Vincent Han

Lab Report 2

EE 4301

July 29th, 2023

1. Introduction

The Cordic computer is an algorithm used in digital signal processing and trigonometric calculations, based on rotations and vectoring. It efficiently computes functions like sine and cosine by iteratively rotating an initial vector towards the target vector. Bit-serial implementations process data one bit at a time, requiring fewer hardware resources and offering simpler debugging. However, they have slower execution speeds. Bit-parallel implementations process multiple bits simultaneously, providing faster computation for high-throughput applications but at the cost of increased hardware complexity. From an FPGA perspective, bit-serial designs are suitable for resource-constrained FPGAs and easier verification, while bit-parallel designs excel in high-throughput scenarios but require careful resource management. The choice between the implementations depends on the specific application and available resources.

2. Bit Serial Implementation

The bit-serial implementation architecture is a digital hardware approach where data is processed one bit at a time. Unlike bit-parallel designs that handle multiple bits simultaneously, bit-serial implementations process data in a sequential manner. This means that each bit is processed one after the other. As a result, the hardware structure is simpler and more compact compared to bit-parallel designs. It allows for the use of narrower data paths and reduces the number of required logic elements, making it suitable for resource-constrained scenarios or applications where minimizing hardware complexity is important. However, the trade-off for this simplicity is a decrease in computation speed compared to bit-parallel implementations. Bit-serial architectures find their usefulness in situations where computational efficiency takes a back seat to conserving hardware resources or when the processing demands are not time-critical.

3. Discussion of Results

Cordic computers benefit from precomputed tables and simulation waveforms, improving their efficiency and validating the design. Utilization is a key consideration, as it determines how effectively hardware resources are utilized, directly impacting the overall performance of the

Cordic computer. The timing report plays a vital role in ensuring correct functionality by analyzing timing constraints, and engineers can use this information to perform optimizations and meet required timing specifications. On the other hand, the power report is essential for optimizing power consumption, which holds significant importance in various applications, especially in portable devices where battery life is crucial. By analyzing the power report, engineers can implement techniques such as clock gating or power-aware optimizations to reduce power consumption without compromising the Cordic computer's functionality. These aspects collectively contribute to the successful implementation and performance of Cordic computers in various hardware systems.

4. **Summary and Conclusion**

Cordic computers are efficient algorithms used for digital signal processing and trigonometric calculations through rotations and vectoring. They compute functions like sine and cosine by iteratively rotating an initial vector toward the target vector. Bit-serial implementations process data one bit at a time, making them suitable for resource-constrained scenarios, but they have slower execution speeds compared to bit-parallel designs that handle multiple bits simultaneously. From an FPGA perspective, bit-serial designs are easier to verify and suitable for resource-constrained FPGAs, while bit-parallel designs excel in high-throughput applications but require careful resource management. Precomputed tables and simulation waveforms enhance Cordic efficiency and validate designs. Utilization affects performance by determining hardware resource efficiency, while the timing report ensures correct functionality by analyzing timing constraints. Engineers can optimize power consumption using the power report, especially important for portable devices with limited battery life. These aspects collectively contribute to successful Cordic computer implementation and performance in various hardware systems.

