ACKNOWLEDGEMENTS

I would like to offer my gratitude to my family who gave me the support and encouragement I needed while working on this project. Thanks also to fellow students past and present who gave advise and feedback on areas of difficulty. Lastly and most importantly I would like to thank my supervisor, Michael Manzke who was not only instrumental in my project progress, but who also showed understanding and compassion when they were needed the most. Special thanks to all of the above.

Contents

ABSTRACT

The overall aim of this project was to design, implement and evaluate the performance of a kalman filter using FPGAs. The projects main concern is the design of a synthesisable VHDL model for the algorithm which defines this filter. From the project viewpoint it was not essential for me to become an expert in minimum mean square error filtering and state space methods. What was required, however, was for me to be familiar with the algorithm, that defines the kalman filter. The set of equations, their relevance to one another and indeed the overall functionality of the algorithm required complete comprehension. If successful the resulting program would then be implemented with field programmable gate arrays, enabling the end result to be appreciated visually.

Chapter 1

First Chapter

1.1 Introduction

The overall aim of this project was to design, implement and evaluate the performance of a kalman filter using FPGAs. The projects main concern is the design of a synthesisable VHDL model for the algorithm which defines this filter. From the project viewpoint it was not essential for me to become an expert in minimum mean square error filtering and state space methods. What was required, however, was for me to be familiar with the algorithm, that defines the kalman filter. The set of equations, their relevance to one another and indeed the overall functionality of the algorithm required complete comprehension. If successful the resulting program would then be implemented with field programmable gate arrays, enabling the end result to be appreciated visually. This chapter firstly presents these equations, describes their meaning in terms of the next and the previous equation and outlines what would be needed for successful implementation in a hardware descriptive language. A complete breakdown of all follows.

1.1.1 The Kalman Filter Equations

The kalman filter equations are a set of mathematical equations that provide an efficient computational means to estimate the state of a process, in a way that minimizes the mean of the squared error. The filter is a very powerful device as it supports the estimation of past, present and future states. It even extends its functionality so it can carry out this procedure when the precise nature of the modelled system is unknown. The system may or may not be subjected to a series of random disturbances, when this occurs it is required to estimate the state variables from noisy observations. The kalman filter equations uses two different types of equations in a prediction of the state variable, these being both the time update equations and the measurement update equations.

1.2 Two Different Sets Of Equations

The filter estimates its process by using a form of feedback control, as implied in the previous section. The filter will estimate the process state at some time and then obtains its feedback in the form of noisy measurements. These equations fall into the category of either Time update equations or measurement update equations.

1.2.1 Time Update Equations

The time update equations are used to predict the current state and covariance matrix, used in time t+1 to predict the previous state. These equations can be generally seen as predictor equations as they are responsible for projecting forward in time. K is representative of the time step, so the time update equations are basically indicative of K+1.

1.2.2 Measurement Update Equations

The measurement equations are responsible for feedback and for correcting the errors that have been made in the time update equations. In a sense they are back propagating to get new values for the prior state to improve the "guess" for the next state. These equations can be seen as corrector equations and the final estimation algorithm resemble that of a predictor-corrector algorithm. So by definition measurement equations adjust the projected estimate by an actual measurement at that time.

1.3 Specific Kalman Filter Equations And Their Role

$$K_k = P_{\bar{k}} H^T (H P_{\bar{k}} H^T + R)^{-1} \tag{1.1}$$

$$\hat{x}_k = \hat{x}_{\bar{k}} + K_k (z_k - H\hat{x}_{\bar{k}}) \tag{1.2}$$

$$P_k = (I - K_k H) P_{\bar{k}} \tag{1.3}$$

$$\hat{x}_{\bar{k}} = A\hat{x}_{k-1} + Bu_k \tag{1.4}$$

$$P_{\bar{k}} = AP_{k-1}A^T + Q \tag{1.5}$$

The initial task when dealing with the measurement update equation is to compute the kalman gain denoted by Kk. K is an n by m matrix and is chosen to be a blending factor or a gain factor that minimizes the error covariance. The next step is to measure the process to obtain a value for Zk and then to generate a next state estimate incorporating the previous result for the kalman gain equation. The last of the measurement update equations is responsible for obtaining the posterior error covariance which is denoted by Pk. After each time and measurement update equation, the process is repeated with the previous 'a posteriori' estimates used to project the new 'a priori' estimates. The fact that the kalman filter is designed with recursive functionality is one of its many appealing characteristics when placed in contrast with alternative filters.

