FPGA Qubit Package
Final Project Report

Francisca Vasconcelos and Megan Yamoah

6.111 Fall 2018
Hom and Steinmeyer
Project Supervisor: Diana Wofk

Abstract

FPGAs have wide-ranging applications across a multitude of disciplines, from bioinformatics to signal processing. In the field of quantum information and, specifically, quantum computing, FPGAs have unique application as a fast classical interface for pre-processing quantum signals. As such, we propose to implement these features on an FPGA to speed up the measurement process for the Engineering Quantum Systems superconducting quantum processors.

Contents

1 System Overview

Our project aimed to tackle a key component of superconducting qubit measurement: state discrimination. This requires computing the phase shift between two voltage signals, mapping on the complex I-Q plane, followed by either linear classification or the creation of a 2D histogram.

Since our project is based on current work in the quantum information, it was instructive to work with hardware used in practice in research labs. Specifically, we used the PXI platform developed by Keysight which provides boards based on the Xilinx Kintex-7 family. These Keysight boards include various useful connectivity and IC options such as ADCs, DACs, and SMA connections which allow the FPGAs to be use more effectively as digitizers or arbitrary waveform generators, both invaluable systems for our project. Further, we have, in our research group, access to signal generators which were used to represent expected input signals from a qubit device.

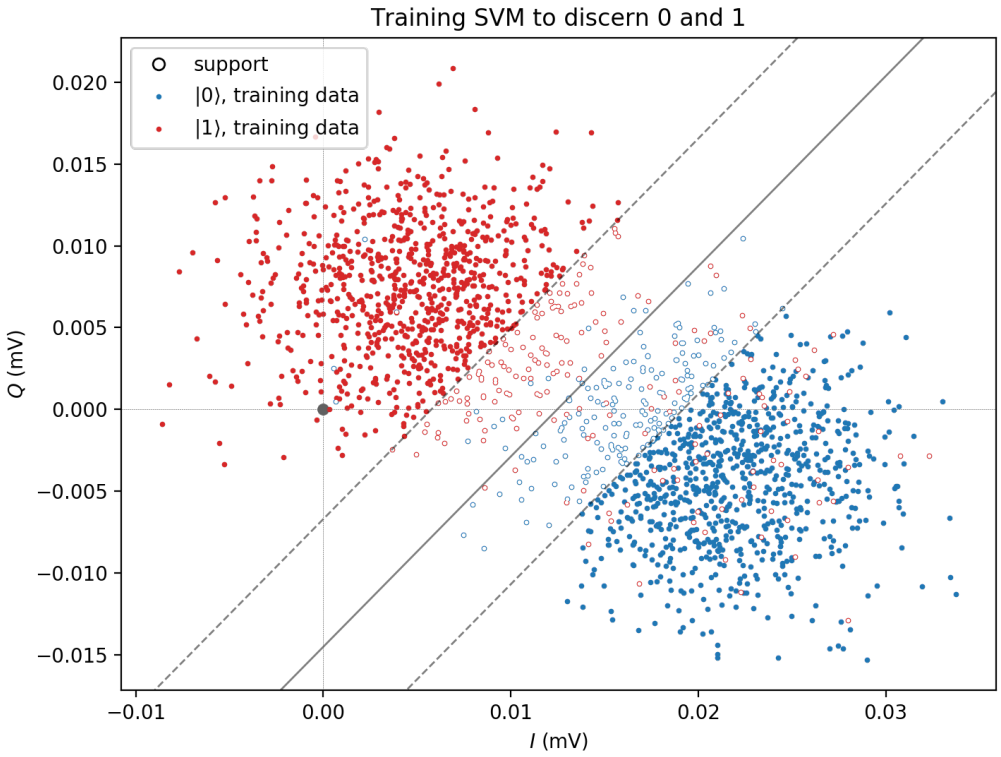

Figure 1: Sample plot showing complex plane with measured states in the ground (blue) and excited (red) state. The SVM was trained on this data to learn the classification between the ground and excited state, serving as input for the linear classification module in our own work.

The project has been split into two halves: data demodulation (pre-processing) and data analysis within the FPGA with Megan taking the former and Fran the latter. Specifically, we use the FPGA to extract a phasor from analog quantities using a rotation on the down-converted microwave signals. Once the states are thus mapped onto the complex, I-Q plane, we then classify the state as described in Fig. 1. To do so, we load a linear discriminator onto our FPGA, which would have been found using previous measurement data and machine learning (i.e. an SVM) to discriminate between the ground and excited state of our qubit. This linear classifier will then be used to perform a continuous

binning of qubit data over a series of a few thousand measurements.

Outside of our FPGA, we will use a pulse generator to generate appropriately phase shifted signals at 10 MHz, representing the I and Q components of our qubit response. These signals will then be fed through an ADC and into a "sampling" module. The sampling module will output the two voltage signals over a specified time interval over which another module will perform an rotation to determine the phase shift. The full system block diagram is shown in Fig. 2.

We note that additional work was done to interface the FPGA with the external lab hardware. This included writing drivers which interface with the FPGA, visualize the output data coming from the FPGA, integrate the system with Labber (our lab's core instrumentation and measurement software), and easily load the classifier parameters onto the FPGA.

2 Physics Background

State discrimination is an important and non-trivial task for quantum computing due to the central difference between quantum and classical computers. Namely, quantum bits, or qubits, store information in a complex sphere. One qubit holds a state $|\psi\rangle$ such that

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$ to maintain the quantum mechanical normalization constraint on wavefunctions. States $|0\rangle$ and $|1\rangle$ are the ground and excited state of the qubit respectively and are the closest representation of the classical 0 and 1 with which we are familiar.

Since a single qubit state exists in a two-dimensional complex space, it in fact lives, so to speak, on a three dimensional sphere, called the Bloch sphere as shown in Fig. 3. To measure quantum states, since we cannot measure complex values, we must choose a basis onto which to project our state. In practice, this is conventionally defined as the z-axis. Further, when we measure the qubit, it will collapse into either the +1 or -1 direction in our chosen basis, so any one measurement on a qubit can only produce a discrete value on a chosen basis. To determine the angle of a state along an axis, quantum mechanics requires us to repeat our measurement a large number of times N (error $\propto 1/N$).

For superconducting qubits, a single measurement step involves measuring the phase shift of a sinusoidal signal which is sent to the qubit which sits at 10mK. The qubit has a state dependent effect on the resonator resonance frequency, inducing a qubit-state dependent phase shift in microwave transmission signals sent to the qubit. In what is termed "heterodyne" detection, a signal $\omega_{RF} \approx 5$GHz near the qubit frequency is sent through a transmission line to which a qubit resonator is coupled. Upon exit, this signal is then I-Q mixed with a local oscillator ω_{LO} sitting at 10 MHz below ω_{RF}. As such, the down-converted signal is split into two lower frequency signals separated by a $\frac{\pi}{2}$ phase shift whose phase shift from the local oscillator can be represented on the complex plane as a phasor on the I-Q plane.

The qubit-state dependent nature of the phase allows us to then use this value to determine the qubit's collapsed state and, over a series of numerous measurements and rotations, construct the full qubit state. For one qubit, the I-Q values cluster as in Fig. 1 and one can use an SVM determined using software to create a linear discriminator line which classifies the qubit state as ground (0) or excited (1).

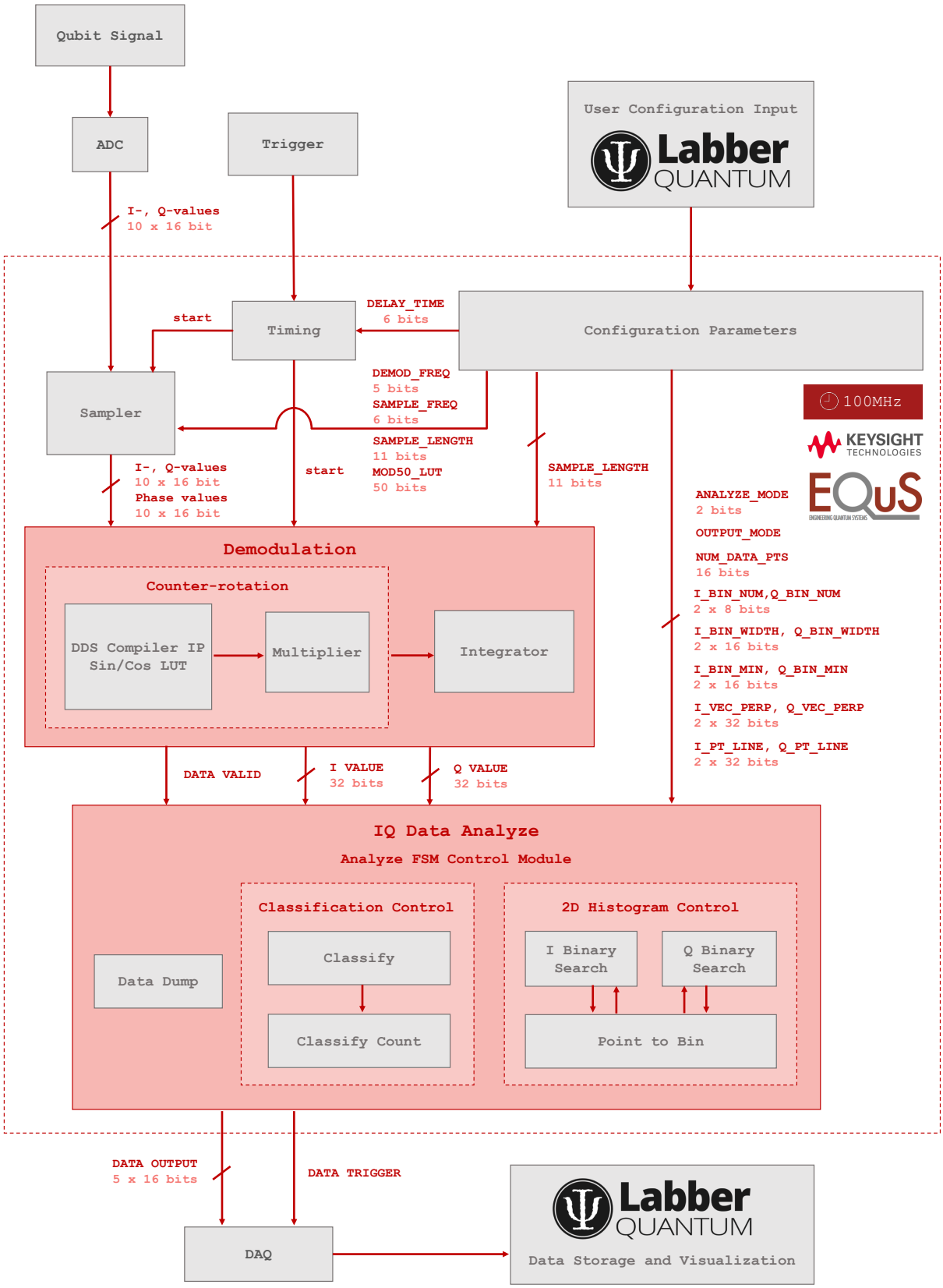

Figure 2: Final project block diagram.

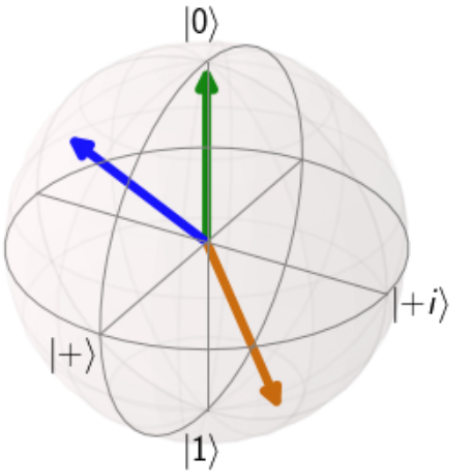

Figure 3: Bloch sphere showing a state vector in the ground (green), or $|0\rangle$, as well as two other arbitrary states (blue, orange). Also shown are the $|+\rangle = 1/\sqrt{2}\,(|0\rangle + |1\rangle)$ and the $|+i\rangle = 1/\sqrt{2}\,(|0\rangle + i\,|1\rangle)$ axes.

3 Demodulation (Megan Yamoah)

3.1 System Function

The demodulation half of the project does several pre-processing functions resulting in averaged I- and Q-values which are subsequently send to the data analysis modules written by Francisca.

