UCSD 237C: Project 2 CORDIC

Steven Daniels

sdaniels@ucsd.edu

Student ID# A53328625

November 5, 2023

1. Introduction

CORDIC (Coordinate Rotation DIgital Computer) is an efficient technique to calculate trigonometric, hyperbolic, and other mathematical functions. It is a digit-by-digit algorithm that produces one output digit per iteration. This report presents various hardware designs for a CORDIC built for converting rectangular coordinates to polar coordinates. The root mean square error of CORDIC output for "R" and "Theta" is used to confirm its accuracy against the expected value. The purpose of this report is to show the trade-offs of varying different hardware parameters when determining the best design for a CORDIC hardware implementation. Five different designs were explored:

1. CORDIC baseline.
2. CORDIC that uses 11 rotations.
3. CORDIC that uses 19 rotations.
4. CORDIC that uses fixed point variables.
5. CORDIC that uses a lookup table.

Each section of the report seeks to answer a different question about the different hardware implementations of the CORDIC.

2. CORDIC Designs

2.1. Baseline

2.1.1. Throughput Max: 522 kHz

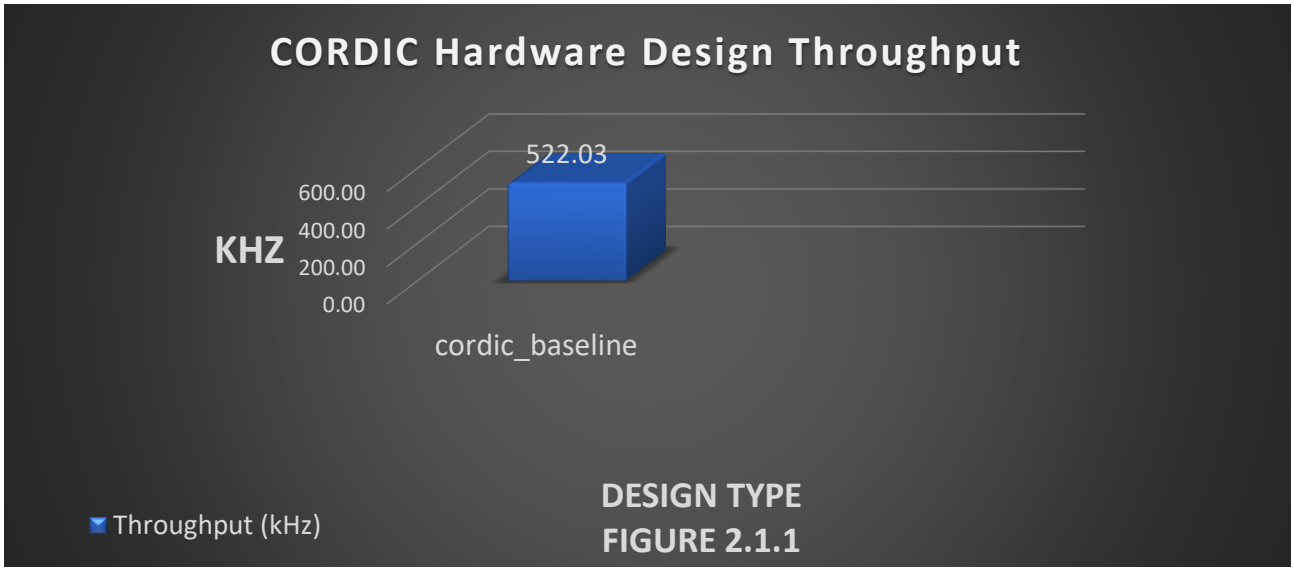

FIGURE 2.1.1

2.1.2. Implementation

```
void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta)
{
    if (x == 1 && y == 0)
    {
        *r=1;
        *theta=0;
        return;
    }
    data_t cur_x = 0;
    data_t cur_y = 0;
    // Use the sign bit of x and y to determine which quadrant the vector is in
    unsigned short x_sign_bit = signbit(x);
    unsigned short y_sign_bit = signbit(y);
    int to_quad = 0;

    if(x_sign_bit == 1.0 && y_sign_bit == 0.0)
    {
        cur_x =  y;
        cur_y = -1.0*x;
        *theta = angles[0]*-2;
    }
    else if(x_sign_bit == 1.0 && y_sign_bit == 1.0)
    {
        cur_x = -1.0*y;
        cur_y = x;
        *theta = angles[0]*2;
    }
    else if(x_sign_bit == 0.0 && y_sign_bit == 1.0)
    {
        cur_x = -1.0*y;
        cur_y = x;
        *theta = angles[0]*2;
    }
    else
    {
        cur_x = y;
        cur_y = x*-1.0;
        *theta = angles[0]*-2.0;
    }
```

```
    data_t sigma = 0;
    data_t tempx = 0;
    data_t tempy = 0;
    for (int i = 0; i < NUM_ITER; i++)
    {
        sigma = (cur_y > 0) ? -1 : 1;

        // perform rotation/transformation
        tempx = cur_x;
        tempy = cur_y;
        cur_x = tempx - (sigma * Kvalues[i] * tempy);
        cur_y = tempy + (sigma * Kvalues[i] * tempx);
        *theta = *theta + sigma * angles[i];
    }
    // Set the final radians and phase
    *r = cur_x*cordic_gain[NUM_ITER-1]; *theta = -1*(*theta);
}
```

