Design and Implementation of a Discrete FIR and CORDIC Conversion System

Alex Stepko — *axstepko.com*

The Pennsylvania State University
School of Electrical Engineering and Computer Science
Zheyu Li, PhD. Candidate

May 4, 2023

Report Contents

Acronyms

AXI Advanced eXtensible Interface

CORDIC coordinate rotation digital computer

FIFO first in first out

FIR finite impulse response

FPGA field programmable gate array

HLS high-level synthesis

IDE integrated development environment

IP intellectual property

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This report discusses the design, optimization, and physical implementation of a complex discrete signal processing system. The processing system involves a 1-D finite impulse response (FIR) filter in conjunction with a coordinate rotation digital computer (CORDIC) resultant in a hardware-optimized low-latency accelerator that can be used as an IP in any Advanced eXtensible Interface (AXI) compliant design. The processing chain consists of the following components:

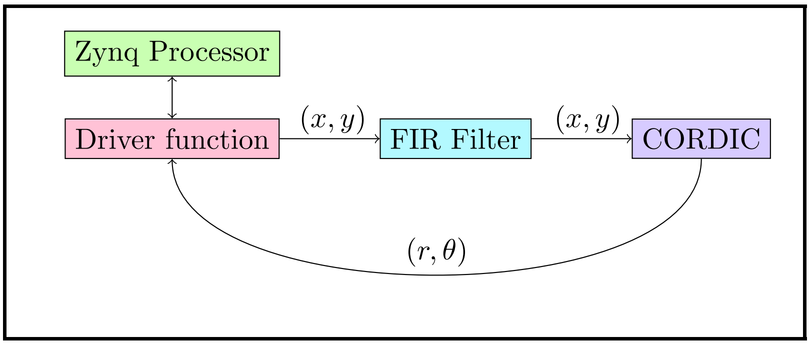


Figure 1.1: Signal processing chain

The processing chain computes a 1-D complex FIR filter on a dataset loaded to the accelerator in cartesian ((x, y)) format. The output of the FIR filter is also of the cartesian form. This cartesian dataset is streamed via a FIFO data channel into a CORDIC accelerator, which performs a conversion of the input vector ((x, y)) to the polar form ((r, θ)). This data is then directed back towards the Zynq processor, which displays the product on the console.

This processing chaing was developed in Vitis HLS, an advanced high-level synthesis (HLS) synthesis tool that allows seemingly linear C (or C++/Python) code into hardware-accelerated models targeted for implementation on an field programmable gate array (FPGA). This chain was then packaged as an intellectual property (IP), and used in a block design in Vivado. Hardware was synthesized and implemented in Vivado, and the exported hardware was then used in the Vitis integrated development environment (IDE) to finish integration of the accelerator into a complete system. The target hardware for this processing chain is a Xilinx Zynq-7000 series chipset, which is held on a Digilent/Avnet/National Instruments Zedboard development board.

Note to Developers: All source code for this project can be found on the GitHub repository (https://github.com/FPGACrashCourse/ComplexFIRFilter).

1.1 Processing chain

1.1.1 FIR filter

The FIR filter is a fundamental technique used in countless systems in the medical, defense, academic, and consumer sectors. Moreover, a high-performance FIR filter is almost always necessary in a signal processing chain. Generically, the formula for a complex FIR filter is

$$Y = \sum_{i=0}^{n} w_i * X[n-i] \tag{1.1}$$

for any arbitrary set of inputs. This is equivalent to a 1-D convolution. This can be extended to complex vectors, which results in an almost identical equation of the form

$$\overline{Y} = \sum_{i=0}^{n} w_i * \overline{X}[n-i] \tag{1.2}$$

where $\overline{Y}$, w_i, and $\overline{X}[n-i]$ are of the general complex form $\overline{Z} = \alpha + j\beta$ (in cartesian coordinates). While this result is desierable in most scenarios, a more useful form of the result exists in polar form, where generically the same vector can be represented as $\overline{Z} = (r, \theta)$ in polar coordinates. In standard trigonometry, this conversion is trivial. For the rectangular vector $\overline{Z} = \alpha + j\beta$, we find its polar form with the following equations

$$r = \sqrt{\alpha^2 + \beta^2} \tag{1.3}$$

$$\theta = \tan^{-1}\left(\frac{\beta}{\alpha}\right) \tag{1.4}$$

which results in the vector $\overline{Z} = (r, \theta)$. While the average individual can quickly compute these values with a calculator, the computational power required to reach these results is rather large. In a setting where realtime performance is necessary, these calculations are nearly impossible to achieve or require an unreasonably large amount of compute power.

1.1.2 CORDIC

A well-known method to perform trigonometry and coordinate system conversion is the CORDIC. This system iteratively rotates a vector using simple matrix multiplication to the desired target, and then applies a simple gain factor. Because it is multiplication on a base-two numerical system, it is possible to achieve this with a simple right-shift and addition/subtraction of the vector to rotate. The generic form for a CORDIC rotation is

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{1.5}$$

i	2^{-i}	Rotating Angle	Scaling Factor	CORDIC Gain
0	1.0	45.000°	1.41421	1.41421
1	0.5	26.565°	1.11803	1.58114
2	0.25	14.036°	1.03078	1.62980
3	0.125	7.125°	1.00778	1.64248
4	0.0625	3.576°	1.00195	1.64569
5	0.03125	1.790°	1.00049	1.64649
6	0.015625	0.895°	1.00012	1.64669

Table 1.1: CORDIC gains for the first six iterations of the algorithm

which is resultant in

$$x_i = x_{i-1}\cos\theta - y_{i-1}\sin\theta \tag{1.6}$$

and

$$y_i = x_{i-1}\sin\theta + y_{i-1}\cos\theta \tag{1.7}$$

being the rotated versions of the original vector $[x_{i-1}, y_{i-1}]$. This algorithm is iterative, and therefore it rotates by decrementing amounts towards the target angle. While the rotation correctly computes to the raw angle, it fails to correctly compute the converted magnitude. This is easily fixed by a single multiplication of the CORDIC gain factor. Gain factors are computed iteratively with the equation

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{1.8}$$

and

$$K = \lim_{n\to\infty} K(n) \approx 0.6072529350088812561694 \tag{1.9}$$

which are normally computed and stored in a lookup table for quick-reference without computing the result in realtime.

As noticed above, the CORDIC gain K_i increases with each iteration. If we compute K_i^{-1}, we then see that the gain decreases as iterations increase. This fact is only important in the sense that after a certain point of iterations, the gains are practically identical until the 10^{-9} decimal point. Essentially, beyond a certain number of iterations the precision of the algorithm plateaus at almost the exact correct value.

Chapter 2

Cartesian FIR Filter

The FIR filter is implemented in Vitis HLS, using equations shown in section 1.1.1 to compute the 1-D convolution. The filter inputs have been specified on a range of $[-50, 50]$, which includes the filter coefficients. The computation of a convolution is rather simple, taking into account the fact that we have complex numbers to multiply (the filter weights, and the raw data to be processed). Generically, for complex vectors

$$a = \alpha + j\beta \tag{2.1}$$

and

$$b = \rho + j\delta \tag{2.2}$$

we find their product $c = ab$ equivalent to

$$c = (\alpha\rho - \beta\delta) + j(\alpha\delta + \beta\rho) \tag{2.3}$$

This can be quickly translated into a processing algorithm shown below in Listing 2.5.

```
    delayLineImg[0] = inputImg;

    // Compute a pass of the filter
FILTER_TAPS:
    for (int j = 0; j < FILTER_SIZE; j++)
    {
#pragma HLS UNROLL
```

Listing 2.1: Convolution algorithm

This algorithm is then instantiated across the whole dataset with the following function behavior:

```
COMPUTE:
    for (int k = 0; k < filterLength; k++)
    {
#pragma HLS PIPELINE
      FIR_INT_INPUT inTempReal = FIR_INT_INPUT(inputReal[
          FILTER_SIZE + k]);
```

```
        FIR_INT_INPUT inTempImg = FIR_INT_INPUT(inputImg[
            FILTER_SIZE + k]);

          // Perform a single pass of an input with the
              coefficients:
          computeComplexFIR(inTempReal, inTempImg, kernelReal,
              kernelImg, &tempR, &tempI);
#ifdef FIR_DEBUG_MODE
        printf("complexFIR: result = %d + j%d\n", tempR.to_int
            (), tempI.to_int());
#endif
          outputReal.write(tempR);
          outputImg.write(tempI);
      }
```

Listing 2.2: Filter pass control

2.1 Performance of the algorithm

The performance of the algorithm can be broadly categorized into two major components:

Latency The computational time (clock cycles) required to process a valid result for a given input dataset.

Utilization The amount of resources required to compute the result at a certain latency.

Both of these metrics provide a general assessment of a given algorithm's performance. A high-efficiency algorithm minimizes utilization, and a high-performance algorithm minimizes latency. The combination of these two results in an ideally performing algorithm.

Latency

As emphasized in the introduction, the performance of the algorithm is a critical factor in its overall usability as a signal processor. One such metric of performance for the FIR filter is the computational latency necessary to compute a filter pass. We can estimate the optimized algorithm's latency prior to optimization with some optimizations:

1. Element access (read/write) takes 1 cycle
2. Addition takes 2 cycles
3. Multiplication takes 4 cycles

If we consider the filter pass shown in Listing 2.5, it is possible to compute the number of cycles Q. Q can be defined as

$$Q = X_i + A_i + M_i \tag{2.4}$$

where X_i, A_i, M_i are defined as

$$X_i = \sum_{i=0}^{numTaps} 1 = numTaps \tag{2.5}$$

$$A_i = \sum_{i=0}^{numTaps} 2 = 2 * numTaps \tag{2.6}$$

$$M_i = \sum_{i=0}^{numTaps} 4 = 4 * numTaps \tag{2.7}$$

and subsequently

$$Q = (1 + 2 + 4)numTaps = 7 * numTaps \tag{2.8}$$

Assuming a filter size of $numTaps = 25$, we determine the latency to be approximately $Q = 175$ cycles. This is, of course, just a single pass of the filter on a single datapoint, so multiple datapoints takes many more cycles. This has significant impact on the performance of the processing chain, and drastically increases system latency for a valid result. We synthesize an un-optimized filter using standard `int` datatypes within the system. The utilizations results below in Figure 2.1.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
polarFir				-	176	2.640E3	-	139	-	dataflow	0	397	48334	50864	0
entry_proc				-	0	0.0	-	0	-	no	0	0	3	38	0
Loop_FILTER_INIT_proc				-	36	540.000	-	36	-	no	0	0	1749	436	0
Block_for_end_proc				-	0	0.0	-	0	-	no	0	0	1602	461	0
complexFIR				-	138	2.070E3	-	138	-	no	0	396	28701	27884	0
COMPUTE				-	136	2.040E3	113	1	25	yes	-	-	-	-	-
computeComplexFIR				-	101	1.515E3	-	1	-	yes	0	396	23350	24768	0
bulkCordicConvert				-	45	675.000	-	45	-	no	0	1	1078	4846	0

Figure 2.1: Un-optimized utilization report

Note the high number of LUTs, FF, and DSPs necessary to run the design. The incredibly high utilization for the system has two problems. 1.) The design cannot be synthesized on the desired hardware target, and that chipset does not have enough resources. 2.) If enough hardware were available, the latency of over 130 cycles for the dataset is unacceptable for a realtime high performance system.

2.2 Optimization

To improve the performance of this algorithm, we leverage the `#pragma HLS` collection of compiler directives. These directives suggests to the compiler that it should rewrite or restructure the compiled result to improve the performance of the algorithm.

2.2.1 Unrolling

Perhaps one of the most basic operations employed in the HLS environment is loop unrolling. Loop unrolling exploits parallelism on the device, and is commonly paired with `for` loop bodies. The behavior of the process effectively duplicates copies of the arguments inside of the loop, and reduces the iteration depth of the loop itself. This optimization is applied below in Listing 2.3.

```
80     FILTER_INIT:for (int i = 0; i < FILTER_SIZE; i++)
81     {
82 #pragma HLS UNROLL
83         kernelReal[i] = FIR_INT_INPUT(inputReal[i]);
84         kernelImg[i] = FIR_INT_INPUT(inputImg[i]);
85     }
```

Listing 2.3: Filter coefficient initialization

`#pragma HLS UNROLL` in line 56 (Listing 2.3) completely unrolls the loop by a factor of `FILTER_SIZE` (which in this design is 25 taps), and as such reduces the computational latency to 1 cycle. This drastically reduces the computational latency required to calculate the result, and allows the core of the algorithm to start more quickly.

Another area this optimization is used is in the shift register within a computation of a filter pass itself. The shift register allows the system to exploit pipelined behavior, and allows data to be accessed in parallel. This is shown below in Listing 2.4.

```
    printf("computeComplexFIR: Recieved input %d + j%d\n",
        inputReal.to_int(), inputImg.to_int());
#endif

PIPELINE_DELAY:
    for (int i = FILTER_SIZE - 1; i >= 1; i--)
    {
#pragma HLS UNROLL
        // Iterate backwards through the array to shift to
            the right.
        delayLineReal[i] = delayLineReal[i - 1];
        delayLineImg[i] = delayLineImg[i - 1];
#ifdef FIR_DEBUG_MODE
```

Listing 2.4: Shift register

2.2.2 Pipelining

While the ability to unroll a loop is useful, the parallelism exploited by unrolling a loop requires more discrete hardware. Resources like DSPs on an FPGA are incredibly precious, and as such it is important to ensure they are used to their maximum potential.

