Characterization and Implementation of a Real-World Target Tracking Algorithm on Field Programmable Gate Arrays with Kalman Filter Test Case

THESIS

Benjamin Hancey, Captain, USAF

AFIT/GE/ENG/08-10

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GE/ENG/08-10

Characterization and Implementation of a Real-World Target Tracking Algorithm on Field Programmable Gate Arrays with Kalman Filter Test Case

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Benjamin Hancey, B.S.E.E.

Captain, USAF

March 2008

AFIT/GE/ENG/08-10

Characterization and Implementation of a Real-World Target Tracking Algorithm on Field Programmable Gate Arrays with Kalman Filter Test Case

Benjamin Hancey, B.S.E.E.

Captain, USAF

Approved:

/signed/	28 Feb 2008
Dr. Yong C. Kim (Chairman)	date
/signed/	28 Feb 2008
Dr. Juan R. Vasquez (Member)	date
/signed/	28 Feb 2008
Dr. Guna S. Seetharaman (Member)	date

Abstract

On today's modern battlefield, the ability to adapt is critical. The asymmetric threats that we now face require that we have the ability to evolve and field new technology quickly and reliably. The Kalman filter is an important algorithm often used in target tracking applications to estimate the future behavior of a system based on a series of past behaviors. There exists an urgent need to provide a flexible Kalman filter implementation in a portable yet synthesizable design. A one dimensional Kalman Filter algorithm provided in `Matlab`® is used as the basis for the Very High Speed Integrated Circuit Hardware Description Language (VHDL) model. The JAVA programming language is used to create the VHDL code that describes the Kalman filter in hardware which allows for maximum flexibility. The internal parameters of the filter such as process noise covariance, measurement noise covariance, data width, and data shape can be adjusted to achieve an optimal design to fit any requirement.

A one-dimensional behavioral model of the Kalman Filter is described, as well as a one-dimensional and synthesizable register transfer level (RTL) model with optimizations for speed, area, and power. These optimizations are achieved by a focus on parallelization as well as careful Kalman filter sub-module algorithm selection. Newton-Raphson reciprocal is the chosen algorithm for a fundamental aspect of the Kalman filter, which allows efficient high-speed computation of reciprocals within the overall system. The Newton-Raphson method is also expanded for use in calculating square-roots in an optimized and synthesizable two-dimensional VHDL implementation of the Kalman filter. The two-dimensional Kalman filter expands on the one-dimensional implementation allowing for the tracking of targets on a real-world Cartesian coordinate system.

An additional goal of this research is to perform an investigation and characterization of how to realize optimal real-time target tracking algorithms in hardware,

such as FPGAs or ASICs, while satisfying real-time throughput and bandwidth requirements.

Acknowledgements

This thesis would not have been possible if not for the help and patience of my family, friends, and professors. First and foremost, I need to thank my wife for her encouragement and understanding. I also need to thank my three children, who never forgot who I was despite many prolonged absences and whose hugs and smiles could bring me up in an instant. I made many new friends here at AFIT that supported me and helped all along the way; thanks guys. I had many great professors to whom I extend a sincere thank you. In particular, I would like to thank Dr. Yong Kim for his mentorship and guidance down this long difficult road.

Benjamin Hancey

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

Abbreviation		Page
FPGA	Field Programmable Gate Array	1
HDL	Hardware Description Language	1
RTL	Register Transfer Level	14
VLSI	Very Large Scale Integration	15
ASIC	Application-Specific Integrated Circuit	15
EN	Equation Number	31
ALU	Arithmetic Logic Unit	39
CLB	Configurable Logic Block	63

Characterization and Implementation of a Real-World Target Tracking Algorithm on Field Programmable Gate Arrays with Kalman Filter Test Case

I. Introduction

1.1 Chapter Overview

In section 1.2 a brief motivation for this thesis is provided. The problem statement and scope of the research is presented in Sections 1.3 and 1.4 respectively.

1.2 Research Motivation

In todays military, the need to adapt quickly has become paramount. Quick adaptation means fielding new technologies quickly, efficiently, and reliably. In the past, designing, testing, and fabricating new integrated circuit technology was a long and expensive process. The goal of this research is to speed up this process with respect to target tracking technologies. This thesis characterizes a set of real-time implementation requirements of a Kalman filter for implementation on an FPGA and discusses various means to optimize implementations via scalable architectures. It is through the use of FPGAs that the design process can be minimized both in time as well as expense. By specifying a system in a Hardware Description Language (HDL) and using that description to quickly compare alternatives in design, as well as testing for correctness, tremendous time and expense can be saved over traditional hardware prototyping. A design process that used to take years can now be reduced to as little as a few months.

1.3 Problem Statement

The Kalman filter, an optimal linear estimator, has become widespread in its applications. One notable use of the filter is in target tracking; where noisy, inaccurate sensor data makes the filter invaluable. As described above, the need to adapt technology quickly has led the Air Force towards the use of more economical, and flexible technologies such as the FPGA. This thesis brings these two technologies together, the Kalman filter, and FPGA by implementation of the Kalman filter in VHDL for use on FPGAs. The development of a clear set of guidelines allows for the derivation of a solution given the problem characteristics. In the case of a real-time target tracking algorithm one might look at parallelizability and pipelinability as characteristics of the problem. These characteristics then play an important role in determining a solution. In general the characteristics of real-time target tracking is that of an intense processing requirement. Whether the tracking is done by radar, infrared cameras, passive detection, or visible-spectrum cameras the amount of data coming into the system for processing can be enormous. For this reason it is essential for the designer of a target tracking system to be able to optimize the design for various optimizations. In other words, the designer needs to look at optimization for such system characteristics as speed, area, and power consumption. Often times secondary considerations might be reusability of code or design flexibility (i.e. the ability of a design to adapt and transform to fulfill different requirements).

1.4 Research Scope

The scope of this research is the development and testing of a Kalman filter in VHDL that satisfies the requirements. These requirements are that the VHDL model produce outputs that are reasonably close to outputs produced by the `Matlab`® Kalman filter model. Also, a JAVA program will be described that allows the VHDL Kalman filter model to be more flexible in the sense that a user is able to specify various parameters as well as designate the desired bit-width. Any generalizations in

reference to model optimality are only in reference to the optimality of the Kalman filter algorithm itself.

1.5 Thesis Format

This thesis is presented in five chapters. The motivation for the development of the VHDL Kalman filter model is presented in Chapter 1, along with the problem statement and assumptions. Chapter 2 provides a brief history of Kalman filters as well as a brief discussion of real-time systems. Details of the problem solving approach and considerations made during the design process are presented in Chapter 3, along with implementation details and detailed explanations of the various hardware stages required in the design. Results of the research including testing and design evaluations are presented in Chapter 4, and conclusions and recommendations for future work are found in Chapter 5. Additional information including portions of the VHDL and JAVA code are listed in the appendices in order to facilitate application and future research.

II. Real-Time Systems and the Kalman Filter

2.1 Chapter Overview

This chapter discusses real-time systems, and the Kalman filter. Real-time systems are discussed to demonstrate some of the requirements of implementing a Kalman filter as a real-time system. The Kalman filter is then discussed in detail including a historical perspective, current applications, and the equations that describe the iterative nature of the filter.

2.2 Real-Time Systems

The Kalman filter is an optimal linear estimator which, optimally estimate the behavior of a system that relies on noisy inaccurate data. One example of this is in the use of radar to track aircraft. The data provided by the radar can be noisy and often full of inaccuracies. It is the job of a linear estimator such as the Kalman filter to optimally estimate the state of the system. For target tracking purposes, the estimations provided by the Kalman filter need to complete in real time. In the case of air traffic control, anything less than real time could prove disastrous.

In order to define what a real time system is, it is important to first understand what is meant by 'system'. In [18] a system is defined as "a regularly interacting or interdependent group of items forming a unified whole". The formation of a unified whole implies a boundary between the system in question and the environment or everything else around it [20]. Real-time implies the need to satisfy timing constraints. Combine the two definitions and a real-time system is a system that has specific input, output, and timing constraints.

2.2.1 Problem or Implementation Domination. When developing a real-time system a designer needs to consider whether the system is problem dominated or implementation dominated. This essentially asks the question, where is the most difficulty in designing a system coming from? Does technology exist to implement a system? If so, then the system is problem dominated. If technology doesn't exist

yet or is cost prohibitive then development of a system would be considered implementation dominated. When a technology is just barely capable of solving a type of problem "the engineering of solutions in this phase is inevitably implementation dominated" [20]. In the case of the Kalman Filter specified in this thesis, the technology to build such a system is relatively new and arguably just capable of handling such a problem; therefore the system is implementation dominated. In other words, the form of implementation becomes critically important, which is the primary reason for this research to optimize the speed performance over area.

As mentioned above, one hallmark of realtime systems is the idea of timing constraints. These timing constraints can also be thought of as deadlines that must be met or the system fails. It can also be stated as, a design must satisfy certain timing requirements. When a design does not satisfy timing requirements, it means that the delay of the critical path is greater than the target clock period [21]. The critical path delay is the largest delay between flip-flops. It is a combination of several delays, including: clk-to-out delay, routing delay, setup timing, clock skew, and so on [21]. To illustrate this, consider the example of an optical target tracking system that is required to track twenty thousand targets simultaneously. Assuming that the system takes its inputs from cameras which produce 30 frames per second; the system must be capable of performing:

$$30\ frames \times 20,000\ targets = 600,000\ calculations/second$$

Anything less than this and the system will fail due to an inability to meet the timing requirements. If the system is running on an FPGA that runs at a maximum speed of 40MHz then the maximum time allowed per calculation would be:

$$\frac{40,000,000\ cycles/second}{600,000\ calculations/second} = 66.6\bar{6}\ cycles/calculation$$

Any design that is unable to do at least one calculation every $66.6\bar{6}$ cycles would not meet the real-time requirements.

2.3 The Kalman Filter

The Kalman Filter is a means to predict the future behavior of a system based on past behavior. A system's past behavior is, in a way, remembered and used along with measurements to make the predictions of how the system might behave in the future. According to [13] the reason that tools such as the Kalman Filter are useful to a designer is because virtually all systems are non-deterministic. In other words, few if any systems are devoid of randomness or stochastic behavior. Whether a system inherently contains stochastic processes or the environment that may act upon a system is itself stochastically governed it inevitably is non-deterministic [13]. According to Maybeck "When considering system analysis or controller design, the engineer has at his disposal a wealth of knowledge derived from deterministic system and control theories." He goes on to say:

> There are three basic reasons why deterministic systems and control theories do not provide a totally sufficient means of performing this analysis and design. First of all, no mathematical model is perfect...A second shortcoming of deterministic models is that dynamic systems are driven not only by our own control inputs, but also by disturbances which we can neither control nor model deterministically...A final shortcoming is that sensors do not provide perfect and complete data about a system [13].

It is naive and often inadequate to assume that a designer can have perfect control of all of a systems parameters as well as the environment acting upon it [13]. This is why the Kalman filter, an optimal linear estimator, has become so important and widespread in the technology of today. The Kalman filter takes inaccurate, incomplete, and noisy data combined with environmental disturbances beyond a designers control and over time develops an optimal estimate of desirable quantities.

2.3.1 Historical Perspective.

Historically the Kalman filter owes its origins to a time long preceding that of Rudolf Emil Kalman, the coinventor of the Kalman

filter. The date was 1 January 1801 when an astronomer by the name of Giuseppe Piazzi discovered Ceres (a dwarf planet located in the Solar Systems asteroid belt). He tracked the new planet for several days before illness interrupted his observations. After reporting his discoveries other astronomers were forced to wait due to solar glare before attempting themselves to find Ceres [10]. Attempts to locate it were unsuccessful and it proved too difficult for them to predict its exact position. To locate Ceres, Carl Friedrich Gauss, a mere 24 years old at the time, developed a method called least-squares analysis and successfully predicted the position of Ceres using this method [2]. The method of least-squares analysis is an early example of an estimator that estimates the state of a dynamic system from incomplete and noisy measurements just as the Kalman filter does. However, it was R. E. Kalman who developed and proved the Kalman filter is an *optimal* system-state estimator that minimizes the estimated error covariance [21].

2.3.2 Applications of the Kalman Filter. The Kalman filter is an estimation tool which has been applied over a wide variety of disciplines. Key applications of the Kalman filter are inertial navigation, sensor calibration, radar tracking, manufacturing, economics, signal processing, freeway traffic modeling, and target tracking in general [6].

2.4 Characterization and Implementation of the Kalman Filter

This section discusses the Kalman filter equations in detail and then relates those equations to the `Matlab`® equations used as the basis for the VHDL implementation.

2.4.1 Preliminary Definitions.

1. a priori: knowledge that is independent of experience.
2. a posteriori: knowledge that is dependent on experience.

The Kalman Filter attempts to estimate the state $x \in \mathbb{R}^n$ of a discrete time controlled process, where x is the state which is contained in the set of all reals and

is of dimension n [21]. The dimension could be the x, y on a coordinate plane or the number of variables involved in describing the state of a target (e.g. position, and temperature).

2.4.2 Equations and Explanations. The Kalman filter estimates the state of a discrete-time controlled process governed by the linear stochastic difference equation 2.1 [21].

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{2.1}$$

In Equation 2.1 the $n \times n$ matrix A relates the state x at the previous time step $k-1$ to the current state of x. That is to say, because we assume a linear relationship between x_k and x_{k-1} the relationship can be defined as $x_k = Ax_{k-1}$ in the absence of a driving function Bu_k and process noise w_k [21]. It is important to note that in this thesis it is assumed that A remains constant even though it is possible that it might change with each time step [21].

In equation Equation 2.1 the $n \times l$ matrix B relates the control input u to the state x [21]. For the Kalman filter used in this thesis there are no control inputs so B and u_k were ignored.

The process noise w_{k-1} is any internally occurring noise. That is, no system is perfect and all systems suffer from internal noise. For example, if a Kalman filter were to be made to predict the state of an electronic circuit the noise w_k might be voltage fluctuations inside the circuit due to imperfections caused during the manufacturing process. It is important to remember that for the Kalman filter w is assumed to be white noise with a normal probability distribution.

$$p(w) \sim N(0, Q) \tag{2.2}$$

If these assumptions are not true for any particular system then the Kalman filter becomes less than optimal.

The system measurement equation is

$$z_k = Hx_k + v_k \tag{2.3}$$

where

$$z \in \mathbb{R}^m \tag{2.4}$$

The $m \times n$ matrix H in measurement equation 2.3 relates the state x to the measurement z_k. As with A from equation 2.1 H is assumed to be constant even though in practice it might change with each time step [21]. Any sensor hooked up to a system will have inherent noise included in its signals. Random variable v_k represents the measurement noise [21].

$$p(v) \sim N(0, R) \tag{2.5}$$

As with the process noise, the measurement noise is assumed to be white with a normal probability distribution.

Equations 2.2 and 2.5 have process noise covariance Q and measurement noise covariance R respectively.

When deriving equations for the Kalman filter the goal is to find an equation that computes an *a posteriori* state estimate $\hat{x}_k$ as a linear combination of an *a priori* estimate $\hat{x}_k^-$ as shown below in equation 2.6 [21].

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-) \tag{2.6}$$

Part of equation 2.6, $(z_k - H\hat{x}_k^-)$ is called the measurement innovation, or the residual. This measures the difference between the actual measurement z_k and the predicted measurement $H\hat{x}_k^-$. Any part of an equation seen with a "hat", e.g. $\hat{x}$, represents an estimate.

What makes the Kalman filter optimal is the calculation of the value K. This value is called Kalman gain or blending factor. This was derived to minimize the

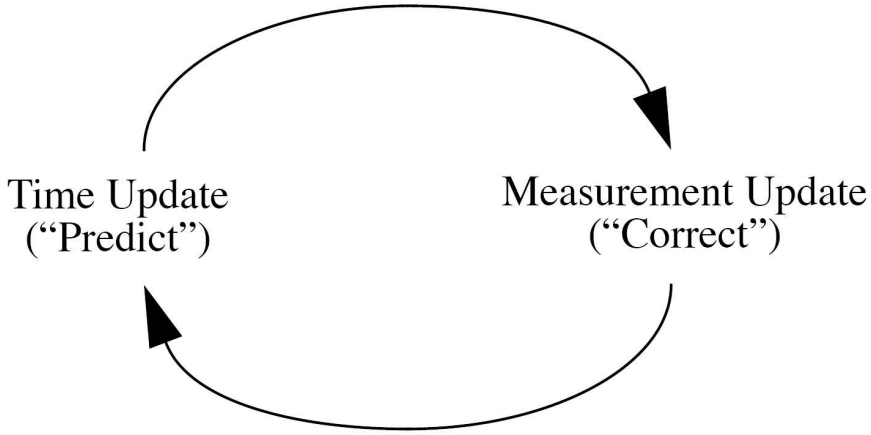


Figure 2.1: Discrete Kalman filter cycle. The *time update* projects the current state estimate ahead in time. The *measurement* update adjusts the projected estimate by an actual measurement at that time [21].

Table 2.1: Discrete Kalman filter time update equations

$$x_k^- = Ax_{k-1} + Bu_k \tag{2.8}$$

$$P_k^- = AP_{k-1}A^T + Q \tag{2.9}$$

a posteriori error covariance [21]. Detailed explanation or derivation of K is beyond the scope of this thesis.

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \tag{2.7}$$

Where P_k^- is the *a priori* estimate error covariance and P_k is the *a posteriori* estimate error covariance [21].

The Kalman filter consists of two stages of equations. Figure 2.1 shows the two stages commonly called the *time update* or predict stage and the *measurement update* or correct stage. The time update equations can be seen in Table 2.1 and the measurement update equations can be seen in Table 2.2 on page 11

Table 2.2: Discrete Kalman filter measurement update equations

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1} \tag{2.10}$$

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-) \tag{2.11}$$

$$P_k = (1 - K_k H) P_k^- \tag{2.12}$$

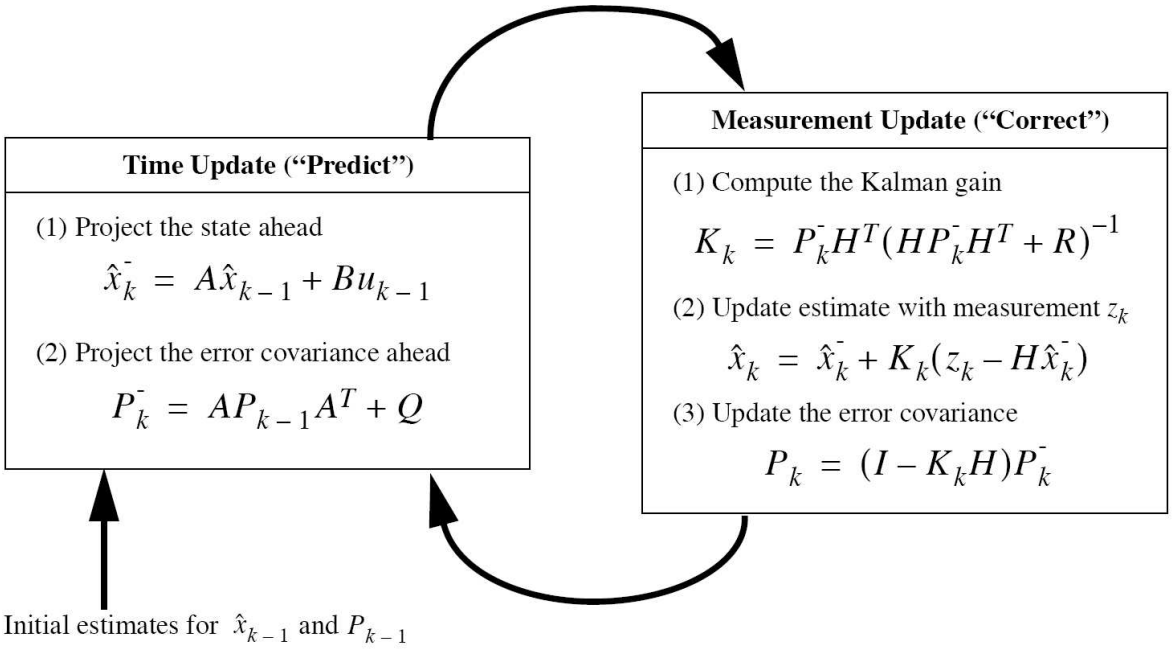


Figure 2.2: A complete picture of the operation of the discrete Kalman filter [21].

