WES 237C: Project 2 CORDIC and Phase Detector

Yi Song
yis057@ucsd.edu
PID# A53266440

Sebastian Nevarez
senevarez@ucsd.edu
PID# A59021322

November 3, 2023

---

1. Introduction

This report details our implementation of the CORDIC algorithm to convert Cartesian coordinates to polar coordinates as well as our optimization strategy. Additionally, section 4 of this report details our work to design a simple phase detector.

2. CORDIC

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?

Answer: Since the size of the Kvalues and angles arrays are 16, we limited our investigation to only changing the number of iterations from 10 to 16. During our testing we were able to see the performance worsen from 10 to 16 iterations with throughput going down from 0.5445MHz to 0.4191MHz respectively. In terms of resource usage, we don't see a change in the resource usage: 5 DSP, 1075 FF, 1746 LUT. This does

make sense as we are not using higher bit data types and instead are merely just increasing the time the program runs. The accuracy of our results does increase, as we would expect. Theta value RMSE accuracy improves 10 times however, R value RMSE did not improve as much. Compared with the error threshold of 0.001, the improvement stops after 11 iterations.

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)?

Answer: Keeping the error threshold of 0.001 in mind, we chose to use the ap_fixed<16,3> data type for all the variables used in our CORDIC algorithm. To get an understanding of the possible range of values in the integer and fraction part of the intermediate cos and sin values we inserted printf() statements in the for loop. This helped us decide on an appropriate size to use and maintain precision. In the future however, our team thought it would be more efficient to use different data types for different values in the loop. For example, the sigma variable in the for loop controls the direction of rotation and can only take on the values of -1 or 1. This would be more appropriately represented using an ap_int<2> data type.

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?

Answer:

1. We replaced multiplication with a cos and sin shift

```
// mytype cos_shift = current_cos * sigma * Kvalues[j];
// mytype sin_shift = current_sin * sigma * Kvalues[j];
```

Given the Kvalues are a sequence of negative integer values of power 2, we can change the multiplication into a right shift of j, the iteration index in the loop. This change maintains the same level of RMSE for R and theta while achieving higher throughput and less resource utilization.

- The throughput doubled from 2.13MHz to 4.2MHz
- DSP utilization decreased from 5 to 3
- FF decresed from 1790 to 1778
- LUT did increase slightly from 5518 to 5624

2. Additionally, we made changes to the if statement for sigma

```
// ap_int<2> sigma = (current_sin > 0) ? -1 : 1;
```

Since we know sigma can only take on values of +/- 1, we can simplify more multiplications using the following logic:

- If current_sin > 0, shift current_cos and current_sin by j and apply ~ to negate the value
- To update theta we can add or subtract angles without multiplying

See our code implementation below. By using this method we can keep the same RMSE while achieving faster throughput of 4.32MHz. FF and LUT both decreased slightly, from 1778 to 1759 and 5624 to 5551 respectively.

```
if (current_sin > 0) {
        cos_shift = ~current_cos >> j;
        sin_shift = ~current_sin >> j;
        current_theta = current_theta + angles[j];
} else {
        cos_shift = current_cos >> j;
        sin_shift = current_sin >> j;
        current_theta = current_theta - angles[j];
}
```

Question 4.1: 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 the input/output data types and the number of bits required for the LUT.

Answer: Having more bits in the input and output makes a larger LUT size, which is determined by the total size of the fixed point representation.

```
#define W 8 // Total size of fixed-point representation
#define LUT_SIZE(1 << (W << 1)) // Size of the LUT = 2^(2*W)
```

Question 4.2: The testbench assumes that the inputs x, y are normalized between [-1, 1]. What is the minimum number of the integer bits required for x and y? What is the minimal number of integer bits for the output data type R and Theta?

Answer: It will take at least 2 bits in the integer part of the ap_fixed data type to represent values between -1 and 1. From the Csim log, we can see the range of theta

and r which need at least 3 bits in the integer part of the ap_fixed data type to be properly represented.

Testbench min_theta = -3.1316, max_theta = 3.1416
Testbench min_r = 0.0000, max_r = 1.4142

Question 4.3: 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?

Answer: We fixed the total length of input and output data type to 8-bits. As previously discussed, 3-bits is the minimum number needed to represent the input and output integer part. If we go below 3-bits, we risk losing precision which will drastically increase the RMSE of theta and r to something above 1. If we go above 3-bits, we risk losing precision in the fractional part which will slightly increase the RMSE by 0.04 per additional bit allocated for integer bits. Based on this, we believe that 3-bits is the sweet spot for the CORDIC LUT.

