Hardware Lab

Embedded Systems[M.Tech]

CORDIC Based Implementation Of Fast Fourier Transform

Submitted to :-

Dr. Amit M. Joshi

[Asst. professor]

(MNIT Jaipur)

Submitted by :-

1. Lucky (2022PEB5136)

CORDIC based implementation of Fast Fourier Transform

Abstract :

In this project , high-speed real-time Fast Fourier Transform (FFT) processor is designed on FPGA which is based on Coordinate Rotation Digital Computer (CORDIC) algorithm. The CORDIC algorithm will reduce the hardware complexity compared to the direct implementation of the butterflies using complex multipliers. Fast Fourier Transform processor based on CORDIC is implemented. The key ideas are replacing the sine and cosine twiddle factors in conventional FFT architecture by iterative CORDIC rotations which allow the reduction in read-only memory (ROM). The use of CORDIC in FFT results in the elimination of multipliers, saves area, power and cost CORDIC finds many applications as it providing a simpler way of computing complex multiplications. It is proved that CORDIC is most suitable alternative.

CORDIC is implemented with the help of XILINX VIVADO 2016.2

Introduction:

CORDIC stands for Coordinate Rotation Digital Computer. It calculates the value of trigonometric functions like sine, cosine, magnitude and phase to any desired precision. It can also calculate hyperbolic functions (such as sinh, cosh and tanh). The CORDIC algorithm does not use calculus based methods such as polynomial or rational function approximation. CORDIC algorithm revolves around the idea of "rotating" the phase of a complex number, by multiplying it by a succession of constant values.

Since it is an iterative method it has the advantage over the other methods of being able to get better accuracy by doing more iteration, whereas the Taylor approximation and the Polynomial interpolation methods need to be averaged to get better results.

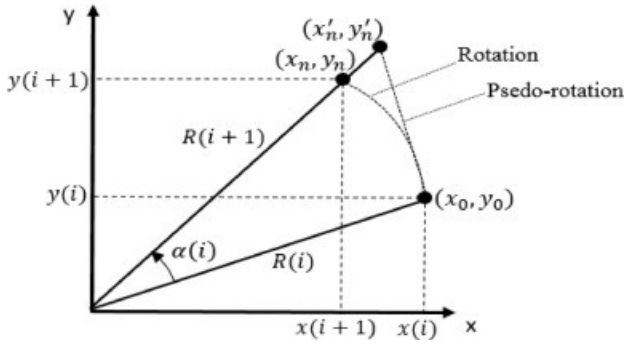
Cordic Algorithm :

Volder's algorithm is derived from the general equations for a vector rotation. If a vector V with coordinates (x, y) is rotated through an angle Ø then a new vector V' with new coordinates (x', y') is formed where x' and y' can be obtained using x, y and Ø from the following method. For the ease of calculation here only rotation in anticlockwise direction is observed first. So the individual equations for x' and y' can be rewritten as

$$x' = \cos Ø(x - y \tan Ø)$$

$$y' = \cos Ø(y + x \tan Ø)$$

$$z' = z + Ø$$



here Ø is the angle of rotation and z is the argument.

$$k = \prod_{i=0}^{7} k_i = \cos\emptyset_0 \cos\emptyset_1 \cos\emptyset_2 \cos\emptyset_3 \cos\emptyset_4 \cos\emptyset_5 \cos\emptyset_6 \cos\emptyset_7$$

$$= \cos 45^0 \cos 26.565^0 \ldots\ldots\ldots\ldots \cos 0.4469^0$$

$$= 0.6073$$

From the previous table it can be seen that precision up to 0.44690 is possible for 8-bit CORDIC hardware. These Øi are stored in the ROM of the hard ware of the CORDIC hardware as a look up table. The CORDIC Algorithm can be used in iterative mode, to simplify each rotation, picking αi(angle of rotation in ith iteration) such that αi= (di .2-i). di is such that it has value +1 or -1 depending upon the rotation i. e. di ϵ {+1,-1} .

After m iteration in rotation mode, when z (m) is sufficiently close to zero. We have Σαi=z, and the CORDIC equations become: The constant K in the preceding equation is k = 1.646760258121……

$$x_{i+1} = \left[x_i - d_i y_i 2^{-i}\right]$$

$$y_{i+1} = \left[y_i + d_i x_i 2^{-i}\right]$$

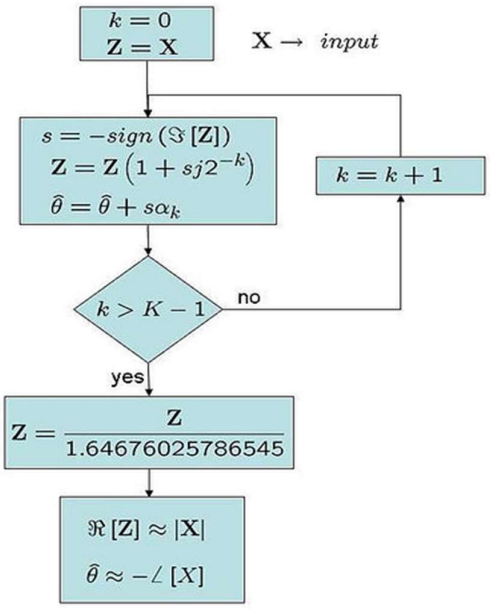
$$z_{i+1} = z_i - d_i \tan^{-1} 2^{-i}$$

$$x_m = k(x\cos z - y\sin z)$$

$$y_m = k(y\cos z + x\sin z)$$

$$z_m = 0$$

Rule: choose di ϵ { -1,1} such that z --> 0 Thus, to compute cos z and sin z, one can start with x = 1/K = 0.607252935….. And y = 0, then, as zm tends to 0 with CORDIC iterations in rotation mode, xm and ym converge to cos z and sin z, respectively. Once sin z and cos z are known, tan z can be found out through necessary division.



