University of Victoria

Electrical & Computer Engineering

ECE 441 Design of Digital and VLSI Systems

FPGA Implementation of the CORDIC Algorithm

Project Final Report Spring 2019

Group Number	2
Project Title	CORDIC Implementation on BASYS 3 (Artix-7 FPGA Dev Kit)

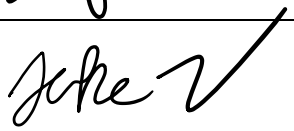
	Name	Signature
Group Members	Alexander Cote	
	Noah Rondeau	
	Jake Vidal	
Submission Date	5th of April, 2019	

Table of Contents

List of Figures

List of Tables

Introduction

Background

Coordinate Rotation Using Digital Computers (**CORDIC**) is a widely adopted iterative algorithm for calculating many non-elementary and transcendental functions [1].

CORDIC forms a family of similar iterative algorithms that all operate on a "shift-and-add" principle; all operations are performed as bit-shifts and additions. This makes CORDIC very efficient to implement on hardware that lacks dedicated multipliers [2]. The algorithm can also be implemented in software for micro processing platforms that only have integer arithmetic logic units. The algorithm is widely used in embedded applications, and has historically been used in calculators, flight navigation systems, digital signal processing, and many other fields [2].

The basic algorithm, called the "trigonometric" or "circular" CORDIC algorithm can compute the sine, cosine, square root, and arctangent; from these results, other functions are computable by running CORDIC multiple times and combining previous results [1,2]. Other algorithms in the family are extensions of the same concept, and allow calculation of hyperbolic functions, among others [2].

The Algorithm

The basic circular CORDIC algorithm is an iterative process that requires n iterations to complete when operating on n-bit data. It operates on vectors in the 2-dimensional plane. This algorithm has two modes: rotation (which allows for calculating sine and cosine) and vectoring (which allows calculation of the square root and the arctangent) [1,2].

The algorithm receives three inputs: x, y, z. The inputs x, y represent the vector data, and z represents a rotation angle. The purpose of the algorithm is either to rotate by z, thereby calculating the new vector coordinates (x, y) (rotation mode); or to translate the tip of the vector to the x-axis (i.e. drive y to 0), thereby calculating the angle argument of the vector (vectoring mode).

Depending on the mode, either y (vectoring mode), or z (rotating mode) should iteratively approach 0, with iterative values of each given by the following equations [1]:

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

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

$$z_{i+1} = z_i - \mu_i \theta_i$$

$$\theta_i = \arctan 2^{-i}$$

$$\mu_i = \begin{cases} +1 & clockwise\ rotation \\ -1 & counterclockwise\ rotation \end{cases}$$

The angles θ_i decrease in size for increasing iteration index i; μ_i is determined for every iteration in order to drive the value of interest (x or y) to zero. On each iteration, the output value may

overshoot or undershoot the desired value, but it will converge in smaller and smaller increments to the desired value.

To produce a correct output, a scale factor is also required at each iteration; however, due to the specific mathematical form of the equations, these can be factored out, and applied only to the final result, such that the scale factor is given by

$$K = \left(\prod_{i=0}^{N-1} \sqrt{1 + (2^{-i})^2} \right)^{-1}$$

A fixed-precision CORDIC can only handle inputs that are bounded by [1]

$$-1 \leq x_0, y_0 < 1, \qquad -1.7433 < z_o < 1.7433\ rad$$

in order to ensure overflow is avoided. Thus, for input values outside the 1st and 4th quadrants, and for vector components greater than length 1, a preprocessor external to the CORDIC core algorithm is required for scaling any data, translating it to the right-half-plane, and then rescaling and translating the output. This controller is also required for interpreting the input and output (for example, setting the correct initial values and mode to calculate a given function).

Though CORDIC can be implemented for arbitrary precision, choosing a fixed size allows the algorithm to be simplified. In particular, for fixed precision, the values of θ_i can be stored in a lookup table instead of being calculated (which itself would require another CORDIC, or some form of recursion). A fixed scale factor K can be calculated in advance. Fixed-point format can be used, with the input pre-scaling factors known in advance.

Design

The following sections detail the problem statement of this project, which is a limited subset of CORDIC on an FPGA. Existing literature is reviewed. A thorough design is proposed. The implementation, validation, testing, hardware performance and results of the design are discussed. Finally, the report makes recommendations for future improvements to the system (including a proposal for incorporating Built-In Self-Test into the design).

Problem Statement

The system designed in the context of this project is a subset of the CORDIC algorithm running on a Xilinx BASYS 3 Artix-7 FPGA development board [3]. Only circular CORDIC is considered in the design. Only the core iterative algorithm should be implemented; no preprocessor capable of mirroring and translating vectors from the left-half-plane and back should be implemented. As scaling the final value by K of CORDIC requires a secondary run of CORDIC, the final values will not be scaled. 16-bit, fixed point precision is used. Initial input values to the algorithm are further constrained to $0 \leq x < 1;\ 0 \leq y < 1; 0 \leq z < 2$. The results are to be displayed on the HEX display included with the BASYS. Inputs must be debounced. A reset input must return the system to its startup state. Any method of input may be used [4].

The system should expect inputs in the input operating ranges, scaled to be represented in HEX with no decimal point, i.e. x_0, y_0, and z_0 should be of the form (where numbers are in 2's complement) [1].

$$x_0\ (unscaled): \pm 2^0.2^{-1}2^{-2}2^{-3}2^{-4}2^{-5}2^{-6}2^{-7}2^{-8}2^{-9}2^{-10}2^{-11}2^{-12}2^{-13}2^{-14}2^{-15}$$

$$\text{Where } x_0\ (scaled) = x_0\ (unscaled) \times 2^{15}$$

$$y_0\ (unscaled): \pm 2^0.2^{-1}2^{-2}2^{-3}2^{-4}2^{-5}2^{-6}2^{-7}2^{-8}2^{-9}2^{-10}2^{-11}2^{-12}2^{-13}2^{-14}2^{-15}$$

$$\text{Where } y_0\ (scaled) = y_0\ (unscaled) \times 2^{15}$$

$$z_0\ (unscaled): \pm 2^1 2^0.2^{-1}2^{-2}2^{-3}2^{-4}2^{-5}2^{-6}2^{-7}2^{-8}2^{-9}2^{-10}2^{-11}2^{-12}2^{-13}2^{-14}$$

$$\text{Where } z_0\ (scaled) = z_0\ (unscaled) \times 2^{14}$$

Additionally, a built-in self-test (BIST) should be designed to ensure system health and the correctness of output [5]. Specifically, this BIST should assess the correctness of the values of θ_i stored in the look-up table. Implementation in hardware of the BIST is not required. It is treated in Appendix A1.

Literature Review

Beyond producing correct output, the chosen design should balance speed, chip usage, and power consumption. CORDIC is a common algorithm, and thus much literature is available. In this section, various sources are reviewed in order to find potentially useful implementation considerations.

From an architectural consideration, there are many ways to design the algorithm in hardware. Andraka [2] provides perhaps the most succinct overview of the different common methods. The principle difference presented is between feedback designs that leverage the iterative nature of the algorithm, and parallel or "unrolled" designs, where each iteration always performs the same calculation.

Andraka notes that though parallel designs use simpler hardware (they do not need memory) and may run faster, they cannot be achieved compactly on an FPGA due to the presence of memory elements and other unused components in each logic block. They also use more power due to the increase in hardware utilization. This would suggest for that for the current implementation, an iterative method should be used.

Andraka also presents a number of iterative designs. Notably he presents a bit-parallel design, which uses full-data-width busses for communication, and a bit-serial design, which uses serial communication of data. Andraka cautions that, though serial implementations using shift registers are by far the most simple, compact and fast, they must be run at a much higher clock rate (shift registers must be clocked fast enough to shift all the data in) [2]. The BASYS development board has a fixed crystal oscillator of 100 MHz which can be further divided to 450MHz [3]. Therefore, using a bit-serial design on this target would actually cause a single

algorithm iteration to take up to 16 times longer than a bit-parallel implementation. Though the complexity of the bit-parallel implementation may be forced to take more than one 100 MHz clock per iteration, this is still faster overall. Thus, due to the constraints of the hardware, a bit-parallel, iterative design should be employed.

Xu et al. [6] discuss potential optimizations employed on a bit-parallel CORDIC. In particular, they describe a bit-parallel CORDIC that is optimized by eliminating the need for a LUT in ROM to store values of rotation angle on most iterations. They achieve this by approximating $\arctan(2^{-i}) \approx 2^{-i}$ for large i, using a Taylor series expansion. This allows using right-shifting to calculate the value of θ, a potentially faster operation than retrieving data from ROM. However, the benefits of this system are only truly felt for large i. Though some speed up can be obtained in a 16-bit CORDIC, we feel that at this bit-length, the additional combinatorial logic and algorithmic complexity required does not outweigh the benefit of reduced ROM size and access times. A conventional LUT will be used instead.

Revathi et al. [7] present a design for a bit-parallel iterative CORDIC co-processor for use in a fingerprint-recognition device. They note that for their purposes, a good measure of performance is the relative error in the trigonometric functions, as a function of the angle of the computation. While this is a useful measure, it is not relevant to this project, as the project scope does not include a controller processor that indicates which function to compute. A different measure of correctness is required. Instead, the number of bits error from a reference implementation will be used. Hardware used, area, time-to-compute, and power consumption should also be evaluated.

Design Discussion

The following sections describe the functional design of the CORDIC processor and supporting modules. The focus is on describing the functionality of each system component, their interconnections with other components, and their expected operation and timings.

The design is broken down into three modules that correspond to three different domains of responsibility. The heart of the system is the CORDIC module. This module is responsible for computing the algorithm for a set of initial values, and then outputting the results (both per iteration results *and* final results). This module is self-contained, but as discussed, would usually be a component in a larger system that would control its operations and interpret its outputs. That is outside the scope of this project. Instead, two modules are used for interfacing the CORDIC to a human user: the Input Driver and the Output Driver. The Input Driver is responsible for collecting the initial values of x, y, and z, and well as the mode (rotation or vectoring) from the user, placing them on CORDIC's inputs, and sending the start signal. The Output Driver is responsible for collecting the CORDIC outputs (both per iteration and final) and storing them for later viewing on a seven-segment display. It provides a user interface to select which result (iteration and corresponding x, y, and z values) to show. This external storage allows the user to better validate the system.

This division of concerns has several motivations. Testability is the primary concern; the separation into modules ensures that each can be verified independently with a testbench.

Complexity is another factor; separated in this way, each module can be fairly simply implemented as an independent state machine that hands-off responsibility in a pipelined fashion.

Top Module Architecture

The Top module is responsible for taking the global hardware inputs from the BASYS board and mapping them to the three modules of the system. Another task accomplished is to properly link the modules together so that there is a seamless path from BASYS board inputs to BASYS board outputs. The Top module is different from the other modules discussed in this report in that it is composed entirely of a set of port maps linking modules.

The following table describes the inputs, outputs, and internal signals of the Top module.

Table 1: Top Module Signal Table

INPUTS		
Signal Name	Type	Function
CLK	STD_LOGIC	Global 100 MHz clock signal.
RESET	STD_LOGIC	An input signal directly from a BASYS board button.
INPUT	STD_LOGIC	An input signal directly from a BASYS board button.
SWITCHES	STD_LOGIC_VECTOR (15 downto 0)	16 input signals directly from the BASYS board switches.
OUTPUTS		
Signal Name	Type	Function
LEDS	STD_LOGIC_VECTOR (15 downto 0)	16 output signals directly to the BASYS board LEDs.
ANODE	STD_LOGIC_VECTOR (3 downto 0)	4 signals directly to the BASYS board seven-segment display anodes.
SEGMENT	STD_LOGIC_VECTOR (6 downto 0)	7 signals directly to the BASYS board seven- segment display cathodes.
INTERNAL SIGNALS		
Signal Name	Type	Function

DEBOUNCED_RESET	STD_LOGIC	A debounced version of RESET provided to internal modules.
DEBOUNCED_INPUT	STD_LOGIC	A debounced version of INPUT provided to internal modules.
DEBOUNCED_SWITCHES	STD_LOGIC_VECTOR (15 downto 0)	16 debounced versions of SWITCHES provided to internal modules.
X_IN_TRANSFER	SIGNED (15 downto 0)	A transfer variable for passing the x value from the Input Driver to the CORDIC.
Y_IN_TRANSFER	SIGNED (15 downto 0)	A transfer variable for passing the y value from the Input Driver to the CORDIC.
Z_IN_TRANSFER	SIGNED (15 downto 0)	A transfer variable for passing the z value from the Input Driver to the CORDIC.
MODE	STD_LOGIC	A transfer variable for passing the CORDIC mode from the Input Driver to the CORDIC.
START	STD_LOGIC	A transfer variable for passing the start signal from the Input Driver to the CORDIC.
X_OUT_TRANSFER	SIGNED (15 downto 0)	A transfer variable for passing the x value from the CORDIC to the Output Driver.
Y_OUT_TRANSFER	SIGNED (15 downto 0)	A transfer variable for passing the y value from the CORDIC to the Output Driver.
Z_OUT_TRANSFER	SIGNED (15 downto 0)	A transfer variable for passing the z value from the CORDIC to the Output Driver.
ITERATION	UNSIGNED (15 downto 0)	A transfer variable for passing the iteration from the CORDIC to the Output Driver.

READY	STD_LOGIC	A transfer variable for passing the data ready signal from the CORDIC to the Output Driver.

The Top module includes four major sub-components, the Input Driver, CORDIC, Output Driver and debouncers. The debouncer modules are directly connected to the BASYS board inputs in order to debounce the signals prior to use by any other module. The debounced signals are fed into Input Driver and Output Driver and are used in their user interfaces. All the other internal signals in the Top module act as transfer variables. There are five total signals which facilitate the transfer of outputs from the input driver to the CORDIC and five signals which facilitate the transfer of outputs from the CORDIC to the output driver. The following diagram shows the organization of the sub-modules and the signals used in their interconnections.

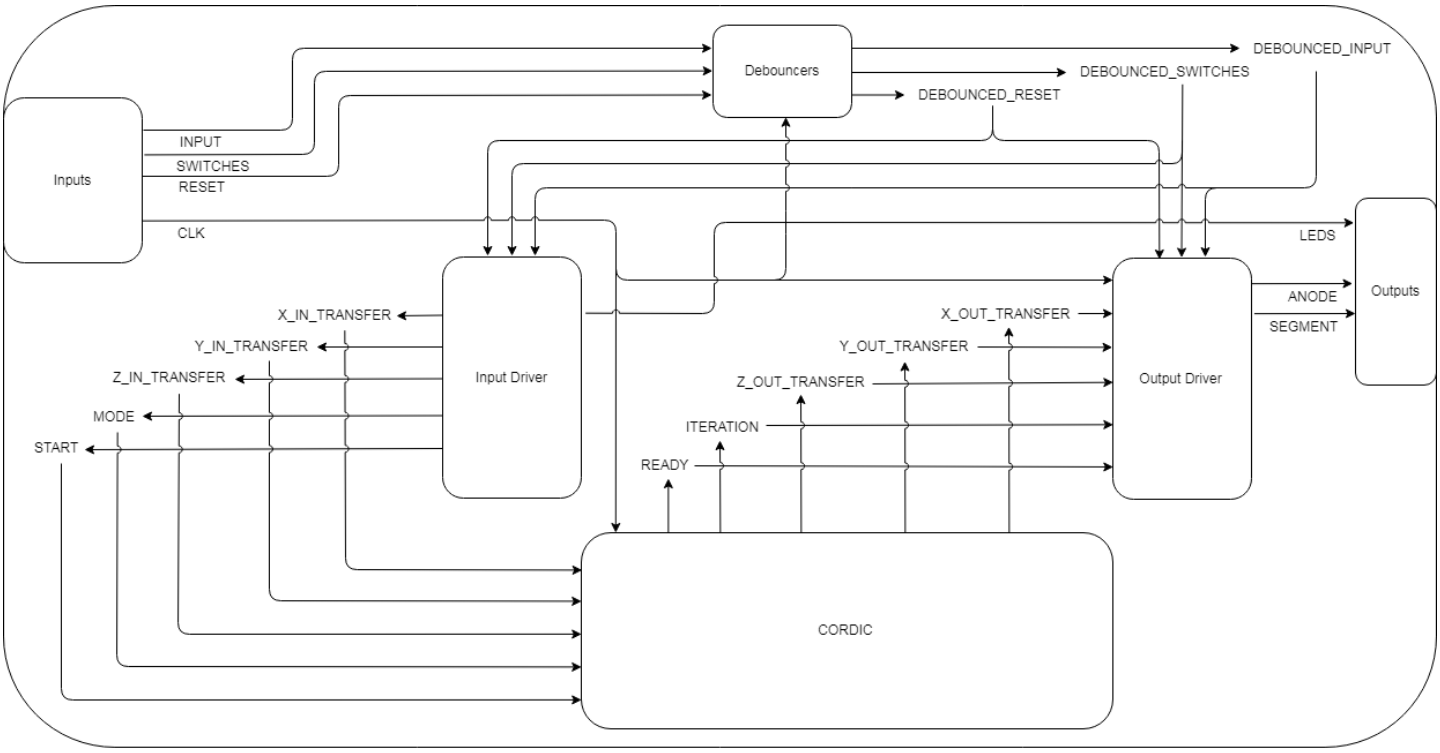


Figure 1: Top module internal architecture

The Top module is synchronous to the global CLK signal only so far as to provide the CLK signal to its various sub-modules. Since the debouncers, CORDIC, and the Output Driver are synchronous with the CLK signal, the Top module must provide them with a consistent CLK signal. The following timing diagram demonstrates the behaviour of the debouncers.

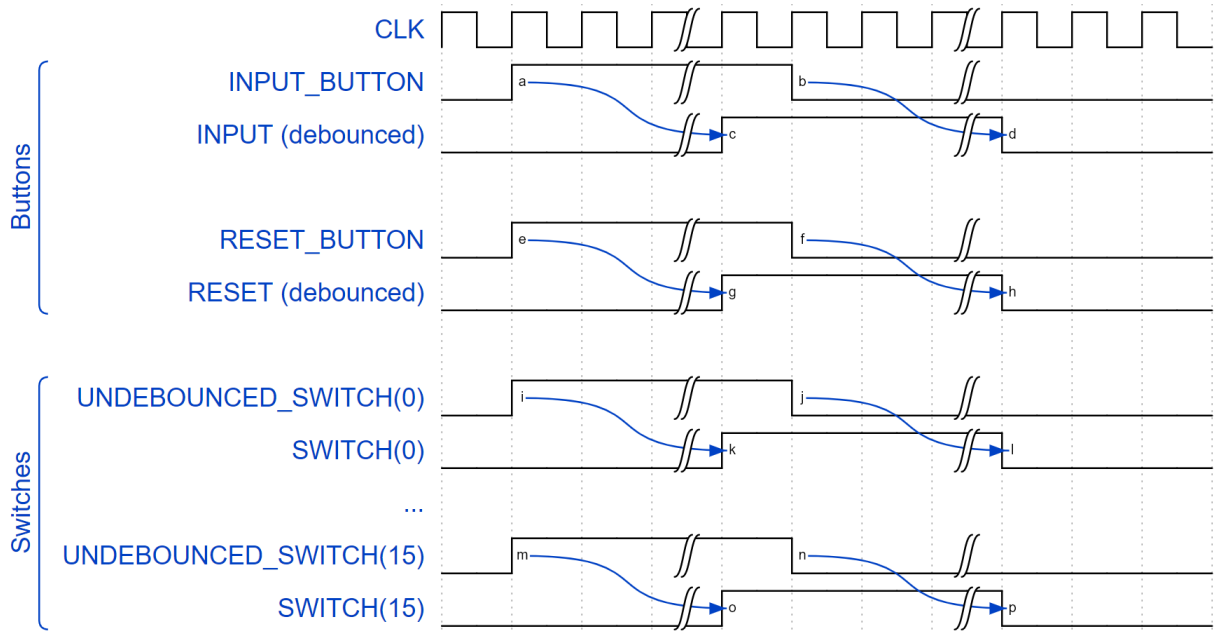


Figure 2: Debouncer module timing diagram

The debouncers, with respect to the processing time of other parts of the design, take a very long time to produce the output signal. Due to the average contact time of the button, the debounce delay from input to output is on the order of milliseconds.

Input Driver

The Input Driver module is responsible for receiving the initial values of x, y and z used for the CORDIC algorithm, as well as the mode, the start signal, and the reset signal. It propagates these values to the CORDIC module. One critical point to understand about this design is that the input driver and output driver manage the inputs and outputs respectively *of the CORDIC module*. It is in this sense, and this sense only, that they are input and output drivers. They both deal with *user inputs*.

The input driver has 16 switch inputs which represent 16 bits of input data. Two buttons are used: one to step through the stages of input, and another to reset the system. These input signals are debounced at the top level of the design.

The following table describes the inputs, outputs, and internal signals of the Input Driver.

Table 2: Input Driver Module Signal Table

INPUTS		
Signal Name	Type	Function
INPUT	STD_LOGIC	Pre-debounced input button signal.
RESET	STD_LOGIC	Pre-debounced reset button signal.
VALUE	STD_LOGIC_VECTOR (15 downto 0)	Pre-debounced switches for bit-wise data entry.
OUTPUTS		
Signal Name	Type	Function
X_VALUE	STD_LOGIC_VECTOR (15 downto 0)	Data representing initial value x_0. Used by CORDIC module.
Y_VALUE	STD_LOGIC_VECTOR (15 downto 0)	Data representing the initial value y_0. Used by CORDIC module.
Z_VALUE	STD_LOGIC_VECTOR (15 downto 0)	Data representing the initial value z_0. Used by CORDIC module.
MODE	STD_LOGIC	Signal representing the CORDIC operation mode. '0' = rotation, '1' = vectoring.
LED	STD_LOGIC_VECTOR (15 downto 0)	16 status indicator outputs. Mapped to BASYS 3 LED bank.
START	STD_LOGIC	Start signal for CORDIC module.
INTERNAL SIGNALS		
Signal Name	Type	Function
STATE	Compound Type:	Represents the current state of the FSM: BEGIN, X_INPUT, Y_INPUT, Z_INPUT, MODE_INPUT, START, END.

Because three different 16-bit numbers and a mode must be input to the system, but only a single 16-bit field is available, the input must occur in sequential stages. This lends itself well to a state-machine design (specifically a Mealy machine).

A block diagram of the Input Driver Finite State Machine is shown below (Figure 3):

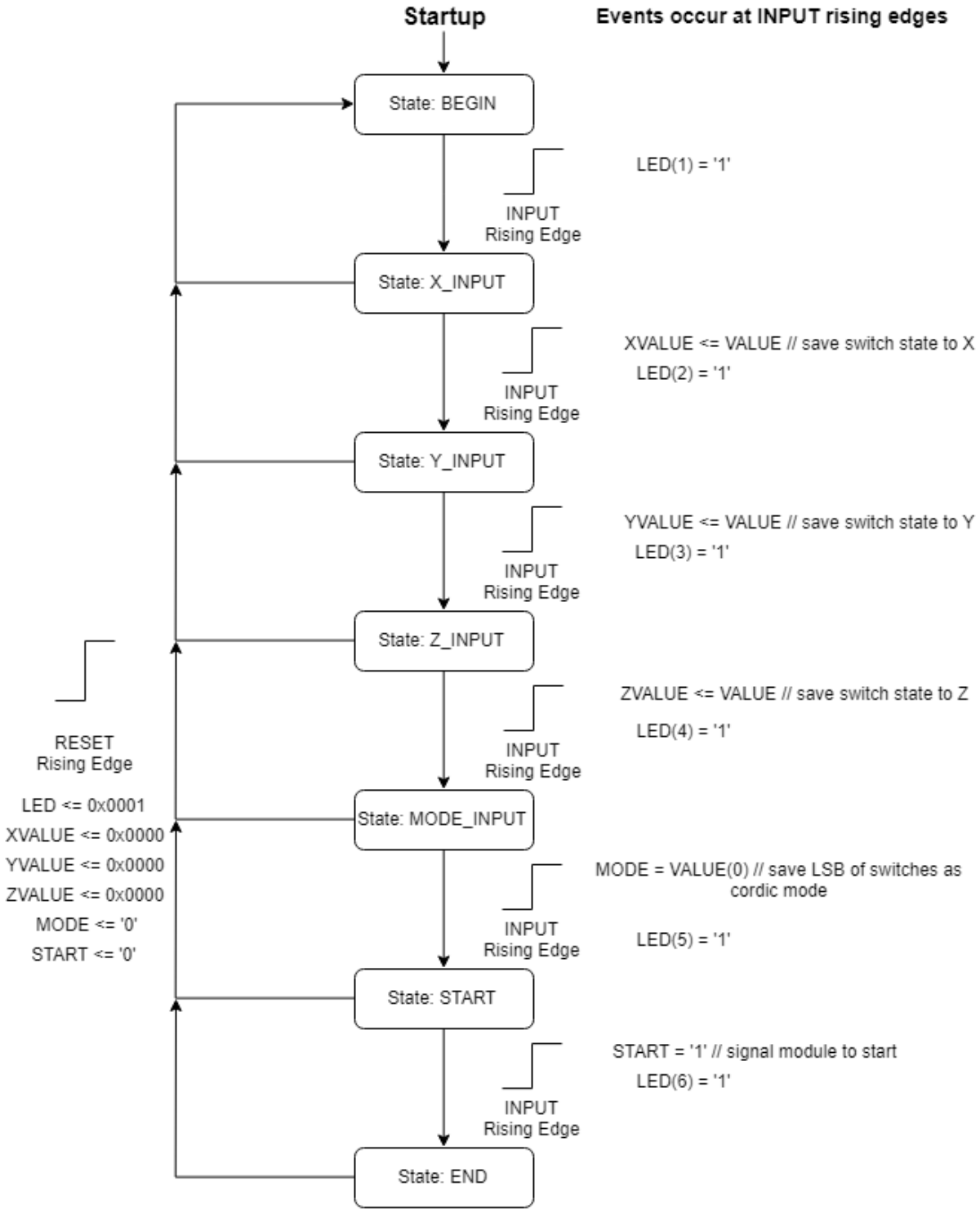


Figure 3: Input Driver FSM Flowchart

This is a linear Mealy state machine. Each state can only progress either to the next state, or back to the “begin” state upon reset. The state transitions are at the rising edge of the INPUT button signal. Therefore, a given state is conceptually just an indefinite wait for a rising edge. Any changes to the VALUE bitfield are captured at the rising edge of this input and routed to the correct output (depending on state). Once the last state is reached, no inputs have any effect. This is to allow the Output Driver module to use the same user inputs to control output display, without mangling the Input Driver FSM internal state.

