R3CAP: Real-Time 3D Motion Capture
Final Project - ECE-395-A

Spring 2019

Partners: Brian Frost-LaPlante
Nikola Janjušević
Karol Wadolowski

Advisor: Sam Keene

Abstract

In this paper we propose a system for the real-time tracking of multiple RF sources in three-dimensional space without the use of RSSI data. We propose the use of angle-of-arrival estimation using antenna arrays at two distinct locations, followed by triangulation based on these two computed angles. All computations are to be distributed between an FPGA and a connected host computer. The estimated locations are to be visualized on the host computer. We outline the design and operation of a functioning prototype system, and describe methods by which it can be scaled to a commercially viable motion capture device.

Contents

1 Introduction

Many modern systems for motion capture, such as infrared reflectometry and computer vision, rely on the maintenance of line-of-sight between the tracking apparatus and the object in question. Inspired by the ubiquity of radio frequency identification (RFID) tags in industry, as well as their small size, we considered the plausibility of motion capture of Radio Frequency (RF) sources without maintenance of line-of-sight.

There has been significant activity around the localization of RFID tags, but most 3D motion capture implementations rely upon Received Signal Strength Index (RSSI) to determine the distance from the receiver to the transmitter. The existence of line-of-sight affects RSSI directly, so these tracking methods fail outside of heavily controlled environments.

We thereby propose a method of motion capture which does not rely on RSSI. Instead, we implement Angle-Of-Arrival (AOA) estimation at two receiver locations and localize using triangulation. This method is independent of line-of-sight between the receivers and the RF source.

We do not focus on RFID tags due to the added complexity incurred by wireless power transmission. Instead, we design for the motion capture of active RF transmitters, and perform experiments with a Software-Defined Radio (SDR).

Using this method, we were able to construct a system which can localize an SDR in space. This document outlines the design of this system, the observed functionality of the prototype device, and a procedure by which to improve the prototype device to achieve higher accuracy.

2 Related Works

2.1 RFID Tracking

We have found several papers from the past decade which implement RFID angle-of-arrival estimation, localization and tracking, although none do so for multiple tags in three dimensions. Although we are not using RFID tags specifically, they operate as constant-frequency RF transmitters, so results about their localization have clear applications to our proposed design.

In 2010, Angerer et al [1] performed two-dimensional angle-of-arrival estimation of a single RFID tag using a field programmable gate array (FPGA) and antenna array. The FPGA acts as an RFID interrogator and reader, and communicates read data to a host computer. This computer then performs an AOA estimation algorithm based on the multiple signal classification (MUSIC) algorithm. With only two antennas in their array, they found an error of only 3.3°. This is a subset of our project in some sense; they are doing only AOA in two dimensions rather than three-dimensional AOA with localization, they are only tracking one tag; but the results serve to inform our project greatly. Using only two antennas, they were able to achieve a relatively high accuracy, meaning that the accuracy of a similar system could potentially be improved even further by a larger array. It also informs the decision to use an FPGA. This paper most importantly serves as a proof of concept for a large subset of our project.

In 2017, Qiu et al [3] created a 2D RFID localization system for multiple tags using a single phased array antenna and RSSI data. This system is the most similar to what we would like to create, specifically due to the use of antenna arrays, and the tracking of several tags in space. This implementation is rather limited, due to its reliance upon RSSI data, but it provides a good point to build off of.

2.2 Digital Downconversion and Quadrature Demodulation

In 2008, Bernal et al [4] published a paper titled *Digital IQ Demodulation in array processing: Theory and Implementation*. They present a very simple scheme for performing quadrature demodulation and downconversion in digital hardware, given a certain arithmetic relationship between the sampling frequency and the carrier frequency. The methods presented in this paper suggest a signal flow for a project like ours in which quadrature demodulation and downconversion occur on an FPGA.

3 Implemented Design

To localize RF sources through triangulation, as described in the introduction, we must estimate the AOA from two distinct locations in space. We thus have two RF-frontends – antenna arrays with additional analog circuitry – fixed at either end of a wooden mount. The purpose of the RF-frontends is to pick up signals from the transmitters, amplify them, and downconvert them to the point that they can be sampled and processed digitally. This process can be visualized with the block diagram in Figure 1.

To produce AOA estimates we use the MUSIC algorithm, first proposed by Schmidt [20], which uses phase data at a number of evenly spaced antennas to determine the incoming angle. Thus, each RF-frontend must contain an array of antennas, each with an associated receiver chain.

The MUSIC algorithm takes as an input an autocorrelation matrix generated from the signals at each antenna. The signals must first be quadrature demodulated (i.e. IQ demodulated) to generate such a matrix. We use an FPGA to sample the downconverted data from the RF-frontend, perform quadrature demodulation and generate the autocorrelation matrices for both arrays at once. The FPGA also controls the Automatic Gain Control (AGC) on the RF-frontends.

The autocorrelation matrices are then sent to a host computer which performs the MUSIC algorithm for each array. A peak finding algorithm is used to find the AOAs of the RF sources relative to each of the two arrays. It then performs triangulation, and lastly displays the location of the sources. For this display and computation, we use Unity – an open-source game development software. It is free, and can easily be made to interface with the FPGA.

Before implementing and designing any physical portions of our project we ran some simulations in MATLAB. The simulations were based around the MUSIC algorithm and its viability for our application. With these simulations, we were able to determine the type of localization accuracy that we would see with various amounts of antennas in each array and various antenna spacings. These simulations also helped us decide on the the number of samples to use for autocorrelation matrices. From these simulations we determined it was best to have 3x3 arrays. Unfortunately,

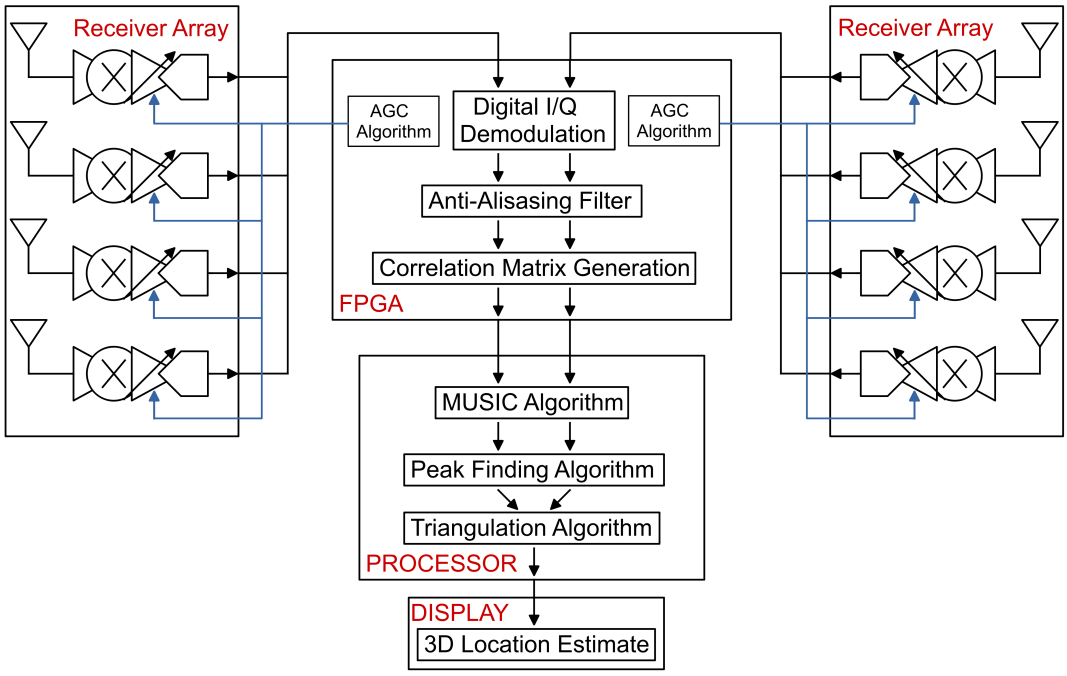

Figure 1: Block Diagram of R3CAP. Two arrays, each of 4 receivers, provide sampled data to the FPGA which processes the data to produce two correlation matrices. In the background, the FPGA digitally controls the amplification of each receiver to ensure constant amplitude signals. The FPGA passes the correlation matrix data to a computer processor to produce separate MUSIC spectra and AOA estimates for each array. The processor then combines the AOA estimates from each array to produce a 3D location estimate (via triangulation). Lastly, the location is displayed to the user in a graphical environment.

due to the number of available pins on the FPGA, the size of the arrays had to be decreased to 2x2. The number of samples that we decided to use for the autocorrelation matrices was 1024. The spacing between antenna arrays was chosen to be 1 m.

In this section, we describe the operation of and design process behind the parts of R3CAP. We do so in order of signal path.

3.1 RF-Frontend

The RF-frontend is responsible for interfacing the digital processing with the physical world via signal acquisition. R3CAP's frontend consists of two identical, custom-designed, phased array antennas separated by a fixed distance of one meter to allow for signal triangulation. In order to be able to extract 2D angle-of-arrival information from the incoming signals, the phased array antennas were constructed as a 2D grid. The array was designed for a digital beamforming scheme,

meaning that each receiver must provide sampled data (ie. have its own analog to digital converter (ADC)). As such, each receiver is simply an identical implementation of one mother-receiver design. The final implemented array design consists of a 2x2 rectangular grid of these receivers, spaced one half-wavelength apart ($\frac{\lambda}{2} \approx 16$cm). The subsequent subsections will deal with the design of the mother-receiver first, and then detail the phased array antennas.

3.2 Mother-Receiver

The mother-receiver (RX) itself consists of several sub-components: the antennas, RX-backbone, automatic gain control, and analog to digital conversion, each of which were designed and prototyped separately before being assembled as a fully functioning wireless receiver on the final implemented array. The RX design went through three iterations and was designed primarily through the method of selecting components via inputting parameters of prospective devices in a gain and compression table (see Section 10 Appendix). Figure 2 shows the block diagram of the mother-receiver.

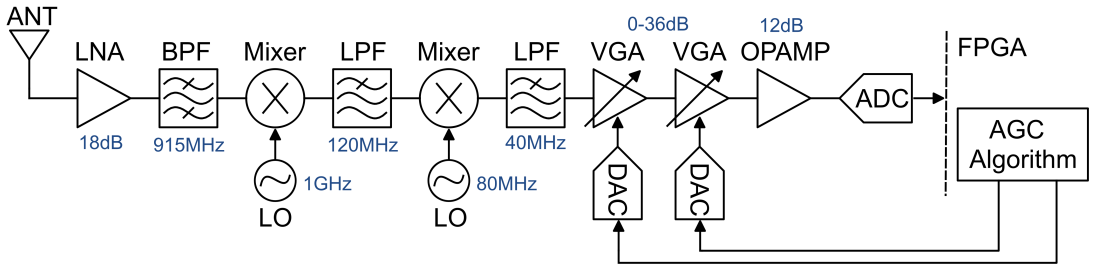

Figure 2: Mother-Receiver block diagram.

3.3 Antenna

Johanson Technology chip antennas were chosen primarily for the reason of size and price. The design of their layout on the array PCBs was followed from the recommendations of the datasheet, [9]. In the design process, a small prototype PCB, see Figure 3, was designed and ordered for a proof of concept to demonstrate that these components would work for the project, and that matching could be performed. This step proved crucial in showing just how non-trivial producing matching circuits could become.
The goal of antenna matching is to reduce the reflection coefficient seen by the incoming signals at the antenna. This is given by the scattering parameter $S_{1,1}$ for the antenna at 915 MHz, our frequency of interest. However, the relationship is somewhat indirect, as we wish to match the antenna to the input impedance of our receiver chain, as opposed to an ideal 50Ω load.

Matching a single antenna takes several steps. The first step is to calibrate a Vector Network Analyzer (VNA). This is necessary as we need to ensure that the reflection coefficients we will be measuring are accurate. After calibrating the VNA, we power on our full RF-frontend (Figure 38) and connect the VNA to the SMA connection on one of our chains. With this setup, the input impedance of our RF chain can be measured. On the VNA, we save the file corresponding to this

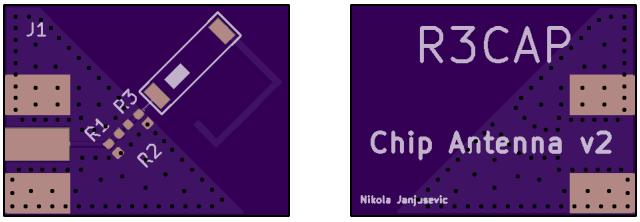

Figure 3: Render of antenna prototype boards, frontside left, backside right. The PCB has four layers (to simulate the four layer environment of the final board) and has space for a pi-matching circuit and chip-antenna.

input impedance. An example of the type of response we see at the input to our RF-frontend can be seen in Figure 4. After obtaining this input impedance data, the receiver chain was disconnected from the SMA. A 0 Ω resistor was then used to short the SMA connector to ground. This was done so that we could apply proper port extension to the VNA. After port extension, the resistor was removed and the antenna was soldered on along with two 0 Ω resistors in series, so that our SMA was connected to the antenna. The VNA was then used to measure the reflection coefficient of the antenna. This data was saved as well. An example of the antenna reflection coefficient can be seen in Figure 5.

Using the Optenni Lab software, we loaded the antenna's reflection coefficient data. Optenni Lab allows you to choose an impedance file to match to, for which we select our input impedance file from the RF-frontend's receiver chain. We can then select the matched frequency, the range of component values we have and the supplier they come from. Optenni Lab can produce an optimal matching circuit using these components in the Π circuit topology. After their generation, these circuits can be tuned in software, either as a result of us not having the correct value or manufacturer associated with the part. Once we find a circuit which seems to be a viable match, we solder on the recommended components to the PCB.

The reflection coefficient of the antenna circuit is then remeasured with the VNA. An example of a matched circuit's reflection coefficient can be seen in Figure 6. If the reflection coefficient after the Π circuit's addition is not satisfactory, then Optenni Lab is used again to recommend a single additional element in shunt. This would then be implemented and viewed again. This entire process, not including VNA calibration, was done for the matching of each of the 8 antennas.

3.4 RF-Backbone

The RF-backbone is implemented as a double Intermediate Frequency (IF)-stage receiver. It takes an incoming 915 MHz signal, downconverts it to 85 MHz with a 1 GHz local oscillator (LO), and then downconverts this signal again by a 80 MHz LO to 5 MHz. It is then amplified and sampled by the AGC and ADC blocks respectively (as seen in Figure 7). The details of the schematic for the backbone were mostly implemented by following the respective datasheets for each of the components. These schematics can be seen in Section 9 Appendix.

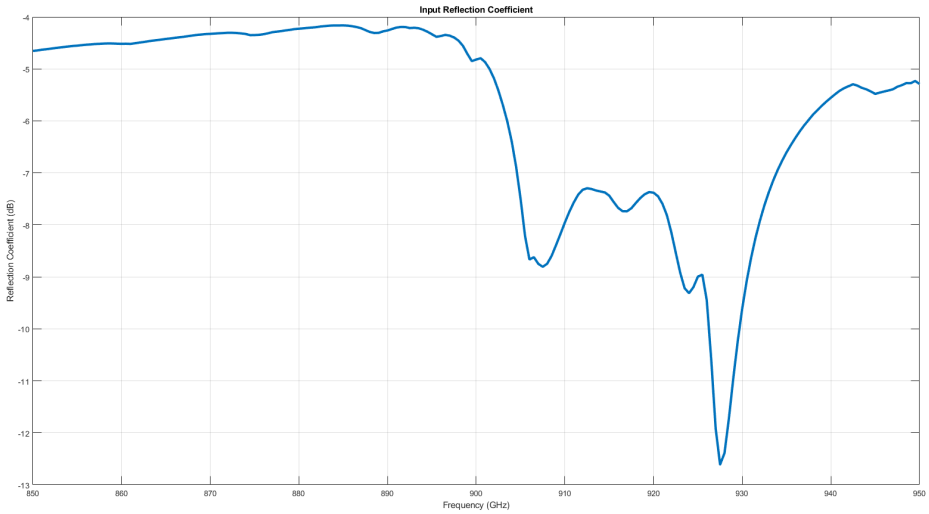

Figure 4: The reflection coefficient as seen at the input of the receiver chain.

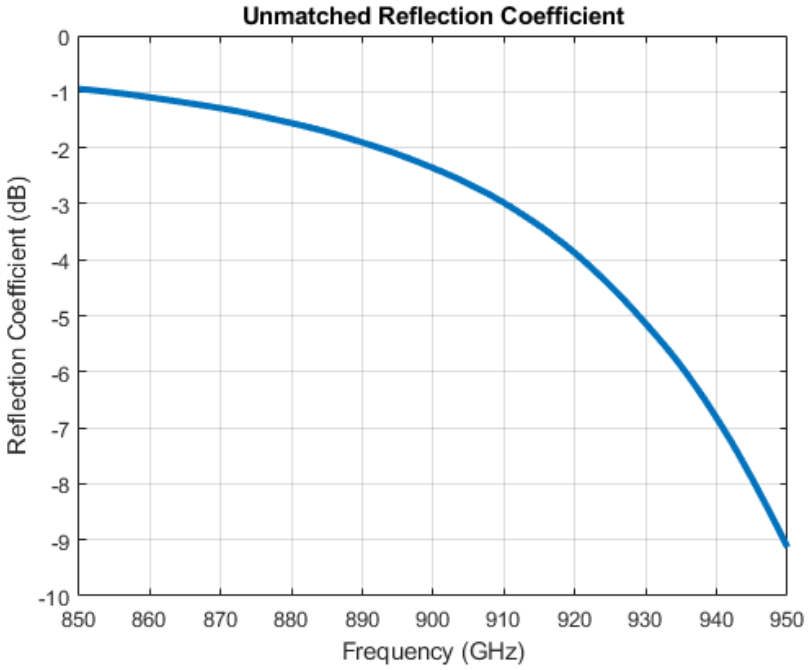

Figure 5: The reflection coefficient of the unmatched antenna.

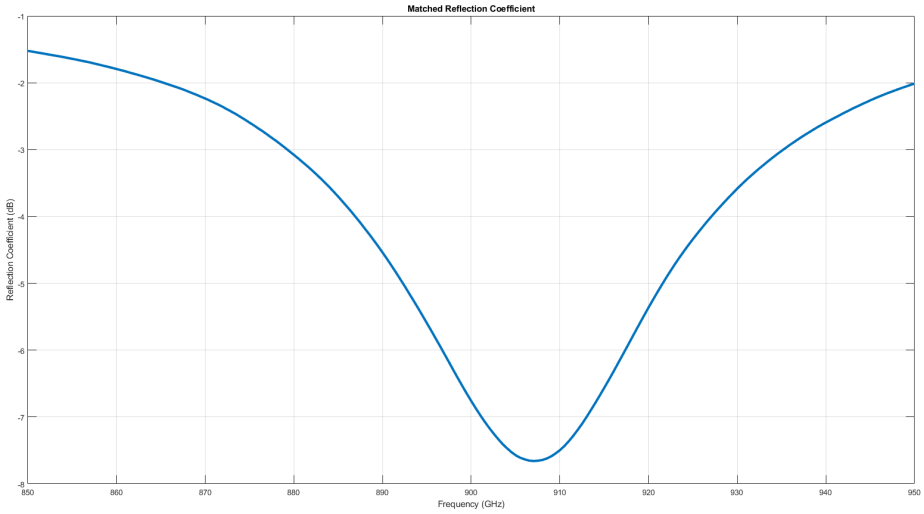

Figure 6: The reflection coefficient of the matched antenna circuit.

The signal from the antenna is first taken to the low noise amplifier (LNA). This component is important for setting the noise figure (the ratio of input noise to output noise) of the receiver as it dominates the noise figure equation in a cascaded system. The LNA feeds the signal into a bandpass filter (BPF) [11] to eliminate out-of-band interference from other sources. The filter chosen is a surface acoustic wave filter with a center frequency of 915 MHz and a 10 MHz bandwidth. To help reduce the effects of the filters undesirable max voltage standing wave ratio of 1.5, 3 dB attenuators [10] were added before and after the filter. This effectively decreases the return loss of the filter by 6 dB at each port.

Next, the BPF feeds into the two IF stages. Each of these stages comprises a double-balanced mixer [12] followed by a low-pass filter (LPF). The mixers were chosen because they have a relatively high input compression point (5 dBm) that allows for higher-power input signals to the chain. The mixer also has an internal output amplifier giving it an overall positive conversion gain of 0.6 dB. Additionally, the mixers are able to use quite low power levels at the LO input ports (-10 dBm), allowing the 5 dBm local oscillators to not require amplifiers after being split for the four receivers. One downside to the chosen mixer is that the input and output ports are both balanced lines, complicating the design. We opted to keep most of the receiver single-ended, so the mixers both required balanced-to-unbalanced conversion (known as the use of a "balun"). This was achieved using discrete components as per recommendation on the datasheet for narrow-band applications [12].

The output of the unbalanced mixer is then fed into a low-pass filter to remove the upper-sideband signal from the mixing process. In the first IF stage, the LPF used has a 3 dB cutoff frequency at 120 MHz, easily cancelling out the undesired 1.915 GHz components from mixing the 915 MHz signal and 1 GHz LO. At the second IF stage, a 40 MHz LPF is used. This both cancels out the 120 MHz upper-sideband mixer output and suppresses any LO leakage into the IF output. Overall, the

RF-backbone of the mother-receiver contributes a net positive gain of 6.5 dB to the signal path.

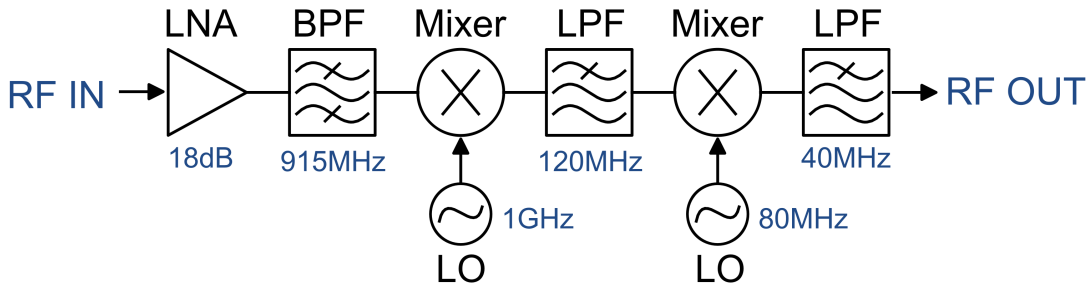

Figure 7: Render of the 2x2 Array PCB, front side.

Figure 8 below shows the prototype of the RX RF backbone. It was used to confirm correct implementation of the components and to debug our initial design.

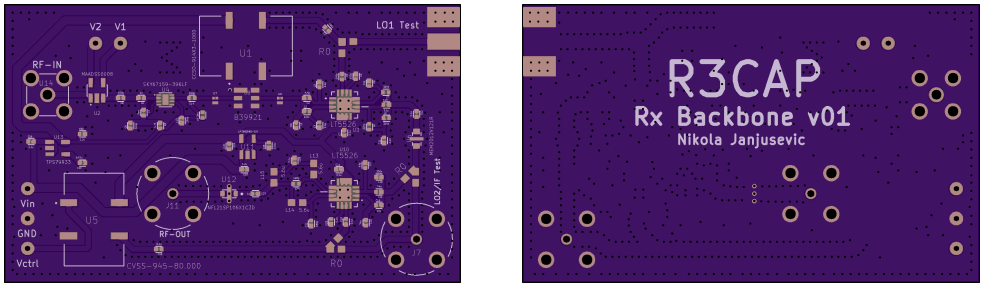

Figure 8: Render of the RX Backbone prototype boards, frontside left, backside right. The board connects an input RF signal to a 15 dB attenuator (for testing purposes) followed by an LNA and two IF stages (mixers bringing the 915 MHz signal to 85 MHz and 5 MHz).

3.5 Automatic Gain Control and Analog-to-Digital Conversion

Automatic gain control is used to present the analog-to-digital converter a signal that appears to be of constant amplitude regardless of proximity to the transmitter. The implemented scheme uses two cascaded analog variable gain amplifiers (VGAs) [14], each controlled by a separate AGC algorithm on the FPGA. The FPGA interfaces to the analog VGAs through digital-to-analog converters (DACs). Figure 9 An output voltage of 3.3 V – the rail voltage of the ADC – was chosen so that the full resolution of the ADC could be taken advantage of. This rail-to-rail output voltage is, however, not achieved through the use of the cascaded VGAs alone due to the compression points of the amplifiers. The chosen VGAs [14] have variable compression points, approximately linear in dB from -5 dBm to 8 dBm, dependent on the current gain of the amplifier. This compression range makes it impossible to bring a signal up to 20 dBm (approximatley 3.3 V), as any input power level

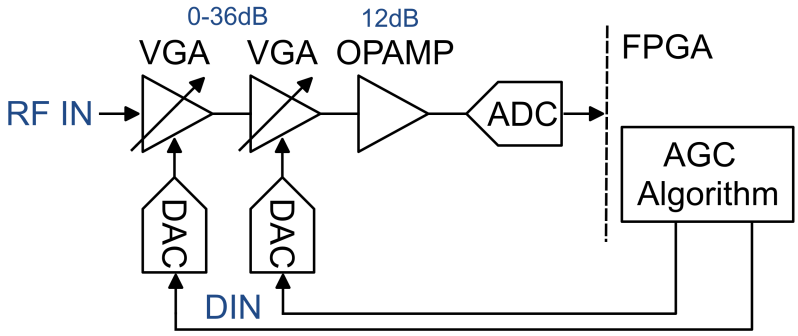

Figure 9: Block Diagram of the AGC and ADC sections of the Mother Receiver design.

theoretically able to be pushed up by the VGA will put the amplifier in compression. To mitigate this, a high-speed op amp [15] was used to bring down the required power that the VGAs need to output from 20 dBm to 8 dBm. To do so, the op amp is configured to provide 12 dB of gain. Placing the op amp at the output of the VGAs has the added effect of providing a low-impedance input to the ADC (as requested in its datasheet) [16]. The op amp feeds into the ADC, which is controlled digitally by the FPGA. The sampled data is used by the AGC algorithm to adjust the gain of the VGAs accordingly. Details of the AGC and ADC circuits can be seen in Figures 42 and 43 in Section 9 Appendix. Details on the AGC algorithm are discussed in Section 3.8.5.

Automatic gain control and analog-to-digital conversion function blocks were prototyped separately. The ADC prototype board, seen in Figure 10, contains two ADCs on a single board and BNC connectors for function generator input. This board was instrumental in kicking off some basic testing and debugging on the FPGA design, although it lacked the op amp driver circuit used in the final implementation.

Figure 11 shows the AGC prototype boards. Each board is a reproduction of the AC-coupled evaluation board schematic found on the VGA datasheet [14], along with the eval-board schematic for the DAC, found in its datasheet [17].

3.6 Local Oscillators

For the implemented array design, involving a 2x2 RX grid, a single local oscillator was routed to each RX for each mixing stage. Both the 1 GHz [18] and 80 MHz [19] oscillators were chosen for their sinewave outputs (ultimate spectral purity), and +5 dBm output power. This allows them to be split into 4 paths for the 4 receivers without requiring extra amplification, as the mixers require an LO input above -10 dBm, and the splitting brings the LO power level to only -1 dBm. These components were first tested on the RX backbone prototype boards. Figure 45 and 46 show the schematics for the 1 GHz and 80 MHz LO configurations used in the final implementation. Transformer-based power splitters were used to ensure maximum power efficiency (some resistor-based splitters end up reducing signal power by more than half).

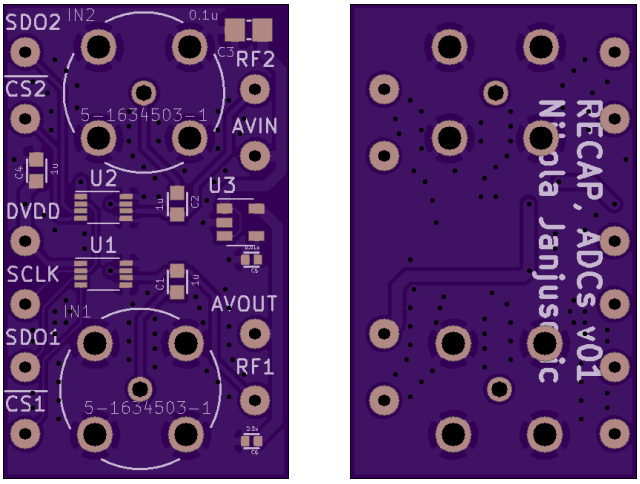

Figure 10: Render of ADC prototype boards, frontside left, backside right. The board has space for two ADCs and two RF inputs.

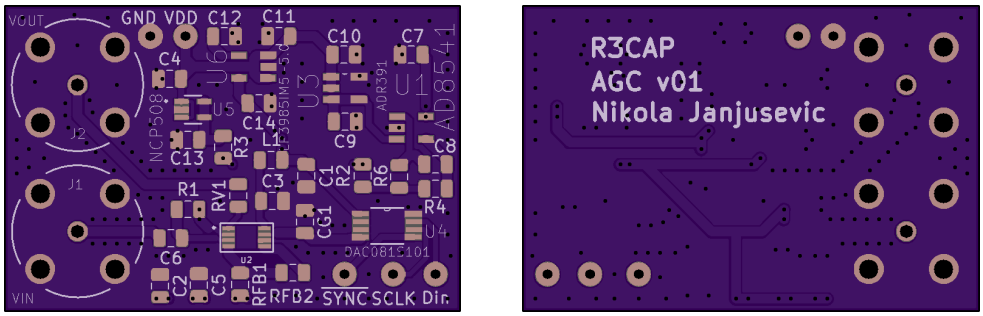

Figure 11: Render of the AGC prototype boards, frontside left, backside right. The board has a variable gain amplifier connected to a DAC, with according digital inputs and RF inputs/outputs.

3.7 2x2 Array

The 2x2 array was implemented on a 4-layer FR4 PCB. It has 4 identical layouts of the mother receiver schematic arranged radially, with the antennas at the corners separated by a distance of a half-wavelength at 915 MHz. The bottom and top layers of the PCB were used for routing signals, while the middle two layers were used for routing power. The board also houses the two LOs, each of which are split into four separate signals. A 5 V and 3.3 V voltage regulator are used to power every component on the board with the exception of the digital supply of the ADCs. The ADCs are powered by the FPGA's fixed 3.3V output.

In PCB design, special attention was paid to ensure that crossing RF signals were separated by ground planes. Figure 44 shows a rendered version of the 2x2 array layout in all its glory. Figure 40 in Section 9 Appendix shows the schematic for the entire array, with sub-schematics presented in subsequent figures of the Appendix.

