Design an architecture for the CORDIC With 16-Stage Pipeline

Subham Ball
Dept. of System-on-chip Design
Indian Institute Of Technology–Palakkad
152202017@smail.iitpkd.ac.in

***Abstract*—This project goal is to implement and validate the design of an architecture that computes the CORDIC algorithm with pipelining.**

I. INTRODUCTION

This assignment focuses on the pipelining implementation of the well-known CORDIC algorithm. Before we go into pipelined CORDIC, let's first define ordinary CORDIC.
The Coordinate Rotation Digital Computer (CORDIC) algorithm is a commonly used technique for calculating trigonometric, logarithmic, and hyperbolic functions. It is particularly beneficial for digital signal processing (DSP) and machine learning (ML) techniques that demand fast and efficient calculations. The demand for specialized hardware to expedite signal processing and machine learning has expanded dramatically in recent years, and the CORDIC method is a popular alternative because to its simplicity and adaptability.
Pipelined CORDIC is a CORDIC algorithm version that divides the algorithm into numerous pipeline phases to boost throughput and minimise latency. Each pipeline stage of a pipelined CORDIC implementation handles a distinct set of algorithm iterations, allowing many iterations to be handled concurrently.
The pipelined CORDIC architecture supports a wide range of pipeline stages, from basic two-stage designs to more complicated systems with dozens of pipeline stages. The number of pipeline stages necessary is determined by the required precision of the computations and the system's expected throughput.
The pipelined CORDIC architecture has the benefit of being able to attain greater clock rates than non-pipelined devices, resulting in quicker calculation times. Pipelined designs can also minimise the hardware complexity required for a given degree of precision since each pipeline stage is responsible for a smaller portion of iterations.
For pipelined CORDIC systems, there are several trade-offs to consider. For example, the increased complexity of the pipeline stages might raise the system's overall power consumption. Moreover, because data must be propagated across the pipeline phases, pipelined architectures might incur extra delay.
Notwithstanding these drawbacks, pipelined CORDIC is a common approach in VLSI designs for signal processing and machine learning, and it is frequently utilised to construct high-speed and efficient calculations of complicated mathematical functions.