1.3.1 Additional Variable In The Equations

$$P_k : PriorErrorConvergence \tag{1.6}$$

$$K : KalmanGain \tag{1.7}$$

$$Z_k : StateMeasurement \tag{1.8}$$

$$\hat{x} : PosteriorStateEstimate \tag{1.9}$$

$$R_k : MeasurementErrorCovariance \tag{1.10}$$

$$Q_k : RandomWhiteNoise \tag{1.11}$$

$$A_k : Variable \tag{1.12}$$

$$B_k : Variable \tag{1.13}$$

$$\mu_k : ControlVariable \tag{1.14}$$

$$H_k : Matrix - valuedFunction \tag{1.15}$$

Above is a table which provides a definition of additional variables in the kalman filter equations. In the equations , a measurement of the process, Zk and Xk are previously defined by linear stochastic difference equations equations. The random variables in these equations, w and v represent the process and measurement noise and are assumed independent of one another. They are also assumed to possess normal probability distribution. For practical examples, process noise covariance Q and measurement noise covariance R matrices, might change with each time step or measurement. However for the purposes of my project, I have assumed them to be constant values. A is an n by n matrix in the difference equation and relates the state at the previous time step k - 1 to the state at the current time step k, without the presence of process noise. Once again A is assumed to be fixed despite the fact that this would more realistically be susceptible to change with each time step. Matrix B relates the control variable to the state x. Matrix H relates the state to the measurement Zk.

Chapter 2

Second Chapter

2.1 Background Mathematics

A complete examination of matrices, their functionality and effects for my particular design was the next step required. This chapter consists of an overview of vector and matrix fundamentals acquired previously in the course and reviewed and revised courtesy of linear algebra books and websites. A little time was allocated to using matrices in matlab which was in itself beneficial and worthwhile.

2.1.1 Vectors and Matrices

As is probably evident at this stage all equations are comprised of matrices, vectors or single values. Some arithmetic function is performed on these components to result in the output value which is passed to the next equation. Below I have taken each equation independently and examined the operations that are required to provide the output function.

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1} \tag{2.1}$$

In this equation P, H and R are all matrices. For the purpose of my project I assumed that all matrices were n by n despite the fact that the kalman gain itself can be defined in terms of an n by m matrix. With the aid of more time, experience and expertise this could have been accounted for. H is transposed and multiplied by matrix P, the result is stored. The transpose of H is multiplied by P and the result is multiplied by H, this is then added to R. This new result is inverted and multiplied by the previously stored result which determines the kalman gain. The Kalman gain is then taken in by the next equation where similar operations are performed. Variables which posess the hat are vector values with all the other variables representing matrices. The time update equations are basically similar with the primary difference being that the time step has now been updated and we are looking at new time t and new time k.

A Breakdown of the Algorithm

After the algorithm was successfully broken down, it was evident that understanding every single part of this algorithm came a poor second to actually being able to implement it in VHDL. Although the algorithm primarily had all the appearance of being intriguingly complex and indeed the theory behind it perplexing at times it soon became obvious that its implementation required no more than a full comprehension of mathematical operations on n sized matrices and vectors which had previously been studied in the course.

It was then apparent that a full overview of matrices and vectors was required in order to become proficient in this area and to successfully code these operations in VHDL.

2.1.2 Vector addition and subtraction

Adding two vectors is an extremely simple process, one basically adds the elements in the same position in the vector.

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} + \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix} = \begin{bmatrix} X_1 + Y_1 \\ X_2 + Y_2 \\ X_3 + Y_3 \\ X_4 + Y_4 \end{bmatrix} \tag{2.2}$$

It is important to note that vector addition is only feasible when vectors of a similar size are being dealt with. Vector addition is also reflexive which means x + y = y + x. Vector subtraction is simply vector addition of the negative.

