Square Root Computation using CORDIC

Square root computation using CORDIC (Coordinate Rotation Digital Computer) is a numerical method that efficiently calculates the square root of a given non-negative real number. The CORDIC algorithm relies on iterative shifts and rotations to approximate the square root using only simple arithmetic operations like additions, subtractions, and bit shifts.

The basic idea behind CORDIC is to iteratively adjust the angle of a vector in a coordinate system until it converges to the desired value. In the context of square root computation, the algorithm starts with an initial estimate and then iteratively refines it using CORDIC rotations until the desired precision is achieved.

CODE

```
`include "rotation.sv"
`include "vectoring.sv"
`include "input_scaling.sv"
`include "output_scaling.sv"

module cordic_division(
  input clk,reset,
  input operands_val,
  input signed [15:0] A,
  output signed [15:0] sqrt_x,
  output ready, sqrt_valid);

  reg[1:0] state, state_next;

  parameter width=16;

  wire signed [15:0] X_scaled;
  wire [2:0] k;
  wire scale_valid, shift_sig;
  scale_num scale_num(
    .clk(clk),.reset(reset),.ack(sqrt_valid),
    .num(A),
    .inp_valid(operands_val),
    .X_scaled(X_scaled),
    .k(k),
    .out_valid(scale_valid), .ready(ready), .shift_sig(shift_sig)
```

```
  );
  localparam signed val_1 = 16'hf800; //-1
  wire signed [15:0] x_2_1 = (X_scaled<<1) + val_1; //2x-1
  wire beta_valid;
  wire signed [15:0] beta;
  wire vect_ready;
  wire count_eq_15;

  CORDIC_vectoring CORDIC_vectoring(
      .clk(clk),.reset(reset),
    .operands_val(shift_sig),.out_valid(beta_valid),
    .A(x_2_1), .B(16'h04dc),
    .theta(beta), .ready(vect_ready),.count_eq_15(count_eq_15)
  );

  // For rotation mode
  wire rot_ready;
  wire signed [15:0] rot_x_start=16'h04dc;
  wire signed [15:0] rot_y_start=16'h0;
  wire signed [15:0] rot_sin;
  wire rot_valid;
  wire count_2_eq_15;
  wire signed [15:0] rot_out;

  CORDIC_rotation CORDIC_rotation(
    .clk(clk),.reset(reset),
    .operands_val(count_eq_15),.out_valid(rot_valid),
    .cos(rot_out),.sin(rot_sin),
    .x_start(rot_x_start),.y_start(rot_y_start),
    // beta_2 should be put instead of beta
    .angle(beta>>>1),.ready(rot_ready), .count_eq_15(count_2_eq_15)
  );

  output_scaling output_scaling(
    .clk(clk),.reset(reset),
    .inp_valid(count_2_eq_15),
    .k(k),
    .rot_inp(rot_out),
    .out(sqrt_x),
    .out_valid(sqrt_valid)
);

  //-----------------------------------------------//
```

```
  localparam IDLE = 2'd0;
  localparam BUSY = 2'd1;
  localparam DONE = 2'd2;

  assign ready = (state==IDLE);
  assign sqrt_valid = (state==DONE);

  always @(*)
  begin
  // Default is to stay in the same state

  case (state)
    IDLE :
      if (operands_val)
        state_next = BUSY;
      else
        state_next = IDLE;

    BUSY :
      if (count_2_eq_15)
        state_next = DONE;
      else
        state_next = BUSY;

    DONE :
        state_next = IDLE;

  endcase
 end

   always @(posedge clk)
    if(reset)
      state <= IDLE;
    else
      state <= state_next;

endmodule


module CORDIC_rotation(clk, reset, operands_val, out_valid, cos, sin,
x_start, y_start, angle, ready, count_eq_15);
```

```
 // Inputs
 input clk;
 input reset;
 input operands_val;
 input signed [15:0] x_start, y_start;
 input signed [15:0] angle;

 // Outputs
 output signed [15:0] sin, cos;
 output out_valid;
 output reg count_eq_15;
 output ready;

localparam IDLE = 2'd0;
localparam BUSY = 2'd1;
localparam DONE = 2'd2;