II. IMPLEMENTION

A. Architecture of Pipelined CORDIC

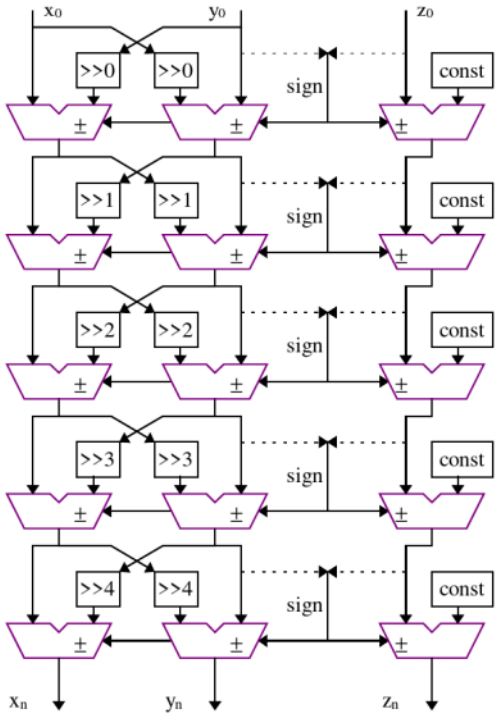

Fig. 1. Datapath

B. Verilog Code for Architecture

design.sv

```
module cordicpipe(input clk,
                  input signed [15:0] x_in,
                  input signed [15:0] y_in,
                  input signed [15:0] theta_in,
                  output signed [15:0] sinx,
                  output signed [15:0] cosx);

  reg [15:0]
x_reg0,x_reg1,x_reg2,x_reg3,x_reg4,x_reg5,x_reg6,x_reg7,x_reg8,x_reg9,x_reg
10,x_reg11,x_reg12,x_reg13,x_reg14,x_reg15,x_reg16;
  reg  [15:0]
y_reg0,y_reg1,y_reg2,y_reg3,y_reg4,y_reg5,y_reg6,y_reg7,y_reg8,y_reg9,y_reg
10,y_reg11,y_reg12,y_reg13,y_reg14,y_reg15,y_reg16;
  reg  [15:0]
theta0,theta1,theta2,theta3,theta4,theta5,theta6,theta7,theta8,theta9,theta
10,theta11,theta12,theta13,theta14,theta15,theta16;
  wire [15:0]
x_add0,x_add1,x_add2,x_add3,x_add4,x_add5,x_add6,x_add7,x_add8,x_add9,x_add
10,x_add11,x_add12,x_add13,x_add14,x_add15,x_add16;
  wire [15:0]
y_add0,y_add1,y_add2,y_add3,y_add4,y_add5,y_add6,y_add7,y_add8,y_add9,y_add
10,y_add11,y_add12,y_add13,y_add14,y_add15,y_add16;
  wire [15:0]
t_add0,t_add1,t_add2,t_add3,t_add4,t_add5,t_add6,t_add7,t_add8,t_add9,t_add
10,t_add11,t_add12,t_add13,t_add14,t_add15,t_add16;
  //stage0
  assign x_add0=x_reg0 - y_reg0;
  assign y_add0=y_reg0 + x_reg0;
  assign t_add0=theta0 - 'b0011001001000100;

  //stage1
   assign x_add1=theta1[15]?(x_reg1 + {y_reg1[15],y_reg1[15:1]}):(x_reg1 -
{y_reg1[15],y_reg1[15:1]});
   assign y_add1=theta1[15]?(y_reg1 - {x_reg1[15],x_reg1[15:1]}):(y_reg1 +
{x_reg1[15],x_reg1[15:1]});
   assign t_add1=theta1[15]?(theta1 +  16'h1DAC):(theta1 -  16'h1DAC);
   //stage2
   assign x_add2=theta2[15]?(x_reg2 + {{2{y_reg2[15]}},y_reg2[15:2]}):
(x_reg2 - {{2{y_reg2[15]}},y_reg2[15:2]});
   assign y_add2=theta2[15]?(y_reg2 - {{2{x_reg2[15]}},x_reg2[15:2]}):
(y_reg2 + {{2{x_reg2[15]}},x_reg2[15:2]});
   assign t_add2=theta2[15]?(theta2 +  16'hFAD):(theta2 -  16'hFAD);
   //stage3
   assign x_add3=theta3[15]?(x_reg3 + {{3{y_reg2[15]}},y_reg3[15:3]}):
(x_reg3 - {{3{y_reg2[15]}},y_reg3[15:3]});
   assign y_add3=theta3[15]?(y_reg3 - {{3{x_reg3[15]}},x_reg3[15:3]}):
(y_reg3 + {{3{x_reg3[15]}},x_reg3[15:3]});
   assign t_add3=theta3[15]?(theta3 +  16'h7F5):(theta3 -  16'h7F5);
   //stage4
   assign x_add4=theta4[15]?(x_reg4 + {{4{y_reg4[15]}},y_reg4[15:4]}):
(x_reg4 - {{4{y_reg4[15]}},y_reg4[15:4]});
   assign y_add4=theta4[15]?(y_reg4 - {{4{x_reg4[15]}},x_reg4[15:4]}):
(y_reg4 + {{4{x_reg4[15]}},x_reg4[15:4]});
   assign t_add4=theta4[15]?(theta4 +  16'h3FE):(theta4 -  16'h3FE);
   //stage5

   assign x_add5=theta5[15]?(x_reg5 + {{5{y_reg5[15]}},y_reg5[15:5]}):
(x_reg5 - {{5{y_reg5[15]}},y_reg5[15:5]});
   assign y_add5=theta5[15]?(y_reg5 - {{5{x_reg5[15]}},x_reg5[15:5]}):
(y_reg5 + {{5{x_reg5[15]}},x_reg5[15:5]});
   assign t_add5=theta5[15]?(theta5 +  16'h200):(theta5 -  16'h200);
   //stage6
   assign x_add6=theta6[15]?(x_reg6 + {{6{y_reg6[15]}},y_reg6[15:6]}):
(x_reg6 - {{6{y_reg6[15]}},y_reg6[15:6]});
   assign y_add6=theta6[15]?(y_reg6 - {{6{x_reg6[15]}},x_reg6[15:6]}):
(y_reg6 + {{6{x_reg6[15]}},x_reg6[15:6]});
   assign t_add6=theta6[15]?(theta6 +  16'h100):(theta6 -  16'h100);
   //stage7
   assign x_add7=theta7[15]?(x_reg7 + {{7{y_reg7[15]}},y_reg7[15:7]}):
(x_reg7 - {{7{y_reg7[15]}},y_reg7[15:7]});
   assign y_add7=theta7[15]?(y_reg7 - {{7{x_reg7[15]}},x_reg7[15:7]}):
(y_reg7 + {{7{x_reg7[15]}},x_reg7[15:7]});
   assign t_add7=theta7[15]?(theta7 +  16'h80):(theta7 -  16'h80);
   //stage8
   assign x_add8=theta8[15]?(x_reg8 + {{8{y_reg8[15]}},y_reg8[15:8]}):
(x_reg8 - {{8{y_reg8[15]}},y_reg8[15:8]});
   assign y_add8=theta8[15]?(y_reg8 - {{8{x_reg8[15]}},x_reg8[15:8]}):
(y_reg8 + {{8{x_reg8[15]}},x_reg8[15:8]});
   assign t_add8=theta8[15]?(theta8 +  16'h40):(theta8 -  16'h40);
   //stage9
