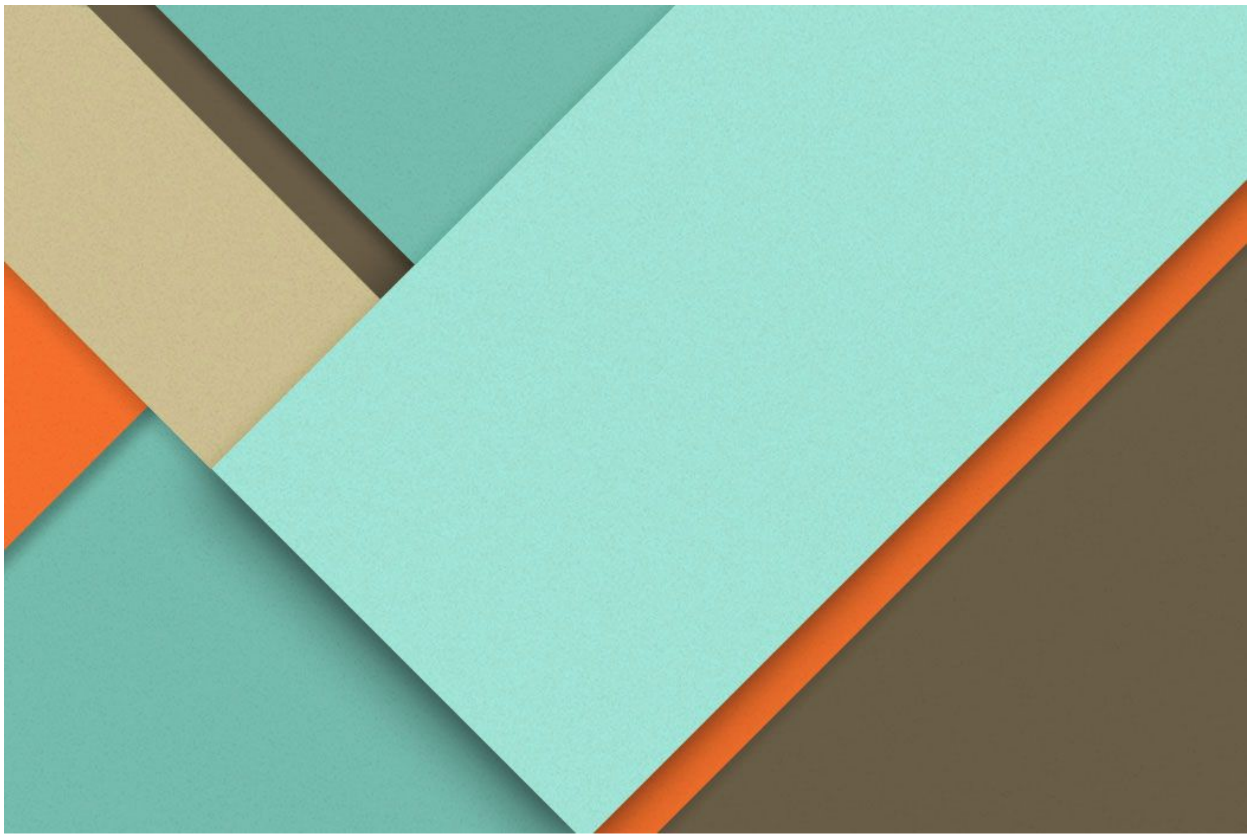


Hyperbolic Vectoring Cordic Algorithm to find square root

07.04.2019

GROUP - 8

Lagadapati Vinay(16EC10032)
Md Danish Imam(16EC10036)

Objective

Derive the architecture of a sequential CORDIC (Coordinate Rotation Digital Computer) processor working in Hyperbolic Vectoring Mode for computing square root of a Real Number. The architecture has 20 bit input data which iteratively passes through the stages of the architecture. The user given initial input data value is $X_0 = R + 1 \; \& \; Y_0 = R - 1$, where R is our input real number. Thus, after iteratively passing through the architecture 16 times the results are as follows: $X_{16} = 2K_h\sqrt{R} \; \& \; Y_{16} \to 0$.

