# On Kalman Filter Implementation on FPGAs

by

Zorawar Bhatia
B.Eng., University of Victoria, 2008

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

© Zorawar Bhatia, 2012
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

# Supervisory Committee

On Kalman Filter Implementation on FPGAs

by

Zorawar Bhatia
B.Eng., University of Victoria, 2008

**Supervisory Committee**

Dr. Mihai Sima, (Department of Electrical and Computer Engineering)
**Co-Supervisor**

Dr. Michael McGuire, (Department of Electrical and Computer Engineering)
**Co-Supervisor**

# Abstract

**Supervisory Committee**
Dr. Mihai Sima, (Department of Electrical and Computer Engineering)
**Co-Supervisor**
Dr. Michael McGuire, (Department of Electrical and Computer Engineering
**Co-Supervisor**

The following dissertation attempts to highlight and address the implementation and performance of a Kalman filter on an FPGA. The reasons for choosing the Kalman filter and the platform for implementation are highlighted as well as an in depth explanation of the components and theory behind both are given.

A controller system which allows the optimal performance of the Kalman filter on it is developed in VHDL. The design of the controller is dictated by the analysis of the Kalman filter which ensures only the most necessary components and operations are built into the instruction set. The controller is made up of several components including the loader, the ALU, Data RAM, KF IO, Control Store and the Branch Unit. The components working in conjunction allows the system to interface though a handshaking protocol with a peripheral of arbitrary latency. The control store is loaded with program code that is determined by converting human readable assembler into machine code through a Perl encoder. The controller system is tested and verified though an extensive testbench environment that emulates all outside signals and views internal operations. The controller system is capable of five matrix operations which are computed in parallel due to the FPGA development environment, which is far superior in this case to the alternative: a software solution, due to the vector operations inherent in the Kalman filter algorithm.

The Kalman filter operation is analyzed and simulated in a MATLAB environment and this analysis confirms the need for the parallel processing power of the FPGA system upon which the controller has been built. FPGA statistical analysis confirms the successful implementation of the system meeting all criteria set at the outset of the project, including memory usage, IO usage and performance and accuracy benchmarks.

# Table of Contents

Supervisory Committee....ii
Abstract....iii

Table of Contents....iv
List of Tables....vi
List of Figures....vii

Acknowledgements....viii
Dedication....ix

Chapter 1: Introduction....1
- 1.1 Kalman Filter....1
- 1.2 Implementation Options....3
- 1.3 Target Specifications....6
- 1.4 Work Completed....8
- 1.5 Report Organization....9

Chapter 2: Kalman Filter Controller....11
- 2.1 Controller Elements....13
  - Loader....13
  - Control Store....17
  - Branch Unit....18
  - KF_IO Unit....19
  - Data Store ....23
  - KF ALU....24
- 2.2 Diagrams....28
- 2.3 Instruction Set....34
  - Instruction format ....34
  - Execution....35
  - Operation Cycles....36
  - Input/Output....38
  - Assembler: Perl Script for Program Code....39
- 2.4 Implementation....42
  - Compiling....42
  - Debugging....42
  - Number Representation....44

Chapter 3: Kalman Filter....47
- 3.1 Intro....47
  - Adaptive vs. Non-adaptive....48
- 3.2 Theory....49
  - Stochastic Estimation....49

Observer Design Problem....50
Kalman Filter....51
The Kalman Filter Equations....53
3.3 Kalman Filter Implementation....56
Stages of Kalman filter....56
Computations Required Per Step....59
Design Considerations....61
MATLAB Routine for Testing Kalman Filter Operation ....62
3.4 HDL Implementation....67
Designing the system....67
Coding....68
Problems encountered....69
Debugging....72
MATLAB Simulation....73
3.5 Using the system....75
Degrees of freedom....75
Programming process....75
Performance....76

Bibliography....77

Appendix A: MATLAB Code....80
Appendix B: Kalman Filter Code....83
Appendix C: FPGA Project Statistics....88

# List of Tables

Table 1: Assembly Matrix Initialization Example....................17
Table 2: ALU Operations....................25
Table 3: Assembly Read Example....................39
Table 4: Assembly Write Example....................40
Table 5: Assembly Arithmetic Operation Example....................40
Table 6: Assembly Jump Example....................40
Table 7: Computations Required for Each Step in KF....................61

# List of Figures

Figure 1: FPGA vs ASIC Development [11]....5
Figure 2: Controller Block Diagram....11
Figure 3: System Initialization....15
Figure 4: KF_IO Signals....19
Figure 5: Handshaking Signals....20
Figure 6: Timing Diagram, Read....21
Figure 7: Timing Diagram, Write....22
Figure 8: Matrix Multiplication Example....26
Figure 9: Controller Top Level....28
Figure 10: Controller, One Level In....29
Figure 11: Control Store, Top Level....30
Figure 12: Control Store, One Level In....31
Figure 13: KF IO Unit....32
Figure 14: Loader Unit....32
Figure 15: Branch Unit....33
Figure 16: Instruction Format....35
Figure 17: Cycles Per Instruction....37
Figure 18: Feedback Cycle of the Kalman Filter....54
Figure 19: Complete Kalman Filter Equations Diagram....55
Figure 20: Discrete Time Linear System [16]....56
Figure 21: Discrete-Time Linear System -- Kalman Predictor [16]....59
Figure 22: KF Example, Measurement....63
Figure 23: KF Example, Covariance....64
Figure 24: KF Example, Kalman Gain, K....65
Figure 25: KF Example, R = .0001....66
Figure 26: Finding System Changes Using Highlighted Variables....71
Figure 27: Multiply Module Waveforms....73

# Acknowledgements

I would like to take this opportunity to express my gratitude to my supervisors, Dr. Mihai Sima, and Dr. Michael McGuire. Without their support, this thesis would not have been possible. I often went to Dr. Sima with questions and problems and was always given excellent guidance. Dr. McGuire was also always available and had great insights on this or related lines of enquiry. I greatly enjoyed accompanying Dr. Sima to the conferences related to this project and having his help in preparing the material that we successfully presented.

I would also like to acknowledge a fellow graduate student of the same supervisors, Scott Miller, who was very helpful during the early stages of this degree, providing advice that proved to be invaluable throughout my time in our lab.

Thank you to my other fellow graduate students. I enjoyed our time together, be it playing Foosball during our breaks, or during the long hours working in the office and lab. I am honoured to have served as your Graduate Student Adviser, during my terms on campus at the University of Victoria.

Finally, thank you to my family. I appreciate you all.

# Dedication

Dedicated to my Father and Mother.

Thank you for everything.

# Chapter 1: Introduction

The following dissertation attempts to highlight and address the implementation and performance of a Kalman filter on an FPGA. The reasons for choosing the Kalman filter and the platform for implementation are highlighted as well as an in depth explanation of the components and theory behind both are given. In this way the goal of the following report is to provide the reader with the background and ability to utilize the tools developed to implement a system that is specific to his or her application. The system has been implemented in as flexible a way as possible, so as to allow an easy extension to a wider word-length and further dimensions. A brief introduction to the Kalman filter follows.

## 1.1 Kalman Filter

The following section serves as an introduction to the Kalman filter, our method, and the attractiveness of FPGAs as implementation platforms.

The Kalman filter was co-invented by Rudolf (Rudy) Emil Kálmán, a Hungarian-American electrical engineer, mathematical system theorist, and college professor. During the 1950s, while employed as a researcher at the Research Institute for Advanced Study in New York, he developed what would be his most well known work, the so called "Kalman Filter" [1]. Kalman's blend of earlier work in filtering by Wiener, Kolmogorov, Bode, Shannon, Pugachev and others with the modern state-space has become perhaps the most widely applied by-product of modern control theory today. It's popularity is partly due to the fact that the digital computer is used in both the design phase, as well as the implementation phase, and it brings together concepts in filtering and control, and the duality between these problems [2].

Due to the strengths of the Kalman filter, many of which will be explored below, it has found a wide area of application, in everything from space vehicle navigation and control, (eg. the Apollo vehicle) to socioeconomic systems. Our contribution has been to bring the Kalman filter to an implementation system in which it can be used to its highest capabilities, the FPGA development environment, and at the same time take advantage of the FPGA's great potential for widely used and economical application [3].

The Kalman filter is a powerful and widely applicable system, however it does have some requirements which can make it difficult to implement. It is a recursive filter, which means that any system implementing it requires the inputs to wait for the outputs, which can take a lot of computing time on a traditional sequential computing system. Also, it is not unusual for there to be a large block of inputs to the system. The system is able to model the internal state of large and complex system, therefore one expects there to be a large number of variables involved in the processing. This process may be very slow on a traditional computing system.

By utilizing the FPGA environment, the inherent parallel computation ability of the Kalman filter is exposed and exploited for maximum computing performance. The way to leverage this ability is to take advantage of the many computing resources available on an FPGA. FPGAs come with blocks of hardware resources such as DSP units, embedded memories, complex clocking structures, etc, which can all be used to further the main overall goal: to make the system as fast and seamless as possible. Plus, by utilizing an FPGA system, there is no need to send the hardware description to a foundry to have to produced in silicon. The FPGA can be configured by the end user as needed.

The applications for a Kalman filter system are many. The number of those applications that would benefit from a fast or real time system response are the majority, if not all of them. Our contribution here brings us closer to that objective. The way to that goal is to make use of the available hardware on the FPGA system; since we have it, we use it to the fullest. This is the reason we use 9 RAMs for data. Since the data is in the form of a 3x3 matrix, one option would have been to use 3 RAMs: one for each column of data, however, since the FPGA chosen has ample RAMs and other resources, the data was spread out as thin as possible, making the parallelism as wide as possible. This results in the greatest speed-up. We also use one RAM to store the code, and many of the DSP units to compute the results of computations. By using as much of the resources as possible at the same time, the inherent parallelism of the Kalman filter is used to the greatest effect. For example in a matrix multiplication, the operation is completed in parallel by many DSP units, as opposed to a step by step operation by one unit.

Algorithmically speaking, the Kalman filter, or linear quadratic estimation, is a recursive solution to the discrete-data linear filtering problem. It uses a series of noisy measurement inputs observed over time to estimate unknown state variables of the underlying system. The estimates are the statistically optimal solution, and tend to be more precise than an estimate based on a single measurement alone.

The question that the Kalman filter attempts to answer is: based on a set of observations and an initial state, what is the state at time N? For example, if we have a guidance system, we have a set of observations, (the location/speed/etc estimates from the sensors) and an initial state (the location/speed when we began) and we take a decision (act on the environment based on the estimation of the state) that we can correct later as new observations become available. Through the Kalman filter, as we continue to estimate the state based on past estimated states, and current observations, the current estimate becomes clearer and clearer by using the internal model/initial state, and the noisy current observations, than it would have been by using the model or the observations alone. The estimates are done through the use of a Linear Minimum Mean Square Error (LMMSE) estimator, and are therefore statistically optimal.

Regardless of whether the observations are noisy or the internal model is flawed, the use of both of them in conjunction allows for one system to weigh the model more heavily, if it computes that it is more statistically accurate, and for another system to weigh the observations more heavily, if necessary. As the system operates, the filter gain, k, which signifies whether the model or the observations are weighed more, changes. After many

iterations, the filter gain will approach a constant value, which is the most optimal place to estimate values for that current situation.

Applications for the Kalman filter are numerous. In most cases where you have a noisy set of data, the Kalman filter can be used to improve the resulting output. For example the Kalman filter has found use in the following fields:

- Tracking objects. One of the most popular uses of the Kalman filter is for guidance, tracking [4] and control of vehicles, including aircraft and spacecraft [5], however the object could be as ordinary as a hand or face in a motion detection type of situation.
- Data smoothing and curve fitting. Fitting bezier patches to point data. Data Fusion/Integration
- Robotics [6]. In order to model the robot position, measurements from many different sources need to be combined, (vision measurements, beacon data, internal motors, detectors, accelerometers, etc) which the Kalman is able to do. Also, noisy data can make it appear as if the robot is "jumping" from one sample point to the next. The Kalman filter is able to recognize the noisy measurements and smooth out the state variable changes by utilizing the less noisy sources. It is also able to provide an estimate of the state variable vector uncertainty, thus giving the system an idea of how confident the estimate is.
- Econometrics: the application of mathematics and statistical methods to economic data. The ability of the Kalman filter to predict future measurements has been invaluable in many economic applications [7].
- Many others, including computer vision [8], video processing, feature extraction, virtual reality [9], etc.

All of the above varying applications and the specific arenas that they would be used in intuitively shows that there are a number of options available for implementation. The next section highlights our analysis on the options available to us, and our reasons for ultimately choosing the platform that we did.

## 1.2 Implementation Options

When considering a complex algorithm such as the one in question, there are always two avenues that one may consider, implementing it in: software or hardware.

The first option is to have an all software solution. The benefits to this method is that it can be a simple and quick option, and there are many different architectures and powerful programming languages available that are capable of implementing this algorithm. However, the disadvantage is that software is sequential. The instruction set may not be optimum for the considered application.

For example, in any normal application, where all the variables are scalars, the software solution is fast and efficient. In this case however, all of the variables are vectors or matrices, which instantly increases the amount of computing that is required. Even for a

small matrix such as a 3x3 one, the computations jump at least nine-fold, from one, say, addition, to nine.

For a more complex operation such as multiplication or finding the determinant, there are much more than nine operations required for a 3x3 matrix. Therefore, in each step of say, addition, the software processor would be required to traverse through a long loop. In the hardware case, assuming that the resources are available, a single variable addition would take comparably the same amount of time as computing an addition for a large matrix, due to the inherent parallel processing ability of hardware circuitry [10].

Software inherently provides a sequential implementation. The Kalman filter, as discussed above, has inherent parallelism potential, in the form of matrix operations that can be computed on each individual cell in the matrix independently, or close to independently, without regard to the other cells. Software is unable to utilize this parallelism to any benefit. This is one advantage of a hardware solution.

However, software does also have its advantages. Software is flexible. To update the system or to fix a bug requires minimal investment. The code is available and easily modifiable. Hardware, once fabricated, is "set in stone," so to speak. However, when considering the specific platform of the FPGA, which is discussed below, the case for software flexibility is essentially nullified as it is shown that some hardware can in fact be flexible as well as fast.

For the above performance reasons, it was decided that the greatest benefit would come from a hardware solution to implement the predominately matrix operational heavy Kalman filter algorithm. Other reasons to develop on the specific system chosen are outlined below.

### Platform for Implementation

Considering the above argument for a hardware solution, the platform for implementation is decided as follows. There are three main options in this regard:

1. An Application Specific Integrated Circuit, (ASIC).
2. Use a popular multiple core system to take advantage of the parallel processing power and high speed.
3. An FPGA system.

The first option, an ASIC system, is a highly customizable solution, and offers the ability to define the packaging and processing of the system in a very specific way. However, this advantage is also its disadvantage, as once the system is built, it cannot be changed. Any updates or bugs to the system would require a complete redesign and re-fabrication of the entire system [11].

The redesign and re-fabrication process is a long and time-consuming one. In this system, after one successful design, if it was successfully implemented, the design team would be

reluctant to change or add any functionality to the system, let alone change it completely for a new application. This means that the scope of the system will be very limited. We decided right away that due to the wide area application that the Kalman filter has, the final system had to be as flexible as possible.

As well, an ASIC system can be an expensive endeavour, as it requires many steps to finalize and manufacture a finished product, as shown in the following figure, Figure 1. On top of this is the cost of any re-manufacturing that may be required due to changing requirements or re-design. The above reasons take the ASIC system out of consideration for final implementation.

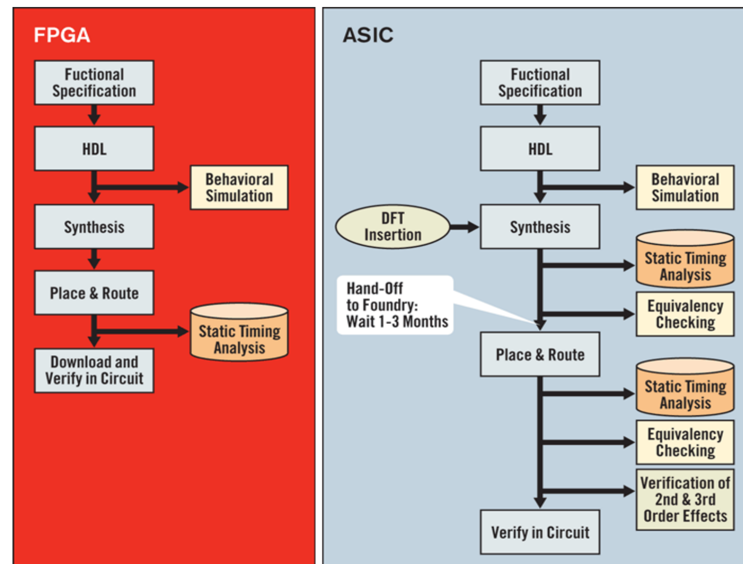


**Figure 1: FPGA vs ASIC Development [11]**

The second option is to use a multi-core system such as a Pentium or a DSP processor. These systems are widely used, and this option makes the software case again a viable option. Due to the parallel processing ability of a multi-core system, the weakness of a software solution is alleviated and it's strengths made more attractive. An implementation on a multi-core system would be fast and flexible.

However, disadvantages again take the software solution out of consideration. The multi-core system is an expensive solution. It requires at the very least a fulling functioning multi-core processing system, complete with the accompanying RAM, hard drive, input/ outputs. This would require some sort of tablet or laptop computer to be connected to the measuring devices. Not very practical for embedded applications, when the system must fit under the hood of a car, or in the black-box of an aeroplane.

This brings us to the last option, an FPGA. Luckily, not only does the FPGA win by default, but it is in actuality a great solution to the problem. The FPGA offers the flexibility of a software solution combined with the speed and processing power of a hardware system. Since the FPGA is “field programmable,” it can be wiped clean and re-

programmed, should the requirements of the system change. Since it is predominately a hardware system, optimization of the computing resources to application, and parallel processing are possible. In this way the benefits of both a software and hardware solution are realized. The system is flexible like a software system, yet fast parallel processing is possible like in a pure hardware system.

Also, on an FPGA, primitive coarse grained computational blocks are available, which means that very basic operations, such as Multiply and Accumulates, or Look Up Tables, do not need to be designed, allowing for faster ramp up and easier modifications. Certain FPGAs come with more or less of these systems and the correct FPGA for the task at hand is chosen as a result of the preliminary design process. For example [12] makes use of the Altera FLEX 8000 family of FPGAs which lack coarse-grained units, and all of the computation was implemented within the fine-grained fabric, which can be an inefficient method of design.

The fine grained fabric of the FPGA cannot support complex DSP operations and an FPGA with existing DSP units are available and must be chosen. Then, these DSP units must be activated and used by programming the FPGA, (through a hardware description language such as VHDL) in a specific way. We will outline in the following chapters the methods and results of such specific choices, design and programming.

Lastly, as FPGAs become more and more popular, the price of an FPGA system is dropping constantly. Not only is the initial investment much lower than the other two options presented, but the cost of re-design is only the cost of the time it takes to change the system [13].

With the above advantages in mind, we have some idea of the type of performance we expect to see. We further develop those expectations into specific target specifications in the next section so as to have a benchmark, or goals to reach for, and to help determine the success of our system.

## 1.3 Target Specifications

Once the implementation platform has been established, certain criteria for performance benchmarking must be defined. The target specifications to strive for, and the reasons for them are presented below.

### Word Length

In any system, the first criteria to consider is the word length. A word length too large, and the system becomes unnecessarily complex and can slow down. If the word length is too small, the system may not be able to meet accuracy requirements, and may require more than one cycle for tasks that can and should be completed in one. For example, if the word length can only hold a number that is a certain size, the program would have to somehow compress the larger numbers, or the numerical system would have very coarse quantization.

As a proof of concept, the system should have a word length of eight bits. This is a standard size, and for good reason. It is a power of two, and many systems support and recognize this word length. The system was designed with this word length in mind.

Through the use of an eight bit system, the system memories built into the FPGA system (FPGA RAM) can be used in blocks of eight bits. For example, the main memory unit which stores the program code for the entire program is a block of code with a 32 bit width or 4x8 width.

### Matrix Size

As has been established above, the Kalman filter is a matrix heavy algorithm. However, these matrices can be of any size. The larger the matrix, the more information that can be stored and used, however, a large matrix does not offer any advantages from a conceptual standpoint. The computational time will be roughly constant for large and small matrices due to parallel processing, and in a well thought out design, changing from one matrix to another should be a simple change.

The benefit to using a smaller matrix is that the instructions can be tested and computed by hand, making debugging far easier. In this case, a 3x3 matrix is chosen for implementation purposes. This can cover the main situation of practical interest for a tracking algorithm. A 3x3 matrix accommodates three input and outputs, which is exactly the amount that would be required in this application that has displacement, speed, and acceleration input signals.

### Memory

Naturally, on the FPGA, there is a maximum amount of memory that is available. The more memory that the system has, the more expensive it tends to be, which brings up the final cost of the system. Therefore, it is a goal to be able to use the least amount of memory to complete the task as possible. The target for the maximum number of instructions is chosen as 512. The system should be able to read inputs, initialize, write outputs, and perform the Kalman filter algorithm in it's entirety in a recursive fashion using only 512 instructions in the control memory.

Since the data is in the form of 3x3 matrices, the data memory should have plenty of room to store an ample amount of matrices. Thus there are no limitations from a memory standpoint. If during expansion, it is apparent that more memory is needed, an alternative FPGA, with two or three or more times the amount of memory can be chosen. This is one of the benefits of choosing an FPGA system. The underlying structure can be removed and exchanged with minimum hassle on the part of the user [13].

### Input/Output

Another limitation on the FPGA is the number of input and output lines. The number of input and output lines should easily fit on even the most basic of FPGAs. Once the FPGA was chosen, this was never found to be a problem, and it stays constant throughout the implementation. For example, on the Virtex-6 FPGA that was used for implementation, there are at maximum 360 I/O pins available to the user. This is more than enough for this application. In fact, for the main use of I/O, the reading and writing of the data matrices, only one ninth of the total amount is required, as the matrix is read and written sequentially, one element at a time [14]. The Virtex-6 is a relatively newer FPGA, introduced in 2009 [15] to satisfy the need for higher bandwidth at lower cost and lower power usage. Older systems, such at the one used in [10] or [12] do not have as many dedicated multipliers into the reconfigurable fabric. Thus, not all of the parallelism or advanced systems can be exploited, as they are in this system.

#### Efficient Use of Resources

The computational resources are limited on the FPGA, and the amount of resource use should be kept to a minimum. At most, the FPGA should be used up to 75% of it's capabilities. As the results section shows, this target was easily met in most areas.

### 1.4 Work Completed

With the above specifications in mind, the following work was completed in the designing of a robust and reliable Kalman filter computational system.

The first area of work was the “Controller” system: In order to run the Kalman filter algorithm on an FPGA, an underlying system was designed and built to be a flexible and efficient way to run a matrix based algorithm. The Kalman filter algorithm was analyzed and certain operations were determined to be necessary for the successful operation of a Kalman filter algorithm, and these operations were optimized for the controller system. The decoder of operations such as matrix multiply, matrix addition, finding the determinant, among others was built into the controller system. The controller system was optimized for branch instructions, due to the recursive nature of the Kalman filter algorithm. The algorithm requires input and output operations that read and write from peripherals. The peripherals represent the measurement instruments that would be required in a real world application of the Kalman filter. The controller is able to access these peripherals, and can interface with peripherals of varying performance, due to the implementation of a handshaking protocol. All of the above capabilities were designed to meet and exceed the performance specifications defined in the initial stages in the design.