2.1.3. Optimizations

None

2.1.4. Resources

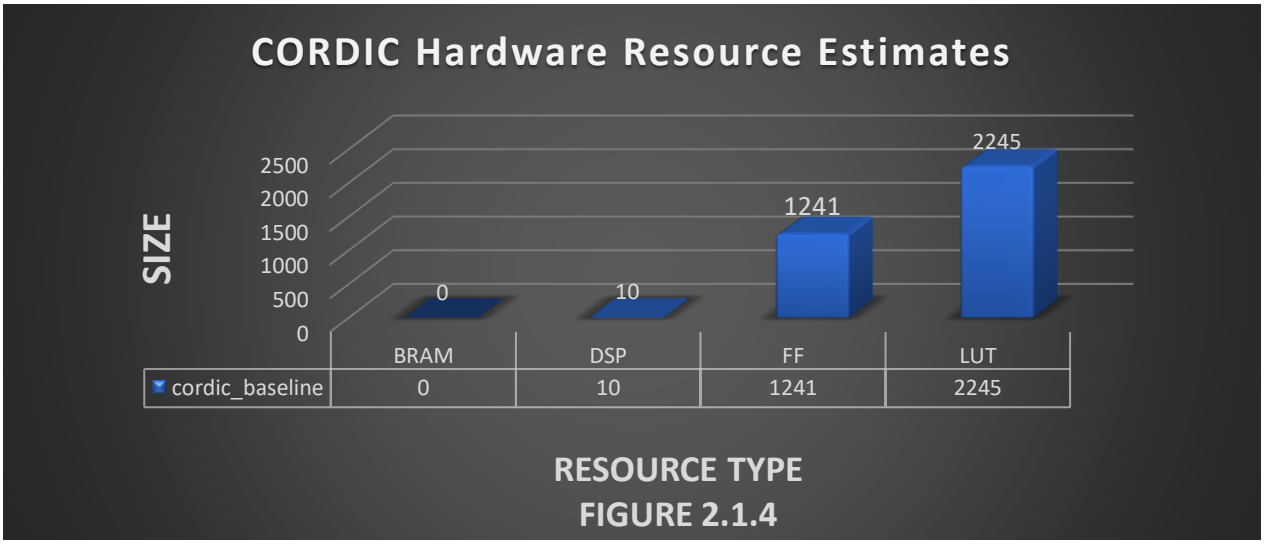

FIGURE 2.1.4

2.1.5. Analysis

This section presents a baseline implementation of the CORDIC algorithm using floating variables as the data type and a CORDIC gain array to scale the final vector. This implementation returns the correct magnitude and phase for all test vectors implemented in the test bench.