The demodulation module takes as input the signals from the I-Q mixer. These signals are

$$\mathrm{I} = \cos(\omega_{IF} t + \phi) \tag{2}$$

$$\mathrm{Q} = \sin(\omega_{IF} t + \phi) \tag{3}$$

where ω_{IF} is an intermediate frequency equal to $|\omega_{RF} - \omega_{LO}| = 10$ MHz. Our goal is to extract the qubit-state-dependent ϕ. To do so, we need to perform a counter-rotation operation such that:

$$\begin{aligned} \mathrm{I}' &= \cos\phi \\ &= \mathrm{I} \cdot \cos(\omega_{IF} t) + \mathrm{Q} \cdot \sin(\omega_{IF} t) \\ \mathrm{Q}' &= \sin\phi \\ &= \mathrm{Q} \cdot \cos(\omega_{IF} t) - \mathrm{I} \cdot \sin(\omega_{IF} t). \end{aligned} \tag{4}$$

Our new values I' and Q' now represent a location on the complex plane which is dependent on the qubit's state. We aim to perform this calculation on each of the points sampled on the incoming I- and Q- singals for each measurement run and then integrate (sum) all of them to average out noise.

To complete this task, the demodulation process involves:

- Storing and updating configuration parameters
- Acquiring and sampling value from the ADC

- Determining the corresponding counter-rotation phase values
- Inputting the phase values in a DDS Compiler LUT
- Multiplying the phase values with the original sampled values
- Integrating the values to average out noise
- Passing integrated values to data analysis modules

A detailed block diagram of these modules is shown in Fig. 4.

The FPGA must sample over a series of five input values for both the I- and Q- inputs every clock cycle since the FPGA runs at a clock of 100 MHz while the ADC samples at a rate of 500 MHz. More details on how these values are sampled are presented in Section 3.2.3. While the FPGA samples values, corresponding phase values must be determined to counter-rotate the signal at its down-converted frequency, extracting the relevant phasor as described in Section 2. These phase values are then used to calculate sine and cosine values which are then multiplied with the original incoming data. Finally, the values are averaged by integration before being passed to the data analysis modules detailed in Section 4.

3.2 Module Overview

The demodulation process involves four distinct modules along with a configuration module which sets the parameters across the entire device. A summary and detailed technical description of our modules follow.

- Configuration
  - Configure parameters by interfacing with Labber software
  - Pass parameters to lower modules
- Timing
  - Implement delay between trigger and start of signal sampling
- Sampling
  - Sample incoming data at appropriate time stamps set by configuration parameters
  - Calculate and set corresponding phase values
- Multiplication
  - Rotate incoming data to extract phase parameters by multiplying appropriate sine and cosine values
  - Instantiate required DDS compilers (Sine/Cosine LUT only mode)
- Integration
  - Integrate rotated values over the sampled time for averaging and noise reduction
  - Output integrated I- and Q-values to Data Analysis modules

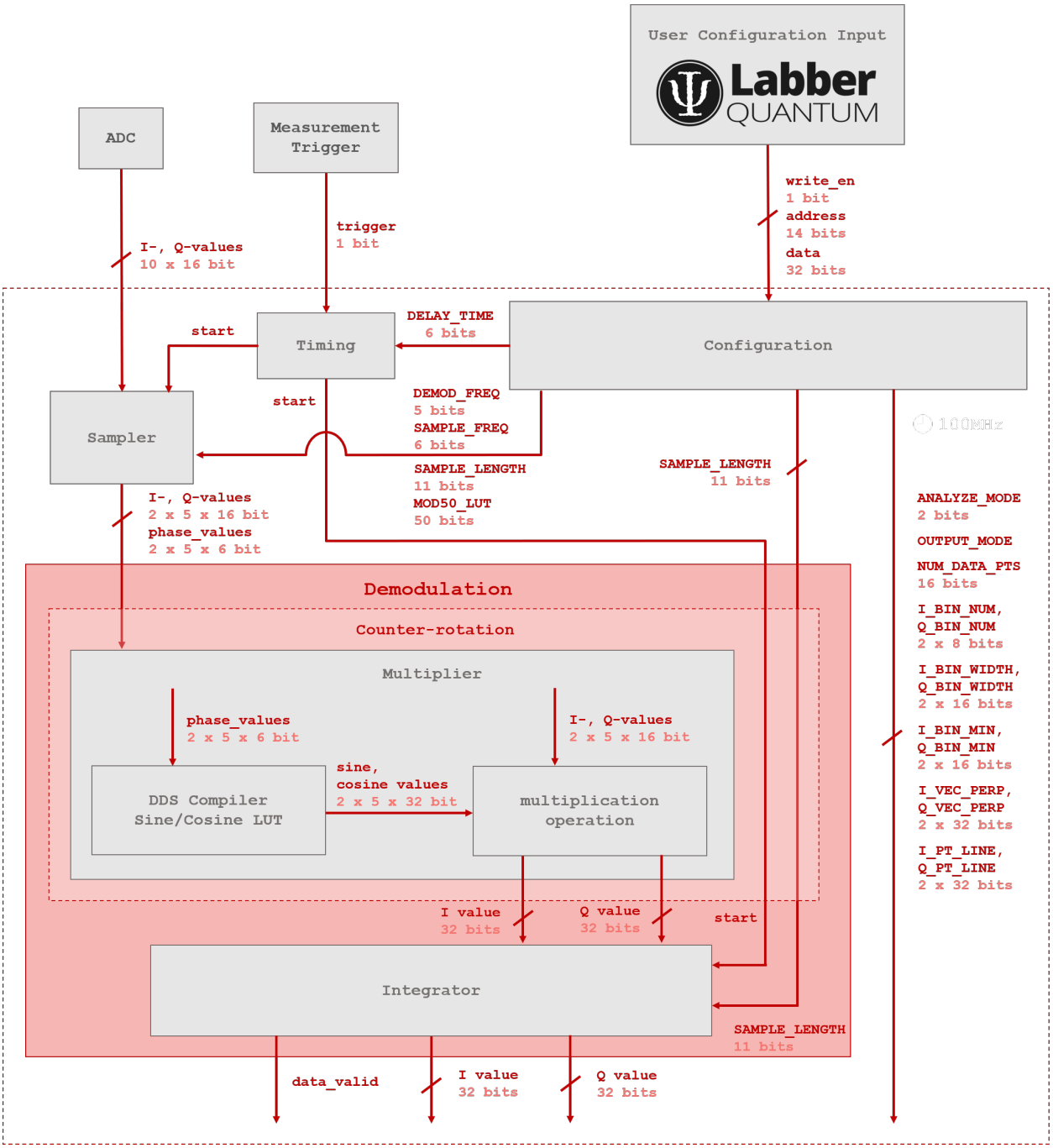

Figure 4: Block diagram of top modules written by Megan.

3.2.1 Configuration

The Configuration module works in tandem with a Python-based Labber driver to receive and set various configuration parameters for the demodulation process such has sampling length, frequency, skip time as well as various data analysis parameters. The module interfaces with the computer through the Python driver using a simple master-slave protocol. The master (Python driver/computer) asserts a write signal along with an address corresponding to the parameter to be modified and a

data string containing the new parameter value. Upon receiving the write signal, the configuration module in the FPGA updates the relevant parameter register which is wired to downstream modules as necessary. Here, we do not the use read functionality of the protocol.

3.2.2 Timing

The Timing module employs a simple state machine and counter to add a “skip time" before the Sampling module begins sampling. The module takes in a delay time parameter from the configuration module as well as the full system trigger sent to the FPGA by Labber at the beginning of the measurement process. Then, it outputs a start signal for the Sampling module.

3.2.3 Sampling

The Sampling module performs two important tasks: 1. sampling the incoming data at appropriate time stamps and 2. calculating the corresponding phase values for the Multiplication module. The data is presented by the ADC to the FPGA as five 16 bit values per I and Q channel at each clock cycle. The sampling frequency requested by the user can then range from 500 MHz (5 times the clock frequency) to an arbitrarily small value. We choose a minimum of 10 MHz. An interesting task is, then, to appropriately sample and store the values of interest. From user inputs and through the driver, we can provide the Sample module with a *demod_freq*, *sample_length*, and *sample_freq*, as well as the input data. The sample frequency is supplied to the FPGA as the number of samples to skip from the 500 MHz data sampled by the ADC. For example, if the user requests a sampling frequency of 50 MHz, the “sample frequency" given to the FPGA is the value 10.

Because in the process of managing five separate inputs for each of our two channels, we find that using arrays and addressing indices are extremely useful for simplifying code, we turn to the efficient packed array type available in SystemVerilog and take advantage of its functionality for much of the demodulation code.

For the sampling process, we must treat sampling skip values of less than five differently than values greater than five since one can use a simple countdown counter for sampling skips greater than five. For sampling skips less than five, we have four possibilities: 1, 2, 3, or 4. While determining the samples to keep on the first clock cycle is straightforward (say, every other sample for a skip value of two, ending with a total of three samples in the first clock cycle), subsequent cycles are influenced by the location of the last point sampled. Continuing the example with a skip value of two, on the first clock cycle, we chose the first, third, and fifth values provided by the ADC. For the second clock cycle, we chose the second and fourth, and so on. Since these patterns become increasing complicated with skip values like three or four, we create a simple LUT that determines the number of samples to skip over in the next clock cycle based on the skip value of the current cycle as well as the sample frequency. For completeness, our LUT is shown in Table 1.

Next, the Sampling module is also responsible for determining the phase values ($\omega_{IF}t$, from before) associated with each sampled point. We mark non-sampled points with a phase value of zero. Of importance is the fact that the DDS compilier LUT which we will use in the next module only takes in values up to a certain magnitude (we choose 50 for reasons to be discussed). This requires us to perform a modulo 50 operation which is non-trivial in hardware. However, we are able

Sample Frequency (skip) \ Skip Value	0	1	2	3
1	0	0	0	0
2	1	0	1	0
3	1	2	0	1
4	3	0	1	2

Table 1: Sampling LUT.

to simplify the modulo operation and create a 50-entry LUT of 6 bit values which we can load onto the FPGA as a configuration parameter—offloading the complex mod 50 calculation to software.

The DDS LUT, set in rasterized mode, splits the unit circle into some user-defined modulus M-1 points and creates an table entry for sine and cosine at each of these points. We choose M such that 1/M defines the smallest change in phase we expect to need to calculate. In other words, we choose our required precision.

Namely, we note that our phase, normalized by $2 * \pi$, is equal to the demodulation frequency (f_{IF}) multiplied by the corresponding phase time step. Our minimum phase, considering the ADC's 500 MHz sampling rate, is then

$$\frac{\phi_{\min}}{2\pi} = \Phi_{\min} = \text{min. demod frequency} \times \text{min. time step} \tag{5}$$

$$= 10 \text{ MHz} \times \frac{1}{500 \text{ MHz}} \tag{6}$$

$$= \frac{1}{50}. \tag{7}$$

Thus, we choose M = 50. Now, however, we must create a method for solving for arbitrary values of mod 50. We can solve this by first simplifying our modulus operation and then creating an LUT dependent on the demodulation frequency. We establish a counter c which equals the current clock cycle where $c = 0$ occurs when *start* is asserted as well as a variable i which refers to the values index within the 5 value array presented by the ADC at each clock cycle. Each phase value will then be represented as

$$\Phi_{\min} = (5 \cdot c + i) \cdot \frac{f_{IF}}{10 \text{ MHz}} \pmod{50} \tag{8}$$

since each tick of one represents one time step normalized by the minimum demodulation frequency of 10 MHz and minimum time step of 0.02 ns. The demodulation frequency presented to the FPGA by the Configuration module is already normalized by 10 MHz. We can simplify the above to eliminate the need for a LUT which spans the entire range of the counter c (up to the sample length of around 20 μs / 2000 clock cycles or more). Namely, we create an inner counter c_m which performs (mod 10) on the counter c. Then, we can simplify $\Phi_{\min}$ in the following way:

$$\begin{aligned}
\Phi_{\min} &= (5 \cdot c + i) \cdot f_{\text{demod}} \pmod{50} \qquad (9)\\
&= ((5 \cdot c + i) \pmod{50} \cdot f_{\text{demod}} \pmod{50}) \pmod{50}\\
&= ((5 \cdot (10n + c_m) \pmod{50} + i) \pmod{50} \cdot f_{\text{demod}} \pmod{50}) \pmod{50}\\
&= ((5 \cdot c_m + i) \pmod{50} \cdot f_{\text{demod}} \pmod{50}) \pmod{50}\\
&= (5 \cdot c_m + i) \cdot f_{\text{demod}} \pmod{50} \qquad (10)
\end{aligned}$$

for n, $c_m \in \mathbb{Z}$ and $0 \leq c_m < 10$. Now, we only need to create an LUT of 50 values ranging from 0 to 49 (6 bit). We create this LUT in the Python driver and load it onto the FGPA as a configuration parameter which is accessed by the Sampling module as appropriate.

