VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS ENGINEERING

…☼…

# CAPSTONE PROJECT I

# CORDIC PROCESSOR

**Instructor: MEng Phan Võ Kim Anh**

| **Student name** | **Student ID** |
|---|---|
| Nguyễn Đình Huy | 2211208 |

*Ho Chinh Minh city, May 2025*

# PROJECT SUMMARY

This project focuses on the design and implementation of a 32-bit multicycle processor integrated with a CORDIC (COordinate Rotation DIgital Computer) computation unit, aimed at performing efficient trigonometric calculations using floating-point arithmetic. The processor architecture builds upon the foundational concepts from the Digital Design course and is extended to support advanced numerical operations suitable for embedded applications.

The CORDIC algorithm is implemented to compute sine and cosine values through iterative rotation techniques, with optimizations tailored for IEEE-754 floating-point representation. The design emphasizes hardware efficiency by utilizing exponent adjustments in place of conventional shift operations.

Through simulation, synthesis, and on-board validation using an FPGA development platform, the processor's functionality was thoroughly verified. The inclusion of RAM for output visualization further supports system-level integration and testing. This work lays the groundwork for scalable processing units capable of handling real-time mathematical computations in embedded environments.

## Contents

## Figures

# Tables

# 1. Design interests and theory

## *1.1 Design interests*

Building upon the processor architecture introduced in the Digital Design course (Figure 1), the original 9-bit processor was expanded into a 32-bit architecture incorporating a CORDIC-based computation unit, referred to as the CORDIC Processor.

CORDIC (COordinate Rotation DIgital Computer) is a hardware-efficient algorithm widely used for computing trigonometric, hyperbolic, exponential, and logarithmic functions. Its shift-and-add iterative structure eliminates the need for multipliers, making it well-suited for embedded and real-time systems.

However, in this design, a floating-point unit is used, so the typical bit-shifting operations for divided by 2 are replaced by adjusting the exponent of the floating-point number to achieve a similar result.

In the initial development phase, accurate sine and cosine functions were implemented using the CORDIC algorithm. The design was verified through simulation using testbenches and waveform analysis and later validated on actual hardware via FPGA deployment.

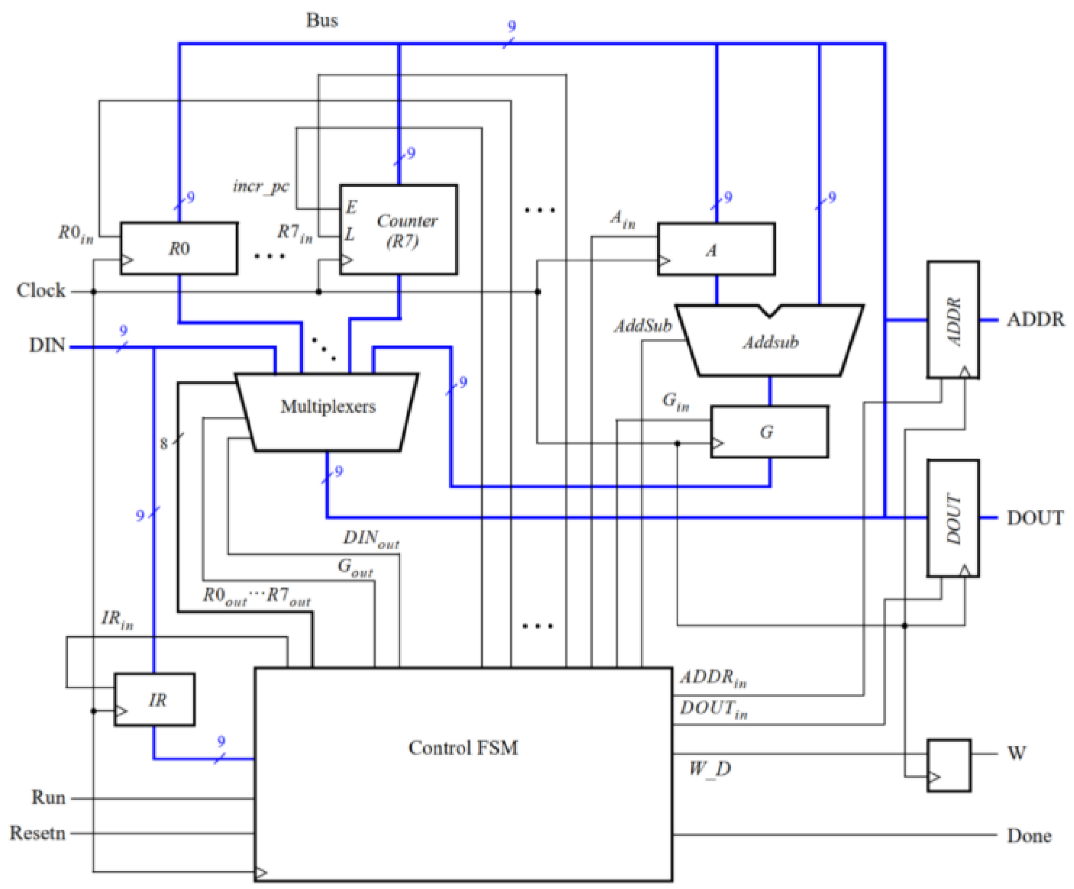


*Figure 1.1: Processor from Digital System*

## *1.2 Design methodology*

The implementation of the CORDIC Processor is based on the mathematical principles of vector rotation in polar coordinates. The CORDIC algorithm leverages a sequence of rotations to converge to a desired angle, enabling efficient computation of trigonometric functions.

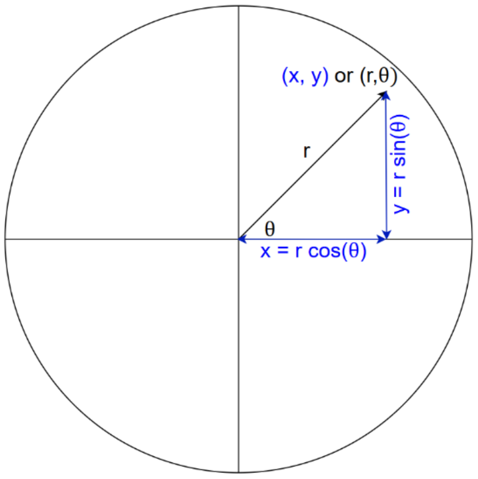


*Figure 1.2: Polar derivatives*

As shown in the figure above, any point on a circle can be represented as:

$$x = r \times \cos(\theta)\,, y = r \times \sin\,(\theta)$$

By applying an angle offset $\alpha$, the rotated coordinates become:

$$x_i = r \times \cos(\theta + \alpha)\,, y_i = r \times \sin\,(\theta + \alpha)$$

Using basic trigonometric identities:

$$\sin(x + y) = \sin(x)\cos(y) + \cos(x)\sin(y)$$

$$\cos(x + y) = \cos(x)\cos(y) - \sin(x)\sin(y)$$

Applying these formulas results in:

$$x_i = r \times [cos(\theta)\ cos(\alpha) - sin\ (\theta)\ sin(\alpha)]$$
$$= r \times cos(\theta)cos(\alpha) - r \times sin(\theta)sin(\alpha)) = x\ cos(\alpha) - y\ sin(\alpha)$$

$$y_i = r \times [sin(\theta)\ cos(\alpha) - \cos(\theta)\ sin(\alpha)]$$
$$= r \times sin(\theta)cos(\alpha) + r \times cos(\theta)sin(\alpha)) = y\ cos(\alpha) + x\ sin(\alpha)$$

For a small $\theta$ angle, the rotated vector $(x_i, y_i)$ becomes:

$$x_{i+1} = x_i\ cos(\theta_i) - y_i\ sin(\theta_i) \Rightarrow x_{i+1} = cos(\theta_i)[x_i - y_i\ tan(\theta_i)]$$

$$y_{i+1} = y_i\ cos(\theta_i) + x_i\ sin(\theta_i)\ \Rightarrow y_{i+1} = cos(\theta_i)[y_i + x_i\ tan(\theta_i)]$$

The scale factor $cos(\theta_i)$ only affects the vector's magnitude and not its direction, so it is removed:

$$x_{i+1} = x_i - y_i\, tan(\theta_i)$$

$$y_{i+1} = y_i + x_i\, tan(\theta_i)$$

Assuming $tan(\theta) = 2^{-i}$, the equation becomes:

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

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

To steer the vector toward the desired angle, the rotation direction is controlled by a coefficient $d_i \in \{-1,1\}$

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

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

The variable z stores the residual angle between the desired angle and the current angle, and it converges to zero over successive iterations:

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

Arctangent Lookup Table for CORDIC Iterations:

| Index (i) with Expression | IEEE-754 32-bit Floating Point | Decimal value |
|---|---|---|
| $tan^{-1}(2^{-0})$ | 32'b0_01111110_10010010000111111011011 | 0.7853981634 |
| $tan^{-1}(2^{-1})$ | 32'b0_01111101_11011010110001100111000 | 0.463647609 |
| $tan^{-1}(2^{-2})$ | 32'b0_01111100_11110101101101110110000 | 0.2449786631 |
| $tan^{-1}(2^{-3})$ | 32'b0_01111011_11111101010110111010101 | 0.1243549945 |
| $tan^{-1}(2^{-4})$ | 32'b0_01111010_11111110101010011011110 | 0.06241881 |
| $tan^{-1}(2^{-5})$ | 32'b0_01111001_11111111110101010101110 | 0.03123983343 |
| $tan^{-1}(2^{-6})$ | 32'b0_01111000_11111111111101010101011 | 0.01562372862 |
| $tan^{-1}(2^{-7})$ | 32'b0_01110111_11111111111111010101011 | 0.00781234106 |
| $tan^{-1}(2^{-8})$ | 32'b0_01110110_11111111111111110101011 | 0.003906230132 |
| $tan^{-1}(2^{-9})$ | 32'b0_01110101_11111111111111111101011 | 0.001953122516 |
| $tan^{-1}(2^{-10})$ | 32'b0_01110100_11111111111111111111011 | 9.7656219e-4 |
| $tan^{-1}(2^{-11})$ | 32'b0_01110011_11111111111111111111111 | 4.88281211e-4 |
| $tan^{-1}(2^{-12})$ | 32'b0_01110011_00000000000000000000000 | 2.4414062e-4 |
| $tan^{-1}(2^{-13})$ | 32'b0_01110010_00000000000000000000000 | 1.22070312e-4 |
| $tan^{-1}(2^{-14})$ | 32'b0_01110001_00000000000000000000000 | 6.10351562e-5 |
| $tan^{-1}(2^{-15})$ | 32'b0_01110000_00000000000000000000000 | 3.05175781e-5 |

*Table 1: Precomputed Arctangent Values for 16-Step CORDIC Rotation (IEEE-754 32-bit Format)*

In the CORDIC algorithm, each iteration omits $cos(\theta_i)$ term to simplify hardware implementation. As a result, the vector magnitude scales by a constant known as the scale factor K, defined as:

$$K = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1 + 2^{-2i}}} \approx 0.60725296$$

For a sufficient number of iterations, K converges to approximately 0.60725296. To correct the final magnitude, the output must be multiplied by K. In floating-point designs, this compensation can be done by a dedicated multiplier block or absorbed in the final stage if only directional values (e.g., sin, cos) are required. In this design, the scale factor is set to K = 0_01111110_00110110111010011101110 in IEEE-754 32-bit floating-point format.

The mathematical derivations presented above form the foundation for the subsequent section, where the corresponding hardware implementation is explored. In the next chapter, these equations are transformed into structural components through a block diagram representation.

# 2. Block Diagram

## *2.1 CORDIC*

The block diagram illustrates the architecture of a CORDIC-based sine and cosine calculation unit, including angle input handling and result adjustment. The block receives an input angle and a start signal to initiate computation. After 17 clock cycles, it outputs the corresponding sine and cosine values.

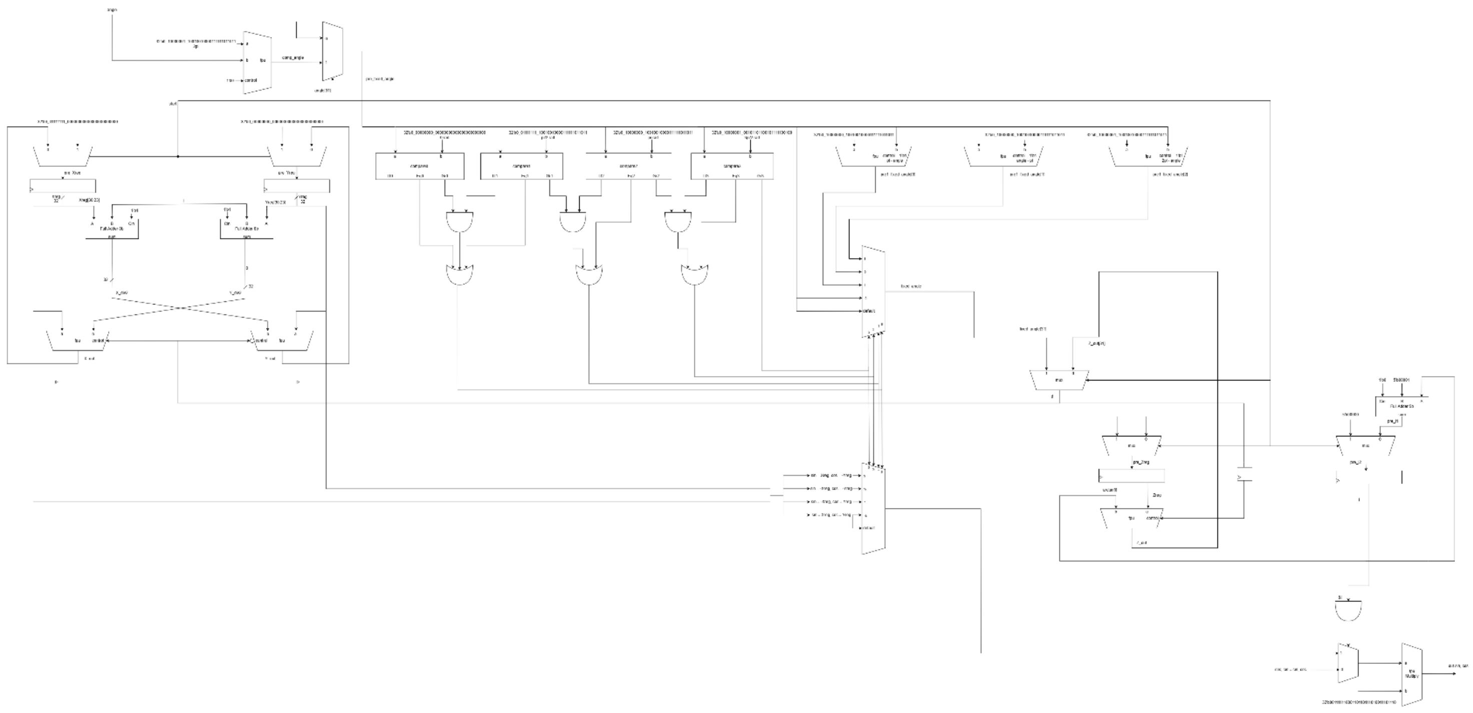


*Figure 2.1: Complete CORDIC Architecture*

This block handles angle adjustment, with the input angle given in floating-point radians. Managing angles across all four quadrants can be complex and error-prone, so the strategy is to map all angles to the first quadrant for easier control and consistency. The adjustment begins by taking the complement of the input if the angle is negative.

