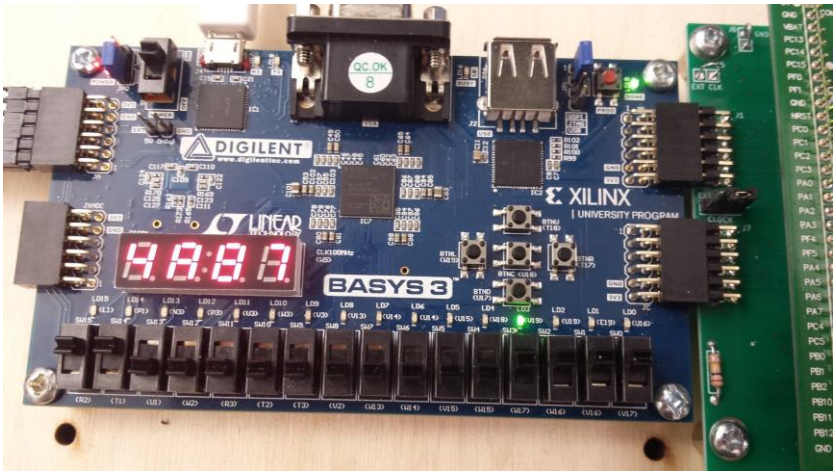


Figure 24: Vectoring Mode, Selection 2 Input Data, X = 4A87

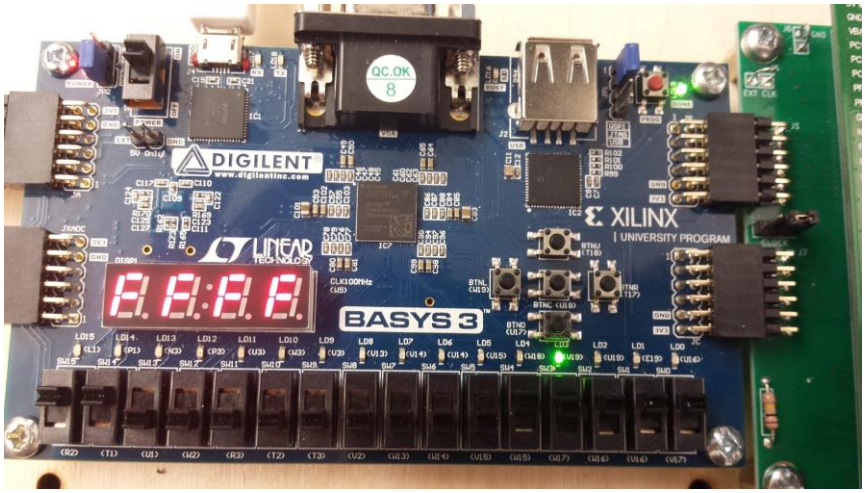


Figure 25: Vectoring Mode, Selection 2 Input Data, Y =FFFF

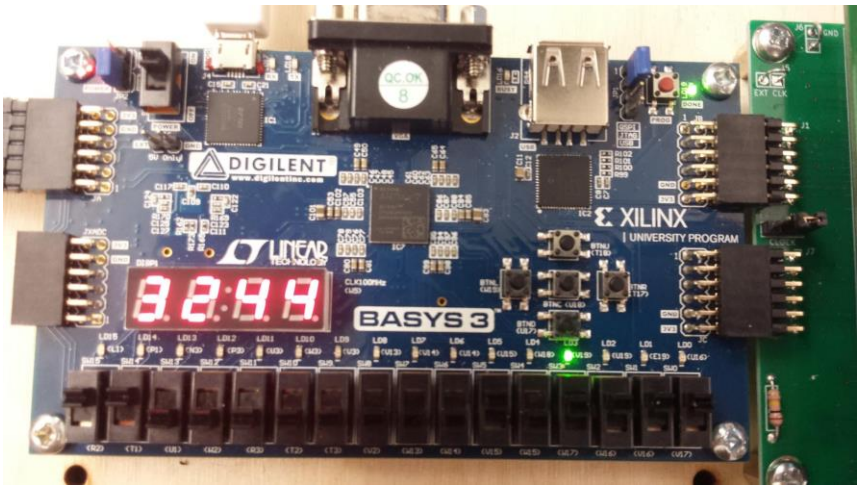


Figure 26: Vectoring Mode, Selection 2 Input Data, Z = 3244

This discrepancy can be explained from the different methods used to calculate each data set. The given test data was most likely calculated in a program such as Matlab, where complex computations such as multiplication can be directly calculated on a computer using higher bit calculations (e.g. 32 or 64). The Basys 3 board must utilize addition and shifting operations calculated on two's complement data and is limited to 16 bit data. Therefore, while comparing the performance of the Basys 3 CORDIC to data generated by a program such as Matlab, Matlab will provide a more precise answer than the CORDIC algorithm. That being said, slight variances in data are negligible since a variance of 1-2 bits is in the range of a 2^{-14} to 2^{-15} error depending on the variables X, Y or Z [2].

Taking into account the lower level of complexity the CORDIC needs to perform these computations, as well as the negligible time taken to compute through 16 iterations, it was determined that the CORDIC hardware implementation performed well compared to other computational methods.

8.0 Simulation and Timing Diagrams

An ALU testbench testing the first 2 iterations of selection 0 data (rotation mode, mode = 0) can be seen below in Figure 27. The first iteration runs a clockwise vector rotation on (4000, 0000, 2183), resulting in the input for the next iteration (4000, 4000, ef3f). The next iteration then performs a counterclockwise rotation, resulting in the data seen in iteration 2 (6000, 2000, 0CEB).

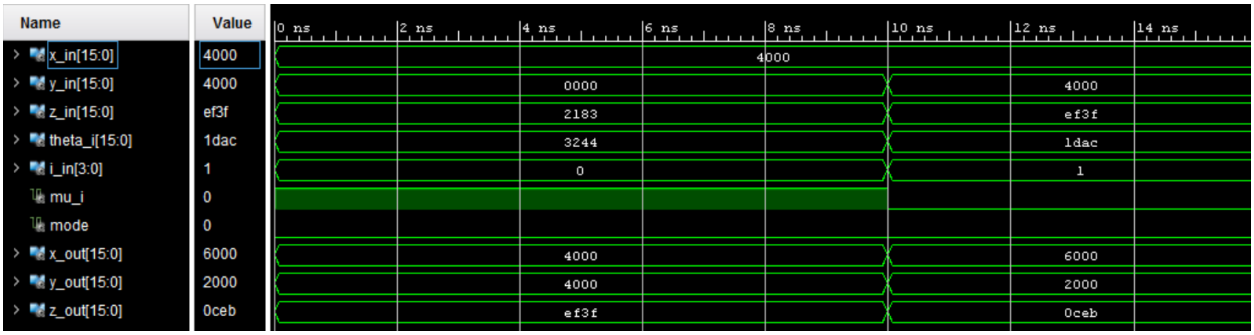


Figure 27: ALU Testbench

Functionality of the storage block can be shown in the testbench simulation below (Figure X). The testbench first writes data into the first 3 positions of the storage arrays (0,1, and 2). Afterwards the block switches to a ‘done’ mode, corresponding to ‘display mode’ for the CORDIC, and reads from position 1 (resulting in the 0xBBBB data on the output). Finally, the testbench resets, forcing the arrays and output data to 0x0000.