2.2. CORDIC that uses 11 rotations.

2.2.1. Throughput Max: 787 kHz

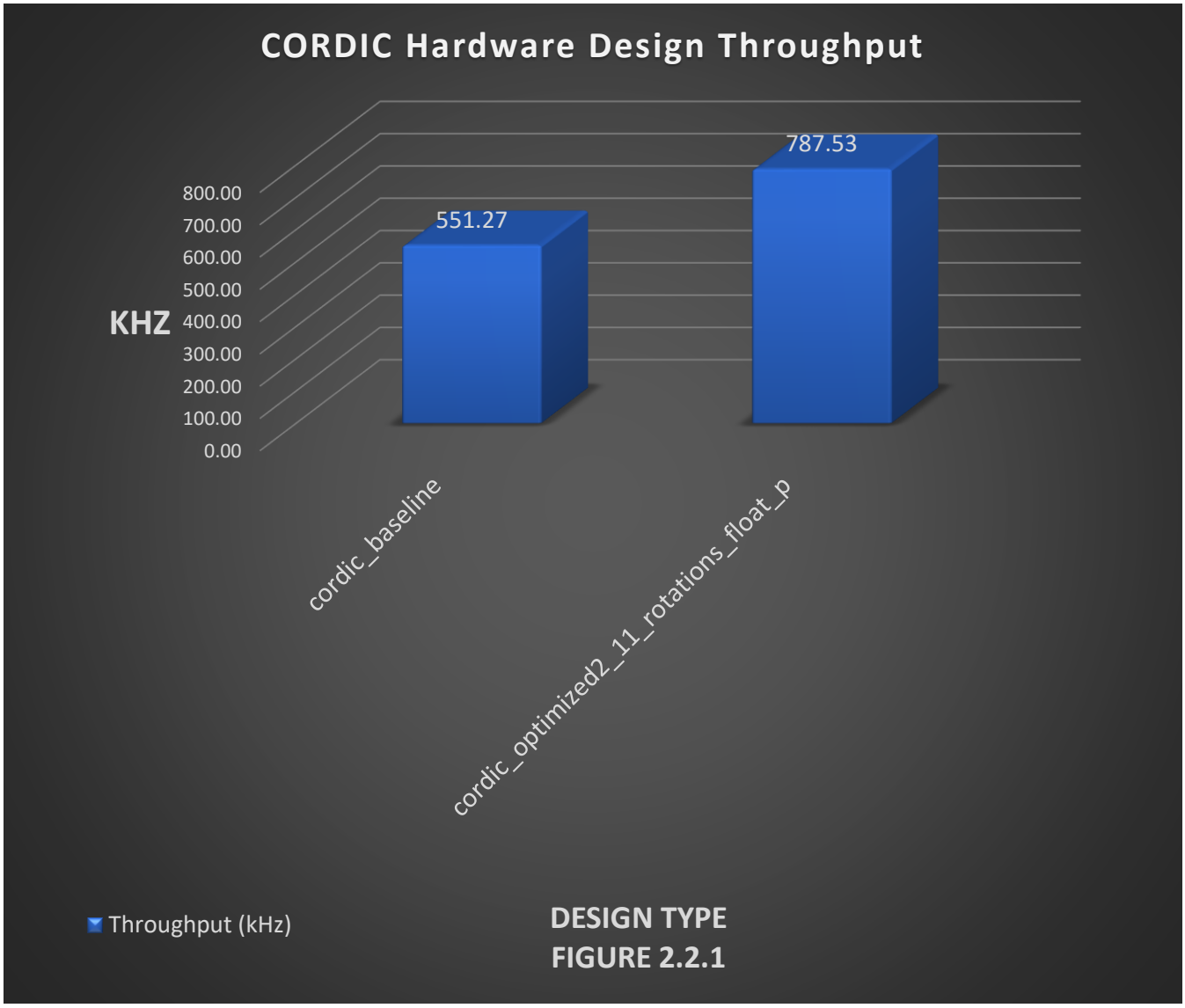

FIGURE 2.2.1

2.2.2. Implementation

The algorithm for calculating the angle remained the same. Only the number of rotations was changed.

```
#ifndef CORDICCART2POL_H
#define CORDICCART2POL_H

#define NUM_ITER 11
#define LUT_SIZE 20
#define NUM_OF_QUADRANTS 5
#define NUM_GAIN_VALS 27

typedef int   coef_t;
typedef float data_t;
typedef float acc_t;

void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta);

#endif
```

2.2.3. Optimizations

- Decreased number of rotations from baseline value of 16 to 11.

2.2.4. Resources

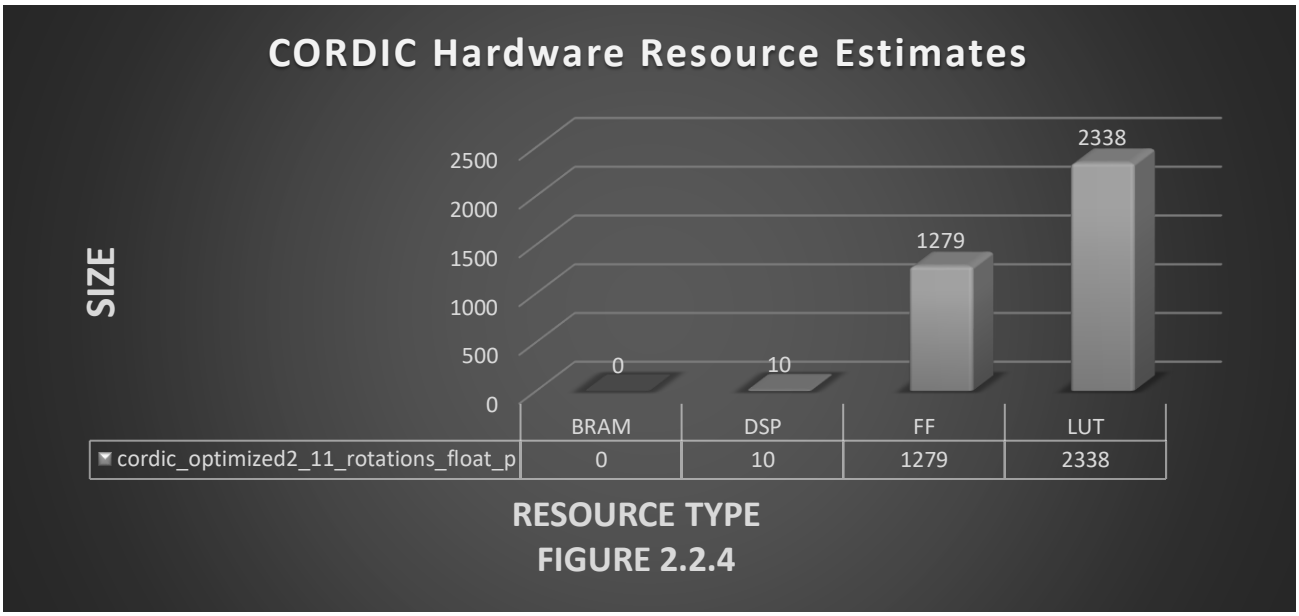

FIGURE 2.2.4

2.2.5. Analysis

This design attempts to optimize the CORDIC by decreasing the number of rotations needed to reach the target angle. The target angle was reached but provided worse accuracy when compared to the baseline. This would be a be good design to use if precise accuracy is not a requirement for the system.