Write the Verilog Code for the above sequential CORDIC (Coordinate Rotation Digital Computer) processor working in hyperbolic vectoring Mode which accepts 20 bit input $X_0 \; \& \; Y_0$. Demonstrate the functionality of the sequential CORDIC processor with exhaustive post route simulation outputs for a varied possible set of given input value $X_0 = R + 1 \; \& \; Y_0 = R - 1$ that leads to computation of square root of R. Follow a structural style of Verilog coding

Description of Algorithm

All of the trigonometric functions can be computed or derived from functions using vector rotations. Vector rotation can also be used from polar to rectangular and rectangular to polar conversions, for vector magnitude, and as a building block in certain transforms such as DFT and DCT. The CORDIC algorithm provides an iterative method of performing vector rotations by arbitrary angles using only shifts and adds. The algorithm is derived from the general rotation transform.

$$\begin{aligned} x' &= xCos\phi - ySin\phi \\ y' &= xSin\phi + yCos\phi \end{aligned}$$

This rotates a vector in a Cartesian plane by the angle ϕ.
These can be rearranged so that

$$\begin{aligned} x' &= Cos\phi(x - ytan\phi) \\ y' &= Cos\phi(xTan\phi + y) \end{aligned}$$

From these equations we can observe that if the rotation angles are restricted so that $Tan\phi = \pm 2^{-i}$, the multiplication by the tangent term is reduced to simple shift operation.

Arbitrary angles of rotation are obtainable by performing a series of successively smaller elementary operations. If the decision at each iteration i, is which direction to rotate rather than whether or not to rotate, then the $Cos\delta_i$ term becomes a constant because $Cos\delta_i = Cos(-\delta_i)$. The iterative rotation can now be expressed as:

$$x_{i+1} = Ki(x_i - d_i + 2^{-i} * y_i)$$
$$y_{i+1} = Ki(y_i - d_i + 2^{-i} * x_i)$$

where:

$$K_i = Cos(Tan^{-1}2^{-i}) = \frac{1}{\sqrt{1+2^{-2i}}}$$
$$d_i = \pm 1$$

Removing the scale constant from the iterative equations yields a shift add algorithm for vector rotation. The product of the $K_i's$ can be applied elsewhere in the system or treated as a part of the system processing gain. The product approaches 0.6073 as the number of iterations goes to infinity. Therefore, the rotation algorithm has a gain An of approximately 1.647. The exact gain depends on the no of iterations and obeys the relation

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations. That sequence can be represented by a decision vector. The set of all possible decision vectors is an angular measurement system based on binary arctangents. Conversions between the angular systems and any other can be accomplished using a look up. A better conversion method uses an additional adder-subtractor that accumulate the elementary rotation angles at each iteration. The elementary angles can be expressed in any convenient angular unit. Those angular values are supplied by a small look up table or are hardwired depending on the application. The angle accumulator adds a third difference equation to the CORDIC algorithm:

$$z_{i+1} = z_i - d_i * Tan^{-1}2^{-i}$$

Obviously, in cases where the angle is useful in the arc tangent base, this extra element is not needed. The CORDIC rotator is normally operated in one of two modes. The first called rotation in which we rotate the input vector by a specified angle. The second mode called vectoring in which we rotate the input vector to the x axis while recording the angle required to make that rotation.

Rotation Method

In rotation mode, the angle accumulator is initialized with the desired rotation angle. The rotation decision at each iteration is made to diminish the magnitude of the residual angle in the angle accumulator. The decision at each iteration is therefore based on the signs of the residual angle after each step. Naturally, if the input angle is already expressed in the binary arctangent base, the angle accumulator may be eliminated. For rotation mode, the CORDIC equations are:

$$x_{i+1} = x_i - d_i * 2^{-i} * y_i$$
$$y_{i+1} = y_i + d_i * 2^{-i} * x_i$$
$$z_{i+1} = z_i - d_i * Tan^{-1}(2^{-i})$$

where

$$d_i = -1 \ if \ z_i < 0, +1 \ otherwise$$

which provides the following results:

$$x_n = A_n(x_0 Cos\phi - y_0 Sin\phi)$$
$$y_n = A_n(x_0 Sin\phi + y_0 Cos\phi)$$
$$z_n = 0$$