Question 4.4: What is the performance (throughput, latency) of the LUT implementation. How does this change as the input and output data types change?

Answer: The throughput of the LUT implementation is 51.219MHz and the latency is 20ns. By fixing the total length of the LUT data type to be 8, and changing only the partition between integer and fraction, the resource allocation does not seem to change.

- 8 BRAM
- 3 FF
- 48 LUT

The throughput and latency does not seem to change either which makes sense because changing the partition only changes the precision of RMSE and changing the data length only changes the amount of memory used for LUT. None of this will affect the throughput and latency.

Question 4.5: What advantages/disadvantages of the CORDIC implementation compared to the LUT-based implementation?

Answer: The advantage of the CORDIC implementation is the precision of the RMSE. In this implementation we have about 100x smaller RMSE than in the LUT-based implementation, even when using the same ap_fixed data type - ap_fixed<16, 3>. The disadvantage of the CORDIC however, is the throughput, latency, and resource

allocation. Using a pre-computed LUT in CORDIC, we were able to achieve a much higher throughput with less register and hardware logic. In terms of number, the CORDIC implementation has a throughput of 4.32MHz whereas the LUT-based implementation has a throughput of 51.21MHz. Below we have included screenshots and tables of our outputs to compare the two implementations.

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../cordiccart2pol_test.cpp in debug mode
   Compiling ../../../../cordiccart2pol.cpp in debug mode
   Generating csim.exe
---Testing results----------------------------------
Test: x=0.8147, y=0.1269, golden theta=0.1545, golden r=0.8245, your theta=0.1531, your r=0.8251
Test: x=0.6323, y=-0.2785, golden theta=-0.4149, golden r=0.6909, your theta=-0.4160, your r=0.6913
Test: x=-0.5469, y=-0.9575, golden theta=-2.0898, golden r=1.1027, your theta=-2.0898, your r=1.1031
Test: x=-0.4854, y=0.7003, golden theta=2.1769, golden r=0.8521, your theta=2.1769, your r=0.8525
---RMS error---------------------------------------
-----------------------------------------------
   RMSE(R)           RMSE(Theta)
0.000466571567813 0.000917539931834
-----------------------------------------------
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Figure 1: CORDIC implementation RMSE

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../cordiccart2pol_test.cpp in debug mode
   Compiling ../../../../cordiccart2pol.cpp in debug mode
   Generating csim.exe
Testbench min_theta=-3.1316, max_theta=3.1416
Testbench min_r=0.0000, max_r=1.4142
   RMSE(R)           RMSE(Theta)
0.023094084113836 0.051045734435320
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Figure 2: LUT-based implementation RMSE

Implementation	RMSE (R)	RMSE (Theta)
CORDIC	0.000466571567813	0.000917539931834
LUT-based	0.023094084113836	0.051045734435320

Table 1: RMSE comparison

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.222 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cordiccart2pol				-	32	320.000	-	33	-	no	0	3	1759	5551	0
cordiccart2pol_Pipeline_VITIS_LOOP_48_1				-	13	130.000	-	13	-	no	0	0	102	364	0

Figure 3: CORDIC implementation throughput and resource utilization

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	6.508 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cordiccart2pol				-	2	20.000	-	3	-	no	8	0	3	48	0

Figure 4: LUT-based implementation throughput and resource utilization

3. PYNQ Demo

In this section we have included screenshots of our run on the PYNQ boards.

Verifying Functionality

In [11]:
```
sum_sq_r=0
sum_sq_theta=0
for i in range(NUM_SAMPLES):
    sum_sq_r =sum_sq_r+(r_error[i]*r_error[i])
    r_rmse = np.sqrt(sum_sq_r / (i+1))
    sum_sq_theta =sum_sq_theta+(theta_error[i]*theta_error[i])
    theta_rmse = np.sqrt(sum_sq_theta / (i+1))
print("Radius RMSE: ", r_rmse, "Theta RMSE:", theta_rmse)
if r_rmse<0.001 and theta_rmse<0.001:
    print("PASS")
else:
    print("FAIL")
```

```
Radius RMSE:  0.0006289146046960588 Theta RMSE: 0.0008373055356320056
PASS
```

