EE5516 VLSI Architectures for Signal Processing and Machine Learning

Lab Report - Experiment No.4

Nakul C - 122101024

Abstract

The goal of this experiment is to design and implement CORDIC (Coordinate Rotation Digital Computer) both Rotation and Vectoring in both iterative and pipelined using Verilog. CORDIC is a commonly used algorithm in digital signal processing for calculating trigonometric and logarithmic functions, Here we are implementing rotation CORDIC to find cosine and sine of an angle given an angle as the input and vectoring CORDIC to find the magnitude and angle of an point given its Cartesian coordinate as its input. The implementation of pipelining technique aims to improve the performance of the circuits by allowing multiple operations to be executed in parallel.

1 Introduction

CORDIC (Coordinate Rotation Digital Computer) is an algorithm used for efficiently computing various mathematical functions, such as trigonometric and hyperbolic functions, as well as matrix and vector operations. It was first introduced by Jack E. Volder in 1959. Pipelined CORDIC is a variant of the original algorithm that takes advantage of pipelining techniques to increase the speed of computation. Pipelining involves breaking down the CORDIC algorithm into smaller stages and processing multiple data inputs simultaneously, with each stage working on a different data input. This results in a significant improvement in processing speed and efficiency.

2 Implementation

CORDIC (Coordinate Rotation Digital Computer) is a hardware-efficient iterative method which uses rotations to calculate a wide range of elementary functions.

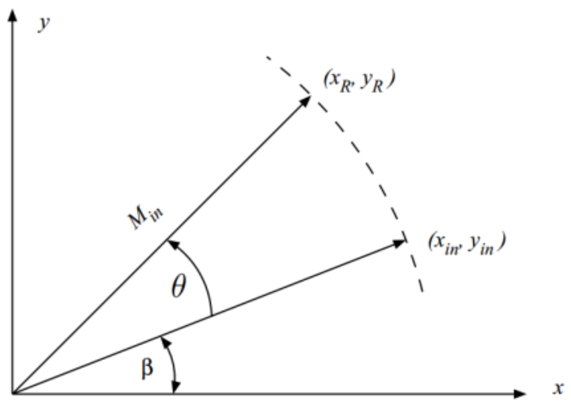


Figure 1: Rotating a vector $(x_{\text{in}}, y_{\text{in}})$ by an angle θ

After rotation we can find the new vector (x_R, y_R) as

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_{\text{in}} \\ y_{\text{in}} \end{bmatrix}$$

If we choose $x_{\text{in}} = 1$ and $y_{\text{in}} = 0$, then $x_R = \cos(\theta)$ and $y_R = \sin(\theta)$. The above equation can be rewritten in matrix form as:

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = k * \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x_{\text{in}} \\ y_{\text{in}} \end{bmatrix}$$

The above equation shows that rotation can be converted to multiplication, and a single rotation requires four multiplications. CORDIC algorithm tries to achieve rotation without multiplication. This first fundamental idea resorted by CORDIC is that rotating an input vector by an arbitrary angle θ_i is equal to rotating the vector by several small angles $\Delta\theta_i$, where $i = 0, 1, 2, \ldots, n$.

$$\theta = \sum_{i=0}^{n} \Delta\theta_i$$

The second fundamental idea is to choose the small elementary angles in such a way that $\tan \Delta\theta_i = 2^{-i}$. In this way multiplication by $\tan \Delta\theta_i$ becomes bitwise shift which is much faster compared to multiplication. It was found that sum of these smaller angle converges to desired angle by choosing the value of n. Here were choosing n as 15. These smaller angles are now called CORDIC angles, where $\Delta\theta_i = \tan^{-1} 2^{-i}, i = 0, 1, 2, \ldots, 15$.

Now we have to multiply the result with $\prod_{i=0}^{15} \cos \Delta\theta_i$. This multiplication acts as a gain which is equal to $K \approx 0.6072$. To remove this gain, we can give our inputs x and y scaled by a factor of $1/K$.

Therefore, after removing the scaling factor, we obtain the following equations of CORDIC as:

$$x[i+1] = x[i] \mp 2^{-i} y[i]$$
$$y[i+1] = y[i] \pm 2^{-i} x[i]$$

The above equations show that the algorithm will always perform a certain number of rotations with the predefined angles. The algorithm determines in which direction the rotation will happen, clock-wise or anti-clock wise during each iteration. The signs +/- will be decided based on the direction of rotation. This is accomplished by recording the angle of previous rotations and comparing the overall achieved rotation with the desired angle. If the desired rotation is larger than previously achieved rotation, then we need to rotate anti-clockwise in the next iteration. If the achieved angle is smaller than the desired angle then we have to rotate in clock-wise direction in the next iteration.

For example, if we have θ_{in} as 58° and at the beginning of the algorithm we have achieved zero degree of rotation, so desired angle $58° > 0$, so we will rotate the vector by 45° in anticlockwise direction. In second iteration $58° > 45°$, that is achieved angle is still smaller than desired angle, so we will rotate the vector by 26.565° in anticlockwise direction. The resulting angle greater than desired angle so we will rotate in clockwise direction in next iteration. And the process will go on like this until 16 iterations are performed.

$$\theta_{\text{error}} = \theta_{\text{desired}} - \theta_{\text{achieved}}$$

$$\theta_{\text{desired}} = \tan^{-1} \left(\sum_{i=0}^{n} \Delta\theta_i \right)$$

This equation can be included in previous set of equations as,

$$\theta[i+1] = \theta[i] \mp \tan^{-1} 2^{-i}$$

Where θ_i is the error signal. If the error signal is greater than zero then we have to rotate in anti-clockwise direction and the equation become,

$$x[i+1] = x[i] - 2^{-i} y[i]$$
$$y[i+1] = y[i] + 2^{-i} x[i]$$
$$\theta[i+1] = \theta[i] - \Delta\theta_i$$

When the error signal is less than zero, we have to rotate in clock wise direction and the equations become,

$$\begin{aligned}x[i+1] &= x[i] + 2^{-i}y[i]\\ y[i+1] &= y[i] - 2^{-i}x[i]\\ \theta[i+1] &= \theta[i] + \Delta\theta_i\end{aligned}$$

These three iterations are basis of CORDIC algorithm.

3 Structural Design

To effectively manage the CORDIC algorithm, Three 2:1 multiplexers (MUXes), one each for the X, Y, and θ registers are used. Each MUX is responsible for controlling the inputs to its respective register, providing flexibility in data manipulation and ensuring accurate computation of the CORDIC algorithm.

The system initializes in the IDLE state and transitions to an operational state upon the assertion of the "operands valid" qualifier. Once activated, the system progresses to the BUSY state, facilitating the transmission of Cartesian coordinates (X, Y) and the input angle θ to the input ports of the registers. These inputs are processed by the registers at the subsequent positive clock edge, guided by their respective enable signals. Also the control path for the vectoring and rotation CORDIC will be similar

During the rotation stage of the CORDIC algorithm, the system computes the new X and Y coordinates based on the current angle and magnitudes. Each MUX selects the appropriate inputs for its corresponding register, ensuring seamless interchange of values and precise rotation calculations.

In the vectoring stage, the system computes the magnitude and angle of the given Cartesian coordinates (X, Y). The magnitude is extracted directly from the output of the X register, while the angle is calculated based on the relationship between X and Y values.

By leveraging three 2:1 MUXes and carefully coordinating their inputs with the register values, our system efficiently executes the CORDIC algorithm, ensuring accurate computation of trigonometric functions and effective management of larger values within the algorithm's framework.

In pipelined CORDIC, the system is divided into 16 stages, designated from stage 0 to stage 15. Each stage comprises three registers, three adders, and two shifters. The pipelining strategy allows for parallel processing of CORDIC iterations, significantly improving performance and throughput.

In rotation CORDIC, the system adjusts the signs of the coordinates if the angle (θ) becomes negative to ensure correct computation of trigonometric functions. Conversely, in vectoring CORDIC, the system changes signs when the output y-coordinate (y_{out}) becomes negative, aligning with the algorithm's requirements for magnitude and angle computation.

Unlike traditional CORDIC implementations, pipelined CORDIC does not incorporate a control path. Instead, it relies solely on the pipelined structure to manage data flow and computation across stages. Each stage operates independently, processing input data and passing results to the subsequent stage without the need for centralized control.