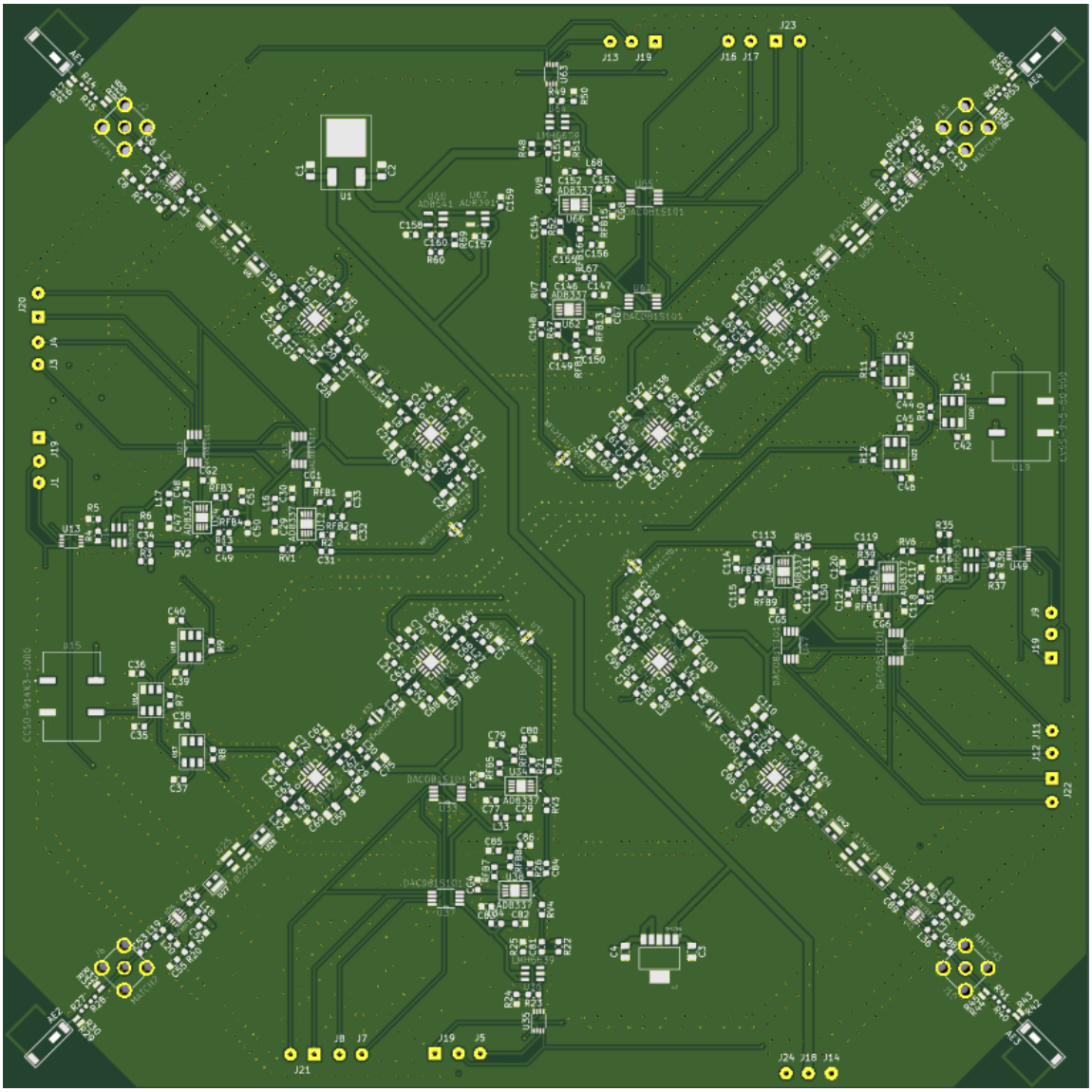

Figure 12: Render of the 2x2 Array PCB, front side.

3.8 FPGA

In order to sample the data from all eight ADCs in parallel and create autocorrelation matrices for both arrays, as well as perform automatic gain control, we use the ZedBoard FPGA [22]. While a microcontroller could just as well be used for sampling and gain control, an FPGA is chosen for the ease with which all 8 signals can be sampled and processed in parallel. This parallel acquisition and processing is necessary for beamforming, as we require accurate phase data from all 8 signals

at specific snapshots in time.

Furthermore, the ease with which Finite Impulse Response (FIR) filters can be implemented on an FPGA is of great use. We also implement a Digital Downconversion (DDC) and IQ demodulation method which can be in low-level logic, further justifying the FPGA choice. We use the ZedBoard FPGA because it was available at the school, and FPGAs are otherwise expensive.

The ZedBoard FPGA contains both a Corex-A9 Processing System (PS) and 85,000 Series-7 Programmable Logic (PL) cells. The PL can be programmed using a hardware description language (HDL) such as Verilog or VHDL within the Xilinx Vivado software. Through the creation of Intellectual Property (IP) blocks, the PL can pass data to the PS. The PS is programmable through the Xilinx Software Development Kit (SDK), using high-level languages like C or C++. The PS can then send messages through the Universal Asynchronous Receiver-Transmitter (UART) protocol to any computer to which it is connected. All HDL code can be found in Section 11 Appendix.

The FPGA must first acquire the data from the ADCs. Then, it must perform IQ demodulation on each incoming signal, and apply an anti-aliasing filter to both the I and Q signals for each input. Then, it must take the required outer product computations and average as necessary to generate the autocorellation matrix entries. All of these tasks can be performed in the PL, after which the PS can retrieve these matrix entries and send them to the host computer through UART for further processing. This must all happen quite quickly as well, or else the implementation cannot be considered real-time.

3.8.1 Data Acquisition

We look to sample at about four times the bandwidth (BW) of our signal so that we safely sample above the Nyquist rate. Our RF transmitter has, optimally, no bandwidth, so we design around the maximum bandwidth of Ultra-High Frequency (UHF) RFID tags. These tags are rated as having 250 kHz bandwidth, meaning that a sampling rate of 1 Msps satisfies our requirements. Our MATLAB simulations show that 8-bit quantization does not compromise our results, so we use the ADS7040 1 Msps 8-bit serial ADC from Texas Instruments.

We choose a serial ADC rather than a parallel one, as parallel ADCs require more general purpose input/output (GPIO) pins on the FPGA to use. To illustrate, a single 8-bit parallel ADC requires 8 pins for parallel input and one for a clock signal from the FPGA. On the other hand, an 8-bit serial ADC requires only one pin for serial input, one for a clock signal and one for a select signal from the FPGA for a total of only three GPIO pins.

The unfortunate drawback of serial ADCs is that they have a more complex protocol. The ADS7040 uses a variant of the Serial Peripheral Interface (SPI) protocol which uses two control signals and one serial data output (SDO) signal. The control signals are a negated chip select, $\overline{CS}$, and a serial clock, $SCLK$. To start up the ADCs, the chip is selected and clocked at least 16 times. After startup, the ADC must be clocked according to the timing diagram in Figure 13. We have no trouble meeting the timing requirements, as the ZedBoard has a 100 MHz on-board clock.

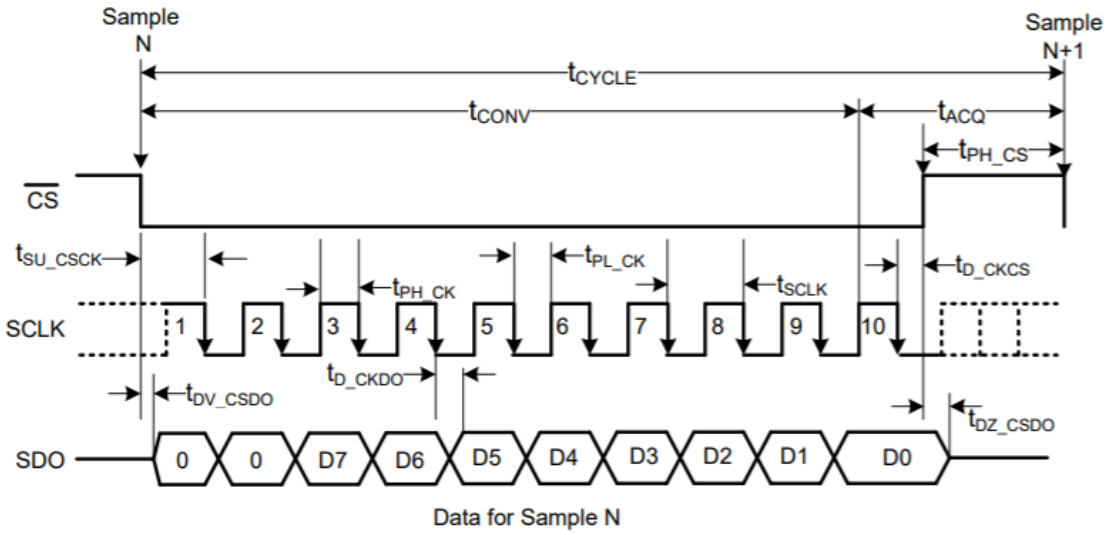

Figure 13: ADC timing diagram in normal operation mode, from the datasheet for the ADS7040. The specific timing intervals are specified in the datasheet, but are omitted here for brevity.

We have written Verilog code which, on startup, satisfies the startup sequence of the ADCs, and then proceeds clocking the ADCs according to the datasheet's timing diagram. We also must, from this diagram, determine when the data can be read into the FPGA from the *SDO* pin. To do this, we implement D-type flip-flops on the FPGA for each data bit. Each flip-flop clocks in the input at the *SDO* pin on the rising edge of *SCLK* for a single data bit, so that at the end of chip select period, the entire data byte is stored in D-type flip-flops. This byte is then moved to a register before the next sample is taken.

We test the clock signals first by simulating the operation of this code in Vivado. The output of this simulation can be seen in Figure 14.

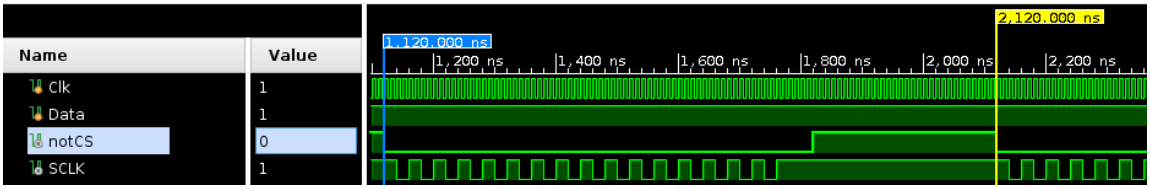

Figure 14: Vivado simulation of the ADC clocks, corresponding to the datasheet's timing diagram and having a 1 μs sample period.

We then perform a test by loading this code onto the FPGA and observing the outputs with an oscilloscope. Screenshots from this test can be seen in Figure 15. When compared to the timing diagram, you can see the clocking is proper. The shapes of the pulses in these plots are not square,

but this is merely an artifact of the scope probe's inability to capture signals at this frequency well.

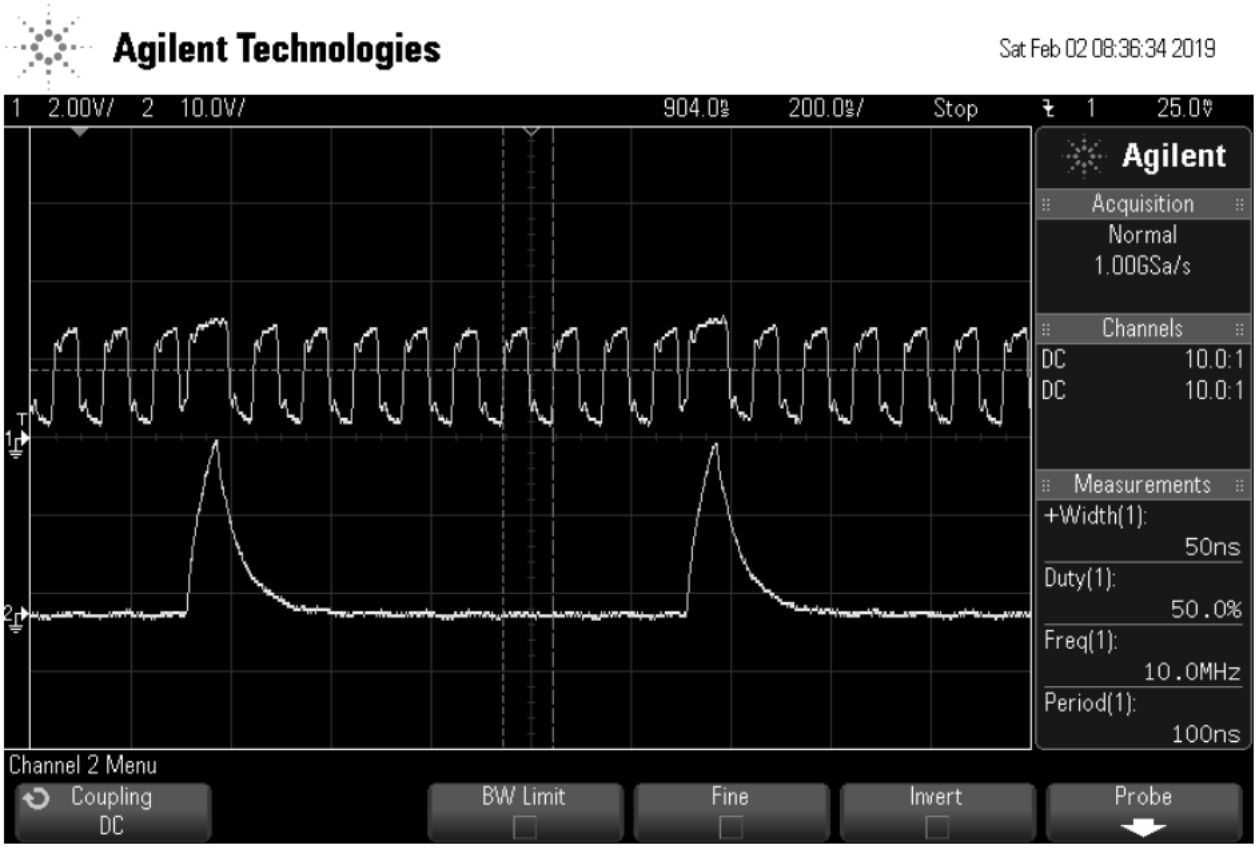

Figure 15: Oscilloscope screenshot of $SCLK$ and $\overline{CS}$ outputs from the FPGA when the ADCs are in standard operation mode (as opposed to startup). The top signal is $SCLK$, while the lower one is $\overline{CS}$.

Once we had ADC prototype boards, we could then verify the operation of the ADCs themselves. We did this first by observing an ADC's SDO output on a scope while it was being clocked by the FPGA, as seen in Figure 16. We can see key properties of the output this way, but the exact values are difficult to read given only the signal in time. To test the output values more precisely, we compiled an IP which allows us to print ADC samples to a serial console on a computer. This allows us to verify that we are reading in bytes correctly.

It is also important to note that the ADCs output values between 0 and 255, corresponding to positive voltages at their inputs. This DC bias is a result of our single-ended frontend, but is undesirable for our digital signal processing. On account of biasing in the RF frontend, we expect the ADC input to be centered at half-rail, or 128 in binary. To remove this DC bias, we can thereby subtract 128 from each ADC input. Using the 2's complement system for signed binary numbers,

subtraction of 128 is equivalent to flipping the most significant bit. This is implemented before any further processing.

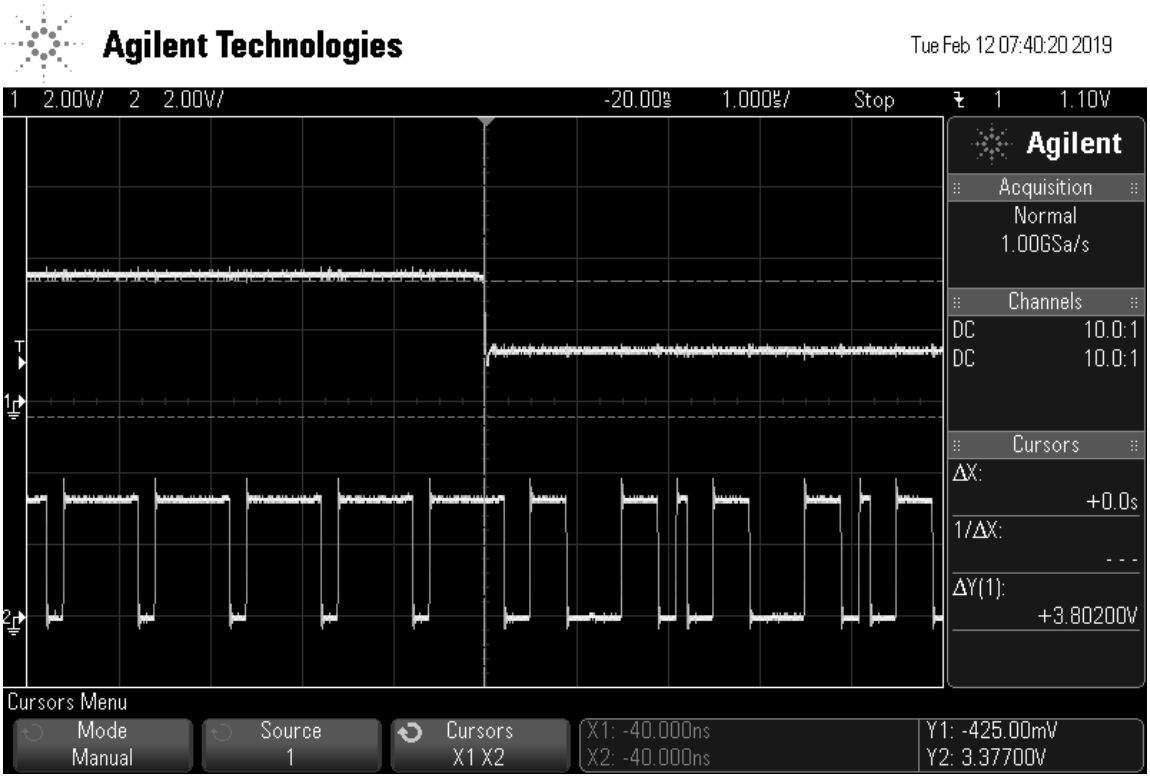

Figure 16: The top signal is the input to the ADC going from high to low in time. The bottom signal is SDO for that ADC. The outputs of the ADC are periodic while the input is constant, as expected.

3.8.2 Digital Downconversion and IQ Demodulation

To perform DDC and quadrature demodulation we followed the work of [4]. We are sampling an incoming signal with an assumed bandwidth of 250 kHz at 1 Msps, and our center frequency f_{IF} is 5.25 MHz. This can be expressed as $f_{IF} = kf_s \pm \frac{f_s}{4}$ for some integer k. The transformation from analog frequency to digital frequency is given by $w_d = \frac{2\pi f}{f_s}$, so the digital signal has a spectrum like that shown in Figure 17.

To perform DDC, we need to shift the spectrum in either direction by $-\frac{\pi}{2}$. Once this is done, there will be one copy of the signal at baseband, and one at a higher frequency. After an anti-aliasing filter, this gives us a baseband spectrum as in Figure 18. In the time domain this shift is equivalent to multiplying the sample at time n by $e^{j\frac{\pi}{2}n}$. This multiplication can be done using the circuit in Figure 19.

The 2's complement negation blocks in this circuit require a bit of extra explaining – for an 8-bit input, one cannot represent the 2's complement negative of every input with an 8-bit number. More precisely, the number -128 (represented in binary as 10000000) has no 2's complement negative within an 8-bit system. We thus implement a multiplexer within the 2's complement block which carries out standard 2's complement negation if the input is not -128, and outputs 127 if the input is -128.

This, of course, incurs an error of 1 bit in the case of a negative-rail input. The 8-bit ADC operates over an input range of 0 V to 3.3 V, so a single bit difference represents a voltage difference of $\frac{3.3\text{V}}{255}$, or approximately 13 mV. This is not significant enough to cause major errors.

Note that the two outputs of this circuit are the real and imaginary parts of our baseband signal, or the in-phase component (I) of the signal and the quadrature component (Q) of the signal respectively. The circuit works by following the state transition table shown in Table 1.

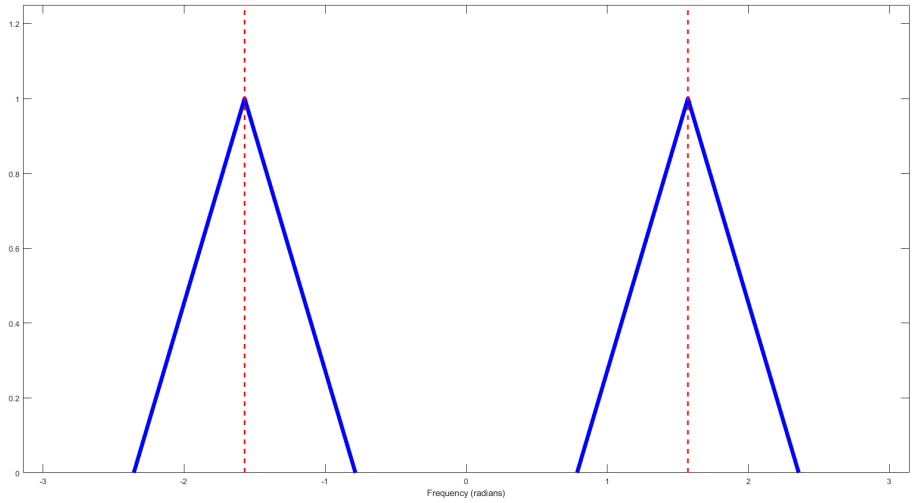

Figure 17: Output spectrum of the ADC.

Current State		Next State		Outputs	
S1[n]	S0[n]	S1[n+1]	S0[n+1]	$x_I[n]$	$x_Q[n]$
0	0	0	1	x[n]	0
0	1	1	0	0	-x[n]
1	0	1	1	-x[n]	0
1	1	0	0	0	x[n]

Table 1: Outputs of IQ demodulator and next state depending on current state.

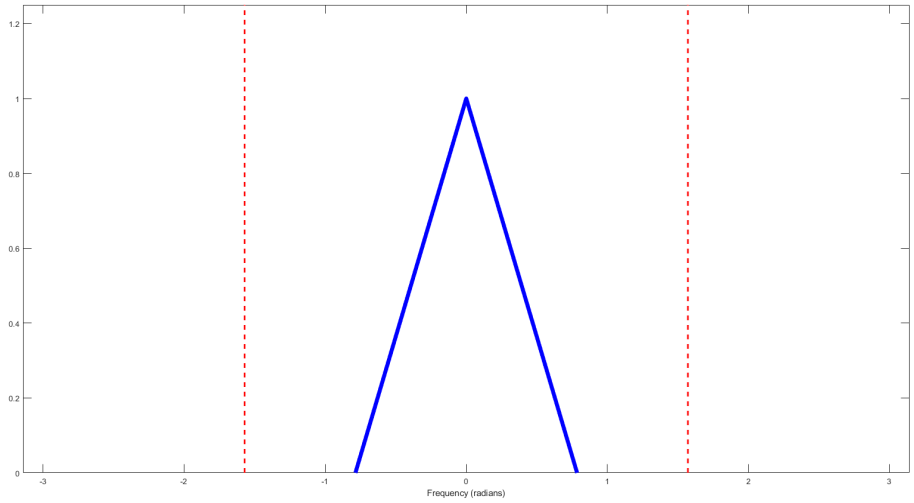

Figure 18: ADC output after a shift to baseband and low-pass filtering.

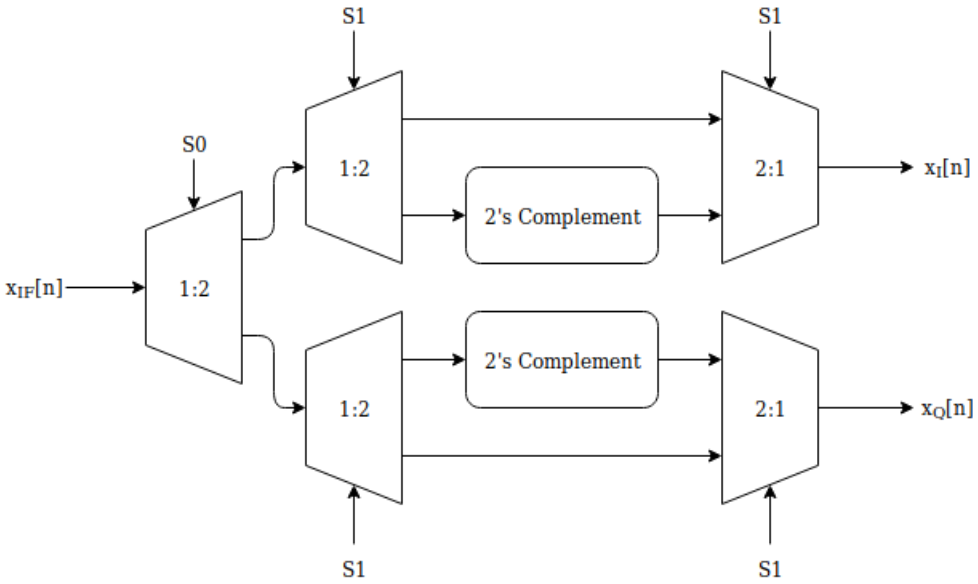

Figure 19: Digital quadrature demodulation circuit block diagram. 2's complement blocks output the negative of the input according to the 2's complement system for signed binary numbers.

This method of DDC and quadrature demodulation facilitates implementation in digital logic. Verilog code is written which implements the circuit in Figure 19 at the gate level, which imposes

minimal delay. Note that each 8-bit input is now represented by a 16-bit complex number.

3.8.3 Low-Pass Filter

As noted above, the applied digital downconversion and quadrature demodulation method produces an aliased signal centered at $\omega_d = \pi$. Our desired signal is in the frequency range of $\omega_d \in [-\pi/4, \pi/4]$. We choose a filter whose stopband begins at $\omega_d = \pi/2$, and whose passband ends at $\pi/3$. This safely ensures that our signal passes through the filter, and the stopband begins at a lower frequency than necessary so as to remove both the aliased signal and noise in intermediate bands.

We sacrifice a bit of passband ripple for an attenuation of 50 dB in the stopband. This is motivated by the fact that our desired spectrum is heavily concentrated at a single frequency, so ripple in the passband is of little effect. On the other hand, our tests showed that lower attenuation values on the order of 35 dB were not sufficient to eliminate aliasing effects in our autocorrelation matrices.

We use MATLAB's fdatool to design the filter. We found that using the p-norm minimizing filter with $p = 128$, we can achieve a filter with the desired specifications using 22 filter taps. Using fdatool's quantization features, we are able to ensure that the filter still meets the specifications with 8-bit inputs and 16-bit coefficients (where the coefficients have 15 fractional bits).

The coefficients are thus generated and fed into Vivado's FIR compiler IP. The frequency response of this filter can be seen in Figure 20.

We implement 36 such LPFs on the FPGA – one for each I and Q component – and apply them in parallel to our I and Q data immediately after they have been generated.

To test the operation of the IQ demodulator and LPF, we generate two out-of-phase sine waves at 1.25 MHz in the Vivado software and input them into our ADC module. We feed this output to the IQ demodulator and LPF module, and observe the output in Vivado. For the first sine wave, we expect to see Q constant and nonzero, and I to be constant at zero after filtering. For the second sinewave, we expect Q to be constant at zero, and I to be constant and nonzero. This is observed in Vivado's simulator, as shown in Figure 21.

Note that the low-pass filter incurs another enlargement of the data – each of the I and Q components for a given signal is 24 bits. Thus, after the filter, each value is represented by a 48-bit complex number. When designing the filter, we specified that 22 of these 24 output bits are fractional, meaning that from this point forward we must keep track of the fixed decimal point in each binary number – Vivado does not do this for us.

3.8.4 Autocorrelation Matrix Generation

The input to the MUSIC algorithm is the autocorrelation matrix for each array generated from the I and Q data over several samples. At a given time step n, we have 4 IQ pairs associated with each array, which we denote as

$$x_{i,n} = I_{i,n} + jQ_{i,n},\ i \in \{1, 2, 3, 4\}$$

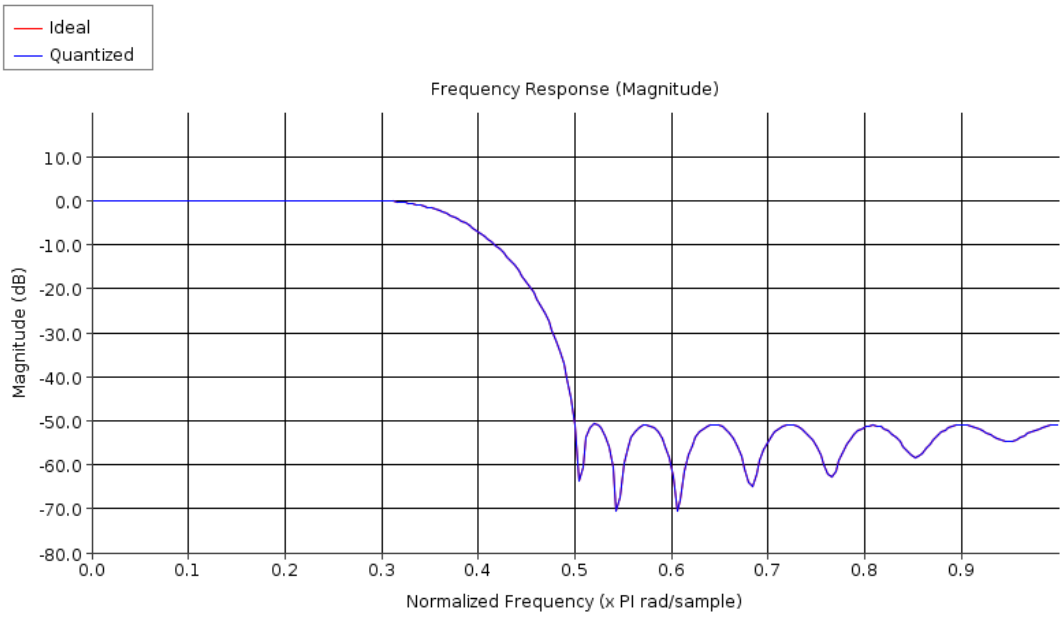

Figure 20: Frequency response of the implemented low-pass filter as generated by Vivado's FIR compiler IP.

The n^{th} autocorrelation sample is a 4x4 matrix denoted R_n, and has entries given by

$$\left(R_n\right)_{i,j} = x_i x_j^* \tag{1}$$

We then take the average over N samples to get

$$R = \frac{\sum_{n=1}^{N} R_n}{N}$$

One thing that is clear from Equation 1 is that R_n is Hermitian for all n, and so too is R. For our implementation, we take $N = 1024$. We compute the entries of R_n at each time step on the FPGA, and average continuously so that R is immediately available after time step 1024.

The complex conjugation in Equation 1 is achieved simply by taking the 2's complement negative of the Q signal, and the outer product is handled with Vivado's complex multiplier modules. By the Hermitian property of this matrix, we need only to compute the upper triangular entries.