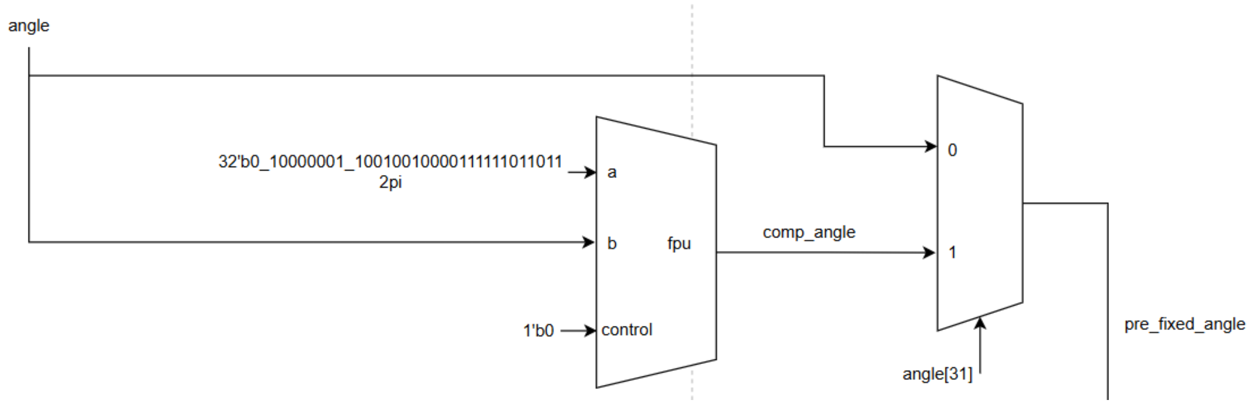


*Figure 2.2: Angle Complementation*

The comparison blocks in the diagram are used to identify the quadrant of the input angle. Based on this classification, the angle is adjusted to lie within the first quadrant $\left[0, \frac{\pi}{2}\right]$ for simplified and consistent trigonometric computation.

- **First Quadrant** $\left[0, \frac{\pi}{2}\right]$: The angle remains unchanged.
- **Second Quadrant** $\left(\frac{\pi}{2}, \pi\right]$: The angle is transformed to $\pi - \theta$.
- **Third Quadrant** $\left(\pi, \frac{3\pi}{2}\right]$: The angle is transformed to $\theta - \pi$.
- **Fourth Quadrant** $\left(\frac{3\pi}{2}, 2\pi\right)$: The angle is transformed to $2\pi - \theta$.

These transformations not only normalize the angle but also ensure proper adjustment of the signs of sine and cosine values based on their original quadrant. This method simplifies implementation and enhances accuracy in trigonometric calculations.

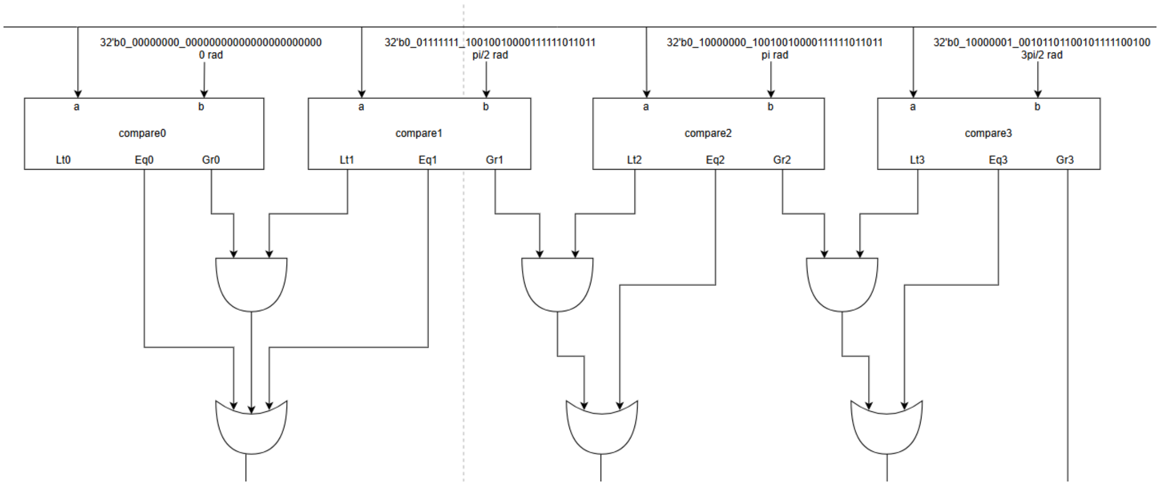

32'b0_00000000_00000000000000000000000
0 rad
32'b0_01111111_10010010000111111011011
pi/2 rad
32'b0_10000000_10010010000111111011011
pi rad
32'b0_10000001_00101101100101111100100
3pi/2 rad
a
b
compare0
Lt0
Eq0
Gr0
compare1
Lt1
Eq1
Gr1
compare2
Lt2
Eq2
Gr2
compare3
Lt3
Eq3
Gr3

*Figure 2.3: Angle Position Selection*

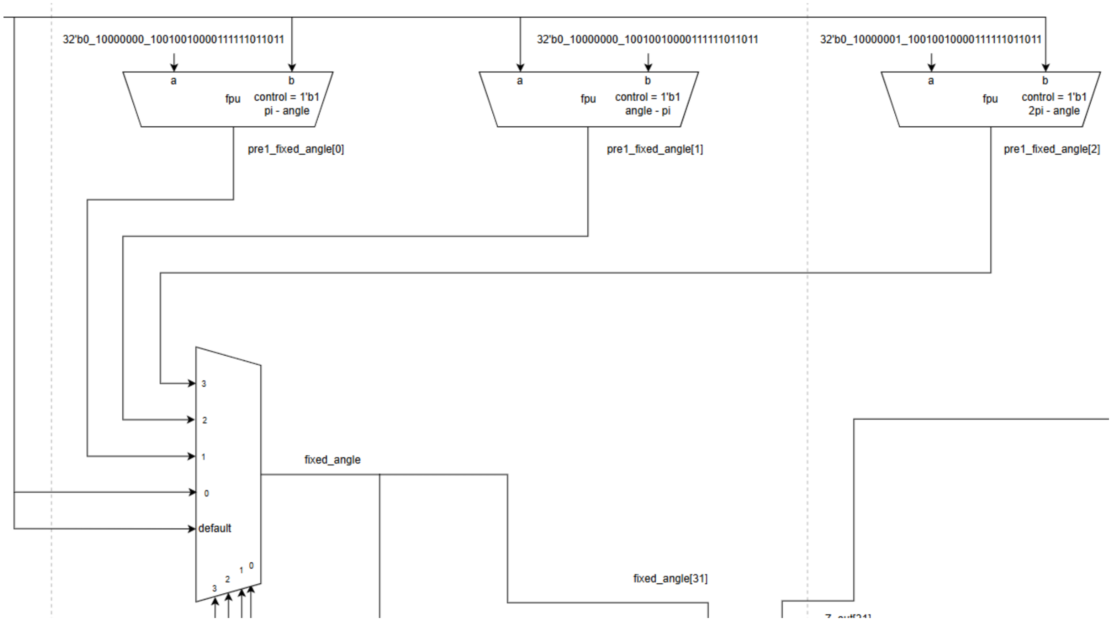


*Figure 2.4: Angle Calculation*

Although all angles are normalized to the first quadrant for computation, the final sine and cosine results must be corrected based on the original quadrant to maintain accuracy. This is because sine and cosine exhibit different signs in each quadrant.

- **First Quadrant** $\left[0, \frac{\pi}{2}\right]$: $sin(\theta), cos(\theta)$ remains unchanged
- **Second Quadrant** $\left(\frac{\pi}{2}, \pi\right]$: $sin(\theta) = sin(\pi - \theta),\ cos(\theta) = -cos(\pi - \theta)$
- **Third Quadrant** $\left(\pi, \frac{3\pi}{2}\right]$: $sin(\theta) = -sin(\theta - \pi),\ cos(\theta) = -cos(\theta - \pi)$
- **Fourth Quadrant** $\left(\frac{3\pi}{2}, 2\pi\right)$: $sin(\theta) = -sin(2\pi - \theta),\ cos(\theta) = cos(2\pi - \theta)$

Thus, although the core computation is performed using angles in the first quadrant, proper sign adjustments are necessary afterward to reflect the true values corresponding to the original angle.

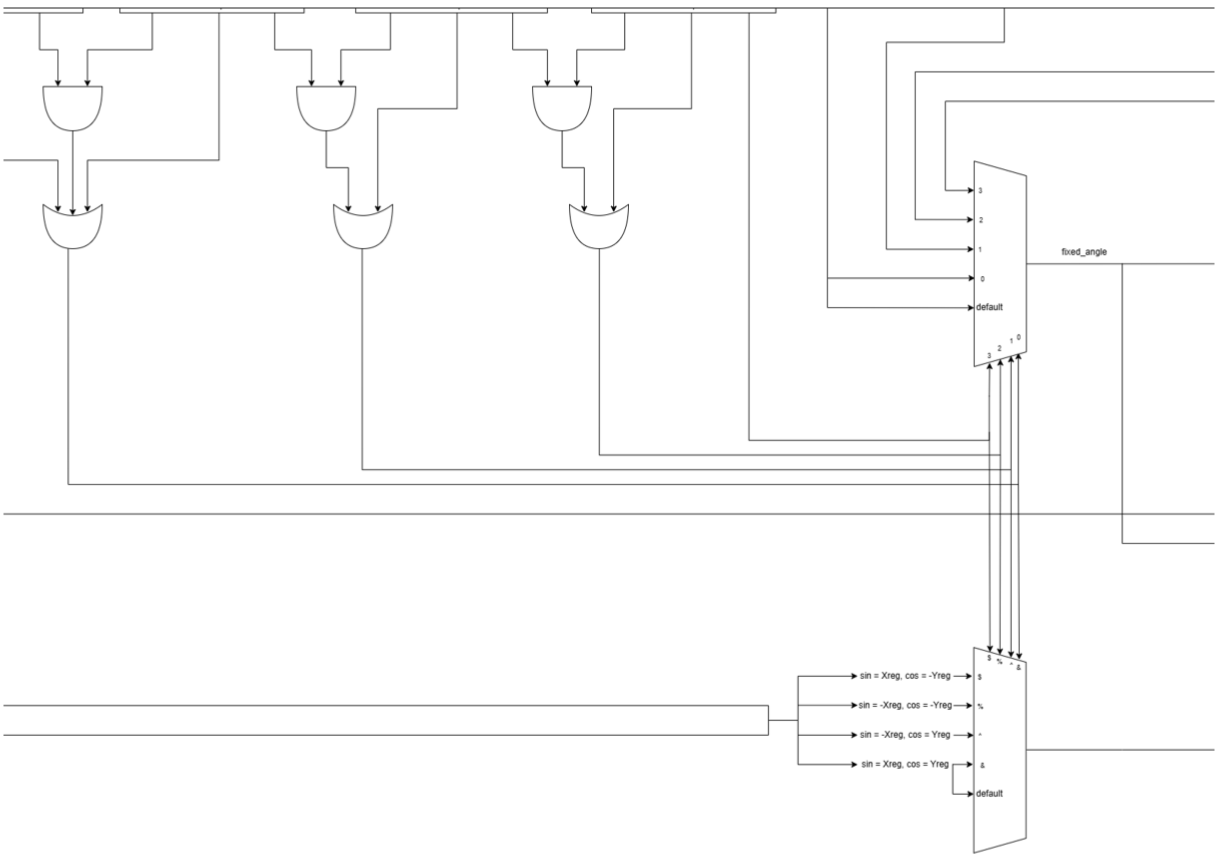


*Figure 2.5: Pre-Result Adjustment*

This block illustrates the control logic designed to generate the iteration variable used to count the number of CORDIC rotation steps. It features a 5-bit full adder for incrementing the count, a multiplexer for selecting between an initial or updated value, and a register to store the current iteration index. This mechanism ensures proper step tracking throughout the CORDIC computation process.

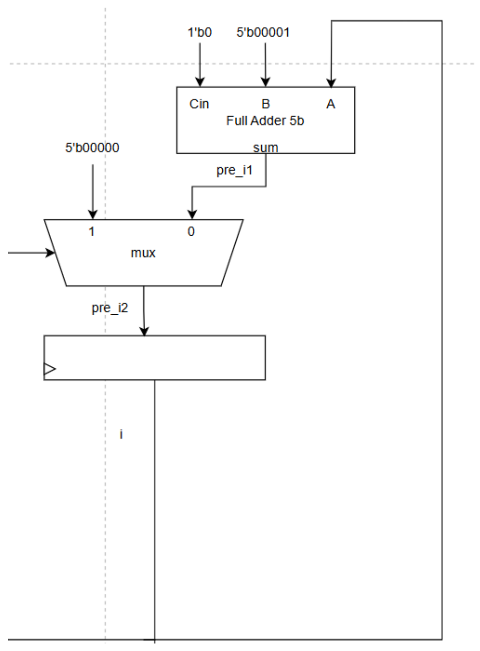


*Figure 2.6: Rotation Iteration Control*

This block performs residual angle computation in each CORDIC iteration. It determines the direction of rotation based on the sign of the current angle value. If the residual angle is positive, the control signal d is set to 0; if negative, d is set to 1. The output angle is updated through a floating-point unit (FPU), and the intermediate value is stored in a flip-flop (Zreg). This ensures the residual angle can be accessed and used in the following clock cycle for continued iteration and accurate convergence.

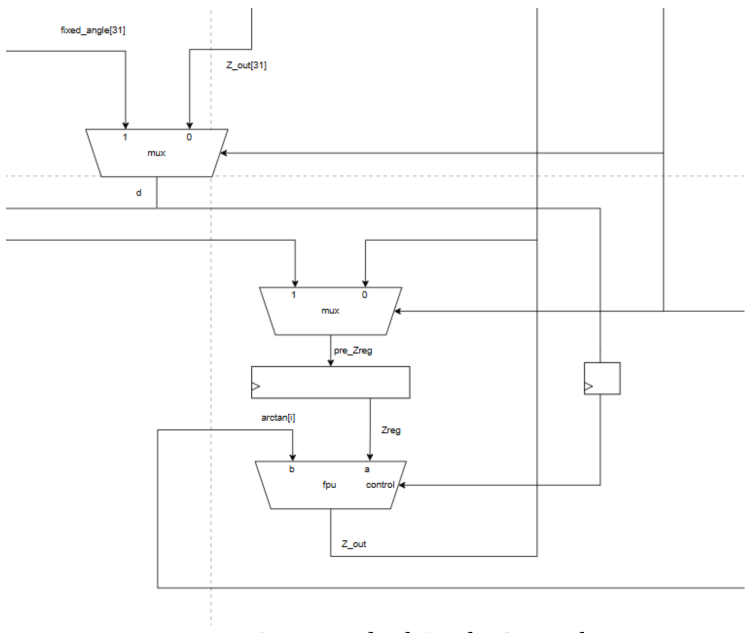


*Figure 2.7: Residual Angle Control*

Based on the iterative CORDIC equations:

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

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

The following block diagram was designed to implement trigonometric rotations.