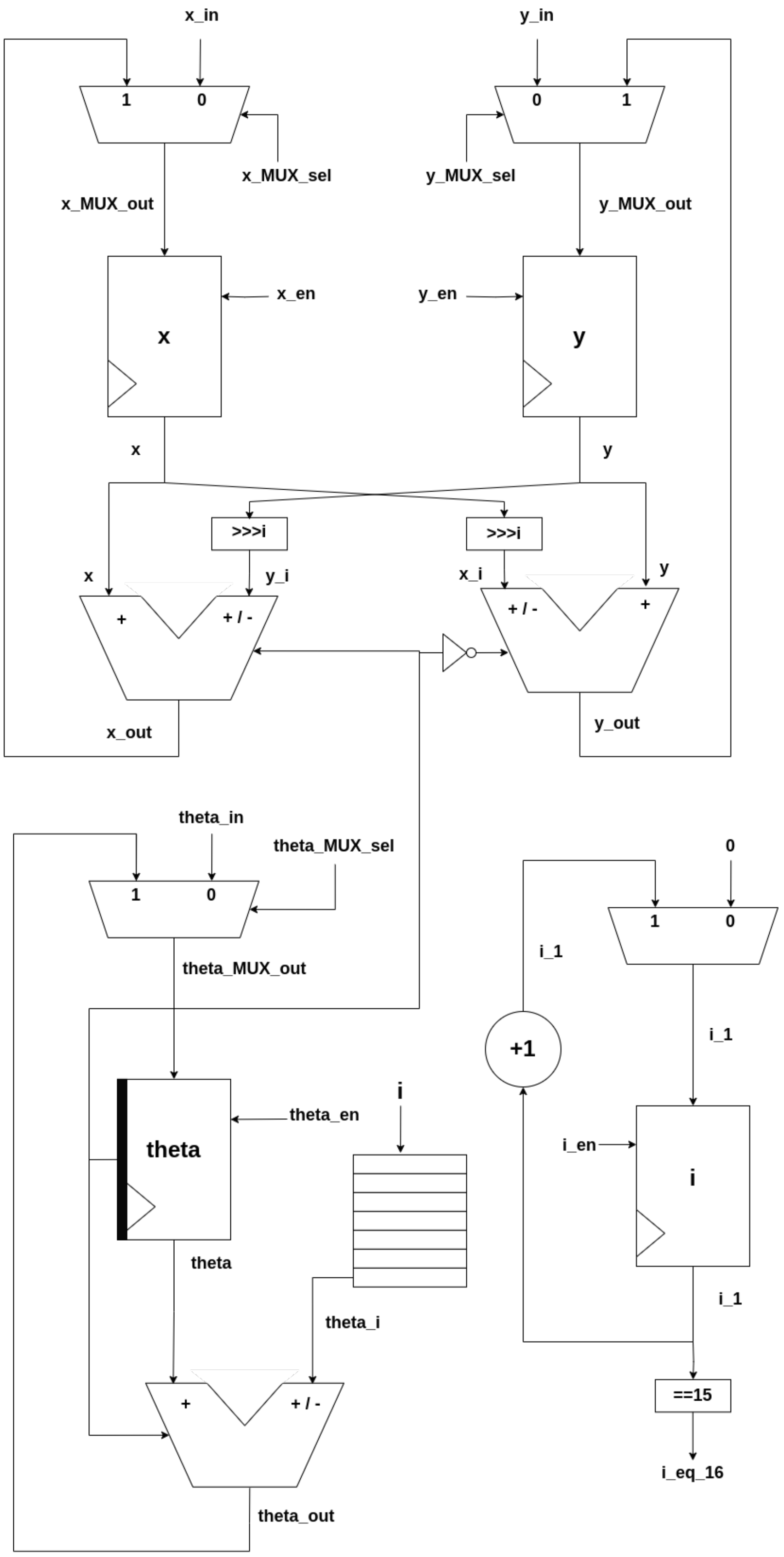


Figure 2: Architecture for data path of Rotation CORDIC

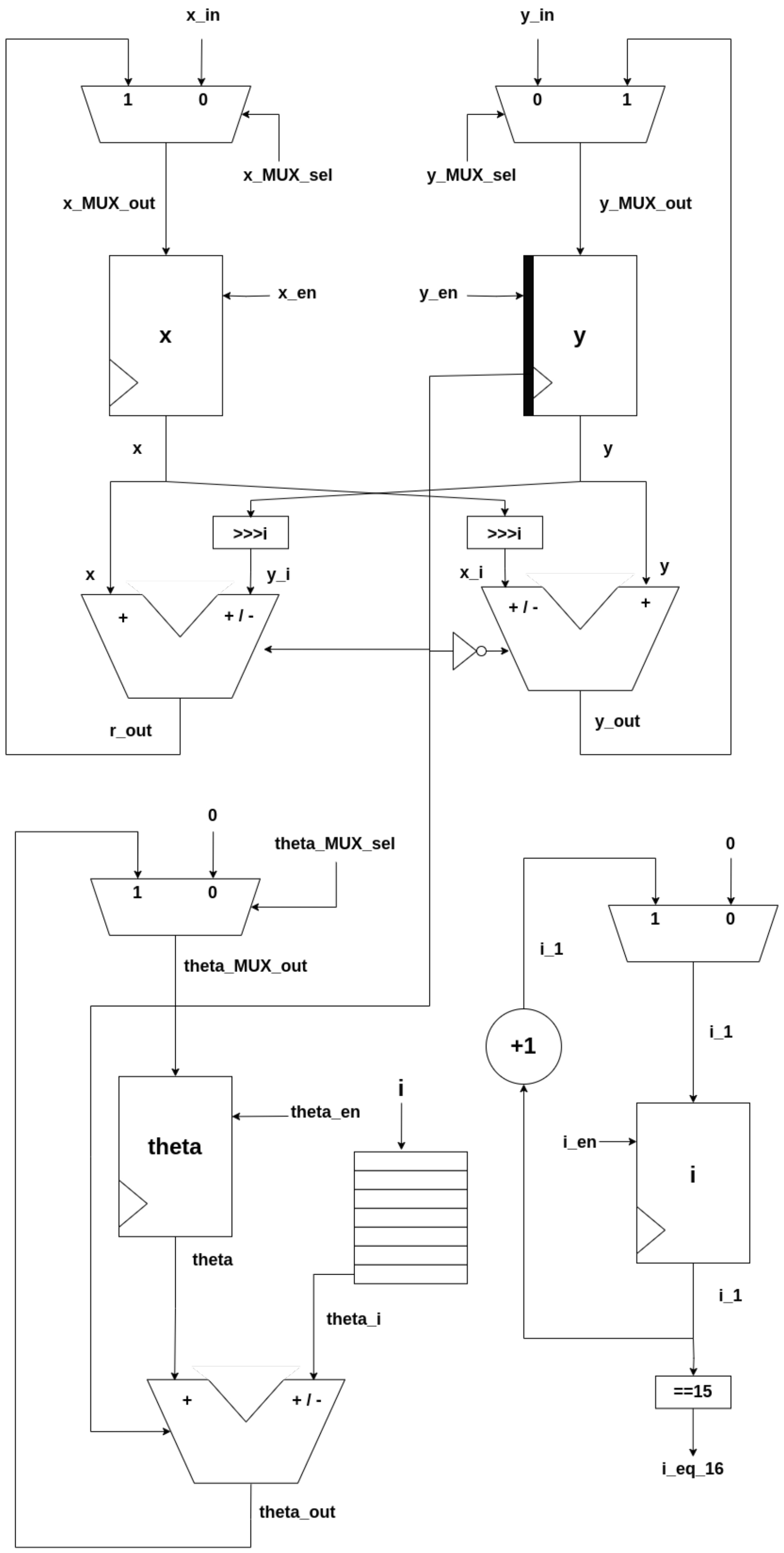


Figure 3: Architecture for data path of Vectoring CORDIC

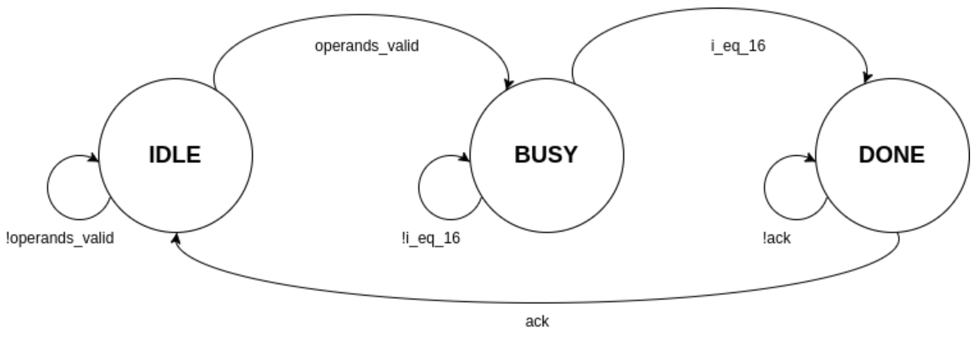


Figure 4: Architecture for control path of Rotation and vectoring CORDIC

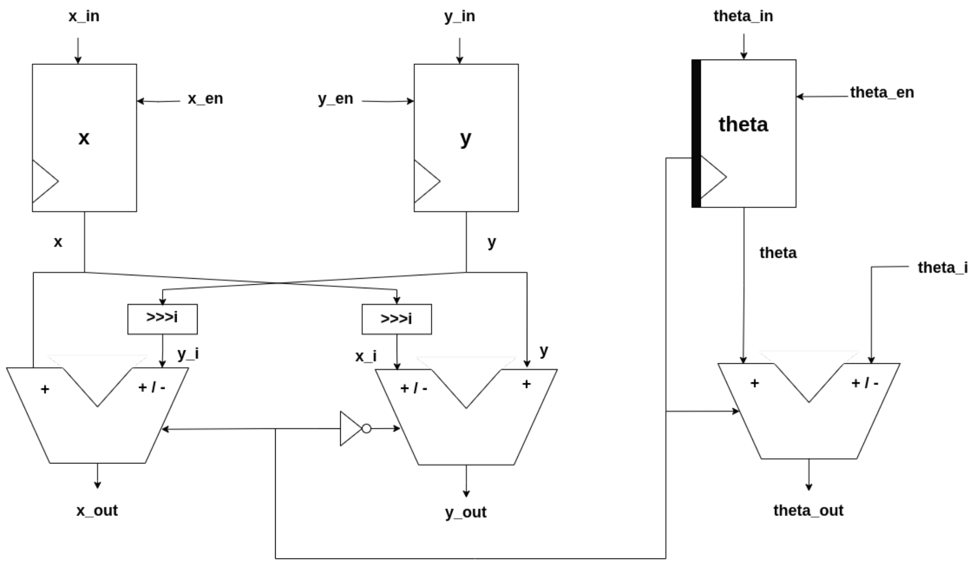


Figure 5: Architecture of Rotation pipelined CORDIC

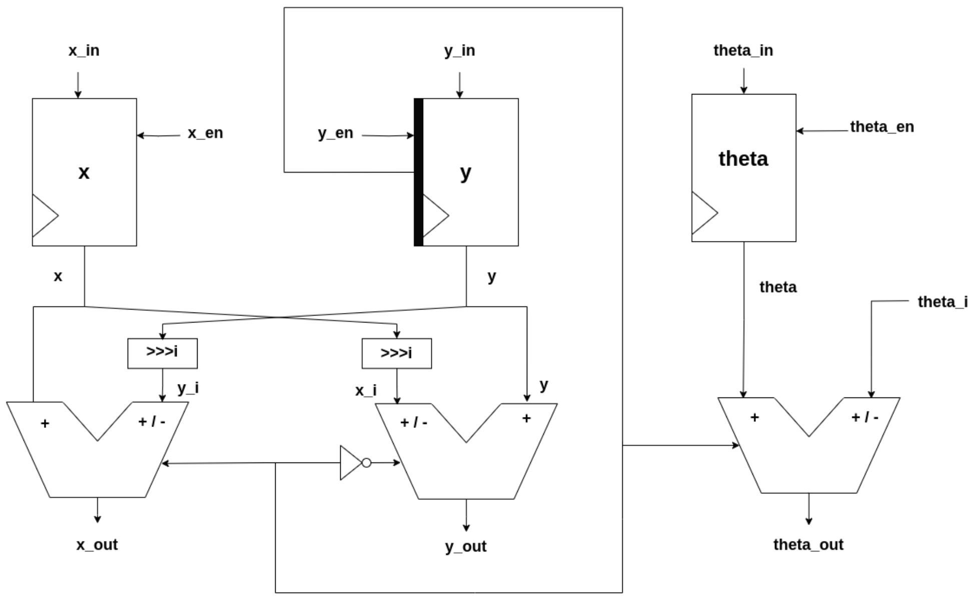


Figure 6: Architecture of Vectoring pipelined CORDIC

4 Experimental Procedure

4.1 Iterative Rotation CORDIC

```

module top_module(
    input signed [15:0] x_in, y_in, theta_in,
    input operands_val,
    input Clk, Rst, ack,
    output signed [15:0] x_out, y_out,
    output out_valid,
    output [1:0] state
);

    wire x_MUX_sel, y_MUX_sel, theta_MUX_sel, i_MUX_sel, x_en, y_en, theta_en,
       i_en;
    wire i_eq_16;


    datapath d1 (
        .x_in(x_in),
        .y_in(y_in),
        .theta_in(theta_in),
        .Clk(Clk),
        .x_MUX_sel(x_MUX_sel),
        .y_MUX_sel(y_MUX_sel),
        .theta_MUX_sel(theta_MUX_sel),
        .i_MUX_sel(i_MUX_sel),
        .x_en(x_en),
        .y_en(y_en),
        .theta_en(theta_en),
        .i_en(i_en),
        .x_out(x_out),
        .y_out(y_out),
        .theta_out(theta_out),
        .i_eq_16(i_eq_16)
    );

    controlpath c1 (
        .ack(ack),
        .Rst(Rst),
        .Clk(Clk),
        .operands_val(operands_val),
        .i_eq_16(i_eq_16),
        .x_MUX_sel(x_MUX_sel),
        .y_MUX_sel(y_MUX_sel),
        .theta_MUX_sel(theta_MUX_sel),
        .i_MUX_sel(i_MUX_sel),
        .x_en(x_en),
        .y_en(y_en),
        .theta_en(theta_en),
        .i_en(i_en),
        .out_valid(out_valid),
        .state(state)
    );

endmodule



//Datapath

module datapath(
    input signed [15:0] x_in, y_in, theta_in,
    input Clk,
    input x_MUX_sel, y_MUX_sel, theta_MUX_sel, i_MUX_sel, x_en, y_en, theta_en, i_en,
    output reg signed [15:0] x_out, y_out, theta_out,
    output i_eq_16
    );

    wire signed [15:0] x_MUX_out, y_MUX_out, theta_MUX_out, x_i, y_i;
    reg signed [15:0] x, y, theta, atan_out;
    reg [3:0] address, i;
    wire [3:0] i_1;

    // Look Up Table

    always @(*)
        begin
        address <= i;
        begin
            case(address)
                4'b0000: atan_out <= 16'b0011001001000100;
                    4'b0001: atan_out <= 16'b0001110110101100;
                    4'b0010: atan_out <= 16'b0000111110101101;
                    4'b0011: atan_out <= 16'b0000011111110101;
                    4'b0100: atan_out <= 16'b0000001111111110;
                    4'b0101: atan_out <= 16'b0000001000000000;
                    4'b0110: atan_out <= 16'b0000000100000000;
                    4'b0111: atan_out <= 16'b0000000010000000;
                    4'b1000: atan_out <= 16'b0000000001000000;
                    4'b1001: atan_out <= 16'b0000000000100000;
                    4'b1010: atan_out <= 16'b0000000000010000;
                    4'b1011: atan_out <= 16'b0000000000001000;
                    4'b1100: atan_out <= 16'b0000000000000100;
                    4'b1101: atan_out <= 16'b0000000000000010;
                    4'b1110: atan_out <= 16'b0000000000000001;
                    4'b1111: atan_out <= 16'b0000000000000000;
                endcase
            end
        end
    always @(posedge Clk)
        begin
            if (x_en)
                x <= x_MUX_out;
            if (y_en)
                y <= y_MUX_out;
            if (theta_en)
                theta <= theta_MUX_out;
            if (i_en)
                i <= i_1;
        end


    always @(*)
        if (theta[15] == 1)
        begin
            x_out = x + y_i;
            y_out = y - x_i;
            theta_out = theta + atan_out;
            end
        else
        begin
            x_out = x - y_i;
            y_out = y + x_i;
```

```verilog
              theta_out = theta - atan_out;
              end


    assign i_1 = i_MUX_sel ? i + 1 : 0;
    assign x_MUX_out = x_MUX_sel ? x_out : x_in;
    assign y_MUX_out = y_MUX_sel ? y_out : y_in;
    assign theta_MUX_out = theta_MUX_sel ? theta_out : theta_in;
    assign i_eq_16 = (i == 15);

    barrel_shifter b1 (x, i, x_i);
    barrel_shifter b2 (y, i, y_i);



endmodule

// implementation of Barrel Shifter

module barrel_shifter(
    input [15:0] input_data,
    input [3:0] control,
    output reg [15:0] output_data);

always @(*)
begin
  case(control)
    4'b0000: output_data <= input_data;
    4'b0001: output_data <= {1'b0, input_data[15:1]};
    4'b0010: output_data <= {2'b00, input_data[15:2]};
    4'b0011: output_data <= {3'b000, input_data[15:3]};
    4'b0100: output_data <= {4'b0000, input_data[15:4]};
    4'b0101: output_data <= {5'b00000, input_data[15:5]};
    4'b0110: output_data <= {6'b000000, input_data[15:6]};