5. **Source Code**

```
`timescale 1ns / 1ps

module cordic(
    input clk,
    input [1:0] btn,
    input [15:0] sw,
    output [6:0] disp,
    output [3:0] anode,
    output reg [12:0] LED
```

```
    );

    parameter width = 16;

    wire signed [width-1:0] x_start,y_start;
    reg signed [15:0] angle;

    wire signed [15:0] ROM [0:12];

    reg [3:0] dig1;
    reg [3:0] dig2;
    reg [3:0] dig3;
    reg [3:0] dig4;

    disp D(clk, dig1, dig2, dig3, dig4, disp, anode);

    assign x_start = 16'b0010011011011101;
    assign y_start = 0;

    always @(posedge clk) begin
        if(btn == 2'b10)
            angle <= sw;
    end

    assign ROM[0] = 16'b0011001001000100;
    assign ROM[1] = 16'b0001110110101100;
    assign ROM[2] = 16'b0000111110101110;
    assign ROM[3] = 16'b0000011111110101;
    assign ROM[4] = 16'b0000001111111111;
    assign ROM[5] = 16'b0000001000000000;
    assign ROM[6] = 16'b0000000100000000;
    assign ROM[7] = 16'b0000000010000000;
    assign ROM[8] = 16'b0000000001000000;
    assign ROM[9] = 16'b0000000000100000;
    assign ROM[10] = 16'b0000000000010000;
    assign ROM[11] = 16'b0000000000001000;
    assign ROM[12] = 16'b0000000000000100;

    reg signed [15:0] x [0:12];
    reg signed [15:0] y [0:12];
```

```
reg signed [15:0] z [0:12];
wire signed [15:0] x_sum [0:11];
wire signed [15:0] y_sum [0:11];
wire signed [15:0] z_sum [0:11];
wire signed [15:0] x_a [0:11];
wire signed [15:0] y_a [0:11];
wire signed [15:0] z_a [0:11];
wire signed [15:0] x_b [0:11];
wire signed [15:0] y_b [0:11];
wire signed [15:0] z_b [0:11];

wire donex [0:11];
wire doney [0:11];
wire donez [0:11];

always @(posedge clk)
begin
    x[0] <= x_start;
    y[0] <= y_start;
    z[0] <= angle;
end

serial addX0(x_sum[0], x_a[0], x_b[0], clk, donex[0]);
serial addY0(y_sum[0], y_a[0], y_b[0], clk, doney[0]);
serial addZ0(z_sum[0], z_a[0], z_b[0], clk, donez[0]);

serial addX1(x_sum[1], x_a[1], x_b[1], clk, donex[1]);
serial addY1(y_sum[1], y_a[1], y_b[1], clk, doney[1]);
serial addZ1(z_sum[1], z_a[1], z_b[1], clk, donez[1]);

serial addX2(x_sum[2], x_a[2], x_b[2], clk, donex[2]);
serial addY2(y_sum[2], y_a[2], y_b[2], clk, doney[2]);
serial addZ2(z_sum[2], z_a[2], z_b[2], clk, donez[2]);

serial addX3(x_sum[3], x_a[3], x_b[3], clk, donex[3]);
serial addY3(y_sum[3], y_a[3], y_b[3], clk, doney[3]);
serial addZ3(z_sum[3], z_a[3], z_b[3], clk, donez[3]);

serial addX4(x_sum[4], x_a[4], x_b[4], clk, donex[4]);
serial addY4(y_sum[4], y_a[4], y_b[4], clk, doney[4]);
```

```
    serial addZ4(z_sum[4], z_a[4], z_b[4], clk, donez[4]);

    serial addX5(x_sum[5], x_a[5], x_b[5], clk, donex[5]);
    serial addY5(y_sum[5], y_a[5], y_b[5], clk, doney[5]);
    serial addZ5(z_sum[5], z_a[5], z_b[5], clk, donez[5]);

    serial addX6(x_sum[6], x_a[6], x_b[6], clk, donex[6]);
    serial addY6(y_sum[6], y_a[6], y_b[6], clk, doney[6]);
    serial addZ6(z_sum[6], z_a[6], z_b[6], clk, donez[6]);

    serial addX7(x_sum[7], x_a[7], x_b[7], clk, donex[7]);
    serial addY7(y_sum[7], y_a[7], y_b[7], clk, doney[7]);
    serial addZ7(z_sum[7], z_a[7], z_b[7], clk, donez[7]);

    serial addX8(x_sum[8], x_a[8], x_b[8], clk, donex[8]);
    serial addY8(y_sum[8], y_a[8], y_b[8], clk, doney[8]);
    serial addZ8(z_sum[8], z_a[8], z_b[8], clk, donez[8]);

    serial addX9(x_sum[9], x_a[9], x_b[9], clk, donex[9]);
    serial addY9(y_sum[9], y_a[9], y_b[9], clk, doney[9]);
    serial addZ9(z_sum[9], z_a[9], z_b[9], clk, donez[9]);

    serial addX10(x_sum[10], x_a[10], x_b[10], clk, donex[10]);
    serial addY10(y_sum[10], y_a[10], y_b[10], clk, doney[10]);
    serial addZ10(z_sum[10], z_a[10], z_b[10], clk, donez[10]);

    serial addX11(x_sum[11], x_a[11], x_b[11], clk, donex[11]);
    serial addY11(y_sum[11], y_a[11], y_b[11], clk, doney[11]);
    serial addZ11(z_sum[11], z_a[11], z_b[11], clk, donez[11]);

    genvar i;

    generate
    for (i=0; i < 12; i = i+1)
    begin: xyz
        wire z_sign;
        wire signed [15:0] x_shr, y_shr;

        assign x_shr = x[i] >>> i;
        assign y_shr = y[i] >>> i;
```

```
    assign z_sign = z[i][15];

    assign x_a[i] = x[i];
    assign y_a[i] = y[i];
    assign z_a[i] = z[i];
    assign x_b[i] = z_sign ? y_shr : 0 - y_shr;