2.1.3 Matrix addition and subtraction

$$\begin{bmatrix} 2 & 4 & 6 \\ 1 & 8 & 3 \\ 4 & 2 & 7 \end{bmatrix} + \begin{bmatrix} 3 & 9 & 1 \\ 2 & 3 & 4 \\ 7 & 5 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 13 & 7 \\ 3 & 11 & 7 \\ 11 & 7 & 8 \end{bmatrix} \tag{2.3}$$

The procedure for adding and subtracting matrices is very simple. First it must be made sure that the dimensions of the matrices are the same. For this reason I stuck to fixed size n by n matrices and discarded using m by n matrices. In the eventuality that I would have to add an m by n matrix to an n by n matrix this would have caused problems so I rejected this straight off. Dimensions refer to the size of the matrix. You can add or subtract matrices that have the same dimensions, i.e three by three matrices or five by five matrices but one can not add or subtract those which contain different dimensions such as a three by two matrix multiplied by a ten by six. The resulting matrix will have dimensions the same size as the input matrices. When adding matrices each element of matrix one is simply added to the corresponding element of matrix 2 in the same cell.

$$\begin{bmatrix} 4 & 2 & 3 \\ 9 & 6 & 7 \\ 12 & 2 & 1 \end{bmatrix} - \begin{bmatrix} 2 & 3 & 6 \\ 4 & 1 & 5 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 2 & -1 & -3 \\ 5 & 4 & 2 \\ 5 & -6 & -8 \end{bmatrix} \tag{2.4}$$

When subtracting matrices, again, the same rule is applied and the elements that correspond to each other in the same cells are subtracted. In the above matrix you would subtract the value in cell 1A in the second matrix from the value in cell 1A in the first matrix to obtain the result, 1A in the resulting matrix.

2.1.4 Matrix Multiplication

When multiplying matrices it is essential that the dimension of the column in the first matrix is the same as the dimension of the row in the second matrix. If this is not the case successful multiplication can not be achieved.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix} \tag{2.5}$$

Due to the fact that the dimensions of my matrices were a fixed, n by n size this would always be the case. To obtain the resulting matrix, you must multiply all the numbers in the first row of the first matrix by all the values in the first column and all that follow of the second matrix. After this you multiply the numbers in the second row of the first matrix by all the values in the first column, and any others that follow of the second matrix. Once the multiplying is complete, the products are added together to produce the resulting matrix.

$$\begin{aligned} (1 \times 7) + (2 \times 8) + (3 \times 9) &= 50 \\ (1 \times 10) + (2 \times 11) + (3 \times 12) &= 68 \\ (4 \times 7) + (5 \times 8) + (6 \times 9) &= 122 \\ (4 \times 10) + (5 \times 11) + (6 \times 12) &= 167 \end{aligned} \tag{2.6}$$

this gives the resulting two by two matrix

$$\begin{bmatrix} 50 & 68 \\ 122 & 167 \end{bmatrix} \tag{2.7}$$

Generally the product C of matrices A and B is defined as:

$$C_{ik} = A_{ij}B_{jk} \tag{2.8}$$

Where, j is summed over all possible values of i and k. In order for matrix multiplication to be defined, the dimensions of the matrices must satisfy

$$(n \times m)(m \times p) = (n \times p) \tag{2.9}$$

$$\begin{bmatrix} C_{11} & C_{12} & ..... & C_{1p} \\ C_{21} & C_{22} & ..... & C_{2p} \\ ..... & ..... & ..... & ..... \\ ..... & ..... & ..... & ..... \\ C_{n1} & C_{n2} & ..... & C_{np} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & ..... & A_{1p} \\ A_{21} & A_{22} & ..... & A_{2p} \\ ..... & ..... & ..... & ..... \\ ..... & ..... & ..... & ..... \\ A_{n1} & A_{n2} & ..... & A_{np} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & ..... & B_{1p} \\ B_{21} & B_{22} & ..... & B_{2p} \\ ..... & ..... & ..... & ..... \\ ..... & ..... & ..... & ..... \\ B_{n1} & B_{n2} & ..... & B_{np} \end{bmatrix} \tag{2.10}$$

The exact algorithm for multiplication of n by n matrices is as follows:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} + ... + a_{1m}b_{m1} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} + ... + a_{1m}b_{m2} \\ c_{1p} &= a_{11}b_{1p} + a_{12}b_{2p} + ... + a_{1m}b_{mp} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} + ... + a_{2m}b_{m1} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} + ... + a_{2m}b_{m2} \\ c_{2p} &= a_{21}b_{1p} + a_{22}b_{2p} + ... + a_{2m}b_{mp} \\ c_{n1} &= a_{n1}b_{11} + a_{n2}b_{21} + ... + a_{nm}b_{m1} \\ c_{n2} &= a_{n1}b_{12} + a_{n2}b_{22} + ... + a_{nm}b_{m2} \\ c_{np} &= a_{n1}b_{1p} + a_{n2}b_{2p} + ... + a_{nm}b_{mp} \end{aligned} \tag{2.11}$$