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

Vector Method

In the vectoring mode the CORDIC vector rotates the input vector through whatever angle is necessary to align the result vector with the x axis. The result of the vectoring operation is a rotation angle and the scaled magnitude of the original vector (the x component of the result. The vectoring function works by seeking to minimize the y component of the residual vector at each rotation. The sign of the residual y component is used to determine which direction is to rotate next. If the angle accumulator is initialized with zero, it will contain the traversed angle at the end of the iterations. In vectoring mode the CORDIC equations are:

$$x_{i+1} = x_i - d_i * 2^{-i} * y_i$$
$$y_{i+1} = y_i + d_i * 2^{-i} * x_i$$
$$z_{i+1} = z_i - d_i * Tan^{-1}(2^{-i})$$

where

$$d_i =+ 1 \; if \; y_i < 0, - 1 \; otherwise$$

Then:

$$x_n = A_n * (\sqrt{x_0^2} + y_0)$$
$$y_n = 0$$
$$z_n = z_0 + Tan^{-1}(\frac{y_0}{x_0})$$
$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

The CORDIC rotation and vectoring algorithms as stated are limited to rotation angles between -90^0 and 90^0. For composite rotations larger than 90^0 an additional rotation is required. This gives the correction iteration:

$$x' =- d * y$$
$$y' = d * x$$
$$z' = x + d * \pi/2$$

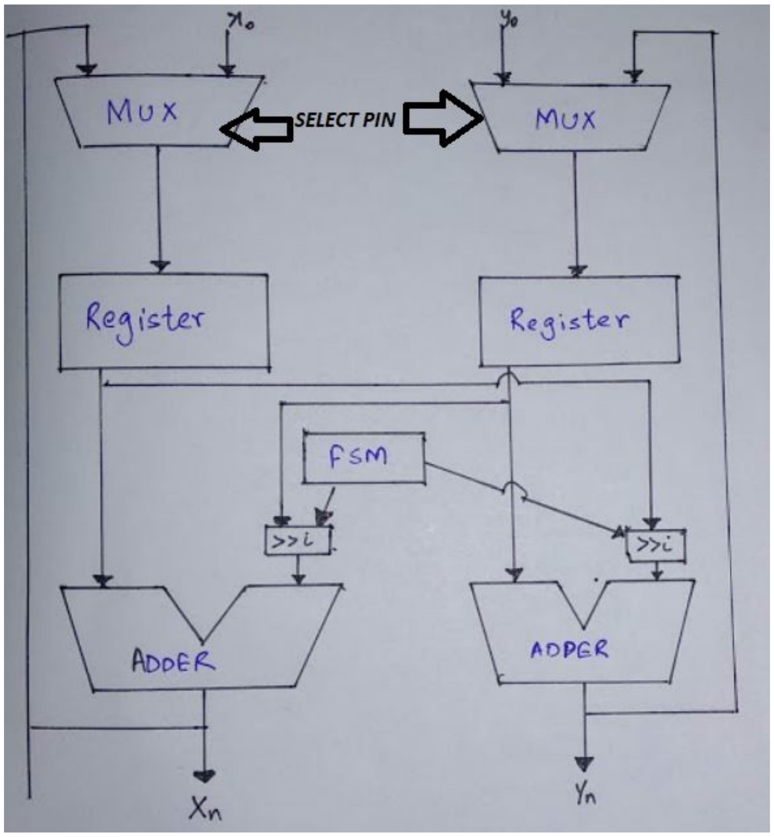
where

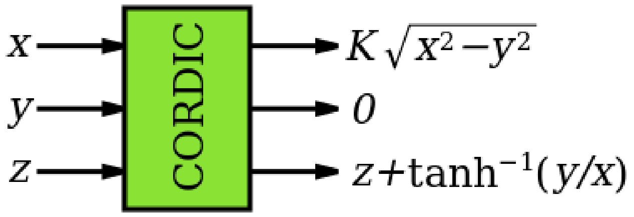
$$d = 1 \; if \; y < 0, \;\; - 1 \; otherwise$$

There is no growth for this initial rotation. Alternatively, an initial rotation of either $\pi \; or \; 0$ can be made avoiding the reassignment of the x and y components to the rotator elements. Again, there is no growth due to the initial rotation

$$x' = d * x$$
$$y' = d * y$$
$$z' = z \; if \; d = 1, \; or \; z = \pi \; if \; d =- 1$$
$$d =- 1 \; if \; x < 0, \;\; + 1 \; otherwise$$

