Implementation of Complex Arithmetic Functions in Hardware using the CORDIC Algorithm

Jason Math
Department of Electrical and Computer Engineering
University of Texas at Austin,
Austin, TX 78712
jasonmath@utexas.edu

Sidharth Nair
Department of Electrical and Computer Engineering
University of Texas at Austin,
Austin, TX 78712
sidnair@utexas.edu

***Abstract*—This project report explores the hardware implementation of complex arithmetic functions, specifically using the CORDIC algorithm. This algorithm has been found to leverage clever lightweight hardware calculation to approximate functions that are generally considered slow. We use Verilog to create the CORDIC algorithm and a Taylor series accelerator to calculate sine and cosine and synthesize a gate-level schematic and physical layout for the modules to more accurately measure the power, performance, and area (PPA) to compare these implementations for different accuracy levels and other constraints.**

I. Introduction

If you were asked to implement a complex function like sine or cosine on a computer, your first thought would likely be implementing the Taylor series representation of the function. However, this can become very expensive since Taylor series terms involve computing multiplications, exponents, factorials, and divisions; these are especially costly to implement in hardware, and you would need different logic to implement different Taylor series formulas. Another option could be to use a lookup table for certain inputs of a function and perform interpolation to get an estimate for arbitrary inputs. This can perform much faster than a Taylor series, and we can bound the size of our lookup tables in the case of periodic functions like sine and cosine. However, if we wanted to increase the precision of our implementation, we would need to exponentially increase the size of our table, which is not a scalable solution.

CORDIC (Coordinate Rotation Digital Computer) is an algorithmic technique proposed by Volder [1] used in digital signal processing and numerical computation to efficiently approximate complex mathematical functions, particularly trigonometric and hyperbolic functions. CORDIC relies on a series of iterative rotations and vector manipulations to perform these calculations using only basic arithmetic operations like shifts and additions. CORDIC's iterative nature allows it to converge to accurate approximations of trigonometric functions while consuming fewer resources and less processing time compared to traditional methods, making it a valuable tool in various applications, including navigation systems, graphics rendering, and communication systems. It achieves impressive computational efficiency by breaking down complex functions into a sequence of simple operations, making it suitable for hardware implementations like FPGAs or custom hardware accelerators. CORDIC algorithms are used in modern computer engineering computational tasks such as Digital Signal Processing (DSP), Graphics and Image Processing, and Robotics and Control systems to quickly calculate accurate approximations of complex algorithms.

In this project, we use Verilog to implement the CORDIC algorithm to compute the sine and cosine of an angle. We synthesized our design and performed automatic place and route (APR) to generate a physical layout of the module. Additionally, we implemented a Taylor series accelerator that computes the cosine of an angle to compare the power, performance, and area (PPA) of both solutions.

II. Background

As mentioned in the previous section, the CORDIC algorithm can be used to compute complex arithmetic functions, such as trigonometric and hyperbolic functions using only shifts and adds. In this section, we will explain how the algorithm works to compute the cosine and sine of an angle.

Given a vector to a point on the unit circle, we know that the cosine and sine of the angle that the vector makes with x-axis basis vector $(1\ 0)$ are the x and y components of that vector respectively. The main idea in CORDIC is to perform iterative rotations of a vector to compute these corresponding coordinates. Specifically, we use a binary search approach by rotating the vector in decreasing angle steps toward our target angle; whether we have rotated a total angle that is more or less than our target angle determines the direction of the next rotation.

As an example, let's say we are trying to compute the cosine and sine of $35°$. Then we would first start with vector $(1\ 0)$, and rotate it $45°$ counterclockwise. Now our total angle of rotation is $45°$, which is above our target, so in our next iteration, we will rotate clockwise $22.5°$. This process will repeat several times until we are satisfied with our accuracy, and in every step, we will decrease the angle we are rotating by and rotate in the direction that will make our total angle closer to the target. Finally, the cosine and sine will be the x and y components of our resultant vector.