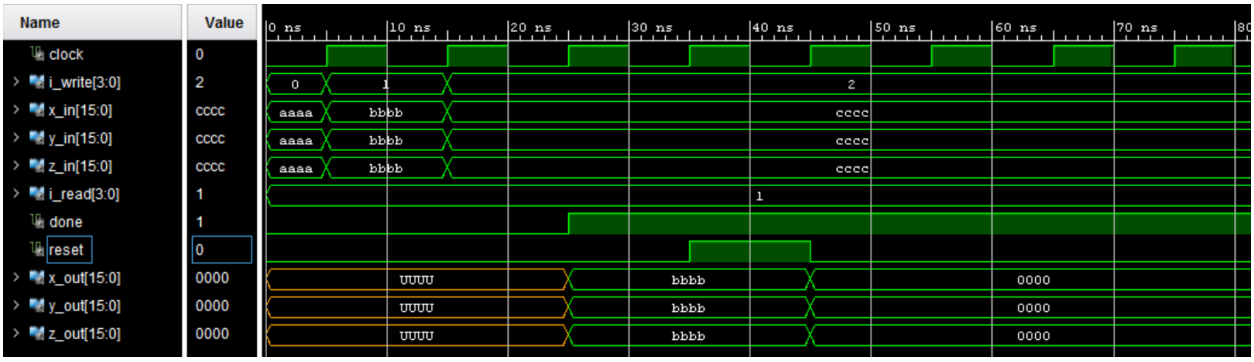


Figure 28: Storage Testbench

A simple simulation covering the first four angles of the lookup table can be seen in Figure X below. The values correspond to angles of 0.78540, 0.46360, 0.24500, and 0.12440 radians respectively.

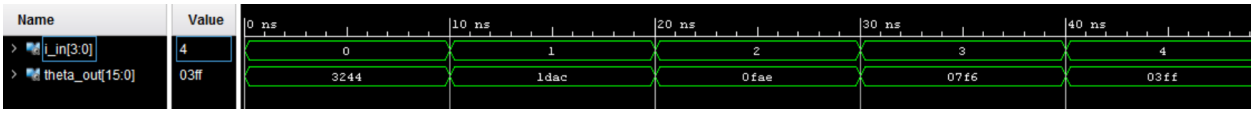


Figure 29: LUT Testbench

The testbench for the debounce code (Figure 30) was run with COUNT_MAX = 10 for the sake of the simulation time (rather than ~ms long time range). The input signal is only passed through to the output after being held constant for enough time.

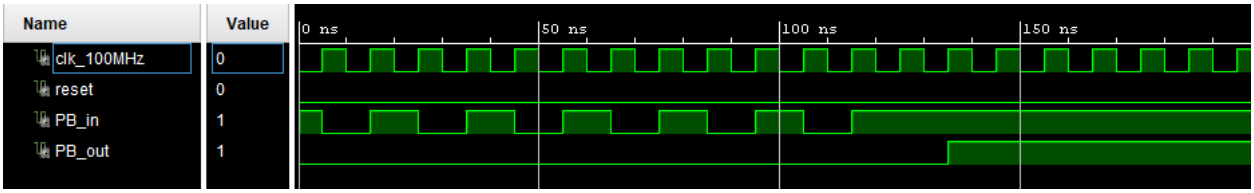


Figure 30: Debounce Testbench

A testbench can be seen covering operation of the CORDIC calculating in rotation mode, with initial data being set according to selection 0's data on the course website (x = 4000, y = 0000, z = 2183). The inital vector corresponds to a 30 degree counterclockwise rotation on the vector (0.5, 0), while the final vector confirms proper operation with a z value of 0 (vector fully rotated). Both the initial wait mode, start mode, and iterate mode can be seen in Figure 31 (part 1), while the display mode and wait mode (after reset is asserted again) can be seen in Figure 32 (part 2).