Due to the use of IEEE-754 floating-point format, division by 2 is efficiently achieved by simply subtracting 1 from the exponent field of the number, rather than performing full division. This reduces hardware complexity and enhances processing speed. In the IEEE-754 format, halving a value corresponds to shifting the exponent down by one bit (i.e., exponent = exponent - 1), while keeping the mantissa and sign unchanged.

In this design, at each clock cycle, the system computes $\frac{x}{2}$ and $\frac{y}{2}$ using exponent adjustment, stores them in registers, and then updates x and y using the formulas above. The process is repeated over 16 clock cycles to iteratively converge to the final sine and cosine results.

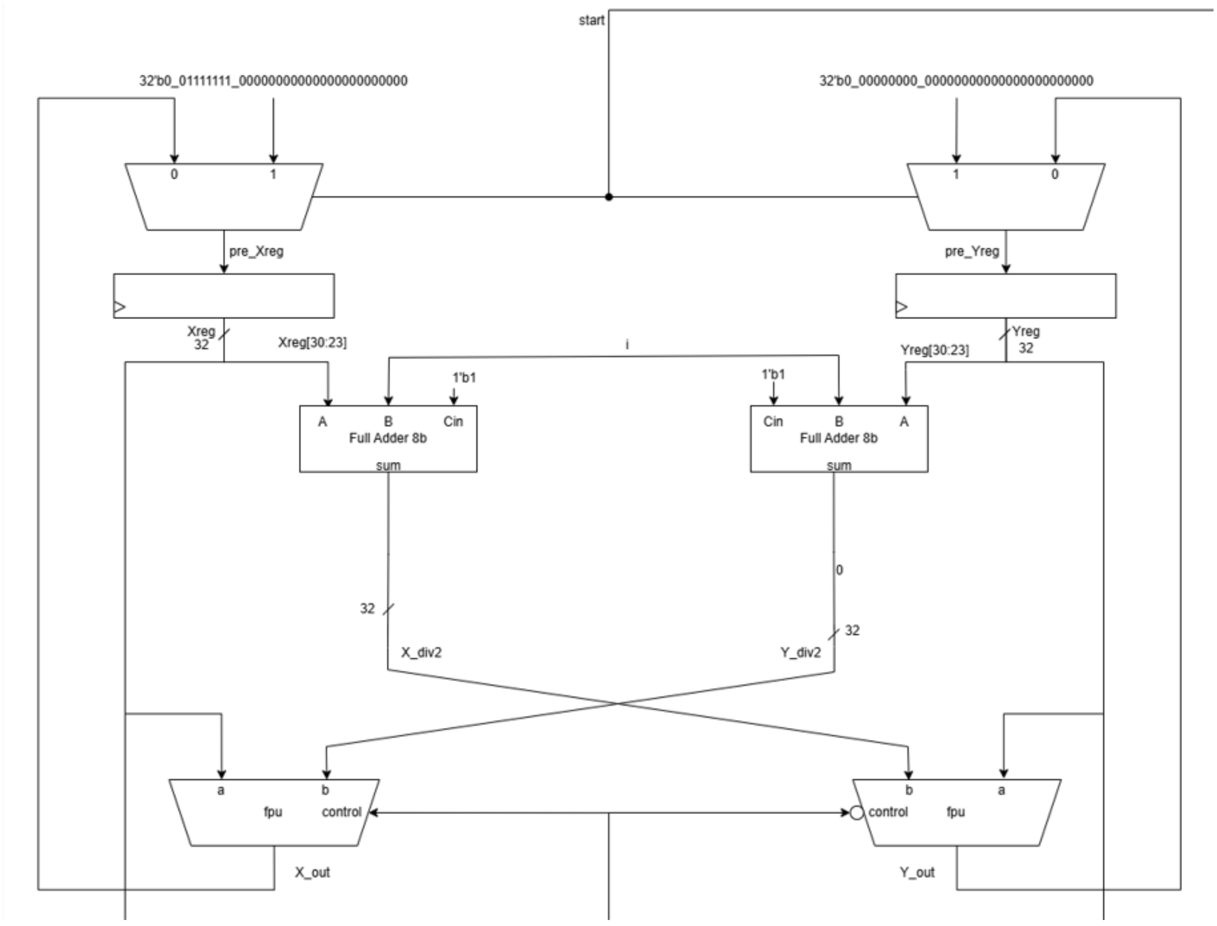


Figure 2.8: Sine and Cosine Computation

The diagram above illustrates the final output multiplication block used to compute the sine and cosine values after CORDIC processing. The sine and cosine outputs are multiplied by the scale factor K to compensate for the amplitude reduction inherent in the CORDIC algorithm.

A multiplexer conditionally swaps the sine and cosine values based on the (&i) control signal, ensuring the correct orientation according to the input angle's quadrant. These values are then passed to a floating-point multiplier, where they are scaled using the constant K, represented in IEEE-754 32-bit format. The final output is the normalized sine and cosine values.

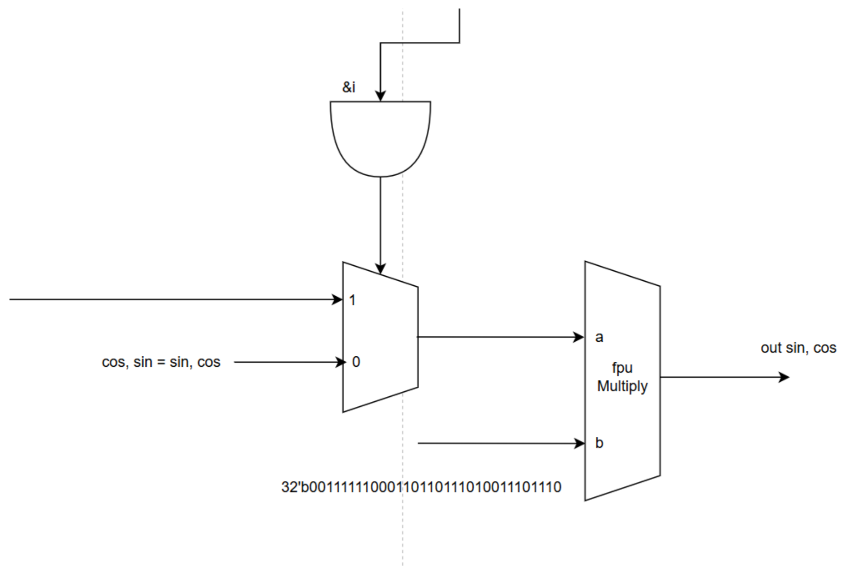


*Figure 2.9: Scale Factor Compensation*

## 2.2 Processor

The diagram represents the overall architecture of a 32-bit processor. It features 31 general-purpose registers (R0 to R30) and a dedicated 32nd register (R31) used as a counter. The IR block is responsible for loading instructions into the Control FSM, which orchestrates the processor's operation.

The multiplexer selects appropriate data sources under the FSM's control and routes them onto the 32-bit data bus. The processor includes a 32-bit ALU capable of performing arithmetic (addition, subtraction), logical operations, comparisons, bitwise shifts, and floating-point computations.

A dedicated CORDIC unit is integrated for trigonometric calculations, with sine and cosine outputs. The processor also includes external interface lines: ADDR (address bus), DOUT (data output), and WR_EN (write enable) for memory communication, enabling read/write operations with external RAM.

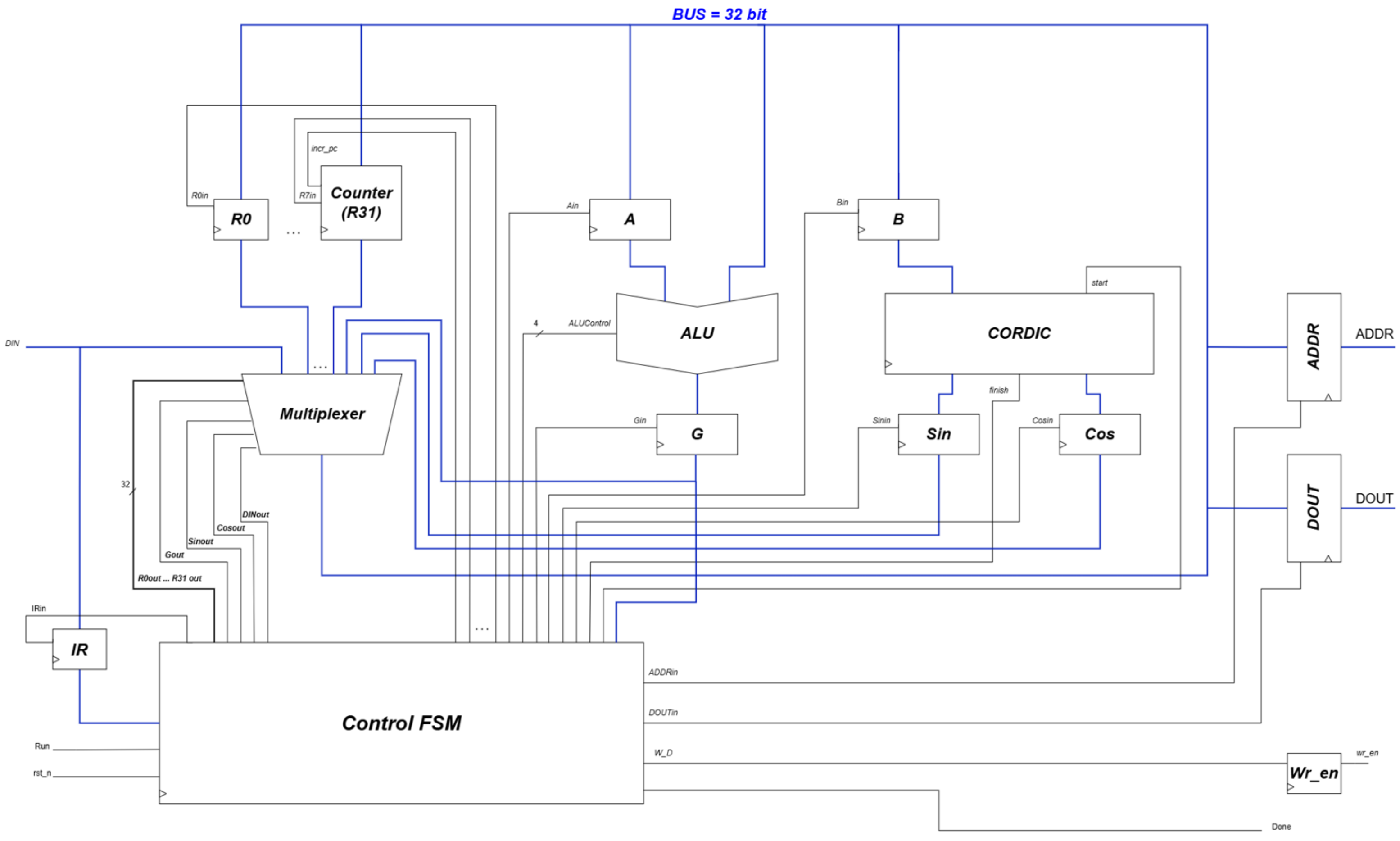


*Figure 2.10: General Block Diagram of the CORDIC-Integrated Processor*

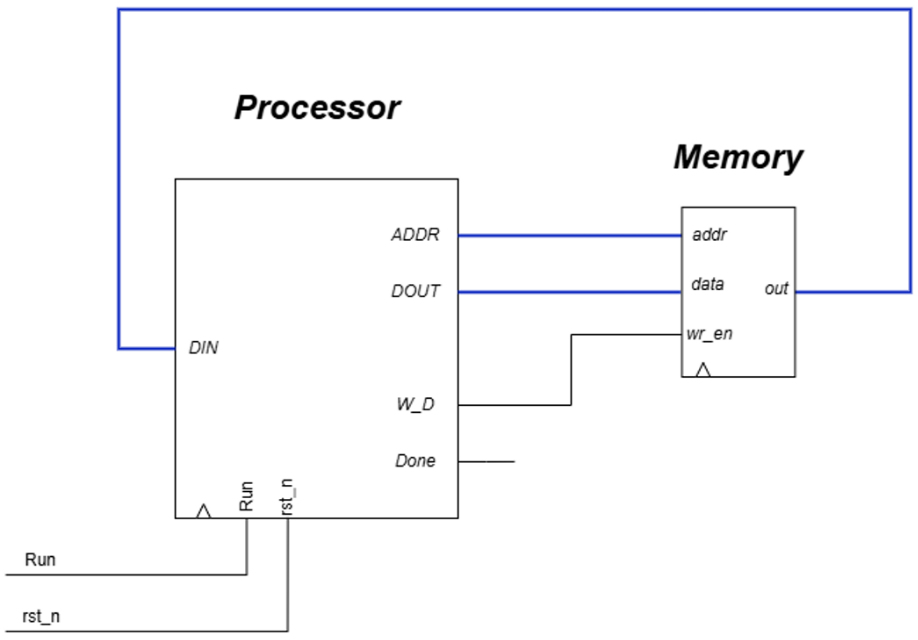


*Figure 2.11: 32-bit Processor with RAM Interface*

The Arithmetic Logic Unit (ALU) is a critical component responsible for executing various arithmetic and logical operations. This unit performs calculations including addition and subtraction, logical operations such as AND, OR, XOR, and NOT, as well as logical left and right shifts, arithmetic right shifts and comparation. Additionally, it supports floating-point computations handled by the floating-point unit (FPU). The complete set of operations, controlled by the 4-bit ALUControl signal [3:0], is summarized in the table below.

| ALUControl [3:0] | Operation |
|---|---|
| 0000 | Addition |
| 0001 | Subtraction |
| 0010 | And |
| 0011 | Or |
| 0100 | Xor |
| 0101 | Not |
| 0110 | Right logical shift |
| 0111 | Left logic shift |
| 1000 | Right arithmetic shift |
| 1001 | Compare equal |
| 1010 | Compare less |
| 1011 | Compare greater or equal |
| 1100 | Compare not equal |
| 1101 | Addition floating point |
| 1110 | Subtraction floating point |
| 1111 | Multiplication floating point |