 // Generate table of atan values
reg signed [15:0] atan_out;
reg signed [15:0] x_reg;
reg signed [15:0] y_reg;
reg signed [15:0] theta_reg;

wire [15:0] x_mux_out, y_mux_out, theta_mux_out;
reg [15:0] x_add_out, y_add_out, theta_add_out;

reg [15:0] x_shift, y_shift;

reg [3:0] count;
wire [3:0] address;
reg [1:0] state, state_next;
wire count_en;

assign address = count;

// Look up table in 5.11 format

always @(*)
   begin
     case(address)
      4'd0:  atan_out = 'b0000011001001000 ;
      4'd1:  atan_out = 'b0000001110110110;
      4'd2:  atan_out = 'b0000000111110110;
```

```
        4'd3:  atan_out = 'b0000000011111111;
        4'd4:  atan_out = 'b0000000010000000 ;
        4'd5:  atan_out = 'b0000000001000000;
        4'd6:  atan_out = 'b0000000000100000;
        4'd7:  atan_out = 'b0000000000010000;
        4'd8:  atan_out = 'b0000000000001000;
        4'd9:  atan_out = 'b0000000000000100;
        4'd10: atan_out = 'b0000000000000010;
        4'd11: atan_out = 'b0000000000000001;
        4'd12: atan_out = 'b0000000000000000;
        4'd13: atan_out = 'b0000000000000000 ;
        4'd14: atan_out = 'b0000000000000000 ;
        4'd15: atan_out = 'b0000000000000000 ;
       endcase
   end


  // 2:1 Multiplexers

 assign x_mux_out = (state == IDLE) ? x_start : x_add_out;
 assign y_mux_out = (state == IDLE) ? y_start : y_add_out;
 assign theta_mux_out = (state == IDLE) ? angle : theta_add_out;
 assign count_eq_15 = (count == 4'd15);
 assign out_valid = (state == DONE);
 assign count_en = (state == BUSY);
 assign ready = (state == IDLE);
 assign cos = x_reg;
 assign sin = y_reg;



//  Barrel Shifter
  always@(*)
    begin
      case(count)
        0: begin
          x_shift={x_reg[15:0]};
          y_shift={y_reg[15:0]};
           end
        1: begin
          x_shift={x_reg[15:1]};
          y_shift={y_reg[15:1]};
           end
```

```
        2: begin
          x_shift={x_reg[15:2]};
          y_shift={y_reg[15:2]};
           end
        3: begin
          x_shift={x_reg[15:3]};
          y_shift={y_reg[15:3]};
           end
        4: begin
          x_shift={x_reg[15:4]};
          y_shift={y_reg[15:4]};
           end
        5: begin
          x_shift={x_reg[15:5]};
          y_shift={y_reg[15:5]};
           end
        6: begin
          x_shift={x_reg[15:6]};
          y_shift={y_reg[15:6]};
           end
        7: begin
          x_shift={x_reg[15:7]};
          y_shift={y_reg[15:7]};
           end
        8: begin
          x_shift={x_reg[15:8]};
          y_shift={y_reg[15:8]};
           end
        9: begin
          x_shift={x_reg[15:9]};
          y_shift={y_reg[15:9]};
           end
        10: begin
          x_shift={x_reg[15:10]};
          y_shift={y_reg[15:10]};
           end
        11: begin
          x_shift={x_reg[15:11]};
          y_shift={y_reg[15:11]};
           end
        12: begin
          x_shift={x_reg[15:12]};
          y_shift={y_reg[15:12]};
```

```
          end
        13: begin
          x_shift={x_reg[15:13]};
          y_shift={y_reg[15:13]};
          end
        14: begin
          x_shift={x_reg[15:14]};
          y_shift={y_reg[15:14]};
          end
        15: begin
          x_shift={x_reg[15]};
          y_shift={y_reg[15]};
          end
      endcase
    end

// add_out muxes
always @ (*)
  if (theta_reg[15] == 1)
     begin
       x_add_out = x_reg + y_shift;
       y_add_out = y_reg - x_shift;
       theta_add_out = theta_reg + atan_out;
      end
     else
      begin
       x_add_out = x_reg - y_shift;
       y_add_out = y_reg + x_shift;