Cordic hardware and Architecture :

CORDIC is generally faster than other approaches when a hardware multiplier is unavailable (e.g. in a microcontroller) or when the number of gates required to implement the function is to be minimized (e.g. in an FPGA). On the other hand, when a hardware multiplier is available (e.g. in a DSP microprocessor), table lookup methods and power series are generally faster than CORDIC. A straight forward hardware implementation for CORDIC arithmetic is shown below. It requires three registers for x, y and z, a look up table to store the values of αi=tan-1 2 -i , and two shifter to supply the terms 2-i x and 2-i y to the adder/subtractor units. The di factor (-1 and 1) is accommodated by selecting the (shift) operand or its complement.

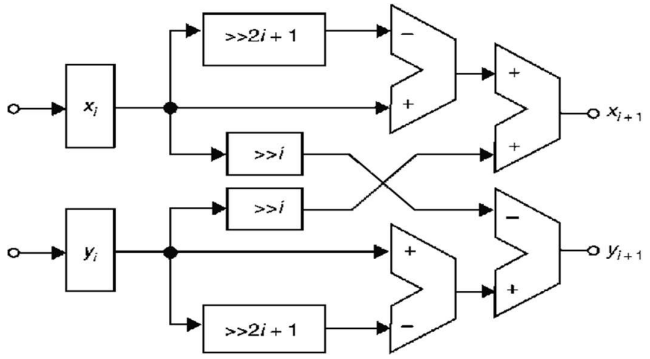


Fig: Cordic Architecture

Digital signal processing is needed more as technology spreads. Today, people want lower cost, area, electricity, and speed prompted the creation of more advanced DSP algorithms to improve performance.

DSP's most powerful tool is DFT. DFT uses arithmetic to breakdown a sequence. Slow and difficult computing. Formula to calculate the FFT:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}$$

$$k = 0, 1, 2 \ldots \ldots N\text{-}1$$

X(k) is the Discrete Fourier Transform and x(n) is a sequence of samples W_N is called twiddle factor. Due to its complexity, Fast Fourier Transform has been proposed by Cooley and Tukey [2]. Fast Fourier Transform is a cleaver trick for obtaining the same result as the DFT, using less computation time. FFT decomposes the sequence, it follows the rule of divide and conquer. Plenty of work has been done on FFT like radix-2, radix-4, split, Hybrid.

$$W_N = e^{-j2\pi/N} = cos\frac{2\pi}{N} - j\,sin\frac{2\pi}{N}$$

Let, Q be a complex number

$$Q = b_R + jb_I$$

When Q multiplied by twiddle factor, we get

$$QW_N^{kn} = (b_R + jb_I) \times \left(cos\frac{2\pi kn}{N} - j\,sin\frac{2\pi kn}{N}\right)$$

$$QW_N^{kn} = \left(b_R \cos\left(\frac{2\pi kn}{N}\right) + b_I \sin\left(\frac{2\pi kn}{N}\right)\right) + j\left(b_I \cos\left(\frac{2\pi kn}{N}\right) - b_R \sin\left(\frac{2\pi kn}{N}\right)\right)$$

There are two methods for computing FFT are when input sequence is broken into even and odd sequences called Decimation in Frequency.

In FFT processor, butterfly is the basic unit and using single butterfly whole computation is done. So, main concern is on butterfly unit. It comprises of Twiddle factor and complex addition and multiplication. FFT requires $N\log_2 N$ computations. Using CORDIC in place of twiddle factor can solve the problem of complex multiplication. It can also solve the problem of space required on ROM and also time required to perform the operation.

Fast Fourier Transform

An efficient algorithm proposed by Cooley-Tukey is the fastest version of DFT. In this paper, implementation of radix-2 DIT algorithm for FFT computations is presented. A Discrete Fourier Transform X (k), $0 \leq k \leq N-1$ is defined in (1) can be computed using FFT. This N point DFT is broken into two N/2 point DFT groups of even and odd. Thus FFT uses a recursive algorithm.

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} a(r) W_N^{k(2r)} + \sum_{r=0}^{\frac{N}{2}-1} b(2r+1) W_N^{k(2r+1)}$$

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} a(r) W_N^{k2r} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} b(2r+1) W_N^{k2r}$$

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} a(r) W_{N/2}^{kr} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} b(2r+1) W_{N/2}^{kr}$$

$$X(k) = A(k) + W_N^k B(k) \qquad k=0,1.....N/2$$

Thus, A(k) and B(k) are obtained by decimating x(n) by a factor of 2, and resulting FFT algorithm is called a decimation-in-time algorithm. W^K_N is called the twiddle factor.