   assign x_add9=theta9[15]?(x_reg9 + {{9{y_reg9[15]}},y_reg9[15:9]}):
(x_reg9 - {{9{y_reg9[15]}},y_reg9[15:9]});
   assign y_add9=theta9[15]?(y_reg9 - {{9{x_reg9[15]}},x_reg9[15:9]}):
(y_reg9 + {{9{x_reg9[15]}},x_reg9[15:9]});
   assign t_add9=theta9[15]?(theta9 +  16'h20):(theta9 -  16'h20);

   //stage10
    assign x_add10=theta10[15]?(x_reg10 +
{{10{y_reg10[15]}},y_reg10[15:10]}):(x_reg10 -
{{10{y_reg10[15]}},y_reg10[15:10]});
   assign y_add10=theta10[15]?(y_reg10 -
{{10{x_reg10[15]}},x_reg10[15:10]}):(y_reg10 +
{{10{x_reg10[15]}},x_reg10[15:10]});
   assign t_add10=theta10[15]?(theta10 +  16'h10):(theta10 -  16'h10);
   //stage11
   assign x_add11=theta11[15]?(x_reg11 +
{{11{y_reg11[15]}},y_reg11[15:11]}):(x_reg11 -
{{11{y_reg11[15]}},y_reg11[15:11]});
   assign y_add11=theta11[15]?(y_reg11 -
{{11{x_reg11[15]}},x_reg11[15:11]}):(y_reg11 +
{{11{x_reg11[15]}},x_reg11[15:11]});
   assign t_add11=theta11[15]?(theta11 +  16'h0008):(theta11 -  16'h0008);
   //stage12
  assign x_add12=theta12[15]?(x_reg12 + {{12{y_reg12[15]}},y_reg12[15:12]}):
(x_reg12 - {{12{y_reg12[15]}},y_reg12[15:12]});
   assign y_add12=theta12[15]?(y_reg12 -
{{12{x_reg12[15]}},x_reg12[15:12]}):(y_reg12 +
{{12{x_reg12[15]}},x_reg12[15:12]});
   assign t_add12=theta12[15]?(theta12 +  16'h0004):(theta12 -  16'h0004);
  //stage13;
  assign x_add13=theta13[15]?(x_reg13 + {{13{y_reg13[15]}},y_reg13[15:13]}):
(x_reg13 - {{13{y_reg13[15]}},y_reg13[15:13]});
   assign y_add13=theta13[15]?(y_reg13 -
{{13{x_reg13[15]}},x_reg13[15:13]}):(y_reg13 +
{{13{x_reg13[15]}},x_reg13[15:13]});
   assign t_add13=theta13[15]?(theta13 +  16'h0002):(theta13 -  16'h0002);

  //stage14
  assign x_add14=theta14[15]?(x_reg14 + {{14{y_reg14[15]}},y_reg14[15:14]}):
(x_reg14 - {{14{y_reg14[15]}},y_reg14[15:14]});
   assign y_add14=theta14[15]?(y_reg14 -
{{14{x_reg14[15]}},x_reg14[15:14]}):(y_reg14 +
{{14{x_reg14[15]}},x_reg14[15:14]});
   assign t_add14=theta14[15]?(theta14 +  16'h0001):(theta14 -  16'h0001);
  //stage15
  assign x_add15=theta15[15]?(x_reg15 + {{15{y_reg15[15]}},y_reg15[15:15]}):
(x_reg15 - {{15{y_reg15[15]}},y_reg15[15:15]});
   assign y_add15=theta15[15]?(y_reg15 -
{{15{x_reg15[15]}},x_reg15[15:15]}):(y_reg15 +
{{15{x_reg15[15]}},x_reg15[15:15]});
   assign t_add15=theta15[15]?(theta15 +  16'h0000):(theta15 -  16'h0000);

  //out
   assign cosx=x_add15;
   assign sinx=y_add15;
  always @(posedge clk)
    begin
       //stage0
       x_reg0<=x_in;
       y_reg0<=y_in;
       theta0<=theta_in;
       //stage1
       x_reg1<=x_add0;
       y_reg1<=y_add0;
       theta1<=t_add0;

        //stage2
         x_reg2<=x_add1;
        y_reg2<=y_add1;
        theta2<=t_add1;
        //stage3
         x_reg3<=x_add2;
        y_reg3<=y_add2;
        theta3<=t_add2;
        //stage4
         x_reg4<=x_add3;
        y_reg4<=y_add3;
        theta4<=t_add3;
        //stage5
         x_reg5<=x_add4;
        y_reg5<=y_add4;
        theta5<=t_add4;
        //stage6
         x_reg6<=x_add5;
        y_reg6<=y_add5;
        theta6<=t_add5;
        //stage7
         x_reg7<=x_add6;
        y_reg7<=y_add6;
        theta7<=t_add6;
        //stage8
         x_reg8<=x_add7;
        y_reg8<=y_add7;
        theta8<=t_add7;
        //stage9

         x_reg9<=x_add8;
        y_reg9<=y_add8;
        theta9<=t_add8;
        //stage10
         x_reg10<=x_add9;
        y_reg10<=y_add9;
        theta10<=t_add9;
        //stage11
         x_reg11<=x_add10;
        y_reg11<=y_add10;
        theta11<=t_add10;
        //stage12
         x_reg12<=x_add11;
        y_reg12<=y_add11;
        theta12<=t_add11;
        //stage13
         x_reg13<=x_add12;
        y_reg13<=y_add12;
        theta13<=t_add12;
        //stage14
            x_reg14<=x_add13;
        y_reg14<=y_add13;
        theta14<=t_add13;
        //stage15
            x_reg15<=x_add14;
        y_reg15<=y_add14;
        theta15<=t_add14;
    end

endmodule

