Hardware Implementation of Extended Kalman Filter

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronic Systems

by

N.Soumya

Roll No: 123076008

under the guidance of

Prof. V Rajbabu

Department of Electrical Engineering

Indian Institute of Technology, Bombay

June, 2015

Dissertation approval

This dissertation entitled *Hardware Implementation of Extended Kalman Filter* by N.Soumya is approved for the degree of Master of Technology.

Examiners

_____

_____

Supervisor

_____

Chairman

_____

Date: _____

Place: _____

Declaration

I declare that this written submission represents my ideas in my own words and where others ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Roll No: 123076008)

N.Soumya

Contents

List of Figures

List of Tables

Abstract

Extended Kalman filter is one of the standard non-linear estimation algorithm used in realtime target tracking. If it is to be used in realtime applications, the estimation time has to be very less (of the order of milliseconds). The objective of the work presented in this report is to design and realize a scalable extended Kalman filter (EKF) for non-linear tracking applications. A scalable design facilitates reusability of the design for varying inputs and models. To achieve this, a parameterizable architecture was used for the implementation of individual modules of the EKF building blocks. These modules mainly comprise of matrix operations such as matrix addition, multiplication and inversion. However, certain sections of the EKF algorithm cannot be modularized as they are specific to the inputs and models used. Here, Jacobian calculation and prediction of states in terms of measurement parameters, are the two main non-modularizable sections.

A performance comparison of the VHDL implementation with that of Matlab implementation is done in terms of estimated state values. The observations indicate similar outputs for FPGA and Matlab implementations. Further, timing comparison show comparable performance between FPGA and Matlab implementations. The FPGA implementation at $34.417\mu s$ is slightly faster than the Matlab implementation which takes $\approx 38.09\mu s$ to complete one iteration. Resource utilization and approximate execution times are also summarized. The maximum frequency of operations obtained was $330.834MHz$. The time taken for a single iteration was obtained as $34.417\mu s$.

Chapter 1

Introduction

State estimation is a key aspect of any tracking application. States of a system are parameters that completely describe the behaviour of a system at any given time. In the context of tracking, these parameters are mainly position, velocity, bearing, and perhaps acceleration. Often, models cannot completely capture the dynamics of a system and hence do not account for deviations in the state values. Measurements are most often noisy and do not provide accurate state values either. In such situations, state estimation algorithms such as Kalman filters, extended Kalman filters, unscented Kalman filters provide a means of using the measurement and state update models to estimate state assuming Gaussian noise.

Most tracking systems find compact portable units more desirable as they promote agility and freedom of movement of a system. This necessitates tracking algorithms to be on an independent hardware unit. Porting estimation algorithms to FPGA's, that provide processing power along with portability, is a viable option for making a tracking system autonomous, and is the subject of interest in this report. We specifically focus on extended Kalman filter.

1.1 Literature review

Hardware implementation of extended Kalman filters (EKF) has been an active research topic for the past decade [1], [2], [3], [4]. One of the papers discusses the computational

complexity incurred in implementation of EKF and possible solutions such as hardware (FPGA) - software (DSP) effort distribution, use of embedded software processors and use of C-to-Hardware for ease of design [1].

Simultaneous localization and tracking (SLAM) is another active area of research on mobile robotics that extensively use hardware implementation of EKF estimation algorithms for tracking. Other works in [2], [3] and [4] describe use of EKF for tracking in systems that use SLAM. In [2] the authors describe an FPGA based EKF algorithm implementation for SLAM problem. Odometric sensor and exteroceptive sensor outputs are used as the measurements obtained for the estimation of states in a SLAM problem for a mobile robotic system. They compared EKF performance on FPGA with that on Pentium M 1.6GHz processor and ARM920T 200MHz, in terms of power consumption for each feature ($0.7mW$ in FPGA as against $54mW$ for Pentium M processor and $5.7mW$ for ARM920T). The authors focused on reusing on-chip data to bring down the computation complexity associated with the covariance update stage, which is the most computationally expensive stage for SLAM application. The hardware architecture uses a light embedded processor to compute the prediction and measurement models with the associated Jacobians. The number of features computed is 1800. The authors have used Handel-C language to describe the hardware architecture. The target device used is EP2S90F1020C4.

The authors of paper [3] describe use of EKF in SLAM for a mobile robotic device. The implementation is based on NIOS II softcore ported on Altera FPGA. EKF algorithm has been coded in C. Additional hardware accelerator blocks have been implemented for unspecified blocks of EKF algorithm. The authors have experimented with C-to-Hardware and custom instructions to generate the hardware accelerator blocks. The inputs for the robot is from odometer to get position information and image sensors for fixing the landmarks. A performance comparison (in terms of speed) of this FPGA implementation with a purely software approach involving Pentium IV 3GHz system was done by the authors. The purely software approach on a PC was found faster at $0.63GHz$, but the FPGA based system operating at $50MHz$ was found acceptable for indoor robotic applications.

In another paper [4], the authors have described an FPGA based implementation of Sequential EKF update algorithm for self localization task for small robots. The

system performs fusion of data taking inputs from ultrasonic (sonar) and LASER range finder. The authors have dealt only with the localization problem and only the gain and update blocks of EKF. The computation of Jacobian matrix is done offline on a Nios II software and the results are used in the FPGA implementation. Gain calculation in the EKF algorithm is done using FSM and the matrix inversion is obtained by calculating the adjoint and determinant of the matrix. The design was realized on a Cyclone IV EP4CE22F17C6N FPGA. Analysis of hardware resource used and power consumed was done by the authors. The authors compared individual modules that computes gain, states, and covariance with respect to the number of DSP blocks used, maximum clock frequency and power consumption in mW. They concluded that the gain module uses more DSP's (29%) as compared to states module (4%) and covariance module (12%). However they found the power consumption of gain module and covariance modules to be comparable at $170.87mW$ and $170.46mW$ respectively.

These three references explore hardware-software division of efforts and the effect on performance of the system. The most widely used and preferred estimation algorithm for non-linear systems, seems to be EKF. However, the popular approach seems to be the use of embedded processor core such as Nios II or use of C to Hardware or Matlab to hardware approach. Also the prediction part of the algorithm seems to be mostly performed externally and mainly the gain calculation and update stages are implemented on FPGA. This could be due to the much larger matrix dimensions owing to large state vectors.

This report however addresses the implementation of a scalable Extended Kalman filter with the aim of realizing a complete hardware solution to achieve a real-time system with fast response time.

1.2 Extended Kalman filter

Kalman filter, a linear state estimation algorithm, has been widely used to estimate a system's state by drawing upon in-exact measurements and an appropriate mathematical model. In Kalman filter, estimation process is the propagation of expectation or mean of the state and covariance of the state, in time, for linear systems. These parameters together describe the system behaviour. The expected value or the mean value denotes

the estimate of the state and the covariance denotes the degree of uncertanity in the estimated state. The equations involved in the Kalman filter deal with computing the *a posteriori* state estimate as a function of linear combination of *a priori* estimate and a weighted difference of the measurement at that time step and the prediction obtained from the *a priory* estimate [7]. The underlying assumptions are that the dynamic system is linear, the noises involved are zero mean, Gaussian, white and uncorrelated.

Kalman filter algorithm provides the optimal solution for estimation of a linear system. However, these equations cannot be directly extended to non-linear systems. To deal with non-linear systems, the approach most commonly used is to approximate all inherent non-linearities of the system with a linear model. Linearization of non-linearities is done using the Taylor series approximations.

Algorithm

Extended Kalman filter is an extension to the Kalman filter algorithm that incorporates linearization of the system about the nominal state trajectory, which is the Kalman filter estimate at every time step. The principle behind EKF is that the mean and covariance of the linearized system is approximately equal to the non linear system's true mean and covariance.

Suppose the system model is of the form

$$X_k = f(X_{k-1}, v_{k-1}) \tag{1.1}$$

where,

f is state transition model or process model,

X_{k-1} is the estimate of the state vector during the previous time step ie $k-1$

X_k is the current state estimate i.e. at time step k

v_{k-1} represented as $v_{k-1} \sim \mathcal{N}(0, Q_k)$, is zero mean, Gaussian, white, uncorrelated process noise.

$$Z_k = h(X_k, w_k) \tag{1.2}$$

where,

h is the observation or measurement model

Z_k is the estimated measurement vector at time step k

w_k is the zero mean, Gaussian, white, uncorrelated measurement noise represented as $w_k \sim \mathcal{N}(0, R_k)$

On performing Taylor series expansion around the value $X = \hat{X}_{k-1}$ and considering $v_{k-1} = 0$, we get

$$X_k = f(X_{k-1}, 0) + \left.\frac{\partial f}{\partial X}\right|_{X=\hat{X}_{k-1}} (X_{k-1} - \hat{X}_{k-1}) + \left.\frac{\partial f}{\partial v}\right|_{\hat{X}_{k-1}} v_{k-1} \tag{1.3}$$

and the measurement model expanded around the new X_k represented as $\hat{X}_k$ and $w_k = 0$, we get

$$Z_k = h(\hat{X}_k, 0) + \left.\frac{\partial h}{\partial X}\right|_{\hat{X}_k} (X_k - \hat{X}_k) + \left.\frac{\partial h}{\partial w}\right|_{\hat{X}_k} w_k \tag{1.4}$$

It is seen from equations 1.3 and 1.4 that implementation of Extended Kalman filter involves the computation of derivative matrices or Jacobians.

The set of equations in the Extended Kalman filter implementation can therefore be summarized as

- Time update

$$\hat{\mathbf{x}}_k = \mathbf{F}\mathbf{x}_{k-1} \tag{1.5}$$

$$\hat{\mathbf{P}}_k = \nabla\mathbf{F}\mathbf{P}_{k-1}\nabla\mathbf{F}^T + \mathbf{Q} \tag{1.6}$$

  where,

  $\mathbf{F}$ is the state transition matrix,

  $\mathbf{x}_k$ is the state vector at instant k,

  $\hat{\mathbf{P}}_k$ is the covariance matrix at instant k,

  $\mathbf{Q}$ is the process noise matrix.