       theta_add_out = theta_reg - atan_out;
      end

  // x and y reg
  always @(posedge clk)
   if(reset)
     begin
     x_reg <= 0;
     y_reg <= 0;
     theta_reg <= 0;
     end
    else
     begin
      x_reg <= x_mux_out;
      y_reg <= y_mux_out;
```

```
       theta_reg <= theta_mux_out;
      end


 always @(*)
  begin

  case (state)
    IDLE :
      if (operands_val)
        state_next = BUSY;
      else
        state_next = IDLE;

    BUSY :
      if (count_eq_15)
        state_next = DONE;
      else
        state_next = BUSY;

    DONE :
        state_next = IDLE;

  endcase
 end


   always @(posedge clk)
    if(reset)
      state <= IDLE;
    else
      state <= state_next;

   always @(posedge clk)
     if(count_en)
      count <= count + 1;
    else
      count <= 0;

endmodule


// Code your design here
```

```
module CORDIC_vectoring(clk, reset, operands_val, out_valid, A, B, theta,
ready, count_eq_15);

   parameter width = 16;

   // Inputs
   input clk;
   input reset;
   input operands_val;
   input signed [width-1:0] A, B;

   // Outputs
   output signed [width-1:0] theta;
   output out_valid;
   output ready;
   output reg count_eq_15;

  localparam IDLE = 2'd0;
  localparam BUSY = 2'd1;
  localparam DONE = 2'd2;

   // Generate table of atan values
  reg signed [width-1:0] atan_out;
  reg signed [width-1:0] x_reg;
  reg signed [width-1:0] y_reg;
  reg signed [width-1:0] A_reg;

  reg signed [width-1:0] theta_reg;

  wire [width-1:0] x_mux_out, y_mux_out, theta_mux_out, A_mux_out;
  reg [width-1:0] x_add_out, y_add_out, theta_add_out;

  reg [width-1:0] x_shift, y_shift;

  reg [3:0] count;
  wire [3:0] address;
  reg [1:0] state, state_next;
  wire count_en;

  reg x_gt_A;

  assign address = count;
```

```
 // Look up table in 5.11 format

 always @(*)
    begin
      case(address)
       4'd0:  atan_out = 'b0000011001001000 ;
       4'd1:  atan_out = 'b0000001110110110;
       4'd2:  atan_out = 'b0000000111110110;
       4'd3:  atan_out = 'b0000000011111111;
       4'd4:  atan_out = 'b0000000010000000 ;
       4'd5:  atan_out = 'b0000000001000000;
       4'd6:  atan_out = 'b0000000000100000;
       4'd7:  atan_out = 'b0000000000010000;
       4'd8:  atan_out = 'b0000000000001000;
       4'd9:  atan_out = 'b0000000000000100;
       4'd10: atan_out = 'b0000000000000010;
       4'd11: atan_out = 'b0000000000000001;
       4'd12: atan_out = 'b0000000000000000;
       4'd13: atan_out = 'b0000000000000000 ;
       4'd14: atan_out = 'b0000000000000000 ;
       4'd15: atan_out = 'b0000000000000000 ;
      endcase
  end


 // 2:1 Multiplexers

assign x_mux_out = (state == IDLE) ? B : x_add_out;
assign y_mux_out = (state == IDLE) ? 0 : y_add_out;

assign A_mux_out = (state==IDLE) ? A : A_reg;

assign theta_mux_out = (state == IDLE) ? 0 : theta_add_out;
assign out_valid = (state == DONE) ;
assign count_en = (state == BUSY);
assign ready = (state == IDLE);

assign x_gt_A = (x_reg>A_reg);
assign count_eq_15 = (count == 4'd15);

assign theta = theta_reg;

//  Barrel Shifter
```

```
always@(*)
  begin
    case(count)
      0: begin
        x_shift={x_reg[15:0]};
        y_shift={y_reg[15:0]};
         end
      1: begin
        x_shift={x_reg[15:1]};
        y_shift={y_reg[15:1]};
         end
      2: begin
        x_shift={x_reg[15:2]};
        y_shift={y_reg[15:2]};
         end
      3: begin
        x_shift={x_reg[15:3]};
        y_shift={y_reg[15:3]};
         end
      4: begin
        x_shift={x_reg[15:4]};
        y_shift={y_reg[15:4]};
         end
      5: begin
        x_shift={x_reg[15:5]};