Operation is as follows. The user first presses the input button to start. Then, the user sets each bit of the input to the desired value to indicate the initial value of the numerical input (either x, y, or z); the user then presses the input button, and the value is placed on the corresponding output (for use by the CORDIC module). This is repeated for each numerical input. A similar operation is performed for the CORDIC mode, where only the LSB of the bitfield is kept, with a 0 indicating rotation mode, and a 1 indicating vectoring; the user then presses the button to save this value, and the corresponding output is set. Finally, the button is pushed one last time to indicate the start signal for CORDIC; the value is passed through to the CORDIC module. After this point, no inputs should be able to affect the input driver, except for the reset signal.

The FSM is fundamentally asynchronous, as it waits for user input; it has no need for knowledge of the system clock. As such, the design of waveform timings is relative to the arrival of inputs. In this design, it is desired that all outputs occur at the rising edge of the INPUT signal and the RESET signal. The following timing diagrams illustrate the desired behavior of the module for each state.

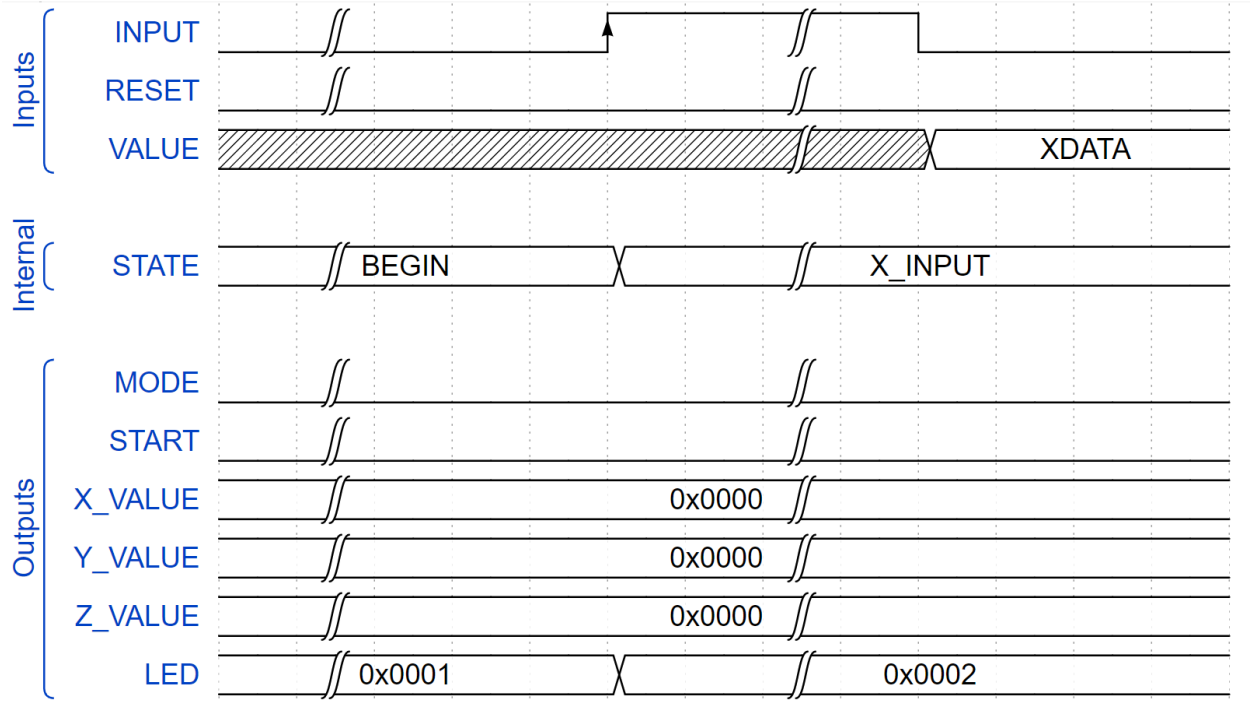


Figure 4: State BEGIN to State X_INPUT Transition Timings

Note that in the figure above, the duration spent waiting for the rising edge of the INPUT signal, the time it is high for, do not matter. The response on the STATE and LED signals should be immediate and synchronized to the edge.

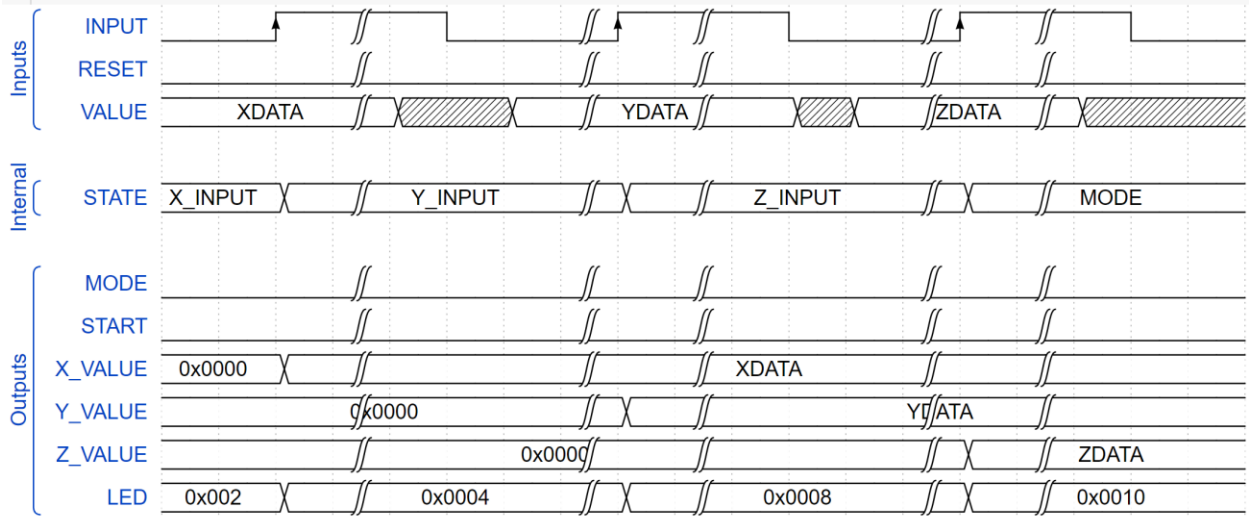


Figure 5: X_INPUT, Y_INPUT, Z_INPUT Transitions Timing Diagram

As the diagram above clearly shows, at each rising edge of the INPUT signal, the data on the VALUE field is transferred directly to the corresponding output depending on the state. The State transitions at the edge, and the indicator LED as well.

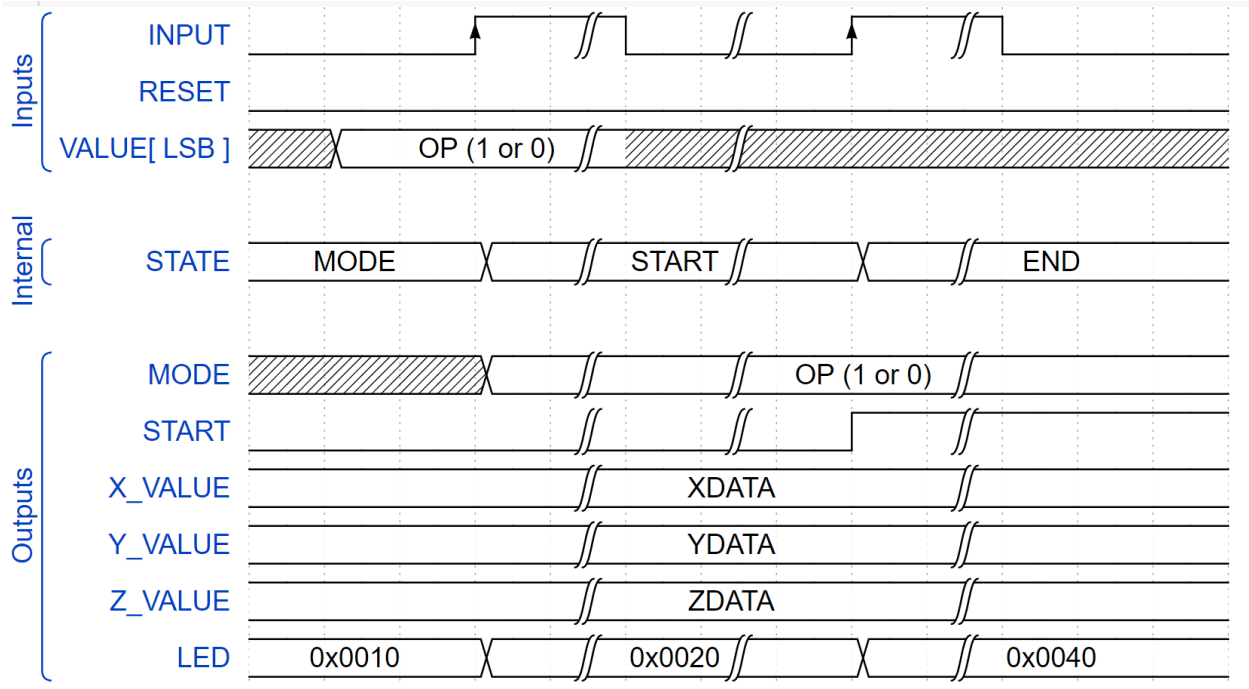


Figure 6: Mode, Start and End State Transition Timing Diagrams

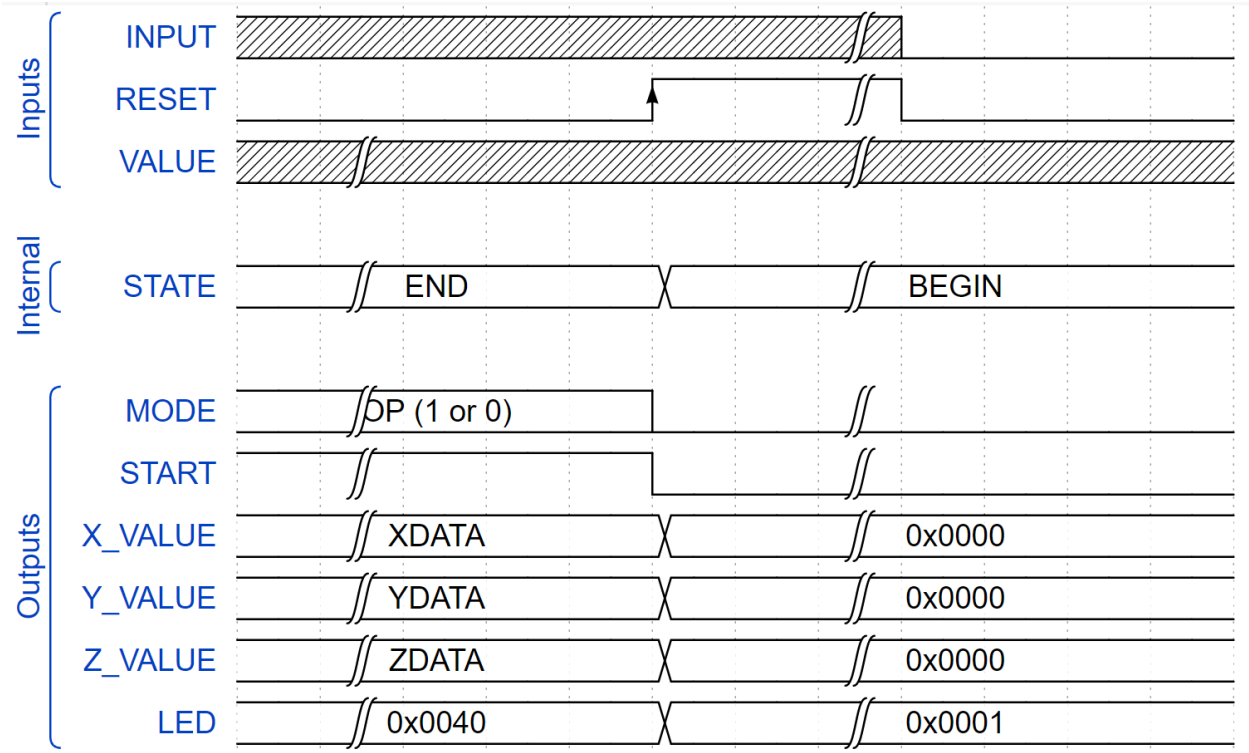


Figure 7: END State and Reset Timing Diagram

Here, the value of the INPUT signal is “don’t care” so long as the state is still END or the RESET signal has not returned to ‘0’ after assertion. All other signals return to default values at the rising edge of the reset signal.

More generally, the reset behaves in the same way out of any state, as shown below.

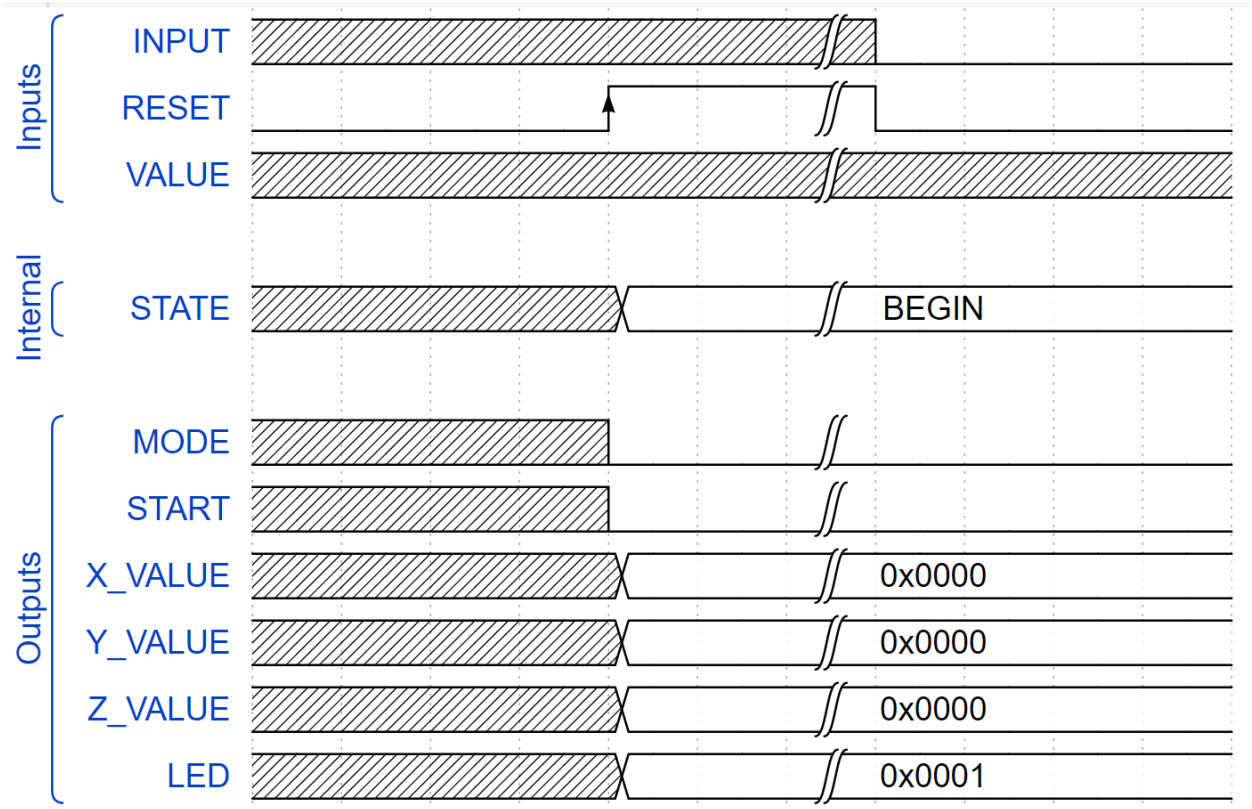


Figure 8: Reset from any state timing diagram

CORDIC Module

The CORDIC module contains both the CORDIC controller, and the CORDIC ALU. As described in the introduction, CORDIC will perform a certain number of iterations based on the number of data bits. As we are implementing a 16-bit rotation and vectoring mode CORDIC, 16 iterations are required.

The main idea behind the CORDIC module is the following: after being fed the correct input data, and given a start signal, CORDIC will process all 16 iterations sequentially, sending the result each iteration to the output driver to store for user access. Internally, the CORDIC module will hold all the information it requires to calculate the next iteration. This includes a look up table (LUT) for the theta values, and the previous value of x, y and z. From these, the value of mu can be calculated.

The following table describes the inputs, outputs, and internal signals of the CORDIC module.

Table 3: CORDIC Module Signal Table

INPUTS		
Signal Name	Type	Function

CLK	STD_LOGIC	Global 100 MHz clock signal.
RESET	STD_LOGIC	Pre-debounced reset button signal.
X_ IN_VALUE	SIGNED (15 downto 0)	Data representing initial value x_0.
Y_ IN_VALUE	SIGNED (15 downto 0)	Data representing the initial value y_0.
Z_ IN_VALUE	SIGNED (15 downto 0)	Data representing the initial value z_0.
MODE	STD_LOGIC	Signal representing the CORDIC operation mode. '0' = rotation, '1' = vectoring.
START	STD_LOGIC	Start signal for CORDIC module.
OUTPUTS		
Signal Name	Type	Function
X_RESULT	SIGNED (15 downto 0)	The value of the x result for the current iteration calculated by CORDIC.
Y_RESULT	SIGNED (15 downto 0)	The value of the y result for the current iteration calculated by CORDIC.
Z_RESULT	SIGNED (15 downto 0)	The value of the z result for the current iteration calculated by CORDIC.
ITERATION	UNSIGNED (3 downto 0)	The current value of the CORDIC iteration, provided as an output to the output driver.
MU	STD_LOGIC	The value of mu used in the previous CORDIC iteration: $\mu_i = \begin{cases} +1 & clockwise\ rotation \\ -1 & counterclockwise\ rotation \end{cases}$
DATA_READY	STD_LOGIC	A signal notifying the output driver that the data output from CORDIC is correct.
INTERNAL SIGNALS		
Signal Name	Type	Function
ITERATION	UNSIGNED (3 downto 0)	The current value of the CORDIC iteration, stored internally so it can be incremented.
X_ALU_INPUT	SIGNED (15 downto 0)	The value x_i placed in the ALU formula:

		$x_{i+1} = x_i - \mu_i y_i 2^{-i}$
Y_ALU_INPUT	SIGNED (15 downto 0)	The value y_i placed in the ALU formula: $y_{i+1} = y_i + \mu_i x_i 2^{-i}$
Z_ALU_INPUT	SIGNED (15 downto 0)	The value z_i placed in the ALU formula: $z_{i+1} = z_i - \mu_i \theta_i$
X_CURRENT	SIGNED (15 downto 0)	The value x_{i+1} returned from the ALU formula: $x_{i+1} = x_i - \mu_i y_i 2^{-i}$
Y_CURRENT	SIGNED (15 downto 0)	The value y_{i+1} returned from the ALU formula: $y_{i+1} = y_i + \mu_i x_i 2^{-i}$
Z_CURRENT	SIGNED (15 downto 0)	The value z_{i+1} returned from the ALU formula: $z_{i+1} = z_i - \mu_i \theta_i$
THETA	SIGNED (15 downto 0)	The value of theta from a look up table, based on the iteration of CORDIC. $\theta_i = \arctan 2^{-i}$
MU	STD_LOGIC	The value of mu used in the previous CORDIC iteration: $\mu_i = \begin{cases} +1 & clockwise\ rotation \\ -1 & counterclockwise\ rotation \end{cases}$
STATE	Compound Type:	Represents the current state of the FSM: MODE_IDLE, MODE_CALCULATE, MODE_OUTPUT, MODE_COMPLETED.

The reasoning for duplicate signals (internal signals and output signals) such as the iteration and mu value, are due to the fact that output signals cannot be "read" from in VHDL, so in order to modify the signals (e.g. increment the iteration), internal copies of the signals must be maintained.

A block diagram of CORDIC's system architecture can be found below in Figure 9:

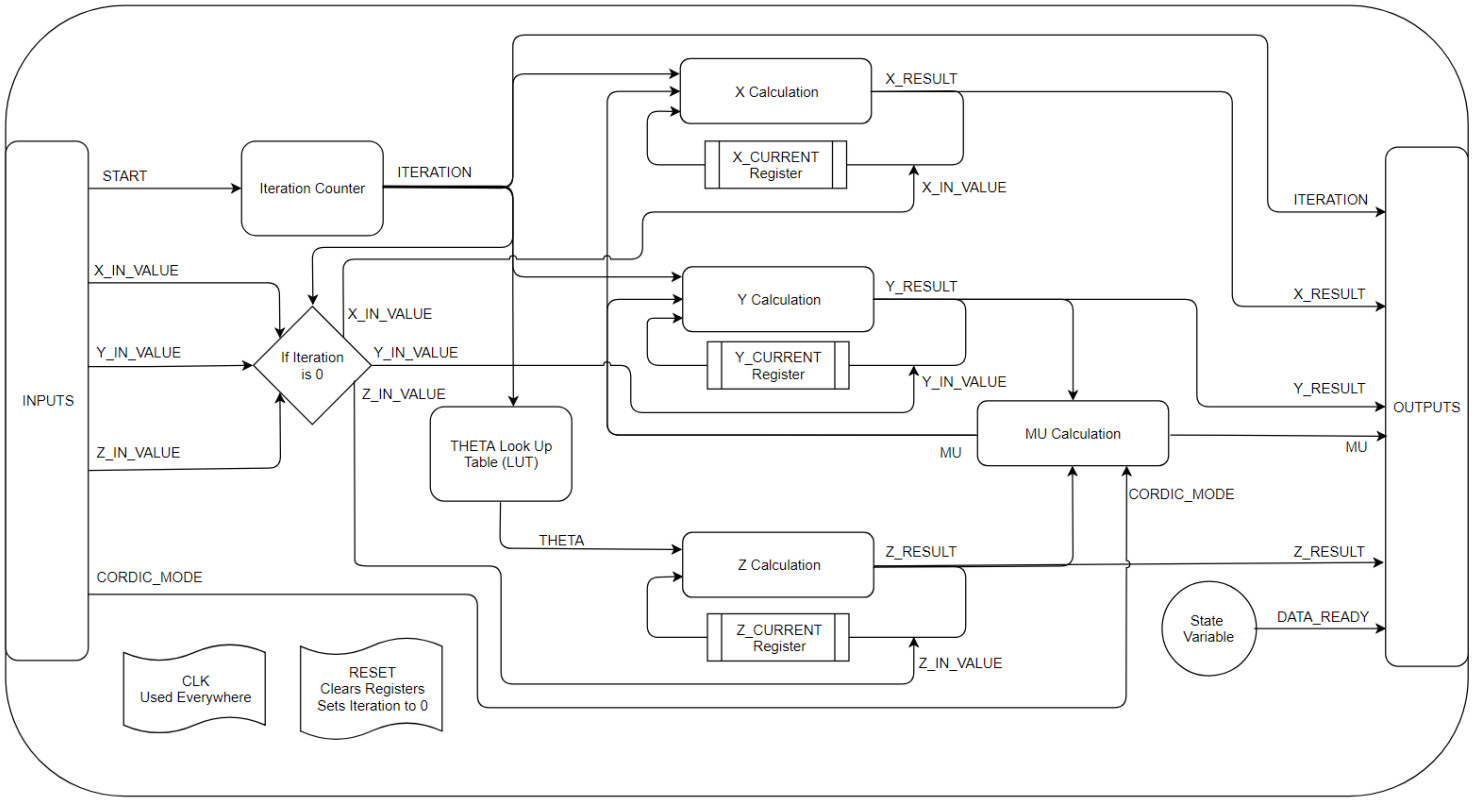


Figure 9: CORDIC Module Architecture - both controller and ALU

The CORDIC module contains the following logic, outlined from the above system architecture. The iteration counter increments from 0 to 16 and begins once the CORDIC module receives a start signal. The initial values are fed into the outputs, and stored in the x, y and z current registers. These registers are then fed back into the x, y and z calculation modules which form the ALU. The calculation modules each contain a register that stores the new value of x, y and z for the next iteration.