- Measurement update

  - Computing gain

$$\mathbf{G} = \hat{\mathbf{P}}_k\nabla\mathbf{H}^T[(\nabla\mathbf{H}\hat{\mathbf{P}}_k\nabla\mathbf{H}^T) + \mathbf{R}]^{-1} \tag{1.7}$$

    where,

    $\mathbf{H}$ is the measurement matrix,

    $\mathbf{R}$ is the measurement noise matrix,

    $\mathbf{G}$ is the gain,

    $\nabla\mathbf{H}$ is the Jacobian of measurement matrix.

– Update estimate with measurements

$$\mathbf{x}_k = \hat{\mathbf{x}}_k + [\mathbf{G}(Z_k - (\mathbf{H}\hat{\mathbf{x}}_k))] \tag{1.8}$$

– Compute error covariance for updated estimate

$$\mathbf{P}_k = \hat{\mathbf{P}}_k - (\mathbf{G}\nabla\mathbf{H}\hat{\mathbf{P}}_k) \tag{1.9}$$

where, $\nabla\mathbf{F}$ is the Jacobian of state transition matrix and $\nabla\mathbf{H}$ is the Jacobian of measurement matrix [7]. The above set of equations of EKF, give fairly good estimation for most non-linear systems and EKF is therefore widely used.

1.3 FPGA

Field programmable gate array (FPGA), is made up of a large number of configurable logic blocks along with a distributed interconnect structure. Configurable logic blocks (CLB) are the basic units of FPGA's and contain a cluster of slices or logic cells. Slices or logic cells contain a highly flexible switch matrix, flip-flops, multiplexers, and lookup tables (LUT's). The switch matrix can be configured to form combinatorial logic, shift registers or memory. The CLB's are connected to each other and external world by interconnects. The interface to the external world is via Input-Output Blocks (IOBs). Block RAM provide the on-chip memory on the FPGAs. Other resources available on FPGA includes multipliers, global clock buffers and boundary scan logic. The matrix of CLBs and the mesh of interconnects, lend to the programmability of FPGAs [8]. Their reconfigurability, reusability and a shorter development time as against ASIC's, make them an attractive choice for quick design realizations.

The objective of this project is to design efficient hardware for extended Kalman filter. The focus is on introducing maximum re-usability of the design by using a modular approach. The idea is to make individual blocks scalable so that they can be reused in the design as well as in a different system model. Possibility of introducing parallelism in discrete modules is also explored in this report.

Organization of report

Chapter 2 describes initially the high level view of the complete EKF algorithm in the form of modules. The various blocks in the modules and their implementation, are discussed. Implementation details such as data representation, target device resources and IP cores used are also discussed.

Simulation results, timing analysis and hardware utilization are discussed in chapter 3. Conclusions and future scope are mentioned in chapter 4.

Chapter 2

Implementation of EKF on FPGA

2.1 Extended Kalman filter implementation

The extended Kalman filter is most often used when the states of the dynamic system to be estimated, involves non-linearity in them. In this implementation therefore, a constant turn model was used as the state transition model. The observations thus obtained were in range r and bearing θ rather than the cartesian x and y position. The extended Kalman filter algorithm tackles the non-linearity of the system by using the derivative of the state transition and measurement model. This involves computation of Jacobian of state transition matrix ($\mathbf{F}$) and measurement matrix ($\mathbf{H}$). The state transition used was split into constant velocity and constant turn (manoeuvring) parts. However, for estimation, only the constant velocity model was used and hence necessitated the derivation of Jacobians for only the constant velocity state transition model.

The input state vector in the cartesian coordinate system is

$$\mathbf{x} = \begin{bmatrix} x & v_x & y & v_y \end{bmatrix}^T \tag{2.1}$$

The state transition model used i.e. the constant velocity model is described as

$$\mathbf{F}_1 = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

The Jacobian of state transition model with respect to each of the input states is the same as the state transition model. The constant turn state transition model [5] used to

generate the true states and measurements is

$$\mathbf{F}_2 = \begin{bmatrix} 1 & \frac{\sin(\Omega T)}{\Omega} & 0 & -\frac{1-\cos(\Omega T)}{\Omega} \\ 0 & \cos(\Omega T) & 0 & -\sin(\Omega T) \\ 0 & \frac{1-\cos(\Omega T)}{\Omega} & 1 & \frac{\sin(\Omega T)}{\Omega} \\ 0 & \sin(\Omega T) & 0 & \cos(\Omega T) \end{bmatrix}$$

where Ω is the constant turn rate. This model is not used in estimation. It is only used to generate ground truth in a Matlab program, which is finally used for performance comparison.

The measurement is in terms of range and bearing and is given by

$$\mathbf{z} = \begin{bmatrix} \sqrt{x^2+y^2} \\ \tan^{-1}(\frac{y}{x}) \end{bmatrix} \tag{2.3}$$

The Jacobian of the measurement model $\nabla\mathbf{H}$ is

$$\nabla\mathbf{H} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & 0 & \frac{y}{\sqrt{x^2+y^2}} & 0 \\ \frac{-y}{(x^2+y^2)} & 0 & \frac{x}{(x^2+y^2)} & 0 \end{bmatrix} \tag{2.4}$$

Trigonometric calculation on digital systems are done using CORDIC algorithms - an acronym for COordinate Rotation DIgital Computer. Xilinx logicores set has a Cordic core, that takes in values in fixed point format and computes the output - also in fixed point form. The arctan logicore requires the input to be between the range -1 and +1. Also, the input word is expected to have exactly 2 integer bits for any word length. The output is a fixed point value with 3 bit integer and $(wordlength - 3)$ fraction bits. As the format used in the implementation was 32 bit floating point, logicores to convert from floating point value to fixed point value and vice versa were used.

The Jacobian evaluation requires the computation of square root of the sum of squares of the x and y coordinates. The Xilinx floating point logicore includes square root computation functionality as well, which is used in the implementation. Jacobian evaluation is problem specific and is one section of the EKF algorithm that could not be modularized and generalized. Separate finite state machines were designed for obtaining the arctan of the ratio of x and y coordinates and for computing the Jacobian of measurement model. These blocks were then integrated in the main finite state machine. Apart from the Ja-

cobian computations, the other parts of extended Kalman filter implementation include prediction of new state and covariance, gain evaluation and state and covariance update.

The equations thus computed in the extended Kalman filter implementation are summarized in Table 2.1.

The control path finite state machine (FSM) for extended Kalman filter is as shown in Figure 2.1. The control path is built as a Moore machine i.e. the control path output depends only on the current state of the FSM. Each of the states in the main state machine, either enables a sub-state machine designed for matrix operations or enables registers to store intermediate values. State transition occurs either with clock transition or when there is an assert from the data path indicating completion of a matrix operation.

The data path of extended Kalman filter implementation is shown in Figure 2.2. The data path includes the modules to compute the state vector prediction, covariance matrix prediction, Jacobian calculation, prediction of state in terms of the measurement parameters of range and angle, matrix inversion by LU decomposition, gain module and finally the state and covariance updates. Barring the Jacobian and prediction of state in terms of the measurement parameters of range and angle, every other module is built in a generic manner using the parameter feature of VHDL. If the design needs to be modified to a different model, only some parts needs to be changed.

The hardware implementation of extended Kalman filter was realized with Virtex-6 FPGA (xc6vlx75t and xc6vl130t) as the target device. Virtex-6 FPGAs are some of the higher end FPGA's from the Xilinx family of FPGAs [8]. A single configurable logic block (CLB) in a Virtex-6 FPGA comprises of two slices, each containing four 6-input LUTs and eight flip-flops [13]. The logic resources available on xc6vlx75t include 11,640 slices, 74,496 logic cells, 93,120 CLB flip-flops and 288 DSP48E1 Slices. A larger variant of Virtex 6 FPGA, xc6vl130t, includes 20,000 slices, 128,000 Logic cells, 160,000 CLB flip-flops and 480 DSP48E1 Slices.

Table 2.1: Equations and Computational complexity in EKF for state dimension M

Equation	Computational Complexity	Dimensions
$\hat{\mathbf{x}}_k = \mathbf{F} * \mathbf{x}_{k-1}$	1 Matrix multiplication	$(M \times M)$ * $(M \times 1)$
∂H	2 Multiplications	Scalar
	1 Addition	Scalar
	1 Square root	Scalar
	4 Divisions	Scalar
$\hat{\mathbf{x}z}_k = \frac{\sqrt{(x)^2+(y)^2}}{arctan(\frac{y}{x})}$	CORDIC function	Scalar
$\hat{\mathbf{P}}_k = \nabla\mathbf{F} * \mathbf{P}_{k-1} * \nabla\mathbf{F}_T + \mathbf{Q}$	2 Matrix multiplications	$(M \times M)$ * $(M \times M)$
		$(M \times M)$ * $(M \times M)$
	1 Matrix Addition	$(M \times M)$
$\mathbf{S} = \nabla\mathbf{H} * \hat{\mathbf{P}}_k * \nabla\mathbf{H}_T + \mathbf{R}$	2 Matrix multiplications	$(N \times M)$ * $(M \times M)$
		$(N \times M)$ * $(M \times N)$
	1 Matrix Addition	$(N \times N)$
$inv(\mathbf{S}) = (\mathbf{S})^{-1}$	1 Matrix Inverse	$(N \times N)$
$\mathbf{G} = \hat{\mathbf{P}}_k * \nabla\mathbf{H}_T * inv(\mathbf{S})$	2 Matrix multiplications	$(M \times M)$ * $(M \times N)$
		$(M \times N)$ * $(N \times N)$
$\mathbf{x}_k = \hat{\mathbf{x}}_k + [\mathbf{G} * (\mathbf{z}_{measured} - \hat{\mathbf{x}z}_k)]$	1 Matrix multiplication	$(M \times N)$ * $(N \times 1)$
	1 Matrix Subtraction	$(N \times 1)$
	1 Matrix Addition	$(M \times 1)$
$\mathbf{P}_k = \hat{\mathbf{P}}_k - (\mathbf{G} * \nabla\mathbf{H} * \hat{\mathbf{P}}_k)$	2 Matrix multiplications	$(M \times N)$ * $(N \times M)$
		$(M \times M)$ * $(M \times M)$
	1 Matrix subtraction	$(M \times M)$