Figure 5: Jupyter notebook execution

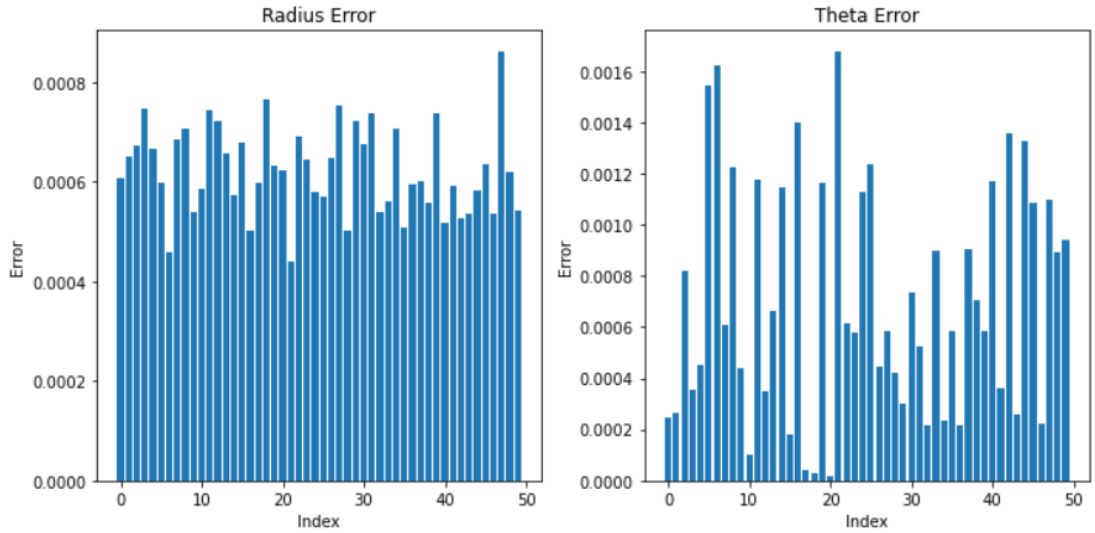

Figure 6: Demo on ZYNQ board

4. Phase Detector

Question 1: What is the throughput of your Phase Detector? How does that relate to the individual components (FIR, CORDIC, etc.)? How can you make it better?

Answer: The throughput of our phase detector is 0.42MHz, see Figure 7 below, which is about the same throughput as the baseline FIR and CORDIC implementations. One way we thought about making our phase detector better is to change the floating point data type used universally into a customized data type using ap_fixed. Additionally, since our FIR filter is 32 integers long, we can unroll the entire loop by a factor 32.

Target	Estimated	Uncertainty
10.00 ns	7.256 ns	2.70 ns

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
phasedetector				-	-	-	-	-	-	no	0	405	66501	83248	0
VITIS_LOOP_28_1				-	-	-	326	1	-	yes	-	-	-	-	-
firI1				-	160	1.600E3	-	1	-	yes	0	64	12225	14035	0
firI2				-	160	1.600E3	-	1	-	yes	0	64	12225	14035	0
firQ1				-	159	1.590E3	-	1	-	yes	0	64	12192	14002	0
firQ2				-	159	1.590E3	-	1	-	yes	0	64	12192	14002	0

Figure 7: Phase detector performance

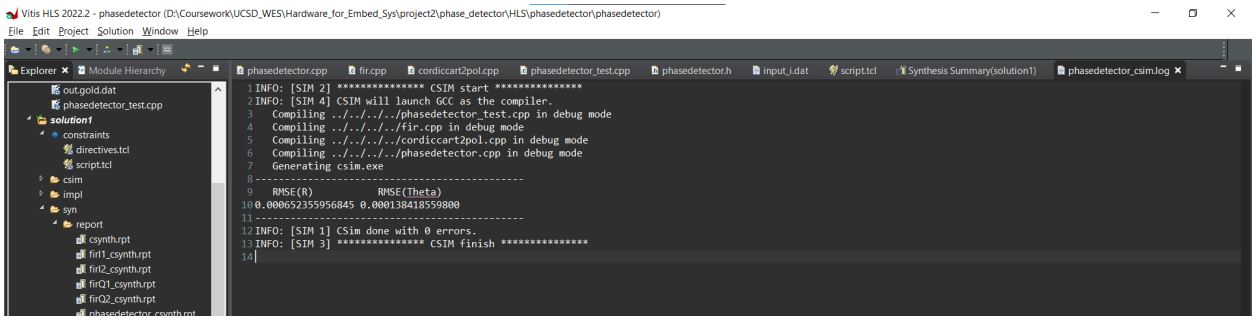

Figure 8: Phase detector simulation result (Pass)