3.2.4 Multiplication

The Multiplication module simply takes the outputs of the Sampling module, feeds them into five separately instantiated DDS compilers to acquire the corresponding sine and cosine values, and performs the multiplication shown in Eq. 4.

We set up the DDS compiler in its sine/cosine LUT rasterized mode such that the IP divides the unit circle into M = 50 points and creates an LUT entry for sine and cosine at each point. For maximum precision, we configure the IP for the maximum output bits of 26 for each of sine and cosine. By feeding the DDS compiler our phase values, we essentially create two signals shifted by 90 deg at ω_{IF}. This is exactly the signal needed for Eq. 4 which creates counter-rotated values at each sampled point. For non-sampled points, the rotated data is set to zero so as to not interfere with the subsequent step.

Namely, these points need to be integrated over several periods $\frac{2\pi}{\omega_{\text{IF}}}$ to obtain the time averaged value.

3.2.5 Integration

The Integration module is the last step in the demodulation process. At the start of sampling, the module waits the appropriate number of clock cycles (two) to account for latency in the Sampling module and DDS compiler before summing each index in the five sampled (or not sampled and therefore set to zero by the Multiplication module) points.

At the end of the sampling period (a “configurated" parameter), the module sums over all of the five indices for the I-value and Q-value independently to create the final I- and Q-values to pass onto the Data Analysis modules. The Integration module also asserts a bit, *iq_ valid*, when the new values are updated.

3.3 Goals and Checklist

- Baseline
  - Set-up qubit signals ✓
  - Sample and output digitized signal values ✓

  - Parameter configuration module ✓
  - Timing module ✓
- Expected
  - Phasor rotation / IP (DDS compilier) configuration ✓
  - Integration / Averaging ✓
- Stretch
  - Full Labber integration
    - Receive and set configuration parameters ✓
  - De-noising ✗
    - No fridge available for experiment before project due date :(

Note: The original checklist included the need for a CORDIC module. However, we realized that since we only sample at set phase values determined by the 500 MHz ADC sampling rate, we can greatly reduce latency (from ≈40 to 1) in the Multiplication module by using the sine/cosine LUT included in the DDS compiler IP.

4 Data Analysis (Francisca Vasconcelos)

4.1 System Function

The second portion of this project focuses on analysis of the processed I-Q values. Over an experiment of several thousand individual measurements, we receive a phasor on the IQ plane and it is up to the experimenter to determine how they would like the data processed. We provide three different data post-processing options: **Data Dump Mode**, **Linear Classification Mode**, and **2D Histogram Mode**.

The **Data Dump Mode** is the default configuration, in which the I-Q values stream out, unaltered, as they come in from the Demodulation module. This is an important feature for the FPGA to be adopted in the lab measurement chain, since the user can bypass analysis and collect raw data, if necessary.

In **Linear Classification Mode**, the I-Q values are automatically assigned to the $|0\rangle$ (ground) or $|1\rangle$ (excited) state, depending on which side of the classification line they fall (there is a third value output for points that fall exactly on the classification line). There is both a stream mode and non-stream mode, allowing the user to receive the classifications in real-time or receive counts of the total number of points in each state after all measurements are made. This speeds up the measurement process significantly by performing analysis on a rolling basis in the FPGA (which is much faster than the control computer).

Finally, in **2D Histogram Mode**, the resolution of the output is decreased by binning values into a 2-dimensional histogram. In the current implementation, the bin values are streamed out as they are calculated. However, we have also begun implementation of a Histogram BRAM module to store values and construct the histogram within the FPGA. It will output the bin counts after all the measurements have finished.

4.2 Goals and Checklist

- Baseline
  - Determine configuration parameters ✓
  - Data dump module ✓
- Expected
  - Binning ✓
  - Horizontal Classifier ✓
  - Linear Classifier ✓
  - Interface with Labber ✓
- Stretch
  - Full Labber integration (visualiztion) ✓
  - Run QST on data from experiment ✗
    - No fridge available for experiment before project due date :(

Note: A few things that were not included in the original goals and checklist were added after work began, when recognizing that they might be useful features. Among these were a streaming and no-streaming option for the classification module, as well as work on a histogram storage and construction module within the FPGA (essentially a no-stream option for the histogram as well). Furthermore some debugging tools for the operator were added, such as the ability to check the input signals by feeding them to the DAQ.

4.3 FPGA Implementation

4.4 Module Overview

- Analyze FSM Module
  - Highest module in hierarchy
  - Selects which mode: Data Dump (coded inside module), Histogram 2D, or Classify
  - Routes input into different mode modules
  - Sends output to DAQ to be read into lab computer with Verilog
- Hist2D Pt To Bin Module
  - Module for Histogram 2D Mode, which streams out histogram bin coordinate for given I-Q value
  - Also made significant progress on modules to do Histogram storage and construction within the FPGA (not completed or required)
- Bin Binary Search Module

  - Used by the Histogram 2D Pt to Bin Module to find which bin the I-Q point falls into along either the I or Q axis
  - Implemented by using binary search algorithm to reduce clock cycle counts

- Classify Master Module
  - Controls how classified state values are being output: stream or no stream
  - Stream mode is coded inside this module and outputs what state the current I-Q value is classified as
  - also feeds input to and takes output from Classify Count Module

- Classify Count Module
  - Counts the number of points that have been classified in the Ground, Excited, or Line state (until the total number of measurements has been taken), then returns all three values

4.4.1 Linear Classification

Linear classification poses a challenge in Verilog due to the requirement for floating point precision (only division by two is truly feasible in Verilog), memory issues, and timing constraints. To ensure no accuracy is lost in the classification, we propose a method using inner-products, illustrated in Figure 6. The SVM classification is a line in the I-Q plane, which can be defined by a point $\vec{p}$ on the line and a vector $\vec{a}$ orthogonal to it (oriented towards the excited state). The user must configure the FPGA with these parameters. As shown in Figure 6, the point $\vec{b}$ can be eliminated by re-centering the coordinate frame so that it becomes the origin. This corresponds to subtracting $\vec{p}$ from the received I-Q values. A vector, $\vec{b}$, connecting the I-Q point to the origin (which is now a point on the line) is then defined, and the inner product, $\vec{a} \cdot \vec{b}$ computed. This is positive when the I-Q point lies in the excited state (shown as $\vec{e}$ in the figure), negative when the point lies in the ground state (shown as $\vec{g}$), and zero when the point lies exactly on the classification boundary. This can be summarized by

$$\text{sign}(\vec{a} \cdot \vec{b}) = \begin{cases} 1, & \text{if } \vec{b} \in E \\ 0, & \text{if } \vec{a} \perp \vec{b} \\ -1, & \text{if } \vec{b} \in G \end{cases}, \tag{11}$$

where E is the half-space into which $\vec{a}$ points (corresponding to the excited state) and G the complementary half-space (ground state). Hence, the classification can be implemented by simply computing $\text{sign}(\vec{a} \cdot \vec{b})$. This classification is performed for all measured values and the number of points that lie in the ground and excited states (as well as on the line, if any) are reported. Given that our module only requires two multiplications and an addition on the computational end, we can afford to use as many wires as necessary and bram in order to ensure no floating point accuracy is lost in the calculation.

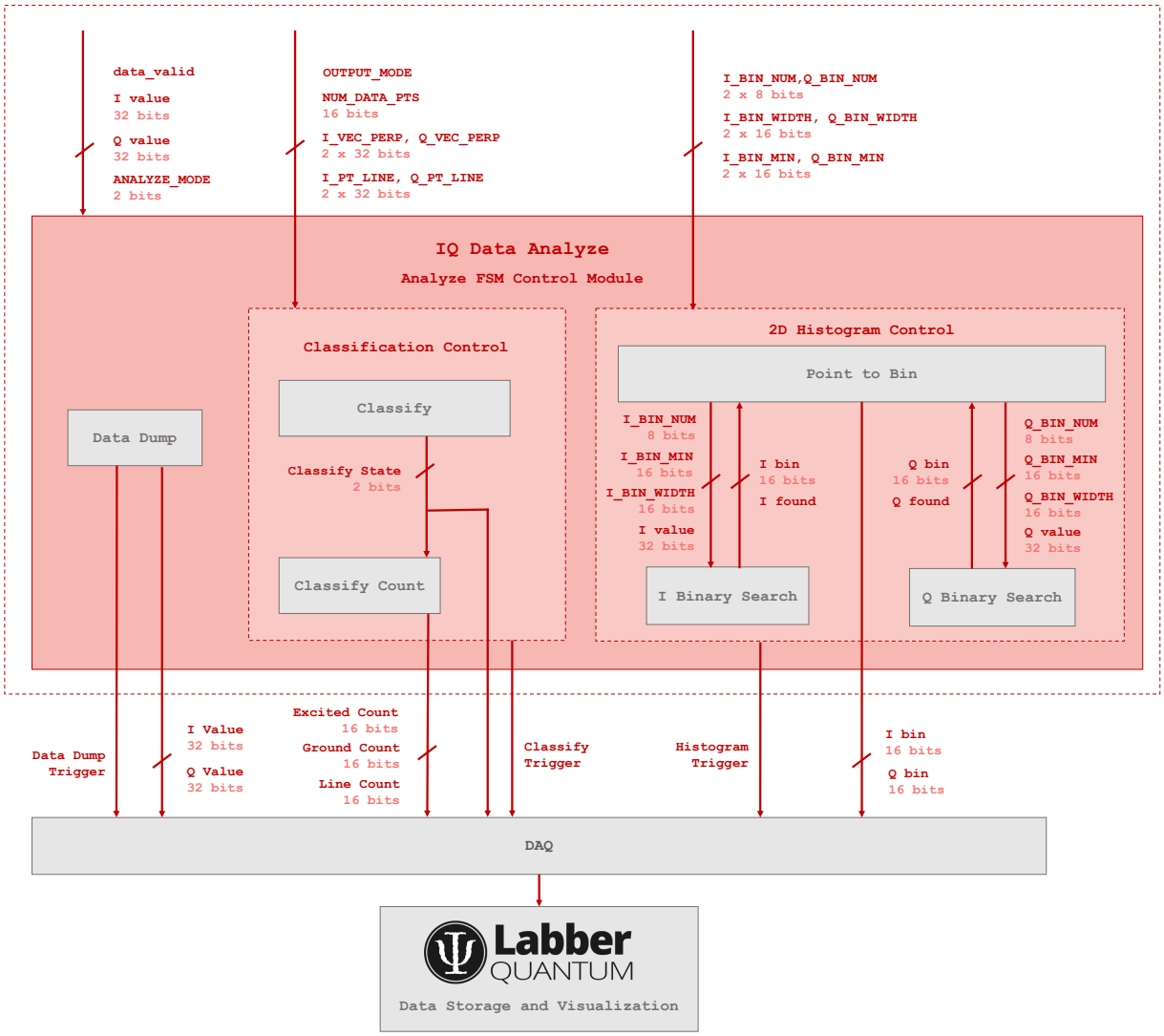

Figure 5: Block diagram of analysis modules written by Fran.

4.4.2 2D Histogram with Binary Search

For the 2D histogram, the main concern is storage, since the Block RAM of an FPGA is limited and the use of external memory would be too costly. The user inputs the minimum values of I and Q ($I_{\min}, Q_{\min}$), the number of bins per axis ($n_{\mathrm{I}}, n_{\mathrm{Q}}$), and the width of these bins ($w_{\mathrm{I}}, w_{\mathrm{Q}}$). Given an I-Q point, (i, q), the corresponding bin is found by performing a binary search, as illustrated in Figure 7, along each axis. This has a $O(\log(\max\{n_{\mathrm{I}}, n_{\mathrm{Q}}\}))$ runtime, as opposed to the $O(n_{\mathrm{I}} * n_{\mathrm{Q}})$ runtime of an exhaustive search. We use an index based method to identify the bins and for each comparison multiply the bin width by the index and add it to the min value in order to compare it to the point. This use of indices instead of storing the actual values significantly reduces the storage requirements. An example of 2D histogram is shown in Figure 8.