*Table 2: ALU Control Table*

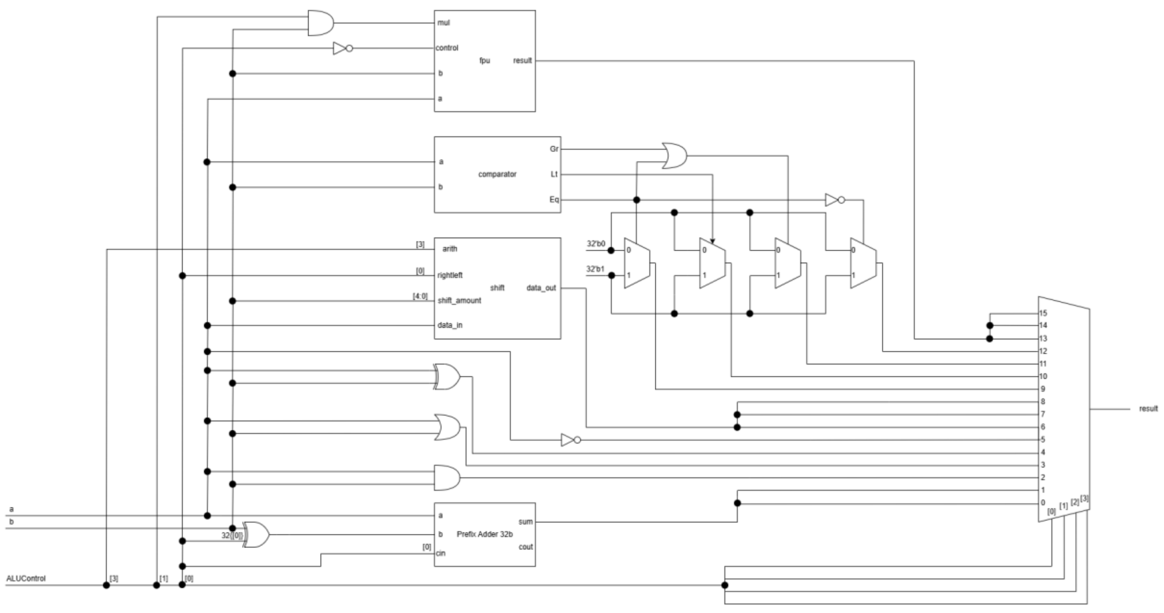


*Figure 2.12: ALU Block Diagram*

The processor is implemented as a 32-bit multicycle architecture, enabling efficient instruction execution by breaking operations into multiple clock cycles. Integration of the CORDIC unit extends the processor’s capabilities to perform trigonometric computations directly within the data path. This design approach balances computational complexity and hardware resource utilization, providing flexibility for a variety of arithmetic, logical, and specialized mathematical operations. The processor’s modular structure supports clear control flow management through a finite state machine, ensuring precise sequencing of multicycle instructions.

# 3. I/O Description

## *3.1 CORDIC Processor*

The tables categorize the input/output ports into two groups: processor signals and CORDIC signals.

Processor signals include clock (***clk***) and reset (***rst_n***) inputs, which synchronize and initialize the processor. The run signal (***run***) enables the processor's operation. The address (***ADDR***), data input (***DIN***), data output (***DOUT***), and write enable (***wr_en***) signals enable communication with memory, managing standard processor data flow, while the ***Done*** signal Denotes that one instruction has been fully executed.

CORDIC signals control the trigonometric computation unit embedded within the processor. The ***start*** input triggers the CORDIC calculation based on the provided floating-point angle (***angle***). Upon computation completion, the sine and cosine values are output through ***sin*** and ***cos*** ports respectively, while the ***finish*** signal is asserted, it sends a notification to the Control FSM, which then captures the computed sine and cosine values and initiates writing them into dedicated registers reserved for sine and cosine data storage using ***Sinin*** and ***Cosin***.

This clear separation enhances modularity and maintainability of the design by isolating CORDIC control from the main processor control signals.

| Name | Direction | Bit width | Description |
|---|---|---|---|
| clk | Input | 1 | System clock signal controlling synchronous operations |
| rst_n | Input | 1 | Asynchronous negative reset signal to initialize processor states |
| run | Input | 1 | Enabling the processor's operation |
| DIN | Input | 32 | Data bus input for reading data from memory |
| ADDR | Output | 8 | Address bus for memory access |
| DOUT | Output | 32 | Data bus output for writing data to memory |
| wr_en | Output | 1 | Write enable signal for memory write operations |
| Done | Output | 1 | Signals the completion of a single instruction execution |

*Table 3: Processor Signals*

| Name | Direction | Bit width | Description |
|---|---|---|---|
| start | Input | 1 | Signal to initiate the CORDIC computation process |
| angle | Input | 32 | Floating-point input angle in radians for CORDIC unit |
| sin | Output | 32 | Floating-point sine value output from CORDIC unit |
| cos | Output | 32 | Floating-point cosine value output from CORDIC unit |
| finish | output | 1 | Signal indicating completion of CORDIC computation |

*Table 4: CORDIC signals*

| Name | Direction | Bit width | Description |
|---|---|---|---|
| clk | Input | 1 | System clock signal controlling synchronous operations |
| rst_n | Input | 1 | Asynchronous negative reset signal to initialize processor states |
| run | Input | 1 | Enabling the processor’s operation |
| finish | Input | 1 | The signal indicating the completion of the CORDIC operation |
| IR | Input | [31:0] | Instruction sets |
| G | Input | [31:0] | Output of ALU |
| ALUControl | Output | [3:0] | Control signals of ALU |
| IRin | Output | 1 | The write enable signal for IR flip-flop |
| Gout | Output | 1 | The selection signal for BUS |
| DINout | Output | 1 | The selection signal for BUS |
| Sinout | Output | 1 | The selection signal for BUS |
| Cosout | Output | 1 | The selection signal for BUS |
| Incr-pc | Output | 1 | Increase counter R31 |
| Ain | Output | 1 | The write enable signal for A flip-flop |
| Bin | Output | 1 | The write enable signal for B flip-flop |
| Gin | Output | 1 | The write enable signal for G flip-flop |
| Sinin | Output | 1 | The write enable signal for Sin flip-flop |
| Cosin | Output | 1 | The write enable signal for Cos flip-flop |
| ADDRin | Output | 1 | The write enable signal for ADDR flip-flop |
| DOUTin | Output | 1 | The write enable signal for DOUT flip-flop |

| | | | |
|---|---|---|---|
| W-D | Output | 1 | The write enable signal for Wr_en flip-flop |
| Done | Output | 1 | Signals the completion of a single instruction execution |
| start | Output | 1 | Signal to initiate the CORDIC computation process |
| R0out … R31out | Output | 1 | The selection signal for BUS |
| R0in … R31in | Output | 1 | The write enable signal for registers flip-flop |

*Table 5: ControlFSM signals*

## 3.2 Interface with FPGA DE10 Kit

The I/O interface with the FPGA development kit uses physical switches (SW), buttons, and 7-segment HEX LEDs for control and status display. Switches (SW[7:0]) select the memory address for read operations, while the HEX LEDs display the data at that address. Due to limited display capacity, the most significant bits (MSB) are shown in two separate registers toggled by KEY[2], which switches between high and low data portions. KEY[1] serves as the run control signal to start the processor.

An additional RAM, identical to the main memory, is included solely for displaying data on the HEX LEDs via a multiplexer, allowing independent output visualization without affecting processor operations.

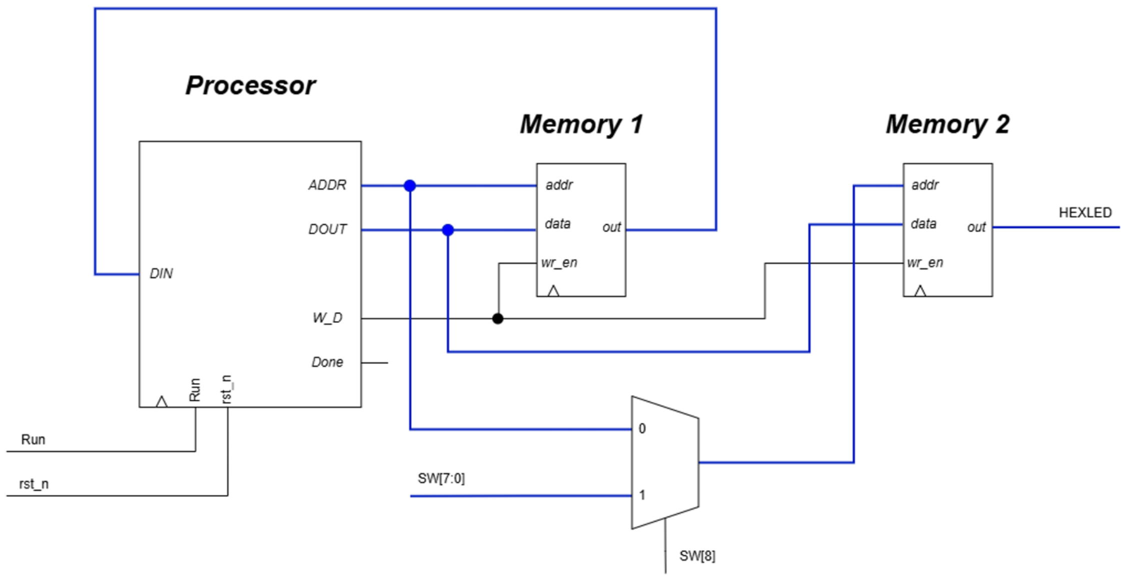


*Figure 3.1: Processor with Two RAM Modules for FPGA Board Implementation*

# 4. Main operation

### *4.1 CORDIC operation*

The CORDIC unit performs iterative vector rotations to compute trigonometric functions such as sine and cosine. Upon receiving a start signal along with the input angle in radian floating-point format, the unit initiates a sequence of rotations. Each iteration adjusts the vector coordinates by adding or subtracting scaled values based on precomputed arctangent angles.

The iterative process continues for a fixed number of cycles, progressively reducing the residual angle until convergence is achieved. Throughout the operation, exponent adjustments replace traditional bit-shifting, leveraging the floating-point format to optimize hardware efficiency.

Once the computation is complete, the unit outputs the normalized sine and cosine values, scaled appropriately by the CORDIC gain factor. A finish signal indicates completion, allowing the control logic to capture and store the results for subsequent use by the processor.

### *4.2 Processor operation*

Initially, the processor resides in the **reset state**, during which no operations occur until the next phase. When the **run signal is asserted** and **rst_n = 1**, the processor begins execution.

The processor places the value of the **counter register R31** onto the **BUS** and loads it into the **ADDR** port to read data from the RAM. The data retrieved from RAM is fed back into the **DIN** port.

Upon completion of this task, the **counter R31 is incremented by one**.

Next, the processor requires **two clock cycles to fetch** the instruction from memory and load it into the control FSM. Based on the fetched instruction, the control FSM decodes and directs the processor to perform the corresponding operation.

Once the execution of the current instruction is complete, the **Done signal is asserted** (set to 1), signaling the processor to return to the **address loading step** to fetch and execute the next instruction.

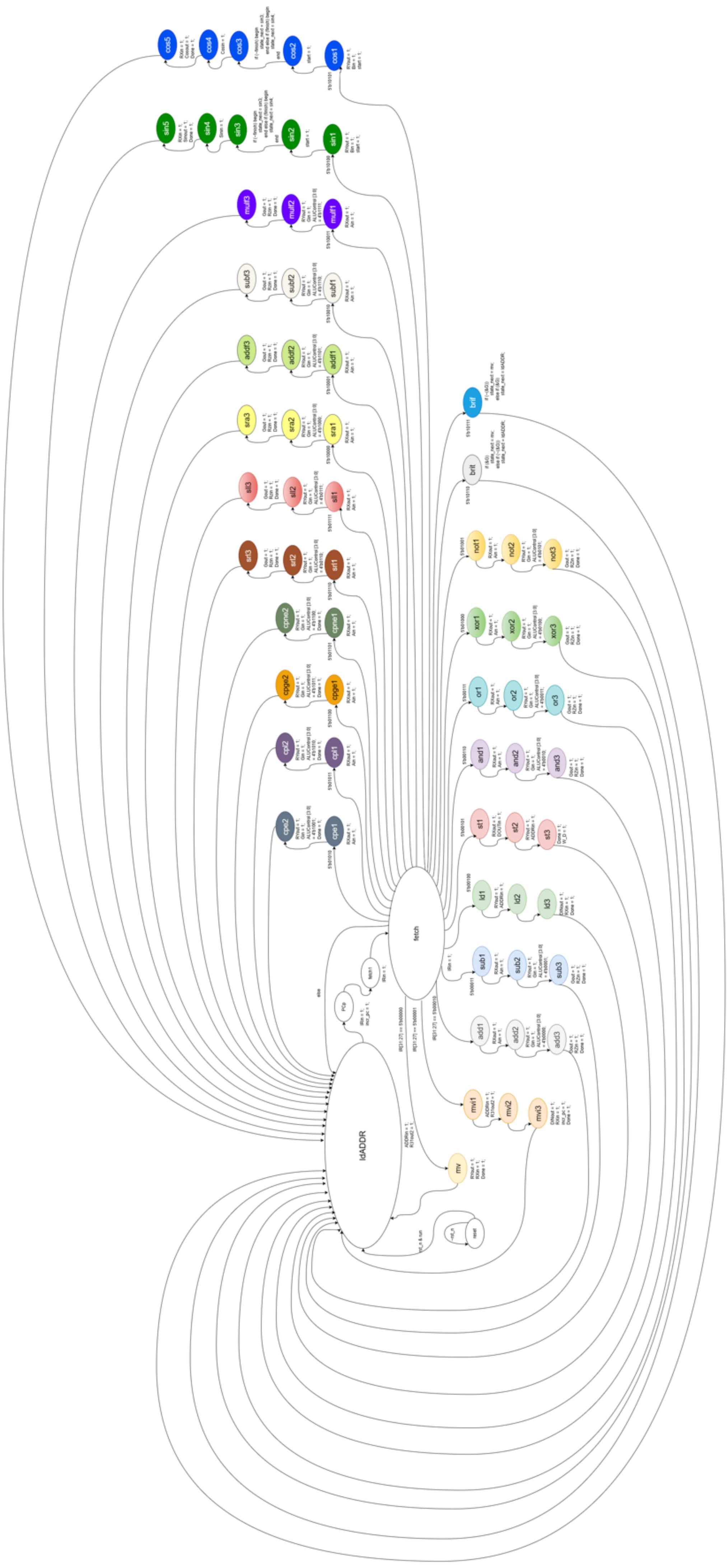


*Figure 4.1: Moore Finite State Machine*

# 5. Detailed functions description

The Moore finite state machine (FSM) presented above provides a comprehensive framework that serves as a solid foundation for detailed explanation of the implementation of the 24 instructions within the processor. This FSM enables precise control over each instruction's execution cycle, ensuring correct sequencing and coordination across all operational stages.