Extensive debugging and testing was undertaken to ensure the reliability of the system. Since the Kalman filter can be used in a wide variety of fields and situations, it is essential that it be able to perform without unexpected errors. In a mission critical task such as tracking a missile or an aeroplane, reliability is paramount. After heavy simulation examining and using large amounts of data and computing time, the system proved to be robust and performed admirably under all tests.

After testing and perfecting the controller system, the next main area, the Kalman filter code itself was developed. This code, as explained below uses a Perl scripting system to convert human readable instructions into machine executable code. This code is then stored in the control store memory of the controller, and easily fits into the 512 memory limit we set for ourselves.

A smooth transition from the software code to the hardware system was ensured by the use of an automated Perl script to convert the code, thereby mating the software world to the hardware system.

Due to the complex nature of the system, a complete explanation and user guide was developed, much of it in the work below. Instruction sets were explained, guidelines were given, and operation was documented heavily, all in the name of making the user experience as frictionless as possible.

The Kalman filter code with the controller system was rigorously tested under simulation, and many of the waveforms from these tests are given below. The results of all tests are documented below and we are pleased to report that all of the target benchmarks developed above were met and in many cases exceeded by wide margins.

Taking into account the above main ideas, the following specific contributions were made:

- Development of a controller that has 11 instructions (5 for computation, and 6 for I/O). Instructions are: NOP, JUMP_U, JUMP_DRDY, JUMP_ACK, READ, WRITE, ADD_33, MULT_31, MULT_33, DET_33, MULT_3S.
- Development of a precise and practical handshaking system for efficient handshaking of I/O.
- Writing, debugging and finalizing code for a controller with the above instruction set and I/O capabilities.
- Testing and developing overall system through the use of test-benches and code verification methodology.
- In depth analysis on an operation by operation basis of the overall controller system.
- Developing and testing of the Kalman filter algorithm including simulation of the Kalman filter algorithm code for verification purposes.
- Quantifying the performance of the Kalman filter though the use of FPGA usage statistics and results.

### 1.5 Report Organization

The report given below is organized to highlight the many strengths of the system, and to provide the user with guidelines to its use and limitations.

### Chapter 2: Controller

In the next chapter, the Controller system, which forms the foundation of the Kalman filter system, is documented. Instructions and processing paths are explained, so as to allow the user to use the controller system with the Kalman filter code developed, and to be able to intelligently modify and redesign it to suit his or her purposes.

Instruction sets, timing diagrams, code examples, and waveforms are all used to explain and document the operation of the Controller.

### Chapter 3: Kalman Filter

The Kalman filter itself is developed in Chapter 3. The basics and the theory behind the Kalman filter is given in the first section, then the user is taken step by step through all of the stages in this recursive filter. By the end of this section, the user should have a good understanding of the reason why certain instructions were chosen for the Controller to implement, as well as having a strong handle on the Kalman filter itself.

An example Kalman filter algorithm is given, illustrating the operation of the Kalman filter, which puts the user in a good position to understand the next section in the chapter, which details how to use the Kalman filter with the Controller system.

### Appendices

Appendix A gives the MATLAB code for the Kalman filter example in Chapter 3. Appendix B gives the Perl code to implement the Kalman filter on the Controller, and Appendix C contains detailed performance figures for the implemented system on the FPGA.

# Chapter 2: Kalman Filter Controller

In this chapter, we present the design and implementation details of a controller system, that is developed with the goal of programing a version of the Kalman filter algorithm onto it. It is designed with the objective of making it as flexible and user friendly as possible. Certain elements are present which separate the user from the details of actual processing. For example, a handshaking system is built in to interface with peripherals of differing characteristics. The operation of the handshaking protocol is explained below, to allow for future expansion of the design. Due to the widely varying applications of the Kalman filter, the system has been built with an eye to versatility. Logically separate functions are divided into separate units, which are fully modular and can therefore be removed, or used in alternate hierarchies for maximum utility.

Figure 2 is a block diagram of the main elements of the controller: the ALU, Data RAM, KF IO, Loader, Control Store and Branch Unit. All of these elements are discussed in detail in the following sections.

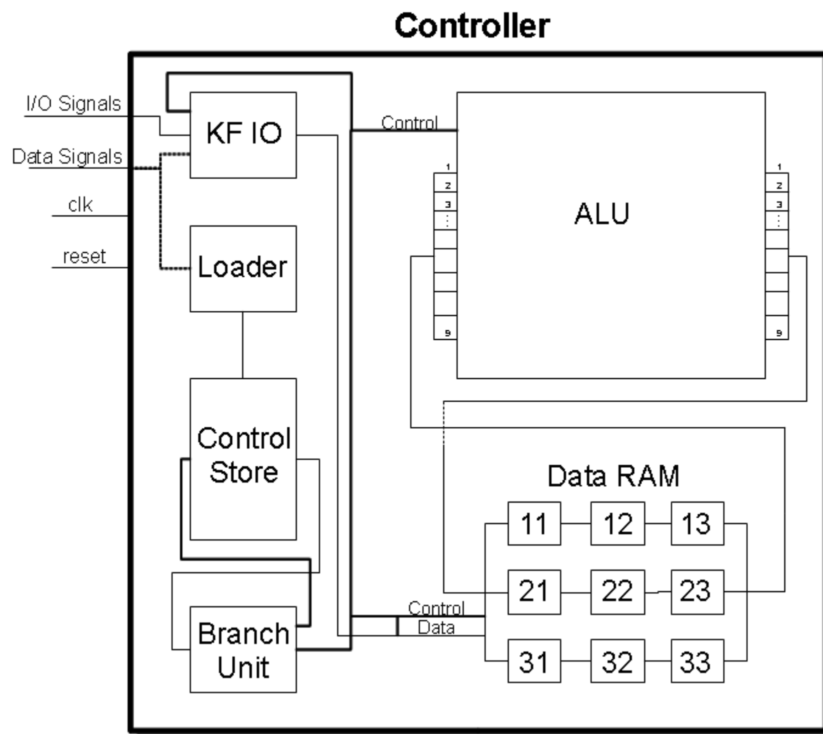


**Figure 2: Controller Block Diagram**

The chapter is organized in the following way:

Section **2.1 Controller Elements**, lists all of the elements which, when arranged together, form the controller system. The blocks discussed are:

- Loader
- Control Store
- Branch Unit
- KF IO Unit
- Data Store
- KF ALU

After each element is presented, the overall system is illustrated in Section 2.2. The diagrams are taken from the HDL simulation tool used to develop FPGA systems, Xilinx ISE Tools. There are many detailed systems which make up the FPGA, and one of the benefits of using an FPGA system is that the user can disregard the details of chip implementation, and focus on the logical design. Therefore, individual flip-flops and latches are left up to the simulation tool to place and route. Data sheets such as [14] provide a summary of (Virtex-6) FPGA features, configuration information, clock management, RAM and IO information. These sheets are useful and necessary for any detailed Xilinx FPGA work.

In Section 2.3, the programmer's interaction with the system is discussed. Through the use of the instructions which we have developed, the system user is able to develop the desired implementation of the Kalman filter which best fits his or her needs. Through detailed discussion in this section of the Instruction format, Execution, Operation cycles, and Input/Output, the user is able to know the exact method for implementation. Therefore the user can potentially change or upgrade the system to tailor the solution to his or her exact application.

A Perl script, which is used to compile the program code is discussed. This is another tool which abstracts the implementation details from the user, as the Perl script is used to take human readable code and convert it into machine loadable code. In this way, the user is able to debug and design in an environment much more comfortable than the ones and zeros of machine code.

Finally in Section 2.4, the details of implementing the Controller system are discussed, including compiling, debugging and number representation. This section is useful for the user that wishes to modify the base code and expand on this project.

## 2.1 Controller Elements

The main elements of the controller, and a brief description of how they fit into the overall structure of the system are listed as follows.

- Loader
  The Loader is only active during the initialization phase. It is responsible for loading the program code into the system, directly via the control store memory
- Control Store
  This unit stores the program code. After initialization, the program code dictates the operation of the system
- Branch Unit
  This unit works in conjunction with the control store to traverse the program code. Specifically, it is responsible for determining the next step in the program run, which may or may not be the very next instruction in the program, due to the use of program jumps, and KF IO handshaking timing and operations
- KF IO Unit
  The KF IO Unit interfaces the controller with the outside world, inputting and outputting data and handshaking signals, which ensure the integrity of the data transferred, as well as ensuring reliable coordination with peripherals of varying performance characteristics
- Data Store
  The matrix data which is used for all arithmetic operations and calculations is stored here. The data store is in the form of nine memory elements, which together are conceptually thought of and organized as the nine elements of a 3x3 matrix
- KF ALU
  The KF Arithmetic and Logic unit is responsible for all computational operations, including matrix multiplications, additions and others, many of which are computed in parallel due to the planning and full use of FPGA resources available to the system.

The following sections discuss in further detail the above elements which make up the controller system.

### Loader

The loader, along with the control store, are the first units to become active once the Controller is initialized. The loader is used to fill the control store with the program data--the data that specifies the entire functionality and operation of the Controller. In testing, data coming to the loader is hard-coded into the program, in the test-bench that wraps around the system. The user enters code by running assembler written in Perl which

converts human understandable code into the hex code used by the controller. This hex code is what is stored in the control store with assistance from the loader.

Once the controller is initialized, the loader begins operation. It stores each hex code into the control store at the correct address. The addresses are contiguous, placing each instruction next to the previous one. The control store uses these well organized instructions once the system is operational, after the loader has finished loading. After the loader has loaded all of the hex codes that make up the program, it passes control onto the control store and becomes idle. The control store then runs though, one by one, in sequence, the code that the user has specified. (See Figure 3: System Initialization).

The important thing to note is that the loader has a mutually exclusive run time with the control store. If one is running, the other is not. At the beginning of the system's operation, or after a reset, the loader is initialized, the loader then begins loading the program instructions into the control store. The control store is not active, except as a memory bank in which the loader places instruction data.

### Advantages and Disadvantages

There are advantages and disadvantages to this form of set up, where the loader loads the control store with program instructions, then freezes for the remainder of the system run. For example, since the loader is frozen, loading a program dynamically is not possible. If we would like to change what is running on the system, this is impossible without a total reset, in which case, the loader would again reinitialize the system and the control store with what could be a totally new program. However, all of the steps to initialize the system, including reading in data matrices would have to be repeated.

This way of operating, in which the program is only loaded at boot and never changed in that run is similar to the way a FPGA works. There is no “run-time swapping” function. In an FPGA, the system is initialized with a certain hardware set up, and this is kept for the entirety of the system run, until the system is reset and re-configured. The advantage to this system is that there is no operating system needed. This makes the code more compact, thus smaller.

To make changes to the system mid-run, and operating system would be required. The operating system is able to reach into the system and make changes to the core of the program, while still allowing the system to run. An operating system takes a lot of memory. In this case, since only the control store memory is available, it was necessary to avoid the large operating system. Without an operating system, there is no operating system overhead in terms of memory, memory management and processing. Thus, the entire code memory (program memory) can be used for computation. There are no resources dedicated to tasks such as garbage collection, defragmentation, resource allocation, or other operating system tasks.

With this system, the entire program is loaded at boot-time. It can be tested and perfected before loading, and there is no need to put testing resources towards determining how the

program will interact and react to the operating system processes, as one would have to do, if the program was loaded at run-time. Run time loading is far more prone to errors due to the many different scenarios that can arise during the run of a program including program modification. In boot time loading, once the program is tested and loaded, it will stay the same, and if it runs once without errors, it will continue to run indefinitely in the same error free way.

It is necessary, however, even without an operating system, that the program fit into the memory bank, (control store) right from the the start. Since there is no operating system, there is no dynamic swapping ability of memory. The program must be a relatively straightforward, self-contained system, for as we have seen, once it is loaded and started by the loader, there is no going back without a total reset.

**System Initialization**

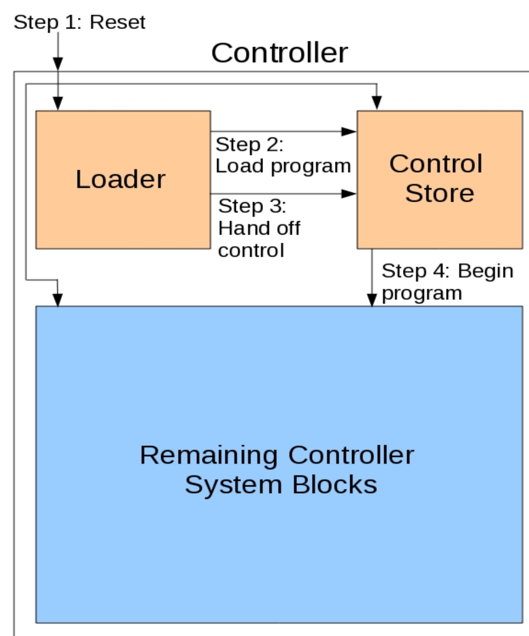


**Figure 3: System Initialization**

The above Figure 3 shows the four initializing steps taken at the start of every run of the system.

**Step 1:** At reset, the reset signal travels to every block in the code and puts each respective one back to its initial state.

**Step 2:** The next step is always step two above, a loading of the program into the control store by the loader.

**Step 3:** The loader then hands off control in step three, and freezes itself.

**Step 4:** Then the actual program code begins, in step four, where the remaining control blocks come into the picture.

Through the use of this four step process, all of the program code is at this time, loaded into the system. Therefore, although the system has the capability to load/store, i.e. read/write from an outside source, there is no need to do so for program code. There are never any slow read/write operations to the program memory during the operation of the program. This make for a very efficient system, in which the entire program can be run in a much better known time frame, and it can be analyzed and optimized long before it is loaded into the system, making execution time deterministic.

### Program to Load

The program that the loader loads onto the control store is relatively simple and can include: branches, loops, reads, writes, etc. For example, at the start of the program, one may wish to load the operating matrices with initial values. Therefore, the loader should load a program that contains, in this case, nine read instructions for one matrix, and then nine more read instructions for the second matrix. Through this initialization process of nine reads per operand, a significant portion of the program memory is used to initialize the system. This is one area that could be improved, if the memory requirements ever became too large, though the use of loops, or of a larger memory. In this case, the system was especially chosen so as to ensure there was ample memory for all of the initialization, and the operational code itself [16].

The following table is an excerpt of the code and shows an example of one matrix being initialized though the use of read instructions. The READ instructions are used to load outside data into the internal memory of the controller, through the use of the IO capabilities of the system. ARG 1 signifies the place in the 3x3 matrix in which to store the current information, which is why it changes from 0 to 8, using nine different locations. ARG 2 stays the same. All of the nine data points are conceptually tied together by the use of ARG 2 which, in this case, is the memory location of the 3x3 matrix.

| OPCODE | ARG 1 | ARG 2 |
|---|---|---|
| NOP | | |
| READ | 0 | D10 |
| READ | 1 | D10 |
| READ | 2 | D10 |
| READ | 3 | D10 |
| READ | 4 | D10 |
| READ | 5 | D10 |
| READ | 6 | D10 |
| READ | 7 | D10 |
| READ | 8 | D10 |

**Table 1: Assembly Matrix Initialization Example**

Once the controller is running, it will move through this initializing section of the code and the IO unit will be called, due to the opcode that control store will have passed onto the controller, which is loaded at initialization. After this part of the code is complete, the rest of the program may be made up of many arithmetic instructions, loops, branches, and writes. For example to read two matrices and to add them and then export the result to an outside bank, the code would include 18 reads, one addition and nine writes. Or, if the reads had already been completed, the code would only need the addition command which includes in it two reads, a computation cycle, and a write cycle, which stores the result in an internal memory location specified by the control store.

All of the locations of the two input matrices and the write matrix are specified by the control store. The instruction that the control store sends to the Controller contains in it all of this information. (See Figure 16: Instruction Format, on Page 35 below)

**Control Store**

Control store is the unit that contains the actual program that the Controller runs. At the initialization of the system, the loader loads the program into the Control Store. (see Loader). Once the loader has handed off control to the control store, the control store runs the system. The loader freezes and is not called again until the system is reset.

The word length of program memory is 32 bits. The control store program is in the form of a 32 bit hexadecimal number which contains all of the information needed by the rest of the system. Each block of the code store is able to direct a complete cycle of the program. For example, in an addition operation, the code will direct the system to the correct memory location for the first matrix, which can be accessed by the system. Then, the next section of the line will direct it to the second matrix. The opcode section of the

line will signify that it is an add operation, and the final section of the line will direct the system to the address in which the store the result of the addition operation.

Depending on which operation is to be computed, (signified by the op-code), the order of the arguments may differ. (See Instruction Format, on Page 35, below)

Control store is implemented with a two port memory unit on the FPGA, RAM_512x32_2Port_SC. The two ports are used separately, the A port is used by the loader to load the program data into the control store. Once the control store is fully loaded, the A port is no longer used for the duration of the system operation. During the system run, at each instruction cycle, each instruction is fed from the control store to the rest of the system using the B port. The B port is used as long at the program is running.

This use of a two port system seems excessive since the two ports are not used at the same time, but in reality, the use of two ports saves a great deal of time and effort on the designer's part, and in terms of computation time during program run. If a one port memory would have been used, it would have required a block of combinational logic external to it, to determine which resources could use the port and at which time. The exact timings for loading or reading would have had to have been established to ensure there was no data collisions. All of these considerations would have been implemented in an "arbitration hardware" which would have taken up resources on the FPGA.

The two port solution does use up more resources than the one, but overall, taking into consideration the time to design, and the added logic needed for a one port solution, the two port method is the most efficient solution. It is akin to having a one lane road or a two lane road. By using a two lane road, each of the lanes can be dedicated to one direction. There is no chance of confusion, and traffic is much more smoother.

### Branch Unit

The branch unit is responsible for all branch operations. The next step in a sequence of instructions can be at any location in the code, thorough the use of an "absolute jump."

First, the code is set up with flags, or labels, to indicate specific locations in the process that one may need to jump to. For example a branch flag can be placed at the beginning of a loop that checks for input. When the assembler code is converted to machine code, the converter replaces the flags/labels with absolute addresses, that the machine can use to go to specific locations in the code. See Table 6: Assembly Jump Example, on Page 40, for an example of a label used for jumping purposes.

In this way, one can ensure that the processing of the code does not move forward until a certain condition has been met. For instance, if one is waiting for a peripheral to input data that is necessary for the next operations, there is no point to moving forward unless the data has been received.  With the branch unit, the code can be set up to check if the data has been received before moving on to subsequent operations, ensuring that those

operations are operating on useful information. The system performs these checking functions though the use of handshaking. Handshaking allows the system to interface with peripherals with arbitrarily latency. In other words the latency of any peripheral is transparent to the user.

The same can be done for outputting data, where a branch can ensure it does not move on, i.e., it does not stop sending data to the peripheral as long as the peripheral has not received said data.

## KF_IO Unit

The KF_IO Unit is the input and output unit of the controller. Through the KF_IO Unit data is send and received from data stores inside the controller to data memory. The data is of length "Wordlength" which is consistent with the rest of the controller unit.

The KF_IO operates by being aware of the current opcode being used. The decoding of opcodes is done locally in the KF_IO, thereby keeping the global operations to a minimum. If it detects an opcode associated with input or output functions, it will become active and perform the functions that that opcode require.

There are four data paths that the KF_IO unit uses: IO_source (inward and outward) and IO_target (inward and outward). As the names suggest, these signals are hooked up to each side of the KF_IO unit, depending on which direction the data required is meant to travel. The directions are summed up in Figure 4 below [17].

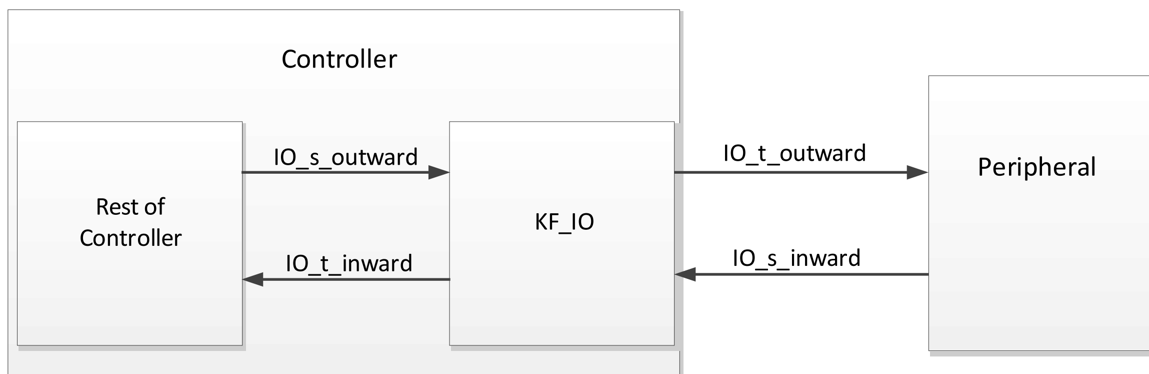


**Figure 4: KF_IO Signals**

### Handshaking

The KF_IO unit is capable of initiating and receiving handshaking protocols. Handshaking is necessary to fulfil the functional goal of being able to interface with peripherals with arbitrary latency. There is a handshaking signal associated with each of the respective data paths that the KF_IO uses.

A diagram of the main handshaking signals is shown below. On the left is the controller unit, and within it is the unit responsible for the handshaking, the IO Unit. Within the IO Unit there are two main blocks, the read and write blocks. These two blocks use different signals to communicate with the outside block. The outside unit, or the peripheral, is shown below on the right. The peripheral and the controller communicate through the IO Unit through the use of a data line, shown below as IO_DATA, and the four handshaking signals, discussed below.

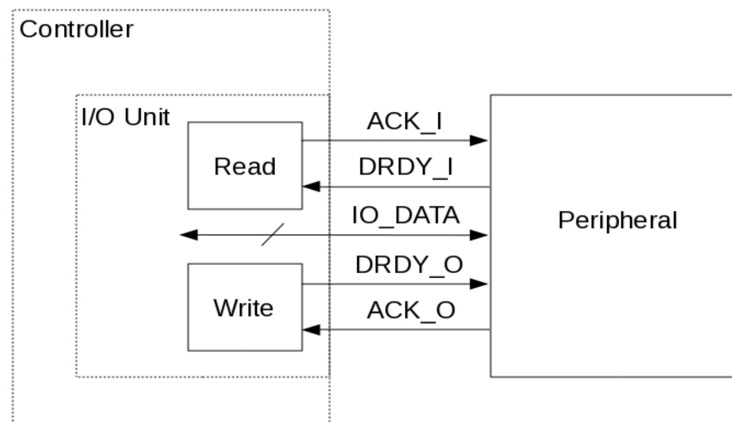


**Figure 5: Handshaking Signals**

The handshaking signals are: DRDY (input and output) and ACK (input and output), which signify "data ready" and "acknowledge." The "data ready" is used when sending data (in either direction) and this signal indicates that the data that KF_IO is providing is now stable and can be used by the receiving party. ACK, or the "acknowledge" signal is used in the opposite case. When the KF_IO successfully receives data, it asserts this signal to indicate to the sending party that it is now safe to change the data on the bus, as the data has now been stored by KF_IO.

The protocols by which the four handshaking signals are used in situations such as read or write operations are implemented in software, as presented below.

**Read Protocol**

A read is the operation that is used when the controller inputs in data from an outside source, (the peripheral) and stores it into internal data memory. Therefore it makes sense to have some sort of signal through which the peripheral knows when to send data. If the peripheral is sending data, and the controller has not yet stored it, that data needs to be held on the data line between the controller and the peripheral until the controller is done with the data. We then expect there to be between the controller and peripheral the following: a data bus, a signal to signify to the controller that the peripheral has data to send, and a signal to let the peripheral know that the controller has received the data.

