Project 4: Implementing a CORDIC module in VivadoHLS

Jennifer Hellar

ELEC 522 - Fall 2018

Introduction

In this project, I implemented a dual-mode CORDIC module in VivadoHLS that can perform either vector rotation (*Equations 1 & 2*) or arctangent and magnitude computations (*Equations 3 & 4*).

Rotation Mode:

$$a = cos(z) * x - sin(z) * y \quad (1)$$

$$b = sin(z) * x + cos(z) * y \quad (2)$$

Arctan Mode:

$$a = \sqrt{x^2 + y^2} \quad (3)$$

$$c = arctan(\frac{y}{x}) \quad (4)$$

The module has four inputs:

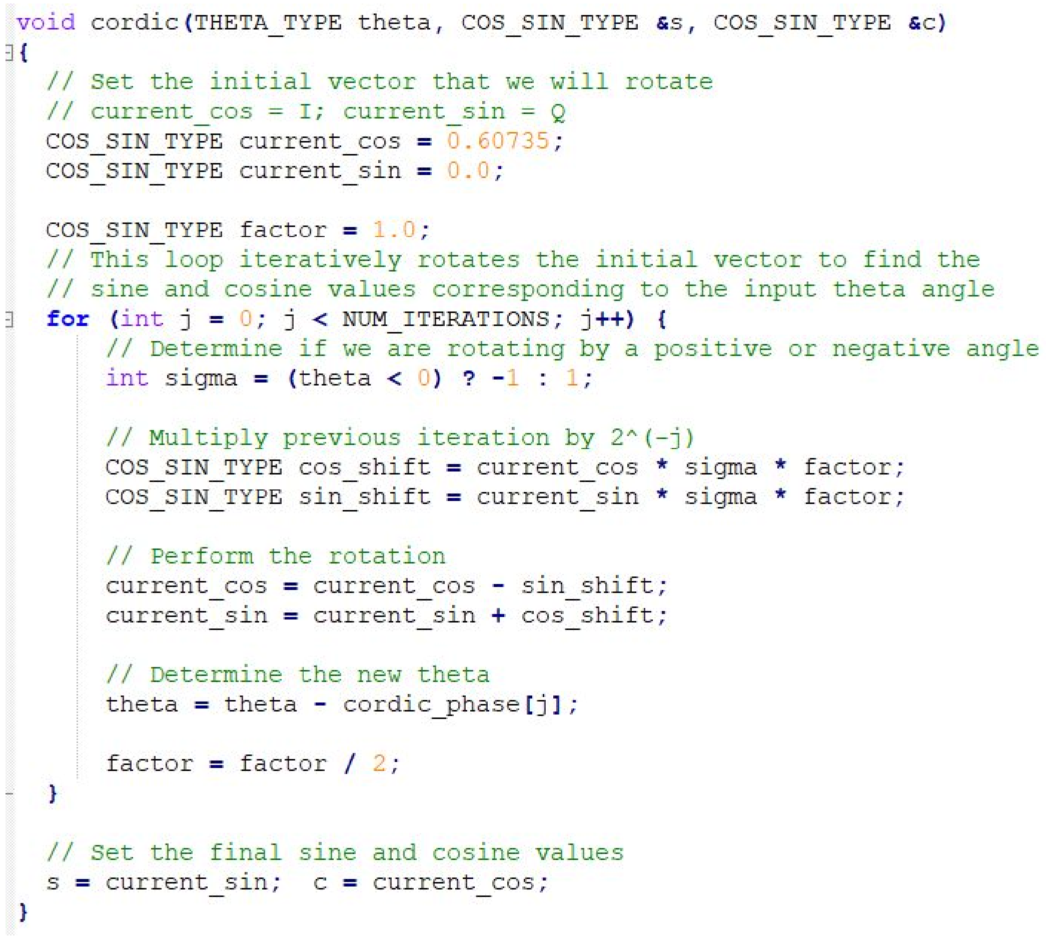
1. *mode* - boolean mode selection (rotation: 1, arctan: 0)
2. *x, y* - input vector
3. *z* - rotation mode: angle of rotation, arctan mode: 0

The three outputs are similar:

1. *a* - rotation mode: x value of resulting vector, arctan mode: magnitude of input vector
2. *b* - rotation mode: y value of resulting vector, arctan mode: 0
3. *c* - rotation mode: 0, arctan mode: angle of input vector

Vivado HLS: C++ Code Design

The initial code I started with was the example from the book shown in ***Figure 1*** below, which performed basic sin/cos computations using iterative rotations.



```
void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c)
{
    // Set the initial vector that we will rotate
    // current_cos = I; current_sin = Q
    COS_SIN_TYPE current_cos = 0.60735;
    COS_SIN_TYPE current_sin = 0.0;

    COS_SIN_TYPE factor = 1.0;
    // This loop iteratively rotates the initial vector to find the
    // sine and cosine values corresponding to the input theta angle
    for (int j = 0; j < NUM_ITERATIONS; j++) {
        // Determine if we are rotating by a positive or negative angle
        int sigma = (theta < 0) ? -1 : 1;

        // Multiply previous iteration by 2^(-j)
        COS_SIN_TYPE cos_shift = current_cos * sigma * factor;
        COS_SIN_TYPE sin_shift = current_sin * sigma * factor;

        // Perform the rotation
        current_cos = current_cos - sin_shift;
        current_sin = current_sin + cos_shift;

        // Determine the new theta
        theta = theta - cordic_phase[j];

        factor = factor / 2;
    }

    // Set the final sine and cosine values
    s = current_sin;  c = current_cos;
}
```