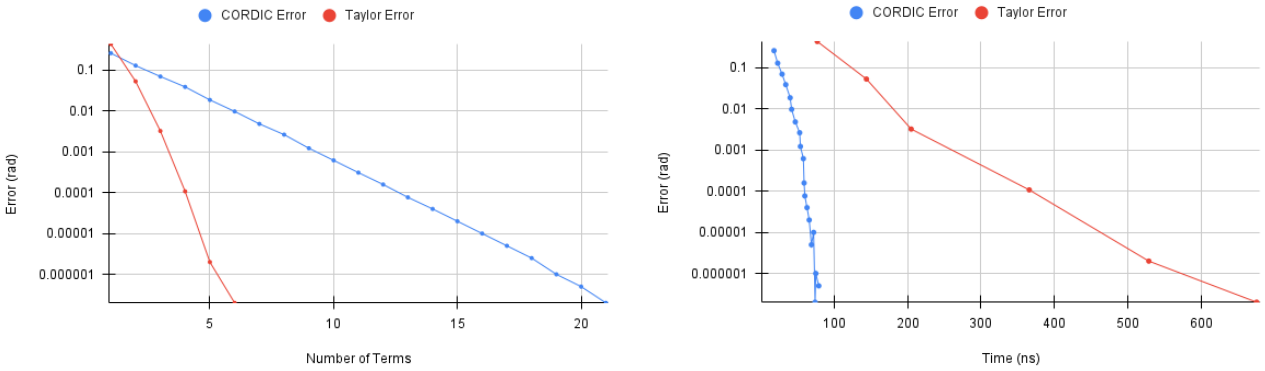

Fig. 1: Using software to model error (rad) against number of terms and time (ns)

With this model, all we need is logic that can carry out rotations on an input vector. We know that rotations can be performed using the following transformation matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta) \begin{bmatrix} x \\ y \end{bmatrix} \text{ where } R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad (1)$$

We can't use this as is since it involves cosine and sine which is the function we want to compute. However, CORDIC uses some clever simplifications that enable us to perform this computation using only shifts and adds. Factoring out cosine, yields:

$$R(\theta) = \cos\theta \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix} \quad (2)$$

The key simplification that CORDIC does is instead of using $45°$ and dividing by 2 each step, we use $\tan^{-1}(2^{-i}) = 45°, 26.565°, 14.036°, ...$ as our angle for each iteration step $(i = 0, 1, 2, ...)$. This reduces the strength of our binary search (since we are not cutting our angle step exactly in half each time), but it is close enough and will still converge to our output given enough terms. Additionally, since tangent is an odd function $(\tan(-\theta) = -\tan\theta)$, we can choose the rotation direction by choosing a value of $d_i = \pm 1$:

$$R(\theta_i) = \cos\theta_i \begin{bmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{bmatrix} \quad (3)$$

Note that the product of the matrix part of the rotation with an input vector can be computed entirely with shifts and adds. For the cosine terms, if we know how many iterations we want our CORDIC module to perform, we can compute this in advance and store it as a gain factor K which we can scale our initial vector $(1\ 0)$ to get $(K\ 0)$:

$$K = \prod_{i=0}^{N-1} \cos\theta_i \quad (4)$$

The only thing remaining is to store the actual angle values for each iteration since we will need to accumulate these to keep track of the total current rotation angle (which will be used to determine d_i. As a result, for the entire system, we will store the value of K and an angle table with an entry for each iteration we want to carry out. Using this model, we can implement the CORDIC module in hardware entirely using shifts and adds.

III. Software Modeling

In this section, we provide some baseline comparison results generated through a software model of both implementations. This helps establish a general ballpark and intuition for what we should expect, along with guaranteeing computational correctness and estimating performance and accuracy. We implemented cosine and sine using both a Taylor series formula and the CORDIC algorithm in C. We ran benchmarks, measuring precision and runtime against the number of Taylor terms and the number of CORDIC terms respectively. We used signed fixed point 32-bit numbers with 30 fractional bits for the CORDIC calculations.