Hardware architecture and implementation





$$x_{i+1} = x_i - \mu d_i y_i 2^{-i}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i}$$

$$z_{i+1} = z_i - d_i \alpha_i$$

$$d_i = -\operatorname{sgn}(x_i y_i), \quad y \to 0$$

μ = -1

$\alpha_i = \tanh^{-1} 2^{-i}$

Modules of Architecture

Sequence Generator

To generate 1, 2, 3, 4, 4, 5, 6..., 12, 13, 13, 14, 15.(4 & 13 repeated twice). This is done by using FSM. The sequence generated by this is fed to Barrel Shifter, so that it shifts the inputs given to the Adders accordingly.

Barrel Shifter

We have used this to divide the number by $2^i,\ i \in \textit{the above sequence}$, if the dividend is a positive number, then we need to right shift the dividend by i, and append $0's$ in the beginning, and if the dividend is negative then right shift by i and append $1's$ in the beginning.

Ripple Carry Adder

Depending on sign of Y_i addition or subtraction of inputs are done. For implementing addition or subtraction ripple adder is used. Incase of addition, both the numbers are fed directly to the adder; and incase of subtraction, the 2nd number is converted to 2's complement form and fed to the adder accordingly, so that it can be used for subtraction, in the adder.

2's complement

2's complement is done in adder module. By converting it to 1's complement, by inverting all the numbers and adding 1 to it. We can implement 1's complement by passing each bit of the number to a *XOR* gate, whose other input is more significant bit of the Y_i, and 1 is added by giving it as carry input to adder.

Registers

To output of each iteration is stored in register. 20 Bit register is implemented using 20 D-flip flops.

MUX Row

We have used 20 multiplexers, such that the 20 bit number $\{X_0 \;\&\; X_i\}\ and\ \{Y_0 \;\&\; Y_i\}$ is chosen accordingly. In the 1st iteration $X_0 \;\&\; Y_0$ is chosen and for the further iterations $X_i \;\&\; Y_i$ are chosen.

MUX

We have used 20 MUX, in the above MUX Row. MUX is implemented using *AND* and *OR* gates.

D Flip-FLop

We have used 20 D Flip-Flops in each register and 5 D Flip-Flops in Sequence Generator. All D Flip-Flops are positive edge triggered.

FSM based Sequence Generator

Truth Table:

Q_5	Q_4	Q_3	Q_2	Q_1	q_5	q_4	q_3	q_2	q_1
0	0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	0	1	1
0	0	0	1	1	0	0	1	0	0
0	0	1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	1	0	1
1	0	1	0	1	1	0	1	1	0
1	0	1	1	0	1	0	1	1	1
1	0	1	1	1	1	1	0	0	0
1	1	0	0	0	1	1	0	0	1
1	1	0	0	1	1	1	0	1	0
1	1	0	1	0	1	1	0	1	1
1	1	0	1	1	1	1	1	0	0
1	1	1	0	0	1	1	1	0	1
1	1	1	0	1	0	1	1	0	1
0	1	1	0	1	0	1	1	1	0
0	1	1	1	0	0	1	1	1	1

We have used Q_5 & q_5 to detect the occurrence of two 4’s and two 13’s.
When we get the first 4(0100), which is detected when $q_5 = 0$, we make $q_5 = 1$ as shown in Truth Table, which indicates that both 4’s have occured.
For 13(1101), it’s first occurrence is detected when $q_5 = 1$, in it’s next occurrence we make $q_5 = 0$, to indicate that 13 is occurred twice.