This chapter provides two detailed tables describing the 24 instructions supported by the processor. The first table lists the binary opcode for each instruction along with a brief description of its operation. The second table details the execution stages of each instruction, explaining how each stage functions and how the control signals are managed by the control finite state machine (FSM).

| **IR** | **DIN** | | | | | **Description** |
|---|---|---|---|---|---|---|
| | [31:27] | [26:15] | [14:10] | [9:5] | [4:0] | |
| mv Rx, Ry | 00000 | X | X | Rx | Ry | Move data from Ry to Rx (Rx ← Ry) |
| mvi Rx, data | 00001 | X | X | Rx | X | Move immediate a 32-bit data to Rx (takes 2 instruction, Rx ← data) |
| | data | | | | | |
| add Rz, Rx, Ry | 00010 | X | Rz | Rx | Ry | Addition of Rx and Ry and stored in Rz (Rz ← Rx + Ry) |
| sub Rz, Rx, Ry | 00011 | X | Rz | Rx | Ry | Subtraction of Rx and Ry and stored in Rz (Rz ← Rx – Ry) |
| ld Rx, Ry | 00100 | X | X | Rx | Ry | Load the value stored at the memory address contained in Ry into Rx (Rx ← [Ry]) |
| st Rx, Ry | 00101 | X | X | Rx | Ry | Store the value of Ry to the memory address contained in Rx ([Rx] ← Ry) |
| and Rz, Rx, Ry | 00110 | X | Rz | Rx | Ry | Logical And (Rz ← Rx & Ry) |
| or Rz, Rx, Ry | 00111 | X | Rz | Rx | Ry | Logical Or (Rz ← Rx \| Ry) |

| | | | | | | |
|---|---|---|---|---|---|---|
| xor Rz, Rx, Ry | 01000 | X | Rz | Rx | Ry | Logical Xor (Rz ← Rx ^ Ry) |
| not Rz, Rx, Ry | 01001 | X | Rz | Rx | Ry | Logical Not (Rz ← Rx ^ Ry) |
| cpe Rx, Ry | 01010 | X | X | Rx | Ry | Compare Rx and Ry are equal, if true, G = 1 |
| cpl Rx, Ry | 01011 | X | X | Rx | Ry | Compare Rx less then Ry, if true, G = 1 |
| cpge Rx, Ry | 01100 | X | X | Rx | Ry | Compare whether Rx greater or equal Ry, if true, G = 1 |
| cpne Rx, Ry | 01101 | X | X | Rx | Ry | Compare whether Rx and Ry are not equal, if true, G = 1 |
| srl Rz, Rx, Ry | 01110 | X | Rz | Rx | Ry | Rx shift right logical based on Ry and store result in Rz (Rz ← Rx >> Ry) |
| sll Rz, Rx, Ry | 01111 | X | Rz | Rx | Ry | Rx shift left logical based on Ry and store result in Rz (Rz ← Rx << Ry) |
| sra Rz, Rx, Ry | 10000 | X | Rz | Rx | Ry | Rx shift right arithmetic based on Ry and store result in Rz (Rz ← Rx >> Ry) |
| addf Rz, Rx, Ry | 10001 | X | Rz | Rx | Ry | Addition floating point of Rx and Ry and stored in Rz (Rz ← Rx + Ry) |
| subf Rz, Rx, Ry | 10010 | X | Rz | Rx | Ry | Subtraction floating point of Rx and Ry and stored in Rz (Rz ← Rx – Ry) |
| mulf Rz, Rx, Ry | 10011 | X | Rz | Rx | Ry | Multiplication floating point of Rx and Ry and stored in Rz (Rz ← Rx × Ry) |
| sin Rx, Ry | 10100 | X | X | Rx | Ry | Calculate Sin(Ry) and store result in Rx |

| | | | | | | |
|---|---|---|---|---|---|---|
| cos Rx, Ry | 10101 | X | X | Rx | Ry | Calculate Cos(Ry) and store result in Rx |
| brit Rx, Ry | 10110 | X | X | Rx | Ry | If compare is true, mv |
| brif Rx, Ry | 10111 | X | X | Rx | Ry | If compare is false, mv |

*Table 6: Instruction Sets description*

| IR | Stage 1 | Stage 2 | Stage 3 | Stage 4 | Stage 5 |
|---|---|---|---|---|---|
| mv | RYout = 1<br>RXin = 1<br>**Done = 1** | | | | |
| mvi | ADDRin = 1<br>R31out2 = 1 | | DINout = 1<br>RXin = 1<br>incr_pc = 1<br>**Done = 1** | | |
| add | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b0000 | Gout = 1<br>RZin = 1<br>**Done = 1** | | |
| sub | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b0001 | Gout = 1<br>RZin = 1<br>**Done = 1** | | |
| ld | RYout = 1<br>ADDRin = 1 | | DINout = 1<br>RXin = 1<br>**Done = 1** | | |
| st | RXout = 1<br>DOUTin = 1 | RYout = 1<br>ADDRin = 1 | **Done = 1**<br>W_D = 1 | | |
| and | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b0010 | Gout = 1<br>RZin = 1<br>**Done = 1** | | |
| or | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b0011 | Gout = 1<br>RZin = 1<br>**Done = 1** | | |
| xor | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b0100 | Gout = 1<br>RZin = 1<br>**Done = 1** | | |

| | | | | | |
|---|---|---|---|---|---|
| not | RXout = 1<br>Ain = 1 | Gin = 1<br>ALUControl [3:0] = 4'b0101 | Gout = 1<br>RZin = 1<br>**Done = 1** | | |
| cpe | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1001<br>**Done = 1** | | | |
| cpl | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1010<br>**Done = 1** | | | |
| cpge | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1011<br>**Done = 1** | | | |
| cpne | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1100<br>**Done = 1** | | | |
| srl | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b0110 | Gout = 1<br>Rzin = 1<br>**Done = 1** | | |
| sll | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b0111 | Gout = 1<br>Rzin = 1<br>**Done = 1** | | |
| sra | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1000 | Gout = 1<br>Rzin = 1<br>**Done = 1** | | |
| addf | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1101 | Gout = 1<br>Rzin = 1<br>**Done = 1** | | |

| | | | | | |
|---|---|---|---|---|---|
| subf | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1110 | Gout = 1<br>Rzin = 1<br>**Done = 1** | | |
| mulf | RXout = 1<br>Ain = 1 | RYout = 1<br>Gin = 1<br>ALUControl [3:0] = 4'b1111 | Gout = 1<br>Rzin = 1<br>**Done = 1** | | |
| sin | RYout = 1<br>Bin = 1<br>start = 1 | start = 1 | Wait until finish = 1 | Sinin = 1 | RXin = 1<br>Sinout = 1<br>**Done = 1** |
| cos | RYout = 1<br>Bin = 1<br>start = 1 | start = 1 | Wait until finish = 1 | Cosin = 1 | RXin = 1<br>Cosout = 1<br>**Done = 1** |
| brit | True ⇒ state_next = mv | | | | |
| brif | False ⇒ state_next = mv | | | | |

*Table 7: Stages of Instructions for ControlFSM*

The previous section detailed the step-by-step operation of each instruction. However, the current FSM design is not yet optimized, as many instructions share identical final stages. These common stages could be consolidated to reduce the total number of states, thereby simplifying the overall design and improving efficiency.

# 6. RTL Simulation results

## 6.1 *CORDIC Simulation*

The following waveform illustrates the simulation results of the CORDIC module.

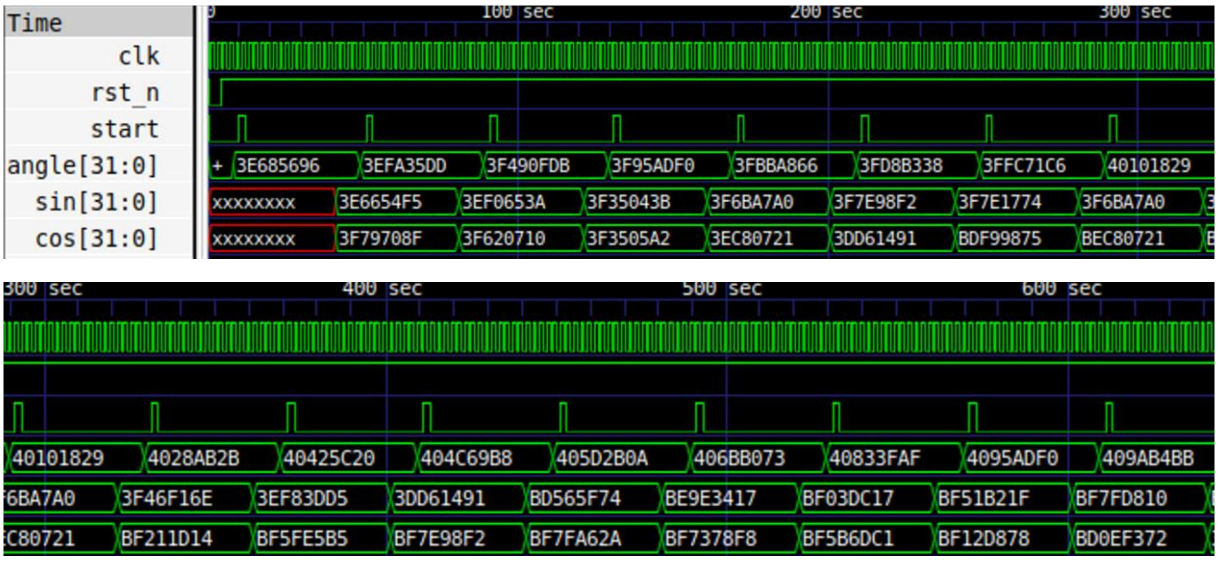


*Figure 6.1: CORDIC Waveform*

This waveform demonstrates the correct operation of the CORDIC module. Each time the **start** signal is asserted, the CORDIC unit begins computation based on the input angle provided in floating-point radians. After 17 clock cycles, the computed result is output, and the **finish** signal is asserted (set to 1) to notify the control FSM of the computation completion.

Below are the testbench output results, all of which correspond exactly to the values shown in the waveform:

VCD info: dumpfile cordic_wave.vcd opened for output.

==== TEST CASE 0 ====

Angle: 00111110011010000101011010010110 // 13° = 0.2268928028 rad

Sin: 00111110011001100101010011110101 // 0.22493346 ≈ 0.2249510543

Cos: 00111111011110010111000010001111 // 0.97437376 ≈ 0.9743700648

Time: 45

==== TEST CASE 1 ====

Angle: 00111110111110100011010111011101 // 28° = 0.4886921906 rad

```
Sin: 00111110111100000110010100111010    // 0.4695223 ≈ 0.4694715628
Cos: 00111111011000100000011100010000    // 0.88292027 ≈ 0.8829475929
Time: 85

==== TEST CASE 2 ====
Angle: 00111111010010010000111111011011  // 45° = 1.169370599 rad
Sin: 00111111001101010000010000111011    // 0.7070958 ≈ 0.7071067812
Cos: 00111111001101010000010110100010    // 0.7071172 ≈ 0.7071067812
Time: 125

==== TEST CASE 3 ====
Angle: 00111111100101011010110111110000  // 67° = 0.7853981634 rad
Sin: 00111111011010111010011110100000    // 0.9205265 ≈ 0.9205048535
Cos: 00111110110010000000011100100001    // 0.3906794 ≈ 0.3907311285
Time: 165

==== TEST CASE 4 ====
Angle: 00111111101110111010100001100110  // 84° = 1.466076572 rad
Sin: 00111111011111101001100011110010    // 0.99452126 ≈ 0.9945218954
Cos: 00111101110101100001010010010001    // 0.104531415 ≈ 0.1045284633
Time: 205

==== TEST CASE 5 ====
Angle: 00111111110110001011001100111000  // 97° = 1.692969374 rad
Sin: 00111111011111100001011101110100    // 0.99254537 ≈ 0.9925461516
Cos: 10111101111110011001100001110101    // -0.12187282 ≈ -0.1218693434
Time: 245

==== TEST CASE 6 ====
Angle: 00111111111111000111000111000110  // 113° = 1.972222055 rad
Sin: 00111111011010111010011110100000    // 0.9205265 ≈ 0.9205048535
Cos: 10111110110010000000011100100001    // -0.3906794 ≈ -0.3907311285
Time: 285

==== TEST CASE 7 ====
Angle: 01000000000100000001100000101001  // 129° = 2.251474735 rad
Sin: 00111111010001101111000101101110    // 0.7771214 ≈ 0.7771459615
Cos: 10111111001000010001110100010100    // -0.62934995 ≈ -0.629320391
Time: 325

==== TEST CASE 8 ====
Angle: 01000000001010001010101100101011  // 151° = 2.635447171 rad
Sin: 00111110111110000011110111010101    // 0.48484674 ≈ 0.4848096202
Cos: 10111111010111111110010110110101    // -0.8745988 ≈ -0.8746197071
Time: 365

==== TEST CASE 9 ====
Angle: 01000000010000100101110000100000  // 174° = 3.036872898 rad
Sin: 00111101110101100001010010010001    // 0.104531415 ≈ 0.1045284633
Cos: 10111111011111101001100011110010    // -0.99452126 ≈ -0.9945218954
Time: 405

==== TEST CASE 10 ====
Angle: 01000000010011000110100110111000  // 183° = 3.193952531 rad
Sin: 10111101010101100101111101110100    // -0.052337125 ≈ -0.05233595624
Cos: 10111111011111111010011000101010    // -0.9986292 ≈ -0.9986295348
Time: 445
```