Figure 1: Book example code

Using this framework, I modified the datatypes, I/Os, and control logic to achieve my first mostly functional C++ design, shown in below.

```
// The file cordic.h holds definitions for the data types and constant values
#include "cordic.h"

void cordic(char mode, FIX_16_2 x, FIX_16_2 y, FIX_16_2 z, FIX_16_2 &a, FIX_16_2 &b,
FIX_16_2 &c)
{
        // Initial vector that we will rotate by angle
        FIX_16_2 current_x;
        FIX_16_2 current_y;
        FIX_16_2 angle;

        FIX_16_2 theta = 0;
        FIX_16_2 x_shift, y_shift = 0;
        FIX_16_2 phase;
```

```
char factor;
char region = 0;

if(mode){
        /* Rotation mode: z = angle (in radians) to rotate [x,y]' vector */
        if(z < -pihalf){
                angle = z + pi;
                region = 1;
        }
        else if(z > pihalf){
                angle = z - pi;
                region = 1;
        }
        else{
                angle = z;
                region = 0;
        }
        current_x = x;
        current_y = y;
        theta = 0;
        RotationLoop: for (int k = 0; k < 32; k++){
                // Multiply by 2^(-k) at each iteration
                x_shift = (current_y >> k);
                y_shift = (current_x >> k);
                phase = cordic_phase[k];

                // Perform rotation by pos or neg angle
                if(theta < angle){
                        current_x = current_x - x_shift;
                        current_y = current_y + y_shift;
                        theta = theta + phase;
                }
                else if(theta > angle){
                        current_x = current_x + x_shift;
                        current_y = current_y - y_shift;
                        theta = theta - phase;
                }
        }

        /* Approximate scale factor correction */
        ScaleLoop: for (int i = 0; i < 4; i++){
                factor = scaling_factors[i];
                if(i == 0 || i == 2){
                        current_x = current_x - (current_x>>factor);
                        current_y = current_y - (current_y>>factor);
                }
                else{
                        current_x = current_x + (current_x>>factor);
                        current_y = current_y + (current_y>>factor);
                }
        }
        if(region){
                a = -current_x;         // Rotation by more
                b = -current_y;
        }
        else{
                a = current_x;          // Rotation by +/- pi/2 or less
                b = current_y;
        }
        c = 0;
}
else{
        /* arctan mode: z = 0, a = |[x,y]'|, b = 0, c = theta */
        if(x < 0){
                if(y > 0){                              // Quadrant II
                        current_x = y;
                        current_y = -x;
                        region = 1;
                }
                else{
                        current_x = -y;         // Quadrant III
                        current_y = x;
                        region = 2;
                }
        }
        else{                                           // Quadrant I or IV
                current_x = x;
                current_y = y;
                region = 0;
        }
        theta = 0;
        AtanRotationLoop: for (int k = 0; k < 32; k++){
                // Multiply by 2^(-k) at each iteration
                x_shift = (current_y >> k);
                y_shift = (current_x >> k);
                phase = cordic_phase[k];

                // Perform rotation by pos or neg angle
                if(current_y < 0){
                    current_x = current_x - x_shift;
                    current_y = current_y + y_shift;
                    theta = theta + phase;
                }
                else if(current_y > 0){
                    current_x = current_x + x_shift;
                    current_y = current_y - y_shift;
                    theta = theta - phase;
                }
            }
            AtanScaleLoop: for (int i = 0; i < 4; i++){
                factor = scaling_factors[i];
                if(i == 0 || i == 2)
                    current_x = current_x - (current_x>>factor);
                else
                    current_x = current_x + (current_x>>factor);
            }
            a = current_x;
            b = 0;
            if(region){
                if(region==1)
                    c = - (theta + pihalf);         // Quadrant II
                else
                    c = - (theta - pihalf);         // Quadrant III
            }
            else
                c = -theta;             // Quadrant I or IV
        }
    }
```

The latency and interval for this code after C Synthesis was 75, with an estimated clock of 7.43 ns. The approximate utilization was 1800 LUTs (3% of the total). Since this will be implemented as a QR decomposition module, the utilization was higher than I liked, so I redesigned the code to be shorter and more efficient.

A. First Improvement: C++ Code Redesign

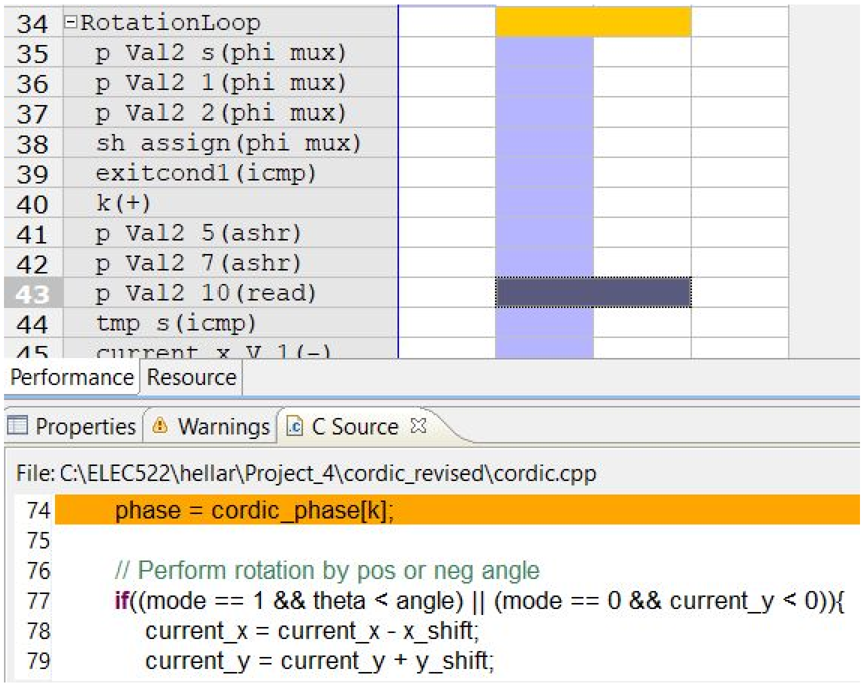
The original code I wrote was quite redundant in some areas, so for my first improvement, I combined several of the loops together and added some control logic, as shown below.

```
// The file cordic.h holds definitions for the data types and constant values
#include "cordic.h"

// The cordic_phase array holds the angle for the current rotation

void cordic(bool mode, FIX_16_3 x, FIX_16_3 y, FIX_16_3 z, FIX_16_3 &a, FIX_16_3 &b,
FIX_16_3 &c)
{
	// Initialize outputs to zero
	a = 0;
	b = 0;
	c = 0;
	FIX_16_3 current_x;
	FIX_16_3 current_y;

	/* angle is the angle of rotation (= z +/- pi if z is out of our normal range)
	 * theta tracks the accumulated cordic phases
	 *
	 * phase holds the current cordic phase taken from the array
	 * x_shift and y_shift are temporary variables to store the shifted (multiplied by 2^-k)
versions of x, y
	 * flag tracks initial conditions to determine necessary post-processing
	 */

	FIX_16_3 angle;
	FIX_16_3 theta = 0;
	FIX_16_3 x_shift, y_shift = 0;
	FIX_16_3 phase;
	char flag = -1;

	/* Pre-processing Control Logic */
	if(mode){						// Rotation
		if(z < -pihalf){			// If angle less than - pi/2, shift back into range and
set flag
			angle = z + pi;
			flag = 2;
		}
		else if(z > pihalf){		// If angle greater than pi/2, shift back into range
and set flag
```

```
                angle = z - pi;
                flag = 1;
            }
            else{                                       // Otherwise, normal rotation
                angle = z;
                flag = 0;
            }
            current_x = x;
            current_y = y;
        }
        else{                                       // Arctan mode
            if(x < 0){
                if(y > 0){                          // Quadrant II (pre-rotate into Quadrant I or
IV)
                    current_x = y;
                    current_y = -x;
                    flag = 3;
                }
                else{
                    current_x = -y;             // Quadrant III (pre-rotate into Quadrant I or
IV)

                    current_y = x;
                    flag = 4;
                }
            }
            else{                                   // Quadrant I or IV (normal calculation)
                current_x = x;
                current_y = y;
                flag = 0;
            }
        }

        /* Main loop */
        RotationLoop: for (int k = 0; k < 32; k++){
            // Multiply by 2^(-k) at each iteration
            x_shift = (current_y >> k);
            y_shift = (current_x >> k);
            phase = cordic_phase[k];

            // Perform rotation by pos or neg angle
            if((mode == 1 && theta < angle) || (mode == 0 && current_y < 0)){
                current_x = current_x - x_shift;
                current_y = current_y + y_shift;
                theta = theta + phase;
            }
            else{
                current_x = current_x + x_shift;
                current_y = current_y - y_shift;
                theta = theta - phase;
            }
        }

        bool negshift;
        char factor;

        /* Approximate scale factor correction */
        ScaleLoop: for (int i = 0; i < 4; i++){
            factor = scaling_factors[i];
            x_shift = current_x >> factor;
            y_shift = current_y >> factor;
            negshift = (i == 0 || i == 2);
            if(negshift)
                current_x = current_x - x_shift;
            else
                current_x = current_x + x_shift;

            if(negshift && (mode == 1))
                current_y = current_y - y_shift;
            else
                current_y = current_y + y_shift;
        }

        /* Post-processing adjustments and assignment to output */
        if(mode){                               // Rotation mode
            if(flag){
                a = -current_x;         // Flip signs if z was beyond -pi/2 < z < pi/2
                b = -current_y;
            }
            else{
                a = current_x;          // Normal case
                b = current_y;
            }
            c = 0;
        }
        else{                                   // Arctan mode (c is opposite the true angle of our
vector)
```

```
                a = current_x;
                b = 0;
                c = -theta;
                if(flag==3)
                                c = c + pihalf;         // Quadrant II
                if(flag==4)
                                c = c - pihalf;         // Quadrant III
        }
}
```