Listing II.1:

```
1  for k=2:t_last
2     x(:,k)= phi*x(:,k-1);
3     P(:,:,k)= phi*P(:,:,k-1)*phi' + Qd;
4     A(:,k)= H*P(:,:,k)*H' + R;
5     K=P(:,:,k)*H'*(inv(A(:,k)));
6     residual(:,k)= z(:,k) - H*x(:,k);
7     x(:,k)= x(:,k) + K*residual(:,k);
8     P(:,:,k)= P(:,:,k) - K*H*P(:,:,k);
9     sigma_f(:,k)=sqrt(diag(P(:,:,k)));
10    KK(:,k)=K;
11 end % End time loop
```

2.4.3 Kalman Filter ***Matlab***® *Code.* The code seen in Listing II.1 is designed to produce data that can be plotted using Matlab® . The data is stored in large matrices; this was done solely to allow for the generation of data plots and is not part of the Kalman filter algorithm. A real-world implementation of a Kalman filter, such as that described in this thesis, does not have to store information beyond the previous estimation. The Matlab® variable $x(:,k)$ stores k column vectors for the Kalman filter equations variable $\hat{x}_k$ and $\hat{x}_k^-$. That is, the estimate x for both the time update stage as well as the measurement update stage.

2.4.3.1 ***Matlab***® *Code Association with Kalman Filter Equations.* The following section contains a comparison and association of the Matlab® code and equations for the Kalman filter. Table 2.3 on page 13 shows how the equations for the Kalman filter are paired up with those from the Matlab® code. Note, as shown in Listing II.2, the Kalman gain K is calculated in two separate calculations.

Listing II.2:

```
1     A(:,k)= H*P(:,:,k)*H' + R;
2     K=P(:,:,k)*H'*(inv(A(:,k)));
```

This is also true for the system state $\hat{x}_k$ calculated during the measurement update stage(see Listing II.3).

Listing II.3:

```
1    residual(:,k)= z(:,k) - H*x(:,k);
2    x(:,k)= x(:,k) + K*residual(:,k);
```

Table 2.3: Comparison: Matlab® code and Kalman filter equations

$$x_k^- = Ax_{k-1} + Bu_k \Longleftrightarrow x(:,k) = phi * x(:,k-1)$$

$$P_k^- = AP_{k-1}A^T + Q \Longleftrightarrow P(:,:,k) = phi * P(:,:,k-1) * phi' + Qd$$

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \Longleftrightarrow K = P(:,:,k) * H' * (inv(A(:,k)))$$

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-) \Longleftrightarrow x(:,k) = x(:,k) + K * residual(:,k)$$

$$P_k = (1 - K_k H)P_k^- \Longleftrightarrow P(:,:,k) = P(:,:,k) - K * H * P(:,:,k)$$

III. Approach

3.1 Chapter Overview

This chapter contains the problem definition, the goals and hypothesis, and a discussion on number representation format. It also discusses the design process used to develop the VHDL Kalman filter implementations including behavioral, and Register Transfer Level (RTL) models.

3.2 Problem Definition

This thesis characterizes a set of real-time implementation requirements of a Kalman filter for implementation on an FPGA and discusses various means to optimize the implementation via scalable architectures.

3.2.1 Goals and Hypothesis. The primary goal of this design is to maximize speed/throughput as much as possible. As a secondary goal, minimization of area and power, will also be considered.

In general, the requirement for a digital system must be specified and categorized in order to determine hardware requirements. In other words, system requirements must be paired up with appropriate hardware capabilities. There are many different FPGAs with varying capabilities for a designer to consider. Which FPGA is best suited for a particular design depends on the specific design specifications. System specifications can include many different requirements. For example, power, speed and area may need to be balanced in a weighted fashion to achieve the required results. How to achieve this is the question. Power speed and area are all intertwined, changing one will inevitably cause changes to another. For example, as area increases, often times power requirements will also increase as a result of the increased number of transistors being utilized. If a low power design is required it may be necessary to make speed concessions that may also result in a design requiring less area. It is considerations such as these that determine the most appropriate combination of implementation and hardware to satisfy design requirements.

The traditional VLSI design process is costly both in time and money. Because of this it also involves a tremendous amount of risk.

The traditional design process includes:

- Design
- Verification and testing
- Prototyping

Each of these steps may have to be visited several times as the process is iterative in nature. As the design process moves forward problems can force designers to backtrack in the design process which will always add to the total cost. Tremendous man-hours are required along with an enormous associated cost.

This thesis proposes a system that allows a designer to greatly reduce the time needed for design, verification, and testing; as such, the risk factor is also reduced tremendously. In terms of the case study, specifications for a design requiring or benefiting from the use of a Kalman Filter can be entered into the system and an efficient, and optimized hardware description suitable for implementation on an FPGA or Application-Specific Integrated Circuit (ASIC) is automatically generated.

3.3 Number Representation Format

With respect to the design of the Kalman filter in VHDL it is necessary to choose a method of representing both the integer and fractional portion of a number. Floating point is an option that provides a large dynamic range but with relatively poor precision. After examination of the `Matlab`® code it was decided that dynamic range [14] was not the main issue and that an alternative to floating point could prove to be easier for implementation purposes as well as provide advantages in speed of computation. Fixed point was chosen due to the precision it allows for and the speed benefits it affords. If an application can be done in fixed-point arithmetic, it will probably run faster than in any other format because fixed point is the natural language of the processor [14].

Traditionally the choice would have to be made as to where to place the radix; how many bits to the right and left of the radix would be required. The approach of this thesis to design allows a designer to make this kind of decision at the end of the design process after testing gives further insight into the most appropriate format. Using Java code, abstraction of the Kalman filter internal parameters is performed, thus allowing those abstracted parameters to be defined by the designer whenever most convenient or practical. For initial testing purposes all numbers are represented by a 32-bit fixed-point binary number, and for secondary testing a 64-bit fixed-point binary representation is used. The 32-bit fixed-point representation uses 15 bits for the integer portion, 16 bits for the fractional portion and one bit for the sign. The 15-bit integer can display numbers as large as 32,767 and as small as -32,768. The fractional 16 bits can display a fraction as small as $1.52587890625 \times 10^{-5}$ or as large as $9.999847412109375 \times 10^{-1}$. The 64-bit fixed-point representation uses 31 bits for the integer portion, 32 bits for the fractional portion and one bit for the sign. The 31-bit integer can display numbers as large as 2^{32} or $2,147,483,647$ and as small as $-2^{32} + 2^{31}$ or $-2,147,483,648$. The fractional 32 bits can display a fraction as small as $2.32830643654 \times 10^{-10}$ or as large as $9.999999997671694 \times 10^{-1}$. Table 3.3 on page 16 shows the format used for number representation. In the case of the 32-bit representation, bits 0 through 15 are used to represent the fractional portion of the number, bits 16 through 30 represent the integer, and bit 31 is the sign bit.

Table 3.1: Number representation formats

32-Bit Representation			
	sign	integer	fraction
bits:	31	30 down to 16	15 down to 0

64-Bit Representation			
	sign	integer	fraction
bits:	63	62 down to 32	31 down to 0

If additional bits were available for the fraction, the more accurately a decimal fraction can be represented in binary. Different applications may require different levels of accuracy. This research provides a means for a designer to implement a Kalman Filter that has been optimized for speed, power, and area that also maintains some minimum level of accuracy as specified by the designer.

3.4 Design Optimization

Design optimization can be accomplished in several ways depending on what type of optimization is required. For the Kalman filter design described in this thesis, optimization for speed is most critical. Parallelization and pipelining are two methods used to help create a hardware design that fulfills this requirement. This section discusses various optimizations and their interrelationship. These optimizations or means of optimization are discussed in order of priority as they relate to this thesis.

3.4.1 Optimization by Parallelization. Parallelization in hardware is the simultaneous processing of data. In modern day processors, such as the Pentium line of processors, all instructions given to the computer are processed in series or one-at-a-time. This means that even when performing calculations that lend themselves well to parallelization they cannot take advantage of this. This is one reason why custom designs such as the one described in this thesis can be much faster, and much more efficient than a multipurpose microprocessor. The custom design allows the designer to parallelize or simultaneously process data to the full extent allowed by an FPGA or ASIC of a given capacity. Figure 3.4.1 on page 18 shows a small portion of the Kalman filter algorithm flow where MMult is matrix multiplication, x^- is the *a priori* system state, x^+ is the *a posteriori* system state, P^- is the *a priori* estimate error covariance, P^+ is the *a posteriori* estimate error covariance, Qd is the process noise covariance, H relates the state x to the measurement z, and *phi* relates the state x at the previous time step $k-1$ to the current state of x. Observe how multiple arithmetic operations can be performed simultaneously or in parallel throughout the

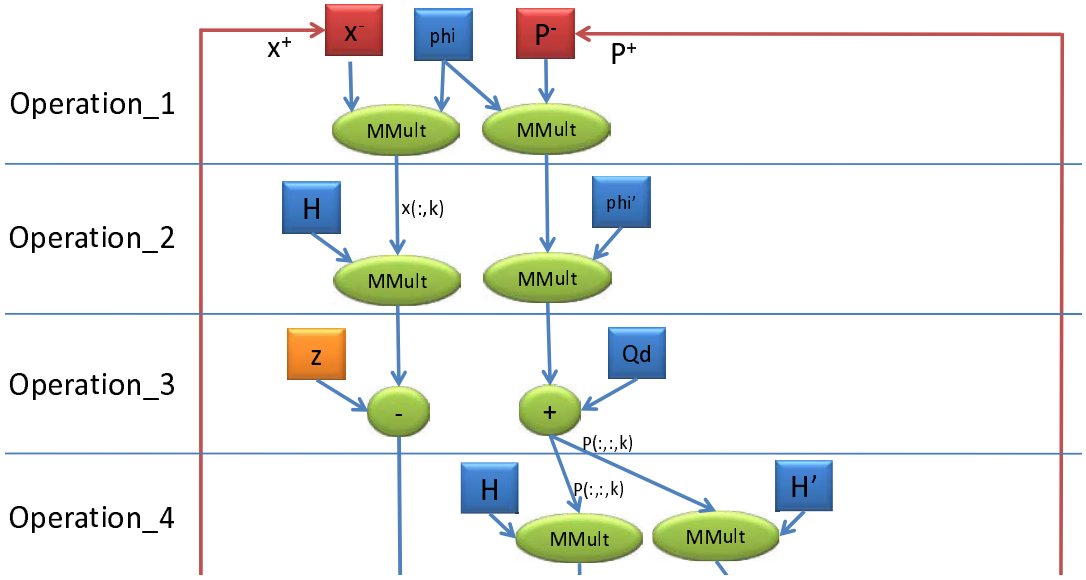


Figure 3.1: This figure shows a portion of the Kalman filter algorithm in flow chart form. Parallelization can be observed in each cycle; multiplications are performed simultaneously. Data dependency can also be observed; Operation_1 must complete before Operation_2 can begin due to its dependency on $x(:, k)$.

algorithm. It also demonstrates data dependencies; Operation_1 must complete before Operation_2 can begin due to its dependency on $x(:, k)$ being completed prior.

3.4.2 Optimization by Pipelining. Pipelining is a common method designers have used to speed up sequential processing. Sequential processing, meaning the need to run data in order, is unfortunately required by some algorithms. Pipelining is a method commonly used to improve throughput of a system. Pipelining creates sequential processing stages that once completed allow the current data to be forwarded to the next stage and new data to be brought in for processing. Pipelining, however, is not always possible at a particular level of circuit abstraction but may be possible at a lower or higher level. If a particular operation requires a computation such as a multiply to be performed multiple times then that multiply can be pipelined. It is important to remember that pipelining improves throughput, not response time. Throughput is the total amount of work done in a given time and response time or execution time is the time between the start and completion of a task [9].

3.4.3 Optimization for Speed. Optimization for speed goes hand-in-hand with the aforementioned optimization methods. However, depending on the type of algorithm being implemented, pipelining may or may not improve performance. If response time is to be optimized then a deep pipeline may actually hinder. For example, if an algorithm is heavily sequential, that is to say order of computation cannot be deviated from, pipelining may actually hinder performance. As a general rule of thumb, faster circuits will require more parallelism, which in turn increases area. This is the trad-off between speed and area: a faster circuit will require more parallelism and therefore suffer an increase in area [12]. The symbiotic existence between speed and parallelism unfortunately does not usually exist between speed and area.

3.4.4 Optimization of Area. Optimization of area may be accomplished in several ways. Resource sharing being the method that will be focused on in this thesis. Resource sharing is the use of a resource by multiple processes. For example, if two distinct processes both require a multiply of the same dimension then they can, at different times, use the same multiplier. That is to say that a resource sharing design may suffer the penalty of diminished throughput unless the operations are mutually exclusive [12]. Although resource sharing allows for the reuse of hardware it typically will also require a more complex control system. Resource sharing is only useful if the increased control complexity results in a smaller area increase than if the resource were simply cloned; this is usually the case. Some synthesis tools that support resource sharing can automatically optimize a design by allowing mutually exclusive operations to share resources. However, there is no guarantee that a tool will do so, therefore, to guarantee resource sharing it is part of the RTL design for this thesis; this allows for more flexibility when choosing a synthesis tool. As mentioned above, as a general rule of thumb, faster circuits require more area.

Table 3.2: Matrix position designators

i	j	k	l
0	0	0	0
0	1	1	0
0	0	0	1
0	1	1	1
1	0	0	0
1	1	1	0
1	0	0	1
1	1	1	1

3.5 Behavioral Model of the Kalman Filter

One of the first steps towards a synthesizable design is to build a behavioral model to help determine correct function of the Kalman filter in VHDL. A full RTL model is difficult and time consuming to design and therefore verification of upper-level behavior is most quickly and effectively done behaviorally.

3.5.1 Matrix Multiplication. Efficiently multiplying matrices is of paramount importance for any Kalman Filter implementation. The vast majority of mathematical operations are performed on matrices. Determination of an efficient method for multiplying matrices in VHDL for behavioral coding is presented next. The first step was to build a table that shows the positions of the individual parts of the matrices that are multiplied.

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} \tag{3.1}$$

Table 3.2 is used to help determine how to set up the nested-for-loops such as those seen in Listing III.1. The position variables i, j, k, and l represent the matrix

$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} \tag{3.2}$$
$$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11} \tag{3.3}$$
$$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10} \tag{3.4}$$
$$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11} \tag{3.5}$$

position designators; e.g. $A_{00} \Leftrightarrow A_{ij}$ where $i = 0$ and $j = 0$ and $B_{00} \Leftrightarrow B_{kl}$ where $k = 0$ and $l = 0$.

Listing III.1:

```
1  --****************************************
2  --* Function to multiply two 2x2 matrices *
3  --****************************************
4  Function matrix_mult_2x2 (A,B: matrix_2x2)
5          Return matrix_2x2 Is
6          Variable result : matrix_2x2;
7          Variable func_temp1 : Signed(63 Downto 0) :=
8          (OTHERS => '0');
9          Begin--Begin function code.
10         For i In 1 to 2 Loop
11          For L In 1 to 2 Loop
12           For j In 1 to 2 Loop
13                 func_temp1 := (A(i,j)*B(j,L)) + func_temp1;
14           End Loop;
15           result(i,L) := func_temp1(47 Downto 16);
16           func_temp1 :=
17           (OTHERS => '0');
18          End Loop;
19         End Loop;
20         Return result;
21 End matrix_mult_2x2;
```

If matrices A and B are traversed left to right from equation 3.2 through equation 3.5 and the position values recorded such as in Table 3.2 on page 20 this

Listing III.2:

```
1 func_temp1 := (A(i,j)*B(j,L)) + func_temp1;
```

information can be used to determine the for-loops. Table 3.2 shows that for-loop variables j and k are identical and therefore one of them can be left out leaving three nested for-loops, one nested inside the other. Determining the order of the for-loop variables is a simple process. Assuming that column k is chosen to be left out since it is identical to column j then the for-loop variable i and j are associated with A and j and L are associated with B. This determines the order of the for-loop variables in the code segment seen in Listing III.2.

3.5.2 VHDL Types. Numeric_std is the type used, as signed numbers are required. Numeric_std is the standard VHDL synthesis package along with numeric_bit. Numeric_std is defined as unconstrained arrays of std_logic elements:

Listing III.3:

```
1     Type unsigned is array (natural range <>) of std_logic;
2     Type signed is array (natural range <>) of std_logic;
```

Using the numeric_std type is not always required, as many VHDL compilers/synthesizers can synthesize other types such as std_logic. However, use of numeric_std helps to ensure synthesis.

3.5.3 Project Code. This section presents a small fraction of the behavioral VHDL representation of the Kalman filter.

3.5.3.1 Behavioral Model. The behavioral model uses 32-bit fixed-point binary numbers. The numbers are broken up into a 16-bit integer portion and a 16-bit fractional portion. The goal is to achieve results that at least approximate those found using the Kalman filter `Matlab`® simulation.

Table 3.3: Constant/variable approximations

Constant/Variable	Matlab® Representation	Fixed-Point Representation
dt	0.1000	0.099908447265625
R	10.0000	10.0000
G	$\begin{pmatrix} 0.0000 \\ 1.0000 \end{pmatrix}$	$\begin{pmatrix} 0.0000 \\ 1.0000 \end{pmatrix}$
B	$\begin{pmatrix} 0.0000 \\ 1.0000 \end{pmatrix}$	$\begin{pmatrix} 0.0000 \\ 1.0000 \end{pmatrix}$
Bd	$\begin{pmatrix} 0.0050 \\ 0.1000 \end{pmatrix}$	$\begin{pmatrix} 0.0049896240234375 \\ 0.099908447265625 \end{pmatrix}$
H	$\begin{pmatrix} 1.0000 & 0.0000 \end{pmatrix}$	$\begin{pmatrix} 1.0000 & 0.0000 \end{pmatrix}$
F	$\begin{pmatrix} 0.0000 & 1.0000 \\ 0.0000 & 0.0000 \end{pmatrix}$	$\begin{pmatrix} 0.0000 & 1.0000 \\ 0.0000 & 0.0000 \end{pmatrix}$
phi	$\begin{pmatrix} 1.0000 & 0.1000 \\ 0.0000 & 1.0000 \end{pmatrix}$	$\begin{pmatrix} 1.0000 & 0.099908447265625 \\ 0.0000 & 1.0000 \end{pmatrix}$
Qd	$\begin{pmatrix} 0.0333 & 0.5000 \\ 0.5000 & 10.0000 \end{pmatrix}$	$\begin{pmatrix} 0.033294677734375 & 0.5000 \\ 0.5000 & 10.0000 \end{pmatrix}$
x(1)	$\begin{pmatrix} 1.0000 \\ 1.0000 \end{pmatrix}$	$\begin{pmatrix} 1.0000 \\ 1.0000 \end{pmatrix}$
P(1)	$\begin{pmatrix} 0.2500 & 0.0000 \\ 0.0000 & 0.2500 \end{pmatrix}$	$\begin{pmatrix} 0.2500 & 0.0000 \\ 0.0000 & 0.2500 \end{pmatrix}$

Due to the limitations of having a fixed number of bits to represent the fractional portion of the numbers for the behavioral model, the results are not exact duplicates of the Matlab® simulation results. However, the results are very close approximations. There are several constants and variables that must be set before simulation of the VHDL Kalman Filter. These constants and variable initializations are taken from the Matlab® Kalman filter code. To see the fixed point representations of these constants and variable initializations see 3.3.

The behavioral model consists of 11 mathematical functions including various matrix manipulation algorithms. It also consists of various constants which are predefined in the Matlab® code. Here are three examples of behavioral code used for matrix manipulation:

Listing III.4:

```
1  --************************************
2  --* Function to add two 2x2 matrices *
3  --************************************
4  Function matrix_add_2x2 (A,B: matrix_2x2)
5          Return matrix_2x2 Is
6          Variable result : matrix_2x2;
7          Begin--Begin function code.
8          For i In 1 to 2 Loop
9                  For j In 1 to 2 Loop
10                         result(i,j) := A(i,j)+B(i,j);
11                 End Loop;
12         End Loop;
13         Return result;
14 End matrix_add_2x2;
15
16 --******************************************
17 --* Function to add a scalar to a 2x2 matrix *
18 --******************************************
19 Function matrix_add_int_2x2 (A: matrix_2x2 ;
20     B: Signed(31  Downto 0))
21         Return matrix_2x2 Is
22         Variable result : matrix_2x2;
23         Begin--Begin function code.
24         For i In 1 to 2 Loop
25                 For j In 1 to 2 Loop
26                         result(i,j) := A(i,j)+B;
27                 End Loop;
28         End Loop;
29         Return result;
30 End matrix_add_int_2x2;
31
32 --************************************************
33 --* Function to multiply a 2x2 with a 2x1 matrix *
34 --************************************************
```

```
35 Function matrix_mult_2x2_2x1 (A: matrix_2x2 ;
36     B: matrix_2x1)
37         Return matrix_2x1 Is
38         Variable result : matrix_2x1;
39         Variable func_temp1 : Signed(63 Downto 0) :=
40         (OTHERS => '0');
41         Begin--Begin function code.
42         For i In 1 to 2 Loop
43                 For j In 1 to 2 Loop
44                         func_temp1 := (A(i,j)*B(j))
45                         + func_temp1;
46                 End Loop;
47                 result(i) := func_temp1(47 Downto 16);
48                 func_temp1 :=
49                 (OTHERS => '0');
50         End Loop;
51         Return result;
52 End matrix_mult_2x2_2x1;
```

Figure 3.2 on page 26 shows the output for a test ran on the behavioral Kalman filter with inputs generated by the Matlab® Kalman filter code. All test inputs into the behavioral VHDL simulation resulted in outputs that closely approximate outputs generated by the Matlab® Kalman filter code. Table 3.4 on page 26 shows the Matlab® inputs and outputs along side their respective VHDL model inputs and outputs. The VHDL model inputs are different due to the translation from decimal to fixed-point binary. The results from the VHDL model are very close approximations to those calculated in Matlab® . This small difference can be attributed to the difference in the number of bits used to represent values inside each model.