The signals that are used in the Read case are: ACK_I, a signal used by the controller to acknowledge receipt of data; DRDY_I, a data ready signal sent by the peripheral; and a data bus, DATA.

In an I/O Read operation, the following occurs:

- DRDY_I issued by peripheral to signify new data is stable and ready to be stored by the controller.
- ACK_I issued by the IO Unit to signify that the data has been received and safely stored in the controller. The data can now be changed by the peripheral at its convenience.

The timing for the read handshaking is given in the diagram below. Notice that due to the nature of handshaking, the ACK_I signal given by the controller ends up acting like a *strobe* signal for the data transfer, however, it is in no way a clock signal. The exact times for the transfer depend on the speed of the peripheral, but the order of events is consistent during a transfer [16].

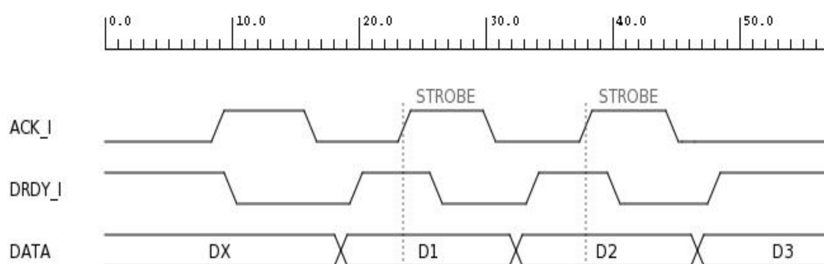


**Figure 6: Timing Diagram, Read**

Coding the read handshaking protocol

During a program run, there is an operation that is especially suited to implement in software the read handshaking protocol. That operation is JUMP_DRDY_I. In a written program, this operation would be placed at the appropriate place, which is just before a READ operation so that the READ would not be executed until the DRDY_I signal was triggered, just as the protocol dictates.

Code is written in the following format for both the read and write handshaking.

| Label field: | Mnemonic field | Operand field |
|---|---|---|
| | JUMP DRDY I | <program memory address> |
| | JUMP ACK O | <program memory address> |

Example code for the READ operation with handshaking is given below.

```
READ ; dummy instruction for initialization
...
```

```
L1:   ...
      <some other instructions>
      ...
      JUMP_DRDY_I L1   ; deactivates ACK_I ('0') and loop back to
                       ; address L1 until DRDY_I becomes active
                       ; ('1')
                       ; ("new data is available")
      READ             ; read the data and activate ACK\_I ('1')
```

**Write Protocol**

The write protocol is similar to the read protocol. In this case, the controller waits for the peripheral to be ready to write more data into its memory banks. Therefore the signal that the controller sends to the peripheral is a DRDY_O (output) signal and the peripheral returns the handshake with an ACK_O signal. The events for the handshaking protocol are as follows [16].

In an I/O Write operation, the following occurs:

- DRDY_O is issued by the IO Unit in the controller to signify to the peripheral that it has data ready to be written into the peripheral
- ACK_O is issued by the peripheral once the data has been written into it's memory banks

In this case, the DRDY_O signal issued by the controller acts as a *strobe* signal, (not a clock signal), as shown in the following timing diagram, Figure 7. Once again, the exact timings of events depends on the speed and power of the peripheral in question, but the order of events in the write protocol is consistent across all peripherals.

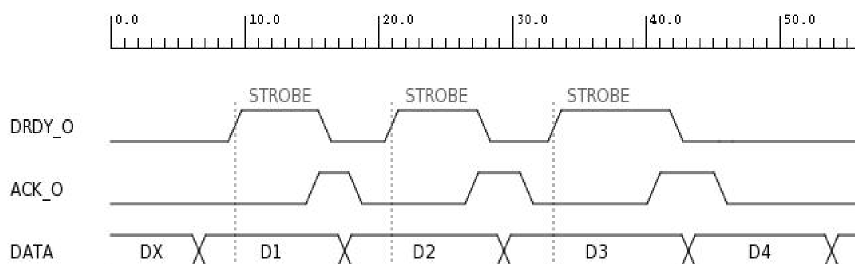


**Figure 7: Timing Diagram, Write**

Coding the write handshaking protocol

In this case, in the write protocol the WRITE operation is required to wait until the ACK_O signal is triggered before new data is written on the bus that is sent to the peripheral. In this way, the data is held constant until the peripheral is done with it.

Code is written as follows, and is in the same style as the above read code with a label signifying where in the code for the JUMP_ACK_O to jump to while it waits for the write operation to complete:

```
L1:    ...
       <some other instructions>
       ...
       JUMP_ACK_O   L1  ; deactivate DRDY_O ('0') and loop back to
                        ; address L1 until ACK_0 becomes active
                        ; ('1')
       WRITE            ; write the data and activate DRDY_O ('1')
```

As the above shows, handshaking is an essential and important part of the controller as it decreases the chances of data corruption, and make the overall system more efficient as time is not wasted over-engineering the data transfers to ensure successful transfers in all situations.

### Data Store

There are nine memory banks in the controller system used to store data. Data can be thought of as a 3x3 matrix with each memory bank serving as an individual spot on the matrix. One of the reasons for choosing the 3x3 data structure is that it was determined that for this application, and many others, there were three main pieces of information that would be needed. In this case, the three spots can be used to hold position, velocity and acceleration information, which is useful for a tracking type of device.

Data is linked together in the form of a matrix. Each memory unit is sent the same address when accessing or writing data. Therefore the "1" spot in each memory bank is associated with the "1" spot in every other memory bank. This way, a vector-valued argument is being accessed. The only other consideration is which spot in the matrix each data element is placed. This is solved by the naming convention of the data banks.

The argument for an instruction is, in most cases, a 3x3 matrix. Therefore, an address to data memory drives all nine data elements in parallel. The names of the nine units are: data_store_11, data_store_12, 13, 21 ... all the way up to data_store_33. It is always known which position in the 3x3 matrix each element belongs to.

In this way, data can be written in parallel. Only one address is needed, and this address goes to all of the memory elements. As well, a write or read signal and of course a data bus is also sent in parallel to each unit. The data to each unit may or may not be the same, depending on the matrix that is to be written, or read. The data lines are unique to each memory location.

Once the read or write signal is received, it only takes the amount of time it take to write one memory block to write all of the memory blocks, due to the parallel set up of the memory system.

Each memory unit has the following signals going to it:

```
clk : in  STD_LOGIC;
ce : in  STD_LOGIC;
we : in  STD_LOGIC;
address : in  STD_LOGIC_VECTOR (8 downto 0);
data_in : in  STD_LOGIC_VECTOR (31 downto 0);
data_out : out  STD_LOGIC_VECTOR (31 downto 0));
```

"ce" is chip enable, and it should always be on for the duration of the program operation. "we" or write enable is the bit that specifies whether the memory is to write or read in each clock cycle, clk.

The "address" is the specific location on the memory which is the same for each unit. In this case, since the address is 9 bits long, there are $2^9$ possible data locations, and subsequently $2^9$ different 3x3 matrices that can be used during the run of a single program. "data_in" and "data_out" are the individual data lines (32 bit width in this case). Therefore data store can be thought of as an area of blocks of nine values which are 32 bits in width [17].

**KF ALU**

The Arithmetic Logic Unit, or ALU, is the main computation engine of the system. It is connected to the rest of the system by the wires representing the opcode, 18 input lines and 9 output lines. The 18 input lines are made up of 9 lines for the first input and 9 lines for the second input. There are nine lines per input due to the fact that most of the operations are computed on 3x3 matrices connected through nine address lines. Therefore, all of the relevant numbers making up the matrices can be loaded into the ALU in one clock cycle. The inputs are distinguished by the "s1" or "s2" marker, indicating source 1 or source 2. There are also nine outputting lines marked "t" for target which represent the 3x3 result of the ALU operations.

Once data is loaded into the ALU, the arithmetic or logic operation to perform is determined by the opcode, which is also loaded into the ALU in parallel with the data. There are five different operations that can be performed by the ALU: Add 3x3 to 3x3, Multiply: 3x3 by 3x3, or 3x3 by 3x1, or 3x3 by 1, and Determinant of a 3x3.

The ALU accepts two input arguments and returns one output. All decoding is done locally, which means that a common, and large, central decoder is not necessary. This makes for faster processing speed, as the ALU can be dedicated to its purpose.

The following table summarizes the ALU operations and provides a brief description of each, for reference. Each operation is discussed in further detail following the table below, Table 2.

| ALU Operation | Input | Description |
|---|---|---|
| Add 3x3 to 3x3 | Two 3x3 Matrices | Adds two 3x3 matrices using a matrix addition, that is, adding each element individually |
| Multiply 3x3 by 3x3 | Two 3x3 Matrices | Matrix multiplication of two 3x3 matrices. |
| Multiply 3x3 by 3x1 | One 3x3 and one 3x1 matrix | Matrix multiplied by a vector. Result is a 3x1 vector output |
| Multiply 3x3 by 1 | One 3x3 matrix and one number | All of the elements of the matrix are scaled by the number |
| Determinant of a 3x3 | One 3x3 matrix | The determinant of the 3x3 matrix computed. Output is one number. |

**Table 2: ALU Operations**

Each operation, the Add, the 3 multiply operations and the determinant operation are written to separate files which are integrated by the overall ALU unit. This makes for somewhat encapsulated code which can be individually tested and debugged.

Depending on which mathematical operation is called for by the opcode, the input data is routed to the correct file, and the output is read from that location. The overall ALU file organizes the individual operations.

"Add 3x3 to 3x3" adds each individual number in the first matrix to the corresponding number in the second matrix and outputs the respective results on the nine "target" output lines.

"Multiply 3x3 by 3x3" performs a matrix multiplication on the two source matrices and outputs the 3x3 result. This is not a simple one to one operation like addition, therefore care must be taken in the area of numerical representation to ensure that the result is interpreted in the correct light.

For example, since the result for one element of the output from matrix multiplication takes the form

**c12 = a11xb12 + a12xb22 + a13xb32**

As shown in Figure 8,

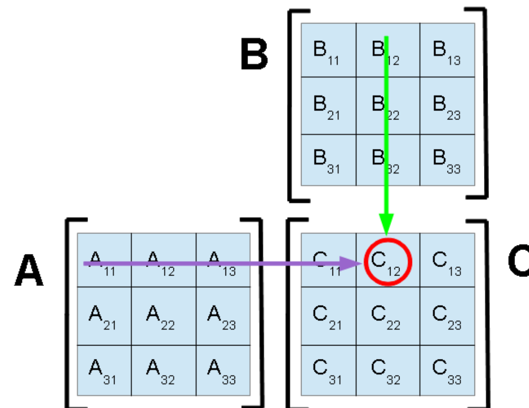


**Figure 8: Matrix Multiplication Example**

where each output, say c12, is computed through three multiplications and two additions, care must be taken to ensure that the numerical representation is consistent with those operations. See Numerical Representation section below.

"Multiply 3x3 by 3x1" is a matrix multiplied by a vector. Like multiply 3x3 by 3x3, there is not a simple one to one operation for each result. Each output is computed through 3 multiplies and 2 additions. However, the difference now is that the result is a 3x1 vector and not a matrix. In this case, the output lines 21 to 33 are simply ignored. Since the opcode is sent everywhere in the program, the rest of the code is aware of this change in output.

"Multiply 3x3 by 1" is a multiplication of a matrix by a scalar. This is a direct one to one output as the matrix is simply scaled by the scaling factor. The output is once again a 3x3 matrix, which must be interpreted as a two number simple multiplication per output.

"Determinant of a 3x3" is the determinant of a 3x3 matrix, which is computed using the following formula:

$D = a_{11}\{a_{22}a_{33} - a_{32}a_{23}\}$
$+a_{12}\{a_{23}a_{31} - a_{33}a_{21}\}$
$+a_{13}\{a_{21}a_{32} - a_{31}a_{22}\}$

In this case, there are nine multiplications and five additions/subtractions. This changes the output and these operations must be taken into account when interpreting the results.

**Testing ALU**

The testing of the ALU was computed by comparing the ALU generated outputs with a reference implementation in MATLAB. Each operation was run with certain inputs

which were replicated in the MATLAB world, and the outputs were compared for consistency [16].

### Hexadecimal Formatting

In the case of a matrix addition, certain numbers were loaded into the controller via the input/read function. The numbers that are loaded are specified in the hexadecimal format for clarity. In the program that was used to run the VHDL testbench, (ISE), numbers can be specified in any format, however, once the simulation is run, it must be kept in mind that there are no actual numbers being input into the FPGA system. In VHDL, and in other hardware description languages, numbers are not represented in hardware, only bits are. That is to say, everything is a signal, and signals are wires. To represent the number 5, the program first verifies that the signal is wide enough to handle the number. If it is eight bits, the number “5” is converted to its binary equivalent, 0000_0101, and each digit in this sequence is assigned to a wire. That is, all the wires except the 1st and 3rd from the left are held in the logical “low” state.

To simplify debugging, the numbers are converted to hexadecimal format. Each four bit group of binary bits is translated to a hex number from 0 to F, or zero to fifteen. These same hex numbers are then simultaneously entered into the MATLAB script that was created for debugging purposes. This script then runs the same operations as the simulation. In this way, there are two separate but consistent systems working on the same problem. Since MATLAB is ideal for working on complex operations on matrices, (it is named after the phrase “matrix laboratory,” after all), it is the perfect tool to use for comparing matrix operations.

In this way the matrix operations of the controller system were refined and confirmed to be correctly operational, without resorting to complex, time-consuming, and prone to error by-hand computations.

## 2.2 Diagrams

Figure 9, below, shows the outermost level of the controller. Looking at it as a black box, it has certain inputs and outputs, which are shown in this diagram. On the left are the inputs, data_in, clk, rst, and the handshaking and I/O signals. Of course one would expect these to be on the very top level, as these signals take information in and out of the overall system.

Most of the output signals on the right side of the figure are signals used for debugging the system. Signals like ARGUMENT_1/2/3, PC_current, what_branch, etc, are peeks into the inner workings of the system. These signals can easily be tied off, or left open during normal operation, without consequence.

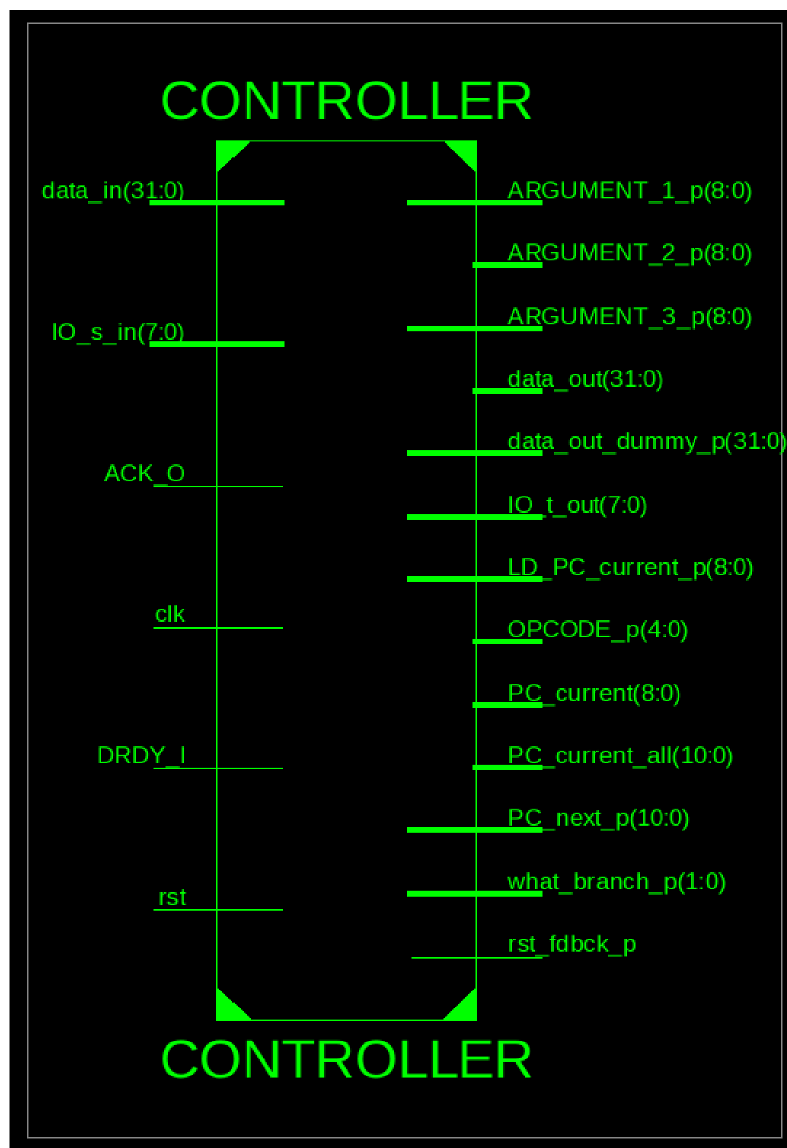


**Figure 9: Controller Top Level**

The next figure, Figure 10, is one level into the controller. Here, we can see the actual inner units that make up the overall system. Units have been discussed above, such as the Branch unit, the ALU, the control store, and the IO unit, as well as the Loader.

The three memory units showing are the data_store_11 data_store_33 and the control_store. There are actually nine data_stores going from 11 to 33, but for clarity, this diagram shows two.

The connections between the units are also shown, with some units communicating directly as inputs or outputs to the overall system.

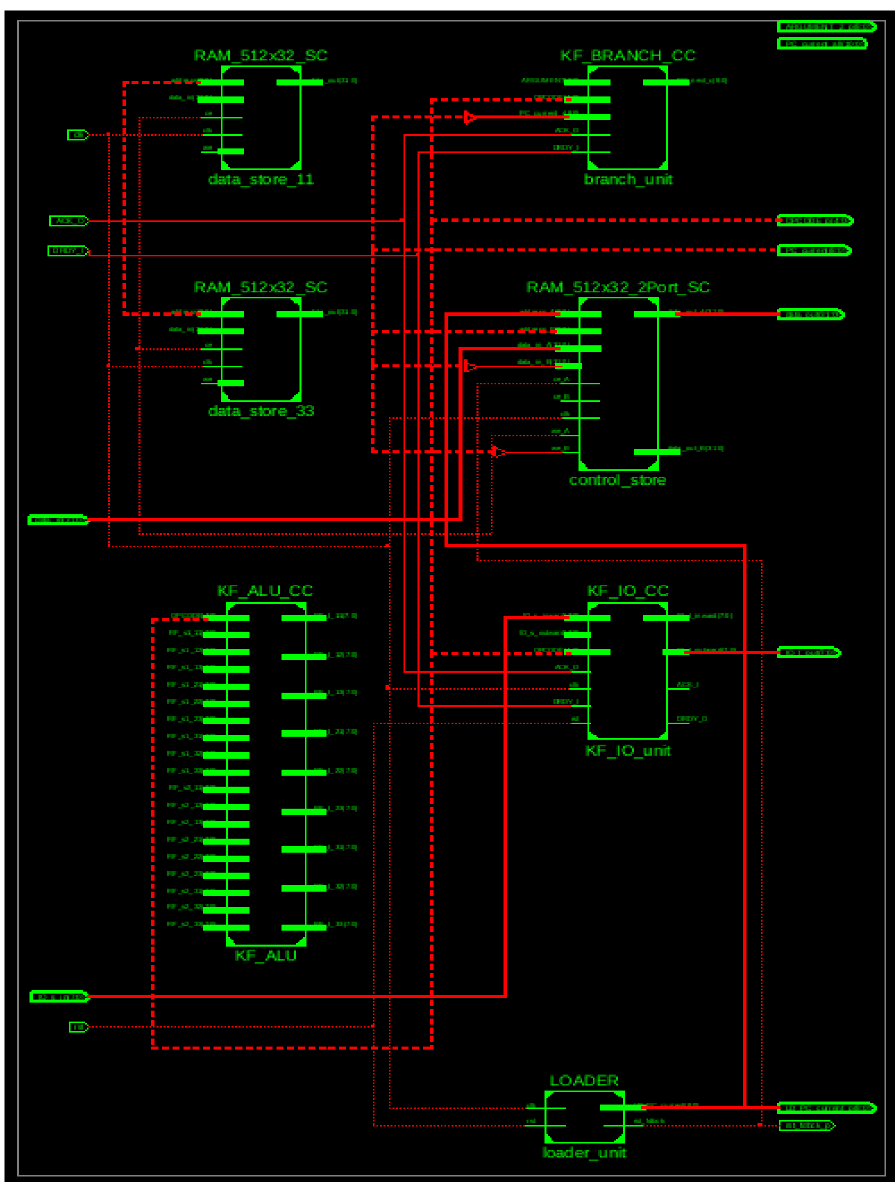


**Figure 10: Controller, One Level In**

The control store is presented next, in Figure 11, below. The top level of the control store is shown. As can be seen, there are two avenues for data going in or out of the control store, A and B. Indeed, as discussed above, the control store uses a two port memory system. Each avenue has its respective inputs on the right, address_A/B and data_in_A/B.

There are also control signals being input on the right, ce (chip enable) and we (write enable) for both A and B to activate or deactivate each port.

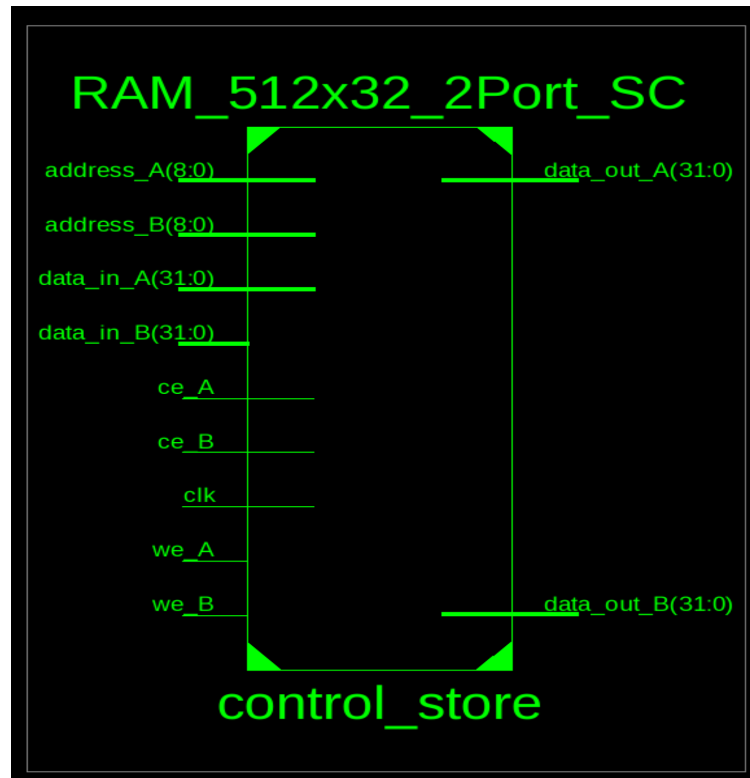


**Figure 11: Control Store, Top Level**

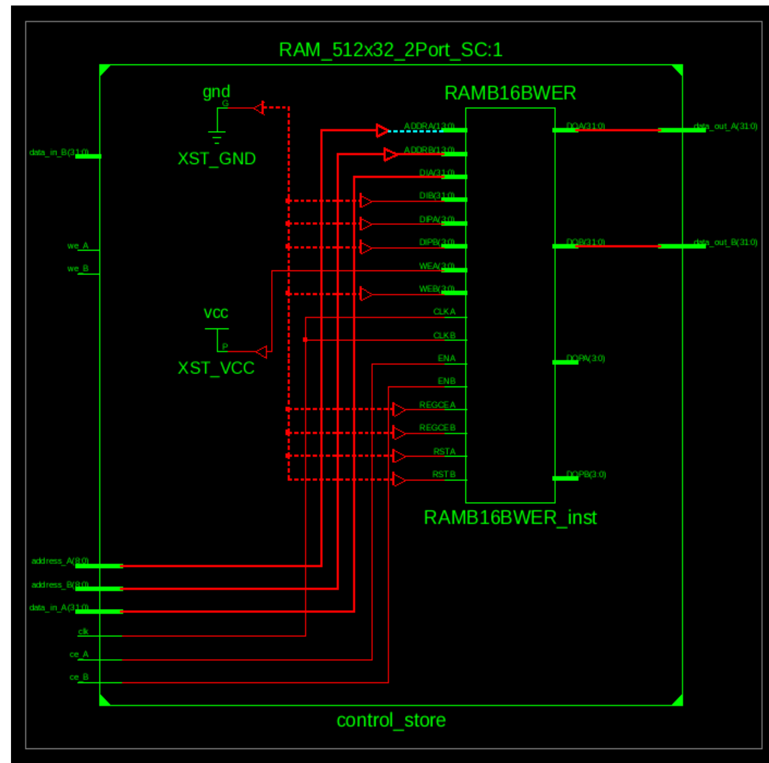


**Figure 12: Control Store, One Level In**

The next diagram, Figure 12, shows the inside of the control store. As is expected, since it is a two port memory unit, the main body of the control store is a block of memory. All of the other signals are used to act upon that memory, transferring to/from it data and control signals.