This code had a latency and interval of 74, an estimated clock period of 7.87 ns, and used approximately 1300 LUTs (~500 less than the original code). The iteration latency for each loop was 2 cycles, which I couldn't find a way to decrease. The limiting factor was reading in the next cordic phase, as shown in the Analysis view in ***Figure 2*** below.



```
34 ⊟RotationLoop
35   p Val2 s(phi mux)
36   p Val2 1(phi mux)
37   p Val2 2(phi mux)
38   sh assign(phi mux)
39   exitcond1(icmp)
40   k(+)
41   p Val2 5(ashr)
42   p Val2 7(ashr)
43   p Val2 10(read)
44   tmp s(icmp)
45   current x V 1(-)
```

File: C:\ELEC522\hellar\Project_4\cordic_revised\cordic.cpp

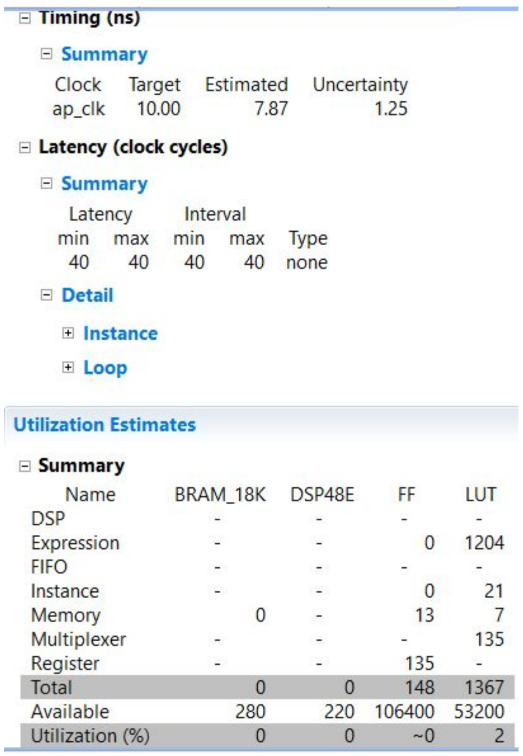
```
74        phase = cordic_phase[k];
75
76        // Perform rotation by pos or neg angle
77        if((mode == 1 && theta < angle) || (mode == 0 && current_y < 0)){
78            current_x = current_x - x_shift;
79            current_y = current_y + y_shift;
```

Figure 2: Rotation loop latency limited by 2-cycle read of cordic phase

Rotating around the unit circle in increments of 1°, the accumulated errors using the C testbench were: Total_Error_a=0.517808, Total_Error_b=0.450419, Total_Error_mag=0.396362, Total_Error_theta=0.097859. Using 32 iterations, the CORDIC module was quite accurate in both modes.

B. Second Improvement: Pipelining

After decreasing the LUT utilization, I focused on decreasing the latency and throughput. I pipelined both loops, resulting the Synthesis output shown below.


Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.87	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
40	40	40	40	none

Detail

Instance

Loop

Utilization Estimates

Summary

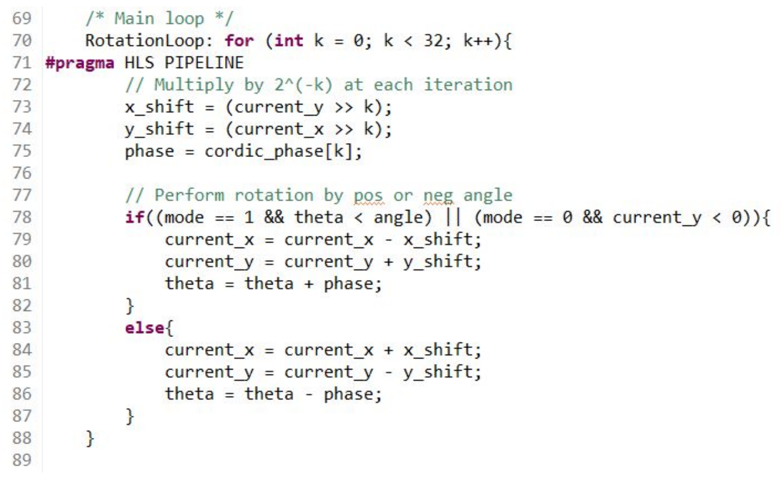
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1204
FIFO	-	-	-	-
Instance	-	-	0	21
Memory	0	-	13	7
Multiplexer	-	-	-	135
Register	-	-	135	-
Total	0	0	148	1367
Available	280	220	106400	53200
Utilization (%)	0	0	~0	2

Figure 3: Synthesis timing and utilization estimates

This design had good resource utilization (2% of LUTs and no BRAMs) as well as decent throughput, so I used this for my final design.