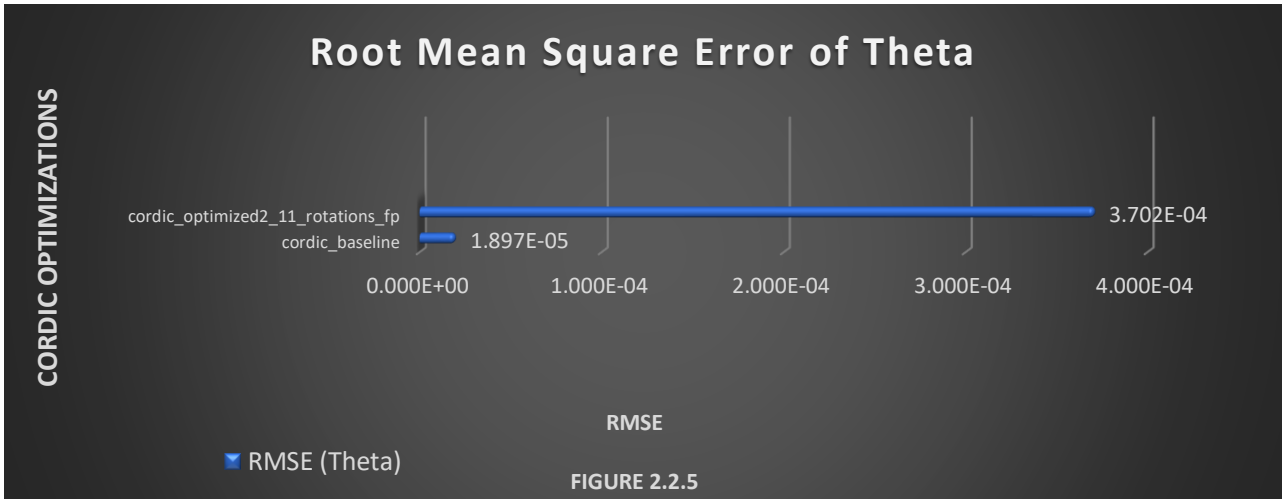

FIGURE 2.2.5

2.3. CORDIC that uses 19 rotations

2.3.1. Throughput Max: 467 kHz

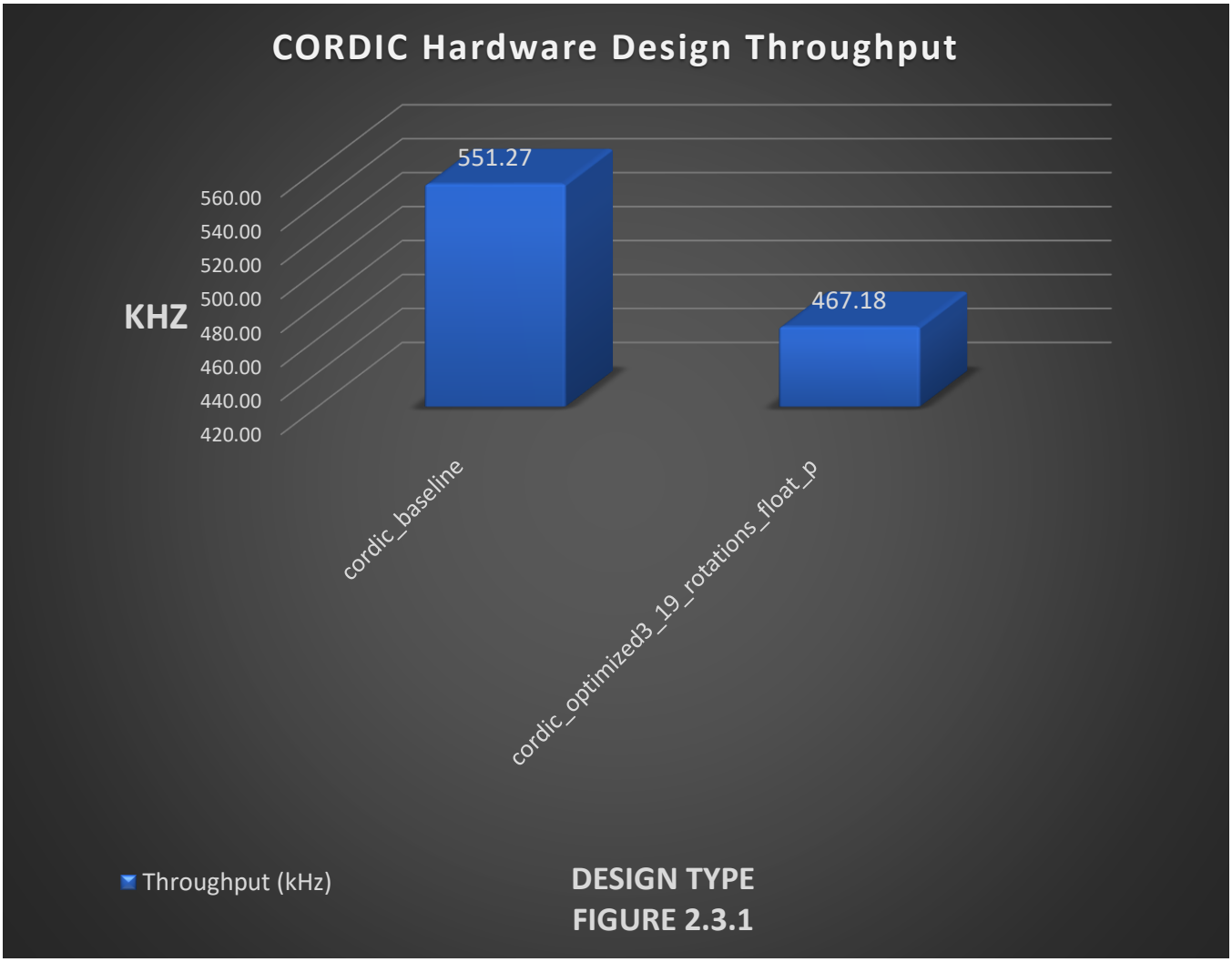

FIGURE 2.3.1

2.3.2. Implementation

The algorithm for calculating the angle remained the same. Only the number of rotations was changed.

```
#ifndef CORDICCART2POL_H
#define CORDICCART2POL_H

#define NUM_ITER 19
#define LUT_SIZE 20
#define NUM_OF_QUADRANTS 5
#define NUM_GAIN_VALS 27

typedef int   coef_t;
typedef float data_t;
typedef float acc_t;

void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta);

#endif
```

2.3.3. Optimizations

- Updated rotations from baseline value of 16 to 19.