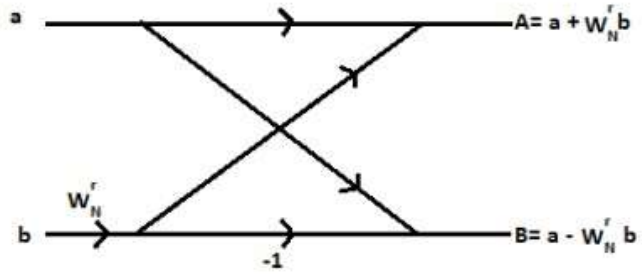


Fig : A basic Butterfly Structure

In this implementation the output is in normal sequence but input is bit reversed. 8-point radix-2 FFT is shown below. Radix-2 butterfly computation element diagram is shown. It comprises of sine generator, multipliers, and addition/subtraction units. FFT require ROM and complex multipliers for computation. Due to which FFT become more complex for computation.Butterfly unit require multipliers, addition and subtraction unit for computation

and this acquire large area, cost and power. To cut down cost, area and power and to increase speed, use of CORDIC in butterfly unit is an unavoidable demand of today.

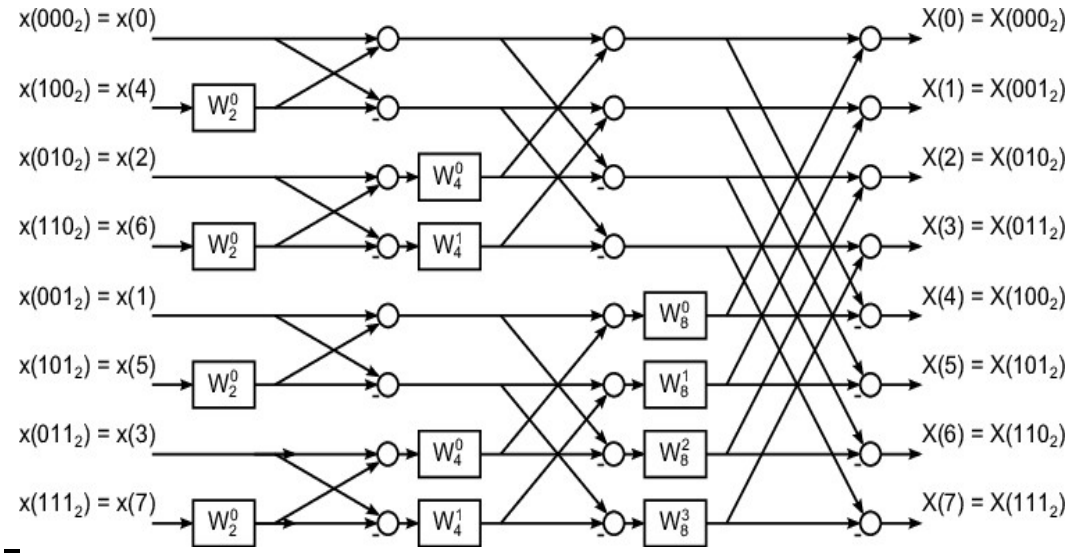


Fig: 8-point radix 2 DIT FFT

Implementation:

The main module is main_fft. Further, instance of Cordic and Butterfly module is created inside main_fft. Implementation is done using VIO (Virtual Input/Output) where input is given using VIO and we get output with the help of ILA (Integrated Logic Analyzer). There is another module by which we have displayed the output on seven segment display.