Figure 1 shows our results of initial benchmarking of both angles. Note that the error is on a log scale in both plots. We see that the Taylor series converges with far fewer terms than the CORDIC algorithm. However, each Taylor term takes significantly longer to evaluate; the second plot shows that the time for CORDIC to evaluate is much faster than the Taylor series. It scales much better with the number of terms (and correspondingly with precision) since evaluating another term just requires us to perform another rotation. This is in contrast to the Taylor series, where each term gets increasingly more complex to compute due to the higher-order polynomial and factorial.

IV. Hardware Implementation

We used Verilog to implement a Taylor series accelerator to compute cosine, and the CORDIC algorithm to compute cosine and sine. Synthesis and APR were done using the 45nm technology node from FreePDK45.

A. Taylor Series Accelerator

1) Design: Since there were no readily available open-source Taylor Series HDL implementations, we created a

custom Taylor Series accelerator to use as our comparison point. Since this hardware is made specifically for calculating these exact series, we can assume that more generalized hardware implementations will be slower and larger.

In our implementation, we factor out the constants from each term ($-1/2!$, $1/4!$, $-1/6!$) and store these values as constants in a look-up table to heavily save on computation of factorial, division, and negation. From here, we can cleverly use IEEE 754 floating point adders and multipliers in a tree-like structure to maximize parallel computation. This approach yields a critical path of 3 multiplies and 2 adds:

Multiplies:

1) $x * x = x^2$
2) $x^2 * x^2 = x^4$, and $x^2 * -1/6! = -x^2/6!$ in parallel
3) $-x^2/6! * x^4 = -x^6/6!$

Adds:

1) $1 - x^2/2!$, and $x^4/4! - x^6/6!$ in parallel
2) $(1 - x^2/2!) + (x^4/4! - x^6/6!)$

Using IEEE 754 floating point fits this application well as the range of output is also -1 to 1 and the multiplies are generally faster compared to fixed point at the expense of the adds.

2) Testing: We use a simple Verilog test bench to verify the functionality and find that using our 4-term design, the values are very accurate ($< .5\%$) in the expected range ($-\pi/2$ to $\pi/2$) and fall off drastically past that (for an input of π, the error was calculated to be $> 20\%$). We also find this design takes on average 80 clock cycles to fully compute for random values, going as low as about half of that for simple values such as 0 radians.

3) Synthesis: Using the 45nm technology node from FreePDK45, we used Design Vision to synthesize a design for 2, 3, and 4 terms. The schematic shown in figure 2 shows the schematic of the clocked implementation with 4 terms.

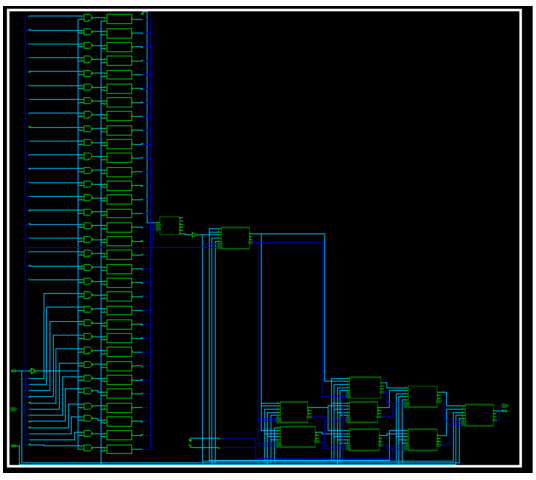

Fig. 2: Taylor Schematic

B. CORDIC Algorithm

1) Design: For our implementation of the CORDIC algorithm, we explored two possible options: a clocked circuit that evaluates a set number of CORDIC rotations (iterations) in a cycle and a fully combinational alternative. The input to our system is a 32-bit signed fixed-point angle (theta), with 30 fractional bits. In the case of the clocked implementations, we also have a clock (clk) and reset (rst) signal as inputs. The output of the system is the cosine and sine of the input angle (which are both in the same format as the input), and a done signal in the case of the clocked implementation, which indicates the output is ready.