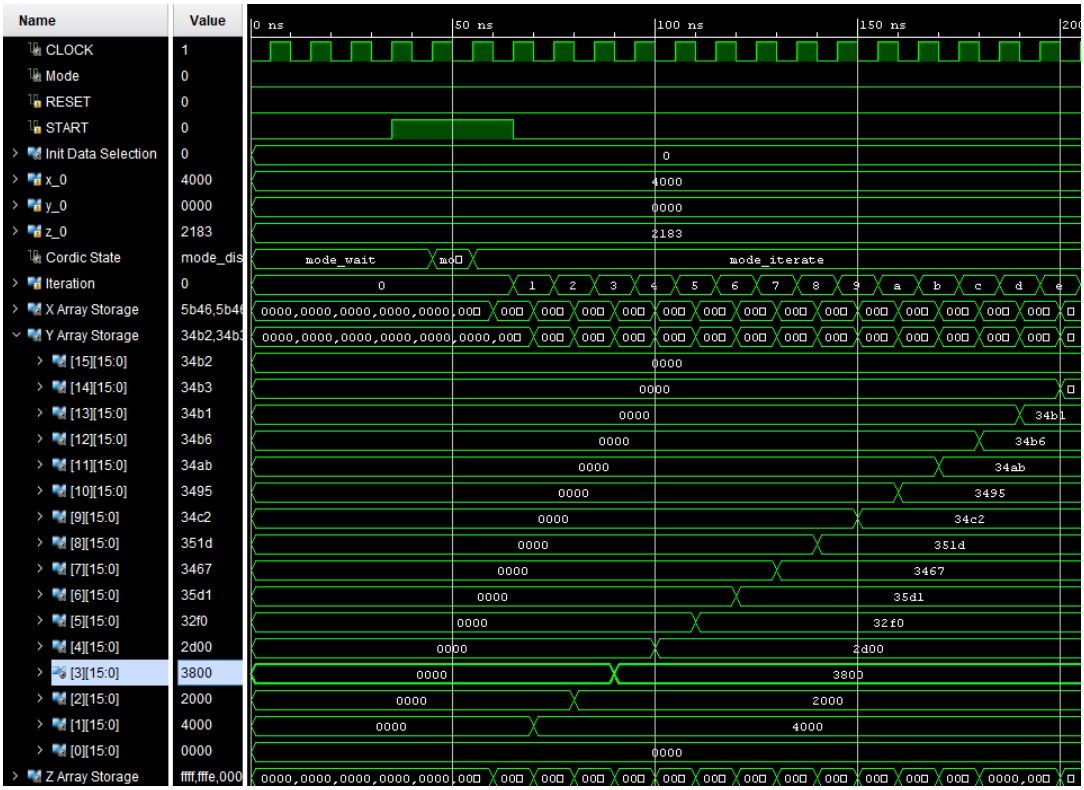


Figure 31: Top (CORDIC) Testbench Pt. 1

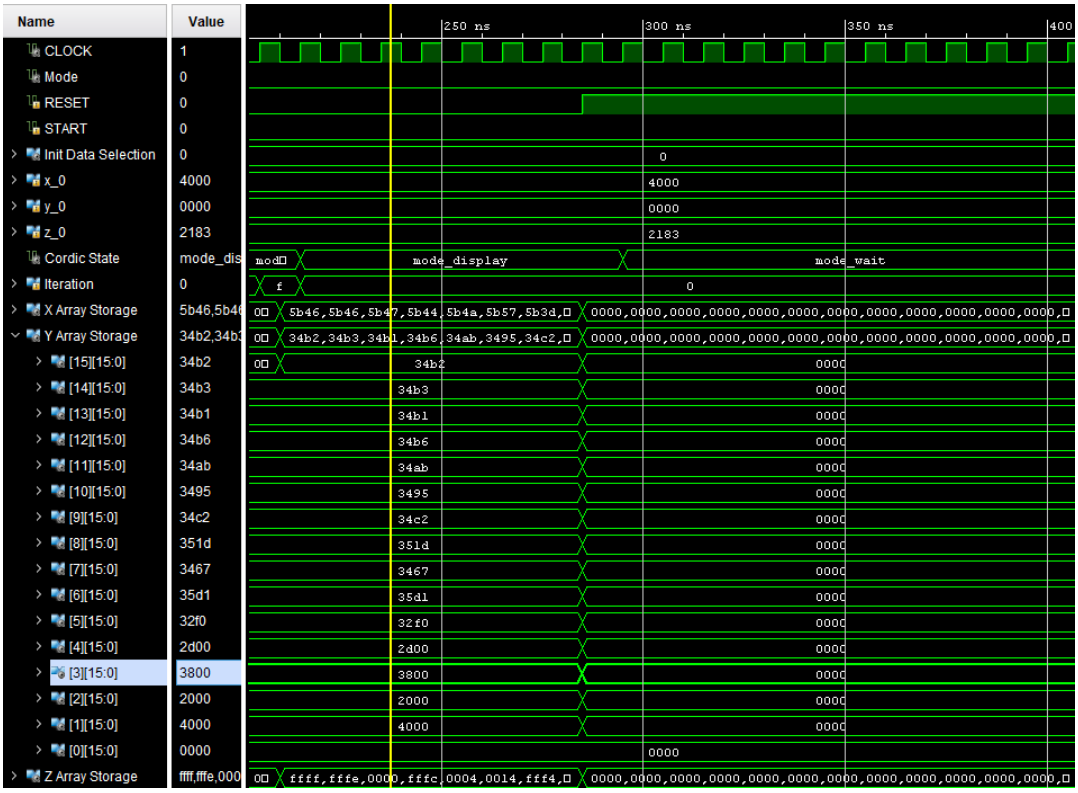


Figure 32: Top (CORDIC) Testbench Pt. 2

Simple timing diagrams can be seen below, representing the expected responses to different input combinations (pushbutton timings). The first Figure shows a full computation runthrough of the finite state machine, going from wait to start mode as the start button is pressed, then switching to iterate mode the next clock, then to display once the iteration count hits 15. In this Figure, a reset is asserted during display mode, forcing the processor back to it's default wait mode. In the second Figure, the reset is asserted much earlier (during the iteration phase), again forcing the CORDIC state to reset to the wait mode rather than completing to the display mode. During the third Figure, both buttons are pushed at the same time. Since the reset button holds priority, the CORDIC state does not exit it's wait mode.

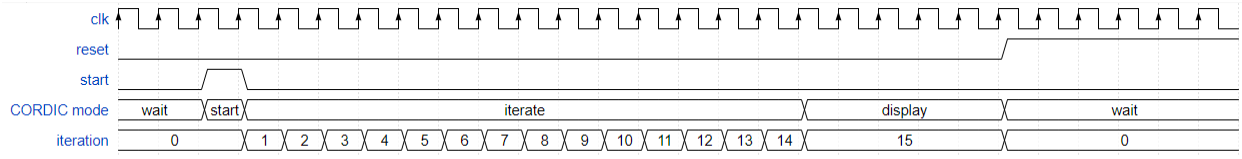


Figure 33: Reset-During-Display CORDIC FSM Timing Diagram

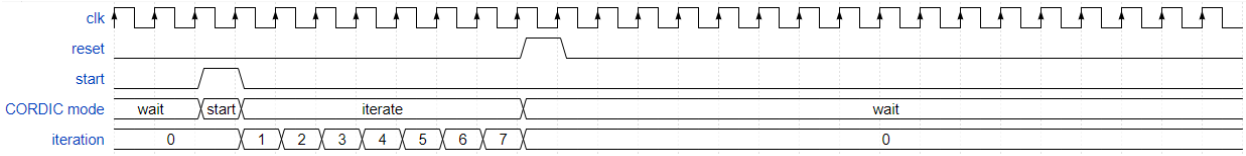


Figure 34: Reset-During-Iterate CORDIC FSM Timing Diagram

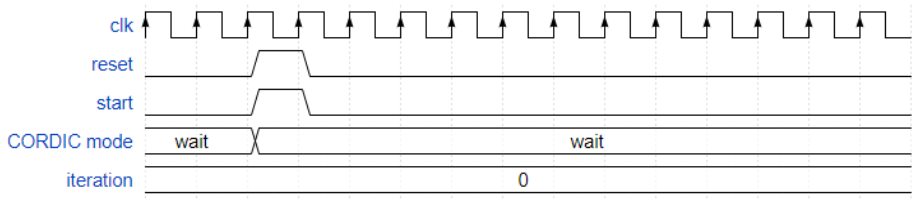


Figure 35: Reset-During-Start CORDIC FSM Timing Diagram

9.0 Conclusions

The final CORDIC Processor Design and Implementation was successful in performing iterative calculations on various 16-bit input data sets. Project requirements were met by implementing an iterative CORDIC algorithm, which was able to perform rotation and vectoring calculations on said 16-bit input data. This was achieved through simple shifting and addition operations. The data being processed was formatted to represent specific data ranges ($0 \leq x \leq 1$, $0 \leq y \leq 1$, $0 \leq z \leq 2$ radians). Final CORDIC results were able to be displayed on a 7-Segment Hex Display.

The CORDIC design was implemented by creating various VHDL components in Vivado. These components were troubleshooted by using testbench files. Resulting timing diagrams were then analyzed to ensure the functionality of each code segment.

Once verified, the VHDL implementation was programmed onto the Basys 3 board. The final board design used start and reset inputs (debounced pushbuttons) for state control, and various switches for input data and output display data selection. LEDs were also utilized as a state indicator for the user. The displayed CORDIC results were compared with the provided test results generated in Matlab, and the functionality of the CORDIC algorithm was verified.