After using K-Map for reduction of $Q_i, \ \forall i \in \{1,\ 2,\ 3,\ 4,\ 5\}$

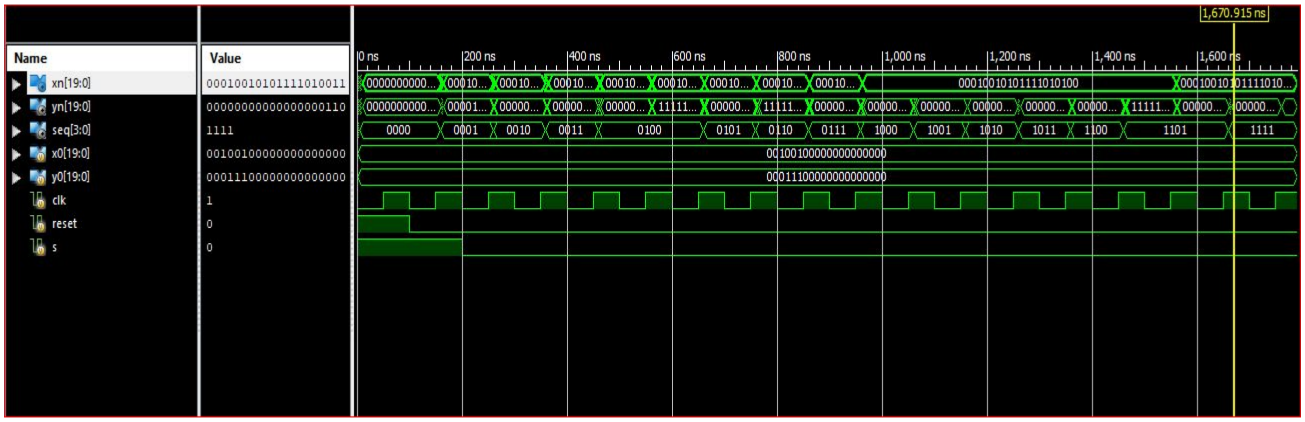
We get,

$$
\begin{gathered}
Q_1 = \overline{q_2}q_3 + \overline{q_3}q_2 + \overline{q_5}q_1 \\
Q_2 = q_2 + q_3q_4q_5 \\
Q_3 = \overline{q_4}q_3 + \overline{q_5}q_4 + \overline{q_3}q_4q_5 \\
Q_4 = \overline{q_5}q_4 + \overline{q_4}q_5\overline{q_2} + \overline{q_1}q_2 + \overline{q_3}q_5\overline{q_4} \\
Q_5 = q_1q_2q_3 + \overline{q_5}q_1 + \overline{q_5}q_4 + \overline{q_3}\,\overline{q_5}
\end{gathered}
$$