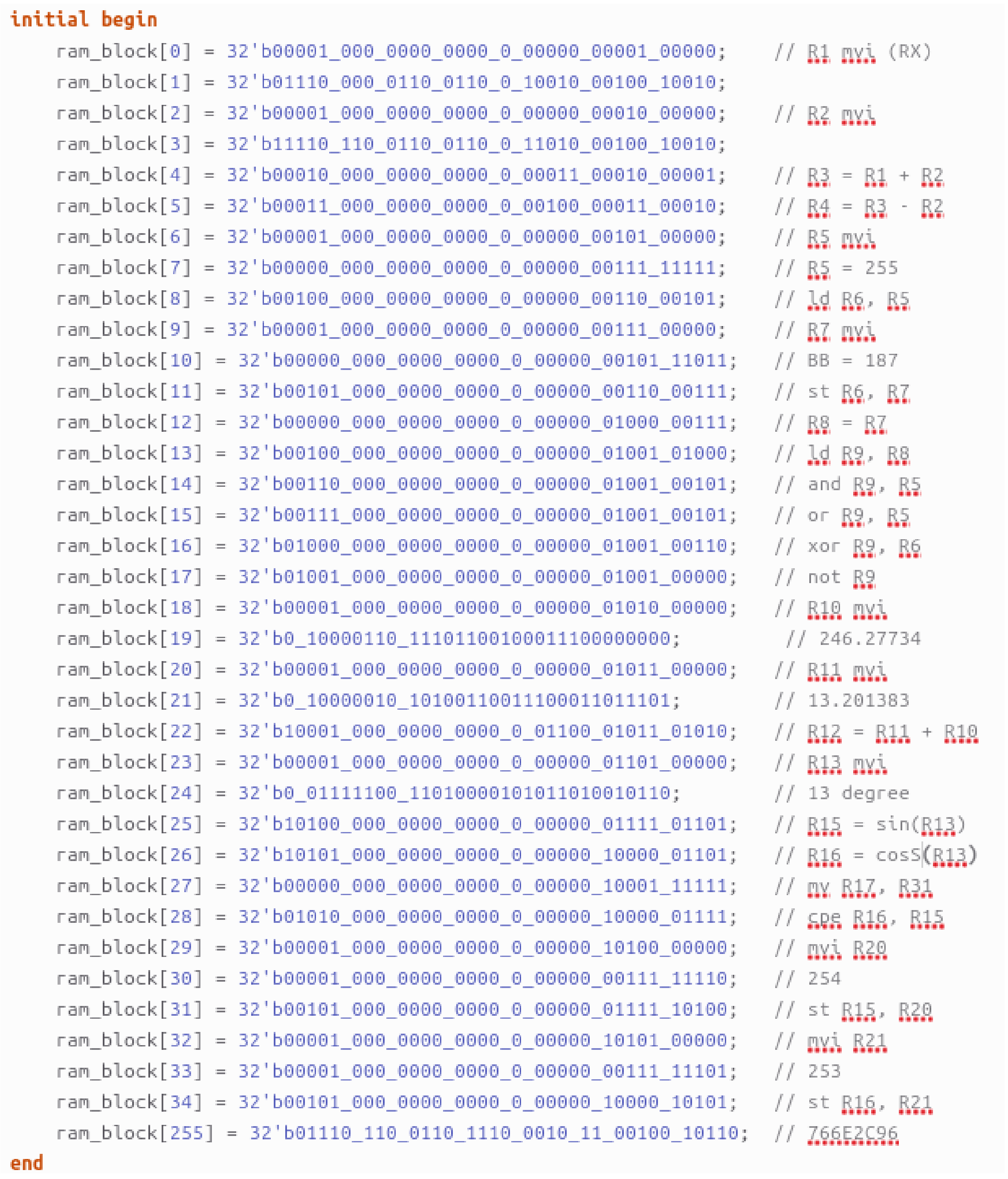
```
==== TEST CASE 11 ====
Angle: 01000000010111010010101100001010  // 198° = 3.455751919 rad
Sin: 10111110100111100011010000010111    // -0.30899116 ≈ -0.3090169944
Cos: 10111111011100110111100011111000    // -0.9510646 ≈ -0.9510565163
Time: 485

==== TEST CASE 12 ====
Angle: 01000000011010111011000001110011  // 211° = 3.682644722 rad
Sin: 10111111000000111101110000010111    // -0.51507705 ≈ -0.5150380749
Cos: 10111111010110110110110111000001    // -0.85714346 ≈ -0.8571673007
Time: 525

==== TEST CASE 13 ====
Angle: 01000000100000110011111110101111  // 235° = 4.101523742 rad
Sin: 10111111010100011011001000011111    // -0.81912416 ≈ -0.8191520443
Cos: 10111111000100101101100001111000    // -0.57361555 ≈ -0.5735764364
Time: 565

==== TEST CASE 14 ====
Angle: 01000000100101011010110111110000  // 268° = 4.677482395 rad
Sin: 10111111011111111101100000010000    // -0.9993906 ≈ -0.999390827
Cos: 10111101000011101111001101110010    // -0.034900136 ≈ -0.0348994967
Time: 605

==== TEST CASE 15 ====
Angle: 01000000100110101011010010111011  // 277° = 4.834562028 rad
Sin: 10111111011111100001011101110100    // -0.99254537 ≈ -0.9925461516
Cos: 00111101111110011001100001110101    // 0.12187282 ≈ 0.1218693434
Time: 645
```

```
==== TEST CASE 16 ====
Angle: 01000000101000101000011001101010  // 291° = 5.078908123 rad
Sin: 10111111011011110000000000100011    // -0.93359584 ≈ -0.9335804265
Cos: 00111110101101110111011010100001    // 0.35832694 ≈ 0.3583679495
Time: 685

==== TEST CASE 17 ====
Angle: 01000000101011111110110111011111  // 315° = 5.497787144 rad
Sin: 10111111001101010000010000111011    // -0.7070958 ≈ -0.7071067812
Cos: 00111111001101010000010110100010    // 0.7071172 ≈ 0.7071067812
Time: 725

==== TEST CASE 18 ====
Angle: 01000000101111001100011001011010  // 338° = 5.899212872 rad
Sin: 10111110101111111100010101010001    // -0.37455228 ≈ -0.3746065934
Cos: 00111111011011010101110101010110    // 0.92720544 ≈ 0.9271838546
Time: 765

==== TEST CASE 19 ====
Angle: 01000000110010001000000011100000  // 359° = 6.265732015 rad
Sin: 10111100100011101111010010010111    // -0.017450614 ≈ -0.01745240644
Cos: 00111111011111111111011000000000    // 0.9998474 ≈ 0.9998476952
Time: 805
../dv/cordic_tb.sv:34: $finish called at 805 (1s)
```

Based on the 20 test cases, the CORDIC unit demonstrates accurate performance. Although minor errors exist, the results are acceptable with approximately four to five decimal places of precision. The accuracy of the CORDIC algorithm can be further improved by increasing the number of iterations, allowing the computed values to converge closer to the expected results. However, additional iterations incur longer computation times and slow down the processor’s operation. Therefore, 16 iterations represent a reasonable balance between accuracy and performance.

## 6.2 Processor

For the processor, a program was preloaded into the RAM to enable immediate execution. The fundamental instructions supported by the design were thoroughly tested to verify correct functionality. The testing involved running this program step-by-step and observing the processor’s responses to ensure expected behavior. Below is the exact program code used during the testing phase.



```
initial begin
    ram_block[0] = 32'b00001_000_0000_0000_0_00000_00001_00000;    // R1 mvi (RX)
    ram_block[1] = 32'b01110_000_0110_0110_0_10010_00100_10010;
    ram_block[2] = 32'b00001_000_0000_0000_0_00000_00010_00000;    // R2 mvi
    ram_block[3] = 32'b11110_110_0110_0110_0_11010_00100_10010;
    ram_block[4] = 32'b00010_000_0000_0000_0_00011_00010_00001;    // R3 = R1 + R2
    ram_block[5] = 32'b00011_000_0000_0000_0_00100_00011_00010;    // R4 = R3 - R2
    ram_block[6] = 32'b00001_000_0000_0000_0_00000_00101_00000;    // R5 mvi
    ram_block[7] = 32'b00000_000_0000_0000_0_00000_00111_11111;    // R5 = 255
    ram_block[8] = 32'b00100_000_0000_0000_0_00000_00110_00101;    // ld R6, R5
    ram_block[9] = 32'b00001_000_0000_0000_0_00000_00111_00000;    // R7 mvi
    ram_block[10] = 32'b00000_000_0000_0000_0_00000_00101_11011;   // BB = 187
    ram_block[11] = 32'b00101_000_0000_0000_0_00000_00110_00111;   // st R6, R7
    ram_block[12] = 32'b00000_000_0000_0000_0_00000_01000_00111;   // R8 = R7
    ram_block[13] = 32'b00100_000_0000_0000_0_00000_01001_01000;   // ld R9, R8
    ram_block[14] = 32'b00110_000_0000_0000_0_00000_01001_00101;   // and R9, R5
    ram_block[15] = 32'b00111_000_0000_0000_0_00000_01001_00101;   // or R9, R5
    ram_block[16] = 32'b01000_000_0000_0000_0_00000_01001_00110;   // xor R9, R6
    ram_block[17] = 32'b01001_000_0000_0000_0_00000_01001_00000;   // not R9
    ram_block[18] = 32'b00001_000_0000_0000_0_00000_01010_00000;   // R10 mvi
    ram_block[19] = 32'b0_10000110_11101100100011100000000;          // 246.27734
    ram_block[20] = 32'b00001_000_0000_0000_0_00000_01011_00000;   // R11 mvi
    ram_block[21] = 32'b0_10000010_10100110011100011011101;         // 13.201383
    ram_block[22] = 32'b10001_000_0000_0000_0_01100_01011_01010;   // R12 = R11 + R10
    ram_block[23] = 32'b00001_000_0000_0000_0_00000_01101_00000;   // R13 mvi
    ram_block[24] = 32'b0_01111100_11010000101011010010110;         // 13 degree
    ram_block[25] = 32'b10100_000_0000_0000_0_00000_01111_01101;   // R15 = sin(R13)
    ram_block[26] = 32'b10101_000_0000_0000_0_00000_10000_01101;   // R16 = cosS(R13)
    ram_block[27] = 32'b00000_000_0000_0000_0_00000_10001_11111;   // mv R17, R31
    ram_block[28] = 32'b01010_000_0000_0000_0_00000_10000_01111;   // cpe R16, R15
    ram_block[29] = 32'b00001_000_0000_0000_0_00000_10100_00000;   // mvi R20
    ram_block[30] = 32'b00001_000_0000_0000_0_00000_00111_11110;   // 254
    ram_block[31] = 32'b00101_000_0000_0000_0_00000_01111_10100;   // st R15, R20
    ram_block[32] = 32'b00001_000_0000_0000_0_00000_10101_00000;   // mvi R21
    ram_block[33] = 32'b00001_000_0000_0000_0_00000_00111_11101;   // 253
    ram_block[34] = 32'b00101_000_0000_0000_0_00000_10000_10101;   // st R16, R21
    ram_block[255] = 32'b01110_110_0110_1110_0010_11_00100_10110;  // 766E2C96
end
```

*Figure 6.2: Test code*

With the given test code, the resulting waveform is as follows.

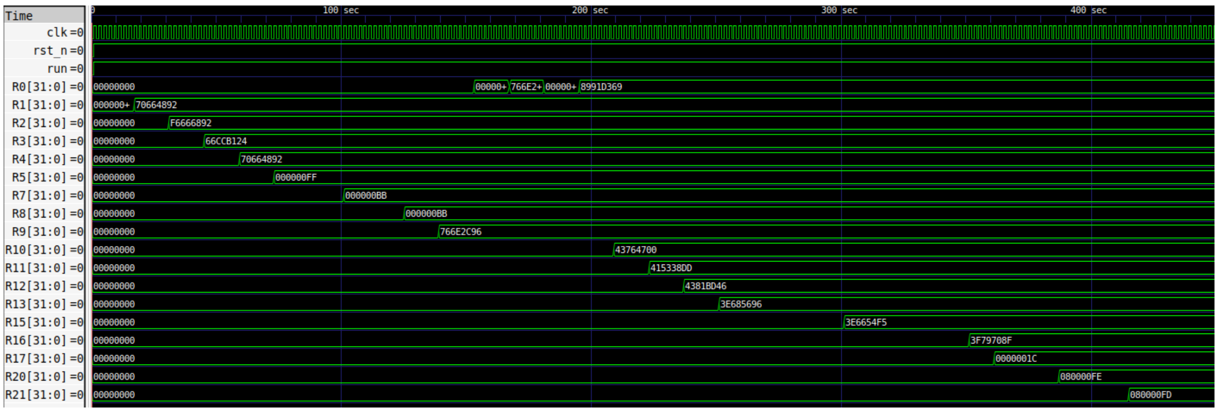

Time
100 sec
200 sec
300 sec
400 sec
clk =0
rst_n =0
run =0
R0[31:0] =0 00000000 00000+ 766E2+ 00000+ 8991D369
R1[31:0] =0 000000+ 70664892
R2[31:0] =0 00000000 F6666892
R3[31:0] =0 00000000 66CCB124
R4[31:0] =0 00000000 70664892
R5[31:0] =0 00000000 000000FF
R7[31:0] =0 00000000 000000BB
R8[31:0] =0 00000000 000000BB
R9[31:0] =0 00000000 766E2C96
R10[31:0] =0 00000000 43764700
R11[31:0] =0 00000000 415338DD
R12[31:0] =0 00000000 4381BD46
R13[31:0] =0 00000000 3E685696
R15[31:0] =0 00000000 3E6654F5
R16[31:0] =0 00000000 3F79708F
R17[31:0] =0 00000000 0000001C
R20[31:0] =0 00000000 080000FE
R21[31:0] =0 00000000 080000FD

*Figure 6.3: Processor Waveform*

Although other instructions have been thoroughly tested and verified for correctness, this section focuses on the sine and cosine computations. According to the test code, register **R15** holds the sine value and register **R16** holds the cosine value of the angle stored in **R13**. The results produced are **sin(R13) = 3E6654F5** (00111110011001100101010011110101) and **cos(R13) = 3F79708F** (00111111011110010111000010001111) , which match the values obtained from the previously shown testbench:

==== TEST CASE 0 ====

Angle: 00111110011010000101011010010110 // 13° = 0.2268928028 rad

Sin: 00111110011001100101010011110101 // 0.22493346 ≈ 0.2249510543

Cos: 00111111011110010111000010001111 // 0.97437376 ≈ 0.9743700648

Time: 45

With the consistent evidence from both the waveform simulation and testbench outputs, it is confirmed that the design operates correctly according to the specifications defined earlier. These results demonstrate the reliability and accuracy of the implemented CORDIC unit and its integration within the processor architecture.