2.1.5 Matrix-Vector Multiplication

The definition of multiplication between a matrix A and a vector B can only be obtained for the case when the number of columns in

A equals the number of rows in B. The general formula for matrix-vector multiplication is:

$$\begin{bmatrix} a_{11} & a_{12} & ..... & a_{1n} \\ a_{21} & a_{22} & ..... & a_{2n} \\ ..... & ..... & ..... & ..... \\ ..... & ..... & ..... & ..... \\ a_{m1} & a_{m2} & ..... & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ ... \\ ... \\ x_n \end{bmatrix} \tag{2.12}$$

$$= \begin{matrix} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \\ \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \end{matrix} \tag{2.13}$$

The process of matrix-vector multiplication is one which takes the dot-product of B with each of the rows of A (hence the reason why the number of columns in A has to be equal to the number of components in vector B). The first element of the matrix-vector product is the dot-product of B with the first row of A.

For example, if A is the matrix

$$\begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \tag{2.14}$$

And B is the vector, (2, 1, 0) then

$$\begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \end{bmatrix} \tag{2.15}$$

When multiplying a vector by a scalar value, each element of the vector must be multiplied by this value. Similarly when multiplying a matrix by a scalar the same procedure is adhered to and the resulting matrix is the product of each element of the matrix and the scalar.

2.1.6 Matrix Transpose

To transpose a matrix all that is necessary is for the columns to be converted to rows and vice versa. The result is the object obtained by replacing all objects of Aij with Aji. The matrix transpose, most commonly denoted AT, is the matrix obtained by exchanging A's rows and columns and satisfies the identity:

$$(A^T)^{-1} = (A^{-1})^T \tag{2.16}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} => \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \tag{2.17}$$

When dealing with an n by m matrix the transposed result matrix will be an m by n matrix, a matrix is symmetric if AT = A and is antisymmetric if AT = -A. For the purpose of my project this is useless trivia. The transpose of a matrix is the matrix product of the transposed matrix in reverted order,

$$(AB)^T = A^T B^T \tag{2.18}$$

2.1.7 Matrix Inversion

There were a couple of different matrix inversion methods that I examined before making a decision as to which one would be most feasible to implement in VHDL and which one would be most suitable in terms of the algorithm.

Matrix Inversion using Gaussian Elimination

For moderate and large matrices the best approach for inversion is the use of Gaussian elimination more commonly known as Gauss

Jordan elimination. When using Gaussian elimination some matrix A is augmented with an identity matrix of the same size. For example matrix A is:

$$\begin{pmatrix} 1 & -.6 & 0 & 0 \\ 0 & 1 & -.5 & 0 \\ 0 & 0 & 1 & -.4 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.19}$$

augmented with the identity matrix,

$$\begin{pmatrix} 1 & -.6 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & -.5 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -.4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.20}$$

A row is picked where the first element is non zero and this is named the pivot row. This row is then added to the remaining rows, scaling it by some constant such that the first element of these selected rows deduces to zero. Let us say that the pivot row starts with the value Z and the soon to be processed row is headed by X, the pivot row would be scaled by -X/Z to obtain an outcome of zero for the first element of X when the rows are added. Each time this scale and add operation is performed the same procedure is carried out on the identity matrix. In the example given, the pivot row of the identity matrix would be taken, multiplied by -X/Z and added to the corresponding row in the identity matrix. This should result in the first column of the matrix being all zero with the exception of the pivot row. From here a row is chosen where the second element in that row is non zero. The process discussed is then repeated. This is continually repeated for all columns which should produce a matrix where only elements of the diagonals are non zero. In the event of an entire row being zero this will mean that the original matrix is singular and can not be inverted. Each row is scaled by 1/A, where A represents the non zero value in the matrix. The inverse of the

matrix has now been obtained. Each step of the procedure below:

$$\begin{pmatrix} 1 & -.6 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & -.5 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -.4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.21}$$

$$R_1 = R_1 + .6R_2 \begin{pmatrix} 1 & 0 & -.3 & 0 & 1 & .6 & 0 & 0 \\ 0 & 1 & -.5 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -.4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.22}$$