The value of theta used in the z calculation is dependent only on the value of the iteration, allowing a simple look up table to hold the values. The calculation for MU is more complex, using either the value of y or z depending on which mode of CORDIC is selected.

However, the block diagram of CORDIC's architecture only paints one half of the CORDIC module. A necessary secondary piece is the state machine used for the CORDIC controller. This shows the use of the system clock, and the DATA_READY output of the CORDIC module.

The state machine of the CORDIC controller is shown below, in Figure 10.

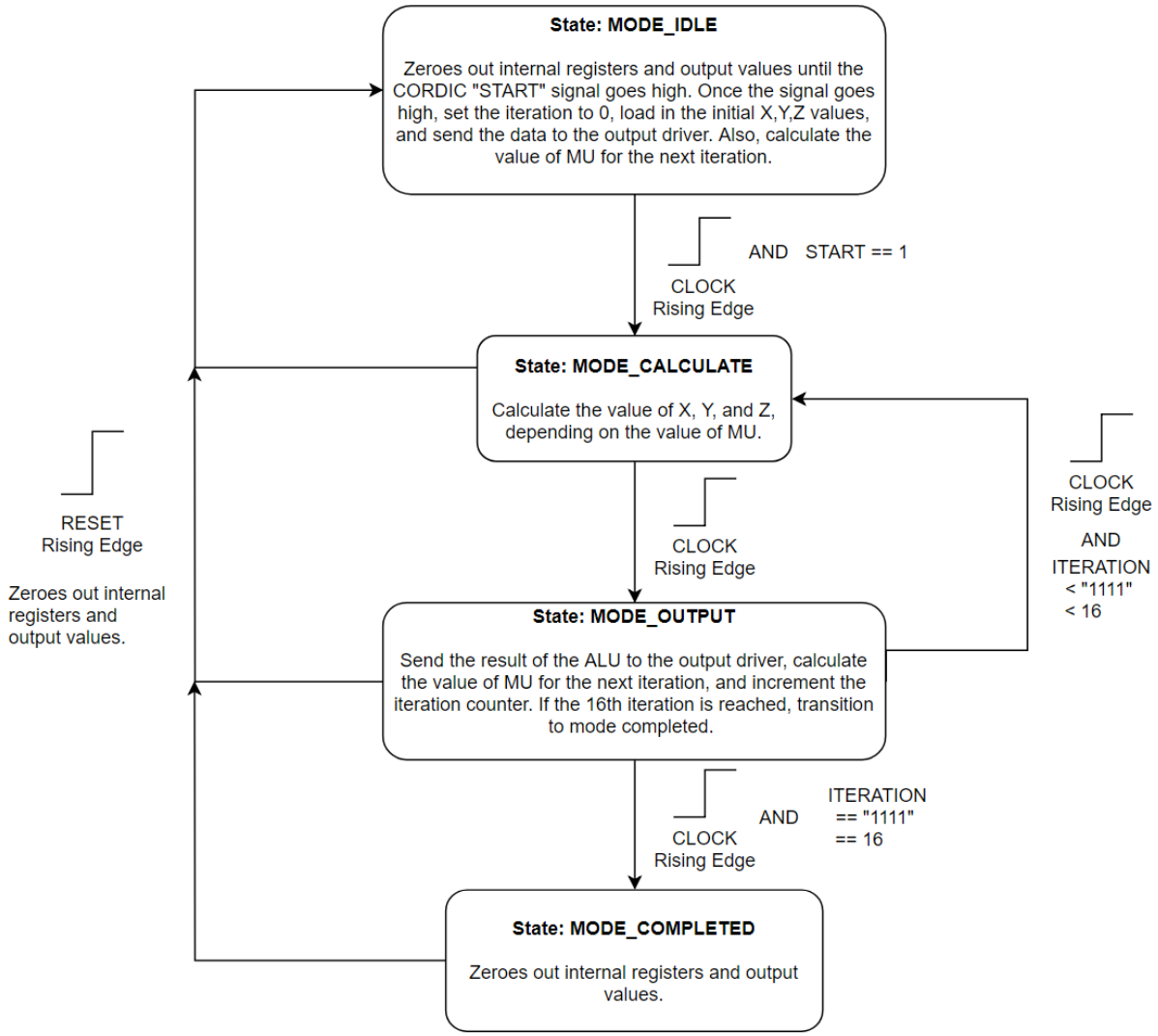


Figure 10: State Machine of the CORDIC Module.

The finite state machine of the CORDIC module has four states: MODE_IDLE, MODE_CALCULATE, MODE_OUTPUT, MODE_COMPLETED. Each transition between the states occurs on the rising edge of the clock. The CORDIC ALU and output are in the two middle states, MODE_CALCULATE and MODE_OUTPUT respectively. A condition allows the entry into MODE_CALCULATE from MODE_IDLE: the CORDIC START signal going high. The condition that stops CORDIC from continuing is the iteration counter reaching 16, which forces MODE_OUTPUT to transition to MODE_COMPLETED.

MODE_IDLE is the initial state. It sets the internal registers and output values to 0 until START goes high. Once the START signal goes high, the initial values are loaded into the required registers and fed to the outputs as iteration 1. It also calculates the value of MU for the first iteration of CORDIC.

MODE_CALCULATE runs the ALU component of CORDIC, calculating the next value of x, y and z depending on the value of theta, mu, and the current value of x, y and z.

MODE_OUTPUT signals to the Output Driver that the data is ready to be read and stored. It also calculates the next value of MU for the CORDIC ALU. MODE_OUTPUT and MODE_CALCULATE loop for 16 iterations until CORDIC is completed.

MODE_COMPLETED returns all the internal values and output values to 0. As a design choice, CORDIC was built to be a “one-shot” calculator, so in order to calculate another set of values, the system must be reset.

To summarize, the CORDIC Module is purely synchronous, so every signal changes on the clock edge. The following timing diagrams demonstrate the systems changes between states, as each state of the CORDIC calculation only requires one clock cycle.

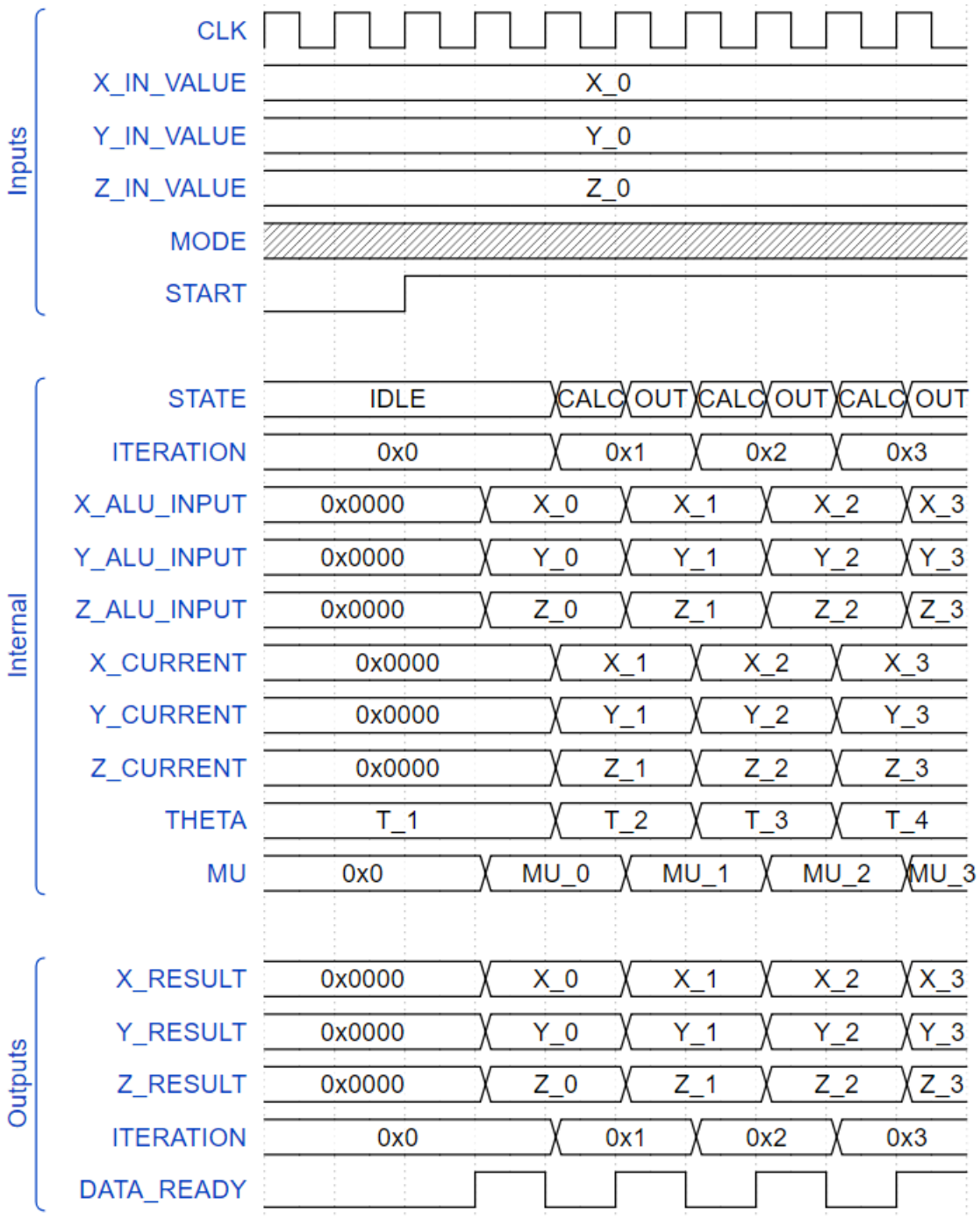


Figure 11: Timing Diagram of the CORDIC Module on startup

From the above timing diagram, it can be seen that CORDIC sleeps until the START signal goes high. Once that signal goes high, data becomes ready on every second clock cycle. The MODE_CALCULATE state generates the value of x, y and z CURRENT, and the OUTPUT state loads those values back into the ALU, creating the looping nature of the CORDIC algorithm. This timing diagram repeats until iteration 16 is reached, as seen in the figure below.

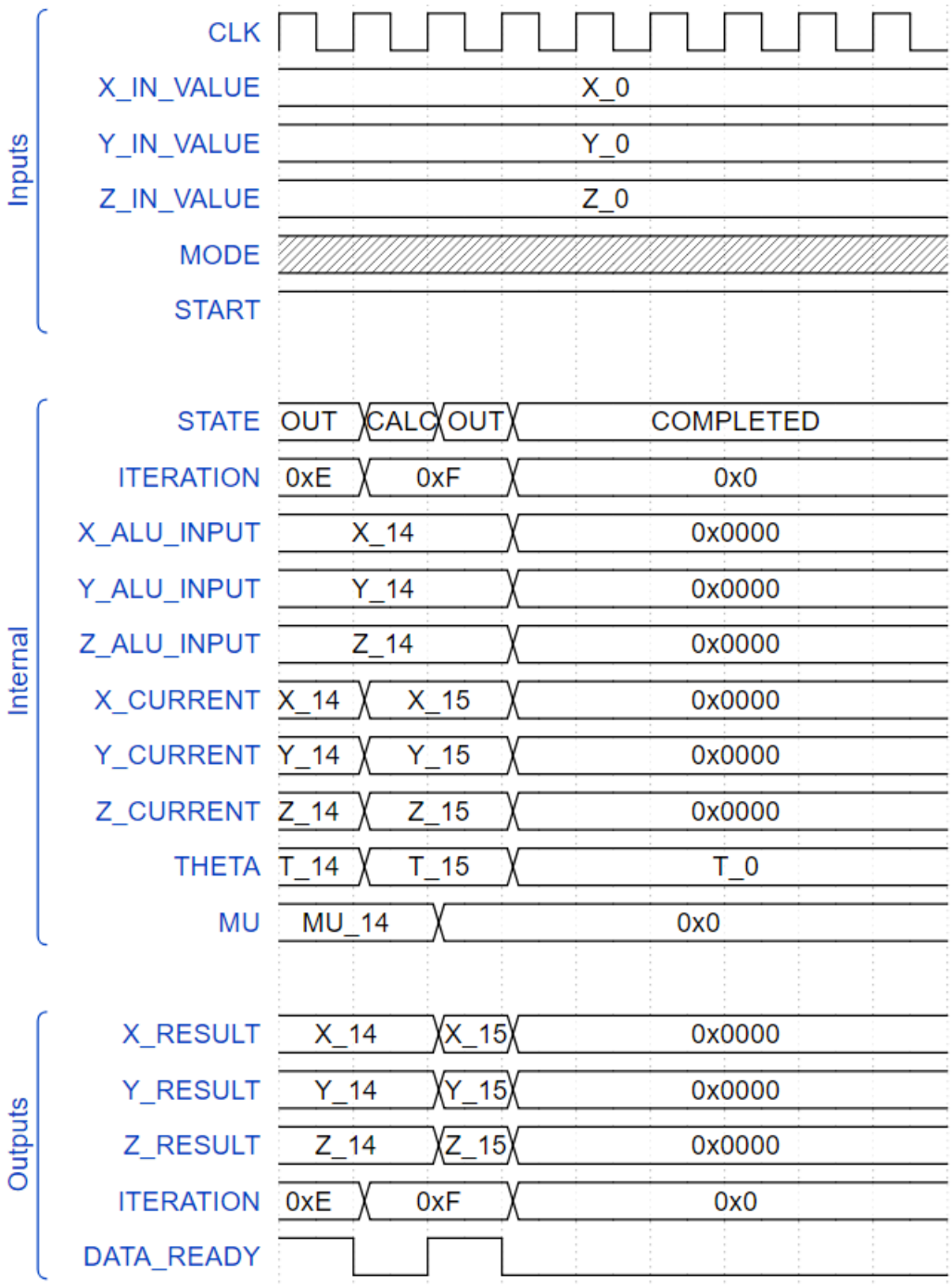


Figure 12: Timing Diagram of the CORDIC Module once completed

Here, the same pattern is followed until the COMPLETED state is reached. Once CORDIC is done and has sent the final set of data to the output driver, it returns everything to zero to minimize power usage. To use CORDIC for more operations, the RESET signal must go high, as seen in the following timing diagram.

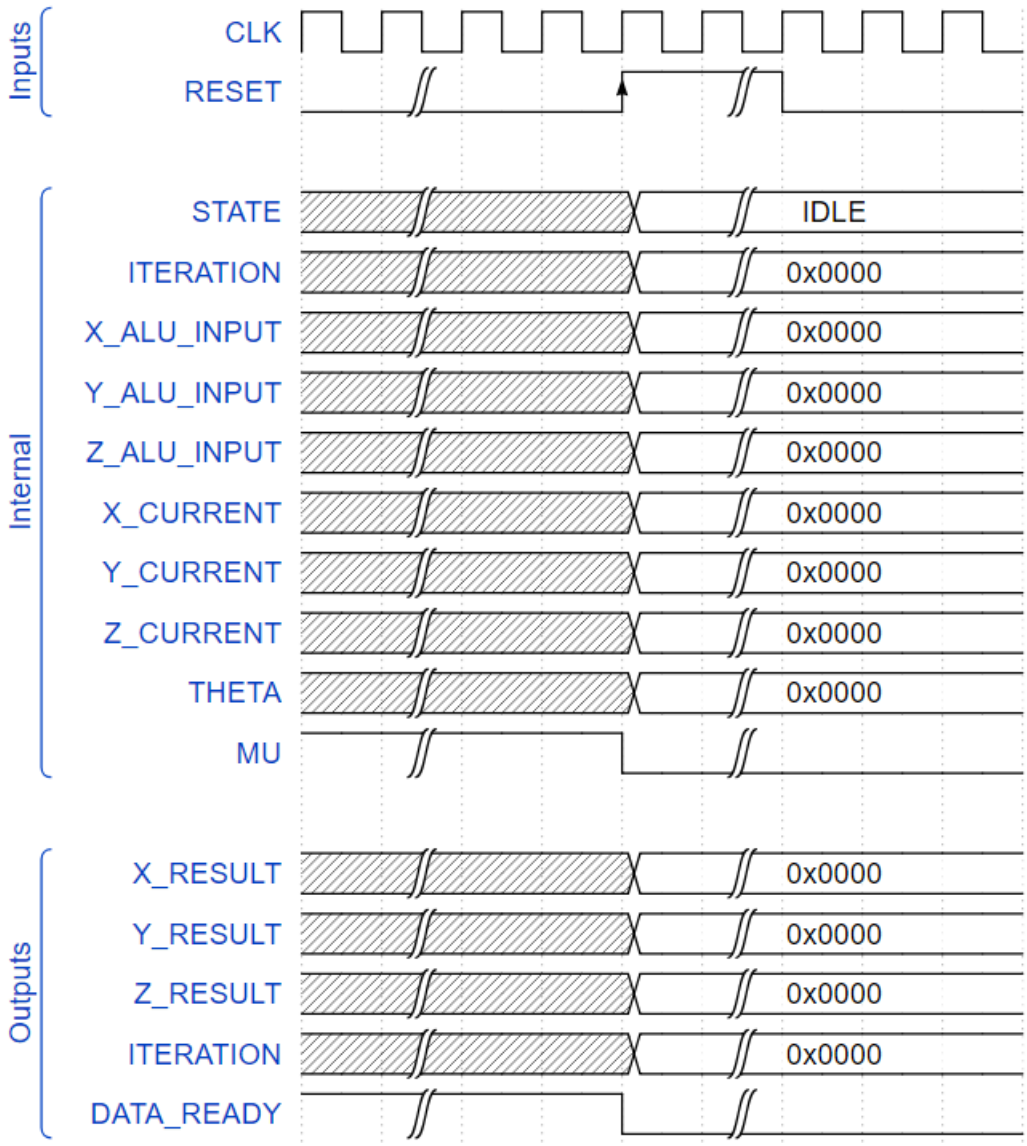


Figure 13: Timing Diagram of the CORDIC Module when subjected to a reset signal

The RESET signal going high returns CORDIC to the IDLE state and sets all internal values and outputs to 0.

Output Driver

The Output Driver module handles both the storage of the CORDIC x, y and z values after each iteration, as well as the display of the results after all 16 CORDIC iterations have completed. The x, y and z values from each CORDIC iteration are stored in RAM modules which are then read to obtain the desired value for display. The display is handled by providing a desired value to the hex driver module which then calculates the correct seven-segment display outputs.

The output driver has 7 switch inputs, 3 for selecting whether to display the x, y and z value and 4 for selecting the iteration to view in binary. There are also 5 input signals provided by the CORDIC module that provide data to store as well as a ready signal to indicate that the data is

valid. Lastly there are two inputs from the Top module, the clock signal and reset signal. All the physical button inputs used in this module are debounced at the top level of the design.

The following table describes the inputs, outputs, and internal signals of the output driver.

Table 4: Output Driver Module Signal Table

INPUTS		
Signal Name	Type	Function
CLK	STD_LOGIC	Global 100 MHz clock signal.
RESET	STD_LOGIC	Pre-debounced reset button signal.
X_RESULT	SIGNED (15 downto 0)	A signal provided by the CORDIC that holds the value of the x result for the current iteration.
Y_RESULT	SIGNED (15 downto 0)	A signal provided by the CORDIC that holds the value of the y result for the current iteration.
Z_RESULT	SIGNED (15 downto 0)	A signal provided by the CORDIC that holds the value of the z result for the current iteration.
ITERATION	UNSIGNED (3 downto 0)	A signal provided by the CORDIC indicating the iteration of the x, y and z results.
READY	STD_LOGIC	A signal provided by the CORDIC that indicates the iteration results are ready to be stored.
X_SELECT	STD_LOGIC	Pre-debounced input switch for choosing to display the x value of the selected iteration.
Y_SELECT	STD_LOGIC	Pre-debounced input switch for choosing to display the y value of the selected iteration.
Z_SELECT	STD_LOGIC	Pre-debounced input switch for choosing to display the z value of the selected iteration.
ITER_SELECT	STD_LOGIC_VECTOR (3 downto 0)	Pre-debounced input switch for selecting the CORDIC iteration to view the values of.
OUTPUTS		
Signal Name	Type	Function

ANODE	STD_LOGIC_VECTOR (3 downto 0)	A signal mapped to the BASYS seven-segment display that selects which digit to turn on.
SEGMENT	STD_LOGIC_VECTOR (6 downto 0)	A signal mapped to the BASYS seven-segment display that selects which segment to turn on.
INTERNAL SIGNALS		
Signal Name	Type	Function
ENABLEWRITE	STD_LOGIC	A signal provided to each internal RAM module that either enables writing when set to '1'.
RAMADDRESS	STD_LOGIC_VECTOR (3 downto 0)	A signal provided to each internal RAM module that selects the address to read or write from.
X_STOREDVAL	STD_LOGIC_VECTOR (15 downto 0)	A variable that stores the x value read from the internal RAM module's current address.
Y_STOREDVAL	STD_LOGIC_VECTOR (15 downto 0)	A variable that stores the y value read from the internal RAM module's current address.
Z_STOREDVAL	STD_LOGIC_VECTOR (15 downto 0)	A variable that stores the z value read from the internal RAM module's current address.
STARTDISPLAY	STD_LOGIC	A variable that makes the Hex Driver start generating outputs for the seven-segment display.
SELECTEDVAL	STD_LOGIC_VECTOR (15 downto 0)	A variable that holds the hex value that is currently selected to be shown on the seven-segment display.
STATE	Compound Type:	Represents the current state of the FSM: MODE_READ, MODE_WRITE.

The Output Driver as a module includes several sub-modules in order to accomplish its stated task. There are three RAM modules for storing x, y and z values as well as a Hex Driver module which facilitates the placing of a selected value on the BASYS board seven-segment display. The three RAM modules were generated using Xilinx IP and the HEX driver module code was provided on the course website. The three RAM modules each share their RAMADDRESS and ENABLEWRITE bits to ensure that x, y and z values are stored synchronously at the same address. The x, y and z values are stored in three different RAM modules in order to allow this

simple method of addressing. With this configuration, there is no requirement to decode the address or extract a single value out of a data string because the data is held separately. The outputs of the three RAM modules can then be used to choose a SELECTEDVAL for display on the seven-segment display. The following diagram displays the sub-modules of the Output Driver and their input and output signals however, it does not show the logic governing the values of the signals.

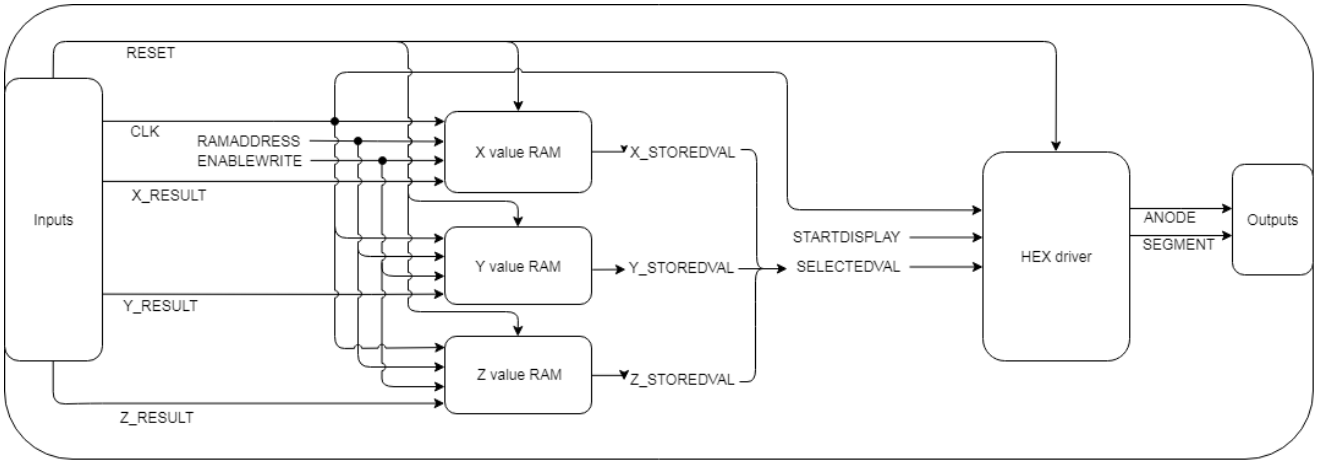


Figure 14: Output Driver module internal architecture

Because the behavior of the Output Driver is different before and after the CORDIC iterations have finished, it is easy to implement as a simple two state FSM. Since there are two input sources of the RAM address, it makes sense to use a Moore machine which selects the RAM address source based on state only.

