

FPGA Based Kalman Filter

A Major Qualifying Project Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Eduardo Pizzini

Daniel Thomas

April 26, 2012

Professor David Cyganski, Primary Advisor

Professor James Duckworth, Co-Advisor

Abstract

This project was undertaken to support the WPI Precision Personal Locator (PPL) project by prototyping an FPGA implementation of a Kalman Filter to perform inertial system signal processing. Currently, the Kalman Filter operations are carried out in software on a single core CPU. This project converts the software design into a parallel processing hardware. A simplified Kalman Filter was designed and implemented within the FPGA and tested with an ADC and DAC which were integrated into the design to support analog signals at the input and output of the system. The successful demonstration of this system shows the viability of using an FPGA based Kalman Filter to perform the signal processing for the PPL system in real time.

Executive Summary

This project was undertaken to support the WPI Precision Personal Locator (PPL) project by prototyping an FPGA implementation of a Kalman Filter to perform inertial system signal processing. Currently, the Kalman Filter operations are carried out in software on a single core CPU. This project converts the software design into a parallel processing hardware. A simplified Kalman Filter was designed and implemented within the FPGA and tested with an ADC and DAC which were integrated into the design to support analog signals at the input and output of the system. The successful demonstration of this system shows the viability of using an FPGA based Kalman Filter to perform the signal processing for the PPL system in real time. The Precision Personal Locator device is designed to protect first responders during firefighting by monitoring their location within buildings. This is important so that they can more easily find their way around the inside of a burning building, but it also allows for other responders outside of the building to locate them in the case of an emergency. The device uses sophisticated signal processing and geolocation to perform its functions. In this project, we focus on the inertial system signal processing, which is done by a Kalman Filter. Currently, the Kalman Filter is only realized as a software implementation in Matlab. However, the goal for this device is to use an FPGA to realize a hardware version of the Kalman Filter, allowing data to be processed in real time.

A Kalman Filter is a statistical algorithm used to compute an accurate estimate of a signal contaminated by noise. It uses measurements observed over time to reduce noise and produce estimates of the true values, and estimate other parameters related to the signal model. The locator uses multiple input signals which when alone, can each be used to determine location of a device. Depending on different circumstances, some inputs are more contaminated with noise than others. The Kalman Filter uses all the available information from all of the input signals to minimize the noise and determine the best estimate for the location of the device.

This MQP project contains three different design blocks. First, analog inputs are streamed in using the ADC and ADC controller and converted to digital values. The next block is the actual Kalman Filter logic used to digitally process the data, which is implemented within an FPGA. Lastly, outputs of the Filter are streamed out of the system using the DAC and DAC controller to produce an analog signal.

The first section of the design to be treated is the main component of this project. This is where the Matlab code of the Kalman Filter logic is converted into a hardware design using the FPGA. To implement the hardware design, we used a Spartan3 FPGA which is located on the Nexys2 board from Digilent. The FPGA design was programmed using hardware description language. This section of the design is the most important because it is where all of the signal processing is done. It performs the calculations by using the current input, as well as previously calculated values stored in registers, to produce an output. This section of the design is fully digital and involves mapping various operations which were floating point based in the original software implementation into fixed point form.

There are some calculations within the Kalman Filter design which are more difficult to implement in hardware than in software. One line of code within a software program can turn into a large process within a hardware design. There are some calculations within the Kalman Filter logic which require a matrix multiplication of two 3×3 matrices. To do this, 27 multiplies are required, but since there are only 20 multipliers in the FPGA, the calculation could not be done all at once. We decided to multiplex 3 different sets of operands to 9 multipliers to perform each matrix multiplication. Division is difficult to realize in hardware and uses a lot of resources. We used the core generator tool in the Xilinx software to create a division module which takes in 2 16-bit values and produces a 16-bit quotient. The core generator uses the resources of the FPGA very efficiently. In some calculations within the Kalman Filter, the previous outputs are used to calculate current values. For this, we used flip flops to act as registers which hold previously calculated values. Those values are then loaded into the system for on the next clock cycle to be used in the current calculations.

After the Kalman Filter logic was designed within the FPGA, we performed tests using a set of known inputs and outputs from Matlab. We first performed testing by creating a test bench with our design and used the Matlab inputs. The outputs of our system were observed and compared to Matlab's. After we verified that our design was working using a test bench, we tested the synthesized design by using a UART to input signals to the Kalman Filter and receive outputs from the Kalman Filter. After we verified that our outputs matched Matlab's once again, we could say for sure that our Kalman Filter design within the FPGA was working correctly.

After we were able to successfully implement the Kalman Filter within the FPGA, the next step was to implement a DAC so that we could output an analog signal instead of just digital values. We chose to use a 12 – bit digital to analog converter from National Semiconductor. This DAC has a 0- 3.3V range which is sufficient for this project. It also can easily satisfy our output rate of 25 KHz. The DAC was convenient to use because it is located on a PMOD from Digilent, which connects directly to the Nexys2 board to easily integrate with the FPGA. We also had to design a DAC controller within the FPGA so that the DAC can properly integrate with the Kalman Filter design.

To test that the DAC was working with our Kalman Filter design, we stored Matlab input values within the FPGA. We then loaded these inputs to the Kalman Filter and then sent the outputs of the filter through the DAC. We verified that everything was working correctly by comparing the oscilloscope capture of the outputs to the Matlab graph.

The remaining hardware section which needed to be implemented were the ADC and ADC controllers. Implementing these allowed for the Inputs to be streamed in as an analog signal instead of storing the inputs in the FPGA as we did in the last section. A 12-bit analog to digital converter from Analog Devices was used, which also had a voltage range of 0-3.3V. The possible sampling rate of this ADC was far above our sampling rate 25 KHz. Just like the DAC, this ADC is located on a PMOD from Digilent which connects directly to the Nexys2 board, allowing for easy connection to the FPGA. We also

created an ADC controller within the FPGA. This allows for the ADC to be properly integrated with the Kalman Filter design.

Before we connected the ADC and ADC controller to the rest of the system, we wanted to make sure we could input an analog signal and correctly send digital values to the FPGA. Once the values were sent to the FPGA, they were sent to the UART and out to the PC to be captured using a terminal. We tested that the ADC was successfully sending digital values to the FPGA by plotting the captured values and comparing the waveform to the analog input signal.

After we verified that the ADC was working properly, the final step was putting the entire system together. Adding the ADC and DAC to the Kalman Filter allows us to input and output an analog signal. To test our complete design, we input a mixed signal which consisted of a 500 Hz sine wave added to a 6.25 KHz sine wave. If the Kalman Filter system is performing properly, it should pass the 500 Hz sine wave and try to reject the 6.25 KHz sine wave. Since we were able to match our oscilloscope capture of the input and output signals to the Matlab graph, we could determine that our entire design was working and that we had successfully completed our MQP.

Our original goal was to convert the Kalman Filter from software to hardware using an FPGA so that signal processing can be done in real time. At the end of this project, we were able to successfully convert the design from software to hardware. We were able to integrate an ADC and DAC with the FPGA to use analog input and output signals. Our final testing showed that we could process the data in real time to produce the correct output. Lastly, we have demonstrated the viability of using an FPGA based Kalman Filter in the Precision Personal Locator Device to perform the signal processing. We believe that this project can be taken further by another group in the future. The new group should enhance our FPGA Kalman Filter design to allow for multiple inputs and outputs so that it can perform the processing required for the Precision Personal Locator Device. Secondly, the current software

version of the Kalman Filter used by the Locator Device should be replaced by this hardware system to perform real time processing.

Table of Contents

Abstract	1
Executive Summary.....	2
Chapter 1: Introduction	9
1.1 Precision Personal Locator Device	9
1.2 Complete Kalman Filter System	10
Chapter 2: Background	12
2.1 Precision Personal Locator Device	12
2.2 Kalman Filter:	13
Chapter 3: Implementing the Kalman Filter within the FPGA	15
3.1 Matlab Version of Kalman Filter	15
3.2 Designing the FPGA Based Kalman Filter	17
3.2.1 Matrix Multiplications.....	17
3.2.2 Division.....	21
3.2.3 Using Registers to Store Previous Values.....	22
3.2.4 Representation of Values.....	23
3.3 Testing the Design with Test Benches	24
3.3.1 Text Files with the Test Bench	25
3.4 Testing the Synthesized Hardware	27
3.5 Testing Using a UART	28
3.5.1 Beginning to Work with the UART	29
3.5.2 Inputting/Outputting Strings	32
3.5.3 UART with the Kalman Filter	38
Chapter 4: Using the Kalman Filter with Real-Time Signals.....	40
4.1 DAC Selected	40
4.2 Implementation of the DAC	41
4.3 Results of DAC Implementation	43
4.4 Choosing an ADC.....	46
4.5 Results of ADC Implementation	48
4.6 Results of ADC implementation with UART	50
Chapter 5: Implementing the Design of the Complete System	53
5.1 Implementing the Kalman Filter with the DAC	53

5.2 Implementation of ADC to DAC	53
5.3 Implementation of Complete Design.....	54
5.3.1 Modifications to the Overall Design	55
Chapter 6: Results of Overall Implementation	62
6.1 Kalman Filter to DAC	62
6.2 Results of ADC to DAC.....	63
6.3 Mixed Signal	65
Chapter 7: Conclusion	72
References	74
Appendix A: Top Level Design of Complete System	75
Appendix B: VHDL Design of Kalman Filter Logic.....	81
Appendix C: VHDL Code for ADC and DAC Controllers	89
Appendix D: VHDL Code for Multiplier Modules used for Calculations.....	95
Appendix E: UCF File for the Complete Kalman Filter System	99

Chapter 1: Introduction

The advisors of this Major Qualifying Project (MQP) were Professors James Duckworth and David Cyganski. Both are Professors of Electrical and Computer Engineering at Worcester Polytechnic Institute. Currently, they are working on the development of technology to save the lives of first responders. This project was undertaken to support the WPI Precision Personal Locator (PPL) project by prototyping an FPGA implementation of a Kalman Filter to perform inertial system signal processing. Currently, the Kalman Filter operations are carried out in software on a single core CPU. This project converts the software design into a parallel processing hardware. A simplified Kalman Filter was designed and implemented within the FPGA and tested with an ADC and DAC which were integrated into the design to support analog signals at the input and output of the system. The successful demonstration of this system shows the viability of using an FPGA based Kalman Filter to perform the signal processing for the PPL system in real time.

1.1 Precision Personal Locator Device

The goal of this MQP is to assist the Precision Personal Locator Project by development of new technology to enhance the PPL device. This device is designed to protect first responders during firefighting by tracking their location within buildings. This protects them because it allows them to find their way through a burning building without getting lost. Also, if they become trapped in the building, then other responders are able to identify their location very rapidly. To perform its functions, the device uses signal processing which is done by a Kalman Filter. Currently, the Kalman Filter used by this device is designed in software. However, the goal for this device is to use a hardware version of this Kalman Filter that can be located within the device and process the data in real time. This MQP will show how such a Kalman Filter can be designed and just how well it can process the data in real time.

1.2 Complete Kalman Filter System

The other goal of this MQP is to show how an FPGA based Kalman Filter could be beneficial to the Locator Device. We will do this by creating a Complete Kalman Filter System, including an FPGA based Kalman Filter, analog to digital converter (ADC), and digital to analog converter (DAC). The goal of this part of the project is to produce a system where data can be streamed into and out of the Kalman Filter as analog signals and processed in real time. To do this, the software version (Matlab code) of the Kalman Filter needed to be converted into a hardware version (VHDL code). The FPGA that was used was the Spartan 3E, which was included with the Nexys2 board from Diligent. A picture of this board and FPGA can be seen below.

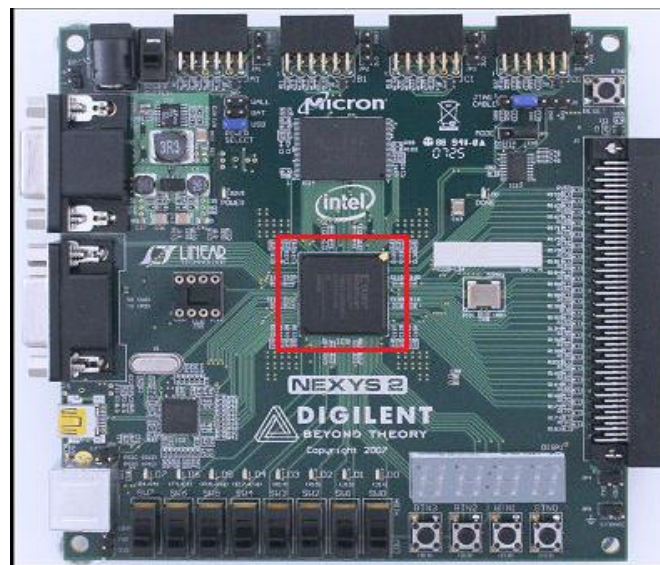


Figure 1 – Nexys2 Board and Spartan 3E

Besides the FPGA, the ADC and DAC were two other hardware devices which were needed for the system. The ADC connects to the Kalman Filter by the ADC controller, designed within the FPGA. The ADC controller directs the ADC when to take a sample of the analog input, and sends the digital value to the input of the Kalman Filter. Like the ADC, the DAC connects to the Kalman filter using a FPGA design,

the DAC controller. This module sends the digital output of the Kalman Filter to the DAC, and then instructs it when to output that value as an analog signal. This completes the system.

The responsibility of this Kalman Filter design is to reject higher frequency signals from passing through the system, but allow lower frequencies to pass unaffected. For example, the final test for this Kalman Filter was to input a mixed signal composed of a 500Hz sine wave and 6.25 KHz sine wave, but only pass the 500 Hz signal to the output. The following figure shows the overview of the system.

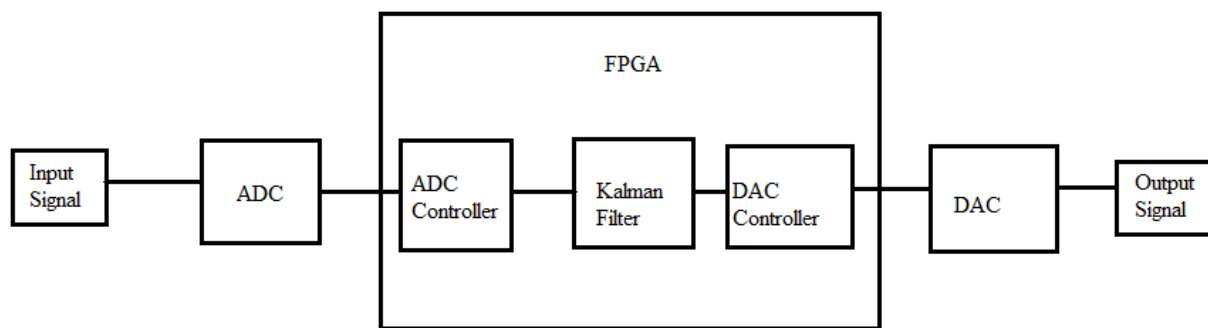


Figure 2 – Block Diagram of Complete System

The figure below shows the Matlab graphs of what the input and output signals of the system should look like. The input signal is shown in red, and the output signal is shown in green.

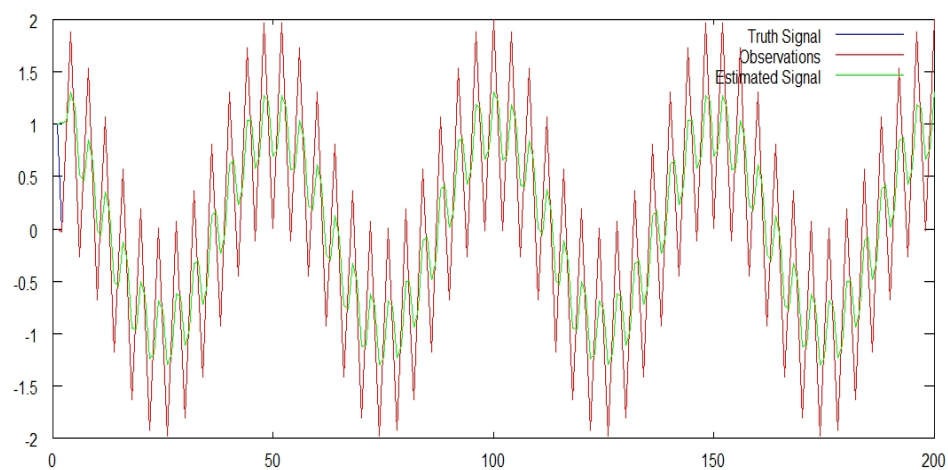


Figure 3 – Input and Output signals of the system

Chapter 2: Background

This section will contain important information that is necessary to completely understand this MQP and what it is about. The Precision Personal Locator Device will be discussed to show its importance and explain why our project can be an important tool in its implementation. Also, the overall idea of a Kalman Filter will be discussed to understand exactly what is being designed in this project, and what its purpose is.

2.1 Precision Personal Locator Device

The idea for the Precision Personal Locator Device was first raised after an incident in 1999. A warehouse fire in Worcester led to the death of six firefighters who could not find their way out of the building. This incident led a team of WPI researchers to develop a locator system. This locator uses extremely sophisticated signal processing to determine the exact location of a person within a building. The technology use for this project consists of OFDM. OFDM stands for orthogonal frequency division multiplexing which transmits either wired or wireless data. The firefighters will wear transmitters that constantly emit customized OFDM signals. Receivers will be placed on emergency vehicles to determine the exact location of the transmitters. This works by placing three or more emergency vehicles at different locations around the building equipped with the receiver to capture the location of the transmitter.

The overall goal of this system is to help protect firefighters by tracking their location of inside burning buildings. This protects them because these responders can more easily find their way through the building without getting lost. Also, if they become trapped, others can use the device to locate them more rapidly. The main idea is to retrieve the precise location of anyone wearing the tracking device in the building. To perform its functions, the device uses sophisticated signal processing and geolocation [1]. This MQP will focus on the signal processing part, which is performed using a Kalman Filter.

Currently, the Kalman Filter used by the Locator Device is designed in software. However, the goal is for this Kalman Filter to be designed in a hardware version so that it can be located within the device and process the data in real time. The MQP will be used to show the viability of this option.

2.2 Kalman Filter:

The Kalman Filter is an important part of this project. The purpose of the Kalman Filter is to use measurements observed over time, which contains random variations of noise, and produce a value that is accurate to the true values of the measurements [2]. It does this by predicting a value, estimating the uncertainty of the predicted value, and computing a weighted average of the predicted value and calculated value. The Kalman Filter first predicts the next value as well as the error covariance. When the next value comes into the filter, the Kalman gain is computed, the estimate is updated with the observed value, and the error covariance is updated [3]. This helps to get rid of the noise within the signal.

A Kalman Filter is different from other filters such as low pass or high pass filters. These filters are linear, time invariant systems which are designed with frequency response in mind. These filters tend to be single input single output systems. In this MQP, the Kalman Filter we are designing could be replaced by one of these filters because it has all of the characteristics just described. However, this MQP is used to show how an FPGA based Kalman Filter could be beneficial to the Locator Device. The Kalman Filter within the Locator device is designed with the characteristics of a normal Kalman Filter. These characteristics involve being a multiple input, multiple output system. Also, they are linear, time variant systems which are designed with a mean square error approach.

The way in which the Locator device uses its Kalman Filter is by using multiple input signals. Each of these signals by themselves could be used to determine location. However, the system cannot rely on any one source because in different scenarios, each of these input signals could be contaminated by

different amounts of noise. So, the Kalman Filter takes the information from all of these signals, uses it to reduce the noise and produces its best estimate for the location of the first responders.

For this particular project, we will be working with an FPGA to implement a Kalman Filter. FPGA stands for field programmable gate array, and the design that it implements is realized using hardware description language. Any functions performed by an application-specific integrated circuit (ASIC) can be performed using the FPGA. FPGA's contain logic blocks which can be configured to perform many operations. In order to implement the Precision Personal Locator Device, a Kalman Filter is used to read the transmitter and get rid of the noise and return back the exact location of a particular person. As of right now, data is recorded and then input to the Matlab version of the Kalman Filter, which then produces the output. The problem with this is that that information needs to be taken in, processed, and sent back out in real time and not computed later on. The Precision Personal Locator Device will be discussed in greater detail in the next section.

Chapter 3: Implementing the Kalman Filter within the FPGA

Implementing a Kalman Filter design on an FPGA is difficult. Although the Matlab code contains the necessary calculations to simulate the performance of this Kalman Filter, some functions which are simple in Matlab become quite complex when implementing them in a hardware description language. Functions like matrix multiplication, sine functions, and division, which are easily executed in software programs, present a challenge when creating them in hardware. As previously mentioned, the FPGA is accompanied by an ADC as well as a DAC, so that it can use digital values even though the inputs and outputs of the overall system are analog voltages.

3.1 Matlab Version of Kalman Filter

The first step in the project was to become familiar with how the Kalman Filter worked and try to implement each part of the design step by step. The first step was to convert the Matlab code into fixed point representation. Doing this allows us to represent the Matlab values as digital values, which our Kalman Filter uses. After the code was successfully converted into fixed point representation, graphs were created to show what the input signal was, and what the output of the Kalman Filter should be using that that input. This graph can be seen below.

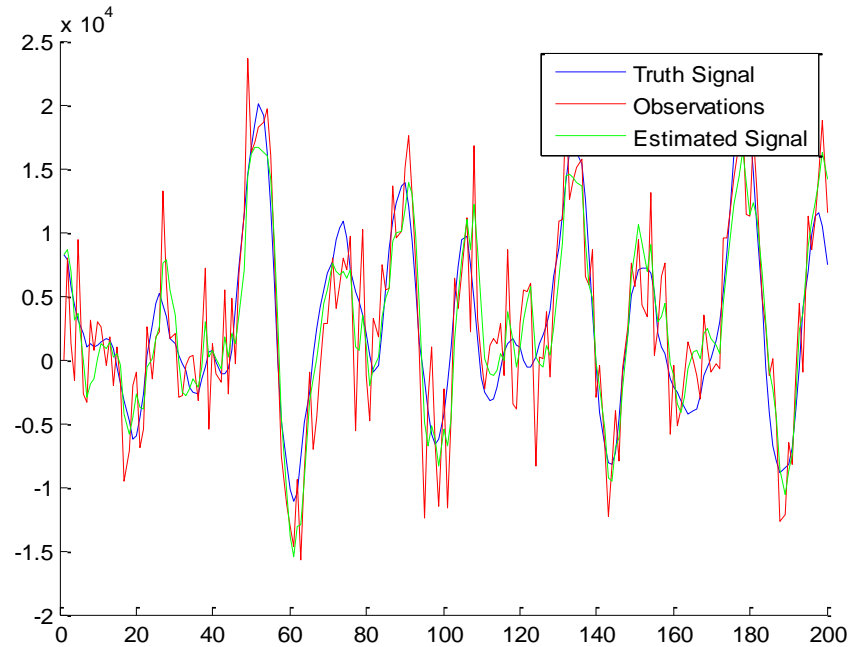


Figure 4 – Output of Kalman Filter Compared to Truth Signal

After creating this graph, 100 input values and output values from Matlab were recorded so they could be used for testing the FPGA version of the Kalman Filter. Besides the inputs and outputs, values from each calculation were recorded so that small sections of the Kalman Filter could be tested individually. Once the FPGA based Kalman Filter was completely designed, it would be tested using the input values, and then comparing the output values to the values calculated using Matlab. This would be done by first using a test bench to test the design, and then using a UART to test the actual hardware. These tests will be discussed in more detail in the Kalman Filter results section. The testing approach can be seen in the block diagram below.

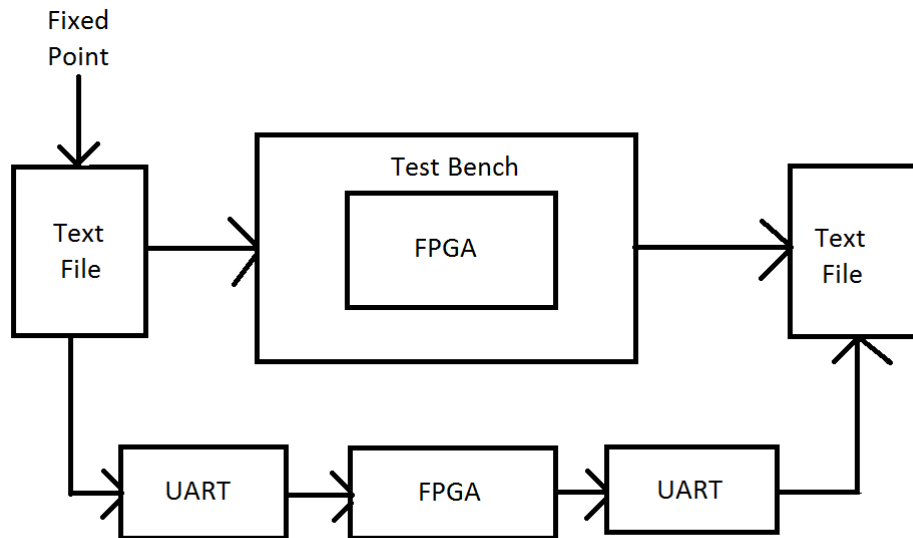


Figure 5 – Block Diagram of Design Approach

3.2 Designing the FPGA Based Kalman Filter

Since the design was quite complicated, it was determined that the best approach was to break the design down into small pieces. These sections were divided based on each calculation made in the Matlab code. After implementing each of these calculations using VHDL hardware language, the Matlab values were used to verify that the hardware was correctly designed. Along the way there were some Matlab code calculations which were not so easy to convert into a hardware design. These required some extra thought on the most efficient way to perform the calculations, and some manipulations to make them simpler.