3.5.4 The Reciprocal Function. It is necessary to calculate the reciprocal $(1/A)$ as part the calculation of K which is the Kalman gain or blending factor [21], see Listing III.5. Note, A is not a fundamental part of the Kalman Filter; it exists only as a sub-calculation of K.

Figure 3.2: Behavioral Model Simulation For Kalman Filter

Table 3.4: Test results for behavioral model (outputs are for position)

Matlab® Input	Matlab® Output	Model Input	Model Output
1.9934	1.1248	1.9933929	1.1247863
7.0719	1.5157	7.0718994	1.5156707
4.0100	1.9602	4.0099945	1.9602203
0.8519	2.0217	0.8518981	2.0217437

Listing III.5:

```
1 K=P(:,:,k)*H'*(inv(A(:,k)));
```

A problem arises when performing division in binary. Without properly shifting the decimal place to the left, the answer will not be correct. To achieve the correct answer the dividend must be left-shifted by the number of bits that one wishes to have to the right of the radix point. To see this in mathematical terms please refer to equation 3.6.

$$\frac{x \times 2^{32}}{y \times 2^{16}} = z \times 2^{16} \tag{3.6}$$

The following example also illustrates this.

$$\frac{1}{8} = 0.125 \tag{3.7}$$

$$\frac{01.00000000000000000000000000000000_2}{0000000000001000.0000000000000000_2} = 0000000000000000.0010000000000000_2 \tag{3.8}$$

In equation 3.8 the dividend is left-shifted by 32 places and the divisor is left-shifted by 16 places. This results in an answer with a 16-bit fractional portion. The location of the radix is actually only virtual. The division is done without the computer having any sense of where the radix point is actually located.

3.6 Top Level Schematic

The top level schematic of the design can be seen in figure 3.6. This section breaks down the design process into its constituent pieces and discusses each of those pieces.

3.6.1 The Kalman Filter Equations. First, a close look at the Matlab® equations that describe the Kalman filter.

A close examination of the Matlab® equations made it apparent that at least this particular implementation of the Kalman filter is highly order dependent. That is, a majority of steps, i.e. lines one through eight in Figure 3.3 on page 29, are dependent on previous steps. For example, in the following listing lines two and three cannot start until line one has finished. Although it is true that the majority of the Kalman

Listing III.6:

```
1    P(:,:,k)= phi*P(:,:,k-1)*phi' + Qd;
2    A(:,k)= H*P(:,:,k)*H' + R;
3    K=P(:,:,k)*H'*(inv(A(:,k)));
```

filter algorithm requires sequential processing there is room to process portions of the code out of order. The algorithm is broken up into its constituent operations. These operations include: addition, subtraction, multiplication, and reciprocal function. In Figure 3.3 on page 29 each calculation cycle involves at least one constituent operation. For example, calculation cycle 1, which encompasses calculation numbers 1 and 2, each perform one multiply. Because the two multiplies in calculation cycle 1 are independent, in the sense that the input of one does not depend on the output of the other, they can be calculated at the same time. Figure Figure 3.3 on page 29

Listing III.7:

```
1      x(:,k)= phi*x(:,k-1);
2      P(:,:,k)= phi*P(:,:,k-1)*phi' + Qd;
3      A(:,k)= H*P(:,:,k)*H' + R;
4      K=P(:,:,k)*H'*(inv(A(:,k)));
5      residual(:,k)= z(:,k) - H*x(:,k);
6      x(:,k)= x(:,k) + K*residual(:,k);
7      P(:,:,k)= P(:,:,k) - K*H*P(:,:,k);
```

shows the order of calculation for all constituent calculations in the Kalman filter algorithm. This can also be seen in flow-chart form in Figure 3.5 on page 34. It had to be decided how many calculations to allow in parallel at any one time. The more parallel calculations that can be done the fewer clock cycles overall that would be required to complete one iteration of the Kalman filter. Hardware area becomes a concern as any parallel calculation will require additional area unless the calculations are of a different nature. That is, i.e. if two identical multiplications are to be done in parallel then double the hardware is required verses if the multiplications were to be done sequentially. In contrast, if an addition and multiply are required for any particular design then there is not an area penalty for allowing the calculations to occur in parallel except for possibly controller overhead. In general, if two calculations require different hardware then there are not any penalties for performing them in parallel except for possible controller overhead.

3.6.1.1 Development of Figure 3.3. Figure 3.3 on page 29 was used to develop the final RTL model; it stems from an examination of the equations seen in Listing III.7.

Calculation Number	Calculation Cycle	Current Calc	Equation	Calculating	Notes	Saved Into Reg Position	Next Used Calcuation Cycle
1	1	x(:,k)	phi * x(:,k-1)	x1	phi * reg0 => reg0	0	2,9
2	1	P(:,:,k)	phi * P(:,:,k-1)	P1	phi * reg1 => reg1	1	
3	2	P(:,:,k)	phi * P(:,:,k-1) * phi'	P1	reg1 * phi' => reg1	1	
4	2	residual	H * x(:,k)	residual	H * reg0 => reg2	2	
5	3	P(:,:,k)	phi * P(:,:,k-1) * phi' + Qd	P1	reg1 + Qd => reg1	1	4,9
6	3	residual	z(:,k) – H * x(:,k)	residual	input - reg2 => reg2	2	9
7	4	K	P(:,:,k) * H'	K	reg1 * H' => reg3	3	
8	4	K	H * P(:,:,k)	A	H * reg1 => reg5	5	
9	5	K	H * P(:,:,k) * H'	A	reg5 * H' => reg4	4	
10	6	K	H * P(:,:,k) * H' + R	A	reg4 + R => reg4	4	
11	6	K	Inv(H * P(:,:,k) * H' + R)	K	Inv(reg4) => reg4	4	
12	7	K	P(:,:,k) * H' * Inv(H * P(:,:,k) * H' + R)	K	reg3 * reg4 => reg4	4	9
13	8	x(:,k)	K * residual(:,k)	x2	reg4 * reg2 => reg2	2	
14	8	P(:,:,k)	K * H * P(:,:,k)	P2	reg4 * reg5 => reg5	5	
15	9	x(:,k)	x(:,k) + K * residual(:,k)	x2	reg0 + reg2 => reg0	0	1
16	9	P(:,:,k)	P(:,:,k) - K * H * P(:,:,k)	P2	reg1 - reg5 => reg1	1	1

Matrix Multiplication
Addition/Subtraction
Yellow indicates a finished calculation
Special case requiring extra clock cycles

Figure 3.3: Kalman Filter Equation Table

Figure 3.4: Timing for each calculation cycle. Each cycle takes four clock cycles to complete. This includes a write-to-memory. An exception to the number of clock cycles required occurs for the calculation of K during the reciprocal function and consumes 13 additional clock cycles.

The following references to equation number(s) (EN) refer to the line numbers in Listing III.7. Already completed calculations or calculations completed in a previous step are designated by surrounding them with square brackets [ ].

Equation

$$phi * x(:, k-1)$$

EN 1, consists of one multiply and can therefore be completed in one calculation cycle. It is designated to be started and completed in calculation cycle 1. The next equation

$$phi * P(:, :, k-1) * phi' + Qd$$

EN 2, contains three constituent calculations and therefore requires three calculation cycles to complete. It was designated to be started in calculation cycle 1. Calculation cycle 1 is

$$phi * x(:, k-1)$$

$$phi * P(:, :, k-1)$$

Due to data hazards, at most two parallel calculations are performed at any one time.

Because EN 3 and EN 4 require EN 2 to be completed before they can start, calculation cycle 2 continues the calculation of EN 2 as well as beginning the calculations for EN 5. Calculation cycle 2 is

$$[phi * P(:, :, k-1)] * phi'$$

$$H * x(:, k)$$

Calculation cycle 3 finishes calculating both EN 2 as well as EN 5. Calculation cycle 3 is

$$P(:, :, k) = [phi * P(:, :, k-1) * phi'] + Qd$$

$$residual(:, k) = z(:, k) - [H * x(:, k)]$$

Calculation cycle 4 begins the calculation of EN 3 and therefore EN 4 as well. This is due to the fact that EN 3 is a sub-calculation of EN 4. Calculation cycle 4 is

$$P(:,:,k) * H'$$

$$H * P(:,:,k)$$

Calculation cycle 5 consists of only a single calculation; no parallel calculations could be performed at this stage. Calculation of EN 3 is continued. Calculation cycle 5 is

$$[H * P(:,:,k)] * H'$$

Calculation cycle 6 is a special case requiring extra clock cycles to accommodate the reciprocal calculation. In this calculation cycle, the special case of performing two constituent calculations in series is addressed. First, the following calculation is performed

$$[H * P(:,:,k) * H'] + R$$

then immediately following the completion of this calculation the reciprocal or inverse is calculated.

$$inv([H * P(:,:,k) * H' + R])$$

As with calculation cycle 5, calculation cycle 7 consists of only a single calculation. Calculation of the EN 4 is finished. Calculation cycle 7 is

$$K = [P(:,:,k) * H'] * [(inv(A(:,k)))]$$

Calculation cycle 8 begins calculation of EN 6 and EN 7 or the measurement update stage, see Figure 2.2 on page 11. Calculation cycle 8 is

$$[K] * [residual(:,k)]$$

$$[K] * [H * P(:,:,k)]$$

Calculation cycle 9 completes calculation of EN 6 and EN 7 and also completes one iteration of the Kalman filter algorithm. Calculation cycle 9 is

$$x(:,k) = [x(:,k)] + [K * residual(:,k)]$$

$$P(:,:,k) = [P(:,:,k)] - [K * H * P(:,:,k)]$$

Figure 3.4 on page 30 shows the timing for the start and finish of each calculation. Note that the calculation for A from Listing III.7 is actually just a sub-calculation of K and therefore does not appear in 3.4. Also, K appears twice in Listing III.7 because portions of K are calculated simultaneously; this occurs in calculation cycle 4.

3.6.2 The Controller. The controller is the brains of the entire system. It coordinates the various other components to register their values during certain clock cycles. Among other things, the controller's job is to control the flow of data into and out of the memory unit. The controller is a one-hot encoded state machine with nine states, s1 through s9 (000000001_2 through 1000000000_2). State changes occur based on the value of an internal counter that increments and resets based on the clock and conditions within the states. Listing III.8 shows a portion of the VHDL code including state s1.

Listing III.8:

```
1  Architecture behav Of controller Is --This controller will ...
       use one-hot encoding
2
3      Type state_type Is (s1, s2, s3, s4, s5, s6, s7, s8, s9);
4      Attribute enum_encoding: string;
5      Attribute enum_encoding of state_type: type is
6      "000000001 000000010 000000100 000001000 000010000 ...
          000100000 001000000 010000000 100000000";
```

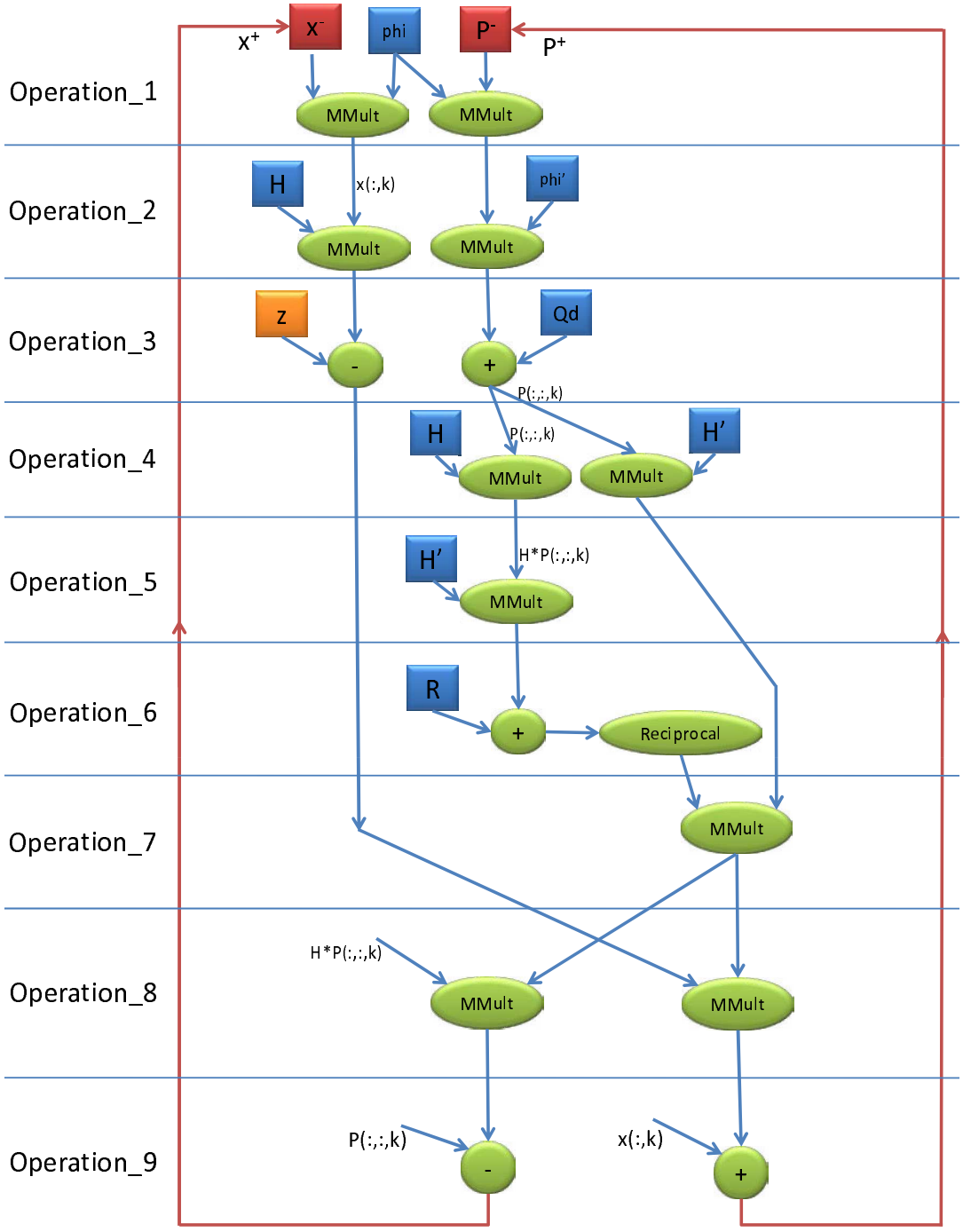


Figure 3.5: Kalman filter algorithm flowchart.

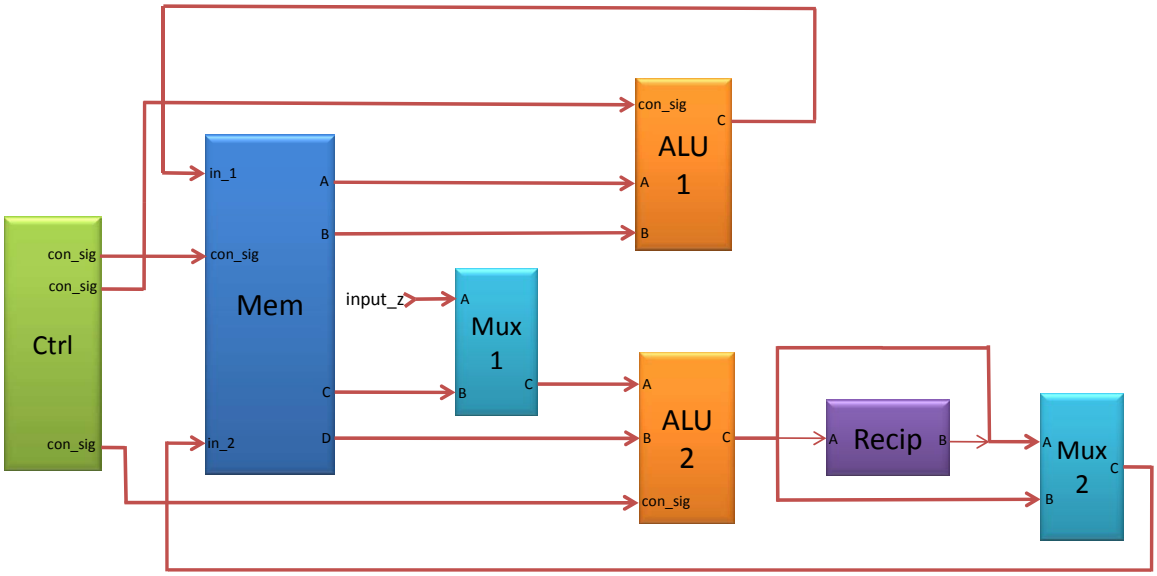


Figure 3.6: Top Level diagram of the Kalman filter VHDL model.

```
7
8          Signal CS, NS : state_type;
9          Signal counter_out : signed(4 Downto 0);
10
11           Begin
12
13 --*****************Begin comb_proc Process...
     ****************************************
14
15               comb_proc: Process (clk, reset)
16
17               Variable counter : signed(4 Downto 0) := "00000";
18
19               Begin
20
21                 If (reset = '1') Then
22                      counter := "00000";
23                      mem_control       <= "0000";
24                      ALU_1             <= "0011";
25                      ALU_2             <= "0011";
```

```
26                    mux_1             <= '1';
27                    mux_2             <= '1';
28                    reciprocal_reset       <= '1';
29                    reciprocal_load        <= '0';
30                    reciprocal_mux_control <= '0';
31                    CS <= S1;
32                    NS <= S1;
33
34                Elsif (clk'event And clk = '1')Then
35                   CS <= NS;
36                   counter_out <= counter;
37                   counter := counter + 1;
38
39                Case CS Is
40                    When S1 =>
41
42                      mem_control      <= "0000";
43                      ALU_1            <= "0011";
44                      ALU_2            <= "0011";
45                      mux_1            <= '1';
46                      mux_2            <= '1';
47                      reciprocal_reset       <= '1';
48                      reciprocal_load        <= '0';
49                      reciprocal_mux_control <= '0';
50
51                      If (counter = "00010") Then
52                          output_reg_load <= '1';
53
54                      Elsif (counter = "00011") Then
55                          NS <= S2;
56                          output_reg_load <= '0';
57
58                      Elsif (counter = "00100") Then
59                          mem_control      <= "1001";
60                          ALU_1            <= "0011";
```

```
61                          ALU_2             <= "0011";
62                          mux_1             <= '1';
63                          mux_2             <= '1';
64                          reciprocal_reset       <= '1';
65                          reciprocal_load        <= '0';
66                          reciprocal_mux_control <= '0';
67
68                          counter := "00000";
69
70                      End If;
71
72                  When S2 =>...
```

3.6.3 Memory Unit. The memory unit is a combination RAM and ROM that allows for both hard-coded constants and writable variables to exist inside a single unit. The memory uses a four bit address to access data in chunks of four. That is, each address effectively is associated with four data that are simultaneously output on four separate buses. Each bus consists of four separate sub-buses that each make up one of four components of a matrix. The way that this memory associates an address with the data that is to be output makes it a unique and highly custom memory. The memory has an asynchronous reset that returns all memory locations to a default starting value. Every number is stored in the form of a 2×2 matrix inside the memory.

Variables (matrix variables) are stored in six separate locations with each location (registers) composed of four numbers. These location names and their associated stored variables can be seen in Table 3.5 on page 38.

Constants are also stored in the form of a 2×2 matrix inside the memory. These values represent various parameters of the Kalman filter and can be set by the designer prior to compilation and synthesis. See Listing III.10 for an example of the

Table 3.5: Memory locations and their associated stored variable.

reg_0	$\Longleftrightarrow$	x
reg_1	$\Longleftrightarrow$	P
reg_2	$\Longleftrightarrow$	$residual$
reg_3	$\Longleftrightarrow$	$H \times P$
reg_4	$\Longleftrightarrow$	K
reg_5	$\Longleftrightarrow$	$K \times H \times P$

signal assignments for a 32-bit example. See Listing III.9 for the input and output ports. Also, see Listing III.11 for output assignments.

