UCSD CSE 237C
Project 2: CORDIC

Naman Sehgal

nsehgal@ucsd.edu

A59005471

Apurva Salvi

asalvi@ucsd.edu

A59005541

Objective:

- To create a functional CORDIC architecture for cart2pol() function
- Analyze the performance, resource and accuracy metrics with variation in data types
- Study the LUT implementation and analyze the trends with variation in data types
- Integrate the CORDIC IP core onto the programmable logic using PYNQ

Brief Overview:

We have implemented a functional code of CORDIC and tested it extensively with a modified testbench. We changed the data type for angle and width from float to fixed type. Subsequently, we converted the multiplication operation to a shift-based operation. The information of the design and its name is given below.

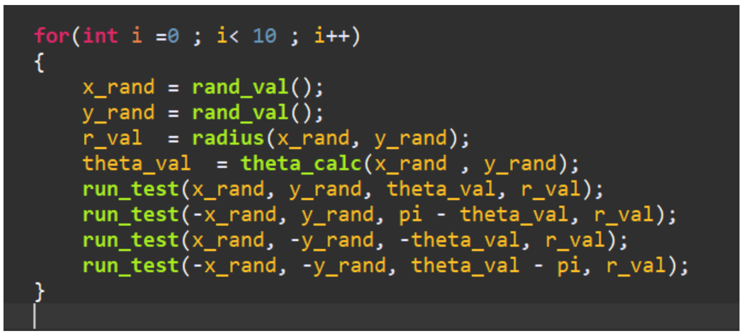
Description of the designs submitted:

Version	Description
cordic_baseline	Functional code working on original testbench and modified testbench
cordic_optimized1	Changed the data and angle to the fixed point representation ap_fixed<16,6>
cordic_optimized2	Replaced the multiplication with Kvalues[i] with shift operation
cordic_optimized3	Optimized the data width and #iterations; this is the best solution
cordic_LUT	LUT implementation for cartesian to polar converter

Testbench Alteration:

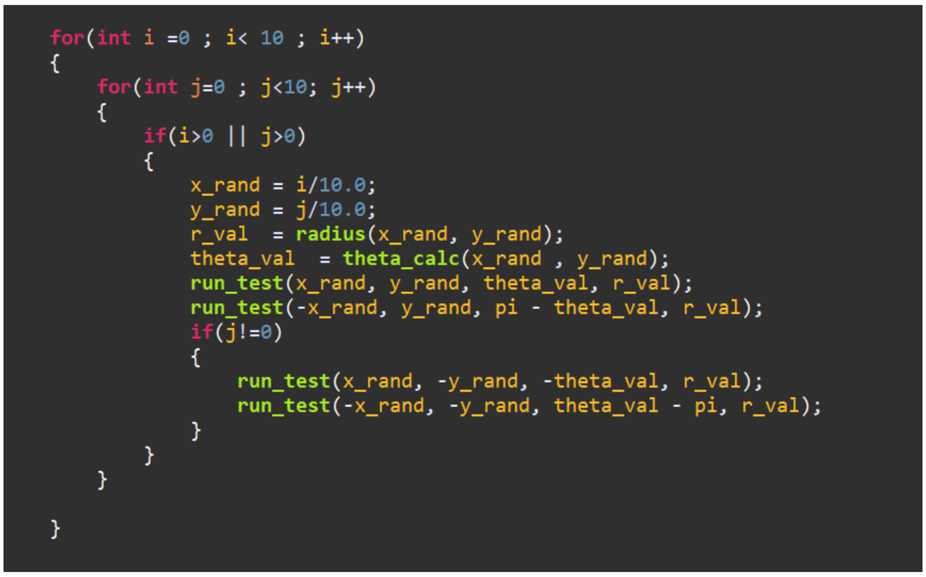
We created functions radius() and theta_calc() to return the golden radius and golden theta values respectively. These take x and y as inputs.