C. Final HLS Code Design

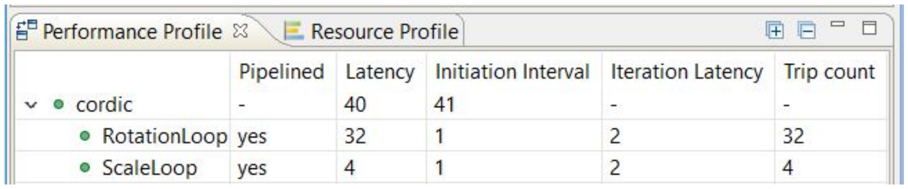
My final design (the code in Part A with additional pipelining pragmas) used an iterative approach, with a total of 32 iterations. The input and output data types were FIX_16_3 (signed 16-bit numbers with 13 fractional bits). Scale factor correction was done within the module using fixed bit shifts given by values stored in the array *scaling_factors*. The cordic micro-rotation values were likewise stored in the array *cordic_phase*. The main processing block of code was the rotation loop shown in ***Figure 4***, with control logic before and after based on the inputs and the mode selected.



```
    /* Main loop */
    RotationLoop: for (int k = 0; k < 32; k++){
#pragma HLS PIPELINE
        // Multiply by 2^(-k) at each iteration
        x_shift = (current_y >> k);
        y_shift = (current_x >> k);
        phase = cordic_phase[k];

        // Perform rotation by pos or neg angle
        if((mode == 1 && theta < angle) || (mode == 0 && current_y < 0)){
            current_x = current_x - x_shift;
            current_y = current_y + y_shift;
            theta = theta + phase;
        }
        else{
            current_x = current_x + x_shift;
            current_y = current_y - y_shift;
            theta = theta - phase;
        }
    }

```

Figure 4: Main rotation loop for final design

The performance profile for the function and the loops is shown in ***Figure 5***. As described above, both loops have an iteration latency of 2 because of having to read values from an array. I tried including the arrays as inputs with various interfaces, but nothing improved this result.



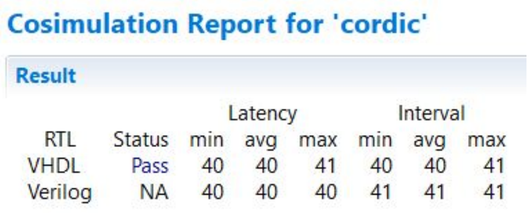
Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
cordic	-	40	41	-	-
RotationLoop	yes	32	1	2	32
ScaleLoop	yes	4	1	2	4

Figure 5: Performance profile (Analysis view)

The accumulated errors for each output in any mode while rotating all the way around the unit circle in 360 steps were each less than 0.52 (using the C testbench) so the design performed very well numerically.

It passed both VHDL and Verilog C/RTL cosimulation, as shown in ***Figure 6*** below.



Cosimulation Report for 'cordic'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	Pass	40	40	41	40	40	41
Verilog	NA	40	40	40	41	41	41

Figure 6: C/RTL cosimulation report

SysGen Design Flow

Using the HDL code described above, I exported the RTL for SysGen and imported into SysGen using the VivadoHLS block.

A. SysGen Model

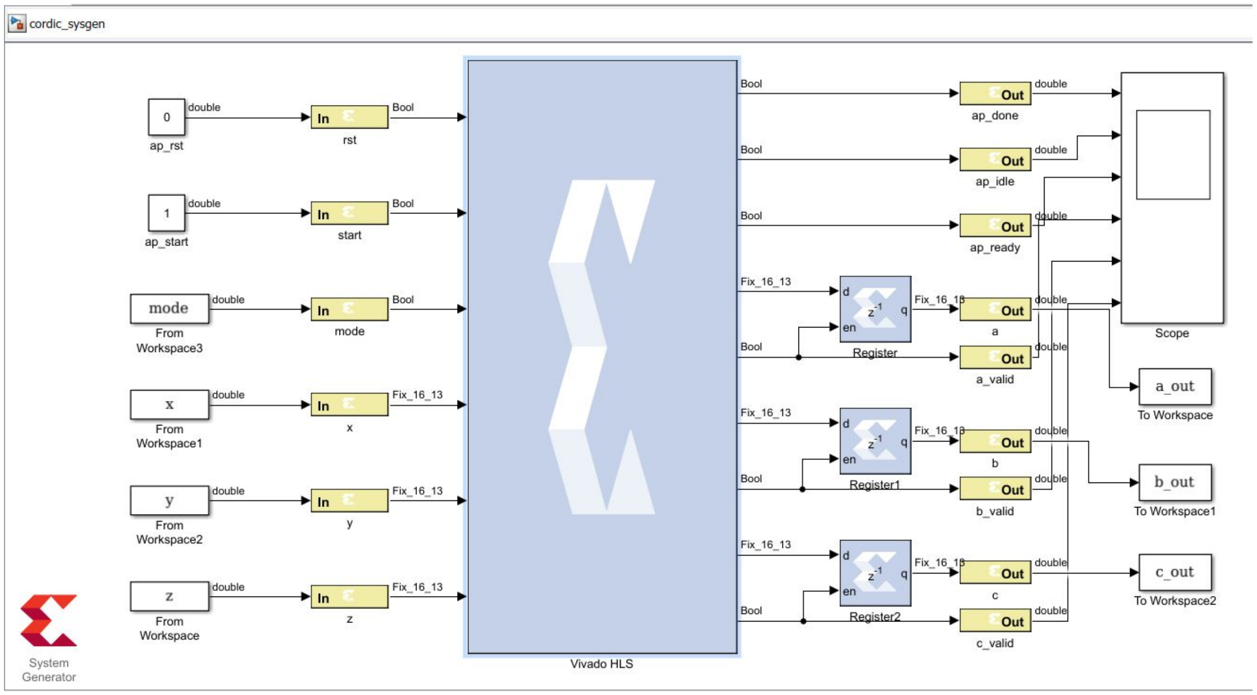


Figure 7: System Generator model

B. MATLAB Code & Simulation Procedure

The *load_data.m* file contains sections of MATLAB code which load data in for each mode and then calculate the error in the results. To test the module, run either Section 1 (rotation mode) or Section 3 (arctan mode), then run the Simulink model. Run the corresponding Section 2 (rotation mode) or Section 4 (arctan mode) to calculate the resulting errors.

C. SysGen Simulation Results

After performing the procedure described above for rotation mode, I received the outputs shown in ***Figure 8***, with average errors of 0.0015 for the x-coordinate and 0.0012 for the y-coordinate. The test started with the vector [1,0] and rotated it by -179 to 180 degrees.

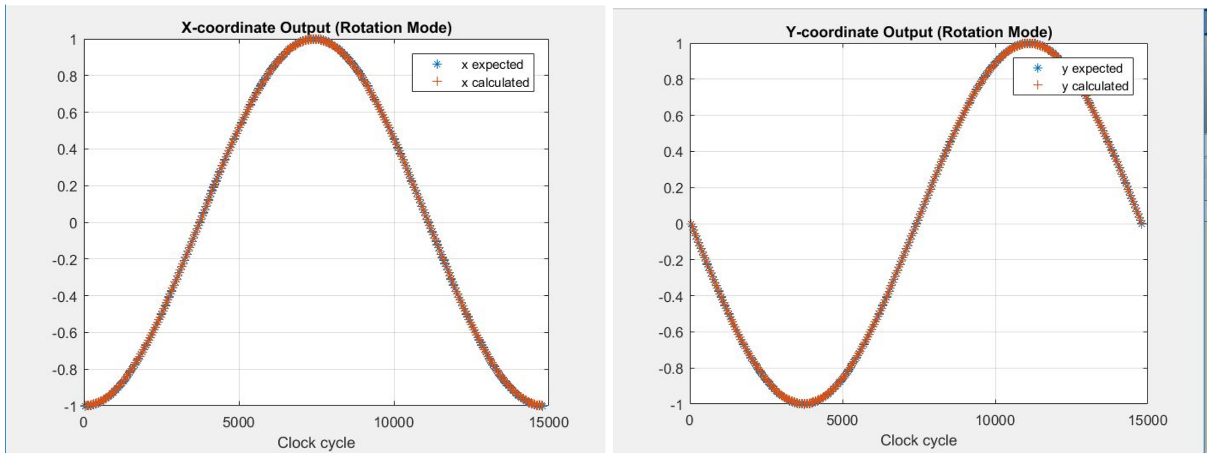


Figure 8: Rotation mode SysGen results

I repeated this with the arctan mode, and received the outputs shown in ***Figure 9***, with average errors of 0.0012 for the magnitude and 0.000319 for the angle.