Figure 13, below, shows the inside of the KF_IO unit. In here, things are a bit more complex than the memory unit above, and as can be seen on the left there is some combinational circuitry to determine when and if to out data or control signals which are being used by the KF_IO system to input and output signals to/from the controller. Again, due to the benefits of the FPGA system, the user need not know exactly how the combinational circuitry works. The implementation tool, in this case, ISE, will take care to implement the HDL (hardware description language, which describes and defines the operation of the system) in the most efficient fashion [18].

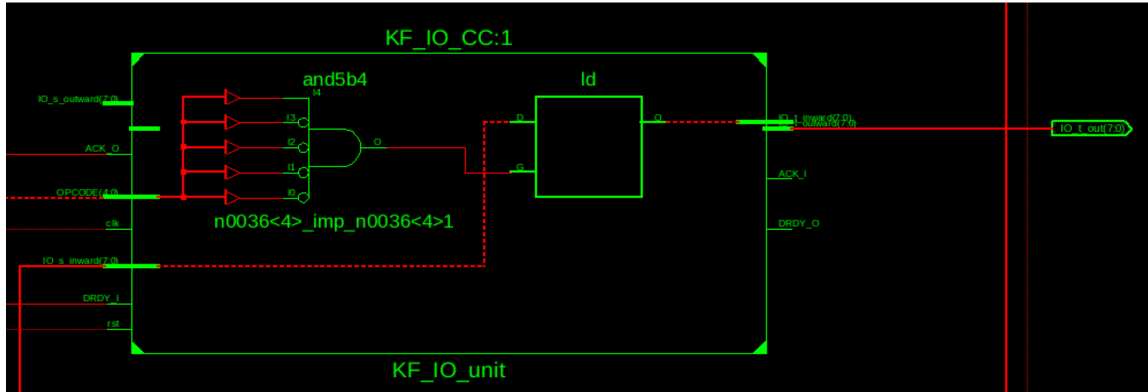


**Figure 13: KF IO Unit**

The Loader Unit is shown next in Figure 14, below. In this case, we have some combinational logic, as well as a MUX and some latches, which are used to store and then transfer the correct program code to the control store, during the operation of the Loader.

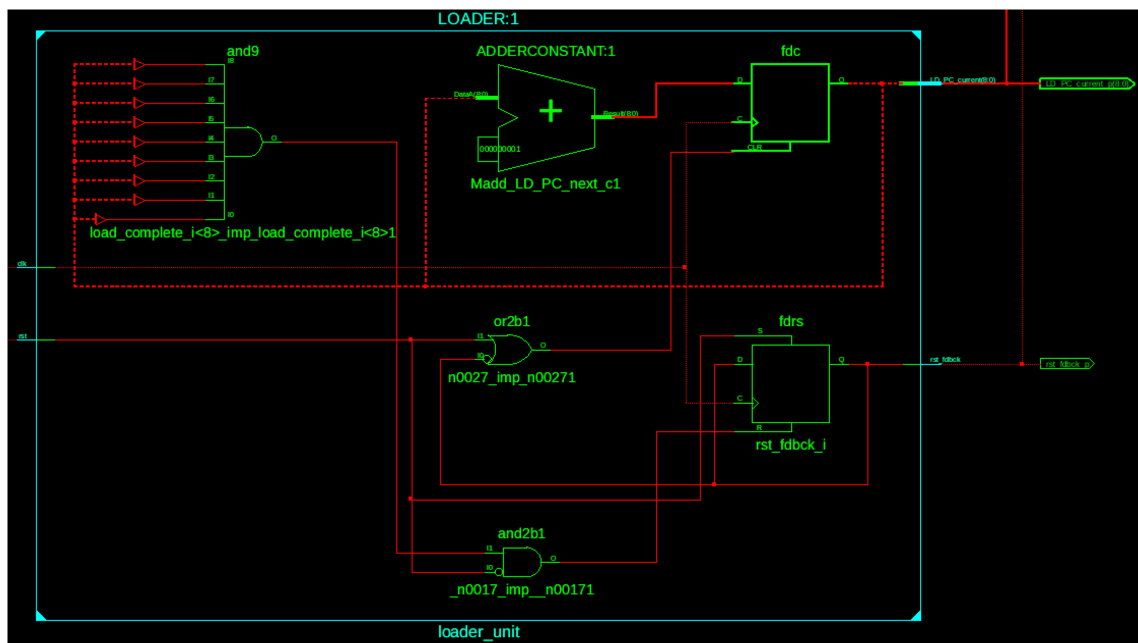


**Figure 14: Loader Unit**

The most logically complex unit yet is shown in the following figure, Figure 15, which details the Branch Unit. The complexity is to be expected, as the Branch Unit is a decision making unit. Options are weighed using predefined options though the multiple multiplexers visible on the right hand side of the diagram. On the left there is a logical operator which is no doubt necessary and the most efficient way to compute the algorithm programed into the FPGA, as determined by the compiler at the time of programming the code onto the FPGA.

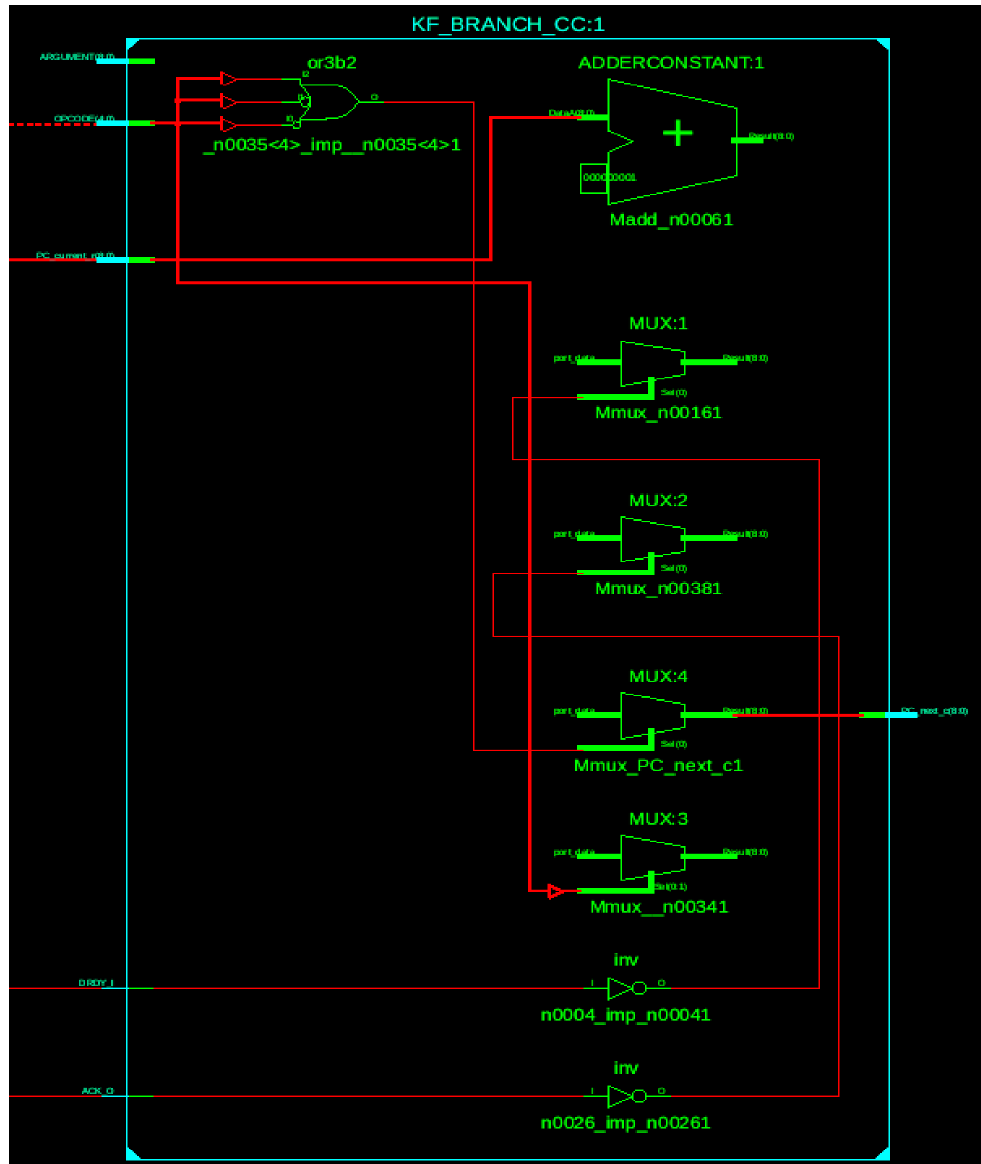


**Figure 15: Branch Unit**

## 2.3 Instruction Set

There are 11 instructions in the instruction set: NOP, JUMP_U, JUMP_DRDY, JUMP_ACK, READ, WRITE, ADD_33, MULT_31, MULT_33, DET_33, MULT_3S.

These 11 instructions are converted into the following operation codes for use in the controller:

NOP => 0x00,
JUMP_U => 0x14,
JUMP_DRDY => 0x15,
JUMP_ACK => 0x16,
READ => 0x10,
WRITE => 0x11,
ADD_33 => 0x08,
MULT_31 => 0x09,
MULT_33 => 0x0A,
DET_33 => 0x0B,
MULT_3S => 0x0C.

IE, JUMP_U is converted to opcode 0x14, which is 0001_0100. However, the opcode is only five bits long. The first three bits are omitted, and hex format 0001_0100 is actually represented at 0_0100. Read, Write, and other instructions beginning with "1" are converted to 1_0000 for read, 1_0001 for write, etc.

By using a five bit long opcode, $2^5$ or 32 different opcodes can be represented in the system. Since there are only 11 currently being used, that means that there could be 21 new opcodes added to the system without any need for extensive updating. This gives us peace of mind for future expansions of the system.

### Instruction format

Each instruction is 32 bits long. It is loaded into the control store, and then used in each clock cycle by the program. The 32 bits are made up of 5 bits for the opcode concatenated with 9 bits for the first argument, 9 bits for the second argument and 9 bits for the third argument, totalling 5 + 27 the 32 bits total, as illustrated in the diagram shown in Figure 16. The numbers on the top indicate the number of bits, and the numbers below the diagram show the length of each unit. The opcode is at the front of the instruction, as it acts as the switch that dictates how the rest of the instruction is interpreted. For example, if the opcode indicated a read instruction, only the first argument would be used, and the remaining arguments would not be relevant. However, if the instruction was a multiply instruction, all of the three arguments would be used. They would indicate the addresses of the two numbers to multiply, and the last argument would be the address where the result is to be stored. In the case of unused arguments, they would be padded with zeros so as not to disturb the spacing of instructions. That is,

all instructions take the same number of bits, regardless of whether all of the arguments are used or not. In this way, timing, program code layout, and other considerations are preserved [16].

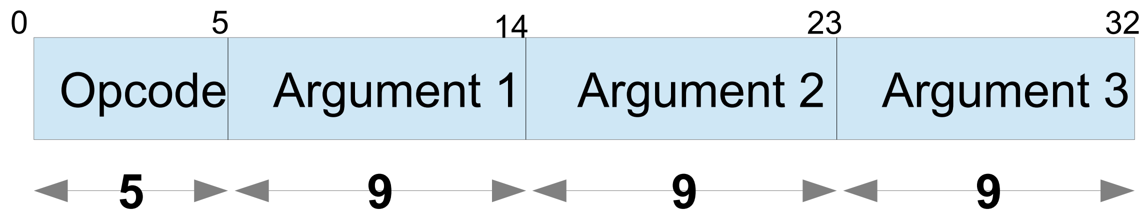


**Figure 16: Instruction Format**

The three 9 bit arguments serve different functions depending on which opcode is at the head of the instruction, however, they are mostly used to indicate the addresses of the data blocks in which the numbers to use for the current cycle are located. For example, during a read cycle, the program accepts an outside value from a peripheral, (or the testbench, during testing), and writes it into the certain memory location. Two memory addresses are necessary. One indicates the memory location to write to, and the other indicates a number from 0 to 8, one of nine possibilities. This is used to store the value in a certain location in the conceptual 3x3 matrix, (the nine memory banks), which is used throughout the program. Therefore, out of the three arguments available in the instruction, two are used. In a read cycle, the first argument are always all zeros. The second argument contains the address for the inputted values to be stored, and the third argument contains the position in the matrix in which to store the value (0-8).

Similarly, during a write, the processor takes an internal value and exports it to outside the controller. Therefore, again, only two values are necessary, the address from which to take the value to be exported, and the position in the matrix from which to take the value. In a write instruction, the third argument is always all zeros, the first argument is the address, and the second argument is the position in the matrix (0-8).

### Execution

In the case of a full run with outside hardware implemented on a physical FPGA board, the controller code will be loaded onto the FPGA, the program code will be on a memory device so that it can be fed into the controller, and outside peripherals will have to be ready and able to interface with the controller device. However, in the case of testing, optimizing and debugging the system, most operations are done through the use of software, and the excellent tools available from Xilinx to simulate the chip operation. In the ISE program, everything about the chip and the chip design can be simulated [19].

A testbench is built to the exact specifications of the unit under test. All signals leading in and out of the program are accounted for. As the program runs, the testbench is able to delay the sending of signals so that it emulates the communication that the chip might receive from a peripheral, including handshaking. Although no actual handshaking

occurs, in terms of the testbench actually acting upon signals received from the program, but a very good emulation can occur by assuming the receipt of certain handshaking signals, and returning others. From the perspective of the controller system, there is no difference, which shows our testing approach is accurate.

The testbench is written will all controller elements instantiated. Other signals used for the purposes of testing are also generated, and a block of linear code is written which emulates the changes in time for certain signals. All of this is run through the ISE tool chain, and the output is a complex waveform window which displays in time all of the signals inside and outside of the controller program.

### Operation Cycles

In the controller, the signal "pc_current" is a program counter. At each instruction cycle, the program counter is incremented. Therefore it is possible to use program counter to assess exactly where the system is in the operating process. This counter counts the actual instructions coming out of the control store. Each one of these instructions is a fully encapsulated process with the opcode, indicating the operation to undertake, and the data labels and addresses on which to read or write the data to or from. Of course, this means that not everything is completed in one cycle of the processor underlying the instruction. That is to say one instruction has more information in it than can be completed in one clock cycle. For example, for a write instruction, first the opcode would have to be decoded, then the correct operation, a write in this case, would have to be sent to the IO unit. There, the code would have to access the correct data spot in which to write the data, which is indicated by the arguments in the instruction. While all of this is happening, the IO unit would be reading the data from an outside peripheral or the testbench. Finally, the data would actually be written into the correct location.

Similarly, read or mathematical instructions require many cycles to complete the tasks that are associated with their successful completion. This is the reason that the instruction counter, pc_current, is actually synchronized with eight clock cycles. That is to say the pc_current increments every eight clock cycles.

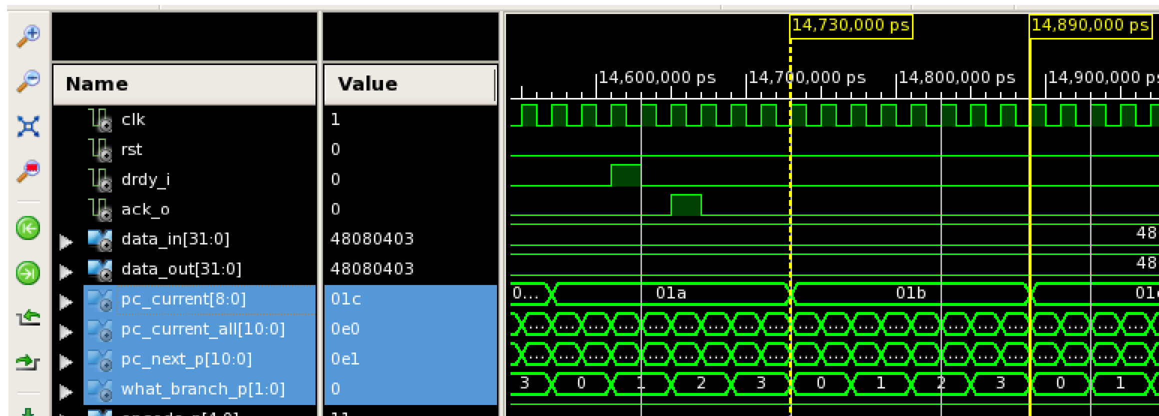


**Figure 17: Cycles Per Instruction**

In the diagram above, Figure 17, the what_branch_p line indicates which part of the execution cycle the system is in. For every pc_current cycle, the what_branch_p goes through four different phases. This is to signify that the system is in one of the following parts.

Instruction cycle sub-sections:
Every instruction is executed in four phases.

1. The **First Fetch** Cycle: The first argument (address) is used to access the first data element, if necessary (some instructions will not require this cycle, such as read and write).
2. The **Second Fetch** Cycle: The second argument is used to load the second data element.
3. The **Execute** Cycle: The two arguments, if loaded, are operated on. The operation used depends on the opcode at the beginning of the instruction.
4. The **Write Back** Cycle: The result of the operation on the two data elements is written back to the memory location indicated by the last argument in the instruction.

As noted above, some instructions do not require some parts of the instruction cycle. For example, read and write do not need the first two parts of the cycle, as they do not use data in the memory elements in the same way as say an addition cycle would. However, for processor synchronizing, ease of implementation, and stability purposes, all instructions take the same amount of time. Therefore, for the read and write cycles, some extra time is built into the instruction cycle that is not necessarily needed, but is very important for aligning up the rest of the controller system.

To conclude, each of the four sub-cycles in the instruction cycle are timed to take exactly two clock cycles. Therefore, the overall instruction cycle is exactly eight clock cycles in length. This consistent instruction cycle is by design. When the system knows beforehand exactly how long the next program cycle will be, a lot of design considerations are simplified. For example, it is known when the current instruction will end and the next one begin, therefore, the operators for the next instruction can be initialized as the current

instruction ends, for every cycle. This saves a lot of planning, time, and resources, on the part of the designer, as well as the system itself, which does not have to use resources to check whether the current instruction is done before it begins the next one. Every sub-system knows exactly when that will happen, and can begin its respective task at exactly the correct time, trusting it is synchronized with the rest of the system.

In the above diagram, Figure 17, the main clock is the first signal at the top of the diagram, clk. As the system runs for the time in between the two yellow markers, (14,730,000 ps to 14,890,000 ps) the clock has eight cycles, the pc_current completes one iteration, and what_branch_p, indicating the sub-cycle, goes through all four of it's options, as will be the case for all instructions.

### Input/Output

In a real world situation, the controller program would be implemented on an FPGA that would be connected to certain outside units. Obviously, not everything is written into the controller program. Much of the operations of the system rely on outside data. Data flows in and out of the system. This is part of what makes it so flexible and efficient. The raw data in the system could be anything. If the system is built into a navigation system, the data coming in could be from a GPS, an accelerometer, or some other device. All that this system would require would be that the data be in the form of a matrix. In this case, the matrix would be a 3x3 matrix, but the code is easily modifiable to accommodate larger matrices or different data sets.

Because of this flexibility, the program relies on the outside systems to feed in the raw data that it will use for computations. Even at the very start of the program, the actual operation code is fed into the system from an outside memory bank. This could easily be modified to update the program or to implement a different algorithm.

Therefore, since both the operational code, and the data itself come from outside the system, it is essential to have an efficient input/output block in the system. In this case, the KF_IO block is responsible for receiving and sending data. (See KF_IO). The KF_IO block is also capable of handshaking to accommodate differing types of peripherals and different speeds. (See Handshaking)

The KF_IO is the interface between peripherals and outside sources and targets. If the conditions are met, the KF_IO essentially acts as a connecting line from one side to the other, with handshaking lines alongside. It does this by actively checking at each cycle the state of the overall program. Central to the KF_IO is a case statement that cycles through each option for the opcode. The opcode is fed into the KF_IO and it decodes and acts upon each situation, including conditional and unconditional jumps. In the case of a jump, the handshaking signals are maintained as they were and changed at the next cycle depending on the next opcode.

Also, Since the I/O opcodes are decoded locally, the decoding is done at a high speed. There is no need for a central/large decoder which decodes for all of the subsystems of the program.

See “Handshaking” in Section 2.1.4 KF_IO Unit.

**Assembler: Perl Script for Program Code**

In this implementation, there is a Perl script that is used as a translator from assembly code to machine code. As has been shown above, the instructions that are loaded into the controller must be in a very specific format. They must be 32 bits long, with 5 bits at the beginning for the opcode, and the remaining 27 bits for the 3 arguments which specify addresses for memory locations. However, as we have seen, in some cases, the arguments are all used, in others they are not. Also, for some opcodes, the order of the arguments is different. That is even though read and write both use only 2 of the 3 available arguments, the blank, all zero argument is in different locations for the two opcodes.

These are all issue we would like to have removed from the mind of the user. To make it as simple as possible for the user to enter in instructions, there has been developed a Perl script to automate the process of conversion from assembly to machine code.

The input to the Perl script is a list of instructions such as the following:

| OPCODE | ARG 1 | ARG 2 |
|---|---|---|
| READ | 0 | L5 |
| READ | 1 | L5 |
| READ | 2 | L5 |

**Table 3: Assembly Read Example**

The first line of the above code simply asks the program to load in (read) a value, (from the outside or a testbench, as the case may be), and store it in the $0^{th}$ location of the L5 memory bank. The second and third lines similarly load values into the $1^{st}$ and $2^{nd}$ locations of the same memory bank. In this way, if nine READ instructions are used, the program can be loaded with an initializing 3x3 matrix. Using nine loads in such a fashion is common at the start of the program. In fact, often more than one matrix will be loaded, and for each additional matrix, the READ instruction may simply be written out nine more times per matrix.

For a write operation, as was established above, the instruction format requires very specific formatting of the arguments, which is different from the read instruction. However, the user is unaware of this complication, and simply writes in the following fashion.

| OPCODE | ARG 1 | ARG 2 |
|---|---|---|
| WRITE | L5 | 0 |
| WRITE | L5 | 1 |
| WRITE | L5 | 2 |

**Table 4: Assembly Write Example**

The above code takes the first three values from the same L5 memory bank and, one-by-one, outputs them to the peripheral (or testbench). It is very similar to the READ instruction and is very intuitive to use.