3.2.1 Matrix Multiplications

The first obstacles presented were the three matrix multiplications. After looking closely at the Matlab code and performing some calculations by hand, it was clear some of these calculations could be greatly reduced into one or two multiplications. However, even after this, there were still three instances where 3x3 matrix multiplications were necessary. The Spartan 3E is equipped with 18

multipliers, which meant not all of these multiplications could be done at once. Also, since some of the multipliers were already being used in single multiplications, there were not 18 multipliers available.

We came to the conclusion that we had to multiplex what values are being multiplied at certain times, so that each multiplier could be used more than once. The way we approached this was to use 9 multipliers at once. This allowed us to input a three value row as well as an entire 3x3 matrix. This means that an entire row could be calculated with each set of the inputs of the multiplexer. The entire 3x3 matrix remains the same for these calculations but the row of 3 is what is getting multiplexed for another 3 x 3 matrix. A module was created to perform the 9 multiplications between a 1x3 matrix and a 3x3 matrix.

In order to perform a matrix multiplication with two 3x3 matrices, this module needs to be used 3 times. The entire input matrix remains the same, but the 3 different rows of the other matrix are alternated as the input. In the next state logic, there are three 3x3 matrix multiplications that need to be calculated. The output to all of these is a 3x3 matrix. Since we can only calculate 1 row at a time using our module, we need to multiplex 9 different sets of inputs. To do this, we create a signal named count. At the rising edge of each clock cycle, count is incremented. Once its value gets to 9, it returns to 1 and continues to increment to 9 once again. A block diagram of this process is shown below.

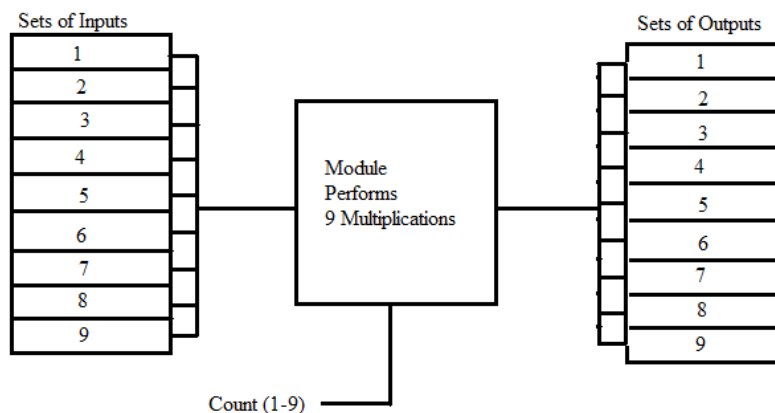


Figure 6 – Multiplexing to Perform Matrix Multiplications

As previously mentioned, this module accounts for 9 of the multipliers which are available in the Spartan 3E FPGA. The figures below show that as the value of count changes between 1 and 9, different rows are calculated, proving that the multiplexing is indeed working correctly.

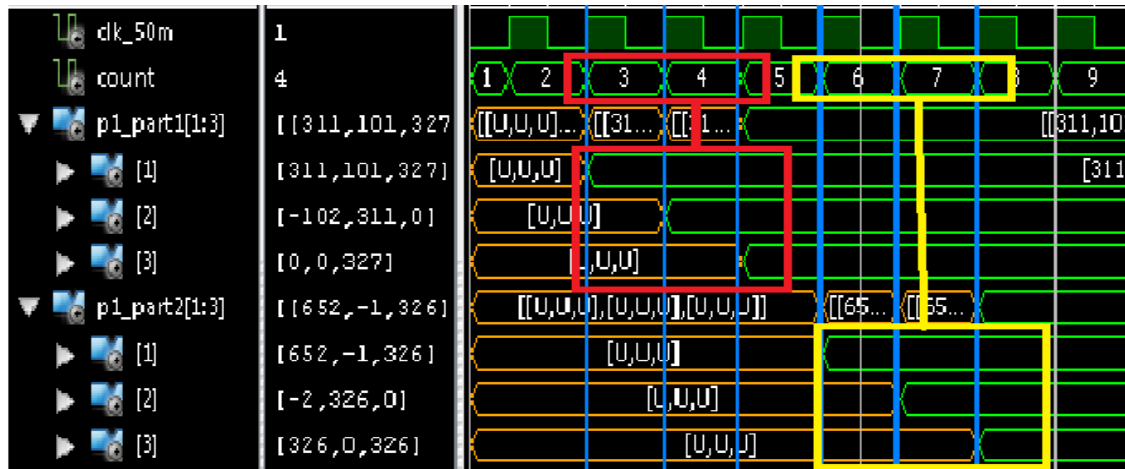


Figure 7 – Multiplexing for Count 3-8

In the next figure, the orange shows the final three rows being multiplexed with values of count from 9 to 2.

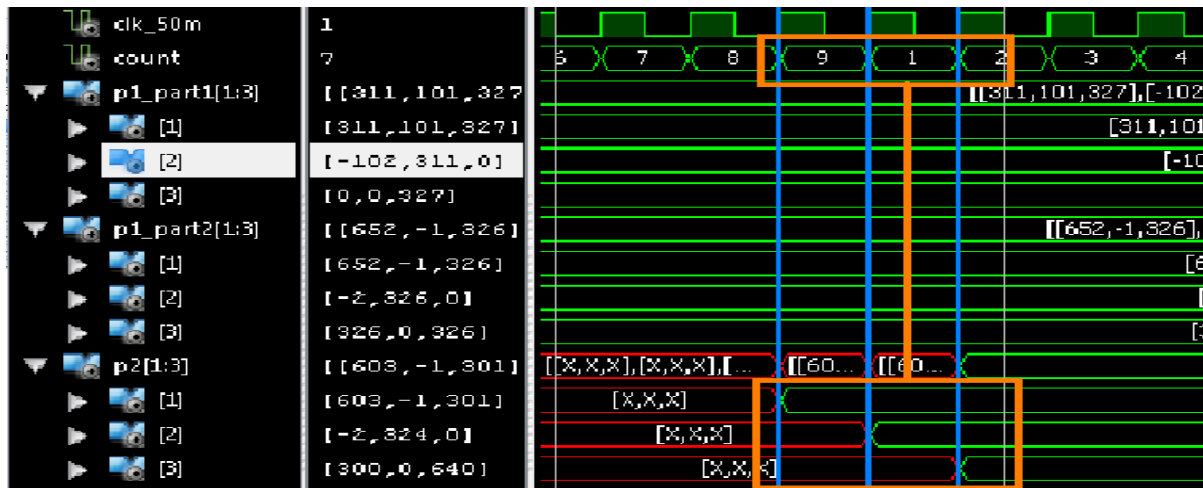


Figure 8 – Multiplexing for count 9-2

Aside from this module performing matrix multiplications, there was also another module which used multipliers. The second multiplier module takes in 6 different inputs to compute 7 multiplications. So, this module uses 7 multipliers. These inputs are not multiplexed, so the input signals are always the same. The outputs are calculated on the next clock cycle after a change in the input is detected. This

figure shows that when new values of x_in1 and x_in2 are detected, 4 different products are computed simultaneously at the next rising edge of the clock.

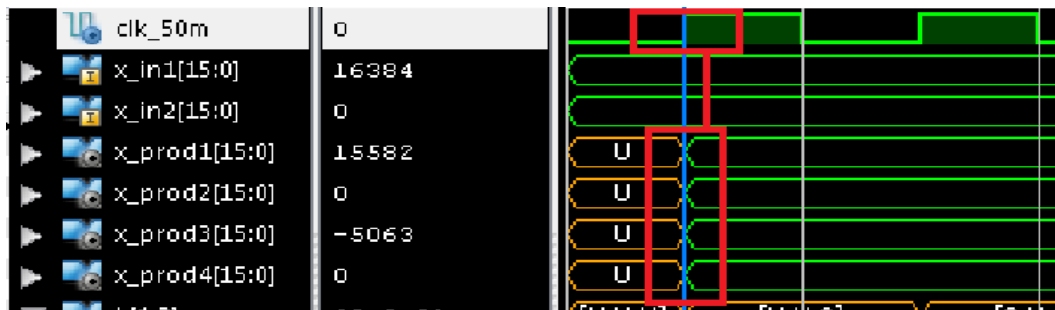


Figure 9 - Test Bench of First 4 Multiplications

The figure below shows the other three multiplications performed in this module. As soon as the values of K(1:3) change, they are multiplied by z_diff to produce kz(1:3). Since the values of K are changing at different times, the different values of Kz change with them.

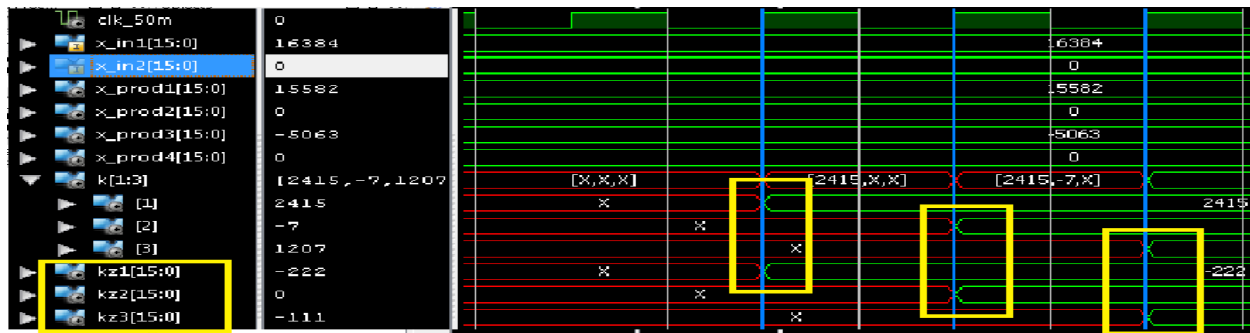


Figure 10 - Test Bench of The Remaining Three Multiplications

The previous two figures show the 7 multipliers being used. These are the only multipliers used in the entire design. This means that a total of 16 multipliers should be used. As we can see from the figure below, 16 multiplies are in fact being used.

SimpleKalmanFilter Project Status			
Project File:	SimpleKalmanFilter.xise	Parser Errors:	No Errors
Module Name:	SimpleKalmanFilter	Implementation State:	Programming File Generated
Target Device:	xc3s500e-4fg320	• Errors:	No Errors
Product Version:	ISE 13.2	• Warnings:	9 Warnings (6 new)
Design Goal:	Balanced	• Routing Results:	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• Timing Constraints:	All Constraints Met
Environment:	System Settings	• Final Timing Score:	0 (Timing Report)


Device Utilization Summary					
Logic Utilization	Used	Available	Utilization	Note(s)	
Number of Slice Flip Flops	2,617	9,312	28%		
Number of 4 input LUTs	2,598	9,312	27%		
Number of occupied Slices	2,321	4,656	49%		
Number of Slices containing only related logic	2,321	2,321	100%		
Number of Slices containing unrelated logic	0	2,321	0%		
Total Number of 4 input LUTs	2,659	9,312	28%		
Number used as logic	2,562				
Number used as a route-thru	61				
Number used as Shift registers	36				
Number of bonded IOBs	16	232	6%		
Number of BUFGMUXs	1	24	4%		
Number of MULT18X18SIOs	16	20	80%		

Figure 11 - Resources Used for Kalman Filter

3.2.2 Division

Another task that we needed to perform was dividing two values within the Kalman Filter. Performing division is a difficult task because it takes a lot VHDL code and uses a lot of resources. It was decided that the best approach was to use the built in core generator in the Xilinx software that the VHDL design was being written in. The core generator can create a number of different functions and it uses an efficient amount of resources. Once we created this module, we worked to include it within our Kalman Filter design. The schematic of this function can be seen below.

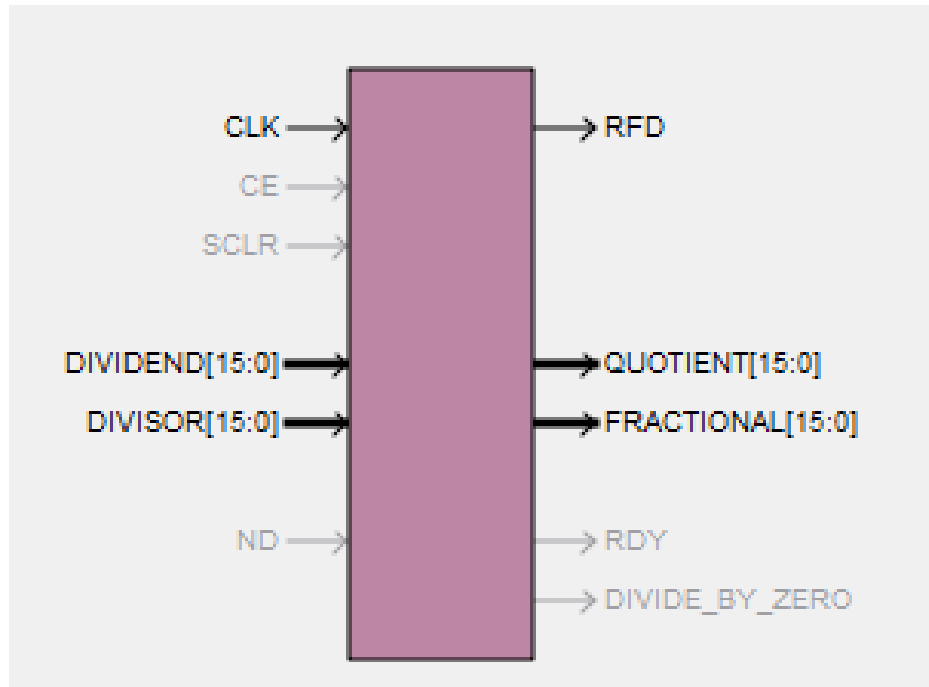


Figure 12 - Divide Function

Two 16 bit signed values are entered into the function. A signed 16 bit quotient is output, as well as a remainder. This module proved to be very useful to us within our design, and was quite simple to implement.

3.2.3 Using Registers to Store Previous Values

The use of registers was very important to this design. Since the design uses previous values in the current calculations, the previous values needed to be stored in registers. Also, the values in these registers have to be loaded into the design along with the input. Without using registers and loading values in on each clock cycle, the design would cause a continuous loop. This happens because as the output changes, the current calculations would change causing the output to change again, and this would keep happening. So, once we knew we needed to use registers to store previous values, a block diagram was drawn first to better understand how this could be done. The block diagram proved to be very helpful. A simple project was first created to make sure the process worked before it was added to the overall project. This turned out to be a success and we could clearly see values being stored in

registers, and then loaded from registers. The block diagram for using the registers can be seen below. On the rising edge of the clock, or for testing purposes, when a button is pressed, flip flops load the previous output values as well as the current input. After going through the next state logic, the output values are stored in the registers and remain there until the next load.

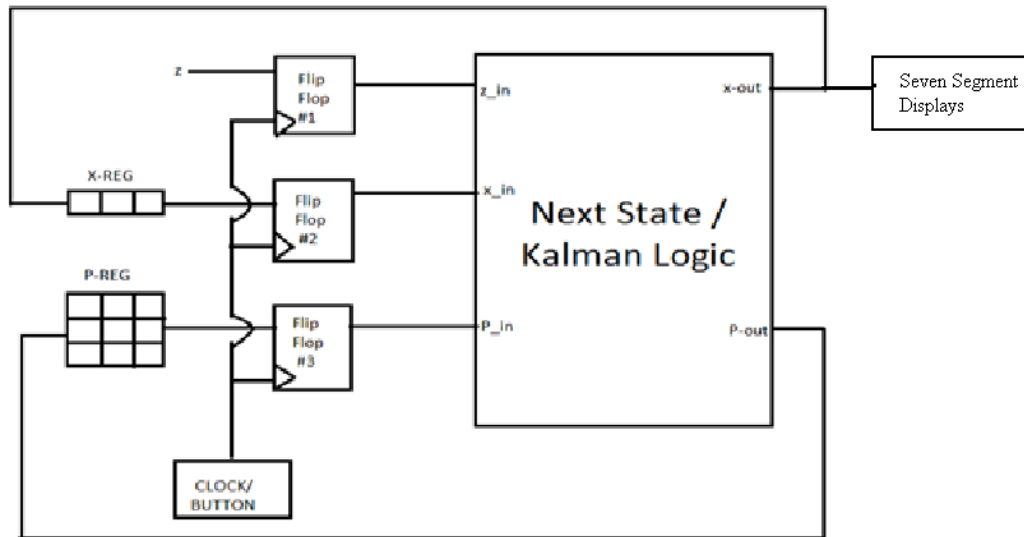


Figure 13 - Main Block Diagram

3.2.4 Representation of Values

Since the Matlab values are what we will be using to test the FPGA based Kalman Filter, digital values needed to be able to represent the floating point values of Matlab. The Kalman Filter uses 16-bit values. It was decided that the best thing to do was scale the Matlab values down so that all of the inputs are values between -1 and 1. This way, there is no possibility of overflow when multiplications are performed. To represent these values as digital numbers, Q-15 representation was used. This means that of the 16 bits, there is one sign bit, followed by 15 fractional bits. So, even though fixed point values are used, they are still representing floating point numbers.

3.3 Testing the Design with Test Benches

The first testing that was done was using Test Benches. Test Benches allow for the testing of the actual design before the actual hardware is tested. The first set of Test Benches was used to see values being loaded and going through the calculations of the Kalman Filter. The figure below shows the Test Bench used. The yellow cursor marks the time where the first input (z) value is loaded into the Kalman Filter/Next state logic block. All the blue markers show the next six times that the input value, as well as the previous output values (x_reg and P_reg) are loaded into the Logic. Looking at the colored boxes on the left hand side, we can see that when butpress is high, and there is a rising edge of the clock, z_in gets the value of z. We can also see that the initial values of x_in are correct, as well as the first outputs, x_est. Lastly, the x register values store the values of the outputs. Looking closer at the test bench, we can see that when butpress is high, and there is a rising edge of the clock, the x_reg values get loaded into the x_in values. Looking at the input and output values of Matlab, it was proven that the design of the Kalman Filter was working properly, because the output values shown in the test bench match the output values of Matlab code.

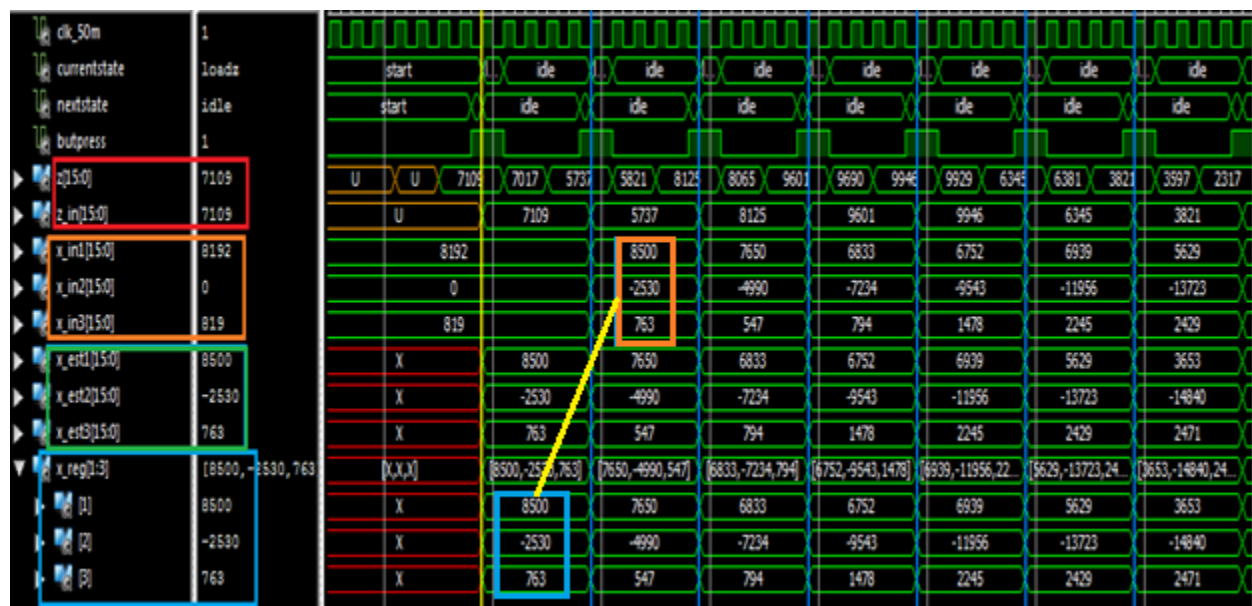


Figure 14 - Kalman Filter Test Bench

3.3.1 Text Files with the Test Bench

Another step that was taken was inputting values from text files into our test bench. This makes things easier because the outputs are written to a text file as well, which makes things quicker and easier to debug. Some simple projects were created to become familiar with inputting and outputting values with text files. After we were able to complete these small projects successfully, this feature was added to the overall design. We used the input file to read the input of the first 5 z values of the logic, and wrote the result in another file. We wanted to be able to load in our values in floating point, just as they are seen in Matlab. Since it was previously decided that the Matlab values needed to be scaled down to values between -1 and 1, the Matlab values would be contained within the text file and scaled down by the test bench. After the calculations are performed, the values are scaled up again to match the Matlab values. The test bench takes in a floating point value, scales it down and converts it into a 16 bit fixed point value with Q-15 representation. This is the value that is used as the input to the Kalman Filter. After all of the logic has been completed, the Kalman Filter produces an output of the same representation. Once again, the test bench converts this value by scaling it back up, and then converts it to a floating point value. This value is outputted to the results text file. With this test bench, the inputs and outputs of the Kalman Filter to be easily compared to those of Matlab since they are in the same form. Five inputs, as well as five outputs and the expected outputs can be seen below.

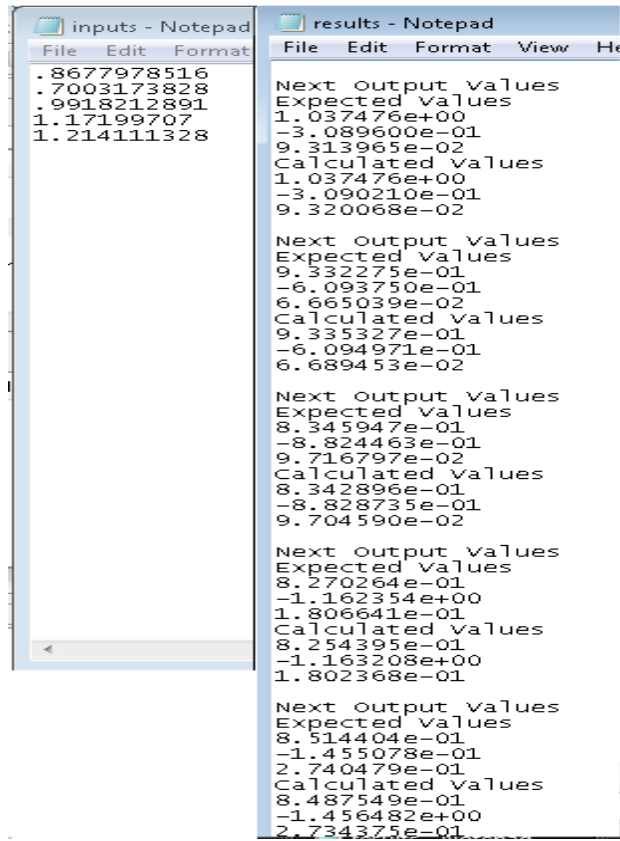


Figure 15 - Kalman Filter Input and Output Text Files

By looking at the input values, we can see why it is necessary to scale them down. Since we want to keep all of the values between -1 and 1 to avoid overflow, we had to divide the inputs by 2 to keep them in this range. Looking at the results file, we see that there are five groups of outputs. The first three values in each group are the expected outputs for x_{est1} , x_{est2} , and x_{est3} , respectively. The next three values are the calculated values for these same three signals. These are the three state outputs of the kalman filter. By looking at these results, we see that the calculated values are very accurate when compared to the expected values. In most circumstances, the values are no more than a few thousandths apart, and in the worst cases, a few hundredths.

In the next two figures, our values are within the Kalman filter and are represented as 16 bit fixed point values. For the first figure, the red box shows that as butpress goes high, z_{in} gets the value of z . This means that the input has been loaded into the Kalman filter. The orange box that is connected

to this red box by the yellow line shows the corresponding three output values. If we convert these Q-15 values into floating point, we would see that they match the first group of calculated outputs on the results file. The next two groups of red and orange boxes show the next two inputs being loaded in, and then producing an output.

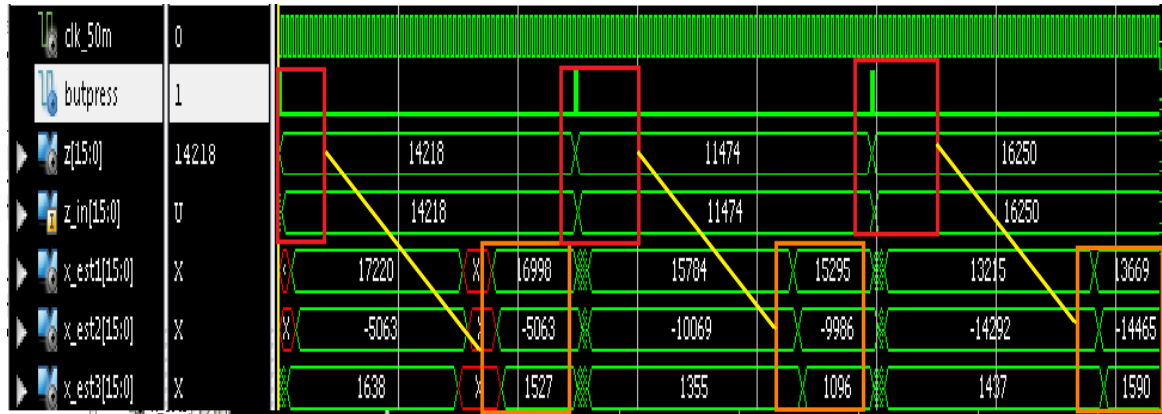


Figure 16 - Test Bench of First Three Inputs Being Loaded

The next graph shows the 4th and 5th group of input and outputs.