Implementation Results

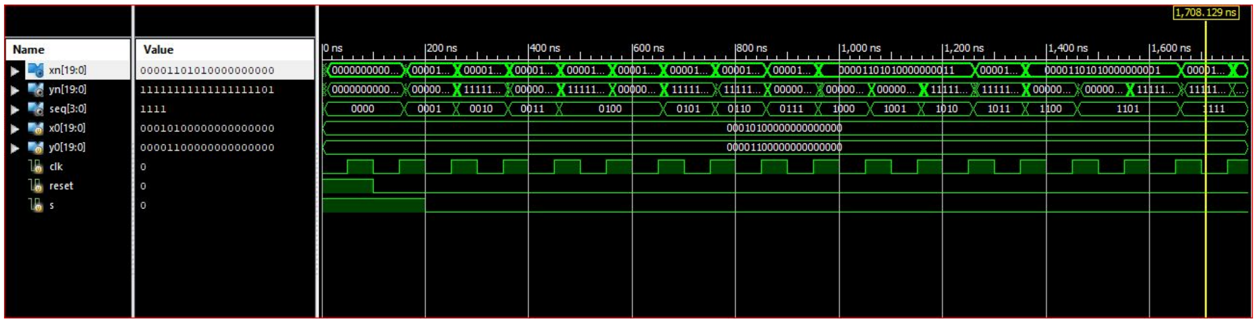
POST PLACE AND ROUTE SIMULATION OUTPUTS

Results for R = 8

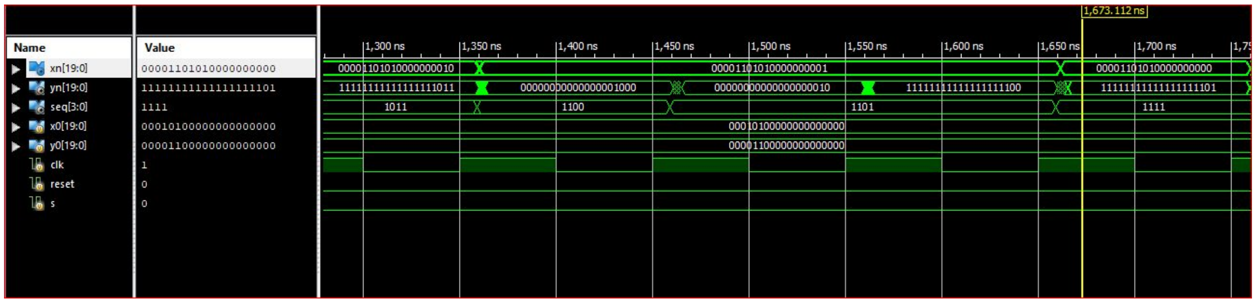


Name	Value
xn[19:0]	00010010101111010011
yn[19:0]	00000000000000000110
seq[3:0]	1111
x0[19:0]	00100100000000000000
y0[19:0]	00011100000000000000
clk	1
reset	0
s	0

Results for R = 4

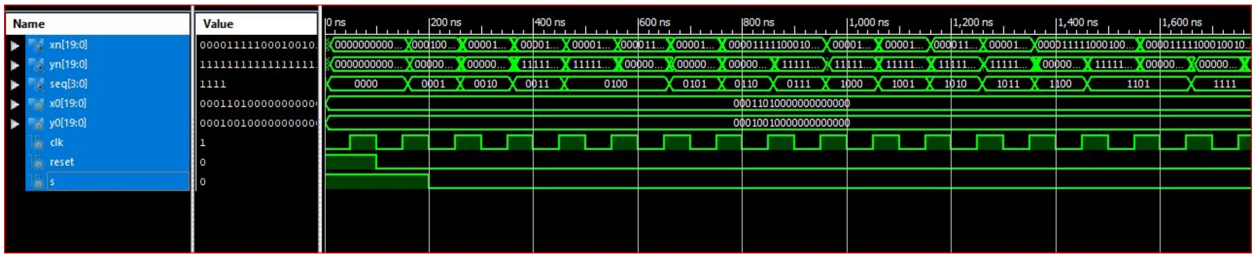


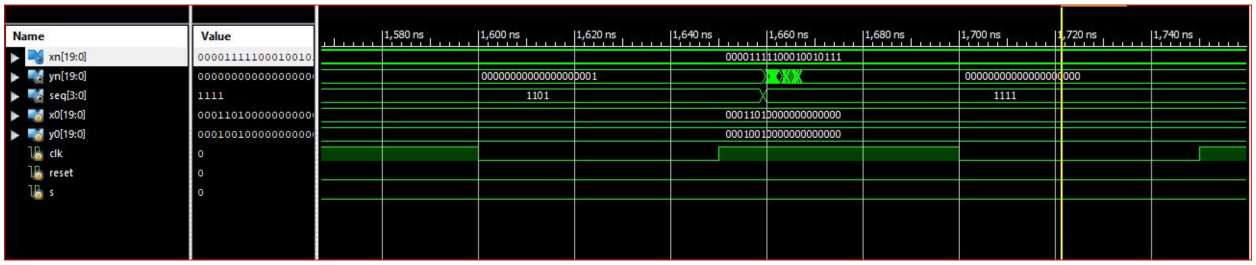
Name	Value
xn[19:0]	00001101010000000000
yn[19:0]	11111111111111111101
seq[3:0]	1111
x0[19:0]	00010100000000000000
y0[19:0]	00001100000000000000
clk	0
reset	0
s	0



Name	Value
xn[19:0]	00001101010000000000
yn[19:0]	11111111111111111101
seq[3:0]	1111
x0[19:0]	00010100000000000000
y0[19:0]	00001100000000000000
clk	1
reset	0
s	0

Results for R = 5.5





2'S complement Binary Representation

Decimal Values are represented in 20 bits. First bit is sign bit. Integral part of decimal value is denoted by next 5 bits . Remaining 14 bits denote fractional part.