An add, multiply or a determinant operation is specified in the following way.

| OPCODE | ARG 1 | ARG 2 | ARG 3 |
|---|---|---|---|
| ADD_33 | L5 | L4 | L6 |
| MULT_33 | L6 | L7 | L8 |
| DET_33 | L4 | L5 | |

**Table 5: Assembly Arithmetic Operation Example**

Add and multiply require three memory locations. The first two matrices, (L5 and L4 in this case) are loaded, operated on, and stored in the third location specified, (L6 in this case). Multiply works the same way, and the determinant only requires two memory locations. The determinant operation takes the determinant of the first matrix and stores the result in the second memory location.

Jumps

It is also possible to specify jumps in the code in the following way.

| LABEL | OPCODE | ARG 1 | ARG 2 | ARG 3 |
|---|---|---|---|---|
| L1: | JUMP_U | L2 | | |
| | MULT_33 | L3 | L3 | L4 |
| L2: | MULT_3S | L4 | L4 | L3 |
| | JUMP_DRDY | L1 | | |
| | NOP | | | |

**Table 6: Assembly Jump Example**

In the code above, the "L1:" and "L2:" specify lines for the program to remember, called Labels. These are locations in the stream of operations that the code can jump to at any time, (when the conditions for the jump are met). In the above code, the operations on the "L1" line specifies that the program is to always jump to the L2 location, which is two

lines below the L1 line in this case. (JUMP_U is short for JUMP Unconditionally). Since it is an unconditional jump from L1 to L2, the line just under L1 is never executed.

Jumps do not have to occur on jump locations, as they do in this case, where the L1 jump location also contains a jump command. As in the line just below L2, the JUMP_DRDY command jumps to L1. However, in this case, it is not an unconditional jump. This jump will only occur if DRDY is true.

The last line of the code is a No Operation, and does nothing. It is used if the program has some need to wait between commands or to fill out an operational lineup to meet some set number of operations (for timing purposes, for example).

### Converting to machine code

Using a list of such assembly like commands it is possible for the user to write a program that he/she wishes to load onto the controller. Since the controller has strict requirements as to the format of the instructions, the Perl script is executed on the assembly code file.

./tinyasm <source_file>.asm > <output_file>.txt

The above command uses the Perl script (tinyasm) to convert the source file to the output file. The output file contains a detailed description of each instruction, indicating which argument in the instruction is what value, (for debugging purposes), and a section of lines which are direct machine code 32 bit instruction in hexadecimal format. The machine code is then directly loaded into the controller program using the testbench or an outside peripheral memory source.

## 2.4 Implementation

The controller is organized into a group of modules instantiated under the overall top module, the "Controller," located in the CONTROLLER.vhd file. Under the controller is the loader_unit (LOADER.vhd), control_store (RAM_512x32_2Port_SC.vhd), branch_unit (KF_BRANCH_CC.vhd), KF_IO_unit (KF_IO_CC.vhd), data_store_11 to data_store_33 (RAM_512x32_SC.vhd), and KF_ALU (KF_ALU_CC.vhd).

KF_ALU itself contains sub-modules: MATRIX_ADD_3x3_CC_unit_01 (MATRIX_ADD_3x3_CC.vhd), MATRIX_MULT_3x3_3x1_CC_unit_02 (MATRIX_MULT_3x3_3x1_CC.vhd), MATRIX_MULT_3x3_3x3_CC_unit_03 (MATRIX_MULT_3x3_3x3_CC.vhd), MATRIX_DET_3x3_CC_unit_04 (MATRIX_DET_3x3_CC.vhd), MATRIX_MULT_3x3_1_CC_unit_05 (MATRIX_MULT_3x3_1_CC.vhd).

All of the above modules plus individual testbenches for most of the modules are incorporated into the overall top level testing unit, the controller testbench, or CONTROLLER_TB (CONTROLLER_TB.vhd).

See Figure 2: Controller Block Diagram on Page 11, above.

### Compiling

Compiling of the program is completed through the program on which the FPGA is emulated, and all of the HDL code is written, as well as the testbench. The program used in this case the the Xilinx tool, ISE Project Navigator, in the ISE Design suite, which is available as a free download from Xilinx [20].

In ISE, the controller is the main component, and it contains within it all other modules. One step above the controller is the overall testbench. Since the testbench instantiates the controller and then uses other new signals to emulate the world outside the controller, it is a step up in the hierarchy. The testbench is run and in the testbench is the original user given instructions that are to be used to program the controller. In this case, the test signals, and all of the testing code is hardcoded into the testbench. This is the code that is the output of the Perl script which generates machine code for the controller to use. This is simply copied and pasted into the testbench due to the Perl script simplifying the writing of the instructions, thus making it a part of the testbench testing system.

The simulation is done is iSim, another Xilinx product associated with the ISE Tools package.

### Debugging

### Evaluation of Code

The code was extensively tested. The code was split into modules and testbenches were developed to input the appropriate inputs to that system and to collect the respective outputs. Waveforms and loops were used to input a wide variety of input signals and the outputs at each step were examined. (See verification waveforms)

### Testbench

In order to test the system, a specially designed testbench was created. The testbench serves three purposes. First of all, the testbench must test each module to ensure proper functioning. Then, the communication between modules is tested. This is achieved by loading each module and combining them to form the unit under test (UUT). Finally, the module as a whole is tested, including the signals in and out of the module. Therefore, the testbench must be able to emulate outside influences on the UUT.

#### Test each module

There is a testbench for each module. It must be able to provide the signals that the module is expected to receive, and also be able to receive signals the the module will send. For example, in the matrix determinant test, the test bench loads different signals onto the determinant module, and the output is checked for consistency. In this way, even if just one case is incorrect, the test bench is able to find it before the error cascades onto other areas of the system, making the error source that much more difficult to locate. Most of the testing was done by hand, but for larger cases, in VHDL, loops and "ensure" statements are also available for quicker verification. The ensure statement is an automated  way to check, or "ensure" that the output of a test matches a predetermined value. In this case, the system was tested for consistency by entering in known inputs and outputs and determining whether the output of the system is consistent with calculated results. This was done at various points in the processing chain, thereby making it easier to determine the exact location of inconsistencies, if/when any were present.

Each module has been thoroughly tested and verified.

#### Test inter-module signals and the overall system

All of the individual modules are loaded and arranged in the correct hierarchy and structure. There are signals that travel from the outside to into a specific module, or to a combination of modules. Thus, each module often has inter-dependencies that range from outside signals to inter-module signals and combinations of such signals.

A crucial part of the overall test is the generation of outside signals. The controller must be made to feel as if it is communicating with an outside peripheral, as it is designed to do. Since this is done all in software, on the ISE Tools system, there are not yet actual outside peripherals that can be used. Even if there were such peripherals available, it would be premature to hook up the system to them at this stage. Therefore, it is highly

convenient to be able to emulate such peripherals through the use of signals generated by the testbench.

In the eyes of the system, it is operating in its normal way, as it would if it were out in the field or in a complex real world system. However, since this system is at this point completely simulated, it is the testbenches responsibility to completely simulate everything that the controller system would see in the "real world."

All data going into the controller is generated by the testbench. For example, the bank of program data, which as stated above is routed to the control store memory stack with the assistance of the loader unit, is generated by the user and written into the test procedure of the testbench. It is as if the program has initialized, and the outside memory, say an external flash drive, is feeding in the program data to the FPGA. The controller, in how it operates or runs, sees no difference between data it would get from a peripheral, or data that is loaded by the testbench.

The actual data that is loaded by the testbench is generated by the accompanying Perl script. (See Perl script)

### Number Representation

The numbers in this system are, by design, 8 bits in length, i.e., the word length for the system is 8 bits. The opcode is 5 bits, addresses are 9 bits, and the control code is 32 bits. Most of these lengths are not manipulated or altered in any way as to disrupt the length considerations. However, the numbers that represent the operational matrices are almost constantly manipulated. How does the act of adding, multiplying, finding the determinant, etc. effect the numerical representations? And, what can one do to maintain consistency and understandability of the numbers, which, in the end, are the raw data of the operation, and represent the results of the entire system?

This section explores such important issues.

First of all, one must decide in which way to represent the numbers at hand. There are two main choices, each which brings its own opportunities and challenges. The two methods to represent the raw data are fixed point and floating point. Deciding on one of the above is like coming to a fork in the road and deciding weather to go left or right. All subsequent decisions are a result of choosing the direction at this juncture. So it pays to understand the pros and cons of each option [16].

Fixed point

Fixed point is where the radix point of each number is fixed and known throughout the operation. This is several benefits and shortcomings.

Benefits of fixed point system

In fixed point there is no need to shift the numbers and line up each number in order to perform operations such as addition or subtraction on it. For example, if 3.14 and 2.71 are stored in two different data blocks, the additions is straightforward and one to one on each digit, as expected. However, if the 3.14 is actually $3.14 \times 10^{4}$ and 2.71 is actually $2.71 \times 10^{-3}$ then the numbers cannot be added as they are. The 2.71 will have to be shifted down three spots to 0.00271 and the 3.14 will need to be shifted to 3140. Therefore the result will be 3140.00271, not 5850 or 0.005850. Therefore, for addition and subtraction, fixed point is preferred. However, multiplication is a different story. Note that the above example is in base 10, however, the system itself runs in base 2. The argument carries over to any base, and is adjusted accordingly.

Issues with fixed point system

There is one major consideration to make before the use of any numerical system, and fixed point is no exception. On the processor, the numbers are a certain finite width. In this case, the numbers are 8 bits in width. This means that from the processor's viewpoint, it is dealing with at most a number as large as $2^{8}$, if you are dealing with unsigned numbers, or if signed, the range of numbers for the processor to deal with is positive $2^{7}$ to negative $(2^{7})-1$, which would be 128 to -127. This is a range of just 256. It may be possible that the only numbers used are in this range, but when you start to get into operations such at addition, multiplication, determinants, etc, all performed in loops that run indefinitely with input from outside sources, it quickly become apparent that this range is not sufficient to cover all operational scenarios. Therefore, even before entering in a single number into the data base, it must be decided, what range is necessary and sufficient for this program, and at what precision. These are the two main issues when considering what numerical system to deploy, fixed or floating point, for as will be shown, one is better for the former and the other, the latter [21].

For example, [21] and [22] use floating-point, which results in a complex system with large amounts of swapping and rotating of operands. [23] claims to use fixed-point but does not discuss accuracy or precision of results.

If fixed point is used, it can be scaled to any range. However, if the range is too large, you end up with large jumps between subsequent digits. That is to say, if you have 3 logical steps in a range, the numbers being the largest, middle and lowest points in the range, if you scale it to from 0 to 100, you have jumps of 50 in between blocks. Whereas if you have a range of 0 to 10, one must only jump from 0 to 5 and 5 to 10. Of course, you loose the possibility of representing all of the numbers from 11 to 100.

The precision issue is similar to quantization of an analog signal when sampling, as there are only certain points in the logical scale where the number can map to.

In this case, the $2^{8}$ or $2^{7}$ were assumed to be sufficient to represent the scale of the project. If it seemed that they were not, then it was always a possibility to make the numbers larger in width, which is the beauty of designing the system from the ground up [23].

Therefore, the issue now was to decide the range that would be used to scale into and out of the program before using the 8 bit operations.

Method Used

In this case, the method that is implemented is a fixed point numerical system. This is due to the fact that it is simpler and more robust to design, build and verify a system based on fixed point. In floating point a lot of shifting and matching of numbers is needed [22]. This is expensive from a computational point of view. Also in this case, the advantages that floating point offers are not necessary if a well thought out numerical scale is used.

# Chapter 3: Kalman Filter

## 3.1 Intro

The Kalman filter is a linear recursive algorithm that solves the discrete data linear filtering problem. It uses a series of noisy measurement inputs observed over time to estimate state variables, which make up the system in question. The estimates are statistically optimal in the mean square error sense, and tend to be more precise than an estimate based on a single measurement alone [24].

The Kalman filter has found success in many diverse applications, due to its flexibility and robust nature. In this project we attempt to provide a platform for implementation on the FPGA. In Chapter 2, above, it was shown how the controller developed was ideal for certain matrix operations and other functions. These primitive operations were chosen after studying the Kalman filter and determining the instruction-set that would be provided in an ideal system to make the development of a Kalman filter application as easy as possible. By predetermining the instruction-set needed, the controller was then designed and implemented to best handle these operations.

Due to the wide range of applications for the Kalman filter, this system had to be as flexible as possible, which is the reason for some of the decisions that were made above in terms of implementation details. For example, in this system, many fundamental areas are easily modifiable, by changing certain variables or interfacing signals. The wordlength can be increased or decreased to suit the application in question. The RAM can be increased in width and/or size. Due to the redundancy built into the system, the system functions can be expanded, and the slack in the number of opcodes allowed versus the number used is easily able to accommodate the expansion. The following chapter discusses some of the reasons for those decisions with a view to the fundamentals of the Kalman filter, and how it performs its functions.

The Kalman filter that is implemented in this system is a non-adaptive filter. This means that at the beginning of the operation of the program, the coefficients that will be used for the filter are known. They are known beforehand, and do not change for the duration of the program run. Therefore, the first instructions in any program instruction set when the program is loaded by the loader and run by the control store must be read operations. These read operations are to load the Kalman filter coefficients into the data banks of the system. These coefficients serve to define the filter. The benefit of this approach is that if a new filter is developed, or a new strategy is to be implemented, the program can easily be adapted and/or updated. It will still have to read in the coefficients at the start of the program, however, instead of one set of coefficients, another set can be fed into the system, thereby possibly completely changing the operation of the system. It is as easy as that [25].

For flexibility and usability reasons, the main form of data is a matrix in the 3x3 form. All of the data, as discussed in the above sections on the Controller, is stored in a memory bank of nine locations which make up the 3x3 matrix, and can be accessed in parallel, due to the benefits, i.e., inherent parallelism, of the FPGA system [26].

Since each matrix is a 3x3 matrix, if we have four sets of coefficients to read in to define the filter, we need only 36 read instructions at the start of the program. In this system, there are 512 memory locations to store the program. Therefore after the 36 read instructions, there are still 512-36=476 lines of code that can be written to form the body of the program to run. This is only 7% of the program to load the filter, which is central to the operation of the program. This is a very efficient ratio, and leaves plenty of room for other steps in the program code.

### Adaptive vs. Non-adaptive

In this case, the system models are static. For this reason, as shown above, the filter is loaded with only 7% of the code space having been used. If an adaptive filter was desired, there would need to be a lot more overhead, and possibly an operating system needed in order to load new state matrices at run-time. That filter would then have to be optimized, the coefficients could have to be altered and modified, which would have to be done while the program was on hold. It is a complicated process, and this system does not deal with it. In the adaptive case, the 512 memory locations for the code would most likely be insufficient. In the non-adaptive case, as we have seen, 512 memory locations if far more than enough, and the best part is that if a new filter is required for a new problem, it can simply be loaded at boot time through the initializing read instructions.

## 3.2 Theory

In this section, we discuss the theory behind the operation of the Kalman filter algorithm. The section begins with a general discussion of stochastic estimation, and the observer design problem, which provides the impetus to develop an efficient implementation such as the Kalman filter to tackle the challenges associated with this common situation. The general case of the observer design problem is one which is found in many fields, and reinforces the reason why the Kalman filter is found in such diverse applications and areas. [27]

After a discussion of the theory of the Kalman filter, the actual equations that are used to implement the filter are presented. The section ends with a diagram highlighting the main equations, and the order in which they are deployed, within the feedback loop that forms the backbone of this algorithm.

### Stochastic Estimation

The dynamic process described by an n-th order difference equation is [28]:

$$y_{i+1}=a_0 y_i+\cdots+a_{n-1} y_{i-n+1}+u_i, i\geq 0 \tag{1}$$

where $u_i$ is a zero-mean white random "noise" process with auto-correlation

$$E(u_n, u_{n+d}) = Q_n \delta_{n,n+d} \quad , \quad \delta_{i,j}=\{\begin{matrix} 0, & i=j \\ 1, & i\neq j \end{matrix}\} \tag{2}$$

The initial values $\{y_{0,} y_{-1,} \cdots, y_{-n+1}\}$ are zero-mean random variables with a known n x n covariance matrix

$$P_0 = E(y_{-j}, y_{-k}), j, k\in\{0, n-1\} \tag{3}$$

Also assume

$$E(u_i, y_i) = 0 \textit{ for } -n+1\leq j\leq 0 \textit{ and } i\geq 0 \tag{4}$$

so that

$$E(u_i, y_i) = 0, \ i\geq j\geq 0 \tag{5}$$

That is, the noise is statistically independent from the process to be estimated.

The difference equation can be re-written in matrix form as:

$$\underbrace{\begin{bmatrix} y_n \\ y_{n-1} \\ \vdots \\ y_{n-m+1} \end{bmatrix}}_{X_n} = \underbrace{\begin{bmatrix} a_0 & a_1 & \cdots & a_{m-2} & a_{m-1} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} y_{n-1} \\ y_{n-2} \\ \vdots \\ y_{n-m} \end{bmatrix}}_{X_{n-i}} + \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{W_n} u_n \quad (6)$$

The state space model, (dynamic equation or system evolution) is:

$$X_n = Ax_{n-1} + W_n$$

and the measurement equation is

$$Z_u = Y_u + V_u = H_u X_u + V_u, H_n = [1 \;\; 0 \;\; 0 \;\; \cdots \;\; 0] \quad (7)$$

$Z_n$ is the measurement at time n, where $V_n$ is the measurement noise. $V_n$ is a white Gaussian process of the mean; $E[V_n]=0$ , $Cov[V_n]=Q_n$ , and $E[V_n \;\; V_n+d]=0$ .

One very general case in control theory which applies the above concepts and is a useful problem to try to solve is the general case of the observer design problem. In the following section, the observer design problem is introduced and forms the basis from which the usefulness of the Kalman filter is discussed [27].

### Observer Design Problem

The "Observer," or "State Observer" as it is referred to in control theory is the name for a dynamic system $\hat{S}$ , whose purpose is to estimate the state of another dynamic system, $S$ , using only the measured input and output of the latter. Since the internals of a system are in most cases not visible to the outside world, it can be thought of as a "Black box" problem [29].

During the operation of the state observer, the observer calculates the "residual," which is the difference between the measured output and the corresponding output generated by the model of $S$ . The residual is multiplied by a "gain" and then used as an input to a model of $S$ . As long as the gain is chosen appropriately, the system will be an asymptotically stable dynamic system, and the error will converge to zero.

The observer gain is optimized for the noise input to $S$ and the sensor(s). In further detail, it is worked out as shown below.

There is typically a process model that models the transformation of the process state. represented as a linear stochastic difference equation similar to equation (7)

$$X_n = Ax_{n-1} + W_n \quad (8)$$

Also a measurement model that describes the relationship between the process state and the measurements, similar to above

$$Z_n = Y_n + V_n = H_n X_n + V_n \tag{9}$$

$w_k$ and $V_n$ are random vectors representing the process and measurement noise respectively. Use of $Z_n$ shows that measurements do not have to be measurements of the state specifically, but any linear combination of the state elements [30].

There is noise associated with each measurement. The sensors operate in the physical realm, and there are certain physical limitations that each sensor deals with, whether it be due to temperature, pressure, humidity, human error, etc. Even in a perfect situation, the sensor operates in certain bounded operating zones, the limit of which may be reached. There are also always random electrical noise signals added due to circuits surrounding the sensors. The signal to noise ratio is something that must be accounted for, and it adds some measure of uncertainty to all measurements.

Like measurements, process models are also not perfect. Some measure of randomness or uncertainty must be accounted for. The actual state transform model is unknown. Therefore ongoing measures of the state are analyzed and qualified as they are combined with the measurements to form overall system estimates.

One of the best ways to combine such noisy data from measurements and models is through the use of the Kalman Filter. The Kalman filter is tuned through the Kalman Gain during operation to the amount of noise present. This is one example of why the Kalman filter is as popular as it is, and one of the reasons we chose to use it.

**Kalman Filter**

The Kalman filter is used to solve the general problem of trying to estimate the state $x \in \Re^n$ of a discrete time control process that is represented by the linear stochastic difference equation [27]

$$X_u = A x_{u-1} + W_u \tag{10}$$

With a measurement $z \in \Re^m$

$$Z_u = H_u X_u + V_u \tag{11}$$

$V_u$ and $W_u$ are assumed to be random vector processes that are independent of each other, white and with normal probability distributions [31]

$$P(W_n) \sim N(0, Q_u) \tag{12}$$

$$P(V_u) \sim N(0, R_n) \tag{13}$$

The n x n matrix A defines the relationship between the state at n-1 time and n time, that is, it relates the previous step with the current. The m x n matrix H in (11) relates the state to the measurement $Z_n$ . All of these matrices may be dynamic and change from step to step but here we assume they remain constant [24].

Define $\hat{x}_n^- \in \Re^N$ to be the *a priori* state estimate at step n given knowledge of the previous steps and $\hat{x}_n \in \Re^N$ to be the *a posteriori* state estimate given the measurement $Z_n$ . Then the *a priori* and *a posteriori* estimate errors are

$$e_n^- \equiv x_n - \hat{x}_n^- \tag{14}$$

and

$$e_n \equiv x_n - \hat{x}_n \tag{15}$$

The *a priori* estimate error covariance is then

$$P_n^- = E[e_n^- e_n^{-T}] \tag{16}$$

and the *a posteriori* estimate error covariance is

$$P_n = E[e_n e_n^T] \tag{17}$$

Now, to derive the Kalman Filter equations, we look for an equation that computes an *a posteriori* state estimate, $\hat{x}_n$ as a linear combination of an *a priori* estimate $\hat{x}_n^-$ and a weighted difference between an actual measurement $z_n$ and a measurement prediction $H\hat{x}_n^-$ as follows

$$\hat{x}_n = \hat{x}_n^- + K(z_n - H\hat{x}_n^-) \tag{18}$$

Where $(z_n - H\hat{x}_n^-)$ is known as the *innovation sequence* or the *residual.* The residual is the difference between the measurement and the measurement prediction. If the residual is zero then the filter has predicted perfectly for that step.

In the above equation (18), K is an N x M matrix that is the *Kalman gain* or *blending factor* that minimizes the *a posteriori* error covariance in equation (17). After substitution and reduction, K is given as follows [27].

$$K_n = P_n^- H^T (H P_n^- H^T + R_n)^{-1} \tag{19}$$

By observation, one can see that as the measurement error covariance approaches zero, the gain K weighs the residual H more heavily:

$$\lim_{R_n \to 0} K_n = H^{-1} \quad , \tag{20}$$

if H is invertible. In the opposite case, as the *a priori* estimate error covariance $P_n^-$ approaches zero, the gain K weighs the residual less heavily:

$$\lim_{P_n^- \to 0} K_n = 0 \tag{21}$$

In other words, as the measurement error covariance R approaches zero, the actual measurement $Z_n$ is trusted more and more, and the predicted measurement $H\hat{x}_n^-$ predicted state is trusted less. Inversely, if the *a priori* estimate error covariance $P_k^-$ gets smaller, the actual measurement $Z_n$ is trusted less, and the predicted measurement $H\hat{x}_k^-$ (predicted state) is trusted more.

Using the above analysis the following section illustrates in detail the exact equations that the Kalman filter uses to obtain its estimates.