The logic that performs an actual rotation (described in the background section) is shown in figure 3. The two adders on the left perform the shift and addition needed to compute the dot product of the matrix with the input vector $(x_i\ y_i)$, and the right adder updates the angle accumulator. The input angle z_i tells us which direction our next rotation should be in.

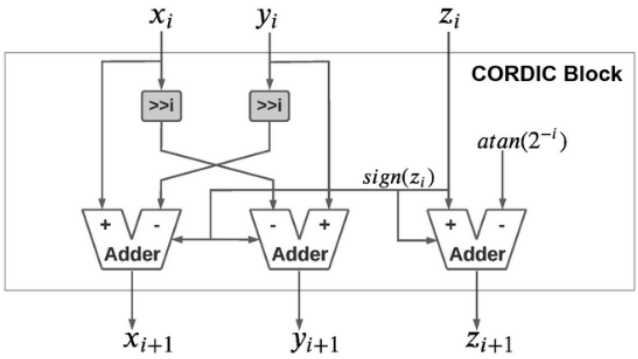

Fig. 3: CORDIC Rotation Logic Block [6]

This logic block is used to construct the following CORDIC modules. In the clocked implementation we feed the output back into the input, using a register to maintain the current value. Internally there is a counter that keeps track of the number of iterations after rst is set low, which will tell us when to gate the final output. The fully combinational logic is simply N CORDIC blocks chained together directly.

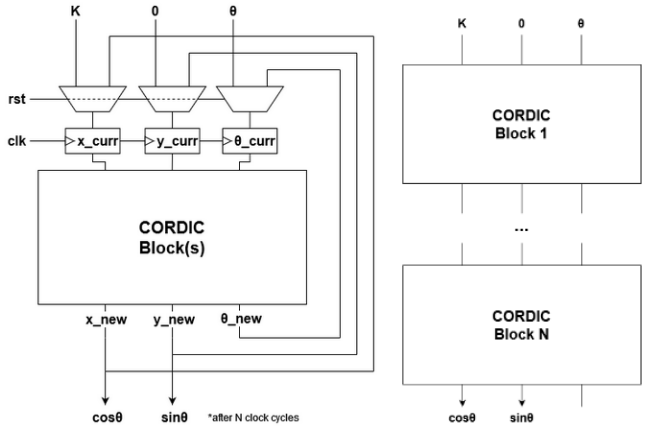

Fig. 4: Clocked vs. Fully Combinational CORDIC

2) Testing: We tested the fully combinational implementation and the clocked implementation by evaluating 1 and