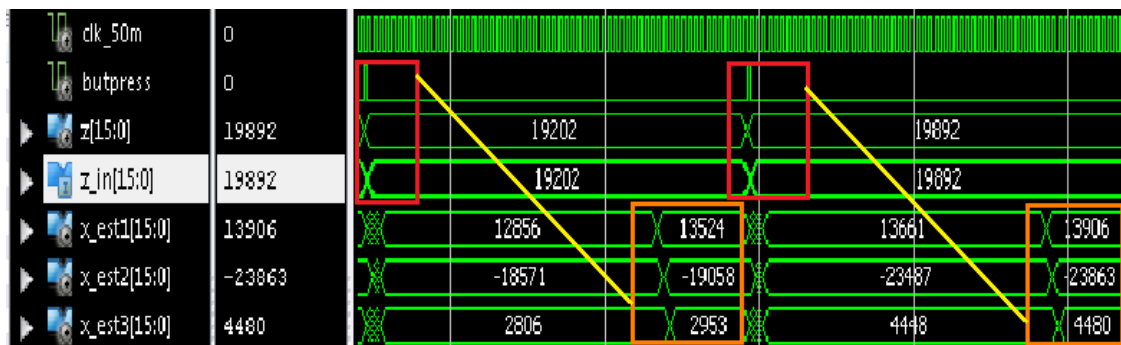


Figure 17 - Test Bench of Next Two Inputs Being Loaded

3.4 Testing the Synthesized Hardware

Once it was verified that the VHDL design was successfully performing the calculations of the Kalman Filter, it was time to synthesize the design. This means that the VHDL code which had been written will now implement the design using the actual hardware within the FPGA. Just like the test bench, it was determined that inputting a handful of known values, and comparing the outputs to

Matlab's would be sufficient to test the hardware. At this point, the switches and buttons on the Nexys2 board were being used to input values, and the seven segment displays were being used to show the outputs. This is a rather slow process, so inputting a large number of values would be illogical.

After performing this process for about five inputs, it was seen that the outputs exactly matched what the test bench had shown. This meant that the design was successfully synthesized and that the Kalman Filter was working properly. Now that we had shown that for at least a small number of values, the Kalman Filter was working, it was time to test it with a large number of values. This would show if the accuracy of the Kalman Filter would diminish after a while or if it continued to work. To do this testing and to complete the overall system which we have set out for, it is necessary to stream data.

3.5 Testing Using a UART

The first step taken to stream data into and out of the Kalman Filter was to employ the use of a UART. The UART can receive data from the serial port sent by the computer, and convert this serial data into the desired length digital value. It can also take a digital value and send it bit by bit through the serial port and to the computer. It provides an easy and convenient solution to the streaming of data since it does not involve the use of analog signals to test the functioning of the Kalman Filter. At this point in the project, the performance of the filter is assessed by comparing the output results with the outputs of the Matlab code. As previously shown, the inputs and outputs used by Matlab are being represented as signed, 16-bit values. With the UART, 16-bit values can be sent to the UART by the computer, from the UART to the FPGA, then back out of the FPGA to the UART, and then can be sent from the UART to the computer. This allows the inputs to quickly be sent and for the output to easily be seen on the terminal screen. The UART allows inputs to be loaded and outputs to be displayed at a much faster pace, meaning a greater number of outputs can be compared. This streaming of data can

prove that the Kalman Filter is not just accurately outputting the first few values, but it continues to be accurate as more and more inputs are loaded.

3.5.1 Beginning to Work with the UART

This was the first time that either member of the project team has done work with a UART. Due to this, it was decided that a few small projects would be completed to get a handle of how to implement the UART with FPGA modules. The Picoblaze microcontroller was used so that the built in UART could be incorporated. To get an understanding of how this UART works the notes from ECE 574 were examined. This provided an understanding of how to go about using the UART and how the UART works. Along with this, example code from ECE 574 was used as a starting point and some helpful explanations about which files needed to be adjusted was given by Professor Duckworth. The first goal that was set for working with the UART was to be able to transmit a string of characters, on a loop, which would say both of the names of the team members. For this task, data was not being sent to the UART, but rather this string was already stored in the UART. The purpose of this task is to show that data can be sent from the UART to the terminal successfully. From the figure below, this is confirmed. This was a rather simple step which involved adjusting a string of characters that was stored within the memory file.

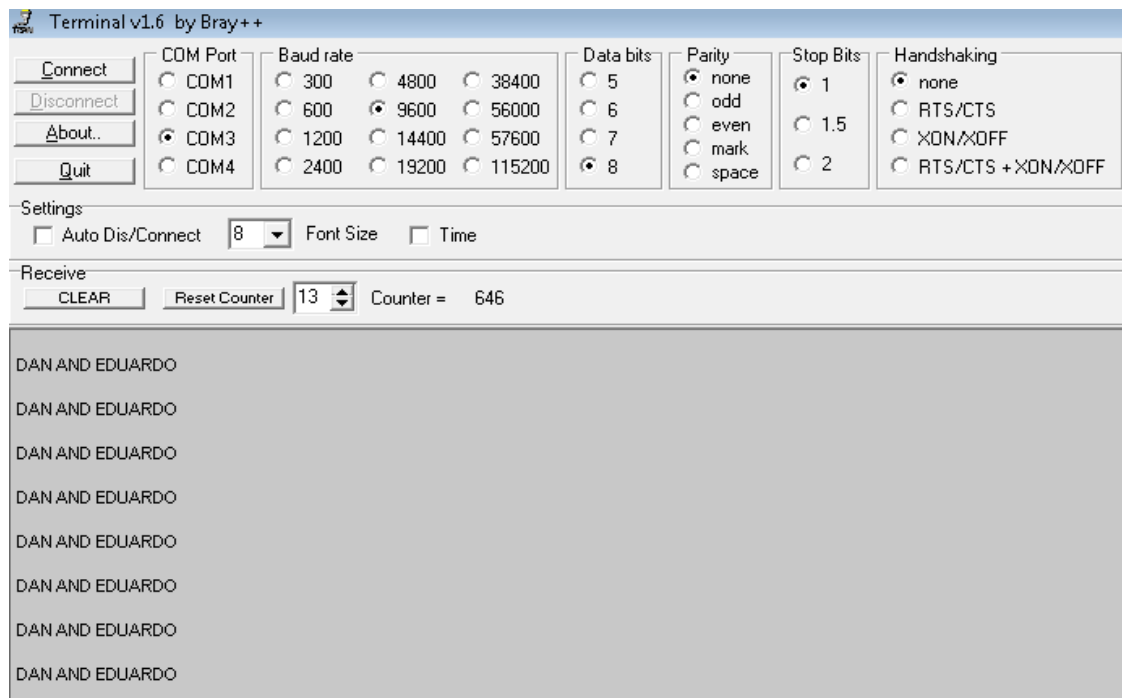


Figure 18 - Terminal Capture 1

The next goal was to be able to send a character to the UART from the terminal, to the FPGA from the UART, to the UART from the FPGA, and then send that character out of the UART and to the terminal. The first step is for the UART to receive a character from the terminal and store it in a register. After this is done, the data in the register can then be transmitted to the terminal. To complete this process, manipulation to the read and send functions within the assembly file was required. Simple changes were made to what registers the incoming data was stored and what registers the output data was taken from. Also, to show the 8 bit representation of the characters, the received data was displayed on the last two 7-segment displays. It was observed that any number sent was represented by sending a hex value of 3 for the first four bits, followed by the actual value in the last four bits. For letters, it is the same structure except that the first four bits represent a hex value of 4. This tells the computer that the next four bits determine the letter to display. This is important to know for future steps when processing the data inside the FPGA. The figure below shows that a character was received and

stored, and that value was transmitted each time through the main loop until a different value was received and stored. The numbers on the bottom of the screen are the numbers being sent from the terminal to the UART, and the numbers on the top section of the screen are the numbers being received by the terminal from the UART.

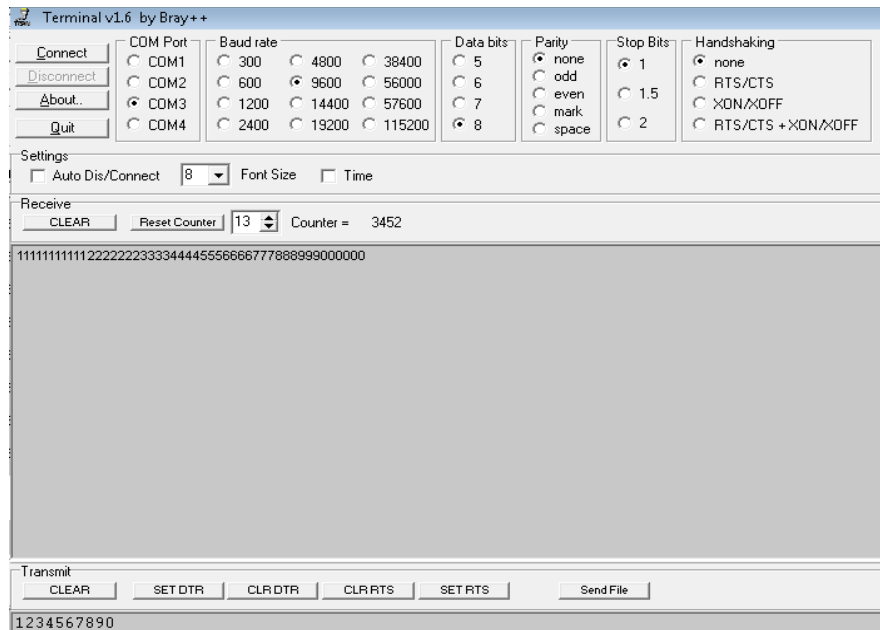


Figure 19 - Terminal Capture 2

For this project, data does not only have to be streamed into and out of the FPGA design, but there also needs to be some data processing performed by the logic module on the incoming values. In this stage some simple processing is implemented to verify that the correct values are produced when using the UART to input and output data at a fast rate. Just as in the previous step, data was stored in a register when once it was sent to the UART. To accomplish the new goal, the FPGA then took the data from that register, and added 2 to it to represent some simple processing before the data is sent out. Finally, the UART took that data out of the register and transmitted it to the terminal. Again, whatever value is stored in the register will be sent to the terminal every time through the main loop. To test that each value was being processed correctly,

values ranging from 0 to 7 were sent from the terminal, meaning the expected values of 2 to 9 should have been transmitted back to the terminal. The successful test results are shown in the figure below.

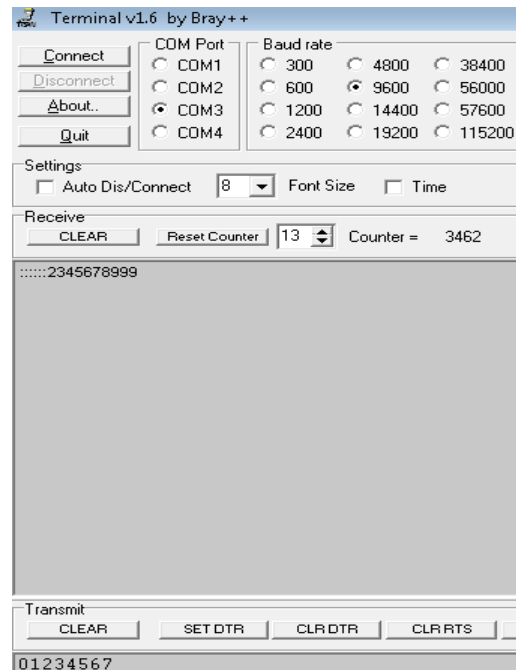


Figure 20 - Terminal Capture 3

Although these steps were small and simple, they were a huge help to the groups understanding of the UART. They were vital in understanding exactly how the Picoblaze incorporates the UART, and where and how the design must be adjusted to create the desired functioning of the UART. At this point, the group was ready to expand the use of the UART and integrate the functions which are necessary to test the overall Kalman Filter Design.

3.5.2 Inputting/Outputting Strings

The FPGA Kalman Filter design loads in 16-bit values and also outputs 16-bit values. For the terminal and Picoblaze UART to be used to test the Kalman Filter, it must be able to send a string of 16-bit values to the FPGA and also output a string a 16-bit values from the FPGA. Now

that it has been confirmed that a single hex value can be streamed in, processed, and streamed out, the next step is to do the same with multiple hex values at once.

The first goal that was set in this section of the project was to send 2 hexadecimal values together. Although the final goal is to send four at once, it was determined that it would be better to start small and validate that multiple values could be sent together. Again, when a value is sent from the terminal, it is sent as 8 bits. It can be viewed as 1 hexadecimal value though because only the last 4 bits contain useful information about the value. The first four bits are used as a code to tell the terminal what is being displayed on the screen. So when two values are sent, although that equates to 16 bits, the FPGA will only use 8 bits. When looking at the previous tasks using the UART, the other problem that is apparent is that the value in the register continues to be displayed during every loop. For each input, it is expected that it produced one output. To fix this, the send function was put into the read function so that values are only being sent one time after they are read.

The read function was recreated to receive 2 characters and create the 8 bit input. When the return character is sent, the 8 bit value is loaded into the logic. Also, the send function was recreated to break up the 8 bit output from the FPGA and send it as 2 characters to the terminal. To make this possible, an FPGA module was created with a state machine to work with the UART code. The functions from the UART code sent signals to the FPGA state machine to tell it when it should be setting up the upper and lower 4 bits of the input, when it should load the input, and when it should send the lower and upper 4 bits to the terminal. The diagram for this state machine can be seen below. Uin is the output of the UART which the FPGA takes in to control the state machine.

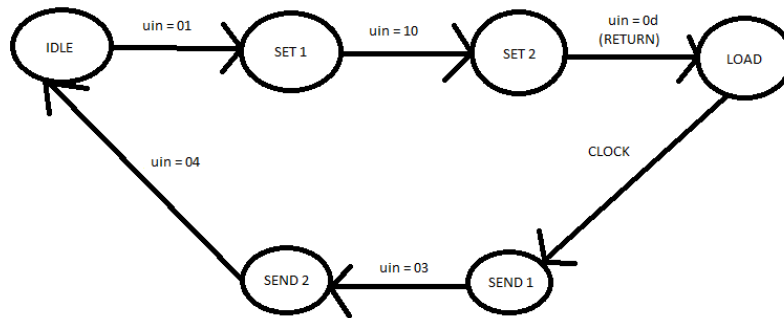


Figure 21 - State Machine of UART

The block diagram shown below shows the data path used to make this task possible. The data from the terminal is sent through the serial port and into the UART of the Picoblaze, which is then sent to the top level design. This design creates the 8 bit input and loads that into the logic of main design. The output of the design is then sent back to the top level component to prepare the data for transmission. The data is separated into 4 bit segments and sent along with four code bits to the UART. The UART then sends that data through the serial port bit by bit to the terminal. The terminal takes the 4 code bits and 4 value bits and uses the information to display each hexadecimal value as a character.

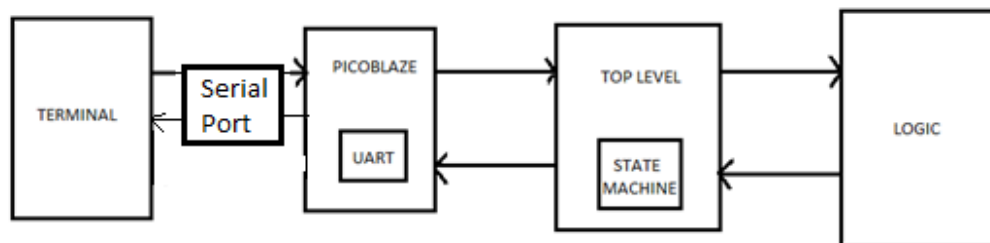


Figure 22 - Block Diagram of UART

The first figure shows that an 8 bit hex value could be sent from the terminal, and without any manipulation, the value could be sent back to the UART. This proves the VHDL top level module, as well as the UART functions are properly functioning.

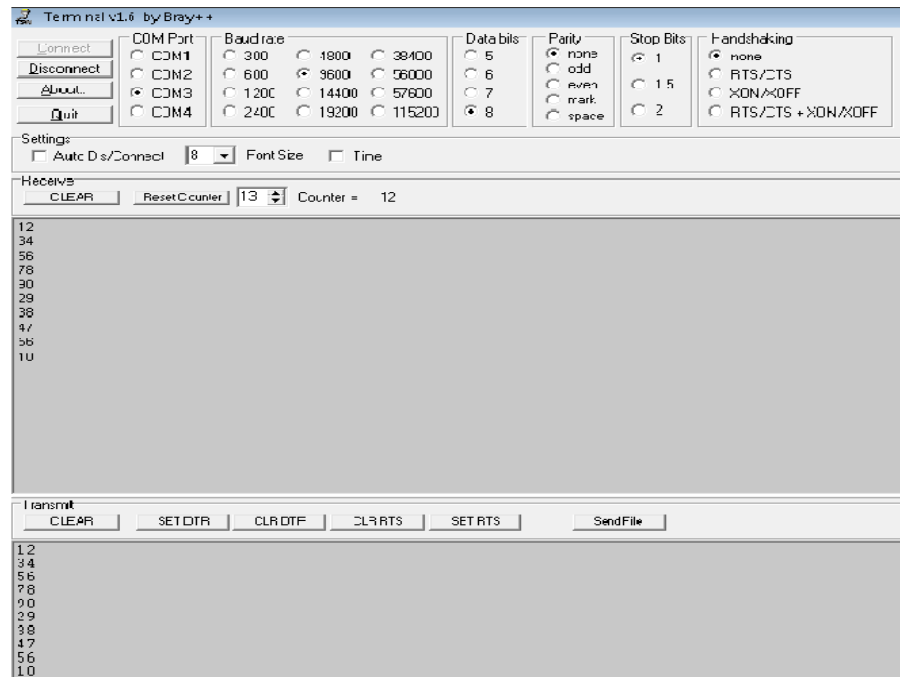


Figure 23 - Terminal Capture 4

The next figure shows that after some manipulation of the input inside the FPGA design, the output is properly sent to the terminal. The processing performed is adding 2 to each of the 4 bit segments.

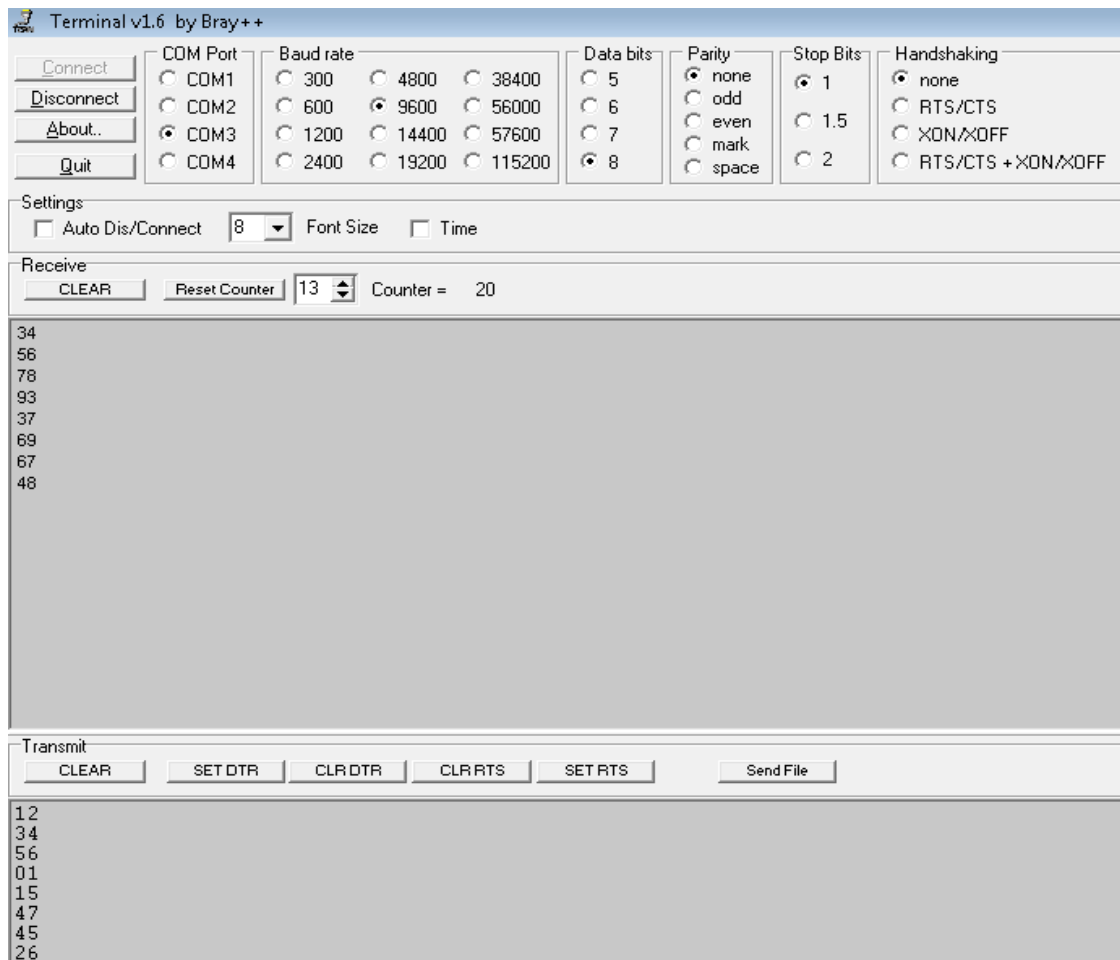


Figure 24 - Terminal Capture 5

Going through this process provided great experience and familiarized the group with using the UART functions and FPGA modules collectively to achieve the desired function. Since our project works with 16 bit values, the next step was to make sure we could successfully transmit and receive 16 bits of data using the terminal. There was also something else that we wanted to include in this test. Our Kalman Filter has some initial conditions meaning that the values of the P registers and X registers do not need to be loaded in along with the first 16 bit input. The first input is loaded in by itself and then after that, the registers will be loaded in along with the input. To show that the state machine for this process is working successfully, when the first 16 bits are transmitted, those exact bits are sent back to the terminal. However, for every other 16 bit input, each of the four hexadecimal values is incremented by 1. This helps to show

that the state machine went from the load input state, to the load all state after the first input. The figure below shows that the system is working properly. There was a smooth transition from the sending and receiving of 8 bit values to the sending and receiving of 16 bit values. The functions in the UART had to go through the same process two more times, and four states had to be added to the FPGA state machine, two to receive 8 more bits or 2 more hex values, and two to send 8 more bits or 2 more hex values.

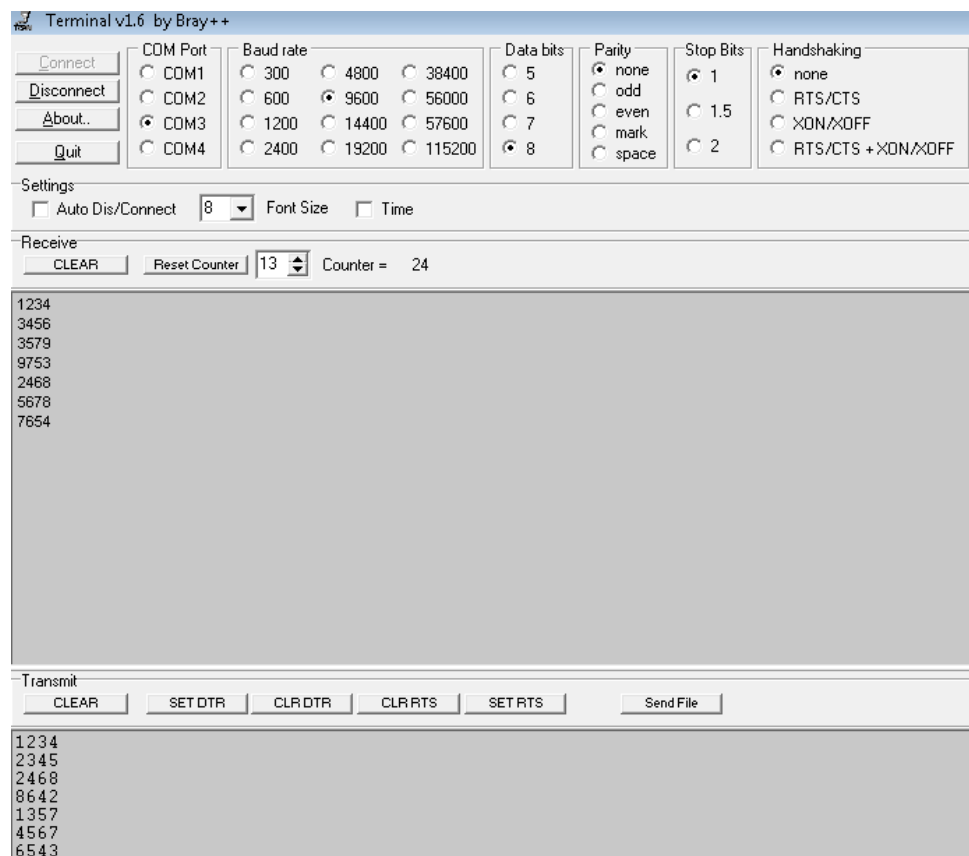


Figure 25 - Terminal Capture 6

3.5.3 UART with the Kalman Filter

After 16 bits were able to be transmitted from the terminal and then back to the terminal, the UART project was then connected to the Kalman Filter design. The connections can be seen in the block diagram below.