A commonplace method to increase throughput of a design in general is to pipeline the computational chain. This behavior is seen in nearly every modern-day processor, and enables high average resource utilization across the whole system. Of course, this also requires the system to be designed in such a manner conducive to pipeline architecture.

In the case of the FIR filter, it is possible to pipeline the design to achieve near-single-cycle latency. This is accomplished via the shift register implemented in Listing 2.4, and the usage of `#pragma HLS PIPELINE`.

It should be noted that the ability for a design to be pipelined also depends on parallel memory element access, and later sections will discuss this in greater detail.

`#pragma HLS PIPELINE` effectively converts loops that are a deterministic length at compile time into a pipelined design. At compile time, this precompiler directive attempts to achieve an iteration interval of 1 (`II = 1`), if this is not possible, it attempts to find the shortest-possible iteration interval that successfully pipelines the design.

For example, consider the filter pass:

```
COMPUTE:
    for (int k = 0; k < filterLength; k++)
    {
#pragma HLS PIPELINE
        FIR_INT_INPUT inTempReal = FIR_INT_INPUT(inputReal[FILTER_SIZE + k]);
        FIR_INT_INPUT inTempImg = FIR_INT_INPUT(inputImg[FILTER_SIZE + k]);

          // Perform a single pass of an input with the coefficients:
          computeComplexFIR(inTempReal, inTempImg, kernelReal, kernelImg, &tempR, &tempI);
#ifdef FIR_DEBUG_MODE
        printf("complexFIR: result = %d + j%d\n", tempR.to_int(), tempI.to_int());
#endif
          outputReal.write(tempR);
          outputImg.write(tempI);
    }
```

Listing 2.5: Convolution algorithm

Without the precompiler directive `#pragma HLS PIPELINE`, this loop takes a large amount of time to compute. This amount is estimated in equation 2.8.

Upon re-synthesis of the design with the precompiler directive the loop's performance is improved dramatically, reducing the number of cycles necessary to compute a result to an iteration interval of 1 (`II = 1`). This is a dramatic improvement in comparison to the old design. Additionally, the efficiency of the design also ensures that resources allocated for that element of the processing chain are at a maximal utilization.

2.2.3 Dataset manipulation

Perhaps not exactly "optimization", dataset manipulation in the HLS environment is the second step (first is a good design) to structure code for optimization. Standard C/C++

languages transfer data between functions as linear allocation. Consider an array `X` which contains N elements. The array is typically organized in memory as a sequential set of data:

X[0]	X[1]	···	X[N]

Figure 2.2: Standard array format

This structure works rather well for traditional linear programs, as it allows efficient allocation of the program memory. However, parallel access of the array is not possible, as the same chunk of memory would be referenced by multiple scheduled calls. This fails to access at the best, and crashes the system at the very worst. To mitigate this problem, it is possible to reshape the array into a format conducive to parallel access.

The usage of `#pragma HLS ARRAY_PARTITION` in the HLS environment provides the ability to transpose any given array with a variety of configurable parameters. A trivial method is to partition the array completely, resultant in the following data structure:

X[0]
X[1]
···
X[N]

Figure 2.3: Partitioned array (complete)

The structure shown in Figure 2.4 utilizes complete partioning. This means that each element can be accessed in parallel at any time, without read/write conflicts. However, this also increases the number of storage elements necessary to allow this transposition. In situations where a complete partition is not viable or computationally necessary, it is possible to partition the array with a varying factor. For example, partitioning the array X with a factor of 3 achieves the following data structure:

X[0]	X[1]	X[2]
···	···	···
···	···	···
X[N-3]	X[N-1]	X[N]

Figure 2.4: Partitioned array (factor of 3)

This allows array elements to be accessed in multiples of three. To utilize the array in this manner, a factor of 3 should be present in the design when unrolling or pipelining. The advantages of higher-factor partitioning reduces the discrete hardware necessary to store an element of data, at the potential cost of increased design latency.

Despite the advantages of higher-factor partitioning, the simplicity of the FIR filter and its small number of taps is conducive to complete array partitions on data in the system. For example, Listing 2.6 shows how the pipeline shift registers are partitioned.

```
#pragma HLS ARRAY_PARTITION variable = delayLineReal type =
    complete
#pragma HLS ARRAY_PARTITION variable = delayLineImg type =
    complete
```

Listing 2.6: Array partition precompiler directive

2.2.4 Datatypes

An interesting area for optimization in an HLS environment is the usage of arbitrary-width datatypes. In a traditional C/C++ environment, the amount of memory available usually provides no reason to use custom types. Additionally, because a traditional C program runs on a fixed-width data structure, anything other than 8/32/64/128-bit architecture is inefficient and often unnecessary.

The difference between a traditional C program and a HLS product is that HLS products are synthesized into a product for deployment on an FPGA. Because FPGAs have fully reprogramable logic/DSP slices, it is possible to use data widths of any kind in a design.

That being said, there are a few technical constraints or considerations to be made in choosing the size of a given datatype. The DSPs on an FPGA are limited in their input data sizes. In the case of Xilinx's DSP48 on the Zynq-7000 series FPGA, it is possible to multiply a 25 and 18 bit number on a single DSP slice. Therefore, operations that exceed 25x18 bits will leverage multiple DSPs to compute the result. Standard `int` and `float` datatypes require 32 bits to calculate, and therefore would require at least 3 DSP slices to compute.

Another consideration is the input/output range of the design, in addition to signage or required precision. It is possible to compute the number of bits required for any given number using the equation

$$K = \lceil log_2|x| \rceil \tag{2.9}$$

where K is the computed number of bits, and x is the number in question. If x is a decimal number, it's necessary to shift the number's decimal to the end of the decimal for the calculation. For example the number $x = 9000$ requires 14 bits, and the number $x = 10.0025$ requires 17 bits.

We can also use this equation to determine the number of bits necessary to handle data in/out of the FIR filter. Consider a worst-case scenario for the input dataset $\overline{X}$ and filter coefficients w_i of

$$\overline{X} = [50 + j50, 50 + j50, ..., 50 + j50]^T$$

and

$$w_i = [50 + j50, 50 + j50, ..., 50 + j50]^T$$

where the input data size is arbitrary, but the span of w_i is fixed to 25 taps. Applying the FIR filter to a datapoint, we find the output of the filter $\overline{Y}$ to be

$$\overline{Y}_{max} = 2 * 50^2 * 25 = 125,000$$

as a worst-case maximum. Of course, this also works for a worst-case minimum, which generates $\overline{Y}_{min} = -125,000$. These results are applied to equation 2.9, determining the results of K_{in} for the inputs, and K_{out} for the output bitwidth required:

$$K_{in} = \lceil log_2|50|\rceil = 6 \tag{2.10}$$

$$K_{out} = \lceil log_2|125000|\rceil = 17 \tag{2.11}$$

These results are applied to arbitrary precision datatypes in the design, and contribute significantly to the reduction of resource utilization in the design.

Arbitrary precision

The HLS environment offers arbitrary precision datatypes for signed or unsigned integers, and fixed-point decimals. When leveraged properly, the usage of these data types dramatically increases design performance.

To use this functionality, we access functions in the `ap_int.h` and `ap_fixed` libraries. The results obtained for K_{in} and K_{out} translate into the following design datatypes seen in Listing 2.7

```
#define FIR_INT_INPUT_WIDTH 6 // 6 bits wide
#define FIR_INT_OUTPUT_WIDTH 18 // 18 bits wide
typedef ap_int<FIR_INT_INPUT_WIDTH> FIR_INT_INPUT; //!<
    Fixed-width integer datatype for FIR inputs
typedef ap_int<FIR_INT_OUTPUT_WIDTH> FIR_INT_OUTPUT; //!<
    Fixed-width integer datatype for FIR outputs
```

Listing 2.7: System datatypes

Utilization of these new datatypes as `typedefs` is not incredibly complex, but does require consultation of HLS documentation for utilization functions. Conversion between datatypes, or printing results to the console requires specific syntax.

2.2.5 Optimization results

Optimization performed to the algorithm has dramatically reduced the design's latency and utilization. This result was only possible after the careful insertion of precompiler directives, and one's ability to structure the algorithm in a manner conducive to parallel processing. It should be noted that there are still design limitations that exist. Namely, the ability to load data input into the filter itself. Because the top-level architecture requires variable input lengths, there's nothing that can be done to improve element access, as those elements are passed as a pointer of unknown size (i.e. `int *foo` instead of `int foo[4]`).

Below in Figure 2.5 is the utilization report generated from HLS upon synthesis of the computational system.

Modules & Loops	Issue Type	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT
polarFir		-	-	-	-	-	dataflow	0	3	4016	7279
entry_proc		0	0.0	-	0	-	no	0	0	3	38
complexFIR		-	-	-	-	-	no	0	2	1491	1825
COMPUTE		-	-	79	-	-	no	-	-	-	-
computeComplexFIR		29	290.000	-	29	-	no	0	2	670	643
FILTER_TAPS		27	270.000	4	1	25	yes	-	-	-	-
bulkCordicConvert		-	-	-	-	-	no	0	1	953	3112
BULK_CORDIC		-	-	51	-	-	no	-	-	-	-
bulkCordicConvert_Pipeline_ROTATOR		34	340.000	-	34	-	no	0	0	143	526
ROTATOR		32	320.000	2	1	32	yes	-	-	-	-

Figure 2.5: Optimized utilization report

Analysis of this report, along with the un-optimized report shown in Table 2.1, shows the following changes in resource utilization for the FIR filter, with amazing results.

Resource type	**Before**	**After**	**% Change**
DSP	396	2	-99.5%
Flip-flops	28701	670	-97.6%
Lookup tables	27884	643	-97.7%

Table 2.1: FIR filter utilization before and after optimization

The original design was so inefficient that the FIR in combination with the CORDIC was impossible to implement on the Zynq chipset in use. Optimization of the program structure and datatypes has dramatically increased the design's performance.

Another area of optimization for the design is latency. Perhaps not as dramatic a change as the utilization, there was still a substantial increase in the design performance. These results are summarized in Table

Latency cycles	**Before**	**After**	**% Change**
Total	136	27	-80.2%
Iteration latency	113	79	-30.1%

Table 2.2: FIR filter latency before and after optimization

Again, this result is well-recieved for the design. The results suggest that the design has been highly optimized for computational latency and resource utilization. This design would be adventageous to integrate into a more complete processing chain, as it uses a small amount of resources continually and computes a result with minimal latency.

Chapter 3

CORDIC

The next unit in the processing chain, as shown in Figure 1.1, is the CORDIC rotational computer. The CORDIC is implemented in Vitis HLS using equations shown in section 1.1.2. The CORDIC's inputs are specified on a range generated by the worst-case scenario of the FIR filter's behavior, which is ± 125000 given an input to the FIR of ± 50 (refer to section 2.2.4). Additionally the FIR filter outputs data in a cartesian (x, y) form. The CORDIC recieves these vectors of the cartesian form

$$d = \alpha + j\beta \tag{3.1}$$

and converts them into polar form

$$d = (r, \theta) \tag{3.2}$$

where

$$r = \sqrt{a^2 + b^2} \tag{3.3}$$

and

$$\theta = \tan^{-1}\left(\frac{\beta}{\alpha}\right) \tag{3.4}$$

As discussed in section 1.1.2, this calculation is not trivial. However, it is possible to translate the coordinate system shift into the algorithm seen below in Listing 3.1.

```
// Rotate the vector back to 0deg to find the thetaRotated value
ROTATOR:
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
#pragma HLS PIPELINE II = 1
                // Compute a shift iteration for the system:
                FIXED_POINT cosShift = currentCos >> i;
                FIXED_POINT sinShift = currentSin >> i;
                // Determine which direction to rotate the vector,
                    and rotate accordingly:
#ifdef DEBUG_MODE
                printf("Prior to rotation: X = %f, Y = %f, theta = %
                    f\n", (float)currentCos, (float)currentSin, (
                    float)thetaRotated);
#endif
```

```
                if (currentSin > 0) // Quadrant I, rotate clockwise
                {
#ifdef DEBUG_MODE
                        printf("Clockwise rotation\n");
#endif
                        // Perform a clockwise rotation down to the X-axis
                        currentCos = currentCos + sinShift;
                        currentSin = currentSin - cosShift;
                        // Update the rotated theta
                        thetaRotated = thetaRotated + cordicPhase[i];
                }
                else // if (currentSin < 0) // Quadrant IV, rotate counter-clockwise
                {
#ifdef DEBUG_MODE
                        printf("Counter-clockwise rotation\n");
#endif
                        // Counter-clockwise rotation up to the X-axis
                        currentCos = currentCos - sinShift;
                        currentSin = currentSin + cosShift;
                        // Update the rotated theta
                        thetaRotated = thetaRotated - cordicPhase[i];
                }
#ifdef DEBUG_MODE
                printf("End of iteration\n");
#endif
                // End of iterations, so value can be updated
```

Listing 3.1: CORDIC algorithm

Analysis of this algorithm shows that it is an iterative process that over/under rotates the vector at decrementing amounts until the target angle has been reached. The actual rotation of the vectors is accomplished via simple multiplication of a matrix, shown in equations 1.6 and 1.7.