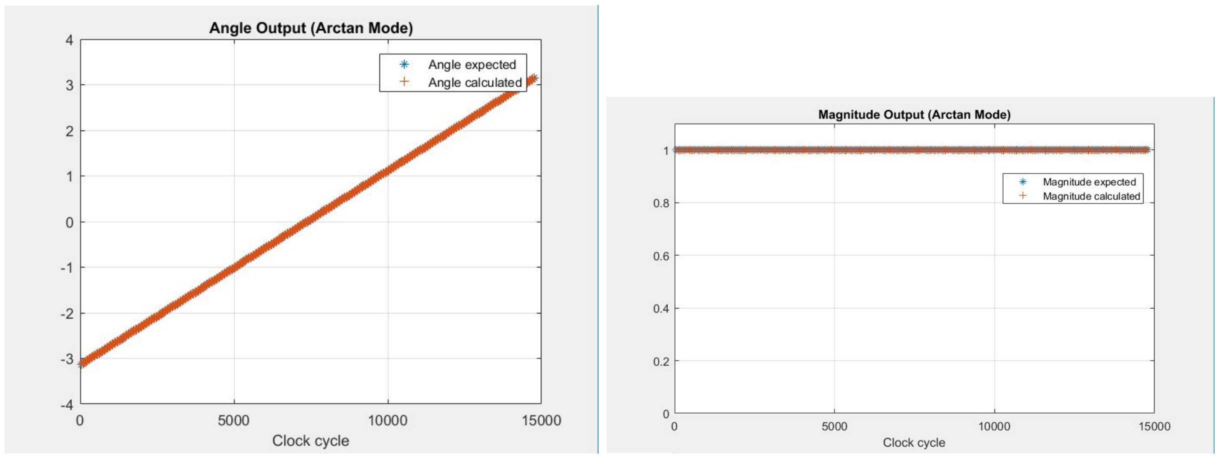


Figure 9: Arctan mode SysGen results

D. Hardware Cosim Results

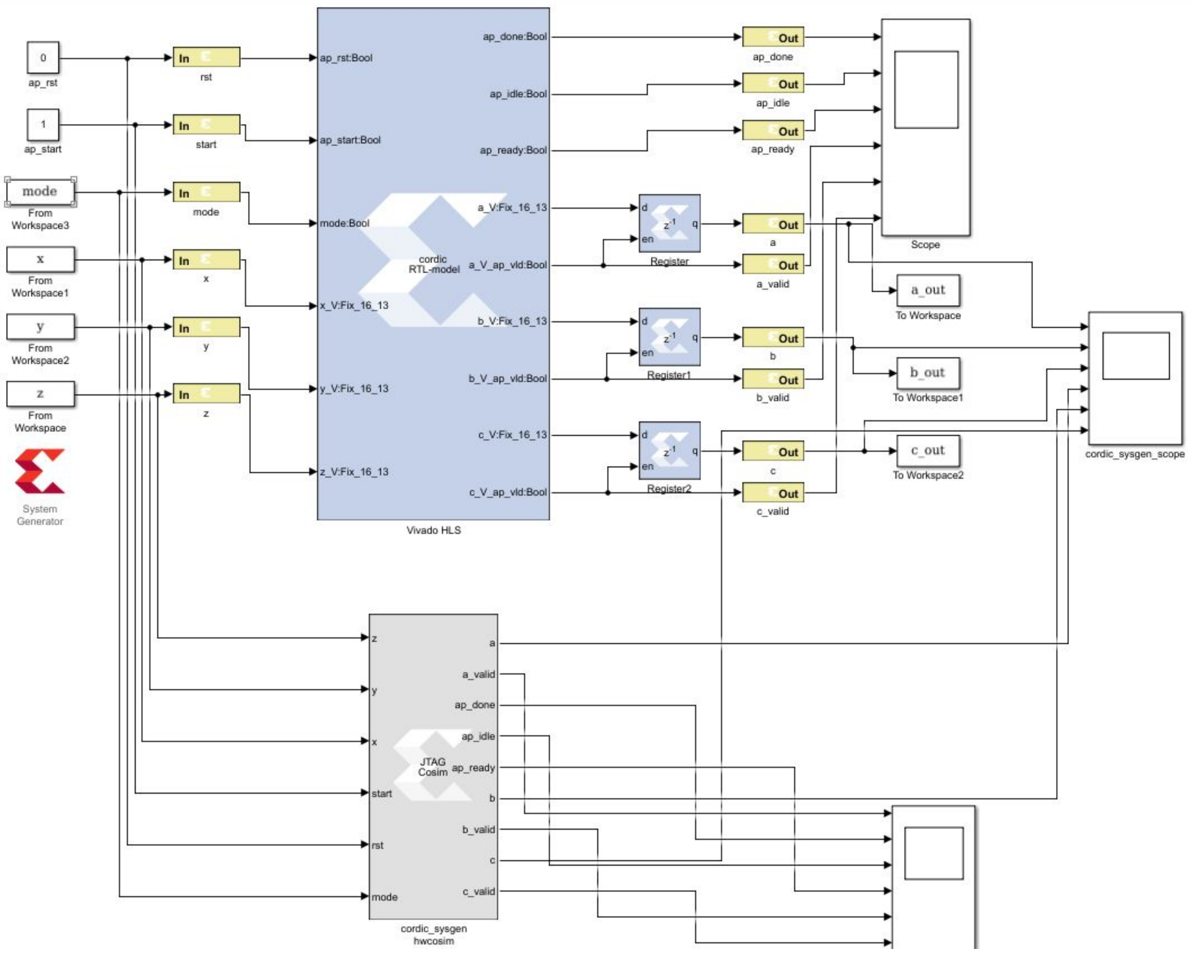


Figure 10: SysGen hardware cosim model

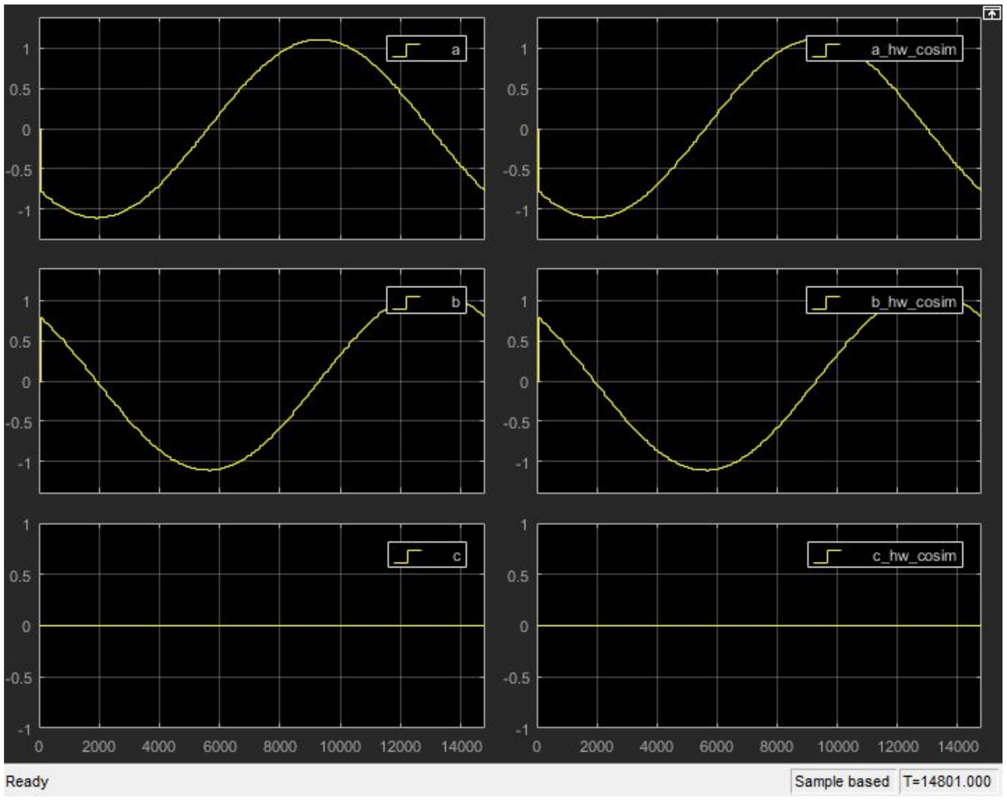


Figure 11: Rotation mode results for software (left) and hardware (right)

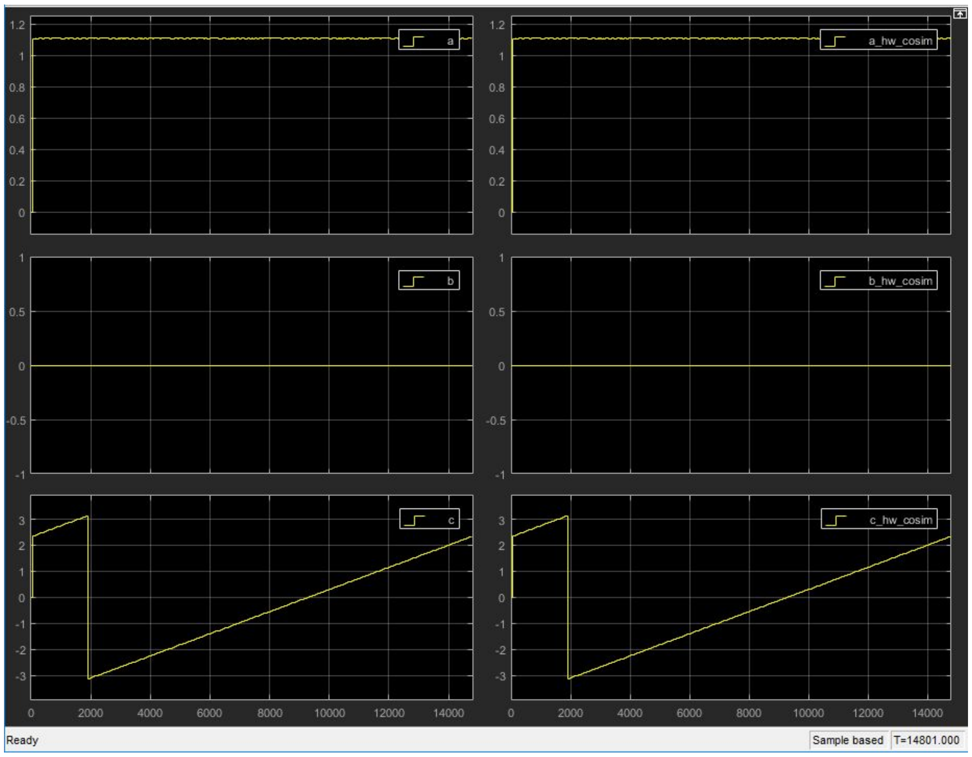


Figure 12: Arctan mode results for software (left) and hardware (right)

From the screenshots above, I verified my results in hardware around the whole unit circle.

E. Vivado Synthesis & Implementation Results

I exported by SysGen design into Vivado and ran synthesis and implementation with the following results.