## 7. RTL View check

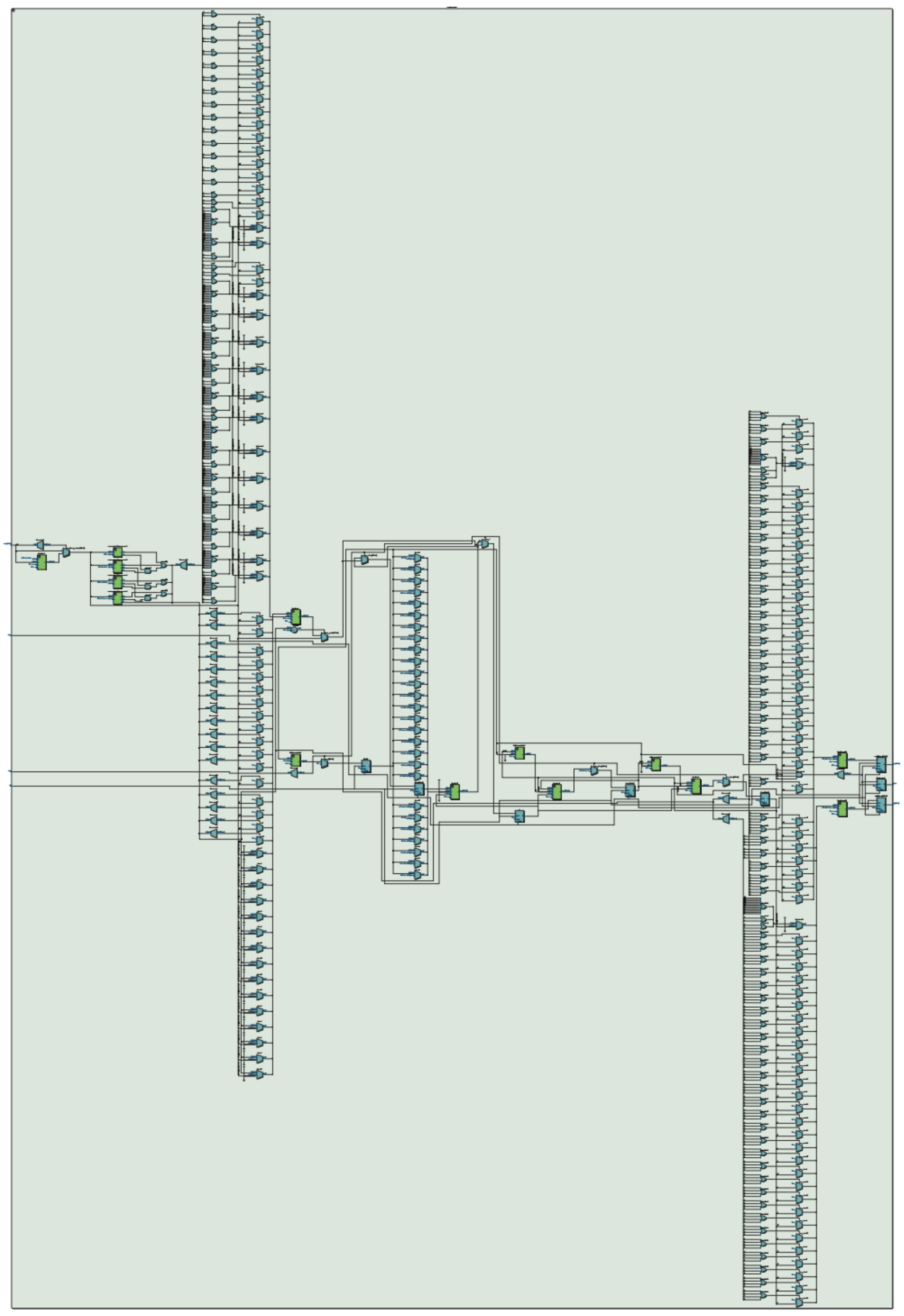


*Figure 7.1: CORDIC RTL view*

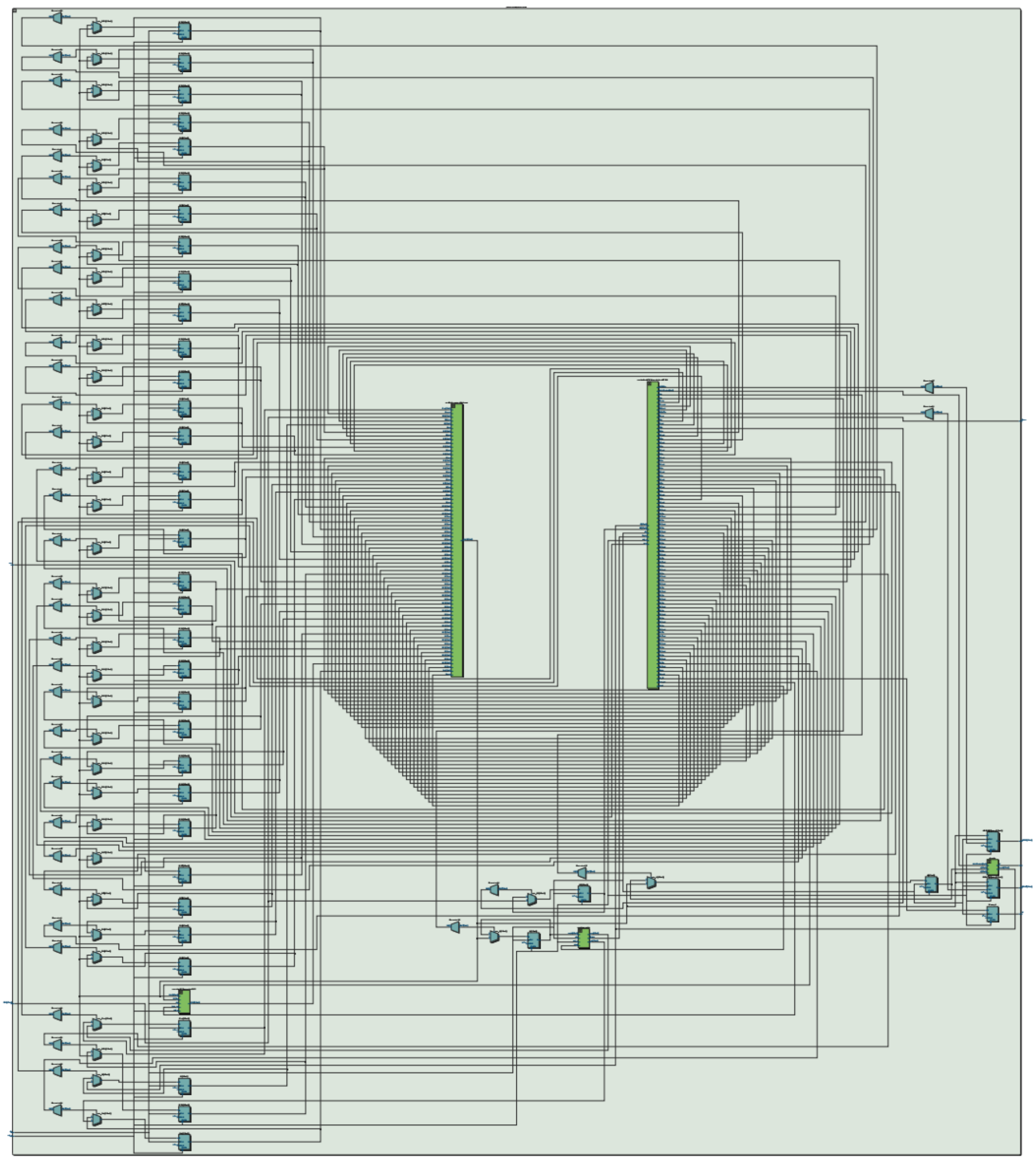


*Figure 7.2: Processor RTL view*

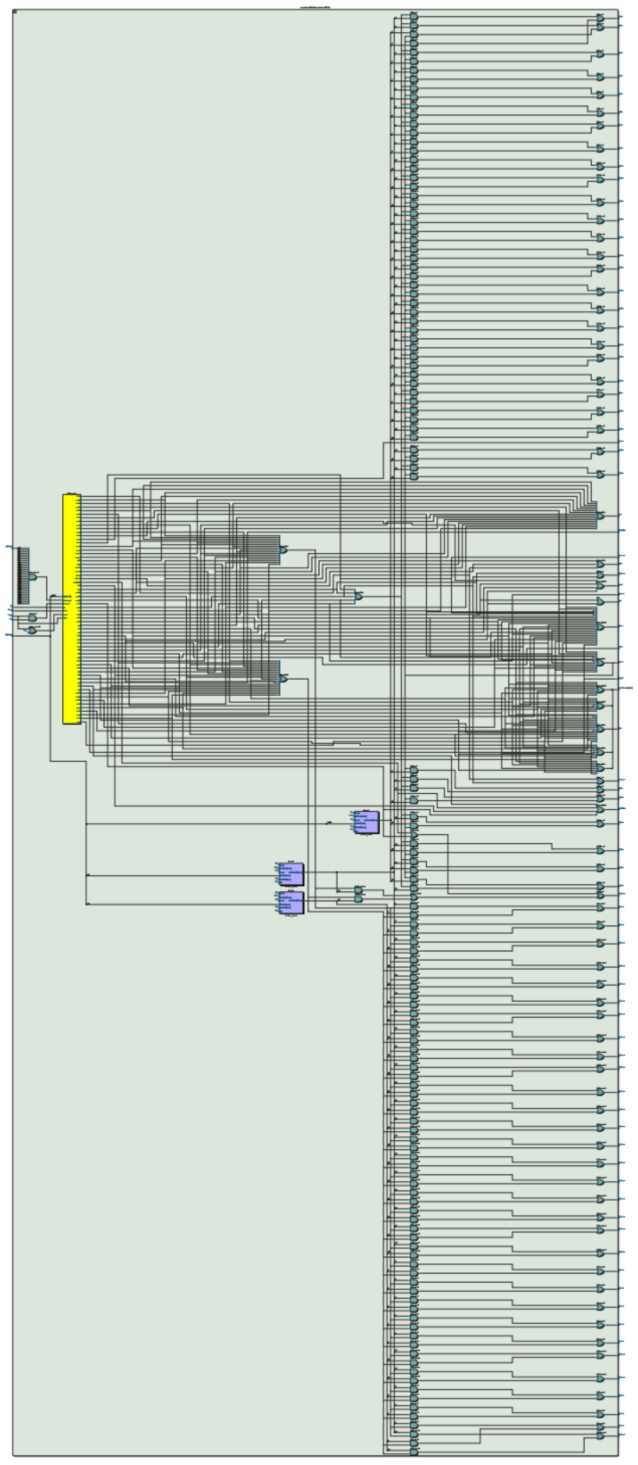


*Figure 7.3: ControlFSM RTL View*

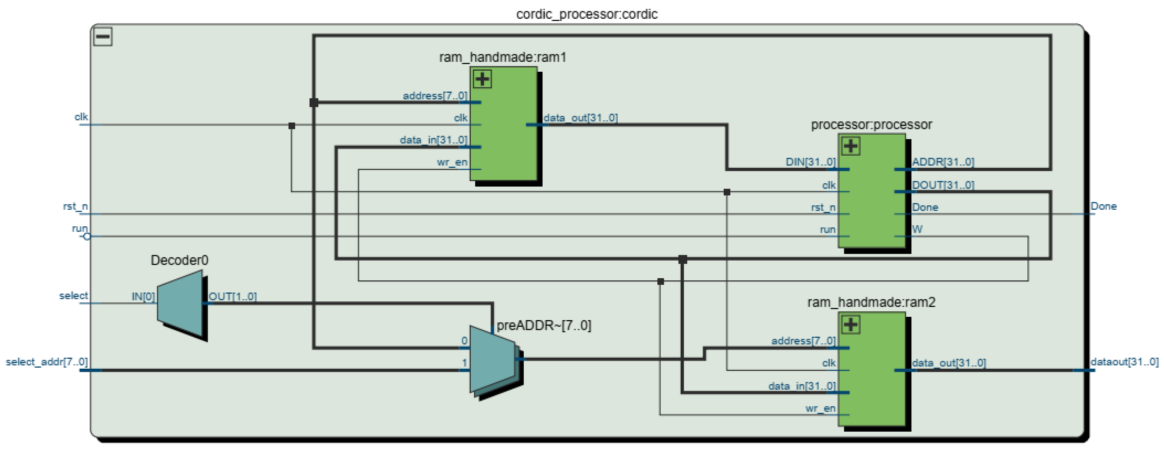


*Figure 7.4: CORDIC Processor RTL View*

The figures above represent the RTL views of the CORDIC module and the overall processor, respectively. A detailed inspection of each connection within these views confirms that all signals and components are correctly wired in accordance with the previously drawn block diagram. This precise alignment ensures structural consistency between the design schematic and its hardware description.

Moreover, the RTL views correspond accurately with the simulation results, validating that the implementation faithfully reflects the intended architecture. It can be confidently stated that the RTL representation matches the block diagram with absolute accuracy, reinforcing the correctness and integrity of the design.

# 8. FPGA Proven

The following is the block diagram representing the overall design intended for deployment on the DE10 FPGA development kit. This diagram illustrates the interconnection of the processor, two memory modules and I/O interfaces tailored specifically to the hardware resources available on the DE10 platform.

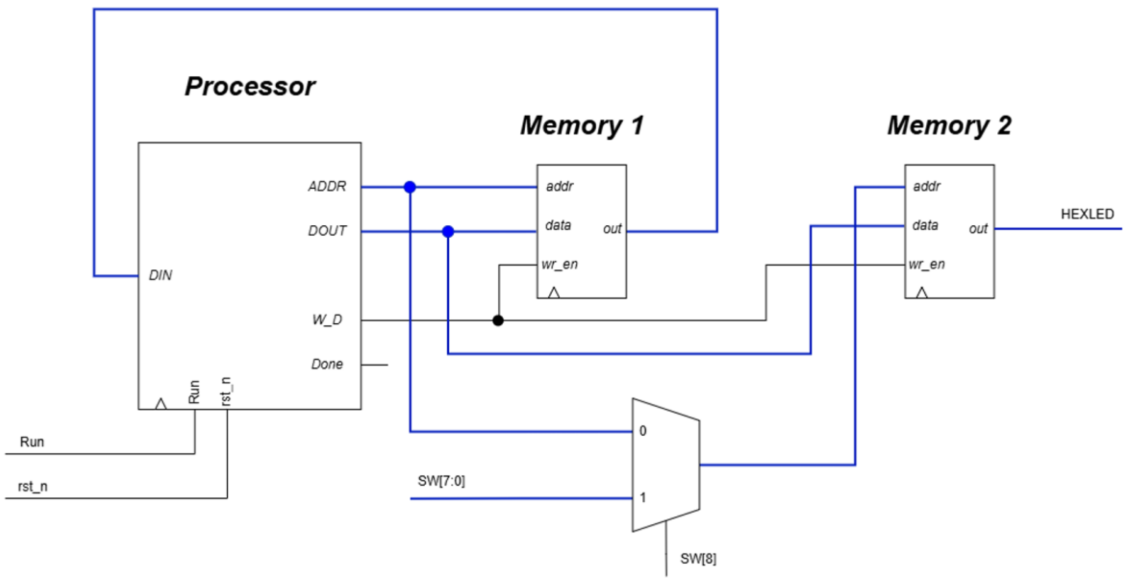
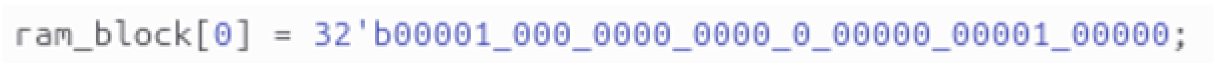


*Figure 8.1: CORDIC Processor with 2 RAMs for FPGA implementation*

This block diagram ensures that the processor maintains proper communication and operation with the primary RAM module. In addition to the main memory, a second RAM is included specifically to facilitate output display selection via address control. This secondary RAM allows the system to independently access and present data on the 7-segment HEX LEDs without interfering with the processor's primary memory operations.