    4'b0111: output_data <= {7'b0000000, input_data[15:7]};
    4'b1000: output_data <= {8'b00000000, input_data[15:8]};
    4'b1001: output_data <= {9'b000000000, input_data[15:9]};
    4'b1010: output_data <= {10'b0000000000, input_data[15:10]};
    4'b1011: output_data <= {11'b00000000000, input_data[15:11]};
    4'b1100: output_data <= {12'b000000000000, input_data[15:12]};
    4'b1101: output_data <= {13'b0000000000000, input_data[15:13]};
    4'b1110: output_data <= {14'b00000000000000, input_data[15:14]};
    4'b1111: output_data <= {15'b000000000000000, input_data[15]};
  endcase
end

endmodule

// ControlPath

module controlpath(
    input ack, Rst, Clk, operands_val, i_eq_16,
    output reg x_MUX_sel, y_MUX_sel, theta_MUX_sel, i_MUX_sel,
    output reg x_en, y_en, theta_en, i_en,
    output out_valid,
    output reg [1:0] state = 2'b00
);


    reg [1:0] state_next;


//parameters to define the states
```

```

    parameter idle = 2'b00;
    parameter busy = 2'b01;
    parameter done = 2'b10;


// Finding current state

always @(posedge Clk or posedge ack)
    begin
        if (Rst)
            state <= idle;
        else
            state <= state_next;
    end

//combinational logic to find next state

always @(*)
    case(state)
        idle : begin
            x_MUX_sel = 0;
            y_MUX_sel = 0;
            theta_MUX_sel = 0;
            i_MUX_sel = 0;
            x_en = 1;
            y_en = 1;
            theta_en = 1;
            i_en = 1;

            if (operands_val  == 1)  state_next = busy;
            else state_next = idle;

            end

        busy : begin
            x_MUX_sel = 1;
            y_MUX_sel = 1;
            theta_MUX_sel = 1;
            i_MUX_sel = 1;
            x_en = 1;
            y_en = 1;
            theta_en = 1;
            i_en = 1;

            if (i_eq_16  == 1)  state_next = done;
            else state_next = busy;

            end

        done : begin
            x_MUX_sel = 0;
            y_MUX_sel = 0;
            theta_MUX_sel = 0;
            x_en = 0;
            y_en = 0;
            i_en = 0;
            theta_en = 0;

            if (ack == 1)  state_next = idle;
            else state_next = done;

            end
        endcase

//Assigning Output

assign out_valid = (i_eq_16 == 1);

endmodule
```

4.2 Pipelined Rotation CORDIC

```
module pipelined_cordic(
    input Clk, Rst,
    input signed[15:0] x_in, y_in, theta_in,
    output reg signed[15:0] sine_theta,
    output reg signed[15:0] cosine_theta
);

reg signed [15:0] x[0:15], y[0:15], theta[0:15], x_out[0:15], y_out[0:15],
    theta_out[0:15];
wire signed [15:0] atan_out [0:15];
reg [3:0] stage; // Counter to track pipeline stages

assign atan_out[0] = 16'b0011001001000100;
assign atan_out[1] = 16'b0001110110101100;
assign atan_out[2] = 16'b0000111110101101;
assign atan_out[3] = 16'b0000011111110101;
assign atan_out[4] = 16'b0000001111111110;
assign atan_out[5] = 16'b0000001000000000;
assign atan_out[6] = 16'b0000000100000000;
assign atan_out[7] = 16'b0000000010000000;
assign atan_out[8] = 16'b0000000001000000;
assign atan_out[9] = 16'b0000000000100000;
assign atan_out[10] = 16'b0000000000010000;
assign atan_out[11] = 16'b0000000000001000;
assign atan_out[12] = 16'b0000000000000100;
assign atan_out[13] = 16'b0000000000000010;
assign atan_out[14] = 16'b0000000000000001;
assign atan_out[15] = 16'b0000000000000000;

// Generate block for pipeline stages
genvar i;
generate
    for (i = 0; i < 16; i = i + 1) begin : stages
        always @(posedge Clk or posedge Rst) begin
            if (Rst)
                begin
                    x[i] <= 0;
                    y[i] <= 0;
                    theta[i] <= 0;
                end
            else
                begin
                    if (i == 0) begin
                        x[i] <= x_in;
                        y[i] <= y_in;
                        theta[i] <= theta_in;
                    end else begin
                        x[i] <= x_out[i-1];
                        y[i] <= y_out[i-1];
                        theta[i] <= theta_out[i-1];
                    end
                end
        end

        always @(*) begin
            if (theta[i][15] == 1) begin
                x_out[i] = x[i] + (y[i] >> (i));
                y_out[i] = y[i] - (x[i] >> (i));
                theta_out[i] = theta[i] + atan_out[i];
            end else begin
                x_out[i] = x[i] - (y[i] >> (i));
                y_out[i] = y[i] + (x[i] >> (i));
                theta_out[i] = theta[i] - atan_out[i];
            end
        end
    end
endgenerate

always @(posedge Clk or posedge Rst) begin
    if (Rst) begin
        sine_theta <= 0;
        cosine_theta <= 0;
    end else begin
        sine_theta <= y_out[15]; // Output from the last stage
        cosine_theta <= x_out[15]; // Output from the last stage
    end
end

endmodule
```