Table 1: Vivado synthesis and implementation utilization and timing results (SysGen flow)

LUTs	596
Slices	180
I/O	106
DSPs	0
WNS (ns)	1.662
Max clock (MHz)	120

F. Comparison with CORDIC block performance

To compare our module's performance with the Vivado CORDIC module, I used the model shown in ***Figure 13***.

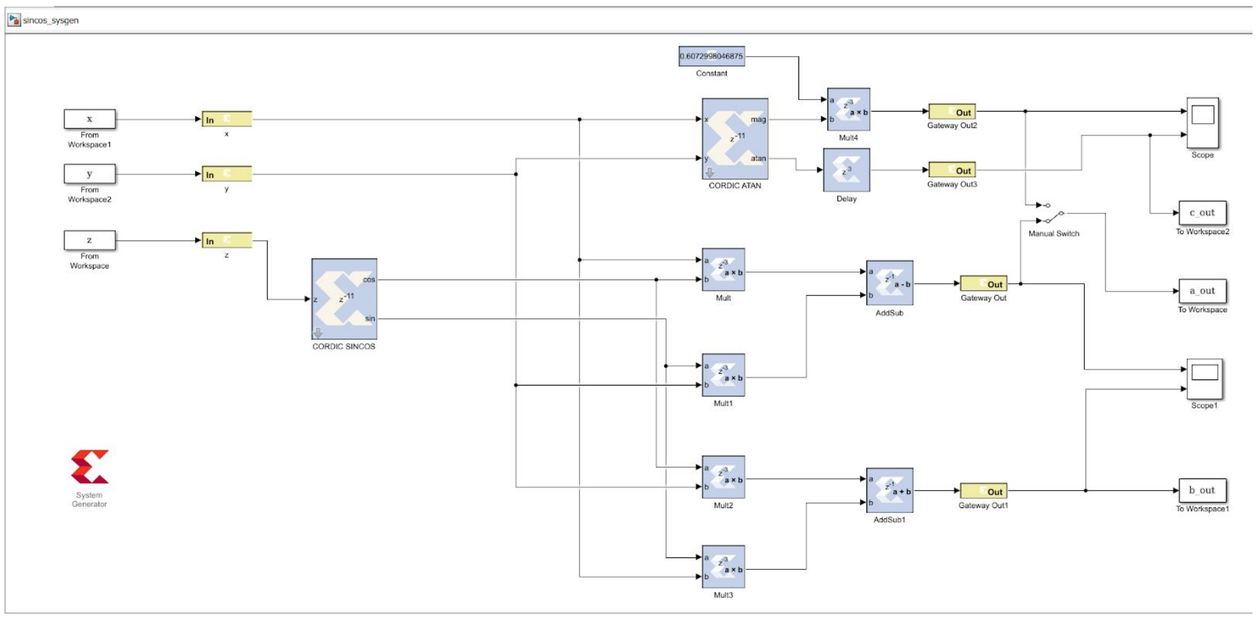


Figure 13: SysGen model for SINCOS and ATAN blocks

To test it in rotation mode, I ran section 5 of *load_data.m*, checked that the manual switch was down in the model, ran the model, then ran section 6 of the MATLAB code to calculate the average error. Note that both CORDIC blocks were set to use 32 iterations also.