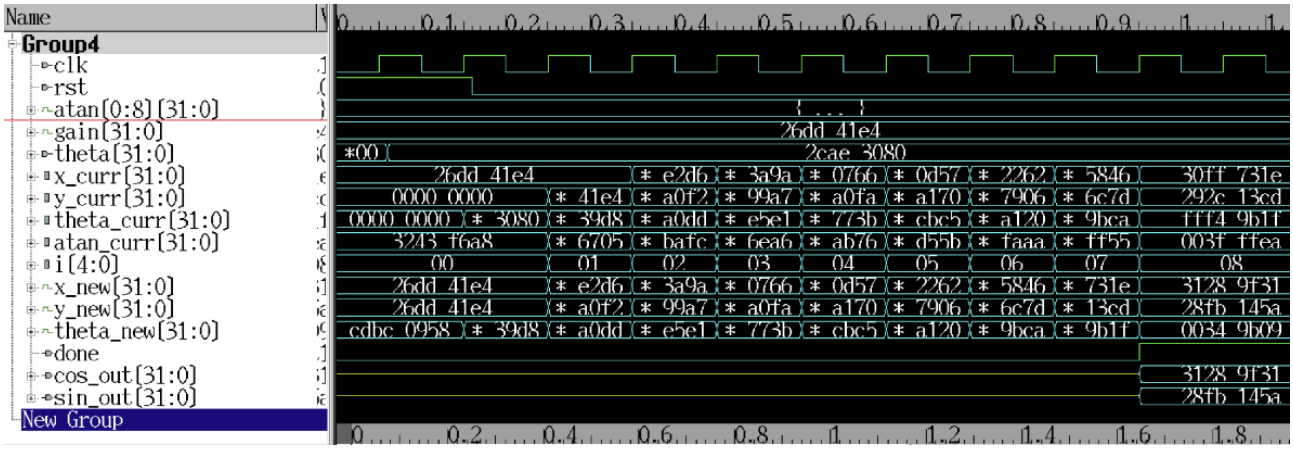

Fig. 5: Detailed Clocked (9 terms) CORDIC Waveform

2 blocks per cycle. Using the software model we created at the beginning of this project, we were able to generate a large set of test inputs and expected results. Additionally, we created a Python script that generates the Verilog files for a given number of CORDIC terms so that we could efficiently measure PPA scaling. We used Verilog simulations to verify the behavior of our design. Figure 6 shows the test bench view of each module and figure 5 shows a detailed look at the waveform for the clocked (1 block) implementation).

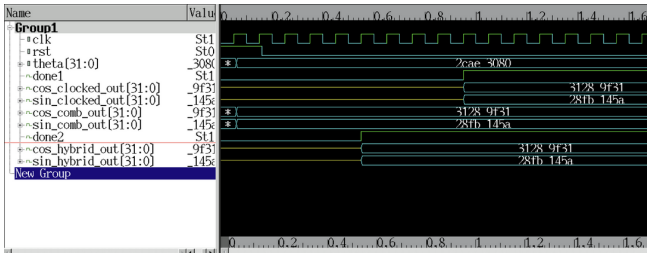

Fig. 6: Top Level CORDIC Waveform

3) Synthesis: Using the 45nm technology node from FreePDK45, we used Design Vision to synthesize each design multiple times using a different number of CORDIC terms. The schematic shown in figure 7 shows the schematic of the clocked implementation with 13 terms. From the synthesized RTL, we performed APR using Innovus to generate the physical layout; the layout for the 13-term clocked implementation is shown in figure 8.