When multiplying two entries, we need to store the output in a register with size at least the sum of the inputs' lengths. We also need to add up the fractional bits, and specify the input and output fractional bit numbers in the instantiation of the complex multiplier IPs. Thus, as we had 24-bit I and Q signals with 22 fractional bits, our output has real and imaginary parts each with at least 48 bits, 44 of which are fractional bits. As we have to add up these values later, we should have a bit more space to avoid overflow. We thus use 56 bits for each of the real and imaginary parts, still

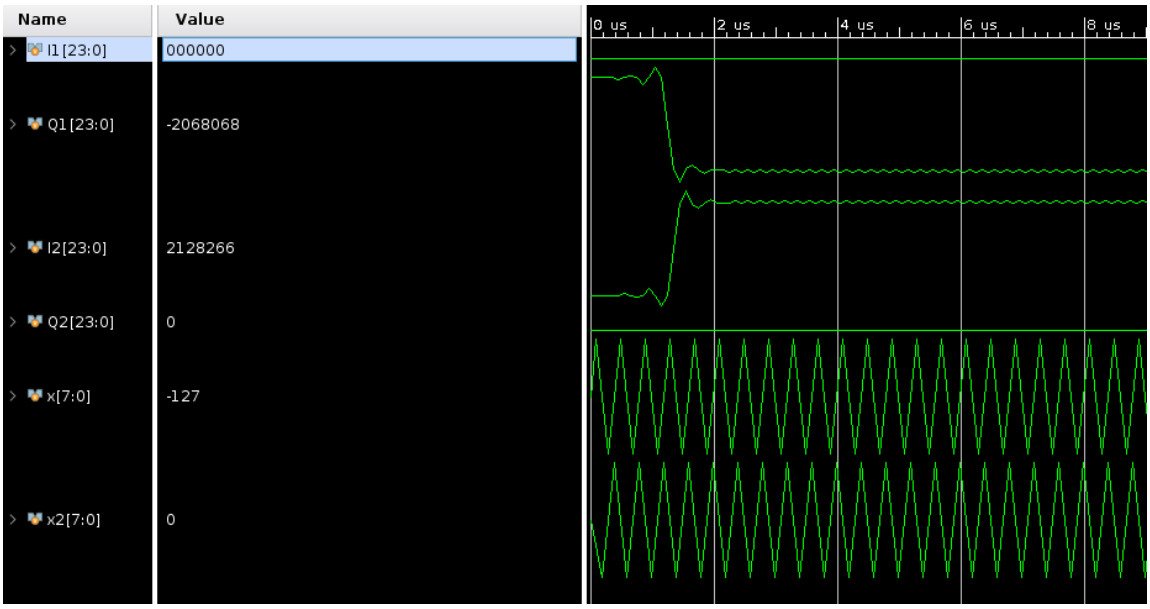

Figure 21: IQ Demodulator and LPF in Vivado simulation. Here input x corresponds to outputs I and Q, while input $x2$ corresponds to outputs $I2$ and $Q2$. The first few samples are strange due to unspecified values in the FIR filter registers at startup, but the filters converge to approximately constant output for constant input signal. The exact values are not themselves meaningful, on account of the fact that Vivado is not keeping track of the fixed decimal point in each of I and Q.

with 44 fractional bits. Thus, each entry in R_n requires a 112-bit register.

We then need to average the data over samples. We are taking 1024 samples per estimation of the autocorrelation matrix, so we must perform 1024 additions and then divide by 1024. However, should we simply add all of the data before dividing, we run the risk of overflow. We solve this problem by writing a module which performs division by 1024 on 2's complement numbers combinatorially; that is, this division takes minimal time.

The trick is to use the fact that $1024 = 2^{10}$, so dividing a positive number by 1024 is equivalent to a right bitshift by 10, replacing all shifted bits with zeros. For negative numbers, it is equivalent to a right bitshift by 10 as well, but the digits are merely replaced with ones.

With this method, we can divide each sample entry by 1024 first, quickly. Thus, when we add, we are adding much smaller numbers, and overflow will not occur. We tested this method on the data shown in Figure 21, and were able to estimate the autocorrelation matrix with less than 1% error in the spectral norm.

Each ADC sample takes 1 μs, and the demodulation, filtering and R_n generation at each time step takes less than 1 μs. Thus, every 1.024 ms we can generate an R matrix for both arrays.

3.8.5 Automatic Gain Control

While generating autocorrelation matrices, the FPGA must also control the automatic gain control units in each receiver chain. This control occurs independent of (and in parallel with) the digital signal processing.

For a single receiver chain, there are two VGAs and two corresponding DACs as described in the section on the RF frontend. The DACs are controlled through their own SPI protocol, and the analog value at their output controls the gain of the VGA.

To determine whether to raise or lower the gain, the FPGA computes the l^1 norm (also known as the taxicab norm or Manhattan norm) of the incoming signal over 50 samples. Recall that the l^1 norm of a discrete-time signal x with support of length 50 is given by

$$||x||_1 = \sum_{n=1}^{50} |x[n]|$$

This is computed quite easily in hardware – we subtract 128 from the ADC output (as in the IQ demodulator) and then take the absolute value of the signal and add it to an accumulating sum. The absolute value is implemented as a multiplexer which is addressed by the most significant bit of the data byte, outputting the byte itself if the MSB is 0 (where the byte is positive) or the byte's 2's complement negative otherwise. This l^1 norm is then compared to theoretically determined satisfactory bounds.

To determine these bounds, we considered 8-bit-sampled rail-to-rail 5.25 MHz signals with a range of possible phases in MATLAB and took their l^1 norms. The satisfactory range, within which we do not alter the gain, is determined by the minimum and maximum values seen in this simulation

across phases. If the l^1 norm is lower than this range, we raise the gain by increasing one DAC's input by 1. If it is higher than this range, we lower one DAC's input by 1. Figure 22 shows the thresholds determined by simulation.

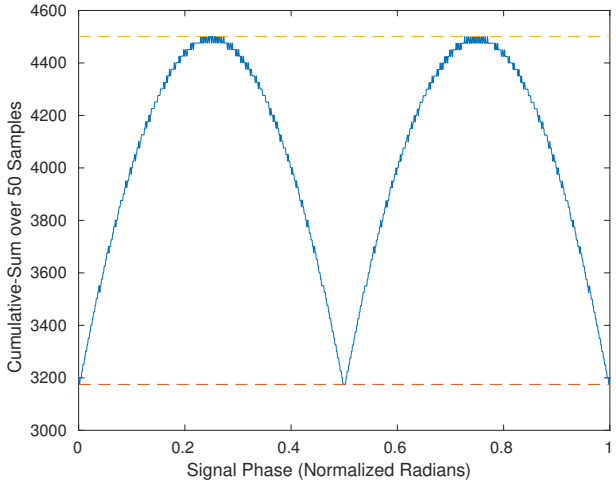

Figure 22: Cumulative-Sum Thresholds for AGC Algorithm.

When raising the gain of the VGAs, it is important that the second VGA (in terms of signal path) is updated first, as to ensure that it is not put into compression by the output of the first VGA. Thus, if we must raise the gain, we raise the second DAC input until it is at maximum, then the first until it is at maximum. At this point, if the l^1 norm is still too small, we do not change the DAC inputs. Conversely, if we must lower the gain, we first lower the input of the first DAC until it is at minimum and then the second. This process is illustrated in the pseudocode of Algorithm 1.

To actually implement Algorithm 1, however, we first need to determine the minimum and maximum DAC inputs in correspondence with our minimum and maximum VGA gain inputs. The VGA used is the AD8337 by Analog Devices, and in our frontend it operates in a range of 0 V to 5 V. The datasheet gives an analog gain input vs. gain curve for the VGA with a dual-ended supply from -2.5 V to 2.5 V, seen in Figure 23. Shifting the curve so that it is centered at 2.5V, we see that the analog input should go from about 1.9 V to 3.1 V for maximum gain swing.

We then need the DAC inputs that correspond to these voltages. The DAC used is Texas Instruments' DAC081S101, and its datasheet gives a mapping from digital input to analog output voltage replicated in Figure 24. For us, V_A is 5 V, so to find the input D that encodes the closest output to $0V < V_0 < 5V$, we must solve

$$V_0 = \frac{V_A D}{256}$$

for D, and then round to an integer. For the minimum output of 1.9 V, D should be about 97 (in

Algorithm 1: Automatic Gain Control

$S = 0$
for n **from** *1* **to** *50* **do**
    $S+ = |x[n]|$
end for
if $S <$ lower bound **then**
    if DAC2 input $<$ max **then**
        DAC2 input $++$
    else if DAC1 input $<$ max **then**
        DAC1 input $++$
    end if
else if $S >$ upper bound **then**
    if DAC1 input $>$ min **then**
        DAC1 input $--$
    else if DAC2 input $>$ min **then**
        DAC2 input $--$
    end if
end if

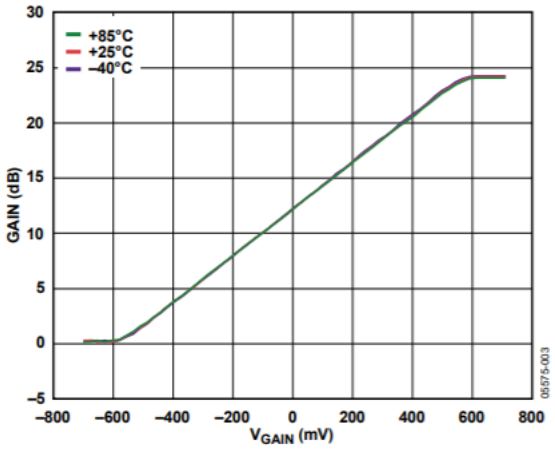

Figure 23: VGA gain input vs. gain as given by the datasheet for a dual-ended supply.

decimal), and for the maximum output of 3.1 V, D should be about 159 (in decimal).

We lastly need to understand the DAC SPI protocol. This is very similar to the ADC protocol, except that we now have to provide the input to the DAC as well as the clock and chip select signals. Each DAC has 3 inputs: the clock $SCLK$, the negated chip select $\overline{SYNC}$ and the digital input D_{IN}. The timing diagram for the update of the DAC is given in Figure 25.

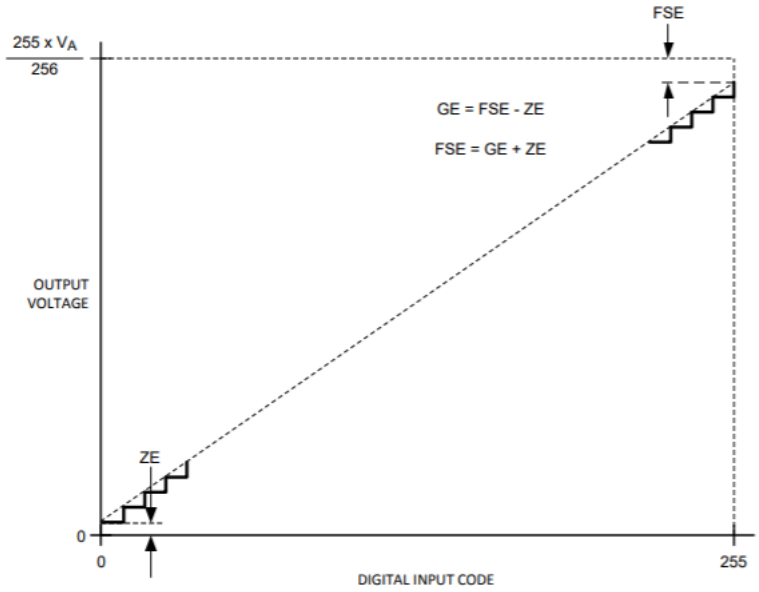

Figure 24: DAC input vs. analog output as given by the DAC's datasheet.

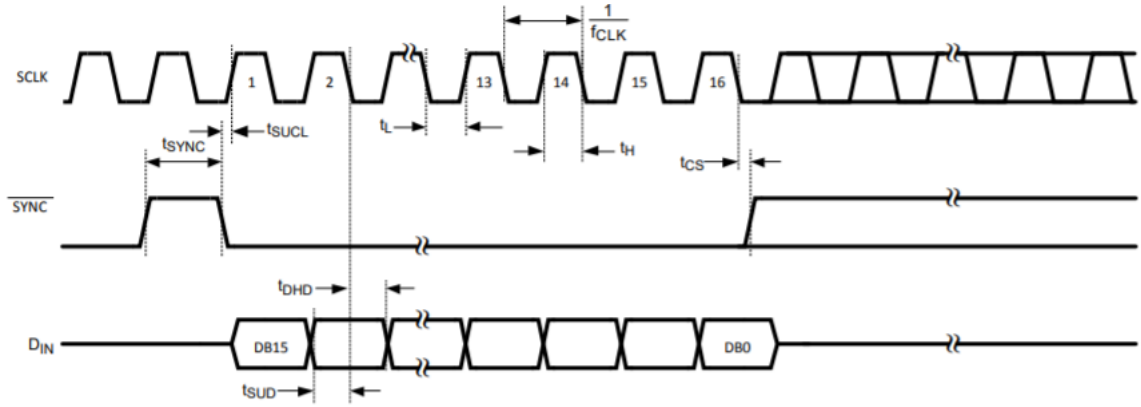

Figure 25: DAC timing diagram given by the datasheet.

The DAC does not require a separate startup sequence, as the first instance of the usual timing sequence after the chip is powered on acts as a startup. Both DACs in a chain can be controlled by a single clock and chip select, and simply have different inputs, as they do not interact with one another directly.

We compiled an IP which carries out Algorithm 1 for a single AGC unit, which also relies on the aforementioned data acquisition Verilog code. This IP prints to a serial console the perceived l^1

norm of the incoming signal, allowing us to determine if the norm generation and gain control are working as expected.

From Algorithm 1, it is clear that we are only altering the gain by a small amount every step. This is a very simple control scheme with a slow update rate relative to the sample period, but as the AGC units are updated every 50 samples, this update happens every 50 μs. The worst case amount of time it would take for the AGC to adapt is if it must adapt across the entire range, i.e. from minimum DAC input to maximum DAC input. This worst case will take about 2 ms. This is a very short time when compared to the frame-rate of the final program (at 60 frames per second, a frame is more than 16 ms long), and it is unlikely that an AGC will actually need to switch from maximum to minimum gain (or from minimum to maximum gain) in application. This upper bound is thus deemed acceptable.

We tested the AGC with our AGC prototype boards, and varied the amplitude of a connected function generator signal to see the gain adapt. The worst case, as described above, was tested and captured on an oscilloscope, and is shown in Figure 26. It should be noted that the test was performed with a 30 kHz signal rather than a 5.25 MHz signal, as high frequencies are harder to capture high-quality images of on our oscilloscope. However, the parts in question operate identically (according to their datasheets) at 5.25 MHz.

3.8.6 Complete IP and GPIO Constraints

We have a master Verilog file which performs AGC for four receiver chains and creates an autocorrelation matrix for a single 2x2 antenna array. We wish to turn this into an IP so that we can send autocorrelation matrix values to the host computer.

To do this we use the Vivado IP generator to create an Advanced Extensible Interface 4 (AXI4) IP. AXI4 is a protocol which stores specified updating PL-side values in buffers and makes them available to the PS. Beyond this, it also allows for inputs and outputs to the IP to be constrained to physical pins on the FPGA.

Each AXI4 buffer is 32 bits long, and we wish to make available each of the upper triangular autocorrelation matrix entries and a single bit which represents that a new autocorrelation matrix is ready. Each of the entries is 112 bits, and we need to pass 10 of them to the PS, so along with the bit to show the matrix is ready we need to pass 1121 bits to the PS. This means we need 36 buffers.

We also need an input for each ADC. All four ADCs can be controlled with a single $SCLK$ and $\overline{CS}$, and all eight DACs can be controlled by a single $SCLK$ and $\overline{SYNC}$. Each DAC, however, requires its own digital input pin. In total, this is 12 GPIO pins required for each IP.

We then, of course, need two such IPs so that we can get the autocorrelation matrices for both arrays. This gives a total of 24 required GPIO pins. The FPGA allows us to access exactly 24 GPIO pins with our own AXI4 IPs.

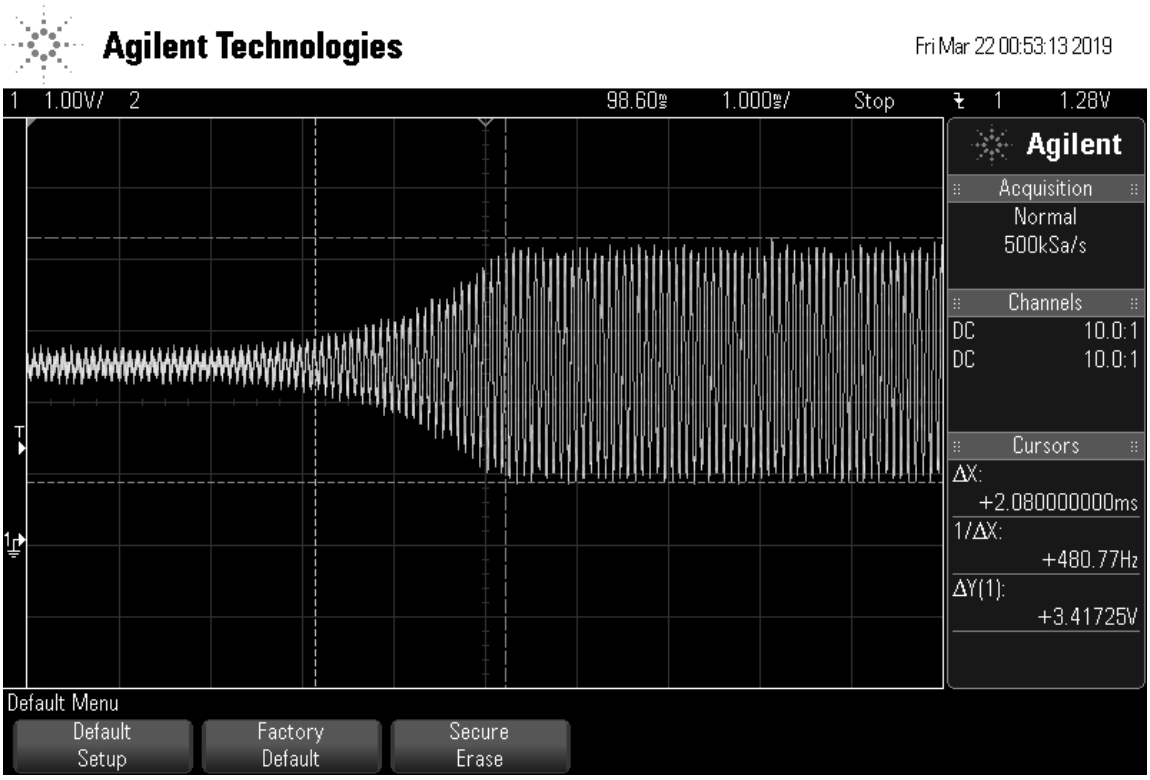

Figure 26: Demonstrated operation of a single AGC unit, measured at the output of the second VGA. This is worst-case gain adjustment occurring in approximately 2 ms as predicted by our theoretical bound. This is generated by inputing a DC 3.3 V signal into the FPGA as the ADC input, causing it to think that it should operate the AGCs at minimum gain. Then, at the time marked by the first horizontal marker, the true ADC input is fed into the FPGA, and it begins to adapt to the required maximum gain.

3.8.7 Block Design

Vivado requires the creation of a high-level block diagram to detail the interface between the PS, IPs and GPIO pins. Figure 27 shows the block diagram for our final implemented system, with two autocorrelation matrix generation IPs. Also included in the block diagram is the ZYNQ7, which is the PS, and a Processor System Reset and AXI Interconnect block which serve as overhead for the connection between the PL and PS. The inputs and outputs aside from "CLK" are constrained to GPIO pins through a constraint file. "CLK" is constrained to the on-board 100 MHz clock through this same constraint file. The ZYNQ7, Interconnect and Reset block are from Vivado's IP catalog, and were not altered by us.

Once a block diagram is created, it must pass a Design Rules Check (DRC) at three levels: behavioral, synthesis and implementation. The behavioral DRC functions similarly to a compiler, checking that the written code does not have syntax errors. The synthesis DRC checks that the

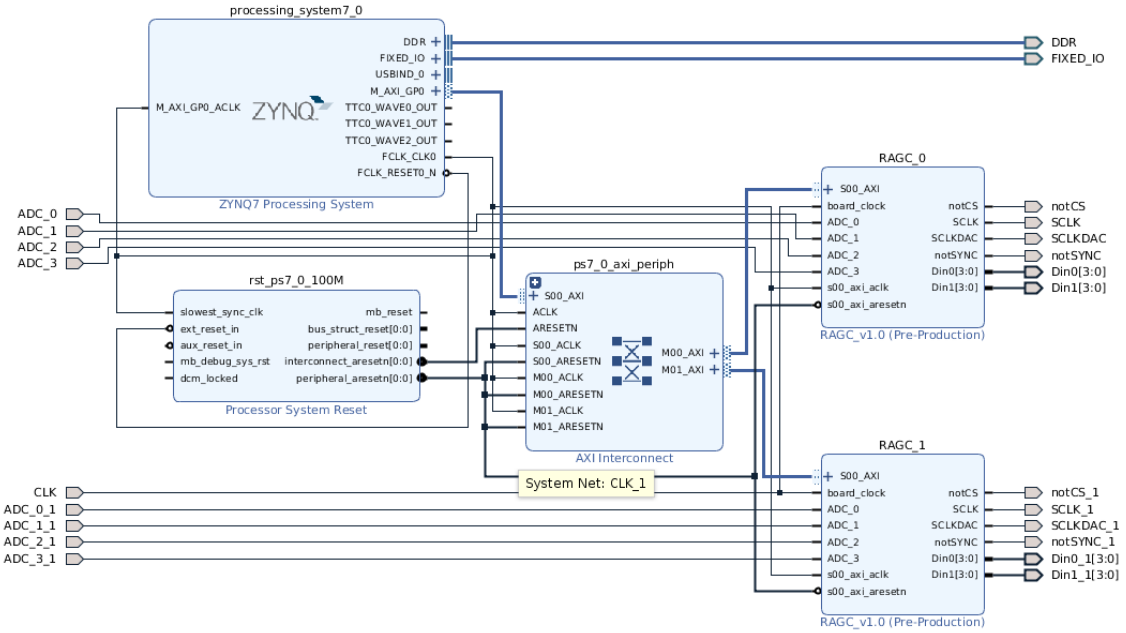

Figure 27: Vivado block design, in which the blocks labeled RAGC_0 and RAGC_1 are our IPs.

written code can actually be implemented in hardware, ensuring that no signal is driven by hanging inputs or by multiple inputs. The implementation DRC checks that the code can actually be implemented on the specified FPGA.

Running implementation also generates a number of figures regarding the utilization of the FPGA. We are concerned mostly with use of space on the FPGA and power usage so that we can consider the plausibility of scaling up this design for larger arrays.

The power usage is given by Vivado as in Figure 28. Here DSP means Digital Signal Processors, which are used in the FIR filter and complex multiplier IPs from Vivado's IP catalog. MMCM stands for Mixed-Mode Clock Manager, and these are required for the clock dividers which control the timing of the SPI protocols.

It is clear that the majority of power consumption is from the PS, which is to be expected. As the power consumed by the PS does not scale with the number of operations being performed on the PL, the use of larger arrays would not impact this 1.529 W figure. The next largest consumer is the MMCM. As we increase the number of antennas, we do increase the number of SPI inputs and outputs, but as the ADCs and DACs are all controlled by the same clocks, the number of MMCMs does not increase. Thus, the 0.436 W figure would not be impacted by scale-up either. Logic, signals and DSP all would necessarily increase as a result of scale-up, but they together only make up 17% of the total power usage.

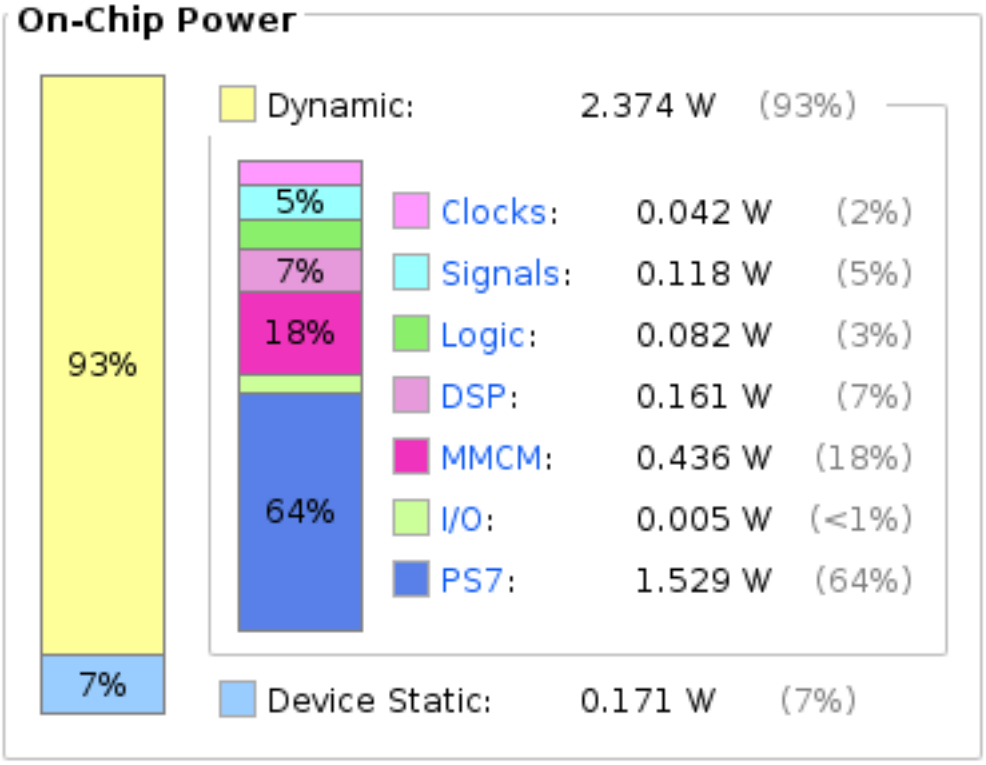

Figure 28: Vivado's estimated power budget of the FPGA.

The estimated PL current draws are given in the ZedBoard Hardware User's Guide [22], which say 1.2 A can be drawn at 1 V, 300 mA can be drawn at 1.8 V and 50 mA can be drawn at 3.3 V. This totals to 1.905 W of estimated PL power draw, which is far lower than what is seen in our current design. Thus, we can assume a scale-up would not be detrimental to our power budget.

We are similarly concerned with the FPGA's spacial utilization. To determine the use of PL cells, we can look at the "Device" image created by Vivado implementation as shown in Figure 29. It can be seen visually that less than half of the available PL cells are used. Figure 30 shows the same "Device" image with routing overlayed, showing that most routing occurs from the PL to the PS.

As for use of specific hardware elements, such as Look-Up Tables (LUTs), DSPs, Flip-Flops (FFs), MMCMs and Random Access Memory (RAM), we can look at the generated utilization summary shown in Figure 31.

From this figure, LUTs, RAM, FFs and IO aren't a significant problem for our design when we consider scaling up. MMCM is already at 100% due to the fact that only four are available. The MMCM units are used in each autocorrelation matrix generator to control the SPI protocols of the DAC and of the ADC. There are thereby two in each IP. Scaling up to larger arrays does not

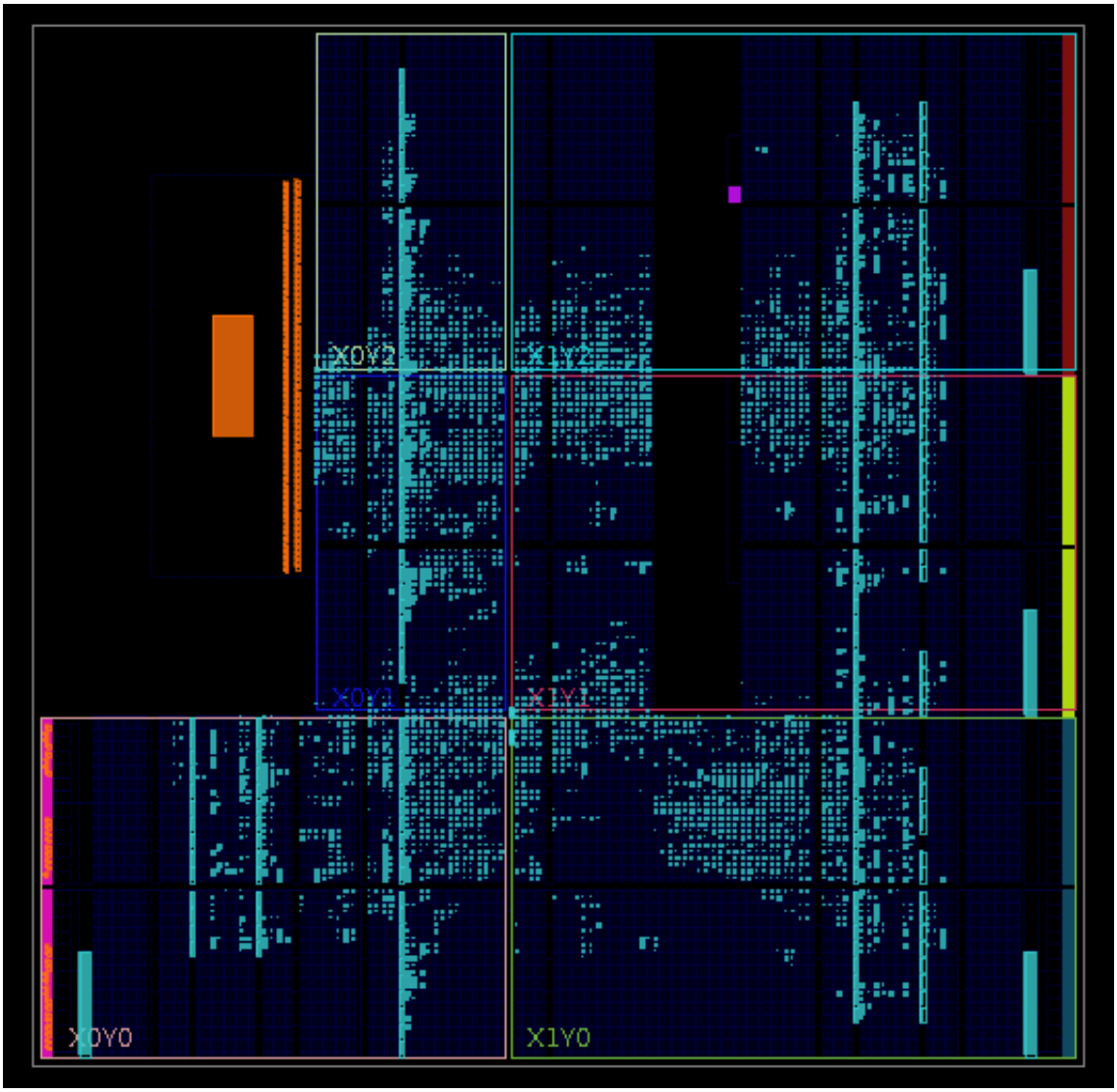

Figure 29: FPGA Device image. Here the light blue squares are used PL cells, the dark blue squares are unused. Orange represents the PS, and pink represents GPIO.

increase the number of MMCM instances, as each DAC and each ADC is subject to the same clocks.

The only significant problems are thereby DSP units, which are used in the FIR filter and the complex multiplier IPs generated by Vivado. We can reduce usage of these on a larger scale by writing our own multipliers or FIR filters, with the understanding that more cells of the PL would be used in exchange for freeing up DSP space. This is not a problem, as there are many free LUTs and PL cells.