4.3 Iterative Vectoring CORDIC

```

module CORDDICvectoring(
  input signed [15:0] x_in, y_in,
    input operands_val,
    input Clk, Rst, ack,
    output signed [15:0] x_out, y_out, theta_out,
    output out_valid
);

    wire x_MUX_sel, y_MUX_sel, theta_MUX_sel, i_MUX_sel, x_en, y_en, theta_en,
       i_en;
    wire i_eq_16;


    datapath d1 (
        .x_in(x_in),
        .y_in(y_in),
        .Clk(Clk),
        .Rst(Rst),
        .x_MUX_sel(x_MUX_sel),
        .y_MUX_sel(y_MUX_sel),
        .theta_MUX_sel(theta_MUX_sel),
        .i_MUX_sel(i_MUX_sel),
        .x_en(x_en),
        .y_en(y_en),
        .theta_en(theta_en),
        .i_en(i_en),
        .x_out(x_out),
        .y_out(y_out),
        .theta_out(theta_out),
        .i_eq_16(i_eq_16)
    );
```

```verilog

    controlpath c1 (
        .ack(ack),
        .Rst(Rst),
        .Clk(Clk),
        .operands_val(operands_val),
        .i_eq_16(i_eq_16),
        .x_MUX_sel(x_MUX_sel),
        .y_MUX_sel(y_MUX_sel),
        .theta_MUX_sel(theta_MUX_sel),
        .i_MUX_sel(i_MUX_sel),
        .x_en(x_en),
        .y_en(y_en),
        .theta_en(theta_en),
        .i_en(i_en),
        .out_valid(out_valid)
    );

endmodule



//Datapath

module datapath(
  input signed [15:0] x_in, y_in,
    input Clk, Rst,
    input x_MUX_sel, y_MUX_sel, theta_MUX_sel, i_MUX_sel, x_en, y_en, theta_en, i_en,
    output reg signed [15:0] x_out, y_out, theta_out,
    output i_eq_16
    );

    wire signed [15:0] x_MUX_out, y_MUX_out, theta_MUX_out, x_i, y_i;
    reg signed [15:0] x, y, theta, atan_out;
    reg [3:0] i;

    // Look Up Table

    always @(*)
        begin
          case(i)
                4'b0000: atan_out <= 16'b0011001001000100;
                    4'b0001: atan_out <= 16'b0001110110101100;
                    4'b0010: atan_out <= 16'b0000111110101101;
                    4'b0011: atan_out <= 16'b0000011111110101;
                    4'b0100: atan_out <= 16'b0000001111111110;
                    4'b0101: atan_out <= 16'b0000001000000000;
                    4'b0110: atan_out <= 16'b0000000100000000;
                    4'b0111: atan_out <= 16'b0000000010000000;
                    4'b1000: atan_out <= 16'b0000000001000000;
                    4'b1001: atan_out <= 16'b0000000000100000;
                    4'b1010: atan_out <= 16'b0000000000010000;
                    4'b1011: atan_out <= 16'b0000000000001000;
                    4'b1100: atan_out <= 16'b0000000000000100;
                    4'b1101: atan_out <= 16'b0000000000000010;
                    4'b1110: atan_out <= 16'b0000000000000001;
                    4'b1111: atan_out <= 16'b0000000000000000;
                endcase
        end
    always @(posedge Clk)
        begin
          if (Rst)
          begin
          x_out <= 0;
          y_out <= 0;
          theta_out <= 0;
          end
          else
          begin
          if (x_en) x_out <= x_MUX_out;
          if (y_en) y_out <= y_MUX_out;
          if (theta_en) theta_out <= theta_MUX_out;
          end
          if (i_en) i <= i + 1; else i <= 0;
        end


    always @(*)
      if (y_out[15] == 0)
        begin
            x = x_out + y_i;
            y = y_out - x_i;
            theta = theta_out + atan_out;
            end
        else
        begin
            x = x_out - y_i;
            y = y_out + x_i;
            theta = theta_out - atan_out;
            end


    assign x_MUX_out = x_MUX_sel ? x : x_in;
    assign y_MUX_out = y_MUX_sel ? y : y_in;
    assign theta_MUX_out = theta_MUX_sel ? theta : 0;
    assign i_eq_16 = (i == 15);
    assign x_i = x_out >>> i;
    assign y_i = y_out >>> i;

endmodule

// ControlPath

module controlpath(
    input ack, Rst, Clk, operands_val, i_eq_16,
    output reg x_MUX_sel, y_MUX_sel, theta_MUX_sel, i_MUX_sel,
    output reg x_en, y_en, theta_en, i_en,
    output out_valid
);

  reg [1:0] state;
  reg [1:0] state_next;


//parameters to define the states

    parameter idle = 2'b00;
    parameter busy = 2'b01;
    parameter done = 2'b10;


// Finding current state

always @(posedge Clk)
    begin
```

```verilog
        if (Rst)
            state <= idle;
        else
            state <= state_next;
    end

//combinational logic to find next state

always @(*)
    case(state)
        idle : begin
            x_MUX_sel = 0;
            y_MUX_sel = 0;
            theta_MUX_sel = 0;
            i_MUX_sel = 0;
            x_en = 1;
            y_en = 1;
            theta_en = 1;
            i_en = 0;

            if (operands_val  == 1)  state_next = busy;
            else state_next = idle;

            end

        busy : begin
            x_MUX_sel = 1;
            y_MUX_sel = 1;
            theta_MUX_sel = 1;
            i_MUX_sel = 1;
            x_en = 1;
            y_en = 1;
            theta_en = 1;
            i_en = 1;

            if (i_eq_16  == 1)  state_next = done;
            else state_next = busy;

            end

        done : begin
            x_MUX_sel = 0;
            y_MUX_sel = 0;
            theta_MUX_sel = 0;
            x_en = 0;
            y_en = 0;
            i_en = 0;
            theta_en = 0;
            i_MUX_sel = 0;

            if (ack == 1)  state_next = idle;
            else state_next = done;

            end
        endcase

//Assigning Output

  assign out_valid = (state == done);

endmodule
```

4.4 Pipelined Vectoring CORDIC

```verilog
module pipelined_cordic(
    input Clk, Rst,
    input signed[15:0] x_in, y_in,
    output reg signed[15:0] magnitude,
    output reg signed[15:0] angle
);

reg signed [15:0] x[0:15], y[0:15], theta[0:15], x_out[0:15], y_out[0:15],
    theta_out[0:15];
wire signed [15:0] atan_out [0:15];
reg [3:0] stage; // Counter to track pipeline stages

assign atan_out[0] = 16'b0011001001000100;
assign atan_out[1] = 16'b0001110110101100;
assign atan_out[2] = 16'b0000111110101101;
assign atan_out[3] = 16'b0000011111110101;
assign atan_out[4] = 16'b0000001111111110;
assign atan_out[5] = 16'b0000001000000000;
assign atan_out[6] = 16'b0000000100000000;
assign atan_out[7] = 16'b0000000010000000;
assign atan_out[8] = 16'b0000000001000000;
assign atan_out[9] = 16'b0000000000100000;
assign atan_out[10] = 16'b0000000000010000;
assign atan_out[11] = 16'b0000000000001000;
assign atan_out[12] = 16'b0000000000000100;
assign atan_out[13] = 16'b0000000000000010;
assign atan_out[14] = 16'b0000000000000001;
assign atan_out[15] = 16'b0000000000000000;

// Generate block for pipeline stages
genvar i;
generate
    for (i = 0; i < 16; i = i + 1) begin : stages
        always @(posedge Clk or posedge Rst) begin
            if (Rst)
                begin
                    x[i] <= 0;
                    y[i] <= 0;
                    theta[i] <= 0;
                end
            else
                begin
                    if (i == 0) begin
                        x[i] <= x_in;
                        y[i] <= y_in;
                        theta[i] <= 16'd0;
                    end else begin
                        x[i] <= x_out[i-1];
                        y[i] <= y_out[i-1];
                        theta[i] <= theta_out[i-1];
                    end
                end
        end

        always @(*) begin
            if (y[i][15] == 0) begin
                x_out[i] = x[i] + (y[i] >>> (i));
                y_out[i] = y[i] - (x[i] >>> (i));
                theta_out[i] = theta[i] + atan_out[i];
            end else begin
                x_out[i] = x[i] - (y[i] >>> (i));
                y_out[i] = y[i] + (x[i] >>> (i));
                theta_out[i] = theta[i] - atan_out[i];
            end
        end
    end
endgenerate

always @(posedge Clk or posedge Rst) begin
    if (Rst) begin
        angle <= 0;
        magnitude <= 0;
    end else begin
        angle <= theta_out[15]; // Output from the last stage
        magnitude <= x_out[15]; // Output from the last stage
    end
end

endmodule
```

5 Experimental Results

In this section, we delve into a thorough evaluation of our design's performance by constructing a comprehensive test bench in Verilog. The primary aim of this test bench is to provide a varied set of inputs, known as stimuli, to our design. Subsequently, we analyze the resulting outputs meticulously to assess the effectiveness and reliability of our system.

Employing a test bench serves as a crucial step in validating our design's functionality and ensuring its alignment with the specified requirements. Through careful planning and thoughtful selection of stimuli, we conduct a comprehensive testing process to validate the integrity and robustness of our system's operation.

Here the test bench in the case of rotation CORDIC iterates through a list of ten angles from an external txt file named angle_hex.txt:

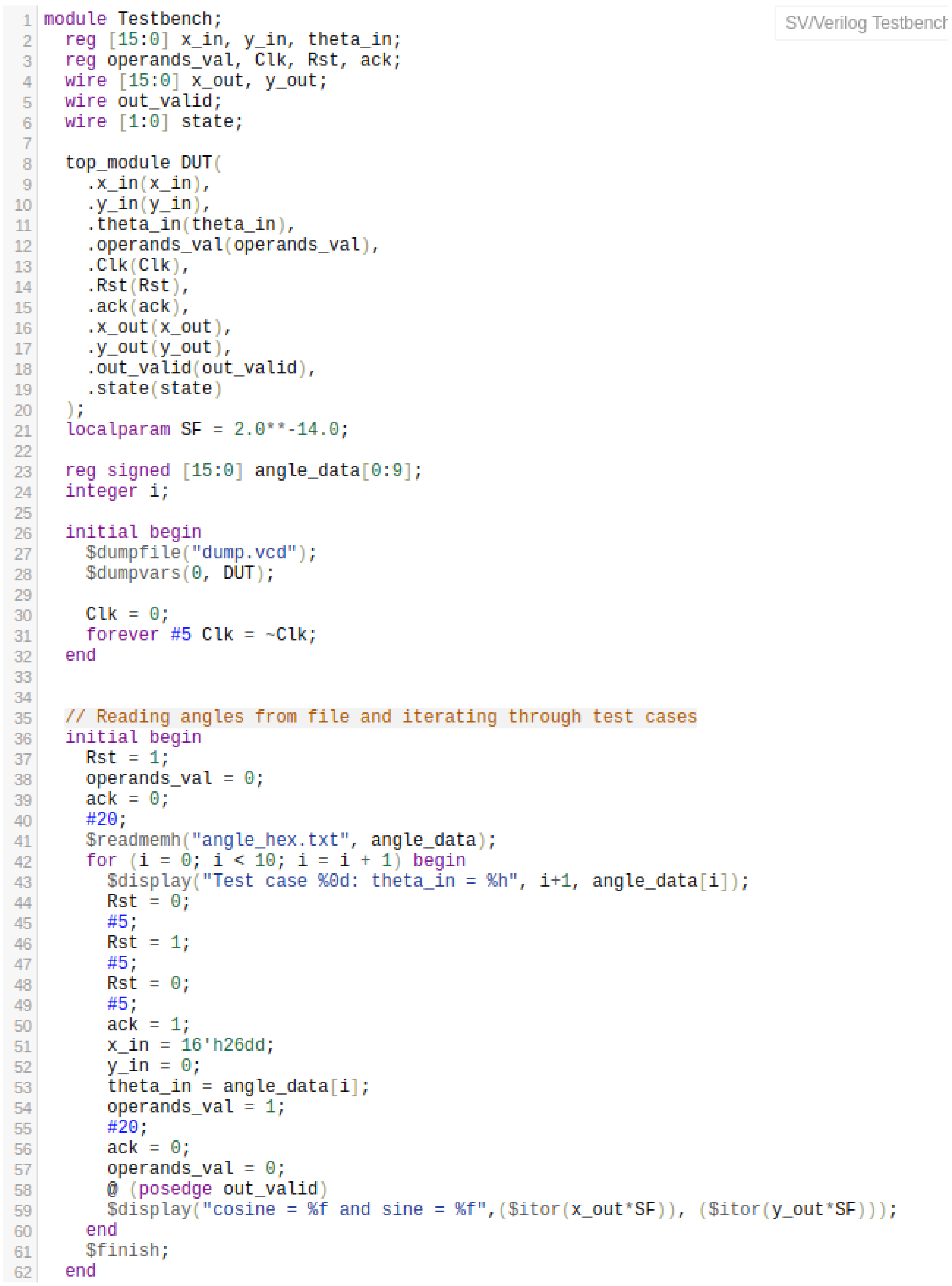
```
0b2c 1657 2183 2cae 3244 37da 4305 4e31 595c 6488
```