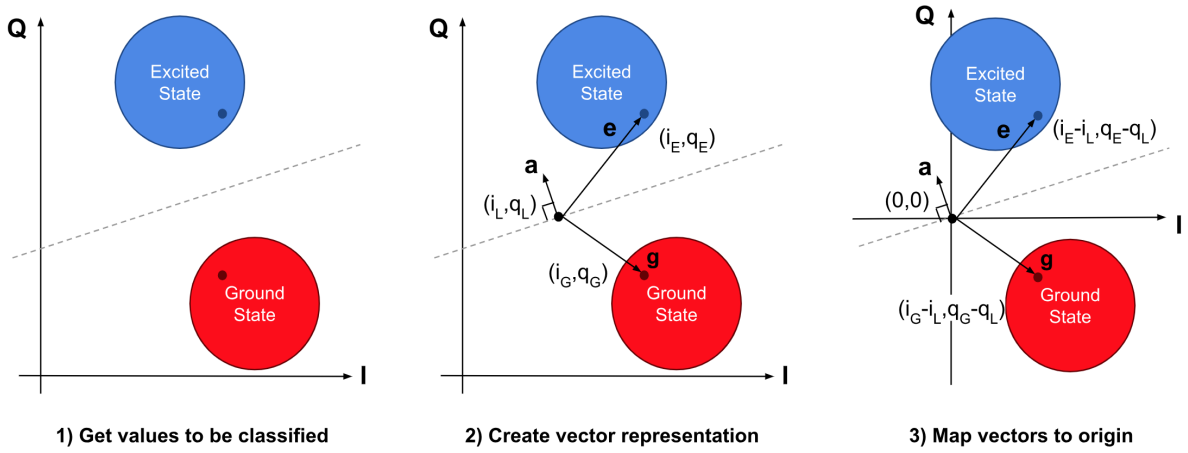

Figure 6: BlA mathematical simplification of linear classification for implementation in the FPGA.

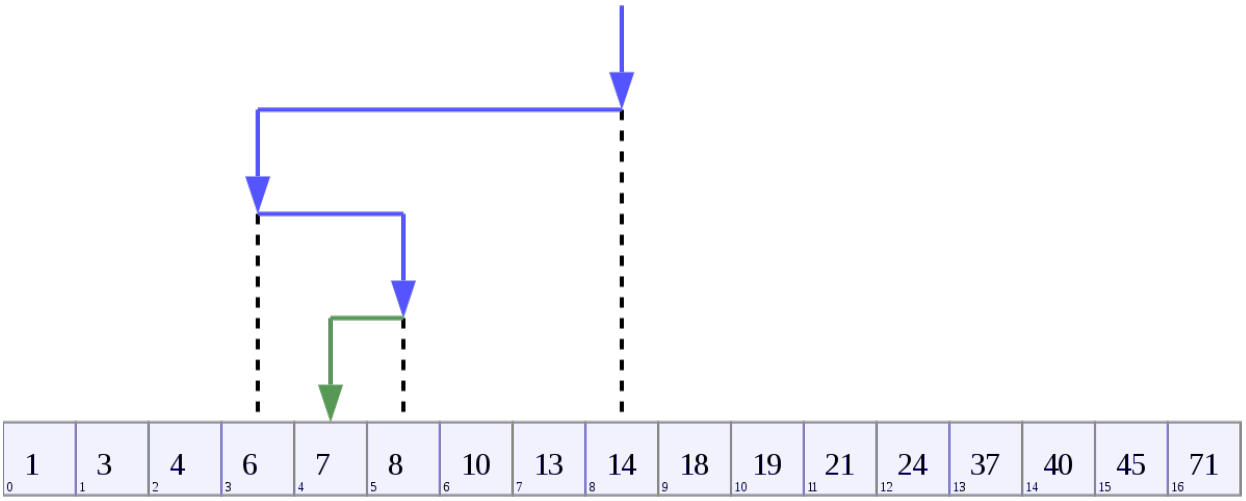

Figure 7: Example of binary search. Source: Wikipedia

4.4.3 Output

The final task in making a fully integratable system for our measurement chain was outputting our data to Labber, the lab software. In terms of hardware, we had 4 DAQs (data acquisition units) available. Each contained 5 16-bit wires for transmitting output. We formatted our output such that each time we wanted to send a data point, all the necessary information would be contained in one DAQ and a trigger would be sent. Different driver codes were then written to receive the output and, depending on the configuration specified by the user, properly load them into Labber for storage and plotting.

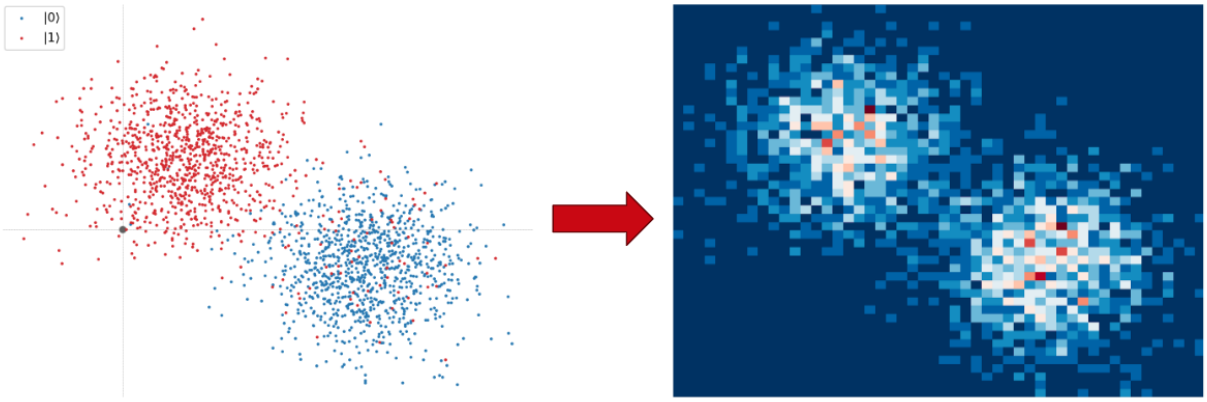

Figure 8: Example of data point conversion to 2d histogrammed values.

5 Conclusions and Outlook

We were successful in implementing most of the data analysis that we set out to do. Namely, we demonstrated signal demodulation, data dump, classification, and histogram binning. However, the is progress to be made in eliminating small sources of error (such as latency with the DDS compiler) within the FPGA itself and which interfacing and displaying data. Further, achieving on board binning (rather than streaming) will be a next goal.

The ability to work with a different FPGA system was extremely helpful in that we were able to, at the beginning, integrate with our research group's set-up. While not trivial, the process allowed us to gain a better understanding and appreciation of the difficulties of hardware-to-software interfacing, contributing to a valuable learning experience beyond just learning FPGA and Verilog.

Furthermore, the setup in our lab, involving the Keysight chasis made it impossible to debug our system by testing with LEDs or other visualization components hooked up to our FPGA, as is frequently done in the 6.111 lab. This meant we had to perform extensive testbenching and verify that our code worked perfectly in simulation before implementing it on the FPGA. Even then, we still had bugs, mainly coming from our attempt to integrate with the Labber software by writing our own driver.