    assign y_b[i] = z_sign ? 0 - x_shr : x_shr;
    assign z_b[i] = z_sign ? ROM[i] : 0 - ROM[i];

    always @(posedge clk)
    begin
        if(donex[i] && doney[i] && donez[i]) begin
            x[i+1] <= x_sum[i];
            y[i+1] <= y_sum[i];
            z[i+1] <= z_sum[i];
        end
    end
end
endgenerate

reg [3:0] count = 0;
reg change = 0;

always @ (posedge clk) begin
    if(btn == 2'b01) begin
        if(change == 0) begin
            count <= (count+1)%13;
            change <= 1;
        end
    end
    else
        change <= 0;

    dig1 <= x[count][15:12];
    dig2 <= x[count][11:8];
    dig3 <= x[count][7:4];
    dig4 <= x[count][3:0];
    LED <= 13'b0000000000000;
    LED[count] <= 1;
```

```
    end
endmodule
```

```
`timescale 1ns / 1ps

module disp(
    input clk,
    input [3:0] dig1,
    input [3:0] dig2,
    input [3:0] dig3,
    input [3:0] dig4,
    output reg [6:0] disp,
    output reg [3:0] anode
    );

    wire [6:0] disp1;
    wire [6:0] disp2;
    wire [6:0] disp3;
    wire [6:0] disp4;

    seg7 d1(disp1, dig1);
    seg7 d2(disp2, dig2);
    seg7 d3(disp3, dig3);
    seg7 d4(disp4, dig4);

    localparam N = 19;
    reg [N-1:0] count = 0;

    always @ (posedge clk)
    begin
        count <= count + 1;
    end

    always @ (*)
    begin
        case(count[N-1:N-2])
            2'b00: begin
                disp = disp4;
                anode = 4'b1110;
            end
```

```
            2'b01: begin
                disp = disp3;
                anode = 4'b1101;
            end
            2'b10: begin
                disp = disp2;
                anode = 4'b1011;
            end
            2'b11: begin
                disp = disp1;
                anode = 4'b0111;
            end
        endcase
    end
endmodule
```

```
`timescale 1ns / 1ps

module seg7(output reg [6:0]disp, input [3:0] x);

    always @(*)
    begin
        case (x)
            4'b0000:                    //Hexadecimal 0
                disp = 7'b1000000;
            4'b0001:                    //Hexadecimal 1
                disp = 7'b1111001;
            4'b0010:                   //Hexadecimal 2
                disp = 7'b0100100;
            4'b0011:                   //Hexadecimal 3
                disp = 7'b0110000;
            4'b0100:                   //Hexadecimal 4
                disp = 7'b0011001;
            4'b0101:                   //Hexadecimal 5
                disp = 7'b0010010;
            4'b0110:                   //Hexadecimal 6
                disp = 7'b0000010;
            4'b0111:                   //Hexadecimal 7
                disp = 7'b1111000;
            4'b1000:                   //Hexadecimal 8
```

```
                disp = 7'b0000000;
            4'b1001:                    //Hexadecimal 9
                disp = 7'b0011000;
            4'b1010:                    //Hexadecimal A
                disp = 7'b0001000;
            4'b1011:                    //Hexadecimal B
                disp = 7'b0000011;
            4'b1100:                    //Hexadecimal C
                disp = 7'b1000110;
            4'b1101:                    //Hexadecimal D
                disp = 7'b0100001;
            4'b1110:                    //Hexadecimal E
                disp = 7'b0000110;
            4'b1111:                    //Hexadecimal F
                disp = 7'b0001110;
            default:
                disp = 7'b1111111;
        endcase
    end
endmodule
```

```
`timescale 1ns / 1ps

module serial(sum, a, b, clk, done);
    input [15:0] a;
    input [15:0] b;
    input clk;
    reg [15:0] x;
    reg [15:0] z;
    output reg [15:0] sum;
    output reg done;
    reg cout;
    reg s;
    reg cin;
    reg [15:0] sum_temp;

    initial begin
        sum_temp <= 0;
        cin <= 0;
        done <= 0;
```

```
    end

    integer i;
    always @(posedge clk) begin
        for(i = 0; i < 16; i=i+1) begin
            x = a >>> i;
            z = b >>> i;

            {cout, s} = x[0] + z[0] + cin;

            sum_temp = {s, sum_temp[15:1]};
            cin = cout;

            if(i == 15) begin
                sum = sum_temp;
                done = 1;
            end
        end
    end
endmodule
```