2.3.4. Resources

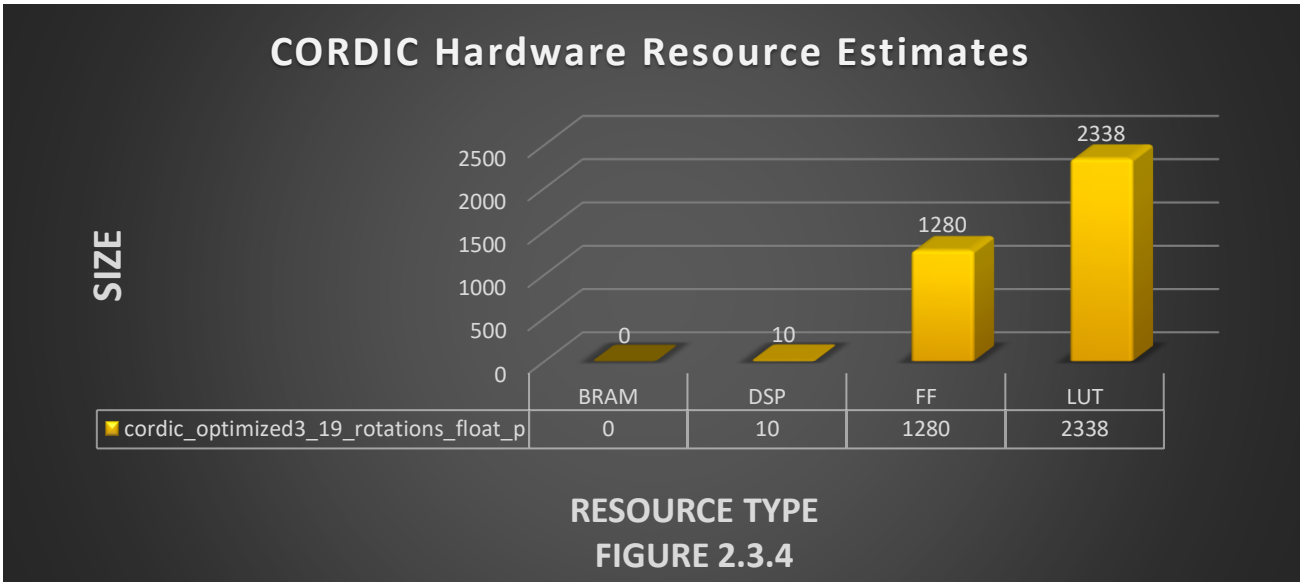

FIGURE 2.3.4

2.3.5. Analysis

This design attempts to optimize the CORDIC by increasing the number of rotations needed to reach the target angle. The target angle was reached with a major improvement in accuracy over the baseline. This would be a be good design to use if precise accuracy is a requirement for the system.

