CSE237C: Lab 2 CORDIC

Weihong Xu
wexu@ucsd.edu

October 24, 2020

1 Introduction

This report includes several types of COordinate Rotation DIgital Computer (CORDIC) designs. The goal is to make tradeoffs between complexity, arithmetic precision, and resource consumption.

2 Baseline

The folder ***cordic_ baseline*** includes a functionally correct design of the CORDIC baseline. The core function in *cordiccart2pol.cpp* is given as follows:

```
    unsigned char i;
        float x_old;

        *theta = 0.0;

Preprocessing:
        if (x < 0)
        {
                x_old = x;

                if (y < 0)
                {
                        x = -y;
                        y = x_old;
                        *theta = -1.57079632;
                }
                else
                {
                        x = y;
                        y = -x_old;
                        *theta = 1.57079632;
                }
        }

Rotation:
        for (i = 0; i < ITER_RUN; i++)
        {
                x_old = x;

                if (y > 0)
                {
                        x += y * shift[i];
                        y -= x_old * shift[i];
                        *theta += angles[i];
                }
                else
                {
                        x -= y * shift[i];
                        y += x_old * shift[i];
                        *theta -= angles[i];
                }
        }

        *r = x * 0.60725293510314;
```

The function includes two steps. The first step is to make some preprocessings for those points on Quadrant II and Quadrant III. They will be rotated for $\pi/2$ before calculating by CORDIC. The second step is to perform CORDIC rotation.

The design is verified using the test bench, where I add a test function that generates 100 samples to validate the algorithm correctness. Given the clock period constraint of 10 ns, the synthesis results are shown as follows:

Table 1: Synthesis results and throughput of baseline design.

Est. Clk (ns)	Clock Cycles		Utilization				Throughput (MHz)
	Latency	Interval	BRAM	DSP48E	FF	LUT	
8.440	171	171	0	21	1965	1867	0.693

3 Testbench Modification

I made some changes to the original testbench by adding a *gen_random_data* function to randomly generate testing samples. The test ia done at least 100 rounds to guarantee extensive testing.

```
void gen_random_data(data_t *a, data_t *b, data_t *golden_theta, data_t *golden_r)
{
        *a = (data_t)((rand() % 20000) - 1e4) / 1e4;
        *b = (data_t)((rand() % 20000) - 1e4) / 1e4;

        *golden_theta = atan(*b / (*a));

        if ((*a < 0) && (*b > 0))
                *golden_theta += PI;

        if ((*a < 0) && (*b < 0))
```

```
                *golden_theta += -PI;

        *golden_r = sqrt((*a) * (*a) + (*b) * (*b));
}
```

4 Design and Optimization

4.1 Question 1

4.1.1 Solution

I observe that the CORDIC computation has become stable before reaching the predefined maximum iterations. Based on the baseline design, the folder ***cordic_ optimized1*** includes an optimized design that reduces the number of rotations.

Table 2 summarizes the RMSE values under different iteration numbers. I select the iteration number of 10 as the parameter of optimized design since it introduces acceptable error.

Table 2: The RMSE values as the rotation number decreases.

Iter.	16	15	14	13	12	11	10	9	8
RMSE(R)	8.6×10^{-8}	8.6×10^{-8}	8.6×10^{-8}	8.6×10^{-8}	1.1×10^{-7}	3.1×10^{-7}	1.1×10^{-6}	4.6×10^{-6}	1.8×10^{-5}
RMSE(Theta)	1.8×10^{-5}	3.5×10^{-5}	7.5×10^{-5}	1.4×10^{-4}	3.0×10^{-4}	5.7×10^{-4}	1.1×10^{-3}	2.3×10^{-3}	4.9×10^{-3}

The selected design passes the simulation and matches the ground truth results. The synthesis results and corresponding performance for baseline and optimized design are given in Table 3. The design presented in this section is called **Opt. 1**.

Table 3: Synthesis results and throughput comparison for various CORDIC designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				Throughput (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.440	171	171	0	21	1965	1867	0.693
Opt. 1	8.178	123	123	0	21	1907	3320	0.994

4.1.2 Analysis

- *One important design parameter is the number of rotations. Change that number and describe the results. What happens to performance? Resource usage? Accuracy of the results? Why does the accuracy stop improving after so many iterations? Can you precisely state when that occurs?*

  The required clock cycles to finish the CORDIC computation declines as the rotation number decreases by 28%. The LUT consumption increases by 78%. As a result, the throughput increases to 0.994 MHz. The accuracy of results under different rotation numbers are given in Table 3.

The accuracy stops improving because the precision improvements from rotation updating reach to its limit. The limitation is caused by the assumption that the updating rotation angle is half of the previous one. The predefined tables of angles and shift values are on the basis of 2, where the arithmetic precision is not enough after around 13 iterations.