Using the test code described above, testing was performed sequentially on the first memory address as well as addresses 254 and 253, where store the sine and cosine values corresponding to an input angle of 13 degrees.



```
ram_block[0] = 32'b00001_000_0000_0000_0_00000_00001_00000;
```

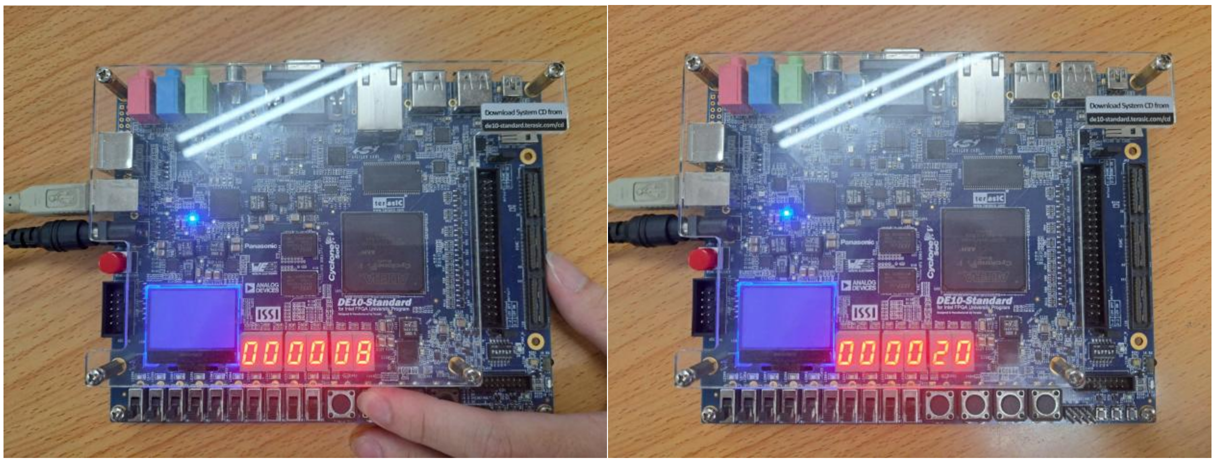
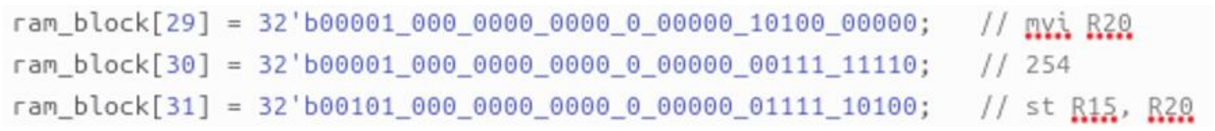


*Figure 8.2: Read data from RAM Address 0*



```
ram_block[29] = 32'b00001_000_0000_0000_0_00000_10100_00000;    // mvi R20
ram_block[30] = 32'b00001_000_0000_0000_0_00000_00111_11110;    // 254
ram_block[31] = 32'b00101_000_0000_0000_0_00000_01111_10100;    // st R15, R20
```

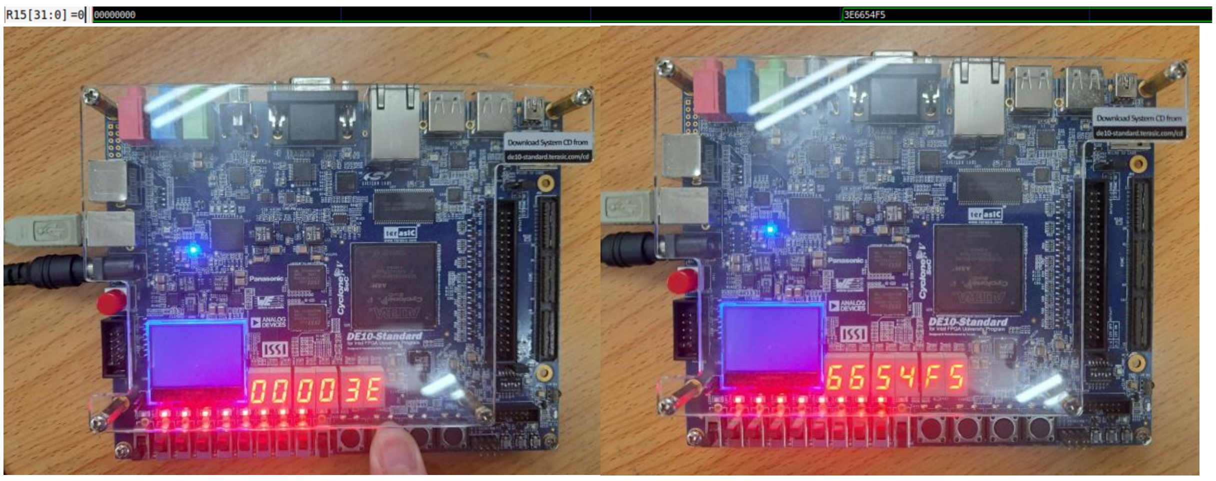
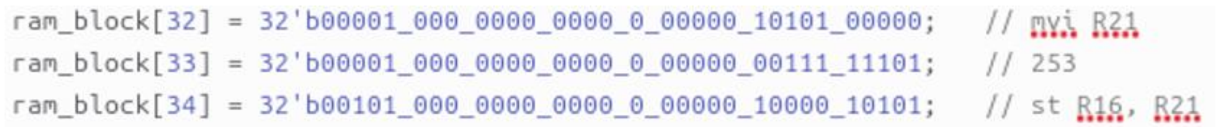


*Figure 8.3: Read data from RAM Address 254*



```
ram_block[32] = 32'b00001_000_0000_0000_0_00000_10101_00000;   // mvi R21
ram_block[33] = 32'b00001_000_0000_0000_0_00000_00111_11101;   // 253
ram_block[34] = 32'b00101_000_0000_0000_0_00000_10000_10101;   // st R16, R21
```

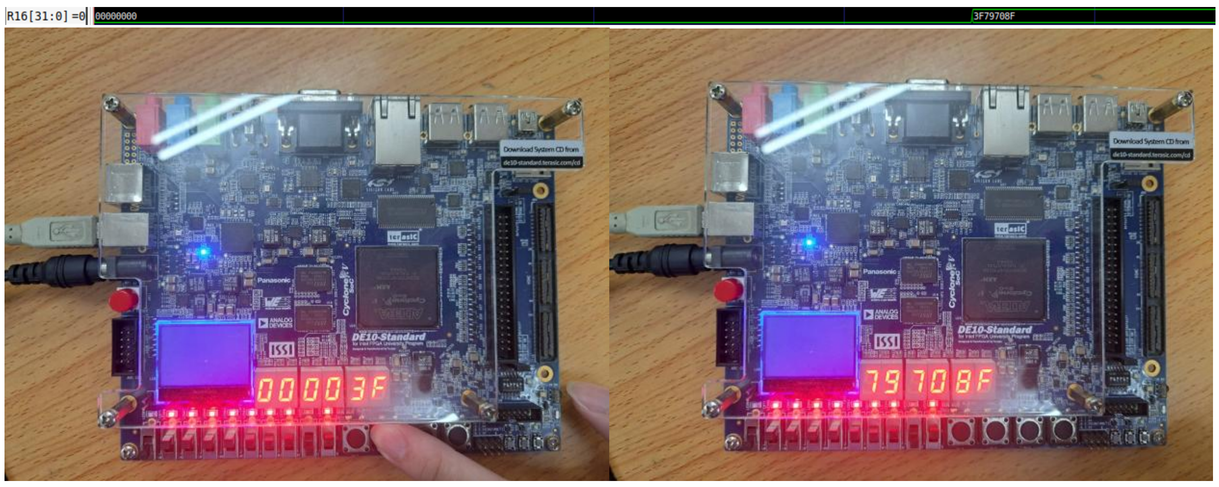


*Figure 8.4: Read data from RAM Address 253*

The design was successfully deployed onto the FPGA development kit. Upon programming the device, the memory contents at the specified addresses were accurately read and verified. The data retrieved from the targeted memory locations matched the expected values precisely, confirming the correct operation of both the memory interface and the processor's functionality on the hardware platform. This successful validation demonstrates the effectiveness of the implementation and its readiness for further testing and integration.

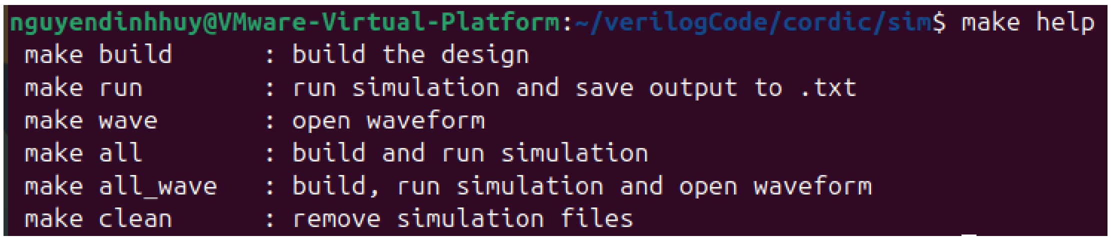
# 9. Working directory

All of the source code for this project has been uploaded and is available at the following GitHub repository:

huynguyendinhhcmut/CORDIC-Processor

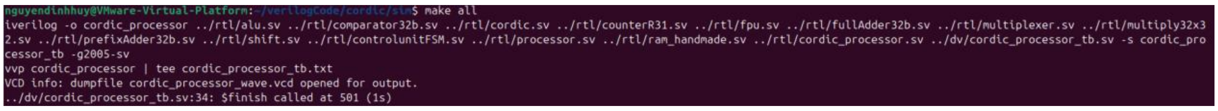
The repository is organized into folders separating the processor design files, testbenches, and simulation scripts.

To run the simulations, please download the repository and navigate to the sim directory. Within this directory, use the command make help to display the available targets and operations supported by the Makefile. This will provide guidance on how to compile, simulate, and verify the design components effectively.



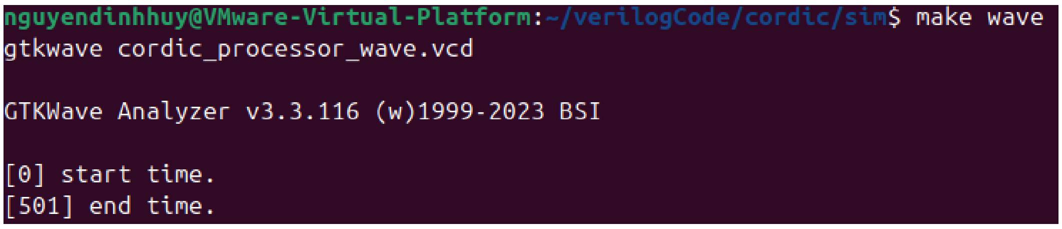
```
nguyendinhhuy@VMware-Virtual-Platform:~/verilogCode/cordic/sim$ make help
 make build       : build the design
 make run         : run simulation and save output to .txt
 make wave        : open waveform
 make all         : build and run simulation
 make all_wave    : build, run simulation and open waveform
 make clean       : remove simulation files
```

*Figure 9.1: make help*



```
nguyendinhhuy@VMware-Virtual-Platform:~/verilogCode/cordic/sim$ make all
iverilog -o cordic_processor ../rtl/alu.sv ../rtl/comparator32b.sv ../rtl/cordic.sv ../rtl/counterR31.sv ../rtl/fpu.sv ../rtl/fullAdder32b.sv ../rtl/multiplexer.sv ../rtl/multiply32x3
2.sv ../rtl/prefixAdder32b.sv ../rtl/shift.sv ../rtl/controlunitFSM.sv ../rtl/processor.sv ../rtl/ram_handmade.sv ../rtl/cordic_processor.sv ../dv/cordic_processor_tb.sv -s cordic_pro
cessor_tb -g2005-sv
vvp cordic_processor | tee cordic_processor_tb.txt
VCD info: dumpfile cordic_processor_wave.vcd opened for output.
../dv/cordic_processor_tb.sv:34: $finish called at 501 (1s)
```

*Figure 9.2: make all*



```
nguyendinhhuy@VMware-Virtual-Platform:~/verilogCode/cordic/sim$ make wave
gtkwave cordic_processor_wave.vcd

GTKWave Analyzer v3.3.116 (w)1999-2023 BSI

[0] start time.
[501] end time.
```

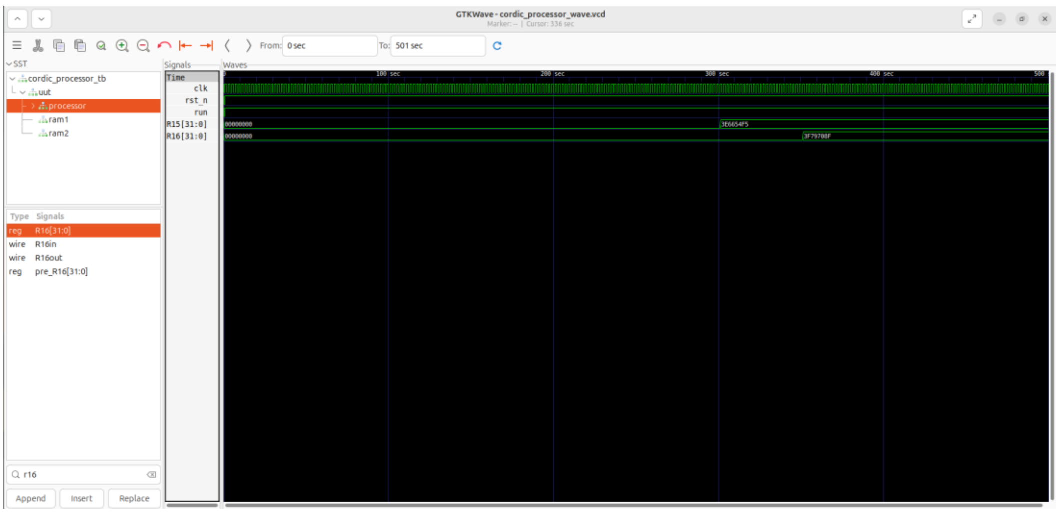


*Figure 9.3: make wave*

# 10. Conclusion

This project successfully implemented a 32-bit multicycle processor integrated with a CORDIC computation unit for efficient trigonometric calculations. The design leverages floating-point arithmetic to optimize hardware operations, particularly by replacing traditional bit-shifting with exponent adjustments. Comprehensive simulation and testing, including waveform analysis and testbench validation, confirmed the accuracy and reliability of the CORDIC module and the processor.

The inclusion of a secondary RAM for output display on the FPGA development kit demonstrates practical considerations for hardware interfacing and debugging. While the finite state machine controlling instruction execution is functional, opportunities for optimization remain, especially by consolidating common instruction stages to reduce complexity.

Overall, the project achieves a balanced trade-off between computational accuracy and performance, providing a solid foundation for further enhancements and practical applications in embedded systems.

## 11. Appendix

[1]. Sharanagouda N Patil, P.V.Hunagund, R.M.Vani, A VERILOG BASED IMPLEMENTATION OF TRANSCENDENTAL FUNCTION CALCULATOR USING CORDIC ALGORITHM FOR SDR, September 2015

[2]. Suman Deb, Saurabh Chaudhury, Department of Electrical Engineering National Institute of Technology Silchar, Silchar, India, High-Speed Comparator Architectures for Fast Binary Comparison

[3]. Lab reports, Digital Systems (EE2420), Ho Chi Minh University of Technology