In testbench, we added randomized inputs for x and y, which vary from [0.001,1]. We compute the arctan and radius for these values to generate the golden vectors. Further, we transposed these golden vectors into all 4 quadrants and then passed them into the run_test() function



```
for(int i =0 ; i< 10 ; i++)
{
    x_rand = rand_val();
    y_rand = rand_val();
    r_val  = radius(x_rand, y_rand);
    theta_val  = theta_calc(x_rand , y_rand);
    run_test(x_rand, y_rand, theta_val, r_val);
    run_test(-x_rand, y_rand, pi - theta_val, r_val);
    run_test(x_rand, -y_rand, -theta_val, r_val);
    run_test(-x_rand, -y_rand, theta_val - pi, r_val);
}
```

Also, in order to increase the coverage of our testcases, we gave x and y all possible values from [0,1] with interval step of 0.1 using for loops. We created an exception for the case of x=0 and y=0.
Again, we transposed the values across 4 quadrants and passed to the run_test() function.



```
for(int i =0 ; i< 10 ; i++)
{
    for(int j=0 ; j<10; j++)
    {
        if(i>0 || j>0)
        {
            x_rand = i/10.0;
            y_rand = j/10.0;
            r_val  = radius(x_rand, y_rand);
            theta_val  = theta_calc(x_rand , y_rand);
            run_test(x_rand, y_rand, theta_val, r_val);
            run_test(-x_rand, y_rand, pi - theta_val, r_val);
            if(j!=0)
            {
                run_test(x_rand, -y_rand, -theta_val, r_val);
                run_test(-x_rand, -y_rand, theta_val - pi, r_val);
            }
        }
    }

}
```

The modified testbench with the above changes has been added to the home directory in our repository.

Questions:

Question 1: One important design parameter is the number of rotations. Change that number to numbers between 10 and 20 and describe the trends. What happens to performance? Resource usage? Accuracy of the results? Why does the accuracy stop improving after some number of iterations? Can you precisely state when that occurs?

We took a version of "cordic_optimized1" with angle_width set to ap_fixed<16,3> and data_width set to ap_fixed<16,3>. With this we varied the number of rotations to observe the trends. We had to analyze #rotations from 4 to 20 as that was required to capture the trends in RMSE(r).

Trend in Performance:
The throughput reduced with increase in number of rotations. This was due to the increasing latency with a constant clock period

	Performance		
No. of Rotations	Latency	Clock period(ns)	Throughput(MHz)
4	31	7.238	4.456764923
5	35	7.238	3.947420361
6	39	7.238	3.542556734
7	43	7.238	3.213016573
8	47	7.238	2.939568354
9	51	7.238	2.709013973
10	55	7.238	2.511994775
11	59	7.238	2.341690045
12	63	7.238	2.193011312
13	67	7.238	2.062085263
14	71	7.238	1.945911445
15	75	7.238	1.842129502
16	79	7.238	1.748857122
17	83	7.238	1.664574851
18	87	7.238	1.588042674
19	91	7.238	1.5182386
20	95	7.238	1.454312765

Trend in Resource Usage:
The resource usage had negligible (less than 0.3%) change across the entire sweep of #rotations from 10 to 20. This happened as our design is a pipelined implementation so a common set of logic gets reused every cycle.

	Utilization				
No. of Rotations	BRAM	DSP	FF	LUT	URAM
4	0	3	1040	3852	0
5	0	3	1040	3852	0
6	0	3	1040	3851	0
7	0	3	1040	3851	0
8	0	3	1042	3851	0
9	0	3	1042	3851	0
10	0	3	1042	3851	0
11	0	3	1042	3851	0
12	0	3	1042	3851	0
13	0	3	1042	3851	0
14	0	3	1042	3852	0
15	0	3	1042	3853	0
16	0	3	1044	3854	0
17	0	3	1044	3853	0
18	0	3	1044	3854	0
19	0	3	1044	3855	0
20	0	3	1044	3855	0

Trend in Accuracy:
We have taken the RMSE values as a metric for comparing the accuracy. We will answer all questions asked about accuracy separately for RMSE(θ) and RMSE(r).

	Accuracy	
No. of Rotations	RMSE(r)	RMSE(θ)
4	0.00398	0.06702
5	0.00092	0.03365
6	0.00017	0.01655
7	7.7E-05	0.00895
8	0.0001	0.0052
9	0.0001	0.00237
10	0.0001	0.00112
11	0.0001	0.00053
12	0.0001	0.00036
13	0.0001	0.00024
14	0.0001	0.0002
15	0.0001	0.0002
16	0.0001	0.0002
17	0.0001	0.0002
18	0.0001	0.0002
19	0.0001	0.0002
20	0.0001	0.0002