```

C. Testbench

Now let's see the testbench for the same

testbench.sv

```
module tb;
  reg clk;
  reg signed [15:0] x_in,y_in,theta_in;
  wire signed [15:0] sinx,cosx;
  cordicpipe
dut(.clk(clk),.x_in(x_in),.y_in(y_in),.theta_in(theta_in),.sinx(sinx),.cosx
(cosx));
  always #3 clk=~clk;
  initial
    begin
      clk='d0;
      drive_values();
      #150
      $finish;
    end
  task drive_values();
    @(negedge clk)
    x_in ='h26DD;
    y_in='h0;
    theta_in= 'b0011001001000100;
    @(negedge clk)
      x_in ='h26DD;
    y_in='h0;
    theta_in= 'd0;
  endtask
  initial
    begin
      $dumpfile("dump.vcd");
      $dumpvars(0);
    end
endmodule
```

Fig. 2. Testbench

III. Output

Let us now examine the results of the same. We can see from the above graphic that (1, 0) has rotated to (1, 1) due to the 45 degrees that we have indicated. Because we did not account for scaling, x out and y out are not 1.

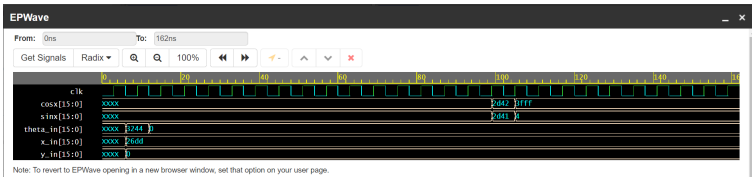

Fig. 3. Waveform

IV. Conclusion

Finally, Pipelined CORDIC is a CORDIC algorithm version that separates the algorithm into numerous pipeline phases to boost throughput and minimise latency. Each pipeline step processes a distinct set of algorithm iterations, allowing numerous iterations to be processed at the same time.
This architecture allows for greater clock frequencies while reducing hardware complexity, however it may raise power consumption and bring extra delay. Pipelined CORDIC is widely utilised in VLSI designs for signal processing and machine learning, allowing for the fast and efficient computing of complicated mathematical functions. .The link to the EDA Playground is: playground link