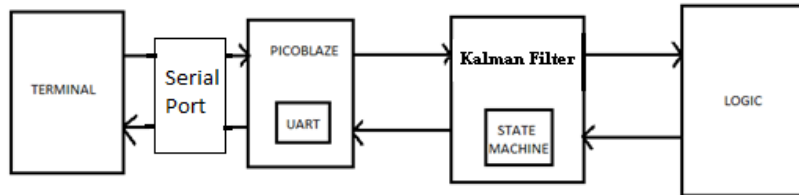


Figure 26 - Block Diagram of UART with Kalman Filter

To make sure this system was working properly, the first 5 input values that were obtained from the Matlab version of the Kalman Filter were sent from the terminal. The actual outputs were then compared to the expected outputs. From the figure below, it is confirmed that the Kalman Filter receiving data from the terminal and sending data to the terminal is working as expected. The values on the top left are the output values of the Kalman Filter, and the values on the top right are the expected output values. Although there are minor differences, these only occur in the least significant bits, so there is no significant difference between these values.

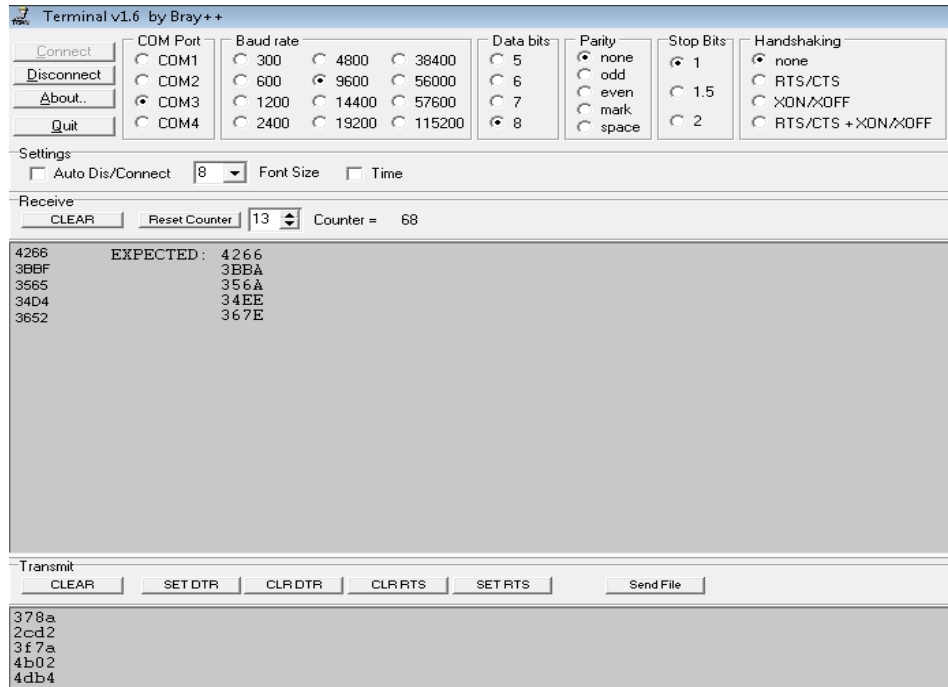


Figure 27 - Terminal Capture 7

After this test, it was clear that data could be streamed into, as well as out of, the Kalman Filter. Although it was already concluded that the Kalman Filter could produce the outputs of the Matlab code when using the Matlab inputs, the UART showed that it can be done much quicker and the results could be seen much easier. These tests not only showed that data could be streamed into and out of the Filter, but this will now be used as a beneficial tool going forward. For example, when ADC testing begins, the digital outputs produced by the ADC from the analog signal can be sent to the UART, which will allow for the quick and easy analysis of the ADC and ADC controller performance.

Chapter 4: Using the Kalman Filter with Real-Time Signals

After completing the tests with the UART, it was time to start working with analog signals. After all, the final goal of the project is to be able to send an analog signal into the project, and output the resulting analog signal. Individual output values have been tested using the UART, but to be sure the Kalman Filter was completely working, it was necessary to duplicate the Matlab graph showing the signal containing all of the outputs. To successfully do this, it is necessary to use an analog to digital converter (ADC) and digital to analog converter (DAC) to stream data into and out of the system using analog signals.

4.1 DAC Selected

The first step to this process is using a DAC to output the values of the Kalman Filter as an analog signal. The first thing we had to do was to choose which DAC to work with. After researching and looking at different DAC's, the easiest way to approach this was to find a PMOD that is easy to use with the Nexys2 board. Using a PMOD give the ability to directly connect the pins of the PMOD into the peripherals of the Nexys2 board, which connect to the FPGA. Digilent does not offer PMODS containing 16 bit DACs, but they do offer PMODS which include 12 bit DACs. The following figures show the schematic of the PMOD with two DACs, and then the actual PMOD itself. Although the PMOD contains two DACs, we will only be using one of the 12-bit DACs because there is only 1 input to our Kalman System. Using a 12-bit DAC instead of a 16-bit DAC means that there is a loss of precision, but since only the 4 least significant bits are lost, the difference in precision does not affect the performance of the Kalman Filter.

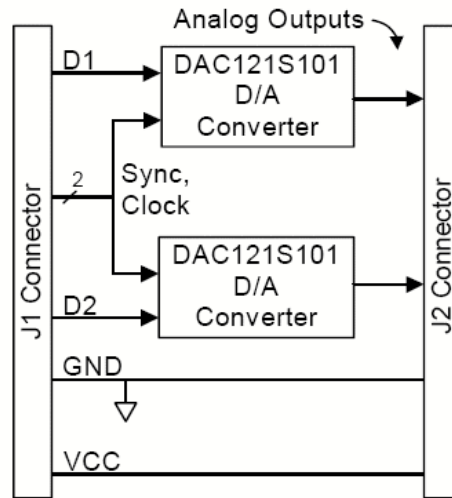


Figure 28 - Schematic of PMOD with two DACs

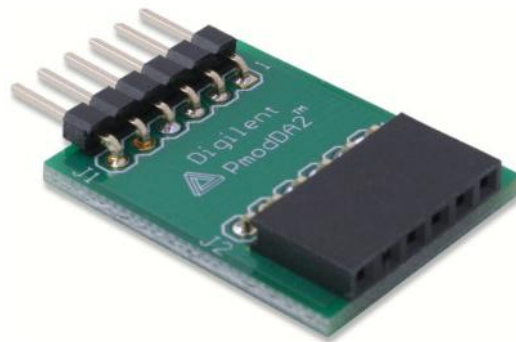


Figure 29 - Digilent PMOD with DAC

This DAC has a settling time of 8 μ s which means that sampling can easily be done at the desired frequency of 25 kHz. Also, it has an output voltage from about 0 to the supply voltage of 3.3 V, which is sufficient for the outputs [4].

4.2 Implementation of the DAC

The next step in our project was to integrate the DAC. If we used the same inputs that the MATLAB program uses, then we could examine how well our Kalman Filter and DAC were working together by comparing the oscilloscope output with the Matlab graph. The figure below shows a block diagram of how we went about our design.

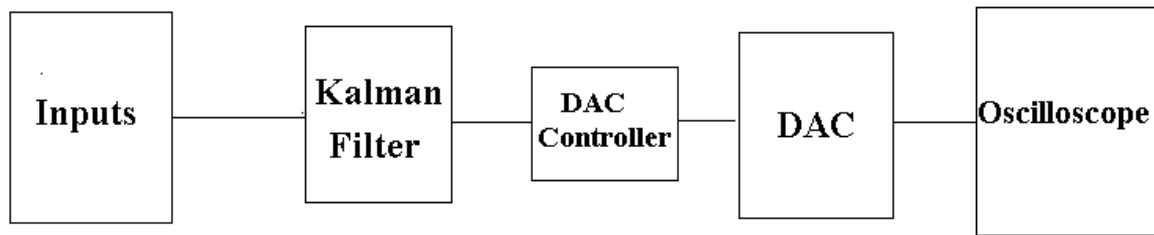


Figure 30 - Block Diagram of DAC Design

We had a series of inputs stored in a file, in which our Kalman filter had access to. We decided to use a 25 kHz clock rate to input the values from that file into the Kalman filter. At the same rate, the Kalman filter would output the calculated values to the DAC. A state machine was used to help with this process of input values to the Kalman filter and then outputting them to the DAC at the correct times. Before we could actually connect the DAC with the Kalman Filter, we had to verify that we could use the DAC properly. From ECE 574, we worked with the other Pmod DAC that Diligent carries. This is an 8 bit DAC. From a project in that class, we were able to display different voltage values depending on the dip switches. Taking this code as a base, we worked on the DAC controller for the DAC which would be used in this project. The figure below shows the approach that we took to implement the DAC controller.

DAC State Machine

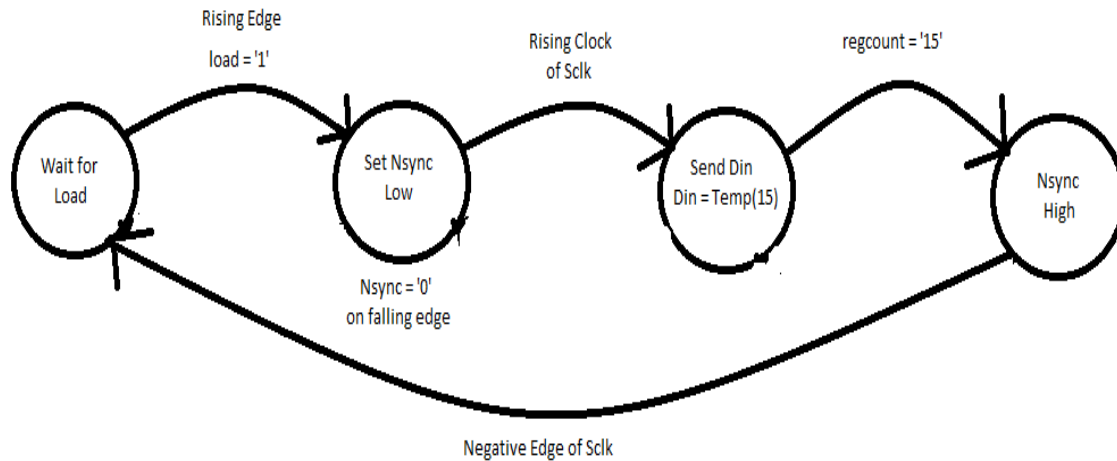


Figure 31 - State Machine of DAC

4.3 Results of DAC Implementation

For this DAC, we are using a sclk of 25MHz to output the serial data. Looking at the diagram, we see that once sync goes low, 16 bits need to be clocked into the DAC on the rising edge of sclk. However, before the first bit is clocked in, the setup time needs to be accounted for. The figure below shows the timing diagram of the DAC that we are using with our project.

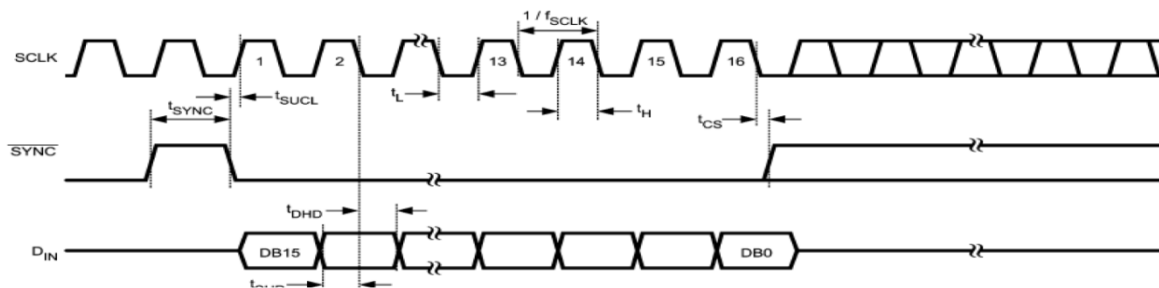


Figure 32 - Timing Diagram of DAC

We used a signal called load to decide when the module was ready to output a signal through the DAC. When load goes high, a 16 bit value called data is loaded into the shift register temp on the rising edge of the clock. The areas surrounded in red on the next figure show this. On the next falling edge of the clock, Nsync is set low. Since the falling edge is used, there will be a half clock cycle before the first rising edge. This accounts for the setup time before the rising edge of the clock starts to send out data. Nsync stays low for 16 rising edges of sclk, so that all 16 bits are clocked out, and then set high on the next rising edge. This can be seen in the yellow areas of the next figure. All 16 bits of temp are shifted out to Din, and this sequence is what the DAC uses to create an analog signal. For our first test, we just wanted to be able to shift out one sequence. The sequence we used was 0000100101100101. When looking at the blue area of the next figure, it can be seen that Din represents that exact value.

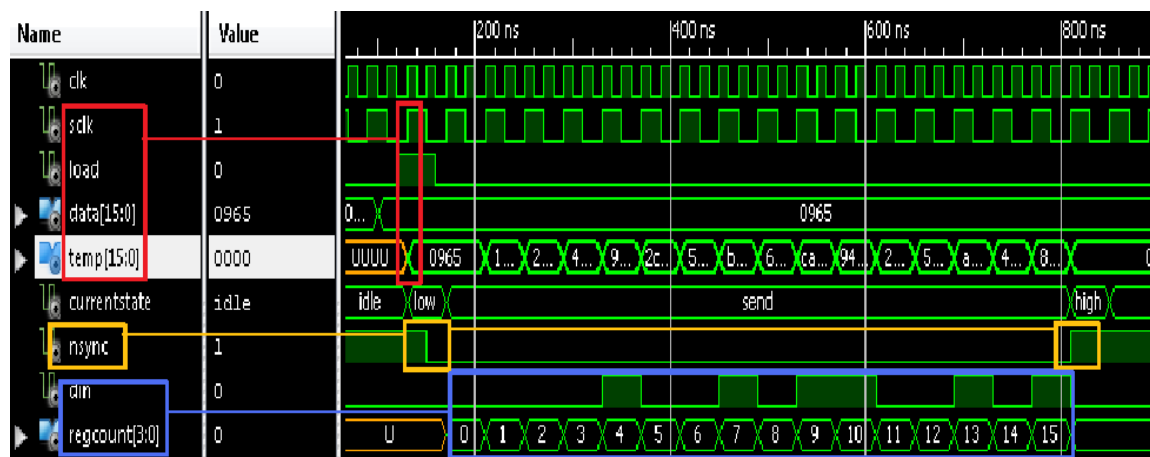


Figure 33 - Test Bench of DAC

After we were able to get our DAC controller working properly with the test bench, we wanted to display the signals on the oscilloscope. The yellow signal represents Nsync. Once that goes low, Din starts to produce the sequence that is being sent to the DAC. This is represented by the green signal. The following figure shows the oscilloscope capture of Din and Nsync.

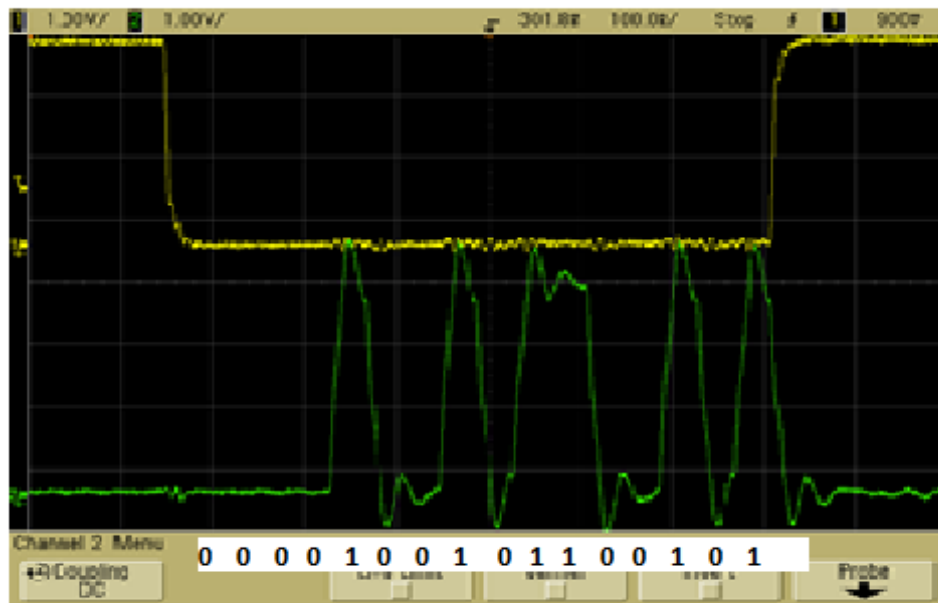


Figure 34 - Oscilloscope Capture of Din and Nsync

As we can see, Nsync has almost a perfect square edge and Din is getting the exact values that were being sent to the DAC. The next step was to use the oscilloscope to make sure our DAC controller worked properly with the DAC. To test this, we changed the value that was being sent to the DAC. The value we send is a 12 bit value, and we continue to increment the value by 1 at a 25KHz rate. When the value consists of all 1s, it rolls over back to all 0s. The signal that this creates is a saw tooth waveform. When looking at the next figure, we can see that our DAC controller and DAC worked perfectly together to produce an analog saw tooth signal.

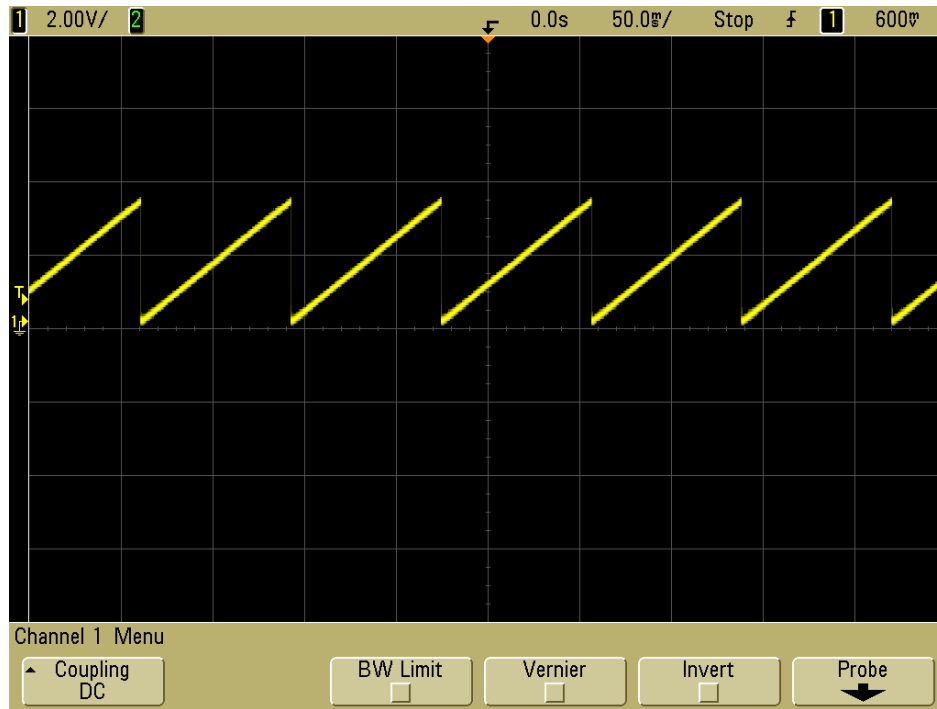


Figure 35 - Capture of Sawtooth Signal

After creating the DAC controller and verifying that it was working it was time to do the same with the ADC.

4.4 Choosing an ADC

After the completion of the DAC, all that was left was the conversion of the input signal from analog to digital. Up to this point, input values were either stored within the FPGA, or they were sent in by dip switches or the UART. Successfully integrating the ADC means that an analog signal can be sampled at the desired frequency of 25 kHz, and those samples can be converted to 16 bit digital values to be sent to the Kalman Filter Logic block. This is the last component that needs to be integrated with the FPGA Kalman Filter design and the DAC. The first step to implementing this design was to choose which ADC to work with. After researching and looking at different ADC's, the easiest way to approach this was to find a

PMOD that is easy to use with the Nexys2 board. Digilent does not offer PMODS which include 16 bit DACs, but they do offer PMODS which include 12 bit ADCs, which again will not lose enough precision to affect the overall results. This ADC has a range of 0 to 3.3 V and can sample at 1 million samples per second, which is well above our sampling frequency of 25 kHz [5]. The following figures show the ADC schematic.

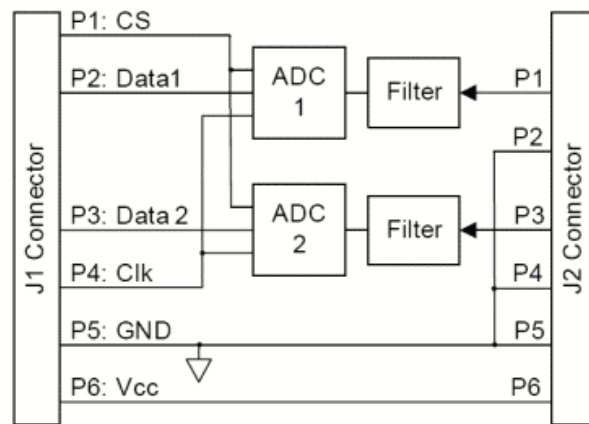


Figure 36 - Schematic of ADC

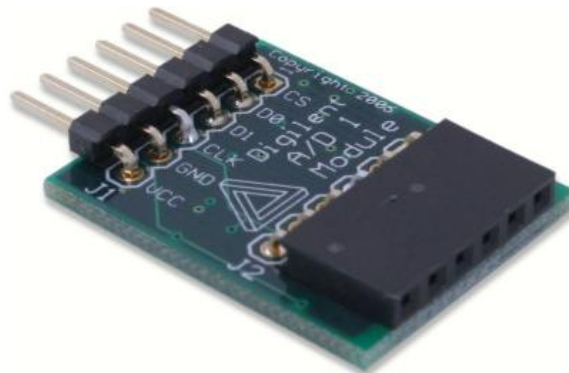


Figure 37 - Digilent PMOD with ADC

After selecting the ADC to work with, we had to come up with a plan to implement the ADC controller. The following figure shows the approach used to implement the ADC.

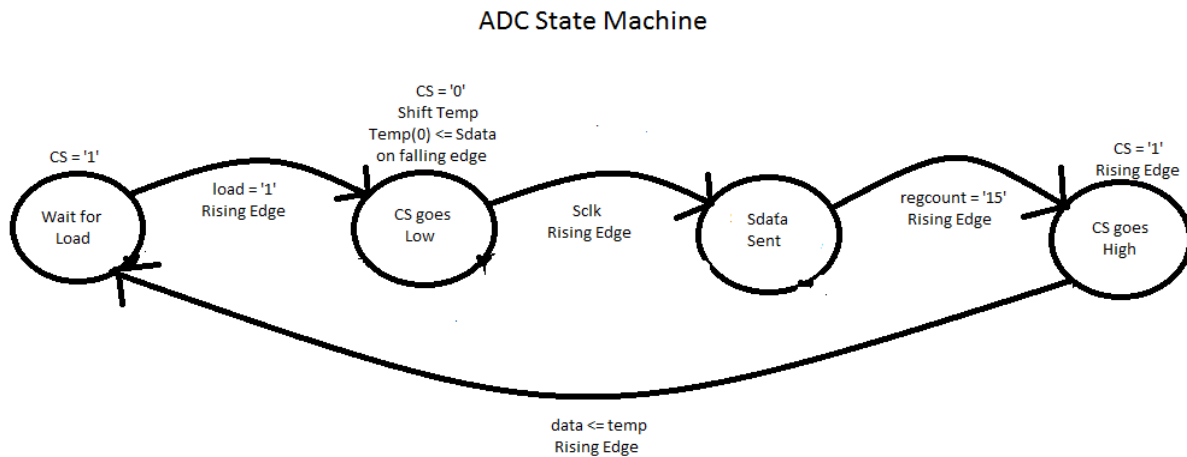


Figure 38 - State Machine of ADC

4.5 Results of ADC Implementation

We then had to look at the data sheet of the ADC to see how it is controlled as well as the timing specifications. The figure below shows the timing diagram of the ADC.