SPEED

Timing Summary:

Minimum period: 4.350ns (Maximum Frequency: 229.874MHz)

Minimum input arrival time before clock: 2.496ns

Maximum output required time after clock: 6.685ns

Maximum combinational path delay: No path found

AREA

HDL Synthesis Report

<u>Macro Statistics</u>

```
# Registers                    : 45
 1-bit register                : 45
# Xors                         : 123
 1-bit xor2                    : 123
```

Device Utilization Summary:

<u>Slice Logic Utilization:</u>

```
  Number of Slice Registers:            45 out of 126,800    1%
    Number used as Flip Flops:          45
    Number used as Latches:              0
    Number used as Latch-thrus:          0
    Number used as AND/OR logics:        0
  Number of Slice LUTs:                314 out of  63,400    1%
    Number used as logic:              314 out of  63,400    1%
      Number using O6 output only:     294
      Number using O5 output only:       0
      Number using O5 and O6:           20
      Number used as ROM:                0
    Number used as Memory:               0 out of  19,000    0%
    Number used exclusively as route-thrus:    0
```

Slice Logic Distribution:

```
  Number of occupied Slices:                   119 out of  15,850    1%
  Number of LUT Flip Flop pairs used:          314
  Number with an unused Flip Flop:             274 out of     314   87%
  Number with an unused LUT:                   0 out of     314    0%
  Number of fully used LUT-FF pairs:           40 out of     314   12%
  Number of slice register sites lost to control set restrictions:           0 out of
126,800    0%
```

Challenges and Difficulties

We initially tried to implement sequence generator by modifying ripple counter such the 4 and 13 are appearing twice. We tried that using other D FLip-flop as a flag, initially the flag is set to 0, when we get the first four we set the D-Flip flop to 1; similarly in case of 13 we set the flag to 1, when we encounter 1st 13 in our counter. When our flag DFF is set to 1, we stop the clock that goes to counter for one cycle.

But we could not implement it that way because ripple counter values changes one bit at a time and we will get 0100 while the counter is changing from 0111 to 1000 and the clock gets stopped there also. So we changed the plan and implemented sequence generator using FSM.

Novelty of architecture

- The sequence generator is implemented using FSM using 5 DFFs .So that we can pass this sequence to the shifter so we can shift the inputs to the desired amount using barrel shifter.
- The area of the design is very less because we have implemented it iteratively and storing the output of each iteration in registers.
- We used 2's complement form everywhere in our design so that it takes care of negative numbers also and we can shift negative numbers also to divide it by 2^i by appending ones in front of it instead of zeroes.

Resources

https://en.wikibooks.org/wiki/Digital_Circuits/CORDIC?fbclid=IwAR0SuKZiScX6jE5TjoLIwdf8FMJnUjjgMIRVU65RmFD7h4dQAgs6Lw-LQ8U

https://en.wikipedia.org/wiki/CORDIC?fbclid=IwAR0SuKZiScX6jE5TjoLIwdf8FMJnUjjgMIRVU65RmFD7h4dQAgs6Lw-LQ8U#Vectoring_mode

https://apps.dtic.mil/dtic/tr/fulltext/u2/a242318.pdf

https://link.springer.com/content/pdf/10.1007%2F978-1-4419-9660-2_6.pdf

https://www.mit.bme.hu/system/files/oktatas/targyak/8497/CORDIC_ppt.pdf

VERILOG Modules

DFF Module

```
module dff (q,qbar,d,reset, clk);
input d,reset, clk;
output reg q;
output qbar;
not g(qbar,q);
always@(posedge clk)
        begin
        if (reset == 1) q<=1'b0;
        else q<=d;
        end
endmodule
```

MUX Module

```
module mux(
  input sel,
  input i0,
  input i1,
  output out);
wire t1,t2;
and g1(t1,i0,~sel);
and g2(t2,i1,sel);
or g3(out,t1,t2);
endmodule
```

SUM Module

```
module sum(
input a,
input b,
input c_in,
output s);

wire t1;

xor g1 (t1,a,b);

xor g2 (s,t1,c_in);

endmodule
```

Carry Module

```
module carry(
input a,
input b,
input c_in,
output c_out);

wire t2, t3, t4, t5;
and g3(t2,a,b);
and g4(t3,b,c_in);
and g5(t4,a,c_in);
or g6(t5, t2, t3);
or g7(c_out, t5, t4);

endmodule
```

Full adder Module

```
module f_adder(
input a,
input b,
input c_in,
output s,
output c_out);

sum s1(a, b, c_in, s);
carry carry1(a, b, c_in, c_out);

endmodule
```