A block diagram of the Output Driver Finite State Machine is shown below (Figure 15).

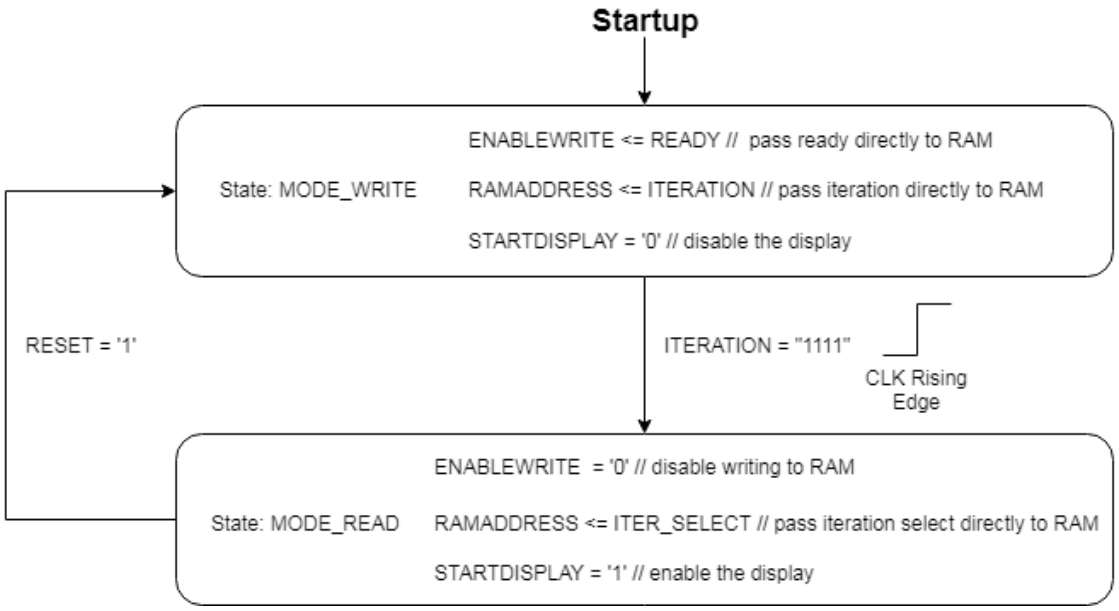


Figure 15: Output Driver FSM Flowchart

The finite state machine for the output driver only has two states, MODE_WRITE and MODE_READ. Before the CORDIC module has finished all its iterations, the output driver needs to be prepared to store the data from each iteration for later display to the user. To enable this, the output driver directly maps the CORDIC READY and ITERATION signals to the three internal RAM modules which hold the x, y and z values respectively. This causes the CORDIC module to automatically select the correct address in the RAM to store the values and prevents writing to the RAM unless the CORDIC validates the data it is providing. A final signal set in the MODE_WRITE state is the STARTDISPLAY signal, which is set to '0' in order to disable the display.

After the CORDIC has completed its final iteration, the state machine transitions to the MODE_READ state. In the MODE_READ state, the output driver is required to display the data selected by the user. To achieve this, the output driver remaps the three internal RAM modules to receive their addresses from the user controlled ITER_SELECT signal and sets the ENABLEWRITE signal to '0', preventing writes to the RAM. The state also sets the STARTDISPLAY signal to '1' to enable the seven-segment display.

From the user's point of view, the operation of the output driver is relatively straightforward. The most important aspect of the user interface is the ITER_SELECT switches. These four switches represent the iteration of the CORDIC algorithm for which the user would like to view the values. The ITER_SELECT switches represent the iteration using binary such that the switches being in positions "0010" will select data from iteration 2 of the CORDIC algorithm. The remaining three select switches simply determine whether the BASYS board will display the x, y and z value of the chosen iteration.

The output driver is synchronous to the global CLK signal and thus, in the following timing diagrams, it is important to note that signal changes will take place on the rising edge of the CLK. Due to the synchronicity of the design, signals must be available for two total CLK edges in order to be registered. An important note is that the greyed-out signals have values, but they are not considered vital to the timing diagram in order to show the desired logic.

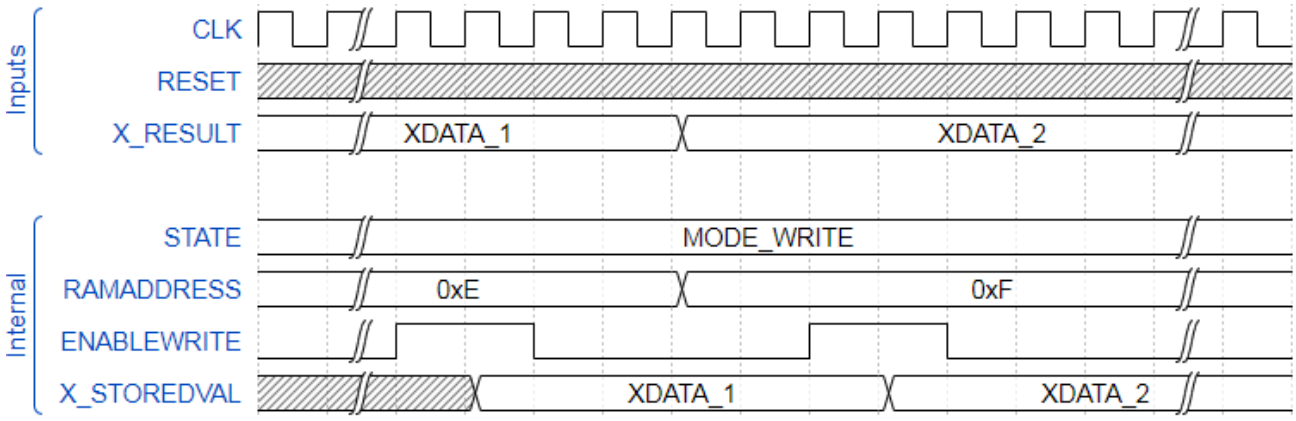


Figure 16: RAM module timing diagram

As shown in the figure above, each of the RAM modules only updates its output value on a CLK rising edge when the ENABLEWRITE signal is set to '1'. This timing diagram also models the behavior of the RAM in the state MODE_READ, since the only difference is that the

ENABLEWRITE signal will be set to ‘0’ and the RAMADDRESS will be set by a different source.

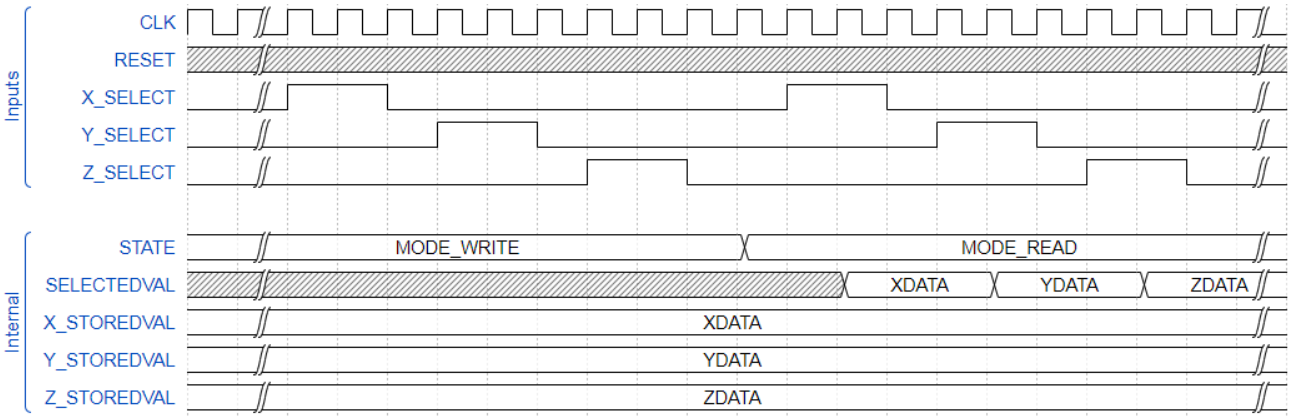


Figure 17: User interface timing diagram

The user interface timing diagram shows the effect that the user input switches have on the SELECTEDVAL which is sent to the hex driver for display. When the state is MODE_WRITE, the inputs are ignored but, when the state is MODE_READ, the CLK rising edge sets SELECTEDVAL to the desired value.

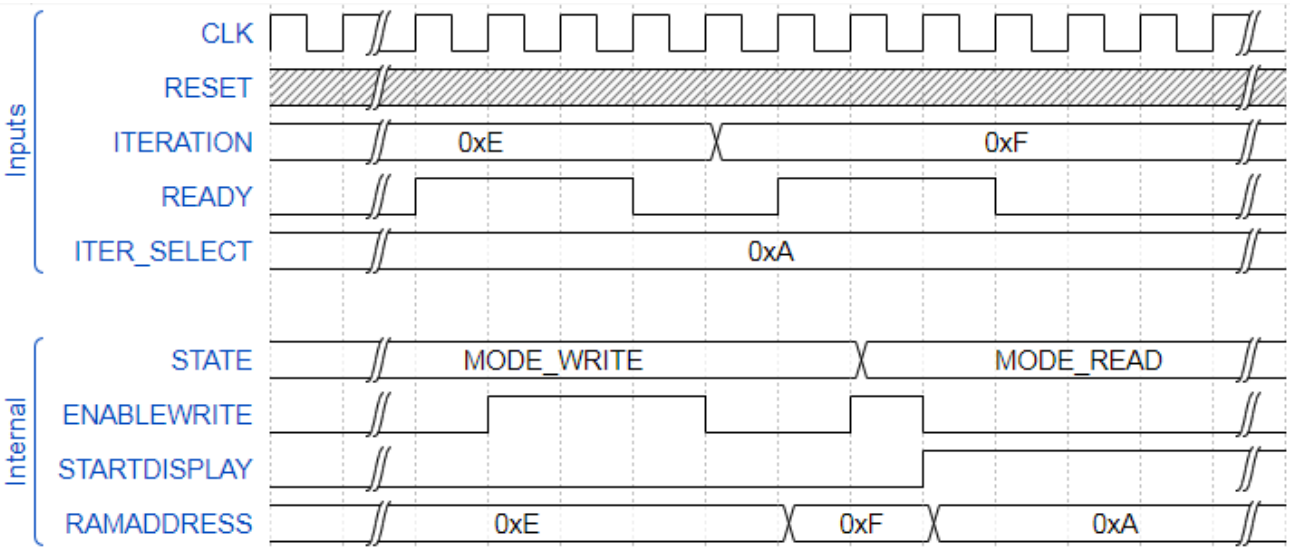


Figure 18: State machine timing diagram

The timing diagram above represents the core logic of the finite state machine in the output driver. Initially, the FSM is in MODE_WRITE where ENABLEWRITE and RAMADDRESS are set by signals from the CORDIC. After transitioning to the state MODE_READ, the ENABLEWRITE and RAMADDRESS signals are instead set by the signals from the user input. At the point where the states switch, it is important to note that the requirement for a CLK rising edge allows the ENABLEWRITE and RAMADDRESS signals to write the last iteration data to the RAM modules before they are set to the values required by MODE_READ.

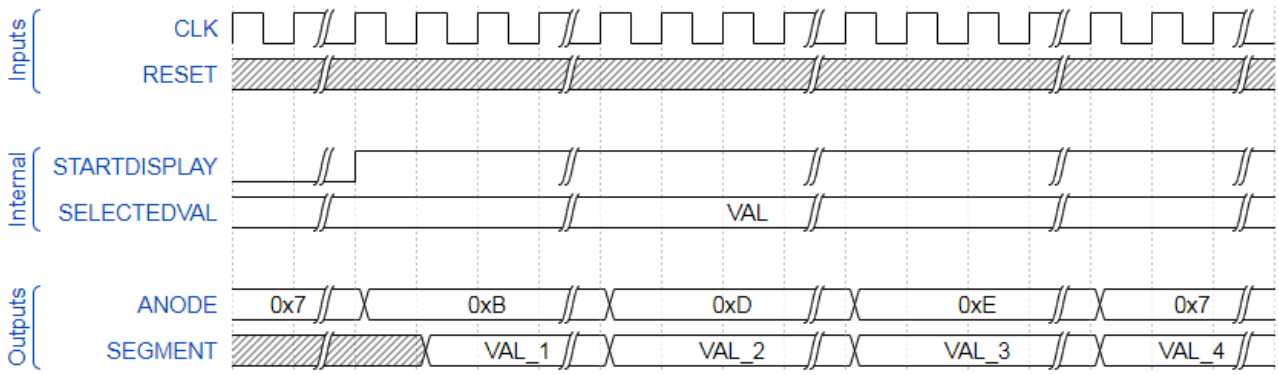


Figure 19: Display timing diagram

The figure above shows the operation of the Hex Driver and its cyclical output of SELECTEDVAL. Each of the anode and segment values shown are actually on the order of milliseconds in length and are shown compacted to illuminate the way that the seven-segment display works. The hex driver only shows one digit of the SELECTEDVAL at a time in order to conserve power.

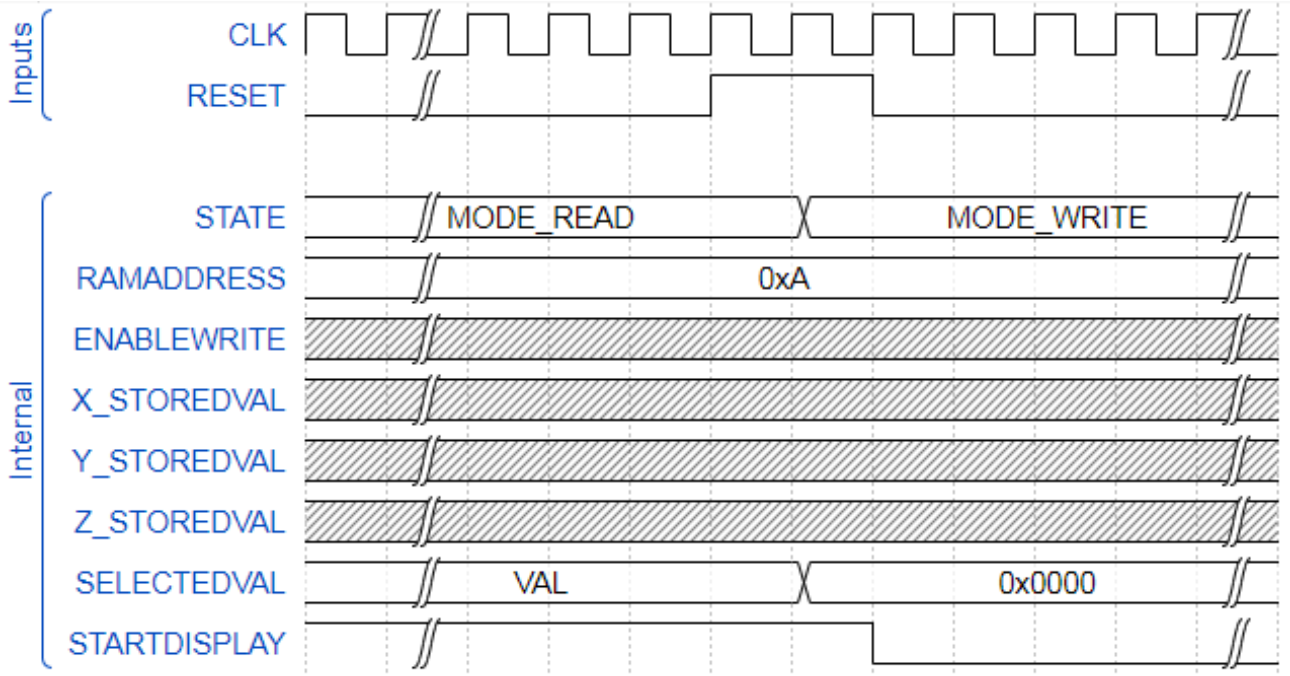


Figure 20: Output Driver Reset timing diagram

The timing diagram above shows the behavior of all internal signals in the case of a reset button press. Many of the signals are already set in the finite state machine and don't require a reset for full functionality to be maintained.

Implementation, Validation and Testing

In this section, details relative to the implementation, testing, and validation of the design presented above are discussed.

The system was implemented in VHDL. Thanks to the partition of the design into three modules with defined interface requirements it was possible to work in parallel.

Iterative design methods were also used; the design presented was refined from earlier drafts, partially as a response to events that occurred during the implementation phase.

For each module, the design was tested using testbench simulations during development. Once all modules were in a working state, they were interfaced in a VHDL Top module, and the Top module was tested via testbench as well, and then synthesized. Any subsequent changes were always tested again, both in isolation and as part of the integrated system. Once the system was ready, it was implemented in hardware and tested.

Input Driver Implementation Discussion

The initial design of the Input Driver did not use the switches and buttons located on the BASYS for input, but rather the PMOD Hexadecimal keypad [8]. After an initial exploration of the feasibility of implementing a hexpad driver, it became clear that it had a high relative complexity compared to the switch-and-button design. We decided that time-cost of development for this feature was too high, as it did not aid in demonstrating a functional CORDIC. In addition, the initial design contained the debouncers which are located in the Top module in the final design. This was changed to increase testability of the module in isolation.

The final design of the Input Driver was simple and straightforward to implement. The linear finite state machine design could be simply mapped to a single process in VHDL, sensitive the debounced input button signal and the debounced reset button signal.

CORDIC Core Implementation Discussion

The CORDIC module was implemented in VHDL as described by the finite state machine, architectural, and timing diagrams. The FSM was implemented as a case statement embedded within a process that is sensitive to the clock. The look-up table (LUT) for theta values was generated using a Xilinx IP LUT.

The design did change somewhat in response to discoveries during early implementation, though the conceptual operation did not change. Initially, the early design proposed to separate the calculations of x, y, z for a single iteration into a separate ALU block, implemented as a VHDL entity, that would be used by the CORDIC controller module finite state machine. This partition of the system was chosen for better testability; the correction of the mathematical operations could be independently verified. However, it was discovered using a separate entity – with an interface and its own internal processes – actually slowed down the calculations, as clock cycles were needed prepare signals for communication with the ALU and vice versa. The FSM (and the 4^{th} FSM for the ALU itself) was significantly more complex because of this. Once the correctness of the mathematical operations was verified through simulations, it was decided to

place those operations directly within the CORDIC module state machine. The resulting design and implementation were more efficient, more compact, and easier to reason about and debug.

For generating values to store in the theta LUT, it was necessary to convert the values of theta to binary, which allowed the z value to be calculated with a simple addition or subtraction of these values. The method used for converting the theta values to binary can be found in Appendix B.

Table 5: Theta Values in Binary

Iteration	Theta (radians)	Theta (Hex)
0	0.7854	0x3244
1	0.4636	0x1DAD
2	0.245	0x0FAE
3	0.1244	0x07F6
4	0.0624	0x03FF
5	0.0312	0x0200
6	0.0156	0x0100
7	0.0078	0x0080
8	0.0039	0x0041
9	0.002	0x0020
10	0.00098	0x0011
11	0.00049	0x0009
12	0.00024	0x0004
13	0.00012	0x0002
14	0.000061	0x0001
15	0.000031	0x0001

Output Driver Implementation Discussion

The Output Driver was implemented according to the design documents. The FSM was implemented using the same method as the CORDIC above. The user interface elements for selecting the data to display was implemented as a process running in parallel to the FSM. Xilinx IP RAM was used to store the data from each of the CORDIC iterations and to output the values to the hex driver and subsequently the seven-segment display.

The design of the Output Driver encountered one major change from the initial design phase to the final design. Initially, the Output Driver was designed to use registers to store the x, y and z values for each CORDIC iteration. This was changed to three separate RAM modules for ease of

addressing. With the original register implementation, the addressing of a particular value from a particular iteration proved difficult. The RAM modules on the other hand have a clear method of selecting data from a particular iteration in that the iteration input was fed directly to the RAM modules address input. This change made the data interface for accessing values after the CORDIC was finished all 16 iterations much simpler.

Testbenches and Simulations

The figure below shows the testbench for the Input Driver module. This testbench mimics typical operation of the Input Driver as it cycles through a set of x, y and z inputs in order to start CORDIC. The six button presses each have the goal of advancing the state of the finite state machine as well as setting the initial values for the CORDIC. The test shows that with each button press, the input driver stores the initial data value and increments the state (as shown by the LED value incrementing). Lastly, the test proves that the input driver correctly sends the start CORDIC signal once the inputs are ready.

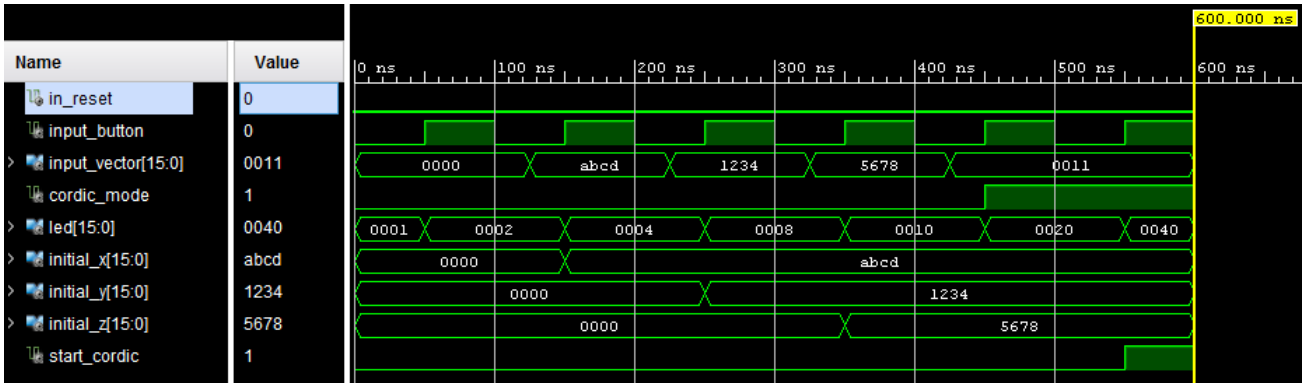


Figure 21: Input driver testbench waveform

The following figure shows the operation of the CORDIC module testbench. Once the CORDIC is triggered by a start signal, it accepts the provided input values and runs through 16 iterations. The test shows that the values of the CORDIC iterations are correct and agree with the provided test cases (further discussed in the analysis of results section). The test also shows that the CORDIC properly sends a ready signal when the iteration data is ready.

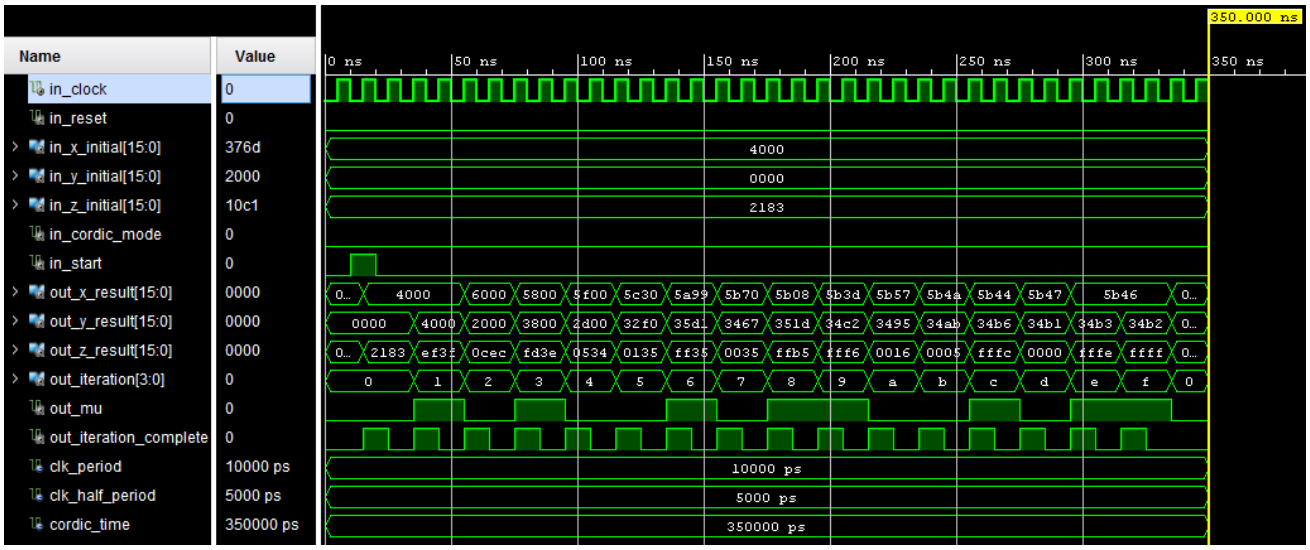


Figure 22: CORDIC testbench waveform

Additionally, from the CORDIC testbench, it can be seen that the total execution time of CORDIC is 32 clock cycles, or 320ns.