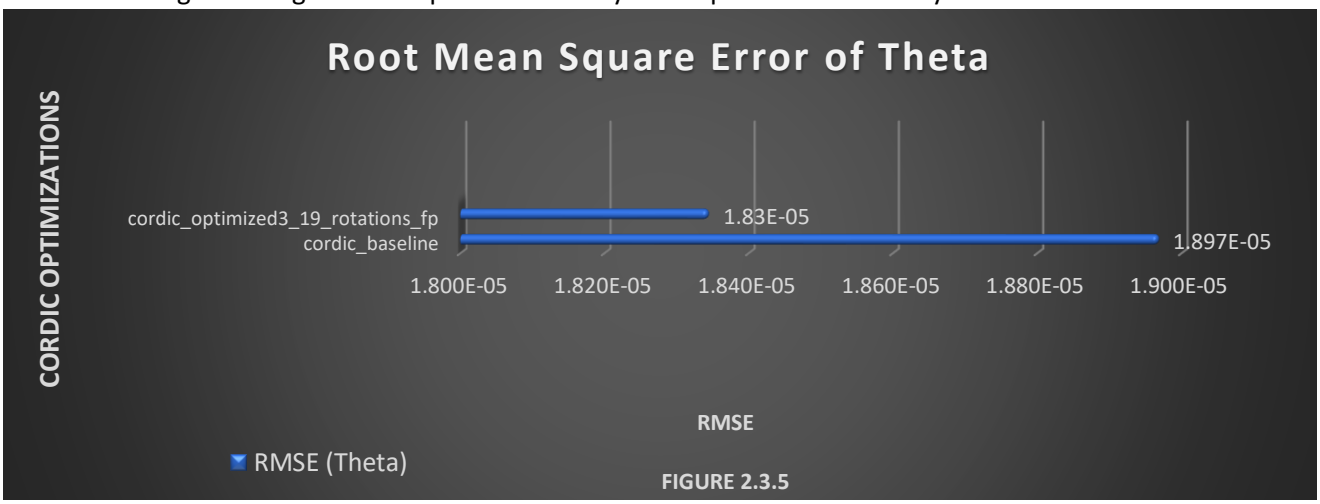

FIGURE 2.3.5

2.4. CORDIC that uses fixed point variables

2.4.1. Throughput Max: 3.4 MHz

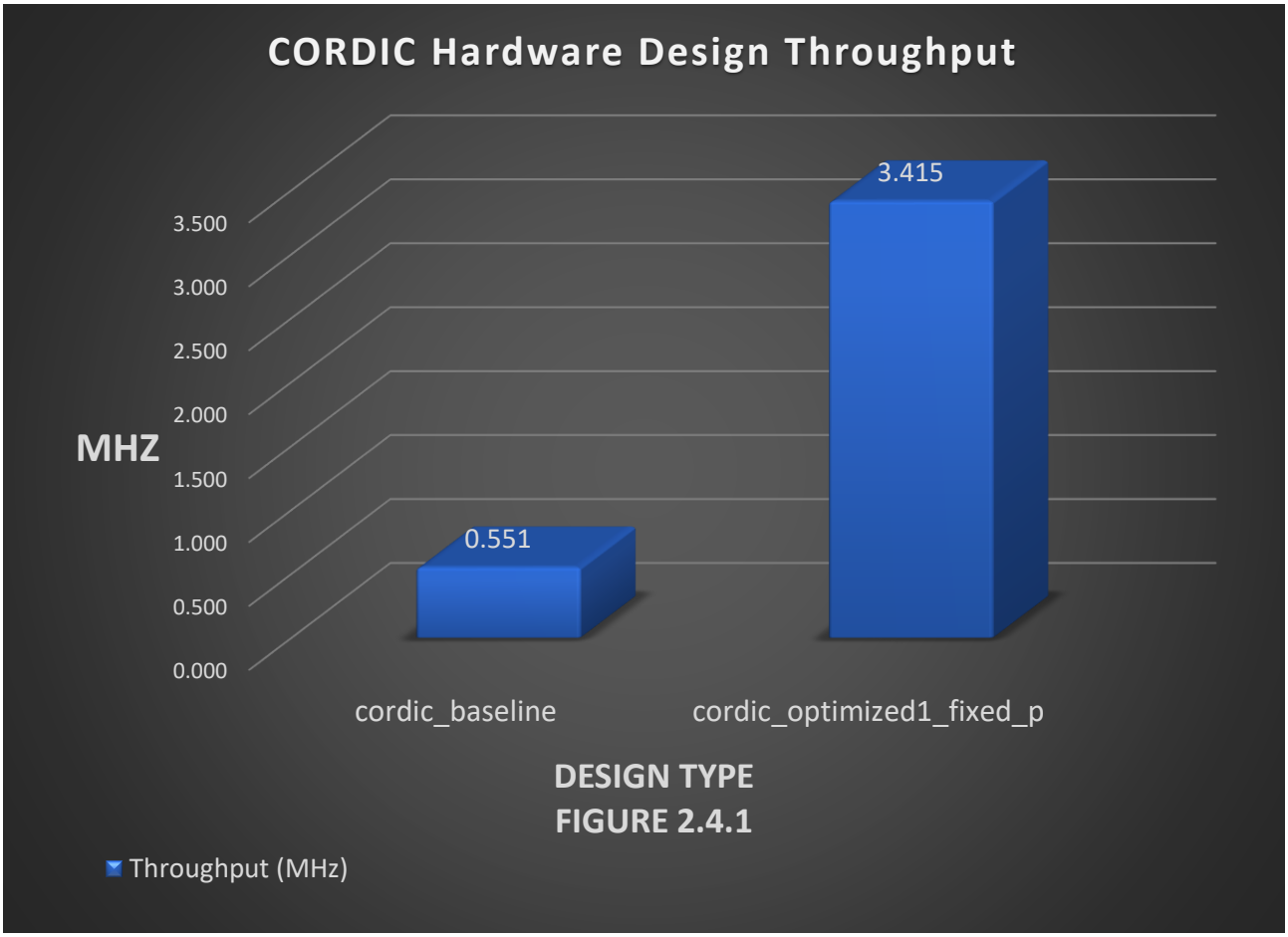

FIGURE 2.4.1

2.4.2. Implementation

- Cordiccart2pol.cpp

```
void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta)
{
    data_t two_multiplier  = 2;

    if (x == 1 && y == 0)
    {
        *r=1;
        *theta=0;
        return;
    }
    data_t cur_x = 0;
    data_t cur_y = 0;
    // Use the sign bit of x and y to determine which quadrant the vector is in

    if(x < 0 && y >= 0.0)
    {
        cur_x =  y;
        cur_y = -x;
        *theta = -(angles[0] + angles[0]);
    }
    else if(x < 0 && y < 0)
    {
        cur_x = -y;
        cur_y = x;
        *theta = angles[0] + angles[0];
```

```
    }
    else if(x >= 0 && y < 0)
    {
        cur_x = -y;
        cur_y = x;
        *theta = angles[0] + angles[0];
    }
    else
    {
        cur_x = y;
        cur_y = -x;
        *theta = -(angles[0] + angles[0]);
    }

    data_t sigma = 1;
    data_t tempx = 0;
    data_t tempy = 0;
    data_t y_to_shift = 0;
    data_t x_to_shift = 0;
    for (int i = 0; i < NUM_ITER; i++)
    {
        if(cur_y>0)
        {
            y_to_shift = -(cur_y >> i);
            x_to_shift = -(cur_x >> i);

            *theta = (*theta - angles[i]);
        }
        else
        {
            y_to_shift = (cur_y >> i);
            x_to_shift = (cur_x >> i);
            *theta = *theta + angles[i];
        }

        // perform rotation/transformation
        tempx = cur_x;
        tempy = cur_y;
        cur_x = tempx - (y_to_shift);
        cur_y = tempy + (x_to_shift);

    }


    // Set the final radians and phase
    *r = cur_x*cordic_gain[NUM_ITER-1]; *theta = -(*theta);
}
```

- Cordiccart2pol.h

```
#ifndef CORDICCART2POL_H
#define CORDICCART2POL_H

#define NUM_ITER 16
#define NUM_OF_QUADRANTS 5
#define NUM_GAIN_VALS 27
#include "ap_fixed.h"

typedef ap_fixed<18,3> coef_t;
typedef ap_fixed<18,3> data_t;
typedef ap_fixed<18,3> acc_t;


void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta);

#endif
```

2.4.3. Optimizations

- Replaced multiply operations with bitwise shift operations.
- Added Fixed point representation to all variables.