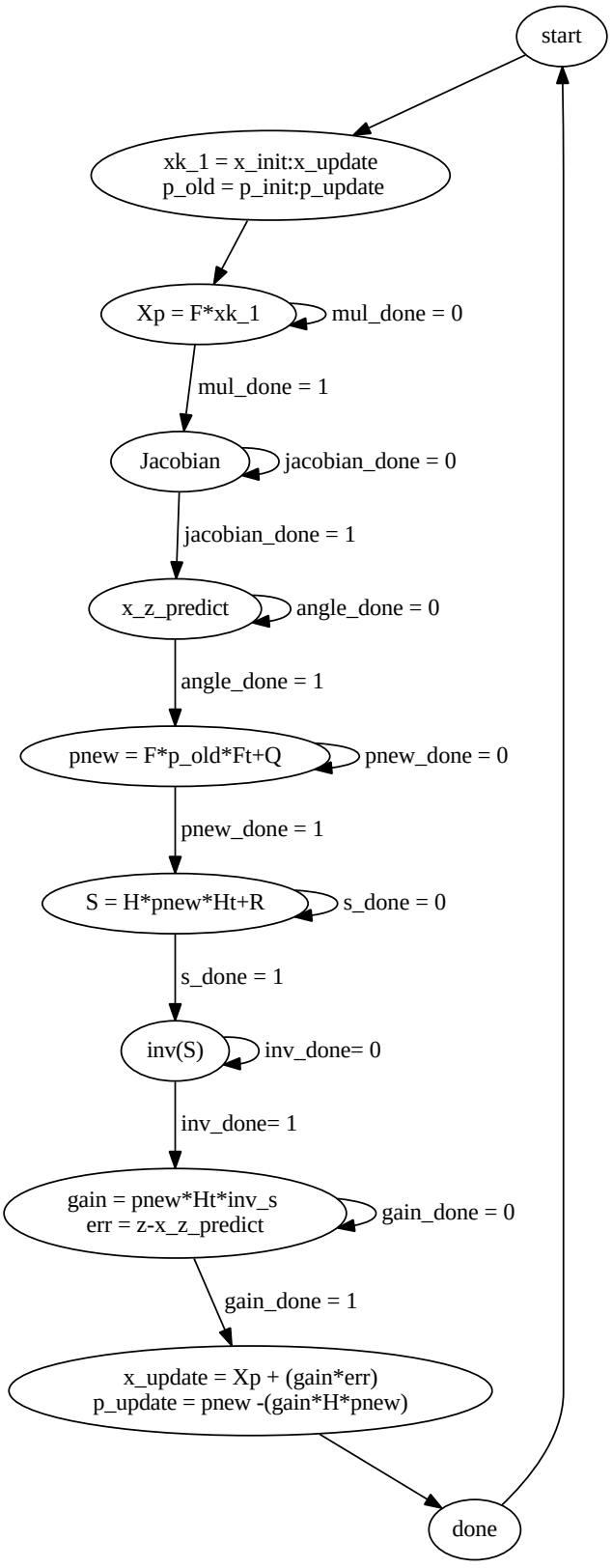

Figure 2.1: Finite state machine depicting control path of EKF implementation

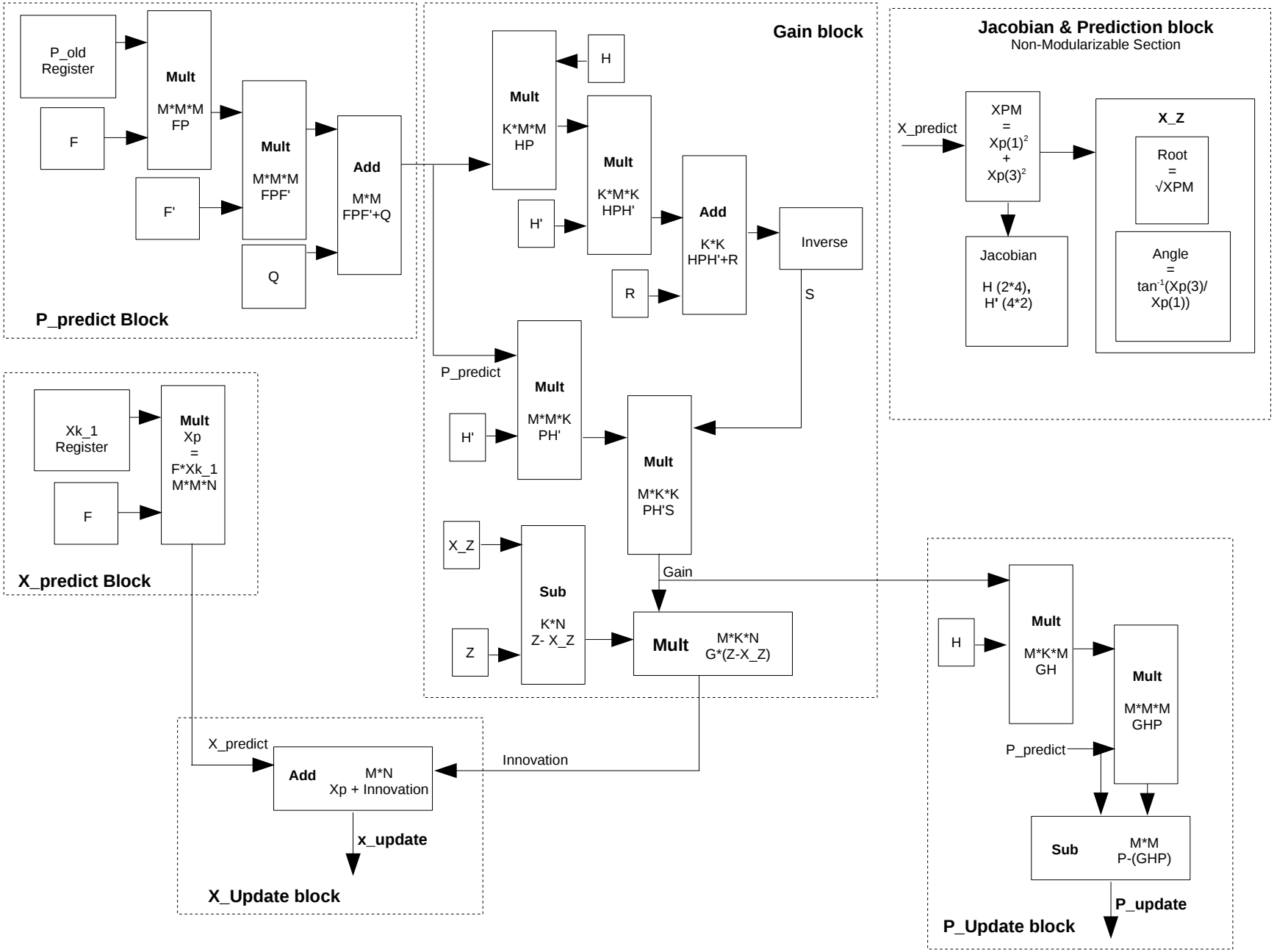

Figure 2.2: Major blocks in the data path of EKF implementation

2.2 Data representation format

IEEE 754 single precision floating point format was used in the hardware implementation. This format includes 1 bit to represent sign of the number S, 8 bits to represent the exponent E and 23 bits to represent the mantissa b. The word length W_{len} is therefore of 32 bits. A real number is thus represented as

$$\textbf{Value} = (-1)^S * 2^E * b_0 b_1 b_2 .....b_{22} \tag{2.5}$$

Here the exponent is biased by 127 , which is incremented or decremented appropriately to represent the floating point number. The range available with this format is quite large (maximum positive number that can be represented being $\approx 3.4 * 10^{38}$). The current implementation of extended Kalman filter does not require this high range. In fact, such a large word length increases the resource utilization and execution time exponentially. Usage of half precision floating point (16 bits word length with 5 exponent bits and 10 fraction bits) or lower word length fixed point format would have been sufficient for this implementation. However, adopting the single precision format provides the advantage of utilizing readily available Xilinx IP cores for numerical operations [9]. The implementation described in this report, has benefited from the use of many of these logicores [9], which provide reliable and accurate results on numerous arithmetic operations. Hence the use of single precision floating point data format makes sense.

2.3 Xilinx IP cores

IP cores (or logicores), are synthesizable logical blocks or units that are designed to perform some specific task, generally in the most optimum manner. They have standard interfaces or handshake signals which enable them to be incorporated in a design seamlessly. Xilinx provides a number of useful logicores suitable for different data formats. For the single precision floating point operations, logicores are available to perform addition, multiplication, subtraction, division, square root and more. Further, there are also cores that convert data format from single precision to fixed point and vice versa. In addition, logicores are also available to tackle trignometric functions under the CORDIC logicore. The implementation described in this chapter, utilizes all of these features provided under the floating point logicore.

From the data path figure 2.2, it can be seen that every module utilizes the arithmetic logicores while the Jacobian and prediction block uses arctan from the CORDIC IP core along with the associated data format conversions, in addition to the arithmetic logicores.