$$R_1 = R_1 + .3R_3 \begin{pmatrix} 1 & 0 & 0 & -.12 & 1 & .6 & .3 & 0 \\ 0 & 1 & -.5 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -.4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.23}$$

$$R_2 = R_2 + .5R_3 \begin{pmatrix} 1 & 0 & 0 & -.12 & 1 & .6 & .3 & 0 \\ 0 & 1 & 0 & -.2 & 0 & 1 & .5 & 0 \\ 0 & 0 & 1 & -.4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.24}$$

$$R_1 = R_1 + .12R_4 \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & .6 & .3 & .12 \\ 0 & 1 & 0 & -.2 & 0 & 1 & .5 & 0 \\ 0 & 0 & 1 & -.4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.25}$$

$$R_2 = R_2 + .2R_4 \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & .6 & .3 & .12 \\ 0 & 1 & 0 & 0 & 0 & 1 & .5 & .2 \\ 0 & 0 & 1 & -.4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.26}$$

$$R_3 = R_3 + .4R_2 \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & .6 & .3 & .12 \\ 0 & 1 & 0 & 0 & 0 & 1 & .5 & .2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & .4 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.27}$$

hence the inverse

$$\begin{pmatrix} 1 & .6 & .3 & .12 \\ 0 & 1 & .5 & .2 \\ 0 & 0 & 1 & .4 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.28}$$

Matrix Inversion using the Adjoint Matrix Formula

For smaller matrices, those of dimensions two, three and four, a less complex approach to the problem is using the adjoint matrix formula. When dealing with any non singular matrix, let us call it A, the inverse of A can be successfully computed by dividing the adjoint of A by the overall determinant of A. The procedure for calculating the adjoint of A is quite simple and poses minimum difficulty for the mathematician. The first step is to find the matrix of minors for A. This is done by effectively eliminating the ith and jth row and column of the matrix and computing the determinants of the resulting two by two matrix at each step. Each of these corresponding results will produce the adjoint A. To find the cofactors of A a sign change is applied to selected elements of the matrix, this is discussed in more detail below. To find the adjoint, the matrix of cofactors is then transposed. After finding the adjoint each value in this new matrix is divided by the determinant of the original matrix. The resulting matrix is the inverted matrix.

Matrix Determinants

To find the determinant of a two by two matrix, each element is aligned to a matrix with elements A, B, C and D respectively. The algorithm for the determinant is AD - BC, so the resulting determinant will be a single value.

In attempting to find the determinant of a three by three matrix, the procedure is simply extended. The overall determinant can be

achieved using cofactor expansion along a chosen row. To obtain cofactors as stated above, the ith row and jth column are covered and the resulting two by two determinant is deduced. This is done for all elements, A11 to A33, and is followed by a sign change on selected elements as shown in the diagram below. The resulting determinants C11 to C33:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2.29}$$

$$C_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} C_{12} = \begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} C_{13} = \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \tag{2.30}$$

$$C_{21} = \begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{pmatrix} C_{22} = \begin{pmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{pmatrix} C_{23} = \begin{pmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{pmatrix} \tag{2.31}$$

$$C_31 = \begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{pmatrix} C_32 = \begin{pmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{pmatrix} C_33 = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{2.32}$$

the sign change applied to these results is:

$$\begin{pmatrix} + & - & + \\ - & + & - \\ + & - & + \end{pmatrix} \tag{2.33}$$

Cofactor expansion can be achieved by using any row in the matrix and multiplying the determinants in that row by the elements in the corresponding original matrix. The formula for cofactor expansion along the first, second and third rows of the matrix is determined by substituting values into the equations below:

$$det(A) = |A| = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = a_{11} \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{12} \begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} + a_{13} \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \tag{2.34}$$

Chapter 3

Third Chapter

3.1 Algorithms for Adding, Subtracting and Multiplication

After the preliminary step of refreshing the matrix mathematics course most of which had been covered previously in first year maths, it was now inevitable that these formulae be converted to VHDL code. After some thought I came to the realization that writing code for adding, subtracting and multiplying logic vectors was what was required. These entities would then be used when attempting to carry out matrix addition and multiplication. With some deliberation I finally came up with an algorithm that I felt would be feasible for implementation in my project but this was only after I had toyed with various ways to do each. One by one, for some reason or another I felt I had to discard these procedures in pursuit of code that would cater for all eventualities. Below is a run through of some of the algorithms I experimented with for both addition and subtraction and then for multiplication of logic vectors. I have also accounted for the reasons why I then found it necessary to pursue alternatives. Unfortunately for a lot of the time it was only after the code had been written that I found it to be inappropriate for certain cases or in some small way flawed.