In the case of vectoring CORDIC, it iterates through a list of ten Cartesian coordinates from an external file named coordinate_hex.txt:

```
3F06 0B1F 3C21 15E5 3767 1FFF 3107 2923 2D42 2D41 2923 3107 1FFF 376F 15E5 3C21 0B1F
3F06 003E 2BFC
```

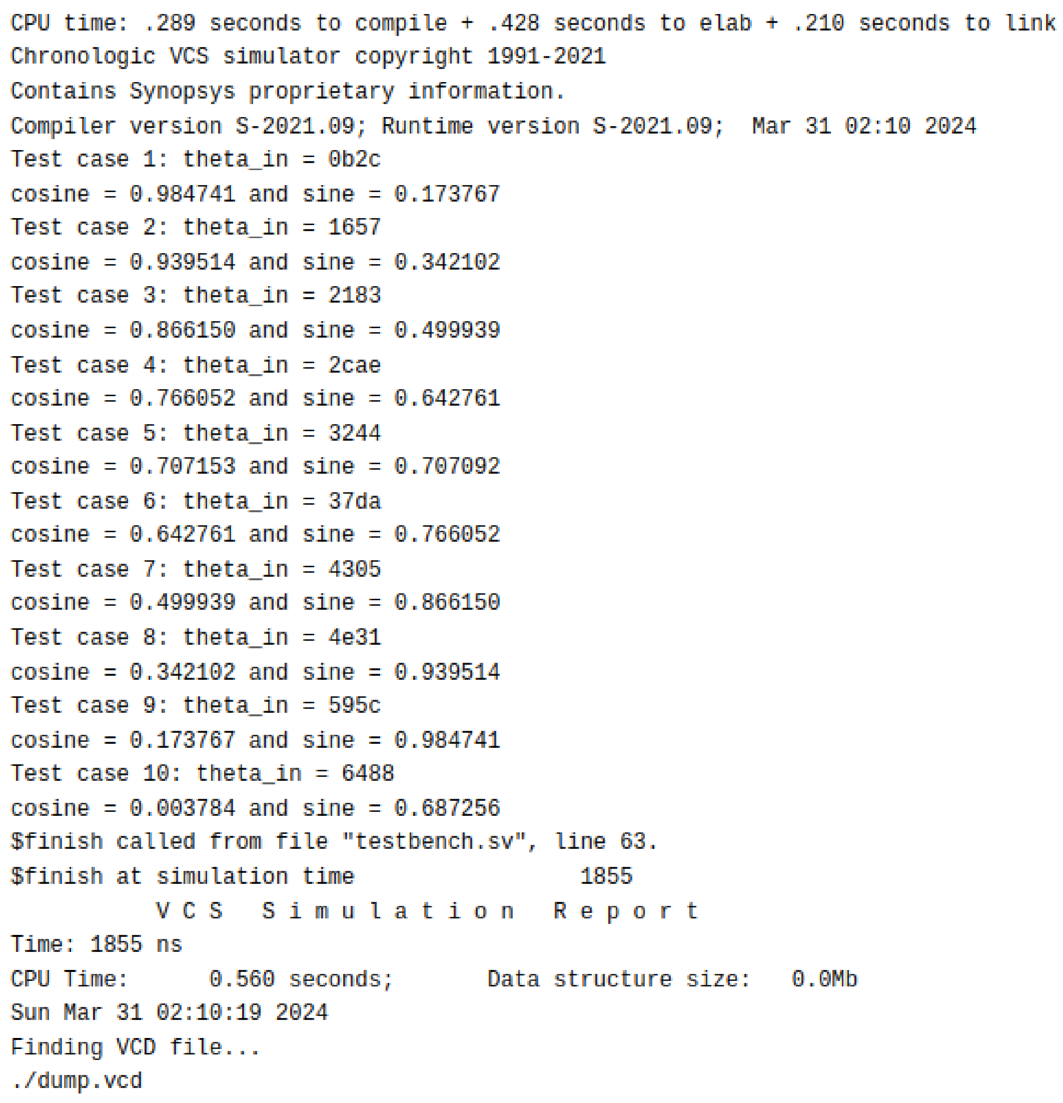
Each odd position corresponds to the abscissa and each even position corresponds to the ordinate

5.1 Iterative Rotation CORDIC



```verilog
module Testbench;
  reg [15:0] x_in, y_in, theta_in;
  reg operands_val, Clk, Rst, ack;
  wire [15:0] x_out, y_out;
  wire out_valid;
  wire [1:0] state;

  top_module DUT(
    .x_in(x_in),
    .y_in(y_in),
    .theta_in(theta_in),
    .operands_val(operands_val),
    .Clk(Clk),
    .Rst(Rst),
    .ack(ack),
    .x_out(x_out),
    .y_out(y_out),
    .out_valid(out_valid),
    .state(state)
  );
  localparam SF = 2.0**-14.0;

  reg signed [15:0] angle_data[0:9];
  integer i;

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, DUT);

    Clk = 0;
    forever #5 Clk = ~Clk;
  end


  // Reading angles from file and iterating through test cases
  initial begin
    Rst = 1;
    operands_val = 0;
    ack = 0;
    #20;
    $readmemh("angle_hex.txt", angle_data);
    for (i = 0; i < 10; i = i + 1) begin
      $display("Test case %0d: theta_in = %h", i+1, angle_data[i]);
      Rst = 0;
      #5;
      Rst = 1;
      #5;
      Rst = 0;
      #5;
      ack = 1;
      x_in = 16'h26dd;
      y_in = 0;
      theta_in = angle_data[i];
      operands_val = 1;
      #20;
      ack = 0;
      operands_val = 0;
      @ (posedge out_valid)
      $display("cosine = %f and sine = %f",($itor(x_out*SF)), ($itor(y_out*SF)));
    end
    $finish;
  end
```

Figure 7: Testbench for iterative rotation CORDIC



```
CPU time: .289 seconds to compile + .428 seconds to elab + .210 seconds to link
Chronologic VCS simulator copyright 1991-2021
Contains Synopsys proprietary information.
Compiler version S-2021.09; Runtime version S-2021.09;  Mar 31 02:10 2024
Test case 1: theta_in = 0b2c
cosine = 0.984741 and sine = 0.173767
Test case 2: theta_in = 1657
cosine = 0.939514 and sine = 0.342102
Test case 3: theta_in = 2183
cosine = 0.866150 and sine = 0.499939
Test case 4: theta_in = 2cae
cosine = 0.766052 and sine = 0.642761
Test case 5: theta_in = 3244
cosine = 0.707153 and sine = 0.707092
Test case 6: theta_in = 37da
cosine = 0.642761 and sine = 0.766052
Test case 7: theta_in = 4305
cosine = 0.499939 and sine = 0.866150
Test case 8: theta_in = 4e31
cosine = 0.342102 and sine = 0.939514
Test case 9: theta_in = 595c
cosine = 0.173767 and sine = 0.984741
Test case 10: theta_in = 6488
cosine = 0.003784 and sine = 0.687256
$finish called from file "testbench.sv", line 63.
$finish at simulation time                 1855
           V C S   S i m u l a t i o n   R e p o r t
Time: 1855 ns
CPU Time:      0.560 seconds;       Data structure size:   0.0Mb
Sun Mar 31 02:10:19 2024
Finding VCD file...
./dump.vcd
```