Listing III.9:

```
1  Entity mem Is
2      Generic(high_bit : natural := 31);
3      Port(clk, reset    : In std_logic;
4           control : In signed(3 Downto 0);
5           in1_00, in1_01, in1_10, in1_11 : In signed (high_bit ...
               Downto 0);
6           in2_00, in2_01, in2_10, in2_11 : In signed (high_bit ...
               Downto 0);
7           A_00, A_01, A_10, A_11 : Out signed (high_bit Downto 0);
8           B_00, B_01, B_10, B_11 : Out signed (high_bit Downto 0);
9           C_00, C_01, C_10, C_11 : Out signed (high_bit Downto 0);
10          D_00, D_01, D_10, D_11 : Out signed (high_bit Downto 0));
11      End entity mem;
```

Listing III.10:

```
1  --K * H * P(:,:,k)
2           Signal reg5_00 : signed(high_bit Downto 0)       := "...
               00000000000000000000000000000000";
3           Signal reg5_01 : signed(high_bit Downto 0)       := "...
               00000000000000000000000000000000";
4           Signal reg5_10 : signed(high_bit Downto 0)       := "...
               00000000000000000000000000000000";
5           Signal reg5_11 : signed(high_bit Downto 0)       := "...
               00000000000000000000000000000000";...
```

Listing III.11:

```
1  Elsif (clk'event And clk = '1') Then
2                          case control is
3
4                              when "0000" => --1
5                                 A_00 <= phi_00;
6                                 A_01 <= phi_01;
7                                 A_10 <= phi_10;
8                                 A_11 <= phi_11;
9
10                                B_00 <= reg0_00;
11                                B_01 <= reg0_01;
12                                B_10 <= reg0_10;
13                                B_11 <= reg0_11;...
```

3.6.4 Arithmetic Logic Unit. The Arithmetic Logic Unit (ALU) is the portion of the design that performs the actual calculations. The overall design consists of two ALUs working in parallel. Each ALU is capable of performing one of three tasks: matrix addition, matrix subtraction, and matrix multiplication. The matrix addition and matrix subtraction are performed by adding or subtracting corresponding elements of the matrices. For two matrices of size $m \times n$ their addition or subtraction produces an $m \times n$ matrix result. The matrix multiplication is accomplished as shown in Figure 3.7 on page 40. Also see Figure 3.8 on page 40 for the matrix multiplication process.

3.6.5 Newton-Raphson Reciprocal. The behavioral model of the Kalman filter in VHDL is not required to synthesize; this allows for a simpler implementation. As mentioned in Section 3.5.4 on page 25 it is only necessary to carefully bit shift in order to achieve a correct outcome when performing the reciprocal. For synthesis purposes an RTL description is required to not only allow for optimizations but to also

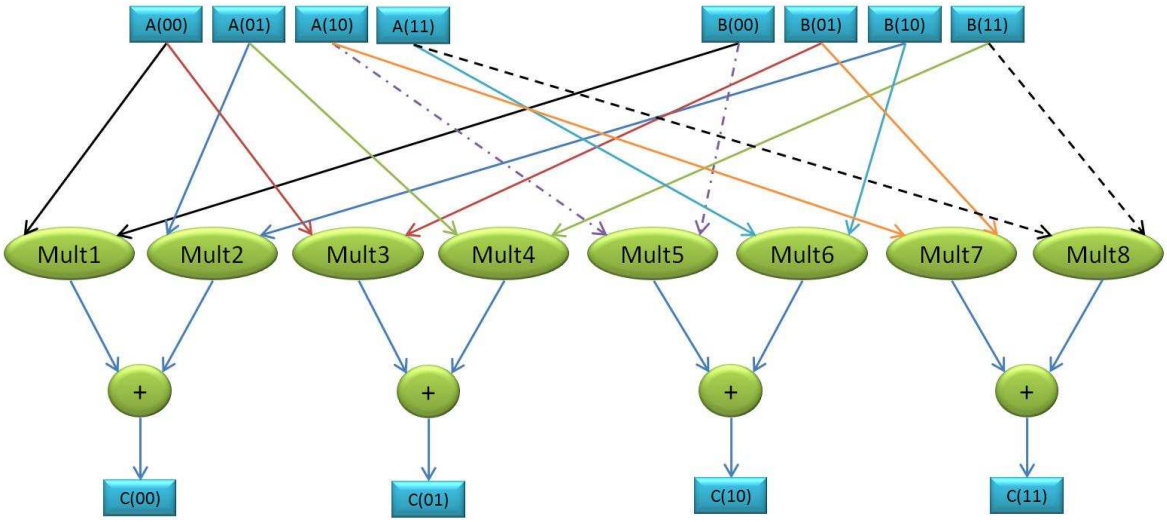


Figure 3.7: Graphical representation of matrix multiplication.

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$$
$$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11}$$
$$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10}$$
$$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11}$$

Figure 3.8: 2×2 matrix multiplication.

implement a synthesizable reciprocation function. The chosen method for reciprocation is called Newton-Raphson Division. The Newton-Raphson division method lends itself well to the special case of finding the reciprocal. There are several ways to perform division in digital designs. The methods can be classified as either a fast division algorithm or a slow division algorithm. The slow division methods produce one digit of the quotient per iteration while the fast division methods start with an estimate and arrive at a quotient through multiple iterations of an algorithm; doubling the number of correct bits each time through. In order to achieve maximum speed in the overall design, the slow methods were not considered. The two methods considered for fast division are the Newton-Raphson method and the Goldschmidt method. Both

methods are iterative and require initial approximations. The Goldschmidt method is a variation of Newton-Raphson that lends itself well to pipelining [4]. Because pipelining of the divider is unnecessary due to data hazards, it was decided to use the Newton-Raphson method which directly computes the reciprocal. Although the method can be used to find the quotient, the process first finds the reciprocal of the divisor and then multiplies the reciprocal by the dividend to produce the quotient. This method converges to the reciprocal quadratically [11]. For the special case of:

$$\frac{1}{H \times P(:,:,k) \times H' + R} \tag{3.9}$$

the Newton-Raphson method is used to calculate the reciprocal which precludes the final step of multiplying the reciprocal by the dividend.

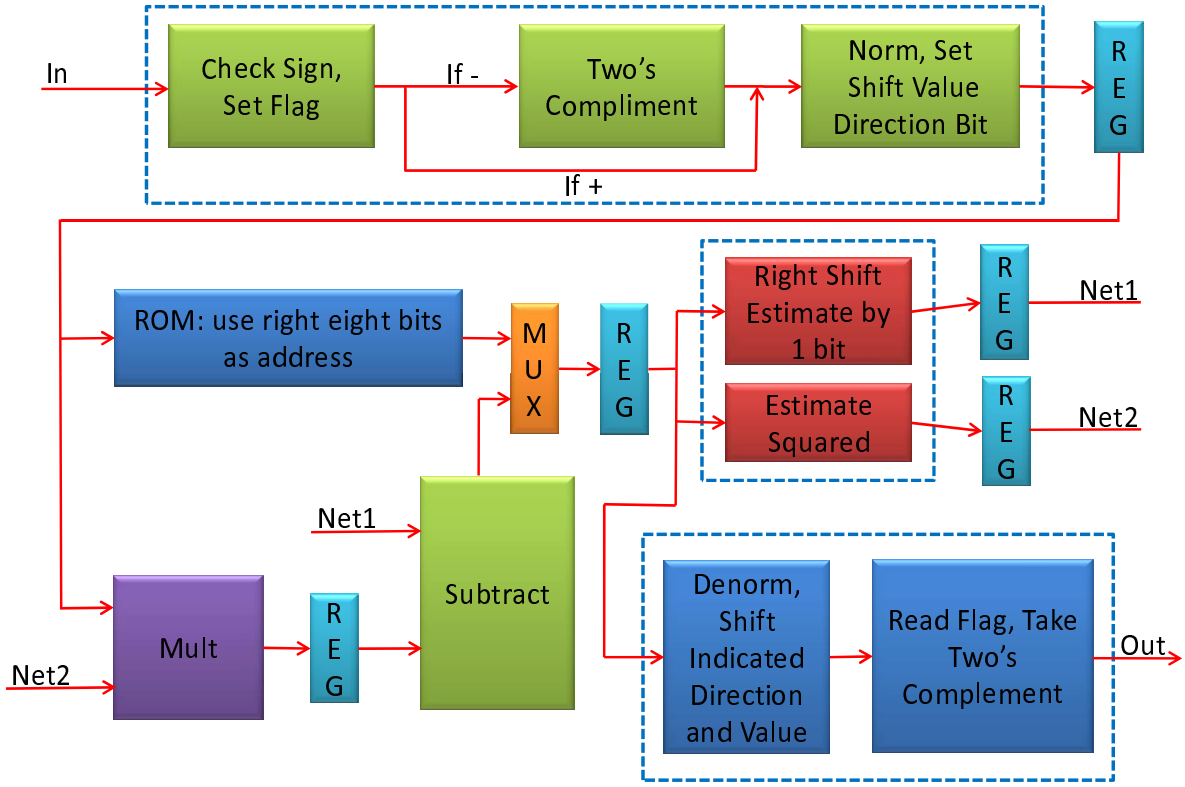


Figure 3.9: Top level schematic of the Newton-Raphson reciprocal VHDL model.

3.6.6 Newton-Raphson Division Algorithm. The Newton-Raphson iteration is used to approximate the root of a non-linear function. For a well behaved function $f(x)$ let r be a root of $f(x) = 0$. Now let x_0 be an estimate of r where $r = x_0 + h$; h is the difference between the estimate and truth. Assuming the estimate is sufficiently accurate, it can be concluded that by linear approximation:

$$0 = f(r) = f(x_0 + h) \approx f(x_0) + h'f(x_0) \tag{3.10}$$

And therefore,

$$h \approx -\frac{f(x_0)}{f'(x_0)} \tag{3.11}$$

It follows that

$$x_0 + h \approx x_0 - \frac{f(x_0)}{f'(x_0)} \tag{3.12}$$

and therefore

$$r \approx x_0 - \frac{f(x_0)}{f'(x_0)} \tag{3.13}$$

r then becomes a new and improved estimate x_1. In general this can be stated as:

$$x_{i+1} = x_i - \frac{f(x_0)}{f'(x_0)} \tag{3.14}$$

For the case of the reciprocal

$$f(x) = \frac{1}{x} - D \tag{3.15}$$

$$f'(x) = -\frac{1}{x^2} \tag{3.16}$$

Substituting equations 3.15 and 3.16 into equation 3.14 yields

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - D}{\frac{1}{x_i^2}}$$

$$\Downarrow$$

$$x_{i+1} = 2x_i - x_i^2 D \tag{3.17}$$

Here is a simple example of this algorithm. For $\frac{1}{2}$, a relatively bad estimate of 0.3 can be made. This means that $D = 2$ and x_i starts at 0.3.

$$2(0.3) - (0.3)^2(2) = 0.42$$

$$2(0.42) - (0.42)^2(2) = 0.4872$$

$$2(0.4872) - (0.4872)^2(2) = 0.49967232$$

$$2(0.49967232) - (0.49967232)^2(2) = 0.49999978$$

As can be seen, in only four iterations the Newton-Raphson approximation to the reciprocal problem of $\frac{1}{2}$ rapidly approaches the correct answer. Even with the bad first estimate the algorithm converges quadratically requiring relatively few iterations. The maximum relative error for any k-bits-in m-bits-out ROM reciprocal table is the result of the relative errors obtained between the actual reciprocal $\frac{1}{x}$ and the lookup table value of x for $1 \leq x < 2$. A table precision of α bits, i.e. α entries will always yield a maximum error of at most $\frac{1}{2^\alpha}$ [1]. As a worst case example, if 16 bits of accuracy is required in an initial estimate and only one bit is available for lookup purposes then the worst case error will be $\frac{1}{2^1} = 0.5$. Using this worst case scenario in the example above yields an initial estimate of 1.0. Therefore, $D = 2$ and x_i starts at 1.0.

$$2(1.0) - (1.0)^2(2) = 0$$

$$2(0) - (0)^2(2) = 0$$

As can be seen, the equations do not converge. At least two bits are required when calculating the initial estimate in order to get convergence. As another example, if two bits are available to calculate or lookup the initial estimate then the worst case error will be $\frac{1}{2^2} = 0.25$. Once again, using the example from above; $D = 2$ and x_i starts at 1.

$$2(0.75) - (0.75)^2(2) = 0.375$$

$$2(0.375) - (0.375)^2(2) = 0.46875$$

$$2(0.46875) - (0.46875)^2(2) = 0.498046875$$

$$2(0.498046875) - (0.498046875)^2(2) = 0.499992370606$$

$$2(0.499992370606) - (0.499992370606)^2(2) = 0.499999999884$$

As can be seen, in order to achieve at least the same level of precision as seen in the first example, one more iteration is required. Note, with the extra iteration, the level of precision of the last example exceeds that of the first.

3.6.7 Initial Estimate. The number of iterations required to converge to an acceptable answer using the Newton-Raphson reciprocal algorithm described above depends on the accuracy of the approximation [5]. Two algorithms were tested for calculating the initial estimates. The first algorithms tested is Equation 3.18.

$$D_{stored}^{-1} = \frac{1}{D' + 2^{-M-1}} + 2^{-M-2} \tag{3.18}$$

Where, $D' = [1.d_1d_2...d_M]$ and $(M+1)$ is the accuracy in bits of the initial approximation [5]. Despite providing accurate estimates for the Newton-Raphson reciprocal function the second algorithm tested provided even greater accuracy. The algorithm chosen for generating estimates for the Newton-Raphson reciprocal can bee seen in Equation 3.19.

$$C_{stored}^{-1} = \frac{1}{\frac{(2+\frac{C'}{2^{\alpha}-1})+\frac{1}{2^{\alpha}})}{2}} \tag{3.19}$$

Where $C' = [d_1d_2...d_M]$; i.e. D' is the address into the memory and the first eight bits after the radix of the normalized number for which the reciprocal is being calculated. The number of bits contained in C' is called α. Eight bits were chosen for C' to keep the size of the ROM as small as possible while maintaining the desired accuracy. The eight bit address into the ROM corresponds to a ROM with 256 entries.

3.6.8 Newton-Raphson Hardware Implementation. The top-level view of the hardware implementation of the Newton-Raphson reciprocal can be seen in Figure 3.9 on page 41. The Newton-Raphson reciprocal only calculates the reciprocal for a single number and not an entire matrix. As can be seen, it is broken up into three main stages:

1. Preliminary sign checking and normalization.
2. Approximation lookup in ROM.
3. Iterative calculations.

The following sub-sections contain explanations of the three stages mentioned above.

3.6.8.1 Preliminary Sign Checking and Normalization. The first step, at the input point, is to check to see if the number x for which you want to know the reciprocal $\frac{1}{x}$ is positive or negative. If the number is negative then a flag is set to later indicate that the number is negative. If the number was negative then it is made positive by taking the two's complement. Finally the number must be normalized to produce a number x where $1 \leq x < 2$ which is required for approximation lookup in the lookup table. Normalization is done by shifting the number either left or right. For example, the 8-bit number 0100.1000_2 is normalized by right shifting by two bits thus producing 0001.0010_2. A direction bit for de-normalizing is set as well as the shift quantity value. The direction bit indicates the direction of the original shift and the shift quantity designates how many bits to shift.

3.6.8.2 Getting the Estimate: Lookup-Table. The normalized value is then passed on to the ROM as well as the Multiplier. The ROM uses the first eight bits from the right side of the radix of the normalized value as an address into the ROM. The approximation to the reciprocal is passed through a multiplexor and into a register. It is at this point that the iterative portion of the algorithm begins.

3.6.8.3 Iterative Portion. The iterative portion performs:

$$x_{i+1} = 2x_i - x_i^2 D$$

Where D is the denominator and x_i is the current estimate for the reciprocal. Three iterations of the above equation are performed producing the reciprocal in its normalized form. At this point it is necessary to de-normalize the number by shifting the appropriate direction by the appropriate number of bits. If the sign bit, set in the preliminary sign checking and normalization portion, indicates the number was negative a two's complement conversion is performed and the final output is ready.

3.7 Two Dimensional Implementation

The Kalman filter design discussed previously is a one-dimensional Kalman filter. That is, linear position is input into the filter and an estimate of the linear position and linear velocity is output. The number representing the velocity has a magnitude represented by the absolute value of the number and a direction indicated by the sign of the number.

For target tracking purposes, a one dimensional Kalman filter would be of limited utility. However, combining two filters together and appropriately combining the velocities produces an efficient real-world position and velocity estimator of much greater utility.

3.7.1 Combination of Two Linear Filters. A two-dimensional Kalman filter is created by combining two one-dimensional filters together. Each filter takes as input either the x coordinate or the y coordinate from a cartesian plane. Each of the Kalman filters output a separate position and velocity. The position value represents where along the associated axis the target is located. The velocities must be combined using Equation 3.20.

$$velocity = \sqrt{v_x^2 + v_y^2} \tag{3.20}$$

Equation 3.20 shows that the combined velocities of the one-dimensional Kalman filters is equal to the square-root of the sum of the squares; Pythagorean's theorem.

3.7.1.1 Hardware Implementation. Implementing equation 3.20 requires three main steps:

1. Squaring of the one-dimensional velocities.
2. Addition of the squares.
3. Taking the square root.

In VHDL, multiplication is considered a basic operator and is part of the VHDL synthesis package numeric_std. Addition, likewise, is also include in numeric_std. Neither of these operators require special programming in order to be synthesizable. However, the square-root operator is not part of the numeric_std synthesis package and therefor presents a designer with the difficult task of implementation.

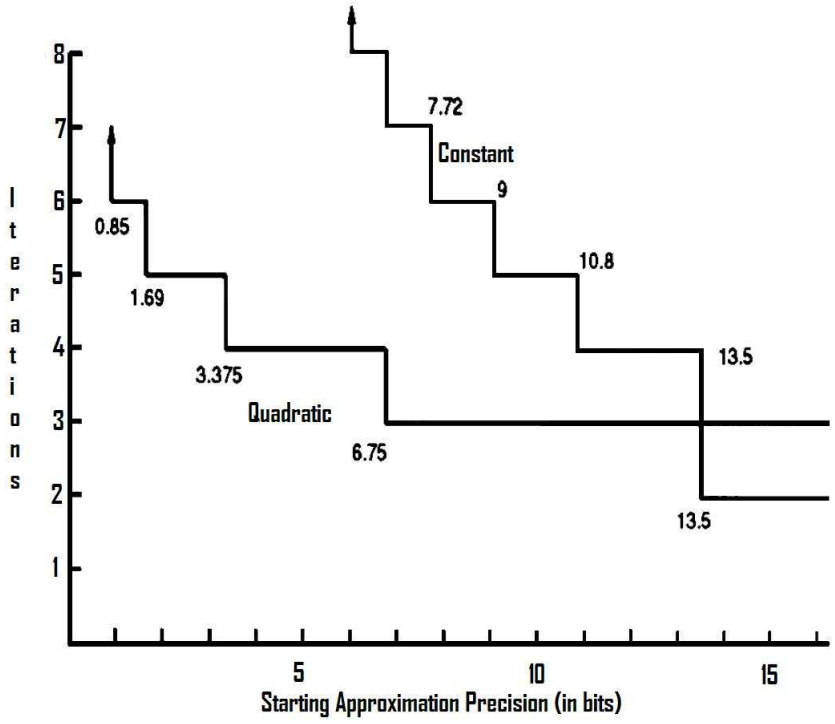


Figure 3.10: Number of iterations versus starting approximation.

Considering the requirement of this thesis to produce fast, small circuits, a square-root method using the Newton-Raphson method similar to that used to find the reciprocal is implemented. The iterative equation for finding the reciprocal of the square-root is [8]:

$$x_{i+1} = \frac{x_i(3 - Dx_i^2)}{2} \tag{3.21}$$

Where D is the number for which it is desired to find the square root; i.e. $1/\sqrt{D}$. As mentioned above, Equation 3.21, finds the reciprocal of the square-root. In order to find the square-root from its reciprocal it is required to multiply by D.

$$\frac{1}{\sqrt{D}}D = \sqrt{D}$$

As with finding the reciprocal, the initial estimate will, in part, determine the number of iterations of equation 3.21 that must be performed to achieve a desired accuracy. Figure 3.10 on page 47 shows approximately how many iterations of Equation 3.21 are required to achieve 53 bits of accuracy [16]. In the case of the two dimensional Kalman filter, excess clock cycles that arise from calculation of the linear filters allows for five iterations with clock cycles to spare.

Normalization of the square-root operand is required in order to both initially populate lookup tables as well as for accessing estimates from those tables. Two types of normalization are required. If $\alpha = log_2 A$ and A is the value of the highest order bit then the first type of normalization is for even numbered α and the second type is for odd numbered α. For example, the binary number 1010.1100_2 has a highest ordered bit that has a value of $2^3 = 8$; because α is equal to an odd number, (3), to normalize the binary number it must be right-shifted $\alpha + 1$ times to produce a number of the form $0.1xxxxxxx_2$. Binary numbers greater than or equal to one and with an even numbered α must be right-shifted α times to produce a number of the form $1.xxxxxxxx_2$. Binary numbers greater than or equal to one and with an

odd numbered α must be right-shifted $\alpha + 1$ times to produce a number of the form $0.1xxxxxx_2$. Binary numbers less than or equal to one and with an even numbered α must be left-shifted α times to produce a number of the form $1.xxxxxxx_2$. Binary numbers less than or equal to one and with an odd numbered α must be left-shifted $\alpha - 1$ times to produce a number of the form $0.1xxxxxx_2$.