Most of the floating point math function logicores, provide customizability in terms of extent of utilization of DSP blocks and architecture optimization in terms of speed and latency. Choosing latency optimized architecture results in increased resource utilization. The floating point logicores provide options of latency and rate configuration, which determine the number of clock cycles required to generate an output, given an input. This latency can be set between 0 and specified maximum value, which is different for different operations, data widths and DSP usage [9]. The resource usage and maximum clock rate are affected by chosen latency values. The LUT and FF usage reduces with reduction in latency and so does the maximum operation frequency. This was observed in the implementation of EKF, where change in latency from maximum to a value of 2 drastically reduced the maximum operating frequency in addition to reducing the resource utilization.

2.4 Building blocks

The extended Kalman filter implementation mainly involves matrix operations such as matrix multiplication, matrix addition, matrix subtraction and matrix inversion. These operations in turn utilize basic arithmetic operations such as addition, multiplication, subtraction and division. Xilinx Logicore IP floating point operator version 5.0 has been used to perform the basic arithmetic opertions. The interface to the cores typically include input arguments, output argument, and handshake signals. Among the available handshake signals, signal for enabling the core (CE), signal for clearing the previous status of the core (SCLR), and ready signal (RDY) generated by the core to indicate the completion of operation, were used. The implementation follows the control path - data path approach. The control path is concerned with handshake signals such as ready, clear, and enable. The data path in turn comprises of multiplexers, IP cores, registers and so on and directly handle the data. The entire design is synchronous and finite state machine (FSM) based. Though there is a lot of scope for introduction of concurrency, the design introduces concurrency only in matrix addition and subtraction modules leaving the rest

of the design to follow a purely serial form of execution. Hence, at any given time, only one state - performing one operation is active in the sequential part.

2.4.1 Matrix addition

Matrix addition and subtraction blocks were built around the floating point adder and subtractor logicores provided by Xilinx. From the data path figure 2.2, it can be seen that except for the X predict block and Jacobian and prediction block, every other block needs matrix addition or matrix subtraction module.

Initially a serial adder/ subtractor was built which used a control path - data path format. This design used only one adder/subtractor and data was provided sequentially to be processed. Subsequently, in keeping with the endeavour towards introducing parallelism, a different design was used, where the data was provided to the adder module all at once and as many adders / subtractors as the matrix dimension, were instantiated. This approach works well as long as the matrix size is small. Resource utilization can get prohibitive for large matrix dimensions. A comparison of number of cycles versus resource utilization was performed for a 2 by 2 matrix of 32 bit floating point numbers. The observations are as tabulated in Table 2.2. It can be seen from table 2.2 that serial implementation uses fewer hardware resources with 0.22% of available slice registers and 1.134% of available slice LUTs, but is 10.4 times slower than the parallel implementation. The parallel implementation is resource greedy with usage of 0.32% of available slice registers and 3.857% of available slice LUTs.

Table 2.2: Serial and parallel adder comparison for 2×2 matrix of 32 bit word length numbers

Adder Type	Number of cycles	Resource Usage
Serial	26	Slice register = 209; Slice LUT = 528
Parallel	2.5	Slice register = 300; Slice LUT = 1796

2.4.2 Matrix multiplication

Matrix multiplication is one of the most important module in the EKF implementation. From the data path figure 2.2, it can be seen that the major modules of state and covariance prediction, gain calculation and covariance update, uses matrix multiplication of varying matrix dimensions. This calls for a generalized multiplier design that can be reused for multiple matrix dimensions. Usage of generics and parameters helped ensuring reusability across multiple matrix dimensions. However, the objective of introducing concurrency was not successful. Deploying as many floating point multipliers as the number of columns in first matrix, still necessitates the process of accumulation of products to be serial. This serial addition defeats the purpose of parallel multiplication. Hence a sequential design with one multiplier and one adder logicore, was adopted.

In literature Faddeev's algorithm [6],[14] and Systolic array architecture [6],[14] are popular dense matrix multiplication algorithm. Though faster, the method would involve use of multiple multipliers and adders in each processing element in the output matrix. There may be re-usability to some extent but at the cost of making the design more complex by introducing multiplexers, demultiplexers and many more registers with their corresponding addressing logic. Newer implementations, where matrix multiplication plays an important role [2], [4], use methods such as C to hardware or hardware software co-design and hence do not specify the method used for matrix multiplication.

The matrix multiplication algorithm used in this implementation emulates the nested for loop algorithm used in high level languages. The pseudocode for matrix multiplication is as in listing 2.3.

The module is built around a multiplier and an adder logicore and computes the multiplication serially. An user defined array type was defined for the matrices, the dimensions of which were decided using the 'generic' feature provided by VHDL. The indices to access specific matrix locations, were generated by counters. Comparators were used to keep track of the index limit. The control signals for enabling and clearing the counters, accumulation registers and the logicores, were generated from the control path of matrix multiplication finite state machine. Status signals from the data path indicating the status of processes such as multiplication and addition, or the comparator status for

Table 2.3: Pseudocode for matrix multiplication

```
arg0 = mat_mult(arg1, arg2, row1, col1row2, col2)
    for i=1:1:row1
        for j=1:1:col2
            arg0(i,j)=0;
            for k=1:1:col1row2
              arg0(i,j)= arg0(i,j) + (arg1(i,k)*arg2(k,j)
            end
         end
     end
```

each index comprised the input signals to the control path finite state machine. A block diagram of the multiplier design is as shown in Figure 2.3.

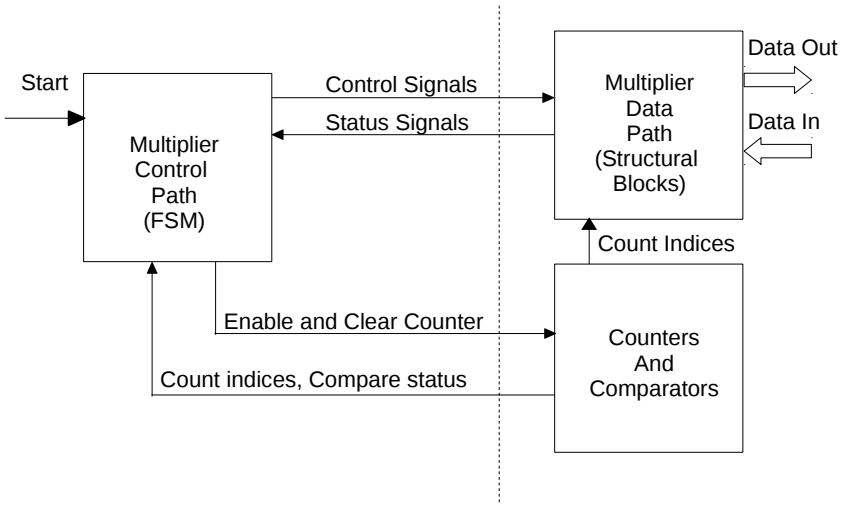

Figure 2.3: Matrix multiplication design

2.4.3 Matrix Inversion

Computation of inverse of a matrix **S** implies finding a matrix **X** such that the product of **S** and **X** results in an Identity matrix **I**. Mathematically, this is expressed as $\mathbf{S} * \mathbf{X} = \mathbf{I}$,

where $\mathbf{S}$, $\mathbf{X}$ and $\mathbf{I}$ belong to $\mathbb{R}^{N\times N}$. This expression is similar to the simultaneous linear equations form $AX = B$, whose solution is computed using Gauss elimination method. The process of forward elimination is the most computationally intensive part of Gauss elimination algorithm. This process inherently uses both the left hand side and right hand side of the equation $AX = B$. If B is a vector (i.e. matrix of dimension $N \times 1$), then the one time computational expense incurred during forward elimination is acceptable. However, as in the case of matrix inverse computation, if the right hand side of the expression $AX = B$ is also an $N \times N$ matrix, then the Gauss elimination process has to be repeated N times - once for every column vector. The computation involved can thus get prohibitive. This calls for an algorithm that factors the matrix, independent of the right hand side, and finally uses the factored parts repeatedly for the different right hand side columns. LU decomposition process satisfies this requirement and is often called a better method to implement Gauss elimination method.

LU decomposition method is an $O(N^3)$ process while Gauss elimination is an $O(N^4)$ method, where N is considered the number of rows or columns. Also designing a general purpose matrix inversion algorithm is one of the key steps in realization of modular EKF.

In [11] one of the papers on hardware implementation of matrix inversion algorithm, the authors have used Gauss Jordan (GJ) elimination algorithm as a low complexity solution for matrix inversion. The low number of memory accesses and use of only three different arithmetic operations viz. addition, multiplication and division, has been proposed as the advantage of using GJ elimination method. The Matlab VHDL code automatic generator was used to generate VHDL code. IP cores have been used to perform arithmetic operations. The design was ported and tested on Virtex 5 at a clock rate of 50MHz. The objective of the authors was to design a scalable low area matrix inversion solution. In [4] authors use adjoint-determinant computation method to calculate the matrix inversion, which is an extremely restrictive approach in terms of scalability. In that respect, the LU decomposition approach followed in this report is better.

In [12], Cholesky decomposition method has been used for matrix inversion. The arithmatic operations involved in Cholesky decomposition includes calculation of square root, in addition to division, multiplication and subtraction. This would imply additional hard-

ware utilization as against LU decomposition. The computational complexity is $O(N^3)$. As in LU decomposition, the lower triangular matrix is first derived and subsequently, two equations viz. $L^T y = b$ and $Lx = y$ are solved. The steps closely resemble LU decomposition method. While Cholesky decomposition is supposed to require only half the computations of LU decomposition, they require the matrix to be positive definite. This method does provide an attractive alternative to LU decomposition, though with increased hardware utilization. However, this report describes a scalable matrix inversion implementation using LU decomposition with only basic arithmatic operations of addition, subtraction, multiplication and division, achieving accuracy sufficient for the EKF implementation.