**The Kalman Filter Equations**

The Kalman filter is essentially a feedback estimation system. There are two steps that the Kalman filter takes. The first is to calculate a prediction of the state. Next, feedback is given to the filter in the form of measurements, which are of course noisy and not perfect. Therefore, the filter is split into two parts: the time predictions and the measurement updates. Each of these two steps has its own respective filter equations which are used in the filter process.

The time update equations project forward into time. Therefore *a priori* estimates are computed for the state and error covariance with the time update equations for the next time step. The feedback from the measurement updates is used to incorporate the current measurements into the *a priori* estimates to obtain an improved *a posteriori* estimate.

The two distinct sides of the Kalman filter, (the time prediction and the measurement update equations) are a form of a predictor-corrector algorithm. In this way a feedback cycle is obtained which is able to use only the previous step to improve the next step, and continuously get closer to a more accurate state than would be possible with only the prediction capabilities, or only the measurement instruments.

The feedback cycle for the Kalman filter is shown below.

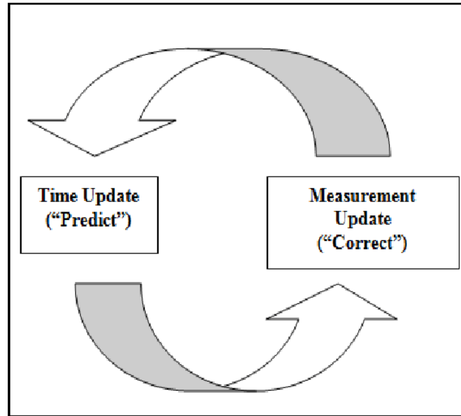


**Figure 18: Feedback Cycle of the Kalman Filter**

The general equations used for each of the time and measurement update sections are given below [24].

$$\hat{x}_n^- = A\hat{x}_{n-1} \tag{22}$$

$$P_n^- = AP_{n-1}A^T + Q_n \tag{23}$$

Notice that the time update equations in the above table use the estimates from the "k-1" step to compute the "n" step. They are using the previous step to predict the state in the current step. Matrix A is from equation (10) while matrix $Q_n$ is from equation (12).

The Kalman filter measurement update equations are given below.

$$K_n = P_n^- H^T (H P_n^- H^T + R_n)^{-1} \tag{24}$$

$$\hat{x}_n = \hat{x}_n^- + K_k (z_n - H\hat{x}_n^-) \tag{25}$$

$$P_n = (I - K_n H) P_n^- \tag{26}$$

Incorporating the equations from the above two tables and the overall Kalman filter high level diagram above, gives us the following complete Kalman filter equations diagram below.

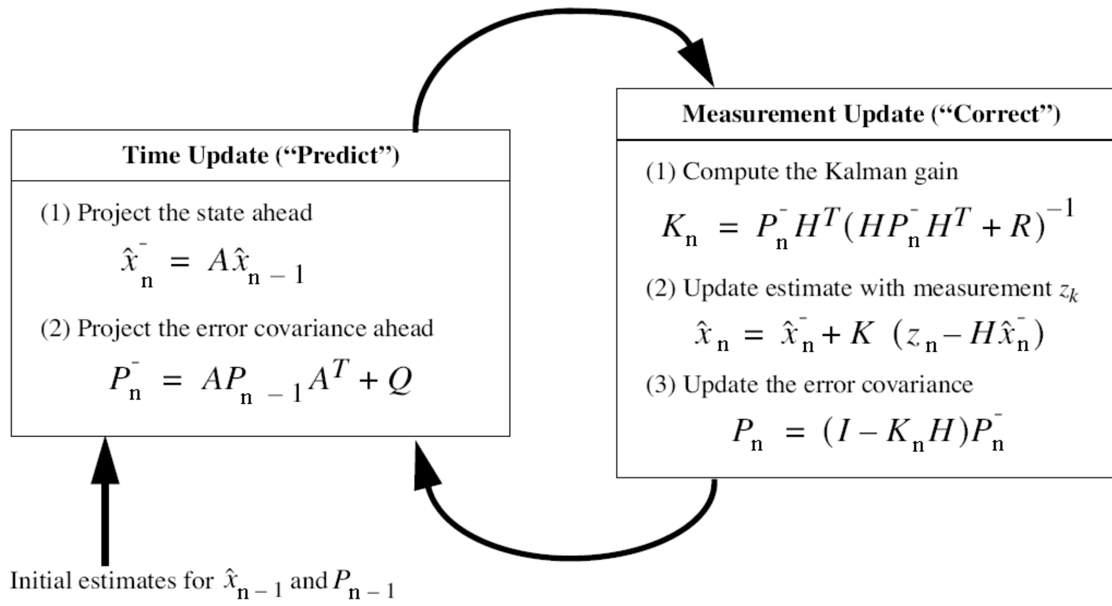


**Figure 19: Complete Kalman Filter Equations Diagram**

### 3.3 Kalman Filter Implementation

The linear system discussed above, can be graphically represented in the following form:

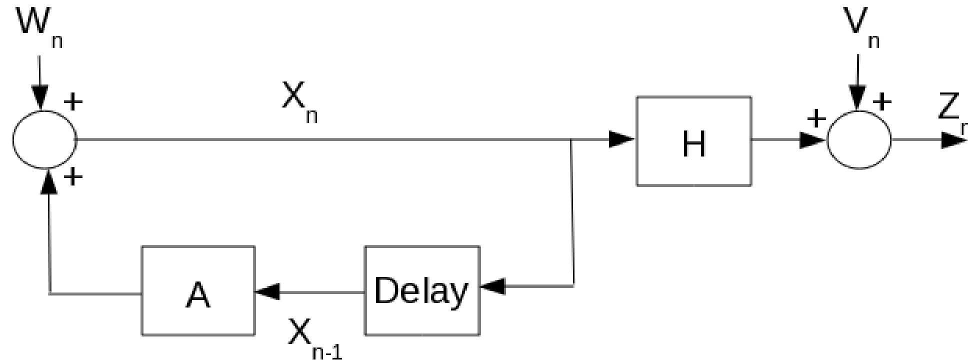


**Figure 20: Discrete Time Linear System [16]**

As mentioned above, the measurements may not be direct measurements of the internal states of the system. The measurements can be linear combinations of the internal states, along with the changes depicted on the diagram above, as follows:

$$Z_n = HX_n + V_n \quad , \tag{27}$$

where H is the N x M measurement matrix and V is an N x 1 matrix representing the measurement noise. $X_n$ is a true state, and $Z_n$ is a true measurement.

#### Stages of Kalman filter

In order to run the Kalman filter, the above Kalman filter equations, the inputs, and the outputs from them are required. The run can be decomposed into a number of stages, as follows [16].

1. State prediction

If we know the estimated state at time n-1, since we have all of the measurements up to and including time n-1, $X_{n-1}$, what is the predicted state at time n, without the measurements at time n, $X_{n|n-1}$?

$$\hat{\hat{X}}_{n|n-1} = A\,\hat{\hat{X}}_{n-1|n-1} \tag{28}$$

The result, $\hat{\hat{X}}_{n-1|n-1}$ is still an estimated state, the true state can only be known with zero-error measurements and with a true initial state.

2. Error Calculation

Since the above, equation (27), is an estimate, and is not the true value, there is a certain error associated with it. We calculate the covariance of that error when predicting the state at time n given measurements up to and including time n-1:

$$P_{n|n-1} = Cov(\hat{\hat{X}}_{n|n-1} - X_n) = AP_{n-1|n-1} A^H + Q \tag{29}$$

where $X_n$ is the true state at time n.

3. Predict the measurements

Now to predict the measurements at time n given the estimated state, (in fact, the mean of estimated state) based on measurements up to and including n-1:

$$\hat{Z}_n = H \hat{\hat{X}}_{n|n-1} \tag{30}$$

4. Error Calculation

Again, since the above is an estimate and a prediction, an error is associated with Equation (30). This error is called the measurement prediction error, and is given by the covariance matrix $M_n$. $M_n$ depends only on what is happening at time n.

$$M_n = Cov(\hat{Z}_n - Z_n) = HP_{n|n-1} H^H + R \quad , \tag{31}$$

where $Z_n$ is true measurement at time n. Equation (31) shows how much spread there is in estimating $Z_n$, and therefore shows how precisely the measurements can be predicted.

5. Innovation

The innovation at time n is given by $N_n$:

$$N_n = Z_n - \hat{Z}_n \tag{32}$$

6. Kalman gain

The Kalman gain, K, which is a M x N matrix is given by:

$$K = P_{n|n-1} H^H M_n^{-1} \tag{33}$$

If H = I, then a noisy measurement of the state is obtained directly, (see Equation (27))

$$Z_n = X_{n-1} + V_n \tag{34}$$

That is, we measure the state directly (see Equation (30))

$$\hat{Z}_n = \hat{\hat{X}}_{n|n-1} \tag{35}$$

When H = $I$, and of the uni-dimensional case, the Kalman gain can be regarded as being

$$K = \frac{state\ prediction\ error}{state\ prediction\ error + measurement\ error} \tag{36}$$

If the measurement noise is large, the Kalman gain, K --> 0. Thus, the Kalman filter model is trusted more and the measurement less, since the measurement is not very accurate.

If the measurement noise is small, then the opposite is true: K --> 1, and the measurements are more valuable. In this case, a lot of information is obtained from measurements. This is demonstrated in Equation (37).

7. Updates

Update the mean of the estimated state:

$$\hat{\hat{X}}_{n|n} = \hat{\hat{X}}_{n|n-1} + KN_n = A\hat{\hat{X}}_{n-1|n-1} + KN_n \tag{37}$$

Update the covariance of the estimated state:

$$P_{n|n} = (I - KH) P_{n|n-1} \tag{38}$$

$P_{n|n}$ is a matrix which gives the covariance of the error of the state estimation done in Equation (37). Equations (37) and (38) are not predictions any more; they are only estimations (thus the “hat” symbol).

Through the use of the Kalman filter, the discrete linear system presented above can be modified. Though the discrete system is included in the Kalman filter system, it serves as one half of the innovation computation section, from which the other half is subtracted to find $N_n$, the innovation. The updated system is as follows.

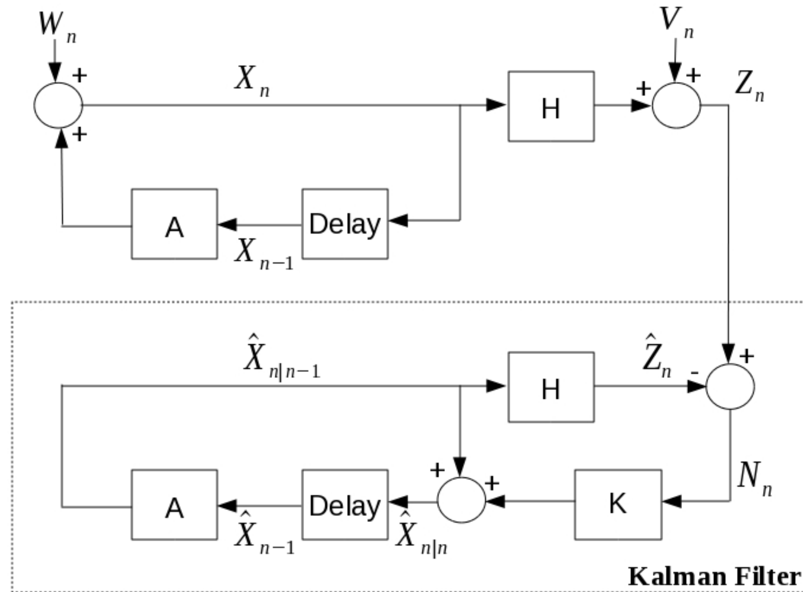


**Figure 21: Discrete-Time Linear System -- Kalman Predictor [16]**

In the next section, we determine the exact number of computations required per step. This is an important consideration as it helps us determine the amount of computing resources that will be required from the system in each iteration of the Kalman filter.

### Computations Required Per Step

In each of the stages above, there are certain computations undertaken from one step to the next. When programing this algorithm onto a hardware system, it is very important to know exactly how many steps each computation will require. In other words, how many clock cycles can be expected to pass during each stage. Once known, this number should stay the same despite any change in the actual numerical values of states, as the computational engine will be consistent in each iteration [32].

These values are crucial from a timing perspective, as once timing for each stage is computed and set, the entire system can be optimized to ensure the greatest efficiency overall. The following is a computation of the number of clock cycles each stage in the Kalman filter algorithm as it is presented above will require.

Step 1. State prediction.

In this first stage, the previous state is used to find the next state estimate using Equation (28): $\hat{X}_{n|n-1} = A\hat{X}_{n-1|n-1}$ . This requires a matrix multiplication to go from $X_{n-1|n-1}$ to $X_{n|n-1}$.

**Computations required:** One matrix multiplication.

Step 2. Error Calculation.

The error is calculated using Equation (29):

$P_{n|n-1}=Cov(\hat{\hat{X}}_{n|n-1}-X_n)=AP_{n-1|n-1}A^H+Q$

**Computations required:** This requires 2 matrix multiplications, a matrix addition and a transposition of matrix A.

Step 3. Predict the measurements.

In this step Equation (30) is used: $\hat{Z}_n=H\hat{\hat{X}}_{n|n-1}$ , which is again a matrix multiplication.

**Computations required:** One matrix multiplication

Step 4. Error Calculation

The error is calculated in this step using Equation (31): $M_n=Cov(\hat{Z}_n-Z_n)=HP_{n|n-1}H^H+R$ . This is similar to the previous error calculation and requires the same number of computations.

**Computations required:** 2 matrix multiplications, a matrix addition and a transposition of A.

Step 5: Innovation

The innovation is computed using Equation (32): $N_n=Z_n-\hat{Z}_n$ which is a direct subtraction.

**Computations required:** One matrix addition.

Step 6: Kalman gain calculation

The Kalman gain is the N x M matrix which tells us the relative precision of the measurements and the predictions. It is calculated using equation (33): $K=P_{n|n-1}H^H M_n^{-1}$ . This step is a bit more complex as it requires a matrix inversion [33]. Other than that it also requires two multiplications and a transposition.

**Computations required:** One matrix inversion, two matrix multiplications, one transposition.

Step 7: Updates

In this stage, the mean and the covariance of the estimated states are updated using Equation (37) and (38) which are given here again for reference: $\hat{\hat{X}}_{n|n}=\hat{\hat{X}}_{n|n-1}+KN_n=A\hat{\hat{X}}_{n-1|n-1}+KN_n$ and $P_{n|n}=(I-KH)P_{n|n-1}$ .

The computations are again multiplications and additions.

**Computations required:** (37): Two matrix multiplications and one matrix addition. (38): One matrix addition and one matrix multiplication.

The following is a table summarizing the above computational analysis.

| Step | Multiplications | Additions | Transpositions | Inversions |
|---|---|---|---|---|
| 1. Prediction | 1 | | | |
| 2. Error | 2 | 1 | 1 | |
| 3. Prediction | 1 | | | |
| 4. Error | 2 | 1 | 1 | |
| 5. Innovation | | 1 | | |
| 6. K, Gain | 2 | | 1 | 1 |
| 7. Updates | 3 | 2 | | |
| **Total** | **11** | **5** | **3** | **1** |

**Table 7: Computations Required for Each Step in KF**

### Design Considerations

Looking at the above totals in Table 7, it is clear that any machine or system that is to implement the Kalman filter algorithm in real time must have an instruction set oriented to matrix computations, such as matrix multiplications, matrix additions, etc. Transpositions are also somewhat important, and a good inversion would be a nice bonus [34]. For example in [18] the author maps the matrix inversion to a systolic array. The Kalman filter is a system with feedback, thus the latency of matrix inversion (and all the other operations) is of paramount importance (so, a systolic array does not provide a significant speed-up, if any). [34] uses Gaussian elimination, which is known to be unstable under certain conditions. Gaussian elimination can be made stable if pivoting (that is, row swapping) is employed, but this increases the traffic with memory or require if-then-else operations. QR decomposition is a much better option.

All of the above totals are for just one iteration, therefore the main task for the implementing machine is to focus on the matrix multiplication subsystem, which will be called upon an outstanding 11 times per iteration.

It is through taking into account such considerations that the controller system is built. The controller has in it a very capable multiplication/addition system. All of the matrix operations, matrix multiplication, addition, multiplication by a constant, etc, are given full hardware support. That is, all of the matrix operations are performed in hardware and in parallel. For example, to operate on a nine element matrix in hardware will take the same amount of time as one element, where as in software, it would take nine times as much time.

The matrix multiplier and adder are able to fully complete their respective operations, which include reading two matrices, executing, and writing the matrix result back to memory, in one instruction cycle. In this way synchronization is possible and timing can be computed without regard to the complexity of the computations to be completed. In this case there are 11 multiplications and 5 additions required per iteration, but even if the

numbers were different, say 15 multiplications and 1 addition, the time required for completion would be the same; in both cases there are 16 operations, and due to the hardware design of the adder and multiplier, the sum of operations in these two examples would both be 16 instruction cycles.

**MATLAB Routine for Testing Kalman Filter Operation**

The Kalman filter was extensively tested in MATLAB. The following is an illustration of the way that the Kalman filter works. In the following example, a script is written in MATLAB for the Kalman filter operation and results are analyzed. This allows us to get a better feel for the functionality of the Kalman filter, and allows us to modify and iteratively improve upon its metrics [35].

In the MATLAB example, the most simplest case is taken to determine if the Kalman filter is useful and able to hone in on the correct value, how fast, and how accurately. During the filtering, key variables are extracted and analyzed.

For this illustration, a sample signal is chosen. This signal could be a voltage, speed, position, etc. It could be any signal that is being measured by some measurement equipment and being used to signify a state in a system. Notice that it is signifying a state directly. This is for added simplicity. Since it is a measurement of the state directly, the "H" in Figure 21 above can be taken as a "1".

In another attempt to further simplify things, the states in this case is assumed to be constant. This implies that the "A" in the above figure is also "1".

With the above assumptions in place, the Kalman filter equations are programmed into a MATLAB script (See Appendix), and the resulting code is run.

During the coding, other assumptions are made. A small process variance is assumed, so Q = 1e-5. A seed value of "0" is given to the code, that is $\hat{X}_{n-1|n-1}=0$ . The initial value for $P_{n-1|n-1}$ or $P_0$ is chosen to be "1". If the initial state for $\hat{X}_{n-1|n-1}$ was known with 100% certainty then one could let $P_0=0$ . However, then the filter output would not change value at all. Why change when one is sure that the value is correct? Any value for $P_0$ not equal to zero would work. The filter would eventually converge. Here, $P_0=1$ .

A random scalar value is chosen as the target for estimation, z = -.25. Again, this could be any measured value. The key to remember is that no matter what is being measured, it is being measured by an imperfect measuring tool. A Gaussian white noise of .1 RMS units is chosen to corrupt the ideal signal. This results in values from the measurement tool which vary from near 0 to near -0.5, which is a large range in any situation, but especially if an accurate value is required [36].

As will be shown below, if a simple moving average is used, the signal is seen to be varying wildly, when in fact it is constant.

In the first simulation, the correct value for R, $R = (0.1)^2 = 0.01$ is used. This is due to the noise having an RMS value of 0.1. Since this is the "true" value for the variance, the best results are expected from the Kalman filter. The results of this simulation are below.

In the following figure, the true value of -0.25 is represented by a solid black line. The Kalman filter estimate is the blue line, the black plus signs are the actual measurement values that the filter sees, and the pink line is a moving average of the measurement values, without the use of the Kalman filter. The performance difference is readily apparent.

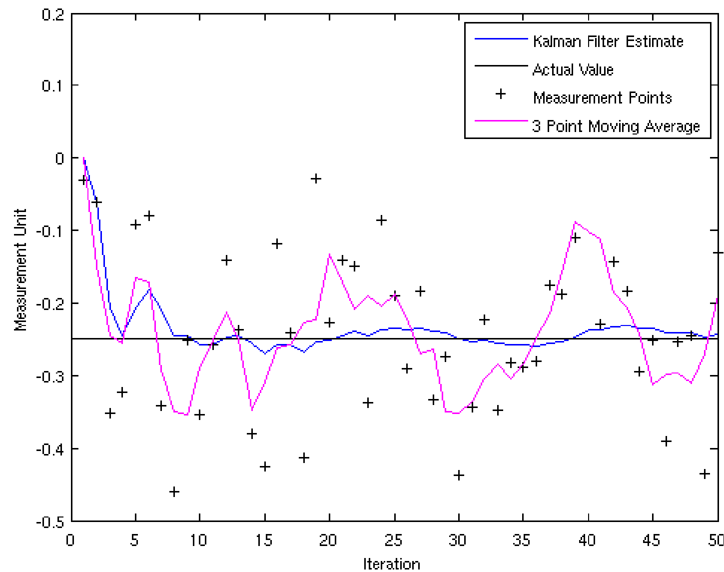


**Figure 22: KF Example, Measurement**

The Kalman filter is able to get very close to the true value of the constant within about 10 iterations, despite the fact that the measurements continue to vary widely. The moving average is effected by this noise and never steadily grasps the correct value, but continues to fluctuate even after 50 estimates from the measurement tool.

The below figure shows the inner operations of the Kalman filter. The figure is of the covariance, and it is easy to see how quickly the error estimate goes down to a very low value.

This shows that not only does the Kalman filter get much closer to the correct value, it is able to identify exactly how much error or uncertainly it still believes to be present between it's output and the actual true value.

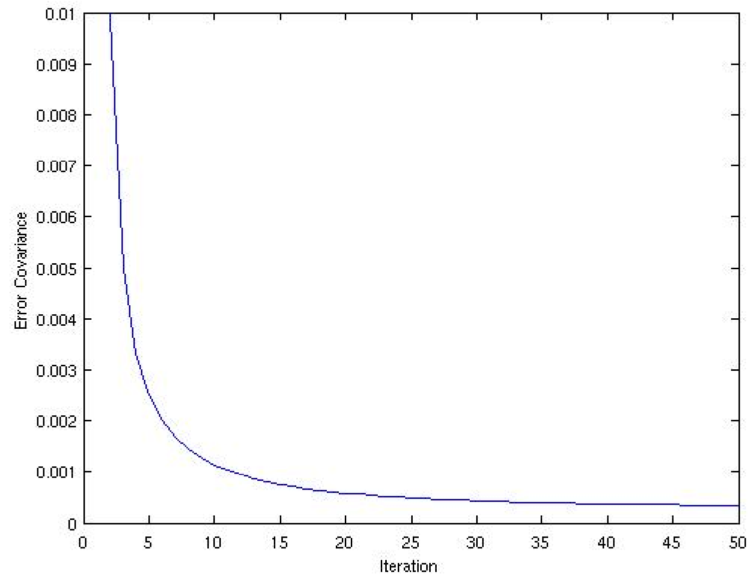


**Figure 23: KF Example, Covariance**

The next figure shows the value of the Kalman gain, K. The Kalman gain is a indicator of the difference between the measured and the estimated value in the Kalman filter. As can be seen, during the start of the estimation, the Kalman gain jumps to a 1 value, as it does not yet have enough information to be able to trust it's state estimate. Therefore in the performance figure above, Figure 22, the blue line is still relatively far from the actual value. As the iterations continue, the Kalman filter gets more and more confident in it's estimation, and even though the measurements continue to be noisy and far from the true value, the Kalman filter trusts it's own previous value more and more, and does indeed get closer and closer to the true value.