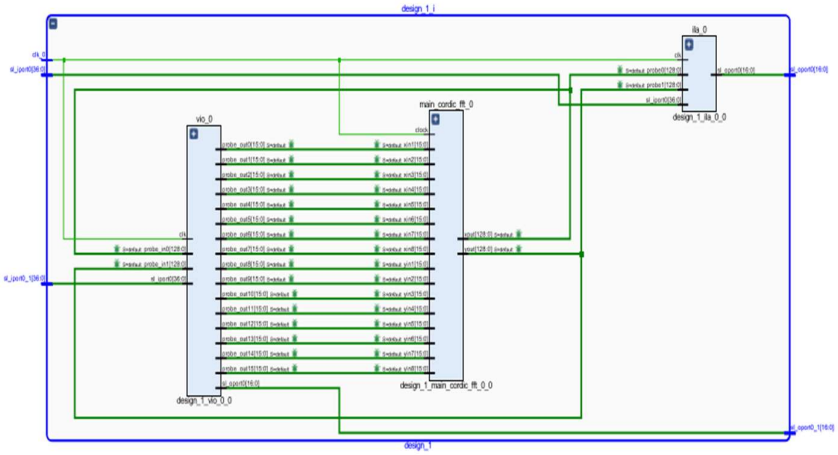
Verilog code for FFT:

```
/*This Main Module takes 8 Real Inputs and 8 Imaginary Inputs each of 16 bits and
Calculate 8-point Fast Fourier Transform and store Output in 16 Different registers each of 16 bits (8 for Real Part and 8 for Imaginary Output), clock we will take from FPGA Board itself
 we have created 12 instances of butterfly module as per DIT FFT algorithm,
and 5 Instances for Multiplication of Butterfly Results with Twiddle FACTOR.
For angle of Each Twiddle factor there is a Lookup Table Already Declared as (angle_lut)
*/
`timescale 1ns / 1ps
module main_cordic_fft(xin1,xin2,xin3,xin4,xin5,xin6,xin7,xin8,yin1,yin2,yin3,yin4,yin5,yin6,yin7,yin8,
xout1,yout1,clock,,xout2,yout2,xout3,yout3,xout4,yout4,xout5,yout5,xout6,yout6,xout7,yout7,xout8,yout8);

        input signed [15:0] xin1,xin2,xin3,xin4,xin5,xin6,xin7,xin8;
        input signed [15:0] yin1,yin2,yin3,yin4,yin5,yin6,yin7,yin8;
        output signed [15:0] xout1,xout2,xout3,xout4,xout5,xout6,xout7,xout8;
        output signed [15:0] yout1,yout2,yout3,yout4,yout5,yout6,yout7,yout8;
    input clock;

wire signed [31:0] angle_lut [0:7];

assign angle_lut[0] = 'b00000000000000000000000000000000;
assign angle_lut[1] = 'b11100000000000000000000000000000;
assign angle_lut[2] = 'b11000000000000000000000000000000;
assign angle_lut[3] = 'b10100000000000000000000000000000;

wire signed[15:0] xtemp_in[0:7],ytemp_in[0:7];
```

```
assign
{xtemp_in[0],xtemp_in[1],xtemp_in[2],xtemp_in[3],xtemp_in[4],xtemp_in[5],xtemp_in[6],
xtemp_in[7]} = {xin1,xin2,xin3,xin4,xin5,xin6,xin7,xin8};

assign
{ytemp_in[0],ytemp_in[1],ytemp_in[2],ytemp_in[3],ytemp_in[4],ytemp_in[5],ytemp_in[6],
ytemp_in[7]} = {yin1,yin2,yin3,yin4,yin5,yin6,yin7,yin8};


wire                                                    signed[15:0]
xtemp1[0:7],ytemp1[0:7],xtemp2[0:7],ytemp2[0:7],xtemp3[0:7],ytemp3[0:7],xtemp_1[0:7],
ytemp_1[0:7],xtemp_2[0:7],ytemp_2[0:7];


        butterfly
b4(.clock(clock),.x1(xtemp_in[0]),.y1(ytemp_in[0]),.x2(xtemp_in[4]),.y2(ytemp_in[4]),.xou
t1(xtemp1[0]),.yout1(ytemp1[0]),.xout2(xtemp1[1]),.yout2(ytemp1[1]));

        butterfly
b5(.clock(clock),.x1(xtemp_in[2]),.y1(ytemp_in[2]),.x2(xtemp_in[6]),.y2(ytemp_in[6]),.xou
t1(xtemp1[2]),.yout1(ytemp1[2]),.xout2(xtemp1[3]),.yout2(ytemp1[3]));

    butterfly
b6(.clock(clock),.x1(xtemp_in[1]),.y1(ytemp_in[1]),.x2(xtemp_in[5]),.y2(ytemp_in[5]),.xou
t1(xtemp1[4]),.yout1(ytemp1[4]),.xout2(xtemp1[5]),.yout2(ytemp1[5]));

        butterfly
b7(.clock(clock),.x1(xtemp_in[3]),.y1(ytemp_in[3]),.x2(xtemp_in[7]),.y2(ytemp_in[7]),.xou
t1(xtemp1[6]),.yout1(ytemp1[6]),.xout2(xtemp1[7]),.yout2(ytemp1[7]));

    cordic c7(clock,angle_lut[2],xtemp1[3],ytemp1[3],xtemp_1[3],ytemp_1[3]);

    cordic c8(clock,angle_lut[2],xtemp1[7],ytemp1[7],xtemp_1[7],ytemp_1[7]);

        butterfly
b8(.clock(clock),.x1(xtemp1[0]),.y1(ytemp1[0]),.x2(xtemp1[2]),.y2(ytemp1[2]),.xout1(xtem
p2[0]),.yout1(ytemp2[0]),.xout2(xtemp2[2]),.yout2(ytemp2[2]));

        butterfly
b9(.clock(clock),.x1(xtemp1[1]),.y1(ytemp1[1]),.x2(xtemp_1[3]),.y2(ytemp_1[3]),.xout1(xt
emp2[1]),.yout1(ytemp2[1]),.xout2(xtemp2[3]),.yout2(ytemp2[3]));

    butterfly
b10(.clock(clock),.x1(xtemp1[4]),.y1(ytemp1[4]),.x2(xtemp1[6]),.y2(ytemp1[6]),.xout1(xte
mp2[4]),.yout1(ytemp2[4]),.xout2(xtemp2[6]), .yout2(ytemp2[6]));

        butterfly
b11(.clock(clock),.x1(xtemp1[5]),.y1(ytemp1[5]),.x2(xtemp_1[7]),.y2(ytemp_1[7]),.xout1(x
temp2[5]),.yout1(ytemp2[5]),.xout2(xtemp2[7]),.yout2(ytemp2[7]));
```

```
    cordic c9(clock,angle_lut[1],xtemp2[5],ytemp2[5],xtemp_2[5],ytemp_2[5]);

    cordic c10(clock,angle_lut[2],xtemp2[6],ytemp2[6],xtemp_2[6],ytemp_2[6]);

    cordic c11(clock,angle_lut[3],xtemp2[7],ytemp2[7],xtemp_2[7],ytemp_2[7]);

        butterfly
b12(.clock(clock),.x1(xtemp2[0]),.y1(ytemp2[0]),.x2(xtemp2[4]),.y2(ytemp2[4]),.xout1(xte
mp3[0]),.yout1(ytemp3[0]),.xout2(xtemp3[4]),.yout2(ytemp3[4]));

        butterfly
b13(.clock(clock),.x1(xtemp2[1]),.y1(ytemp2[1]),.x2(xtemp_2[5]),.y2(ytemp_2[5]),.xout1(x
temp3[1]),.yout1(ytemp3[1]),.xout2(xtemp3[5]),.yout2(ytemp3[5]));

    butterfly
b14(.clock(clock),.x1(xtemp2[2]),.y1(ytemp2[2]),.x2(xtemp_2[6]),.y2(ytemp_2[6]),.xout1(x
temp3[2]),.yout1(ytemp3[2]),.xout2(xtemp3[6]),.yout2(ytemp3[6]));

        butterfly
b15(.clock(clock),.x1(xtemp2[3]),.y1(ytemp2[3]),.x2(xtemp_2[7]),.y2(ytemp_2[7]),.xout1(x
temp3[3]),.yout1(ytemp3[3]),.xout2(xtemp3[7]),.yout2(ytemp3[7]));


                assign xout1 = xtemp3[0];
                assign xout2 = xtemp3[1];
                assign xout3 = xtemp3[2];
                assign xout4 = xtemp3[3];
                assign xout5 = xtemp3[4];
                assign xout6 = xtemp3[5];
                assign xout7 = xtemp3[6];
                assign xout8 = xtemp3[7];
                assign yout1 = ytemp3[0];
                assign yout2 = ytemp3[1];
                assign yout3 = ytemp3[2];
                assign yout4 = ytemp3[3];
                assign yout5 = ytemp3[4];
                assign yout6 = ytemp3[5];
                assign yout7 = ytemp3[6];
                assign yout8 = ytemp3[7];
endmodule
```