Briefly, LU decomposition method can be described as follows. Suppose the matrix inversion equation is represented as

$$\mathbf{SX} = \mathbf{I} \tag{2.6}$$

Matrix **S** can be factored as a product of upper (**U**) and lower (**L**) triangular matrices such that

$$\mathbf{LU} = \mathbf{S} \tag{2.7}$$

Substituting for **S** in equation 2.6, we get

$$\mathbf{LUX} = \mathbf{I} \tag{2.8}$$

Substituting

$$\mathbf{UX} = \mathbf{Z} \tag{2.9}$$

in equation 2.8 we get,

$$\mathbf{LZ} = \mathbf{I} \tag{2.10}$$

Once **Z** is obtained, substituting in equation 2.9

yields **X**, which is the required inverse of the matrix **S** [10]. The VHDL implementation of LU decomposition involves three sections viz.

- Factoring of the given matrix to upper (**U**) and lower (**L**) triangular parts
- Solving equation 2.10
- Solving equation 2.9

Each of these three parts uses control path - data path approach. The control path is a Moore machine. It deals mainly with enabling and clearing different data blocks towards controlling data flow. The data blocks include counter, comparator, multiplexer, registers and logicores that perform floating point arithmetic operations. The functionality of 'nested-for' loops to generate matrix indices, is performed by the counters and comparators. The mathematical operations to be performed include division, multiplication, addition and subtraction. Logicores provided by Xilinx have been used to perform these operations. Registers are used to store the intermediate and final matrices which include the upper and lower triangular matrices, the Z matrix and finally the inverse or Y matrix. The pseudocode for obtaining upper and lower triangular matrices are as shown in listing 2.4.

Figure 2.4 and 2.5 depict the control path and data path for matrix factorization.

The pseudocode for solving equation 2.10 is as shown in listing 2.5.

Figure 2.6 and 2.7 depict the control path and data path for solving the equation 2.10.

The pseudocode for solving equation 2.9 is as shown in listing 2.6.

Figure 2.8 and 2.9 depict the control path and data path for solving the equation 2.9.