One area not discussed in detail thus far is the range of input the CORDIC is able to take. The accelerator is only capable of operation in quadrants I and IV. Therefore, an initial coordinate shift may be necessary before the raw data is passed to the rotator algorithm itself. The specifics and details of this process are shown in Listing C.2.

3.1 Performance of the CORDIC

The performance of the CORDIC hinges on three main concepts. Precision, latency, and utilization. Latency and utilization carry the same nomenclature as the discussion in the FIR algorithm's performance (see section 2.1). Precision of the CORDIC is a separate metric from the other two, yet has substantial impact on those two parameters. The precision of the CORDIC algorithm is broadly dependent on the number of iterations the algorithm goes through to calculate the desired angle. As noted in section 1.1.2, each subsequent iteration of the algorithm rotates the vector by a lesser amount, and as such we find for a given target angle Θ, CORDIC-calculated angle θ, and number of iterations n,

$$\Theta \approx lim_{n \to \infty} \theta \tag{3.5}$$

is a valid approximation for the convergence of any given angle calculation.

A critical parameter of the convergence of the algorithm is the number of iterations, or rotations, to perform. A method to estimate the convergence of the CORDIC algorithm is to inspect the CORDIC gain coefficents calculated using equation 1.8. The inverse of these coefficients (K_i^{-1}) can be plotted to show convergence of the coefficients, as shown in Figure ??.

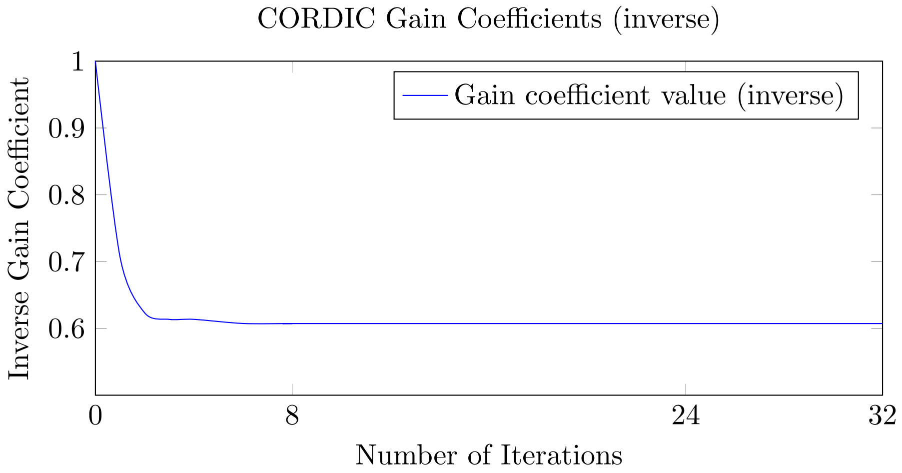


Figure 3.1: Plot of CORDIC gain

We observe a substantial convergence of the algorithm after approximately 8 iterations of the algorithm. The 8th iteration addresses rotations on the magnitude of 2^{-8}, which is an extremely small rotation. Traditional `float` datatypes do not consider levels beyond this amount of percision, although it is important to preserve such precision as long as possible while performing rotations of a given vector.

A similar pattern is observed with the rotational angle of a given increment. We construct a similar plot as shown below:

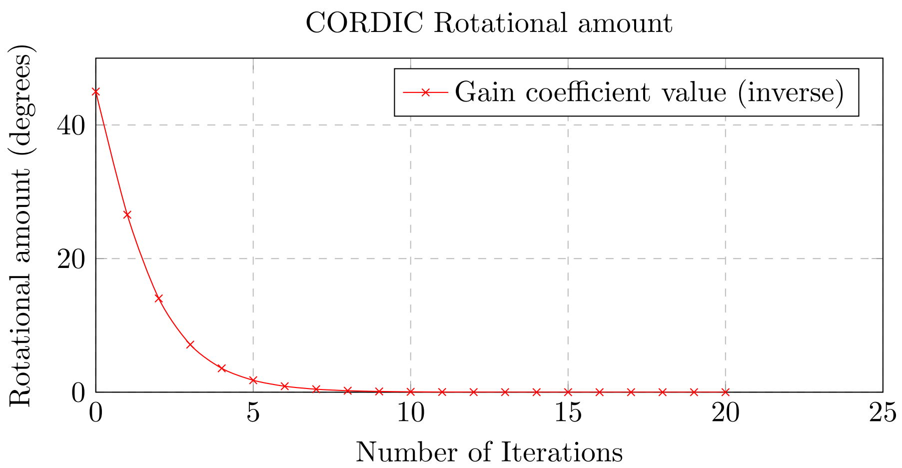


Figure 3.2: Plot of CORDIC rotation per iteration

While this is a useful plot at showing how quickly the angle changes converge to near-zero change, there is a near-direct logarithmic relationship to this result, as presented by the chart below:

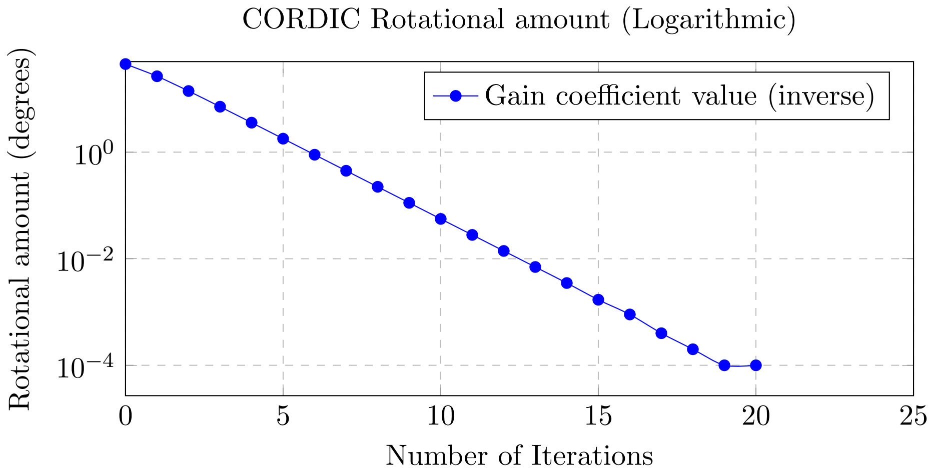


Figure 3.3: Plot of CORDIC rotation per iteration (log)

These results suggest that beyond approximately 8-10 levels of precision, there's not much room to gain in terms of approaching the true value of a given angle calculation.

3.1.1 Determination of precision

To validate the theory of higher-iteration algorithms and their resultant precision, we implement the CORDIC algorithm in Vitis HLS and perform hardware simulations on a datapoint. Of course, the larger the piece of data is, the more deviation will be apparent in a final calculated value. For the test, we use the input cartesian vector

$$\overline{X} = (325, 325) \tag{3.6}$$

which, using traditional trigometry, converts to the polar form of

$$\overline{X^*} = (r, \theta) = (459.619407771256, \pi/4) = (459.619407771256, 0.785398163397448) \tag{3.7}$$

where $\overline{X^*}$ is our "true value" we wish to calculate (or closely approach) using the CORDIC algorithm. Running the CORDIC using the datatype `FIXED_POINT` (refer to Listing D.1), we determine the following results for magnitude and phase:

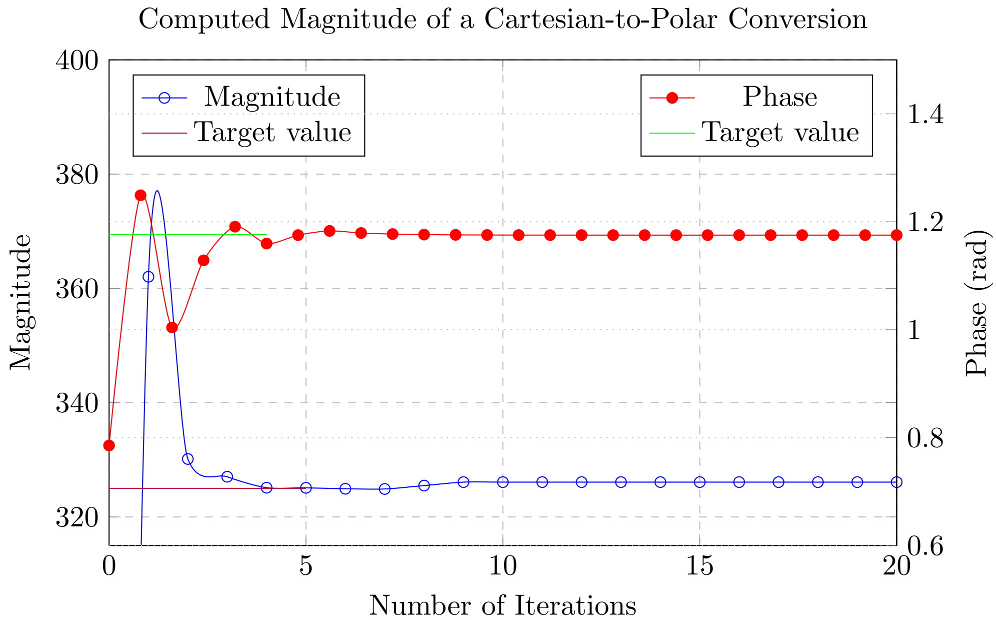


Figure 3.4: Converted magnitude and phase of a cartesian vector

We see that beyond 10 iterations, there is no significant improvement in the algorithm's computed result. Therefore, it is wise to avoid iterations beyond this amount. For the purposes of the accelerator, the number of iterations was fixed to 8.

3.2 Optimization

3.2.1 Pipelining

The algorithmic structure of the CORDIC is conducive to optimization. Unlike a FIR filter's convolutional structure, the CORDIC is a simple fixed-length iterative process that is immediately structured for optimization via parallelism. Given that observation, we optimize the code structure using the precompiler directive `#pragma HLS PIPELINE` within the main rotation algorithm. This is shown below in Listing 3.2.

```
ROTATOR:
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
#pragma HLS PIPELINE II = 1
                // Compute a shift iteration for the system:
                FIXED_POINT cosShift = currentCos >> i;
                FIXED_POINT sinShift = currentSin >> i;
```

Listing 3.2: Pipeline optimization

The pipelined algorithm allows parallel usage of hardware. This reduces latency significantly in the design, and increases the overall throughput of the processing chain. This is the only optimization performed to the CORDIC, which despite its simplicity, achieves a design latency as noted in Figure 2.5.

3.2.2 Unrolling

Further optimization can be made to the design. While an individual iteration of the CORDIC computes a vector pair, the processing chain requires that the coordinate system conversion is performed across the entire dataset. Much like concepts in the FIR filter, the usage of the precompiler directive `#pragma HLS UNROLL` increases parallelism within the design. Because the CORDIC is an iterative algorithm, it is not possible to include loop unrolling in the actual calculation itself. However, in the higher-level loop that calls the algorithm, it's entirely possible. This is shown in Listing 3.3.

```
   BULK_CORDIC: for(int i = 0; i < convertSize; i++)
   {
#pragma HLS UNROLL
      //Read from the stream and convert to fixed point:
      cos.read(cosInt);
      sin.read(sinInt);
      cosFixed = (FIXED_POINT)cosInt;
      sinFixed = (FIXED_POINT)sinInt;

      //Send to CORDIC:
      cordic(cosFixed, sinFixed, &outMag, &outTheta);
      theta[i] = outTheta.to_float();
      mag[i] = outMag.to_float();
#ifdef DEBUG_MODE
      printf("CORDIC.cpp: Magnitude: %f, Phase: %f\n", outMag.
         to_float(), outTheta.to_float());
#endif
   }
```

Listing 3.3: Loop optimization

The usage of this particular precompiler directive increases the number of calls to the CORDIC converter, and subsequently increases design throughput. The results from this optimization is shown in Figure 2.5.

Chapter 4

Complete Processing Chain

The complete processing chain consists of the FIR filter and CORDIC cartesian-to-polar converter shown in Figure 1.1. Systems which may employ such a chain commonly include real-time audio processing, wireless communication, medical imaging, and radar systems - to name a few.

The internal architecture of the complete processing chain is rather simple, it serves as a top-level I/O handler for the chain, and handles data between processing elements. Data is transferred between the FIR filter and CORDIC via `HLS::stream` objects. These objects act as a first in first out (FIFO) queue between functions to ensure efficient single-element data transfer without substantial read/write delays. The syntax to declare datastreams between the two chain elements is shown in Listing 4.1

```
    hls::stream<FIR_INT_OUTPUT> realStream; //!< Stream between FIR output and CORDIC input (real)
    hls::stream<FIR_INT_OUTPUT> imgStream; //!< Stream between FIR output and CORDIC input (imaginary)

#pragma HLS STREAM variable=realStream
#pragma HLS STREAM variable=imgStream

#pragma HLS DATAFLOW
    //Declare the CORDIC and FIR, and connect them with the stream objects:
    complexFIR(inputReal, inputImg, realStream, imgStream, inputLength);
    bulkCordicConvert(realStream, imgStream, outputMag, outputPhase, inputLength);
```

Listing 4.1: Stream syntactical use

Additonally, the complete processing chain uses the appropriate precompiler directives to set the AXI4-compliant interface for the device. Particularly, the usage of a memory-access interface is desired (set/get various data with driver functions). This enables export as a Xilinx IP, and its subsequent use in a Vivado product. An example of the interface is shown in Listing 4.2

```
#pragma HLS INTERFACE mode = m_axi port = inputReal offset = slave
    bundle=inReal
#pragma HLS INTERFACE mode = m_axi port = inputImg offset = slave
    bundle=inImg

// Polar outputs:
#pragma HLS INTERFACE mode = m_axi port = outputMag offset = slave
    bundle=outMag
#pragma HLS INTERFACE mode = m_axi port = outputPhase offset = slave
     bundle=outPhase

#pragma HLS INTERFACE mode = s_axilite port = inputLength
#pragma HLS INTERFACE mode = s_axilite port = return
```

Listing 4.2: AXI4 interface declaration

The resultant top-level function completes the processing chain developed in the HLS environment. No optimization is performed in this top-level function (aside from the `HLS::streams`), as it is unnecessary given the granular development of the component processsing stages.