4.2 Question 2-4

4.2.1 Solution

The floating point-based computation is expensive due to the fact that DSP resources on board are limited. I transfer the design **Opt. 1** into fixed-point format as shown in the following code. The multiplication operations when rotation updating are replaced by right shift.

```
    // Convert the floating-point inputs to fixed-point representation
        data_fixed fixed_x = x;
        data_fixed fixed_y = y;
        data_fixed x_old;
        data_fixed fixed_theta = 0.0;
        bool flag_neg_x, flag_neg_y;

        flag_neg_x = fixed_x[W - 1]? 1: 0;
        flag_neg_y = fixed_y[W - 1]? 1: 0;

        Preprocessing:
        if (flag_neg_x)
        {
                x_old = fixed_x;

                if (flag_neg_y)
                {
                        fixed_x = -fixed_y;
                        fixed_y = x_old;
                        fixed_theta = -1.57079632;
                }
                else
                {
                        fixed_x = fixed_y;
                        fixed_y = -x_old;
                        fixed_theta = 1.57079632;
                }
        }

        Rotation:
        for (unsigned char i = 0; i < ITER_RUN; i++)
        {
                x_old = fixed_x;

                if (flag_neg_y)
                {
                        fixed_x -= fixed_y >> i;
                        fixed_y += x_old >> i;
                        fixed_theta -= angles[i];
                }
                else
                {
                        fixed_x += fixed_y >> i;
                        fixed_y -= x_old >> i;
                        fixed_theta += angles[i];
                }

                flag_neg_y = fixed_y[W - 1]? 1: 0;
        }
```

In this case, the bit length will significantly impact the result accuracy. I search the accuracy loss using different bit lengths. The integer part is fixed to 3 bits while the rotation number is fixed to 10 iterations. The associated RMSE values are summarized in Table 4.

Table 4: The RMSE values under different bit lengths.

Bit length	18	17	16	15	14	13
RMSE(R)	2.8×10^{-5}	5.8×10^{-5}	1.7×10^{-4}	2.0×10^{-4}	7.6×10^{-4}	1.4×10^{-3}
RMSE(Theta)	1.1×10^{-3}	1.1×10^{-3}	1.1×10^{-3}	1.3×10^{-3}	1.5×10^{-3}	2.7×10^{-3}

I select the bit length of 14 as the optimized design. The design presented in this section is called **Opt. 2**. The synthesis results and performance comparison are listed in Table 5.

Table 5: Synthesis results and throughput comparison for various CORDIC designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				Throughput (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.440	171	171	0	21	1965	1867	0.693
Opt. 1	8.178	123	123	0	21	1907	3320	0.994
Opt. 2	8.702	29	29	0	1	1330	5474	3.967

4.2.2 Analysis

- *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?*

  In most cases, one data type is sufficient for every variable. However, it may not yield a good system efficiency. It is better to use different data types for different variables since various fractional and integer bit combinations can well accommodate the data range.

  The best data type heavily depends on the input data since the dynamic range of different input data may vary, depending on the algorithms.

  A good way to determine the data type is first to identify the dynamic ranges for each variable. Select the suitable quantization scheme to accommodate the data range of associated variables. Then perform simulation to see if the adopted data quantization satisfies the accuracy requirement.

- *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?*

  In this section, I use simple fixed-point operations rather than floating-point operations. The arithmetic complexity is significantly reduced. This design only requires 1 DSP at the expense of additional LUT consumption. Besides, the timing performance is improved, where the latency is shortened from 123 to 29 cycles. The throughput reaches to 3.967 MHz. The accuracy loss is around 10^{-3}, which is acceptable.

- *How does the ternary operator '?' synthesize? Is it useful in this project?*

  The ternary operator is synthesized to architectures similar to conditional branch. I use the ternary operator to calculate the flag that represents negative values.

4.3 Question 5

Table 6: The RMSE values of R under different bit lengths and LUT fillings.

	W=9	W=8
Fully	2.3×10^{-3}	4.1×10^{-3}
Partially	1.0×10^{-4}	1.0×10^{-4}

Table 7: The RMSE values of theta under different bit lengths and LUT fillings.

	W=9	W=8
Fully	2.8×10^{-3}	3.2×10^{-2}
Partially	2.8×10^{-2}	3.1×10^{-2}

- *These questions all refer to the LUT-based CORDIC: Summarize the design space exploration that you performed as you modified the data types of the input variables and the LUT entries. In particular, what are the trends with regard to accuracy (measured as error)? How about resources? What about the performance? Is there a relationship between accuracy, resources, and performance? What advantages/disadvantages does the regular CORDIC approach have over an LUT-based approach?*

  The design space exploration is performed through modifying the data types and LUT entries. The bit length of input variables ranges from 8 to 9 bits, where the integer part is fixed to 2 bits. Table 6 and Table 7 summarize the RMSE values of R and theta under different bit lengths and LUT filling forms. The error slightly increases as the bit length shrinks. Besides, the resulting accuracy of partially LUT filling is higher than fully LUT filling.

The LUT-based CORDIC designs are tested and synthesized. The synthesis results are shown in Table 8. Compared to the previous designs, LUT-based designs do not need any DSP resources and significantly reduce the latency and interval to only 5 clock cycles, improving the throughput to about 23 MHz. The required FF and LUT resources are also reduced at the expense of increased BRAM consumption.

The LUT-based designs realize CORDIC computation by one-shot table lookup. The advantage is it requires much less clock cycles, thus delivering very high throughput. The disadvantage is LUT-based designs cost much more BRAM memory resources, which may be insufficient as the LUT size and required accuracy increase.

The advantage of regular CORDIC design is that the high accuracy can be easily achieved and the resource consumption is relatively balanced. But the iterative processing nature limits the throughput which could be the disadvantage.

Table 8: Synthesis results and throughput comparison for various CORDIC designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (MHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.440	171	171	0	21	1965	1867	0.693
Opt. 1	8.178	123	123	0	21	1907	3320	0.994
Opt. 2	8.702	29	29	0	1	1330	5474	3.967
LUT, W=9, fully	8.425	5	5	1024	0	868	3020	23.739
LUT, W=8, fully	8.670	5	5	256	0	880	3000	23.068
LUT, W=9, partially	8.425	5	5	1024	0	868	3020	23.739
LUT, W=8, partially	8.670	5	5	256	0	880	3000	23.068