The output driver module testbench is shown in the diagram below. For the given input x, y and z values, it can be seen that the segment outputs the correct value (this can be determined using the hex driver code). The test also shows that the segment values are synced with the looping anode values as is expected.

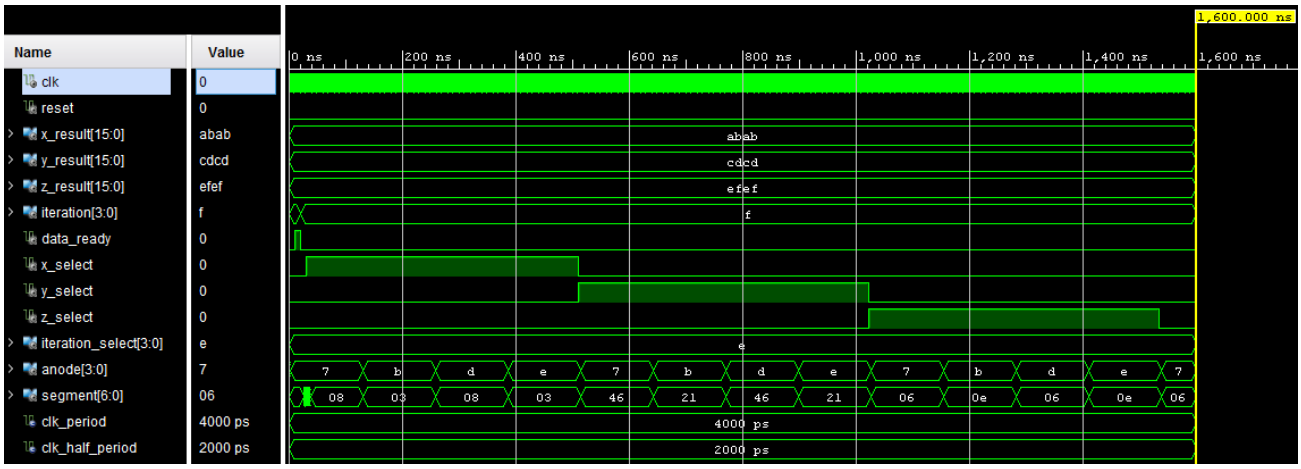


Figure 23: Output driver testbench waveform

The following figure shows a close-up view of the start of the Output Driver testbench. The anode value initially starts with all digits disabled before the first clock cycle and immediately begins entering its loop afterwards. The segment value starts in its default state until the final iteration value (indicating that the CORDIC is complete) after which it displays a default ‘0’ digit until the first value select switch signal arrives.

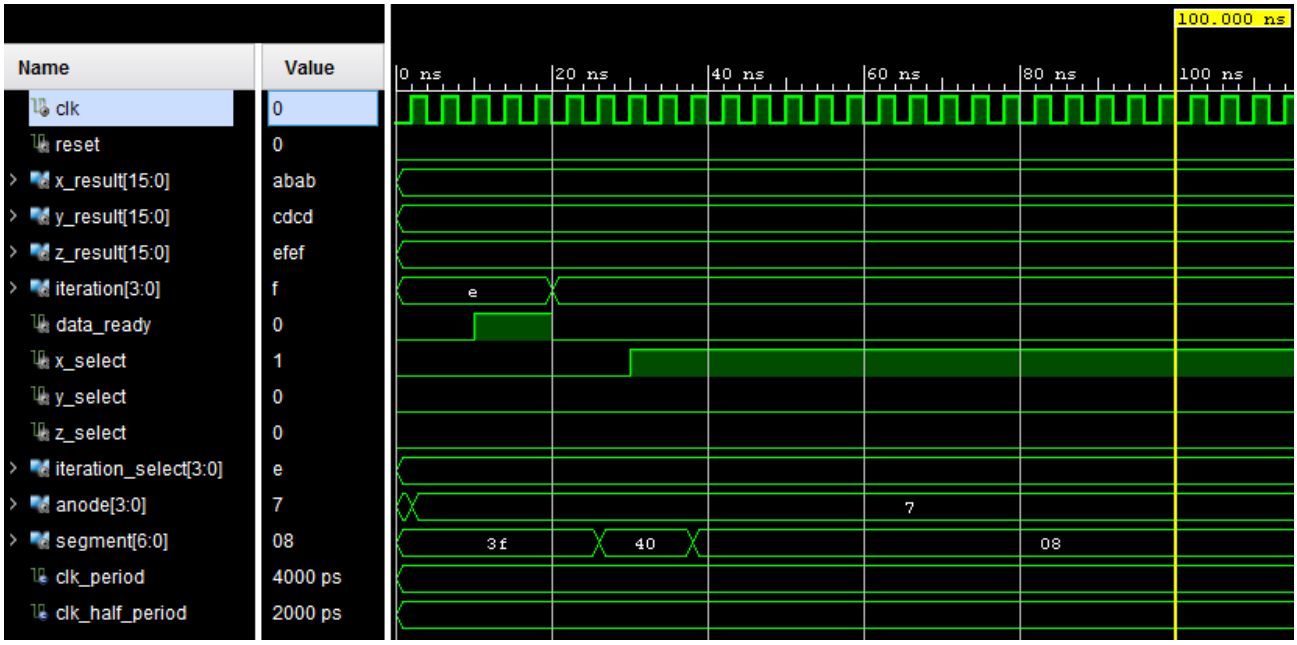


Figure 24: Output driver testbench start-up waveform

The Top module testbench shown below is a combination of the three aforementioned testbenches. The Input Driver is used to select three values from the sample CORDIC data provided, the CORDIC runs the 16 iterations on the data and then the output driver displays the results of an iteration. This test shows that the logic of the design is sound and that the design is ready for physical hardware implementation and testing.

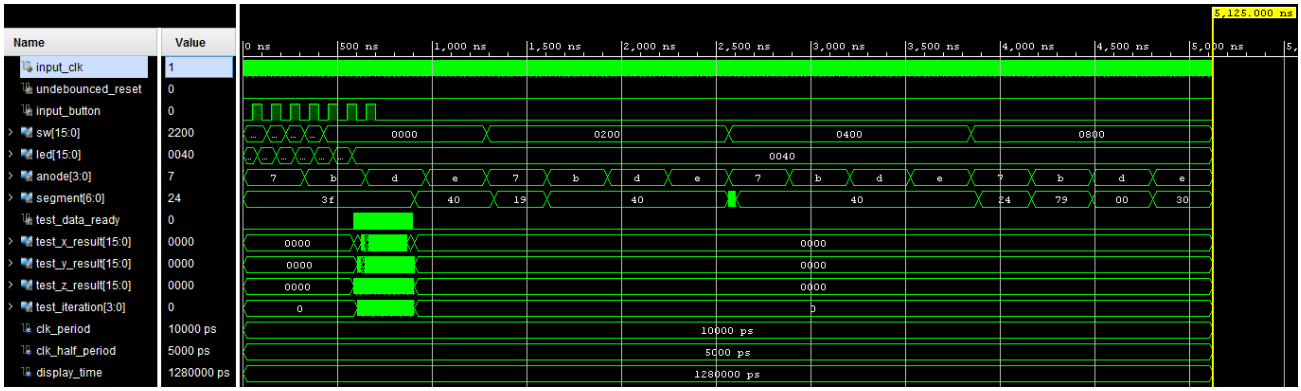


Figure 25: Top testbench full system waveform

The following testbench diagram shows the initial portion of the waveforms. The input driver runs through its expected states and properly triggers the CORDIC algorithm. Additionally, the CORDIC module shows the correct values for the provided inputs.

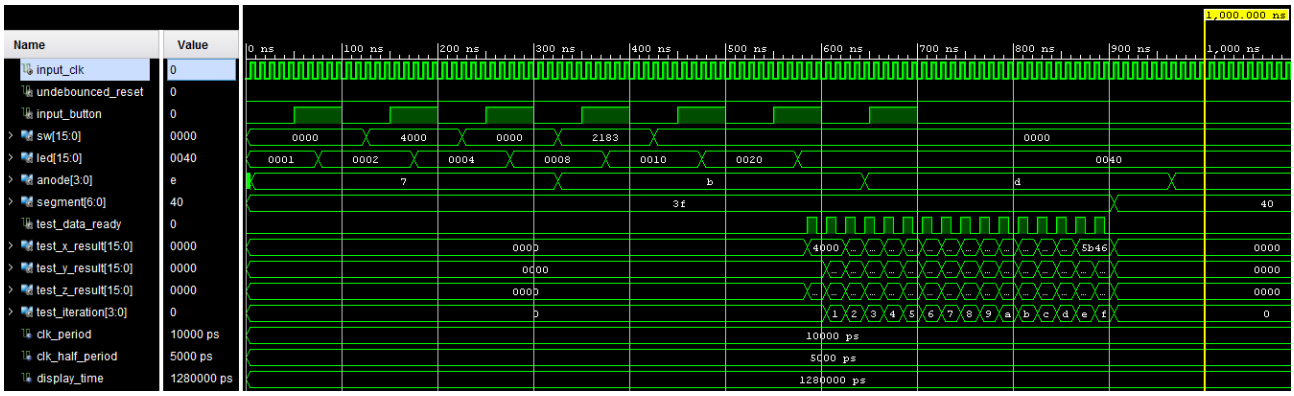


Figure 26: Top testbench input and CORDIC waveform

The following figure shows a close-up of the segment used in the display of the desired values. This part of the testbench shoes that there is potential in the design for small overlaps of the segment value that continue into the next anode cycle. This is not an issue on a practical hardware implementation because the overlap time is such a small percentage of the total display time of the digit.

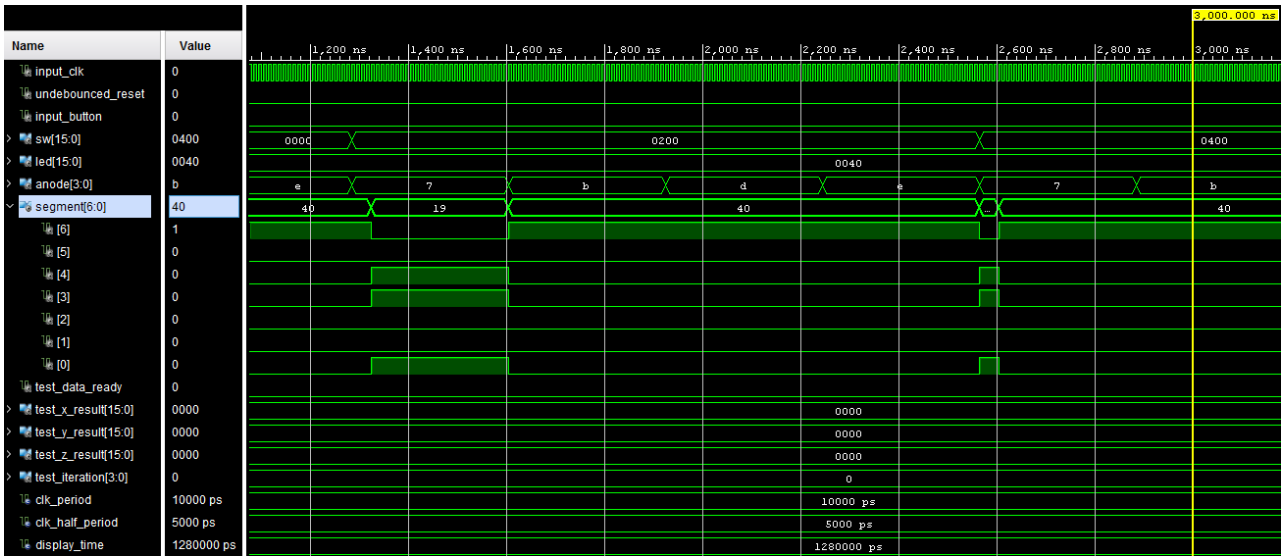


Figure 27: Top testbench display timing waveform

The various testbench waveforms generated all serve the purpose of validating the design prior to implementation. Given that the results of the testbenches all agree with the expected logic of the modules, it is safe to assume that there is a high likelihood of a successful run on hardware if the design is synthesizable and implementable.

Team Work Allocation

Initial work was done in team design meetings. To begin the project, the group met up and reviewed the specifications for the CORDIC algorithm, and the hardware implementation. We discussed the various implementation options and weighed the benefits of different module structures. We then started a basic system design, creating a block diagram starting from the

CORDIC module. Interfacing requirements were made as a team to ensure correct communication between the modules. Once the first draft of the design was complete, implementation began, with each team member developing one module. Depending on the current needs of the group, and on any revisions that needed to be made to the design, further design meetings were held to agree on a solution and reassign work as needed. All team member contributed to the editing and formatting of this report.

Alex Cote

Alex was in charge of designing the CORDIC module. Initially, this was limited to the controller, but expanded to include the ALU (see implementation discussion of CORDIC above). He also developed testbenches for this module. Alex was also entrusted with fixing any errors during the many attempts at "implementation" in Xilinx Vivado, until it was finally programmable onto the FPGA. Alex contributed to this report by writing the formal design documents and design section for the CORDIC module, and the Hardware Performance section.

Noah Rondeau

Noah was initially in charge of the CORDIC ALU when it was still conceived as a VHDL entity. He wrote the entity code and testbenches to ensure the correctness of the mathematical operations. After the ALU logic was integrated directly into the CORDIC controller to form the CORDIC Module (see implementation discussion of CORDIC above), he created the second revision of the input driver, entirely rewriting it and testbenching it. Noah contributed the Introduction, Problem Statement, Literature Review and Input Driver design documents and section for this report.

Jake Vidal

Jake was initially in charge of the Output Driver module and the Input Driver module. He completed the output driver and the testbench for it. When it was decided to change to a different input method, he took charge of writing the Top module; he was also in charge of testing it with a testbench. Jake ensured that the synthesis worked as expected. Jake also programmed and tested the hardware for the first time. Jake contributed the Top Module Architecture Design Discussion and documents, Output Driver Design Discussion and documents, Output Driver Implementation Discussion, and the Analysis of results section.

Hardware Performance

The following diagrams shows all the sub-modules included in the top module as well as their interconnections. Two close-up diagrams show a view of the input and output signal names as well as the ports connected by the signals.

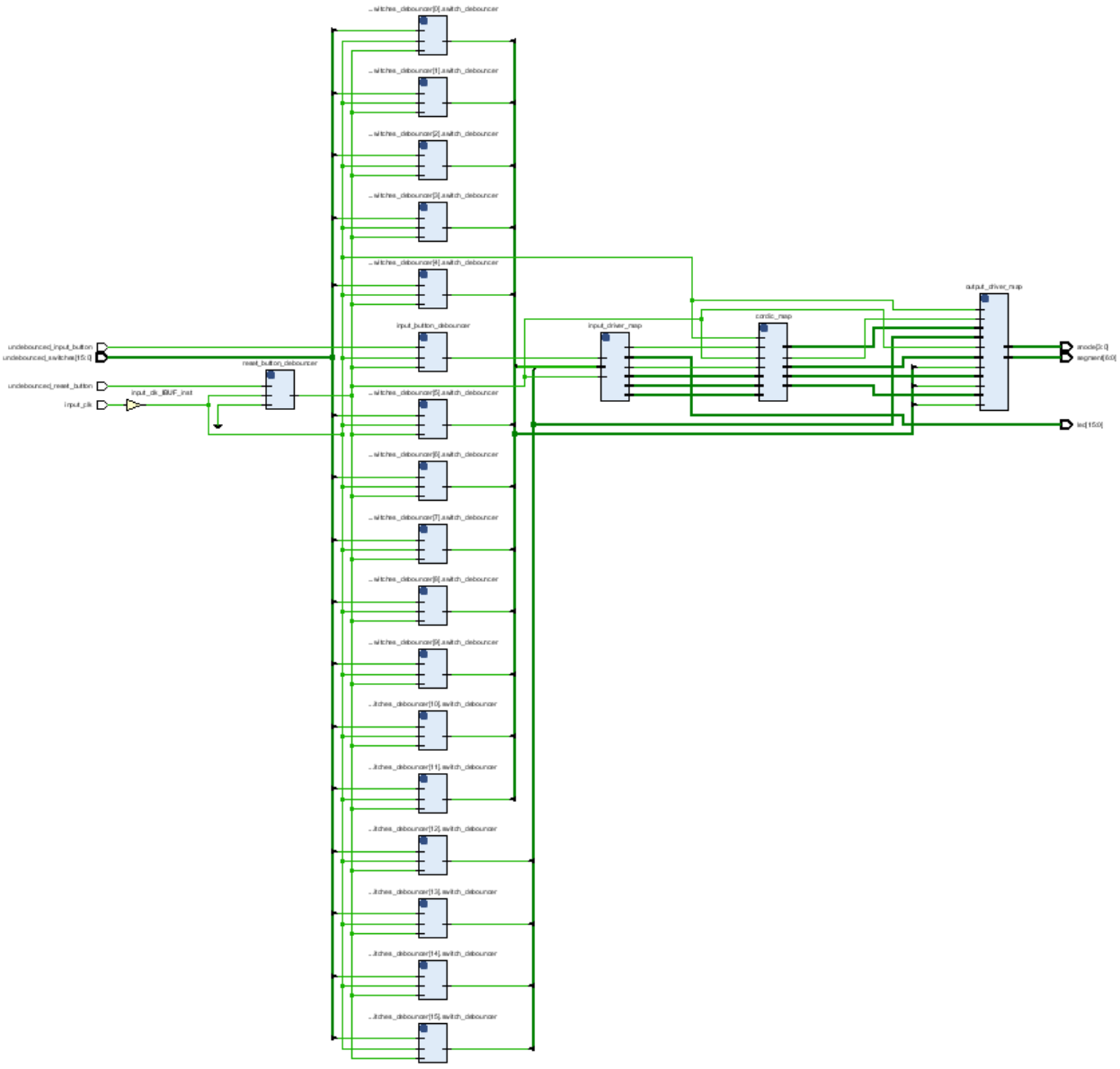


Figure 28: RTL Schematic for the Entire System

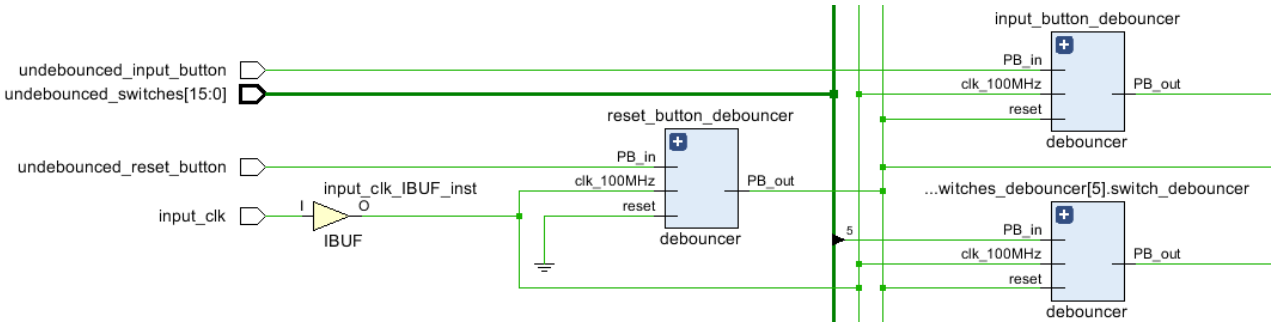


Figure 29: RTL Schematic zoomed in on the input signal debouncers.

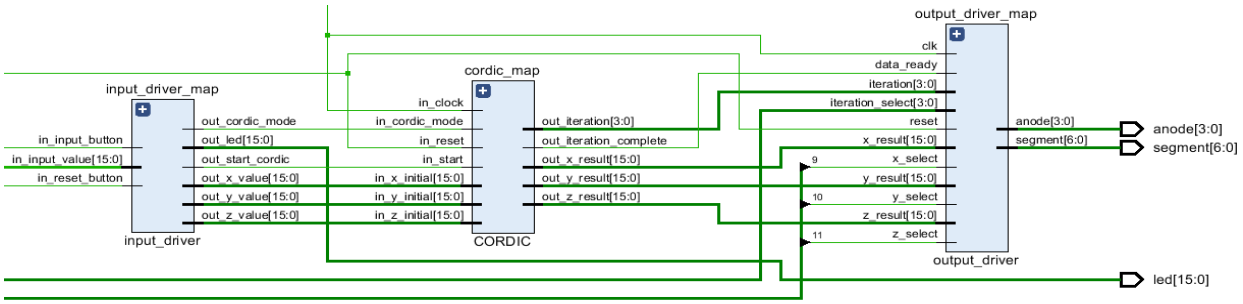


Figure 30: RTL Schematic of the three primary module.

Once the design was completed, an implementation was performed in order to map the logic onto the FPGA blocks available on the processor. The following figure shows the hardware elements selected by the Vivado router and their positions.

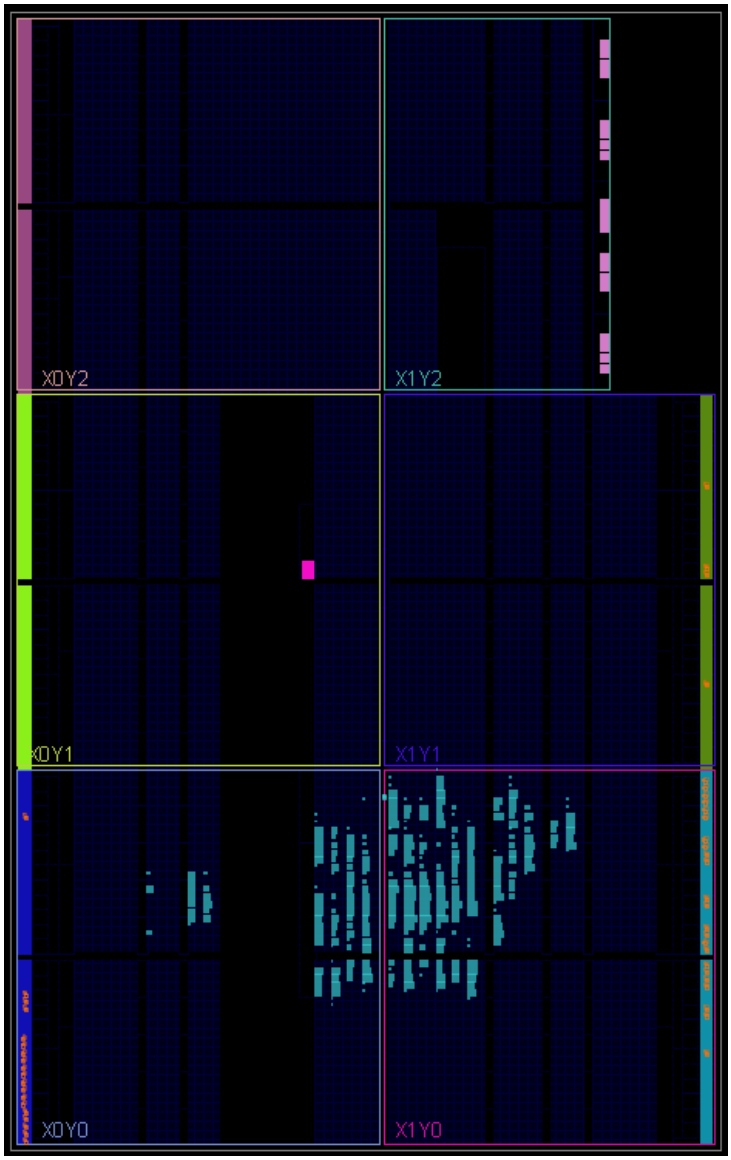


Figure 31: FPGA Device Utilization Diagram after Implementation.

The implementation of the design also produced a timing analysis of the clock signal in order to ensure that realistic constraints do not interfere with the designed logic. As seen in the figure below, the worst-case slack in the design is less than 5ns. This is important because the design uses a 100MHz clock signal to drive the synchronous logic. This 100MHZ signal is low for 5ns and then goes high for the next 5ns. Since the worst-case slack is less than half of the period, no logical errors should occur due to delay.

Clock: sys_clk_pin

Statistics

Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	3.373 ns	0.000 ns	0	1908
Hold	0.033 ns	0.000 ns	0	1908
Pulse Width	3.750 ns	0.000 ns	0	915

Figure 32: Clock Signal Timing Statistics

Another feature of Vivado's analysis of the implemented design is the power usage statistics. As shown in the diagram below, the design primarily uses static power, likely due to the large number of values stored in LUTs and RAM. The static power is the power used in order to hold logic gates at a set value, while the dynamic power is the power used in the process of switching logic gates.