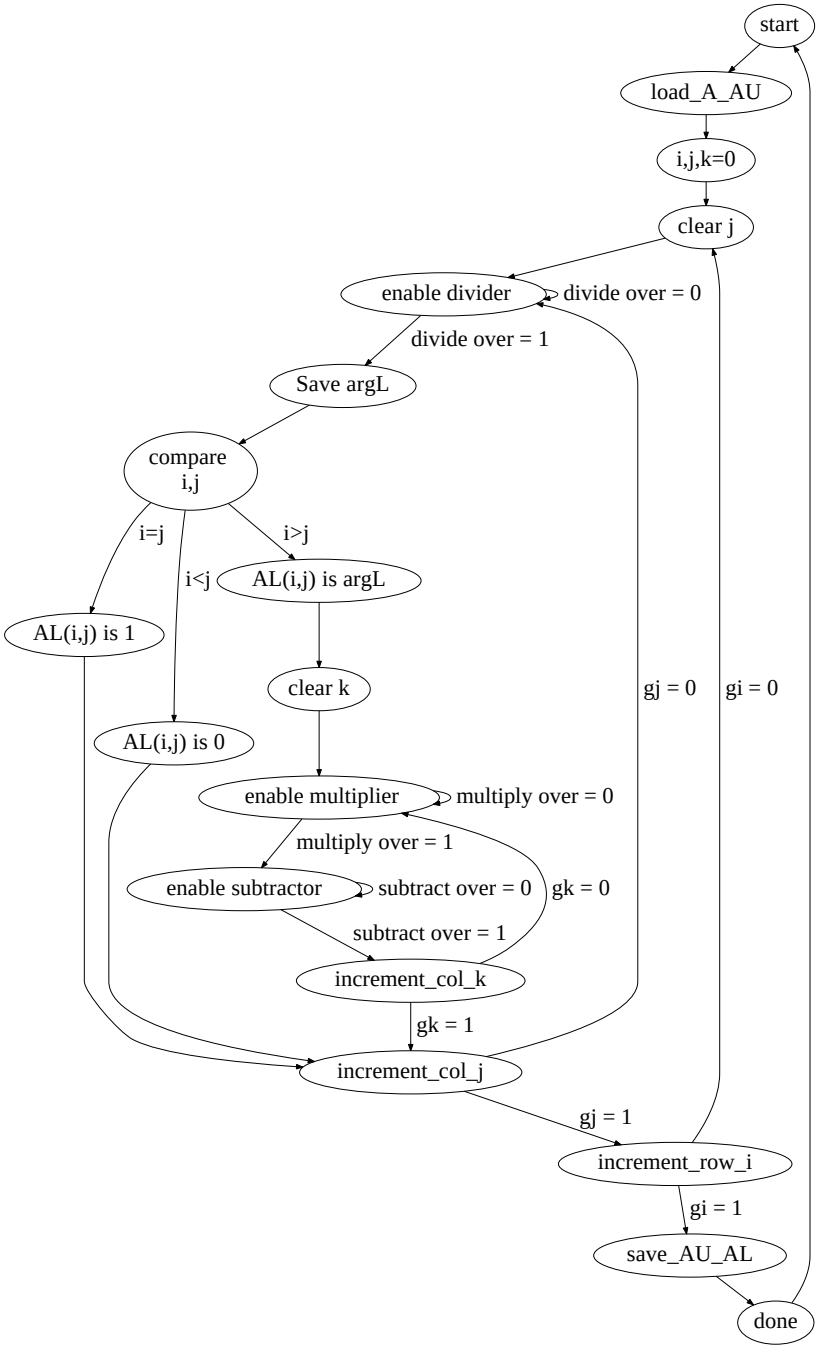

Figure 2.4: Matrix factorization FSM

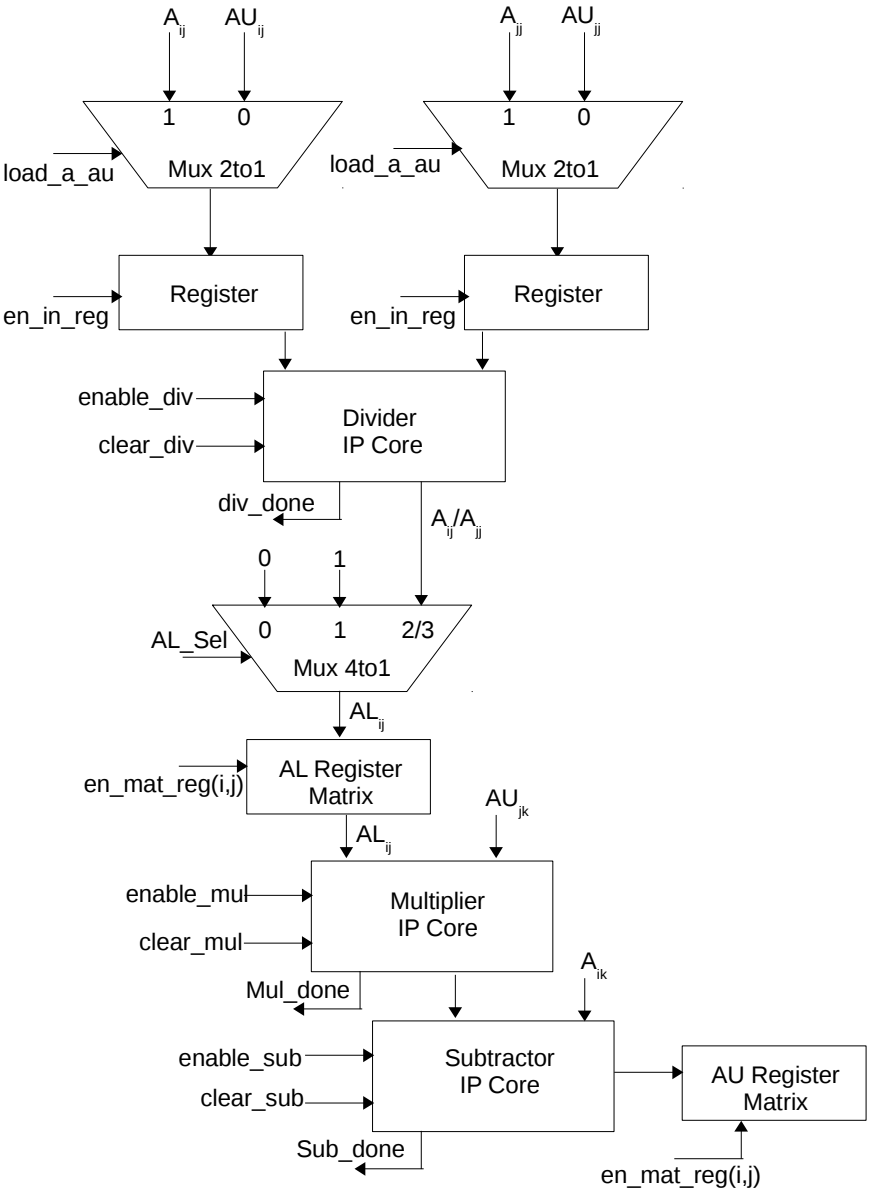

Figure 2.5: Matrix factorization Data Path

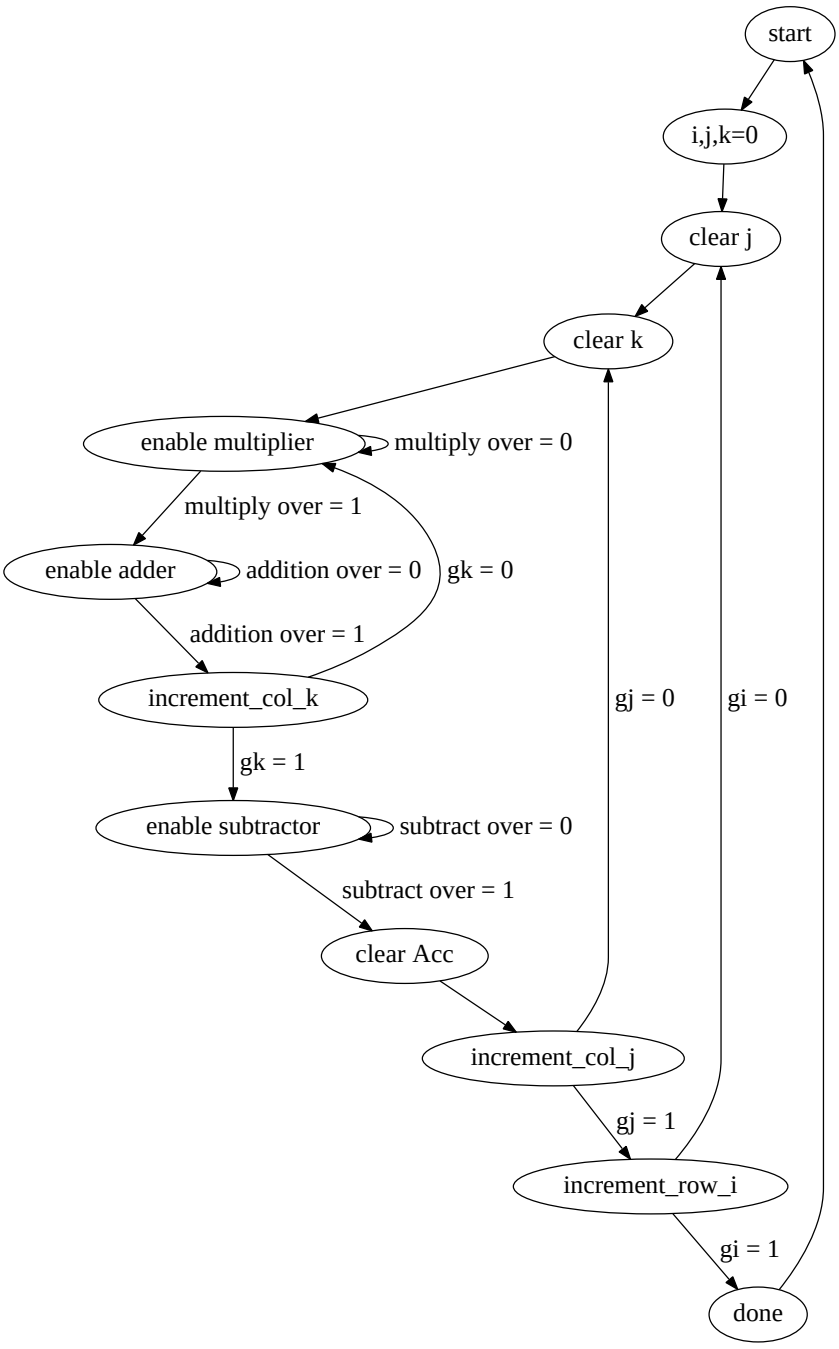

Figure 2.6: LZ part FSM

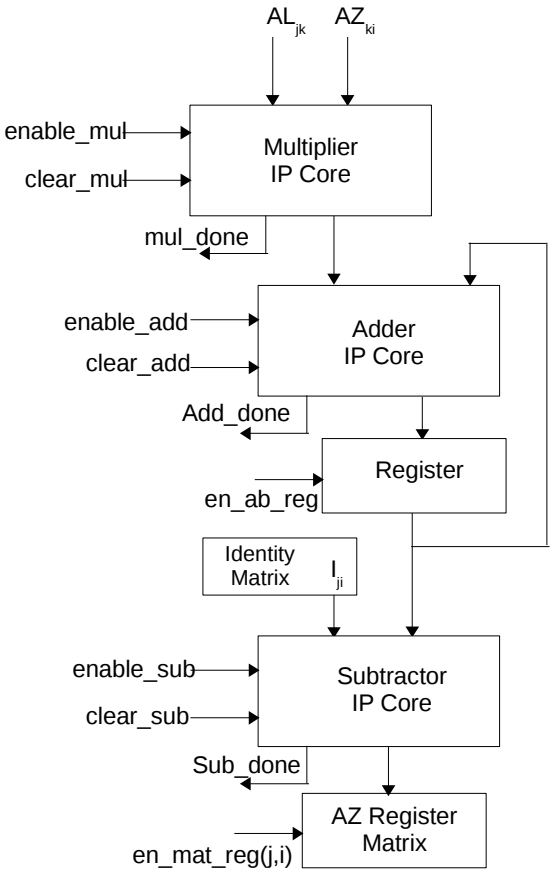

Figure 2.7: LZ part Data Path

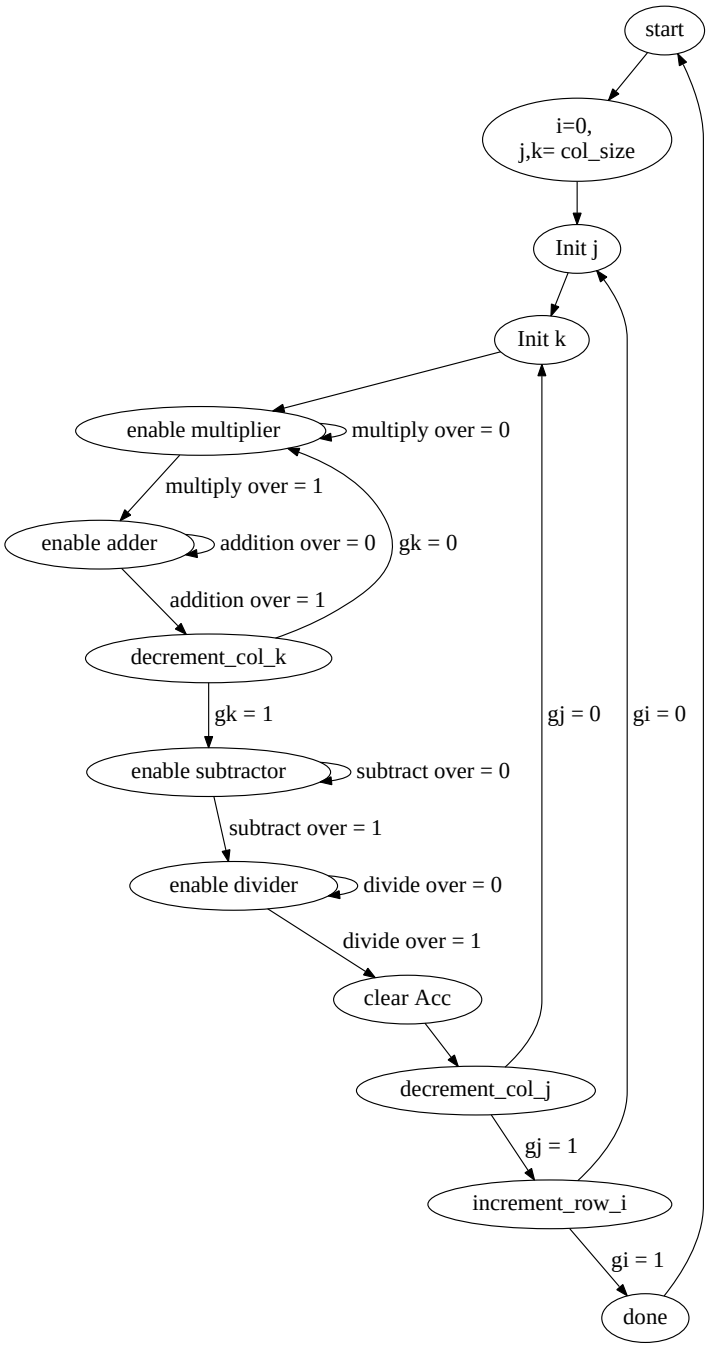

Figure 2.8: UX=Z part FSM

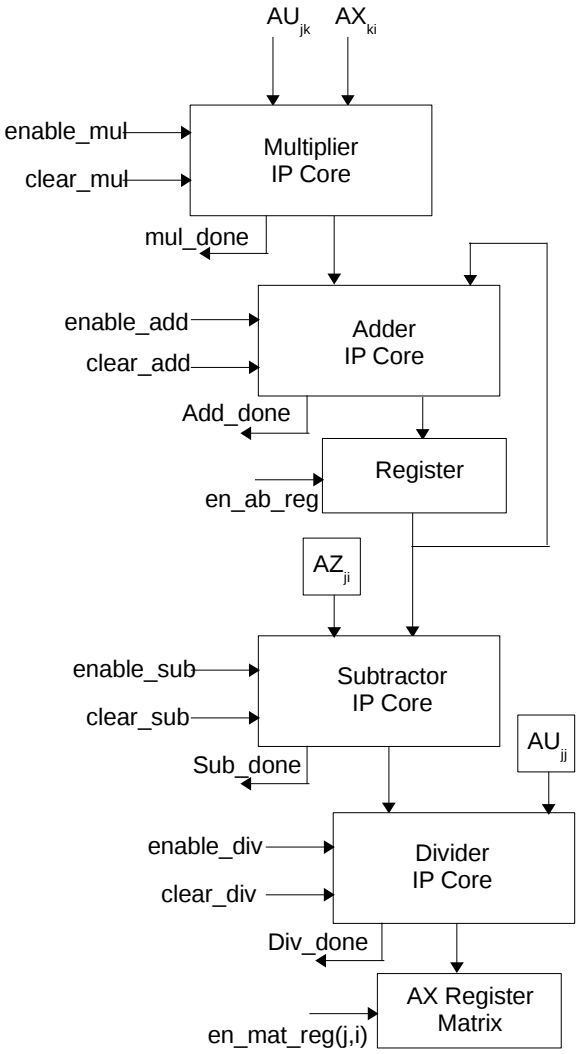

Figure 2.9: UX=Z part Data Path

Table 2.4: Pseudocode for matrix factorization

```
[mat_l,mat_u] = mat_tri(mat_a, row, col)
for i=1:1:row
    for j=1:1:col
        if (i==j)
            mat_l(i,j) = 1
        end
        if i<j
            mat_l(i,j) = 0
        end
        if i>j
            mat_l(i,j) = mat_a(i,j) / mat_a(j,j)
            for k=1:1:col
                temp = mat_l(i,j) * mat_a(j,k)
                mat_u(i,k)= mat_a(i,k) - temp
            end
            mat_a = mat_u;
        end
     end
end
```