A Top-Level: demod_main.v

```
////////////////////////////////////////////////////////////////////////////////
// Demod block version v2
// Author: Megan Yamoah and Francisca Vasconcelos
// Date: 11/12/2018
////////////////////////////////////////////////////////////////////////////////

module demod_main(
    // general inputs
    input clk, rst,

    // PcPort
    input [13:0] MEM_sdi_mem_S_address,
    input MEM_sdi_mem_S_rdEn, MEM_sdi_mem_S_wrEn,
    input [31:0] MEM_sdi_mem_S_wrData,
    output [31:0] MEM_sdi_mem_M_rdData,

    // hvi_port interface
    input [9:0] HVI_sdi_mem_S_address,
    input HVI_sdi_mem_S_rdEn, HVI_sdi_mem_S_wrEn,
    input [31:0] HVI_sdi_mem_S_wrData,
    output [31:0] HVI_sdi_mem_M_rdData,

    // Trigger in
    input [4:0] trigger_in,

    // Input 0
    input [15:0] data0_in_sdi_dataStreamFCx5_S_data_0,
    input [15:0] data0_in_sdi_dataStreamFCx5_S_data_1,
    input [15:0] data0_in_sdi_dataStreamFCx5_S_data_2,
    input [15:0] data0_in_sdi_dataStreamFCx5_S_data_3,
    input [15:0] data0_in_sdi_dataStreamFCx5_S_data_4,
    input data0_in_sdi_dataStreamFCx5_S_valid,

    // Input 1
    input [15:0] data1_in_sdi_dataStreamFCx5_S_data_0,
    input [15:0] data1_in_sdi_dataStreamFCx5_S_data_1,
    input [15:0] data1_in_sdi_dataStreamFCx5_S_data_2,
    input [15:0] data1_in_sdi_dataStreamFCx5_S_data_3,
    input [15:0] data1_in_sdi_dataStreamFCx5_S_data_4,
    input data1_in_sdi_dataStreamFCx5_S_valid,
```

```
// Output 0
output [15:0] data0_out_sdi_dataStreamFCx5_M_data_0,
output [15:0] data0_out_sdi_dataStreamFCx5_M_data_1,
output [15:0] data0_out_sdi_dataStreamFCx5_M_data_2,
output [15:0] data0_out_sdi_dataStreamFCx5_M_data_3,
output [15:0] data0_out_sdi_dataStreamFCx5_M_data_4,
output data0_out_sdi_dataStreamFCx5_M_valid,

// Output 1
output [15:0] data1_out_sdi_dataStreamFCx5_M_data_0,
output [15:0] data1_out_sdi_dataStreamFCx5_M_data_1,
output [15:0] data1_out_sdi_dataStreamFCx5_M_data_2,
output [15:0] data1_out_sdi_dataStreamFCx5_M_data_3,
output [15:0] data1_out_sdi_dataStreamFCx5_M_data_4,
output data1_out_sdi_dataStreamFCx5_M_valid,

// Output 2
output [15:0] data2_out_sdi_dataStreamFCx5_M_data_0,
output [15:0] data2_out_sdi_dataStreamFCx5_M_data_1,
output [15:0] data2_out_sdi_dataStreamFCx5_M_data_2,
output [15:0] data2_out_sdi_dataStreamFCx5_M_data_3,
output [15:0] data2_out_sdi_dataStreamFCx5_M_data_4,
output data2_out_sdi_dataStreamFCx5_M_valid,

// Output 3
output [15:0] data3_out_sdi_dataStreamFCx5_M_data_0,
output [15:0] data3_out_sdi_dataStreamFCx5_M_data_1,
output [15:0] data3_out_sdi_dataStreamFCx5_M_data_2,
output [15:0] data3_out_sdi_dataStreamFCx5_M_data_3,
output [15:0] data3_out_sdi_dataStreamFCx5_M_data_4,
output data3_out_sdi_dataStreamFCx5_M_valid,

// Trigger out
output [4:0] trigger3_out,
output [4:0] trigger2_out

);

wire iq_valid;
wire signed [31:0] i_val;
wire signed [31:0] q_val;
wire [1:0] analyze_mode;
```

```
wire [15:0] num_data_pts;
wire [15:0] i_bin_width, q_bin_width;
wire [7:0] i_bin_num, q_bin_num;
wire signed [15:0] i_bin_min, q_bin_min;
wire signed [31:0] i_vec_perp, q_vec_perp;
wire signed [31:0] i_pt_line, q_pt_line;
wire output_mode;

assign trigger2_out = {4'b0, iq_valid};

// output Megan's I-, Q- data
// make sure top modules are working
assign data2_out_sdi_dataStreamFCx5_M_data_0 = 16'b0;
assign data2_out_sdi_dataStreamFCx5_M_data_1 = q_val[15:0];
assign data2_out_sdi_dataStreamFCx5_M_data_2 = q_val[31:16];
assign data2_out_sdi_dataStreamFCx5_M_data_3 = i_val[15:0];
assign data2_out_sdi_dataStreamFCx5_M_data_4 = i_val[31:16];
assign data2_out_sdi_dataStreamFCx5_M_valid = 1;

// set read data to zero; we'll never read config data from the FPGA
assign MEM_sdi_mem_M_rdData = 32'b0;

// output our processed data
// this is what we're interested in!
wire [79:0] analyze_fsm_output;
assign data3_out_sdi_dataStreamFCx5_M_data_4 = analyze_fsm_output[79:64];
assign data3_out_sdi_dataStreamFCx5_M_data_3 = analyze_fsm_output[63:48];
assign data3_out_sdi_dataStreamFCx5_M_data_2 = analyze_fsm_output[47:32];
assign data3_out_sdi_dataStreamFCx5_M_data_1 = analyze_fsm_output[31:16];
assign data3_out_sdi_dataStreamFCx5_M_data_0 = analyze_fsm_output[15:0];
assign data3_out_sdi_dataStreamFCx5_M_valid = 1;


wire [129:0] sin_theta;
wire [129:0] cos_theta;

// output DDS signal to ensure DDS working
assign data0_out_sdi_dataStreamFCx5_M_valid = 1;
assign data1_out_sdi_dataStreamFCx5_M_valid = 1;
assign data0_out_sdi_dataStreamFCx5_M_data_0 = sin_theta[25:10];
assign data0_out_sdi_dataStreamFCx5_M_data_1 = sin_theta[51:36];
assign data0_out_sdi_dataStreamFCx5_M_data_2 = sin_theta[77:62];
```

```
assign data0_out_sdi_dataStreamFCx5_M_data_3 = sin_theta[103:88];
assign data0_out_sdi_dataStreamFCx5_M_data_4 = sin_theta[129:114];

assign data1_out_sdi_dataStreamFCx5_M_data_0 = cos_theta[25:10];
assign data1_out_sdi_dataStreamFCx5_M_data_1 = cos_theta[51:36];
assign data1_out_sdi_dataStreamFCx5_M_data_2 = cos_theta[77:62];
assign data1_out_sdi_dataStreamFCx5_M_data_3 = cos_theta[103:88];
assign data1_out_sdi_dataStreamFCx5_M_data_4 = cos_theta[129:114];

top_main tm(
    // inputs
    .clk100(clk), .reset(rst),
    // config control
    .MEM_sdi_mem_S_address(MEM_sdi_mem_S_address),
    .MEM_sdi_mem_S_wrEn(MEM_sdi_mem_S_wrEn),
    .MEM_sdi_mem_S_wrData(MEM_sdi_mem_S_wrData),
    // I input values
    .data0_in_0(data0_in_sdi_dataStreamFCx5_S_data_0),
    .data0_in_1(data0_in_sdi_dataStreamFCx5_S_data_1),
    .data0_in_2(data0_in_sdi_dataStreamFCx5_S_data_2),
    .data0_in_3(data0_in_sdi_dataStreamFCx5_S_data_3),
    .data0_in_4(data0_in_sdi_dataStreamFCx5_S_data_4),
    // Q input values
    .data1_in_0(data1_in_sdi_dataStreamFCx5_S_data_0),
    .data1_in_1(data1_in_sdi_dataStreamFCx5_S_data_1),
    .data1_in_2(data1_in_sdi_dataStreamFCx5_S_data_2),
    .data1_in_3(data1_in_sdi_dataStreamFCx5_S_data_3),
    .data1_in_4(data1_in_sdi_dataStreamFCx5_S_data_4),
    .trigger(trigger_in[0]),

    //outputs
    .iq_valid(iq_valid),
    .i_val(i_val), .q_val(q_val),
    // configurated prameters to pass to lower modules
    .analyze_mode(analyze_mode),
    .num_data_pts(num_data_pts),
    .i_bin_width(i_bin_width), .q_bin_width(q_bin_width),
    .i_bin_num(i_bin_num), .q_bin_num(q_bin_num),
    .i_bin_min(i_bin_min), .q_bin_min(q_bin_min),
    .i_vec_perp(i_vec_perp), .q_vec_perp(q_vec_perp),
    .i_pt_line(i_pt_line), .q_pt_line(q_pt_line),
    .output_mode(output_mode),
```

```
        .sin_theta(sin_theta), .cos_theta(cos_theta));

    analyze_fsm analyze_module(

        .clk100(clk), .system_reset(rst),

        //config params
        .analyze_mode(analyze_mode), // fsm state
        .num_data_pts(num_data_pts), // total number of points
        .output_mode(output_mode), // stream or no stream?

        // i-q data parameters
        .data_in(iq_valid),
        .i_val(i_val), .q_val(q_val),

        // histogram inputs
        .i_bin_num(i_bin_num), .q_bin_num(q_bin_num), // number of bins on each
        ↪  axis
        .i_bin_width(i_bin_width), .q_bin_width(q_bin_width), // bin width on each
        ↪  axis
        .i_min(i_bin_min), .q_min(q_bin_min), // origin pt of 0,0 bin

        // classification inputs
        .i_vec_perp(i_vec_perp), .q_vec_perp(q_vec_perp),
        .i_pt_line(i_pt_line), .q_pt_line(q_pt_line),

        // output data
        .data_output_trigger(trigger3_out),
        .output_channels(analyze_fsm_output));

endmodule // demod_top
```

A.1 Megan's Top-Level: top_main.sv

```
////////////////////////////////////////////////////////////////////////////////
// Top Modules Instantiation version v1
// Author: Megan Yamoah
// Date: 11/12/2018
////////////////////////////////////////////////////////////////////////////////

module top_main (
        input clk100, reset,
        // config control
```

```
    input [13:0] MEM_sdi_mem_S_address, // parameter location
input MEM_sdi_mem_S_wrEn, // write on
input [31:0] MEM_sdi_mem_S_wrData, // parameter value
// i input
    input [15:0] data0_in_0, data0_in_1, data0_in_2, data0_in_3, data0_in_4,
    // q input
    input [15:0] data1_in_0, data1_in_1, data1_in_2, data1_in_3, data1_in_4,
    input trigger,
    //outputs
    output iq_valid,
    output [31:0] i_val, q_val,
    // configurated prameters to pass to lower modules
    output [1:0] analyze_mode,
    output [15:0] num_data_pts,
output [15:0] i_bin_width, q_bin_width,
output [7:0] i_bin_num, q_bin_num,
output signed [15:0] i_bin_min, q_bin_min,
output signed [31:0] i_vec_perp, q_vec_perp,
output signed [31:0] i_pt_line, q_pt_line,
output output_mode,
output signed [4:0] [25:0] sin_theta,
output signed [4:0] [25:0] cos_theta);

// configurated values
wire [4:0] demod_freq;
wire [10:0] sample_length;
wire [5:0] sample_freq;
wire [13:0] delay_time;
wire [9:0] [4:0] [5:0] demod_mod50_LUT;

// instantiate parameter configuration
    config_params config_main(.clk100(clk100), .reset(reset),
            // config protocol
            .MEM_sdi_mem_S_address(MEM_sdi_mem_S_address),
    .MEM_sdi_mem_S_wrEn(MEM_sdi_mem_S_wrEn),
    .MEM_sdi_mem_S_wrData(MEM_sdi_mem_S_wrData),
            // parameter
            .demod_freq(demod_freq), .num_data_pts(num_data_pts),
            .demod_mod50_LUT(demod_mod50_LUT),
            .sample_length(sample_length), .sample_freq(sample_freq),
            .delay_time(delay_time),
            .analyze_mode(analyze_mode),
```

```
            .i_bin_width(i_bin_width), .q_bin_width(q_bin_width),
            .i_bin_num(i_bin_num), .q_bin_num(q_bin_num),
            .i_bin_min(i_bin_min), .q_bin_min(q_bin_min),
            .i_vec_perp(i_vec_perp), .q_vec_perp(q_vec_perp),
        .i_pt_line(i_pt_line), .q_pt_line(q_pt_line),
        .output_mode(output_mode));

        // start data collection
        // output from timing module
        wire start_collect;

        timing timing_main(
                // inputs
                .clk100(clk100), .reset(reset), .trigger(trigger),
                .delay(delay_time),
                // ouputs
                .start_collect(start_collect));

        // set up data arrays
        wire signed [4:0] [15:0] data_i_in;
        wire signed [4:0] [15:0] data_q_in;

    // shifted arrays are fed into multiplier
        wire signed [4:0] [15:0] data_i_shift;
        wire signed [4:0] [15:0] data_q_shift;

        // assign data arrays
        assign data_i_in = {data0_in_4, data0_in_3, data0_in_2, data0_in_1,
        ↪  data0_in_0};
        assign data_q_in = {data1_in_4, data1_in_3, data1_in_2, data1_in_1,
        ↪  data1_in_0};

        wire [4:0] [7:0] phase_vals; // create phase value array

        // instantiate sampler
        sampler sampler_main(
                // inputs
                .clk100(clk100), .reset(reset), .start(start_collect),
            .data_i_in(data_i_in), .data_q_in(data_q_in),
            .demod_freq(demod_freq), .demod_mod50_LUT(demod_mod50_LUT),
            .sample_length(sample_length), .sample_skip(sample_freq),
            // outputs
```

```
            .data_i_shift(data_i_shift), .data_q_shift(data_q_shift),
            .phase_vals(phase_vals));
    // shifted arrays ensure matching between phase_vals and I-Q data

        // set up rotated data arrays
        wire signed [4:0] [59:0] data_i_rot;
        wire signed [4:0] [59:0] data_q_rot;

        // instantiate multiplier
        multiplier multiplier_main(
                // inputs
                .clk100(clk100), .reset(reset),
                .phase_vals(phase_vals),
                .data_i_in(data_i_shift), .data_q_in(data_q_shift), // shifted
                ↪  values
                // outputs
                .data_i_rot(data_i_rot), .data_q_rot(data_q_rot),
                .sin_theta(sin_theta), .cos_theta(cos_theta)); // counter rotated
                ↪  outputs

        // instantiate integrator
        integrator integrator_main(
                // inputs
                .clk100(clk100), .reset(reset), .start(start_collect),
                .sample_length(sample_length),
                .data_i_rot(data_i_rot), .data_q_rot(data_q_rot),
                // outputs
                .iq_valid(iq_valid),
                .i_val_tot(i_val), .q_val_tot(q_val));

endmodule // top_main
```