3.2 Addition and Subtraction of Std logic vectors

Due to the fact that the project specification stated that the equations would be evaluated using FPGAs , I felt that an algorithm for addition that would be compatible with FPGAs was necessary. This would mean it would be easier to use the board when the time came in the closing stages of the application. For this reason I opted for a design that would use a series of full-adders to add vectors. These full-adders would be constructed from two half-adders as is the usual implementation style. I rehashed a section of the code from 'HDL Chip Design' which I have referenced in my project and bibliography.

```
entity HALF_ADDER is
 port(A, B: in std_logic; Sum, C_out: out
std_logic);
     end entity;

architecture Logic of HALF_ADDER is

begin
   Sum <= A xor B;
   C_out <= A and B;
end architecture Logic;
```

3.3 Six Bit Adder

The sixbit addsub2bit as the name suggests either adds or subtracts a 2-bit value to or from a 6-bit value. This logical structure is basically modelled by a series of full adders. A full-adder in turn comprises two half adders and an OR gate.

```
begin
 HA1: HALF_ADDER port map (A => A, B => B, Sum => AplusB, C_out =>
 CoutHA1);
 HA2: HALF_ADDER port map(A => AplusB, B => Cin, Sum => Sum, C_out
 => C_outHA2);
 C_out <= CoutHA1 or CoutHA2;
```

A single bit half adder is modelled using a single XOR logical operator and a single AND logical operator. For the adder/subtractor circuit , the entity, is designed by instantiating six of these full adders with a ripple carry chain from one full adder to the next. Input B, the addend uses two XOR functions in order to create the ones complement; they are XORed with the two least significant bits of input A, the augend. The twos complement which is required for subtraction, is designed by connecting input SubAddBar to the carry in of the first, least Significant bit.

Extra logic is modelled to force the output to binary 111111 in the event of overflow caused by addition. Similarly if underflow occurs from subtraction the output will result in binary 000000. An overflow will result in the case where the carry out from the most significant bit full adder, i.e carryOut[5], is at logic 1. An overflow will be created when subtracting and carryOut[5] is at 0. This design only requires minimum changes in order to remodel it with different bit widths. VHDL constants specify the bus width of inputs A and B which will later be referenced in the entities body. The design of the model is such that it uses generate statements to instantiate the single bit adders in a way that only the constants widthA and widthB require changes. This will then change the input and output bit widths.

Problems with the Six Bit Adder

This adder seemed very practical for my application as it allowed for the input sizes to be changed without difficulty. As can be seen from a segment of the code and indeed is implied in the name, this

adder adds a six bit logic vector to a two bit logic vector. In the case of subtract when the input subAddBar is asserted it subtracts a two bit logic vector from a six bit logic vector. The output is six bits which means it is the same size as the first input. This is fine for cases where the second input is small in comparison but this was not always going to hold true for my addition. From the way that the code is designed it was almost impossible for me to change the size of the output to be bigger than the first input, although it was easy enough to change the size of all inputs and outputs provided the output always remained the same size as the first input. To change this would mean completely re writing the code. This was a definite option for me. Another problem that arose was that this algorithm failed to facilitate for the possibility of negatives. In the event of the second number being greater than the first number, with subAddBar remaining low then the output produced was zero. Similarly for subtract, this problem occurred. I spent time trying to adjust the code so that not only would the output be made greater but that the most significant bit of each value, input and output and indeed all internal signal alike would be reserved representing the sign of the number. After failing on occasion, I decided to use this entity in another entity, one in which I could allow the inputs and outputs to be any size desired .I would just call this entity on the part of the vector I wanted to add. In this way I could use the MSB as the sign of the number and still reserve the implementation previously explored. This would take positives and negatives into account. I felt confident that this would work and quickly moved onto designing multipliers without giving it any more time or thought.

```
entity sixBitAdder
port(Sub_AddBar: in std_logic;
   A: in std_logic_vector(5 downto 0);
   B: in std_logic_vector(1 downto 0);
   Y: out std_logic_vector(5 downto 0));
end entity sixBitAdder;
```