For rotation mode, I found an average error of 0.0311 for the x coordinate and 0.0033 for the y coordinate, both of which were higher errors than my own CORDIC block achieved. For arctan mode, I ran sections 7 and 8 respectively, after switching the manual switch up. I received average errors of 0.000888 for the magnitude and 0.000125 for the angle respectively, which were both somewhat better than those of my own CORDIC block.

I also exported this into Vivado and obtained the following results for timing and utilization.

Table 2: Vivado utilization and timing results for design with Xilinx CORDIC blocks

LUTs	2203
Slices	722
I/O	113
DSPs	5
WNS (ns)	-45.414

From this, we see that those block have much higher resource utilization than ours and fails the timing requirement.

ASIC Design Flow

Next, I exported from HLS as an IP catalog, moved the resulting .v files onto CLEAR, edited the .tcl files to match my file names, and ran Design Compiler and Encounter. The highest clock frequency which still met design constraints was 200 MHz.

The Encounter output is shown in ***Figure 14***.

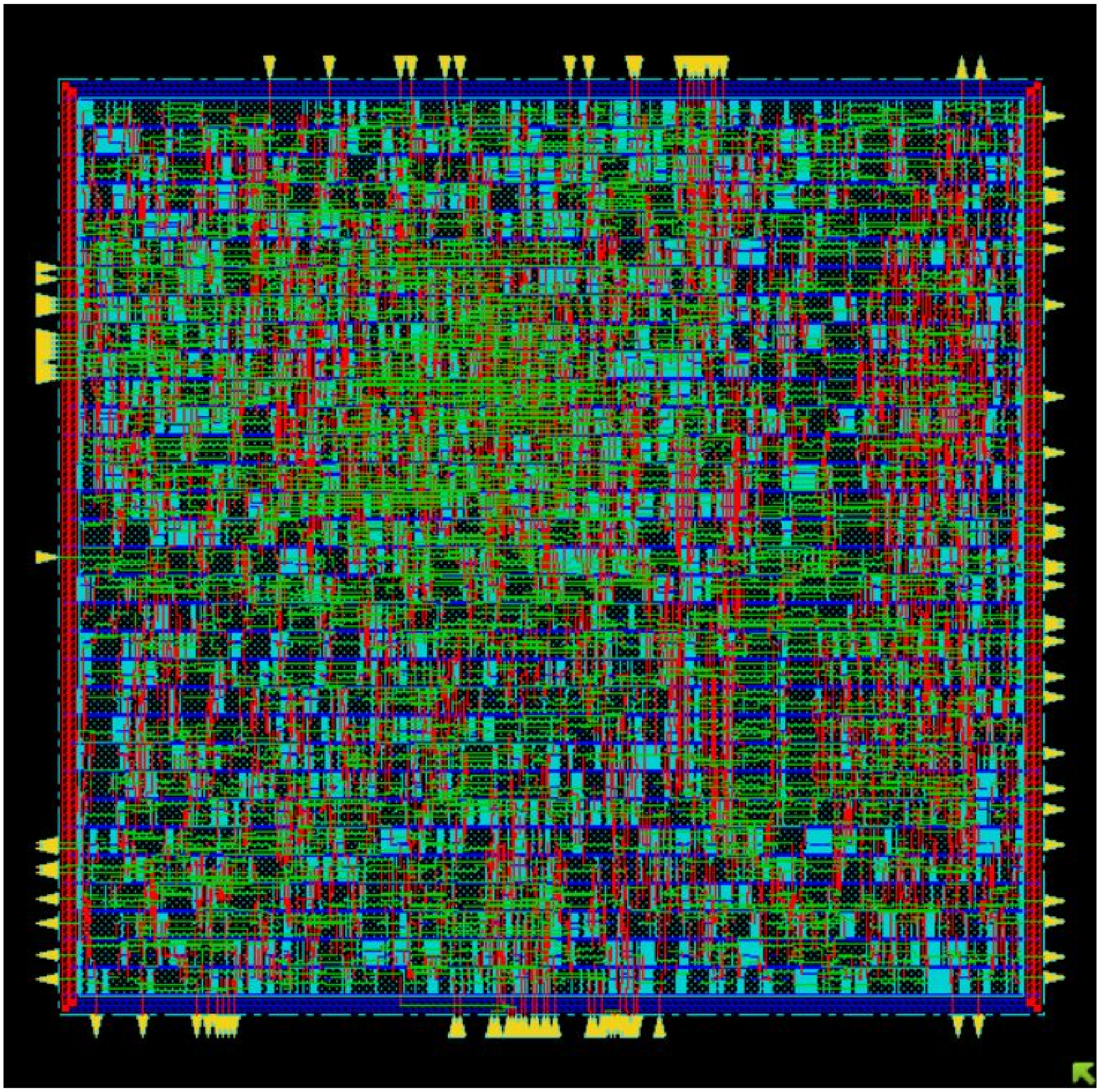
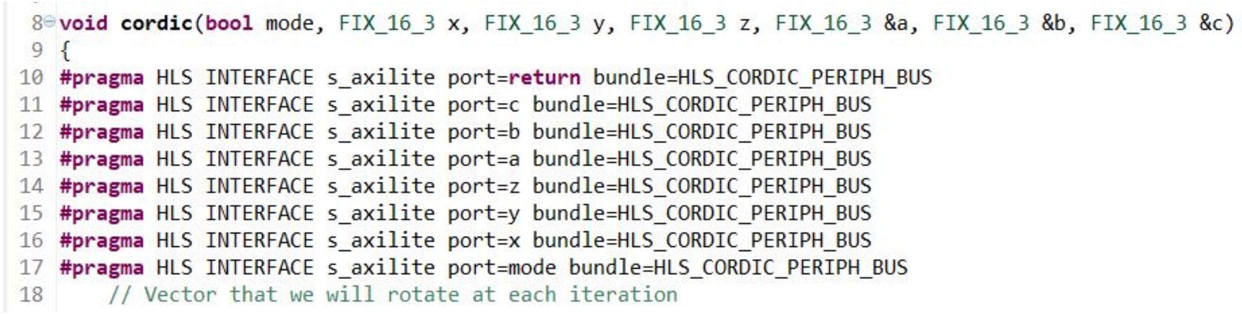


Figure 14: Encounter design

Vivado SDK Design Flow

Next, I edited the HLS file to include AXILITE interfaces as shown in ***Figure 15*** below.



```
void cordic(bool mode, FIX_16_3 x, FIX_16_3 y, FIX_16_3 z, FIX_16_3 &a, FIX_16_3 &b, FIX_16_3 &c)
{
#pragma HLS INTERFACE s_axilite port=return bundle=HLS_CORDIC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=c bundle=HLS_CORDIC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=b bundle=HLS_CORDIC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=a bundle=HLS_CORDIC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=z bundle=HLS_CORDIC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=y bundle=HLS_CORDIC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=x bundle=HLS_CORDIC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=mode bundle=HLS_CORDIC_PERIPH_BUS
    // Vector that we will rotate at each iteration
```

Figure 15: HLS code modification before exporting as IP

I integrated this IP package with the ARM core in Vivado as shown in ***Figure 16*** below.