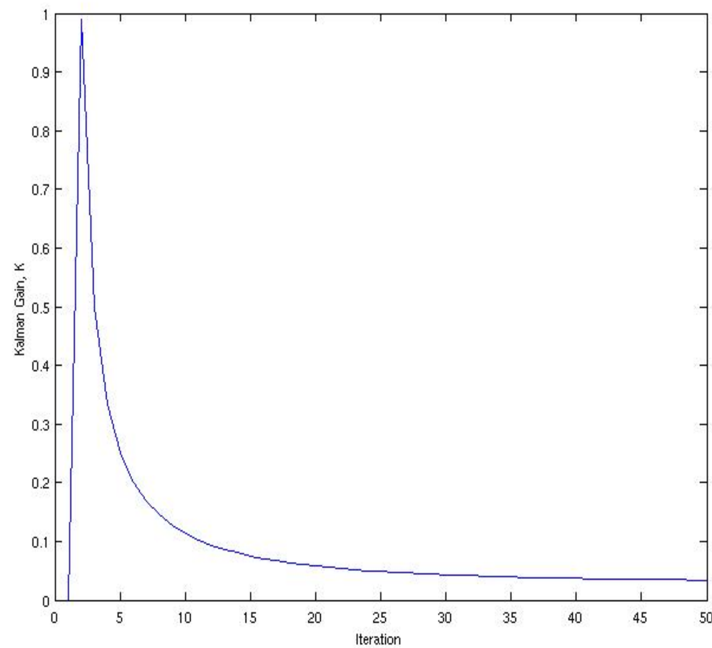


**Figure 24: KF Example, Kalman Gain, K**

In the above case, the variance of the noise is known accurately and this plays to the Kalman filter's advantage. What if the variance is thought to be less than it actually is. That is to say, what if the measurements are thought to be more accurate than they actually are. How does this affect the performance of the Kalman filter.

In the following simulation, the Kalman filter is told that the variance, R is actually 0.0001, a factor of 100 times smaller. We expect that the filter would be more willing to move around as it would expect the measurements to be that much more accurate. Indeed, as the following figure illustrates, when the variance is lower than the correct value, the filter is more strongly influenced by the measurements.

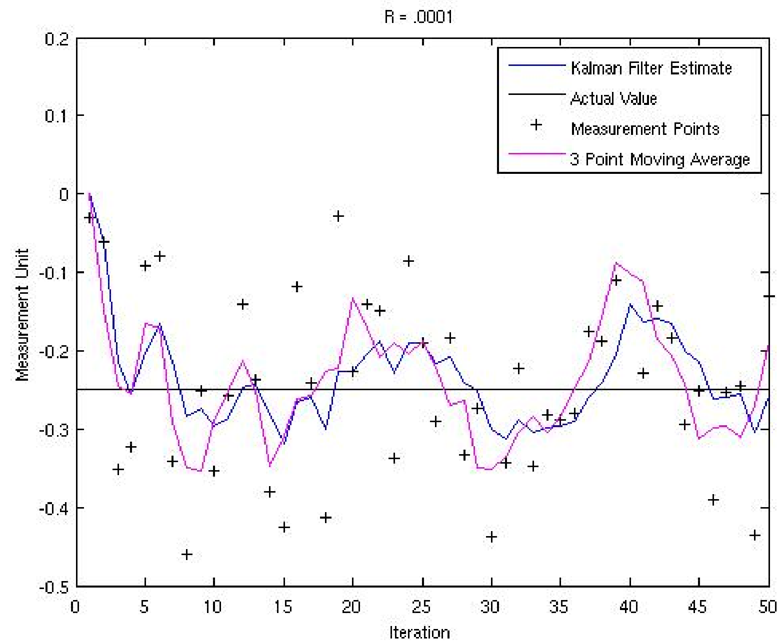


**Figure 25: KF Example, R = .0001**

In the above figure, the blue line is the Kalman filter estimate, and the pink line is again a moving average. It is interesting to note that despite an error in the R value of a factor of 100, the Kalman filter still outperforms the moving average. Of course, it is affected more than in the ideal case by the measurement noise, but not as much as would be expected. If the R value was in fact 0.0001, then one would expect the measurements to be much closer to the true value, and the Kalman filter's movements would correspondingly be much closer to the correct value.

In this way, the Kalman filter is able to balance and modify performance to mach the characteristics of the measuring equipment and other variables. This makes the Kalman filter an ideal solution for many systems and helps to explain it's wide popularity and success in many diverse fields.

## 3.4 HDL Implementation

HDL implementation of the Kalman filter was undertaken. The following section outlines the main issues, problems, solutions and layout of the implementation. The goal being to introduce the reader to the methods used so that the reader is able to understand, use, and modify the implementation to his or her advantage. In fact versatility has been one of the objectives of the system from it's initial conception. What good is a system if it only works for a certain well defined problem. The system described below has the capability to adapt and change to suit the needs of the user.

There are several steps involved in programing the Kalman filter system. The first is design.

### Designing the system

The first step is to chose the system to implement. In this case, a Kalman filter solution is desired. The Kalman filter is a popular and useful system to have, and it would be even better if it was in a hardware environment, where certain properties of hardware systems make it an extremely attractive option. A hardware system is in many cases faster and when the system requires taking measurements and observations from hardware components, it makes sense to design the system to take advantage of the benefits of a hardware implementation. In hardware, two options are to design and implement an IC, or to use a FPGA. Unlike ICs, FPGAs are modifiable at load time, and the investment required for an FPGA implementation is much, much lower than an IC system. For this, and many other reasons an FPGA system was chosen for this implementation.

No matter the environment chosen, some basic design criteria must be established. In this case, the operation of the Kalman filter was analyzed and defined. Certain operations were also deemed as necessary for the system to perform efficiently for the Kalman filter to be successfully implemented. (See Design Considerations, above)

Once the Kalman filter is chosen as the system to implement, two main areas are targeted for detailed design, hardware and software. The components that are to run the controller and filter code are chosen and tested to meet the requirements of the design.

#### Hardware

In this case, hardware is an FPGA system by the Xilinx company. In choosing this vendor, one is constrained to the systems and software used by the company to program it's tools. Namely, in this case, the simulation software provided by the Xilinx company is called ISE.

A free version of Xilinx ISE is available for download and is called the ISE Webpack Tool. This version of the software supports many of the Xilinx FPGAs including most of the Virtex and Spartan family of FPGAs. In general, devices with very large numbers of

I/O pins and gate matrices are not available in the free version. For access to all of the Spartan and Virtex FPGAs the full ISE Design suite can be purchased from Xilinx. Although the full version was available and used during the design and implementation of this system, depending on the size of the project, the Webpack version should be sufficient for most cases, including the Kalman filter system being discussed.

Information regarding built in FPGA functions is available in [17] which outlines Xilinx macros and primitives which are used to streamline the FPGA design process. By using primitives and built in functions, the design can be at a higher level and more robust.

Through the use of the Xilinx ISE simulation package, all of the functionality of the hardware system is easily identified, modified and tested, without the need to export and download to a physical system. Since Xilinx is a reputable and well known vendor, the tools are by now highly sophisticated and accurate. Thousands of signals call be accessed and viewed waveform by waveform or through test-benching software programming. Once functionality is established in simulation, one can be very confident in the performance specifications shown in the simulation tools.

#### Software

Software and hardware both come together in the simulation environment. Hardware is described using a hardware description language, (HDL), which in this case was VHDL, and software tools can be used to import data and analyze results.

An area which is fully software based is the pre-coding section (see Programming Process, in Using the System, below), which is done in Perl. Pre-coding is where the functionality of the Kalman filter is specified before it is loaded into the hardware area, where the controller uses the Kalman filter code to operate the controller hardware system to achieve the end result, which is the Kalman filter working in a hardware environment.

In this section, the code can be debugged and modified to introduce new or mare efficient functionality. In this way the system is upgradeable. In fact, an entirely new system could be coded in the pre-coded section. As long as the hardware tools provided in the controller are sufficient to operate the software code, any system could be programmed into the system by a knowledgeable user.

### Coding

Once the desired system is designed, the next step is to start the coding of that system using the tools chosen in the design phase.

As a result of the design analysis of the Kalman filter, the controller was coded to take advantage of certain properties of the FPGA. Namely the parallel processing ability of the FPGA was used to speed up certain operations such as matrix multiplication, which is one of the operations defined as a priority in the design analysis phase.

The coding of the system was done in VHDL in the ISE environment. The FPGA family chosen to code to was the Virtex6. The exact model of the FPGA is the XC6VLX75T, however, due to the efficiency of the system, many other FPGA models or even families could have been chosen.

The controller is divided up into .vhd files which specify the hierarchy of the system. On the top level is the Controller file, in which all other sections are instantiated. Thus, below the controller level are the loader, control store, branch, IO, ALU, and data store units. Further down, the ALU instantiates the respective mathematical modules, the matrix add, multiply and determinant units for 3x3 by 3x3, 3x3 by 3x1 and 3x3 by 1 operations.

## Problems encountered

### Testing accuracy of system

One of the problems with having such a complex system working in one package is the issue of testing. How accurately does the system work when all parts are working together? One way to check this is to test each individual component separately. Due to the modularity of the system, that is to say, each functional block in the code has been written into a separate module. Therefore, each module can be given an respective testbench. It can then be sent inputs and the outputs can be read, as if it were part of a larger system. In this way, each module can be verified on its own, right down to the individual signal level.

This method works in some respects, however, the problem with this system of checking is that it does not take into account separate systems working in conjunction and in parallel. One module may require a certain result from another module. Individually the testbench writer can give the required inputs to each separate module, but this assumes that the input to that module is being fed in accurately, and at the correct time. This may not be the case, and it is difficult to detect, unless all modules in question are run at the same time, as they would be in an actual system.

Therefore, the best methods of testing, once the basic tests have been done at the individual module level, require most, if not all modules to run as they would, in all of their complexity and hierarchy. Therefore, the original problem remains, except now, we are one step closer, that is, assuming each module works individually, and the modules work together, outputting some consistent result from a certain input pattern. At this stage, the question of accuracy is the next question to answer.

The way that the question of accuracy was answered in this case was through the use of a MATLAB script. Once the system was working in the most general sense, that is, consistent results were obtained from inputs, a MATLAB script was used to emulate the system. The main functionality of the controller system, the matrix arithmetic operations were programmed into a MATLAB script. Since the controller uses signals and not

numbers, it is difficult to assess the accuracy of results. MATLAB is the ideal forum to convert and calculate inputs simultaneously in the controller system and the real world system.

The same inputs being fed into the controller and the same operations being performed by the MATLAB script resulted in the same hexadecimal signals as outputs, verifying the performance of the controller operations. This was a crucial step in the rolling out of the controller system, as all other functionality depends on the basic arithmetic operations and sequences.

**Timing**

Another problem encountered during the implementation of the system was the issue of timing. In a system as complex as the above, many signals need to come together at just the right time for the system to work as it should. A very useful tool for checking the system in that regard is the waveform viewer that comes as a part of ISE. There are other waveform viewers that are compatible with ISE, (for example, ModelSim, by Mentor Graphics, which is a popular program) however the program packaged with ISE, called ISim, is also very popular and it works with ISE very well.

In ISim, all waveforms are clearly visible, and any signal can be loaded into the waveform window, module by module. It can be a daunting task to look through many individual signals and line them up in the waveform view to ensure that functionality is correct. There were, therefore certain steps taken to make this functional checking as efficient as possible. Certain coding practices can be undertaken in the HDL code itself to make sure that the waveforms will be at the correct place, or to make it easier to see the changes in the waveform window.

One method is the use of “ensure” statements in the testbench. This statement does as it sounds, it ensures that certain conditions have been met, and outputs errors, or stops the program completely if they have not. This is ideal for data heavy programs that require the output to be, for example, within certain parameters. Instead of checking each number individually the ensure statement could be used to automate checking. In this case, where the difficulty is not with numbers themselves but more with the behaviour of signals, other strategies must be employed to make the waveform viewing as accurate as possible.

For example, in one case, there was a certain block of variables that would change depending on which stage the controller program was in. One could have zoomed in on that stage, and checked each variable separately, but this would be highly time consuming and prone to error. The method that was instead employed involved changing the HDL code to trigger certain signals if certain other variables were being changed. Similar to the software case where a certain output would be sent if the program was in a certain section of the code, certain hardware signals were changed depending on where the program was, and what variables were being changed. The diagram below illustrates this example.

The question to be answered was what branch of the program was the variable in question affecting. To answer that question, a very obvious signal was changed at the same time as the variable or section of code in question. A “XX” signal was output on the “what_branch_p” signal and the following result was obtained.

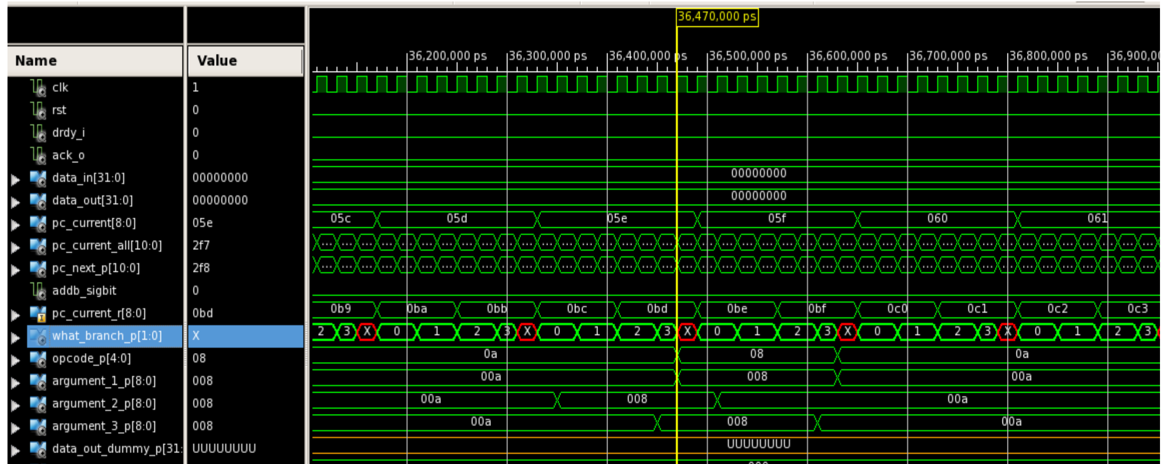


**Figure 26: Finding System Changes Using Highlighted Variables**

As can be seen in the above figure, the result is readily apparent. The signal “what_branch_p” does in fact change to a bright red “X” signal at the later half of the “3” stage. This indicates that the changes to the program, or the variable in question was being acted upon on the second half of the “write” phase. Now, further investigation is possible and debugging is much more focused and fruitful.

The above is just one example of using the capabilities of the waveform program in creative ways to solve what can be very hard to find bugs and issues.

**Memory performance**

In most cases, like the above example, the program requires only minor adjustments to find and eliminate problems. However, in some cases the problem is systemic, and requires fundamental program changes. This example is one of total system changes being required. The problem first became apparent as do many problems, by a system that does not behave as one expects.

In this case, the program was not behaving correctly, and after much debugging, it was established that the memory was the source of the problem. Data was at some times being written, and at other times it was not. When a data address stayed constant for two cycles the data would get written, but when the address changed, that is two subsequent memory locations was written to one after the other, neither of them was able to write back the data that they should have stored.

In the end, the root of the problem was found to be that of the behaviour of the memory itself. The program was written correctly, but it did not anticipate the delay that memory

always had in writing to it. For example, if at one clock cycle one writes to a certain address in the memory, the program assumed that it would write it in that clock cycle. However, the memory being synchronous as opposed to asynchronous caused the memory to not be able to access the correct memory location until the next clock cycle. The *set up time* (the time required for the data to be constant before the clock trigger arrives) of the memory was longer than expected, and the memory would not write until the next rising edge of the clock arrived. This was why if the address was held constant for two clock cycles, the data would be written, but not if the address changed every clock cycle.

To solve this issue required almost a total re-design of the system. The system was originally designed to work through one program cycle in four clock cycles, with an operation being performed each clock cycle. The memory of the FPGA could not handle this system, as at least one extra clock cycle per program stage was required for the signal to reach the memory and for it to recognize the address and data. Therefore, the instruction cycle was expanded from four cycles to eight, and two clock cycles were used in each stage of the program cycle.

Now, at every stage, the first half of the cycle performs the calculations, and gets all of the data and addresses ready and sent to the memory locations, and the second half of the stage is where, when the rising edge of the clock arrives to the memory, the data and addresses have already been set up, and the memory is able to recognize and perform the correct actions.

Since the instruction cycle is fundamental to the program itself and effects all areas of the controller, every section of the system had to be changed, updated and tested to make it compatible with this new mode of operation.

Of course, all of this does not effect the end users in any way, as internal clock cycles are one level of abstraction away from them. The user is more concerned with the operation of the system, that is, the program stages: read, read, execute, and write. These stay constant, and whether each stage takes one or two clock cycles internally has no effect on the outside experience.

**Debugging**

As mentioned above, there were many methods and strategies used to ensure that the system was performing as intended. The debugging stage was a crucial step towards developing confidence in the design. Simulations, test cases, and waveform analysis were conducted using the following tools.

Waveform analysis
Extensive simulations were performed using the ISE tools waveform analysis tool, ISim. In ISim, one is able to load any block in the system, and see all of the signals in that block and how they change over time. This proved to be an invaluable tool. For example, when testing the multiply block, one is able to see each stage in the multiplication, across

all of the points in the matrix, at once. The waveforms are lined up and readily accessible for easy comparison. If one section is not consistent with the others, it can quickly be addressed. In this way, debugging can be a focused and productive effort. A sample waveform from the multiplication block is shown below. Using the waveforms, one is able to follow all of the steps in the operation being performed, in this case, multiplication, and can compute by hand from start to finish, one computation, to find the exact location of any logic errors, if there are any.

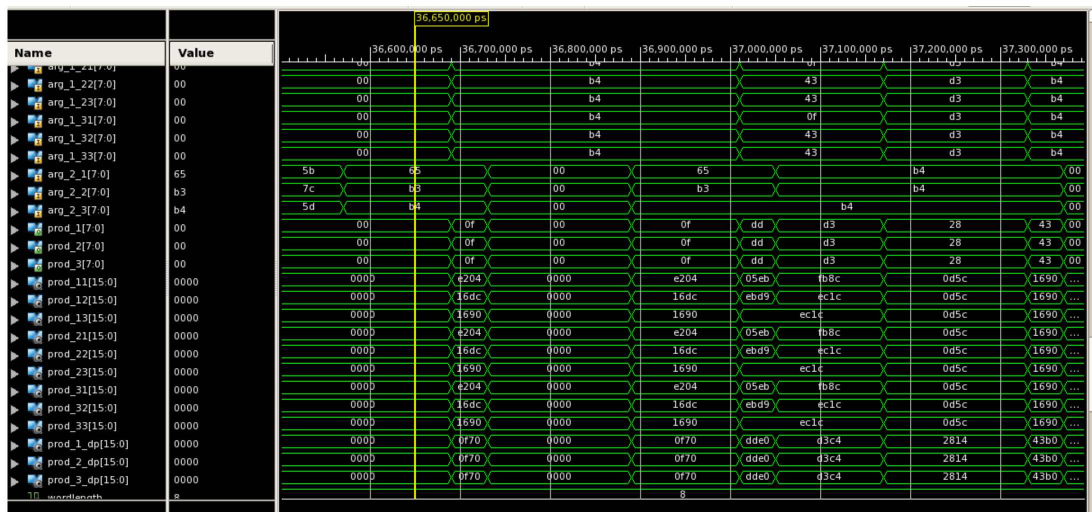


**Figure 27: Multiply Module Waveforms**

On the top left of the above diagram, all of the individual signals are listed. They can each be expanded out further to see exact signal values, but they are left in the hexadecimal format here. These signals are part of the Multiply 3x3 by 3x1 module. The signal names are all the same names used in the programming of this module, and it is apparent that it is the 3x3 by 3x1 block by further inspection. Note that the arg_1 goes from 11 (not shown) down to arg_1_33, which is the 9$^{th}$ position in the 3x3 matrix. The second argument, the arg_2, which is all visible, goes from arg_2_1 down to arg_2_3, which is what one would expect from a 3x1 matrix. These two matrices are multiplied internally by the module, and the output signals are shown below arg_2. The final value, prod_1 is a sum of the prod_11 + prod_12 + prod_13, and so on for prod_2 and prod_3. The prod_XX values are the actual multiplied matrix numbers, depending on the position of the number in the matrix, as is the case for matrix multiplication.

The main point to note is that however the module has been coded, the internal signals are all visible and as their values change, the effect of these changes are readily visible on adjacent signals in the waveform. This makes for a powerful debugging tool.

### MATLAB Simulation

MATLAB is also a very useful tool for simulating a design and ensuring functionally. Due to the heavy use of matrices in MATLAB, which is after all, named after the words “Matrix Laboratory,” using matrices is quick and easy in MATLAB. This enables

extensive pre-design work and consistency checks during development. In fact, below, a simulation of the Kalman filter itself in MATLAB, complete with diagrams and data point visuals, is given to provide the user with a clearer understanding of the algorithm.

Using the testbench itself

The testbench is the overall top level module that wraps around the device under test, in this case, the Controller system. In designing the testbench, one is able to bring forward internal signals or variables that may not be readily visible in the waveform analysis. Diverse signals from different modules and parts of the design can be brought together to the top level for further checking.

These signals can also be acted upon using HDL language constructs as well as “ensure” statements, which will output errors, warnings, or stop the operation of the system if certain conditions are met, or not met.

In this way, the internal workings of the system in question can be readily examined and as the system operates these signals can be used to quickly confirm correct functionality at all times.

## 3.5 Using the system

The following section attempts to provide the user with a quick reference guide with which to understand the steps required for successful operation of the Controller and Kalman filter system.

### Degrees of freedom

Though the system has been designed and implemented for a specific task, there are certain areas in which the end user is able to change to suit the task at hand. The degrees of freedom that the user has are as follows.

KF Coefficients

The end user is easily able to change the operation of the Kalman filter dramatically, simply by keeping everything in the design as is, and only changing the coefficients of the Kalman filter. This creates a Kalman filter that can be optimized for the problem at hand, and is a quick and easy change for fast implementation.

SW layout/design

To modify the design in a more extensive way, the software design for the Kalman filter can be changed as well. In this way, one can implement a completely different filter, if one so wishes. The stage at which this change can be performed is outlined in the next section, the programming process.

### Programming process

The following section outlines the steps required to have a fully functional custom designed Kalman filter system running on the Controller.

1) Define the system

The first step in designing and implementing a Kalman filter system is to define the system and the performance you would like to see from it. Perhaps a simple Kalman filter implementation as it has been presented above will suffice for the application at hand. In this case, the most minimum of changes need be made. However, perhaps you would like to take advantage of the flexibility of the above implementation and define your own type of filter. In this case, the steps for that filter to work need to be articulated.

2) Define KF coefficients

In the case of a Kalman filter implementation, the coefficients for the Kalman filter must be identified. Depending on the system state diagram and the performance of certain system areas, the coefficients will change to match the discrete time linear system that is being modelled. These coefficients must be developed.

3) Use template code provided, or modify KF code

Using the template Kalman filter provided, the system can be used as is with unique coefficients, or the code can be modified to develop a different or altered filter. Developing this code is very intuitive, and this stage has been designed to be very human friendly. The basic instructions, such as READ, WRITE, MULT_33, etc, are easy to understand and no knowledge of the machine code is required. Develop your filter in this stage, and use the given Perl converter to convert this human readable code into machine code for input into the controller system.

4) Set up system, wiring, measurements, etc.

Depending on the number of peripherals, input/outputs, hardware interfaces, etc, the hardware wiring will be unique to the situation at hand. Due to the extensive documentation and commenting in the code itself, the set up of the system should be a relatively straightforward task. Once this system has been set up and is running for the basic code, proceed to the next step.