a.) **RMSE(θ):**
On sweeping #rotations from 20 towards 4, RMSE(θ) remains constant till we reach #rotations=13, at which point it starts degrading (i.e. becomes larger).
Reason for the trend:
We can see that atan(2^{-i}) is the entity creating changes in the value of θ in every cycle from the below equation

$$\theta[i+1] = \theta[i] \pm atan(2^{-i})$$

For small values of an angle, let's say δ, the tan follows the below equation

$$tan(\delta) = \delta$$

Conversely,

$$atan(\delta) = \delta$$

Plugging δ to 2^{-i}

$$atan(2^{-i}) = 2^{-i}$$

So, atan(2^{-i}) which is approximately equal to 2^{-i} will get quantized to 0 when our iterator i reaches a value greater than the number of fractional bits, which is 13 in our design.
For a design with data width ap_fixed<W,I> we can say that accuracy of θ saturates beyond #rotations equal to **(W-I)**

b.) **RMSE(r):** We observed that RMSE(r) did not change at all for our design when sweeping #rotations from 20 to 10. So, we extended the observation data and swept #rotation from 20 to 4, RMSE(r) remained constant till #rotations=7, at which point it became slightly better. After that it constantly degrades (i.e. becoming larger) with every subsequent reduction in #rotations.

We initially expected the iteration at which RMSE(r) saturates to be equivalent to be the same as the number of fractional bits in data. But on close analysis, we found the saturation occurring around half of the expected value (i.e. **(W-I)/2**).

Reason for trend:
The polar radius(r) is the value of x finally computed at the last iteration. Intermediate values of x can be defined by the below equation

$$x[i+1] = x[i] \pm y[i] * 2^{-i}$$

We observed that the absolute value of y is constantly decreasing with every iteration.
This stems from the fact that in CORDIC based cartesian to polar converter, the y converges towards 0 with every iteration so that radius value resides on x-axis. y[i] is decaying by a factor quite close to 2^{-i}.
So its multiplication with 2^{-i} is going to become close to 2^{-2i} (or 4^{-i}), causing a decay by a factor of 4 every cycle. We checked this by plotting values of y after i iterations and its value after multiplication with 2^{-i} and compared it to the lowest quantization in the below table.

#iteration (i.e. value of i)	Average value of y[i]	y[i]*2^-i	(<2^-13)
6	0.016217709	0.00025	No
7	0.009030891	7.1E-05	Yes
8	0.005464815	2.1E-05	Yes

Question 2: Another important design parameter is the data type of the variables. Is one data type sufficient for every variable or is it better for each variable to have a different type? Does the best data type depend on the input data? What is the best technique for the designer to determine the data type(s)?

Based on our observations that θ and r when sized equally, reach RMSE saturation at different iterations, it makes sense to size them differently.
It depends on the range of the data. For example, if the input range is reduced from [-1,1] to [-0.5,0.5], we could remove one bit from the integer side and either reduce the width or allocate that removed bit to fractional bits.
Simultaneously, if there was a requirement for higher bit precision then either float could be used or we could have increased the bit precision.
Typically, the complexity and utilization of our resources increases as we increase the data width. So, it is efficient to size our data width such that it just meets the performance.
For our design, the below table represents the widths that we experimented with. The last row with data width of <15,3> and angle_width of <13,3> is our most optimal design.

Angle width	Data width	iterations	RMSE(r)	RMSE(theta)
<16,3>	<16,3>	14	0.0001	0.00020016
<16,3>	<13,3>	13	0.0007	0.0016995
<13,3>	<16,3>	13	0.0001	0.00089778
<13,3>	<16,3>	11	0.0001	0.00089778
<13,3>	<16,3>	10	0.0001	0.00138613
<13,3>	<15,3>	11	0.00015	0.00094229

Question 3.
What is the effect of using simple operations (add and shift) in the CORDIC as opposed to multiply and divide? How does the resource usage change? Performance? Accuracy?