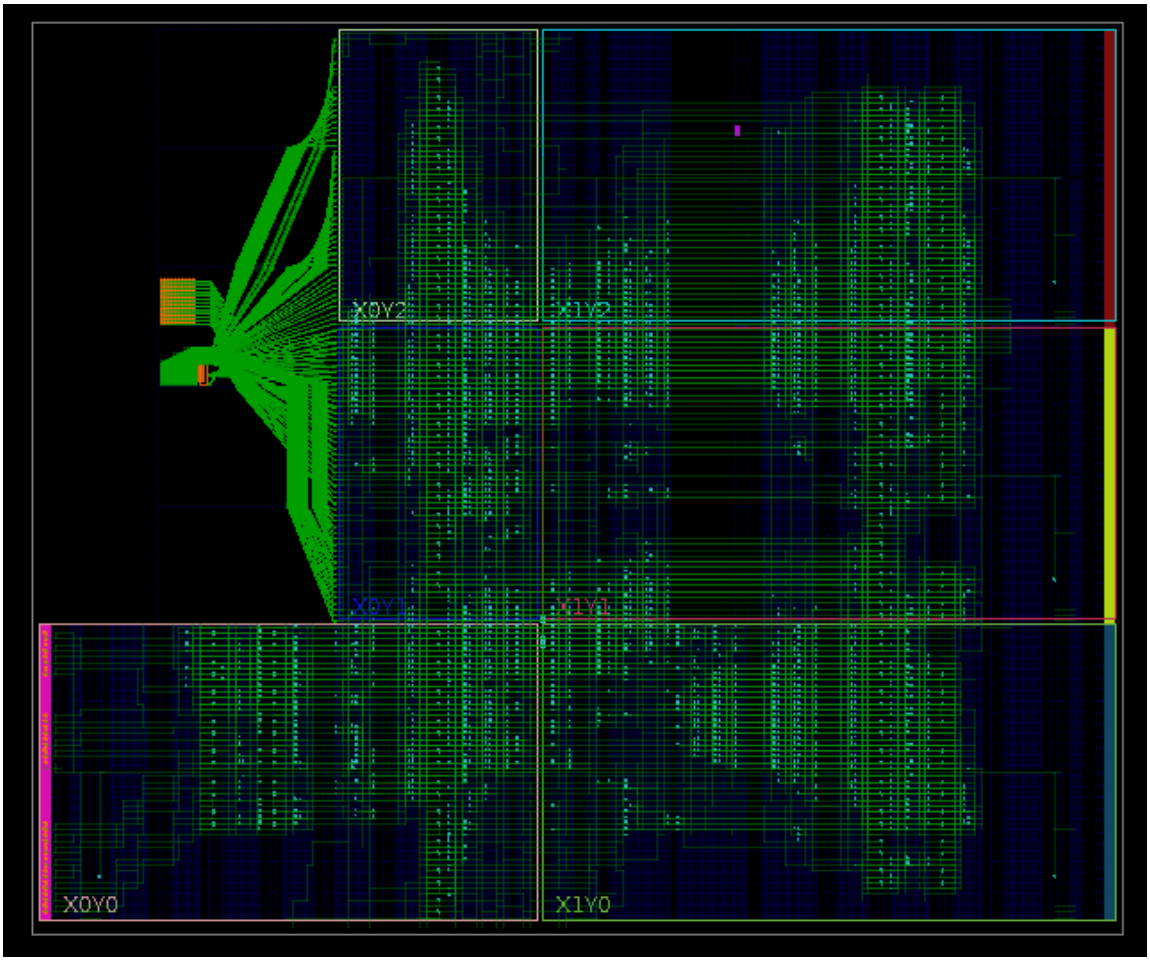

Figure 30: FPGA Device image with routing overlayed.

Resource	Utilization	Available	Utilization %
LUT	7256	53200	13.64
LUTRAM	1376	17400	7.91
FF	12902	106400	12.13
DSP	184	220	83.64
IO	33	200	16.50
MMCM	4	4	100.00

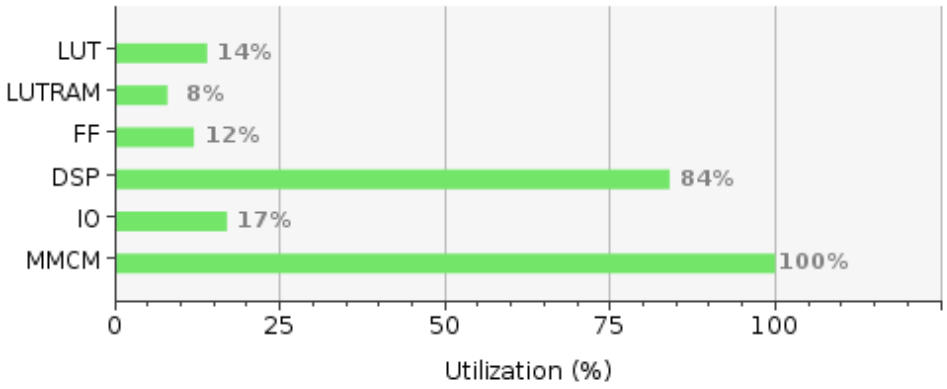

Figure 31: Utilization breakdown by specific hardware component as generated by Vivado implementation.

3.8.8 Serial Communication with Host Computer

We write C++ code in SDK which reads the AXI4 buffers and sends the autocorrelation matrix entries across UART to the host computer. The process is simple – when we see from the AXI4 buffers that new matrices are ready, we reconstruct the fixed-point signed complex entries of the matrices and send them across UART using "std::cout".

Reading the matrix entries from the IP, they are 112 bits long where bits 0 - 55 represent the imaginary component and bits 56 - 112 represent the real component of the autocorrelation value. The string is broken up into its real and imaginary components. The two substrings are both 2's complement numbers. These 2's complement numbers are then converted to a floating point number (i.e. float). The floats are then divided by 2^{43} as 43 of the 56 bits for both the real and imaginary portions represent fractional bits.

Below is an example of converting a 56-bit number from binary to float with 43 fractional bits:
Starting binary number:

10010100010001011110101010001000100100100101111010111011

Find the 2s complement of the starting number:

01101011101110100001010101110111011011010000101000101

Convert the number to decimal: -30322423868793157
Divide by 2^{43}: 3447.2605

After this division, the binary number has finished being converted into a float.

We can send these floats in any format so long as we end each transmission with a carriage return and new line character. We then flash the FPGA's system memory so that the FPGA can operate without being connected to a computer running Xilinx SDK. The associated C++ code can be found in Section 12 Appendix.

3.9 Unity

Unity is used to visualize the location of the RF transmitter as estimated by R3CAP. We include an object which represents the physical board on which the frontends and FPGA are held, upon which there are two squares which represent the position of the frontends. These objects are not movable, and exist for the sake of interpreting our visualized locations. These objects can be seen in Figure 32.

The estimated location of the transmitter is represented by the location of an orange sphere as shown in Figure 33. The tag is small compared to the board, as it is meant to be the position of a point source. It is orange so that it stands out against the grey ground.

A simple camera control mechanism has been implemented as well, which allows users to use the mouse and arrow keys to navigate about. This is necessary to get a good point of view for various motion capture applications. Both the camera control and the ball location update are written as C♯ scripts, which update the game objects' locations at every frame. Figure 34 shows a sample localization from an appropriate angle.

Figure 32: Unity objects to represent the board

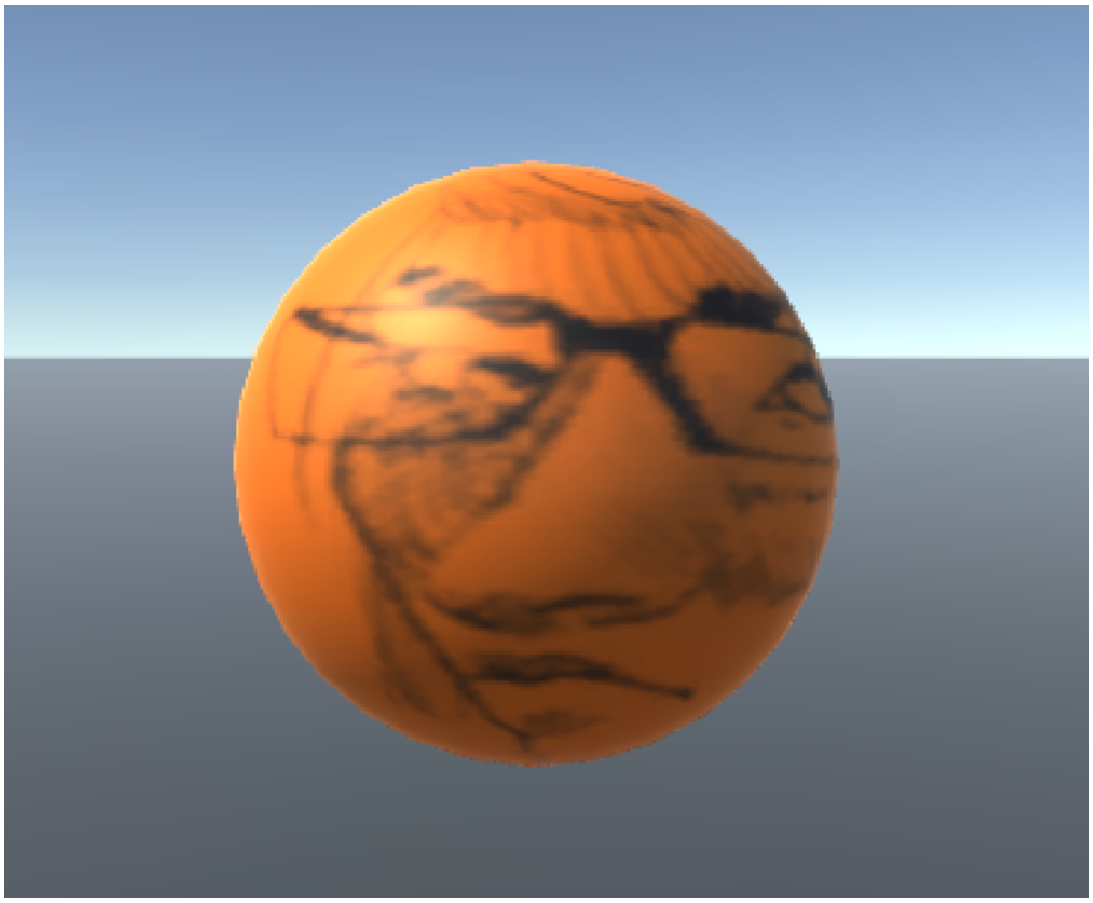

Figure 33: Ball which represents the estimated position of the RF transmitter.

The ball location update is the most important and challenging piece of this Unity program. First, the host computer must know to look for serial messages from the FPGA containing the autocor-

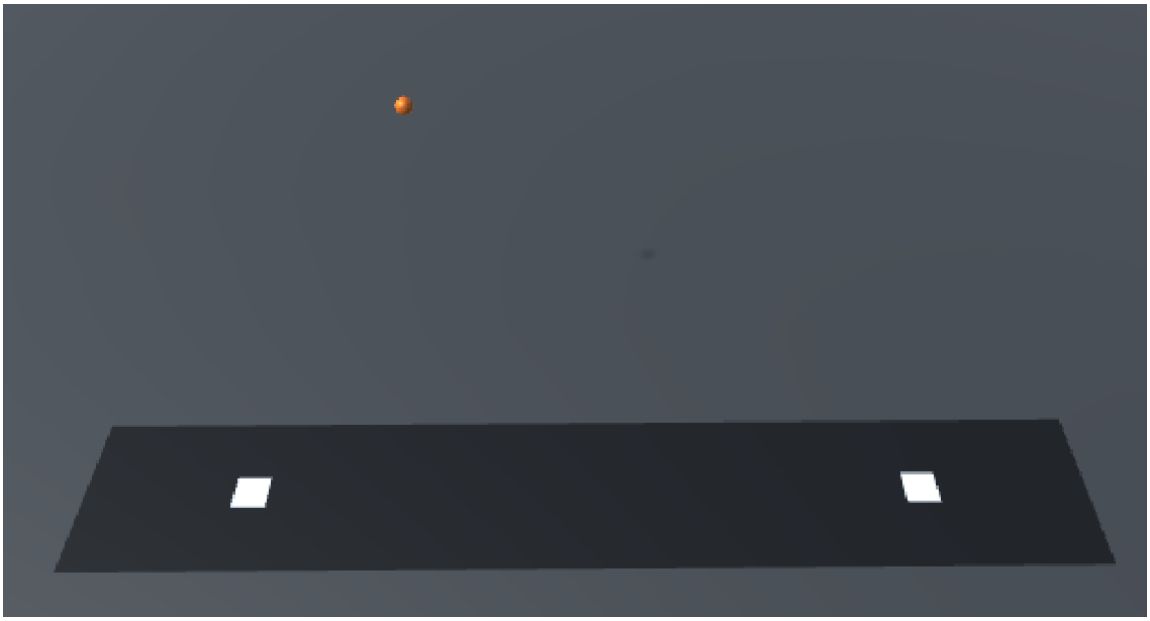

Figure 34: An example of what the Unity UI might look like when a source is actually being localized.

relation matrices. For serial communication, we use the Ardity Unity package. Ardity is designed to interface Arduinos with Unity, but it provides a general protocol for reading serial messages into Unity. When the FPGA sends a new string, a message listener game object updates a public string variable to be the received message.

The tag's movement controller script can then parse this string (knowing in what format it was sent from the FPGA) to extract the autocorrelation matrix entries as floating point numbers. With these entries, the movement controller can call a function which computes the corresponding locations.

This function is implemented using a C++ Dynamic Link Library (DLL), which allows us to call C++ functions from within a Unity C♯ script. All implemented C++ and C♯ code for the host computer can be found in Section 13 Appendix. The details of this C++ code follow.

3.10 C++ DLL

The job of the C++ DLL is to take the incoming autocorrelation matrix values from the FPGA and use them to compute the location of the source. The computed location is then displayed in Unity. The process of converting the autocorrelation matrix values to a location occurs in the following order:

1. Create autocorrelation matrices from the autocorrelation values.

2. Compute the MUSIC spectra from the autocorrelation matrices.

3. Find the peaks of the spectra to identify angles of arrival.

4. Use the AOAs from the two spectra to localize the tags.

3.10.1 Creating the Autocorrelation Matrices From Autocorrelation Values

The C++ code receives the autocorrelation values and then organizes them in the order seen below with the real and imaginary parts split into two separate float arrays:

$$r_{1,1}, r_{1,2}, r_{1,3}, r_{1,4}, r_{2,2}, r_{2,3}, r_{2,4}, r_{3,3}, r_{3,4}, r_{4,4} \tag{2}$$

The real and imaginary float arrays containing the autocorrelation values are then used to create autocorrelation matrices. To do this, the Eigen.h and complex.h C++ libraries were used. An empty 4x4 matrix is instantiated. The upper triangular entries are filled in first with the positions corresponding to the indices in Eq. 2. After filling the upper triangular portion of the matrix, the entries in the lower triangular portion, not including the diagonal, are filled in. For the lower triangular entries $r_{i,j}$ is filled in with $r^*_{j,i}$, since the autocorrelation matrix is Hermitian. For example $r_{4,1} = r^*_{1,4}$. After doing this the autocorrelation matrix R is ready to be used to calculate the MUSIC spectrum.

3.10.2 Computing the MUSIC Spectrum

The first step in calculating the MUSIC spectrum, given the autocorrelation matrix, is to compute the subspace matrix G. G is computed by first finding the eigenvalues and corresponding eigenvectors of R. Suppose there are n antennas in the array and k sources in the environment that are being tracked, where $0 < k < n$. We take the $n-k$ eigenvectors corresponding to the $n-k$ smallest eigenvalues and use them as the columns of a matrix T of size $n \times (n-k)$. Then $G = TT^*$, where T^* is the adjoint of T. After computing the subspace matrix, the MUSIC algorithm is performed. The MUSIC spectrum is computed over a grid of polar angles $\theta \in (-180, 180]$, and azimuthal angles $\phi \in [0, 90]$. The grid is discretized to 200 points for both θ and ϕ (200^2 points in total). For each (θ, ϕ) pair, the wavenumber vector $\vec{k}$ is computed. The wavenumber vector encodes the wavelength λ of our RF signal and the angle-of-arrival vector $\vec{a}$ (Eq. 3) corresponding to the (θ, ϕ) pair. Eq. 4 shows how $\vec{k}$ is computed. Then the steering vector $\vec{s}$ is computed using the wavenumber vector and the location vector $\vec{d}$. The location vector is just the (x, y, z) coordinate of the antennas in an array shifted so that the centroid of the array lies at the origin. This way the steering vector is computed relative to the center of the array. Eq. 5 shows how the steering vector is computed. With the values of G and $\vec{s}$ the value of the MUSIC spectrum M at (θ, ϕ) can be computed using Eq. 6, where $\vec{s}^H$ refers to the Hermitian transpose of $\vec{s}$. We do this for all the grid points to fully compute the MUSIC spectrum. The meshgrid of θ and ϕ values is also stored. An example of a MUSIC spectrum can be seen in Figure 35.

$$\vec{a} = (sin(\phi)cos(\theta), sin(\phi)sin(\theta), cos(\phi)) \tag{3}$$

$$\vec{k} = \frac{2\pi}{\lambda}\vec{a} \tag{4}$$

$$\vec{s} = \frac{e^{-j\vec{k}\cdot\vec{d}}}{\sqrt{n}} \tag{5}$$

$$M(\theta, \phi) = 20log_{10}\Big|\frac{1}{(\vec{s}^H G \vec{s})}\Big| \tag{6}$$

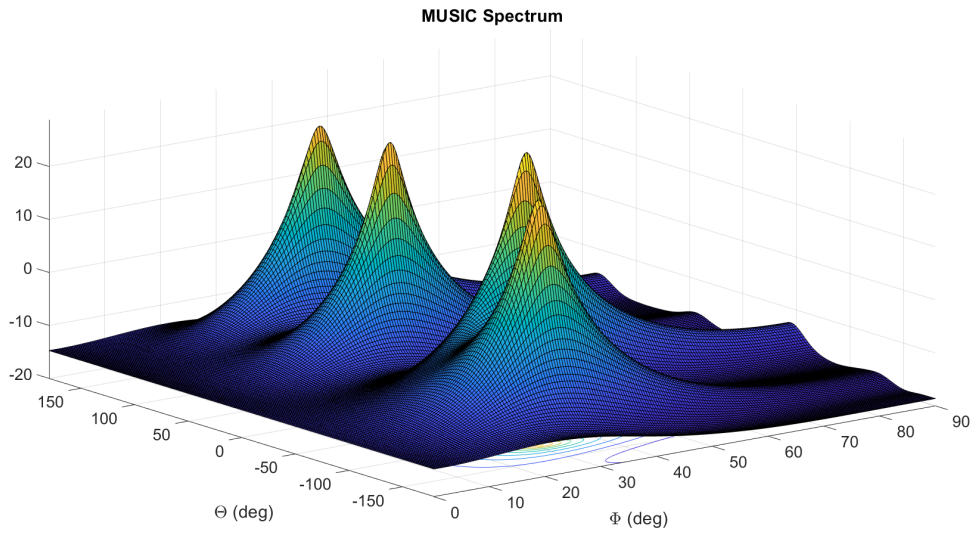

Figure 35: A MUSIC spectrum with 4 peaks, each peak corresponds to a unique source.

3.10.3 Peak Finding to Determine AOAs

A peak finding algorithm is run on the MUSIC spectrum matrix M. If there is only one source to localize, then peak finding is equivalent to finding the maximum entry in the M matrix and its associated angles. For the general case where there are multiple sources, the value of the MUSIC spectrum at (θ_i, ϕ_j), for$0 \leq i, j \leq 199$, is compared against the the value of the MUSIC spectrum at the four adjacent values $(\theta_{i\pm1}, \phi_{j\pm1})$. If $M(\theta_i, \phi_j) \geq M(\theta_{i\pm1}, \phi_{j\pm1})$, i.e. is greater than all of its surrounding values, then there is a peak at (θ_i, ϕ_j). All of the peaks are stored in a list. For $i = 0$ ($\theta = -180°$) and for $i = 199$ ($\theta = 180°$) assuming a 200 point resolution on θ, there is a wrap around corresponding to the physical meaning of the polar angle. This is not necessary for ϕ as only the northern hemisphere of possible AOAs is being considered. Consequently, when j = 0 or 199 then $M(\theta_i, \phi_j)$ is only compared against three other values. After all of the peaks are found, they are sorted in terms of descending values of $M(\theta, \phi)$. The largest value and corresponding AOA is considered an actual peak. The next value on the list is then considered. If the taxicab norm between the AOAs of the first peak and the current peak is less than some threshold value, then the current peak is not considered to be an actual peak. If the taxicab norm is greater than the threshold, the current peak is considered an actual peak. This thresholding is done to ensure that the peaks that correspond to the same source are only counted once as an actual peak. For finding subsequent peaks, we take the next value on the peaks list and see if the taxicab norm is greater than the threshold when comparing it to all the actual peaks that were already located. If the current peak passes the threshold for all the actual peaks already on the list, it is also added to

the list of actual peaks. We keep going through the list of peaks until the number of actual peaks found is equal to the number of sources that are being localized.

3.10.4 AOAs to Location

After finding the AOAs for each of the k sources for each of the two antenna arrays, localization of the sources can occur. Let A_1 be the set of the k AOAs corresponding to array 1 and A_2 be the analogous set for array 2. To localize the source, we take one AOA from A_1 and one from A_2. For each of these AOAs, we define the direction vectors $\vec{d_1}$ and $\vec{d_2}$ corresponding to the AOA from A_1 and A_2 respectively. The direction vectors are computed with Eq. 7. Let $\vec{c_1}$ and $\vec{c_2}$ denote the centers of each array. Note that each of the direction vectors and center vectors uniquely defines a line that goes through the center of the array. To localize the source, we need to find the point, $\vec{p_1}$, on line 1 and the point, $\vec{p_2}$, on line 2 that minimize the distance between the two lines. Using the direction vectors, we calculate calculate the normal vectors $\vec{n_i}$ as defined in Eq. 8 for $i = 1, 2$.

$$\vec{d_i} = (cos(\theta_i), sin(\theta_i), arctan(\phi_i)) \text{ for } i = 1, 2 \tag{7}$$

$$\begin{aligned} \vec{n_1} &= \vec{d_1} \times \vec{d_2} \times \vec{d_1} \\ \vec{n_2} &= \vec{d_2} \times \vec{d_1} \times \vec{d_2} \end{aligned} \tag{8}$$

Given these normal vectors, $\vec{p_1}$ and $\vec{p_2}$ can be calculated with Eq. 9 and the distance between these two points is given by Eq. 10.

$$\begin{aligned} \vec{p_1} &= \vec{c_1} + \frac{(\vec{c_2} - \vec{c_1}) \cdot \vec{n_2}}{\vec{d_1} \cdot \vec{n_2}} \vec{d_1} \\ \vec{p_2} &= \vec{c_2} + \frac{(\vec{c_1} - \vec{c_2}) \cdot \vec{n_1}}{\vec{d_2} \cdot \vec{n_1}} \vec{d_2} \end{aligned} \tag{9}$$

$$dist = \left| \frac{(\vec{n_1} \times \vec{n_2}) \cdot (\vec{c_2} - \vec{c_1})}{|\vec{n_1} \times \vec{n_2}|} \right| \tag{10}$$

Now with the two points that minimize the distance between the two lines, the location of the source can be approximated as $\vec{loc} = \frac{\vec{p_1} + \vec{p_2}}{2}$.

If there is only one source to localize, then the above procedure will work without any issues. With more sources, there arises a problem with associating an AOA for array 1 to its corresponding AOA in array 2. We propose the following solution: pick an AOA from A_1 and find the distance between the chosen AOA from array 1 to all the AOAs in A_2. Match the AOA from array 1 to the AOA from array 2 that minimizes the distance as defined in Eq. 10. Then remove these two AOAs from there respective sets. Pick another AOA from A_1 and match to the distance minimizing AOA from the AOAs that remain in A_2. Continue doing this until all the sources are located. This method is not optimal in locating all the tags – it would be optimal to compare the total distance between the points for all permutations of AOAs from array 1 and array 2, and then choose the permutation that minimizes the distance. However, this optimal method has complexity $\mathcal{O}(n!)$ while our proposed method has complexity $\mathcal{O}(n^2)$ where n is the number of sources. After doing this all the sources are localized. The locations are then passed to Unity where they are displayed.

3.11 Physical Implementation: PLANC

The Poly Localization Array Network Carrier (PLANC) can be seen in Figure 36. PLANC consists of two RF-frontends and an FPGA. The two RF-frontends are separated one meter apart. The FPGA is located in the middle of the board to maintain an equal signal path length to each of the RF-frontends. The RF-frontend PCB without any components can be seen in Figure 37. Each PCB contains four receiver chains. Each chain starts in a corner, works its way to the center, makes a 45 degree turn and outputs at an edge of the board. The PCB also contains the necessary circuitry to provide power to all the components and the circuitry necessary to split the oscillator outputs to all the mixers on the board. The final soldered PCB can be seen in Figure 38. The two RF-frontends and the FPGA were screwed down to the board to ensure the distances between the arrays does not change.

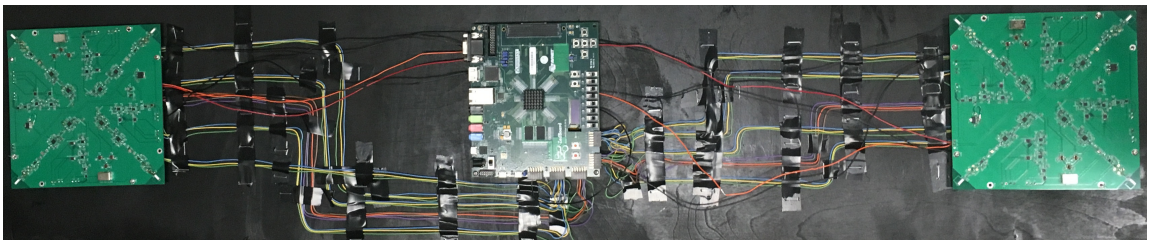

Figure 36: PLANC consists of two RF front ends and an FPGA.

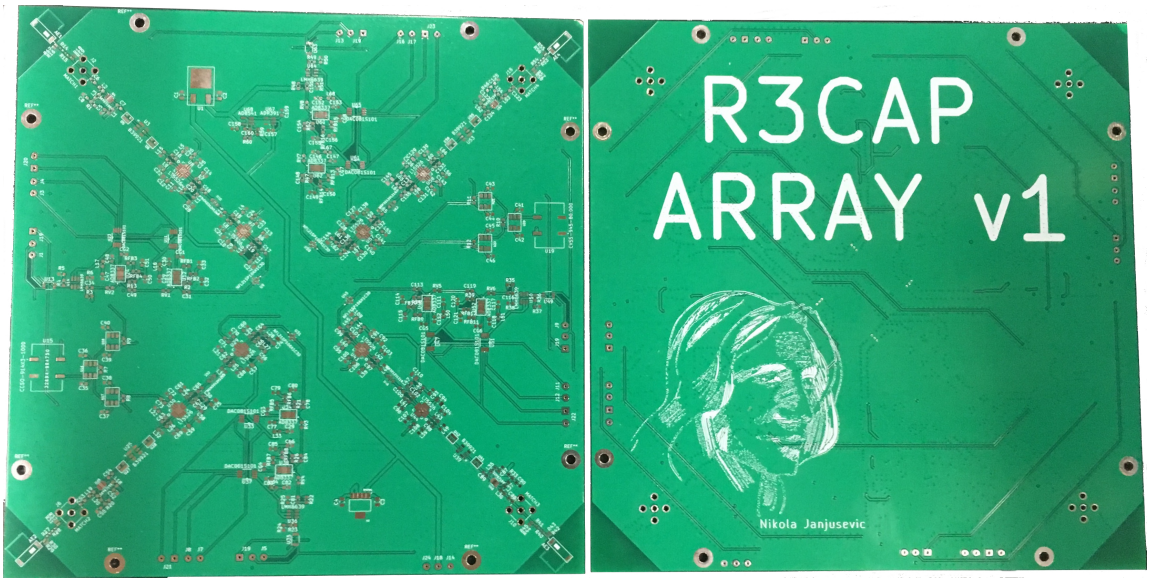

Figure 37: RF-frontend PCB without any components. Left: front of the PCB. Right: back of the PCB.

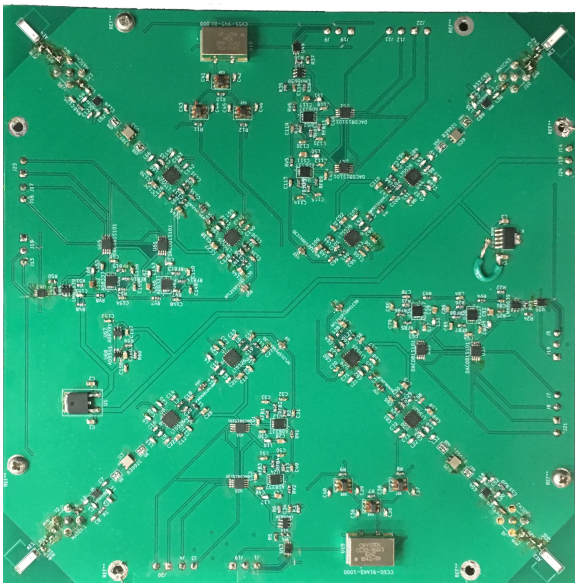

Figure 38: RF-frontend PCB with four receiver chains fully soldered.

4 Functionality

4.1 Observed Capabilities

Our device is able to perform angle-of-arrival estimation from both of the two arrays, as well as localization of a high-power 915.25 MHz source. Furthermore, the source can be localized regardless of line-of-sight between either array. Figure 34 shows a sample snapshot in the localization of a transmitter with total distance error of about 10 cm. Our setup for experiments of this type is shown in Figure 39.

For localization experiments with a fixed-position source, error in the tens of centimeters is seen. This is as predicted by our MATLAB simulations for 2x2 arrays. The orange ball is seen to jump about a 10 cm ball (volume of space) centered around the true location, which remains within our error bounds. The RF source in our experiments is an SDR which is programmed to transmit constantly at 915.25 MHz. We use GnuRadio to program the device through ethernet.

Human motion near the RFFEs can cause substantial reflections and increase the error. Different environments may substantially impact our error, but the SDR's requirement of ethernet and the FPGA's and SDR's requirement of wall power restricts our current environment testing capabilities. The creation of handheld transmitters would mitigate these problems.

Motion of the SDR is tracked, but the error is too high at each snapshot to see smooth motion. The location error at each snapshot is most importantly impacted by the number of antennas in the array according to our simulations.

The power of the transmitter currently has to be quite high – around 25 dBm – for localization at about 1 m from each RFFE. The power and distance at which AOA can be performed successfully are different for each array, however. This is a result of matching. The matching process can be repeated at each antenna to ensure the best possible matching, but this will require more passive components and time.