Schematic Diagram :



Code for Butterfly:

/* Simple Butterfly Module to Calculate 2 Radix Which were used in DIT/DIF FFT algorithms */

```
`timescale 1ns / 1ps
module butterfly(
        input clock,
        input signed [15:0] x1,y1,x2,y2,
        output signed [15:0] xout1,yout1,xout2,yout2);
    assign xout1 = x1+x2;
    assign yout1 = y1+y2;

    assign xout2 = x1-x2;
    assign yout2 = y1-y2;
endmodule
```

Code for Cordic :

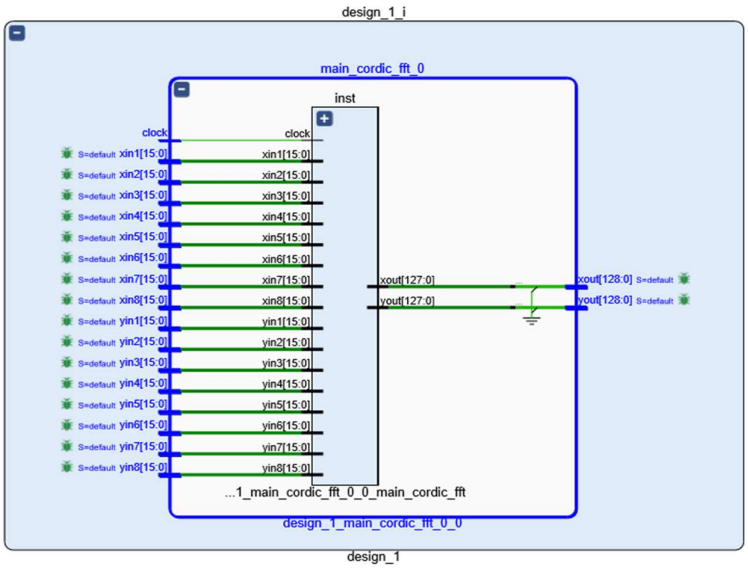
```
/*CORDIC Module will be Used to calculate value of twiddle Factor which have been used in FFT,
as a Input it takes two 16 bit Number i.e Xin1 & Yin1 one for Real part and one for Imaginary Part
and one input is theta (angle)

It will Give Two Output of 16 bit one for real part and another for Imaginary as Follows
    Xout = Xin1*cos(angle)+Xin1*sin(angle);  // Real Part
    Yout = Yin1*cos(angle)+Yin1*sin(angle);  // Imaginary Part
*/

`timescale 1 ns/100 ps

module cordic(clock, angle, Xin1, Yin1, Xout, Yout);

  parameter c_parameter = 16;
  localparam STG = c_parameter ;

  input  clock;
  input  signed  [31:0] angle;
  input  signed  [c_parameter -1:0] Xin1;
  input  signed  [c_parameter -1:0] Yin1;
  output signed  [c_parameter :0] Xout;
  output signed  [c_parameter :0] Yout;
  wire signed [c_parameter -1:0] Xin;
  wire signed [c_parameter -1:0] Yin;
   assign Xin = (Xin1>>>1)+(Xin1>>>4)+(Xin1>>>5)+(Xin1>>>6);
   assign  Yin = (Yin1>>>1)+(Yin1>>>4)+(Yin1>>>5)+(Yin1>>>6);
  wire signed [31:0] atan_table [0:30];
  assign atan_table[00] = 32'b00100000000000000000000000000000;
```