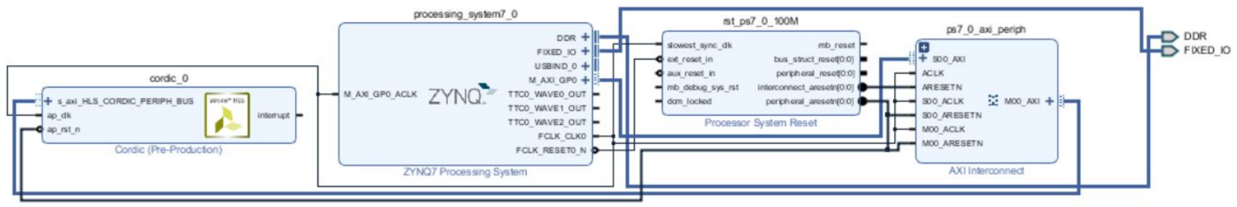
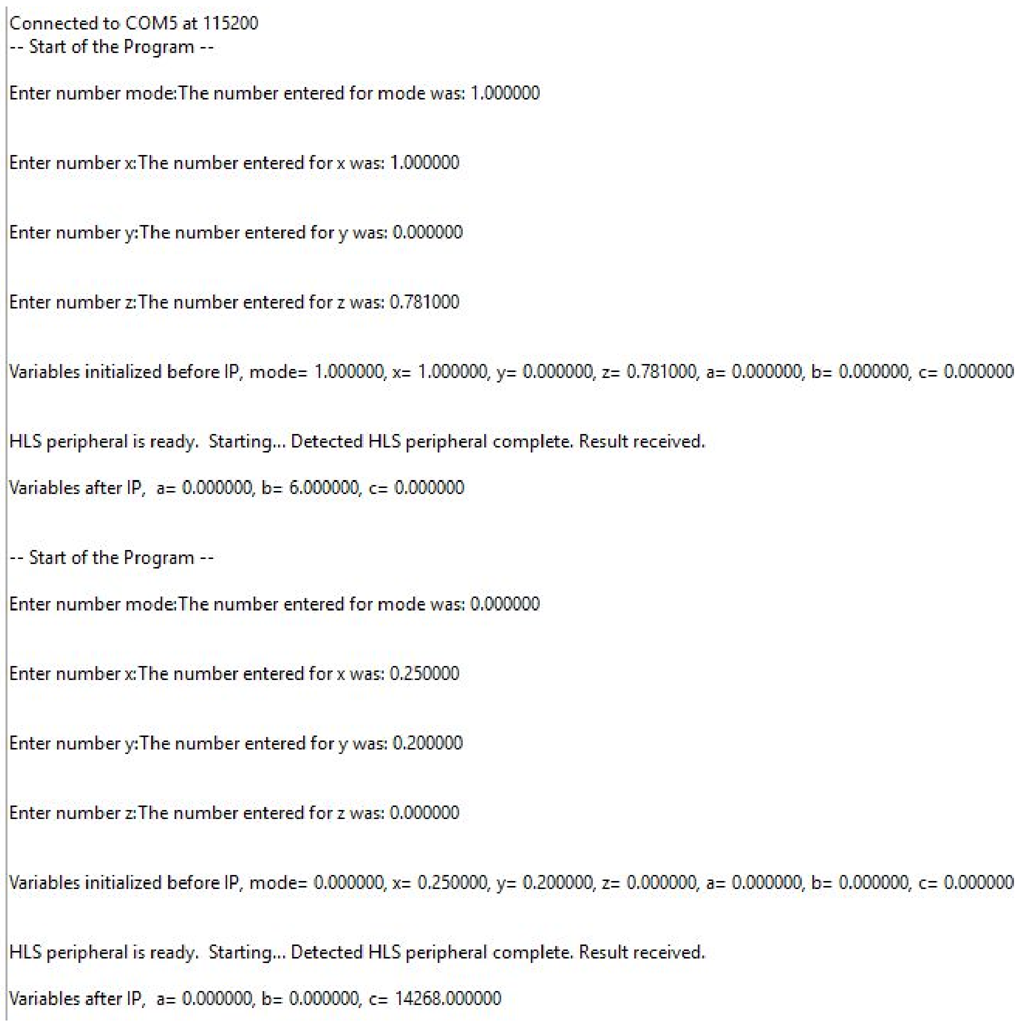


Figure 16: Vivado block diagram including cordic IP block and ARM core

I then verified and wrapped this design, exported the hardware, and launched the SDK. I imported the drivers, and starting with the *treeadd* template that was provided to us, I modified the *helloworld.c* file to communicate with and test our module, but it generated incorrect results, as shown in ***Figure 17***. I suspect that this is because the SDK testbench code does not support FIX_16_3 datatypes, so we are feeding in the data to our module incorrectly. I did not have time to try to find a way around this, unfortunately.



```
Connected to COM5 at 115200
-- Start of the Program --

Enter number mode:The number entered for mode was: 1.000000


Enter number x:The number entered for x was: 1.000000


Enter number y:The number entered for y was: 0.000000


Enter number z:The number entered for z was: 0.781000


Variables initialized before IP, mode= 1.000000, x= 1.000000, y= 0.000000, z= 0.781000, a= 0.000000, b= 0.000000, c= 0.000000


HLS peripheral is ready.  Starting... Detected HLS peripheral complete. Result received.

Variables after IP,  a= 0.000000, b= 6.000000, c= 0.000000


-- Start of the Program --

Enter number mode:The number entered for mode was: 0.000000


Enter number x:The number entered for x was: 0.250000


Enter number y:The number entered for y was: 0.200000


Enter number z:The number entered for z was: 0.000000


Variables initialized before IP, mode= 0.000000, x= 0.250000, y= 0.200000, z= 0.000000, a= 0.000000, b= 0.000000, c= 0.000000


HLS peripheral is ready.  Starting... Detected HLS peripheral complete. Result received.

Variables after IP,  a= 0.000000, b= 0.000000, c= 14268.000000
```

Figure 17: SDK terminal output