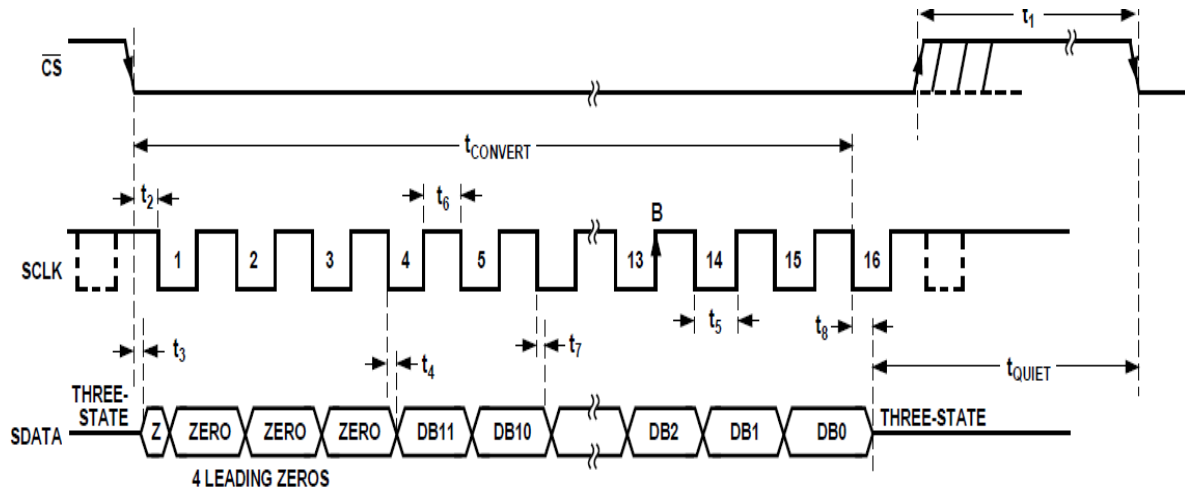


Figure 39 - Timing Diagram of ADC

As we can see, \overline{CS} is what controls when a sample is taken. \overline{CS} is active low and tells the ADC to create a 16 bit value out of the analog sample. Like the DAC, we created a load signal to tell the ADC controller that an input is desired. When load is high, \overline{CS} goes low on the rising edge of $sclk$. This allows for the setup time to be achieved before the first value is input on the

falling edge of the clock. This can be seen in the red area of the following figure. We can also see from the figure below, that temp shifts in one bit at a time on the rising edge of sclk and after 15 cycles, temp is ready to output a 16 bit value, and CS is sent high. The values are available on the falling edge of sclk, but taking them on the rising edge assures that they are valid. The only bit that is taken on the falling edge is the first bit, and this is because it is sent along with the second bit on the first falling edge. This can be seen in the diagram above. Since the first four bits of the input are leading zeros, we only care about the remaining 12 bits because these are the bits that represent sampled value. We then take these twelve bits and store them in adc_out followed by 4 zeros to complete the 16 bit output. To check that our ADC controller was working properly, we inputted the following 16 bit number “0000110111001011”. We then saw that it took one bit at a time and after 15 cycles, it outputted the correct 16 bit output. The values of Sdata were shifted into temp, and this can be seen in the blue area of the following figure. We also checked that adc_out correctly added four zeros to the value of temp to create a 16 bit value for the input to our system.

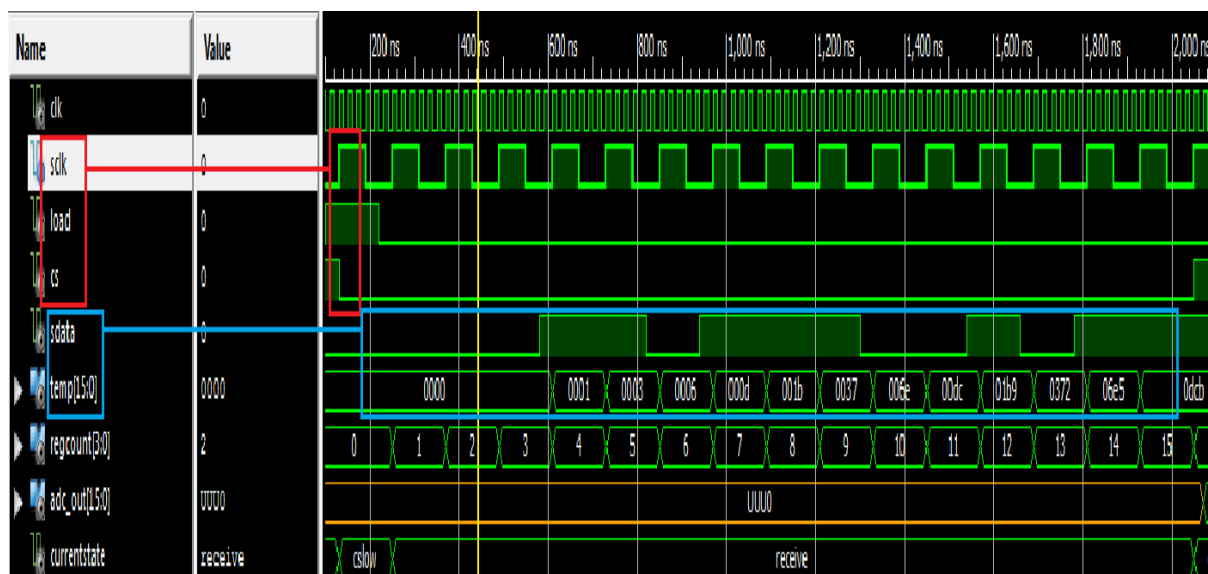


Figure 40 - Test Bench of ADC

Upon completing this small project, we were able to conclude that indeed our design met the specifications for the ADC and was correctly outputting the desired values.

We then hooked up a function generator with a certain voltage and used the seven segment display on the Nexys board to check if the correct voltage was being applied. We put 3.3V through the ADC which corresponds to the Vcc of the Nexys Board, so it is theoretically the highest input voltage to the ADC.

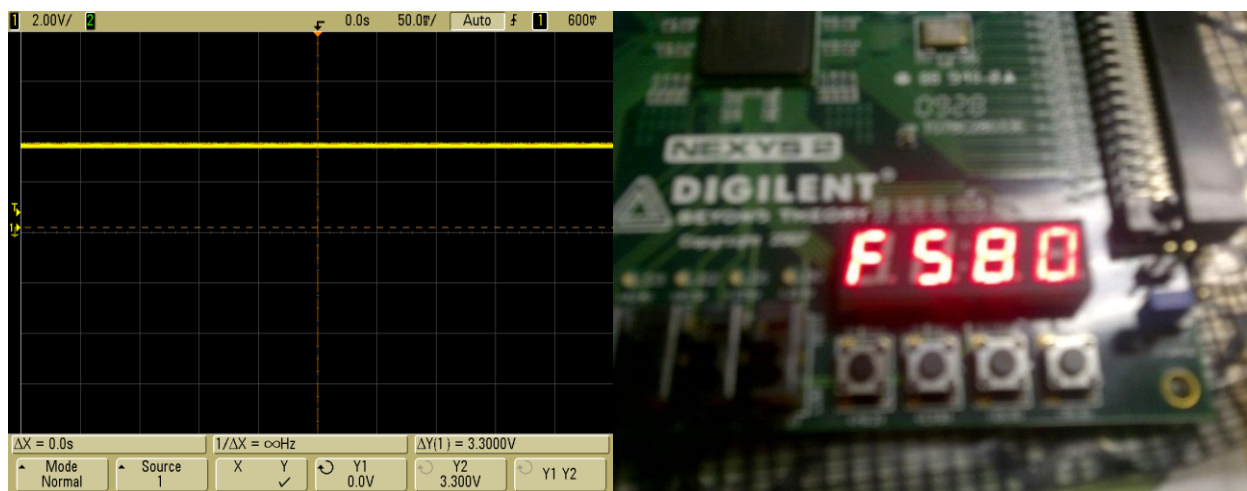


Figure 41 - Seven Segment Display of 3.3V

As we can see from the pictures above, a voltage of 3.3V was close to the max value that the Nexys board can take and once again proved that our ADC was working correctly. We expected the digital value to be around FFF0, and this clearly is.

4.6 Results of ADC implementation with UART

The next thing we did was to make sure we could send a signal from the ADC to the UART and make sure that the values displayed in the UART correctly matched the signal sent from the ADC. This is where working with the UART earlier provided us with a very useful

tool. We used the function generator to create a ramp wave that ranged from 0 to 3.3 V at a frequency of 5 Hz. The graph of this figure can be seen below.

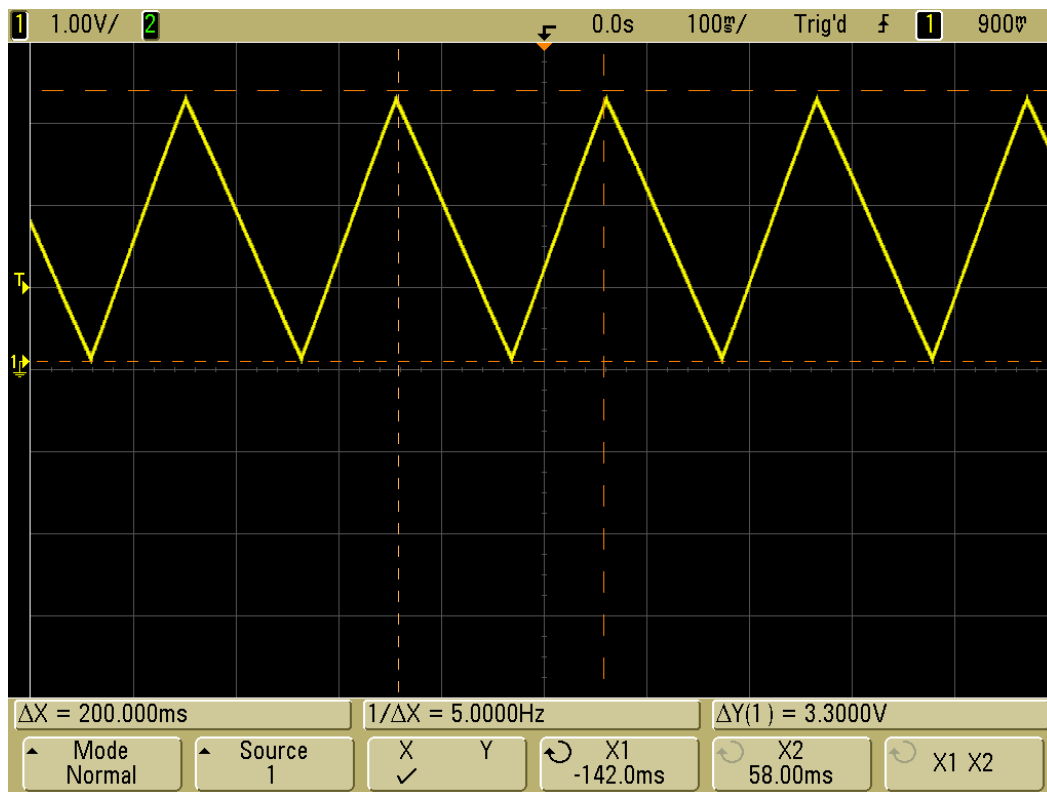


Figure 42 - Function Generator

We used the ADC and ADC controller to sample this signal at 125Hz. To verify that we could sample properly using the ADC we used the UART to show each sample represented digitally. Since this is a ramp wave, the values sent to the UART should start around 0000 and continue to increase until it reaches about FFF0. At this point, the values should decrease until it gets back to 0000. The sequence is then repeated. We took the values captured for one cycle of the waveform and plotted them using Microsoft Excel. As we can see from the figure below, the graph matches one cycle of the analog input signal.

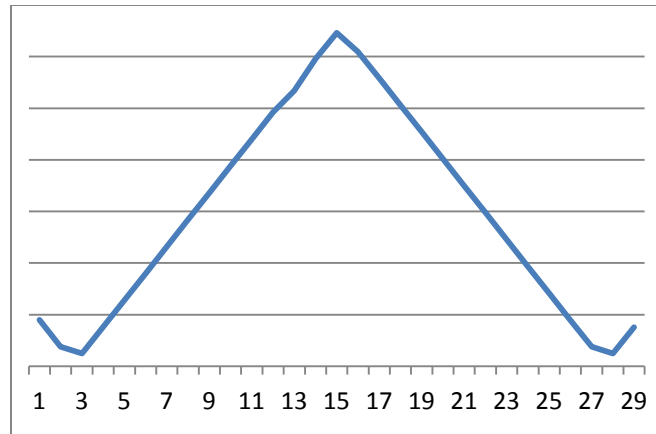


Figure 43 - ADC with UART

Since the outcome from above matches what was expected, this shows that the UART was correctly receiving the 16 bit digital signal to match the right ramp wave sent by function generator. At the end we successfully saw that the ADC was working properly by checking the values sent to the UART.

Chapter 5: Implementing the Design of the Complete System

Designing a complete system to input an analog signal, process the data, and then output an analog system is the overall goal of this project. It involves using the ADC, Kalman Filter, and DAC to accomplish a complete system where data is streamed in and out.

5.1 Implementing the Kalman Filter with the DAC

Now that we knew our DAC was working properly, it was time to implement it with our Kalman Filter. We first took the Matlab code that we converted and produced a graph using 100 inputs. We took these 100 inputs and converted them into 16 bit values which could be sent into our Kalman filter. We stored these values in an array inside our project. The complete design contained the Kalman Filter logic as well as the DAC controller. The design input the stored values into the Kalman filter logic block at a rate of 50KHz. After the values were outputted from the Kalman filter logic, they were sent to the DAC, using the DAC controller, to create an analog signal.

5.2 Implementation of ADC to DAC

We have already shown that the ADC and DAC could successfully be implemented with an FPGA design using the ADC and DAC controllers, so now it was time to tie them together. This is important because it will verify that an analog signal can be sampled, converted to digital, then processed, converted back to analog is output. Once it is proven that this can be done with no processing involved then we can attempt to integrate the Kalman Filter.

In our final project we will be sampling at 25 kHz. So before we connect our Kalman Filter, we wanted to make sure we can sample at this rate and still produce the correct output. In this module, a state machine was created to sample the input signal, and send the output to the DAC. On each rising

edge of the 25 kHz clock, the loads for both the ADC and DAC controller are sent high. The next couple of states determine when the loads of each controller should be sent back low. These states tell the ADC controller to sample a signal, and the DAC controller to output the digital value that is given. This repeats every rising edge of the 25 kHz clock. The state machine can be seen below.

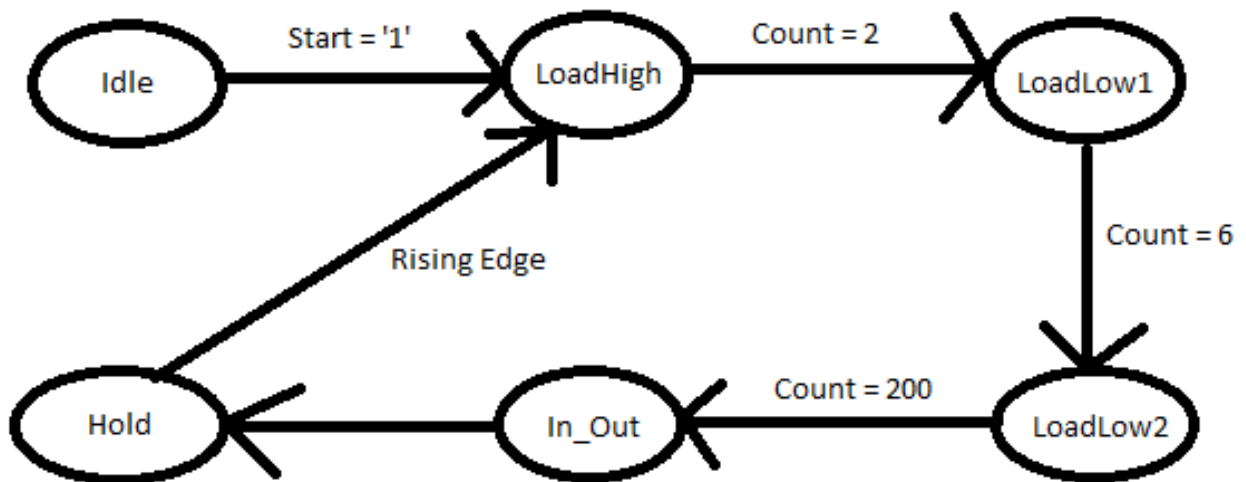


Figure 44 - State Machine of ADC to DAC

5.3 Implementation of Complete Design

After successfully creating a module which used both the ADC and the DAC, the last step that had to be done was to add the Kalman Filter between the two devices. Inside our top level Kalman Filter module there are three main components. The ADC controller, DAC controller, and Kalman Filter Logic block. Also inside this design are the registers which store the previous values of the 3 X values and 9 P values, the current input sample, and the current output value. The X, P, and input value are loaded into the Logic and the output value is loaded to the DAC all on the clock edge. To successfully implement this design, the state machine from the ADC to DAC module was used to take care of the timing of when samples should be taken

and when values should be loaded. The overall block diagram of the entire project can be seen below.

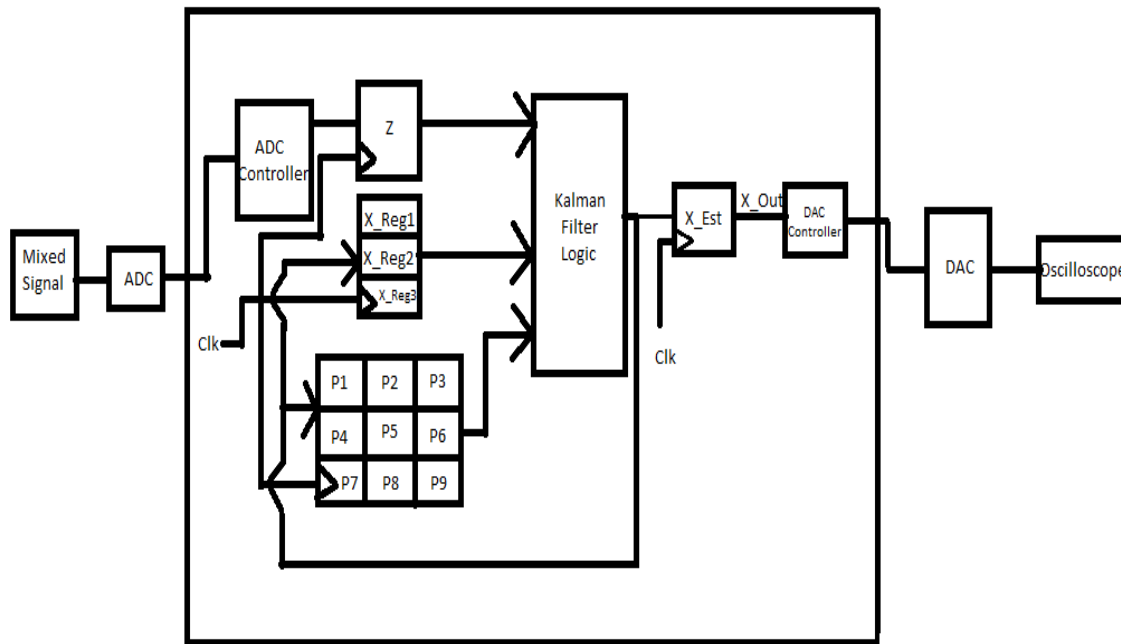


Figure 45 - Block Diagram of ADC to DAC with Kalman Filter

5.3.1 Modifications to the Overall Design

After integrating our ADC and DAC to the Kalman Filter to create a complete system, we performed some tests. These tests will be talked about further in the next section but what we saw in the original results was that there were some problems within our design.

The first modification we made was how the ADC sent value to the Kalman Filter, and how the DAC received values from the Kalman Filter. Our filter expects to take in a signal with no offset, so that the sine wave has negative and positive values. Since both the ADC and DAC range from 0 to 3.3V, we needed to offset the wave forms so there was no negative voltage. This messes with how the Kalman Filter sees the samples, meaning that we will have to go into the design and make the necessary changes to eliminate the offset inside the Kalman Filter.

Although this was a necessary step, it did not solve our problem. We spent a few days going over the DAC and ADC, making sure that the correct values coming out of the Kalman Filter were being sent to the DAC at the correct times. Also, we made sure that the ADC was sampling at the correct rate and sending the correct values to the input of the Kalman Filter. Since we had previously completed smaller projects testing these components in a similar way, we were pretty sure that they were working correctly, and were connected properly. What we found was that our instincts were correct, and that these components were working with the Kalman Filter properly. We did go through our main state machine that controls the sampling and made some minor changes so that it made more sense.

The next thing we wanted to do was look inside the Kalman Filter and see if we could find something wrong. Something that was bothering us was that when we sent the original 200 Matlab inputs through our Kalman Filter, we were able to produce a graph that was very similar to Matlab's. But, when we put in a constant value, such as positive 0.5, our Kalman Filter would not produce a constant output at the voltage representing that value. What it did was continuously rise from that voltage, then decrease back down to that voltage. The following figure shows the Matlab graph of what the Kalman Filter should do.

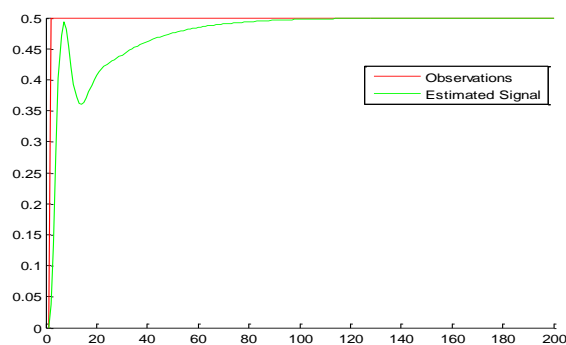


Figure 46 – Matlab output for constant input of 0.5

The next figure shows the actual output of our Kalman Filter.

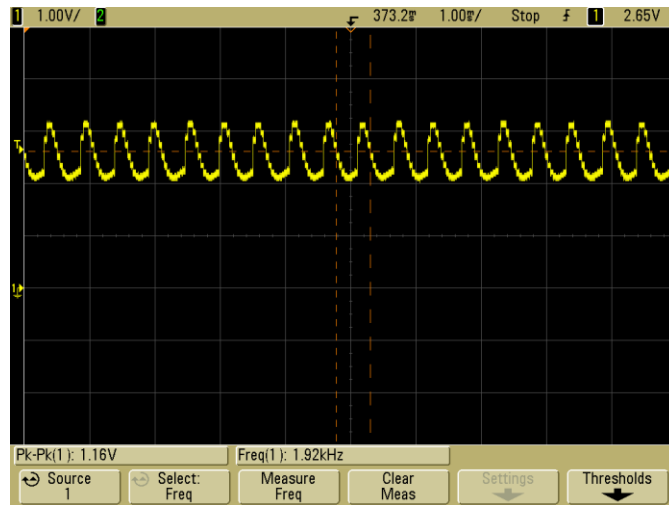


Figure 47 – Output of our Kalman Filter for a constant value of 0.5

So the question we asked ourselves was why does the filter work for the original inputs, but does not work for a constant input like this one? We remembered that the values we used for the Matlab inputs were scaled down to be between values of -1 and 1. In doing this, all values of the three outputs(x_1 , x_2 , x_3) were also between -1 and 1.

When we have been inputting our signal, we have made sure that all of the input values are between -1 and 1. However, we did not look at the three output values to see what they were doing. Using Matlab, we input a constant signal of 0.5. We checked all of the x output values and found that the x_2 values went way above the absolute value of 1. Since our main output, x_1 , depends on the previous values of x_2 , the overflow of x_2 was creating major problems for the rest of our outputs. The values for x_2 can be seen below.

	192	193	194	195	196	197	198	199	200
1	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000
2	-3.1561	-3.1562	-3.1562	-3.1562	-3.1563	-3.1563	-3.1563	-3.1563	-3.1564
3	0.9997	0.9998	0.9998	0.9998	0.9998	0.9998	0.9998	0.9998	0.9998

Figure 48 – Matlab output values for a constant value of 0.5

At this point, we knew the input had to be scaled down. This system could not support an input which is a constant of half of the positive range. We continued to decrease the constant input until we could output the correct signal. This happened around a value of positive 0.15. Looking at the Matlab values here, we see that x_2 does not go over the absolute value of 1.

	327	328	329	330	331	332	333	334	335	336
1	0.1500	0.1500	0.1500	0.1500	0.1500	0.1500	0.1500	0.1500	0.1500	0.1500
2	-0.9471	-0.9471	-0.9471	-0.9471	-0.9471	-0.9471	-0.9471	-0.9471	-0.9471	-0.9471
3	0.3000	0.3000	0.3000	0.3000	0.3000	0.3000	0.3000	0.3000	0.3000	0.3000

Figure 49 – Matlab outputs for a constant input of 0.15

The next thing we did was look at inputting a saw wave in Matlab. We had previously tried this but the output did not look right. Here is the Matlab graph for inputting a sawtooth waveform through the Kalman Filter with a range from -1 to 1.

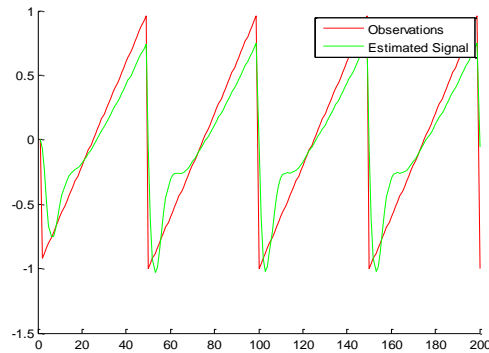


Figure 50 – Matlab output of a 500Hz sawtooth wave

Using a sawtooth waveform that used most of range between -1 and 1, here is the DAC output of our Kalman Filter.

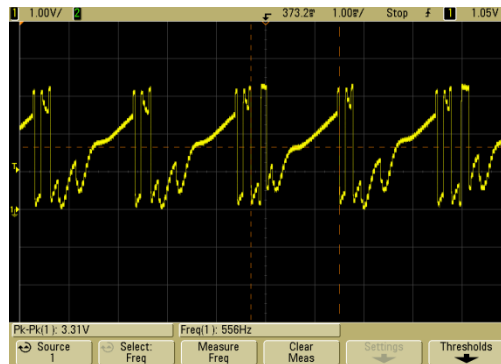


Figure 51 – Output of our Kalman Filter for a 500Hz sawtooth wave

When looking at the Matlab values once again, we saw the same problem with x2.