To implement the CORDIC function, we have to multiply terms in the equations by a factor of 2^{-i}. We have implemented this in two ways. The first utilizes the given Kvalues array and multiplies every term with the corresponding input term, as shown in Figure 3.3. The second way avoids multiplication and instead shifts the corresponding input term by i, as shown in Figure 3.4. The performance, accuracy and resource usage of both implementations are given in Tables 3.1 and 3.2

	Performance		Accuracy (RMSE)	
	Throughput	Latency (clock remains same)	R	Theta
Multiplication with Kvalue	1.7488 MHz	79	0.0001	0.00020016
Shift Operator	4.6053 MHz	30	0.00038	0.00027604

Table 3.1: Performance and Accuracy of both implementations

	DSP	FF	LUT
Multiplication with Kvalue	3	1044	3854
Shift Operator	1	957	3755

Table 3.2: Resource Usage of both implementations

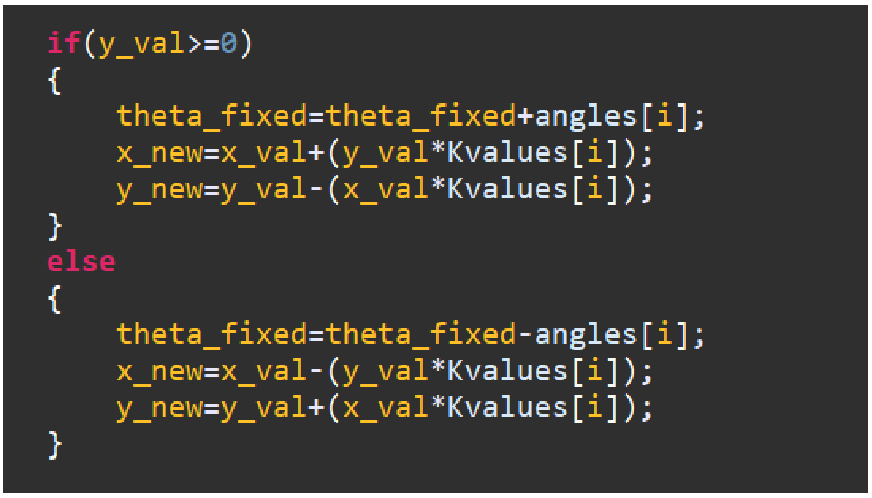
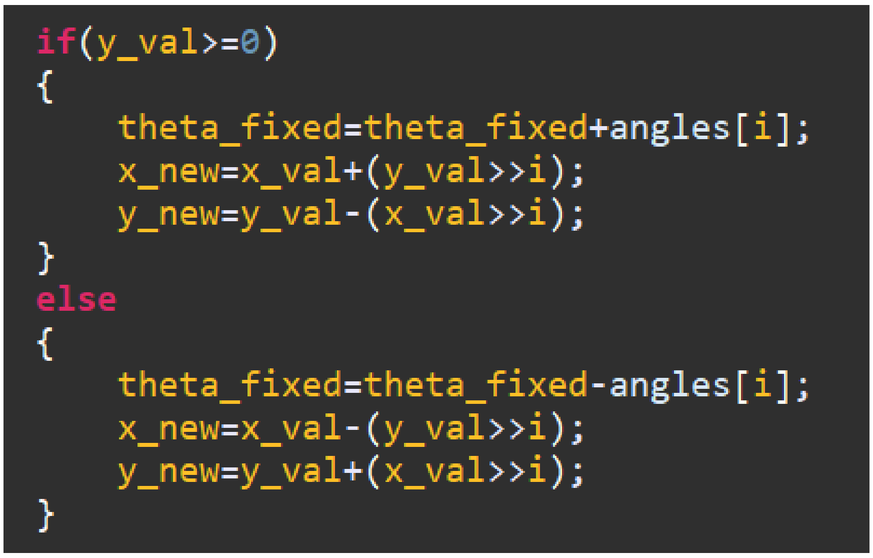

```
if(y_val>=0)
{
    theta_fixed=theta_fixed+angles[i];
    x_new=x_val+(y_val*Kvalues[i]);
    y_new=y_val-(x_val*Kvalues[i]);
}
else
{
    theta_fixed=theta_fixed-angles[i];
    x_new=x_val-(y_val*Kvalues[i]);
    y_new=y_val+(x_val*Kvalues[i]);
}
```