It was determined that the CORDIC algorithm in conjunction with the hardware performed adequately well when compared to the Matlab results. Few minor variances in data were observed, which were later proved to be negligible. In conclusion, the CORDIC algorithm was successfully implemented in order to provide a fast and simple way to compute vector rotation through the use of the Basys 3 board.

10.0 References

[1] Tomar, R. Singh, P Raj, K. (2013). *A Review of CORDIC Algorithms and Architectures with Applications for Efficient Designing.* International Journal of Scientific & Engineering Research, Vol 4. Issue 8. [Accessed 30 Mar. 2019].

[2] Gebali, F. (2019). *The Circular CORDIC Algorithm*. [ebook] Available at: https://www.ece.uvic.ca/~fayez/courses/441/project/cordic_circ.pdf [Accessed 30 Mar. 2019].

[3] V. Padma and P. Sudhakara Reddy, *CORDIC Based DFT on FPGA for DSP Applications*. Available at: https://www.ijert.org/research/cordic-based-dft-on-fpga-for-dsp-applications-IJERTV3IS071310.pdf [Accessed 30 Mar. 2019].

[4] Fayez, G. *CORDIC Hardware PDF*. [ebook] Available at: https://www.ece.uvic.ca/~fayez/courses/441/project/cordic_circ_hw.pdf [Accessed 28 Feb. 2019].

[5] Reference.digilentinc.com. (2019). [online] Available at: https://reference.digilentinc.com/_media/reference/pmod/pmodkypd/pmodkypd_rm.pdf?fbclid=IwAR3YPiR1Q0GlisED9qh02Js9bZh5AN7Z3J4qb2cdeP7imwTHeVsqH0zrXks [Accessed 1 Mar. 2019].

[6] https://www.ece.uvic.ca/~fayez/courses/441/project/project.html website to reference the tests in the hardware section (Needs to be turned into IEEE)

[7] F. Gebali, Datasheet: “Circular Mode Vectoring, Selection 2” ECE 441, University of Victoria, Victoria, BC., Mar., 2019. Available: https://www.ece.uvic.ca/~fayez/courses/441/project/cordic_circular/cordic_circ_vec1.txt [Accessed: 30- Mar- 2019].

[8] *Nptel.ac.in*, 2019. [Online]. Available: https://nptel.ac.in/courses/Webcourse-contents/IIT%20Kharagpur/Embedded%20systems/Pdf/Lesson-40.pdf. [Accessed: 30- Mar- 2019].

[9] F. Gebali, Class Lecture, Topic: “Design for Testability (DFT).” ECE 441, University of Victoria, Victoria, BC., Mar., 2019.

Appendixes

Appendix A: VHDL Code

Top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Top is
    port(
        START_IN, RESET_IN, MODE_IN: in STD_LOGIC;
        CLOCK_IN: in STD_LOGIC;
        ITERATION_SELECT_IN: in STD_LOGIC_VECTOR(3 downto 0);
        DATA_SELECT_IN: in STD_LOGIC_VECTOR(1 downto 0);
        INIT_DATA_SELECTION: in STD_LOGIC_VECTOR(1 downto 0);
        ANODES: out STD_LOGIC_VECTOR (3 downto 0);
        CATHODES: out STD_LOGIC_VECTOR (6 downto 0);
        STATE_OUT: out STD_LOGIC_VECTOR(3 downto 0));
end Top;

architecture Behavioral of Top is
    component CORDIC is
        port(
            START, MODE, RESET: in STD_LOGIC;
            CLOCK: in STD_LOGIC;
            x_0, y_0, z_0: in STD_LOGIC_VECTOR(15 downto 0);
            iteration_select: in STD_LOGIC_VECTOR(3 downto 0);
            x_i, y_i, z_i: out STD_LOGIC_VECTOR(15 downto 0);
            state_out: out STD_LOGIC_VECTOR(3 downto 0));
    end component;

    component hex_driver is
        port(
            clk, reset: in STD_LOGIC;
            hex3, hex2, hex1, hex0: in STD_LOGIC_VECTOR(3 downto 0);
            an: out STD_LOGIC_VECTOR(3 downto 0);
            sseg: out STD_LOGIC_VECTOR(6 downto 0)
        );
    end component;

    component debouncer is
        port(
            clk_100MHz: in STD_LOGIC;
            reset: in STD_LOGIC;
            PB_in: in STD_LOGIC;    -- the input PB that is bouncy
            PB_out: out STD_LOGIC);    -- the de-bounced output
    end component;

    -- signals
    signal reset_START, reset_RESET: STD_LOGIC;
    signal d_in_HEX: STD_LOGIC_VECTOR(15 downto 0);
    signal START_debounced: STD_LOGIC;
    signal RESET_debounced: STD_LOGIC;
    signal CORDIC_x_out, CORDIC_y_out, CORDIC_z_out: STD_LOGIC_VECTOR(15 downto 0);
    signal CORDIC_state_out: STD_LOGIC_VECTOR(3 downto 0);

    signal x_init: STD_LOGIC_VECTOR(15 downto 0);
    signal y_init: STD_LOGIC_VECTOR(15 downto 0);
    signal z_init: STD_LOGIC_VECTOR(15 downto 0);

begin
    CORDIC_processor: CORDIC port map(START => START_debounced, MODE => MODE_IN, RESET => RESET_debounced, CLOCK => CLOCK_IN,
            x_0 => x_init, y_0 => y_init, z_0 => z_init, iteration_select => ITERATION_SELECT_IN, x_i => CORDIC_x_out,
            y_i => CORDIC_y_out, z_i => CORDIC_z_out, state_out => CORDIC_state_out);
    hex_display: hex_driver port map(clk => CLOCK_IN, reset => '0', hex3 => d_in_HEX(15 downto 12), hex2 => d_in_HEX(11 downto 8),
            hex1 => d_in_HEX(7 downto 4), hex0 => d_in_HEX(3 downto 0), an => ANODES, sseg => CATHODES);
    START_debounce: debouncer port map(clk_100MHz => CLOCK_IN, reset => '0', PB_in => START_IN, PB_out => START_debounced);
    RESET_debounce: debouncer port map(clk_100MHz => CLOCK_IN, reset => '0', PB_in => RESET_IN, PB_out => RESET_debounced);
```