2.4.4. Resources

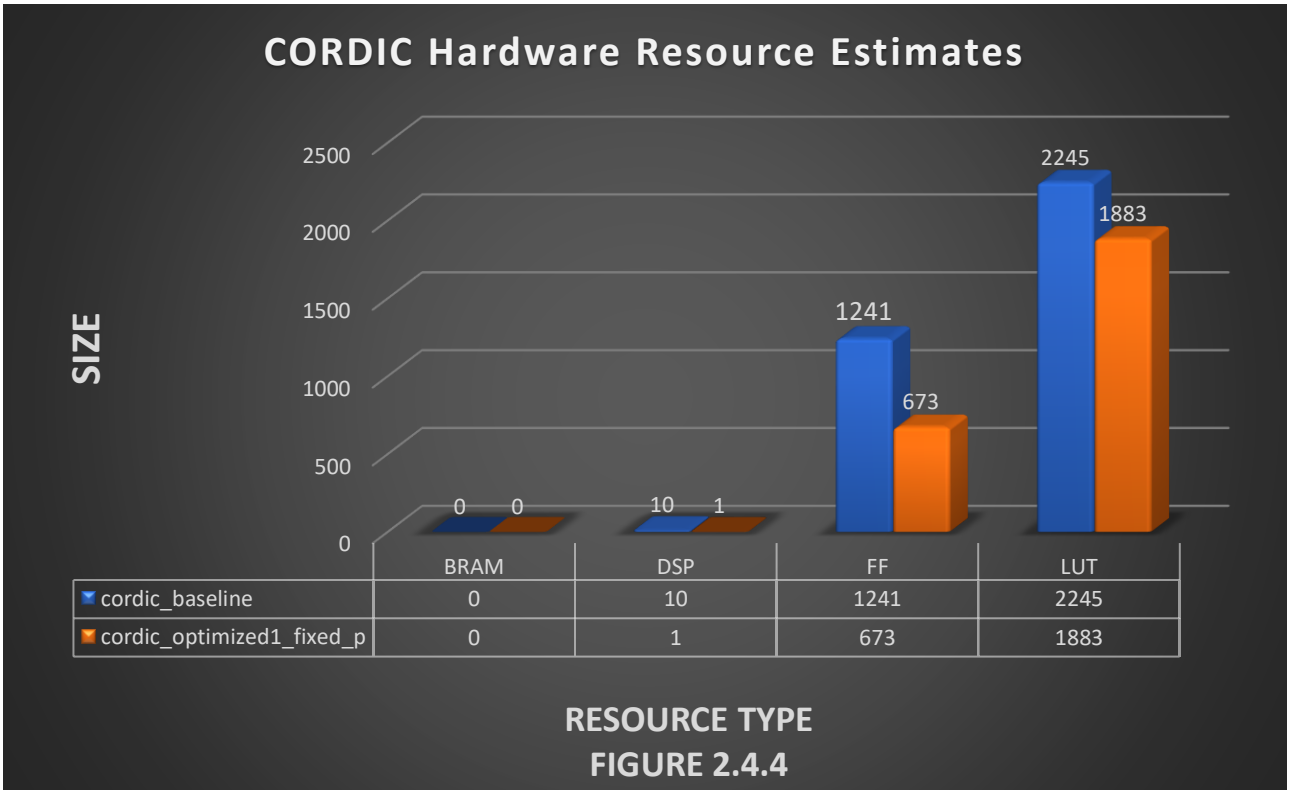

FIGURE 2.4.4

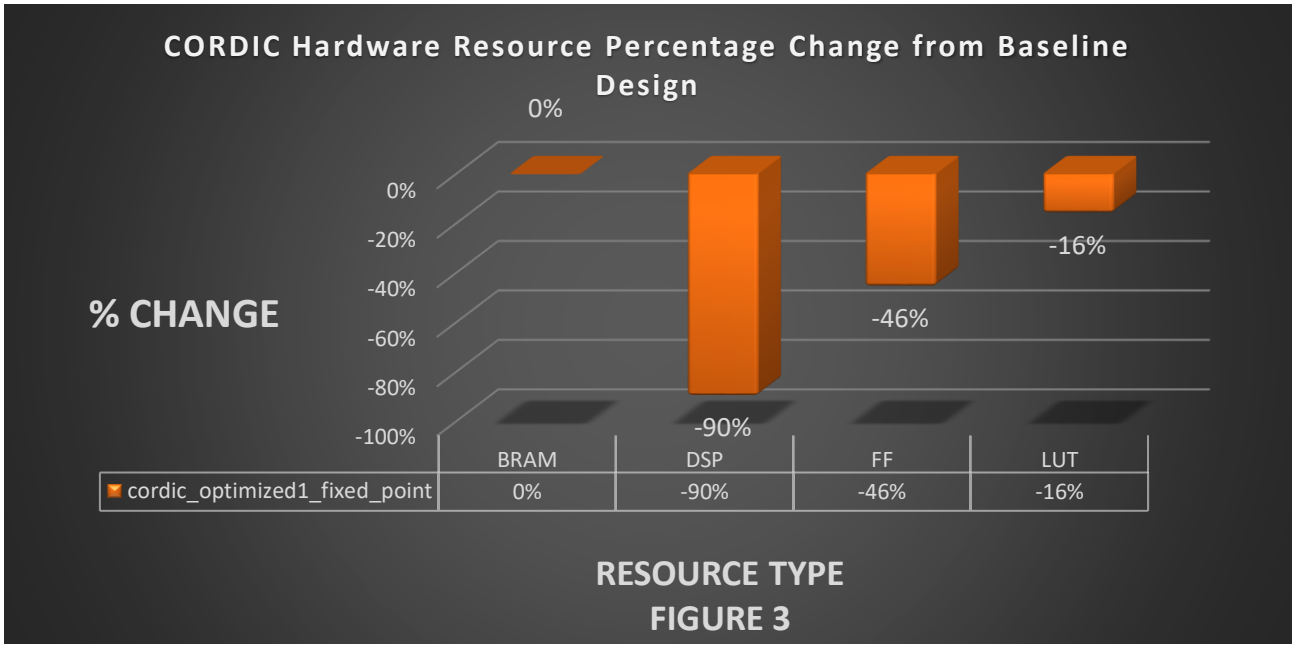

FIGURE 3

2.4.5. Analysis

This design attempts to optimize the CORDIC by using arbitrary precision data types for all variables. This allows for a large reduction in the number of resources needed to get an acceptable output for “R” and “theta”. This is possible because the CORDIC algorithm requires that the rotation angle be reduced or

increased by 2^-i radians each iteration of the loop. This reduction eventually leads to an angle of rotation with a value that provides little to no effect on the final estimate.