Figure 8: Results for iterative rotation CORDIC

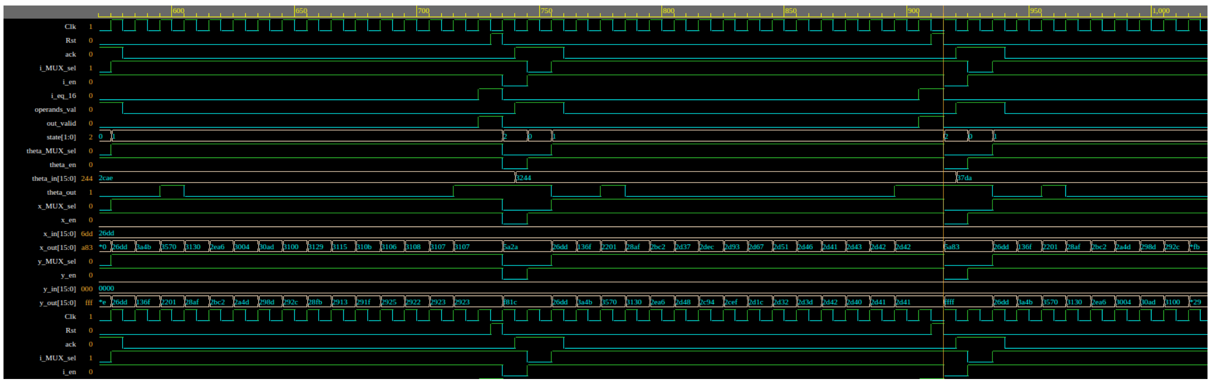
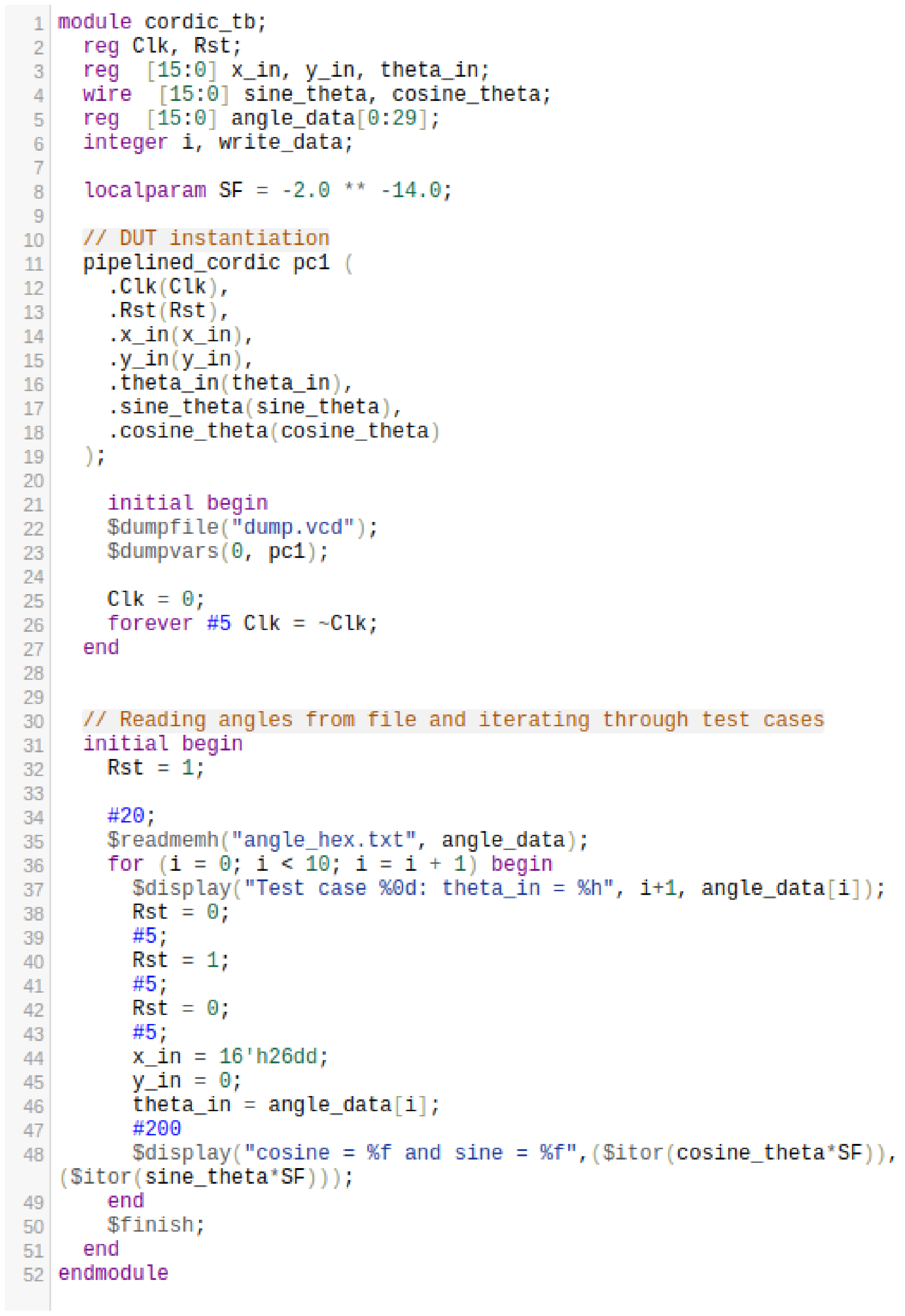


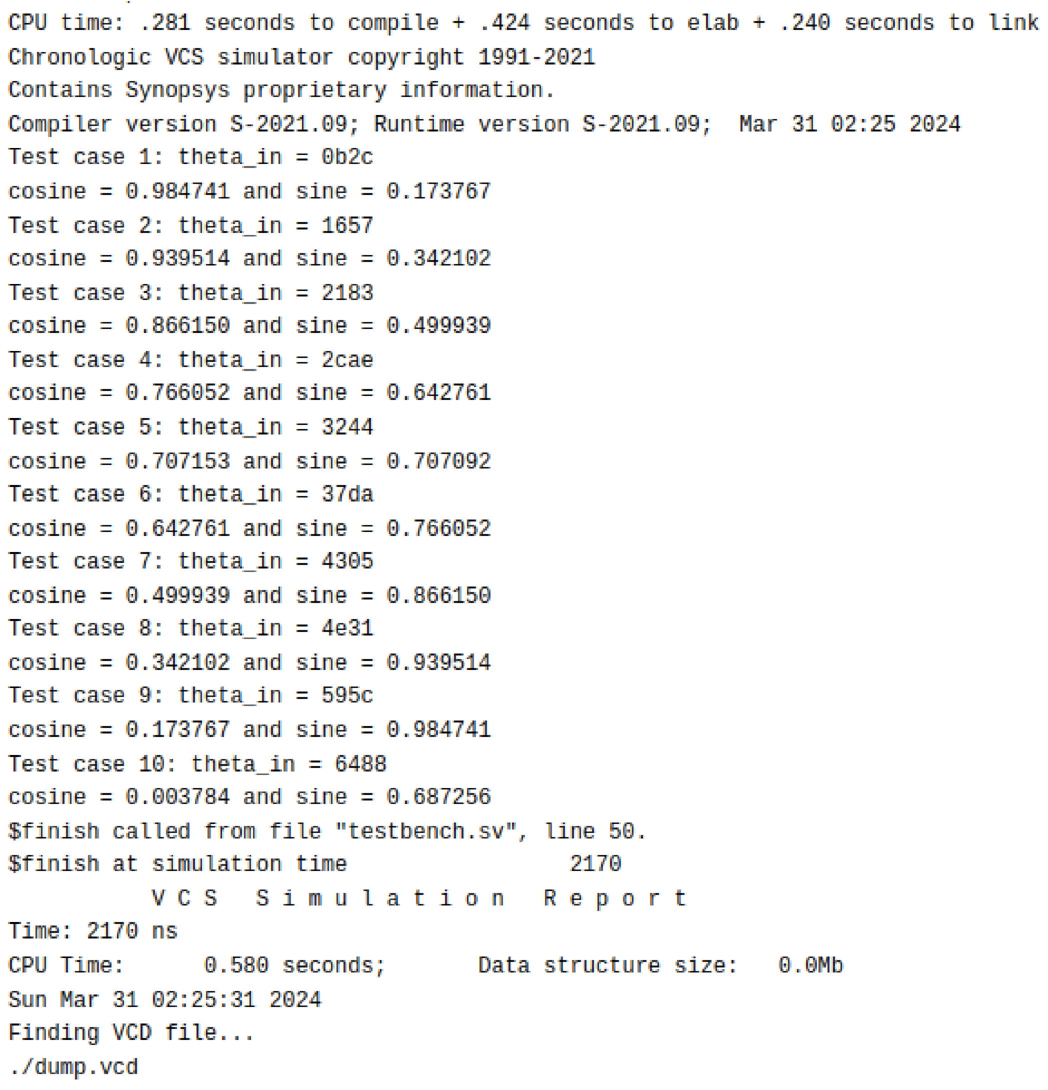
Figure 9: Waveform for iterative rotation CORDIC

5.2 Pipelined Rotation CORDIC



```verilog
module cordic_tb;
  reg Clk, Rst;
  reg  [15:0] x_in, y_in, theta_in;
  wire  [15:0] sine_theta, cosine_theta;
  reg  [15:0] angle_data[0:29];
  integer i, write_data;

  localparam SF = -2.0 ** -14.0;

  // DUT instantiation
  pipelined_cordic pc1 (
    .Clk(Clk),
    .Rst(Rst),
    .x_in(x_in),
    .y_in(y_in),
    .theta_in(theta_in),
    .sine_theta(sine_theta),
    .cosine_theta(cosine_theta)
  );

    initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, pc1);

    Clk = 0;
    forever #5 Clk = ~Clk;
  end


  // Reading angles from file and iterating through test cases
  initial begin
    Rst = 1;

    #20;
    $readmemh("angle_hex.txt", angle_data);
    for (i = 0; i < 10; i = i + 1) begin
      $display("Test case %0d: theta_in = %h", i+1, angle_data[i]);
      Rst = 0;
      #5;
      Rst = 1;
      #5;
      Rst = 0;
      #5;
      x_in = 16'h26dd;
      y_in = 0;
      theta_in = angle_data[i];
      #200
      $display("cosine = %f and sine = %f",($itor(cosine_theta*SF)),
($itor(sine_theta*SF)));
    end
    $finish;
  end
endmodule
```

Figure 10: Testbench for pipelined rotation CORDIC



```
CPU time: .281 seconds to compile + .424 seconds to elab + .240 seconds to link
Chronologic VCS simulator copyright 1991-2021
Contains Synopsys proprietary information.
Compiler version S-2021.09; Runtime version S-2021.09;  Mar 31 02:25 2024
Test case 1: theta_in = 0b2c
cosine = 0.984741 and sine = 0.173767
Test case 2: theta_in = 1657
cosine = 0.939514 and sine = 0.342102
Test case 3: theta_in = 2183
cosine = 0.866150 and sine = 0.499939
Test case 4: theta_in = 2cae
cosine = 0.766052 and sine = 0.642761
Test case 5: theta_in = 3244
cosine = 0.707153 and sine = 0.707092
Test case 6: theta_in = 37da
cosine = 0.642761 and sine = 0.766052
Test case 7: theta_in = 4305
cosine = 0.499939 and sine = 0.866150
Test case 8: theta_in = 4e31
cosine = 0.342102 and sine = 0.939514
Test case 9: theta_in = 595c
cosine = 0.173767 and sine = 0.984741
Test case 10: theta_in = 6488
cosine = 0.003784 and sine = 0.687256
$finish called from file "testbench.sv", line 50.
$finish at simulation time                 2170
           V C S   S i m u l a t i o n   R e p o r t
Time: 2170 ns
CPU Time:      0.580 seconds;       Data structure size:   0.0Mb
Sun Mar 31 02:25:31 2024
Finding VCD file...
./dump.vcd
```