4.1 Hardware development

This HLS product was exported as a fully AXI4-compliant interface IP for usage in a Vivado design. This IP allows transport between different hardware targets, and also protects any sensitive processing trade-secrets from being made public.

This HLS IP was brought into a Vivado block diagram. Additionally, the Zynq processing system and the necessary AXI4 protocol systems was connected using the connection automation function in Vivado. The complete block diagram for the system is seen below in Figure 4.1.

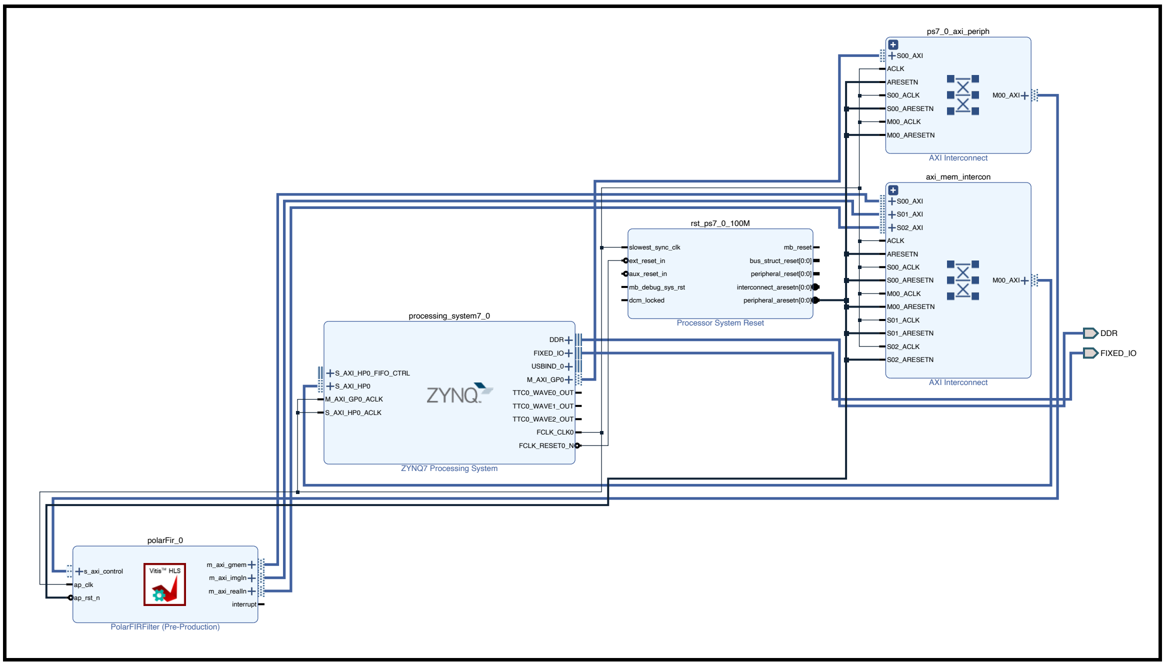


Figure 4.1: Hardware block diagram

The block diagram provides further abstraction from the source code written in C++ (and the Verilog products), and lets the deployment process happen much more quickly. The block diagram was then wrapped, and a RTL diagram was generated. The design was then synthesized implemented, and packaged as a `.xsa` file for usage in Vitis IDE as a development platform.

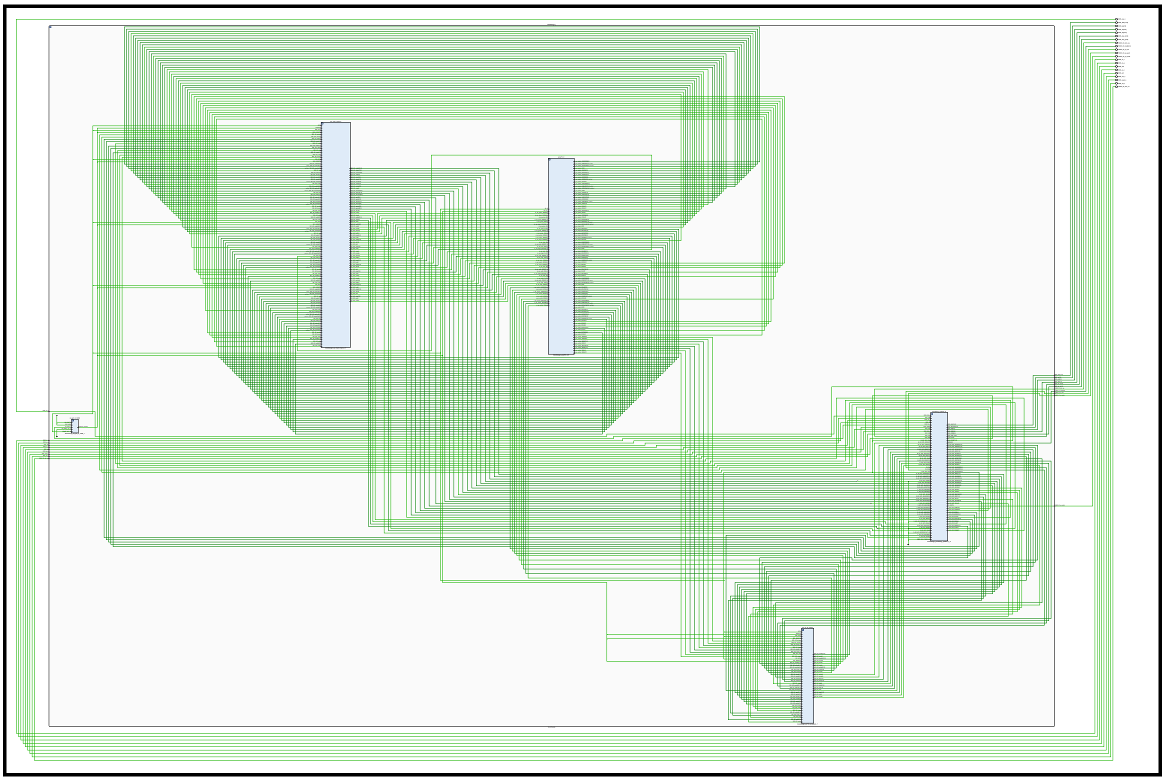


Figure 4.2: RTL schematic

4.2 Vitis IDE interaction development

The hardware design created in section 4.1 was brought into Vitis IDE as the last step in the development stack to generate a functional product. In the IDE, we develop an interaction function to initiate, start, and obtain results from any given processing chain.

In addition to the hardware design, Vitis HLS provides a set of driver functions for utilization of a HLS product in a complete system. These functions are necessary to interact with the HLS IP. The generated drivers include typical hardware interaction functions, such as set/get of data, initialization, start/stop, and status returns. The function declarations are included in Listing G.1, although the specifics don't essentially matter.

What is crucially important is that the order-of-operations is followed to initialize the hardware to its proper state before running the accelerator. This is accomplished with a set of functions provided in the driver, but is abstracted slightly for ease-of-development. The source code for this abstraction layer is provided in Listing G.3.

With a functional driver, it is possible to interact with the accelerator and validate its functionality. This is accomplished via a top-level application, who's source code can be found in Listing E.1.

The high-level functionality of the source code is to generate an input and filter set, run the accelerator, and print out the results. Data fed into the filter is a u32 datatype, and is memory-mapped to pointers shown below in Listing 4.3

```
  u32 *INPUT_REAL_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR;
  u32 *INPUT_IMG_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x20000;
  u32 *HW_MAG_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR +  0x20000 + 0
     x20000;
  u32 *HW_PHASE_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x20000+ 0
     x20000 + 0x20000;

   memset(INPUT_IMG_BUFFER, 0, (INPUT_LENGTH << 2));
   memset(INPUT_REAL_BUFFER, 0, (INPUT_LENGTH << 2));
   memset(HW_MAG_BUFFER, 0, (DATA_LENGTH << 2));
   memset(HW_PHASE_BUFFER, 0, (DATA_LENGTH << 2));
```

Listing 4.3: Primary datatypes

Then, with the datatypes properly generated and allocated, the accelerator is set with that data. Additionally, the output products of the system are mapped, as shown in Listing 4.4.

```
   XPolarfir_Set_inputReal(&polarFIRinst, (u64)INPUT_REAL_BUFFER);
   XPolarfir_Set_inputImg(&polarFIRinst, (u64)INPUT_IMG_BUFFER);
   XPolarfir_Set_inputLength(&polarFIRinst, (u32)DATA_LENGTH);
   XPolarfir_Set_outputMag(&polarFIRinst, (u64)HW_MAG_BUFFER);
   XPolarfir_Set_outputPhase(&polarFIRinst, (u64)HW_PHASE_BUFFER);
```

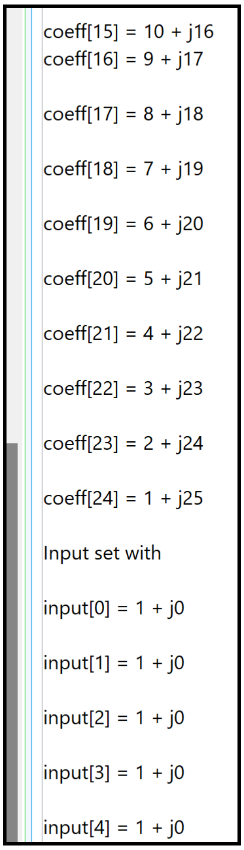
Listing 4.4: Main application source code

Once this process is complete, we run the accelerator using the `runHardware();` function provided by the IP-control abstraction layer. Upon completion of the dataset processing, the accelerator notifies the Zynq processor via a hardware interrupt that a valid result is ready to be read and further processed.

4.3 Validation

Validation of the hardware's functionality in the IDE is rather simple. Because the components were tested in HLS, the final validation is to check if the IDE product can properly interact with the accelerator. This is accomplished via running the hardware (as previously mentioned), and then waiting for the hardware to return it has finished computation via an interrupt.

With the final results, we print the output results. Using a test dataset of the input vectors, filter coefficents, we initialize the dataset and run the test. We obtain the following output from the IDE after running the accelerator:



coeff[15] = 10 + j16
coeff[16] = 9 + j17
coeff[17] = 8 + j18
coeff[18] = 7 + j19
coeff[19] = 6 + j20
coeff[20] = 5 + j21
coeff[21] = 4 + j22
coeff[22] = 3 + j23
coeff[23] = 2 + j24
coeff[24] = 1 + j25
Input set with
input[0] = 1 + j0
input[1] = 1 + j0
input[2] = 1 + j0
input[3] = 1 + j0
input[4] = 1 + j0

Figure 4.3: Input dataset

```
hwMag[0] = 25.019531, ph = 0.039307
hwMag[1] = 49.087646, ph = 0.059814
hwMag[2] = 72.241211, ph = 0.083740
hwMag[3] = 94.519043, ph = 0.105713
hwMag[4] = 115.960205, ph = 0.129150
hwMag[5] = 136.606445, ph = 0.153076
hwMag[6] = 156.506104, ph = 0.179932
hwMag[7] = 175.706055, ph = 0.206787
hwMag[8] = 194.259766, ph = 0.234131
hwMag[9] = 212.223633, ph = 0.260986
hwMag[10] = 229.658691, ph = 0.290771
hwMag[11] = 246.626953, ph = 0.320068
hwMag[12] = 263.198486, ph = 0.353760
hwMag[13] = 279.440430, ph = 0.385498
hwMag[14] = 295.429443, ph = 0.417725
hwMag[15] = 311.243408, ph = 0.452881
hwMag[16] = 326.961426, ph = 0.486084
hwMag[17] = 342.668457, ph = 0.522705
hwMag[18] = 358.447510, ph = 0.557373
hwMag[19] = 374.386719, ph = 0.594482
hwMag[20] = 390.575439, ph = 0.633545
hwMag[21] = 407.098389, ph = 0.669678
hwMag[22] = 424.047852, ph = 0.709229
hwMag[23] = 441.506836, ph = 0.746338
hwMag[24] = 459.562744, ph = 0.785400
hwMag[25] = 459.562744, ph = 0.785400
hwMag[26] = 459.562744, ph = 0.785400
hwMag[27] = 459.562744, ph = 0.785400
hwMag[28] = 459.562744, ph = 0.785400
hwMag[29] = 459.562744, ph = 0.785400
Cleaning system
```

Figure 4.4: Output data

Manual calculation of these results verifies that the FIR filter correctly performs a filter pass using the equations developed in section 1.1.1, and the CORDIC computes the output value of the system using the trigonometry shown in section 1.1.2. Additionally, the accuracy of the CORDIC is quite good, which validates that the accelerator's iteration depth for the CORDIC is satisfactory.