Equation 3.22 shows the method for calculating an estimate for a normalized number with an even shift value where $D'_e = [1.d_1d_2...d_8]$, i.e. the first eight bits of the fraction portion of the normalized number.

$$\frac{1}{\sqrt{\frac{2(1+\frac{D'_e}{2^8})+\frac{1}{2^8}}{2}}} \tag{3.22}$$

Equation 3.23 shows the method for calculating an estimate for a normalized number with an odd shift value where $D'_o = [0.1d_1d_2...d_8]$, i.e. the first eight bits after the first 1 of the fraction portion of the normalized number.

$$\frac{1}{\sqrt{\frac{2(0.5+\frac{D'_o}{2^9})+\frac{1}{2^9}}{2}}} \tag{3.23}$$

Listing III.12:

```
1  Case address Is
2
3                  When "00000000" => --0.9990248656871403
4                       estimate_intermediate := "1111111111000000";
5
6                  When "00000001" => --0.9970831245596092
7                       estimate_intermediate := "1111111101000000";
8
9                  When "00000010" => --0.9951526617137967
10                      estimate_intermediate := "1111111011000010";
```

```
11
12                When "00000011" => --0.9932333683907213
13                     estimate_intermediate := "1111111001000100";
```

Listing III.12 shows four approximation entries for the even α ROM. The approximations are calculated using Equation 3.22 on page 49.

Listing III.13:

```
1  Case address Is
2
3                When "00000000" => --1.4128345142027134
4                     estimate_intermediate := "0110100110101111";
5
6                When "00000001" => --1.4100884775655416
7                     estimate_intermediate := "0110100011111011";
8
9                When "00000010" => --1.407358390827336
10                    estimate_intermediate := "0110100001001000";
11
12               When "00000011" => --1.4046441001796708
13                    estimate_intermediate := "0110011110010110";
```

Listing III.12 shows four approximation entries for the odd α ROM. The approximations were calculated using Equation 3.23 on page 49.

After normalization, and estimate lookup, the iterative portion of the process begins. Five iterations of Equation 3.21 on page 48 are performed followed by multiplication of the reciprocal square-root by the square-root operand. It is this step where the actual square-root is calculated. The final step is that of denormalization, where the answer is shifted in the opposite direction as in normalization. The number of shifts for denormalization is one-half the number of shifts required for normalization. It is now that the combined two-dimensional velocity has been fully calculated and is output.

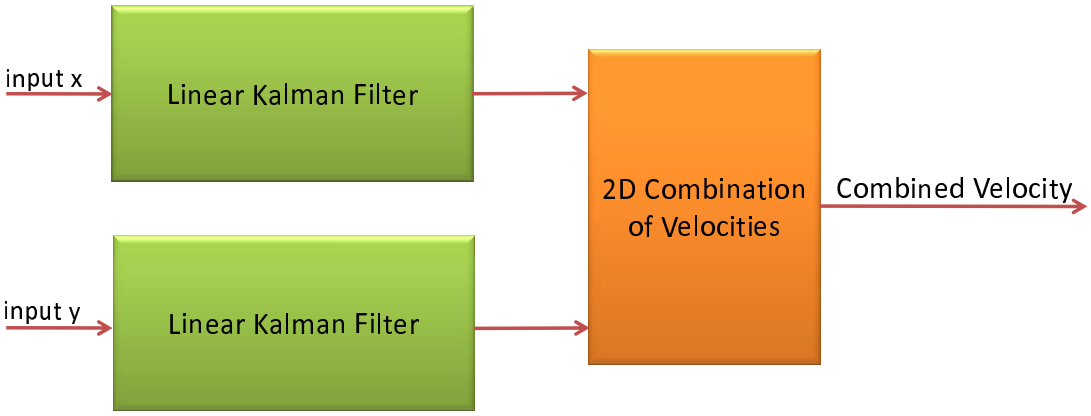


Figure 3.11: Top level schematic of the two-dimensional implementation or combination of linear Kalman filters.

3.8 Design Flexibility

It is desired to make the design flexible. Flexibility allows a designer that wants to utilize the VHDL Kalman filter to designate the value and bit width of the Kalman filter parameters. A Kalman filter has various parameters that affect its overall behavior. For example, process noise covariance Q and measurement noise covariance R can be tuned according a system model ,producing the desired behavior of the filter. In order to give a designer this kind of control and flexibility while still producing synthesizable code an alternate programming language to VHDL is needed. JAVA was chosen for its ability to execute on most multipurpose computer systems.

It is assumed that the reader has at least a basic understanding of programming and JAVA. The JAVA code consists of four packages:

1. Decimal to binary converter: DecimalToBinary.java
2. Code generator: CodeObject.java
3. Kalman filter parameter initializers: Initializers.java
4. A main function: Kf_main.java

3.8.1 Decimal to Binary Converter. The decimal to binary converter allows for the automatic conversion of decimal numbers. Three user defined integers,

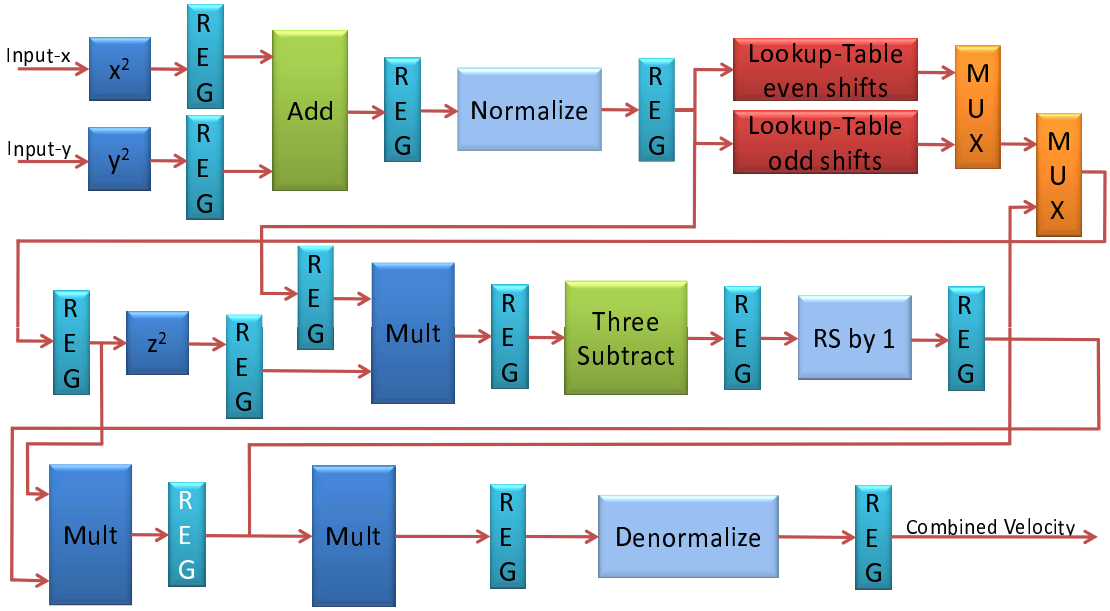


Figure 3.12: Top level schematic of the primary module for the two-dimensional implementation or combination of linear Kalman filters.

data_size, **fracton_size**, and **rom_estimate_size** determine the width of the number in binary and the size of the fraction portion of the number. These three integers are passed to the method when an *object* of type DecimalToBinary is created. The returned value is a string of ones and zeros that is the binary representation of the decimal number . The converter uses various JAVA *methods* to generate a binary two's complement numbers the width of **data_size** or the width of **rom_estimate_size**.

The integer **rom_estimate_size** is used to indicate the size of the data inside the ROM lookup table that is used in the Newton-Raphson reciprocal calculation. It will typically be significantly smaller than **data_size** as it is only an estimate. The smaller size of the estimates also minimizes the size of the ROM. For the two designs tested in this thesis, ROM estimates of size 8 bits and 16 bits are used for the 32-bit and 64-bit versions respectively.

3.8.2 Code Generator. The code generator generates all of the Kalman filter VHDL code. Inside the main function, **Kf_main.jave**, an *object* of type **CodeObject** is created and four integers are passed into it: **rom_estimate_size**, **data_size**,

Listing III.14:

```
1  double dt = 0.1;
2  double R = 10.0;
3  double Q = 100.0;
4  double[] G = {0.0, 1.0};
5  double[] B = {0.0,1.0};
6  double[] Bd = {0.005, 0.1};
7  double[] H = {1.0, 0.0};
8  double[] H_prime = {1.0, 0.0};
9  double[] F = {0.0, 1.0, 0.0, 0.0};
10 double[] phi = {1.0, 0.1, 0.0, 1.0};
11 double[] phi_prime = {1.0, 0.0, 0.1, 1.0};
12 double[] Qd = {0.0333, 0.5, 0.5, 10.0};
13 double[] Gd = {1.0, 0.0, 0.0, 1.0};
14 double[] x = {1.0, 1.0};
15 double[] P = {0.25, 0.0, 0.0, 0.25};
```

fraction_size, and **array_size**. The new *object* of type **CodeObject** can then be used to call the method inside **CodeObject.java** that creates the VHDL modules. Table 3.6 on page 53 shows the *methods* within CodeObject:

Table 3.6: These CodeObject *methods* each generate a corresponding VHDL file.

kf_top();	kf_top_tb();	add_sub_behavioral();
sub_behavioral();	ALU();	controller();
KF_RTL_top();	KF_RTL_top_tb();	mem();
mux_4_to_2();	mux_2_to_1();	mult_behavioral();
mux();	reciprocal_top();	reciprocal_stage1();
reciprocal_stage3();	reciprocal_stage6();	register_ALU();
NR_LT_ROM();		

The user adjustable Kalman filter parameters are initialized inside **CodeObject.java**. See Listing III.14 for a list of these parameters. Many of these parameters are matrices and are created as type array in JAVA. The integer **array_size** is passed into a *method* called **Array_size_initializer** where it is used to designate the size of the arrays.

3.8.3 Initializers. This file creates the *objects* seen in Listing III.14. As mentioned above, the user settable integer constant that determines the size of the arrays is called **array_size**. This constant can be set by the user inside *main*.

3.8.4 Main. The *method main* is where the integer constants: **rom_estimate_size**, **data_size**, **fracton_size**, and **array_size** are set. Also, it is inside *main* where an object(s) of type **CodeObject** is created. That **CodeObject** is then used to call the various methods inside **CodeObject.java** which then writes the VHDL Kalman filter to file.

IV. Testing and Evaluation

This chapter discusses the testing approach, test application, and test results.

4.1 Testing Approach

Testing of the Kalman filter VHDL model was performed to verify function of the model in comparison with output data produced by the `Matlab`® version of the filter. Identical inputs were run through both versions of the filter and the outputs compared. Input vector $z(:,k)$ consists of 500 pseudo-random noise-corrupted measurements. The input values as generated in Matlab have a standard deviation of 323.9967 indicating a wide range of values. The values range from a maximum of 452.907548 to a minimum of -856.481355. Two versions of the VHDL Kalman filter were tested: a 32-bit version and a 64-bit version.

4.1.1 The Test Bench. The 32-bit and 64-bit test benches were created using JAVA to populate each of them with their fixed point radix binary numbers. The test bench consists of 500 inputs z that are cycled through the VHDL Kalman filter to produce 500 position data and 500 velocity data. A simulation was run in Mentor Graphics `ModelSim`® SE Plus 6.3c revision 2007.09 for both the 32-bit and 64-bit versions of the VHDL Kalman filter with their respective test benches. A *list* of the output was created from the wave diagram and exported to a file. The binary outputs were then converted to integers using a JAVA binary-to-decimal converter written specifically for this thesis. The JAVA binary-to-decimal converter produced a text file containing the integer version of the `ModelSim`® output that was then imported into a spreadsheet for analysis.

4.1.2 Analysis. Analysis consisted of a look at possible error sources, followed by calculation of the standard deviation for the difference and a percentage-difference between the output produced by `Matlab`® and the `ModelSim`® simulation output for the VHDL Kalman filter.

4.1.2.1 Error Analysis. Errors can arise from various sources. One way that error can be introduced is in normalization of numbers. The reciprocal square-root function required normalization of the input in order to use the even and odd ROM lookup tables. The worst case scenario is that a 32-bit binary number with a 16-bit fraction of the form

$$0111111111111111.1111111111111111_2$$

is normalized by right shifting by 14 bits. The normalized number would look like:

$$0000000000000001.1111111111111111_2$$

which means the right 14 bits of the fraction are lost.

$$\frac{2^{14}}{2^{16}} = +0.25 \tag{4.1}$$

Equation 4.1 shows the maximum error that might occur due to normalization. To avoid this error, padding of the numbers prior to normalization would preserve the accuracy. The required bit-padding would be the size of the fraction portion minus two divided by two. If the number of bits to be padded is b_+ and the number of bits in the fraction is x then:

$$b_+ = \frac{x-2}{2}$$

The divide by two is due to the fact that when performing a reciprocal square-root, denormalization requires a shift that is half the value of the normalization shift and in the opposite direction.

For the reciprocal function used in the Kalman filter, both normalization and denormalization bit shifts are in the same direction and of the same magnitude. This means that no error will occur due to the normalization process. For example, if the binary number $0111111111111111.1111111111111111_2$ is normalized it takes on

the form $0000000000000001.1111111111111111_2$. In this example, 14 bits were shifted and lost. However, because denormalization for the reciprocal function requires a shift in the same direction and of the same magnitude as normalization, any bit-padding that would have preserved the bits is lost when converting the number back to its original format of 32 bits. To further demonstrate this, consider the following example.

$$\frac{1}{0111111111111111.1111111111111111_2} = 0.0000000000000010_2$$

Now the reciprocal for the normalized number is found. Normalization required a right shift by 14 bits.

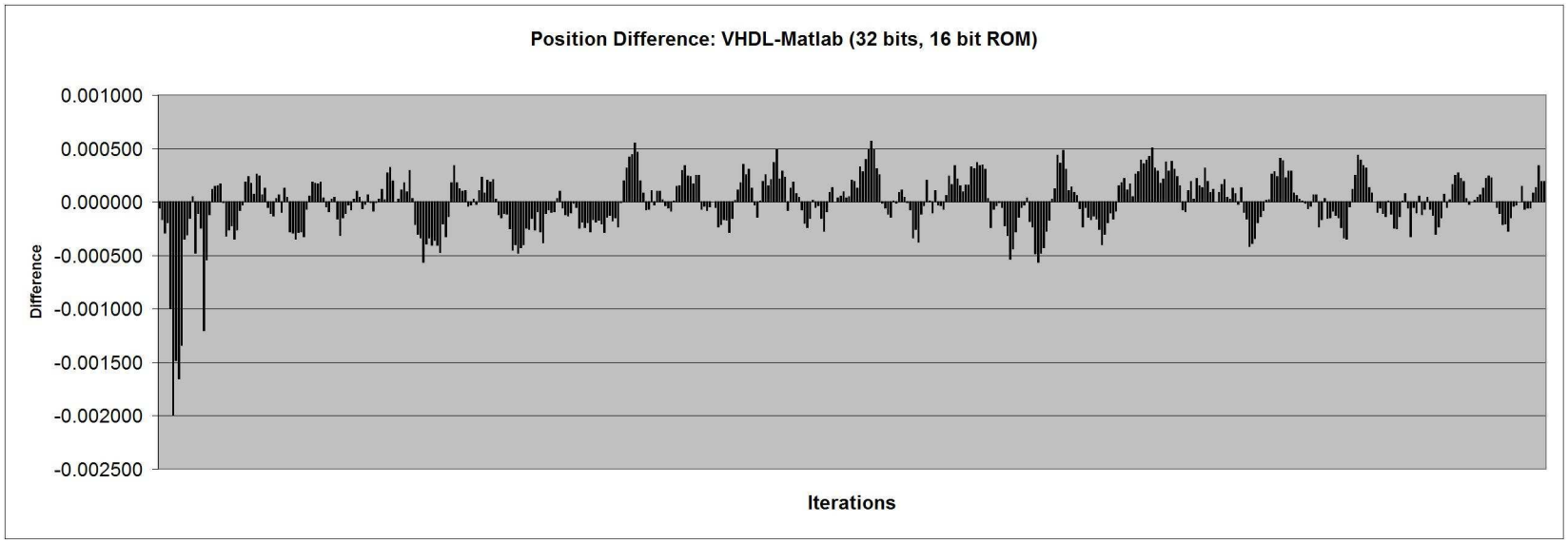
$$\frac{1}{0000000000000001.1111111111111111_2} = 0.1000000000000000_2$$

Denormalizing the answer from the above equation produces the number:

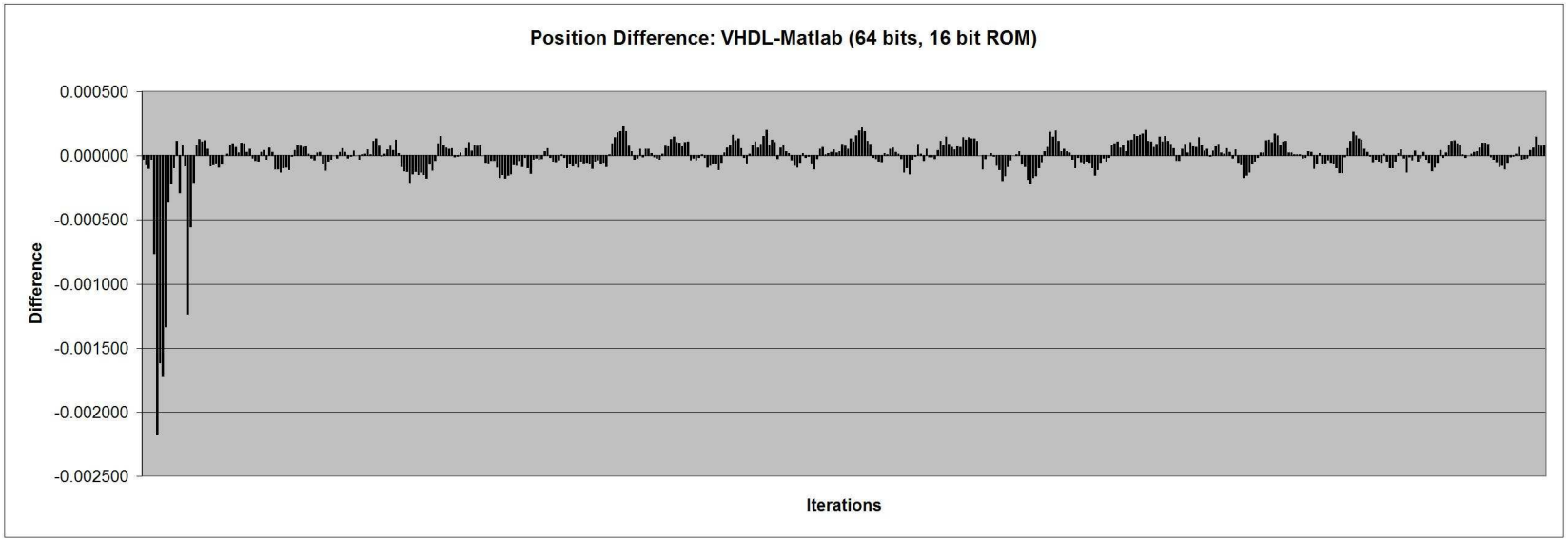
$$0.0000000000000010_2$$

which is exactly identical to the answer for the non-normalized reciprocal. There is no loss of data.

4.1.2.2 Difference Comparison. For the difference comparison the difference was taken between the output produced by `Matlab`® and the `ModelSim`® simulation output for the VHDL Kalman filter for all 1000 outputs(500 position outputs, and 500 velocity outputs). These differences for position are plotted for both the 32-bit and 64-bit test cases and can be seen in Figure 1(a) on page 58 and Figure 1(b) on page 58 respectively. The standard deviation was then calculated for the difference. See Figure 4.3 on page 61 for the standard deviation.



(a)



(b)

Figure 4.1: Difference taken between the VHDL Kalman filter output with 32-bit and 64-bit fixed point representations and the `Matlab`® Kalman filter output. There are 500 differences shown for each figure.