```
    STATE_OUT <= CORDIC_state_out;  -- feed CORDIC mode to LEDs

    process(data_select_in, CORDIC_x_out, CORDIC_y_out, CORDIC_z_out, RESET_debounced, CORDIC_state_out)
    begin
        case DATA_SELECT_IN is          -- display selected X/Y/Z data to display
            when "00" =>
                d_in_HEX <= CORDIC_x_out;
            when "01" =>
                d_in_HEX <= CORDIC_y_out;
            when "10" =>
                d_in_HEX <= CORDIC_z_out;
            when others =>
                d_in_HEX <= x"0000";
        end case;
    end process;

    process(MODE_IN, INIT_DATA_SELECTION)
    begin                               -- select inital data for x0, y0, z0
        if MODE_IN = '1' then   -- vectoring (y -> 0)
            case INIT_DATA_SELECTION is
                when "00" =>
                    x_init <= x"0000";  -- x = 0
                    y_init <= x"4000";  -- y = 1/2
                    z_init <= x"0000";  -- z = 0
                when "01" =>
                    x_init <= x"2000";  -- x = 1/4
                    y_init <= x"376D";  -- y = sqrt(3)/4
                    z_init <= x"0000";  -- z = 0
                when "10" =>
                    x_init <= x"2000";  -- x = 1/4
                    y_init <= x"2000";  -- y = 1/4
                    z_init <= x"0000";  -- z = 0
                when "11" =>
                    x_init <= x"376D";  -- x = sqrt(3)/4
                    y_init <= x"2000";  -- y = 1/4
                    z_init <= x"0000";  -- z = 0
                when others => NULL;
            end case;
        else                    -- rotation
            case INIT_DATA_SELECTION is
                when "00" =>
                    x_init <= x"4000";  -- x = 1/2
                    y_init <= x"0000";  -- y = 0
                    z_init <= x"2183";  -- z = 30 degrees (CCW)
                when "01" =>
                    x_init <= x"376D";  -- x = sqrt(3)/4
                    y_init <= x"2000";  -- y = 1/4
                    z_init <= x"10C1";  -- z = 15 degrees (CCW)
                when "10" =>
                    x_init <= x"2000";  -- x = 1/4
                    y_init <= x"376D";  -- y = sqrt(3)/4
                    z_init <= x"2183";  -- z = 30 degrees (CCW)
                when "11" =>
                    x_init <= x"2000";  -- x = 1/4
                    y_init <= x"376D";  -- y = sqrt(3)/4
                    z_init <= x"2183";  -- z = 30 degrees (CW)
                when others => NULL;
            end case;
        end if;
    end process;
end Behavioral;
```

CORDIC.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CORDIC is
    port(
        START, MODE, RESET: in STD_LOGIC;
        CLOCK: in STD_LOGIC;
        x_0, y_0, z_0: in STD_LOGIC_VECTOR(15 downto 0);
        iteration_select: in STD_LOGIC_VECTOR(3 downto 0);
        x_i, y_i, z_i: out STD_LOGIC_VECTOR(15 downto 0);
        state_out: out STD_LOGIC_VECTOR(3 downto 0));
end CORDIC;

architecture behavioral of CORDIC is
    component ALU is
        port(
            x_in, y_in, z_in, theta_i: in STD_LOGIC_VECTOR(15 downto 0);
            i_in: in STD_LOGIC_VECTOR(3 downto 0);
            mu_i: in STD_LOGIC;
            x_out, y_out, z_out: out STD_LOGIC_VECTOR(15 downto 0));
    end component;

    component LUT is
        port(
            i_in: in STD_LOGIC_VECTOR(3 downto 0);
            theta_out: out STD_LOGIC_VECTOR(15 downto 0));
    end component;

    component Storage is
        port(
            x_in, y_in, z_in: in STD_LOGIC_VECTOR(15 downto 0);
            done: in STD_LOGIC; -- if done = 0 => write mode, if done = 1 => read mode
            clock: in STD_LOGIC;
            i_read: in STD_LOGIC_VECTOR(3 downto 0);
            i_write: in STD_LOGIC_VECTOR(3 downto 0);
            reset: in STD_LOGIC;
            x_out, y_out, z_out: out STD_LOGIC_VECTOR(15 downto 0));
    end component;

    -- signals
    signal x_in_ALU, y_in_ALU, z_in_ALU: STD_LOGIC_VECTOR(15 downto 0);
    signal i_in_ALU: STD_LOGIC_VECTOR(3 downto 0) := x"0";
    signal i_in_Storage: STD_LOGIC_VECTOR(3 downto 0);
    signal mu_i_ALU: STD_LOGIC;
    signal x_out_ALU, y_out_ALU, z_out_ALU: STD_LOGIC_VECTOR(15 downto 0);

    signal theta_out_LUT: STD_LOGIC_VECTOR(15 downto 0);

    signal x_in_Storage, y_in_Storage, z_in_Storage: STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
    signal i_read_Storage: STD_LOGIC_VECTOR(3 downto 0) := x"F";
    signal reset_Storage: STD_LOGIC;
    signal x_out_Storage, y_out_Storage, z_out_Storage: STD_LOGIC_VECTOR(15 downto 0);
    signal done_Storage: STD_LOGIC := '0';

    -- 4 modes for finite state machine
    type state_modes is (mode_wait, mode_start, mode_iterate, mode_display);
    signal state: state_modes := mode_wait;

begin
    CORDIC_ALU: ALU port map(x_in => x_in_ALU, y_in => y_in_ALU, z_in => z_in_ALU, theta_i => theta_out_LUT,
            i_in => i_in_ALU, mu_i => mu_i_ALU, x_out => x_out_ALU, y_out => y_out_ALU, z_out => z_out_ALU);
    CORDIC_LUT: LUT port map(i_in => i_in_ALU, theta_out => theta_out_LUT);
    CORDIC_Storage: Storage port map(x_in => x_in_Storage, y_in => y_in_Storage, z_in => z_in_Storage,
            done => done_Storage, clock => CLOCK, i_read => i_read_Storage, i_write => i_in_ALU, reset => reset_Storage,
            x_out => x_out_Storage, y_out => y_out_Storage, z_out => z_out_Storage);
```

```
reset_Storage <= RESET; -- if reset button is pressed, set storage data to 0's

process(START, RESET, CLOCK, x_0, y_0, z_0, iteration_select, x_out_ALU, y_out_ALU, z_out_ALU, state,
        x_out_Storage, y_out_Storage, z_out_Storage, done_Storage, i_read_Storage)
begin
    if RISING_EDGE(CLOCK) then
        if RESET = '1' then
            state <= mode_wait;
        elsif START = '1' and state = mode_wait then
            state <= mode_start;
        end if;

        case state is
            when mode_wait =>   -- idle
                state_out <= "0001";
                done_Storage <= '0';
                i_in_ALU <= x"0";

            when mode_start =>  -- initalization, i=0
                state_out <= "0010";

                -- write initial data to slot 0 in storage arrays
                x_in_Storage <= x_0;
                y_in_Storage <= y_0;
                z_in_Storage <= z_0;

                -- feed initial data to ALU for first iteration
                x_in_ALU <= x_0;
                y_in_ALU <= y_0;
                z_in_ALU <= z_0;

                if MODE = '1' then  -- vectoring mode (y -> 0)
                    if y_0(15) = '0' then               -- y > 0
                        mu_i_ALU <= '0';
                    else                                -- y < 0
                        mu_i_ALU <= '1';
                    end if;
                else                -- rotation mode (z -> 0)
                    if z_0(15) = '0' then               -- z > 0
                        mu_i_ALU <= '1';
                    else                                -- z < 0
                        mu_i_ALU <= '0';
                    end if;
                end if;

                state <= mode_iterate;  -- switch to iterate mode next clock cycle

            when mode_iterate =>    -- iterate from i=1->15
                state_out <= "0100";
                i_in_ALU <= STD_LOGIC_VECTOR(UNSIGNED(i_in_ALU) + "1");

                -- feed previous iteration results to ALU
                x_in_ALU <= x_out_ALU;
                y_in_ALU <= y_out_ALU;
                z_in_ALU <= z_out_ALU;

                if MODE = '1' then  -- vectoring mode (y -> 0)
                    if y_out_ALU(15) = '0' then         -- y > 0
                        mu_i_ALU <= '0';
                    else                                -- y < 0
                        mu_i_ALU <= '1';
                    end if;
                else                -- rotation mode (z -> 0)
                    if z_out_ALU(15) = '0' then         -- z > 0
                        mu_i_ALU <= '1';
                    else                                -- z < 0
                        mu_i_ALU <= '0';
                    end if;
                end if;
```

```
                -- write values to storage(i)
                x_in_Storage <= x_out_ALU;
                y_in_Storage <= y_out_ALU;
                z_in_Storage <= z_out_ALU;

                if i_in_ALU = x"F" then     -- finished iterations at i = 15
                    state <= mode_display;
                    done_Storage <= '1';
                end if;

            when mode_display =>    -- done iteration, send data to display
                state_out <= "1000";
                i_read_Storage <= iteration_select;
                x_i <= x_out_storage;
                y_i <= y_out_storage;
                z_i <= z_out_storage;

            when others =>          -- error state
                state_out <= "0000";
        end case;
    end if;
```