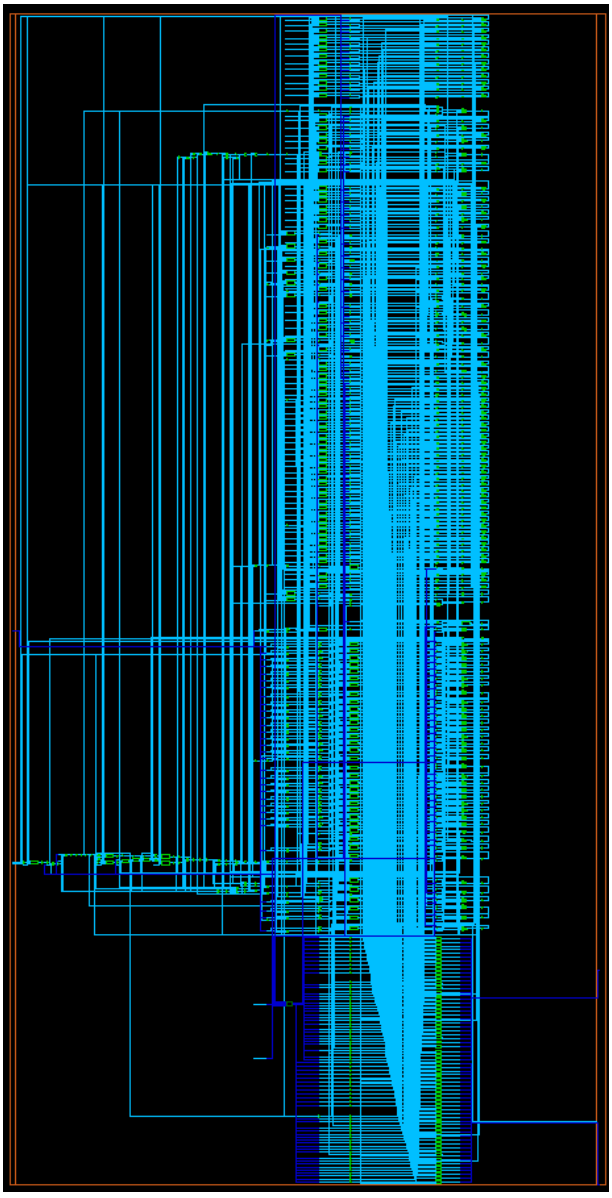

Fig. 7: CORDIC Schematic

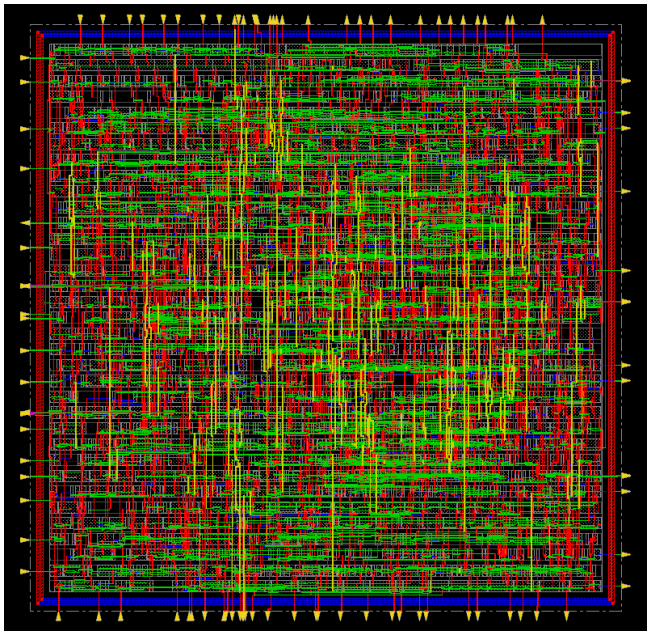

Fig. 8: CORDIC Physical Layout

V. Results

In this section, we detail the power, performance, and area characteristics of our modules, and provide a comparison for designs with approximately the same accuracy.

A. Taylor Series Accelerator

1) Performance: As shown in figure 9, the performance of our accelerator scales roughly linear to super linear. This is because we can compute terms in parallel at the cost of more functional units and use a tree to combine everything. We can also ”memoize” and reuse calculations across different terms to save on any redundant calculations. Because the four-term implementation resulted in a very small margin of error, we did not think it was necessary to compute more terms for scaling.

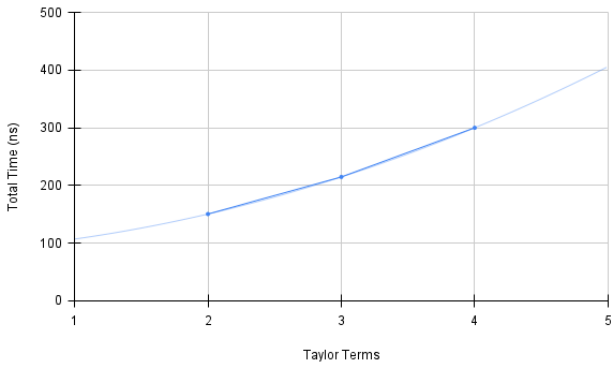

Fig. 9: Time (ns) vs. Number of Taylor Terms

2) Area: We see a similar linear trend in our area, as each additional term on average will require one to two additional multipliers and one to two additional adders. We do not show data for a singular term as that is a static ”one” and provides no value to our measurements.

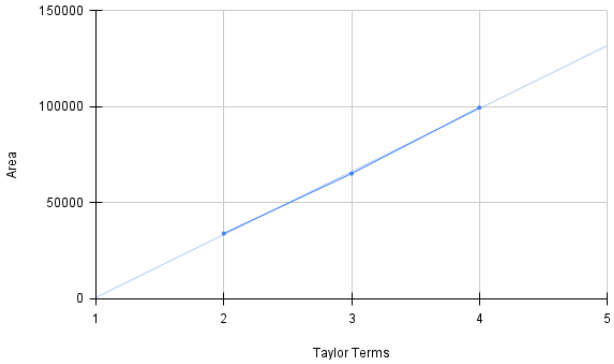

Fig. 10: Area vs. Number of Taylor Terms

3) Power: The power also scales nearly linearly, with most of the power requirement coming from powering all of the internal registers of the adders and multipliers.