Figure 3.3: Multiplication with term from Kvalues

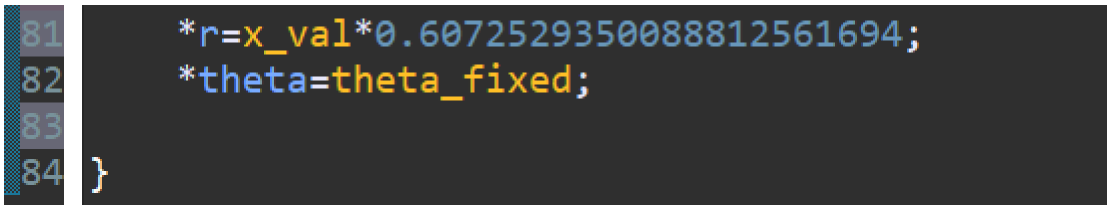


```
if(y_val>=0)
{
    theta_fixed=theta_fixed+angles[i];
    x_new=x_val+(y_val>>i);
    y_new=y_val-(x_val>>i);
}
else
{
    theta_fixed=theta_fixed-angles[i];
    x_new=x_val-(y_val>>i);
    y_new=y_val+(x_val>>i);
}
```

Figure 3.4: Using Shift Operator instead of multiplication

We can see that though accuracy is slightly reduced when we use shift operators instead of multiplication, there is a significant increase in the performance of the design as well as lesser usage of resources. We can observe that there is only 1 DSP being used, which is for the constant multiplication with the inverse of CORDIC gain, as seen in Figure 3.5.

If the design or application permits a processing gain, then the final multiplication required for gain adjustment can be removed. This will lead to further simplification of the CORDIC algorithm.



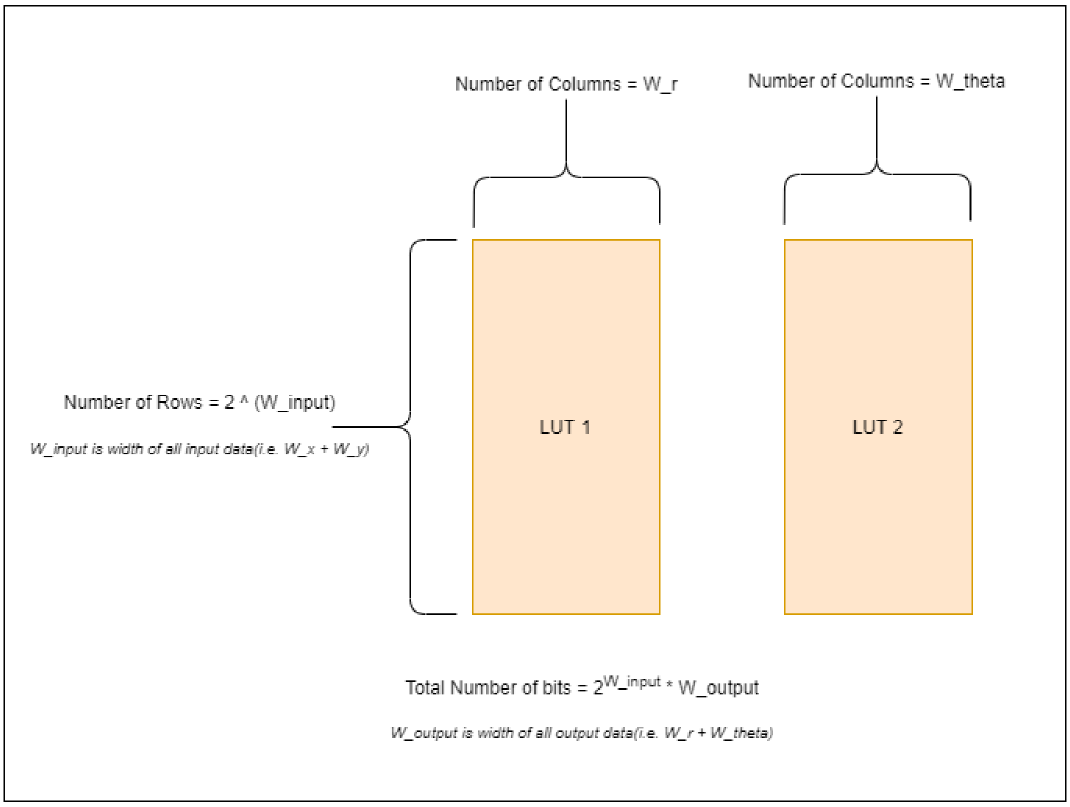
```
    *r=x_val*0.6072529350088812561694;
    *theta=theta_fixed;

}
```

Figure 3.5: Multiplication with inverse of CORDIC gain

Question 4.