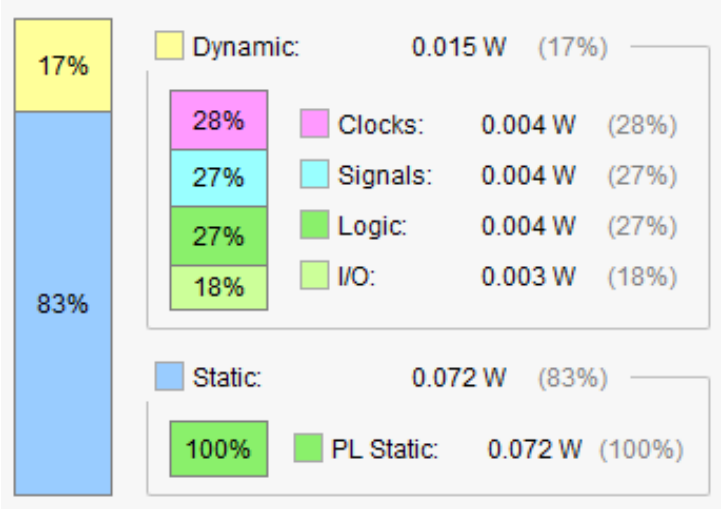


Figure 33: Total Power Usage of the Design

The utilization of the FPGA blocks on the processor is also reported by Vivado. The design uses a significant amount of the available I/O blocks compared to the other types of blocks. This is because there are only a set number of physical I/O inputs on the BASYS board. It is expected that if almost 50% of the interactive elements on the BASYS board are used, that a similar percentage of the hardware connected to those elements will be required.

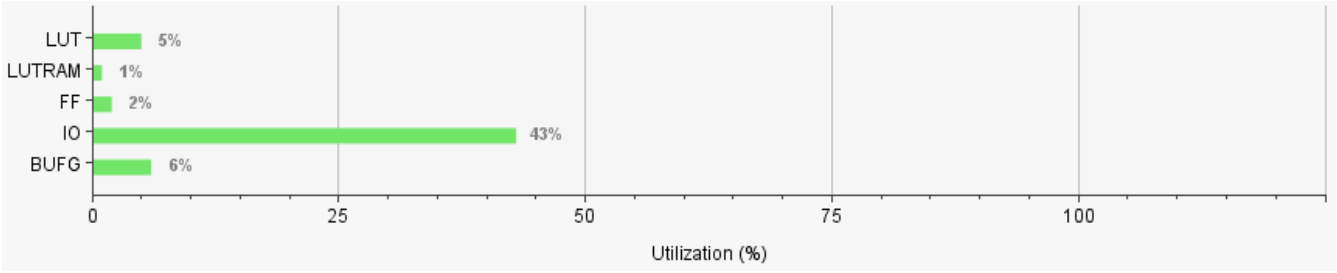


Figure 34: Design FPGA Block Usage

A breakdown of the usage of the FPGA blocks by module is shown below. This figure is useful in the analysis of the efficiency of the design. From the data, it appears that the CORDIC only requires roughly 1/5th of the FPGA blocks on the processor. This shows the efficiency of the CORDIC algorithm and also highlights the issue of user interface and data storage. Since the project problem statement requires a user to be able to input and read values, there is a significant hardware overhead. In a realistic system, the user interface elements may be implemented in software in order to preserve valuable hardware resources.

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)
top	1039	929	473	991	48	571
cordic_map (CORDIC)	209	156	67	209	0	102
output_driver_map (output_driver)	96	98	30	48	48	88
input_driver_map (input_driver)	2	63	23	2	0	1
reset_button_debouncer (debouncer_16)	39	34	20	39	0	17
input_button_debouncer (debouncer_15)	33	34	18	33	0	17
generate_switches_debouncer[9].switch_debouncer (debouncer_14)	45	34	21	45	0	17
generate_switches_debouncer[8].switch_debouncer (debouncer_13)	44	34	22	44	0	17
generate_switches_debouncer[7].switch_debouncer (debouncer_12)	50	34	23	50	0	34
generate_switches_debouncer[6].switch_debouncer (debouncer_11)	34	34	19	34	0	17
generate_switches_debouncer[5].switch_debouncer (debouncer_10)	53	34	24	53	0	33
generate_switches_debouncer[4].switch_debouncer (debouncer_9)	34	34	19	34	0	17
generate_switches_debouncer[3].switch_debouncer (debouncer_8)	36	34	19	36	0	17
generate_switches_debouncer[2].switch_debouncer (debouncer_7)	37	34	19	37	0	18
generate_switches_debouncer[1].switch_debouncer (debouncer_6)	35	34	19	35	0	17
generate_switches_debouncer[15].switch_debouncer (debouncer_5)	45	34	21	45	0	18
generate_switches_debouncer[14].switch_debouncer (debouncer_4)	38	34	20	38	0	18
generate_switches_debouncer[13].switch_debouncer (debouncer_3)	38	34	20	38	0	18
generate_switches_debouncer[12].switch_debouncer (debouncer_2)	34	34	19	34	0	17
generate_switches_debouncer[11].switch_debouncer (debouncer_1)	34	34	20	34	0	17
generate_switches_debouncer[10].switch_debouncer (debouncer_0)	51	34	25	51	0	34
generate_switches_debouncer[0].switch_debouncer (debouncer)	52	34	26	52	0	33

Figure 35: Design FPGA block usage broken down by module

Once the design was implemented, it was programmed to the FPGA and run in order to test its final behavior. The process of testing the design on hardware proved somewhat difficult due to the lack of ability to observe key variables during the process of running the design. This was

solved through careful analysis of the VHDL code as well as testing of the signal debouncing through the use of an oscilloscope.

Analysis of Results

Provided to the group were eight sets of data in order to validate the accuracy and performance of the CORDIC module. Four of these sets were for rotation mode, and the other four were for vectoring mode. They were generated using MATLAB. Shown below is one data set for rotation, and another data set for vectoring, with our simulation results shown after.

Table 6: Data to Test the 16-Bit Circular CORDIC Rotation Mode. Selection 0, Initial vector (0.5, 0) rotate anticlockwise by 30°.

Iteration	X (Hex)	Y (Hex)	Z (Hex)	Mu
0	4000	0	2183	1
1	4000	4000	EF3F	-1
2	6000	2000	0CEB	1
3	5800	3800	FD3D	-1
4	5F00	2D00	533	1
5	5C30	32F0	134	1
6	5A99	35D2	FF34	-1
7	5B70	3467	34	1
8	5B07	351E	FFB4	-1
9	5B3C	34C3	FFF4	-1
10	5B56	3495	14	1
11	5B49	34AC	4	1
12	5B43	34B8	FFFC	-1
13	5B46	34B2	0	1
14	5B44	34B5	FFFE	-1
15	5B45	34B3	FFFF	-1
	5B46	34B3	0000	

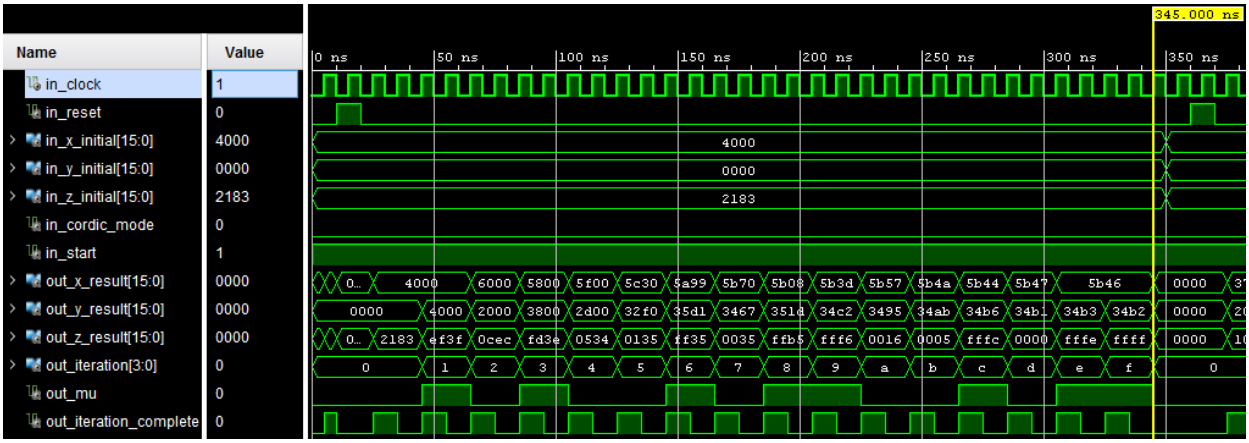


Figure 36: CORDIC Simulation results for Selection 0 of Rotation Mode CORDIC

As the primary goal of CORDIC is reaching the final value, comparing the end results provides the best description of accuracy. Here, the X data matches perfectly, while the Y and Z values are a single bit off. This can be accounted for in the accuracy used for calculating the theta values, which can change the final value slightly. Additionally, with MATLAB providing the data to compare against, the achieving bit accuracy is extremely challenging, as MATLAB data types can have a large effect on errors carrying through. With the rotation mode operating within a tolerable error range, the vectoring mode can be tested, as seen below.

Table 7: Data to Test the 16-Bit Circular CORDIC Vectoring Mode. Selection 0, Initial vector (0, 0.5).

Iteration	X (Hex)	Y (Hex)	Z (Hex)	Mu
0	0	4000	0	-1
1	4000	4000	3244	-1
2	6000	2000	4FF0	-1
3	6800	800	5F9E	-1
4	6900	FB00	6794	1
5	6950	190	6395	-1
6	695D	FE45	6595	1
7	6963	FFEB	6495	1
8	6964	00BE	6415	-1
9	6964	54	6455	-1
10	6964	20	6475	-1
11	6965	5	6485	-1
12	6965	FFF8	648D	1

13	6965	FFFF	6489	1
14	6965	2	6487	-1
15	6965	0	6488	-1
	6965	0	6488	

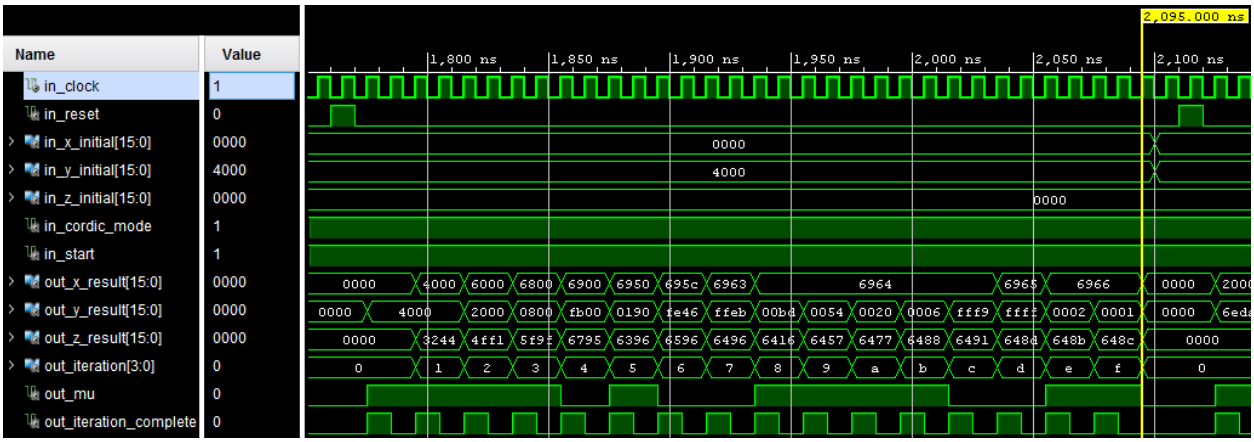


Figure 37: CORDIC Simulation results for Selection 0 of Rotation Mode CORDIC

As was done with the rotation mode, comparing the final values of vectoring will best describe the accuracy. Here, the X and Y value are off by one bit, while the Z value is off by 4 bits. This is a much larger error than with rotation. Going back to the start, iteration 2 is already off by one bit for the Z value. This seems to indicate that the accuracy used in the theta values for the reference data wasn't the same as the one used in the VHDL implementation. However, being within 4 bits of the reference data provides conclusive evidence that the VHDL implementation of CORDIC is successful in providing accurate data.

Conclusions and Recommendations

After implementing the CORDIC processor design as described, the system was successfully able to achieve correct computation of results. The accurate and repeated results show that the design is a valid solution to the problem statement of the project. Despite the success of the design, there are many future improvements that are possible.

One issue with the user interface of the Output Driver is that it can be difficult to tell the iteration being selected without converting the positions of the switches to an integer. This issue could be fixed by displaying the currently selected iteration using the 16 available LEDs on the BASYS board which, currently are unused while the CORDIC outputs are being displayed. Another possible improvement with the user interface involves the input of data prior to running CORDIC. Adding a decoder for the PMOD keypad would allow the user to simply type the hex value of the CORDIC input into the system rather than entering the values in binary.

Besides changes to the user interface, the logical and performance capabilities of the design could be improved. One simple change in this vein would be to increase the clock speed from

100MHz to the maximum value accepted by the BASYS board, 450MHz. This change however, would not be simple as it would require a more stringent set of timing restrictions that may prove difficult to meet. Another performance improvement to the design would be to add in a built-in-self-test for the theta look-up table. This would increase the reliability of the design and prevent stray bit-flips from causing incorrect results.

A final future direction to take the project would be the addition of an interpreter for the CORDIC algorithm results. This interpreter would convert the inputs and outputs of the CORDIC processor into human-readable mathematical functions, similar to a hand-held calculator.

References

[1] Gebali, F. Fayez, 'CORDIC Algorithm', 2019. [Online]. Available: https://www.ece.uvic.ca/~fayez/courses/441/project/cordic_circ.pdf [Accessed 30th Mar 2019]

[2] Andraka, R., "A survey of CORDIC algorithm for FPGA based computers", Proceedings of the ACM/SIGDA, pages 191-200, 1998.

[3] Digilentinc.com, 'Basys-3 reference manual', 2017. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual [Accessed 30th Mar 2019]

[4] Gebali, F. Fayez, 'Circular CORDIC Design Specifications', 2019. [Online]. Available: https://www.ece.uvic.ca/~fayez/courses/441/project/project.html [Accessed 30th Mar 2019]

[5] Gebali, F. Fayez, 'Circular CORDIC Final Report Specifications', 2019. [Online]. Available: https://www.ece.uvic.ca/~fayez/courses/441/project/project.html [Accessed 30th Mar 2019]

[6] R. Xu, Z. Jiang, H. Huang, C. Dong 'An Optimization of CORDIC Algorithm and FPGA Implementation', International Journal of Hybrid Information Technology Vol. 8, No.6, pp.217-228, 2015. [Online]. Available: https://pdfs.semanticscholar.org/1c30/4394f7b6e04ec9aac27c495231a787176bf1.pdf?_ga=2.213471958.399734865.1553822395-1858563035.1553822395 [Accessed 28th Mar 2019]

[7] P. Revathi, M.V. Nageswara Rao, G.R. Locharla, 'Architecture Design and FPGA Implementation of CORDIC Algorithm for Fingerprint Recognition Applications', Procedia Technology, Vol. 6, pp. 371-378, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212017312005890 [Accessed 28th Mar 2019]

[8] Digilentinc.com, 'PMOD kypd 16 button keypad', 2011. [Online]. Available: https://store.digilentinc.com/pmod-kypd-16-button-keypad/ [Accessed 30th Mar 2019]

[9] Gebali, F. Fayez, 'Design for Testability Lecture Slides, 2019. [Online]. Available: https://www.ece.uvic.ca/~fayez/courses/441/project/project.html [Accessed 30th Mar 2019]

[10] I. Guwahati, 'Module 11: Build in Self test (BIST)', NPTEL, 2013. [Online]. Available: https://nptel.ac.in/courses/106103016/30 [Accessed 30th Mar 2019]

[11] Compaq, HP, Intel, 'Universal Serial Bus Specification 2.0', Rev 2.0, 2000. [Online] Available: http://sdphca.ucsd.edu/lab_equip_manuals/usb_20.pdf [Accessed 30th Mar 2019]

[12] Evgeni, 'Parallel CRC Generator', 2009 [Online]. Available: http://outputlogic.com/?p=158 [Accessed 30th Mar 2019]

Appendix A.1: Design of BIST

The CORDIC algorithm for a fixed data-length is typically implemented, as it is in this project, using a lookup table of pre-calculated rotation angles. Each angle is given by $\theta_i = \arctan(2^{-i})$, where i is the iteration number.

The lookup table can be implemented by many methods, for example, combinatorial logic for a small table, or ROM for a large table. In either case, this dependence on pre-calculated and stored values presents a very clear potential point of failure in the system.

A data corruption in the LUT would be a very difficult problem to diagnose in a deployed product. In isolation, it would not cause system-wide failure, but only corruption of the results. In the best case, it could cause off-by-one errors, or at worst, completely incorrect results. Since CORDIC is used in many high-performance applications such as avionics hardware, radar, and control systems, the results themselves may be critical to the safety of the entire system.

For this reason, it is highly desirable to implement a built-in self-test (BIST) to verify the health of the LUT values. The design presented in this Appendix represents the most important recommendation for future improvements to this system.

General Considerations for a BIST Design

A BIST for internal storage elements is typically [9] implemented by performing a computation on the data under test (DUT) and comparing the result to the expected solution, known as the Golden Signature. This requires storing the Golden Signature for any data to be tested. The most widely used system for this is a cyclic redundancy check (CRC). This algorithm involves computing the remainder of a polynomial long division, where the individual bits of the DUT are used as the coefficients of the input polynomial. The divisor polynomial is chosen in advance to meet certain criteria. Specific CRC's are characterized by the chosen bit-length and polynomial coefficients they use. A consideration in choosing a CRC scheme [10] is how to partition the DUT, as well as the size of the DUT versus the size of the CRC. The individual coefficients of the polynomial also contribute to the types of errors that can be detected; some CRCs can only detect off-by-one errors, but not off-by-two errors. Note that a n-bit CRC uses an (n+1)-bit polynomial.

Some designs run a BIST at start-up [10]. From a team member's experience in industry, many systems also perform BIST periodically during operation, or even against every computation. Systems verifying the correctness of received data (e.g. USB 2.0 [11]) compute CRCs against every received data packet, with the golden signature included after the packet data for comparison. A trade-off between ensuring correctness and simplicity of design will have to be made for this system.

When using CRC for BIST, it is typical to use a linear feedback shift register (LSFR) to compute the signature [9,10]. This is a very straightforward solution for systems dealing with data streams, such as communications protocols (e.g. USB [11]). This is a very economical choice in hardware, as it only requires a single shift register of the same length as CRC. However, this requires shifting each bit of data into the LFSR one clock-cycle at a time. For implementations where the primary constraint is timing, not hardware use, the logic of an LFSR can be

reproduced using combinatorial logic to achieve computation in a single clock cycle. This can only be used if all the data is buffered for immediate use; this requires therefore large hardware buffers. This technique is known as parallel CRC generation [12].

Choosing When to Run BIST in this Implementation of CORDIC

CORDIC is the basis for many high-precision computations is many security and safety-sensitive fields. When considering whether to run BIST at system start-up or periodically, this is an important factor to consider. A bit-flip or other data corruption could occur at any time, not just before start-up. In a safety-critical application such as avionics, where every computation counts, it is not only beneficial, but potentially critical to know that each computation will be flagged as incorrect. The iterative nature of CORDIC means that the LUT accepts the iteration number as its input. These facts together suggest that the best solution for this system is to perform a BIST for every complete CORDIC cycle, and flag the computation result as either pass or fail.

Choosing a CRC for Implementation in CORDIC

An appropriate bit-length for the CRC must be chosen. Because in our system, only sixteen 16-bit values must be stored, it would be advantageous to treat all 256 bits + 16 bits (for signature) as a single stream of data, and to store a golden signature for the entire LUT output. Compared to the alternative (storing a signature for each LUT row), this is economical. To see this, consider computing a 16-bit CRC over 256-bits of data: this requires storing a single 16-bit golden signature. Now consider applying one 3-bit CRC per 16-bit datum; this requires storing a total of 48 bits of signature. It is clear that for the small data size in this CORDIC application, treating all 256 bits of data as a single DUT is the superior choice.

A commonly used CRC length in industry is 16 bits. Among 16-bit CRCs, the CRC-16-IBM is quite common; among other applications, it is used to check data correctness in the USB 2.0 standard [11]. This CRC has 100% coverage for all single-bit and double-bit errors. The USB 2.0 packet size is always of comparable size to the CORDIC LUT, and can therefore be considered a good match.

Its polynomial is given by

$$g(x) = x^{16} + x^{15} + x^2 + 1$$

or, in binary, this is the field

11000000000000101 (17-bit representation of the polynomial) also represented as 1000000000000101 (16-bit representation dropping the implicit x^{16} term).

The design will use this CRC.

Choosing a CRC Computation Method

Using a traditional LFSR, the CRC computation would take as many clock-cycles as there are bits in the DUT. The CORDIC module designed here, on the other hand, is capable of produces an entire computation in 16 clock cycles. Ideally, a BIST should not cause a computation to take multiples of its execution time to complete.

In order to allow the CORDIC to produce its result and a pass/fail BIST signal without significantly increasing execution time, it is clear that a traditional LSFR design cannot be used. A parallel CRC computation will be used instead. This comes with a significant combinatorial logic hardware overhead. However, because the CORDIC implementation is already quite compact, we consider this overhead to be a reasonable trade-off to ensure correctness.

Design Documents for Implementing BIST in CORDIC

The design of the BIST requires a few changes to the CORDIC module.

Firstly, an extra output must be added to indicate the pass/fail condition. Second, an entity containing the BIST must be added to the internal implementation of CORDIC.

The BIST entity receives the iteration number and the 16-bit datum from the LUT as an input. It must have a reset signal input and a start signal input. It outputs a simple pass/fail signal.

A block diagram of where the BIST module fits within CORDIC is shown below.

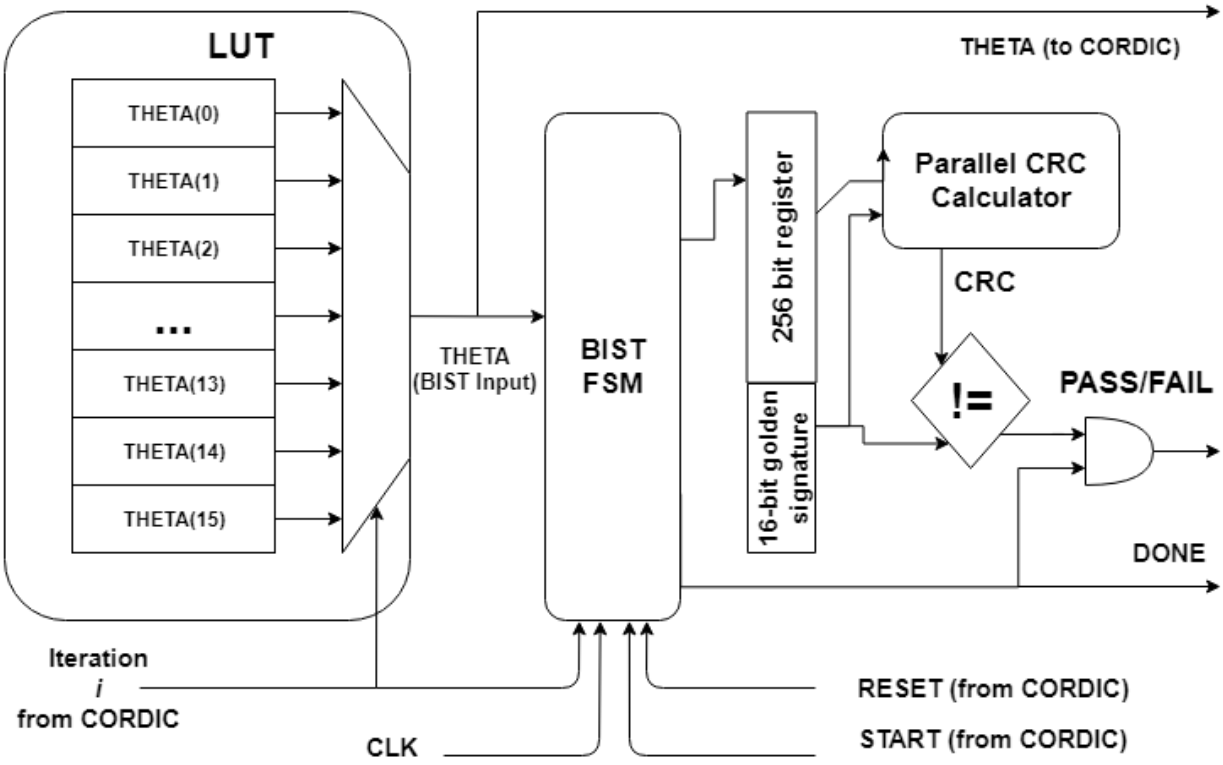


Figure 38: BIST Architecture Block Diagram

In this system, the values of θ are still allowed to pass from the LUT directly to the enclosing CORDIC module. The START and RESET signals are passed directly from the CORDIC input to the BIST input. The BIST controller, implemented as an FSM, controls timing of signal propagation and CRC computation within the BIST module (the multiplexing logic is not shown, to preserve the clarity of the interconnections).

In this system, the role of the test pattern generator common in BIST designs [9] is the CORDIC itself. The test pattern is the sequence of iteration values.