4.3.1 Concluding remarks

The extensive analysis in this report of a FIR-CORDIC processing chain provides an in-depth look at the development of FPGA hardware accelerators in the Vitis HLS development environment. The simplicity of the generated source code demonstrates the ease of creating highly optimized algorithms, without an in-depth understanding of raw Verilog. Additionally, more nuanced design aspects such as pipelining or serial-to-parallel conversion is highly abstracted from the developer. Also, it appears that the development cycle for an HLS product is shorter.

Although using Vitis HLS makes writing high-performance accelerators "easier," it by no means does it automatically. Development must still adhere to common digital design principles such as memory dependencies, pipeline architecture, and dataset manipulation. The abstraction provided by Vitis HLS is no substitute for a proper understanding of digital design. A thorough background in digital discrete signal processing is also necessary, especially when working in both a time and/or frequency domain processing chain.

Despite the depth of analysis provided in this report, the processing chain developed is rather basic. The interaction with the processing chain is also highly limited, and is by no means a product capable of actual signal processing. Future work on driver interaction and perhaps the HLS product itself should look to further develop the design into an actual realtime signal processing system. This is accomplished via using the accelerator in a "stream" configuration, rather than a "load, run, read" architecture. While "load, run, read" is simple to understand and use, it is not scalable for a realtime processing system.

Given this expansion in functionality, it is then possible to use the accelerator in real applications such as audio or 1-D image processing. The speed of the system also enables processing of downconverted RF samples at baseband. The output of the accelerator can then be used as a method to detect phase or amplitude modulation for, for example, a very poor digital radio reciever. We say poor, because nothing in the design accounts for timing errors/oscillator offset, nor does it allow the means to synchronize phase-locked-loops between the transmitter and reciever. Nevertheless, a reliable FIR filter is at the center of this system, and its importance should be emphasized.

Appendices

Appendix A

Top-level processing chain source code

```
/**
 * @file polarFIR.h
 * @author Alex Stepko (axstepko.com)
 * @brief Header file for cartesian-to-polar FIR filter driver architecture.
 * @version 0.1
 * @date 2023-04-30
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
```

```
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY
   USE THIS SOFTWARE
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE
   FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY
   STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND
    PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING
   BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE
    WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY
   VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR
   VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef POLAR_FIR_H
#define POLAR_FIR_H
#include <stdio.h>
#include <math.h>
#include "hls_stream.h"

#include "CORDIC.h"
#include "complexFIR.h"
#include "datatypes.h"

//#define POLAR_FIR_DEBUG_MODE

void polarFir(int *inputReal, int *inputImg, float *outputMag, float
   *outputPhase, int inputLength);

#endif // POLAR_FIR_H
```

Listing A.1: Top-level processor code (header)

```
/**
 * @file polarFIR.cpp
 * @author Alex Stepko (axstepko.com)
 * @brief Cartesian-to-polar FIR filter driver and interface
   architecture
 * @version 0.1
 * @date 2023-04-30
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All
   rights reserved.
 *
 */
```

```c

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef POLAR_FIR_INCLUDES
#define POLAR_FIR_INCLUDES
#include <stdio.h>
#include <math.h>
#include <ap_fixed.h>

#include "polarFIR.h"

#endif // POLAR_FIR_INCLUDES

```

```
/**
 * @brief Computes a 1-d FIR complex FIR filter and displays the result in polar coordinates.
 *
 * @param inputReal Real inputs, first 25 elements are the filter
 * @param inputImg Imaginary inputs, first 25 elements are the filter
 * @param outputMag Output magnitude
 * @param outputPhase Output phase
 * @param inputLength Length of the input dataset to compute
 */
void polarFir(int *inputReal, int *inputImg, float *outputMag, float *outputPhase, int inputLength)
{

// Define the system's AXI interface:
// Data inputs:
#pragma HLS INTERFACE mode = m_axi port = inputReal offset = slave bundle=inReal
#pragma HLS INTERFACE mode = m_axi port = inputImg offset = slave bundle=inImg

// Polar outputs:
#pragma HLS INTERFACE mode = m_axi port = outputMag offset = slave bundle=outMag
#pragma HLS INTERFACE mode = m_axi port = outputPhase offset = slave bundle=outPhase

#pragma HLS INTERFACE mode = s_axilite port = inputLength
#pragma HLS INTERFACE mode = s_axilite port = return


#ifdef POLAR_FIR_DEBUG_MODE
    printf("INPUTS:\n");
    for(int j = 0; j < inputLength; j++)
    {
      printf("POLARFIR: inputReal[%d] = %d, inputImg[%d] = %d\n", j, inputReal[FILTER_SIZE + j], j, inputImg[FILTER_SIZE + j]);
    }
#endif


    //Declare stream objects:
    hls::stream<FIR_INT_OUTPUT> realStream; //!< Stream between FIR output and CORDIC input (real)
    hls::stream<FIR_INT_OUTPUT> imgStream; //!< Stream between FIR output and CORDIC input (imaginary)

#pragma HLS STREAM variable=realStream
#pragma HLS STREAM variable=imgStream

```

```
#pragma HLS DATAFLOW
    //Declare the CORDIC and FIR, and connect them with the stream
       objects:
    complexFIR(inputReal, inputImg, realStream, imgStream,
       inputLength);
    bulkCordicConvert(realStream, imgStream, outputMag, outputPhase,
        inputLength);

}
```

Listing A.2: Top-level processor code

Appendix B

Cartesian FIR Filter code

```
/**
 * @file complexFIR.h
 * @author Alex Stepko (axstepko.com)
 * @brief Header file for a 1-d rectangular coordinate FIR filter
 * @version 1.0
 * @date 2023-04-24
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
```

```
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef COMPLEX_FIR_H
#define COMPLEX_FIR_H
#include <stdio.h>
#include <math.h>
#include "hls_stream.h"

#include "datatypes.h"

//#define FIR_DEBUG_MODE //!< Enables rich debugging support within these functions

#define DEBUG_SIZE 25  //!< Constant-width known-values for debug.
#define FILTER_SIZE 25 //!< Constant-width value for HLS optimization

//const int FILTER_SIZE = 25;

static int debugInputReal[DEBUG_SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
static int debugInputImg[DEBUG_SIZE] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

static int debugCoeffReal[DEBUG_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25};
static int debugCoeffImg[DEBUG_SIZE] = {25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

void complexFIR(int *inputReal, int *inputImg, hls::stream<FIR_INT_OUTPUT> &outputReal, hls::stream<FIR_INT_OUTPUT> &outputImg, int filterLength);
void computeComplexFIR(FIR_INT_INPUT inputReal, FIR_INT_INPUT inputImg, FIR_INT_INPUT filterReal[FILTER_SIZE], FIR_INT_INPUT filterImg[FILTER_SIZE], FIR_INT_OUTPUT *outputReal, FIR_INT_OUTPUT *outputImg);
```

```c
#endif // COMPLEX_FIR_H
```

Listing B.1: Cartesian FIR filter source code (header)

```c
/**
 * @file complexFIR.c
 * @author Alex Stepko (axstepko.com)
 * @brief Hardware-accelerated 1-d rectangular FIR filter
 * @remark Outputs values in rectangular float form
 * @version 1.0
 * @date 2023-04-16
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
```

```
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY
    VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR
    VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef INCLUDES
#include <stdio.h>
#include <stdlib.h>

#endif // INCLUDES

#ifndef DEPENDENCIES
#define DEPENDENCIES
#include "complexFIR.h"

#endif
/**
 * @brief Complete processor for a 1-d FIR filter
 * @remark The real and imaginary parts of the input, and kernel,
    must have the same length.
 *
 * @param inputReal Real part of data input
 * @param inputImg  Imaginary part of data input
 * @param kernelReal Real part of filter coefficents
 * @param kernelImg Pointer to imaginary part of filter coefficents
 * @param outputReal Real part of filter output
 * @param outputImg Imaginary part of filter output
 */
void complexFIR(int *inputReal, int *inputImg, hls::stream<
   FIR_INT_OUTPUT> &outputReal, hls::stream<FIR_INT_OUTPUT> &
   outputImg, int filterLength)
{

//Partition the arrays accordingly:
#pragma HLS ARRAY_PARTITION variable = outputReal type = complete
#pragma HLS ARRAY_PARTITION variable = outputImg type = complete

#ifdef FIR_DEBUG_MODE
    printf("Start of hardware FIR...\n");
#endif

   FIR_INT_INPUT kernelImg[FILTER_SIZE]; //!< Imaginary filter
      coefficient storage location
    FIR_INT_INPUT kernelReal[FILTER_SIZE]; //!< Real filter
       coefficient storage location

#pragma HLS ARRAY_PARTITION variable=kernelReal type=complete
#pragma HLS ARRAY_PARTITION variable=kernelImg type=complete

    // Initialize kernel with filter coefficients, using data from
```

```
        the first 0 -> FILTER_SIZE-1 spaces
#ifdef POLAR_FIR_DEBUG_MODE
    printf("FILTER:\n");
#endif
    FILTER_INIT:for (int i = 0; i < FILTER_SIZE; i++)
    {
#pragma HLS UNROLL
        kernelReal[i] = FIR_INT_INPUT(inputReal[i]);
        kernelImg[i] = FIR_INT_INPUT(inputImg[i]);
    }

#ifdef FIR_DEBUG_MODE
    printf("complexFIR: Loaded coefficients:\n");
    for (int a = 0; a < FILTER_SIZE; a++)
    {
        printf("complexFIR: kernel [%d] = %d + j%d\n", a, kernelReal[a].to_int(), kernelImg[a].to_int());
    }
    printf("complexFIR: Loaded inputs:\n");
    for(int b = 0; b < filterLength; b++)
    {
      printf("complexFIR: input[%d] = %d + j%d\n", b, inputReal[FILTER_SIZE + b], inputImg[FILTER_SIZE + b]);
    }
#endif

   FIR_INT_OUTPUT tempR = 0;
   FIR_INT_OUTPUT tempI = 0;

COMPUTE:
    for (int k = 0; k < filterLength; k++)
    {
#pragma HLS PIPELINE
      FIR_INT_INPUT inTempReal = FIR_INT_INPUT(inputReal[FILTER_SIZE + k]);
      FIR_INT_INPUT inTempImg = FIR_INT_INPUT(inputImg[FILTER_SIZE + k]);

        // Perform a single pass of an input with the coefficients:
        computeComplexFIR(inTempReal, inTempImg, kernelReal, kernelImg, &tempR, &tempI);
#ifdef FIR_DEBUG_MODE
      printf("complexFIR: result = %d + j%d\n", tempR.to_int(), tempI.to_int());
#endif
        outputReal.write(tempR);
        outputImg.write(tempI);
    }
}

/**
```

```
 * @brief Performs a FIR on a single element of a complex datapoint
 *
 * @param inputReal Real part of input sample
 * @param inputImg Imaginary part of input sample
 * @param filterReal Real part of filter coefficients
 * @param filterImg Imaginary part of filter coefficients
 * @param filterLength Length of the filter
 * @param delayLineReal Real part of pipeline delay component
 * @param delayLineImg Imaginary part of pipeline delay component
 * @param outputReal Real part of discrete output
 * @param outputImg Imaginary part of discrete output
 */
void computeComplexFIR(FIR_INT_INPUT inputReal, FIR_INT_INPUT
    inputImg, FIR_INT_INPUT filterReal[FILTER_SIZE], FIR_INT_INPUT
    filterImg[FILTER_SIZE], FIR_INT_OUTPUT *outputReal,
    FIR_INT_OUTPUT *outputImg)
{
//#pragma HLS PIPELINE
   FIR_INT_OUTPUT resultReal = 0;
   FIR_INT_OUTPUT resultImg = 0; //!< Temporary result hold for the
      filter pass

    static FIR_INT_INPUT delayLineReal[FILTER_SIZE] = {}; //!< Input
        pipeline delay buffer (real)
    static FIR_INT_INPUT delayLineImg[FILTER_SIZE] = {};  //!< Input
        pipeline delay buffer (imaginary

#pragma HLS ARRAY_PARTITION variable = delayLineReal type = complete
#pragma HLS ARRAY_PARTITION variable = delayLineImg type = complete

#pragma HLS ARRAY_PARTITION variable = filterReal type = complete
#pragma HLS ARRAY_PARTITION variable = filterImg type = complete

#ifdef FIR_DEBUG_MODE
    printf("computeComplexFIR: Recieved input %d + j%d\n", inputReal
        .to_int(), inputImg.to_int());
#endif

PIPELINE_DELAY:
    for (int i = FILTER_SIZE - 1; i >= 1; i--)
    {
#pragma HLS UNROLL
        // Iterate backwards through the array to shift to the right
           .
        delayLineReal[i] = delayLineReal[i - 1];
        delayLineImg[i] = delayLineImg[i - 1];
#ifdef FIR_DEBUG_MODE
        printf("computeComplexFIR: delay[%d] = %d + %di\n", i,
            delayLineReal[i].to_int(), delayLineImg[i].to_int());
#endif
    }
```

```
    // Add the new input sample to the beginning of the delay line
       arrays
    delayLineReal[0] = inputReal;
    delayLineImg[0] = inputImg;

    // Compute a pass of the filter
FILTER_TAPS:
    for (int j = 0; j < FILTER_SIZE; j++)
    {
#pragma HLS UNROLL
        resultReal += (delayLineReal[j] * filterReal[j]) - (
           delayLineImg[j] * filterImg[j]);
        resultImg += (delayLineReal[j] * filterImg[j]) + (filterReal
           [j] * delayLineImg[j]);
    }

    //Send outputs of the filter pass
    *outputReal = resultReal;
    *outputImg = resultImg;
}
```