5) Input coefficients and run program

Now, the unique inputs for the solution at hand can be input and, if necessary, debugged. If all of the steps have been followed, and the system has been simulated and tested at each stage for functionality, this step should be a quick one and the system should be functioning enough to provide outputs for the next stage.

6) Obtain results

The results from the system can be taken from simulations, or if the full hardware system has been set up, the hardware itself. At any of the above stages, the documentation and the report above is a resource to use for debugging or understanding the underlying principles. Using these principles and the above steps, a large number of applications should be able to take advantage of the power of the Kalman filter.

System updates

Due to the easy modification of the system that is possible, at any time in the future, new improvements to the system and changes to the system from adapting to new technologies is possible. The user should be aware of any further releases which may provide added functionality or benefit to the implemented system.

### Performance

Performance of the Kalman filter system has been fantastic. All of the objectives laid out at the start of design have been met and in many cases exceeded. The following section highlights some of those performance metrics. For a full list of performance figures straight from the ISE simulation tool itself, please refer to Appendix C.

## Bibliography

[1] R. E. Kalman, "A new approach to linear filtering and prediction problems," Transactions of the ASME Journal of Basic Engineering, pp. 35-45, 1960.

[2] *Rudolf E. Kalman* [online]. (2012). Available from: <http://www.ieeeghn.org/wiki/index.php/Rudolf_E._Kalman>. [Accessed January 2012].

[3] *Xilinx® : What is an FPGA? Field Programmable Gate Array* [online]. (2012). Available from: <http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm>. [Accessed July 2012].

[4] Brookner, Eli (1998). *Tracking and Kalman Filtering Made Easy*. First. ed. New York: Wiley-Interscience.

[5] T. Misu and K. Ninomiya, Optical guidance for autonomous landing of spacecraft," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 459-473, 1999.

[6] V. Bonato, R. Peron, D.F. Wolf, J.A.M.D. Holanda, E. Marques, and J.M.P. Cardoso, "An FPGA Implementation for a Kalman Filter with Application to Mobile Robotics", in Proc. SIES, 2007, pp.148-155.

[7] D.S.G. Pollock, "Recursive estimation in econometrics", presented at Computational Statistics & Data Analysis, 2003, pp.37-75.

[8] J. Gao, A. Kosaka, and A.C. Kak, "A multi-Kalman filtering approach for video tracking of human-delineated objects in cluttered environments", presented at Computer Vision and Image Understanding, 2006, pp.260-316.

[9] Welch, Gregory F. "History: The use of the kalman filter for human motion tracking in virtual reality." *Presence: Teleoperators and Virtual Environments* 18.1 (2009): 72-91.

[10] Eduardo Pizzini, "FPGA Based Kalman Filter," BSc thesis, 2012, WORCESTER POLYTECHNIC INSTITUTE.

[11] *FPGA vs. ASIC* [online]. (2012). Available from: <http://www.xilinx.com/fpga/asic.htm>. [Accessed July 2012].

[12] C.R. Lee, Z. Salcic, "High-performance FPGA-based implementation of Kalman filter," Microprocessors and Microsystems 21 (1997) pp. 257-265

[13] *Low-Cost Embedded Solution* [online]. (2012). Available from: <http://www.altera.com/devices/processor/fpgas/cost/emb-low-cost.html>. [Accessed July 2012].

[14] Virtex-6 Family Overview [online]. (2012). Available from: <http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf>. [Accessed January 2012].

[15] *Virtex-6 Product Brief* [online]. (2009). Available from: <http://www.xilinx.com/publications/prod_mktg/Virtex6_Product_Brief.pdf>. [Accessed November 2011].

[16] M. Sima and M. McGuire, “Kalman Filters”, Private Communication, November 2011.

[17] *Virtex-6 Libraries Guide for HDL Designs* [online]. (2011). Available from: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/virtex6_hdl.pdf>. [Accessed November 2011].

[18] Arvind Sudarsanam, "ANALYSIS OF FIELD PROGRAMMABLE GATE ARRAY-BASED KALMAN FILTER ARCHITECTURES," PhD dissertation, 2010, Univ. Utah.

[19] *ISE Design Suite 14* [online]. (2012). Available from: <http://www.xilinx.com/publications/prod_mktg/ISE_sellsheet.pdf>. [Accessed July 2012].

[20] *Software and Design Tools* [online]. (2011). Available from: <http://www.xilinx.com/products/design-tools/ise-design-suite/>. [Accessed January 2011].

[21] Vanderlei Bonato, Eduardo Marques, George A. Constantinides, "A Floating-point Extended Kalman Filter Implementation for Autonomous Mobile Robots," Journal of VLSI Signal Processing, 2008.

[22] R. Barnes and A. Dasu, "Hardware/Software Co-designed Extended Kalman Filter on an FPGA", in Proc. ERSA, 2008, pp.281-284.

[23] Vikrant Vij and Rajesh Mehra, "FPGA IMPLEMENTATION OF AREA & SPEED EFFICIENT KALMAN FILTER FOR WIRELESS COMMUNICATION APPLICATIONS," ISP JOURNAL OF Electronics Engineering, Volume 1, Issue 1, October’ 2011, (pp. 26-29).

[24] Greg Welch and Gary Bishop (2006). An Introduction to the Kalman Filter [online]. Available from: <http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf>. [Accessed January 2011].

[25] Anderson, Brian D.O. and Moore, John B. (1979). *Optimal Filtering*. First. ed. New York: Dover Publications, Inc..

[26] R. Müller, J. Teubner, and G. Alonso, "Data Processing on FPGAs", presented at PVLDB, 2009, pp.910-921.

[27] Haykin, Simon (2002). *Adaptive Filter Theory*. Fourth. ed. Upper Saddle River, New Jersey: Prentice Hall.

[28] Maybeck, Peter S. (1979). Stochastic models, estimation, and control. 1st. ed. New York: ACADEMIC PRESS.

[29] I. Karafyllis and C. Kravaris, "On the Observer Problem for Discrete-Time Control Systems", presented at IEEE Trans. Automat. Contr., 2007, pp.12-25.

[30] Kamen, E.W. and Su, J.K. (1999). *Introduction to Optimal Estimation*. First. ed. New York: Springer.

[31] Mendel, Jerry M. (1995). *Lessons in Estimation Theory for Signal Processing, Communications, and Control*. First. ed. Englewood Cliffs, New Jersey: Prentice Hall.

[32] H. W. Sorenson (1970). Least-squares estimation: from Gauss to Kalman. *IEEE Spectrum*. **7**, pp.63-68.

[33] Dongdong Chen and Mihai Sima, "Fixed-Point CORDIC-Based QR Decomposition by Givens Rotations on FPGA," International Conference on Reconfigurable Computing and FPGAs (ReConFig 2011), pp. 327-332, Cancun, Mexico, November 2011.

[34] Gang Chen and Li Guo. 2005. The FPGA implementation of Kalman filter. In *Proceedings of the 5th WSEAS International Conference on Signal Processing, Computational Geometry \& Artificial Vision* (ISCGAV'05), Beniamino Castagnolo (Ed.). World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 61-65.

[35] Brown, Robert G. and Hwang, Patrick Y.C. (2012). *Introduction to Random Signals and Applied Kalman Filtering with MATLAB Exercises*. Fourth. ed. Hoboken, NJ: John Wiley and Sons.

[36] Salzmann, M.A. and University of New Brunswick. Dept. of Surveying Engineering (1988). *Some Aspects of Kalman Filtering*. First. ed. New Brunswick: Department of Surveying Engineering, University of New Brunswick.

# Appendix A: MATLAB Code

```
% Kalman Filter Example

%clear all; %commented to preserve constant zk

A=1; H=1; % Both set to one in this case
Q= 1e-5;  % Some small amount used, if zero, there would be no change

xhat_k_1 = 0; % Initial guess used by Kalman filter. Would converge
with
              % any value
Pn(1)=1;
z = -.25;  % Value to estimate, chosen arbitrarily

R = .1^2;  % Value used by KF. Changed to .01^2 for last run

noise = wgn(50,1,.1^2, 'linear'); % Actual noise power value is .1^2
                                  % matching KF estimate in first run

%zk = z +noise; % Commented to preserve zk after 1st run
zk2=[zk;0];

Xn_(1) = xhat_k_1;
Pn_(1)=1;
Xn(1)=0;

ilim = 50; % Number of iterations to use

for(i=2:ilim)

    % Computing moving avg for reference
    zk_avg(i) = (zk2(i-1)+zk2(i)+zk2(i+1))/3;

    % Kalman Filter part
    % Step 1. State prediction.

        Xn_(i) = A*Xn(i-1);

    % Step 2. Error Calculation.

        Pn_(i) = A*Pn(i-1)*A'+Q;

    % Step 3. Predict the measurements.

        % In this case, H = 1

    % Step 4. Error Calculation

        % As for step 4, n this case, H = 1

    % Step 5: Innovation
```

```
        Nn= zk(i) - Xn_(i);

    % Step 6: Kalman gain calculation

        K(i) = Pn_(i) /(Pn_(i)+R);

    % Step 7: Updates

        Xn(i) = Xn_(i) + K(i)*Nn;
        Pn(i) = (1-K(i))*Pn_(i);

end

%close all

% Plotting figures. Figure 1, the measurement values and KF values
figure
plot(1:ilim, Xn)
hold on
plot([0,ilim],[z,z],'k')
plot(1:ilim,zk,'k+')
plot(1:ilim,zk_avg,'m')
xlabel('Iteration')
ylabel('Measurement Unit')
legend('Kalman Filter Estimate', 'Actual Value', 'Measurement Points',
'3 Point Moving Average')
hold off

figure  % Covariance figure
plot(1:ilim, Pn)
axis([0 50 0 .01])
xlabel('Iteration')
ylabel('Error Covariance')

figure % Kalman gain figure
plot(1:ilim, K)
xlabel('Iteration')
ylabel('Kalman Gain, K')

% For reference, the exact values of zk used:
%    zk =
%
%    -0.0310
%    -0.0618
%    -0.3526
%    -0.3226
%    -0.0921
%    -0.0785
%    -0.3409
%    -0.4602
%    -0.2504
%    -0.3531
%    -0.2583
%    -0.1400
%    -0.2361
%    -0.3798
```

```
%    -0.4263
%    -0.1176
%    -0.2408
%    -0.4131
%    -0.0278
%    -0.2277
%    -0.1401
%    -0.1492
%    -0.3375
%    -0.0852
%    -0.1900
%    -0.2897
%    -0.1837
%    -0.3337
%    -0.2744
%    -0.4378
%    -0.3441
%    -0.2231
%    -0.3465
%    -0.2811
%    -0.2885
%    -0.2796
%    -0.1755
%    -0.1869
%    -0.1104
%     0.0343
%    -0.2288
%    -0.1428
%    -0.1836
%    -0.2951
%    -0.2508
%    -0.3896
%    -0.2534
%    -0.2449
%    -0.4361
%    -0.1305
```

# Appendix B: Kalman Filter Code

**Code Excerpt:**

```
        .text
B000:   NOP
        READ    0   D1          -- Initialize matrix "A"
        READ    1   D1
        READ    2   D1
        READ    3   D1
        READ    4   D1
        READ    5   D1
        READ    6   D1
        READ    7   D1
        READ    8   D1

        NOP
        READ    0   D2          -- Initialize matrix "At"
        READ    1   D2
        READ    2   D2
        READ    3   D2
        READ    4   D2
        READ    5   D2
        READ    6   D2
        READ    7   D2
        READ    8   D2

        NOP
        READ    0   D3          -- Initialize matrix "Q"
        READ    1   D3
        READ    2   D3
        READ    3   D3
        READ    4   D3
        READ    5   D3
        READ    6   D3
        READ    7   D3
        READ    8   D3

        NOP
        READ    0   D4          -- Initialize matrix "H"
        READ    1   D4
        READ    2   D4
        READ    3   D4
        READ    4   D4
        READ    5   D4
```

```
READ        6       D4
READ        7       D4
READ        8       D4

NOP
READ        0       D5              -- Initialize matrix "R"
READ        1       D5
READ        2       D5
READ        3       D5
READ        4       D5
READ        5       D5
READ        6       D5
READ        7       D5
READ        8       D5

NOP
READ        0       D6              -- Initialize matrix "Ht"
READ        1       D6
READ        2       D6
READ        3       D6
READ        4       D6
READ        5       D6
READ        6       D6
READ        7       D6
READ        8       D6

NOP
READ        0       D9              -- Initialize matrix "H6"
READ        1       D9
READ        2       D9
READ        3       D9
READ        4       D9
READ        5       D9
READ        6       D9
READ        7       D9
READ        8       D9

NOP
READ        0       D10             -- Initialize matrix "H7"
READ        1       D10
READ        2       D10
READ        3       D10
READ        4       D10
READ        5       D10
READ        6       D10
READ        7       D10
```

```
        READ      8    D10

        NOP
        READ      0    D16                -- Initialize matrix "H16"
        READ      1    D16
        READ      2    D16
        READ      3    D16
        READ      4    D16
        READ      5    D16
        READ      6    D16
        READ      7    D16
        READ      8    D16


L1:     NOP
        MULT_33        D1    D9    D8     -- KF Implementation Code

        MULT_33        D1    D10   T1
        MULT_33        T1    D2    T2
        ADD_33         T2    D3    D11

        MULT_33        D4    D8    D12

        MULT_33        D4    D11   T1
        MULT_33        T1    D6    T2
        ADD_33         T2    D5    D14

        ADD_33         D16   ND12  D15

        NOP
        WRITE          D14   0            -- Updating result of computation
        WRITE          D14   1
        WRITE          D14   2
        WRITE          D14   3
        WRITE          D14   4
        WRITE          D14   5
        WRITE          D14   6
        WRITE          D14   7
        WRITE          D14   8
        NOP
        READ      0    D18                -- Read next input
        READ      1    D18
        READ      2    D18
        READ      3    D18
        READ      4    D18
        READ      5    D18
```

```
        READ        6       D18
        READ        7       D18
        READ        8       D18
        NOP
        MULT_33             D11   D6    T1           -- Continue computation
        MULT_33             T1    D18   D17

                            .                        .
                            .                        .
                            .                        .

        MULT_33             D17   D15   T1
        ADD_33              T1    D8    D19
        MULT_33             D17   D4    T1
        ADD_33              Di    NT1   T2
        MULT_33             T2    D11   D20

        ADD_33              D20   Dz    D10
        ADD_33              D19   Dz    D9

        NOP
        NOP

        JUMP_U              L1

        NOP
        NOP


        .data
D1:     A                         -- Allocating memory for respective matrices
D2:     At
D3:     Q
D4:     H
D5:     R
D6:     Ht
D7:     H4
D8:     H5
D9:     H6
D10:    H7
D11:    H8
D12:    H9
ND12:   H10
D13:    H11
D14:    H12
D15:    H13
```

D16: H14
D17: H15
D18: H16
D19: H17
D20: H18
T1: H19
T2: H20
NT1: H21
Dz: H22
Di: H23

**Machine Code:**

```
x"00000000",
x"80000000",
x"80000200",
x"80000400",
x"80000600",
x"80000800",
x"80000A00",
x"80000C00",
x"80000E00",
x"80001000",

x"00000000",
x"80000001",
x"80000201",
x"80000401",
x"80000601",
x"80000801",
x"80000A01",
x"80000C01",
x"80000E01",
x"80001001",

x"00000000",
x"80000002",
x"80000202",
x"80000402",
x"80000602",
x"80000802",
x"80000A02",
x"80000C02",
x"80000E02",
x"80001002",
x"00000000",
x"80000003",
x"80000203",
x"80000403",
x"80000603",
x"80000803",
x"80000A03",
x"80000C03",
x"80000E03",
x"80001003",

x"00000000",
x"80000004",
x"80000204",
x"80000404",
x"80000604",
x"80000804",
x"80000A04",
x"80000C04",
x"80000E04",
x"80001004",

x"00000000",
x"80000005",
x"80000205",
x"80000405",
x"80000605",
x"80000805",
x"80000A05",
x"80000C05",
x"80000E05",
x"80001005",

x"00000000",
x"80000008",
x"80000208",
x"80000408",
x"80000608",
x"80000808",
x"80000A08",
x"80000C08",
x"80000E08",
x"80001008",

x"00000000",
x"80000009",
x"80000209",
x"80000409",
x"80000609",
x"80000809",
x"80000A09",
x"80000C09",
x"80000E09",
x"80001009",

x"00000000",
x"80000010",
x"80000210",
x"80000410",
x"80000610",
x"80000810",
x"80000A10",
x"80000C10",
x"80000E10",
x"80001010",


x"00000000",
x"50001007",


x"50001215",
x"50540216",
x"4058040A",

x"500C0E0B",

x"500C1415",
x"50540A16",
x"4058080E",

x"4040180F",

x"00000000",
x"88380000",
x"88380200",
x"88380400",
x"88380600",
x"88380800",
x"88380A00",
x"88380C00",
x"88380E00",
x"88381000",
x"00000000",
x"80000012",
x"80000212",
x"80000412",
x"80000612",
x"80000812",
x"80000A12",
x"80000C12",
x"80000E12",
x"80001012",
x"00000000",
x"50280A15",
x"50542411",

x"50441E15",
x"40540E13",
x"50440615",
x"40642E16",
x"50581414",

x"40503009",
x"404C3008",

x"00000000",
x"00000000",

x"A1680000",

x"00000000",
x"00000000",
```

# Appendix C: FPGA Project Statistics

| KF4 Project Status | | | |
|---|---|---|---|
| **Project File:** | KF4.ise | **Implementation State:** | Synthesized |
| **Module Name:** | CONTROLLER | • **Errors:** | |
| **Target Device:** | xc6vlx75t-3ff484 | • **Warnings:** | |
| **Product Version:** | ISE 11.5 | • **Routing Results:** | |
| **Design Goal:** | Balanced | • **Timing Constraints:** | |
| **Design Strategy:** | Xilinx Default (unlocked) | • **Final Timing Score:** | |

| Device Utilization Summary (estimated values) | | | [-] |
|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** |
| Number of Slice Registers | 385 | 93120 | 0% |
| Number of Slice LUTs | 483 | 46560 | 1% |
| Number of fully used LUT-FF pairs | 242 | 626 | 38% |
| Number of bonded IOBs | 191 | 240 | 79% |
| Number of Block RAM/FIFO | 5 | 156 | 3% |
| Number of BUFG/BUFGCTRLs | 4 | 32 | 12% |
| Number of DSP48E1s | 57 | 288 | 19% |

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# ROMs                                                  : 1
 8x10-bit ROM                                           : 1
# Multipliers                                           : 57
 16x8-bit multiplier                                    : 6
 8x8-bit multiplier                                     : 51
# Adders/Subtractors                                    : 41
 11-bit adder                                           : 1
 16-bit adder                                           : 24
```

```
 24-bit adder                                          : 2
 24-bit subtractor                                     : 2
 25-bit subtractor                                     : 1
 8-bit adder                                           : 9
 9-bit adder                                           : 2
# Registers                                            : 23
 1-bit register                                        : 3
 11-bit register                                       : 1
 8-bit register                                        : 18
 9-bit register                                        : 1
# Latches                                              : 229
 1-bit latch                                           : 229
# Multiplexers                                         : 321
 1-bit 2-to-1 multiplexer                              : 266
 11-bit 2-to-1 multiplexer                             : 2
 32-bit 2-to-1 multiplexer                             : 9
 8-bit 2-to-1 multiplexer                              : 40
 9-bit 2-to-1 multiplexer                              : 3
 9-bit 4-to-1 multiplexer                              : 1

=====================================================================
Advanced HDL Synthesis Report

Macro Statistics
# ROMs                                                 : 1
 8x10-bit ROM                                          : 1
# MACs                                                 : 29
 16x8-to-24-bit MAC                                    : 5
 8x8-to-16-bit MAC                                     : 24
# Multipliers                                          : 28
 16x8-bit multiplier                                   : 1
 8x8-bit multiplier                                    : 27
# Adders/Subtractors                                   : 11
 11-bit adder                                          : 1
 8-bit adder                                           : 9
 9-bit adder                                           : 1
# Counters                                             : 1
 9-bit up counter                                      : 1
# Registers                                            : 158
 Flip-Flops                                            : 158
# Multiplexers                                         : 321
 1-bit 2-to-1 multiplexer                              : 266
 11-bit 2-to-1 multiplexer                             : 2
 32-bit 2-to-1 multiplexer                             : 9
 8-bit 2-to-1 multiplexer                              : 40
 9-bit 2-to-1 multiplexer                              : 3
 9-bit 4-to-1 multiplexer                              : 1

=====================================================================

Final Register Report

Macro Statistics
# Registers                                            : 166
 Flip-Flops                                            : 166
```

```
=========================================================================
*                            Design Summary
*
=========================================================================

Top Level Output File Name         : CONTROLLER.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 685
#      GND                         : 1
#      INV                         : 3
#      LUT1                        : 8
#      LUT2                        : 87
#      LUT3                        : 15
#      LUT4                        : 166
#      LUT5                        : 58
#      LUT6                        : 146
#      MUXCY                       : 71
#      MUXF7                       : 48
#      VCC                         : 1
#      XORCY                       : 81
# FlipFlops/Latches                : 410
#      FD                          : 1
#      FDC                         : 21
#      FDE                         : 144
#      LD                          : 244
# Clock Buffers                    : 4
#      BUFG                        : 3
#      BUFGP                       : 1
# IO Buffers                       : 190
#      IBUF                        : 43
#      OBUF                        : 147
# DSPs                             : 57
#      DSP48E1                     : 57
# Others                           : 10
#      RAMB16BWER                  : 10

Device utilization summary:
---------------------------

Selected Device : 6vlx75tff484-3

Slice Logic Utilization:
 Number of Slice Registers:             385  out of  93120     0%
 Number of Slice LUTs:                  483  out of  46560     1%
    Number used as Logic:               483  out of  46560     1%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    626
   Number with an unused Flip Flop:     241  out of    626    38%
   Number with an unused LUT:           143  out of    626    22%
   Number of fully used LUT-FF pairs:   242  out of    626    38%
   Number of unique control sets:        12
```

```
IO Utilization:
 Number of IOs:                         191
 Number of bonded IOBs:                 191  out of    240    79%
    IOB Flip Flops/Latches:              25

Specific Feature Utilization:
 Number of Block RAM/FIFO:                5  out of    156     3%
    Number using Block RAM only:          5
 Number of BUFG/BUFGCTRLs:                4  out of     32    12%
 Number of DSP48E1s:                     57  out of    288    19%

=======================================================================
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 1.568ns (Maximum Frequency: 637.755MHz)
   Minimum input arrival time before clock: 1.696ns
   Maximum output required time after clock: 13.159ns
   Maximum combinational path delay: 2.009ns

Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 1.568ns (frequency: 637.755MHz)
  Total number of paths / destination ports: 495 / 274
-------------------------------------------------------------------------
Delay:               1.568ns (Levels of Logic = 3)
  Source:            PC_current_r_10 (FF)
  Destination:       PC_current_r_10 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: PC_current_r_10 to PC_current_r_10
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              5   0.280   0.564  PC_current_r_10
(PC_current_r_10)
     LUT5:I0->O            1   0.053   0.252  Mmux_PC_next_p251_SW1
(N166)
     LUT6:I5->O            1   0.053   0.313  Mmux_PC_next_p232
(Mmux_PC_next_p232)
     LUT6:I4->O            2   0.053   0.000  Mmux_PC_next_p2133
(PC_next_p_10_OBUF)
     FDC:D                    -0.012          PC_current_r_10
    ----------------------------------------
    Total                      1.568ns (0.439ns logic, 1.129ns route)
                                       (28.0% logic, 72.0% route)
```