The input, output, and internal signals of the BIST module are summarized in the following table.

Table A 1: BIST Module Signal Table

INPUTS		
Signal Name	Type	Function
ITER	STD_LOGIC_VECTOR (3 downto 0)	The iteration value of CORDIC. This serves as the TEST PATTERN.
RESET	STD_LOGIC	The global debounced reset signal.
START	STD_LOGIC	The START signal input to CORDIC.
CLK	STD_LOGIC	The global clock signal.
THETA	STD_LOGIC_VECTOR (15 downto 0)	The LUT(i) input data.
OUTPUTS		
Signal Name	Type	Function
PASS/FAIL	STD_LOGIC	Test pass or fail signal. Goes to '1' when BIST is complete if the test fails.
DONE	STD_LOGIC	Completion signal. Goes to '1' on completion.
INTERNAL SIGNALS		
Signal Name	Type	Function
STATE	Compound Type:	Represents the current state of the FSM: IDLE, TEST, END.
CRC	STD_LOGIC_VECTOR (16 downto 0)	Output of the CRC calculation module.
INTERNAL REGISTERS		
Signal Name	Type	Function
DUT	STD_LOGIC_VECTOR (255 downto 0)	Buffer for holding the entire DUT from the LUT before CRC computation.

Golden Signature	CONSTANT STD_LOGIC_VECTOR (15 downto 0)	The expected result of the CRC computation on the DUT. Equal to 0xBE36.

The main responsibility of the FSM is to place the LUT output for every iteration into the correct location in the DUT register. The format is as follows.

Table A 2: 256-bit DUT Register Format

255:240	**239:224**	**…**	**31:16**	**15:0**
LUT(15)	LUT(14)	…	LUT(1)	LUT(0)

Using this format, the precalculated values of θ stored in the LUT, and a CRC calculator [12], the golden signature for this data is found below.

Table A 3: Golden Signature of LUT data under CRC-16-IBM / CRC-16-USB

Format	**Hexadecimal**	**Binary**
Golden Signature	FA65	1111101001100101

Though the Golden Signature is itself not protected, an error in the signature will cause the CRC check to fail as well. In that sense, it is protected.

A flowchart for the BIST FSM is shown in the following figure.

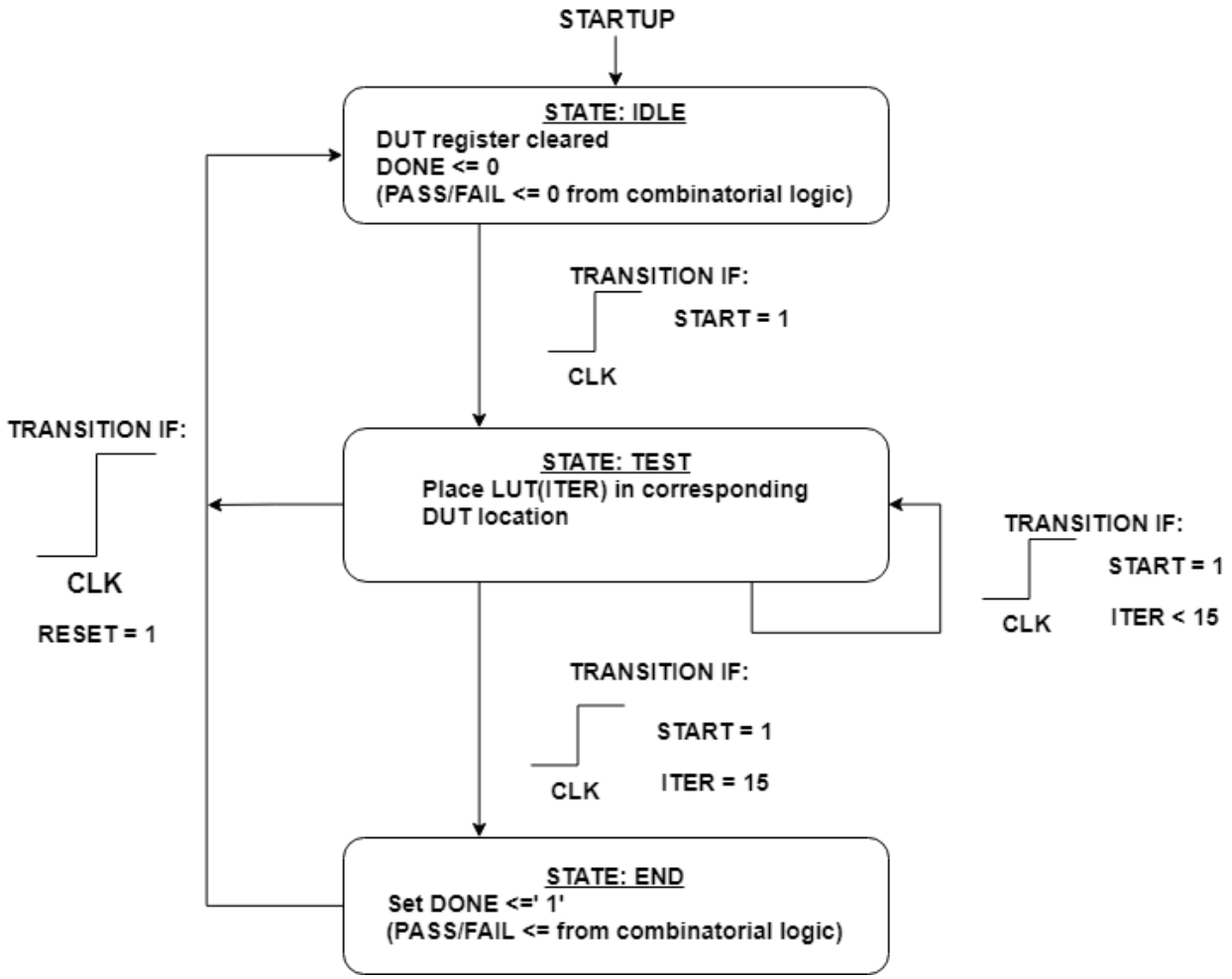


Figure 39: BIST controller FSM

The parallel CRC calculation and the internal PASS/FAIL signal are implemented as combinatorial logic, and therefore the only task of the FSM is to load the LUT data into the corresponding DUT fields and to set the DONE signal after the last iteration. The design is such as that if the CORDIC controller stalls on an iteration for whatever reason, the same value will be loaded into the DUT at the same location, preventing the corruption of the test process.

The desired timing of the BIST subsystem is shown in the figure below.

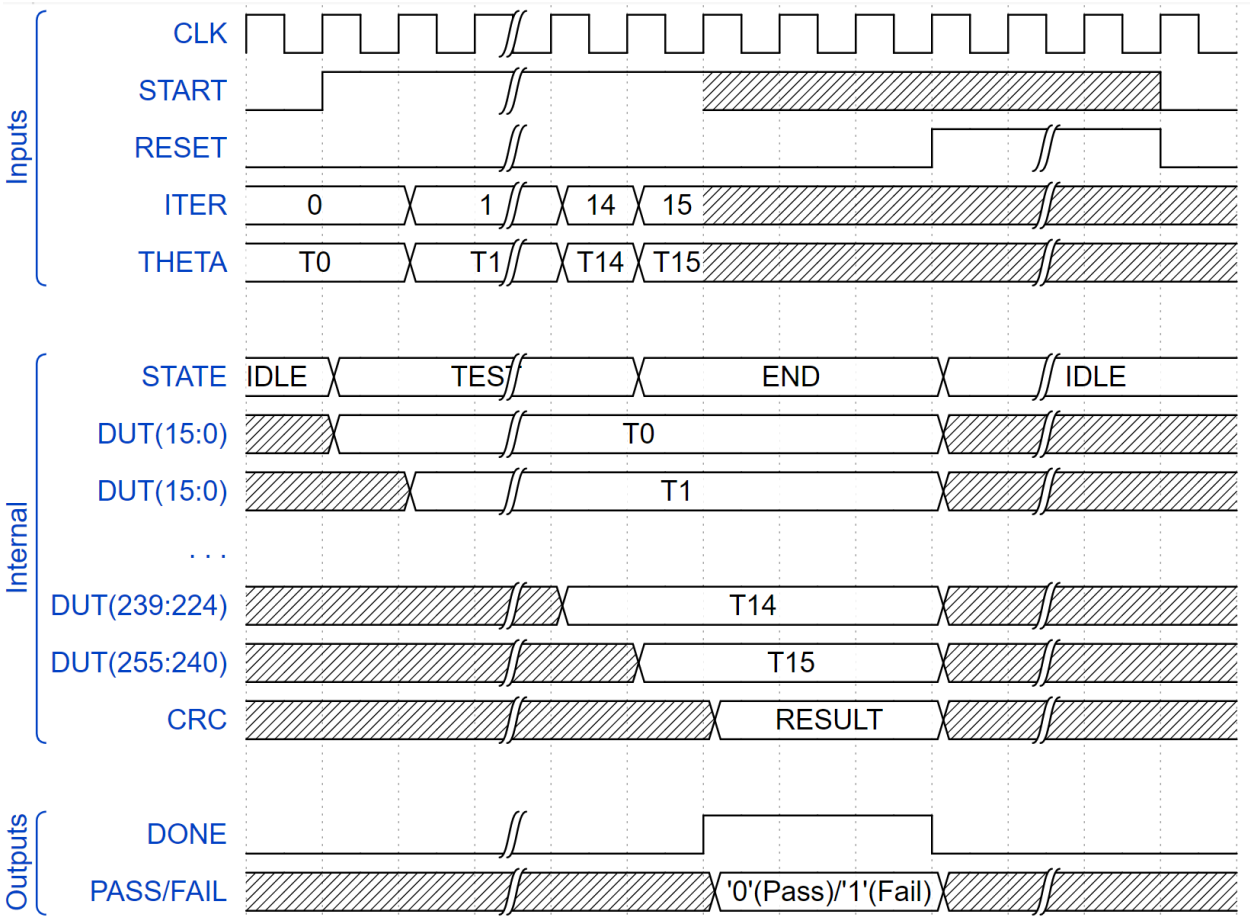


Figure 40: BIST Timing Diagram

The actual implementation of the BIST is outside the scope of this design proposal. However, VHDL code to compute a parallel CRC is widely available from online engineering tools. The code (see Appendix A.2) was generated from one such popular tool [13] and verified in a testbench.

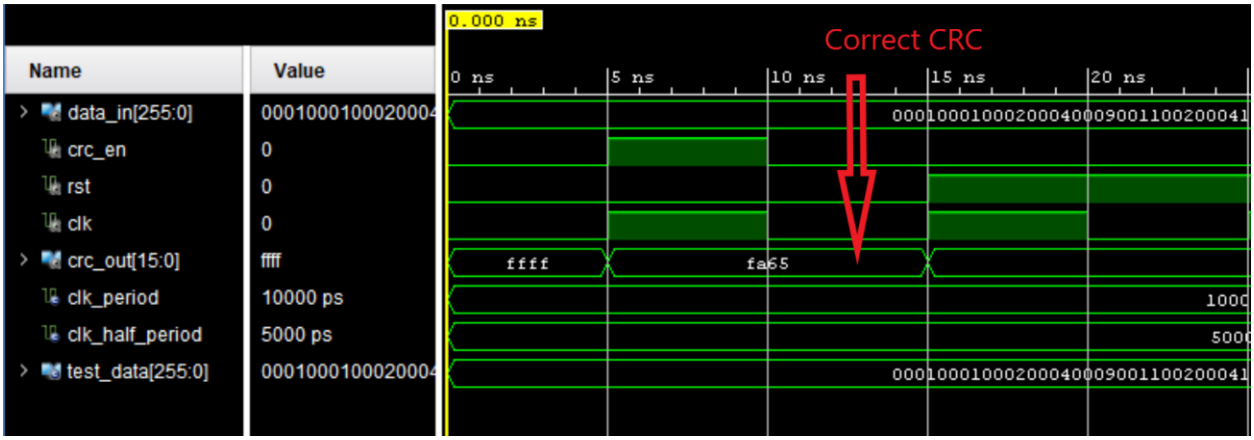


Figure 41: CRC parallel computation simulation

As is shown, the CRC is indeed generated in a single clock cycle, provided the data is already in the 256-bit register, which means that the timings described in the waveform diagram are realizable.

Summary of BIST Design

By taking into consideration the structure of the implemented CORDIC module, a compatible BIST design was developed and proposed. The design is capable of verifying the correctness of the stored angle values in the CORDIC look-up table on-the-fly, allowing each CORDIC computation to be verified in real time. The design requires a minimum of one clock cycle per iteration of CORDIC, which is indeed faster than the CORDIC itself; it will nevertheless run at any slower pace, even if irregular, that the CORDIC module might impose. It is recommended that this design be kept in consideration for future improvements to the system.