Figure 11: Results for pipelined rotation CORDIC

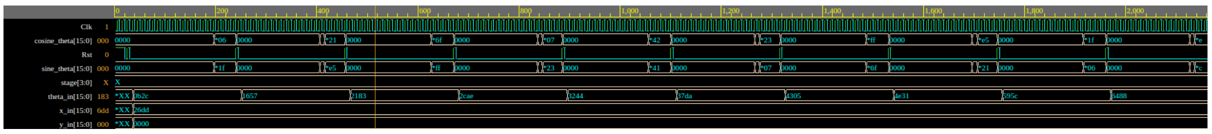
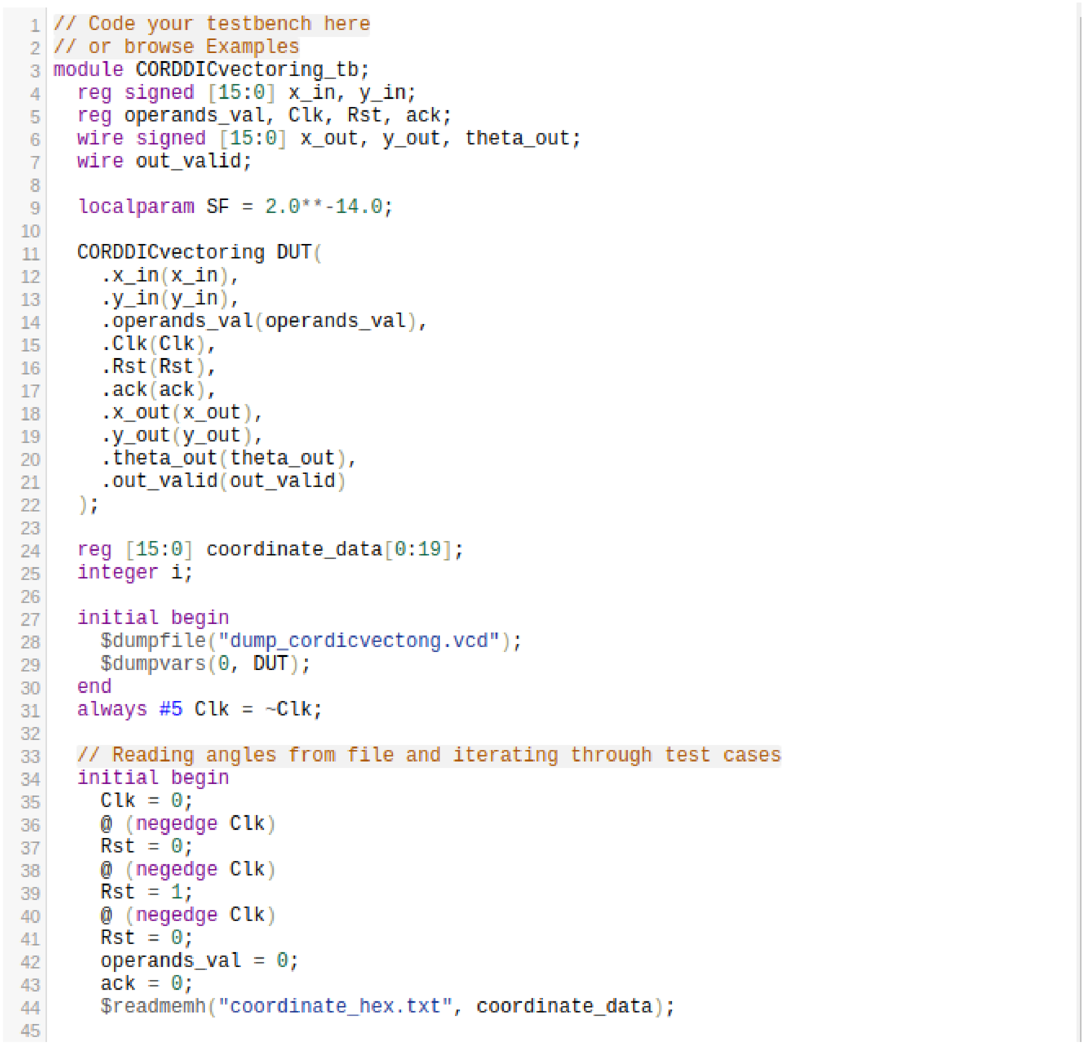
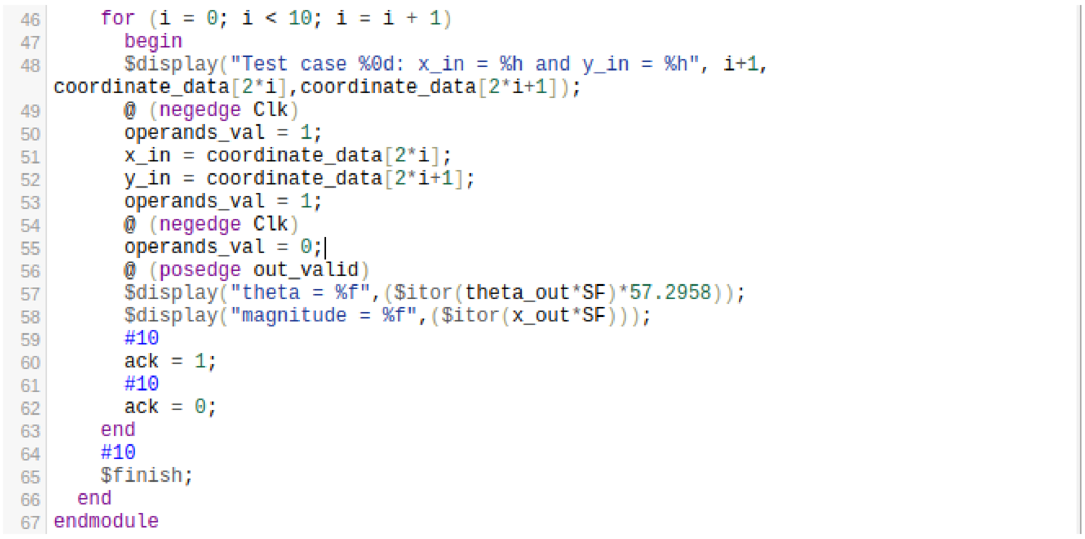


Figure 12: Waveform for pipelined rotation CORDIC

5.3 Iterative Vectoring CORDIC



```verilog
// Code your testbench here
// or browse Examples
module CORDDICvectoring_tb;
  reg signed [15:0] x_in, y_in;
  reg operands_val, Clk, Rst, ack;
  wire signed [15:0] x_out, y_out, theta_out;
  wire out_valid;

  localparam SF = 2.0**-14.0;

  CORDDICvectoring DUT(
    .x_in(x_in),
    .y_in(y_in),
    .operands_val(operands_val),
    .Clk(Clk),
    .Rst(Rst),
    .ack(ack),
    .x_out(x_out),
    .y_out(y_out),
    .theta_out(theta_out),
    .out_valid(out_valid)
  );

  reg [15:0] coordinate_data[0:19];
  integer i;

  initial begin
    $dumpfile("dump_cordicvectong.vcd");
    $dumpvars(0, DUT);
  end
  always #5 Clk = ~Clk;

  // Reading angles from file and iterating through test cases
  initial begin
    Clk = 0;
    @ (negedge Clk)
    Rst = 0;
    @ (negedge Clk)
    Rst = 1;
    @ (negedge Clk)
    Rst = 0;
    operands_val = 0;
    ack = 0;
    $readmemh("coordinate_hex.txt", coordinate_data);

```



```verilog
    for (i = 0; i < 10; i = i + 1)
      begin
      $display("Test case %0d: x_in = %h and y_in = %h", i+1,
coordinate_data[2*i],coordinate_data[2*i+1]);
      @ (negedge Clk)
      operands_val = 1;
      x_in = coordinate_data[2*i];
      y_in = coordinate_data[2*i+1];
      operands_val = 1;
      @ (negedge Clk)
      operands_val = 0;
      @ (posedge out_valid)
      $display("theta = %f",($itor(theta_out*SF)*57.2958));
      $display("magnitude = %f",($itor(x_out*SF)));
      #10
      ack = 1;
      #10
      ack = 0;
    end
    #10
    $finish;
  end
endmodule
```

Figure 13: Testbench for iterative vectoring CORDIC

```
CPU time: .301 seconds to compile + .445 seconds to elab + .255 seconds to link
Chronologic VCS simulator copyright 1991-2021
Contains Synopsys proprietary information.
Compiler version S-2021.09; Runtime version S-2021.09;  Mar 31 02:37 2024
Test case 1: x_in = 3f06 and y_in = 0b1f
theta = 10.012077
magnitude = 1.646973
Test case 2: x_in = 3c21 and y_in = 15e5
theta = 20.006669
magnitude = 1.646912
Test case 3: x_in = 3767 and y_in = 1fff
theta = 30.008256
magnitude = 1.646240
Test case 4: x_in = 3107 and y_in = 2923
theta = 39.995854
magnitude = 1.646973
Test case 5: x_in = 2d42 and y_in = 2d41
theta = 44.989653
magnitude = 1.647034
Test case 6: x_in = 2923 and y_in = 3107
theta = 50.011428
magnitude = 1.647034
Test case 7: x_in = 1fff and y_in = 376f
theta = 60.006020
magnitude = 1.647095
Test case 8: x_in = 15e5 and y_in = 3c21
theta = 70.000612
magnitude = 1.646545
Test case 9: x_in = 0b1f and y_in = 3f06
theta = 79.988210
magnitude = 1.646851
Test case 10: x_in = 003e and y_in = 2bfc
theta = 89.682055
magnitude = 1.132080
$finish called from file "testbench.sv", line 65.
$finish at simulation time                 1945
           V C S   S i m u l a t i o n   R e p o r t
Time: 1945 ns
CPU Time:      0.660 seconds;       Data structure size:   0.0Mb
Sun Mar 31 02:37:07 2024
Finding VCD file...
./dump_cordicvectong.vcd
```