        y_shift={y_reg[15:5]};
         end
      6: begin
        x_shift={x_reg[15:6]};
        y_shift={y_reg[15:6]};
         end
      7: begin
        x_shift={x_reg[15:7]};
        y_shift={y_reg[15:7]};
         end
      8: begin
        x_shift={x_reg[15:8]};
        y_shift={y_reg[15:8]};
         end
      9: begin
        x_shift={x_reg[15:9]};
        y_shift={y_reg[15:9]};
```

```
          end
        10: begin
          x_shift={x_reg[15:10]};
          y_shift={y_reg[15:10]};
          end
        11: begin
          x_shift={x_reg[15:11]};
          y_shift={y_reg[15:11]};
          end
        12: begin
          x_shift={x_reg[15:12]};
          y_shift={y_reg[15:12]};
          end
        13: begin
          x_shift={x_reg[15:13]};
          y_shift={y_reg[15:13]};
          end
        14: begin
          x_shift={x_reg[15:14]};
          y_shift={y_reg[15:14]};
          end
        15: begin
          x_shift={x_reg[15]};
          y_shift={y_reg[15]};
          end
      endcase
    end


always @ (*)
  if (x_gt_A)
     begin
       x_add_out = x_reg - y_shift;
       y_add_out = y_reg + x_shift;
       theta_add_out = theta_reg + atan_out;
      end
     else
      begin
       x_add_out = x_reg + y_shift;
       y_add_out = y_reg - x_shift;
       theta_add_out = theta_reg - atan_out;
      end
```

```
  always @(posedge clk)
   if(reset)
     begin
     x_reg <= 0;
     y_reg <= 0;
     A_reg <= A_mux_out;
     theta_reg <= 0;
     end

   else
     begin
      x_reg <= x_mux_out;
      y_reg <= y_mux_out;
      A_reg <= A_mux_out;
      theta_reg <= theta_mux_out;
     end


always @(*)
 begin

 case (state)
   IDLE :
     if (operands_val)
       state_next = BUSY;
     else
       state_next = IDLE;

   BUSY :
     if (count_eq_15)
       state_next = DONE;
     else
       state_next = BUSY;

   DONE :
       state_next = IDLE;

 endcase
end


  always @(posedge clk)
   if(reset)
```

```
      state <= IDLE;
    else
      state <= state_next;

   always @(posedge clk)
     if(count_en)
      count <= count + 1;
    else
      count <= 0;

endmodule


module scale_num(
  input clk,reset,ack,
  input signed [15:0] num,
  input inp_valid,
  output reg signed [15:0] X_scaled,
  output reg [2:0] k,
  output out_valid, ready, shift_sig
);

  localparam IDLE = 2'b00;
  localparam BUSY = 2'b01;
  localparam DONE = 2'b10;

  reg [1:0] state,next_state;

  assign out_valid = (state==DONE);
  assign ready = (state == IDLE);

  assign shift_sig = (X_scaled<=16'h066e) & (state==BUSY); // 066e is
0.80362
                                                          // i.e, 2x-1 <=
0.6072
  reg [1:0] mux_sig;

  wire signed [15:0] X_shift_2 = X_scaled>>>2; // right shift by 2

  reg signed [15:0] mux_out;
  reg [2:0] k_mux_out;//counter k

  always@(*)
```

```
  begin
    if(inp_valid)
      k_mux_out=0;
    else if(state==BUSY)
      begin
        if(!shift_sig)
          k_mux_out=k+1;
        else
          k_mux_out=k;
      end
  end

// next_state
always@(*)
  begin
    case(state)
      IDLE:
        begin
          if(inp_valid) next_state=BUSY;
          else next_state=IDLE;
        end
      BUSY:
        begin
          if(shift_sig)
            next_state=DONE;
        end
      DONE:
        begin
          if (ack) next_state=IDLE;
          else next_state=DONE;
        end
    endcase
  end

// x_scaled and k mux signals
always@(*)
  begin
    case(state)
      IDLE:
        mux_sig=0;
      BUSY:
        begin
          if(!shift_sig)
```

```
              mux_sig=1;