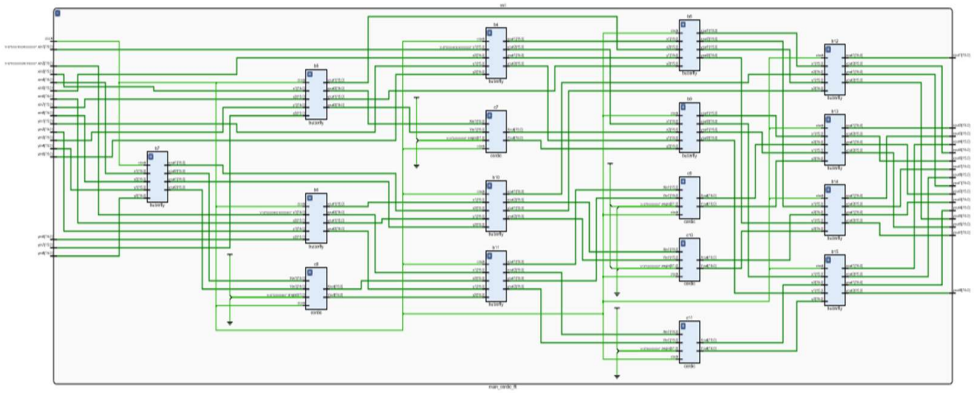
```
assign atan_table[01] = 32'b00010010111001000000010100011101;
assign atan_table[02] = 32'b00001001111111011001110000101101 1;
assign atan_table[03] = 32'b00000101000100010001000111010100;
assign atan_table[04] = 32'b00000010100010110000110101000011;
assign atan_table[05] = 32'b00000001010001011101011111100001;
assign atan_table[06] = 32'b00000000101000101111011000011110;
assign atan_table[07] = 32'b00000000010100010111110001010101;
assign atan_table[08] = 32'b00000000001010001011111001010011;
assign atan_table[09] = 32'b00000000000101000101111100101110;
assign atan_table[10] = 32'b00000000000010100010111110011000;
assign atan_table[11] = 32'b00000000000001010001011111001100;
assign atan_table[12] = 32'b00000000000000101000101111100110;
assign atan_table[13] = 32'b00000000000000010100010111110011;
assign atan_table[14] = 32'b00000000000000001010001011111001;
assign atan_table[15] = 32'b00000000000000000101000101111101;
assign atan_table[16] = 32'b00000000000000000010100010111110;
assign atan_table[17] = 32'b00000000000000000001010001011111;
assign atan_table[18] = 32'b00000000000000000000101000101111;
assign atan_table[19] = 32'b00000000000000000000010100011000;
assign atan_table[20] = 32'b00000000000000000000001010001100;
assign atan_table[21] = 32'b00000000000000000000000101000110;
assign atan_table[22] = 32'b00000000000000000000000010100011;
assign atan_table[23] = 32'b00000000000000000000000001010001;
assign atan_table[24] = 32'b00000000000000000000000000101000;
assign atan_table[25] = 32'b00000000000000000000000000010100;
assign atan_table[26] = 32'b00000000000000000000000000001010;
assign atan_table[27] = 32'b00000000000000000000000000000101;
assign atan_table[28] = 32'b00000000000000000000000000000010;
assign atan_table[29] = 32'b00000000000000000000000000000001;
assign atan_table[30] = 32'b00000000000000000000000000000000;
```

```
reg signed [c_parameter :0] X [0:STG-1];
reg signed [c_parameter :0] Y [0:STG-1];
reg signed    [31:0] Z [0:STG-1];
wire [1:0] quadrant;
assign quadrant = angle[31:30];

always @(posedge clock)
begin
  case (quadrant)
    2'b00,
    2'b11:
    begin
      X[0] <= Xin;
      Y[0] <= Yin;
      Z[0] <= angle;
    end

    2'b01:
    begin
      X[0] <= -Yin;
      Y[0] <= Xin;
      Z[0] <= {2'b00,angle[29:0]};
    end

    2'b10:
    begin
      X[0] <= Yin;
      Y[0] <= -Xin;
      Z[0] <= {2'b11,angle[29:0]};
    end
```

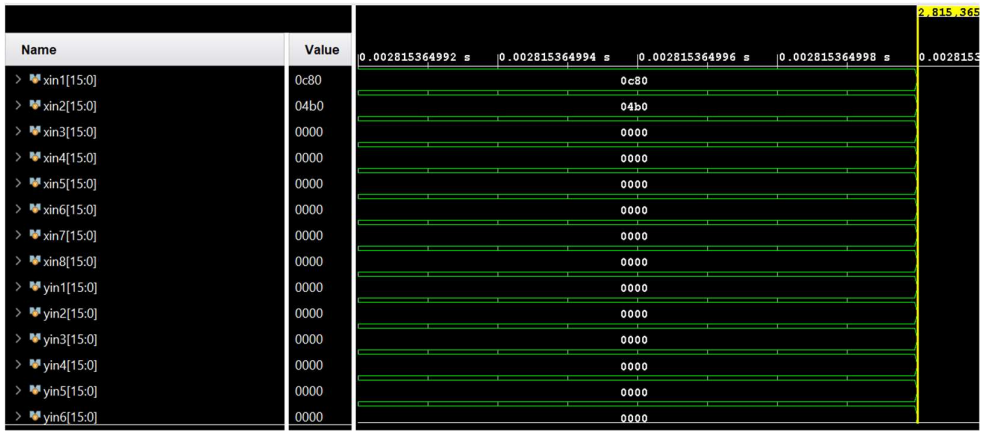
```
    endcase
  end

  genvar i;

  generate
  for (i=0; i < (STG-1); i=i+1)
  begin: XYZ
    wire                Z_sign;
    wire signed  [c_parameter :0] X_shr, Y_shr;

    assign X_shr = X[i] >>> i;
    assign Y_shr = Y[i] >>> i;

    assign Z_sign = Z[i][31];

    always @(posedge clock)
    begin
      X[i+1] <= Z_sign ? X[i] + Y_shr         : X[i] - Y_shr;
      Y[i+1] <= Z_sign ? Y[i] - X_shr         : Y[i] + X_shr;
      Z[i+1] <= Z_sign ? Z[i] + atan_table[i] : Z[i] - atan_table[i];
    end
  end
  endgenerate

  assign Xout = X[STG-1];
  assign Yout = Y[STG-1];

endmodule
```