Listing B.2: Cartesian FIR filter source code

Appendix C

CORDIC source code

```
/**
 * @file CORDIC.h
 * @author Alex Stepko (axstepko.com)
 * @brief Header file for a CORDIC calculation system
 * @version 0.1
 * @date 2023-04-30
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
```

```
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE
    FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY
    STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND
     PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING
    BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE
     WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY
    VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR
    VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef CORDIC_H
#define CORDIC_H
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hls_stream.h"

#include "datatypes.h"

//#define DEBUG_MODE

#define MAX_DEPTH 64 //!< Maximum allowed depth/precision of the
   CORDIC

static FIXED_POINT cordicPhase[MAX_DEPTH] = {0.78539816339744828000,
    0.46364760900080609000, 0.24497866312686414000,
   0.12435499454676144000, 0.06241880999595735000,
   0.03123983343026827700, 0.01562372862047683100,
   0.00781234106010111110, 0.00390623013196697180,
   0.00195312251647881880, 0.00097656218955931946,
   0.00048828121119489829, 0.00024414062014936177,
   0.00012207031189367021, 0.00006103515617420877,
   0.00003051757811552610, 0.00001525878906131576,
   0.00000762939453110197, 0.00000381469726560650,
   0.00000190734863281019, 0.00000095367431640596,
   0.00000047683715820309, 0.00000023841857910156,
   0.00000011920928955078, 0.00000005960464477539,
   0.00000002980232238770, 0.00000001490116119385,
   0.00000000745058059692, 0.00000000372529029846,
   0.00000000186264514923, 0.00000000093132257462,
   0.00000000046566128731, 0.00000000023283064365,
   0.00000000011641532183, 0.00000000005820766091,
   0.00000000002910383046, 0.00000000001455191523,
   0.00000000000727595761, 0.00000000000363797881,
```

```
    0.000000000000181898940, 0.000000000000090949470,
    0.000000000000045474735, 0.000000000000022737368,
    0.000000000000011368684, 0.000000000000005684342,
    0.000000000000002842171, 0.000000000000001421085,
    0.000000000000000710543, 0.000000000000000355271,
    0.000000000000000177636, 0.000000000000000088818,
    0.000000000000000044409, 0.000000000000000022204,
    0.000000000000000011102, 0.000000000000000005551,
    0.000000000000000002776, 0.000000000000000001388,
    0.000000000000000000694, 0.000000000000000000347,
    0.000000000000000000173, 0.000000000000000000087,
    0.000000000000000000043, 0.000000000000000000022,
    0.000000000000000000011};
const FIXED_POINT cordicGain[MAX_DEPTH] = {0.7071067811865476,
    0.6324555320336759, 0.6135719910778963, 0.6088339125177524,
    0.6076482562561682, 0.6073517701412959, 0.6072776440935250,
    0.6072591122988928, 0.6072544793325624, 0.6072533210898752,
    0.6072530315291342, 0.6072529591389448, 0.6072529410413971,
    0.6072529365170102, 0.6072529353859135, 0.6072529351031395,
    0.6072529350324458, 0.6072529350147724, 0.6072529350103540,
    0.6072529350092495, 0.6072529350089730, 0.6072529350089042,
    0.6072529350088871, 0.6072529350088828, 0.6072529350088814,
    0.6072529350088810, 0.6072529350088809, 0.6072529350088809,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808,
    0.6072529350088808, 0.6072529350088808, 0.6072529350088808}; //!<
     CORDIC gain coefficents for quick-access

#define NUM_ITERATIONS 8 //!< Number of iterations for the
    rotational computer


void bulkCordicConvert(hls::stream<FIR_INT_OUTPUT> &cos, hls::stream
    <FIR_INT_OUTPUT> &sin, float * mag, float *theta, int convertSize
    );
void cordic(FIXED_POINT &cos, FIXED_POINT &sin, FIXED_POINT *mag,
    FIXED_POINT *theta);
#endif // CORDIC_H
```

Listing C.1: CORDIC source code (header)

```
/**
```

```c
 * @file CORDIC.c
 * @author Alex Stepko (axstepko.com)
 * @brief Rectangular to polar CORDIC converter
 * @version 0.1
 * @date 2023-04-17
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR VIEWERS OF THIS SOFTWARE.
 *
 */
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "CORDIC.h"

void bulkCordicConvert(hls::stream<FIR_INT_OUTPUT> &cos, hls::stream
    <FIR_INT_OUTPUT> &sin, float * mag, float *theta, int convertSize
    )
{
    FIXED_POINT cosFixed = 0.0;
    FIR_INT_OUTPUT cosInt, sinInt = 0;
    FIXED_POINT sinFixed = 0.0;
    FIXED_POINT outMag = 0.0;
    FIXED_POINT outTheta = 0.0;
    // Initialize a bunch of CORDIC's to convert the incoming data:
    BULK_CORDIC: for(int i = 0; i < convertSize; i++)
    {
#pragma HLS UNROLL
        //Read from the stream and convert to fixed point:
        cos.read(cosInt);
        sin.read(sinInt);
        cosFixed = (FIXED_POINT)cosInt;
        sinFixed = (FIXED_POINT)sinInt;

        //Send to CORDIC:
        cordic(cosFixed, sinFixed, &outMag, &outTheta);
        theta[i] = outTheta.to_float();
        mag[i] = outMag.to_float();
#ifdef DEBUG_MODE
        printf("CORDIC.cpp: Magnitude: %f, Phase: %f\n", outMag.
            to_float(), outTheta.to_float());
#endif
    }
}

/**
 * @brief Rectangular to polar coordinate converter
 *
 * @param cos Rectangular real component input (cos)
 * @param sin Rectangular imaginary component input (sin)
 * @param mag Output vector magnitude
 * @param theta Output vector angle (radians)
 */
void cordic(FIXED_POINT &cos, FIXED_POINT &sin, FIXED_POINT *mag,
    FIXED_POINT *theta)
{
        FIXED_POINT currentCos = cos;
        FIXED_POINT currentSin = sin;
        FIXED_POINT thetaRotated = 0.0;
        FIXED_POINT ninetyDeg = (FIXED_POINT)M_PI_2;
```

```
        FIXED_POINT circleRads = (FIXED_POINT)M_2_PI;
        FIXED_POINT CORDIC_SCALE_FACTOR = 0.60735;

        FIXED_POINT swapTemp = 0.0; // Temporary value used for swapping coordinates to perform initial vector rotation

#ifdef DEBUG_MODE
      printf("Received input %d + j%d\n", cos.to_int(), sin.to_int());
#endif

        // Rotate the vector to within quadrant I or IV, based upon the sin (y) value of the system

        if ((currentSin > 0) & (currentCos < 0)) // Y value is in quadrant II
        {
// Rotate the vector by -90deg:
// This is accomplished by swapping X, Y, and negating the swapped Y.
#ifdef DEBUG_MODE
                printf("QUAD II SWAP\n");
                printf("BEFORE SWAP: X = %f, Y = %f\n", (float)currentCos, (float)currentSin);
#endif
                swapTemp = -1 * currentCos; // Set temp to opposite of X coordinate
                currentCos = currentSin;    // Set Y to X
                currentSin = swapTemp;      // Set X to negated Y
                thetaRotated = ninetyDeg;

#ifdef DEBUG_MODE
                printf("AFTER SWAP: X = %f, Y = %f\n", (float)currentCos, (float)currentSin);
#endif
        }
        else if ((currentSin < 0) & (currentCos < 0)) // Y value is in quadrant III
        {
// Rotate the vector by +90deg:
// This is accomplished by swapping the X and Y, and negating the swapped X.
#ifdef DEBUG_MODE
                printf("QUAD IV SWAP\n");
                printf("BEFORE SWAP: X = %f, Y = %f\n", (float)currentCos, (float)currentSin);
#endif
                swapTemp = -1 * currentSin; // Set temp to opposite of X coordinate
                currentSin = currentCos;    // Set Y to X
                currentCos = swapTemp;      // Set X to negated Y
```

```
                thetaRotated = -1 * ninetyDeg;
#ifdef DEBUG_MODE
                printf("AFTER SWAP: X = %f, Y = %f\n", (float)currentCos, (float)currentSin);
#endif
        }
#ifdef DEBUG_MODE
        else
        {
                printf("No coordinate swap needed.\n");
                thetaRotated = 0.0;
        }
#endif
// ^At this point, the vector is in quadrant I or IV.

// Rotate the vector back to 0deg to find the thetaRotated value
ROTATOR:
        for (int i = 0; i < NUM_ITERATIONS; i++)
        {
#pragma HLS PIPELINE II = 1
                // Compute a shift iteration for the system:
                FIXED_POINT cosShift = currentCos >> i;
                FIXED_POINT sinShift = currentSin >> i;
                // Determine which direction to rotate the vector, and rotate accordingly:
#ifdef DEBUG_MODE
                printf("Prior to rotation: X = %f, Y = %f, theta = %f\n", (float)currentCos, (float)currentSin, (float)thetaRotated);
#endif
                if (currentSin > 0) // Quadrant I, rotate clockwise
                {
#ifdef DEBUG_MODE
                        printf("Clockwise rotation\n");
#endif
                        // Perform a clockwise rotation down to the X-axis
                        currentCos = currentCos + sinShift;
                        currentSin = currentSin - cosShift;
                        // Update the rotated theta
                        thetaRotated = thetaRotated + cordicPhase[i];
                }
                else // if (currentSin < 0) // Quadrant IV, rotate counter-clockwise
                {
#ifdef DEBUG_MODE
                        printf("Counter-clockwise rotation\n");
#endif
                        // Counter-clockwise rotation up to the X-axis
```

```
                        currentCos = currentCos - sinShift;
                        currentSin = currentSin + cosShift;
                        // Update the rotated theta
                        thetaRotated = thetaRotated - cordicPhase[i
                           ];
                }
#ifdef DEBUG_MODE
                printf("End of iteration\n");
#endif
                // End of iterations, so value can be updated
        }

        // Assign output values
        currentCos = currentCos * cordicGain[NUM_ITERATIONS - 1];
        *mag = currentCos;
        *theta = thetaRotated;
#ifdef DEBUG_MODE
        printf("Magnitude: %f\n", (float)currentCos);
        printf("Angle: %f\n", (float)thetaRotated);
#endif
}
```

Listing C.2: CORDIC source code

Appendix D

Custom datatype source code

```
/**
 * @file datatypes.h
 * @author Alex Stepko (axstepko.com)
 * @brief Custom-width arbitrary datatype definitions.
 * @version 1.0
 * @date 2023-04-27
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
```

```
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE
    FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY
    STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND
     PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING
    BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE
     WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY
    VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR
    VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef DATATYPES_H
#define DATATYPES_H
#include <ap_fixed.h>
#include "ap_int.h"

#define FIXED_BITS_M 24 // 24
#define FIXED_BITS_N 17 // 12
typedef ap_fixed<FIXED_BITS_M, FIXED_BITS_N> FIXED_POINT; //!< AP-
   Fixed datype for consistent calculation in the CORDIC

#define FIR_INT_INPUT_WIDTH 6 // 6 bits wide
#define FIR_INT_OUTPUT_WIDTH 18 // 18 bits wide
typedef ap_int<FIR_INT_INPUT_WIDTH> FIR_INT_INPUT; //!< Fixed-width
   integer datatype for FIR inputs
typedef ap_int<FIR_INT_OUTPUT_WIDTH> FIR_INT_OUTPUT; //!< Fixed-
   width integer datatype for FIR outputs

#endif
```

Listing D.1: CORDIC source code

Appendix E

Vitis IDE source code

```
/**
 * @file main.c
 * @author Alex Stepko (axstepko.com)
 * @brief Runs iterations of a 1-d FIR filter
 * @version 0.1
 * @date 2023-04-26
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
```

```
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE
    FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY
    STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND
     PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING
    BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE
     WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY
    VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR
    VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef GLOBAL_DEFINES
#define GLOBAL_DEFINES
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h" // Parameter definitions for processor
   peripherals
#include "xscugic.h"     // Processor interrupt controller device
   driver
#include "xil_cache.h"
#endif // GLOBAL_DEFINES

#ifndef DEPENDENCIES
#define DEPENDENCIES
#include "IPcontrol.h"
#endif // DEPENDENCIES

#define MAIN_DEBUG_MODE

//#define RANDOM_INPUT_GEN

//#define WORST_CASE 50


#define COEFF_LENGTH 25
#define DATA_LENGTH 30
#define INPUT_LENGTH COEFF_LENGTH + DATA_LENGTH

int main()
{
    systemInit();
    int ret = 0; //!< Status return variable