A.1.1 Megan's Modules: top_modules.sv

```
//////////////////////////////////////////////////////////////////////////////////
// Top Modules version v1
// Author: Megan Yamoah
// Date: 10/12/2018
//////////////////////////////////////////////////////////////////////////////////

module config_params(
    // inputs
    input clk100,
```

```
input reset,
// control inputs
input [13:0] MEM_sdi_mem_S_address, // parameter location
input MEM_sdi_mem_S_wrEn, // write enable
input [31:0] MEM_sdi_mem_S_wrData, // parameter value

// outputs
output reg [4:0] demod_freq, // freq divided by 10 MHz; we assume will only
↪  sample at increments of 10 MHz
output reg [15:0] num_data_pts, // total number of measurements
output reg [9:0] [4:0] [5:0] demod_mod50_LUT, // LUT containing values for mod
↪  50 updated for each value of demod_freq
output reg [10:0] sample_length, // max 20 us
output reg [5:0] sample_freq, // min 8 MHz
output reg [13:0] delay_time, // max 163 us
output reg [1:0] analyze_mode, // data dump, binning, classifier
output reg [15:0] i_bin_width, q_bin_width, // width of bin in i, q_direction
output reg [7:0] i_bin_num, q_bin_num, // number of bins along i, q_direction
output reg signed [15:0] i_bin_min, q_bin_min, // start of bins i, q_direction
output reg signed [31:0] i_vec_perp, q_vec_perp, // perpendicular classifier
↪  lines
output reg signed [31:0] i_pt_line, q_pt_line,
output reg output_mode);

always @(posedge clk100) begin
    if (reset) begin // reset to default
        demod_freq <= 5'd1; // 50 MHz
        demod_mod50_LUT <= {{6'd49, 6'd48, 6'd47, 6'd46, 6'd45},
                            {6'd44, 6'd43, 6'd42, 6'd41, 6'd40},
                            {6'd39, 6'd38, 6'd37, 6'd36, 6'd35},
                            {6'd34, 6'd33, 6'd32, 6'd31, 6'd30},
                            {6'd29, 6'd28, 6'd27, 6'd26, 6'd25},
                            {6'd24, 6'd23, 6'd22, 6'd21, 6'd20},
                            {6'd19, 6'd18, 6'd17, 6'd16, 6'd15},
                            {6'd14, 6'd13, 6'd12, 6'd11, 6'd10},
                            {6'd09, 6'd08, 6'd07, 6'd06, 6'd05},
                            {6'd04, 6'd03, 6'd02, 6'd01, 6'd00}};
        sample_length <= 11'd20000;
        sample_freq <= 6'd1; // 100 MHz
        delay_time <= 14'd5; // 50 us
        analyze_mode <= 2'd01;
        i_bin_width <= 16'd100;
```

```
        q_bin_width <= 16'd100;
        i_bin_num <= 8'd10;
        q_bin_num <= 8'd10;
        i_bin_min <= 16'd0;
        q_bin_min <= 16'd0;
        i_vec_perp <= 32'd1;
        q_vec_perp <= 32'd1;
        i_pt_line <= 32'd0;
        q_pt_line <= 32'd1;
        output_mode <= 1'd0;
        num_data_pts <= 16'd10;
    end
    else if (MEM_sdi_mem_S_wrEn) begin // reconfigure values
        // search for correct address
        // reconfigure general values
        if (MEM_sdi_mem_S_address == 14'd0)
            demod_freq <= MEM_sdi_mem_S_wrData[4:0];
        else if (MEM_sdi_mem_S_address == 14'd1)
            num_data_pts <= MEM_sdi_mem_S_wrData[15:0];
        else if (MEM_sdi_mem_S_address == 14'd2)
            output_mode <= MEM_sdi_mem_S_wrData[0];

        // reconfigure LUT values
        else if (MEM_sdi_mem_S_address == 14'd10)
            demod_mod50_LUT[0] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd11)
            demod_mod50_LUT[1] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd12)
            demod_mod50_LUT[2] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd13)
            demod_mod50_LUT[3] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd14)
            demod_mod50_LUT[4] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd15)
            demod_mod50_LUT[5] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd16)
            demod_mod50_LUT[6] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd17)
            demod_mod50_LUT[7] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd18)
            demod_mod50_LUT[8] <= MEM_sdi_mem_S_wrData[29:0];
        else if (MEM_sdi_mem_S_address == 14'd19)
```

```
                demod_mod50_LUT[9] <= MEM_sdi_mem_S_wrData[29:0];

            // reconfigure sampling-type vaules
            else if (MEM_sdi_mem_S_address == 14'd20)
                sample_length <= MEM_sdi_mem_S_wrData[10:0];
            else if (MEM_sdi_mem_S_address == 14'd21)
                sample_freq <= MEM_sdi_mem_S_wrData[5:0];
            else if (MEM_sdi_mem_S_address == 14'd22)
                delay_time <= MEM_sdi_mem_S_wrData[13:0];

            // reconfigure analysis-type values
            else if (MEM_sdi_mem_S_address == 14'd30)
                analyze_mode <= MEM_sdi_mem_S_wrData[1:0];
            else if (MEM_sdi_mem_S_address == 14'd31)
                i_bin_width <= MEM_sdi_mem_S_wrData[15:0];
            else if (MEM_sdi_mem_S_address == 14'd32)
                q_bin_width <= MEM_sdi_mem_S_wrData[15:0];
            else if (MEM_sdi_mem_S_address == 14'd33)
                i_bin_num <= MEM_sdi_mem_S_wrData[7:0];
            else if (MEM_sdi_mem_S_address == 14'd34)
                q_bin_num <= MEM_sdi_mem_S_wrData[7:0];
            else if (MEM_sdi_mem_S_address == 14'd35)
                i_bin_min <= MEM_sdi_mem_S_wrData[15:0];
            else if (MEM_sdi_mem_S_address == 14'd36)
                q_bin_min <= MEM_sdi_mem_S_wrData[15:0];
            else if (MEM_sdi_mem_S_address == 14'd37)
                i_vec_perp <= MEM_sdi_mem_S_wrData[31:0];
            else if (MEM_sdi_mem_S_address == 14'd38)
                q_vec_perp <= MEM_sdi_mem_S_wrData[31:0];
            else if (MEM_sdi_mem_S_address == 14'd39)
                i_pt_line <= MEM_sdi_mem_S_wrData[31:0];
            else if (MEM_sdi_mem_S_address == 14'd40)
                q_pt_line <= MEM_sdi_mem_S_wrData[31:0];
        end
    end

endmodule // config

// timing module
// this module sets how long to wait after the trigger to begin sampling data
// often, data that arrives with the trigger is not yet clean
module timing(
```

```
input clk100,
input reset,
input trigger,
input [13:0] delay,
output reg start_collect);

parameter IDLE = 0;
parameter DELAY = 1;

reg state = IDLE;
reg [13:0] counter = 0;

always @(posedge clk100) begin
    if (reset) begin // at reset, reset our count and wait for trigger
        state <= IDLE;
        counter <= 14'b0;
        start_collect <= 0;
    end

    case (state)
        IDLE:
        begin
            start_collect <= 0;
            if (trigger) begin // once triggered, begin count
                state <= DELAY;
                counter <= counter + 1;
            end
        end

        DELAY:
        begin
            if (counter < delay)
                counter <= counter + 1;
            else if (~reset && counter == delay) begin // only assert
             ↪  start_collect if reset not assert on same clock cycle
                counter <= 0;
                start_collect <= 1; // asserted on same clock cycle of first
                 ↪  value to sample
                state <= IDLE;
            end
        end
```

```
            default:
            begin
                state <= IDLE;
                counter <= 14'd0;
                start_collect <= 0;
            end

        endcase
    end

endmodule // timing


// sampler module
// decide which values to sample each clock cycle
// does this by setting relevant phase values to non-zero values
// samples to ignore we set corresponding phase values to zero
module sampler(
    // inputs
    input clk100,
    input reset,
    input start,
    input signed [4:0] [15:0] data_i_in, // packed
    input signed [4:0] [15:0] data_q_in, // packed
    input [4:0] demod_freq,
    input [9:0] [4:0] [5:0] demod_mod50_LUT,
    input [10:0] sample_length,
    input [5:0] sample_skip, // equal to sample_freq
    // outputs
    output reg signed [4:0] [15:0] data_i_shift, // I and Q values shifted one
    ↪  clock cylce to match phase_vals
    output reg signed [4:0] [15:0] data_q_shift,
    output reg [4:0] [5:0] phase_vals); // packed

    // state parameter
    parameter IDLE = 0;
    parameter SAMPLE = 1;

    reg state = IDLE;
    reg [10:0] counter = 0;
    reg [3:0] count_mult = 0;
    reg [5:0] skip_hold; // stores value which determines offset of each count
    ↪  based on previous cycle
```

```
// for sample_skips less than five
// maps sample_skip and skip_hold at time t to to skip_hold at time t+1
wire [3:0] [3:0] [1:0] skip_hold_LUT;
assign skip_hold_LUT = {{2'd2, 2'd1, 2'd0, 2'd3},
                        {2'd1, 2'd0, 2'd2, 2'd1},
                        {2'd0, 2'd1, 2'd0, 2'd1},
                        {2'd0, 2'd0, 2'd0, 2'd0}};

// for loop interation variable
integer i, j;

always @(posedge clk100) begin
    if (reset) begin // at reset, move to IDLE state and wait for new start
    ↪  signal
        state <= IDLE;
        counter <= 0;
        phase_vals <= {5{8'b0}};
    end

    // make sure data values match phase_vals
    data_i_shift <= data_i_in;
    data_q_shift <= data_q_in;

    case (state)
        IDLE: begin
            if (start) begin // from timing, assert on same clock cycle as
            ↪  value to sample
                state <= SAMPLE; // change states to sampling
                counter <= 1;
                count_mult <= 1;
                // if sample_skip < 5, we need to assign phase values on this
                ↪  cycle
                if (sample_skip < 5) begin
                    // which phase values we sample depends on value of
                    ↪  sample_skip
                    if (sample_skip == 1) begin // every time step
                        phase_vals[0] <= 0; // phase_vals set at next clock
                        ↪  cycle
                        phase_vals[1] <= demod_mod50_LUT[0][1];
                        phase_vals[2] <= demod_mod50_LUT[0][2];
                        phase_vals[3] <= demod_mod50_LUT[0][3];
```

```
                phase_vals[4] <= demod_mod50_LUT[0][4];
                skip_hold <= 0;
            end
            else if (sample_skip == 2) begin // every other
                phase_vals[0] <= 0; // the first value is always
                ↪  sampled, but the phase = 0
                phase_vals[1] <= 0;
                phase_vals[2] <= demod_mod50_LUT[0][2];
                phase_vals[3] <= 0;
                phase_vals[4] <= demod_mod50_LUT[0][4];
                skip_hold <= 1;
            end
            else if (sample_skip == 3) begin // every third
                phase_vals[0] <= 0; // the first value is always
                ↪  sampled, but the phase = 0
                phase_vals[1] <= 0;
                phase_vals[2] <= 0;
                phase_vals[3] <= demod_mod50_LUT[0][3];
                phase_vals[4] <= 0;
                skip_hold <= 1;
            end
            else if (sample_skip == 4) begin // every fourth
                phase_vals[0] <= 0; // the first value is always
                ↪  sampled, but the phase = 0
                phase_vals[1] <= 0;
                phase_vals[2] <= 0;
                phase_vals[3] <= 0;
                phase_vals[4] <= demod_mod50_LUT[0][4];
                skip_hold <= 3;
            end
        end // if (sample_skip < 5)
        else begin // if sample_skip > 5, we don't sample anything
        ↪  except the first value this cycle
            phase_vals[0] <= 0;
            phase_vals[1] <= 0;
            phase_vals[2] <= 0;
            phase_vals[3] <= 0;
            phase_vals[4] <= 0;
            skip_hold <= sample_skip - 5;
        end
    end // if (start)
    else // IDLE && ~start
```

```
        phase_vals <= {5{8'b0}};
end

SAMPLE: begin
    if (counter < sample_length) begin // within sampling range
        counter <= counter + 1;
        if (count_mult < 9)
            count_mult <= count_mult + 1;
        else
            count_mult <= 0;

        if (sample_skip < 5) begin // when sample_skip < 5, skip_hold <
        ↪  4
            // since each cycle of five may have more than one sampled
            ↪  value, we need if statements for each value in our
            ↪  array
            // test each position to assign value
            // set to zero otherwise
            for (i = 0; i < 5; i = i + 1) begin
                if ((skip_hold == i) ||
                   ((skip_hold + sample_skip) == i) || ((skip_hold + 2
                   ↪  * sample_skip) == i) ||
                   ((skip_hold + 3 * sample_skip) == i) || ((skip_hold
                   ↪  + 4 * sample_skip) == i))
                    phase_vals[i] <= demod_mod50_LUT[count_mult][i];
                else
                    phase_vals[i] <= 0;
            end
            skip_hold <= skip_hold_LUT[sample_skip-1][skip_hold]; //
            ↪  set skip_hold for next cycle based on current
            ↪  skip_hold and sample_skip
        end

        else if ((skip_hold < 5) && (sample_skip >= 5)) begin // if
        ↪  skip_hold < 5, we will sample a value this cycle
            // need to determine which value
            // all other set to zero
            for (j = 0; j < 5; j = j + 1) begin
                phase_vals[j] <= (skip_hold == j) ?
                ↪  demod_mod50_LUT[count_mult][j] : 0;
            end
            skip_hold <= sample_skip - (5 - skip_hold); // redefine
            ↪  skip_hold
```

```
                    end

                    else begin
                        skip_hold <= skip_hold - 5; // if we don't sample this
                        ↪  cycle, we decrement skip_hold and check next cycle
                        phase_vals <= {5{8'b0}};
                    end
                end

                else begin // stop sampling, set to IDLE
                    counter <= 0;
                    state <= IDLE;
                    phase_vals <= {5{8'b0}};
                end
            end
        endcase
    end

endmodule // sampler


// multiplier module
// instantiate DDS compiler and multiply relevant values
// uses phase data from sampler module
module multiplier(
    input clk100,
    input reset,
    input [4:0] [5:0] phase_vals, // packed
    input signed [4:0] [15:0] data_i_in,
    input signed [4:0] [15:0] data_q_in,
    // output
    // rotation data values (multiplied by sine and cosine)
    // outputs always active, only non-zero for sampled values
    output reg signed [4:0] [56:0] data_i_rot,
    output reg signed [4:0] [56:0] data_q_rot);

    // create outputs for DDS compiler
    wire signed [4:0] [25:0] sin_theta;
    wire signed [4:0] [25:0] cos_theta;
    wire [4:0] [63:0] sin_cos;

    // set up our concatenated sin_cos bus
```

```
genvar g;
generate
    for (g = 0; g < 5; g = g + 1) begin
        assign sin_theta[g] = sin_cos[g][57:32];
        assign cos_theta[g] = sin_cos[g][25:0];
    end
endgenerate

// initiation values for DDS
wire phase_valid = 1;
wire error0;
wire error1;
wire error2;
wire error3;
wire error4;
wire data_valid0;
wire data_valid1;
wire data_valid2;
wire data_valid3;
wire data_valid4;

// DDS COMPILERS
// mode: sine/cosine LUT
// modulus: 50
// input phase: 8 bit
// output {sine, cosine}: 32 bit
// latency: 1
dds_compiler_0 sincos0(.aclk(clk100), .s_axis_phase_tvalid(phase_valid),
    .s_axis_phase_tdata(phase_vals[0]),
    .m_axis_data_tvalid(data_valid0), .m_axis_data_tdata(sin_cos[0]),
    .event_phase_in_invalid(error0));
dds_compiler_0 sincos1(.aclk(clk100), .s_axis_phase_tvalid(phase_valid),
    .s_axis_phase_tdata(phase_vals[1]),
    .m_axis_data_tvalid(data_valid1), .m_axis_data_tdata(sin_cos[1]),
    .event_phase_in_invalid(error1));
dds_compiler_0 sincos2(.aclk(clk100), .s_axis_phase_tvalid(phase_valid),
    .s_axis_phase_tdata(phase_vals[2]),
    .m_axis_data_tvalid(data_valid2), .m_axis_data_tdata(sin_cos[2]),
    .event_phase_in_invalid(error2));
dds_compiler_0 sincos3(.aclk(clk100), .s_axis_phase_tvalid(phase_valid),
    .s_axis_phase_tdata(phase_vals[3]),
    .m_axis_data_tvalid(data_valid3), .m_axis_data_tdata(sin_cos[3]),
```

```
        .event_phase_in_invalid(error3));
    dds_compiler_0 sincos4(.aclk(clk100), .s_axis_phase_tvalid(phase_valid),
        .s_axis_phase_tdata(phase_vals[4]),
        .m_axis_data_tvalid(data_valid4), .m_axis_data_tdata(sin_cos[4]),
        .event_phase_in_invalid(error4));

    // create registers to hold our I and Q data in order to match with DDS output
    reg signed [4:0] [15:0] data_i_hold;
    reg signed [4:0] [15:0] data_q_hold;
    reg signed [4:0] [5:0] phase_vals_hold;

    integer i;

    always @(posedge clk100) begin
        if (reset) begin
            data_i_rot <= {5{32'b0}};
            data_q_rot <= {5{32'b0}};
        end
        else begin
            // perform multiplication here
            for (i = 0; i < 5; i = i + 1) begin
                if (~(phase_vals[i] == 0)) begin
                    data_i_rot[i] <= data_i_hold[i]*cos_theta[i] +
                    ↪  data_q_hold[i]*sin_theta[i];
                    data_q_rot[i] <= data_q_hold[i]*cos_theta[i] -
                    ↪  data_i_hold[i]*sin_theta[i];
                end
                else begin
                    data_i_rot[i] <= 0;
                    data_q_rot[i] <= 0;
                end
            end
        end

        // clock our I and Q data
        data_i_hold <= data_i_in;
        data_q_hold <= data_q_in;
        phase_vals_hold <= phase_vals;
    end

endmodule // multiplier
```

```
// integrator module
// integrate all of our sampled values to average
module integrator(
    input clk100,
    input reset,
    input start,
    input [10:0] sample_length,
    input signed [4:0] [56:0] data_i_rot, // latency: 2
    input signed [4:0] [56:0] data_q_rot,
    output reg iq_valid, // asserted HIGH on the same clock cycle as when
    ↪  integrated I and Q values are valid
    output signed [31:0] i_val_tot,
    output signed [31:0] q_val_tot);

    // state parameters
    parameter IDLE = 0;
    parameter DELAY = 1; // require an extra state to account for latency of
    ↪  previous modules of 2
    parameter INTEGRATE = 2;

    reg [1:0] state = IDLE;
    reg [10:0] counter = 0;

    // store added I and Q values
    reg [4:0] [56:0] i_vals = 0;
    reg [4:0] [56:0] q_vals = 0;

    reg [56:0] i_val_sum = 0;
    reg [56:0] q_val_sum = 0;

    assign i_val_tot = i_val_sum[56:25];
    assign q_val_tot = q_val_sum[56:25];

    integer i;

    always @(posedge clk100) begin
        if (reset) begin // at reset, stop integrating and reset sums to zero
            state <= IDLE;
            counter <= 11'b0;
            i_vals <= {5{32'b0}};
            q_vals <= {5{32'b0}};
```

```
end

case (state)
    IDLE: begin
        iq_valid <= 0;
        if (start) begin // at start change states
            state <= DELAY;
            i_vals <= {5{32'b0}};
            q_vals <= {5{32'b0}};
        end
    end

    DELAY: begin // account of latency from previous modules of 2
        state <= INTEGRATE;
    end

    INTEGRATE: begin
        counter <= counter + 1;
        if (counter < sample_length) begin // when sampling, sum over each
        ↪  index
            for (i = 0; i < 5; i = i + 1) begin
                i_vals[i] <= i_vals[i] + data_i_rot[i];
                q_vals[i] <= q_vals[i] + data_q_rot[i];
            end
        end
        else if (counter == sample_length) begin
            state <= IDLE;
            counter <= 0;
            // at end of sample length, sum over indices
            // I and Q total values valid until end of next sampling
            ↪  period
            i_val_sum <= i_vals[0] + i_vals[1] + i_vals[2] + i_vals[3] +
            ↪  i_vals[4];
            q_val_sum <= q_vals[0] + q_vals[1] + q_vals[2] + q_vals[3] +
            ↪  q_vals[4];
            iq_valid <= 1;
        end
    end

    default: begin
        state <= IDLE;
        counter <= 0;
```

```
                iq_valid <= 0;
            end
        endcase
    end

endmodule // iq_output
```