            else
              mux_sig=2;
          end
        DONE:
          mux_sig=2;
      endcase
    end

      // mux output
    always@(*)
      begin
        case(mux_sig)
          0: mux_out=num;
          1: mux_out= X_shift_2;
          2: mux_out=X_scaled;
        endcase
      end

  // state assignment
   always@(posedge clk)
     begin
       if(reset)
         begin
           state<=IDLE;
           X_scaled<=0;
           k<=0;
         end
       else
         begin
           state<=next_state;
           X_scaled<=mux_out;
           k<=k_mux_out;
         end
     end

endmodule

module output_scaling
  (
      input clk,reset,
    input inp_valid,
    input [2:0] k,
```

```
    input signed [15:0] rot_inp,
    output reg signed [15:0] out,
    output reg out_valid
);
  wire signed [15:0] rot_shifted;
  assign rot_shifted=rot_inp<<k; // left shifting by k

  // passing values to the register
  always@(posedge clk)
    if(reset)
      begin
        out_valid<=0;
        out<=0;
      end
    else
      begin
        out_valid<=inp_valid;
        if (inp_valid)
          out<=rot_shifted;
        else
          out<=0;
      end

endmodule
```

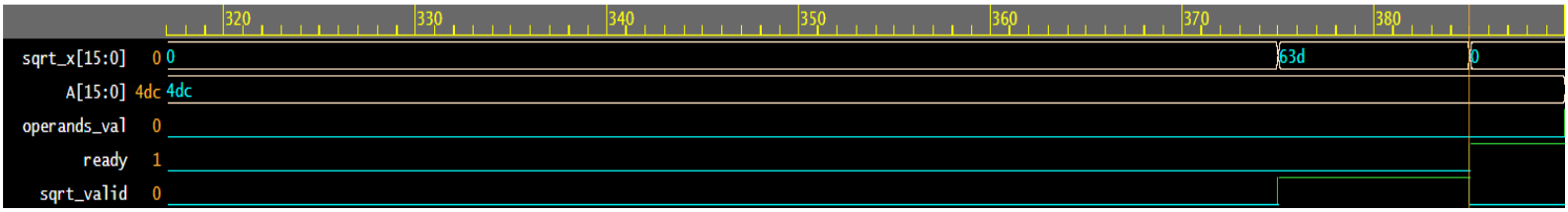
TESTBENCH

```
module SQRT_tb;

  reg signed [15:0] A;
  reg operands_val;
  reg reset;
  reg clk;
  wire sqrt_valid, ready;

  wire signed [15:0] sqrt_x;

  localparam SF = 2.0**-11.0;  // scaling factor is 2^-14

  // Instantiate the Unit Under Test (UUT)
```

```
  cordic_division div_mod  (.operands_val(operands_val),
                            .A(A),
                            .clk(clk),
                            .reset(reset),
                            .sqrt_valid(sqrt_valid),
                            .sqrt_x(sqrt_x),
                            .ready(ready)
                           );


  // Clock generation

  always #5 clk = ~clk;


  // Driving Stimulus

    initial
    begin
      drive_reset();

      drive_input(16'h04dc); // a is 0.6
      check_output(); // outp should be 0.7745 (2*0.6-1, cosin(..),theta/2,
cos(theta/2))

      drive_input(16'h0600); // a is 0.75
      check_output(); // outp should be 0.8366

      drive_input(16'h10cd); // a is 2.1
      check_output(); // outp should be 1.45

      drive_input(16'h199a); // a is 3.2
      check_output(); // outp should be 1.78

      drive_input(16'h4348); // a is 8.41
      check_output(); // outp should be 2.9

      drive_input(16'h4800); // a is 9
      check_output(); // outp should be 3

      drive_input(16'h5800); // a is 11
      check_output(); // outp should be 3.31
```

```
      drive_input(16'h6200); // a is 12.25
      check_output(); // outp should be 3.5

      repeat(5)@(negedge clk)
     $finish;
    end

  task drive_reset();
    $display ("Driving the reset");
    clk = 1'b0;
    @ (negedge clk)
    reset = 0;
    @ (negedge clk)
    reset = 1;
    @ (negedge clk)
    reset = 0;
  endtask

  task drive_input(input signed [15:0] a);

    wait (ready == 1)
       $display ("Received the ready signal and driving the Input");
       @ (negedge clk)
        operands_val = 1;
        A = a;
        @ (negedge clk)
        operands_val = 0;
   endtask


  task check_output();
    @ (posedge sqrt_valid)
    $display ("Recieved Output Valid");
    $display("(%f) = %f ", $itor(A * SF), $itor(sqrt_x * SF));

  endtask

   initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0);
   end
endmodule
```

OUTPUT


320
330
340
350
360
370
380
sqrt_x[15:0] 0
0
63d
0
A[15:0] 4dc
4dc
operands_val 0
ready 1
sqrt_valid 0