```

```
    ret = polarFIRinit(&polarFIRinst);
    if (ret != XST_SUCCESS)
    {
        print("Peripheral setup failed\n\r");
        exit(-1);
    }

    printf("Initializing filter\n");

    int inputReal[INPUT_LENGTH] = {};
    int inputImg[INPUT_LENGTH] = {};
#ifdef WORST_CASE
    COEFF_GEN:for (int i = 0; i < COEFF_LENGTH; i++)
        {
            inputReal[i] = WORST_CASE;
            inputImg[i] = WORST_CASE;
        }
#else
COEFF_GEN:for (int i = 0; i < COEFF_LENGTH; i++)
    {
        inputReal[i] = COEFF_LENGTH - i;
        inputImg[i] = i + 1;
    }
#endif
// Generate data input:
#ifdef RANDOM_INPUT_GEN
INPUT_GEN:for (int i = 0; i < DATA_LENGTH; i++)
    {
        inputReal[COEFF_LENGTH + i] = rand() % 50;
        inputImg[COEFF_LENGTH + i] = -1 * rand() * rand() % 50;
    }
#else
INPUT_GEN:
   for(int i = 0; i < DATA_LENGTH; i++)
   {
      inputReal[COEFF_LENGTH + i] = 1;
      inputImg[COEFF_LENGTH + i] = 0;
   }
#endif

#ifdef WORST_CASE
#ifndef RANDOM_INPUT_GEN
   INPUT_GEN:
      for(int i = 0; i < DATA_LENGTH; i++)
      {
         inputReal[COEFF_LENGTH + i] = 50;
         inputImg[COEFF_LENGTH + i] = 50;
      }
#endif
#endif

```

```
    // Locate and clean memory for the peripheral I/O:
   u32 *INPUT_REAL_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR;
   u32 *INPUT_IMG_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x20000;
   u32 *HW_MAG_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR +  0x20000 + 0
      x20000;
   u32 *HW_PHASE_BUFFER = XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x20000+ 0
      x20000 + 0x20000;

    memset(INPUT_IMG_BUFFER, 0, (INPUT_LENGTH << 2));
    memset(INPUT_REAL_BUFFER, 0, (INPUT_LENGTH << 2));
    memset(HW_MAG_BUFFER, 0, (DATA_LENGTH << 2));
    memset(HW_PHASE_BUFFER, 0, (DATA_LENGTH << 2));

INPUT_BUFF_SET:
    for (int i = 0; i < INPUT_LENGTH; i++)
    {
        INPUT_IMG_BUFFER[i] = inputImg[i];
        INPUT_REAL_BUFFER[i] = inputReal[i];
    }

    XPolarfir_Set_inputReal(&polarFIRinst, (u64)INPUT_REAL_BUFFER);
    XPolarfir_Set_inputImg(&polarFIRinst, (u64)INPUT_IMG_BUFFER);
    XPolarfir_Set_inputLength(&polarFIRinst, (u32)DATA_LENGTH);
    XPolarfir_Set_outputMag(&polarFIRinst, (u64)HW_MAG_BUFFER);
    XPolarfir_Set_outputPhase(&polarFIRinst, (u64)HW_PHASE_BUFFER);

#ifdef MAIN_DEBUG_MODE
    printf("Filter set with\n");
    for (int i = 0; i < COEFF_LENGTH; i++)
    {
        printf("coeff[%d] = %d + j%d\n", i, INPUT_REAL_BUFFER[i],
            INPUT_IMG_BUFFER[i]);
    }

    printf("Input set with\n");
    for (int i = 0; i < DATA_LENGTH; i++)
    {
        printf("input[%d] = %d + j%d\n", i, (int)INPUT_REAL_BUFFER[
            COEFF_LENGTH + i], (int)INPUT_IMG_BUFFER[COEFF_LENGTH + i
            ]);
    }
#endif

    printf("Filter initialized.\n");

    int hardwareRet = runHardware();

    if(hardwareRet == 0)
    {
        printf("Result is ready\n");

```

```
        //Somewhere here, we need to access the output arrays, because the result is done.

        /////////////////////////////////////////////////
        printf("Received output from hardware:\n");
        for(int i = 0; i < DATA_LENGTH; i++)
        {
         //printf("hwMag[%d] = %f, ph = %f\n", i, (float*)(HW_MAG_BUFFER + i), (float*)(HW_PHASE_BUFFER + i));
         printf("hwMag[%d] = %f, ph = %f\r", i, (*(float*)(HW_MAG_BUFFER  + i)), (*(float*)(HW_PHASE_BUFFER  + i)));
        }
    }
    else
    {
        printf("Result failed to generate. No output to show. :( \n");
    }

    systemDeInit();
    return 0;
}
```

Listing E.1: Main application source code

Appendix F

Hardware accelerator source code

```
/**
 * @file fpga417_lab4_test.c
 * @author Alex Stepko (axstepko.com), Web Simmara
 * @brief Test application for a simple FIR filter
 * @version 0.1
 * @date 2023-03-22
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
```

```c
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE
     FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY
     STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND
      PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING
     BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE
      WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY
     VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR
     VIEWERS OF THIS SOFTWARE.
 *
 */
#ifndef HW_VALIDATION_INCLUDES
#define HW_VALIDATION_INCLUDES

#include <stdio.h>
#include <stdlib.h>

#endif // HW_VALIDATION_INCLUDES



#ifndef HW_VALIDATION_DEPENDENCIES
#define HW_VALIDATION_DEPENDENCIES

#include "polarFIR.h"

#endif // HW_VALIDATION_DEPENDENCIES

#define HW_VALIDATION_DEBUG_MODE //!< Enable this to have verbose
    debug console output

static int polarDebugInputReal[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 1, 1,
     1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1};
static int polarDebugInputImg[] = {25, 24, 23, 22, 21, 20, 19, 18,
    17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0};

#define LENGTH 27 //!< Length of the data array

int *hwInputReal;  //!< Hardware-accelerated array pointer real
    components
```

```
int *hwInputImg;   //!< Hardware-accelerated array pointer imaginary components
float *hwOutputReal; //!< Hardware-accelerated array pointer real components
float *hwOutputImg;  //!< Hardware-accelerated array pointer imaginary components
float *hwOutputMag; //!< Hardware-accelerated array pointer to the output magnitude
float *hwOutputPhase; //!< Hardware-accelerated array pointer to the phase

int *swInputReal;  //!< Softwae-accelerated array pointer real components
int *swInputImg;   //!< Software-accelerated array pointer imaginary components
float *swOutputReal; //!< Hardware-accelerated array pointer real components
float *swOutputImg;  //!< Hardware-accelerated array pointer imaginary components

void init();
void deinit();
void initializeArray(int **arr, int length);
int compareArray(float *a, float *b, int length);
void copyArray(int *a, int *b, int length);
void complexFIRReference(int *inputReal, int *inputImg, int *kernelReal, int *kernelImg, float *outputReal, float *outputImg, int inputLength, int kernelSize);
void computeComplexFIRReference(int inputReal, int inputImg, int *filterReal, int *filterImg, int filterLength, int *delayLineReal, int *delayLineImg, float *outputReal, float *outputImg);
/**
 * @brief Main application
 */
int main(void)
{
   printf("-----------------  Start validation  -------------------\n");
    init();
    printf("Arrays initialized.\n");


#ifndef HW_VALIDATION_DEBUG_MODE
    printf("Running filters...\n");
    complexFIR(hwInputReal, hwInputImg, debugCoeffReal, debugCoeffImg, hwOutputReal, hwOutputImg);
    complexFIRReference(swInputReal, swInputImg, debugCoeffReal, debugCoeffImg, swOutputReal, swOutputImg, LENGTH, LENGTH);
    printf("Filters complete.\n");
#else
    //printf("Running filters...\n");
```

```
    //complexFIR(debugInputReal, debugInputImg, debugCoeffReal,
       debugCoeffImg, hwOutputReal, hwOutputImg);
    //complexFIRReference(debugInputReal, debugInputImg,
       debugCoeffReal, debugCoeffImg, swOutputReal, swOutputImg,
       LENGTH, LENGTH);
    //printf("Filters complete.\n");
#endif

    //printf("Comparing arrays...\n");
//    if ((compareArray(hwOutputReal, swOutputReal, LENGTH) == 0) ||
      (compareArray(hwOutputImg, swOutputImg, LENGTH) == 0))
//    {
//        printf("TEST FAILED. ELEMENTS DO NOT MATCH");
//    }
//    else
//    {
//        printf("Test passed\n");
//    }

    deinit();
    //printf("Arrays freed.\n");

    printf("Running CORDIC...\n");
    double cosDouble;
    double sinDouble;

    FIXED_POINT cos = -2.0;
    FIXED_POINT sin = -2.0;
    FIXED_POINT mag = 0.0;
    FIXED_POINT theta = 0.0;

    //cordic(cos, sin, &mag, &theta);

    cosDouble = cos.to_double();
    sinDouble = sin.to_double();

    printf("Running combo system...\n");
    polarFir(polarDebugInputReal, polarDebugInputImg, hwOutputMag,
       hwOutputPhase, LENGTH);

    for(int i = 0; i < LENGTH; i++)
    {
      printf("Output magnitude[%d]: %f, phase: %f\n", i, hwOutputMag
         [i], hwOutputPhase[i]);
    }
```

```
    printf("---------------  End of validation program
        ---------------\n");
    return 0;
}

/**
 * @brief Initializes arrays and copies software array to hardware
     array
 */
void init()
{
    srand(0);
    // Initialize the real input arrays:
    initializeArray(&hwInputReal, LENGTH);
    initializeArray(&swInputReal, LENGTH);
    copyArray(hwInputReal, swInputReal, LENGTH);


    // Initialize the imaginary input arrays:
    initializeArray(&hwInputImg, LENGTH);
    initializeArray(&swInputImg, LENGTH);
    copyArray(hwInputImg, swInputImg, LENGTH);

    //Initialize output arrays (really, allocate memory and set to
       zero)
    hwOutputReal = (float *)malloc(sizeof(float) * LENGTH); //!<
       Allocated space for random number array
    hwOutputImg = (float *)malloc(sizeof(float) * LENGTH); //!<
       Allocated space for random number array
    swOutputReal = (float *)malloc(sizeof(float) * LENGTH); //!<
       Allocated space for random number array
    swOutputImg = (float *)malloc(sizeof(float) * LENGTH); //!<
       Allocated space for random number array
    hwOutputMag = (float* )malloc(sizeof(float) * LENGTH);
    hwOutputPhase = (float* )malloc(sizeof(float) * LENGTH);
    for(int i = 0; i < LENGTH; i++)
    {
      hwOutputReal[i] = 0.0;
      hwOutputImg[i] = 0.0;
      swOutputReal[i] = 0.0;
      swOutputImg[i] = 0.0;
      hwOutputMag[i] = 0.0;
      hwOutputPhase[i] = 0.0;
    }
}

/**
 * @brief Frees space in the created arrays to avoid a memory
     leakage.
 */
void deinit()
```

```
{
    free(hwInputReal);
    free(hwInputImg);
    free(swInputReal);
    free(swInputImg);
}

/**
 * @brief Allocates memory and initializes array to a set of random
     numbers
 *
 * @param arr Pointer to the input array
 * @param length Length of the input array
 */
void initializeArray(int **arr, int length)
{
    *arr = (int *)malloc(sizeof(int) * length); //!< Allocated space
        for random number array
    for (int idx = 0; idx < length; ++idx)
    {
      (*arr)[idx] = rand() % 255;
    }

}

/**
 * @brief Compares two arrays searching for a different value
 *
 * @param a Array 1 to be compared
 * @param b Array 2 to be compared
 * @param length Length of both arrays to be compared
 * @return int return status of the function (0 = ok, 1 = fail)
 */
int compareArray(float *a, float *b, int length)
{

    for (int i = 0; i < length; ++i)
        if (a[i] != b[i])
            return 1;
    return 0;
}

/**
 * @brief Copies array b to a
 *
 * @param a Pointer to an array to be copied into
 * @param b Pointer to an array to be copied from
 * @param length Length of the arrays.
 */
void copyArray(int *a, int *b, int length)
{
```

```c
    for (int i = 0; i < length; ++i)
        a[i] = b[i];
}

/////////////////////// REFERENCE FILTER DESIGNS
    //////////////////////
/**
 * @brief Complete processor for a 1-d FIR filter
 * @remark The real and imaginary parts of the input, and kernel,
     must have the same length.
 *
 * @param inputReal Real part of data input
 * @param inputImg  Imaginary part of data input
 * @param kernelReal Real part of filter coefficents
 * @param kernelImg Pointer to imaginary part of filter coefficents
 * @param outputReal Real part of filter output
 * @param outputImg Imaginary part of filter output
 * @param inputLength Length of the input dataset
 * @param kernelSize Length of the filter coefficent dataset
 */
void complexFIRReference(int *inputReal, int *inputImg, int *
    kernelReal, int *kernelImg, float *outputReal, float *outputImg,
    int inputLength, int kernelSize)
{
#ifdef DEBUG_MODE
    printf("Start of hardware FIR...\n");
#endif
    int filterReal[kernelSize]; //!< Filter coefficent buffer (real)
    int filterImg[kernelSize];  //!< Filter coefficent buffer (
        imaginary)

    int delayLineReal[kernelSize]; //!< Input pipeline delay buffer
        (real)
    int delayLineImg[kernelSize];  //!< Input pipeline delay buffer
        (imaginary)
    // ^^ These must remain in the top-level processing function to
        retain "static" qualities when instantiating multiple filter
        passes.

    float tempR, tempI; //!< Raw output from a filter pass

LOAD_FILTER:
    for (int i = 0; i < kernelSize; i++)
    {
        filterReal[i] = kernelReal[i];
        filterImg[i] = kernelImg[i];
    }

PIPELINE_DELAY_ARRAY_INIT:
    for (int j = 0; j < kernelSize; j++)
    {
```

```
        delayLineReal[j] = 0;
        delayLineImg[j] = 0;
    }

#ifdef DEBUG_MODE
    printf("Loaded coefficients:\n");
    for (int a = 0; a < kernelSize; a++)
    {
        printf("filterReal[%d] = %d, filterImg[%d] = %d\n", a,
           filterReal[a], a, filterImg[a]);
    }
#endif

COMPUTE:
    for (int k = 0; k < inputLength; k++)
    {
        // Perform a single pass of an input with the coefficents:
        computeComplexFIRReference(inputReal[k], inputImg[k],
           filterReal, filterImg, kernelSize, delayLineReal,
           delayLineImg, &tempR, &tempI);
        outputReal[k] = tempR;
        outputImg[k] = tempI;
#ifdef DEBUG_MODE
        printf("outReal = %f, outImg = %f\n", tempR, tempI);
#endif
    }
}

/**
 * @brief Performs a FIR on a single element of a complex datapoint
 *
 * @param inputReal Real part of input sample
 * @param inputImg Imaginary part of input sample
 * @param filterReal Real part of filter coefficents
 * @param filterImg Imaginary part of filter coefficents
 * @param filterLength Length of the filter
 * @param delayLineReal Real part of pipleine delay component
 * @param delayLineImg Imaginary part of pipeline delay component
 * @param outputReal Real part of discrete output
 * @param outputImg Imaginary part of discrete output
 */
void computeComplexFIRReference(int inputReal, int inputImg, int *
   filterReal, int *filterImg, int filterLength, int *delayLineReal,
    int *delayLineImg, float *outputReal, float *outputImg)
{
    float resultReal, resultImg = 0.0; //!< Temporary result hold
       for the filter pass

PIPELINE_DELAY:
    for (int i = filterLength - 1; i >= 1; i--)
    {
```

```
        // Iterate backwards through the array to shift to the right.
        delayLineReal[i] = delayLineReal[i - 1];
        delayLineImg[i] = delayLineImg[i - 1];
    }
    // Add the new input sample to the beginning of the delay line arrays
    delayLineReal[0] = inputReal;
    delayLineImg[0] = inputImg;

FILTER_PASS:
    for (int j = 0; j < filterLength; j++)
    {
        // Calculate the real and imaginary parts of the convolution
        // output using the filter coefficients and the delay line.
        resultReal += (delayLineReal[j] * filterReal[j]) - (delayLineImg[j] * filterImg[j]);
        resultImg += (delayLineReal[j] * filterImg[j]) + (filterReal[j] * delayLineImg[j]);
    }

    // Send the output
    // Update the output pointers with the computed real and imaginary parts
    *outputReal = resultReal;
    *outputImg = resultImg;
}
```