Figure 14: Result for iterative vectoring CORDIC

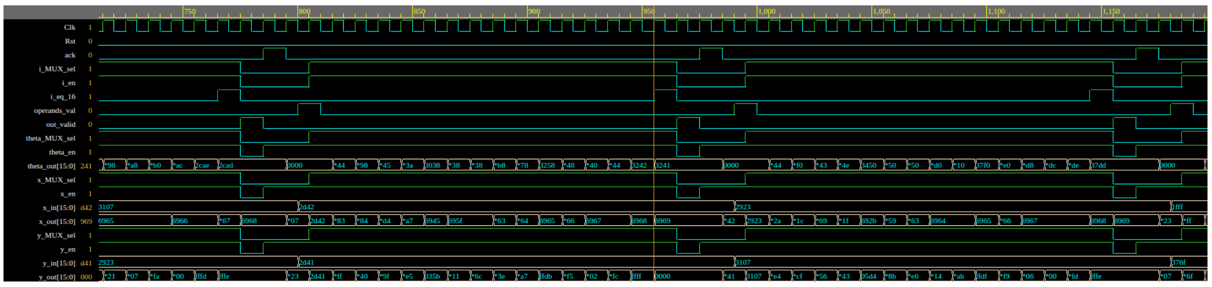
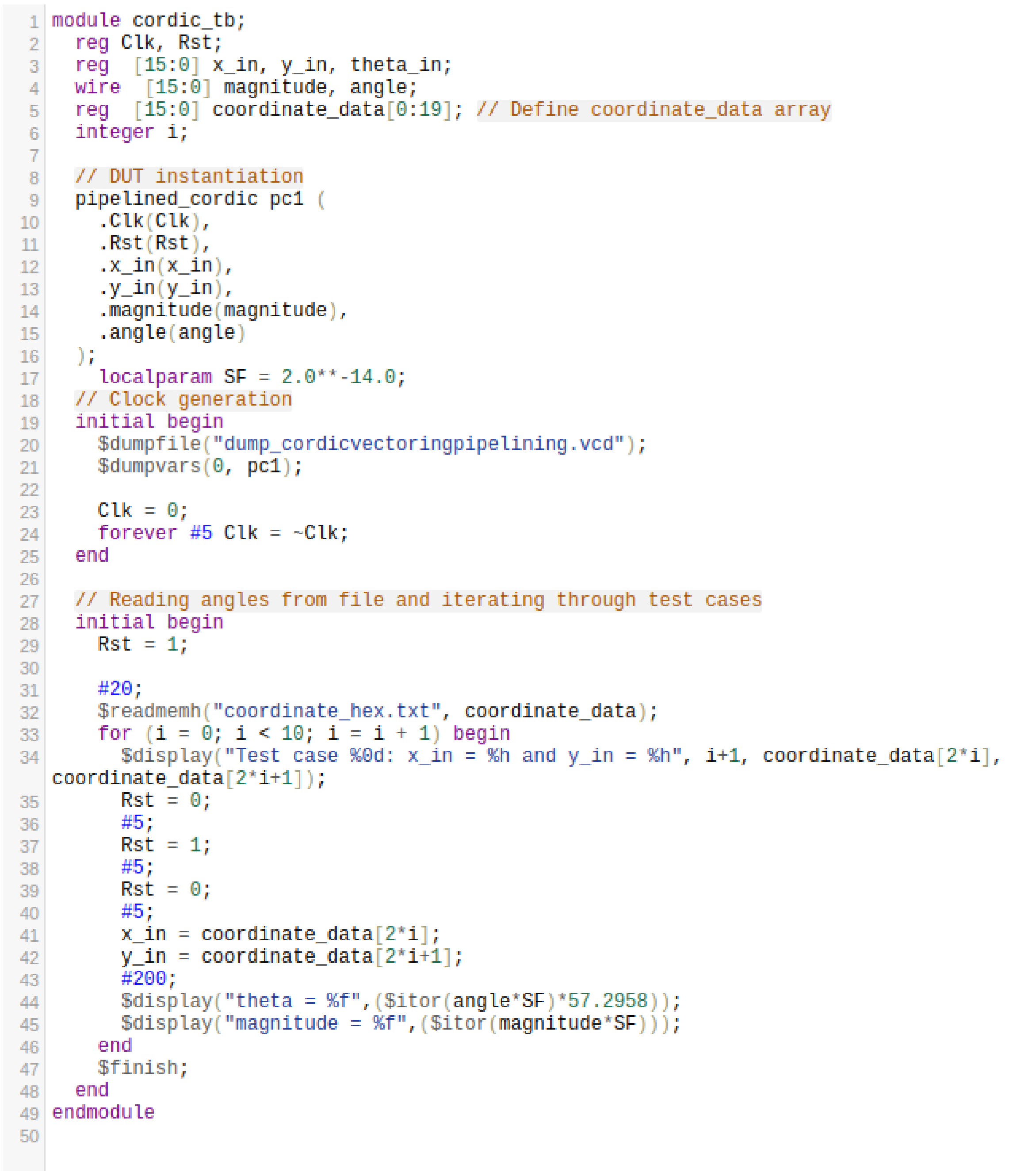


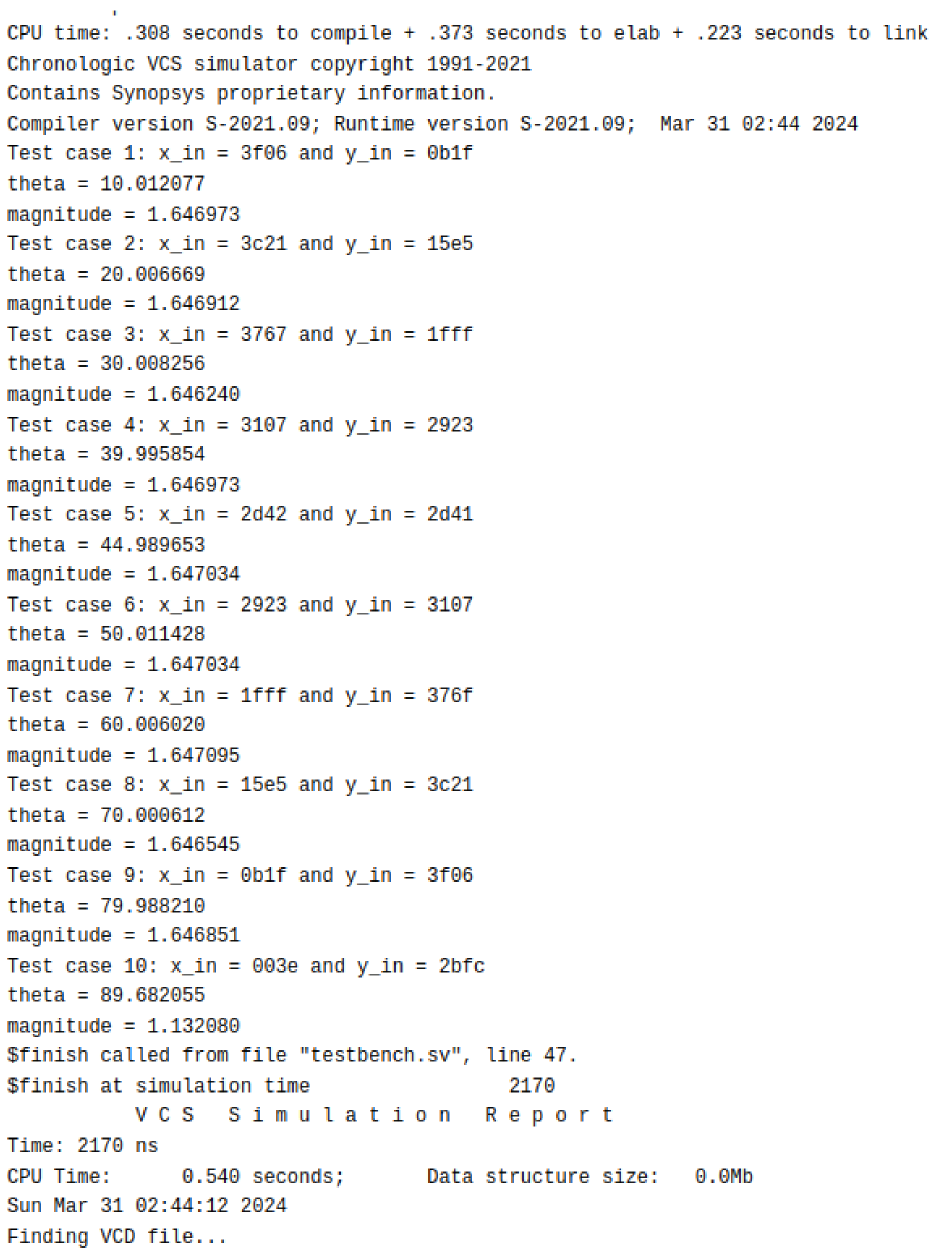
Figure 15: Waveform for iterative vectoring CORDIC

5.4 Pipelined Vectoring CORDIC



```verilog
module cordic_tb;
  reg Clk, Rst;
  reg  [15:0] x_in, y_in, theta_in;
  wire  [15:0] magnitude, angle;
  reg  [15:0] coordinate_data[0:19]; // Define coordinate_data array
  integer i;

  // DUT instantiation
  pipelined_cordic pc1 (
    .Clk(Clk),
    .Rst(Rst),
    .x_in(x_in),
    .y_in(y_in),
    .magnitude(magnitude),
    .angle(angle)
  );
    localparam SF = 2.0**-14.0;
  // Clock generation
  initial begin
    $dumpfile("dump_cordicvectoringpipelining.vcd");
    $dumpvars(0, pc1);

    Clk = 0;
    forever #5 Clk = ~Clk;
  end

  // Reading angles from file and iterating through test cases
  initial begin
    Rst = 1;

    #20;
    $readmemh("coordinate_hex.txt", coordinate_data);
    for (i = 0; i < 10; i = i + 1) begin
      $display("Test case %0d: x_in = %h and y_in = %h", i+1, coordinate_data[2*i], coordinate_data[2*i+1]);
      Rst = 0;
      #5;
      Rst = 1;
      #5;
      Rst = 0;
      #5;
      x_in = coordinate_data[2*i];
      y_in = coordinate_data[2*i+1];
      #200;
      $display("theta = %f",($itor(angle*SF)*57.2958));
      $display("magnitude = %f",($itor(magnitude*SF)));
    end
    $finish;
  end
endmodule
```

Figure 16: Testbench for pipelined vectoring CORDIC



```
CPU time: .308 seconds to compile + .373 seconds to elab + .223 seconds to link
Chronologic VCS simulator copyright 1991-2021
Contains Synopsys proprietary information.
Compiler version S-2021.09; Runtime version S-2021.09;  Mar 31 02:44 2024
Test case 1: x_in = 3f06 and y_in = 0b1f
theta = 10.012077
magnitude = 1.646973
Test case 2: x_in = 3c21 and y_in = 15e5
theta = 20.006669
magnitude = 1.646912
Test case 3: x_in = 3767 and y_in = 1fff
theta = 30.008256
magnitude = 1.646240
Test case 4: x_in = 3107 and y_in = 2923
theta = 39.995854
magnitude = 1.646973
Test case 5: x_in = 2d42 and y_in = 2d41
theta = 44.989653
magnitude = 1.647034
Test case 6: x_in = 2923 and y_in = 3107
theta = 50.011428
magnitude = 1.647034
Test case 7: x_in = 1fff and y_in = 376f
theta = 60.006020
magnitude = 1.647095
Test case 8: x_in = 15e5 and y_in = 3c21
theta = 70.000612
magnitude = 1.646545
Test case 9: x_in = 0b1f and y_in = 3f06
theta = 79.988210
magnitude = 1.646851
Test case 10: x_in = 003e and y_in = 2bfc
theta = 89.682055
magnitude = 1.132080
$finish called from file "testbench.sv", line 47.
$finish at simulation time                 2170
           V C S   S i m u l a t i o n   R e p o r t
Time: 2170 ns
CPU Time:      0.540 seconds;       Data structure size:   0.0Mb
Sun Mar 31 02:44:12 2024
Finding VCD file...
```

Figure 17: Result for pipelined vectoring CORDIC

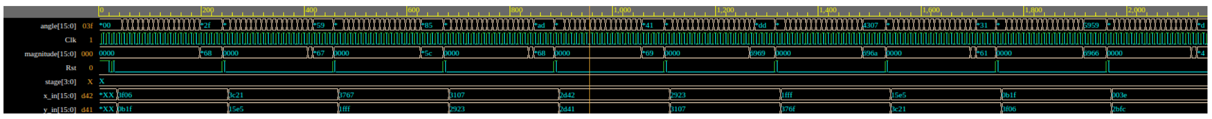


Figure 18: Waveform for pipelined vectoring CORDIC

6 Conclusion

In this experiment we successfully implemented rotation and vectoring CORDIC in both iterative and pipelined fashion. . The pipelined CORDIC have several advantages over iterative CORDIC such as: High throughput, Low latency, Reduced area and Increased accuracy. The pipelined CORDIC doesn't rquire a control path like the iterative CORDIC