Figure 39: Our experimental setup. On the shelf is the USRP N200 SDR. Below is the R3CAP PLANC, where the FPGA is powered by wall and an independent power supply (right) provides the RFFEs with 3.3 V and 5 V rail voltages.

We have thus achieved our goal of proof-of-concept localization, but it is clear that future work will be necessary for R3CAP to be a commercially viable motion capture device.

4.2 Known Sources of Error

The following list contains all known sources of error within our implemented design, all of which can be mitigated in future implementations.

- Antenna mismatching: can add phase noise and decrease signal amplitude, lowering Signal-to-Noise Ratio (SNR)
- Component phase delay: If components in each RF chain have different phase responses, the phase data necessary for the MUSIC algorithm could be corrupted
- Op amp biasing: Our digital backend requires that the input signal is centered at mid-rail to apply AGC and to generate autocorrelation matrices, and resistor values may cause the bias voltage to be offset from mid-rail.
- Electrical noise due to mixed signal design: Due to analog and digital components sharing a ground and power rail with minimal shielding, there is non-negligible noise caused by the analog and digital coupling. A PCB could be designed with this fact in mind.
- Quantization: quantization always leads to error, although our simulations show that 8-bit quantization achieves acceptable accuracy.

5 Future Work

5.1 RF Frontend

5.1.1 Scaling Up

From our simulations, scaling up the arrays to 3x3 would give accuracy more in-line with our original motion tracking goals. However, future work in scaling should also include an investigation into sub-quarter wavelength arrays or dense-arrays. Of course, this work is reliant on creating a cheaper and more compact receiver design.
Due to the time constraints of this project, a time-money trade-off taken advantage of to the detriment of the design of the project. For example, with a smarter LO configuration that takes advantage of a PLL one could replace the two-IF stage design with a single IF-stage, which would not only greatly decrease the cost of a receiver but also save space on a PCB and reduce power consumption. Future work could also involving looking at alternative AGC components to reduce the space and power consumption of those units. Another interesting area of future work is the exploration of using frequencies other than 915 MHz. While a higher frequency would allow for smaller dimension arrays it would also bring a significant price increase in components and PCB (a better PCB substrate, likely Rodgers, would have to be used). A lower frequency could decrease the cost of components and allow for less line-of-sight dependency. In conjunction with dense array configurations, this approach could prove very fruit-full and may be an exciting area of future work.

5.1.2 Interference Cancellation for use with RFID Tags

As the implemented design currently stands, an RF transmitter is required for localization (which for us has been an SDR). Hence, one is required to have several RF transmitters (at the right frequency!) on hand to take advantage of device's multiple source motion capture capabilities. One way to mitigate this problem is to modify the frontend to include a transmitter and interference cancellation so that one could use RFID tags (at the right frequency!) for localization. This would severely complicate the receiver design and only adds the benefit of smaller objects for localization. The authors recommend instead exploring the design and implementation of mobile transmitters (see Section 5.5).

5.2 FPGA

5.2.1 Scaling Up

As described in the Implemented Design section, the main concern in scaling the number of receivers up is DSP usage. Our current design makes use of built-in IPs for complex multiplication and FIR filters, but these operations could easily be written so as to not make use of DSPs. If we built our own FIR filters or multipliers, they would incur similar delay but would likely be space-inefficient, as they would not make use of the specialized DSPs. Although it seems that we have LUT and PL cell space to make up for this, it is worth considering the scenario in which we would have to make our program more space-efficient.

The easiest way to do this would be to curtail the "data explosion" incurred by complex multipliers and FIR filters. As most of these bits are fractional, we can simply drop a number of them at the cost of some precision. Tests would be required to determine how far we could reduce this the precision before it negatively affects our location accuracy.

5.2.2 Computation

Theoretically, the process of localization could be sped up by moving all computations currently performed on the Host computer to the FPGA. However, we currently see quick (with respect to the Unity framerate) computation of locations from autocorrelation matrices on a ThinkPad Yoga 14 (which is not equipped with state-of-the-art processing power). Thus, for real-time functionality, this move is not necessary. This move would also require potentially more space than is possible on the FPGA. As such, this is the least concern in considering improvements that could be made to the FPGA implementation.

5.3 Host Computer

5.3.1 Unity

Our current implementation in Unity could be aesthetically improved greatly by making more detailed game objects. The game objects currently involved in our Unity environment are the bare minimum required to understand the results of our experiment, but more realistic representations of both the PLANC and the transmitter would greatly improve the aesthetic quality of our visualization program.

Furthermore, we only currently have the ability to represent a single transmitter in space. This is a result of our only having one SDR daughterboard capable of transmitting at 915 MHz. We will eventually want the ability to choose how many sources are being tracked, and have a number of selections for visual representation of these sources.

5.3.2 C++ DLL

In the future we would like to test beamforming algorithms other than MUSIC. One particular beamformer that we would want to look into is the Minimum Variance Distortionless Response beamformer (MVDR) [21]. The MVDR algorithm also uses an autocorrelation matrix to obtain AOAs, so this algorithm can be tested with no changes to the physical portions of our project. We

would like to see how its speed and localizing capabilities compare to the MUSIC algorithm that we have already implemented. On top of this, we would like to optimizing our code so that it can run faster and keep up with the influx of data coming from the FPGA.

5.4 Physical Implementation

There are several aesthetic and practical changes that we wish to make to PLANC. On the aesthetic side, we want to clean up the wiring and spray paint the board once more. On the practical side, we want to add power circuitry so that we can have a single plug that we can put into an outlet instead of using a power supply as we do now. This would make testing less difficult, and ensure that our voltages are always set to the right value as opposed to using the power supply. A power supply can be dangerous, as a mistaken turn of the knobs can cause our whole project to be overvolted.

5.5 Transmitters

To take advantage of the full capabilities of the implemented design (multi-source localization), one needs several mobile RF transmitters. Future work should focus on the design and implementation of compact mobile RF sources at the correct frequency.

6 Potential Applications

R3CAP, in its current state, is not yet accurate enough for most commercial applications. However, when scaled up, R3CAP can have a wide range of motion tracking and localization applications in industry settings.

As for industrial applications, R3CAP can perform the currently implemented localization tasks used in warehouses to locate items in storage, with the added benefit that items in motion in warehouses can be tracked if necessary. Larger-scale industrial uses include cargo cranes, which already in some cases employ similar triangulation schemes to localize the cargo they must move.

When compared to other motion capture methods, R3CAP has an advantage because of its lack of necessity of line-of-sight. As a result, a single R3CAP board can be used with small RF transmitters on the body of a tracked subject, as opposed to current methods such as computer vision and infrared reflectometry which require multiple apparatuses to make constant line-of-sight. This reduces both cost and space, and could even be implemented by amateur animators with ease.

Lastly, the implementation of a transmitter could allow for the motion tracking of any passive UHF RFID tags. This could provide a wealth of home and industry applications which we have not even considered, as almost anything can be tracked with RFID tags. Of course, with custom transmitters, this is the case as well, but they are active and would likely be less compact than an RFID tag.

7 Conclusion

R3CAP is a real time beamformer capable of locating RF sources that are nearby. It uses 8 antennas over two arrays to acquire the 915 MHz, downconvert it to 5 MHz, and then performs an analog to digital to conversion. The bit stream coming from each antenna's receiver chain is then processed on the FPGA to compute signal the incoming signal strength so that the AGC can be adjusted to keep our signal rail-to-rail without clipping. The incoming digital data is processed further by the FPGA. Digital IQ demodulation and filtering are performed before the incoming bit stream is used to calculate autocorrelation values. The autocorrelation values are then passed into The PS of the FPGA where they are prepared as float arrays and then sent to the host computer. The host computer then feeds the float arrays into the C++ DLL code. This code performs the MUSIC algorithm to compute the MUSIC spectrum. Peak finding is used on the MUSIC spectrum to find the AOAs of the RF sources relative to each array. The AOAs from each array are then matched appropriately to localize the tag. The localized tag is then displayed in Unity relative to PLANC. PLANC has both antenna arrays screwed down to it along with the FPGA.

R3CAP was able to locate a single RF source within our predicted error bound for the size of arrays that were used. The range can be improved by improving the matching on the antennas, while localization performance can be improved by adding more antennas to each array.

8 Thank You

Thank you Zoga Borova and Henry Kasen from LGS for helping us with the design of our RF-frontend.
Thank you Brian Woods from Maccentric for helping us with RF-frontend deisgn and component selection advice.
Thank you Joni Lappalainen from Optenni Lab for giving us access to their software and giving us advice on antenna matching.
Thank you ENGRTUTOR from youtube.com/user/ENGRTUTOR for providing helpful tutorials on the use of the Vivado software with ZedBoard.
Thank you Jeff Sherman from Columbia University for helping us debug our PCBs.
Thank you Aladino Melendez from The Cooper Union for his help in fixing our FPGA and helping us put PLANC together.
Thank you to Joey Berkovitz and the Microlab staff at The Cooper Union for helping us gain access to the ADS software.
Thank you Prof. Sam Keene for being our adviser, providing a voice of reason, and most of all for believing in us even when we didn't.

References

[1] Christoph Angerer, Robert Langwieser and Markus Rupp. *Direction of Arrival Estimation by Phased Arrays in RFID*. Institute of Communications and Radio-Frequency Engineering. Vienna University of Technology, Austria, 2010.

[2] Ferdews Tlili, Noureddine Hamdi and Abdelfettah Belghith. *Accurate 3D Localization Scheme Based on Active RFID Tags for Indoor Environment*. IEEE 2012 International Conference on RFID -Technologies and Applications (RFID - TA).

[3] Lanxin Qiu, Xiaoxuan Liang and Zhangqin Huang. *PATL: A RFID Tag Localization based on Phased Array Antenna*. Scientific Reports Volume 7, Article number: 44183, 2017.

[4] David Bernal, Pau Closas and Juan A. Fernandez-Rubio. *Digital IQ Demodulation in array processing: Theory and Implementation*. 16th European Signal Processing Conference (EUSIPCO 2008), Lausanne, Switzerland, August 25-29, 2008.

[5] Texas Instruments,"ADCS747x 1-MSPS, 12-Bit, 10-Bit, and 8-Bit A/D Converters," datasheet, April 2003 [Revised May 2015].

[6] ISO/IEC 18000-6, "Information technology Radio frequency identification for item management Part 6: Parameters for air interface communications at 860 MHz to 960 MHz General." Third edition, 2015.

[7] Bin You, Bo Yang, Xuan Wen, Liangyu Qu, "Implementation of Low-Cost UHF RFID Reader Front-Ends with Carrier Leakage Suppression Circuit" Hindawi Publishing Corporation, International Journal of Antennas and Propagation, Volume 2013, 2013

[8] Skyworks, "300 to 2200 MHz Ultra Low-Noise Amplifier", SKY67150-396LF datasheet, May 2017

[9] Johanson Technology, "915 MHz ISM Antenna for small form factor application", P/N 0915AT43A0026 datasheet, 9/7/2016

[10] Susumu, "High Precision Chip Attenuator, PAT Series", PAT1220 datasheet

[11] Qualcomm, "SAW Components, SAW RF filter, Short range devices", B3726 Datasheet, May 16 2013

[12] Linear Technology, "High Linearity, Low Power Downconverting Mixer", LT5526 Datasheet

[13] TDK, "EMC Components, 3-terminal filters", MEM2012V121RT001 Datasheet, 20190207

[14] Analog Devices, "General-Purpose, Low Cost, DC-Coupled VGA", AD8337 Datasheet, 2016

[15] Texas Instruments, "LMH6639 190MHz Rail-to-Rail Output Amplifier with Diasble", LMH6639 Datasheet, 2013

[16] Texas Instruments, "ADS7040 Ultra-Low Power, Ultra-Small Size, 8-Bit, 1-MSPS, SAR ADC, ADS7040 Datahseet", December 2015

[17] Texas Instruments, "DAC081S101 8-Bit Micro Power Digital-to-Analog Converter with Rail-to-Rail Output", DAC081S101, Febuary 2013

[18] Crystek Crystals, "Ultra-Low Phase Noise 1GHz SAW Clock", CCS0-914X3-1000 Datasheet, 25th February 2019

[19] Crystek Crystals, "Ultra-Low Phase Noise SineWave VCXO", CVSS-945X-80.000 Datasheet, 28th March 2018

[20] Schmidt, R. *Multiple emitter location and signal parameter estimation.* IEEE Transactions on Antennas and Propagation, 34(3), 276-280, 1986.

[21] Haykin, S. *Adaptive Filter Theory, Fourth Edition*

[22] ZedBoard: Zynq^{TM} Evaluation Development Hardware User's Guide

9 Appendix – RF-Frontend Schematics

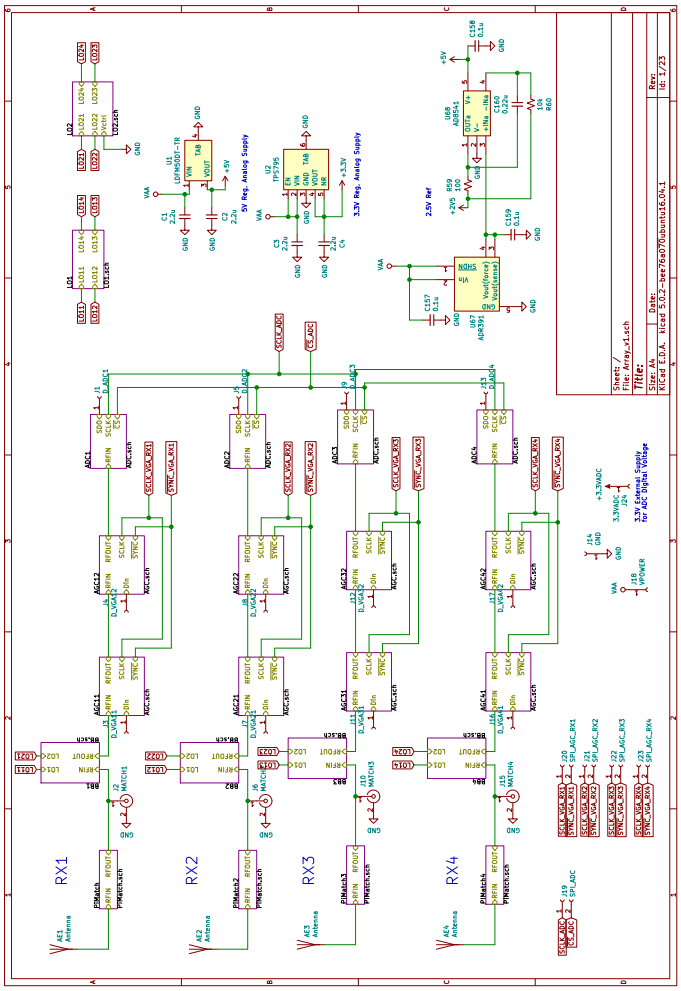

Figure 40: Schematic of 2x2 array. Sub-schematics are presented on subsequent pages.

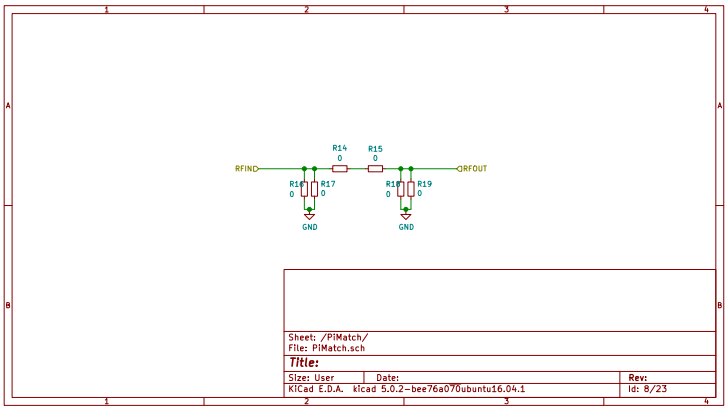

Figure 41: Antenna Pi-Matching sub-schematic

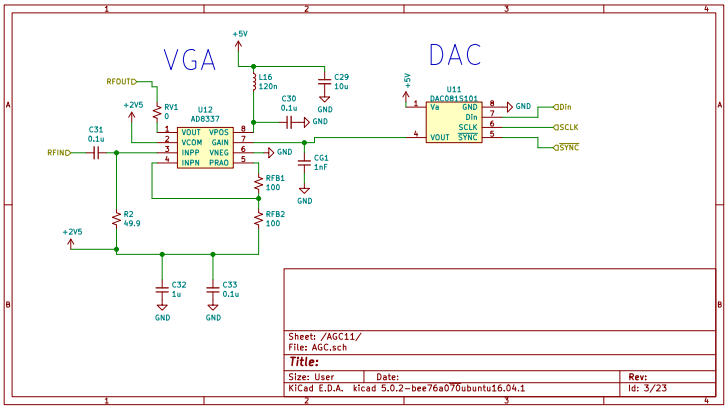

Figure 42: Schematic of the AGC circuit, as used in the Array prototype schematic (Figure 40).

10 Appendix – RF-Frontend Gain Compression Table and Parts-list

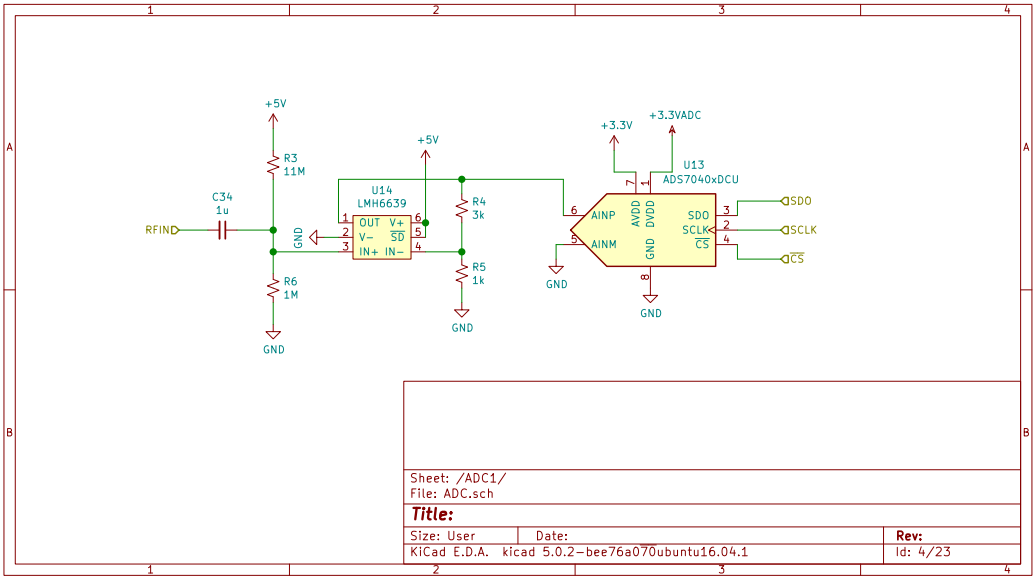

Figure 43: Schematic of the ADC configuration, as used in the Array prototype schematic (Figure 40).

11 Appendix – HDL Code

11.1 Constraints

11.1.1 base_constraints.xdc

```
## Clock signal

set_property PACKAGE_PIN Y9 [get_ports CLK]

    set_property IOSTANDARD LVCMOS33 [get_ports CLK]

    create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports CLK
        ]

##Pmod Header JA

#Sch name = JA1

set_property PACKAGE_PIN Y11 [get_ports {Din0[0]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din0[0]}]

#Sch name = JA2
```

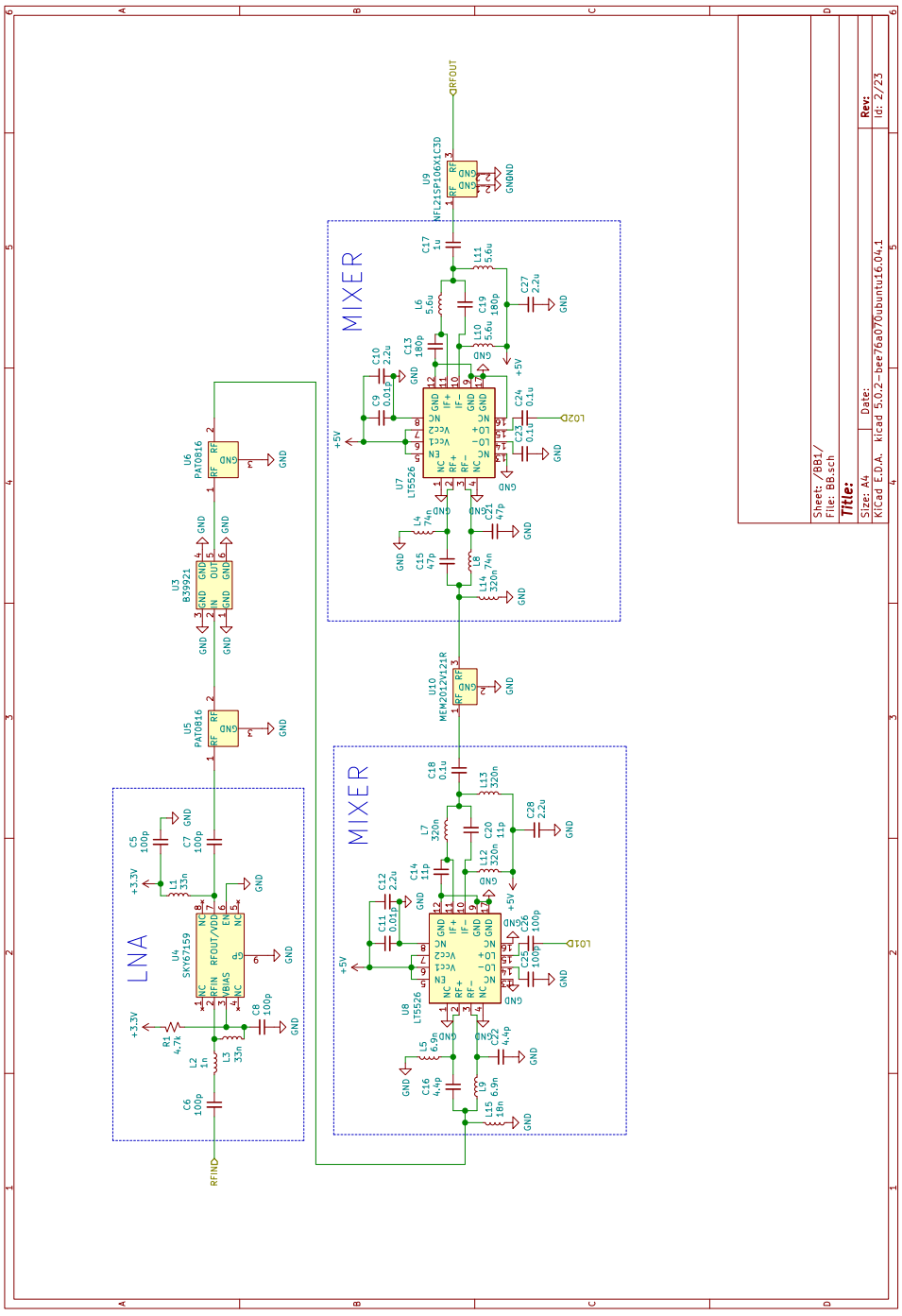

Figure 44: Mother-receiver RF-backbone schematic.

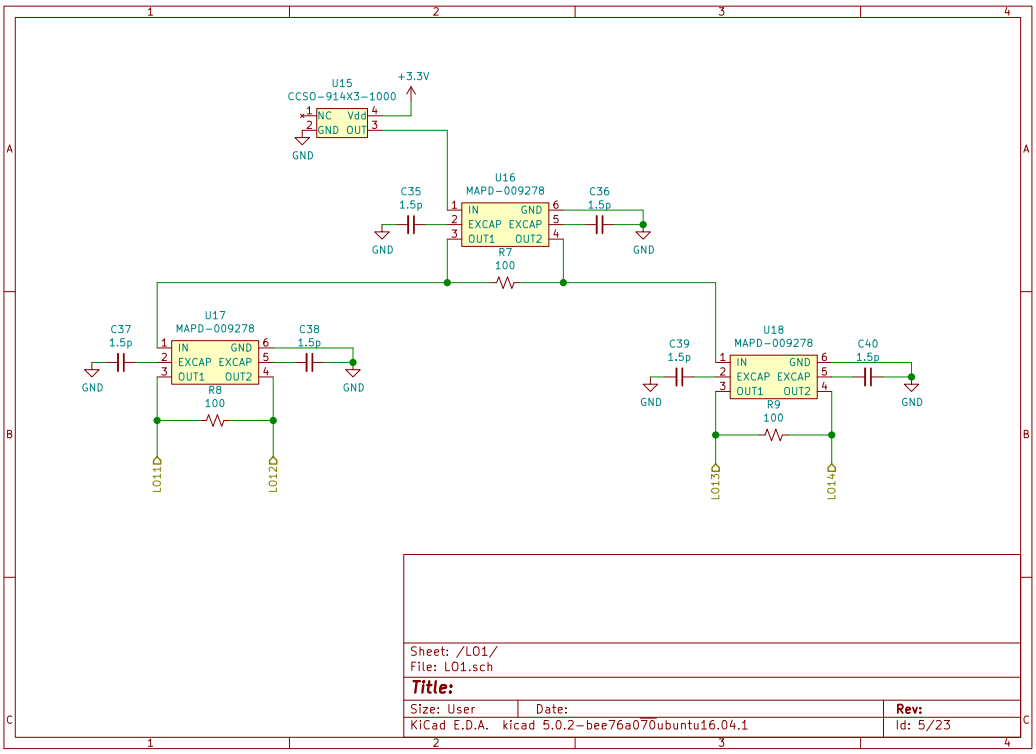

Figure 45: Schematic of the 1 GHz local oscillator configuration, as used in the Array prototype schematic (Figure 40).

```
set_property PACKAGE_PIN AA11 [get_ports {Din0[1]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din0[1]}]

##Sch name = JA3

set_property PACKAGE_PIN Y10 [get_ports {Din0[2]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din0[2]}]

##Sch name = JA4

set_property PACKAGE_PIN AA9 [get_ports {Din0[3]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din0[3]}]

##Sch name = JA7
```

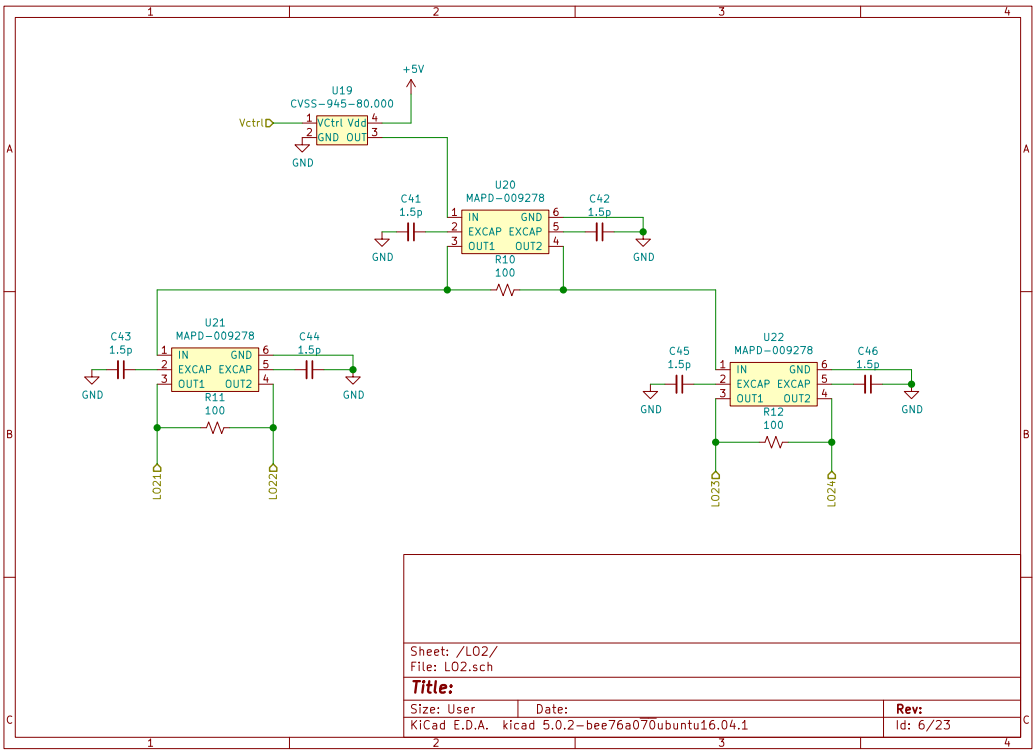

Figure 46: Schematic of the 80 MHz local oscillator configuration, as used in the Array prototype schematic (Figure 40).

```
set_property PACKAGE_PIN AB11 [get_ports {Din1[0]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din1[0]}]

##Sch name = JA8

set_property PACKAGE_PIN AB10 [get_ports {Din1[1]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din1[1]}]

##Sch name = JA9

set_property PACKAGE_PIN AB9 [get_ports {Din1[2]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din1[2]}]

##Sch name = JA10
```

ARRAY PROTOTYPE V1 -- 2X2 RECEIVER ARRAY

Single RX

Componenent	Gain	Max Power (dbm)	IP1 (dbm)	Pwr into cmpt (dBm)	Vsupply (V)	Max Current (mA)	Part #	Features	Footprint	Datasheet	Price ($)	quantity	
ANT	-4	33		-25			0915AT43A0026	Chip	custom	https://www.johar	0.95	10	1
LNA	17.8	21	-1	-29	3.3	100	SKY67159-396LF	RF Choke needed	DFN8	https://www.mou	4.37	10	1
ATTN	-3	20		-11.2			PAT1220-C-3DB-T	Pi Network, Max 1.3 VSWR	0805 / custom	https://www.susu	0.35	18	1
BPF	-3.5	10		-14.2			B39921B3726U41(	10 MHz BW, Max 1.9 VSWR	DDC6C / SMD-6	https://en.rf360jv	1.89	18	1
ATTN	-3	20		-17.7			PAT1220-C-3DB-T	Pi Network, Max 1.3 VSWR	0805 / custom	https://www.susu	0.35		
MIXER	0.6	10	5	-20.7	5	28	LT5526	Differential IO, -10dBm LO	16QFN 4x4	https://www.analo	6.97	18	1
LPF	-2	30		-20.1			MEM2012V121R	100 MHz cutoff	custom	https://product.td	0.46	10	1
MIXER	0.6	10	5	-22.1	5	28	LT5526	Differential IO, -5dBm LO	16QFN 4x4	https://www.analo	6.97		
LPF	-1	30		-21.5			NFL21SP106X1C3	10 MHz cutoff, -35dB@50MHz,	custom	https://www.murz	0.41	10	1
VGA	0		8.2	-22.5	5	25	AD8337	100 MHz BW, DC-coupled	DFN8 3x3	https://www.analo	6.25	18	1
VGA	0		8.2	-22.5	5	25	AD8337	100 MHz BW, DC-coupled	DFN8 3x3	https://www.analo	6.25		
OPAMP	12			-22.5	5		LMH6639MF		SOT23-6	http://www.ti.com	1.6	10	1
ADC		22		-10.5	3.3	0.1	ADS7040IDCUR	1 MHz, single ended, SPI	VSSOP 8	ADS7040	1.2	10	1
total	14.5					206.1					38.02	368.18	