Appendix A.2: Code for CRC-16-IBM Parallel Computation

The following code was generated using the online tools on the website OutputLogic.com [13]. The code defines an entity which implements the combinatorial logic in parallel, and generates the CRC in a single clock cycle.

```
--------------------------------------------------------------------------------
-----
-- Copyright (C) 2009 OutputLogic.com
-- This source file may be used and distributed without restriction
-- provided that this copyright statement is not removed from the file
-- and that any derivative work contains the original copyright notice
-- and the associated disclaimer.
--
-- THIS SOURCE FILE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS
-- OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
-- WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
--------------------------------------------------------------------------------
-----
-- CRC module for data(255:0)
--   lfsr(15:0)=1+x^2+x^15+x^16;
--------------------------------------------------------------------------------
-----
library ieee;
use ieee.std_logic_1164.all;

entity crc16 is
  port (
    data_in     : in std_logic_vector (255 downto 0);
    crc_en      : in std_logic;
    rst         : in std_logic;
    clk         : in std_logic;
    crc_out     : out std_logic_vector (15 downto 0)
    );
end crc16;

architecture crc16 of crc16 is
  signal lfsr_q: std_logic_vector (15 downto 0) := b"1111111111111111";
  signal lfsr_c: std_logic_vector (15 downto 0);
begin
    crc_out <= lfsr_q;

    lfsr_c(0) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(2) xor lfsr_q(3) xor
lfsr_q(4) xor lfsr_q(5) xor lfsr_q(6) xor lfsr_q(7) xor lfsr_q(8) xor
lfsr_q(9) xor lfsr_q(10) xor lfsr_q(11) xor lfsr_q(15) xor data_in(0) xor
data_in(1) xor data_in(2) xor data_in(3) xor data_in(4) xor data_in(5) xor
data_in(6) xor data_in(7) xor data_in(8) xor data_in(9) xor data_in(10)
xor data_in(11) xor data_in(12) xor data_in(13) xor data_in(15) xor
data_in(16) xor data_in(17) xor data_in(18) xor data_in(19) xor
data_in(20) xor data_in(21) xor data_in(22) xor data_in(23) xor
data_in(24) xor data_in(25) xor data_in(26) xor data_in(27) xor
data_in(30) xor data_in(31) xor data_in(32) xor data_in(33) xor
data_in(34) xor data_in(35) xor data_in(36) xor data_in(37) xor
```

```
data_in(38) xor data_in(39) xor data_in(40) xor data_in(41) xor
data_in(43) xor data_in(45) xor data_in(46) xor data_in(47) xor
data_in(48) xor data_in(49) xor data_in(50) xor data_in(51) xor
data_in(52) xor data_in(53) xor data_in(54) xor data_in(55) xor
data_in(60) xor data_in(61) xor data_in(62) xor data_in(63) xor
data_in(64) xor data_in(65) xor data_in(66) xor data_in(67) xor
data_in(68) xor data_in(69) xor data_in(71) xor data_in(72) xor
data_in(73) xor data_in(75) xor data_in(76) xor data_in(77) xor
data_in(78) xor data_in(79) xor data_in(80) xor data_in(81) xor
data_in(82) xor data_in(83) xor data_in(86) xor data_in(87) xor
data_in(90) xor data_in(91) xor data_in(92) xor data_in(93) xor
data_in(94) xor data_in(95) xor data_in(96) xor data_in(97) xor
data_in(99) xor data_in(101) xor data_in(103) xor data_in(105) xor
data_in(106) xor data_in(107) xor data_in(108) xor data_in(109) xor
data_in(110) xor data_in(111) xor data_in(120) xor data_in(121) xor
data_in(122) xor data_in(123) xor data_in(124) xor data_in(125) xor
data_in(127) xor data_in(128) xor data_in(129) xor data_in(130) xor
data_in(131) xor data_in(132) xor data_in(133) xor data_in(135) xor
data_in(136) xor data_in(137) xor data_in(138) xor data_in(139) xor
data_in(142) xor data_in(143) xor data_in(144) xor data_in(145) xor
data_in(146) xor data_in(147) xor data_in(150) xor data_in(151) xor
data_in(152) xor data_in(153) xor data_in(155) xor data_in(157) xor
data_in(158) xor data_in(159) xor data_in(160) xor data_in(161) xor
data_in(163) xor data_in(165) xor data_in(166) xor data_in(167) xor
data_in(172) xor data_in(173) xor data_in(174) xor data_in(175) xor
data_in(180) xor data_in(181) xor data_in(183) xor data_in(184) xor
data_in(185) xor data_in(187) xor data_in(188) xor data_in(189) xor
data_in(191) xor data_in(192) xor data_in(193) xor data_in(195) xor
data_in(198) xor data_in(199) xor data_in(202) xor data_in(203) xor
data_in(206) xor data_in(207) xor data_in(211) xor data_in(213) xor
data_in(215) xor data_in(217) xor data_in(219) xor data_in(221) xor
data_in(223) xor data_in(224) xor data_in(238) xor data_in(240) xor
data_in(241) xor data_in(242) xor data_in(243) xor data_in(244) xor
data_in(245) xor data_in(246) xor data_in(247) xor data_in(248) xor
data_in(249) xor data_in(250) xor data_in(251) xor data_in(255);

    lfsr_c(1) <= lfsr_q(1) xor lfsr_q(2) xor lfsr_q(3) xor lfsr_q(4) xor
lfsr_q(5) xor lfsr_q(6) xor lfsr_q(7) xor lfsr_q(8) xor lfsr_q(9) xor
lfsr_q(10) xor lfsr_q(11) xor lfsr_q(12) xor data_in(1) xor data_in(2) xor
data_in(3) xor data_in(4) xor data_in(5) xor data_in(6) xor data_in(7) xor
data_in(8) xor data_in(9) xor data_in(10) xor data_in(11) xor data_in(12)
xor data_in(13) xor data_in(14) xor data_in(16) xor data_in(17) xor
data_in(18) xor data_in(19) xor data_in(20) xor data_in(21) xor
data_in(22) xor data_in(23) xor data_in(24) xor data_in(25) xor
data_in(26) xor data_in(27) xor data_in(28) xor data_in(31) xor
data_in(32) xor data_in(33) xor data_in(34) xor data_in(35) xor
data_in(36) xor data_in(37) xor data_in(38) xor data_in(39) xor
data_in(40) xor data_in(41) xor data_in(42) xor data_in(44) xor
data_in(46) xor data_in(47) xor data_in(48) xor data_in(49) xor
data_in(50) xor data_in(51) xor data_in(52) xor data_in(53) xor
data_in(54) xor data_in(55) xor data_in(56) xor data_in(61) xor
data_in(62) xor data_in(63) xor data_in(64) xor data_in(65) xor
data_in(66) xor data_in(67) xor data_in(68) xor data_in(69) xor
data_in(70) xor data_in(72) xor data_in(73) xor data_in(74) xor
```

```
data_in(76) xor data_in(77) xor data_in(78) xor data_in(79) xor
data_in(80) xor data_in(81) xor data_in(82) xor data_in(83) xor
data_in(84) xor data_in(87) xor data_in(88) xor data_in(91) xor
data_in(92) xor data_in(93) xor data_in(94) xor data_in(95) xor
data_in(96) xor data_in(97) xor data_in(98) xor data_in(100) xor
data_in(102) xor data_in(104) xor data_in(106) xor data_in(107) xor
data_in(108) xor data_in(109) xor data_in(110) xor data_in(111) xor
data_in(112) xor data_in(121) xor data_in(122) xor data_in(123) xor
data_in(124) xor data_in(125) xor data_in(126) xor data_in(128) xor
data_in(129) xor data_in(130) xor data_in(131) xor data_in(132) xor
data_in(133) xor data_in(134) xor data_in(136) xor data_in(137) xor
data_in(138) xor data_in(139) xor data_in(140) xor data_in(143) xor
data_in(144) xor data_in(145) xor data_in(146) xor data_in(147) xor
data_in(148) xor data_in(151) xor data_in(152) xor data_in(153) xor
data_in(154) xor data_in(156) xor data_in(158) xor data_in(159) xor
data_in(160) xor data_in(161) xor data_in(162) xor data_in(164) xor
data_in(166) xor data_in(167) xor data_in(168) xor data_in(173) xor
data_in(174) xor data_in(175) xor data_in(176) xor data_in(181) xor
data_in(182) xor data_in(184) xor data_in(185) xor data_in(186) xor
data_in(188) xor data_in(189) xor data_in(190) xor data_in(192) xor
data_in(193) xor data_in(194) xor data_in(196) xor data_in(199) xor
data_in(200) xor data_in(203) xor data_in(204) xor data_in(207) xor
data_in(208) xor data_in(212) xor data_in(214) xor data_in(216) xor
data_in(218) xor data_in(220) xor data_in(222) xor data_in(224) xor
data_in(225) xor data_in(239) xor data_in(241) xor data_in(242) xor
data_in(243) xor data_in(244) xor data_in(245) xor data_in(246) xor
data_in(247) xor data_in(248) xor data_in(249) xor data_in(250) xor
data_in(251) xor data_in(252);

    lfsr_c(2) <= lfsr_q(1) xor lfsr_q(12) xor lfsr_q(13) xor lfsr_q(15)
xor data_in(0) xor data_in(1) xor data_in(14) xor data_in(16) xor
data_in(28) xor data_in(29) xor data_in(30) xor data_in(31) xor
data_in(42) xor data_in(46) xor data_in(56) xor data_in(57) xor
data_in(60) xor data_in(61) xor data_in(70) xor data_in(72) xor
data_in(74) xor data_in(76) xor data_in(84) xor data_in(85) xor
data_in(86) xor data_in(87) xor data_in(88) xor data_in(89) xor
data_in(90) xor data_in(91) xor data_in(98) xor data_in(106) xor
data_in(112) xor data_in(113) xor data_in(120) xor data_in(121) xor
data_in(126) xor data_in(128) xor data_in(134) xor data_in(136) xor
data_in(140) xor data_in(141) xor data_in(142) xor data_in(143) xor
data_in(148) xor data_in(149) xor data_in(150) xor data_in(151) xor
data_in(154) xor data_in(158) xor data_in(162) xor data_in(166) xor
data_in(168) xor data_in(169) xor data_in(172) xor data_in(173) xor
data_in(176) xor data_in(177) xor data_in(180) xor data_in(181) xor
data_in(182) xor data_in(184) xor data_in(186) xor data_in(188) xor
data_in(190) xor data_in(192) xor data_in(194) xor data_in(197) xor
data_in(198) xor data_in(199) xor data_in(200) xor data_in(201) xor
data_in(202) xor data_in(203) xor data_in(204) xor data_in(205) xor
data_in(206) xor data_in(207) xor data_in(208) xor data_in(209) xor
data_in(211) xor data_in(224) xor data_in(225) xor data_in(226) xor
data_in(238) xor data_in(241) xor data_in(252) xor data_in(253) xor
data_in(255);
    lfsr_c(3) <= lfsr_q(2) xor lfsr_q(13) xor lfsr_q(14) xor data_in(1)
xor data_in(2) xor data_in(15) xor data_in(17) xor data_in(29) xor
```

```
data_in(30) xor data_in(31) xor data_in(32) xor data_in(43) xor
data_in(47) xor data_in(57) xor data_in(58) xor data_in(61) xor
data_in(62) xor data_in(71) xor data_in(73) xor data_in(75) xor
data_in(77) xor data_in(85) xor data_in(86) xor data_in(87) xor
data_in(88) xor data_in(89) xor data_in(90) xor data_in(91) xor
data_in(92) xor data_in(99) xor data_in(107) xor data_in(113) xor
data_in(114) xor data_in(121) xor data_in(122) xor data_in(127) xor
data_in(129) xor data_in(135) xor data_in(137) xor data_in(141) xor
data_in(142) xor data_in(143) xor data_in(144) xor data_in(149) xor
data_in(150) xor data_in(151) xor data_in(152) xor data_in(155) xor
data_in(159) xor data_in(163) xor data_in(167) xor data_in(169) xor
data_in(170) xor data_in(173) xor data_in(174) xor data_in(177) xor
data_in(178) xor data_in(181) xor data_in(182) xor data_in(183) xor
data_in(185) xor data_in(187) xor data_in(189) xor data_in(191) xor
data_in(193) xor data_in(195) xor data_in(198) xor data_in(199) xor
data_in(200) xor data_in(201) xor data_in(202) xor data_in(203) xor
data_in(204) xor data_in(205) xor data_in(206) xor data_in(207) xor
data_in(208) xor data_in(209) xor data_in(210) xor data_in(212) xor
data_in(225) xor data_in(226) xor data_in(227) xor data_in(239) xor
data_in(242) xor data_in(253) xor data_in(254);

    lfsr_c(4) <= lfsr_q(0) xor lfsr_q(3) xor lfsr_q(14) xor lfsr_q(15) xor
data_in(2) xor data_in(3) xor data_in(16) xor data_in(18) xor data_in(30)
xor data_in(31) xor data_in(32) xor data_in(33) xor data_in(44) xor
data_in(48) xor data_in(58) xor data_in(59) xor data_in(62) xor
data_in(63) xor data_in(72) xor data_in(74) xor data_in(76) xor
data_in(78) xor data_in(86) xor data_in(87) xor data_in(88) xor
data_in(89) xor data_in(90) xor data_in(91) xor data_in(92) xor
data_in(93) xor data_in(100) xor data_in(108) xor data_in(114) xor
data_in(115) xor data_in(122) xor data_in(123) xor data_in(128) xor
data_in(130) xor data_in(136) xor data_in(138) xor data_in(142) xor
data_in(143) xor data_in(144) xor data_in(145) xor data_in(150) xor
data_in(151) xor data_in(152) xor data_in(153) xor data_in(156) xor
data_in(160) xor data_in(164) xor data_in(168) xor data_in(170) xor
data_in(171) xor data_in(174) xor data_in(175) xor data_in(178) xor
data_in(179) xor data_in(182) xor data_in(183) xor data_in(184) xor
data_in(186) xor data_in(188) xor data_in(190) xor data_in(192) xor
data_in(194) xor data_in(196) xor data_in(199) xor data_in(200) xor
data_in(201) xor data_in(202) xor data_in(203) xor data_in(204) xor
data_in(205) xor data_in(206) xor data_in(207) xor data_in(208) xor
data_in(209) xor data_in(210) xor data_in(211) xor data_in(213) xor
data_in(226) xor data_in(227) xor data_in(228) xor data_in(240) xor
data_in(243) xor data_in(254) xor data_in(255);

    lfsr_c(5) <= lfsr_q(1) xor lfsr_q(4) xor lfsr_q(15) xor data_in(3) xor
data_in(4) xor data_in(17) xor data_in(19) xor data_in(31) xor data_in(32)
xor data_in(33) xor data_in(34) xor data_in(45) xor data_in(49) xor
data_in(59) xor data_in(60) xor data_in(63) xor data_in(64) xor
data_in(73) xor data_in(75) xor data_in(77) xor data_in(79) xor
data_in(87) xor data_in(88) xor data_in(89) xor data_in(90) xor
data_in(91) xor data_in(92) xor data_in(93) xor data_in(94) xor
data_in(101) xor data_in(109) xor data_in(115) xor data_in(116) xor
data_in(123) xor data_in(124) xor data_in(129) xor data_in(131) xor
data_in(137) xor data_in(139) xor data_in(143) xor data_in(144) xor
```

```
data_in(145) xor data_in(146) xor data_in(151) xor data_in(152) xor
data_in(153) xor data_in(154) xor data_in(157) xor data_in(161) xor
data_in(165) xor data_in(169) xor data_in(171) xor data_in(172) xor
data_in(175) xor data_in(176) xor data_in(179) xor data_in(180) xor
data_in(183) xor data_in(184) xor data_in(185) xor data_in(187) xor
data_in(189) xor data_in(191) xor data_in(193) xor data_in(195) xor
data_in(197) xor data_in(200) xor data_in(201) xor data_in(202) xor
data_in(203) xor data_in(204) xor data_in(205) xor data_in(206) xor
data_in(207) xor data_in(208) xor data_in(209) xor data_in(210) xor
data_in(211) xor data_in(212) xor data_in(214) xor data_in(227) xor
data_in(228) xor data_in(229) xor data_in(241) xor data_in(244) xor
data_in(255);

    lfsr_c(6) <= lfsr_q(2) xor lfsr_q(5) xor data_in(4) xor data_in(5) xor
data_in(18) xor data_in(20) xor data_in(32) xor data_in(33) xor
data_in(34) xor data_in(35) xor data_in(46) xor data_in(50) xor
data_in(60) xor data_in(61) xor data_in(64) xor data_in(65) xor
data_in(74) xor data_in(76) xor data_in(78) xor data_in(80) xor
data_in(88) xor data_in(89) xor data_in(90) xor data_in(91) xor
data_in(92) xor data_in(93) xor data_in(94) xor data_in(95) xor
data_in(102) xor data_in(110) xor data_in(116) xor data_in(117) xor
data_in(124) xor data_in(125) xor data_in(130) xor data_in(132) xor
data_in(138) xor data_in(140) xor data_in(144) xor data_in(145) xor
data_in(146) xor data_in(147) xor data_in(152) xor data_in(153) xor
data_in(154) xor data_in(155) xor data_in(158) xor data_in(162) xor
data_in(166) xor data_in(170) xor data_in(172) xor data_in(173) xor
data_in(176) xor data_in(177) xor data_in(180) xor data_in(181) xor
data_in(184) xor data_in(185) xor data_in(186) xor data_in(188) xor
data_in(190) xor data_in(192) xor data_in(194) xor data_in(196) xor
data_in(198) xor data_in(201) xor data_in(202) xor data_in(203) xor
data_in(204) xor data_in(205) xor data_in(206) xor data_in(207) xor
data_in(208) xor data_in(209) xor data_in(210) xor data_in(211) xor
data_in(212) xor data_in(213) xor data_in(215) xor data_in(228) xor
data_in(229) xor data_in(230) xor data_in(242) xor data_in(245);

    lfsr_c(7) <= lfsr_q(3) xor lfsr_q(6) xor data_in(5) xor data_in(6) xor
data_in(19) xor data_in(21) xor data_in(33) xor data_in(34) xor
data_in(35) xor data_in(36) xor data_in(47) xor data_in(51) xor
data_in(61) xor data_in(62) xor data_in(65) xor data_in(66) xor
data_in(75) xor data_in(77) xor data_in(79) xor data_in(81) xor
data_in(89) xor data_in(90) xor data_in(91) xor data_in(92) xor
data_in(93) xor data_in(94) xor data_in(95) xor data_in(96) xor
data_in(103) xor data_in(111) xor data_in(117) xor data_in(118) xor
data_in(125) xor data_in(126) xor data_in(131) xor data_in(133) xor
data_in(139) xor data_in(141) xor data_in(145) xor data_in(146) xor
data_in(147) xor data_in(148) xor data_in(153) xor data_in(154) xor
data_in(155) xor data_in(156) xor data_in(159) xor data_in(163) xor
data_in(167) xor data_in(171) xor data_in(173) xor data_in(174) xor
data_in(177) xor data_in(178) xor data_in(181) xor data_in(182) xor
data_in(185) xor data_in(186) xor data_in(187) xor data_in(189) xor
data_in(191) xor data_in(193) xor data_in(195) xor data_in(197) xor
data_in(199) xor data_in(202) xor data_in(203) xor data_in(204) xor
data_in(205) xor data_in(206) xor data_in(207) xor data_in(208) xor
data_in(209) xor data_in(210) xor data_in(211) xor data_in(212) xor
```

```
data_in(213) xor data_in(214) xor data_in(216) xor data_in(229) xor
data_in(230) xor data_in(231) xor data_in(243) xor data_in(246);

    lfsr_c(8) <= lfsr_q(4) xor lfsr_q(7) xor data_in(6) xor data_in(7) xor
data_in(20) xor data_in(22) xor data_in(34) xor data_in(35) xor
data_in(36) xor data_in(37) xor data_in(48) xor data_in(52) xor
data_in(62) xor data_in(63) xor data_in(66) xor data_in(67) xor
data_in(76) xor data_in(78) xor data_in(80) xor data_in(82) xor
data_in(90) xor data_in(91) xor data_in(92) xor data_in(93) xor
data_in(94) xor data_in(95) xor data_in(96) xor data_in(97) xor
data_in(104) xor data_in(112) xor data_in(118) xor data_in(119) xor
data_in(126) xor data_in(127) xor data_in(132) xor data_in(134) xor
data_in(140) xor data_in(142) xor data_in(146) xor data_in(147) xor
data_in(148) xor data_in(149) xor data_in(154) xor data_in(155) xor
data_in(156) xor data_in(157) xor data_in(160) xor data_in(164) xor
data_in(168) xor data_in(172) xor data_in(174) xor data_in(175) xor
data_in(178) xor data_in(179) xor data_in(182) xor data_in(183) xor
data_in(186) xor data_in(187) xor data_in(188) xor data_in(190) xor
data_in(192) xor data_in(194) xor data_in(196) xor data_in(198) xor
data_in(200) xor data_in(203) xor data_in(204) xor data_in(205) xor
data_in(206) xor data_in(207) xor data_in(208) xor data_in(209) xor
data_in(210) xor data_in(211) xor data_in(212) xor data_in(213) xor
data_in(214) xor data_in(215) xor data_in(217) xor data_in(230) xor
data_in(231) xor data_in(232) xor data_in(244) xor data_in(247);

    lfsr_c(9) <= lfsr_q(5) xor lfsr_q(8) xor data_in(7) xor data_in(8) xor
data_in(21) xor data_in(23) xor data_in(35) xor data_in(36) xor
data_in(37) xor data_in(38) xor data_in(49) xor data_in(53) xor
data_in(63) xor data_in(64) xor data_in(67) xor data_in(68) xor
data_in(77) xor data_in(79) xor data_in(81) xor data_in(83) xor
data_in(91) xor data_in(92) xor data_in(93) xor data_in(94) xor
data_in(95) xor data_in(96) xor data_in(97) xor data_in(98) xor
data_in(105) xor data_in(113) xor data_in(119) xor data_in(120) xor
data_in(127) xor data_in(128) xor data_in(133) xor data_in(135) xor
data_in(141) xor data_in(143) xor data_in(147) xor data_in(148) xor
data_in(149) xor data_in(150) xor data_in(155) xor data_in(156) xor
data_in(157) xor data_in(158) xor data_in(161) xor data_in(165) xor
data_in(169) xor data_in(173) xor data_in(175) xor data_in(176) xor
data_in(179) xor data_in(180) xor data_in(183) xor data_in(184) xor
data_in(187) xor data_in(188) xor data_in(189) xor data_in(191) xor
data_in(193) xor data_in(195) xor data_in(197) xor data_in(199) xor
data_in(201) xor data_in(204) xor data_in(205) xor data_in(206) xor
data_in(207) xor data_in(208) xor data_in(209) xor data_in(210) xor
data_in(211) xor data_in(212) xor data_in(213) xor data_in(214) xor
data_in(215) xor data_in(216) xor data_in(218) xor data_in(231) xor
data_in(232) xor data_in(233) xor data_in(245) xor data_in(248);

    lfsr_c(10) <= lfsr_q(6) xor lfsr_q(9) xor data_in(8) xor data_in(9)
xor data_in(22) xor data_in(24) xor data_in(36) xor data_in(37) xor
data_in(38) xor data_in(39) xor data_in(50) xor data_in(54) xor
data_in(64) xor data_in(65) xor data_in(68) xor data_in(69) xor
data_in(78) xor data_in(80) xor data_in(82) xor data_in(84) xor
data_in(92) xor data_in(93) xor data_in(94) xor data_in(95) xor
data_in(96) xor data_in(97) xor data_in(98) xor data_in(99) xor
```

```
data_in(106) xor data_in(114) xor data_in(120) xor data_in(121) xor
data_in(128) xor data_in(129) xor data_in(134) xor data_in(136) xor
data_in(142) xor data_in(144) xor data_in(148) xor data_in(149) xor
data_in(150) xor data_in(151) xor data_in(156) xor data_in(157) xor
data_in(158) xor data_in(159) xor data_in(162) xor data_in(166) xor
data_in(170) xor data_in(174) xor data_in(176) xor data_in(177) xor
data_in(180) xor data_in(181) xor data_in(184) xor data_in(185) xor
data_in(188) xor data_in(189) xor data_in(190) xor data_in(192) xor
data_in(194) xor data_in(196) xor data_in(198) xor data_in(200) xor
data_in(202) xor data_in(205) xor data_in(206) xor data_in(207) xor
data_in(208) xor data_in(209) xor data_in(210) xor data_in(211) xor
data_in(212) xor data_in(213) xor data_in(214) xor data_in(215) xor
data_in(216) xor data_in(217) xor data_in(219) xor data_in(232) xor
data_in(233) xor data_in(234) xor data_in(246) xor data_in(249);

    lfsr_c(11) <= lfsr_q(7) xor lfsr_q(10) xor data_in(9) xor data_in(10)
xor data_in(23) xor data_in(25) xor data_in(37) xor data_in(38) xor
data_in(39) xor data_in(40) xor data_in(51) xor data_in(55) xor
data_in(65) xor data_in(66) xor data_in(69) xor data_in(70) xor
data_in(79) xor data_in(81) xor data_in(83) xor data_in(85) xor
data_in(93) xor data_in(94) xor data_in(95) xor data_in(96) xor
data_in(97) xor data_in(98) xor data_in(99) xor data_in(100) xor
data_in(107) xor data_in(115) xor data_in(121) xor data_in(122) xor
data_in(129) xor data_in(130) xor data_in(135) xor data_in(137) xor
data_in(143) xor data_in(145) xor data_in(149) xor data_in(150) xor
data_in(151) xor data_in(152) xor data_in(157) xor data_in(158) xor
data_in(159) xor data_in(160) xor data_in(163) xor data_in(167) xor
data_in(171) xor data_in(175) xor data_in(177) xor data_in(178) xor
data_in(181) xor data_in(182) xor data_in(185) xor data_in(186) xor
data_in(189) xor data_in(190) xor data_in(191) xor data_in(193) xor
data_in(195) xor data_in(197) xor data_in(199) xor data_in(201) xor
data_in(203) xor data_in(206) xor data_in(207) xor data_in(208) xor
data_in(209) xor data_in(210) xor data_in(211) xor data_in(212) xor
data_in(213) xor data_in(214) xor data_in(215) xor data_in(216) xor
data_in(217) xor data_in(218) xor data_in(220) xor data_in(233) xor
data_in(234) xor data_in(235) xor data_in(247) xor data_in(250);

    lfsr_c(12) <= lfsr_q(8) xor lfsr_q(11) xor data_in(10) xor data_in(11)
xor data_in(24) xor data_in(26) xor data_in(38) xor data_in(39) xor
data_in(40) xor data_in(41) xor data_in(52) xor data_in(56) xor
data_in(66) xor data_in(67) xor data_in(70) xor data_in(71) xor
data_in(80) xor data_in(82) xor data_in(84) xor data_in(86) xor
data_in(94) xor data_in(95) xor data_in(96) xor data_in(97) xor
data_in(98) xor data_in(99) xor data_in(100) xor data_in(101) xor
data_in(108) xor data_in(116) xor data_in(122) xor data_in(123) xor
data_in(130) xor data_in(131) xor data_in(136) xor data_in(138) xor
data_in(144) xor data_in(146) xor data_in(150) xor data_in(151) xor
data_in(152) xor data_in(153) xor data_in(158) xor data_in(159) xor
data_in(160) xor data_in(161) xor data_in(164) xor data_in(168) xor
data_in(172) xor data_in(176) xor data_in(178) xor data_in(179) xor
data_in(182) xor data_in(183) xor data_in(186) xor data_in(187) xor
data_in(190) xor data_in(191) xor data_in(192) xor data_in(194) xor
data_in(196) xor data_in(198) xor data_in(200) xor data_in(202) xor
data_in(204) xor data_in(207) xor data_in(208) xor data_in(209) xor
```

```
data_in(210) xor data_in(211) xor data_in(212) xor data_in(213) xor
data_in(214) xor data_in(215) xor data_in(216) xor data_in(217) xor
data_in(218) xor data_in(219) xor data_in(221) xor data_in(234) xor
data_in(235) xor data_in(236) xor data_in(248) xor data_in(251);

    lfsr_c(13) <= lfsr_q(9) xor lfsr_q(12) xor data_in(11) xor data_in(12)
xor data_in(25) xor data_in(27) xor data_in(39) xor data_in(40) xor
data_in(41) xor data_in(42) xor data_in(53) xor data_in(57) xor
data_in(67) xor data_in(68) xor data_in(71) xor data_in(72) xor
data_in(81) xor data_in(83) xor data_in(85) xor data_in(87) xor
data_in(95) xor data_in(96) xor data_in(97) xor data_in(98) xor
data_in(99) xor data_in(100) xor data_in(101) xor data_in(102) xor
data_in(109) xor data_in(117) xor data_in(123) xor data_in(124) xor
data_in(131) xor data_in(132) xor data_in(137) xor data_in(139) xor
data_in(145) xor data_in(147) xor data_in(151) xor data_in(152) xor
data_in(153) xor data_in(154) xor data_in(159) xor data_in(160) xor
data_in(161) xor data_in(162) xor data_in(165) xor data_in(169) xor
data_in(173) xor data_in(177) xor data_in(179) xor data_in(180) xor
data_in(183) xor data_in(184) xor data_in(187) xor data_in(188) xor
data_in(191) xor data_in(192) xor data_in(193) xor data_in(195) xor
data_in(197) xor data_in(199) xor data_in(201) xor data_in(203) xor
data_in(205) xor data_in(208) xor data_in(209) xor data_in(210) xor
data_in(211) xor data_in(212) xor data_in(213) xor data_in(214) xor
data_in(215) xor data_in(216) xor data_in(217) xor data_in(218) xor
data_in(219) xor data_in(220) xor data_in(222) xor data_in(235) xor
data_in(236) xor data_in(237) xor data_in(249) xor data_in(252);

    lfsr_c(14) <= lfsr_q(10) xor lfsr_q(13) xor data_in(12) xor
data_in(13) xor data_in(26) xor data_in(28) xor data_in(40) xor
data_in(41) xor data_in(42) xor data_in(43) xor data_in(54) xor
data_in(58) xor data_in(68) xor data_in(69) xor data_in(72) xor
data_in(73) xor data_in(82) xor data_in(84) xor data_in(86) xor
data_in(88) xor data_in(96) xor data_in(97) xor data_in(98) xor
data_in(99) xor data_in(100) xor data_in(101) xor data_in(102) xor
data_in(103) xor data_in(110) xor data_in(118) xor data_in(124) xor
data_in(125) xor data_in(132) xor data_in(133) xor data_in(138) xor
data_in(140) xor data_in(146) xor data_in(148) xor data_in(152) xor
data_in(153) xor data_in(154) xor data_in(155) xor data_in(160) xor
data_in(161) xor data_in(162) xor data_in(163) xor data_in(166) xor
data_in(170) xor data_in(174) xor data_in(178) xor data_in(180) xor
data_in(181) xor data_in(184) xor data_in(185) xor data_in(188) xor
data_in(189) xor data_in(192) xor data_in(193) xor data_in(194) xor
data_in(196) xor data_in(198) xor data_in(200) xor data_in(202) xor
data_in(204) xor data_in(206) xor data_in(209) xor data_in(210) xor
data_in(211) xor data_in(212) xor data_in(213) xor data_in(214) xor
data_in(215) xor data_in(216) xor data_in(217) xor data_in(218) xor
data_in(219) xor data_in(220) xor data_in(221) xor data_in(223) xor
data_in(236) xor data_in(237) xor data_in(238) xor data_in(250) xor
data_in(253);

    lfsr_c(15) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(2) xor lfsr_q(3) xor
lfsr_q(4) xor lfsr_q(5) xor lfsr_q(6) xor lfsr_q(7) xor lfsr_q(8) xor
lfsr_q(9) xor lfsr_q(10) xor lfsr_q(14) xor lfsr_q(15) xor data_in(0) xor
data_in(1) xor data_in(2) xor data_in(3) xor data_in(4) xor data_in(5) xor
```

```
data_in(6) xor data_in(7) xor data_in(8) xor data_in(9) xor data_in(10)
xor data_in(11) xor data_in(12) xor data_in(14) xor data_in(15) xor
data_in(16) xor data_in(17) xor data_in(18) xor data_in(19) xor
data_in(20) xor data_in(21) xor data_in(22) xor data_in(23) xor
data_in(24) xor data_in(25) xor data_in(26) xor data_in(29) xor
data_in(30) xor data_in(31) xor data_in(32) xor data_in(33) xor
data_in(34) xor data_in(35) xor data_in(36) xor data_in(37) xor
data_in(38) xor data_in(39) xor data_in(40) xor data_in(42) xor
data_in(44) xor data_in(45) xor data_in(46) xor data_in(47) xor
data_in(48) xor data_in(49) xor data_in(50) xor data_in(51) xor
data_in(52) xor data_in(53) xor data_in(54) xor data_in(59) xor
data_in(60) xor data_in(61) xor data_in(62) xor data_in(63) xor
data_in(64) xor data_in(65) xor data_in(66) xor data_in(67) xor
data_in(68) xor data_in(70) xor data_in(71) xor data_in(72) xor
data_in(74) xor data_in(75) xor data_in(76) xor data_in(77) xor
data_in(78) xor data_in(79) xor data_in(80) xor data_in(81) xor
data_in(82) xor data_in(85) xor data_in(86) xor data_in(89) xor
data_in(90) xor data_in(91) xor data_in(92) xor data_in(93) xor
data_in(94) xor data_in(95) xor data_in(96) xor data_in(98) xor
data_in(100) xor data_in(102) xor data_in(104) xor data_in(105) xor
data_in(106) xor data_in(107) xor data_in(108) xor data_in(109) xor
data_in(110) xor data_in(119) xor data_in(120) xor data_in(121) xor
data_in(122) xor data_in(123) xor data_in(124) xor data_in(126) xor
data_in(127) xor data_in(128) xor data_in(129) xor data_in(130) xor
data_in(131) xor data_in(132) xor data_in(134) xor data_in(135) xor
data_in(136) xor data_in(137) xor data_in(138) xor data_in(141) xor
data_in(142) xor data_in(143) xor data_in(144) xor data_in(145) xor
data_in(146) xor data_in(149) xor data_in(150) xor data_in(151) xor
data_in(152) xor data_in(154) xor data_in(156) xor data_in(157) xor
data_in(158) xor data_in(159) xor data_in(160) xor data_in(162) xor
data_in(164) xor data_in(165) xor data_in(166) xor data_in(171) xor
data_in(172) xor data_in(173) xor data_in(174) xor data_in(179) xor
data_in(180) xor data_in(182) xor data_in(183) xor data_in(184) xor
data_in(186) xor data_in(187) xor data_in(188) xor data_in(190) xor
data_in(191) xor data_in(192) xor data_in(194) xor data_in(197) xor
data_in(198) xor data_in(201) xor data_in(202) xor data_in(205) xor
data_in(206) xor data_in(210) xor data_in(212) xor data_in(214) xor
data_in(216) xor data_in(218) xor data_in(220) xor data_in(222) xor
data_in(223) xor data_in(237) xor data_in(239) xor data_in(240) xor
data_in(241) xor data_in(242) xor data_in(243) xor data_in(244) xor
data_in(245) xor data_in(246) xor data_in(247) xor data_in(248) xor
data_in(249) xor data_in(250) xor data_in(254) xor data_in(255);


    process (clk,rst) begin
      if (rst = '1') then
        lfsr_q <= b"1111111111111111";
      elsif (clk'EVENT and clk = '1') then
        if (crc_en = '1') then
          lfsr_q <= lfsr_c;
           end if;
      end if;
    end process;
end architecture crc16;
```

Appendix B: Calculation of Theta Values

When calculating the fixed-point hexadecimal values of theta to be stored in a look-up table, it was necessary to write a simple Python script. First, the decimal values of theta are calculated using Python math functions. Next, in order to convert the decimal values into fixed point binary, the decimal values are iteratively multiplied by two and checked to see if the integer portion is equal to '0' or '1'. This process determines the 'fractional' part of the fixed-point binary number. Since the fixed-point binary format chosen has two digits for the whole number and 14 digits for the fractional part, the final binary fixed-point value is 16 bits. This value is then adjusted by 1 bit due to the floating-point error in python calculations. Once the correct fixed point binary value is calculated, the python hex() function converts the fixed point binary to fixed point hexadecimal which is used in our final look-up table design.

The Python script for calculating the theta values can be seen below:

```
import math

thetas = []
print("Thetas:")
for i in range(0, 16):
        theta = math.atan( math.pow(2, -i) )
        theta = round(theta, 5)
        thetas.append(theta)
        print(theta)


hex_thetas = []
print("Hex value bit shifted thetas")
for theta in thetas:
        output_value = "00"
        adjustment = "0000000000000001"
        post_decimal = theta
        for i in range(0, 14):
                temp = post_decimal
                post_decimal = temp*2 - int(temp*2)
                temp = temp*2
                output_value = output_value + str(int(temp))

        output_value = int(output_value, 2) + int(adjustment, 2)
        hex_thetas.append(output_value & 0xffff)
        output_value = hex(output_value)
        print(output_value)
```