Adder Module

```
module adder(i1,i2,out);
input [19:0]i1;
input [19:0]i2;
output [19:0]out;

wire [19:0]c;
wire [19:0]t;
wire c_in;

xnor g1(c_in,i1[19],i2[19]);
xor g2[19:0](t,c_in,i2);
f_adder F0(i1[0],t[0],c_in,out[0],c[0]);

genvar i;
generate
        for(i=1;i<20;i=i+1)
        begin:loop

                f_adder F(i1[i],t[i],c[i-1],out[i],c[i]);

        end

endgenerate
endmodule
```

Seqgen Module

```
module seqgen(b,c,d,e,reset,clk);
input reset,clk;
output b,c,d,e;
wire a,b,c,d,e,abar,bbar,cbar,dbar,ebar,w,ck;
wire t[19:1];
nand g19(w,abar,b,c,d,e);
and g20(ck,w,clk);

dff F5(a,abar,t[3],reset,ck);
dff F4(b,bbar,t[6],reset,ck);
dff F3(c,cbar,t[10],reset,ck);
dff F2(d,dbar,t[15],reset,ck);
dff F1(e,ebar,t[19],reset,ck);

and g1(t[1],a,ebar);
xor g2(t[2],b,c);
or g3(t[3],t[1],t[2]);
and g4(t[5],c,d,e);
or g5(t[6],b,t[5]);
and g6(t[7],c,dbar);
and g7(t[8],c,ebar);
and g8(t[9],cbar,d,e);
or g9(t[10],t[7],t[8],t[9]);
or g10(t[11],bbar,cbar);
and g11(t[12],d,ebar);
and g12(t[13],abar,b);
and g13(t[14],t[11],e,dbar);
or g14(t[15],t[12],t[13],t[14]);
and g15(t[16],cbar,ebar);
and g16(t[17],d,ebar);
and g17(t[18],a,b,c);
or g18(t[19],t[16],t[17],t[18],t[1]);
endmodule
```

Shifter Module

```
module shifter(sel,ip,out);
input [3:0]sel;
input [19:0]ip;
output [19:0]out;

wire [19:0]t[3:1];

mux m0[19:0](sel[0],ip,{ip[19],ip[19:1]},t[1]);
mux m1[19:0](sel[1],t[1],{{2{ip[19]}},t[1][19:2]},t[2]);
mux m2[19:0](sel[2],t[2],{{4{ip[19]}},t[2][19:4]},t[3]);
mux m3[19:0](sel[3],t[3],{{8{ip[19]}},t[3][19:8]},out);
endmodule
```

sroot Module

```
module sroot(x0,y0,clk,reset,s,xn,yn,seq);
input [19:0]x0;
input [19:0]y0;
input clk,reset,s;
output [19:0]xn;
output [19:0]yn;
output [3:0]seq;
wire [3:0]sel;
wire [19:0]t[10:1];
assign sel=seq;
mux m1[19:0](s,t[9],x0,t[1]);
mux m2[19:0](s,t[10],y0,t[2]);
dff r1[19:0](t[3],t[5],t[1],reset,clk);
dff r2[19:0](t[4],t[6],t[2],reset,clk);
shifter s1(sel,t[4],t[7]);
shifter s2(sel,t[3],t[8]);
seqgen sg(seq[3],seq[2],seq[1],seq[0],reset,clk);
adder a1(t[3],t[7],xn);
adder a2(t[4],t[8],yn);
```

```
assign t[9]=xn;
assign t[10]=yn;

endmodule
```

tb_sroot Module

```
module tb_sroot;
// Inputs
reg [19:0] x0;
reg [19:0] y0;
reg clk;
reg reset;
reg s;
// Outputs
wire [19:0] xn;
wire [19:0] yn;
wire [3:0]seq;
// Instantiate the Unit Under Test (UUT)
sroot uut (
        .x0(x0),
        .y0(y0),
        .clk(clk),
        .reset(reset),
        .s(s),
        .xn(xn),
        .yn(yn),
        .seq(seq)
);
```

```
initial begin
        // Initialize Inputs
        x0 = 20'b00010100000000000000;
        y0 = 20'b00001100000000000000;
        reset = 1'b1;
        s=1'b1;
        // Wait 100 ns for global reset to finish

        #100;

        reset=1'b0;

        #100;

        s=1'b0
        // Add stimulus here
end

initial begin
clk = 0;
        forever
                begin
                #50 clk=~clk;
                #50 clk=~clk;
                end
        end
endmodule
```