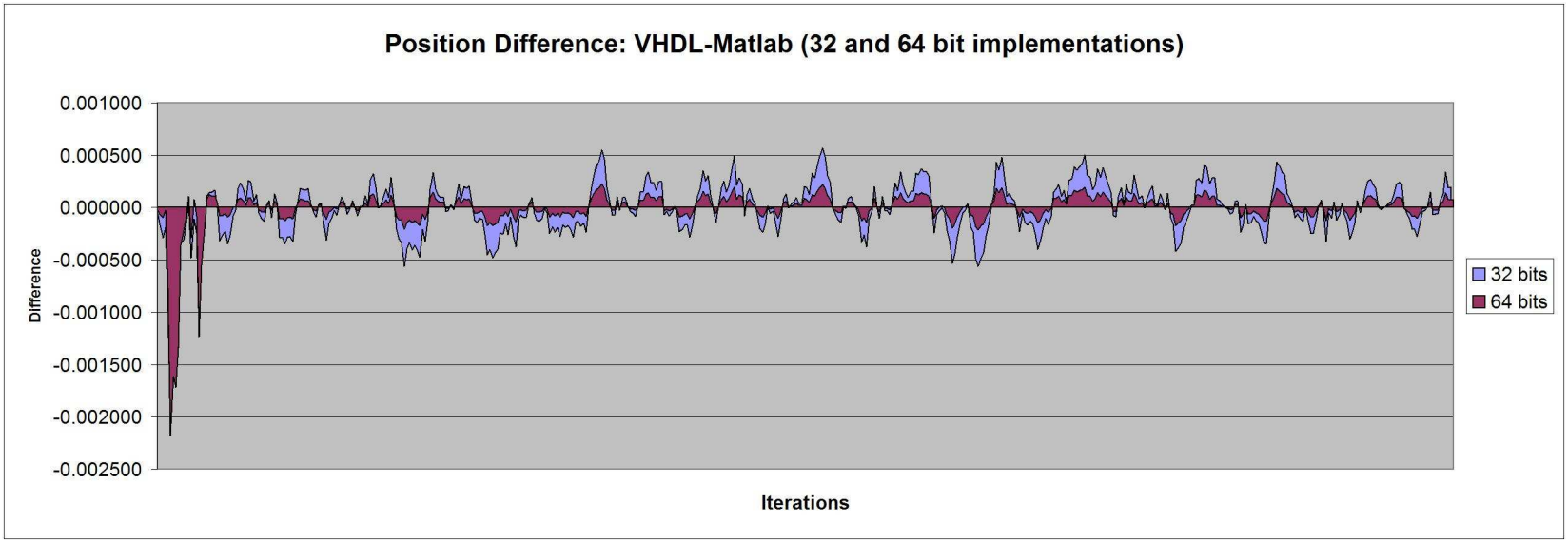


Figure 4.2: Difference taken between the VHDL Kalman filter output with 32-bit and 64-bit fixed point representations and the `Matlab`® Kalman filter output . There are 500 differences shown here.

4.1.2.3 Percentage Difference Comparison. The percentage difference was calculated by taking one minus the difference of the output produced by `Matlab`® and the `ModelSim`® simulation output for the VHDL Kalman filter for all 1000 outputs(500 position outputs, and 500 velocity outputs). The standard deviation was then calculated for the percentage difference. See Figure 4.3 on page 61 for the standard deviation.

4.1.2.4 Difference Comparison Standard Deviation. The standard deviations that was calculated for the 32-bit and 64-bit test cases can be seen in Figure 4.3 on page 61. The extremely small deviation from the mean indicates that the VHDL Kalman filter output is a very good approximation to the `Matlab`® implementation of the Kalman filter. As expected the standard deviation for the 64-bit implementation is smaller than that for the 32-bit implementation. Figure 4.2 on page 59 shows the difference comparison for both tests. As can be seen, the 64-bit implementation varied less overall with the `Matlab`® Kalman filter implementation output.

The large spikes seen in the first iterations of Figure 1(a) and Figure 1(b) on page 58 are due to the Kalman filter being in a transient state. This transient state is caused by the initial default estimates of the filter not being accurate estimates of the state of the system. These initial estimates are not intended or expected to be accurate, rather, the filter must iterate multiple times to reach a steady state. Simulations were performed in `Matlab`® in which a single position measurement was set as the filter input and held for 100 iterations. The filter achieved steady-state on the 55^{th} iteration when the velocity went to zero and the position, as output by the filter, became equal to the input position. The number of iterations required to achieve steady-state is dependent on the filter parameters and therefore can be changed to fit design requirements.

Simulations were run for the two-dimensional implementation that resulted in a standard deviation of 0.021272. As expected, this was greater than the standard

Difference

	Position 500 Samples (16 bit ROM)	Velocity 500 Samples (16 bit ROM)
32bits	0.000269	0.003725
64bits	0.000190	0.002407

Percentage Difference

	Position 500 Samples (16 bit ROM)	Velocity 500 Samples (16 bit ROM)
32bits	0.006000%	0.127800%
64bits	0.006100%	0.338500%

Figure 4.3: Standard deviation for the difference and percentage-difference of the VHDL Kalman filter output (with a 64-bit and 32-bit fixed point representation) and the `Matlab`® Kalman filter output.

deviation for the one-dimensional implementation due to error as discussed in subsection 4.1.2.1. Figure 4.4 on page 62 shows the simulation results as a difference between the two-dimensional VHDL Kalman filter and the two-dimensional `Matlab`® Kalman filter as calculated using Microsoft Excel. The large spikes were expected due to the normalization error. Note that the spikes do not exceed the maximum calculated error of 0.25.

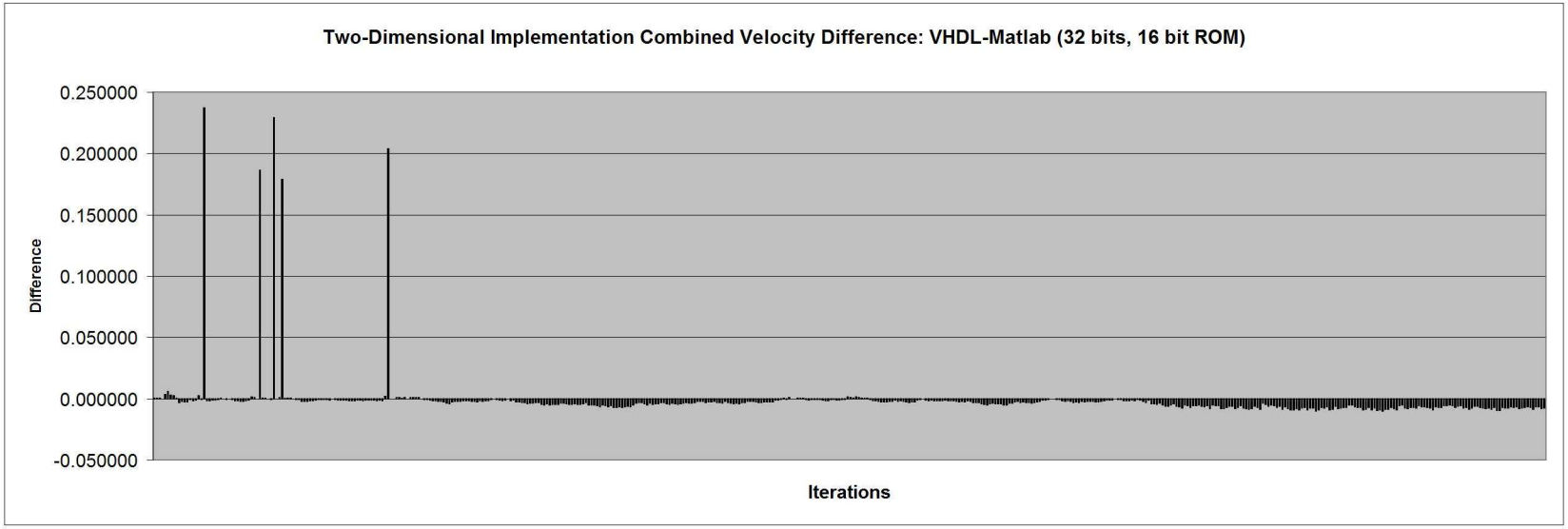


Figure 4.4: Difference taken between the VHDL Kalman filter output for the two-dimensional Kalman filter and the `Matlab`® Kalman filter two-dimensional output as calculated using Microsoft Excel. There are 500 differences shown here.

4.1.3 Speed and Area Analysis. Synthesis testing to determine maximum frequency at which the design will run was performed on a Xilinx Virtex-4 model 4vsx35ff668 speed grade -10. The same synthesis test determines resource usage on the FPGA. The synthesis was performed using Mentor Graphics' Precision RTL Synthesis 2006a.101. Tests were run for both 32-bit as well as 64-bit versions of the VHDL Kalman filter. Table 4.3 on page 65 shows the results for maximum frequency determination. As can be seen, an initial frequency was chosen for the 32-bit test of 100MHz. This test resulted in negative slack, indicating that the design would not run at 100MHz. A frequency recommended by the synthesis tool of 40MHz was then run, resulting in a relatively small positive slack indicating that the design will run at 40MHz. Slack values specify amounts of extra propagation delay available on the critical path. A small positive slack indicates that the design will just barely function at the specified frequency. As with the 32-bit tests, the 64-bit VHDL Kalman filter was first tested at 100MHz. This test resulted in negative slack, indicating that the design will not run at that frequency. The recommended frequency for the second test was 51MHz which resulted in a relatively small positive slack.

The results for area can be seen in Figure 4.1. The 32-bit implementation fit on the Virtex-4 utilizing 13.82% of the Configurable Logic Block (CLB) slices, however, the 64-bit implementation required 142.78% of the CLB slices.

Table 4.1: Synthesis test of the one-dimensional Kalman filter using Precision RTL Synthesis with the Virtex-4 4vsx35ff668 at speed grade -10.

Test	CLB Slices	IOs	DSP48s
32-bit test 100MHz	13.82%	22.32%	37.50%
32-bit test 40MHz	13.82%	22.32%	37.50%
64-bit test 100MHz	142.78%	43.75%	93.75%
64-bit test 51MHz	142.78%	43.75%	93.75%

Further testing was done for the 64-bit model due to the area requirements. Because the Virtex-4 is too small to accommodate the 64-bit model, the Xilinx Virtex-5 model 5vsx95tff1136 speed grade -3 was chosen instead. Table 4.4 on page 65 shows

the frequency results for the two tests performed and Table 4.2 on page 64 shows the results for resource usage.

Table 4.2: Synthesis test of the one-dimensional Kalman filter using Precision RTL Synthesis with the Virtex-5 5vsx95tff1136 at speed grade -3.

Test	CLB Slices	IOs	DSP48s
64-bit test 100MHz	16.51%	30.63%	30.94%
64-bit test 50MHz	16.51%	30.63%	30.94%

In reference to Table 4.3 on page 65, for the case of the 32-bit representation, 40MHz equates to a clock period of 0.000000025 seconds or 25 ns. This means that at 49 clock-cycles per iteration of the one-dimensional VHDL Kalman filter, a total of $1,225$ ns per iteration are required. Also, at 40MHz, 49 clock-cycles per iteration equates to $816,326$ iterations with each iteration corresponding to a potential target being tracked. If data is fed into the one-dimensional VHDL Kalman filter at a rate of 30 times per second for each target, position and velocity estimations for a total of approximately $27,210$ targets can be performed.

Table 4.3: Synthesis test of the one-dimensional Kalman filter using Precision RTL Synthesis with the Virtex-4 4vsx35ff668 at speed grade -10.

Test	Outcome	frequency	Period	Rec. Freq	Slack	Retiming
32-bit test 100MHz	Unsuccessful	100.00MHz	10 ns	40.56MHz	-14.65	Yes
32-bit test 40MHz	Successful	40.00MHz	25 ns	N/A	0.094	Yes
64-bit test 100MHz	Unsuccessful	100.00MHz	10 ns	51.98MHz	-9.24	Yes
64-bit test 51MHz	Successful	51.00MHz	19.60784 ns	N/A	0.362	Yes

Table 4.4: Synthesis test of the one-dimensional Kalman filter using Precision RTL Synthesis with the Virtex-5 5vsx95tff1136 at speed grade -3.

Test	Outcome	frequency	Period	Rec. Freq	Slack	Retiming
64-bit test 100MHz	Unsuccessful	100.00MHz	10 ns	50.00MHz	-3.68	Yes
64-bit test 50MHz	Successful	50.00MHz	20 ns	N/A	6.317	Yes

V. Conclusions and Future Work

This chapter contains conclusions, and future work.

5.1 Conclusions

As warfare evolves so to must the technology we use to fight wars. This thesis has provided a tool as well as a model for tool development that will allow electronics designers to more quickly and easily implement designs utilizing a Kalman filter. This helps to enable rapid fielding of force multiplying technology to the warfighter. It was demonstrated that a near-optimal VHDL Kalman filter model can be programmed onto an inexpensive FPGA for potential implementation into target tracking systems. Algorithms provided in `Matlab`® were implemented using modern FPGA synthesis tools to characterize, experiment, and generate flexible VHDL codes. Fundamental modular VHDL building blocks were formulated that can be used to optimize the target tracking algorithms while considering speed, power, and area for the targeted applications. The code flexibility means that parameters are adjustable allowing for experimentation of various combinations as to optimize or tune the algorithm for different applications.

5.2 Future Work

Future work for expansion of this thesis may include:

- Investigate the implementation of various multipliers for use in the VHDL Kalman filter.
- Error reduction through bit-padding in the Newton-Raphson square-root function.
- Data storage and swapping for multiple-target tracking.

Currently the VHDL Kalman filter implementation relies on the default process for multiplication as defined by the VHDL compiler. An investigation into various RTL

multiplication implementations should be done and a comparison made with the default implementations of the available VHDL compilers.

Currently there exists a maximum error of 0.25 that can occur due to the bit-shifting required for the normalization and denormalization process. It is proposed that future work be done to include bit-padding in order to preserve accuracy during this process.

The current implementation does not address the storage and swapping of data into and out of the VHDL Kalman filter. I propose an onboard memory with approximately 120,000 (6 x 20,000 targets), 32 or 64 (480,000 or 960,000 bytes) bit positions used to store x(:,k) and P(:,:,k). The address to access a targets information comes from outside the system in the form of a target number. The target acquisition program feeds both the target number(address) as well as the input z(:,k). It may also be necessary for the target locating program to send a reset signal along with a particular address that will reset the memory for the case where a target number is reassigned to another target or becomes inactive. In the case of a reassignment or a target number becoming inactive the x and P values must be reset. When the target number changes it is used as the address to access the last x and P value for that target. Those x and P values are loaded into the secondary memory or if the secondary memory is reduced to holding only constants then the x and P values can be used directly and the x(:,k+) and P(:,:,k+) will be saved into the primary memory. This scheme or a similar scheme would allow a single linear Kalman filter to track

Appendix A. Matlab Code

This appendix contains the Matlab® code written by Dr. Juan Vasquez.

Listing A.1: This is the main file that implements the Kalman Filter by calling both equiv_discrete.m as well as KF.m .
(appendix2/main.m)

```
% Main Routine - this is my code for generating truth
clear
clc
randn('seed',0);
cleanup=1;
gentruth=1; % 1=generate truth data, 0=do not
nruns=1;    % # of monte carlo runs
run=1;      % single run to plot
t_final=50;      % Final time
dt=.1;           % Sampling interval
t=0:dt:t_final;     % time vector
t_last=length(t);

% Model parameters
% Truth model - constant components
R_t = 10; % Measurement noise covariance
Q_t = 100; % Dynamics noise strength
G_t = [0; 1]; % Noise injection model
H_t = [1 0];% Measurement model
B_t = [0;1];
F_t = [0 1; 0 0];
[phi_t, Bd_t ,Qd_t]=equiv_discrete(F_t,B_t,G_t,Q_t,dt);
Gd_t=eye(2);

% True Initial conditions
% State=[x v]=[position velocity]
x_t(:,1)=[1 1];
z(1,1)=x_t(1,1);

% Filter Model parameters
R = 10;    % Measurement noise covariance
Q = 100; % Dynamics noise strength
G = [0; 1]; % Noise injection model
H = [1 0]; % Measurement model
B = [0;1];
F = [0 1; 0 0];
[phi, Bd ,Qd]=equiv_discrete(F,B,G,Q,dt);
Gd=eye(2);

% Noise injection matrices for truth model simulation
dynNoise_t = sqrt(Qd_t)';
measNoise_t = sqrt(R_t)';

% Begin Monte Carlo runs
```

```
45 if gentruth==1
46     for i=1:nruns
47         % Generate truth data
48         for k=2:t_last
49             % Generate noise-corrupted states
50             noise=dynNoise_t*randn(2,1);
51             x_t(:,k)= phi_t*x_t(:,k-1) + noise ;
52             % Generate noise-corrupted measurement
53             z(:,k) = H_t*x_t(:,k) + measNoise_t*randn;
54         end
55 
56         % Save truth data
57         eval(['save run' num2str(i) ' x_t z t_final dt'])
58     end
59 end
60 
61 for i=1:nruns
62     clear x
63     % Filter Initial conditions
64     % State=[x v]=[position velocity]
65     x(:,1)=[1 1];
66     P=.25*eye(2);
67     KF       % Kalman Filter subroutine
68 end
69 
70 for i=1:nruns
71     % Generate error statistics
72     eval(['load run' num2str(i) 'f' ])
73     eval(['load run' num2str(i) ])
74     for j=1:2   % 2 states
75         e(j,:,i)= x_t(j,:) - x(j,:);
76     end
77     res(:,:,i)=residual;
78 end
79 
80 % Ensemble stats
81 Me=mean(e,3);   % mean error over number of runs (dimension 3)
82 sigma_e=std(e,0,3); % standard dev of error over number of runs (...
       dimension 3)
83 % Since the noise stats are stationary, all of the gains and ...
       filter
84 % computed covariance values are the same for each run, so just ...
       use the
85 % last run data that was just loaded above
86 
87 % Temporal stats of ensemble data
88 Me_time=mean(Me,2)
89 sigma_e_time=mean(sigma_e,2);
90 
91 figure(1)
92 for j=1:2
93     subplot(2,1,j),plot(t,Me(j,:),'r',...
```

```
        t,Me(j,:)+ 3*sigma_e(j,:),'b',...
        t,Me(j,:)- 3*sigma_e(j,:),'b',...
        t,3*sigma_f(j,:),'m:',...
        t,-3*sigma_f(j,:),'m:')
end
figure(1),subplot(211)
title('State Estimates and Covariances [x    v ]')

% Load a specific single run data file and plot
eval(['load run' num2str(run) 'f' ])
eval(['load run' num2str(run) ])

figure(2) % last run data only
subplot(211),plot(t,x(1,:),'r');hold on;
plot(t,x_t(1,:));hold off;
title('Position')
subplot(212),plot(t,x(2,:),'r');hold on;
plot(t,x_t(2,:));hold off;
title('Velocity')

% Residual data
Mr=mean(res,3); % mean residual
sigma_r=std(res,0,3);   % std of residual
sigma_r_f(1,:)=sqrt(A(1,:));    % filter computed std of residual
figure(3)
plot(t,Mr(1,:),'r',...
    t,-3*sigma_r(1,:),'b',...
    t,3*sigma_r(1,:),'b',...
    t,-3*sigma_r_f,'m',...
    t,3*sigma_r_f,'m'),title('Residual')

% AANES Calculations
for i = 1:nruns
  for j = 1:t_last
    ness(j,i) = (e(:,j,i)'*inv(P(:,:,j))*e(:,j,i))/2;
  end
end
aness = sum(ness,2)/nruns;
figure(4)
plot(aness),title('ANESS');

% File Clean-Up
if cleanup==1
    for i=1:nruns
        eval(['delete run' num2str(i) 'f.mat' ])
        eval(['delete run' num2str(i) '.mat'])
    end
end
```

Listing A.2: This the Kalman Filter algorithm.(appendix2/KF.m)

```

% Load truth data and other parameters
```

```
3  eval(['load run' num2str(i) ])
4
5  for k=2:t_last
6      x(:,k)= phi*x(:,k-1);
7      P(:,:,k)= phi*P(:,:,k-1)*phi' + Qd;
8      A(:,k)= H*P(:,:,k)*H' + R;
9      K=P(:,:,k)*H'*(inv(A(:,k)));
10     residual(:,k)= z(:,k) - H*x(:,k);
11     x(:,k)= x(:,k) + K*residual(:,k);
12     P(:,:,k)= P(:,:,k) - K*H*P(:,:,k);
13     sigma_f(:,k)=sqrt(diag(P(:,:,k)));
14     KK(:,k)=K;
15 end % End time loop
16
17 % Save results
18 eval(['save run' num2str(i) 'f x P sigma_f A residual'])
```

This appendix contains the VHDL code that describes the Kalman Filter behaviorally.