ALU.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALU is
    port(
        x_in, y_in, z_in, theta_i: in STD_LOGIC_VECTOR(15 downto 0);
        i_in: in STD_LOGIC_VECTOR(3 downto 0);
        mu_i: in STD_LOGIC;
        x_out, y_out, z_out: out STD_LOGIC_VECTOR(15 downto 0));
end ALU;

architecture behavioral of ALU is
begin
    process(x_in, y_in, z_in, theta_i, mu_i, i_in)
    begin
        if mu_i = '1' then      -- clockwise rotation
            -- x_i+1 = x_i - y_i * 2^-i
            x_out <= STD_LOGIC_VECTOR(SIGNED(x_in) - SHIFT_RIGHT(SIGNED(y_in), TO_INTEGER(UNSIGNED(i_in))));
            -- y_i+1 = y_i + x_i * 2^-i
            y_out <= STD_LOGIC_VECTOR(SIGNED(y_in) + SHIFT_RIGHT(SIGNED(x_in), TO_INTEGER(UNSIGNED(i_in))));
            -- z_i+1 = z_i - theta_i
            z_out <= STD_LOGIC_VECTOR(SIGNED(z_in) - SIGNED(theta_i));
        else                    -- counterclockwise rotation
            -- x_i+1 = x_i + y_i * 2^-i
            x_out <= STD_LOGIC_VECTOR(SIGNED(x_in) + SHIFT_RIGHT(SIGNED(y_in), TO_INTEGER(UNSIGNED(i_in))));
            -- y_i+1 = y_i - x_i * 2^-i
            y_out <= STD_LOGIC_VECTOR(SIGNED(y_in) - SHIFT_RIGHT(SIGNED(x_in), TO_INTEGER(UNSIGNED(i_in))));
            -- z_i+1 = z_i + theta_i
            z_out <= STD_LOGIC_VECTOR(SIGNED(z_in) + SIGNED(theta_i));
        end if;
    end process;
end behavioral;
```

LUT.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity LUT is
    port(
        i_in: in STD_LOGIC_VECTOR(3 downto 0);
        theta_out: out STD_LOGIC_VECTOR(15 downto 0));
end LUT;

architecture behavioral of LUT is
begin
    process(i_in)
    begin
        case i_in is
            when "0000" =>              -- i = 0, theta = 0.7854
                theta_out <= x"3244";
            when "0001" =>              -- i = 1, theta = 0.4636
                theta_out <= x"1DAC";
            when "0010" =>              -- i = 2, theta = 0.2450
                theta_out <= x"0FAE";
            when "0011" =>              -- i = 3, theta = 0.1244
                theta_out <= x"07F6";
            when "0100" =>              -- i = 4, theta = 0.0624
                theta_out <= x"03FF";
            when "0101" =>              -- i = 5, theta = 0.0312
                theta_out <= x"0200";
            when "0110" =>              -- i = 6, theta = 0.0156
                theta_out <= x"0100";
            when "0111" =>              -- i = 7, theta = 0.0078
                theta_out <= x"0080";
            when "1000" =>              -- i = 8, theta = 0.0039
                theta_out <= x"0040";
            when "1001" =>              -- i = 9, theta = 0.0020
                theta_out <= x"0020";
            when "1010" =>              -- i = 10, theta = 0.00098
                theta_out <= x"0010";
            when "1011" =>              -- i = 11, theta = 0.00049
                theta_out <= x"0008";
            when "1100" =>              -- i = 12, theta = 0.00024
                theta_out <= x"0004";
            when "1101" =>              -- i = 13, theta = 0.00012
                theta_out <= x"0002";
            when "1110" =>              -- i = 14, theta = 0.000061
                theta_out <= x"0001";
            when "1111" =>              -- i = 15, theta = 0.000031
                theta_out <= x"0001";
            when others => NULL;
        end case;
    end process;
end behavioral;
```

Storage.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Storage is
    port(
        x_in, y_in, z_in: in STD_LOGIC_VECTOR(15 downto 0);
        done: in STD_LOGIC; -- if done = 0 => write mode, if done = 1 => read mode
        clock: in STD_LOGIC;
        i_read: in STD_LOGIC_VECTOR(3 downto 0);
        i_write: in STD_LOGIC_VECTOR(3 downto 0);
        reset: in STD_LOGIC;
        x_out, y_out, z_out: out STD_LOGIC_VECTOR(15 downto 0));
end Storage;

architecture behavioral of Storage is
    -- 16 16-bit vectors for x, y, and z data
    type x_data is array (15 downto 0) of STD_LOGIC_VECTOR(15 downto 0);
    type y_data is array (15 downto 0) of STD_LOGIC_VECTOR(15 downto 0);
    type z_data is array (15 downto 0) of STD_LOGIC_VECTOR(15 downto 0);