LOs	Freq (MHz)	output pwr (dbm)	stability (MHz)	Vsupply (V)	Max Current (mA)							
LO1	1000	5	0.015	3.3	35	CCSO-914X3-1000		custom	http://www.cryste	49.5	1	1
LO2	80	5	0.002	5	35	CVSS-945-80.000		same as LO1	http://www.cryste	29.09	1	1
Pwr Splitter	5-1000					MAPD-009278		custom	https://cdn.macor	2.64	14	1
Value	footprint	quantity x1 array									115.55	
1.5pF	0603	12										
100ohm	0603	6										
0.1uF	0603	2										
100pF	0603	2										

LNA LC Components			**ALL RLC COMPONENTS**						
Value	footprint	quantity x1 rx	Value	footprint	quantity	price	part #		
100pF	0603	4	1.5pF	0603	29	0.1	https://www.digikey.com/product	X	1
1nH	0603	1	4.4pF	0603	21	0.26	399-16187-1-ND	X	1
33nH	0603	2	11pF	0603	21	0.1	1276-2173-1-ND	X	1
			47pF	0603	21	0.1	478-1171-1-ND	X	1
4.7kohm	0603	1	100pF	0603	41	0.1	311-1069-1-ND	X	1
			180pF	0603	21	0.1	1276-1255-1-ND	X	1
Mixer Balun LC Components			1nF	0603	21	0.1	1276-1018-1-ND	X	1
Value	footprint	quantity x1 rx	0.1uF	0603	49	0.1	1276-1000-1-ND	X	1
--- RF1 ---			1uF	0603	13	0.1	1276-1036-1-ND	X	1
4.4pF	0603	2	2.2uF	0805	33	0.11	https://www.digikey.com/product	x	1
6.8nH	0603	2							
18nH	0603	1	1nH	0603	13	0.1	712-1424-1-ND	x	1
--- IF1 ---			6.8nH	0603	21	0.1	https://www.digikey.com/product	x	1
330nH	0603	3	18nH	0603	13	0.1	https://www.digikey.com/product	x	1
11pF	0603	2	33nH	0603	21	0.1	https://www.digikey.com/product	x	1
0.1uF	0603	1	68nH	0603	21	0.1	https://www.digikey.com/product	x	1
2.2uF	0805	1	120nH	0603	21	0.1	https://www.digikey.com/product	x	1
--- RF2 ---			330nH	0603	37	0.1	https://www.digikey.com/product	x	1
47pF	0603	2	5.6uH	0603	29	0.1	https://www.digikey.com/product	x	1
68nH	0603	2							
330nH	0603	1	49.9ohm	0603	21	0.1	https://www.digikey.com/product	x	1
--- IF2 ---			3kohm	0603	13	0.1	https://www.digikey.com/product	x	1
5.6uH	0603	3	1kohm	0603	13	0.1	https://www.digikey.com/product	x	1
180pF	0603	2	1Mohm	0603	13	0.1	https://www.digikey.com/product	x	1
1uF	0603	1	11Mohm	0603	13	0.1	https://www.digikey.com/product	x	1
2.2uF	0805	1	4.7kohm	0603	13	0.1	https://www.digikey.com/product	x	1
			100ohm	0603	49	0.1	https://www.digikey.com/product	x	1

Figure 47: Gain Compression Table and Parts-list for 2x2 Array prototype.

```
set_property PACKAGE_PIN AA8 [get_ports {Din1[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Din1[3]}]

#Sch name = JB1
set_property PACKAGE_PIN W12 [get_ports {SCLK}]
    set_property IOSTANDARD LVCMOS33 [get_ports {SCLK}]
#Sch name = JB2
set_property PACKAGE_PIN W11 [get_ports {notCS}]
    set_property IOSTANDARD LVCMOS33 [get_ports {notCS}]
##Sch name = JB3
```

```
set_property PACKAGE_PIN V10 [get_ports {SCLKDAC}]
    set_property IOSTANDARD LVCMOS33 [get_ports {SCLKDAC}]
##Sch name = JB4
set_property PACKAGE_PIN W8 [get_ports {notSYNC}]
    set_property IOSTANDARD LVCMOS33 [get_ports {notSYNC}]
#Sch name = JB7
set_property PACKAGE_PIN V12 [get_ports {ADC_0}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_0}]
#Sch name = JB8
set_property PACKAGE_PIN W10 [get_ports {ADC_1}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_1}]
##Sch name = JB9
set_property PACKAGE_PIN V9 [get_ports {ADC_2}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_2}]
##Sch name = JB10
set_property PACKAGE_PIN V8 [get_ports {ADC_3}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_3}]


#Sch name = JC1_P
    set_property PACKAGE_PIN AB7 [get_ports {SCLK_1}]
        set_property IOSTANDARD LVCMOS33 [get_ports {SCLK_1}]
#Sch name = JC1_N
set_property PACKAGE_PIN AB6 [get_ports {notCS_1}]
    set_property IOSTANDARD LVCMOS33 [get_ports {notCS_1}]
##Sch name = JC2_P
set_property PACKAGE_PIN Y4 [get_ports {SCLKDAC_1}]
    set_property IOSTANDARD LVCMOS33 [get_ports {SCLKDAC_1}]
##Sch name = JC2_N
set_property PACKAGE_PIN AA4 [get_ports {notSYNC_1}]
```

```
    set_property IOSTANDARD LVCMOS33 [get_ports {notSYNC_1}]
#Sch name = JC3_P
set_property PACKAGE_PIN R6 [get_ports {ADC_0_1}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_0_1}]
#Sch name = JC3_N
set_property PACKAGE_PIN T6 [get_ports {ADC_1_1}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_1_1}]

##Sch name = JC4_P
set_property PACKAGE_PIN T4 [get_ports {ADC_2_1}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_2_1}]

##Sch name = JC4_N
set_property PACKAGE_PIN U4 [get_ports {ADC_3_1}]
    set_property IOSTANDARD LVCMOS33 [get_ports {ADC_3_1}]

#Sch name = JD1_P
set_property PACKAGE_PIN V7 [get_ports {Din0_1[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Din0_1[0]}]
#Sch name = JD1_N
set_property PACKAGE_PIN W7 [get_ports {Din0_1[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Din0_1[1]}]
##Sch name = JD2_P
set_property PACKAGE_PIN V5 [get_ports {Din0_1[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Din0_1[2]}]
##Sch name = JD2_N
set_property PACKAGE_PIN V4 [get_ports {Din0_1[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Din0_1[3]}]
#Sch name = JD3_P
set_property PACKAGE_PIN W6 [get_ports {Din1_1[0]}]
```

```
    set_property IOSTANDARD LVCMOS33 [get_ports {Din1_1[0]}]

#Sch name = JD3_N

set_property PACKAGE_PIN W5 [get_ports {Din1_1[1]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din1_1[1]}]
##Sch name = JD4_P

    set_property PACKAGE_PIN U6 [get_ports {Din1_1[2]}]

        set_property IOSTANDARD LVCMOS33 [get_ports {Din1_1[2]}]
##Sch name = JD4_N

set_property PACKAGE_PIN U5 [get_ports {Din1_1[3]}]

    set_property IOSTANDARD LVCMOS33 [get_ports {Din1_1[3]}]
```

11.2 Autocorrelation Generator and Submodules

All HDL code written entirely by us is included herein. IPs generated by Vivado, such as counters, clock wizards, FIR filters and complex multipliers are not included. Furthermore, AXI protocols and block diagram wrappers generated by Xilinx are not included.

11.2.1 autocor_backend.v – Autocorrelation Matrix Generator

```
`timescale 1ns / 1ps
module autocor_backend(
    input Clk,
    input data_in_0, input data_in_1, input data_in_2, input data_in_3, //ADC Inputs
    output reg [111:0] R11, output reg [111:0] R12, output reg [111:0] R13, output
        reg [111:0] R14,
    output reg [111:0] R22, output reg [111:0] R23, output reg [111:0] R24,
    output reg [111:0] R33, output reg [111:0] R34,
    output reg [111:0] R44, //Rij is the ijth entry in autocorr matrix, Hermitian so
         no need for lower indices
    output wire R_ready, // Signals when all 1024 samples have been processed and
        the R matrix can be read
    output ADC_STARTEND,
    output notCS, output SCLK, output [3:0] LPF_out_valid,
    output SCLKDAC, output notSYNC, output [3:0] Din0, output [3:0] Din1 // Control
        for ADCs
    );

    wire [2:0] notSYNCs;
    wire [23:0]  I0, Q0, I1, Q1, I2, Q2, I3, Q3;
    wire [23:0] Q0c, Q1c, Q2c, Q3c; // These are the negative Qs, used for cplx conj
    //wire [3:0] LPF_out_valid;
    wire [10:0] SAMPNO; // Number of samples taken so far

    wire [7:0] go;
    wire mult_in_valid, mda_valid, CLR, ADC_OUT_VALID;
    wire [9:0] mult_out_valid_int, mult_out_valid;
    wire [111:0] Rtemp11, Rtemp12, Rtemp13, Rtemp14,
    Rtemp22, Rtemp23, Rtemp24, Rtemp33, Rtemp34, Rtemp44;
    wire [111:0]  Racc11, Racc12, Racc13, Racc14, Racc22, Racc23, Racc24, Racc33,
        Racc34, Racc44;
```

```
// Rtemp is output of adder, Racc is average accumulating, Rij is full output,
    asserted until next is ready
wire RST;

initial
begin
    R11 <= 0;
    R12 <= 0;
    R13 <= 0;
    R14 <= 0;
    R22 <= 0;
    R23 <= 0;
    R24 <= 0;
    R33 <= 0;
    R34 <= 0;
    R44 <= 0;
end

//  ----------------------- IP Initialization
    ---------------------------------------------//

synth_adc_timing adc (Clk, notCS, SCLK, ADC_OUT_VALID, go, ADC_STARTEND,CLR);
clk_wiz_2 DACclock (SCLKDAC,Clk);


adc_to_iq ADC0 (Clk, go, ADC_STARTEND, ADC_OUT_VALID, SCLK, CLR, data_in_0,
I0, Q0, LPF_out_valid[0], SCLKDAC, notSYNC, Din0[0], Din1[0]);
adc_to_iq ADC1 (Clk, go, ADC_STARTEND, ADC_OUT_VALID, SCLK, CLR, data_in_1,
    I1, Q1, LPF_out_valid[1], SCLKDAC, notSYNCs[0], Din0[1], Din1[1]);
adc_to_iq ADC2 (Clk, go, ADC_STARTEND, ADC_OUT_VALID, SCLK, CLR, data_in_2,
        I2, Q2, LPF_out_valid[2], SCLKDAC, notSYNCs[1], Din0[2], Din1[2]);
adc_to_iq ADC3 (Clk, go, ADC_STARTEND, ADC_OUT_VALID, SCLK, CLR, data_in_3,
            I3, Q3, LPF_out_valid[3], SCLKDAC, notSYNCs[2], Din0[3], Din1[3]);

negate24 neg0 (Q0,Q0c);
negate24 neg1 (Q1,Q1c);
negate24 neg2 (Q2,Q2c);
negate24 neg3 (Q3,Q3c);

//
//      [ 0  1  2  3 ]      0->R11
//      [ -  4  5  6 ]
//      [ -  -  7  8 ]
//      [ -  -  -  9 ]
//



mult_div_add op_0 (Clk, mult_in_valid, {Q0c,I0}, {Q0,I0}, Racc11,
    mult_out_valid_int[0], Rtemp11);
mult_div_add op_1 (Clk, mult_in_valid, {Q1c,I1}, {Q0,I0}, Racc12,
    mult_out_valid_int[1], Rtemp12);
mult_div_add op_2 (Clk, mult_in_valid, {Q2c,I2}, {Q0,I0}, Racc13,
    mult_out_valid_int[2], Rtemp13);
mult_div_add op_3 (Clk, mult_in_valid, {Q3c,I3}, {Q0,I0}, Racc14,
    mult_out_valid_int[3], Rtemp14);
mult_div_add op_4 (Clk, mult_in_valid, {Q1c,I1}, {Q1,I1}, Racc22,
    mult_out_valid_int[4], Rtemp22);
```

```
 mult_div_add op_5 (Clk, mult_in_valid, {Q2c,I2}, {Q1,I1}, Racc23,
     mult_out_valid_int[5], Rtemp23);
 mult_div_add op_6 (Clk, mult_in_valid, {Q3c,I3}, {Q1,I1}, Racc24,
     mult_out_valid_int[6], Rtemp24);
 mult_div_add op_7 (Clk, mult_in_valid, {Q2c,I2}, {Q2,I2}, Racc33,
     mult_out_valid_int[7], Rtemp33);
 mult_div_add op_8 (Clk, mult_in_valid, {Q3c,I3}, {Q2,I2}, Racc34,
     mult_out_valid_int[8], Rtemp34);
 mult_div_add op_9 (Clk, mult_in_valid, {Q3c,I3}, {Q3,I3}, Racc44,
     mult_out_valid_int[9], Rtemp44);

 dff mo0 (mult_out_valid_int[0],Clk,0,1,mult_out_valid[0]);
 dff mo1 (mult_out_valid_int[1],Clk,0,1,mult_out_valid[1]);
 dff mo2 (mult_out_valid_int[2],Clk,0,1,mult_out_valid[2]);
 dff mo3 (mult_out_valid_int[3],Clk,0,1,mult_out_valid[3]);
 dff mo4 (mult_out_valid_int[4],Clk,0,1,mult_out_valid[4]);
 dff mo5 (mult_out_valid_int[5],Clk,0,1,mult_out_valid[5]);
 dff mo6 (mult_out_valid_int[6],Clk,0,1,mult_out_valid[6]);
 dff mo7 (mult_out_valid_int[7],Clk,0,1,mult_out_valid[7]);
 dff mo8 (mult_out_valid_int[8],Clk,0,1,mult_out_valid[8]);
 dff mo9 (mult_out_valid_int[9],Clk,0,1,mult_out_valid[9]);

 and multinvalid (mult_in_valid,LPF_out_valid[0],LPF_out_valid[1],LPF_out_valid
     [2],LPF_out_valid[3]);
 and multoutvalid (mda_valid,mult_out_valid[0],mult_out_valid[1],mult_out_valid
     [2],mult_out_valid[3],
                     mult_out_valid[4],mult_out_valid[5],mult_out_valid[6],
                         mult_out_valid[7],
                     mult_out_valid[8],mult_out_valid[9]);
 // multinvalid control may need to change

 //counter to 1024 counts how many samples have been taken

 sample_counter sc (mda_valid,1,0,SAMPNO);

 assign RST = ~SAMPNO[0] & ~SAMPNO[1] & ~SAMPNO[2] & ~SAMPNO[3] & ~SAMPNO[4] & ~
     SAMPNO[5] & ~SAMPNO[6]
             & ~SAMPNO[7] & ~SAMPNO[8] & ~SAMPNO[9] & ~SAMPNO[10];

 dff112 mao0 (Rtemp11,mda_valid & ~SAMPNO[10],RST,1,Racc11);
 dff112 mao1 (Rtemp12,mda_valid & ~SAMPNO[10],RST,1,Racc12);
 dff112 mao2 (Rtemp13,mda_valid & ~SAMPNO[10],RST,1,Racc13);
 dff112 mao3 (Rtemp14,mda_valid & ~SAMPNO[10],RST,1,Racc14);
 dff112 mao4 (Rtemp22,mda_valid & ~SAMPNO[10],RST,1,Racc22);
 dff112 mao5 (Rtemp23,mda_valid & ~SAMPNO[10],RST,1,Racc23);
 dff112 mao6 (Rtemp24,mda_valid & ~SAMPNO[10],RST,1,Racc24);
 dff112 mao7 (Rtemp33,mda_valid & ~SAMPNO[10],RST,1,Racc33);
 dff112 mao8 (Rtemp34,mda_valid & ~SAMPNO[10],RST,1,Racc34);
 dff112 mao9 (Rtemp44,mda_valid & ~SAMPNO[10],RST,1,Racc44);


always @(posedge SAMPNO[10])
begin
     R11 <= Racc11;
     R12 <= Racc12;
     R13 <= Racc13;
     R14 <= Racc14;
     R22 <= Racc22;
```

```
        R23 <= Racc23;
        R24 <= Racc24;
        R33 <= Racc33;
        R34 <= Racc34;
        R44 <= Racc44;
   end

    assign R_ready = SAMPNO[10];

endmodule
```

11.2.2 dff.v – One-Bit D-Type Flip-Flop

```
‘timescale 1ns / 1ps
module dff(
    input D,
    input Clk,
    input reset,
    input CEn,
    output reg Q
    );

    always @(posedge Clk)
    if (reset)
    begin
        Q <= 1’b0;
    end
    else if (CEn)
    begin
        Q <= D;
    end
endmodule
```

11.2.3 dff112.v – 112-Bit D-Type Flip-Flop

```
‘timescale 1ns / 1ps
module dff112(
input [111:0] D,
    input Clk,
    input reset,
    input CEn,
    output reg [111:0] Q
    );

    always @(posedge Clk)
    if (reset)
    begin
        Q <= 0;
    end
    else if (CEn)
    begin
        Q <= D;
    end
endmodule
```

11.2.4 negate24.v – 24-Bit 2’s Complement Negation

```
‘timescale 1ns / 1ps
```

```
module negate24(
    input [23:0] A,
    output [23:0] negA
    );
    wire cout;
    adder24 adder_0 (~A,23'b0,1,negA,cout);
endmodule
```

11.2.5 adder24.v – 24-Bit Adder

```
`timescale 1ns / 1ps
module adder24(
    input [23:0] A,
    input [23:0] B,
    input CIN,
    output [23:0] S,
    output COUT
    );

    wire Cint[22:0];

    add adder_0 (A[0],B[0],CIN,S[0],Cint[0]);
    add adder_1 (A[1],B[1],Cint[0],S[1],Cint[1]);
    add adder_2 (A[2],B[2],Cint[1],S[2],Cint[2]);
    add adder_3 (A[3],B[3],Cint[2],S[3],Cint[3]);
    add adder_4 (A[4],B[4],Cint[3],S[4],Cint[4]);
    add adder_5 (A[5],B[5],Cint[4],S[5],Cint[5]);
    add adder_6 (A[6],B[6],Cint[5],S[6],Cint[6]);
    add adder_7 (A[7],B[7],Cint[6],S[7],Cint[7]);
    add adder_8 (A[8],B[8],Cint[7],S[8],Cint[8]);
    add adder_9 (A[9],B[9],Cint[8],S[9],Cint[9]);
    add adder_10 (A[10],B[10],Cint[9],S[10],Cint[10]);
    add adder_11 (A[11],B[11],Cint[10],S[11],Cint[11]);
    add adder_12 (A[12],B[12],Cint[11],S[12],Cint[12]);
    add adder_13 (A[13],B[13],Cint[12],S[13],Cint[13]);
    add adder_14 (A[14],B[14],Cint[13],S[14],Cint[14]);
    add adder_15 (A[15],B[15],Cint[14],S[15],Cint[15]);
    add adder_16 (A[16],B[16],Cint[15],S[16],Cint[16]);
    add adder_17 (A[17],B[17],Cint[16],S[17],Cint[17]);
    add adder_18 (A[18],B[18],Cint[17],S[18],Cint[18]);
    add adder_19 (A[19],B[19],Cint[18],S[19],Cint[19]);
    add adder_20 (A[20],B[20],Cint[19],S[20],Cint[20]);
    add adder_21 (A[21],B[21],Cint[20],S[21],Cint[21]);
    add adder_22 (A[22],B[22],Cint[21],S[22],Cint[22]);
    add adder_23 (A[23],B[23],Cint[22],S[23],COUT);

endmodule
```

11.2.6 add.v – One-Bit Full Adder

```
`timescale 1ns / 1ps
module add(
    input a,
    input b,
    input cin,
    output s,
    output cout
    );
```

```
    assign s = (a ^ b) ^ cin;
    assign cout = ((a ^ b) & cin) | (a & b);

endmodule
```

11.2.7 mult_div_add.v – Arithmetic Involved in Averaging

```
`timescale 1ns / 1ps
module mult_div_add(
    input Clk,
    input in_valid,
    input [47:0] aH,
    input [47:0] a,
    input [111:0] Rcurr,
    output out_valid,
    output [111:0] out,
    output [111:0] mult_out, output [111:0] div_out
    );

   // wire [111:0] mult_out;
   // wire [111:0] div_out;

    cmpy_0 mult (Clk, in_valid, aH, in_valid, a, out_valid, mult_out);
    divcplx div (mult_out,div_out);
    addcplx cadd (Rcurr, div_out, out);

endmodule
```

11.2.8 divcplx.v – Division of a Complex Number by 1024

```
`timescale 1ns / 1ps

module divcplx(input [111:0] in, output [111:0] out);

divby1024 re (in[55:0],out[55:0]);
divby1024 im (in[111:56],out[111:56]);

endmodule
```

11.2.9 divby1024.v – Division of 56-Bit 2's Complement Numbers by 1024

```
`timescale 1ns / 1ps
module divby1024(
    input [55:0] a,
    output [55:0] aon1024
    );

    assign aon1024[44:0] = a[55:10];
    assign aon1024[55:45] = {a[55],a[55],a[55],a[55],a[55],a[55],a[55],a[55],a[55],a
        [55],a[55]};
endmodule
```

11.2.10 addcplx.v – Addition of 112-bit Complex Numbers

```
`timescale 1ns / 1ps
module addcplx(
    input [111:0] a,
```

```
    input [111:0] b,
    output [111:0] out
    );

    safe_add re (a[55:0],b[55:0],out[55:0]);
    safe_add im (a[111:56],b[111:56],out[111:56]);

endmodule
```

11.2.11 safe_add.v – 56-Bit Full Adder

```
`timescale 1ns / 1ps
module safe_add(
    input [55:0] a,
    input [55:0] b,
    output [55:0] sum
    );

    wire [3:0] cint;

    adder32 a_0 (a[31:0],b[31:0],1'b0,sum[31:0],cint[0]);
    adder8 a_1 (a[39:32],b[39:32],cint[0],sum[39:32],cint[1]);
    adder8 a_2 (a[47:40],b[47:40],cint[1],sum[47:40],cint[2]);
    adder8 a_3 (a[55:48],b[55:48],cint[2],sum[55:48],cint[3]);

endmodule
```

11.2.12 adder32.v – 32-Bit Full Adder

```
`timescale 1ns / 1ps
module adder32(
    input [31:0] A,
    input [31:0] B,
    input CIN,
    output [31:0] S,
    output COUT
    );

    wire [2:0] CINT;

    adder8 a_0 (A[7:0],B[7:0],CIN,S[7:0],CINT[0]);
    adder8 a_1 (A[15:8],B[15:8],CINT[0],S[15:8],CINT[1]);
    adder8 a_2 (A[23:16],B[23:16],CINT[1],S[23:16],CINT[2]);
    adder8 a_3 (A[31:24],B[31:24],CINT[2],S[31:24],COUT);

endmodule
```

11.2.13 adder8.v – 8-Bit Full Adder

```
`timescale 1ns / 1ps
module adder8(
    input [7:0] A,
    input [7:0] B,
    input CIN,
    output [7:0] S,
    output COUT
    );
```

```
    wire Cint[6:0];

    add adder_0 (A[0],B[0],CIN,S[0],Cint[0]);
    add adder_1 (A[1],B[1],Cint[0],S[1],Cint[1]);
    add adder_2 (A[2],B[2],Cint[1],S[2],Cint[2]);
    add adder_3 (A[3],B[3],Cint[2],S[3],Cint[3]);
    add adder_4 (A[4],B[4],Cint[3],S[4],Cint[4]);
    add adder_5 (A[5],B[5],Cint[4],S[5],Cint[5]);
    add adder_6 (A[6],B[6],Cint[5],S[6],Cint[6]);
    add adder_7 (A[7],B[7],Cint[6],S[7],COUT);

endmodule
```

11.2.14 synth_adc_timing.v – Clock Control for ADC

```
`timescale 1ns / 1ps
module synth_adc_timing(input Clk, output wire notCS, output wire SCLK, output wire
    byte_read, output wire [7:0] go,
output wire START, output reg CLR);

wire [7:0] STATE;
wire [7:0] STARTSTATE;
wire [7:0] DATASTATE;

wire bitread;
wire slowClk;
wire locked;

wire int_SCLK;
wire int_notCS;
wire int_START;
wire int_bitread;
wire int_byteread;



clk_wiz_0 twentyM (slowClk,0,locked,Clk);


initial
begin

    CLR <= 1;
    @(posedge Clk)
    begin
    CLR <= 0;
    end

end

c_counter_binary_0 STATECOUNT (Clk,START,CLR,STATE);
c_counter_binary_0 STARTCOUNT (slowClk,~START,CLR,STARTSTATE);
c_counter_binary_1 DATACOUNT (bitread,START,CLR,DATASTATE);

dff dSCLK (int_SCLK,Clk,CLR,1,SCLK);
dff dnotCS (int_notCS,Clk,CLR,1,notCS);
dff dSTART (int_START,Clk,CLR,1,START);
dff dbitread (int_bitread,Clk,CLR,1,bitread);
```

```
dff dbyteread (int_byteread,Clk,CLR,1,byte_read);

//dff data7 (DATA,SCLK,CLR,go[7],data_byte[7]);
//dff data6 (DATA,SCLK,CLR,go[6],data_byte[6]);
//dff data5 (DATA,SCLK,CLR,go[5],data_byte[5]);
//dff data4 (DATA,SCLK,CLR,go[4],data_byte[4]);
//dff data3 (DATA,SCLK,CLR,go[3],data_byte[3]);
//dff data2 (DATA,SCLK,CLR,go[2],data_byte[2]);
//dff data1 (DATA,SCLK,CLR,go[1],data_byte[1]);
//dff data0 (DATA,SCLK,CLR,go[0],data_byte[0]);

assign int_SCLK = (~START & ((~STARTSTATE[0] | (~STARTSTATE[1] & ~STARTSTATE[2] & ~
    STARTSTATE[3] & ~STARTSTATE[4] & ~STARTSTATE[5])
| (STARTSTATE[1] & STARTSTATE[2] & ~STARTSTATE[3] & ~STARTSTATE[4] & STARTSTATE[5]))
    ))
| (START & (
(STATE[6] & STATE[5] & ~STATE[1]) |
(~STATE[6] & ~STATE[5] & ~STATE[4] & ~STATE[3]) |
(~STATE[5] & ~STATE[4] & ~STATE[3] & ~STATE[2]) |
(STATE[6] & STATE[4] & STATE[2] & STATE[0]) |
(STATE[6] & STATE[4] & ~STATE[3] & STATE[2]) |
(STATE[6] & STATE[4] & STATE[2] & STATE[1]) |
(STATE[5] & ~STATE[4] & STATE[2] & ~STATE[1]) |
(~STATE[6] & ~STATE[4] & ~STATE[3] & ~STATE[2] & STATE[1]) |
(~STATE[6] & ~STATE[5] & STATE[4] & STATE[3] & ~STATE[2]) |
(~STATE[6] & ~STATE[5] & ~STATE[3] & ~STATE[2] & ~STATE[1]) |
(STATE[6] & ~STATE[5] & ~STATE[4] & ~STATE[2] & STATE[1]) |
(~STATE[6] & STATE[4] & STATE[3] & ~STATE[2] & ~STATE[1]) |
(~STATE[6] & ~STATE[4] & ~STATE[3] & ~STATE[1] & STATE[0]) |
(~STATE[6] & ~STATE[4] & STATE[3] & STATE[2] & STATE[1]) |
(~STATE[6] & ~STATE[4] & STATE[3] & STATE[2] & STATE[0]) |
(STATE[6] & ~STATE[5] & ~STATE[4] & ~STATE[2] & STATE[0]) |
(STATE[5] & ~STATE[4] & STATE[3] & STATE[1] & STATE[0]) |
(STATE[5] & STATE[4] & ~STATE[3] & STATE[2] & STATE[1]) |
(STATE[5] & ~STATE[3] & STATE[2] & ~STATE[1] & STATE[0]) |
(STATE[6] & STATE[4] & ~STATE[3] & STATE[1] & STATE[0]) |
(STATE[4] & ~STATE[3] & STATE[2] & STATE[1] & STATE[0]) |
(STATE[5] & STATE[4] & STATE[2] & STATE[1] & STATE[0]) |
(STATE[6] & ~STATE[4] & STATE[3] & STATE[2] & ~STATE[1])
));

assign int_bitread = (~notCS & START & ~SCLK);

assign int_notCS = (~START & ((~STARTSTATE[1] & ~STARTSTATE[2] & ~STARTSTATE[3] & ~
    STARTSTATE[4] & ~STARTSTATE[5])
| ((STARTSTATE[1] | STARTSTATE[2]) & ~STARTSTATE[3] & ~STARTSTATE[4] & STARTSTATE
    [5])))
| (START & (~STATE[6] & ~STATE[5] & ~STATE[4] & ~STATE[3] & (~STATE[2] | ~STATE[1]))
    );

assign int_START = STARTSTATE[0] & STARTSTATE[1] & STARTSTATE[2] & ~STARTSTATE[3] &
    ~STARTSTATE[4] & STARTSTATE[5];

assign go[7] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
~DATASTATE[3] & ~DATASTATE[2] & DATASTATE[1] & ~DATASTATE[0];
assign go[6] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
~DATASTATE[3] & ~DATASTATE[2] & DATASTATE[1] & DATASTATE[0];
assign go[5] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
```

```
~DATASTATE[3] & DATASTATE[2] & ~DATASTATE[1] & ~DATASTATE[0];
assign go[4] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
~DATASTATE[3] & DATASTATE[2] & ~DATASTATE[1] & DATASTATE[0];
assign go[3] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
~DATASTATE[3] & DATASTATE[2] & DATASTATE[1] & ~DATASTATE[0];
assign go[2] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
~DATASTATE[3] & DATASTATE[2] & DATASTATE[1] & DATASTATE[0];
assign go[1] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
DATASTATE[3] & ~DATASTATE[2] & ~DATASTATE[1] & ~DATASTATE[0];
assign go[0] = ~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
DATASTATE[3] & ~DATASTATE[2] & ~DATASTATE[1] & DATASTATE[0];

assign int_byteread = ~STATE[7] & STATE[6] & STATE[5] & ~STATE[4] & ~STATE[3] & ~
    STATE[2]
 & STATE[1] & STATE[0] & START;
//~DATASTATE[7] & ~DATASTATE[6] & ~DATASTATE[5] & ~DATASTATE[4] &
//~DATASTATE[3] & ~DATASTATE[2] & ~DATASTATE[1] & ~DATASTATE[0];

endmodule
```