Variable Editor - x_

Stack: Base

No valid plots for: x_(1,1)

x_ <3x200 double>

No valid plots for: x_(1,1)

	16	17	18	19	20	21	22	23	24
1	-0.2459	-0.2327	-0.2190	-0.2031	-0.1839	-0.1616	-0.1365	-0.1092	-0.0805
2	1.6371	1.6502	1.6537	1.6482	1.6336	1.6102	1.5777	1.5363	1.4860
3	-0.4421	-0.4639	-0.4792	-0.4886	-0.4926	-0.4922	-0.4880	-0.4805	-0.4701

Figure 52 – Matlabe output values for a 500Hz sawtooth wave

Once again, we continued to scale down the Matlab input until we were able to get values which did not exceed the absolute value of 1. For the sawtooth waveform this happened at about 0.4 times the original range. The values for the outputs can be seen below.

Variable Editor - x_										
x_ <3x200 double>										
	15	16	17	18	19	20	21	22	23	24
1	-0.1308	-0.1230	-0.1164	-0.1095	-0.1015	-0.0920	-0.0808	-0.0682	-0.0546	-0.04
2	0.8066	0.8186	0.8251	0.8269	0.8241	0.8168	0.8051	0.7889	0.7682	0.74
3	-0.2070	-0.2211	-0.2319	-0.2396	-0.2443	-0.2463	-0.2461	-0.2440	-0.2403	-0.23

Figure 53 – Matlab output values for 0.4 times a 500Hz sawtooth wave

Now there is no overflow inside the filter. So, now we input a sawtooth waveform to our Kalman Filter which only used 0.4 times the range. The Matlab graph as well as the DAC output can be seen below.

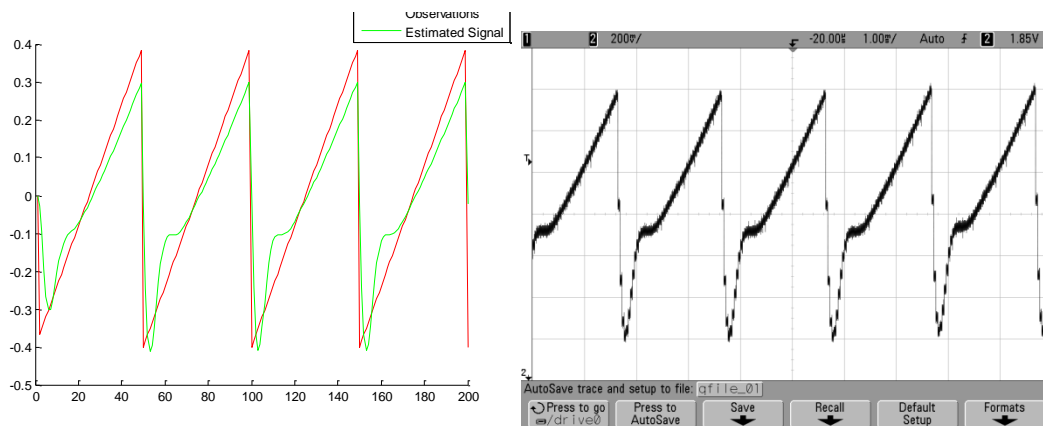
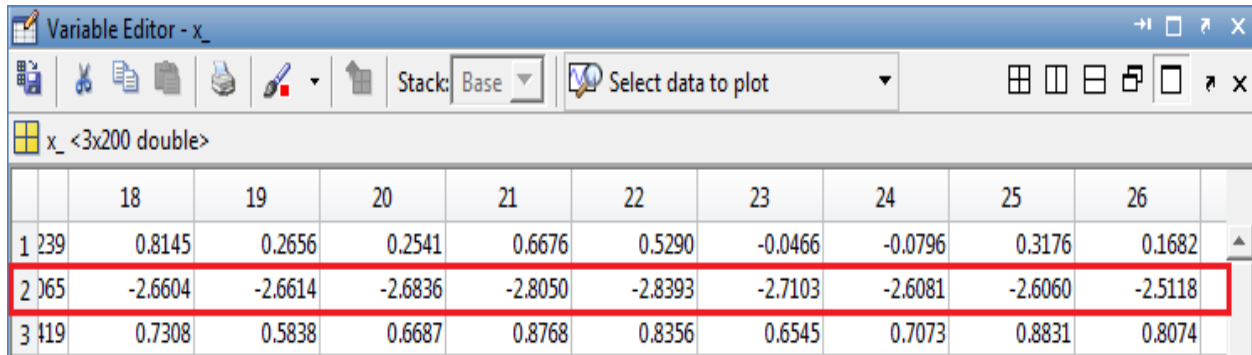


Figure 54 & 55 – Matlab and our Kalman Filter output for 0.4 times 500Hz sawtooth wave

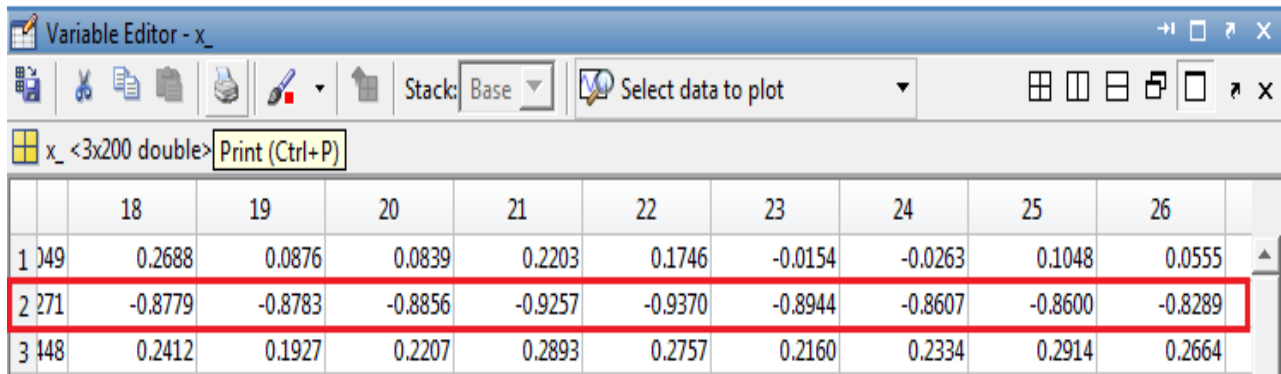
As we can see, our new output exactly matches that of the estimated signal in Matlab which is shown in green. Now that we had figured this out, we wanted to implement it within our overall figure. The first thing we did was look at the Matlab values of the outputs when the input was the mixed signal. As we can see from the figure below, the x2 values once again exceed the absolute values of 1.



		18	19	20	21	22	23	24	25	26
1	239	0.8145	0.2656	0.2541	0.6676	0.5290	-0.0466	-0.0796	0.3176	0.1682
2	065	-2.6604	-2.6614	-2.6836	-2.8050	-2.8393	-2.7103	-2.6081	-2.6060	-2.5118
3	419	0.7308	0.5838	0.6687	0.8768	0.8356	0.6545	0.7073	0.8831	0.8074

Figure 56 – Matlab output values for the mixed signal input

So once again, we scaled down the input until we could see the all of the output values remained within the range of -1 to 1. For this mixed signal, this range was about one third of the entire range. The values of Matlab for this input can be seen below.



		18	19	20	21	22	23	24	25	26
1	049	0.2688	0.0876	0.0839	0.2203	0.1746	-0.0154	-0.0263	0.1048	0.0555
2	271	-0.8779	-0.8783	-0.8856	-0.9257	-0.9370	-0.8944	-0.8607	-0.8600	-0.8289
3	448	0.2412	0.1927	0.2207	0.2893	0.2757	0.2160	0.2334	0.2914	0.2664

Figure 57 – Matlab output values for 0.33 times the mixed signal

This was a long debug process where many options were examined. However, we found that this was the problem causing all of the errors in our overall design, and the solution was to scale the input down so that the amplitude was 1 volt peak to peak with a DC bias of 1.65 Volts. This allowed for our system to work properly. The next section will show all of the results of the tests.

Chapter 6: Results of Overall Implementation

This section will show all the results of the test performed to finish the complete design of the system.

6.1 Kalman Filter to DAC

The next two figures show the results of this test compared to the Matlab graph. The two seem to match perfectly. We used 16 bit values in our Kalman filter, but were only able to send 12 of these bits to the DAC because our DAC can only take in 12 bit values. This is not a something that truly affects the outputs since only the least 4 significant bits are lost, meaning a small amount of precision is lost.

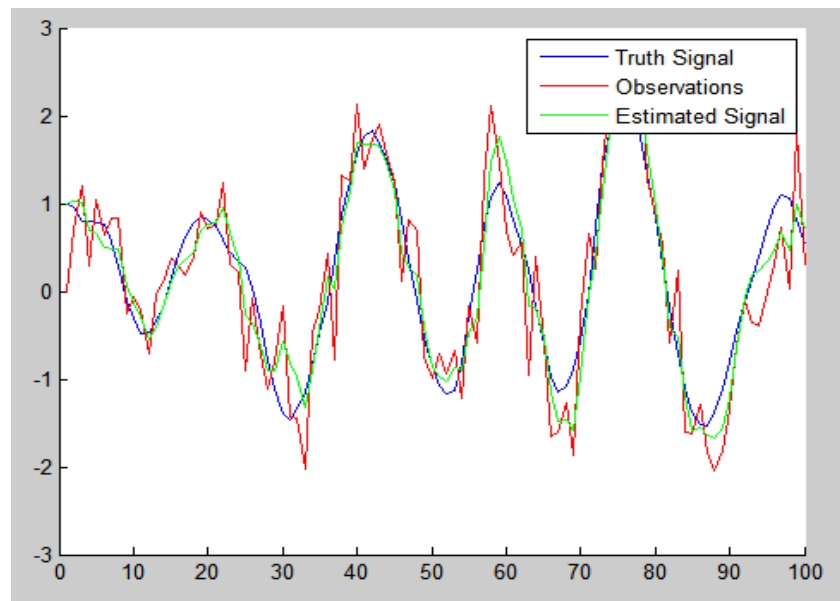


Figure 58 - Matlab Graph

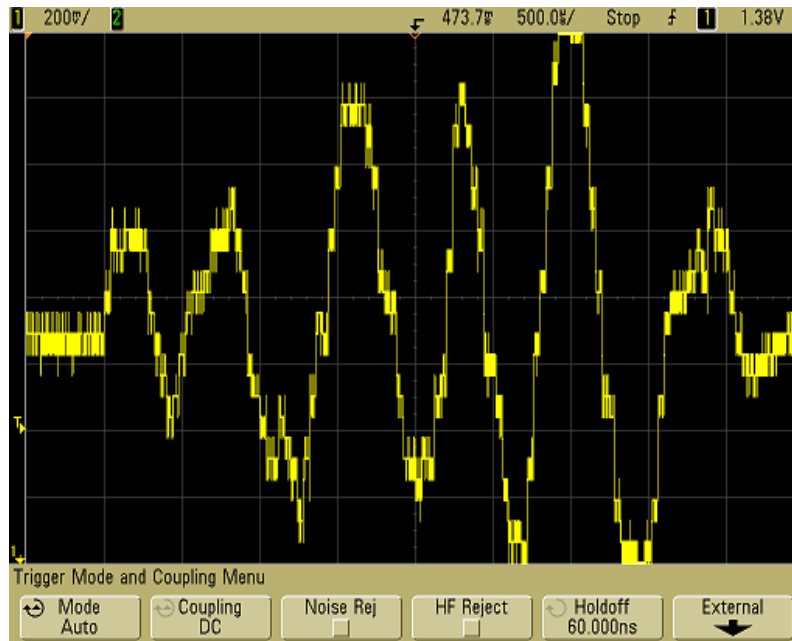


Figure 59 - Graph Produced through DAC

The results were very successful. At the end, the output produced by Matlab was extremely accurate to the output produced by our DAC using the Kalman Filter inputs.

6.2 Results of ADC to DAC

After completing our module to interface the ADC with the DAC, we then tested our module to check the outcome. What is happening is that a sample is being taken by the ADC, converted to a digital value and loaded into the module. That value is then sent to the DAC to be converted and output as an analog value. The output analog signal should recreate the input analog signal after digital processing. The figure shows the input signal in yellow and the output signal in green represented on the oscilloscope. The signal is 100 Hz, so since the sampling is done at 25 kHz, which is far above the Nyquist rate specifications, the two signals should be identical. Since each division on the oscilloscope is 1 V, we see that the signal is 3.3 V. Also, the bottom of the figure shows that the signal is 100 Hz.

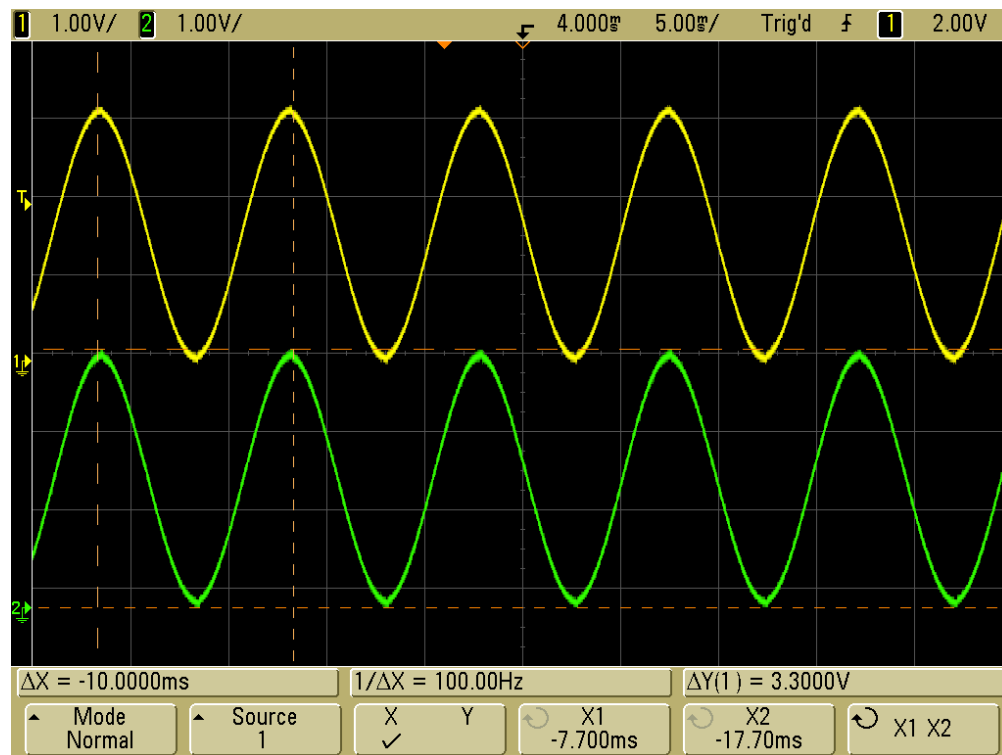


Figure 60 - Oscilloscope Capture of Input and Output Signal at 100Hz

The next step was to input a 2.5 kHz signal. Since the sampling frequency is 10 times greater than this, we will not see a crisp sine wave like above. Instead we will see a 2.5 kHz sine wave which has 10 clear steps every period. This was done to show the 10 samples taken and then sent out during every clock cycle. At the bottom of the figure we can see that the signal is 2.5 kHz.

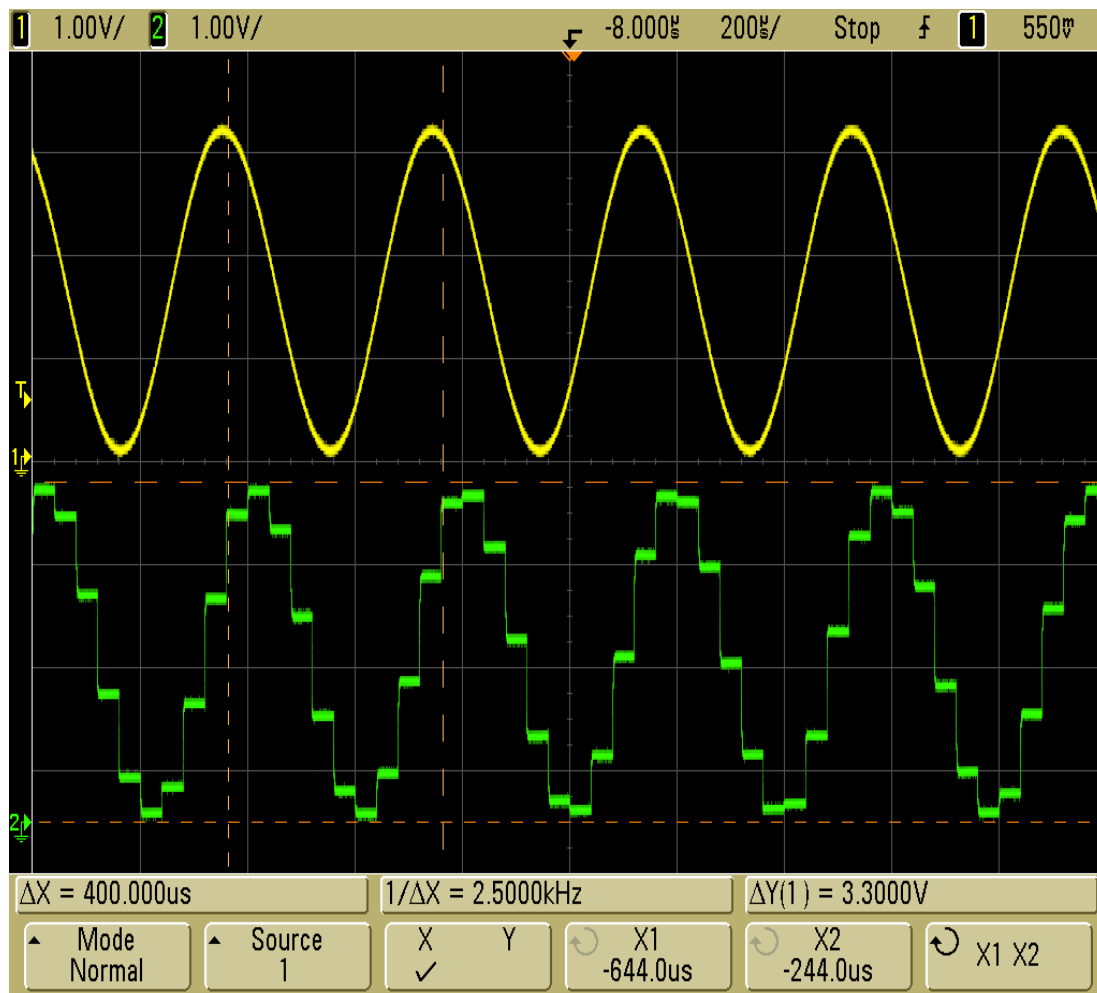


Figure 61 - Oscilloscope Capture of Function Generator Signal at 2.5 kHz

6.3 Mixed Signal

Now that the overall design has been implemented, a mixed signal was created to test the Kalman Filter. The mixed signal consists of the sum of two sine waves of equal amplitude. One of them has a frequency of 6.25 kHz, and the other has a frequency of 500 Hz. The point of this version of the Kalman Filter is to accept signals at a certain frequency and then try to reject signals at other frequencies. In this case, the Kalman Filter wants to accept the 500 Hz sine wave and reject the 6.25 kHz sine wave.

Before creating the mixed signal, we wanted to verify that the Kalman Filter was working properly. To do this we just used the function generator to input a 500 Hz signal. Since we know this signal should be passed through, we should see a 500 Hz signal at the output as well.

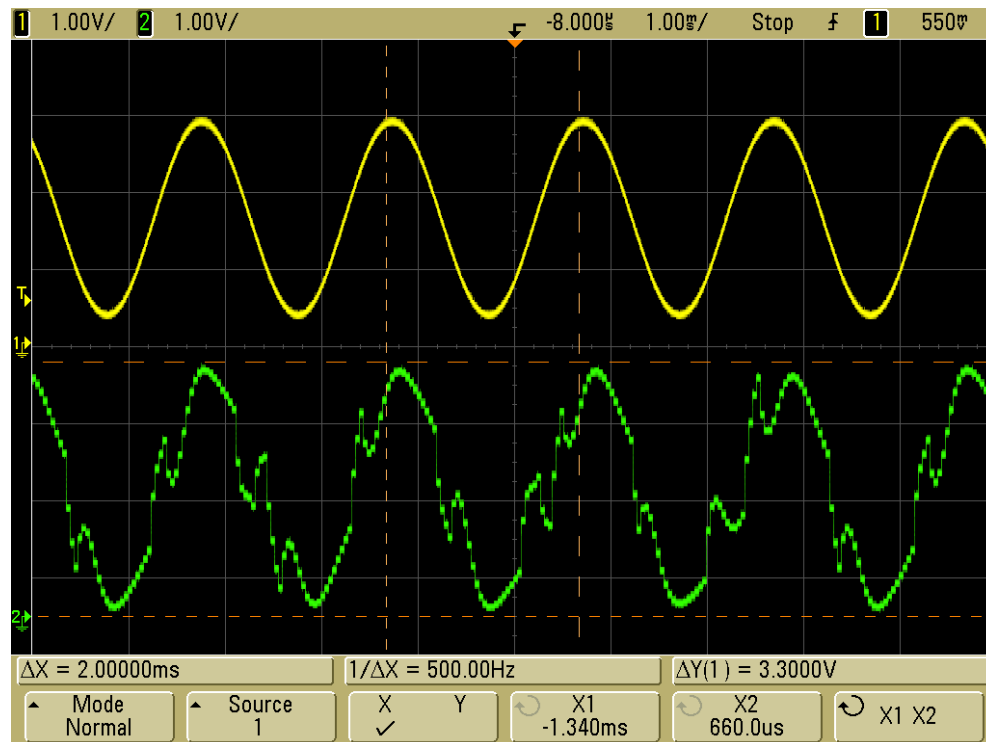


Figure 62 - 500Hz Signal

After seeing this signal we went back and performed the debuggin talked about in the last section. After that was completed, the new oscilloscope figure looked like this.

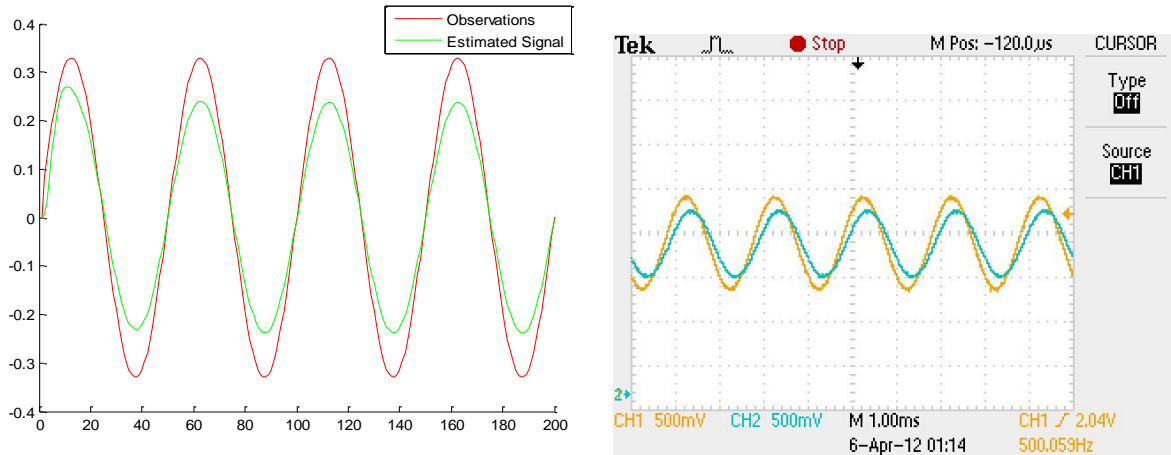
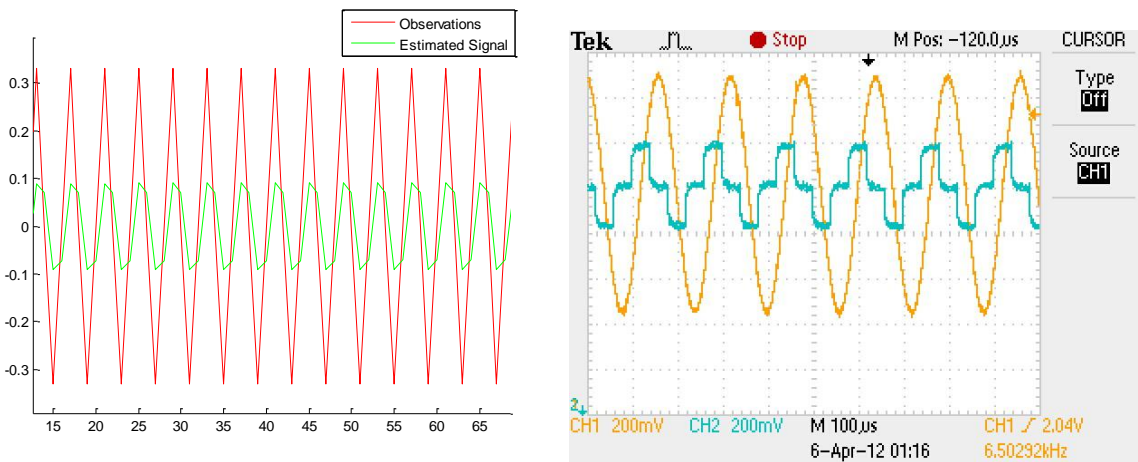


Figure 63 & 64 – Matlab output and our Kalman Filter output of 0.33 times a 500Hz signal

The two figures match which means that our Kalman Filter is working properly. On both figures, we see that the output loses a little bit of amplitude. The only difference in the figures is that our output is shifted by one sample. This is because we take in a sample, perform the calculations on it, and then output it on the next clock cycle. The next step was to perform the same process using the 6.25KHz input. The Kalman Filter should be trying to reject this signal from passing through the filter. The two figures below show the Matlab graph and the DAC output of our filter.