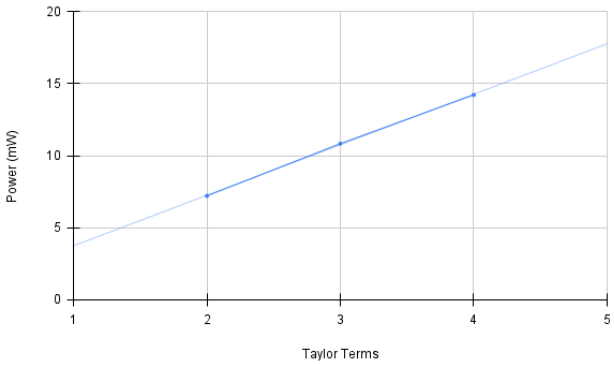

Fig. 11: Power (mW) vs. Number of Taylor Terms

B. CORDIC Algorithm

All of the following plots show how error falls relative to power, performance, or area when we increase the number of CORDIC terms. Error is on a log scale, showing that error falls exponentially as we increase the number of terms.

1) Performance: In figure 12, we show how evaluation time scales with the number of terms. Note that the time for the clocked graphs assumes the *minimum* clock period, and is computed as the number of cycles $\times$ that period. This means that, although we see similar scaling between all implementations, if we are constrained by our system’s clock period, the clocked time will scale up with the number of cycles. However, if we know our system will be clocked at a higher period, we can take advantage of this slack by evaluating multiple CORDIC blocks per cycle (e.g. the 2-block case). This will cut the number of evaluation cycles in half and, assuming this does not affect the system clock period, means we can compute the result in half the time. However, we still need to consider how this will impact power and area.

Avg Error (rad)	Taylor Terms	CORDIC Terms	Taylor Perf (ns)	CORDIC Perf (ns)	Taylor Area (nm^2)	CORDIC Area (nm^2)	Taylor Power (mW)	CORDIC Power (mW)
0.4 - 0.6	2	5	150.5	19.55	34005	7265	7.21	1.406
0.002 - 0.004	3	9	214.8	35.28	65144	7326	10.83	1.626
0.00005 - 0.0001	4	13	300.0	50.96	99458	7388	14.21	1.743

Table 1. PPA comparison of Taylor series and CORDIC implementations

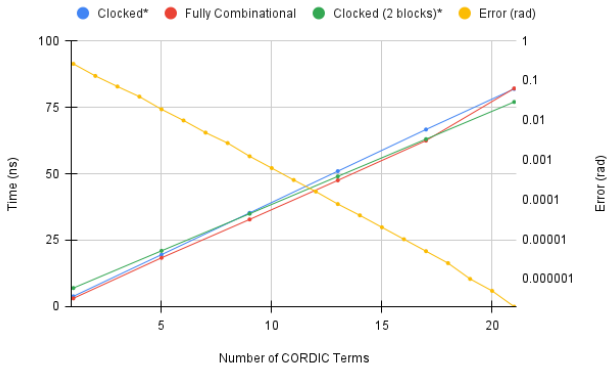

Fig. 12: Time (ns) vs. Number of CORDIC Terms

2) Area: Figure 13 plots area against the number of terms (note that area is on a log scale). We show that the fully combinational logic scales approximately linearly with the number of terms, which makes sense since each term needs the logic for a CORDIC block shown in figure 3. On the other hand, the clocked implementations stay relatively constant, with the only increases being the size of the arctan lookup table (1 entry per term) and the counter to determine when we are done computing the result.