These questions all refer to the lookup table (LUT) implementation of the Cartesian to Polar transformation.

a) How does the input data type affect the size of the LUT? How does the output data type affect the size of the LUT? Precisely describe the relationship between input/output data types and the number of bits required for the LUT.


Number of Columns = W_r
Number of Columns = W_theta
Number of Rows = 2 ^ (W_input)
W_input is width of all input data(i.e. W_x + W_y)
LUT 1
LUT 2
Total Number of bits = 2^W_input * W_output
W_output is width of all output data(i.e. W_r + W_theta)

The number of rows of the two lookup tables is related to the width of the input data in the following way:

$$Number\ of\ Rows\ =\ 2^{W\_input}$$

where W_input is the total width of input data types x and y

The number of columns required in both LUTs are determined by the width of the output data types R and Theta, as shown in the above diagram. Increasing the output data precision linearly increases the LUT size.

The total number of bits in the LUT is related with the input and output data types in the following manner:

$$Total\ Number\ of\ Bits\ =\ W\_output * 2^{W\_input}$$

where W_output is the total width of output data types R and Theta

b) The testbench assumes that the inputs x, y are normalized between [-1,1]. What is the minimum number of integer bits required for x and y? What is the minimal number of integer bits for the output data type R and Theta?

Variables	Minimum Integer Bits	Signed/Unsigned
x, y	2	Signed
R	1	Unsigned
Theta	3	Signed

c) Modify the number of fractional bits for the input and output data types. How does the precision of the input and output data types affect the accuracy (RMSE) results?

Comparison of variation of Input bits with output data types fixed to have 8 total bits, 3 integer bits and 5 fractional bits:

Input Bits			RMSE	
Total Bits	Integer Bits	Fractional Bits	R	Theta
9	2	7	0.023094084	0.051045734
8	2	6	0.023094084	0.051045734
7	2	5	0.023094084	0.051045734
6	2	4	0.023380082	0.531664014
5	2	3	0.038790837	0.741694272
4	2	2	0.072758213	1.021938562

As we can see, increasing the number of fractional bits leads to more accuracy till a point where it saturates. After this point (5 fractional bits), it remains constant and RMSE does not reduce (i.e. improve) further.

Comparison of variation of Output bits with input data types fixed to have 8 total bits, 2 integer bits and 6 fractional bits:

Output Bits			RMSE	
Total Bits	Integer Bits	Fractional Bits	R	Theta
11	3	8	0.004972981	0.011713494
10	3	7	0.006156081	0.012354538
9	3	6	0.011874747	0.025460193
8	3	5	0.023094084	0.051045734
7	3	4	0.045045134	0.094583899
6	3	3	0.086298309	0.17088145

Increasing the number of fractional bits in the output data types has a higher impact on the performance of the design compared to changing the precision of input data.

d) What is the performance (throughput, latency) of the LUT implementation. How does this change as the input and output data types change?

Input Data Types		Output Data Types		Latency	Throughput
Total bits	Fractional bits	Total bits	Fractional bits		
12	2	8	3	2	76.8285188
11	2	8	3	2	76.8285188
10	2	8	3	2	76.8285188
9	2	8	3	2	76.8285188
8	2	8	3	2	76.8285188
8	2	6	3	2	76.8285188
8	2	7	3	2	76.8285188
8	2	9	3	2	76.8285188
8	2	10	3	2	76.8285188
8	2	11	3	2	76.8285188

We observe that the Throughput and Latency of the LUT implementation do not change with changes in the input precision or the output precision.

e) What advantages/disadvantages of the CORDIC implementation compared to the LUT-based implementation?

The CORDIC is an efficient iterative algorithm to calculate trigonometric functions. It has a fair bit of complexity and understanding that is required for its implementation.
LUTs on the other hand are simpler to implement and require storing precomputed values into memory.

Comparison on number of cycles:
If the output can be computed across multiple cycles, then CORDIC is preferred as it typically requires multiple cycles due to its iterative shift and add. An exception would be if we unroll all the loops but that will increase the hardware utilization.
If the output is required in a single cycle, typically LUTs are preferred.

Comparison on Input Data width:
The space complexity of CORDIC increases linearly with the data width of inputs.
LUTs on the other hand increase by a factor of 2 with every bit that is added to input width.
Due to this, we prefer LUTs for small input data width and CORDIC for large data width.