Figures 65 & 66 – Matlab output and our Filter output of 0.33 times a 6.25KHz signal

After verifying that our Kalman Filter did pass the 500 Hz signal and rejected the 6.25 KHz signal, we created the mixed signal. To do this, we used an op amp and three 10 kΩ resistors to create a signal

summer. The schematic for this can be seen below, as well as the pin layout of the op amp that was used.

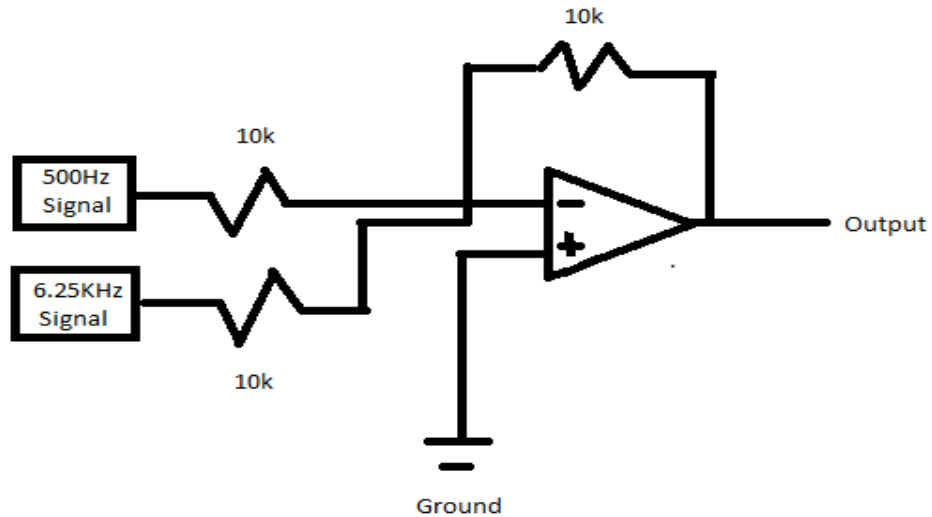


Figure 67 - Schematic of OP AMP

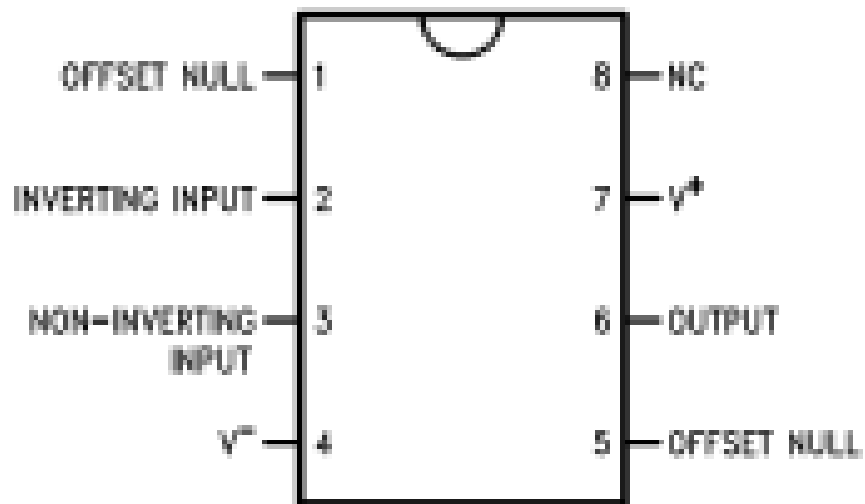


Figure 68 - Pin Layout of Op Amp

After building this op amp, two function generators were connected to this signal summer to create the desired mixed signal. As previously mentioned, the two signals have equal amplitude, but one has a frequency of 500 Hz and the other 6.25 kHz.

The last step was to test our filter with the mixed signal. Once again, our analog input signal was decreased to only a third of its original range. The figure below shows the Matlab graph for the mixed signal. We see that the range has double from the single signal inputs. The 500Hz signal remains pretty much unaffected, but the 6.25KHz signal is greatly rejected.

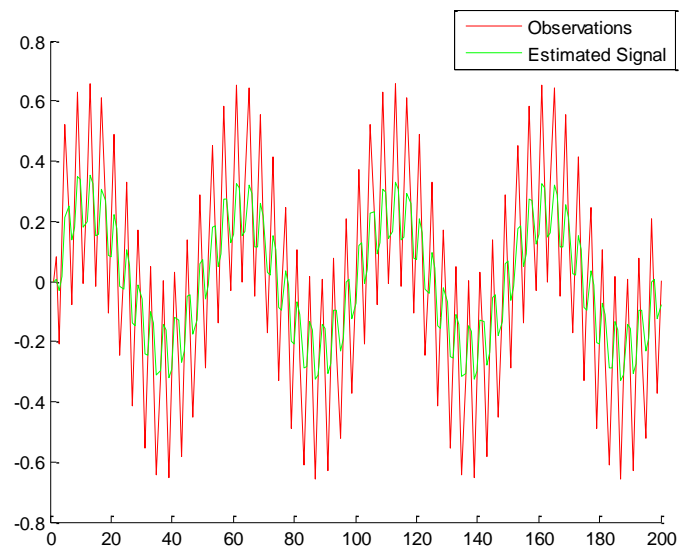
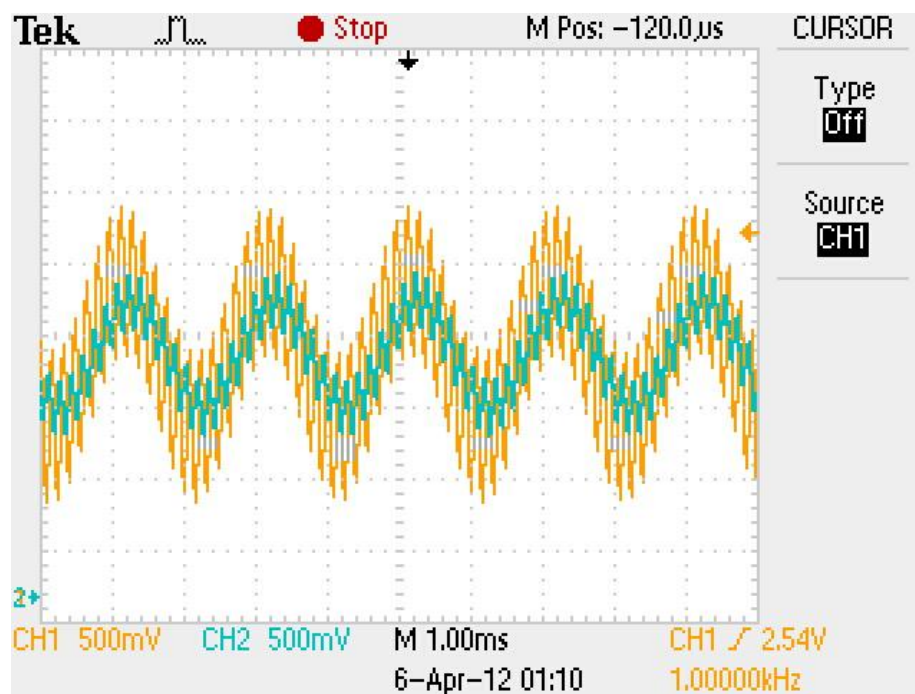
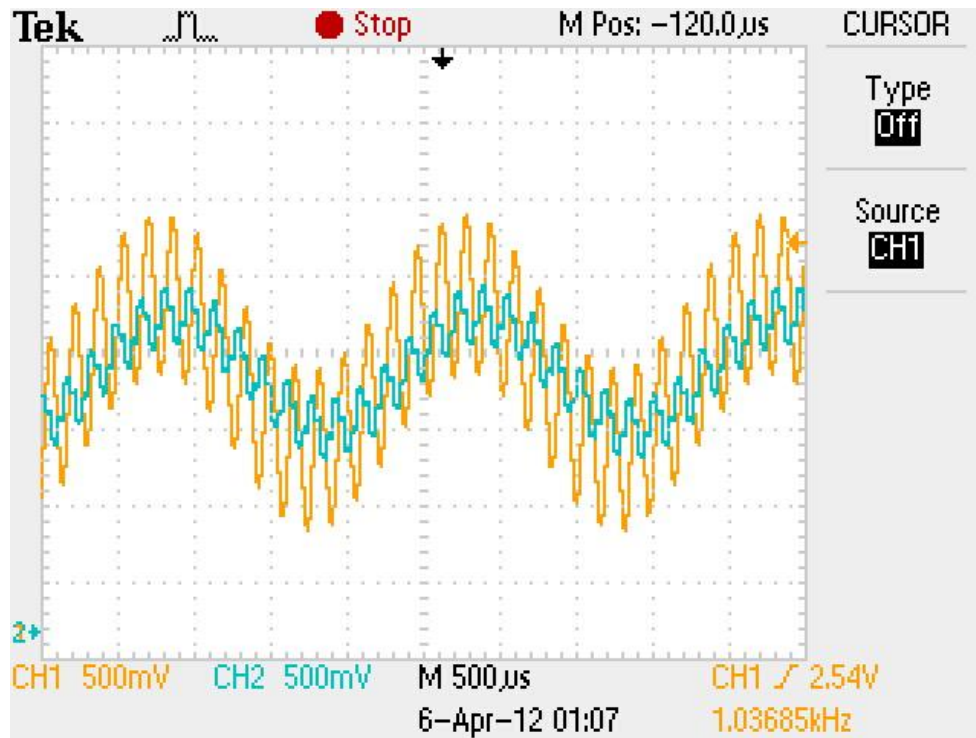


Figure 69 – Matlab output for the 0.33 times the Mixed signal Input

The following two figures show the DAC output of our Kalman Filter.



Figures 70 & 71 – Our Kalman Filter outputs for 0.33 times the mixed signal input

We see that the output of our Kalman Filter matches the Matlab graph of what is expected.

Once again, the only difference is that the output wave is shifted by one sample compared to the input

wave, but overall, our signal completely matches the Matlab version. This shows a successful implementation of our project, meaning the project has been completed.

Chapter 7: Conclusion

In conclusion, we have created a successful FPGA based Kalman Filter, which interfaces with an ADC and DAC to form a complete system that streams analog data in and out. We have matched our output graphs in various tests with those generated using a Matlab based software implementation to prove that our FPGA based version of the Kalman Filter can execute its functions just as it was designed to do. We were able to convert the existing Matlab code from a software version to a hardware design and implement it using an FPGA and a hardware description language. We also created an ADC controller and DAC controller within the FPGA so that the Kalman Filter, ADC and DAC could be integrated together and used for testing purposes. The results showed that this system could stream in a mixed signal containing sine waves with frequencies of 500Hz and 6.25KHz, process the data within the FPGA, and stream out an analog signal which has the 500Hz sine wave unaffected, but the 6.25KHz sine wave greatly diminished in amplitude.

Not only do the results show that this MQP was completed successfully, but this also means that an FPGA based Kalman Filter is a practical solution for the Kalman Filter within the Precision Personal Locator Device. We have shown that an FPGA has the ability to perform the functions of a Kalman Filter, and that it can work with an ADC and DAC to receive and send analog information. A complete system like the one we have built can be altered, and added onto, to perform the tasks of the Kalman Filter in the PPL system, and can be included within the implementation of the actual system to process the data in real time.

With that in mind, we have recommendations for future work which we believe will help to reach the ultimate goal of implementing the Kalman Filter of the Precision Personal Locator device with an FPGA that can process the data while it is received rather than using a software program at a later time. In this report, we have laid out the process we used to create the complete system. In addition some experimentation that was performed and some errors that occurred were documented. If we

were to create another system we would now know the best path to take, which FPGA modules needed to be created, and what other components needed to be used to obtain a better solution. We also now understand how to choose the best components and the most efficient ways to go about converting the Matlab calculations into a hardware design and the best ways to analyze small sections of the system, and what tests should be performed at different steps to allow for the smoothest design process possible. What all of this means is that another group can learn from everything that has been documented here to enhance our design to support a more complicated version of a Kalman Filter. We believe the next step is for a group is to adjust the FPGA design of our Kalman Filter to allow for multiple inputs and outputs so that the filter can perform the processing of the Precision Personal Locator device. Also, for future work, the current software version of the Kalman Filter for the Locator Device should be replaced by the hardware system that we have created so that incoming data is processed in real time.

This MQP was a very important project for the education of this group. It included some difficult periods which gave us the opportunity to increase our knowledge and gain experience with this type of work. During this process, our eyes were opened to a number of different tools which can assist us during a process like this. We also learned the importance of working with small designs, testing these designs, and then building upon them to work towards the overall design. This showed us the best ways to debug certain problems and what kinds of problems cause certain errors to occur. Overall, we feel that we have come out of this experience with much more experience and knowledge, and we have created a design which performs exactly how it is supposed to.

References

- [1] *First Responder Locator System*. (2008, April). Retrieved October 2011, from Precision Personal Locator Project: http://www.wpi.edu/Images/CMS/PPL/PPL_Flier_Apr08.pdf
- [2] Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Research Institute of Advanced Study* , 12.
- [3] Ribeiro, Maria Isabel. *Kalman and Extended Kalman Filters: Concept Derivation and Properties*. February 2004. <http://users.isr.ist.utl.pt/~mir/pub/kalman.pdf> (accessed April 28, 2011).
- [4] Semiconductor, N. (2010, February 19). *Data Sheet of DAC121S101*. Retrieved from Texas Instruments: <http://www.ti.com/lit/ds/symlink/dac121s101.pdf>
- [5] Devices, A. (n.d.). *Data Sheet of AD7476*. Retrieved from Analog.com: http://www.analog.com/static/imported-files/data_sheets/AD7476_7477_7478.pdf

Appendix A: Top Level Design of Complete System

```
-----
-- Engineer: Dan Thomas & Eduardo Pizzini
-- Create Date:   MQP 2011-2012
-- Design Name:
-- Module Name:   KalmanFilter
-- Description: This is the top level vhdl design which controls all of the other modules within the project.
--               This design includes the Kalman Filter Logic and the DAC and ADC controllers to allow
for the
--               DAC and ADC to work with the FPGA.
```

```
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity KalmanFilter is
    Port ( clk_50M : in  STD_LOGIC;
           start : in std_logic;
           Sdata : in  STD_LOGIC;
           Din : out std_logic;
           Nsync : out std_logic;
           sclk_DAC : out std_logic;
           CS : out  STD_LOGIC;
           sclk_ADC : out std_logic);
```

```
end KalmanFilter;
```

```
architecture Behavioral of KalmanFilter is
```

```
    component ADC is
        Port ( clk : in  STD_LOGIC;
              load : in  STD_LOGIC;
              Sdata : in  STD_LOGIC;
              sclk : out std_logic;
              CS : out  STD_LOGIC;
              ADC_out : out STD_LOGIC_VECTOR(15 downto 0));
    end component;
```

```
    component DAC is
        Port ( clk : in  STD_LOGIC;
              DAC_in : in  STD_LOGIC_VECTOR (11 downto 0);
              load : in  STD_LOGIC;
              Din : out  STD_LOGIC;
```

```

Nsync : out STD_LOGIC;
      sclk : out STD_LOGIC);
end component;

component KFLLogic is
  Port (clk_50M : in std_logic;
        z_in : in STD_LOGIC_VECTOR (15 downto 0);
        x_in1 : in STD_LOGIC_VECTOR (15 downto 0);
x_in2 : in STD_LOGIC_VECTOR (15 downto 0);
x_in3 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in1 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in2 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in3 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in4 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in5 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in6 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in7 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in8 : in STD_LOGIC_VECTOR (15 downto 0);
        P_in9 : in STD_LOGIC_VECTOR (15 downto 0);
x_est1 : out STD_LOGIC_VECTOR (15 downto 0);
x_est2 : out STD_LOGIC_VECTOR (15 downto 0);
x_est3 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out1 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out2 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out3 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out4 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out5 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out6 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out7 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out8 : out STD_LOGIC_VECTOR (15 downto 0);
        P_out9 : out STD_LOGIC_VECTOR (15 downto 0));
end component;

type onebythree is array (1 to 3) of std_logic_vector(15 downto 0);
type threebythree is array (1 to 3, 1 to 3) of std_logic_vector(15 downto 0);
type statetype is (Idle, In_Out, LoadHigh, LoadLow, Hold);
signal z : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
signal x_out : STD_LOGIC_VECTOR (11 downto 0) := "000000000000";
signal loadDAC : STD_LOGIC;
signal loadADC : STD_LOGIC;
signal currentstate, nextstate : statetype;
signal count25K : integer range 0 to 2000 := 0;
signal z_in : std_logic_vector(15 downto 0);
signal x_in1 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
signal x_in2 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
signal x_in3 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
signal x_est1 : STD_LOGIC_VECTOR (15 downto 0);
signal x_est2 : STD_LOGIC_VECTOR (15 downto 0);

```

```

signal x_est3 : STD_LOGIC_VECTOR (15 downto 0);
    signal x_reg : onebythree;
    signal P_reg : threebythree;
    signal P_in1 : STD_LOGIC_VECTOR (15 downto 0) := "0000000101001000";
    signal P_in2 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
    signal P_in3 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
    signal P_in4 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
    signal P_in5 : STD_LOGIC_VECTOR (15 downto 0) := "0000000101001000";
    signal P_in6 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
    signal P_in7 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
    signal P_in8 : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";
    signal P_in9 : STD_LOGIC_VECTOR (15 downto 0) := "0000000101001000";
    signal P_out1 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out2 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out3 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out4 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out5 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out6 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out7 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out8 : STD_LOGIC_VECTOR (15 downto 0);
    signal P_out9 : STD_LOGIC_VECTOR (15 downto 0);

```

begin

```

    ADC1 : ADC port map (clk => clk_50M, ADC_out => z, load => loadADC, Sdata => Sdata, cs => cs,
sclk => sclk_ADC);

```

```

    DAC1 : DAC port map (clk => clk_50M, DAC_in => x_out, load => loadDAC, Din => Din, Nsync =>
Nsync, sclk => sclk_DAC);

```

```

    KFLogic1 : KFLogic port map(clk_50M => clk_50M, z_in => z_in, x_in1 => x_in1, x_in2 => x_in2,
x_in3 => x_in3, x_est1 => x_est1, x_est2 => x_est2, x_est3 => x_est3,
                                                                    P_in1 => P_in1, P_in2
=> P_in2, P_in3 => P_in3, P_in4 => P_in4, P_in5 => P_in5, P_in6 => P_in6,
                                                                    P_in7 => P_in7, P_in8
=> P_in8, P_in9 => P_in9, P_out1 => P_out1, P_out2 => P_out2, P_out3 => P_out3,
                                                                    P_out4 => P_out4,
P_out5 => P_out5, P_out6 => P_out6, P_out7 => P_out7, P_out8 => P_out8,
                                                                    P_out9 => P_out9);

```

```

-- Creates a 25kHz clock
process(clk_50M)
begin
    if rising_edge(clk_50M) then
        if currentstate /= Idle then
            if count25K = 2000 then
                count25K <= 1;
            else

```

```

                                count25K <= count25K + 1;
                                end if;
                            end if;
                        end if;
end process;

-- Update current state on every clk cycle
process(clk_50M)
begin
    if rising_edge(clk_50M) then
        currentstate <= nextstate;
    end if;
end process;

-- Next State Logic
process(currentstate, start, count25K)
begin
    case currentstate is
        when Idle =>
            if start = '1' then
                nextstate <= In_Out;
            else
                nextstate <= Idle;
            end if;
        when In_Out =>
            if count25K = 20 then
                nextstate <= LoadHigh;
            else
                nextstate <= In_Out;
            end if;
        when LoadHigh =>
            if count25K = 22 then
                nextstate <= LoadLow;
            else
                nextstate <= LoadHigh;
            end if;
        when LoadLow =>
            if count25K = 26 then
                nextstate <= Hold;
            else
                nextstate <= LoadLow;
            end if;
        when others =>
            if count25K = 2000 then
                nextstate <= In_Out;
            else
                nextstate <= Hold;
            end if;
    end case;
end process;

```

```

        end case;
    end process;

    -- Controls for Loads
    loadADC <= '1' when currentstate = LoadHigh or currentstate = LoadLow else '0';
    loadDAC <= '1' when currentstate = LoadHigh else '0';

    -- Loading Input and Output Signals
    process(clk_50M)
    begin
        if rising_edge(clk_50M) then
            if nextstate = In_Out and currentstate = Idle then
                z_in <= z;
            elsif nextstate = In_Out and currentstate = Hold then
                z_in <= z;
                x_in1 <= x_reg(1);
                x_in2 <= x_reg(2);
                x_in3 <= x_reg(3);
                P_in1 <= P_reg(1,1);
                P_in2 <= P_reg(1,2);
                P_in3 <= P_reg(1,3);
                P_in4 <= P_reg(2,1);
                P_in5 <= P_reg(2,2);
                P_in6 <= P_reg(2,3);
                P_in7 <= P_reg(3,1);
                P_in8 <= P_reg(3,2);
                P_in9 <= P_reg(3,3);
            end if;
        end if;
    end process;

    --Control for Input to DAC
    process(clk_50M)
    begin
        if rising_edge(clk_50M) then
            if nextstate = In_Out and currentstate = Hold then
                x_out <= x_est1(15 downto 4);
            end if;
        end if;
    end process;

    x_reg(1) <= x_est1;
    x_reg(2) <= x_est2;
    x_reg(3) <= x_est3;
    P_reg(1,1) <= P_out1;
    P_reg(1,2) <= P_out2;
    P_reg(1,3) <= P_out3;
    P_reg(2,1) <= P_out4;

```



```
P_reg(2,2) <= P_out5;  
P_reg(2,3) <= P_out6;  
P_reg(3,1) <= P_out7;  
P_reg(3,2) <= P_out8;  
P_reg(3,3) <= P_out9;
```

```
end Behavioral;
```

Appendix B: VHDL Design of Kalman Filter Logic

```
-----  
-- Engineer: Daniel Thomas and Eduardo Pizzini  
-- Create Date: 21:42:23 11/06/2011  
-- Module Name: SKFLogic  
-- Description: This design computes all of the calculations for the  
--              Kalman Filter to perform its functions  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity KFLogic is
```

```
    Port ( clk_50M : in std_logic;  
           z_in : in STD_LOGIC_VECTOR (15 downto 0);  
           x_in1 : in STD_LOGIC_VECTOR (15 downto 0);  
           x_in2 : in STD_LOGIC_VECTOR (15 downto 0);  
           x_in3 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in1 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in2 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in3 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in4 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in5 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in6 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in7 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in8 : in STD_LOGIC_VECTOR (15 downto 0);  
           P_in9 : in STD_LOGIC_VECTOR (15 downto 0);  
           x_est1 : out STD_LOGIC_VECTOR (15 downto 0);  
           x_est2 : out STD_LOGIC_VECTOR (15 downto 0);  
           x_est3 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out1 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out2 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out3 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out4 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out5 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out6 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out7 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out8 : out STD_LOGIC_VECTOR (15 downto 0);  
           P_out9 : out STD_LOGIC_VECTOR (15 downto 0)));
```

```
end KFLogic;
```

```
architecture Behavioral of KFLogic is
```

```

    component Multiplier1 is
Port ( row1 : in STD_LOGIC_VECTOR (15 downto 0);
      row2 : in STD_LOGIC_VECTOR (15 downto 0);
      row3 : in STD_LOGIC_VECTOR (15 downto 0);
      col11 : in STD_LOGIC_VECTOR (15 downto 0);
      col12 : in STD_LOGIC_VECTOR (15 downto 0);
      col13 : in STD_LOGIC_VECTOR (15 downto 0);
          col21 : in STD_LOGIC_VECTOR (15 downto 0);
      col22 : in STD_LOGIC_VECTOR (15 downto 0);
      col23 : in STD_LOGIC_VECTOR (15 downto 0);
          col31 : in STD_LOGIC_VECTOR (15 downto 0);
      col32 : in STD_LOGIC_VECTOR (15 downto 0);
      col33 : in STD_LOGIC_VECTOR (15 downto 0);
      prod1 : out STD_LOGIC_VECTOR (15 downto 0);
          prod2 : out STD_LOGIC_VECTOR (15 downto 0);
          prod3 : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component Multiplier2 is
Port ( xin1 : in STD_LOGIC_VECTOR (15 downto 0);
      xin2 : in STD_LOGIC_VECTOR (15 downto 0);
          K1 : in STD_LOGIC_VECTOR (15 downto 0);
          K2 : in STD_LOGIC_VECTOR (15 downto 0);
          K3 : in STD_LOGIC_VECTOR (15 downto 0);
          zdiff : in STD_LOGIC_VECTOR (15 downto 0);
      xprod1 : out STD_LOGIC_VECTOR (15 downto 0);
      xprod2 : out STD_LOGIC_VECTOR (15 downto 0);
      xprod3 : out STD_LOGIC_VECTOR (15 downto 0);
      xprod4 : out STD_LOGIC_VECTOR (15 downto 0);
          Kz1 : out STD_LOGIC_VECTOR (15 downto 0);
          Kz2 : out STD_LOGIC_VECTOR (15 downto 0);
          Kz3 : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component divide is
    port (
        rfd : out STD_LOGIC;
        clk : in STD_LOGIC := 'X';
        dividend : in STD_LOGIC_VECTOR ( 15 downto 0 );
        quotient : out STD_LOGIC_VECTOR ( 15 downto 0 );
        divisor : in STD_LOGIC_VECTOR ( 15 downto 0 );
        fractional : out STD_LOGIC_VECTOR ( 15 downto 0 ));
    end component;

    type onebythree is array (1 to 3) of std_logic_vector(15 downto 0);
    type threebythree is array (1 to 3, 1 to 3) of std_logic_vector(15 downto 0);