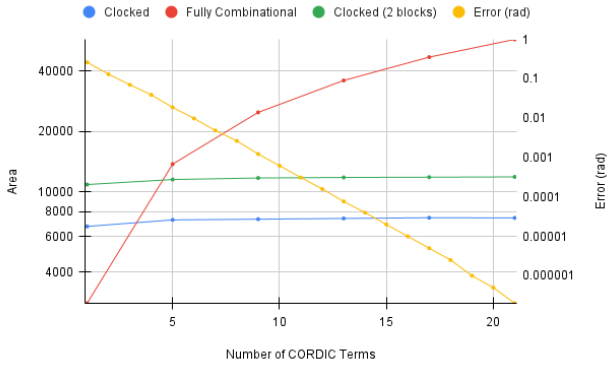

Fig. 13: Area vs. Number of CORDIC Terms

3) Power: Finally, figure 14 shows power scaling against the number of terms. Here we understandably see a similar scaling to area, since the main increase in power is due to increasing the size of combinational logic. However, we notice that evaluating 2 blocks per cycle has a *lower* power consumption than 1 block per cycle. This is because we need to run 2 blocks at a slower clock frequency to ensure we do not have any timing violations, which reduces register switching power considerably. At this stage, this decrease in power offsets the power from increased combinational logic, which is why it uses less power than the single block version.

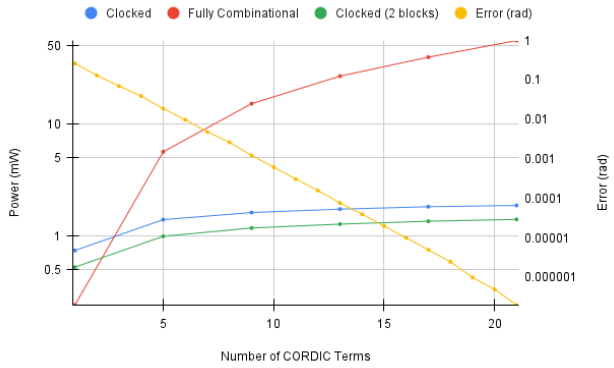

Fig. 14: Power (mW) vs. Number of CORDIC Terms

C. Comparison

In table 1, we show the PPA comparison of the two implementations. We selected the number of Taylor and CORDIC terms such that their average error was approximately the same. We notice much better results across the board, showing the superiority of the CORDIC algorithm in a hardware system.

VI. Team Contributions

We divided this project to have one member (Jason) take ownership of the Taylor Series Implementation and the other member (Sid) take ownership of the CORDIC implementation. This approach allowed us to modularize and parallelize our work. This division of labor is also reflected in the slide deck and report, with each member responsible for their respective sections.

VII. Conclusion

In this project, we implemented functional units to calculate the Taylor Series and CORDIC approximations of common trigonometric functions, sine and cosine. We modeled the implementations in low-level software, implemented in Verilog Hardware Description Language, and then used industry-standard Cadence tools to generate netlists, determine power, performance, and area metrics, and run automatic place and route to generate an on-chip implementation. Through our generated results, we can see that the CORDIC implementation had overall lower power, performance, and area relative to the Taylor series implementation when compared using similar ranges of average error. The CORDIC design also exhibits better scaling to larger more accurate designs, using more

cycles to generate another iteration as opposed to a completely different term. Additionally, our CORDIC design is easily configurable to calculate other trigonometric functions, while the Taylor Series design is made specifically for cosine and would need to be completely remade for the other functions.

REFERENCES

[1] J. E. Volder, "The CORDIC Trigonometric Computing Technique," IRE Trans. on Electronic Computers, 1959, pp. 330-334.

[2] Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing," in IEEE Signal Processing Magazine, vol. 9, no. 3, pp. 16-35, July 1992, doi: 10.1109/79.143467.

[3] Michael Andrews, "Influence of architecture on numerical algorithms", Microprocessors, vol.2, no.3, pp.130, 1978.

[4] Michael Andrews, Thomas Mraz, "Unified elementary function generator", Microprocessors, vol.2, no.5, pp.270, 1978.

[5] Michael Andrews, Daniel A. Eggerding, "A pipelined computer architecture for unified elementary function evaluation", Computers & Electrical Engineering, 1978

[6] E. Manor, A. Ben-David, and S. Greenberg, "Cordic hardware acceleration using DMA-based ISA extension," Journal of Low Power Electronics and Applications, vol. 12, no. 1, p. 4, 2022. doi:10.3390/jlpea12010004