It was only later when designing the matrix multiplier that I once again encountered setbacks. This was mainly as a result of the subAddBar input which needed to be set for each addition or subtraction that took place on an individual logic vector. For matrix addition it was not too problematic as it could just be left at zero the entire time. However with matrix multiplication it had to be firstly set to one for subtracting determinants and then it had to be reset to zero for adding the partial results. This meant that I would need two input subAddBar signals. This seemed extremely messy and I felt that it would be best to do away with this completely and to just have two different entities that would either add or subtract and that would be called separately as needed. When realising that the problem of overflow and the size of the output had not been completely overcome as was previously believed, I decided after some consternation to rewrite this entity doing away with the 'series of full-adder approach' yet drawing on the knowledge previously acquired with respect to integer values as opposed to natural numbers.

In the end this adder was actually used in the very last part of the inverse of a matrix implementation, which will be discussed later. However in reference to matrix multiplication and matrix addition/subtraction it was never used.

3.4 Alternative Adder/Subtractor

This adder was a lot less complex in that it used the arithmetic library defined in VHDL to add and subtract logic vectors. The MSB of the inputs and outputs were reserved to represent the sign of the number, thus indicating whether it was positive or negative. The entity entailed a process that took in the inputs and contained a series of if statements that tested the MSB of the inputs to ascertain their sign and to test which of the values was the greater. Depending on the outcome it would then add or subtract these inputs and set the sign of the output accordingly, this again being its MSB. The output was greater than the inputs which meant that on occasion when the sign of the output was negative larger numbers were being

dealt with in the test bench. The code was changed slightly to deal with subtraction which at most meant readjusting the if statements in the process again for each different case. This algorithm although pretty basic worked for all possible cases.

3.4.1 Addition/Subtraction for Fixed Point Numbers

Obviously my program would have to deal with something more than whole numbers, positive and negative. After exploring the floating point number system for some time and due to the fact that time itself was pushing on I opted to represent all numbers as fixed point as opposed to floating point. This understandably made my job somewhat easier. A fixed point number is basically just a value with a fractional and integer part. As opposed to floating point numbers, fixed point numbers allow you to fix the position of the decimal point and then carry out logical operations. Generally, fixed point representation has "int" bits to the left of the decimal point and has "fract" bits for the fractional part to the right of the decimal point. When "fract" = '0' the number is treated as an integer. After researching the fixed point number system with regard hardware arithmetic functions it became apparent that the logic vectors would be indicative of completely different values when converted to decimal and would be interpreted accordingly. In retrospect this should have been apparent straight away given that negatives in VHDL are also different values when converted to the decimal system. Alas it took time before finally getting to grips with this notion, which in the end proved to be quite simplistic. Working off the premise that a four bit std logic vector, from left to right, would have two bits that entailed the integer part, then the decimal point, and the remaining two bits would be representative of the fractional part of the number. For example with a vector such as 01.01, in decimal form this would be interpreted as 1.25. The MSB would still be maintained to hold the sign of the number.

Fraction Adder/Subtractor

First off was the coding of a fraction adder which added two fractions as normal numbers. The numbers are merely interpreted as fractions and the second most significant bit of the output becomes a one in the event of overflow. This will be representative of a one before the decimal point, i.e an integer. The most significant bit of all numbers is used for the sign and the output is obviously one bit bigger than the inputs to allow for the possibility of an integer. The same holds true for subtraction.

3.4.2 Fixed Point Adder/Subtractor

Much the same idea was used when coding the fixed point adder. The integer and fraction parts of the logic vector were separated. The two fractions were added and if overflow occurred a one was added to the integer parts which were in turn added. If no overflow took place then it was not necessary to update the output of the integer parts. When dealing with subtraction the procedure was changed for the possibility that the integer part of the first number was smaller than the integer part of the second input whilst the fractional part of the first number was greater. This would mean that the first input was greater and that numbers needed to be subtracted as normal and then the decimal point placed in the correct position.

3.5 Algorithms for Multiplication

The first two algorithms which I attempted to use were sequential multipliers also compliments of 'HDL Chip Design'. First was the shift and add algorithm and second was booths multiplier. Booths algorithm was specifically designed to speed up sequential multiplication operations. Sequential multipliers and dividers are often implemented because of the substantial savings in chip area. Combi-

national logic multipliers are faster, but are significantly larger than their sequential counterparts for input bit widths in excess of three. The area of the combinational circuit increases with significance as the input and output bits grow. Contrasting this, sequential circuits are much smaller but will take a set number of clock cycles in which to carry out an operation.