    -- initialize data to 0's
    signal x_array: x_data := (others => (others => '0'));
    signal y_array: y_data := (others => (others => '0'));
    signal z_array: z_data := (others => (others => '0'));

begin
    process (clock, x_in, y_in, z_in, done, i_read, i_write, x_array, y_array, z_array, reset)
    begin
        if reset = '1' then                                 -- reset data to 0's
            x_array <= (others => (others => '0'));
            y_array <= (others => (others => '0'));
            z_array <= (others => (others => '0'));

        else
            if FALLING_EDGE(clock) AND (done = '0') then    -- write mode
                x_array(TO_INTEGER(UNSIGNED(i_write))) <= x_in;
                y_array(TO_INTEGER(UNSIGNED(i_write))) <= y_in;
                z_array(TO_INTEGER(UNSIGNED(i_write))) <= z_in;
            end if;

            if done = '1' then                              -- read mode
                x_out <= x_array(TO_INTEGER(UNSIGNED(i_read)));
                y_out <= y_array(TO_INTEGER(UNSIGNED(i_read)));
                z_out <= z_array(TO_INTEGER(UNSIGNED(i_read)));
            end if;
        end if;
    end process;
end behavioral;
```

Appendix B: Incorporation of DFT

Design for Testability (DFT) is a design methodology which integrates testing mechanisms into hardware design. DFT can be incorporated into the CORDIC design by using a built-in self-test (BIST). This will allow the system to test itself, eliminating the need to create testbench files. Implementing a BIST will increase the reliability of our project, as well as decrease its complexity. As a result, the cost of project implementation will be reduced due to the elimination of the need for complex test equipment as well as decreased labour involved for testing.

Incorporation of BIST will be analyzed to test the angle memory on our initial LUT design.

Figure B1: Initial LUT Design

To implement BIST into our design, an exhaustive testing approach is recommended due to the small amount of iterations the CORDIC algorithm uses (small data set). To achieve this, an iteration counter will be implemented, instead of a pseudorandom number generator, in combination with the cyclic redundancy check (CRC). The benefits of moving to an iteration counter includes the ability to guarantee all combinations of data is tested, as well as a less-complex implementation.

A 2-to-1 multiplexer will then be added in between the data input and LUT blocks to select between normal operation and testing modes. While in test mode, the iteration counter will iterate through values 0 to 15 and feed said values into the lookup table.

A Linear Feedback Shift Register (LFSR) is implemented after the LUT to receive the output data from the LUT. The LFSR then creates a signature for the iterations. Next, this signature will be inputted into to a comparator, which will compare the signature obtained from the LFSR with the expected signature. This expected signature is called the “Golden Signature” and is stored in the system’s memory. If the signatures match, the testing of that iteration is a “Pass”, otherwise the test is a “Fail”. Pass or fail results are then sent to the controller.

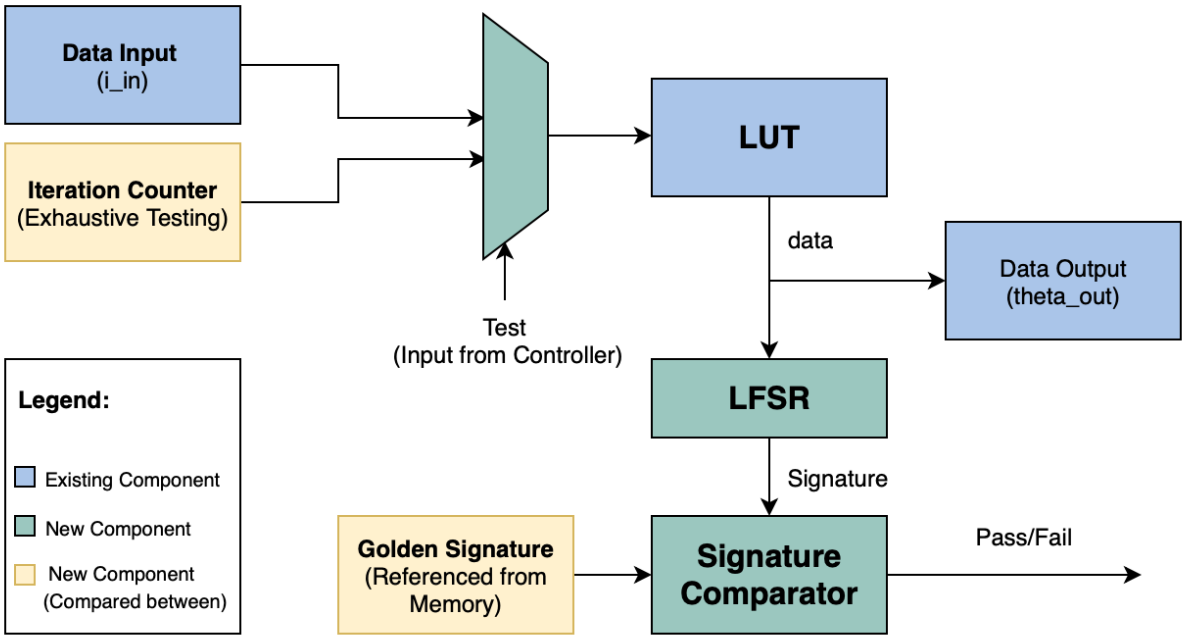


Figure B2: Design for Built-In Self-Test (BIST).

B.1 Iteration Counter:

A correct iteration counter must be able to iterate through values 0 to 15, as well as reset once testing has concluded. The iteration counter must also only send the current value when the system is not handling a previous iteration. In order to implement this in VHDL, a clock will be used to update the system state once it is finished with the previous state. This process will be created to update the state value from 0 to 15. When the state reaches a value of 15, the system will overwrite to 0 on the next increment, since the data is stored in 4-bits.

Iteration Counter Pseudocode:

```
bist_counter = std_logic := “0000”
process (clock)
    If rising_edge(clock)
        bist_counter = bist_counter +1
    end if
end process
```

B.2 2-to-1 Multiplexer:

The multiplexer is used as a switch between testing and normal operation modes. The multiplexer must be able to identify if either data from the controller or testing values from the iteration counter is to be sent to the LUT. A VHDL process can be created using this logic to determine which data the multiplexor sends to the LUT. Using three cases, the multiplexer may relay controller data for a value “0”, counter data for a value “1” and a case to send a value of “0” while in any other states.

Multiplexer Pseudocode:

```
process (multiplexer)
    if test_mode = ‘0’ THEN
        Output = controller_data;
    else
        Output = iteration_counter_data;
    end if
end process
```

B.3 Linear Feedback Shift Register (LFSR):

The LFSR will compress the entire outputted data (LUT) from test iterations 0 to 15 into one signature. This is done by using a cascaded system of flip-flops with added feedback, as seen in Figure B3. Due to this feedback system, values stored in registers D_0 through D_3 will be a combination of data from the current system state, as well as previous states. The current signature will be a combination of data stored in each of these registers. After each iteration, data stored in registers D_0 through D_3 will change, updating to reflect current and previous iterations. This will have the effect of changing the resulting signature, based on the number of iterations completed. On the final iteration (iteration 15), the final signature will be created as a function of all the outputted LUT data from iterations 0 to 15. This signature will then be sent to the signal comparator to be compared with the golden signature. To implement this design, a “clear all” command must be implemented to reset all register values to 0 after the final signature is generated.

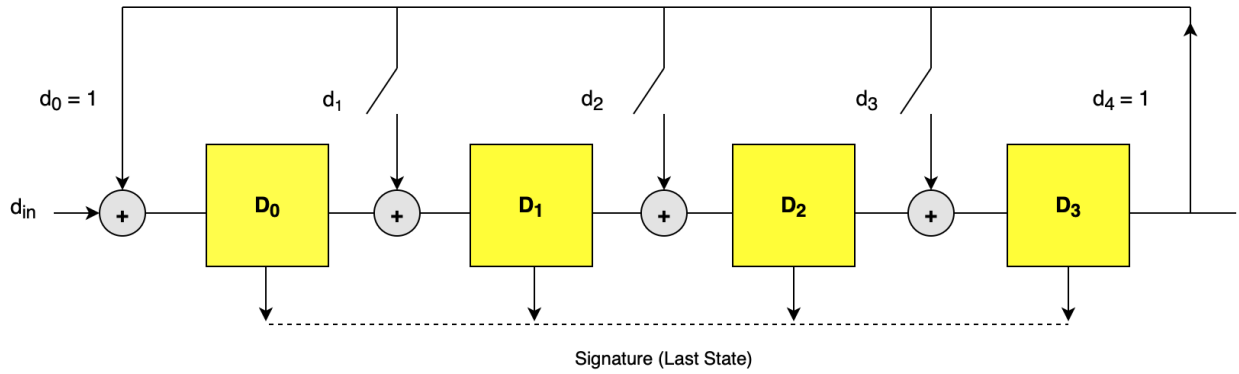


Figure B3: LFSR Block Diagram

Given four registers, D_0 through D_3, a limited amount of unique signatures can be created. An aliasing error occurs when the LFSR produces the same signature as the golden signature, while either the LUT output data is not correct, or the iterations are incomplete. This can be solved by increasing the number of registers to increase the complexity of the signature. A signature created with more registers will significantly decrease the possibility of an aliasing error.

B.4 Signal Comparator:

The signature comparator will compare signatures from the LFSR to the Golden signature. The signal comparator will then send a pass or fail signal depending on if the signatures match. VHDL code for signal comparator implementation may look similar to below.

Signal Comparator Pseudocode:

```
process(test_signature) is
begin
    if test_signature = golden_signature then
        result = pass
    else
        result = fail
    end if
end process
```

B.5 Golden Signature:

The Golden Signature will be referenced from memory. This signature will be implemented in a separate storage block, similar to how LUT.vhd was created. The Golden Signature will be inputted into the signal comparator to be compared with the signature created from the LFSR.

Appendix C: FPGA Layout

The FPGA layout, divided into clock regions, can be seen below. The illuminated blue regions represent the combinational logic blocks (based on LUT architecture) used by the design.

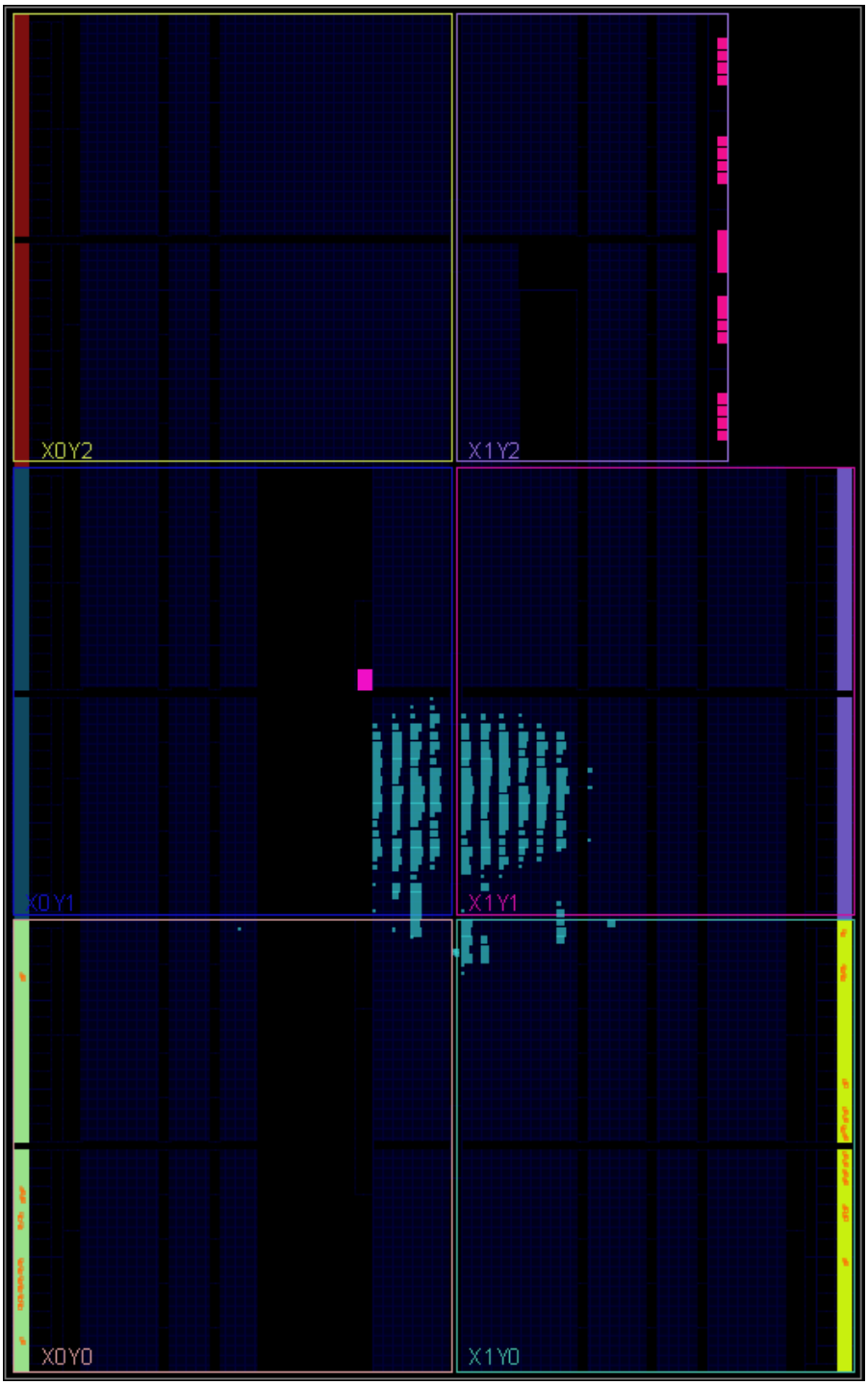


Figure C1: Implementation FPGA Layout