A.2 Fran's Top-Level: analyze_fsm.v

```
// FSM to tie it all together
module analyze_fsm(
    input clk100, system_reset,

    //config params
    input [1:0] analyze_mode, // fsm state
    input [15:0] num_data_pts, // total number of points
    input output_mode, // stream or no stream?


    // i-q data parameters
    input data_in,
    input signed [31:0] i_val, q_val,

    // histogram inputs
    input [7:0] i_bin_num, q_bin_num, // number of bins on each axis
    input [15:0] i_bin_width, q_bin_width, // bin width on each axis
    input signed [15:0] i_min, q_min, // origin pt of 0,0 bin

    // classification inputs
    input signed [31:0] i_vec_perp, q_vec_perp,
    input signed [31:0] i_pt_line, q_pt_line,

    // output data
    output reg [4:0] data_output_trigger,
    output reg [79:0] output_channels);

    // define states
    parameter DATA_DUMP_MODE = 2'b00;
    parameter CLASSIFY_MODE = 2'b01;
    parameter HIST2D_MODE = 2'b11;

    // for reading output of different modules
    wire [79:0] classify_output;
```

```
wire classify_trigger;

wire [7:0] hist_i_output;
wire [7:0] hist_q_output;
wire hist2d_trigger;

// analysis FSM
always @(posedge clk100) begin
    case(analyze_mode)

        DATA_DUMP_MODE: begin
            if (data_in)
                output_channels <= {16'd0, i_val, q_val};
            data_output_trigger <= {4'd0, data_in};
        end

        CLASSIFY_MODE: begin
            output_channels <= classify_output;
            data_output_trigger <= {4'd0, classify_trigger};
        end

        HIST2D_MODE: begin
            output_channels <= {48'd0,8'd0,hist_i_output,8'd0,hist_q_output};
            data_output_trigger <= {4'd0, hist2d_trigger};
        end

        default: output_channels <= 64'b0;

    endcase
end

// linear classification control module
classify_master lin_class(.clk100(clk100), .system_reset(system_reset),
↪  .num_data_pts(num_data_pts), .data_in(data_in), .i_val(i_val),
↪  .q_val(q_val), .stream_mode(output_mode), .i_pt_line(i_pt_line),
↪  .q_pt_line(q_pt_line), .i_vec_perp(i_vec_perp), .q_vec_perp(q_vec_perp),
↪  .fpga_output(classify_output),  .data_output_trigger(classify_trigger));
```

```
    hist2d_pt_to_bin conv(.clk100(clk100), .system_reset(system_reset),
    ↪  .data_in(data_in), .i_val(i_val), .q_val(q_val), .i_bin_num(i_bin_num),
    ↪  .q_bin_num(q_bin_num), .i_bin_width(i_bin_width),
    ↪  .q_bin_width(q_bin_width), .i_min(i_min), .q_min(q_min),
    ↪  .i_q_found_out(hist2d_trigger), .i_bin_coord_out(hist_i_output),
    ↪  .q_bin_coord_out(hist_q_output));

endmodule // analyze_fsm
```

A.2.1 Fran's Module: classify_master.v

```
module classify_master(
        input clk100, system_reset,
        input [15:0] num_data_pts,

        input data_in, // indicates if new i,q data is coming in
        input signed [31:0] i_val, q_val, // pt to be classified

        input stream_mode, // 1 to stream classifications as they come in, 0 to
        ↪  report counts at end

        // static input
        input signed [31:0] i_pt_line, q_pt_line, // pt on classification line
        // vector from origin with slope perpendicular to line, pts in direction of
        ↪  excited state
        input signed [31:0] i_vec_perp, q_vec_perp,

        output reg data_output_trigger,
        output reg [79:0] fpga_output
    );

    // classify output
    wire [1:0] state;
    wire valid_class_pt;
    parameter GROUND_STATE = 2'b01;
    parameter EXCITED_STATE = 2'b10;
    parameter CLASSIFY_LINE = 2'b11;
    parameter ERROR = 2'b00;

    // classify_count input/output
    reg reset;
    reg [15:0] data_pt_count = 0;
    wire [15:0] excited_count;
```

```
wire [15:0] ground_count;
wire [15:0] line_count;

reg [1:0] count_mode;
parameter COUNT = 2'b00;
parameter OUTPUT_FINAL = 2'b01;
parameter RESET = 2'b10;

reg stream_state;
parameter WAIT_FOR_VAL = 1;
parameter STREAM_RESET = 0;

always @(posedge clk100) begin
    // output values as they come in
    if(system_reset) begin
        count_mode = COUNT;
        data_output_trigger = 0;
        fpga_output = 0;
        reset <= 0;
    end
    else begin
        if(stream_mode) begin
            fpga_output <= {64'b0, 14'b0, state[1:0]};

            case(stream_state)

                WAIT_FOR_VAL: begin
                    if(valid_class_pt) begin
                        data_output_trigger <= 1;
                        stream_state <= STREAM_RESET;
                    end
                end

                STREAM_RESET: begin
                    data_output_trigger <= 0;
                    stream_state <= WAIT_FOR_VAL;
                end

            endcase
        end

        // output values while updating and signal when finished
```

```
        else begin
            case(count_mode)

                COUNT: begin
                    reset <= 0;
                    data_output_trigger <= 0;
                    if(data_pt_count < num_data_pts) begin
                        fpga_output <= {16'b0, data_pt_count, excited_count,
                        ↪  ground_count, line_count};
                        reset <= 0;
                        if(valid_class_pt) data_pt_count <= data_pt_count + 1;
                    end
                    else count_mode <= OUTPUT_FINAL;
                end

                OUTPUT_FINAL: begin
                    data_output_trigger <= 1;
                    fpga_output <= {16'd0, data_pt_count, excited_count,
                    ↪  ground_count, line_count};
                    count_mode <= RESET;
                end

                RESET: begin
                    reset <= 1;
                    data_output_trigger <= 0;
                    data_pt_count <= 0;
                    fpga_output <= 0;
                    count_mode <= COUNT;
                end

                default count_mode <= COUNT;

            endcase
        end
    end
end

classify lin_class(.clk100(clk100), .system_reset(system_reset),
↪  .data_in(data_in), .i_val(i_val), .q_val(q_val),
                   .i_pt_line(i_pt_line), .q_pt_line(q_pt_line),
                   ↪  .i_vec_perp(i_vec_perp),
                   .q_vec_perp(q_vec_perp), .state(state),
                   ↪  .valid_output(valid_class_pt));
```

```
    classify_count bin(.clk100(clk100), .system_reset(system_reset),
    ↪  .reset(reset),.data_in(valid_class_pt), .state(state),
                        .excited_count(excited_count), .ground_count(ground_count),
                        ↪  .line_count(line_count));

endmodule
```