Listing F.1: Hardware validation source code

Appendix G

Hardware driver source code

```c
// ==============================================================
// Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2022.2
    (64-bit)
// Tool Version Limit: 2019.12
// Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
// ==============================================================
#ifndef XPOLARFIR_H
#define XPOLARFIR_H

#ifdef __cplusplus
extern "C" {
#endif

/*************************** Include Files
    *********************************/
#ifndef __linux__
#include "xil_types.h"
#include "xil_assert.h"
#include "xstatus.h"
#include "xil_io.h"
#else
#include <stdint.h>
#include <assert.h>
#include <dirent.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stddef.h>
#endif
#include "xpolarfir_hw.h"

/*************************** Type Definitions
    *****************************/
```

```
#ifdef __linux__
typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;
#else
typedef struct {
    u16 DeviceId;
    u64 Control_BaseAddress;
} XPolarfir_Config;
#endif

typedef struct {
    u64 Control_BaseAddress;
    u32 IsReady;
} XPolarfir;

typedef u32 word_type;

/**************** Macros (Inline Functions) Definitions
    *******************/
#ifndef __linux__
#define XPolarfir_WriteReg(BaseAddress, RegOffset, Data) \
    Xil_Out32((BaseAddress) + (RegOffset), (u32)(Data))
#define XPolarfir_ReadReg(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (RegOffset))
#else
#define XPolarfir_WriteReg(BaseAddress, RegOffset, Data) \
    *(volatile u32*)((BaseAddress) + (RegOffset)) = (u32)(Data)
#define XPolarfir_ReadReg(BaseAddress, RegOffset) \
    *(volatile u32*)((BaseAddress) + (RegOffset))

#define Xil_AssertVoid(expr)    assert(expr)
#define Xil_AssertNonvoid(expr) assert(expr)

#define XST_SUCCESS             0
#define XST_DEVICE_NOT_FOUND    2
#define XST_OPEN_DEVICE_FAILED  3
#define XIL_COMPONENT_IS_READY  1
#endif

/************************* Function Prototypes
    ****************************/
#ifndef __linux__
int XPolarfir_Initialize(XPolarfir *InstancePtr, u16 DeviceId);
XPolarfir_Config* XPolarfir_LookupConfig(u16 DeviceId);
int XPolarfir_CfgInitialize(XPolarfir *InstancePtr, XPolarfir_Config
     *ConfigPtr);
#else
int XPolarfir_Initialize(XPolarfir *InstancePtr, const char*
    InstanceName);
```

```
int XPolarfir_Release(XPolarfir *InstancePtr);
#endif

void XPolarfir_Start(XPolarfir *InstancePtr);
u32 XPolarfir_IsDone(XPolarfir *InstancePtr);
u32 XPolarfir_IsIdle(XPolarfir *InstancePtr);
u32 XPolarfir_IsReady(XPolarfir *InstancePtr);
void XPolarfir_EnableAutoRestart(XPolarfir *InstancePtr);
void XPolarfir_DisableAutoRestart(XPolarfir *InstancePtr);

void XPolarfir_Set_inputReal(XPolarfir *InstancePtr, u64 Data);
u64 XPolarfir_Get_inputReal(XPolarfir *InstancePtr);
void XPolarfir_Set_inputImg(XPolarfir *InstancePtr, u64 Data);
u64 XPolarfir_Get_inputImg(XPolarfir *InstancePtr);
void XPolarfir_Set_outputMag(XPolarfir *InstancePtr, u64 Data);
u64 XPolarfir_Get_outputMag(XPolarfir *InstancePtr);
void XPolarfir_Set_outputPhase(XPolarfir *InstancePtr, u64 Data);
u64 XPolarfir_Get_outputPhase(XPolarfir *InstancePtr);
void XPolarfir_Set_inputLength(XPolarfir *InstancePtr, u32 Data);
u32 XPolarfir_Get_inputLength(XPolarfir *InstancePtr);

void XPolarfir_InterruptGlobalEnable(XPolarfir *InstancePtr);
void XPolarfir_InterruptGlobalDisable(XPolarfir *InstancePtr);
void XPolarfir_InterruptEnable(XPolarfir *InstancePtr, u32 Mask);
void XPolarfir_InterruptDisable(XPolarfir *InstancePtr, u32 Mask);
void XPolarfir_InterruptClear(XPolarfir *InstancePtr, u32 Mask);
u32 XPolarfir_InterruptGetEnabled(XPolarfir *InstancePtr);
u32 XPolarfir_InterruptGetStatus(XPolarfir *InstancePtr);

#ifdef __cplusplus
}
#endif

#endif
```

Listing G.1: HLS-generated driver declarations

```
/**
 * @file IPcontrol.h
 * @author Alex Stepko (axstepko.com)
 * @brief Header file for higher-level bulk control of hardware IP
   in the design
 * @version 0.1
 * @date 2023-04-26
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All
   rights reserved.
 *
 */

/**
```

```c
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY STATED ABOVE, IN
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR VIEWERS OF THIS SOFTWARE.
 *
 */

#ifndef IPCONTROL_H
#define IPCONTROL_H
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xpolarfir.h"
#include "xpolarfir_hw.h"

extern XPolarfir polarFIRinst;
```

```

void systemInit();
void systemDeInit();
int polarFIRinit(XPolarfir *instance);
void polarFIRstart(void *instance);
int runHardware();


#endif // IPCONTROL_H
```

Listing G.2: Driver interaction function (header)

```
/**
 * @file IPcontrol.c
 * @author Alex Stepko (axstepko.com)
 * @brief Higher-level bulk control of hardware IP in the design
 * @version 0.1
 * @date 2023-04-26
 *
 * @copyright  Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 */

/**
 * Copyright (c) Alex Stepko (axstepko.com) 2023. All rights reserved.
 *
 * THIS CODE WRITTEN WHOLY BY ALEX STEPKO (AXSTEPKO.COM) Redistribution and
 * use in source and binary forms, with or without modification, are
 * STRICTLY PROHIBITED WITHOUT PRIOR WRITTEN PERMISSION from Alex Stepko (axstepko.com).
 *
 * This software is provided "as is" and any express or implied warranties, including,
 * but not limited to, the implied warranties of merchantability and fitness for a
 * particular purpose are disclaimed. In no event shall Alex Stepko (axstepko.com) be
 * liable for any direct, indirect, incidental, special, exemplary, or consequential
 * damages (including, but not limited to, procurement of substitute goods or services;
 *
 * ACADEMIC INTEGRITY: STUDENTS, RESEARCHERS, OR INSTITUTIONS MAY USE THIS SOFTWARE
 * FOR EDUCATIONAL OR RESEARCH PURPOSES ONLY. YOU ARE RESPONSIBLE FOR ENSURING THAT
 * THE USAGE OF THIS SOFTWARE COMPLIES WITH THE COPYRIGHT POLICY STATED ABOVE, IN
```

```
 * ADDITION TO YOUR INSTITUTION'S POLICIES ON ACADEMIC INTEGRITY AND
    PLAGARISM.
 * IT IS PROHIBITED TO REPRESENT THIS WORK AS YOUR OWN, INCLUDING
   BUT NOT LIMITED
 * TO THE DUPLICATION, MODIFICATION, OR DISTRUBTION OF THIS SOFTWARE
    WITHOUT PRIOR
 * WRITTEN PREMISSION FROM THE SOFTWARE'S OWNER (ALEX STEPKO).
 * ALEX STEPKO (AXSTEPKO.COM) DISCLAIMS ALL LIABILITY FOR ANY
   VIOLATIONS OF ACADEMIC
 * INTEGRITY OR OTHER ETHICAL STANDARDS COMMITTED BY USERS OR
   VIEWERS OF THIS SOFTWARE.
 *
 */

#include "IPcontrol.h"

XPolarfir polarFIRinst;

/**
 * @brief Initializes the processing system
 *
 */
void systemInit()
{
    printf("Initializing processing system.\n");
    init_platform();
    Xil_DCacheEnable();
    srand(0);
    printf("Initialization complete...\n");
}

/**
 * @brief De-initializes the processing system
 *
 */
void systemDeInit()
{
    printf("Cleaning system\n");
    cleanup_platform();
}

/**
 * @brief Initializes an instance of the hardware accelerator
 *
 * @param instance
 * @return int general status return. Getting this return means the
   accelerator is ready to be loaded with input.
 */
int polarFIRinit(XPolarfir *instance)
{
    XPolarfir_Config *configPtr;
```

```
    int status = 0;
    printf("Initializing accelerator...\n");
    configPtr = XPolarfir_LookupConfig(XPAR_POLARFIR_0_DEVICE_ID);
    if (!configPtr)
    {
        fprintf(stderr, "ERROR: Failed to find accelerator
            configuration\n\r");
        return XST_FAILURE;
    }
    print("SUCCESS: Accelerator config found..\n\r");
    status = XPolarfir_CfgInitialize(instance, configPtr);
    if (status != XST_SUCCESS)
    {
        print("ERROR: Failed to initialize accelerator.\n\r");
        return XST_FAILURE;
    }
    print("SUCCESS: Accelerator initialized...\n\r");
    return status;
}

/**
 * @brief Primes the accelerator for run.
 *
 * @param instance Pointer to the accelerator instance
 */
void polarFIRstart(void *instance)
{
    XPolarfir *pAccel = (XPolarfir *)instance;
    XPolarfir_InterruptEnable(pAccel, 1);
    XPolarfir_InterruptGlobalEnable(pAccel);
    XPolarfir_Start(pAccel);
}

/**
 * @brief Runs the hardware accelerator
 *
 * @return int status return when hardware is done writing. 0 is OK,
     1 is error.
 */
int runHardware()
{
   int c = 0;
    Xil_DCacheFlush();
    printf("Wait for accel ready");
    while(!XPolarfir_IsReady(&polarFIRinst))
    {
      printf(".");
    }
    printf("\n");
    if (XPolarfir_IsReady(&polarFIRinst))
    {
```

```
        print("Starting peripheral...\n");
    }
    else
    {
        print("ERROR: peripheral not ready to run...\n\r");
        return 1;
    }
    XPolarfir_Start(&polarFIRinst);
    c = 0;
    while (!XPolarfir_IsReady(&polarFIRinst))
    {
        printf("Waiting for completion... %i\r", ++c);
    }
    Xil_DCacheInvalidate();
    print("Peripheral complete\n");
    return 0;
}
```

Listing G.3: Driver interaction function