Table 2.5: Pseudocode for solving equation 2.10

```
[arr_z] = lz(mat_l, row, col)
for k=1:1:col
    for i=1:1:row
        for j=1:1:col
            temp = temp + (mat_l(i,j) * arr_z(j,k))
        end
        arr_z(i,k) = identity(i,k) - temp
        temp = 0
    end
end
```

Table 2.6: Pseudocode for solving equation 2.9

```
[arr_x] = ux(mat_u, arr_z, row, col)
for i=1:1:col
    for j=row:-1:1
        for k=col:-1:1
            temp = temp + (mat_u(j,k) * arr_x(k,i))
        end
        arr_x(j,i) = (1 / mat_u(j,j)) * (arr_z(j,i) - temp)
        temp = 0
    end
end
```

Chapter 3

Simulations

3.1 Extended Kalman filter simulation

The ground truth for the extended Kalman filter model used in this design, were generated from a Matlab program. Observations were generated in Matlab by adding noise to the true state. The true states are used only for the final result comparison. The observations are however used as test bench inputs to evaluate the error between the states predicted in EKF estimate and the actual measurement. The input state vector was initialized to an initial value of

$$\mathbf{x}_{init} = \begin{bmatrix} 100 & 20 & 95 & 20 \end{bmatrix}^T \tag{3.1}$$

The time factor T in state transition model was set to 1. The covariance matrix was intitalized to

$$\mathbf{P}_{init} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix} \tag{3.2}$$

The process noise covariance matrix used was

$$\mathbf{Q} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4.75 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

and the measurement noise covariance matrix used was

$$\mathbf{R} = \begin{bmatrix} 10 & 0 \\ 0 & 0.01745 \end{bmatrix} \tag{3.4}$$

The target device was Virtex-6 FPGA (xc6vlx75t-1-ff484). The simulation was run for a 100 time steps and the state updates obtained at each step, was plotted along with state update values obtained from a pure Matlab simulation and the expected true state at that time step. The plots thus obtained are as seen in Figures 3.1 to 3.3.

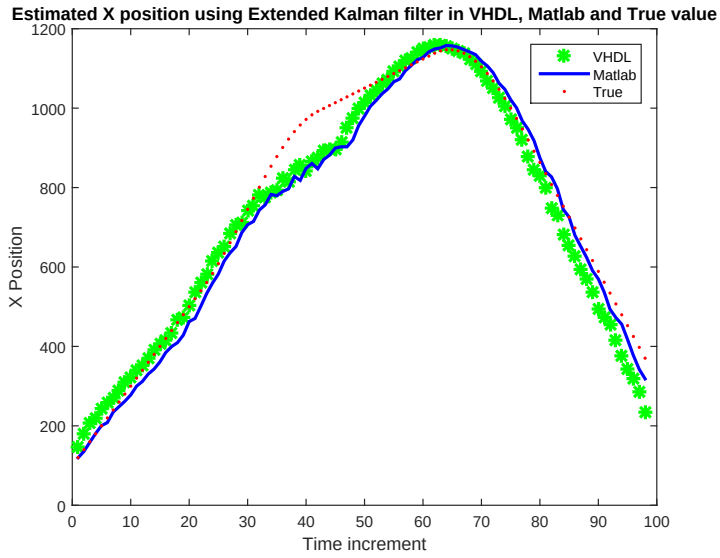

Figure 3.1: True and estimated x-Position in Extended Kalman filter implementation

From the plots in figures 3.1 to 3.3, it is seen that the x, y positions and xy trajectory are faithfully captured in the VHDL implementation, similar to the Matlab implementation and the non-linear tracking response of the VHDL implementation is similar to that of Matlab implementation.

3.2 Timing and logic utilization

Timing and logic utilization analysis was carried out for the target device xc6vlx75t. The logic resources available on xc6vlx75t include 11,640 slices, 74,496 logic cells, 93,120 CLB flip-flops and 288 DSP48E1 Slices.

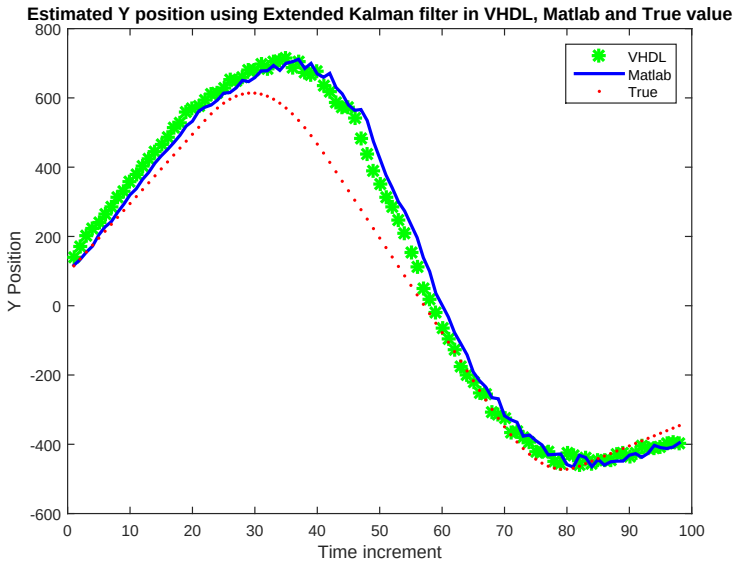

Figure 3.2: True and estimated y-Position in Extended Kalman filter implementation

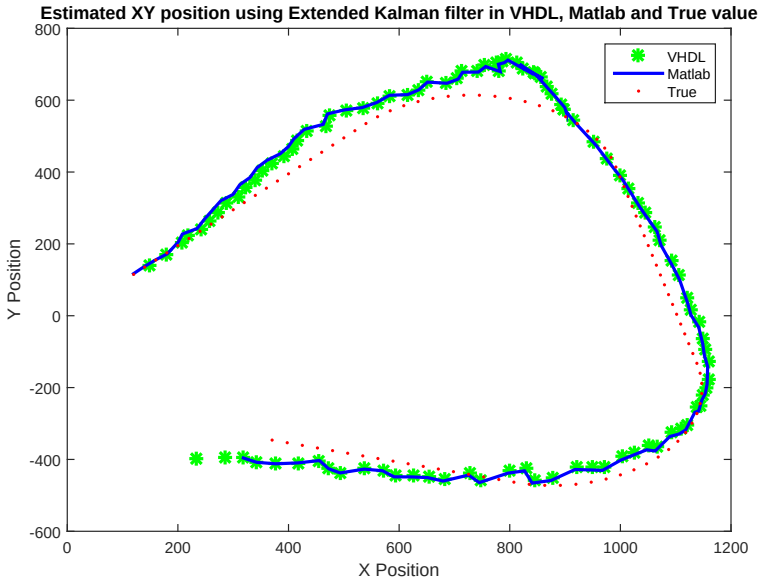

Figure 3.3: True and estimated xy-trajectory in Extended Kalman filter implementation

Logicore configuration

The configuration used in the logicores are as summarized in Table 3.1

Table 3.1: Maximum clock cycle latency Logicore configuration

IP core	Data format	Family optimization	Clock cycle latency	Architecture optimization
Multiplier	Single	Logic only	Maximum (8)	-
Adder	Single	Logic only	Maximum (12)	High speed
Square root	Single	-	Maximum (28)	-
Divider	Single	-	Maximum (28)	-
Subtractor	Single	Logic only	Maximum (12)	High speed
Float to fixed	float : Single fixed : 2 int, 30 fraction	-	Maximum (6)	-
Fixed to float	fixed : 3 int, 29 fraction float : Single	-	Maximum (6)	-
Arc tan	32 bits, radians, truncated 10 iterations, coarse rotation	-	-	-

Logic utilization

Table 3.2 summarises the logic utilization obtained for the Extended Kalman filter implementation. Computation of Jacobian is a resource intensive process requiring additional logicore usage for the square root operation. Further, presence of non-linearity in the system, necessitates the use of CORDIC logicore and the data format conversion logicores. This further adds to resource utilization in the FPGA.

Table 3.2: Logic Utilization in EKF implementation with maximum clock cycle latency

Parameter	Actual Used	Available	Percentage Utilization
Number of slice registers	58348	93120	62%
Number of slice LUT	45481	46560	97%
Number of LUT used as logic	42189	46560	90%
Number used as memory	3292	16720	19%
Number of fully used LUT FF pairs	36443	67386	54%
Number with unused FF in LUT FF pairs	9038	67386	13%
Number with unused LUT in LUT FF pairs	21905	67386	32%

Execution time

From the post synthesis summary, the maximum frequency of operation was obtained as $330.834MHz$. The Extended Kalman filter VHDL implementation was executed for 100 iterations. The number of clock cycles needed to execute 100 iterations were 1138650. Therefore at $330.834MHz$, time required for execution is $3441.7562\mu s$. Thus for 1 iteration, the time required for execution is $\approx 34.417\mu s$. Table 3.3 gives the time taken at different stages in Extended Kalman filter execution.