11.2.15 adc_to_iq.v – AGC, IQ Demodulator and Filter Master

```
`timescale 1ns / 1ps
module adc_to_iq(input Clk, input [7:0] go, input STARTEND, input ADC_OUT_VALID,
    input SCLK, input CLR, input DATA,
output [23:0] xI, output [23:0] xQ, output data_out_valid, input SCLKDAC, output
    notSYNC, output Din0, output Din1);

wire [1:0] S;
wire [7:0] ADC_OUT,ADC_OUT_TR;

agc_test agc (
        Clk,
        DATA,
        SCLK, ADC_OUT_VALID, go, STARTEND, CLR,
        SCLKDAC,
        notSYNC,
        Din0, Din1, ADC_OUT
    );


assign ADC_OUT_TR[7] = ~ADC_OUT[7];
assign ADC_OUT_TR[6:0] = ADC_OUT[6:0];

IQ_LPF iqlpf (ADC_OUT_TR, Clk, S, ADC_OUT_VALID, xI, xQ, data_out_valid);

// IQ internal state counter (S)

S_counter scounter (ADC_OUT_VALID,STARTEND,0,S);

endmodule
```

11.2.16 agc_test.v – AGC Master

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Controlling DAC which controls VGA. To determine value, take sum over one
// of incoming ADC samples. Get 0->24 dB by moving input voltage between 0
```

```
// and 1.4 V or so. We chain two to get 0-48 dB with two DACs.
//
//////////////////////////////////////////////////////////////////////////////////


module agc_test(
        input clk,
        input adc_in,
        input SCLK, input byte_read, input [7:0] go, input START, input CLR,
        input SCLKDAC,
        output notSYNC,
        output Din0, output Din1, output [7:0] data_byte_raw, output reg [15:0]
            total, output reg [15:0] sum
    );

    wire [7:0] go, wrango,data_byte,data_byte_raw;
    wire byte_read;
    wire DACtime;
    wire DACtimed;
    reg SCLKtime = 0;
    wire  CLR;
    reg [15:0] sum = 0;
    reg [15:0] total = 0;
    reg [32:0] dintemp =
        {0,0,0,0,0,0,1,0,1,1,1,0,0,0,0,0,0,0,0,0,0,1,0,1,1,1,0,0,0,0,0,0};
    wire [4:0] STATE;
    wire notSYNCd;
    wire [7:0] k;
    wire trigger; wire triggerd;

    wire keepnotSYNCalive;

    dff DAC (DACtimed, clk, 0 , 1, DACtime);
    dff notS (notSYNCd, clk, 0, 1, notSYNC);
    dff trig (triggerd, clk, 0, 1, trigger);
    dff knsa (1,!SCLKDAC,DACtime,1,keepnotSYNCalive);


    c_counter_binary_4 DACSTATE (SCLKDAC, !notSYNC, !DACtime, STATE);
    c_counter_binary_5 CLKIN (byte_read || (trigger & !SCLKDAC), !DACtime, 0, k);


    assign Din0 = dintemp[16 - STATE[3:0]];
    assign Din1 = dintemp[32 -STATE[3:0]];

    adc_wrangler rango ( adc_in, SCLK, CLR, go, data_byte_raw);

    abs abval (data_byte_raw,data_byte);

    always @(posedge byte_read & !DACtime)
    begin
        if(k<49)
        begin
        sum <= sum+data_byte;
        end
        else
        begin
        total <= sum;
```

```
        sum <= 0;
        end
    end

    always @(posedge DACtime)
    begin
    if(total < 3150 )
    begin
        if(dintemp[11:4] >= 164)
        begin
            if (dintemp[27:20] < 164)
            begin
                dintemp[27:20]<= dintemp[27:20] + 1;
            end
        end
        else
        begin
            dintemp[11:4]<= dintemp[11:4] + 1;
        end
    end
    else if(total > 4550)
    begin
        if(dintemp[27:20] <= 92)
        begin
            if (dintemp[11:4] > 92)
            begin
                dintemp[11:4]<= dintemp[11:4] - 1;
            end
        end
        else
        begin
            dintemp[27:20]<= dintemp[27:20] - 1;
        end
    end

    end

    assign triggerd = STATE[4] & STATE[0];


    assign DACtimed = (k == 50) & !(STATE[4] & STATE[0]);

    assign notSYNCd = (STATE[4] & STATE[0]) || (STATE == 5'b0 & !DACtime) ||
        keepnotSYNCalive;

endmodule
```

11.2.17 adc_wrangler.v – Retrieves Bytes from the ADC

```
`timescale 1ns / 1ps
module adc_wrangler( input wire DATA, input wire SCLK, input wire CLR, input [7:0]
    go, output [7:0] data_byte
    );

    dff data7 (DATA,SCLK,CLR,go[7],data_byte[7]);
    dff data6 (DATA,SCLK,CLR,go[6],data_byte[6]);
    dff data5 (DATA,SCLK,CLR,go[5],data_byte[5]);
    dff data4 (DATA,SCLK,CLR,go[4],data_byte[4]);
```

```
    dff data3 (DATA,SCLK,CLR,go[3],data_byte[3]);
    dff data2 (DATA,SCLK,CLR,go[2],data_byte[2]);
    dff data1 (DATA,SCLK,CLR,go[1],data_byte[1]);
    dff data0 (DATA,SCLK,CLR,go[0],data_byte[0]);

endmodule
```

11.2.18 abs.v – Absolute Value of 2's Complement Number

```
`timescale 1ns / 1ps
module abs(
    input [7:0] data_byte_raw,
    output [7:0] data_byte
    );

    wire [7:0] m128;
    wire [7:0] n;

    assign m128[7] = ~data_byte_raw[7];
    assign m128[6:0] = data_byte_raw[6:0];
    negate neg (m128,n);
    byte_mux bm (m128,n,m128[7],data_byte);

endmodule
```

11.2.19 byte_mux.v – 8-Bit Multiplexer

```
`timescale 1ns / 1ps
module mux(
    input I0,
    input I1,
    input S,
    output OUT
    );

    assign OUT = (~S & I0) | (S & I1);

endmodule
```

11.2.20 mux.v – One-Bit Multiplexer

```
`timescale 1ns / 1ps
module mux(
    input I0,
    input I1,
    input S,
    output OUT
    );

    assign OUT = (~S & I0) | (S & I1);

endmodule
```

11.2.21 IQ_LPF.v – IQ Demodulator and LPF Master

```
`timescale 1ns / 1ps
module IQ_LPF(
    input [7:0] x,
```

```
    input Clk,
    input [1:0] S,
    input data_in_valid,
    output [23:0] xI,
    output [23:0] xQ,
    output data_out_valid
    );

    wire [7:0] Ipath;
    wire [7:0] Qpath;
    wire data_ready;

    IQdemod IQ_0 (x,S,Ipath,Qpath);

    fir_compiler_0 filter (Clk, data_in_valid,data_ready,
                           {Ipath,Qpath},data_out_valid,{xI,xQ});

endmodule
```

11.2.22 IQdemod.v – IQ Demodulator

```
`timescale 1ns / 1ps
module IQdemod(
    input [7:0] x,
    input [1:0] S,
    output [7:0] xI,
    output [7:0] xQ
    );

    wire [7:0] xIroute, xQroute;
    wire [7:0] I0,I1,Q0,Q1;
    wire [7:0] negI1,negQ0;

    byte_demux demux_0 (x,S[0],xIroute,xQroute);

    byte_demux demux_I (xIroute,S[1],I0,I1);
    byte_demux demux_Q (xQroute,S[1],Q0,Q1);

    negate negate_I (I1,negI1);
    negate negate_Q (Q0,negQ0);

    byte_mux mux_I (I0,negI1,S[1],xI);
    byte_mux mux_Q (negQ0,Q1,S[1],xQ);

endmodule
```

11.2.23 byte_demux.v – 8-Bit Demultiplexer

```
`timescale 1ns / 1ps
module byte_demux(
    input [7:0] I,
    input S,
    output [7:0] O0,
    output [7:0] O1
    );

    demux demux_0 (I[0],S,O0[0],O1[0]);
    demux demux_1 (I[1],S,O0[1],O1[1]);
```

```
    demux demux_2 (I[2],S,O0[2],O1[2]);
    demux demux_3 (I[3],S,O0[3],O1[3]);
    demux demux_4 (I[4],S,O0[4],O1[4]);
    demux demux_5 (I[5],S,O0[5],O1[5]);
    demux demux_6 (I[6],S,O0[6],O1[6]);
    demux demux_7 (I[7],S,O0[7],O1[7]);

endmodule
```

11.2.24 demux.v – One-Bit Demultiplexer

```
‘timescale 1ns / 1ps
module demux(
    input I,
    input S,
    output O0,
    output O1
    );

    assign O0 = ~S & I;
    assign O1 = S & I;

endmodule
```

12 Appendix – SDK Code

12.0.1 main.cc

```
#include "stdio.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xbasic_types.h"
#include "unistd.h"
#include <string>
#include <math.h>
#include <iostream>

const int FRAC_BITS = 43;

using namespace std;

void printBits(size_t const size, void const * const ptr);
void print2Bytes(size_t const size, void const * const ptr);
void sprintBits(string * str, size_t const size, void const * const ptr);
void sprint2Bytes(string * str, size_t const size, void const * const ptr);
int bitVal(char bit);
float str2val(char bin[], int fracBits);
void printNums(float *valsR1, float *valsC1, int AVG);
void toNums(const char r1_0[], const char r1_1[], const char r1_2[], const char r1_3
    [],
        const char r1_4[], const char r1_5[], const char r1_6[], const char r1_7[],
        const char r1_8[], const char r1_9[], float *valsR1, float *valsC1);

int main()
{
    int AVG = 1;
    int i = 0; int j = 0; // i counts array 1 R samples, j counts array 2 R samples
    u32 zero,one,two,three,four,five,six, seven,
    eight, nine, ten, eleven, twelve, thirteen,
    fourteen,fifteen,sixteen,seventeen, eighteen,
    nineteen, twenty, twentyone, twentytwo, twentythree, twentyfour,
    twentyfive, twentysix, twentyseven, twentyeight, twentynine,
    thirty, thirtyone, thirtytwo, thirtythree, thirtyfour,
    MAGIC;

    u32 zero2,one2,two2,three2,four2,five2,six2, seven2,
    eight2, nine2, ten2, eleven2, twelve2, thirteen2,
    fourteen2,fifteen2,sixteen2,seventeen2, eighteen2,
    nineteen2, twenty2, twentyone2, twentytwo2, twentythree2, twentyfour2,
    twentyfive2, twentysix2, twentyseven2, twentyeight2, twentynine2,
    thirty2, thirtyone2, thirtytwo2, thirtythree2, thirtyfour2,
    MAGIC2;

    xil_printf("Hello baby!\n");
    // Read the secret message 140 offset from baseaddr 0x43C00000

    float r1[] = {0,0,0,0,0,0,0,0,0,0};
    float c1[] = {0,0,0,0,0,0,0,0,0,0};

    float r2[] = {0,0,0,0,0,0,0,0,0,0};
    float c2[] = {0,0,0,0,0,0,0,0,0,0};
```

```
while(TRUE){


string R11 = "";string R12 = "";string R13 = "";string R14 = "";
string R22 = "";string R23 = "";string R24 = "";
string R33 = "";string R34 = "";
string R44 = "";

string R11_2 = "";string R12_2 = "";string R13_2 = "";string R14_2 = "";
string R22_2 = "";string R23_2 = "";string R24_2 = "";
string R33_2 = "";string R34_2 = "";
string R44_2 = "";

//Xil_Out32(XPAR_PL_AUTOCOR_0_S00_AXI_BASEADDR,0003);
zero = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR);
one = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 4);
two = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 8);
three = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 12);
four = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 16);
five = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 20);
six = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 24);
seven = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 28);
eight = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 32);
nine = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 36);
ten = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 40);
eleven = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 44);
twelve = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 48);
thirteen = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 52);
fourteen = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 56);
fifteen = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 60);
sixteen = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 64);
seventeen = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 68);
eighteen = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 72);
nineteen = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 76);
twenty = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 80);
twentyone = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 84);
twentytwo = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 88);
twentythree = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 92);
twentyfour = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 96);
twentyfive = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 100);
twentysix = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 104);
twentyseven = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 108);
twentyeight = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 112);
twentynine = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 116);
thirty = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 120);
thirtyone = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 124);
thirtytwo = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 128);
thirtythree = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 132);
thirtyfour = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 136);
MAGIC = Xil_In32(XPAR_RAGC_0_S00_AXI_BASEADDR + 140);

zero2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR);
one2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 4);
two2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 8);
three2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 12);
four2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 16);
five2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 20);
six2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 24);
```

```
seven2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 28);
eight2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 32);
nine2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 36);
ten2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 40);
eleven2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 44);
twelve2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 48);
thirteen2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 52);
fourteen2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 56);
fifteen2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 60);
sixteen2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 64);
seventeen2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 68);
eighteen2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 72);
nineteen2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 76);
twenty2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 80);
twentyone2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 84);
twentytwo2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 88);
twentythree2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 92);
twentyfour2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 96);
twentyfive2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 100);
twentysix2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 104);
twentyseven2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 108);
twentyeight2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 112);
twentynine2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 116);
thirty2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 120);
thirtyone2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 124);
thirtytwo2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 128);
thirtythree2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 132);
thirtyfour2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 136);
MAGIC2 = Xil_In32(XPAR_RAGC_1_S00_AXI_BASEADDR + 140);


if((MAGIC == 3) & (i<AVG)){

sprintBits(&R11,sizeof(three)-2,&three);
sprintBits(&R11,sizeof(two),&two);
sprintBits(&R11,sizeof(one),&one);
sprintBits(&R11,sizeof(zero),&zero);

//xil_printf("R11 = %s \n", R11.c_str());

sprintBits(&R12, sizeof(six),&six);
sprintBits(&R12,sizeof(five),&five);
sprintBits(&R12,sizeof(four),&four);
sprint2Bytes(&R12,sizeof(three),&three);

//xil_printf("R12 = %s \n", R12.c_str());

sprintBits(&R13,sizeof(ten)-2,&ten);
sprintBits(&R13,sizeof(nine),&nine);
sprintBits(&R13,sizeof(eight),&eight);
sprintBits(&R13,sizeof(seven),&seven);

//xil_printf("R13 = %s \n", R13.c_str());

sprintBits(&R14, sizeof(thirteen),&thirteen);
sprintBits(&R14, sizeof(twelve),&twelve);
sprintBits(&R14, sizeof(eleven),&eleven);
sprint2Bytes(&R14, sizeof(ten),&ten);
```

```
//xil_printf("R14 = %s \n", R14.c_str());

sprintBits(&R22, sizeof(seventeen)-2,&seventeen);
sprintBits(&R22, sizeof(sixteen),&sixteen);
sprintBits(&R22, sizeof(fifteen),&fifteen);
sprintBits(&R22, sizeof(fourteen),&fourteen);

//xil_printf("R22 = %s \n", R22.c_str());

sprintBits(&R23, sizeof(twenty),&twenty);
sprintBits(&R23, sizeof(nineteen),&nineteen);
sprintBits(&R23, sizeof(eighteen),&eighteen);
sprint2Bytes(&R23, sizeof(seventeen),&seventeen);

//xil_printf("R23 = %s \n", R23.c_str());

sprintBits(&R24, sizeof(twentyfour)-2,&twentyfour);
sprintBits(&R24, sizeof(twentythree),&twentythree);
sprintBits(&R24, sizeof(twentytwo),&twentytwo);
sprintBits(&R24, sizeof(twentyone),&twentyone);

//xil_printf("R24 = %s \n", R24.c_str());

sprintBits(&R33, sizeof(twentyseven),&twentyseven);
sprintBits(&R33, sizeof(twentysix),&twentysix);
sprintBits(&R33, sizeof(twentyfive),&twentyfive);
sprint2Bytes(&R33, sizeof(twentyfour),&twentyfour);

//xil_printf("R33 = %s \n", R33.c_str());

sprintBits(&R34, sizeof(thirtyone)-2,&thirtyone);
sprintBits(&R34, sizeof(thirty),&thirty);
sprintBits(&R34, sizeof(twentynine),&twentynine);
sprintBits(&R34, sizeof(twentyeight),&twentyeight);

//xil_printf("R34 = %s \n", R34.c_str());

sprintBits(&R44, sizeof(thirtyfour),&thirtyfour);
sprintBits(&R44, sizeof(thirtythree),&thirtythree);
sprintBits(&R44, sizeof(thirtytwo),&thirtytwo);
sprint2Bytes(&R44, sizeof(thirtyone),&thirtyone);

//xil_printf("R44 = %s \n", R44.c_str());

toNums(R11.c_str(),R12.c_str(),R13.c_str(),R14.c_str(),R22.c_str(),R23.c_str(),
    R24.c_str(),
                    R33.c_str(),R34.c_str(),R44.c_str(),r1,c1);

i++;

}

if((MAGIC2 == 3) & (j<AVG)){

    sprintBits(&R11_2,sizeof(three2)-2,&three2);
    sprintBits(&R11_2,sizeof(two2),&two2);
    sprintBits(&R11_2,sizeof(one2),&one2);
```

```
sprintBits(&R11_2,sizeof(zero2),&zero2);

//xil_printf("R11 = %s \n", R11.c_str());

sprintBits(&R12_2, sizeof(six2),&six2);
sprintBits(&R12_2,sizeof(five2),&five2);
sprintBits(&R12_2,sizeof(four2),&four2);
sprint2Bytes(&R12_2,sizeof(three2),&three2);

//xil_printf("R12 = %s \n", R12.c_str());

sprintBits(&R13_2,sizeof(ten2)-2,&ten2);
sprintBits(&R13_2,sizeof(nine2),&nine2);
sprintBits(&R13_2,sizeof(eight2),&eight2);
sprintBits(&R13_2,sizeof(seven2),&seven2);

//xil_printf("R13 = %s \n", R13.c_str());

sprintBits(&R14_2, sizeof(thirteen2),&thirteen2);
sprintBits(&R14_2, sizeof(twelve2),&twelve2);
sprintBits(&R14_2, sizeof(eleven2),&eleven2);
sprint2Bytes(&R14_2, sizeof(ten2),&ten2);

//xil_printf("R14 = %s \n", R14.c_str());

sprintBits(&R22_2, sizeof(seventeen2)-2,&seventeen2);
sprintBits(&R22_2, sizeof(sixteen2),&sixteen2);
sprintBits(&R22_2, sizeof(fifteen2),&fifteen2);
sprintBits(&R22_2, sizeof(fourteen2),&fourteen2);

//xil_printf("R22 = %s \n", R22.c_str());

sprintBits(&R23_2, sizeof(twenty2),&twenty2);
sprintBits(&R23_2, sizeof(nineteen2),&nineteen2);
sprintBits(&R23_2, sizeof(eighteen2),&eighteen2);
sprint2Bytes(&R23_2, sizeof(seventeen2),&seventeen2);

//xil_printf("R23 = %s \n", R23.c_str());

sprintBits(&R24_2, sizeof(twentyfour2)-2,&twentyfour2);
sprintBits(&R24_2, sizeof(twentythree2),&twentythree2);
sprintBits(&R24_2, sizeof(twentytwo2),&twentytwo2);
sprintBits(&R24_2, sizeof(twentyone2),&twentyone2);

//xil_printf("R24 = %s \n", R24.c_str());

sprintBits(&R33_2, sizeof(twentyseven2),&twentyseven2);
sprintBits(&R33_2, sizeof(twentysix2),&twentysix2);
sprintBits(&R33_2, sizeof(twentyfive2),&twentyfive2);
sprint2Bytes(&R33_2, sizeof(twentyfour2),&twentyfour2);

//xil_printf("R33 = %s \n", R33.c_str());

sprintBits(&R34_2, sizeof(thirtyone2)-2,&thirtyone2);
sprintBits(&R34_2, sizeof(thirty2),&thirty2);
sprintBits(&R34_2, sizeof(twentynine2),&twentynine2);
sprintBits(&R34_2, sizeof(twentyeight2),&twentyeight2);
```

```
        //xil_printf("R34 = %s \n", R34.c_str());

        sprintBits(&R44_2, sizeof(thirtyfour2),&thirtyfour2);
        sprintBits(&R44_2, sizeof(thirtythree2),&thirtythree2);
        sprintBits(&R44_2, sizeof(thirtytwo2),&thirtytwo2);
        sprint2Bytes(&R44_2, sizeof(thirtyone2),&thirtyone2);

        //xil_printf("R44 = %s \n", R44.c_str());

        toNums(R11_2.c_str(),R12_2.c_str(),R13_2.c_str(),R14_2.c_str(),R22_2.c_str()
            ,R23_2.c_str(),R24_2.c_str(),
                                R33_2.c_str(),R34_2.c_str(),R44_2.c_str(),r2,c2);

        j++;

        }

    if ((i>=AVG) & (j>=AVG))
    {
        printNums(r1,c1,AVG);printNums(r2,c2,AVG);cout << "\r\n";
        for(int k = 0;k<10;k++)
        {
        r1[k] = 0;
        c1[k] = 0;

        r2[k] = 0;
        c2[k] = 0;
        }
        i=0; j=0;
    }

    }
    return 0;
}

void sprintBits(string * str, size_t const size, void const * const ptr)
{
    unsigned char *b = (unsigned char*) ptr;
    unsigned char byte;
    int i, j;
    for (i=size-1;i>=0;i--)
    {
        for (j=7;j>=0;j--)
        {
            byte = (b[i] >> j) & 1;
            (*str).append(to_string(byte));
        }
    }
}

void sprint2Bytes(string * str, size_t const size, void const * const ptr)
{
    unsigned char *b = (unsigned char*) ptr;
    unsigned char byte;
    int i, j;
    i = size-1;
    for (i=size-1;i>=size-2;i--)
    {
```

```
        for (j=7;j>=0;j--)
        {
            byte = (b[i] >> j) & 1;
            (*str).append(to_string(byte));
        }
    }

    //xil_printf("\n");
}

void printNums(float *valsR1, float *valsC1, int AVG)
{

        float favg = (float) AVG;
        int is[] = {1,1,1,1,2,2,2,3,3,4};
        int js[] = {1,2,3,4,2,3,4,3,4,4};

        for(int i=0; i<10; i++){
            cout << fixed << "R" << is[i] <<""<<js[i]<< " = " << valsR1[i]/favg << "
                +" << valsC1[i]/favg <<"j;";
        }
        //cout<<"\r\n";
}

float str2val(char bin[], int fracBits){
    //Takes a string representing a 2s complement binary number and converts to a
        float.
        float val = 0;      //Value of binary string
        int len = strlen(bin) -1;   //How many bits to convert
        int bit;        //Bit value: either 0 or 1

        if(bin[0] == '0'){  //Positive value
            for (int i=len-1; i>=1; i--){
                bit = bitVal(bin[i]);
                val += bit*pow(2,len-1-i);
            }
        }
        else{           //Negative value
            for (int i=len-1; i>=1; i--){
                bit = -(bitVal(bin[i])-1);  //2s complement flip bits
                val += bit*pow(2,len-1-i);
            }
            val++;      //2s complement increment
            val *= -1;  //Negate
        }

        val /= pow(2,fracBits); //Scale appropriately
        return val;
    }


int bitVal(char bit){
//Takes a '0' or '1' char and returns as int.
    int val;
    if(bit == '0'){
        val = 0;
    }
    else{
```

```
        val = 1;
    }
    return val;
}

void toNums(const char r1_0[], const char r1_1[], const char r1_2[], const char r1_3
    [],
        const char r1_4[], const char r1_5[], const char r1_6[], const char r1_7[],
        const char r1_8[], const char r1_9[], float *valsR1, float *valsC1)
{

    int len = strlen(r1_0);

    //Break up char arrays into real and imaginary arrays
    char rr1_0[len/2+1], rc1_0[len/2+1];
    char rr1_1[len/2+1], rc1_1[len/2+1];
    char rr1_2[len/2+1], rc1_2[len/2+1];
    char rr1_3[len/2+1], rc1_3[len/2+1];
    char rr1_4[len/2+1], rc1_4[len/2+1];
    char rr1_5[len/2+1], rc1_5[len/2+1];
    char rr1_6[len/2+1], rc1_6[len/2+1];
    char rr1_7[len/2+1], rc1_7[len/2+1];
    char rr1_8[len/2+1], rc1_8[len/2+1];
    char rr1_9[len/2+1], rc1_9[len/2+1];

    for(int i=0; i<len/2; i++){
        rr1_0[i] = r1_0[i];     rc1_0[i] = r1_0[i+len/2];
        rr1_1[i] = r1_1[i];     rc1_1[i] = r1_1[i+len/2];
        rr1_2[i] = r1_2[i];     rc1_2[i] = r1_2[i+len/2];
        rr1_3[i] = r1_3[i];     rc1_3[i] = r1_3[i+len/2];
        rr1_4[i] = r1_4[i];     rc1_4[i] = r1_4[i+len/2];
        rr1_5[i] = r1_5[i];     rc1_5[i] = r1_5[i+len/2];
        rr1_6[i] = r1_6[i];     rc1_6[i] = r1_6[i+len/2];
        rr1_7[i] = r1_7[i];     rc1_7[i] = r1_7[i+len/2];
        rr1_8[i] = r1_8[i];     rc1_8[i] = r1_8[i+len/2];
        rr1_9[i] = r1_9[i];     rc1_9[i] = r1_9[i+len/2];
    }
    int t = len/2 + 1;
        rr1_0[t] = '\0';    rc1_0[t] = '\0';
        rr1_1[t] = '\0';    rc1_1[t] = '\0';
        rr1_2[t] = '\0';    rc1_2[t] = '\0';
        rr1_3[t] = '\0';    rc1_3[t] = '\0';
        rr1_4[t] = '\0';    rc1_4[t] = '\0';
        rr1_5[t] = '\0';    rc1_5[t] = '\0';
        rr1_6[t] = '\0';    rc1_6[t] = '\0';
        rr1_7[t] = '\0';    rc1_7[t] = '\0';
        rr1_8[t] = '\0';    rc1_8[t] = '\0';
        rr1_9[t] = '\0'; rc1_9[t] = '\0';

        float Rr1_0 = str2val(rr1_0, FRAC_BITS);
        float Rr1_1 = str2val(rr1_1, FRAC_BITS);
        float Rr1_2 = str2val(rr1_2, FRAC_BITS);
        float Rr1_3 = str2val(rr1_3, FRAC_BITS);
        float Rr1_4 = str2val(rr1_4, FRAC_BITS);
        float Rr1_5 = str2val(rr1_5, FRAC_BITS);
        float Rr1_6 = str2val(rr1_6, FRAC_BITS);
        float Rr1_7 = str2val(rr1_7, FRAC_BITS);
        float Rr1_8 = str2val(rr1_8, FRAC_BITS);
```

```
        float Rr1_9 = str2val(rr1_9, FRAC_BITS);
            //R1 imagianry part
        float Rc1_0 = str2val(rc1_0, FRAC_BITS);
        float Rc1_1 = str2val(rc1_1, FRAC_BITS);
        float Rc1_2 = str2val(rc1_2, FRAC_BITS);
        float Rc1_3 = str2val(rc1_3, FRAC_BITS);
        float Rc1_4 = str2val(rc1_4, FRAC_BITS);
        float Rc1_5 = str2val(rc1_5, FRAC_BITS);
        float Rc1_6 = str2val(rc1_6, FRAC_BITS);
        float Rc1_7 = str2val(rc1_7, FRAC_BITS);
        float Rc1_8 = str2val(rc1_8, FRAC_BITS);
        float Rc1_9 = str2val(rc1_9, FRAC_BITS);

        //Complex and Real opposite of what you would expect cause Karol's an idiot
        valsC1[0] = valsC1[0] + Rr1_0;
        valsC1[1] = valsC1[1] + Rr1_1;
        valsC1[2] = valsC1[2] + Rr1_2;
        valsC1[3] = valsC1[3] + Rr1_3;
        valsC1[4] = valsC1[4] + Rr1_4;
        valsC1[5] = valsC1[5] + Rr1_5;
        valsC1[6] = valsC1[6] + Rr1_6;
        valsC1[7] = valsC1[7] + Rr1_7;
        valsC1[8] = valsC1[8] + Rr1_8;
        valsC1[9] = valsC1[9] + Rr1_9;

        valsR1[0] = valsR1[0] + Rc1_0;
        valsR1[1] = valsR1[1] + Rc1_1;
        valsR1[2] = valsR1[2] + Rc1_2;
        valsR1[3] = valsR1[3] + Rc1_3;
        valsR1[4] = valsR1[4] + Rc1_4;
        valsR1[5] = valsR1[5] + Rc1_5;
        valsR1[6] = valsR1[6] + Rc1_6;
        valsR1[7] = valsR1[7] + Rc1_7;
        valsR1[8] = valsR1[8] + Rc1_8;
        valsR1[9] = valsR1[9] + Rc1_9;

        //int is[] = {1,1,1,1,2,2,2,3,3,4};
        //int js[] = {1,2,3,4,2,3,4,3,4,4};

        //for(int i=0; i<10; i++){
        //  cout << fixed << "R" << is[i] <<""<<js[i]<< " = " << valsR1[i] << "+" <<
             valsC1[i] <<"j;";
        //}
        //cout<<"\r\n";
}
```

13 Appendix – Host Computer Code

This appendix contains all C++ code and C♯ implemented on the host computer.

13.1 Unity

13.1.1 MyMessageListener.cs – Read Serial Messages from FPGA

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class MyMessageListener : MonoBehaviour
{
    public string FPGAin;
    // Use this for initialization
    void Start()
    {
    }
    // Update is called once per frame
    void Update()
    {
    }
    // Invoked when a line of data is received from the serial device.
    void OnMessageArrived(string msg)
    {
        //Debug.Log("Arrived: " + msg);
        FPGAin = msg;
        //Debug.Log("Arrived: " + FPGAin);
    }
    // Invoked when a connect/disconnect event occurs. The parameter 'success'
    // will be 'true' upon connection, and 'false' upon disconnection or
    // failure to connect.
    void OnConnectionEvent(bool success)
    {
        Debug.Log(success ? "Device connected" : "Device disconnected");
    }
}
```

13.1.2 CameraMove.cs – Camera Control

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraMove : MonoBehaviour
{
    private readonly float rSpd = 10.0f;
    private readonly float tSpd = 0.5f;
    // Start is called before the first frame update
    void Start()
    {
    }

    private float x;
    private float y;
    private Vector3 rotateValue;
```

```
    void Update()
    {
        if (Input.GetKey(KeyCode.Mouse0))
        {
            y = Input.GetAxis("Mouse X");
            x = Input.GetAxis("Mouse Y");
            rotateValue = new Vector3(x * -rSpd, y * rSpd, 0);
            transform.eulerAngles = transform.eulerAngles - rotateValue;
        }

        if (Input.GetKey(KeyCode.UpArrow))
        {
            transform.Translate(Vector3.forward * tSpd) ;
        }
        if (Input.GetKey(KeyCode.DownArrow))
        {
            transform.Translate(Vector3.forward * -tSpd);
        }
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            transform.Translate(Vector3.right * -tSpd);
        }
        if (Input.GetKey(KeyCode.RightArrow))
        {
            transform.Translate(Vector3.right * tSpd);
        }
        if (Input.GetKey(KeyCode.Space))
        {
            transform.position = new Vector3(0, 0.5f, -10.0f); ;
        }
    }
}
```

13.1.3 PlayerController.cs – Transmitter Location Update