Listing B.1:

```
1  ----------------------------------------------------------------------
2  --Entity : kf_top
3  --
4  --Description : This is a behavioral description of the Kalman
5  --Filter as described by Dr. Juan Vasquez in Matlab.
6  --
7  --Inputs :
8  --NAME        TYPE       DESC
9  --z_position Signed     input
10 --reset       Std_logic reset
11 --
12 --Outputs :
13 --NAME      TYPE    DESC
14 --position Signed position value
15 --velocity Signed velocity value
16 ----------------------------------------------------------------------
17
18 Library IEEE;
19 Use IEEE.std_logic_1164.All;
20 Use IEEE.numeric_std.All;
21
22 Package matrix_types Is
23 Type matrix_2x2 Is Array (1 to 2, 1 to 2) Of
24 Signed(31 Downto 0);--row x column
25
26 Type matrix_2x1 Is Array (1 to 2) Of
27 Signed(31 Downto 0);--row x column
28
29 Type matrix_1x2  Is Array (1 to 2) Of
30 Signed(31 Downto 0);--row x column
```

```
31
32 End Package matrix_types;
33
34 Library IEEE;
35
36 Use IEEE.std_logic_1164.All;
37 Use IEEE.numeric_std.All;
38 Use work.matrix_types.All;
39 Use std.textio.All;
40
41 Entity kf_top Is
42         Port(z_position : In Signed(31 Downto 0);
43                 reset : In Std_logic;
44                 position, velocity : Out Signed(31 Downto 0));
45 End kf_top;
46
47 Architecture kf_behav Of kf_top Is
48
49 --****************************************
50 --* Function to multiply two 2x2 matrices *
51 --****************************************
52 Function matrix_mult_2x2 (A,B: matrix_2x2)
53         Return matrix_2x2 Is
54         Variable result : matrix_2x2;
55         Variable func_temp1 : Signed(63 Downto 0) :=
56         (OTHERS => '0');
57         Begin--Begin function code.
58         For i In 1 to 2 Loop
59                 For L In 1 to 2 Loop
60                         For j In 1 to 2 Loop
61                                 func_temp1 := (A(i,j)*B(j,L)) + ...
                                       func_temp1;
62                         End Loop;
63                         result(i,L) := func_temp1(47 Downto 16);
64                         func_temp1 :=
```

```
65                          (OTHERS => '0');
66                  End Loop;
67          End Loop;
68          Return result;
69  End matrix_mult_2x2;
70
71  --************************************
72  --* Function to add two 2x2 matrices *
73  --************************************
74  Function matrix_add_2x2 (A,B: matrix_2x2)
75          Return matrix_2x2 Is
76          Variable result : matrix_2x2;
77          Begin--Begin function code.
78          For i In 1 to 2 Loop
79                  For j In 1 to 2 Loop
80                          result(i,j) := A(i,j)+B(i,j);
81                  End Loop;
82          End Loop;
83          Return result;
84  End matrix_add_2x2;
85
86  --*****************************************
87  --* Function to add a scalar to a 2x2 matrix *
88  --*****************************************
89  Function matrix_add_int_2x2 (A: matrix_2x2 ;B: Signed(31  Downto ...
        0))
90          Return matrix_2x2 Is
91          Variable result : matrix_2x2;
92          Begin--Begin function code.
93          For i In 1 to 2 Loop
94                  For j In 1 to 2 Loop
95                          result(i,j) := A(i,j)+B;
96                  End Loop;
97          End Loop;
98          Return result;
```

```
99  End matrix_add_int_2x2;
100
101 --************************************************
102 --* Function to multiply a 2x2 with a 2x1 matrix *
103 --************************************************
104 Function matrix_mult_2x2_2x1 (A: matrix_2x2 ;B: matrix_2x1)
105         Return matrix_2x1 Is
106         Variable result : matrix_2x1;
107         Variable func_temp1 : Signed(63 Downto 0) :=
108         (OTHERS => '0');
109         Begin--Begin function code.
110         For i In 1 to 2 Loop
111                 For j In 1 to 2 Loop
112                         func_temp1 := (A(i,j)*B(j)) + func_temp1;
113                 End Loop;
114                 result(i) := func_temp1(47 Downto 16);
115                 func_temp1 :=
116                 (OTHERS => '0');
117         End Loop;
118         Return result;
119 End matrix_mult_2x2_2x1;
120
121 --************************************************
122 --* Function to multiply a 1x2  with a 2x2 matrix *
123 --************************************************
124 Function matrix_mult_1x2_2x2 (A: matrix_1x2  ;B: matrix_2x2)
125         Return matrix_1x2  Is
126         Variable result : matrix_1x2 ;
127         Variable func_temp1 : Signed(63 Downto 0) :=
128         (OTHERS => '0');
129         Begin--Begin function code.
130         For L In 1 to 2 Loop
131                 For j In 1 to 2 Loop
132                         func_temp1 := (A(j)*B(j,L)) + func_temp1;
133                 End Loop;
```

```
134                 result(L) := func_temp1(47 Downto 16);
135                 func_temp1 :=
136                 (OTHERS => '0');
137         End Loop;
138         Return result;
139 End matrix_mult_1x2_2x2;
140
141 --************************************************
142 --* Function to multiply a 1x2  with a 2x1 matrix *
143 --************************************************
144 Function matrix_mult_1x2_2x1 (A: matrix_1x2  ;B: matrix_2x1)
145         Return Signed Is
146         Variable result : Signed(63 Downto 0) :=
147         (OTHERS => '0');
148         Begin--Begin function code.
149         For j In 1 to 2 Loop
150         result := (A(j)*B(j)) + result;
151         End Loop;
152         Return result(47 Downto 16);
153 End matrix_mult_1x2_2x1;
154
155 --************************************************
156 --* Function to multiply a 2x1 and a 1x2  matrix *
157 --************************************************
158 Function matrix_mult_2x1_1x2  (A: matrix_2x1 ;B: matrix_1x2 )
159         Return matrix_2x2 Is
160         Variable result : matrix_2x2;
161         Variable func_temp1 : Signed(63 Downto 0);
162         Begin--Begin function code.
163         For i In 1 to 2 Loop
164                 For L In 1 to 2 Loop
165                         func_temp1 := (A(i)*B(L));
166                         result(i,L) := func_temp1(47 Downto 16);
167                 End Loop;
168         End Loop;
```

```
169         Return result;
170 End matrix_mult_2x1_1x2 ;
171
172 --************************************************
173 --* Function to multiply a 2x1 matrix and a scalar *
174 --************************************************
175 Function matrix_mult_2x1_int (A: matrix_2x1 ;B: Signed(31 Downto ...
    0))
176         Return matrix_2x1 Is
177         Variable result : matrix_2x1;
178         Variable func_temp1 : Signed(63 Downto 0);
179         Begin--Begin function code.
180         For i In 1 to 2 Loop
181                 func_temp1 := A(i)*B;
182                 result(i) := func_temp1(47 Downto 16);
183         End Loop;
184         Return result;
185 End matrix_mult_2x1_int;
186
187 --***************************************
188 --* Function to add a 2x1 to a 2x1 matrix *
189 --***************************************
190 Function matrix_add_2x1_2x1 (A: matrix_2x1 ;B: matrix_2x1)
191         Return matrix_2x1 Is
192         Variable result : matrix_2x1;
193         Begin--Begin function code.
194         For i In 1 to 2 Loop
195                 result(i) := A(i)+B(i);
196         End Loop;
197         Return result;
198 End matrix_add_2x1_2x1;
199
200 --***************************************
201 --* Function to subtract two 2x2 matrices *
202 --***************************************
```

```
203 Function matrix_subtract_2x2 (A,B: matrix_2x2)
204         Return matrix_2x2 Is
205         Variable result : matrix_2x2;
206         Begin--Begin function code.
207         For i In 1 to 2 Loop
208                 For j In 1 to 2 Loop
209                         result(i,j) := A(i,j)-B(i,j);
210                 End Loop;
211         End Loop;
212         Return result;
213 End matrix_subtract_2x2;
214
215 --*********************************************************
216 --* Function to return the diagonal (diag) of a 2x2 matrix *
217 --*********************************************************
218 Function diag_2x2 (A: matrix_2x2)
219         Return matrix_2x1 Is
220         Variable result : matrix_2x1;
221         Begin--Begin function code.
222         For i In 1 to 2 Loop
223                 result(i) := A(i,i);
224         End Loop;
225         Return result;
226 End diag_2x2;
227
228 --***********************
229 --* Begin main process. *
230 --***********************
231 Begin
232         Process (z_position, reset) Is
233
234                 Constant dt : Signed(31  Downto 0) :=
235                 "00000000000000000001100110011001";
236
237                 --R is the measurement noise covariance.
```

```
238                 Constant R : Signed(31  Downto 0) :=
239                 "00000000000010100000000000000000";
240
241                 --Q is "dynamic noise strength" (process noise ...
                        covariance).
242                 Constant Q : Signed(31  Downto 0) :=
243                 "00000000110010000000000000000000";
244
245                 --G is the noise injection model.
246                 --This was intended to be 2 rows, 1 column but is ...
                        represented as
247                 --1 row, 2 columns.
248                 Constant G : matrix_2x1 :=
249                 ("00000000000000000000000000000000",
250                  "00000000000000010000000000000000");
251
252                 --This was intended to be 2 rows, 1 column but is ...
                        represented as
253                 --1 row, 2 columns.
254                 Constant B : matrix_2x1 :=
255                 ("00000000000000000000000000000000",
256                  "00000000000000010000000000000000");
257
258                 --This was intended to be 2 rows, 1 column but is ...
                        represented as
259                 --1 row, 2 columns.
260                 Constant Bd : matrix_2x1 :=
261                 ("00000000000000000000000101000111",
262                  "00000000000000000011001100110001");
263
264                 Constant H : matrix_1x2 :=
265                 ("00000000000000010000000000000000",
266                  "00000000000000000000000000000000");
267
268                 Constant H_prime : matrix_2x1 :=
```

```
269                 ("00000000000000010000000000000000",
270                  "00000000000000000000000000000000");
271
272                 Constant F : matrix_2x2 :=
273                 (("00000000000000000000000000000000",
274                  "00000000000000010000000000000000"),
275                 ("00000000000000000000000000000000",
276                  "00000000000000000000000000000000"));
277
278                 Constant phi : matrix_2x2 :=
279                 (("00000000000000010000000000000000",
280                  "00000000000000000001100110011001"),
281                 ("00000000000000000000000000000000",
282                  "00000000000000010000000000000000"));
283
284                 Constant phi_prime : matrix_2x2 :=
285                 (("00000000000000010000000000000000",
286                  "00000000000000000000000000000000"),
287                 ("00000000000000000001100110011001",
288                  "00000000000000010000000000000000"));
289
290                 Constant Qd : matrix_2x2 :=
291                 (("00000000000000000000100010000110",
292                  "00000000000000010000000000000000"),
293                 ("00000000000000010000000000000000",
294                  "00000000000010100000000000000000"));
295
296                 Constant Gd : matrix_2x2 :=
297                 (("00000000000000010000000000000000",
298                  "00000000000000000000000000000000"),
299                 ("00000000000000000000000000000000",
300                  "00000000000000010000000000000000"));
301
302                 --**************************
303                 --* Definition of Variables *
```

```
304                 --**************************
305
306                 Variable x : matrix_2x1 :=
307                 ("00000000000000010000000000000000",
308                  "00000000000000010000000000000000");
309
310                 Variable P : matrix_2x2 :=
311                 (("00000000000000000100000000000000",
312                  "00000000000000000000000000000000"),
313                 ("00000000000000000000000000000000",
314                  "00000000000000000100000000000000"));
315
316                 Variable A : Signed(31 Downto 0);
317                 Variable residual : Signed(31 Downto 0);
318                 Variable K : matrix_2x1;
319                 Variable K_temp : Signed(33 Downto 0);
320
321                 Begin
322
323                 If reset = '1' Then
324                         x :=
325                         ("00000000000000010000000000000000",
326                         "00000000000000010000000000000000");
327
328                         P :=
329                         (("00000000000000000100000000000000",
330                         "00000000000000000000000000000000"),
331                         ("00000000000000000000000000000000",
332                         "00000000000000000100000000000000"));
333                 End If;
334
335                 x := matrix_mult_2x2_2x1(phi,x);
336                 P := matrix_add_2x2((matrix_mult_2x2(...
                        matrix_mult_2x2(phi,P),phi_prime)),Qd);
```

```
337                 A := matrix_mult_1x2_2x1(matrix_mult_1x2_2x2(H,P),...
                        H_prime)+R;
338                 K_temp := "01000000000000000000000000000000"/A;
339                 K := matrix_mult_2x1_int(matrix_mult_2x2_2x1(P,...
                        H_prime),
340                         K_temp(31 Downto 0));
341                 residual := to_01(z_position) - ...
                        matrix_mult_1x2_2x1(H,x);
342                 x := matrix_add_2x1_2x1(x,matrix_mult_2x1_int(K,...
                        residual));
343                 P := matrix_subtract_2x2(P,matrix_mult_2x2(...
                        matrix_mult_2x1_1x2(K,H),P));
344
345                 position <= x(1);
346                 velocity <= x(2);
347
348         End Process;
349 End kf_behav;
```

This appendix contains the VHDL, RTL Kalman filter implementation entities.

Listing C.1:

```
Entity add_behavioral Is
     generic (high_bit : natural := 31); --This is the highest
                                         --order bit
     Port (reset        : In std_logic;
           in_a         : In signed(high_bit Downto 0);
           in_b         : In signed(high_bit Downto 0);
           output       : Out signed(high_bit Downto 0);
           c_out        : Out std_logic);
       End add_behavioral;
```

Listing C.2:

```
Entity add_sub_behavioral Is
     generic (high_bit : natural := 31); --This is the highest
                                         --order bit
     Port (add_sub     : In std_logic;
           --0 add in_A to in_B
           --1 subtract in_B from in_A
          reset        : In std_logic;
          in_a         : In signed(high_bit Downto 0);
          in_b         : In signed(high_bit Downto 0);
          output       : Out signed(high_bit Downto 0);
          c_out        : Out std_logic);
End add_sub_behavioral;
```

Listing C.3:

```
Entity ALU_2x2 Is
     Generic(high_bit : natural := 31; fraction_size : natural
          := 16);
     Port(A_00, A_01, A_10, A_11 : In signed(high_bit Downto 0);
          B_00, B_01, B_10, B_11 : In signed(high_bit Downto 0);
          reset : In std_logic;
```

```
7           add_sub : In std_logic;
8           reg_load : In std_logic;
9           mux_control : In std_logic;
10          clk : In std_logic;
11          C_00, C_01, C_10, C_11 : Out signed(high_bit Downto 0);
12          overflow : Out std_logic);
13 End ALU_2x2;
```

Listing C.4:

```
1 Entity controller Is
2     Port(clk, reset : In std_logic;
3           mem_control, ALU_1, ALU_2 : out signed(3 Downto 0);
4           mux_1, mux_2, output_reg_load, reciprocal_reset,
5           reciprocal_load, reciprocal_mux_control : out std_logic)...
               ;
6 End controller
```

Listing C.5:

```
1 Entity controller_2D Is
2     Port(clk, reset_external : In std_logic;
3           reset, load_main, load_sp_normhold, mux_control,
4           load_sp_1, load_sp_reg5 : out std_logic);
5 End controller_2D;
```

Listing C.6:

```
1 Entity denormalization Is
2      Generic(high_bit : natural := 31; fraction_size : natural
3           := 16);
4      Port (data_in : In signed(high_bit Downto 0);
5           sign_flag : In std_logic;
6           shift_value : In integer;
7           shift_direction : In std_logic;--'0' for right and
8                                          --'1' for left
9           clk : In std_logic;
```

```
10          data_out : Out signed(high_bit Downto 0));
11 End denormalization;
```

Listing C.7:

```
1  Entity KF_RTL_top Is
2      Generic(high_bit : natural := 31; fraction_size : natural := ...
           16);
3      Port(clk, reset : In std_logic;
4          input : In signed(high_bit Downto 0);--Otherwise known as z
5          x_out1, x_out2 : Out signed(high_bit Downto 0);
6          overflow1 : Out std_logic;
7          overflow2 : Out std_logic;
8          overflow_reciprocal : Out std_logic);
9  End KF_RTL_top;
```

Listing C.8:

```
1  Entity mem Is
2      Generic(high_bit : natural := 31);
3      Port(clk, reset   : In std_logic;
4           control : In signed(3 Downto 0);
5           in1_00, in1_01, in1_10, in1_11 : In signed (high_bit ...
               Downto 0);
6           in2_00, in2_01, in2_10, in2_11 : In signed (high_bit ...
               Downto 0);
7           A_00, A_01, A_10, A_11 : Out signed (high_bit Downto 0);
8           B_00, B_01, B_10, B_11 : Out signed (high_bit Downto 0);
9           C_00, C_01, C_10, C_11 : Out signed (high_bit Downto 0);
10          D_00, D_01, D_10, D_11 : Out signed (high_bit Downto 0));
11 End entity mem;
```

Listing C.9:

```
1  Entity mult_behavioral Is
2       Generic (high_bit : natural := 31;
3                 fraction_size : natural := 16); --This is the
```

```
4                                                    --highest order bit
5       Port (in_a        : In signed(high_bit Downto 0);
6             in_b        : In signed(high_bit Downto 0);
7             clear_async : In std_logic;
8             product     : Out signed(high_bit Downto 0);
9             c_out       : Out std_logic);
10 End mult_behavioral;
```

Listing C.10:

```
1 Entity mux Is
2     Generic(high_bit : natural := 31);
3     Port(control : In std_logic;
4          A : In signed(high_bit Downto 0);
5          B : In signed(high_bit Downto 0);
6          C : Out signed(high_bit Downto 0));
7 End entity mux;
```

Listing C.11:

```
1 Entity mux_2_to_1 Is
2     Generic(high_bit : natural := 31);
3     Port(control : In std_logic;
4          A_00, A_01, A_10, A_11 : In signed(high_bit Downto 0);
5          B_00, B_01, B_10, B_11 : In signed(high_bit Downto 0);
6          C_00, C_01, C_10, C_11 : Out signed(high_bit Downto 0));
7 End entity mux_2_to_1;
```

Listing C.12:

```
1 Entity mux_4_to_2 Is
2     Generic(high_bit : natural := 31);
3     Port(control : In std_logic;
4          A0_00, A0_01, A0_10, A0_11 : In signed(high_bit Downto 0)...
               ;
5          A1_00, A1_01, A1_10, A1_11 : In signed(high_bit Downto 0)...
               ;
```

```
6           B0_00, B0_01, B0_10, B0_11 : In signed(high_bit Downto 0)...
               ;
7           B1_00, B1_01, B1_10, B1_11 : In signed(high_bit Downto 0)...
               ;
8           C0_00, C0_01, C0_10, C0_11 : Out signed(high_bit Downto ...
               0);
9           C1_00, C1_01, C1_10, C1_11 : Out signed(high_bit Downto ...
               0));
10 End entity mux_4_to_2;
```

Listing C.13:

```
1  Entity normalization_sqrt Is
2       Generic(high_bit : natural := 31; fraction_size : natural
3            := 16);
4       Port (data_in : In signed(high_bit Downto 0);
5             data_out : Out signed(high_bit Downto 0);
6             sign_flag : Out std_logic;
7             shift_value : Out integer;
8             shift_direction : Out std_logic;--'0' for right and
9                                             --'1' for left
10            even0_odd1 : Out std_logic);
11 End normalization_sqrt;
```

Listing C.14:

```
1  Entity NR_LT_ROM Is
2       Generic (high_bit : natural := 31;
3                 fraction_size : natural := 16);
4       Port(address : In Signed(7 Downto 0);
5            estimate : Out Signed(high_bit Downto 0));
6  End NR_LT_ROM;
```

Listing C.15:

```
1  Entity reciprocal_stage1 Is
2       Generic(high_bit : natural := 31; fraction_size : natural
```

```
3           := 16);
4      Port (data_in : In signed(high_bit Downto 0);
5           data_out : Out signed(high_bit Downto 0);
6           sign_flag : Out std_logic;
7           shift_value : Out natural;
8           shift_direction : Out std_logic);--'0' for right and
9                                            --'1' for left
10 End reciprocal_stage1;
```

Listing C.16:

```
1 Entity reciprocal_stage3 Is
2      Generic(high_bit : natural := 31; fraction_size : natural
3           := 16);
4      Port (data_in : In signed(high_bit Downto 0);
5            mult2_out : Out signed(high_bit Downto 0);
6            square_out : Out signed(high_bit Downto 0));
7 End reciprocal_stage3;
```

Listing C.17:

```
1 Entity reciprocal_stage6 Is
2      Generic(high_bit : natural := 31; fraction_size : natural
3           := 16);
4      Port (data_in : In signed(high_bit Downto 0);
5           sign_flag : In std_logic;
6           shift_value : In natural;
7           shift_direction : In std_logic;--'0' for right and
8                                          --'1' for left
9           clk : In std_logic;
10          data_out : Out signed(high_bit Downto 0));
11 End reciprocal_stage6;
```

Listing C.18:

```
1 Entity reciprocal_top Is
2     Generic(high_bit : natural := 31; fraction_size : natural := ...
        16);
```

```
3     Port(clk, reset, load, mux_control : In std_logic;
4        data_in : In signed(high_bit Downto 0);
5        data_out : Out signed(high_bit Downto 0);
6        overflow : Out std_logic);
7 End reciprocal_top;
```