The Matlab implementation was also timed to get the approximate execution time. The system on which Matlab code was executed has an Intel core i7 CPU operating at $3.20GHz$. It also has cache of $8192KB$ and has 8 cores. Execution time for a 100 iterations in the Matlab Extended Kalman filter code, took approximately $0.003809seconds$. Therefore, roughly, one run would take $\approx 38.09\mu s$. However, this reading may be far from accurate as the processor may be executing other processes in between, thereby clocking different times for different runs. The EKF implementation in [2] reports a maximum operating frequency of $70MHz$. However the architecture, design approach and matrix sizes are totally different and may not be comparable with the implementation in this report. The hardware-software co-design approach used in [4], has input dimensions similar to the

implementation in this report. The implementation on Cyclone IV FPGA device, reports maximum operating frequency of 49.08MHz. Compared to this, the implementation in this report seems faster. However, the target devices used in the paper and in this work are not the same and so this comparison may not be sufficient and valid.

Table 3.3: Execution time for various stages in EKF implementation with maximum clock cycle latency

Stage	Time in microseconds	Number of clock cycles
State prediction $\hat{\mathbf{x}}_k$	$1.4162\mu s$	467.5
Jacobian $\nabla\mathbf{H}$	$0.6089\mu s$	201
$arctan(\frac{y}{x})$	$0.2060\mu s$	68
Covariance prediction $\hat{\mathbf{P}}_k$	$11.3596\mu s$	3750
S	$4.3076\mu s$	1422
S^{-1}	$2.7899\mu s$	921
$Gain$	$4.3318\mu s$	1430
State update $\mathbf{x}_k$	$0.8270\mu s$	273
Covariance update $\mathbf{P}_k$	$8.6454\mu s$	2854

Variations in the logicore settings in terms of number of cycles latency was made to estimate the effect of logicore configuration changes on hardware utilization and the maximum frequency of operation.

Analysis when latency in number of clock cycles is 2

Logicore configuration

The configuration used in the logicores were as summarized in Table 3.4. In addition to reducing the number of clock cycles in the logicores, DSP blocks were used for the subtraction logicore. This is because, in the current implementation, the number of

subtractors used is less as compared to adders or multipliers. Thus the limited DSP blocks can be used for subtractor logicores.

Table 3.4: Logicore configuration with clock cycle latency of 2

IP core	Data format	Family optimization	Clock cycle latency	Architecture optimization
Multiplier	Single	Logic only	2	-
Adder	Single	Logic only	2	High speed
Square root	Single	-	2	-
Divider	Single	-	2	-
Subtractor	Single	full usage of DSP48E 2*DSP48E	2	High speed
Float to fixed	float : Single fixed : 2 int, 30 fraction	-	2	-
Fixed to float	fixed : 3 int, 29 fraction float : Single	-	2	-
Arc tan	32 bits, radians, truncated 10 iterations, coarse rotation	-	-	-

Logic utilization

Table 3.5 gives the logic utilization obtained for the extended Kalman filter implementation. It can be seen from the logic utilization that, with lower latency in clock cycles, the hardware utilization also reduced by a small margin.

Execution time

From the post synthesis summary, the maximum frequency of operation was obtained as $21.387MHz$. The extended Kalman filter VHDL implementation was executed for

Table 3.5: Logic Utilization in EKF implementation when clock cycle latency is 2

Parameter	Actual Used	Available	Percentage Utilization
Number of slice registers	16116	93120	17%
Number of slice LUT	43296	46560	92%
Number of LUT used as logic	43293	46560	92%
Number used as memory	3	16720	0%
Number of fully used LUT FF pairs	6773	52639	12%
Number with unused FF in LUT FF pairs	36523	52639	69%
Number with unused LUT in LUT FF pairs	9343	52639	17%

100 iterations. The number of clock cycles needed to execute 100 iterations were 505650. Therefore at $21.387MHz$, time required for execution is $23642.867\mu s$. Thus for 1 iteration, the time required for execution is $\approx 236.4287\mu s$. Table 3.6 gives the time taken at different stages in extended Kalman filter execution.

From the comparison of performance between maximum clock cycle latency and low (2) clock cycles latency configurations of EKF implementation, it is seen that with reduction latency in number of clock cycle, the hardware utilization reduces marginally as seen in tables 3.2 and 3.5. However, the performance in terms of execution time alters drastically with an increase of ≈ 7 times. The objective behind looking for lower hardware utilization mode was to fit the design on the FPGA (xc6vlx75t-1-ff484). However, the deterioration in performance in terms of speed makes this option unacceptable. As an alternative, the maximum clock cycle latency design was ported on the next larger version of Virtex 6 FPGA - (xc6vlx130t-1-ff484). The hardware utilization came down by more than 35% as the resources available on this FPGA is much more. Meanwhile the maximum operating frequency remained at $330.834MHz$, which is by far a more acceptable speed.

Table 3.6: Execution time for various stages in EKF implementation when clock cycle latency is 2

Stage	Time in microseconds	Number of clock cycles
State prediction $\hat{\mathbf{x}}_k$	$9.8892\mu s$	211.5
Jacobian $\nabla\mathbf{H}$	$2.2911\mu s$	49
$arctan(\frac{y}{x})$	$0.9351\mu s$	20
Covariance prediction $\hat{\mathbf{P}}_k$	$79.1135\mu s$	1692
S	$30.1118\mu s$	644
S^{-1}	$16.1313\mu s$	345
$Gain$	$30.9534\mu s$	662
State update $\mathbf{x}_k$	$5.8447\mu s$	125
Covariance update $\mathbf{P}_k$	$61.1589\mu s$	1308

Chapter 4

Conclusion

This report covers the details of hardware implementation of Extended Kalman filter for a constant turn model. The design satisfies the requirement of a modular general purpose design, to a large extent. At individual module level comprising of matrix operations, the design is scalable. The scalability of the design as a whole still needs to be verified. The design achieves maximum operating frequency of $330.834MHz$. Other implementations referred to in this report achieve maximum operating frequency of about $70MHz$ [2] and $50MHz$ [4]. The EKF implementation is predominantly serial, with only the matrix addition and matrix subtraction modules being concurrent. With the current architecture, it is difficult to make the matrix multiplication and the LU decomposition modules concurrent. At each stage of the implementation, there is a stall in data flow to allow for completion of processing of all data so that the next stage is presented with a complete data set from the previous stage. Modifications in the architecture to work with partially available data could keep the data flow continuous and make the hardware work at a higher clock rate. However, introduction of such an approach would make the design extremely complex. Also, in order to benefit form the existing logicores, the word length was maintained at 32 bits, which is largely unnecessary for the current simulations. A decrease in word length can drastically reduce the execution speed, area and resources utilized, making for a much lighter and faster design.

Chapter 5

References

References

[1] Hongyan Guo, Hong Chen, Fang Xu, Fei Wang, and Geyu Lu,"Implementation of EKF for Vehicle Velocities Estimation on FPGA", IEEE Transactions on industrial electronics, Vol. 60, No.9, September 2013.

[2] Vanderlei Bonato, Eduardo Marques, George A.Constantinides, "A Floating-point Extended Kalman Filter Implementation for Autonomous Mobile Robots", J Sign Process Syst (2009) 56:41–50.

[3] Vanderlei Bonato, Rafael Peron, Denis F. Wolf, Jose A. M. de Holanda, Eduardo Marques, Joao M. P. Cardoso, "An FPGA implementation for a Kalman filter with application to mobile robotics", SIES '07. International Symposium on Industrial Embedded Systems, 2007.

[4] Sergio Cruz, Daniel M. Munoz, Milton Conde, Carlos H. Llanos, Geovany A. Borges, "FPGA Implementation of a Sequential Extended Kalman Filter Algorithm Applied to Mobile Robotics Localization Problem", IEEE Fourth Latin American Symposium on Circuits and Systems (LASCAS), 2013.

[5] X. Rong Li and Vesselin P. Jilkov, "A Survey of Maneuvering Target Tracking—Part III: Measurement Models", Proceedings of SPIE Conference on Signal and Data Processing of Small Targets, San Diego, CA, USA, July-August 2001.

[6] Padma Rao and Magdy A Bayoumi, "An efficient VLSI implementation of real-time kalman filter", IEEE International Symposium on Circuits and Systems, 1990.

[7] Dan Simon, "Optimal State Estimation Kalman, H Infinity and Nonlinear Approaches", Wiley-Blackwell. 2006

[8] "Xilinx Virtex 6 family overview", 2012.

[9] "Xilinx LogiCORE IP Floating-point operator v6.0 product specification", 2012.

[10] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, "Numerical Recipes in C: The Art of Scientific Computing".

[11] Janier Arias-Garcıa, Ricardo Pezzuol Jacobi, Carlos H. Llanos, Mauricio Ayala-Rincon, "A suitable FPGA implementation of floating-point matrix inversion based on Gauss-Jordan elimination", VII Southern Conference on Programmable Logic (SPL), 2011.

[12] Pentu Salmela, Aki Happonen, Adnan Buriant, and Janno Takala, "Several Approaches to Fixed-Point Implementation of Matrix Inversion", International Symposium on Signals, Circuits and Systems, 2005. ISSCS 2005.

[13] "Virtex-6 FPGA Configurable Logic Block User Guide", 2012.

[14] Mi Lu, Xiangzhen Qiao, Guanrong Chen, "A parallel square-root algorithm for modified extended Kalman filter", IEEE Transactions on Aerospace and Electronic systems VOL. 28. NO. 1 JANUARY 1992

Acknowledgment

First and foremost, I would like to express my sincere thanks to my guide Prof. V Rajbabu for his valuable inputs, patience and encouragement during the course of this work.

I would also like to thank Prof. Shankar Balachandran for his inputs on methods to introduce modularity and parallelism in the design. This work has benefited from his suggestions.

Insightful discussions with Naveen K and Vinay B. Y. Kumar have helped in locating and rectifying various flaws in the design. My heartfelt thanks to them. I would also like to acknowledge the help of Kulshreshth Dhiman and Aakash Paacharne for their sound suggestions to my queries.

Finally I would like to thank everybody who have helped me in every way in my project, enabling me to complete this work.

N.Soumya