```
using System;
using System.Runtime.InteropServices;
using UnityEngine;

public class PlayerController : MonoBehaviour
{

    [DllImport("TestEigenDLL 1", EntryPoint = "R2LOC")]
    public static extern void R2LOC(
    float r1_0, float r1_1, float r1_2, float r1_3, float r1_4,
    float r1_5, float r1_6, float r1_7, float r1_8, float r1_9,
    float r2_0, float r2_1, float r2_2, float r2_3, float r2_4,
    float r2_5, float r2_6, float r2_7, float r2_8, float r2_9,
    float i1_0, float i1_1, float i1_2, float i1_3, float i1_4,
    float i1_5, float i1_6, float i1_7, float i1_8, float i1_9,
    float i2_0, float i2_1, float i2_2, float i2_3, float i2_4,
    float i2_5, float i2_6, float i2_7, float i2_8, float i2_9,
    float[] angles1, float[] angles2, float[] xyz);

    private Transform t;
    private MyMessageListener ml;
    private int num = 0;
    public int AVG = 1;
```

```
void Start()
{
    t = GetComponent<Transform>();
    ml = gameObject.GetComponentInChildren<MyMessageListener>();
}

void Update()
{
    //string test = "R11 = 0.01+1j;R12 = 0.01+1j;R13 = 0.02+2j;R14 = 0.03+3j;R22
         = 0.05+0j;R23 = 0.08+4j;R24 = 0.13+5j;R33 = 0.21+0j;R34 = 0.34+6j;R44 =
         0.55+0j;";
    float [] real = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; float [] imaginary = { 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0 };
    float[] real2 = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; float[] imaginary2 = { 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0 };
    string test = ml.FPGAin;

    //if (Input.GetKey(KeyCode.P))
    //{
        int ind = 6;
        for (int i = 0; i < 10; i++)
        {
            string tempR = "";
            while (test[ind] != '+')
            {
                tempR = tempR + test[ind];
                ind++;
            }
            real[i] = real[i] + (float)Convert.ToDouble(tempR)/AVG;
            ind++;
            string tempI = "";
            while (test[ind] != 'j')
            {
                tempI = tempI + test[ind];
                ind++;
            }
            imaginary[i] = imaginary[i] + (float)Convert.ToDouble(tempI)/AVG;
            ind = ind + 8;
        }
        //Debug.Log("R11 = " + real[0] + "+" + imaginary[0] + "j");
        //float[] valsR1 = { .3650F, .0364F, .2522F, .0207F, .3533F, .0427F,
            .2515F, .3450F, .0305F, .3421F };
        //float[] valsC1 = { 0.000F, -.2565F, -.0124F, -.2541F, .0000F, .2467F,
            -.0100F, .0000F, -.2384F, .0000F };
        //float[] valsR2 = { .3650F, .0364F, .2522F, .0207F, .3533F, .0427F,
            .2515F, .3450F, .0305F, .3421F };
        //float[] valsC2 = { .0000F, -.2565F, -.0124F, -.2541F, .0000F, .2467F,
            -.0100F, .0000F, -.2384F, .0000F };

        for (int i = 0; i < 10; i++)
        {
            string tempR = "";
            while (test[ind] != '+')
            {
                tempR = tempR + test[ind];
                ind++;
            }
            real2[i] = real2[i] + (float)Convert.ToDouble(tempR) / AVG;
```

```
                ind++;
                string tempI = "";
                while (test[ind] != 'j')
                {
                    tempI = tempI + test[ind];
                    ind++;
                }
                imaginary2[i] = imaginary2[i] + (float)Convert.ToDouble(tempI) / AVG
                    ;
                ind = ind + 8;
            }

        //Debug.Log("R11 = " + real2[0] + "+" + imaginary2[0] + "j");

        float[] angles1 = { 0, 0 }; float[] angles2 = { 0, 0 }; float[] xyz = { 0,
            0, 0 };

        if (num == AVG)
        {
            num = 0;
            R2LOC(real[0], real[1], real[2], real[3], real[4], real[5], real[6],
                real[7], real[8], real[9],
    real2[0], real2[1], real2[2], real2[3], real2[4], real2[5], real2[6], real2[7],
        real2[8], real2[9],
    imaginary[0], imaginary[1], imaginary[2], imaginary[3], imaginary[4], imaginary
        [5], imaginary[6], imaginary[7], imaginary[8], imaginary[9],
    imaginary2[0], imaginary2[1], imaginary2[2], imaginary2[3], imaginary2[4],
        imaginary2[5], imaginary2[6], imaginary2[7], imaginary2[8], imaginary2[9],
    angles1, angles2, xyz);
            //angles1[0] = angles1[0] ;

            if (Input.GetKey(KeyCode.A))
            {
                Debug.Log("Theta: " + angles1[0] * 180 / 3.14159 + ", Phi: " +
                    angles1[1] * 180 / 3.14159);
                t.position = new Vector3(5 * Mathf.Sin(angles1[1]) * Mathf.Cos(
                    angles1[0]), 5 * Mathf.Cos(angles1[1]), 5 * Mathf.Sin(angles1
                    [1]) * Mathf.Sin(angles1[0]));
            }
            else if (Input.GetKey(KeyCode.B))
            {
                Debug.Log("Theta: " + angles2[0] * 180 / 3.14159 + ", Phi: " +
                    angles2[1] * 180 / 3.14159);
                t.position = new Vector3(5 * Mathf.Sin(angles2[1]) * Mathf.Cos(
                    angles2[0]), 5 * Mathf.Cos(angles2[1]), 5 * Mathf.Sin(angles2
                    [1]) * Mathf.Sin(angles2[0]));
            }
            else
            {
                Debug.Log("x: " + xyz[0] + ", y: " + xyz[2] + ", z: " + xyz[1]);
                t.position = new Vector3(xyz[0], xyz[2], xyz[1]);
            }
        }
        num++;
        //}
    }
}
```

13.2 C++ DLL

13.2.1 TestEigenDLL.cpp – Declaration of Localization Function

```
#include "stdafx.h"
#include "TestEigenDll.h"


void R2LOC(
    float r1_0, float r1_1, float r1_2, float r1_3, float r1_4,
    float r1_5, float r1_6, float r1_7, float r1_8, float r1_9,
    float r2_0, float r2_1, float r2_2, float r2_3, float r2_4,
    float r2_5, float r2_6, float r2_7, float r2_8, float r2_9,
    float i1_0, float i1_1, float i1_2, float i1_3, float i1_4,
    float i1_5, float i1_6, float i1_7, float i1_8, float i1_9,
    float i2_0, float i2_1, float i2_2, float i2_3, float i2_4,
    float i2_5, float i2_6, float i2_7, float i2_8, float i2_9,
    float *angles1, float *angles2, float *xyz)
{
    float array1[ANTS][3];      //ANTS antennas, (x,y,z) coordinates
    float array2[ANTS][3];      //^ but for array 2

    //Array 1
    array1[0][0] = 0.0*D;
    array1[0][1] = 0.0*D;
    array1[0][2] = 0.0;

    array1[1][0] = 1.0*D;
    array1[1][1] = 0.0*D;
    array1[1][2] = 0.0;

    array1[2][0] = 0.0*D;
    array1[2][1] = 1.0*D;
    array1[2][2] = 0.0;

    array1[3][0] = 1.0*D;
    array1[3][1] = 1.0*D;
    array1[3][2] = 0.0;

    //Array 2
    array2[0][0] = 2.0;
    array2[0][1] = 0.0;
    array2[0][2] = 0.0;

    array2[1][0] = 2.0 + 1.0*D;
    array2[1][1] = 0.0;
    array2[1][2] = 0.0;

    array2[2][0] = 2.0;
    array2[2][1] = 1.0*D;
    array2[2][2] = 0.0;

    array2[3][0] = 2.0 + 1.0*D;
    array2[3][1] = 1.0*D;
    array2[3][2] = 0.0;

    float *a1, *a2;
    a1 = new float[ANTS * 3];
```

```
a2 = new float[ANTS * 3];
a1 = &array1[0][0];
a2 = &array2[0][0];


float valsR1[] = { r1_0, r1_1, r1_2, r1_3, r1_4,
              r1_5, r1_6, r1_7, r1_8, r1_9 };
float valsC1[] = { i1_0, i1_1, i1_2, i1_3, i1_4,
                   i1_5, i1_6, i1_7, i1_8, i1_9 };

float valsR2[] = { r2_0, r2_1, r2_2, r2_3, r2_4,
                   r2_5, r2_6, r2_7, r2_8, r2_9 };
float valsC2[] = { i2_0, i2_1, i2_2, i2_3, i2_4,
                   i2_5, i2_6, i2_7, i2_8, i2_9 };

_Fcomplex *R1;
R1 = new _Fcomplex[ANTS*ANTS];
_Fcomplex *R2;
R2 = new _Fcomplex[ANTS*ANTS];

vec2autocorr(valsR1, valsC1, R1);
vec2autocorr(valsR2, valsC2, R2);

MatrixXcf *eigvecs1, *eigvecs2;
MatrixXf *eigvals1, *eigvals2;
eigvecs1 = new MatrixXcf;
eigvecs2 = new MatrixXcf;
eigvals1 = new MatrixXf;
eigvals2 = new MatrixXf;

autocorr2eig(R1, eigvecs1, eigvals1);
autocorr2eig(R2, eigvecs2, eigvals2);

MatrixXcf *subspace1, *subspace2;
subspace1 = new MatrixXcf;
subspace2 = new MatrixXcf;

subspaceMat(eigvals1, eigvecs1, subspace1);
subspaceMat(eigvals2, eigvecs2, subspace2);

MatrixXf *S_music1, *thetas1, *phis1;
S_music1 = new MatrixXf;        //Music Spectrum values
thetas1 = new MatrixXf;         //Theta grid
phis1 = new MatrixXf;           //Phi grid

MatrixXf *S_music2, *thetas2, *phis2;
S_music2 = new MatrixXf;        //Music Spectrum values
thetas2 = new MatrixXf;         //Theta grid
phis2 = new MatrixXf;           //Phi grid

musicSpectrum(subspace1, a1, S_music1, thetas1, phis1);
musicSpectrum(subspace2, a2, S_music2, thetas2, phis2);

float *thLocs1, *phLocs1;
thLocs1 = new float[TAGS];  //Theta values corresponding to peaks
phLocs1 = new float[TAGS];  //Phi values corresponding to peaks
findPeaks(S_music1, thetas1, phis1, thLocs1, phLocs1);
```

```
    float *thLocs2, *phLocs2;
    thLocs2 = new float[TAGS];  //Theta values corresponding to peaks
    phLocs2 = new float[TAGS];  //Phi values corresponding to peaks
    findPeaks(S_music2, thetas2, phis2, thLocs2, phLocs2);

    *angles1 = *thLocs1;
    *(angles1 + 1) = *phLocs1;
    *angles2 = *thLocs2;
    *(angles2 + 1) = *phLocs2;

    float *dist;
    dist = new float;
    Vector3f *midpoint;
    midpoint = new Vector3f;

    Vector3f *locations;
    locations = new Vector3f[TAGS];
    bestLocal(a1, thLocs1, phLocs1, a2, thLocs2, phLocs2, locations);
    *xyz = (*locations)[0];
    *(xyz + 1) = (*locations)[1];
    *(xyz + 2) = (*locations)[2];
}
```

13.2.2 TestEigenDLL.h – All Auxiliary Functions

```
#define TESTEIGENDLL_API __declspec(dllexport)
#define _USE_MATH_DEFINES
#include<complex.h>
#include "../Eigen/Dense"
#include "../Eigen/Geometry"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <iostream>
#include <utility>
#include <cfloat>
#include <vector>

using namespace std;
using Eigen::MatrixXcf;
using Eigen::VectorXcf;
using Eigen::VectorXd;
using Eigen::MatrixXf;
using Eigen::Vector3f;

const int ANTS = 4;
const int TAGS = 1;
const int GRID_RES = 200;
const int gg = GRID_RES * GRID_RES;

const float FREQ = 915000000;       //Tag freq
const float DOL = 0.5;              //Distance over lambda
const float LAMBDA = 300000000 / FREQ;  //Wavelength
const float D = DOL * LAMBDA;               //Distance
const float oneNorm = 5.0*M_PI / 180.0;

extern "C" {
```

```
    TESTEIGENDLL_API void R2LOC(float r1_0, float r1_1, float r1_2, float r1_3,
        float r1_4,
        float r1_5, float r1_6, float r1_7, float r1_8, float r1_9,
        float r2_0, float r2_1, float r2_2, float r2_3, float r2_4,
        float r2_5, float r2_6, float r2_7, float r2_8, float r2_9,
        float i1_0, float i1_1, float i1_2, float i1_3, float i1_4,
        float i1_5, float i1_6, float i1_7, float i1_8, float i1_9,
        float i2_0, float i2_1, float i2_2, float i2_3, float i2_4,
        float i2_5, float i2_6, float i2_7, float i2_8, float i2_9,
        float *angles1, float *angles2, float *xyz);
}

bool comparator(pair <float, int> p1, pair <float, int> p2) {
    return (p1.first > p2.first);
}

void vec2autocorr(float valsReal[], float valsComp[], _Fcomplex *R) {
    /* Returns autocorrelation matrix formed from the values in vals.
     *
     * Inputs:
     *  valsReal: Array of the real portion of values containing the upper
                  traingular portion
     *         [r11, r12, r13, r22, r23, r33] are the indexes for the 3x3 R case.
     *  valsComp: Array of the complex portion of values containing the upper
                  traingular portion
     *  ants    : Number of antennas (aka size of square matrix)
     *
     * Outputs:
     *  *R  : Pointer to a 2D array of complex values
     */
    _Fcomplex temp;
    for (int i = 0; i < ANTS; i++) {
        for (int j = 0; j < ANTS; j++) {
            if (j >= i) {   //Upper triangular
                int ind = i * ANTS - i * (i + 1) / 2 + j;
                temp = { valsReal[ind],valsComp[ind] }; //Form the complex number. I
                     is imag unit
                *(R + ANTS * i + j) = temp;
            }
            else {      //Lower triangular
                int ind = j * ANTS - j * (j + 1) / 2 + i;
                temp = { valsReal[ind],-valsComp[ind] };    //Form the complex
                    number
                *(R + ANTS * i + j) = temp;
            }
        }
    }
}

void autocorr2eig(_Fcomplex *R, MatrixXcf *eigmat, MatrixXf *eigvals) {
    /*Takes an autocorrelation matrix and returns the eigenvectors and eigenvalues.
    Inputs:
        *R      : Pointer to R matrix values
        *eigmat : Pointer to eigenvectors stored in matrix
        *eigvals: Pointer to eigenvalues sotred in matrix (row vector)
    */
    MatrixXcf Rmat(ANTS, ANTS);
```

```
    //Place R values into appropriate data type
    for (int i = 0; i < ANTS*ANTS; i++) {
        int row = i / ANTS;
        int col = i % ANTS;
        Rmat(row, col) = *(R + i);
    }

    Eigen::SelfAdjointEigenSolver<MatrixXcf> es(Rmat);
    *eigvals = es.eigenvalues();
    *eigmat = es.eigenvectors();

    //  std::cout << "Eigen Values: \n" << *eigvals << std::endl;
    //  std::cout << "\nEigen Vectors: \n" << *eigmat << std::endl;
    //  std::cout << "\nLambda*v: \n" << (*eigvals)(0)*((*eigmat).col(0)) << std::
        endl;
    //  std::cout << "\nR*eig: \n" << (Rmat)*((*eigmat).col(0)) << std::endl;
}

void centroid(float *points, int ants, float *center) {
    /* Finds the centroid of the points.
     *
     * Inputs:
     *  *points : Pointer to a 2D array of size [ants][3] where each row contains (x
         ,y,z) coordinate
     *  ants    : Number of antennas to find the centroid of
     *
     * Outputs:
     *  *center : Pointer to a 1D array of size [3] containing (x,y,z) coordinates
         of centroid
     */
    *center = 0.0;
    *(center + 1) = 0.0;
    *(center + 2) = 0.0;
    //  cout << "AAAA: " << &points << endl;
    for (int i = 0; i < ants; i++) {
        for (int j = 0; j < 3; j++) {
            *(center + j) += *(points + 3 * i + j) / (double)ants;
        }
    }

    //  printf("X: %f, Y: %f, Z: %f\n", *(center), *(center+1), *(center+2));
}

void subspaceMat(MatrixXf *eigvals, MatrixXcf *eigvecs, MatrixXcf *subspace) {
    /*Takes eigenvectors and eigenvalues and returns the appropriate Subspace matrix
        .
    Inputs:
        *eigvals    : Pointer to eigenvalues
        *eigvecs    : Pointer to eigenvector matrix
    Ouputs:
        *subspace   : Pointer to noise subspace matrix
    */
    pair <float, int> lambs[ANTS];
    for (int i = 0; i < ANTS; i++) {
        lambs[i].first = (*eigvals)(i);
        lambs[i].second = i;
    }
```

```
    int n = sizeof(lambs) / sizeof(lambs[0]);
    std::sort(lambs, lambs + n, comparator);

    /*
        for(int i=0; i<ants; i++){
            cout << "Lambs at " << i << ": First  " << lambs[i].first
                 << ", Second " << lambs[i].second << endl;
        }
    */
    MatrixXcf subs(ANTS, ANTS - TAGS);

    for (int i = 0; i < ANTS - TAGS; i++) {
        int ind = lambs[i + TAGS].second;
        subs.col(i) = (*eigvecs).col(ind);
    }

    (*subspace) = subs * subs.adjoint();

    //  std::cout << "\nSubs :\n" << *subspace << std::endl;
}



void musicSpectrum(MatrixXcf *subspace, float *antPos, MatrixXf *S_music, MatrixXf *
    thetas, MatrixXf *phis) {
    /*Calculates the music spectrum based on the noise subspace matrix
    Inputs:
        *subspace   : Noise subspace matrix
        *antPos     : [ants][3] pointer to antenna coordinates
    Outputs:
        *S_music    : [gridRes][girdRes] matrix containing values of the spectrum
                        theta in first dim, phi in second dim
        *thetas     : [gridRes][gridRes] matrix containing theta values
        *phis       : [gridRes][gridRes] matrix containing phis values
    */

    MatrixXcf music_spec(GRID_RES, GRID_RES);   //To be filled in and then sent back

    float *center;
    center = new float[3];
    //  cout << "Antpos: " <<  &antPos << endl;
    //  printf("Center %f, %f, %f\n", *center, *(center+1), *(center+2));
    centroid(antPos, ANTS, center);

    //  printf("Center %f, %f, %f\n", *center, *(center+1), *(center+2));

    MatrixXf centered(3, ANTS); //Shifted antPos by center
    for (int i = 0; i < ANTS; i++) {
        for (int j = 0; j < 3; j++) {
            centered(j, i) = *(antPos + 3 * i + j) - *(center + j);
        }
    }

    //  cout << "Ant_locs\n" <<  centered << endl;

    MatrixXf dir_vec(1, 3);     //Directional Vector
    MatrixXcf steerRow(1, ANTS);    //Steering vector
```

```
    float th, ph;
    MatrixXf theta(GRID_RES, GRID_RES), phi(GRID_RES, GRID_RES);

    float kwav = 2.0*M_PI / LAMBDA;
    MatrixXcf temp1(1, 1);
    for (int i = 0; i < GRID_RES; i++) {
        for (int j = 0; j < GRID_RES; j++) {
            th = -M_PI + i * (2 * M_PI) / (float)GRID_RES;
            ph = j * (M_PI) / (float)(GRID_RES * 2);
            theta(i, j) = th;
            phi(i, j) = ph;
            dir_vec(0, 0) = sin(ph)*cos(th);        //ak x component
            dir_vec(0, 1) = sin(ph)*sin(th);        //ak y component
            dir_vec(0, 2) = cos(ph);                //ak z component

            MatrixXcf Ifm(1,1);
            _Fcomplex If = { 0,1 };
            Ifm(0, 0) = If;
            _Fcomplex neg1 = { -1,0 };

            steerRow = -((float)kwav)*Ifm*dir_vec*centered;
            for (int k = 0; k < ANTS; k++) {
                steerRow(0, k) = exp(steerRow(0, k)) / ((float)(sqrt(ANTS)));
            }

            temp1(0, 0) = ((steerRow.conjugate())*(*subspace)*(steerRow.transpose())
                )(0, 0);
            music_spec(i, j) = (float)1.0 / temp1(0, 0);

            //  cout << "th: " << th << ", ph: " << ph << ", vect: " << steerRow <<
                endl;
        }
    }

    *thetas = theta;
    *phis = phi;
    MatrixXf temp(GRID_RES, GRID_RES);

    //Take abs of music spec and print it
    for (int i = 0; i < GRID_RES; i++) {
        for (int j = 0; j < GRID_RES; j++) {
            temp(i, j) = ((float)20)*log10(abs(music_spec(i, j)));
            //temp(i,j) = abs(music_spec(i,j));
//          printf("%f\n",temp(i,j));
        }
    }
    *S_music = temp;
}

void findPeaks(MatrixXf *S_music, MatrixXf *th, MatrixXf *ph, float *thetas,
    float *phis) {
    /* Find the theta and phi values of the peaks of the Music Spectrum.
     *
     * Inputs:
     *  *S_music: Music spectrum
     *  *th     : Theta value grid
     *  *ph     : Phi value grid
     *
```

```
 * Outputs:
 *  *thetas : theta values of the peaks
 *  *phis   : phi values of the peals
 */
if (TAGS == 1) {
    float thMax;
    float phMax;
    float maxPeak = FLT_MIN;

    for (int i = 0; i < GRID_RES; i++) {
        for (int j = 0; j < GRID_RES; j++) {
            if ((*S_music)(i, j) > maxPeak) {
                thMax = (*th)(i, j);
                phMax = (*ph)(i, j);
                maxPeak = (*S_music)(i, j);
            }
        }
    }
    *thetas = thMax;
    *phis = phMax;

    //cout << "Theta: " << thMax << ", Phi: " << phMax << endl;
}
else {

    bool isPeak = false;
    int im, ip, jm, jp, k;
    im = 0;     //Previous Row
    ip = 0;     //Next Row
    jm = 0;     //Previous Column
    jp = 0;     //Next Column
    k = 0;

    float peaks[gg];
    float thTemp[gg];
    float phTemp[gg];

    for (int i = 0; i < GRID_RES; i++) {        //Theta loop (rows constant)
        //Handle edge cases for rows
        if (i == 0) {
            im = GRID_RES - 1;      //Wrap around on the theta
            ip = 1;
        }
        else if (i == (GRID_RES - 1)) {
            im = i - 1;
            ip = 0;             //Wrap around on the theta
        }
        else {
            im = i - 1;
            ip = i + 1;
        }

        for (int j = 0; j < GRID_RES; j++) {    //Phi loop (cols constant)
            //Handle edge cases for rows
            if (j == 0) {
                jm = 0;
                jp = 1;
            }
```

```
else if (j == (GRID_RES - 1)) {
    jm = j - 1;
    jp = GRID_RES - 1;
}
else {
    jm = j - 1;
    jp = j + 1;
}

//Check if current value is larger or equal to neighbors
isPeak = ((*S_music)(im, j) <= (*S_music)(i, j))
    & ((*S_music)(i, jm) <= (*S_music)(i, j))
    & ((*S_music)(ip, j) <= (*S_music)(i, j))
    & ((*S_music)(i, jp) <= (*S_music)(i, j));

//Store peak value and corresponding theta and phi
bool isok = true;
float differ;
if (isPeak) {
    for (int n = 0; n < k; n++) {
        //                  float tempth = thTemp[n];
        //                  if(thTemp[n] > 0){
        //                      tempth = thTemp[n] - M_PI;
        //                  }
        //                  else{
        //                      tempth = thTemp[n] + M_PI;
        //                  }
        differ = abs(thTemp[n] - (*th)(i, j)) + abs(phTemp[n] - (*ph
            )(i, j));
        //      cout << "Differ = " << differ << ", k = " << k <<
            endl;
        isok = isok && (differ > oneNorm);
    }

    if ((*th)(i, j) < (-M_PI + oneNorm)) {
        for (int n = 0; n < k; n++) {
            differ = abs(thTemp[n] + (float)(2.0*M_PI) - (*th)(i, j)
                + abs(phTemp[n] - (*ph)(i, j)));
            isok = isok && (differ > oneNorm);
        }
    }
    else if ((*th)(i, j) < (M_PI - oneNorm)) {
        for (int n = 0; n < k; n++) {
            differ = abs(thTemp[n] - (float)(2.0*M_PI) - (*th)(i, j)
                + abs(phTemp[n] - (*ph)(i, j)));
            isok = isok && (differ > oneNorm);
        }
    }

    if (isok) {
        //      cout << "End my misery!\n";
        peaks[k] = (*S_music)(i, j);
        thTemp[k] = (*th)(i, j);
        phTemp[k] = (*ph)(i, j);

        //                  if(thTemp[k] > 0){
        //                      thTemp[k] = thTemp[k] - M_PI;
        //                  }
```

```
                //                  else{
                //                      thTemp[k] = thTemp[k] + M_PI;
                //                  }
                k++;
            }
        }

        //              cout << "Peak " << k <<" found! Val: " << peaks[k]
            << ", th: " << thTemp[k] << ", ph: " << phTemp[k] << endl;
    }
}

//Sort the peaks so that highest peak is first (Stronget Signal)
pair <float, int> locs[gg];
for (int i = 0; i < k; i++) {
    locs[i].first = peaks[i];
    locs[i].second = i;
}
int n = sizeof(locs) / sizeof(locs[0]);
std::sort(locs, locs + n, comparator);

float allOrdTh[gg], allOrdPh[gg];
for (int i = 0; i < k; i++) {
    allOrdTh[i] = thTemp[locs[i].second];
    allOrdPh[i] = phTemp[locs[i].second];
}

//The above two for loops organize all the peaks and corresponding in angles
     in
//order of highest peaks to lowest peaks.


/*
for(int i=0; i<k; i++){
    cout << "value: " << locs[i].first << " , Position: " << locs[i].second
        << endl;
}
*/
//Take the n largest peaks where n is the number of tags;
float ordTh[TAGS];      //Ordered thetas
float ordPh[TAGS];      //Ordered phis


for (int i = 0; i < TAGS; i++) {
    //          cout << "Location index: " << locs[i].second << endl;
    //          cout <<"\t TH: " << thTemp[locs[i].second]*(180/M_PI) <<
        endl;
    //          cout <<"\t PH: " << phTemp[locs[i].second]*(180/M_PI) <<
        endl;
    ordTh[i] = thTemp[locs[i].second];
    ordPh[i] = phTemp[locs[i].second];
}

*thetas = ordTh[0];
*phis = ordPh[0];

/*
for(int i=0; i<tags; i++){
```

```
            cout << "ThPh" << i << ": " << ordTh[i] << ", " << ordPh[i] << endl;
        }
        */
    }
}


void ang2loc(float *antloc1, float th1, float ph1, float *antloc2, float th2,
    float ph2, float *dist, Vector3f *midpoint) {
    /* Reutrns the minimum distance between the two rays pertruding from the center
       of the antenna arrays specified by the angles.
    Inputs:
        *antloc : [ANTS][3] Contains the coordiantes of an antenna array
        th      : theta value corresponding to array
        ph      : phi value corresponding to array
    Outputs:
        *dist   : Minimum distance between the two rays
        *midpoint   : Midpoint between the two lines
    */

    float *c1, *c2;
    c1 = new float[3];  //Center of array 1
    c2 = new float[3];  //Center of array 2

    centroid(antloc1, ANTS, c1);
    centroid(antloc2, ANTS, c2);

    //Store centers in vector objects
    Vector3f center1, center2;
    for (int i = 0; i < 3; i++) {
        center1(i) = *(c1 + i);
        center2(i) = *(c2 + i);
    }

    //  cout << "\nAngles1: Th = " << th1 << ", Ph = " << ph1 << endl;
    //  cout << "\nAngles2: Th = " << th2 << ", Ph = " << ph2 << endl;

    Vector3f dir1, dir2;    //Direction of ray
    dir1(0) = cos(th1);     //x component
    dir1(1) = sin(th1);     //y component
    dir1(2) = cos(ph1) / sin(ph1);  //z component
    dir2(0) = cos(th2);     //x component
    dir2(1) = sin(th2);     //y component
    dir2(2) = cos(ph2) / sin(ph2);  //z component

//  cout << "\nDirection 1:\n" << dir1 << endl;
//  cout << "\nDirection 2:\n" << dir2 << endl;

    Vector3f nor1, nor2, distNor;       //Normal vectors
    nor1 = dir1.cross(dir2.cross(dir1));        //For center calc
    nor2 = dir2.cross(dir1.cross(dir2));        //For center calc
    distNor = (dir1.cross(dir2)).normalized();  //For distance calc

//  cout << "\nNormal 1:\n" << nor1 << endl;
//  cout << "\nNormal 2:\n" << nor2 << endl;
//  cout << "\nDistance Normal:\n" << distNor << endl;

    float distance = abs(distNor.dot(center1 - center2));       //Distance between 2
```

```
         skew lines
    *dist = distance;

    Vector3f point1, point2;    //Points on line1 and line2 closest to line 2 and
        line 1 respectively
    point1 = center1 + ((center2 - center1).dot(nor2) / dir1.dot(nor2))*dir1;
    point2 = center2 + ((center1 - center2).dot(nor1) / dir2.dot(nor1))*dir2;

    //  cout << "\nPoint 1:\n" << point1 << endl;
    //  cout << "\nPoint 2:\n" << point2 << endl;

    Vector3f mid;
    mid = (point1 + point2) / (float)(2);
    *midpoint = mid;
}

void bestLocal(float *antloc1, float *thLocs1, float *phLocs1, float *antloc2,
    float *thLocs2, float *phLocs2, Vector3f *locations) {
    //Takes the thetas and phis from each array and finds the best matching to
    //localize the tags.
    vector <int> avail;     //Unpaired AOA indices
    pair <int, int> matches[TAGS];

    for (int i = 0; i < TAGS; i++) {
        matches[i].first = i;
        avail.push_back(i);
    }

    float minDist = FLT_MAX;    //Smallest distance found so far
    float *dist;                //Current distance found
    dist = new float;
    Vector3f *midpoint;         //Current localized point
    midpoint = new Vector3f;
    Vector3f *loc;              //Best localized point
    loc = new Vector3f;
    int bestInd;                //Index of best match

    Vector3f locals[TAGS];      //Array of tags locations

    for (int i = 0; i < TAGS; i++) {
        for (int j = 0; j < TAGS - i; j++) {
            ang2loc(antloc1, *(thLocs1 + i), *(phLocs1 + i), antloc2,
                *(thLocs2 + avail.at(j)), *(phLocs2 + avail.at(j)), dist, midpoint);
            if (*dist < minDist) {
                minDist = *dist;
                bestInd = j;
                loc = midpoint;

                //  cout << "MIDPOINT " << *midpoint << endl;
            };
        }
        matches[i].second = avail.at(bestInd);
        avail.erase(avail.begin() + bestInd);
        locals[i] = *loc;
    }

    *locations = locals[0];
```

```
    for (int i = 0; i < TAGS; i++) {
        cout << "Matches: " << matches[i].first << " and "
            << matches[i].second << endl;
    }
}
```