Listing C.19:

```
1 Entity reg_alu Is
2     Generic(high_bit : natural := 31);
3     Port(clk    : In std_logic;
4          load   : In std_logic;
5          d      : In signed(high_bit Downto 0);
6          q      : Out signed(high_bit Downto 0));
7 End entity reg_alu;
```

Listing C.20:

```
1 Entity RS_by_one Is
2      Generic(high_bit : natural := 31; fraction_size : natural
3           := 16);
4      Port (data_in : In signed(high_bit Downto 0);
5           data_out : Out signed(high_bit Downto 0));
6 End RS_by_one;
```

Listing C.21:

```
1 Entity sqrt_ROM Is
2      Generic (high_bit : natural := 31;
3                fraction_size : natural := 16);
4      Port(address : In Signed(7 Downto 0);
5           estimate : Out Signed(high_bit Downto 0));
6 End sqrt_ROM;
```

Listing C.22:

```
1 Entity sqrt_ROM_even Is
2      Generic (high_bit : natural := 31;
```

```
3               fraction_size : natural := 16);
4      Port(address : In Signed(7 Downto 0);
5           estimate : Out Signed(high_bit Downto 0));
6 End sqrt_ROM_even;
```

Listing C.23:

```
1 Entity sqrt_ROM_odd Is
2      Generic (high_bit : natural := 31;
3               fraction_size : natural := 16);
4      Port(address : In Signed(7 Downto 0);
5           estimate : Out Signed(high_bit Downto 0));
6 End sqrt_ROM_odd;
```

Listing C.24:

```
1 Entity squared_behavioral Is
2      Generic (high_bit : natural := 31;
3               fraction_size : natural := 16); --This is the
4                                               --highest order bit
5      Port (in_a       : In signed(high_bit Downto 0);
6            clear_async : In std_logic;
7            product     : Out signed(high_bit Downto 0);
8            c_out       : Out std_logic);
9 End squared_behavioral;
```

Listing C.25:

```
1 Entity sub_behavioral Is
2      generic (high_bit : natural := 31); --This is the highest
3                                          --order bit
4      Port (reset        : In std_logic;
5            in_a         : In signed(high_bit Downto 0);
6            in_b         : In signed(high_bit Downto 0);
7            output       : Out signed(high_bit Downto 0);
8            c_out        : Out std_logic);
9 End sub_behavioral;
```

Listing C.26:

```
Entity sub_const_behavioral Is
     generic(high_bit : natural := 31;
          fraction_size : natural := 16);
     Port(reset         : In std_logic;
          in_a          : In signed(high_bit Downto 0);
          output        : Out signed(high_bit Downto 0);
          c_out         : Out std_logic);
End sub_const_behavioral;
```

Listing C.27:

```
Entity TwoD_connect_top Is
    Generic(high_bit : natural := 31; fraction_size : natural := ...
       16);
    Port(clk, reset, reset_TwoD_top_controller : In std_logic;
       input_z1 : In signed(high_bit Downto 0);--Input from the x-...
          coordinate
                                                --Kalman filter.
       input_z2 : In signed(high_bit Downto 0);--Input from the y-...
          coordinate
                                                --Kalman filter.
       output : Out signed(high_bit Downto 0);
       overflow : Out std_logic;
       output_position1 : Out signed(high_bit Downto 0);
       output_position2 : Out signed(high_bit Downto 0));
End TwoD_connect_top;
```

Listing C.28:

```
Entity TwoD_top Is
    Generic(high_bit : natural := 31; fraction_size : natural := ...
       16);
    Port(clk, reset_external : In std_logic;
       input_x : In signed(high_bit Downto 0);--Input from the x-...
          coordinate
                                              --Kalman filter.
```

```
6         input_y : In signed(high_bit Downto 0);--Input from the y-...
              coordinate
7                                                  --Kalman filter.
8         output : Out signed(high_bit Downto 0);
9         overflow_2D : Out std_logic);
10 End TwoD_top;
```

Listing C.29:

```
1 Entity TwoD_top_controller Is
2     Port(clk, reset : In std_logic;
3           reset_external : out std_logic);
4 End TwoD_top_controller;
```

Appendix D. Design Schematics

This appendix contains schematics for some of the VHDL Kalman filter modules.

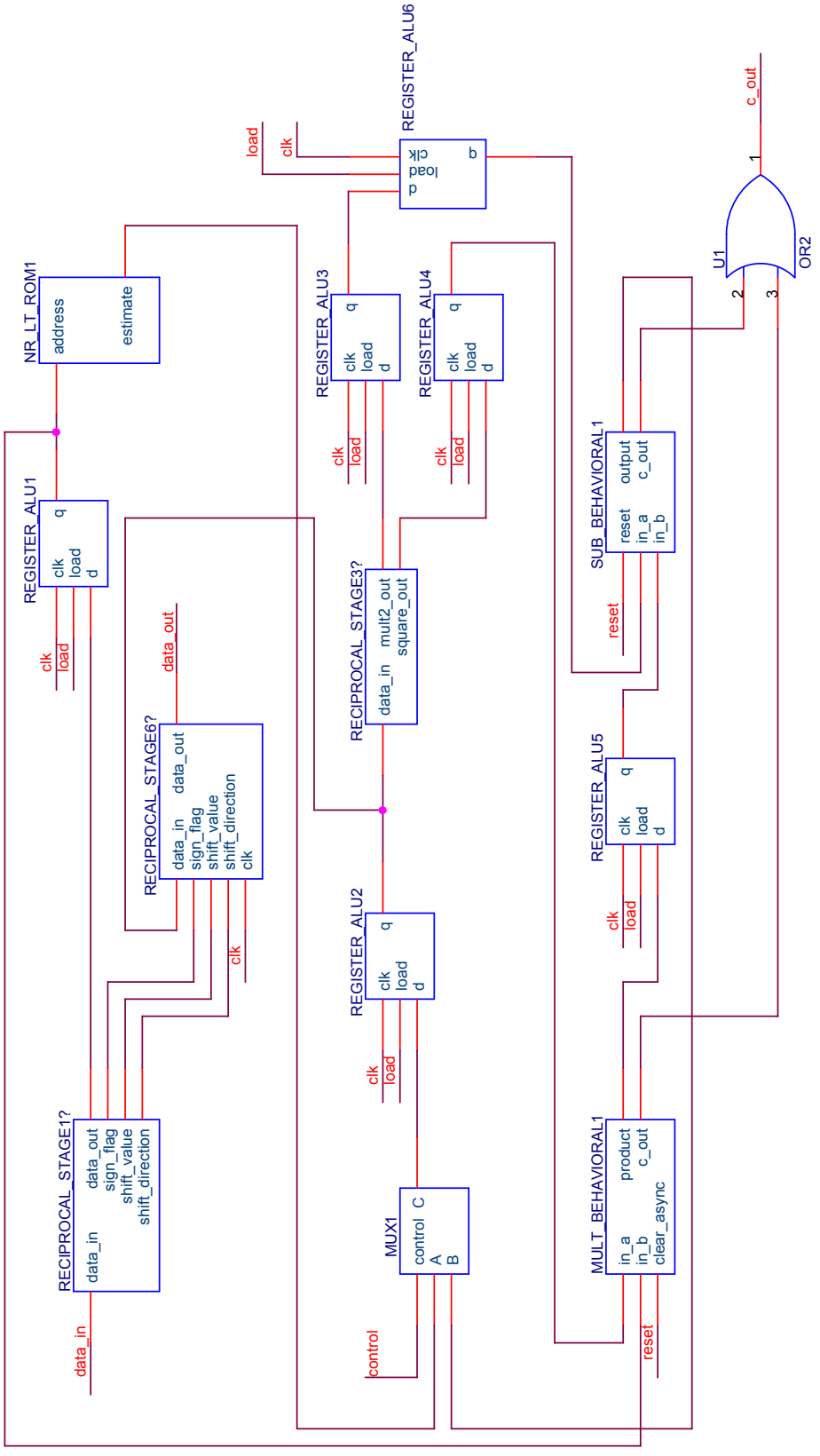


Figure D.1: Schematic for the Newton Raphson reciprocal function.

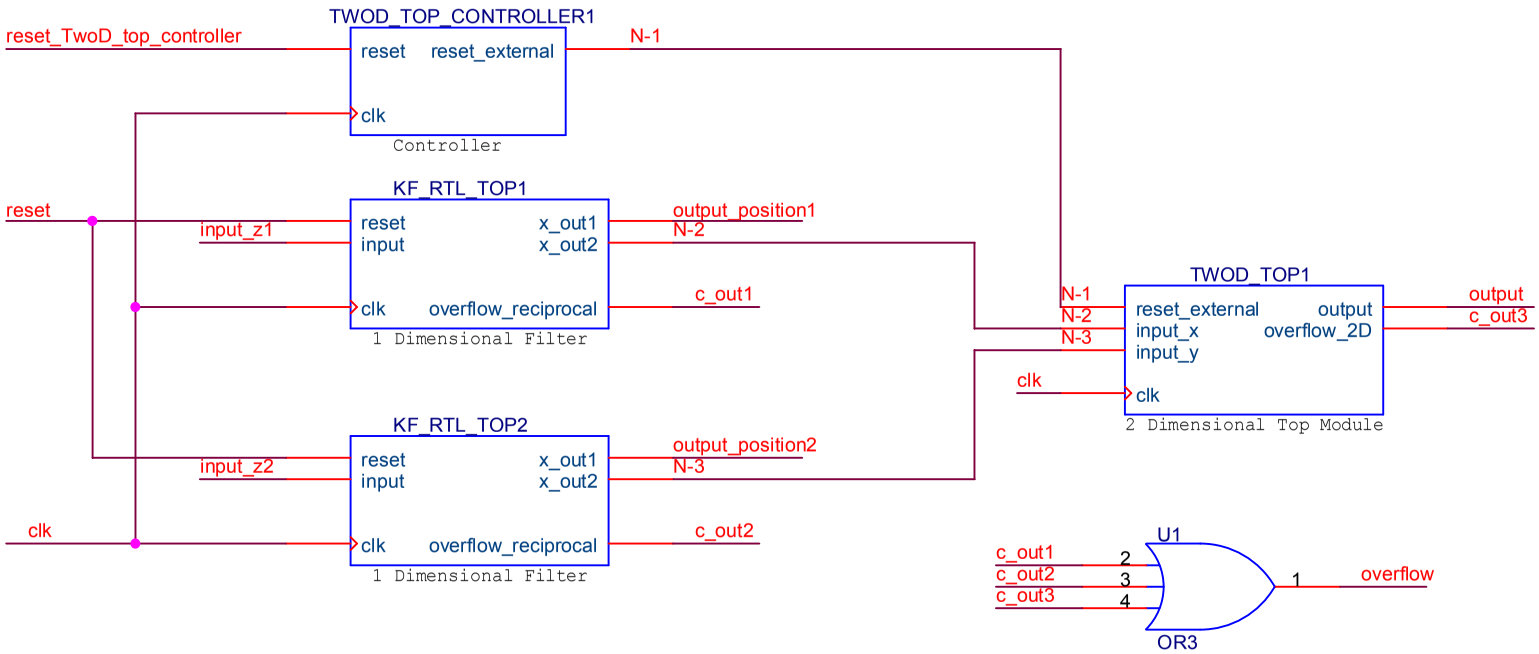


Figure D.2: Top level schematic for the two-dimensional Kalman filter implementation.

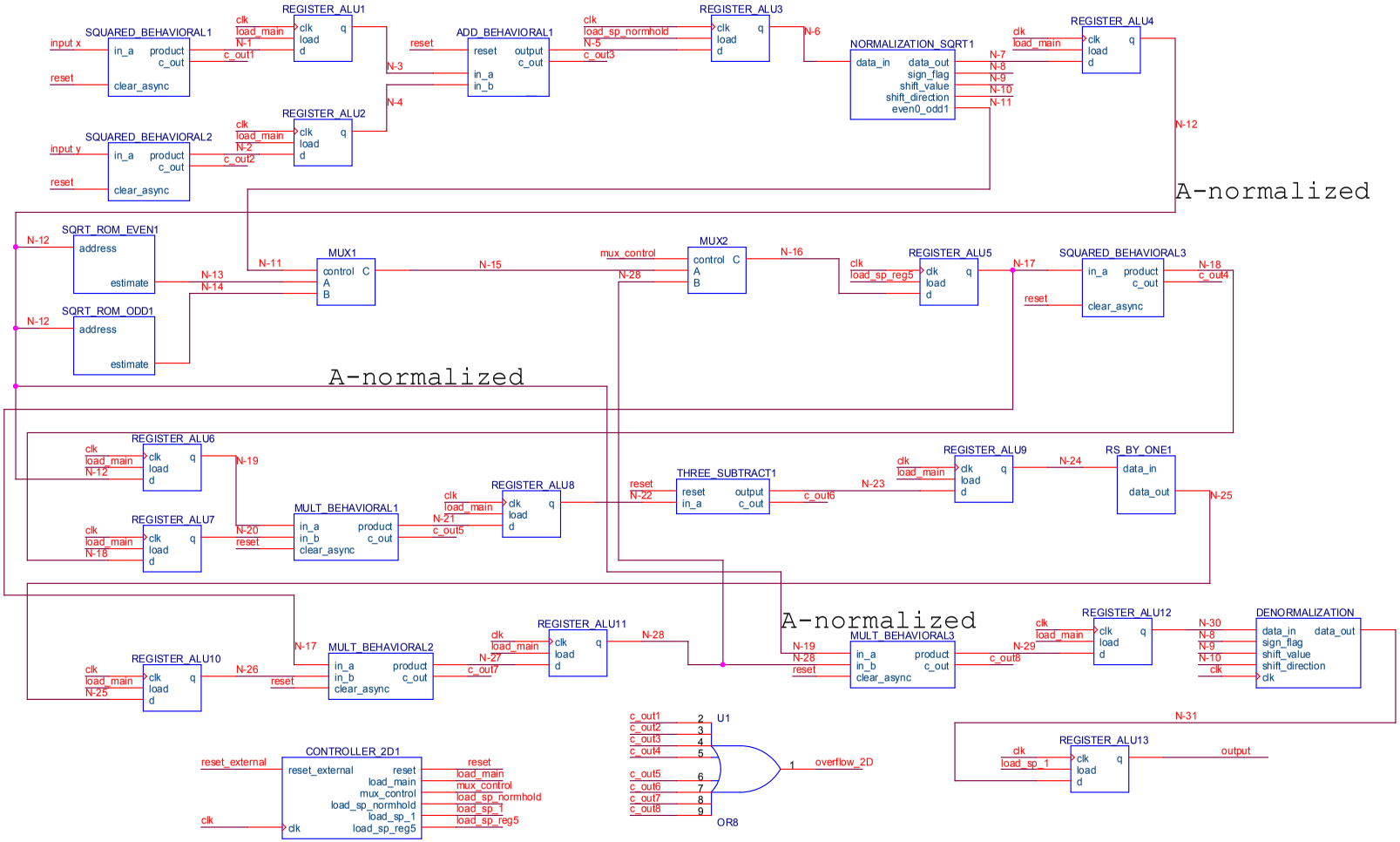


Figure D.3: Bottom level schematic for the two-dimensional Kalman filter implementation.

Bibliography

1. DasSarma, Debjit and David Matula. "Measuring the Accuracy of ROM Reciprocal Tables". *IEEE Transactions on Computers*, 43:932–940, August 1994.

2. Forbes, Eric G. "Gauss and the Discovery of Ceres". *Journal for the History of Astronomy*, 2:195–199, 1971.

3. Fowler, D. L. and J. E. Smith. "An Accurate, High Speed Implementation of Division by Reciprocal Approximation". *Proceedings of the 9th Symposium on Computer Arithmetic*, 60–67, 1989.

4. G. Even, P.M. Seidel and W.E. Ferguson. "A Parametric Error Analysis of Goldschmidt's Division Algorithm". *Journal of Computer and System Sciences*, 70(1):118–139, February 2005. ISSN:0022-0000.

5. Gaurav Agrawal, Ankit Khandelwal and Jr. Earl E. Swartzlander. "An Improved Reciprocal Approximation Algorithm for a Newton Raphson Divider". *Proceeding of SPIE, Advanced Signal Processing Algorithms, Architectures, and Implementations XVII*, 6697, September 2007. The International Society for Optical Engineering.

6. Grewal, Mohinder S. and Angus P. Andrews. *Kalman Filtering: Theory and Practice Using MATLAB*. John Wiley and Sons, Inc., second edition, 2001. ISBN 0-471-39254-5.

7. H. Kabuo, A. Miyoshi H. Yamashita M. Urano H. Edamatsu S. Kuninobu, T. Taniguchi. "Accurate Rounding Scheme for the Newton-Raphson Method Using Redundant Binary Representation". *IEEE Transaction on Computers*, 43(1):43–51, 1994.

8. Hennessy, J. and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, second edition, 1996. Appendix A: Computer Arithmetic by D. Goldberg.

9. Hennessy, John L. and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, third edition, 2005. ISBN 1-55860-604-1.

10. Hoskin, Michael. "Bodes' Law and the Discovery of Ceres", June 1992. Www.astropa.unipa.it/HISTORY/hoskin.html.

11. Hwang, Kai. *Computer Arithmetic: Principles, Architecture and Design*. John Wiley & Sons, Inc., New York, NY, USA, 1979. ISBN 0471052000.

12. Kilts, Steve. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-Interscience, first edition, 2007. ISBN 978-0-470-05437-6.

13. Maybeck, Peter S. *Stochastic models, estimation, and control*, volume 141 of *Mathematics in Science and Engineering*. 1979.

14. Morgan, Don. *Numerical Methods for DSP Systems in C*. Wiley Computer Publishing, 1997. ISBN 0-471-13232-2.

15. Pop, P., P. Eles, and Z. Peng. *Analysis and Synthesis of Distributed Real-Time Embedded Systems*. Kluwer Academic Publishers, 2004. ISBN 1-4020-2872-5.

16. Schwarz, E. M. and M. J. Flynn. "Hardware Starting Approximation For The Square Root Operation". *Proceedings from the 11th Symposium on Computer Arithmetic*, 103–111, 29 Jun-2 Jul 1993.

17. Sorenson, H. W. "Least-squares estimation: from Gauss to Kalman". *IEEE Spectrum*, 7:63–68, July 1970.

18. unknown. "Merriam-Webster OnLine Dictionary". World Wide Web, 2008. Http://www.merriam-webster.com/dictionary/system.

19. Wang, Liang-Kai and Michael J. Schulte. "Decimal Floating-Point Square Root Using Newton-Raphson Iteration". *Proceeding of the 16th International Conference on Application-Specific Systems, Architectures and Processors*, 309–315, July 2005. INSPEC Accession Number: 8745893.

20. Ward, Paul T. and Stephen J. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0917072510.

21. Welch, Greg and Gary Bishop. "An Introduction to the Kalman Filter", July 2006. University of North Carolina at Chapel Hill.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
27-03-2008	**Master's Thesis**	May 2006-March 2008

4. TITLE AND SUBTITLE

CHARACTERIZATION AND IMPLEMENTATION OF A REAL-WORLD TARGET TRACKING ALGORITHM ON FIELD PROGRAMMABLE GATE ARRAYS WITH KALMAN FILTER TEST CASE

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hancey, Benjamin D., Captain, USAF

5d. PROJECT NUMBER
ENG 08-262

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765 DSN: 785-3636

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/08-10

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Devert Wicker, devert.wicker@wpafb.af.mil , Comm.:(937) 674-9871
AFRL/RYAT (AFMC)
2241 Avionics Circle
WPAFB, OH 45433-7765

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A one dimensional Kalman Filter algorithm provided in Matlab is used as the basis for a Very High Speed Integrated Circuit Hardware Description Language (VHDL) model. The JAVA programming language is used to create the VHDL code that describes the Kalman filter in hardware which allows for maximum flexibility. A one-dimensional behavioral model of the Kalman Filter is described, as well as a one-dimensional and synthesizable register transfer level (RTL) model with optimizations for speed, area, and power. These optimizations are achieved by a focus on parallelization as well as careful Kalman filter sub-module algorithm selection. Newton-Raphson reciprocal is the chosen algorithm for a fundamental aspect of the Kalman filter, which allows efficient high-speed computation of reciprocals within the overall system. The Newton-Raphson method is also expanded for use in calculating square-roots in an optimized and synthesizable two-dimensional VHDL implementation of the Kalman filter. The two-dimensional Kalman filter expands on the one-dimensional implementation allowing for the tracking of targets on a real-world Cartesian coordinate system.

15. SUBJECT TERMS
Kalman filter, Newton-Raphson, VHDL, RTL, target tracking

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
REPORT	**ABSTRACT**	**c. THIS PAGE**			**Dr. Yong Kim, PhD (ENG)**
U	**U**	**U**	**UU**	112	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565; email: benjamin.hancey@afit.edu