```

```

constant F : threebythree := (("0111100110111100", "0010011110001110",
"0111111111111111"),

("1101100001110010", "0111100110111100", "0000000000000000"),

("0000000000000000", "0000000000000000", "0111111111111111"));
constant GxQxGT : std_logic_vector(15 downto 0) := "0000000101001000";
constant one : std_logic_vector(15 downto 0) := "0111111111111111";
constant zero : std_logic_vector(15 downto 0) := "0000000000000000";
constant R : std_logic_vector(15 downto 0) := "0010000000000000";
signal x_int : onebythree;
signal x_out : onebythree;
signal z_est : std_logic_vector(15 downto 0);
signal z_diff : std_logic_vector(15 downto 0);
signal P1_Part1 : threebythree;
signal P1_Part2 : threebythree;
signal P1 : threebythree;
signal P2_Part1 : threebythree;
signal P2 : threebythree;
signal S : std_logic_vector(15 downto 0);
signal K : onebythree;
signal K_z : onebythree;

signal xin1 : std_logic_vector(15 downto 0);
signal xin2 : std_logic_vector(15 downto 0);
signal K1 : std_logic_vector(15 downto 0);
signal K2 : std_logic_vector(15 downto 0);
signal K3 : std_logic_vector(15 downto 0);
signal zdiff : std_logic_vector(15 downto 0);
signal Kz1 : std_logic_vector(15 downto 0);
signal Kz2 : std_logic_vector(15 downto 0);
signal Kz3 : std_logic_vector(15 downto 0);
signal xprod1 : std_logic_vector(15 downto 0);
signal xprod2 : std_logic_vector(15 downto 0);
signal xprod3 : std_logic_vector(15 downto 0);
signal xprod4 : std_logic_vector(15 downto 0);
signal x_prod1 : std_logic_vector(15 downto 0);
signal x_prod2 : std_logic_vector(15 downto 0);
signal x_prod3 : std_logic_vector(15 downto 0);
signal x_prod4 : std_logic_vector(15 downto 0);

signal count : integer range 1 to 9 := 1;
signal count2 : integer range 1 to 3 := 1;
signal row1 : STD_LOGIC_VECTOR (15 downto 0);
signal row2 : STD_LOGIC_VECTOR (15 downto 0);
signal row3 : STD_LOGIC_VECTOR (15 downto 0);
signal col11 : STD_LOGIC_VECTOR (15 downto 0);
signal col12 : STD_LOGIC_VECTOR (15 downto 0);

```

```

signal col13 : STD_LOGIC_VECTOR (15 downto 0);
    signal col21 : STD_LOGIC_VECTOR (15 downto 0);
signal col22 : STD_LOGIC_VECTOR (15 downto 0);
signal col23 : STD_LOGIC_VECTOR (15 downto 0);
    signal col31 : STD_LOGIC_VECTOR (15 downto 0);
signal col32 : STD_LOGIC_VECTOR (15 downto 0);
signal col33 : STD_LOGIC_VECTOR (15 downto 0);
signal prod1 : STD_LOGIC_VECTOR (15 downto 0);
    signal prod2 : STD_LOGIC_VECTOR (15 downto 0);
    signal prod3 : STD_LOGIC_VECTOR (15 downto 0);

    signal dividend : STD_LOGIC_VECTOR (15 downto 0);
    signal divisor : STD_LOGIC_VECTOR (15 downto 0);
    signal quotient : STD_LOGIC_VECTOR (15 downto 0);
    signal fractional : STD_LOGIC_VECTOR (15 downto 0);
    signal rfd : std_logic;

begin

    Multiplier11 : Multiplier1 port map(row1 => row1, row2 => row2, row3 => row3,
col11
=> col11, col12 => col12, col13 => col13, prod1 => prod1,
col21
=> col21, col22 => col22, col23 => col23, prod2 => prod2,
col31
=> col31, col32 => col32, col33 => col33, prod3 => prod3);

    Multiplier22 : Multiplier2 port map(xin1 => xin1, xin2 => xin2, xprod1 => xprod1,
xprod2 => xprod2, xprod3 => xprod3, xprod4 => xprod4,

    K1 => K1, K2 => K2, K3 => K3, zdiff => zdiff,

    Kz1 => Kz1, Kz2 => Kz2, Kz3 => Kz3);

    divide1 : divide port map(rfd => rfd, clk => clk_50M, dividend => dividend,
divisor => divisor, quotient =>
quotient, fractional => fractional);

    process(clk_50M)
    begin
        if rising_edge(clk_50M) then
            if count = 9 then
                count <= 1;
            else
                count <= count + 1;
            end if;
        end if;
    end process;

```

```

case count is
  when 1 =>
    row1 <= F(1,1); row2 <= F(1,2); row3 <= F(1,3);
    col11 <= P_in1; col21 <= P_in4; col31 <= P_in7;
    col12 <= P_in2; col22 <= P_in5; col32 <= P_in8;
    col13 <= P_in3; col23 <= P_in6; col33 <= P_in9;
    P2(3,1) <= prod1; P2(3,2) <= prod2; P2(3,3) <= prod3;
  when 2 =>
    row1 <= F(2,1); row2 <= F(2,2); row3 <= F(2,3);
    col11 <= P_in1; col21 <= P_in4; col31 <= P_in7;
    col12 <= P_in2; col22 <= P_in5; col32 <= P_in8;
    col13 <= P_in3; col23 <= P_in6; col33 <= P_in9;
    P1_Part1(1,1) <= prod1; P1_Part1(1,2) <= prod2; P1_Part1(1,3)
<= prod3;
  when 3 =>
    row1 <= F(3,1); row2 <= F(3,2); row3 <= F(3,3);
    col11 <= P_in1; col21 <= P_in4; col31 <= P_in7;
    col12 <= P_in2; col22 <= P_in5; col32 <= P_in8;
    col13 <= P_in3; col23 <= P_in6; col33 <= P_in9;
    P1_Part1(2,1) <= prod1; P1_Part1(2,2) <= prod2; P1_Part1(2,3)
<= prod3;
  when 4 =>
    row1 <= P1_Part1(1,1); row2 <= P1_Part1(1,2); row3 <=
P1_Part1(1,3);

    col11 <= F(1,1); col21 <= F(1,2); col31 <= F(1,3);
    col12 <= F(2,1); col22 <= F(2,2); col32 <= F(2,3);
    col13 <= F(3,1); col23 <= F(3,2); col33 <= F(3,3);
    P1_Part1(3,1) <= prod1; P1_Part1(3,2) <= prod2; P1_Part1(3,3)
<= prod3;
  when 5 =>
    row1 <= P1_Part1(2,1); row2 <= P1_Part1(2,2); row3 <=
P1_Part1(2,3);

    col11 <= F(1,1); col21 <= F(1,2); col31 <= F(1,3);
    col12 <= F(2,1); col22 <= F(2,2); col32 <= F(2,3);
    col13 <= F(3,1); col23 <= F(3,2); col33 <= F(3,3);
    P1_Part2(1,1) <= prod1; P1_Part2(1,2) <= prod2; P1_Part2(1,3)
<= prod3;
  when 6 =>
    row1 <= P1_Part1(3,1); row2 <= P1_Part1(3,2); row3 <=
P1_Part1(3,3);

    col11 <= F(1,1); col21 <= F(1,2); col31 <= F(1,3);
    col12 <= F(2,1); col22 <= F(2,2); col32 <= F(2,3);
    col13 <= F(3,1); col23 <= F(3,2); col33 <= F(3,3);
    P1_Part2(2,1) <= prod1; P1_Part2(2,2) <= prod2; P1_Part2(2,3)
<= prod3;
  when 7 =>

```

```

        row1 <= P2_Part1(1,1); row2 <= P2_Part1(1,2); row3 <=
P2_Part1(1,3);
        col11 <= P1(1,1); col21 <= P1(2,1); col31 <= P1(3,1);
        col12 <= P1(1,2); col22 <= P1(2,2); col32 <= P1(3,2);
        col13 <= P1(1,3); col23 <= P1(2,3); col33 <= P1(3,3);
        P1_Part2(3,1) <= prod1; P1_Part2(3,2) <= prod2; P1_Part2(3,3)
<= prod3;
        when 8 =>
            row1 <= P2_Part1(2,1); row2 <= P2_Part1(2,2); row3 <=
P2_Part1(2,3);
            col11 <= P1(1,1); col21 <= P1(2,1); col31 <= P1(3,1);
            col12 <= P1(1,2); col22 <= P1(2,2); col32 <= P1(3,2);
            col13 <= P1(1,3); col23 <= P1(2,3); col33 <= P1(3,3);
            P2(1,1) <= prod1; P2(1,2) <= prod2; P2(1,3) <= prod3;
        when 9 =>
            row1 <= P2_Part1(3,1); row2 <= P2_Part1(3,2); row3 <=
P2_Part1(3,3);
            col11 <= P1(1,1); col21 <= P1(2,1); col31 <= P1(3,1);
            col12 <= P1(1,2); col22 <= P1(2,2); col32 <= P1(3,2);
            col13 <= P1(1,3); col23 <= P1(2,3); col33 <= P1(3,3);
            P2(2,1) <= prod1; P2(2,2) <= prod2; P2(2,3) <= prod3;
        end case;
    end if;
end process;

process(clk_50M)
begin
    if rising_edge(clk_50M) then
        if count2 = 3 then
            count2 <= 1;
        else
            count2 <= count2 + 1;
        end if;

        case count2 is
            when 1 =>
                dividend <= P1(1,1); divisor <= S; K(3) <= fractional;
            when 2 =>
                dividend <= P1(2,1); divisor <= S; K(1) <= fractional;
            when 3 =>
                dividend <= P1(3,1); divisor <= S; K(2) <= fractional;
        end case;
    end if;
end process;

xin1 <= x_in1;
xin2 <= x_in2;
K1 <= K(1);

```

```

K2 <= K(2);
K3 <= K(3);
zdiff <= z_diff;

process(clk_50M)
begin
    if rising_edge(clk_50M) then
        x_prod1 <= xprod1;
        x_prod2 <= xprod2;
        x_prod3 <= xprod3;
        x_prod4 <= xprod4;
        K_z(1) <= Kz1;
        K_z(2) <= Kz2;
        K_z(3) <= Kz3;
    end if;
end process;

x_int(1) <= x_prod1 + x_prod2 + x_in3;
x_int(2) <= x_prod3 + x_prod4;
x_int(3) <= x_in3;

z_est <= x_int(1);

process(clk_50M)
begin
    if rising_edge(clk_50M) then
        P1(1,1) <= P1_Part2(1,1);
        P1(1,2) <= P1_Part2(1,2);
        P1(1,3) <= P1_Part2(1,3);
        P1(2,1) <= P1_Part2(2,1);
        P1(2,2) <= P1_Part2(2,2);
        P1(2,3) <= P1_Part2(2,3);
        P1(3,1) <= P1_Part2(3,1);
        P1(3,2) <= P1_Part2(3,2);
        P1(3,3) <= P1_Part2(3,3) + GxQxGT;
    end if;
end process;

S <= P1(1,1) + R;
z_diff <= z_in - z_est;

x_out(1) <= x_int(1) + K_z(1);
x_out(2) <= x_int(2) + K_z(2);
x_out(3) <= x_int(3) + K_z(3);

P2_Part1(1,1) <= one - K(1);
P2_Part1(1,2) <= zero;
P2_Part1(1,3) <= zero;

```



```
P2_Part1(2,1) <= zero - K(2);  
P2_Part1(2,2) <= one;  
P2_Part1(2,3) <= zero;  
P2_Part1(3,1) <= zero - K(3);  
P2_Part1(3,2) <= zero;  
P2_Part1(3,3) <= one;
```

```
P_out1 <= P2(1,1);  
P_out2 <= P2(1,2);  
P_out3 <= P2(1,3);  
P_out4 <= P2(2,1);  
P_out5 <= P2(2,2);  
P_out6 <= P2(2,3);  
P_out7 <= P2(3,1);  
P_out8 <= P2(3,2);  
P_out9 <= P2(3,3);
```

```
x_est1 <= x_out(1);  
x_est2 <= x_out(2);  
x_est3 <= x_out(3);
```

```
end Behavioral;
```

Appendix C: VHDL Code for ADC and DAC Controllers

```
-----  
-- Engineer: Daniel Thomas and Eduardo Pizzini  
-- Create Date: 16:48:53 02/15/2012  
-- Module Name: ADC - Behavioral  
-- Description: This is the ADC controller which allows the Kalman Filter design  
--              within the FPGA to work with the ADC component  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity ADC is  
    Port ( clk : in STD_LOGIC;  
          load : in STD_LOGIC;  
          Sdata : in STD_LOGIC;  
          sclk : out std_logic;  
          CS : out STD_LOGIC;  
          ADC_out : out STD_LOGIC_VECTOR(15 downto 0));  
end ADC;
```

```
architecture Behavioral of ADC is
```

```
    type state_type is (Idle, CSLow, Receive, CSHigh);  
    signal temp : STD_LOGIC_VECTOR (15 downto 0) := "0000000000000000";  
    signal data : STD_LOGIC_VECTOR (11 downto 0);  
    signal count : integer range 0 to 5 := 0;  
    signal regcount : STD_LOGIC_VECTOR (3 downto 0) := "0000";  
    signal currentstate, nextstate : state_type;  
    signal tsclk : std_logic := '0';
```

```
begin
```

```
    -- Creates a 8.3 MHz clock  
    process (clk)  
    begin  
        if rising_edge(clk) then  
            if count = 5 then  
                count <= 0;  
            else  
                count <= count + 1;  
            end if  
        end if  
    end process
```

```

        end if;
    end if;
end process;

-- Logic for sclk
sclk <= tsclk;
process(clk)
begin
    if rising_edge(clk) then
        if count = 0 then
            tsclk <= '1';
        elsif count = 3 then
            tsclk <= '0';
        end if;
    end if;
end process;

-- Sets current State on rising edge of sclk
process(clk)
begin
    if rising_edge(clk) then
        if count = 0 then
            currentstate <= nextstate;
        end if;
    end if;
end process;

-- Next State Logic
process(currentstate, load, regcount)
begin
    case currentstate is
        when Idle =>
            if load = '1' then
                nextstate <= CSLow;
            else
                nextstate <= Idle;
            end if;
        when CSLow =>
            nextstate <= Receive;
        when Receive =>
            if regcount = 15 then
                nextstate <= CSHigh;
            else
                nextstate <= Receive;
            end if;
        when CSHigh =>
            nextstate <= Idle;
    end case;
end process;

```

```

end process;

-- CS Logic
CS <= '0' when currentstate = CSLow or currentstate = Receive else '1';

-- Control for regcount
process(clk)
begin
    if rising_edge(clk) then
        if count = 0 then
            if currentstate = CSLow then
                regcount <= "0001";
            elsif currentstate = Receive then
                regcount <= regcount + 1;
            end if;
        end if;
    end if;
end process;

-- Control for temp
process(clk)
begin
    if rising_edge(clk) then
        if nextstate = Receive then
            if count = 0 then
                temp <= temp(14 downto 0) & Sdata;
            elsif count = 3 then
                temp(0) <= Sdata;
            end if;
        end if;
    end if;
end process;

-- Load 12 data bits to data
process(clk)
begin
    if rising_edge(clk) then
        if currentstate = CSHigh then
            data <= temp(11 downto 0);
        end if;
    end if;
end process;

-- data takes the 12 data bits from temp
-- and adds 4 '0's as LSBs to form ADC_out
ADC_out <= not(data(11)) & data(10 downto 0) & "0000";

```

end Behavioral;

```

-----
-- Engineer: Eduardo Pizzini and Daniel Thomas
-- Module Name: DAC - Behavioral
-- Description: This is the DAC Controller which allows the Kalman Filter Design
--              within the FPGA to work with the DAC component
-----

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

entity DAC is

```

    Port ( clk : in  STD_LOGIC;
          DAC_in : in  STD_LOGIC_VECTOR (11 downto 0);
          load : in  STD_LOGIC;
          Din : out STD_LOGIC;
          sclk : out std_logic;
          Nsync : out STD_LOGIC);
end DAC;

```

architecture Behavioral of DAC is

```

    type statetype is (Idle, Low, Send, High);
    signal temp : STD_LOGIC_VECTOR (15 downto 0);
    signal data : STD_LOGIC_VECTOR (15 downto 0);
    signal count : integer range 0 to 1 := 0;
    signal regcount : STD_LOGIC_VECTOR (3 downto 0);
    signal currentstate, nextstate : statetype;
    signal tsclk : std_logic := '0';

begin

    -- Sets sclk for the DAC
    sclk <= tsclk;

    -- data takes the 12 data bits of DAC_in
    -- and adds 4 control to use with the DAC
    data <= "0000" & not(DAC_in(11)) & DAC_in(10 downto 0);

    -- creates 25MHz clk
    process (clk)
    begin
        if rising_edge(clk) then
            if count = 1 then -- falling edge of sclk
                count <= 0;
            end if;
        end if;
    end process;

```

```

        tsclk <= '0';
    else -- rising edge of sclk
        count <= count + 1;
        tsclk <= '1';
    end if;
end if;
end process;

-- Control for temp
process(clk)
begin
    if rising_edge(clk) then
        if count = 0 then
            if load = '1' then
                temp <= data;
            elsif currentstate = Send then
                temp <= temp(14 downto 0) & '0';
            end if;
        end if;
    end if;
end process;

-- Sets current state on every rising edge of sclk
process(clk)
begin
    if rising_edge(clk) then
        if count = 0 then
            currentstate <= nextstate;
        end if;
    end if;
end process;

-- Next State Logic
process(currentstate, load, regcount)
begin
    case currentstate is
        when Idle =>
            if load = '1' then
                nextstate <= Low;
            else
                nextstate <= Idle;
            end if;
        when Low =>
            nextstate <= Send;
        when Send =>
            if regcount = 15 then
                nextstate <= High;
            end if;
        end case;
    end process;

```

```

        else
            nextstate <= Send;
        end if;
    when High =>
        nextstate <= Idle;
    end case;
end process;

-- Output Logic
Din <= temp(15) when currentstate = Send else '0';
Nsync <= '0' when (currentstate = Low and count = 0)
            or currentstate = Send else '1';

-- Control for regcount
process(clk)
begin
    if rising_edge(clk) then
        if count = 0 then
            if currentstate = Low then
                regcount <= "0000";
            elsif currentstate = Send then
                regcount <= regcount + 1;
            end if;
        end if;
    end if;
end process;

end Behavioral;

```

Appendix D: VHDL Code for Multiplier Modules used for Calculations

```
-- This package allows for the multiplication of 2 signed 16-bit values
-- and returns a signed 16-bit product

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

package Mult_Funct is

    function multiply (operand1 : in std_logic_vector(15 downto 0);
                      operand2 : in std_logic_vector(15 downto 0))
        return std_logic_vector;

end Mult_Funct;

package body Mult_Funct is

    function multiply (operand1 : in std_logic_vector(15 downto 0);
                      operand2 : in std_logic_vector(15 downto 0))
        return std_logic_vector is
        variable oprnd1 : std_logic_vector(15 downto 0);
        variable oprnd2 : std_logic_vector(15 downto 0);
        variable prod32 : std_logic_vector(31 downto 0);
        variable prod : std_logic_vector(15 downto 0);
        variable sign : integer range 0 to 1;

    begin

        if operand1(15) = '1' then
            oprnd1 := "1111111111111111" - operand1 + "0000000000000001";
            sign := 1;
        else
            oprnd1 := operand1;
            sign := 0;
        end if;

        if operand2(15) = '1' then
            oprnd2 := "1111111111111111" - operand2 + "0000000000000001";
            if sign = 1 then
                sign := 0;
            else
                sign := 1;
            end if;
        else
            sign := 0;
        end if;

        prod32 := oprnd1 * oprnd2;

        if sign = 1 then
            prod := prod32(16 to 31);
        else
            prod := prod32(0 to 15);
        end if;

    end multiply;

end package body Mult_Funct;
```



```

        oprnd2 := operand2;
        if sign = 1 then
            sign := 1;
        else
            sign := 0;
        end if;
    end if;

    if sign = 1 then
        prod32 := "11111111111111111111111111111111" - (oprnd1 * oprnd2) +
            "00000000000000000000000000000001";
    else
        prod32 := oprnd1 * oprnd2;
    end if;
    prod := prod32(30 downto 15);

    return prod;

end multiply;

end Mult_Funct;

```

```

-----
-- Engineer: Eduardo Pizzini and Daniel Thomas
-- Create Date: 11:46:42 11/26/2011
-- Module Name: Multiplier1 - Behavioral
-- Description: This multiplier performs a matrix multiplication on a row of 3
--              add a 3x3 matrix to produce a product of a row of 3
-----

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.Mult_Funct.ALL;

```

```

entity Multiplier1 is
    Port ( row1 : in STD_LOGIC_VECTOR (15 downto 0);
          row2 : in STD_LOGIC_VECTOR (15 downto 0);
          row3 : in STD_LOGIC_VECTOR (15 downto 0);
          col11 : in STD_LOGIC_VECTOR (15 downto 0);
          col12 : in STD_LOGIC_VECTOR (15 downto 0);
          col13 : in STD_LOGIC_VECTOR (15 downto 0);
          col21 : in STD_LOGIC_VECTOR (15 downto 0);
          col22 : in STD_LOGIC_VECTOR (15 downto 0);
          col23 : in STD_LOGIC_VECTOR (15 downto 0);
          col31 : in STD_LOGIC_VECTOR (15 downto 0);

```

```

col32 : in STD_LOGIC_VECTOR (15 downto 0);
col33 : in STD_LOGIC_VECTOR (15 downto 0);
prod1 : out STD_LOGIC_VECTOR (15 downto 0);
        prod2 : out STD_LOGIC_VECTOR (15 downto 0);
        prod3 : out STD_LOGIC_VECTOR (15 downto 0));
end Multiplier1;

architecture Behavioral of Multiplier1 is

begin

    prod1 <= multiply(row1,col11) + multiply(row2,col21) + multiply(row3,col31);
    prod2 <= multiply(row1,col12) + multiply(row2,col22) + multiply(row3,col32);
    prod3 <= multiply(row1,col13) + multiply(row2,col23) + multiply(row3,col33);

end Behavioral;

```

```

-----
-- Engineer: Eduardo Pizzini and Daniel Thomas
-- Create Date: 15:37:11 11/27/2011
-- Module Name: Multiplier4 - Behavioral
-- Description: This multiplier module takes in 5 operands and uses them to produce
--              7 products
-----

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use work.Mult_Funct.ALL;

entity Multiplier2 is
    Port ( xin1 : in STD_LOGIC_VECTOR (15 downto 0);
          xin2 : in STD_LOGIC_VECTOR (15 downto 0);
              K1 : in STD_LOGIC_VECTOR (15 downto 0);
              K2 : in STD_LOGIC_VECTOR (15 downto 0);
              K3 : in STD_LOGIC_VECTOR (15 downto 0);
              zdiff : in STD_LOGIC_VECTOR (15 downto 0);
          xprod1 : out STD_LOGIC_VECTOR (15 downto 0);
          xprod2 : out STD_LOGIC_VECTOR (15 downto 0);
          xprod3 : out STD_LOGIC_VECTOR (15 downto 0);
          xprod4 : out STD_LOGIC_VECTOR (15 downto 0);
              Kz1 : out STD_LOGIC_VECTOR (15 downto 0);
              Kz2 : out STD_LOGIC_VECTOR (15 downto 0);
              Kz3 : out STD_LOGIC_VECTOR (15 downto 0));
end Multiplier2;

```

```

architecture Behavioral of Multiplier2 is

```

```

constant ccos : std_logic_vector(15 downto 0) := "0111100110111100";
constant csin : std_logic_vector(15 downto 0) := "0010011110001101";
constant ncsin : std_logic_vector(15 downto 0) := "1101100001110011";

begin

    xprod1 <= multiply(ccos,xin1);
    xprod2 <= multiply(csin,xin2);
    xprod3 <= multiply(ncsin,xin1);
    xprod4 <= multiply(ccos,xin2);
    Kz1 <= multiply(K1,zdiff);
    Kz2 <= multiply(K2,zdiff);
    Kz3 <= multiply(K3,zdiff);

end Behavioral;

```

Appendix E: UCF File for the Complete Kalman Filter System

#User Constraints for KalmanFilter

```
net "clk_50M" loc = "B8";  
net "clk_50M" period = 20 ns high 50%;
```

```
net "start" loc = "H13";
```

```
NET "CS" LOC = "L15";  
NET "Sdata" LOC = "K12";  
NET "sclk_ADC" LOC = "M15";
```

```
NET "Nsync" LOC = "M13";  
NET "Din" LOC = "R18";  
NET "sclk_DAC" LOC = "T17";
```