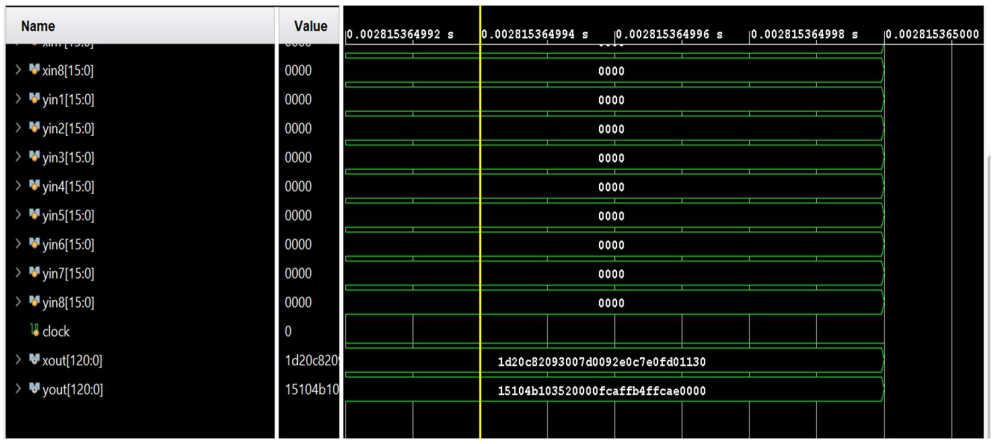
Schematic Diagram:





Output :


Name
Value
0.002815364992 s
0.002815364994 s
0.002815364996 s
0.002815364998 s
2,815,365
xin1[15:0] 0c80
xin2[15:0] 04b0
xin3[15:0] 0000
xin4[15:0] 0000
xin5[15:0] 0000
xin6[15:0] 0000
xin7[15:0] 0000
xin8[15:0] 0000
yin1[15:0] 0000
yin2[15:0] 0000
yin3[15:0] 0000
yin4[15:0] 0000
yin5[15:0] 0000
yin6[15:0] 0000


Name
Value
0.002815364992 s
0.002815364994 s
0.002815364996 s
0.002815364998 s
0.002815365000
xin8[15:0] 0000
yin1[15:0] 0000
yin2[15:0] 0000
yin3[15:0] 0000
yin4[15:0] 0000
yin5[15:0] 0000
yin6[15:0] 0000
yin7[15:0] 0000
yin8[15:0] 0000
clock 0
xout[120:0] 1d20c820
yout[120:0] 15104b10
1d20c82093007d0092e0c7e0fd01130
15104b103520000fcaffb4ffcae0000

Code for Seven Segment:

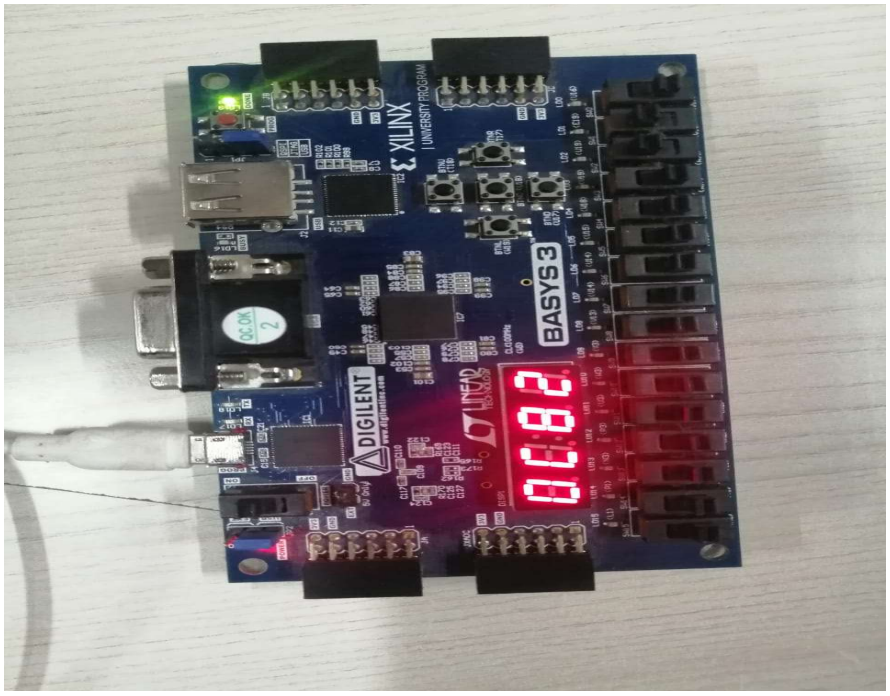
```
/* This Module we have used to display The Output to seven-segment Display, it takes a 4 bit input from the slide switches

which we already have on BASYS3 FPGA Board and This Input will be used to Display Output corresponding to that input Combination.

We have Created an instance of Our main module (main_cordic_fft) and give input with explicit association and store output

to an array of eight 16 bit registers and use that registers to Display Output on Seven-Segment Display

*/


module Seven_segment_LED_Display_Controller(
    input clock_100Mhz,
    input reset,
    input[3:0] w,
    output reg [3:0] Anode_Activate,
    output reg [6:0] LED_out
    );
    reg [15:0] displayed_number1;
    wire [15:0] displayed_number;
    reg [15:0] LED_BCD;
    reg [19:0] refresh_counter;
    wire [1:0] LED_activating_counter;
    wire signed [15:0] xout [7:0];
    wire signed [15:0] yout [7:0];




 main_cordic_fft uut
(.xin1(3200),.xin2(1200),.xin3(0),.xin4(0),.xin5(0),.xin6(0),.xin7(0),.xin8(0),

.yin1(0),.yin2(0),.yin3(0),.yin4(0),.yin5(0),.yin6(0),.yin7(0),.yin8(0),
```

```
 .xout1(xout[0]),.xout2(xout[1]),.xout3(xout[2]),.xout4(xout[3]),.xout5(xout[4]),.xout6(xout[
 5]),.xout7(xout[6]),.xout8(xout[7]),


 .yout1(yout[0]),.yout2(yout[1]),.yout3(yout[2]),.yout4(yout[3]),.yout5(yout[4]),.yout6(yout[
 5]),.yout7(yout[6]),.yout8(yout[7]),
.clock(clock_100Mhz)
        );
    always @(*)
    begin
       if(w==4'b0000)
         displayed_number1 <= xout[0];
       else if (w==4'b0001)
         displayed_number1 <= xout[1];
       else if (w==4'b0010)
         displayed_number1 <= xout[2];
       else if (w==4'b0011)
         displayed_number1 <= xout[3];
       else if (w==4'b0100)
         displayed_number1 <= xout[4];
       else if (w==4'b0101)
         displayed_number1 <= xout[5];
       else if (w==4'b0110)
         displayed_number1 <= xout[6];
       else if (w==4'b0111)
         displayed_number1 <= xout[7];
       else if (w==4'b1000)
         displayed_number1 <= yout[0];
       else if (w==4'b1001)
         displayed_number1 <= yout[1];
       else if (w==4'b1010)
         displayed_number1 <= yout[2];
```

```
    else if (w==4'b1011)
      displayed_number1 <= yout[3];
    else if (w==4'b1100)
      displayed_number1 <= yout[4];
    else if (w==4'b1101)
      displayed_number1 <= yout[5];
    else if (w==4'b1110)
      displayed_number1 <= yout[6];
    else if (w==4'b1111)
      displayed_number1 <= yout[7];
 end

 assign displayed_number = displayed_number1;

 always @(posedge clock_100Mhz or posedge reset)
 begin
    if(reset==1)
       refresh_counter <= 0;
    else
       refresh_counter <= refresh_counter + 1;
 end
assign LED_activating_counter = refresh_counter[19:18];
 always @(*)
 begin
   case(LED_activating_counter)
    2'b00: begin
       Anode_Activate = 4'b0111;
       LED_BCD = displayed_number[15:12];
         end
    2'b01: begin
```

```
    Anode_Activate = 4'b1011;
    LED_BCD = displayed_number[11:8];
     end
2'b10: begin
    Anode_Activate = 4'b1101;
    LED_BCD = displayed_number[7:4];
     end
2'b11: begin
    Anode_Activate = 4'b1110;
    LED_BCD = displayed_number[3:0];
      end
endcase

case(LED_BCD)
4'b0000: LED_out = 7'b0000001; // "0"
4'b0001: LED_out = 7'b1001111; // "1"
4'b0010: LED_out = 7'b0010010; // "2"
4'b0011: LED_out = 7'b0000110; // "3"
4'b0100: LED_out = 7'b1001100; // "4"
4'b0101: LED_out = 7'b0100100; // "5"
4'b0110: LED_out = 7'b0100000; // "6"
4'b0111: LED_out = 7'b0001111; // "7"
4'b1000: LED_out = 7'b0000000; // "8"
4'b1001: LED_out = 7'b0000100; // "9"
4'b1010: LED_out = 7'b0000010; //"a"
4'b1011: LED_out = 7'b1100000;  //"b"
4'b1100: LED_out = 7'b0110001;  //"c"
4'b1101: LED_out <= 7'b1000010;  //"d"
4'b1110: LED_out <= 7'b0110000;  //"e"
4'b1111: LED_out <= 7'b0111000;  //"f"
```

```
        default: LED_out = 7'b0000001; // "0"
        endcase
    end
endmodule
```



Conclusion:

In this project, we have successfully simulated Verilog code and designed VIO for calculating FFT using CORDIC algorithm. Output is shown on seven segment display successfully with minimal error of approx. 0.25 % and accuracy of 99.75%.

Future Scope:

Precision of FFT implementation can be further improved by providing floating point input values (both single and double precision floating point). In future high-performance parallel computers, improvements in floating-point performance are likely to continue to outpace improvements in communication bandwidth. Therefore, important algorithms for the future may trade off arithmetic for reduced communication.