1.1. CORDIC that uses a lookup table

1.1.1. Throughput Max: 51 MHz

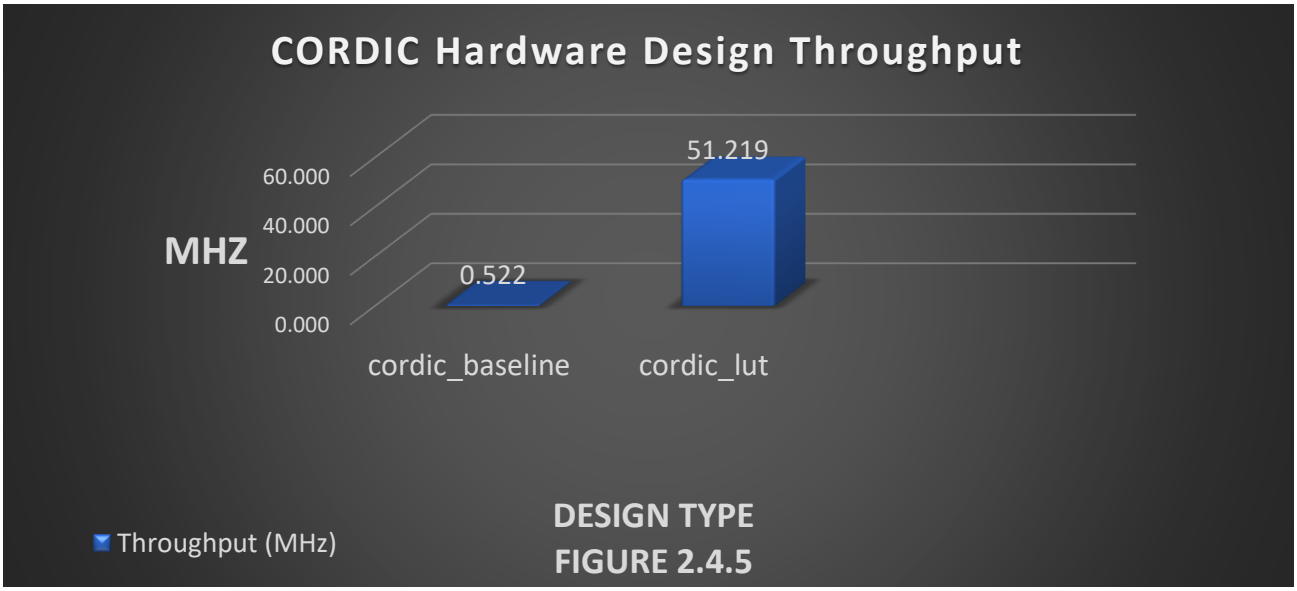

FIGURE 2.4.5

2. CORDIC Questions

2.1. Question 1 – Number of Rotations

Increasing the number of rotations for the CORDIC leads to a drop in performance when compared to the baseline. Decreasing the number of rotations shows the opposite of the former when compared to the baseline. The overall accuracy of the CORDIC increases as the number of rotations increases due to the rotating vector closely approaching its desired target angle while decreasing the number of rotations reduces the accuracy of the CORDIC. The accuracy of the CORDIC is tied to N number of bits of precision the design needs. The CORDIC would stop improving once the number of iterations reduces the angle value beyond the N number of fractional bits specified in the design. Therefore, performing more iterations after reaching that point provides little to no effect on the movement of the vector. Essentially the vector has moved as close as it possibly can to the target angle.

2.2. Question 2 – Variable Data Types

One data type is sufficient for every variable if there are not strict requirements on resource usage. However, if the aim is to reduce the number of resources for a CORDICs design the hardware could be designed to provide the minimum amount of precision needed for each variable used in the CORDIC algorithm. The best data type does the depend on the input data. For example, if designing a CORDIC that only handles data coming from a 16-bit analog to digital converter, the hardware could be designed to use a fixed-point representation with the precision specified by the number of bits coming from the analog to digital converter. The best way for a designer to determine the data types needed for hardware implementation is to consider the size of the data coming into the system. The designer needs to know how much precision is required to process the data and still produce the correct output.

2.3. Question 3 – Simple Operations

Using simple operations in the CORDIC greatly improved the throughput of the design when compared to baseline. The throughput of a design with simple operations is displayed in **Figure 2.4.1.** This figure

demonstrates that the throughput can be approved by greater than 2x the baseline throughput. The accuracy slightly decreased due to moving from floating point to fixed point since there were less bits to available to use in our approximation and there was a major decrease in resource usage especially in the number of DSP's which can save on the overall power consumption as seen in **Figure 2.4.4**.

2.4. Question 4 – Lookup Table Implementation

2.4.1. Question 1

The values in the lookup table are a multiple of the size of the input and output data types. The greater the number of bits needed to represent the input and output data, the larger the lookup table will be. The output data type should have a number of bits greater than or equal to the number bits used to store the lookup table data. The number of bits required to store information in the LUT is smaller than the number of bits needed to store information in the input and output data because it used to store a smaller range of values. It's integer part should only represent the range of values that can be output by the "arctan" and "sqrt" function for inputs between [-1,1].

2.4.2. Question 2

The minimum number of integer bits required for x and y is 2. The minimum number of integer bits required for the output of R is 2 and 5 for theta.

2.4.3. Question 3

Decreasing the number of fractional bits in the input and output data types decreases the accuracy of the CORDIC.

2.4.4. Question 4

The LUT implementation of the CORDIC has a considerably high throughput when compared to the baseline implementation. It provides 51 MHz of throughput as seen in **Figure 2.4.5**. The throughput decreases as data types with more integer bits are used.

2.4.5. Question 5

The advantage of the CORDIC implementation is that you don't have to precompute your values beforehand so that saves an extra step in the processing and you can force an exit from the heart of the loop by decreasing the iterations if you know how many you need to get the type of accuracy you desire. The disadvantage is in performance since using a lookup table to grab the values you need will often be faster than computing them on the fly.