A.2.2 Fran's Module: classify.v

```
// perform linear classification of data points
module classify(
    // dynamic input
    input clk100, system_reset,
    input data_in, // indicates if new i,q data is coming in
    input signed [31:0] i_val, q_val, // pt to be classified

    // static input
    input signed [31:0] i_pt_line, q_pt_line, // pt on classification line
    // vector from origin with slope perpendicular to line, pts in direction of
    ↪  excited state
    input signed [31:0] i_vec_perp, q_vec_perp,

    // classified state of input
    output reg [1:0] state,
    output reg valid_output); // boolean: 1 when there is valid output

    reg signed [31:0] i_vec_pt, q_vec_pt; // vector from origin to pt to classify

    reg signed [65:0] dot_product;

    // output state parameters
    parameter GROUND_STATE = 2'b01;
    parameter EXCITED_STATE = 2'b10;
    parameter CLASSIFY_LINE = 2'b11;
    parameter ERROR = 2'b00;

    reg [1:0] comp_state = 2'b10; // fsm to sequentially perform computation steps
    parameter DOT_PRODUCT = 2'b00;
    parameter CLASSIFY = 2'b01;
    parameter RESET = 2'b10;
```

```
// NOTE: MIGHT NEED TO ADD BUFER STATES TO ACCOUNT FOR OPERATION LAG (IF OPS
↪  EXCEED CLOCK CYCLE)
always @(posedge clk100) begin
    if(system_reset) begin
        comp_state <= RESET;
        valid_output <= 0;
    end
    else begin
        case(comp_state)
            DOT_PRODUCT: begin
                dot_product <= i_vec_pt*i_vec_perp + q_vec_pt*q_vec_perp;
                comp_state <= CLASSIFY;
            end

            CLASSIFY: begin
                // EXCITED STATE CLASSIFICATION
                if(dot_product>0) begin
                    state <= EXCITED_STATE;
                    valid_output <= 1;
                end
                // GROUND STATE CLASSIFICATION
                else if (dot_product<0) begin
                    state <= GROUND_STATE;
                    valid_output <= 1;
                end
                // PT ON CLASSIFICATION LINE
                else if (dot_product==0) begin
                    state <= CLASSIFY_LINE;
                    valid_output <= 1;
                end
                // error case
                else begin
                    state <= ERROR;
                end
                comp_state <= RESET;
            end

            RESET: begin
                valid_output <= 0;
                if(data_in) begin
                    i_vec_pt <= i_val - i_pt_line;
                    q_vec_pt <= q_val - q_pt_line;
```

```
                    comp_state <= DOT_PRODUCT;
                end
            end

            default: comp_state <= RESET;

        endcase
    end
end

endmodule // classify
```

A.2.3 Fran's Module: classify_count.v

```
module classify_count(
    // dynamic input
    input clk100, system_reset,
    input reset,
    input data_in,
    input [1:0] state,

    output reg [15:0] excited_count, ground_count, line_count
);

parameter GROUND_STATE = 2'b01;
parameter EXCITED_STATE = 2'b10;
parameter CLASSIFY_LINE = 2'b11;
parameter ERROR = 2'b00;

always @(posedge clk100) begin
    if(system_reset) begin
        excited_count = 0;
        ground_count = 0;
        line_count = 0;
    end
    else begin
        if(reset) begin
            excited_count <= 16'b0;
            ground_count <= 16'b0;
            line_count <= 16'b0;
        end
        else begin
            if(data_in) begin
```

```
                    case(state)

                        GROUND_STATE: ground_count <= ground_count + 1;

                        EXCITED_STATE: excited_count <= excited_count + 1;

                        CLASSIFY_LINE: line_count <= line_count + 1;

                        ERROR: begin ; end

                    endcase
                end
            end
        end
    end
endmodule
```

A.2.4 Fran's Module: hist_2d_pt_to_bin.v

```
// takes in i,q point and finds bin
// outputs bin and (saves to hist_2d_memory -- not being used currently)
module hist2d_pt_to_bin(
        // dynamic input
        input clk100, system_reset,
        input data_in,
        input signed [31:0] i_val, q_val,

        // static input (from config)
        input [7:0] i_bin_num, q_bin_num, // number of bins along each axis (value
        ↪  must be in range 1-63)
        input [15:0] i_bin_width, q_bin_width, // width of a bin along a given
        ↪  axis
        input signed [15:0] i_min, q_min, // bin origin

        // for accessing histogram memory
        /*input [15:0] mem_read_val,
        output [15:0] mem_address,
        output mem_write,
        output mem_reset,
        output [15:0] mem_write_val,*/

        output i_q_found_out, // boolean - 1 indicated valid data output for !
        ↪  stream mode
```

```
    output [7:0] i_bin_coord_out, // can have up to 255 bins along i direction
    ↪ (256th bin counts # outside range)
    output [7:0] q_bin_coord_out // can have up to 255 bins along q direction
    ↪ (256th bin counts # outside range)
);

wire i_bin_found; // boolean: 1 when bin # for i data pt is found
wire [7:0] i_bin_val;
reg [7:0] i_bin_store;

wire q_bin_found; // boolean: 1 when bin # for q data pt is found
wire [7:0] q_bin_val;
reg [7:0] q_bin_store;

reg [1:0] hist_state = 2'b00; // fsm to do 2d hist sequentially
parameter SEARCHING = 2'b00;
parameter ONE_FOUND = 2'b01;
parameter TWO_FOUND = 2'b10;
parameter SEARCH_RESET = 2'b11;

//reg store_data = 0;

reg i_q_found = 0;
reg [7:0] i_bin_coord = 0;
reg [7:0] q_bin_coord = 0;

assign i_q_found_out = i_q_found;
assign i_bin_coord_out = i_bin_coord;
assign q_bin_coord_out = q_bin_coord;

always @(posedge clk100) begin
    if(system_reset) hist_state = SEARCH_RESET;
    else begin
        case(hist_state)
            SEARCHING: begin
                if(i_bin_found && q_bin_found) begin
                    hist_state <= TWO_FOUND;
                    i_bin_store <= i_bin_val;
                    q_bin_store <= q_bin_val;
                end
                else if(i_bin_found) begin
                    hist_state <= ONE_FOUND;
```

```
                    i_bin_store <= i_bin_val;
                end
                else if(q_bin_found) begin
                    hist_state <= ONE_FOUND;
                    q_bin_store <= q_bin_val;
                end
            end

            ONE_FOUND: begin
                if(i_bin_found) begin
                    hist_state <= TWO_FOUND;
                    i_bin_store <= i_bin_val;
                end
                else if(q_bin_found) begin
                    hist_state <= TWO_FOUND;
                    q_bin_store <= q_bin_val;
                end
            end

            TWO_FOUND: begin
                i_bin_coord <= i_bin_store;
                q_bin_coord <= q_bin_store;
                i_q_found <= 1;
                //store_data <= 1;
                hist_state <= SEARCH_RESET;
            end

            SEARCH_RESET: begin
                //store_data <= 0;
                i_q_found <= 0;
                if(data_in) hist_state <= SEARCHING;
            end

        endcase
    end

end

// perform binary search along i axis
bin_binary_search i_search(.clk100(clk100), .system_reset(system_reset),
↪  .data_in(data_in), .value(i_val), .num_bins(i_bin_num),
↪  .bin_width(i_bin_width), .origin(i_min), .binned(i_bin_found),
↪  .current(i_bin_val));
```

```
    // perform binary search along q axis
    bin_binary_search q_search(.clk100(clk100), .system_reset(system_reset),
    ↪  .data_in(data_in), .value(q_val), .num_bins(q_bin_num),
    ↪  .bin_width(q_bin_width), .origin(q_min), .binned(q_bin_found),
    ↪  .current(q_bin_val));
endmodule
```

A.2.5 Fran's Module: bin_binary_search.v

```
// recursive algorithm to find bin! :D
module bin_binary_search(
    // dynamic input
    input clk100, system_reset,
    input data_in,
    input signed [31:0] value,
    // static input
    input [7:0] num_bins, // value must be in range 1-255
    input [15:0] bin_width,
    input signed [15:0] origin,

    output reg binned, // boolean that outputs 1 when value has been binned
    output reg [7:0] current); // contains value bin when binned=1 (in range 0 to
    ↪  num_bins, 63 if out of range)

    reg signed [31:0] val = 0; // for storing value when data_in = 1;

    reg [7:0] max = 8'd10; // max possible bin value
    reg [7:0] min = 0; // min possible bin value
    reg signed [31:0] bin_value; // current bin value we are comparing to

    wire signed [8:0] current_signed;
    wire signed [16:0] bin_width_signed;
    wire signed [31:0] bin_width_long_signed;

    assign current_signed = current; // convert current to signed
    assign bin_width_signed = bin_width; // convert bin_width to signed
    assign bin_width_long_signed = bin_width; // convert bin_width to longer
    ↪  signed

    reg [2:0] search_state = 3'b011; // make FSM to run search algo and properly
    ↪  update values
    parameter UPDATE_BIN_VAL = 3'b000;
```

```
parameter STALL_1 = 3'b100;
parameter RUN_ALGO = 3'b001;
parameter OUTPUT_RESULT = 3'b010;
parameter RESET = 3'b011;

always @(posedge clk100) begin

    if(system_reset) begin
        search_state = RESET;
        max = num_bins;
    end
    else begin
        case(search_state)
            UPDATE_BIN_VAL: begin
                bin_value <= origin+(current_signed*bin_width_long_signed);
                search_state <= RUN_ALGO;
            end

            RUN_ALGO: begin
                // value falls right on bin boundary
                if(val == bin_value) begin
                    search_state <= OUTPUT_RESULT;
                end
                // value in smaller bin
                else if(val < bin_value) begin
                    // value outside of binning range
                    if(current == min) begin
                       current <= 8'b1111_1111;
                       search_state <= OUTPUT_RESULT;
                    end
                    // boundaries have converged to bin
                    else if (val > bin_value - bin_width_signed) begin
                        current <= min;
                        search_state <= OUTPUT_RESULT;
                    end
                    // continue search
                    else begin
                        max <= current; // set new maximum boundary
                        current <= ((current - min) >> 1) + min; // half way
                        ↪  between current and min
                        search_state <= UPDATE_BIN_VAL;
                    end
```

```
                end
                // value in current or larger bin
                else begin
                    // boundaries have converged to bin
                    if (val - bin_value < bin_width_signed) begin
                        current <= min;
                        search_state <= OUTPUT_RESULT;
                    end
                    // value outside of range
                    else if (current == max-1) begin
                        current <= 8'b1111_1111;
                        search_state <= OUTPUT_RESULT;
                    end
                    // continue search
                    else begin
                        min <= current; // set new minimum boundary
                        current <= ((max - current) >> 1) + current; // half
                        ↪  way between current and max
                        search_state <= UPDATE_BIN_VAL;
                    end
                end
            end

            OUTPUT_RESULT: begin
                binned <= 1;
                search_state <= RESET;
            end

            RESET: begin
                binned <= 0;
                max <= num_bins;
                min <= 8'b0;
                current <= num_bins>>1; // start at middle of range
                bin_value <= 32'd0;
                if(data_in) begin
                    val <= value;
                    search_state <= UPDATE_BIN_VAL;
                end
            end
        endcase
    end
```

```
    end

endmodule
```