3.5.1 Shift And Add Algorithm

A process of successive shift and add operations can be used to achieve multiplication when using binary numbers. This process examines each successive bit of the multiplier in turn, starting with the least significant bit and proceeding left to the most significant bit. When examined, if the multiplier bit is one, the multiplicand is copied down. If the multiplier bit is zero, zeros are copied down. These numbers copied down in successive lines are constantly shifted one position to the left from the previous number. When all bits have been inspected, the shift and adds are complete so the numbers are added and the sum provides the product.

When multiplying two signed numbers together the code is modified to cope with the sign bits. The sign of the product is determined from both the sign of multiplicand and multiplier . If they are alike then the sign will be positive, and if unalike the sign negative.

DECIMAL	BINARY	COMMENTS
39	0000010111	multiplicand
49	0000011001	multiplier
39	0000010111	partial product 1
0	0000000000	partial product 2
0	0000000000	partial product 2
0	0000000000	partial product 3
368	0101110000	partial product 4
736	1011100000	partial product 5
1911	1000111111	product = sum of partial products

3.5.2 Booths Multiplication Algorithm

Booths algorithm, like all multiplication schemes requires the examination of the multiplier bits proceeded by the shifting of all partial products. Booths algorithm is primarily intended for synchronous logic implementation of a multiplier circuit and works using all integers, negative and positive. It treats negatives and positives as one and is most ideally suited for the multiplication of signed twos complement numbers. Booths algorithm works on two principles. The first is that strings of successive zeros will have no effect on the product and thus require no addition, but just that the partial product be shifted as necessary. The second principle is that the string of successive ones that comprise the multiplier can be defined by 2 up + 1 - 2 low where 'up' is the upper weighted bit and 'lo' is representative of the lower weighted bit. The rules below are applied when using booths algorithm. The multiplicand is subtracted from the partial product upon finding the first one in a string of ones in the multiplier. The multiplicand is added to the partial product upon entering the first zero provided that no previous ones in a string of zeros in the multiplier were encountered. The partial product never changes provided the bit is the same as the previous multiplier bit. This algorithm works equally for positives and negatives as a negative multiplier fills the MSBs with a string of 1s and the last operation will be a subtraction of the appropriate weight.

9876543210	bit weighting	number
0000010111	multiplicand	(-9)
0000010011	multiplier	(-13)

0000001001	1st multiplier bit 1 - subtract(add twos complement)
0000000000	2nd multiplier bit also 1 - no change so no add/subtract
1111011100	3rd multiplier bit changes to 0 so add. note sign extension
0000000000	4th multiplier bit also 0 - no change so add/subtract
0010010000	5th multiplier bit changes to 1 so subtract(add 2s complement)
0001110101	product(+117)

3.5.3 Implementation of Sequential Shift and Add Multiplier

This multiplier was modelled on the algorithm previously examined. It also entails a sign bit. The products sign is the result of exclusive ORing the sign of the two inputs. The operation starts when load becomes a logic one and the registers are loaded with values. regA becomes zero, regB is loaded with the multiplicand minus its MSB, similarly regQ holds the multiplier without its sign bit. Ps, holds the sign when exclusive oring takes place between multiplicand and multiplier sign bits. SequenceCounter is the number of bits in the multiplier without the sign bit. Multiplicand and multiplier are now in registers B and Q respectively and operation is ready. As is indicative of this algorithm, a series of consecutive test, add and shift right operations occur. The signal AddShiftB is controlling add or shift operations. When this signal is one the sum of regA and regB become the partial product that is then stored in another reg, EA, a combination of flip-flop E and register A. The carry out from the adder in flip-flop E needs to be stored so that it can be used when it is time to generate the next partial product summation. EAQ is shifted right, the least significant bit of register A is shifted into the most significant bit of register Q. The bit from E is shifted into the MSB of register A, while logic 0 is then shifted into E. This shift causes one bit of the partial product in register A to be shifted into register Q, forcing the multiplier bits one position right. The rightmost flipflop in register Q, now Qn, will hold the bit of the multiplier which is next in line for examination. Once inspected, if Qn is a logic 1 an addition will result before the next shift occurs. However if Qn is a logic 0 no addition is necessary. A single multiplication will take from the width of the multiplier minus 1 multiplied by the width of the multiplicand minus 1 multiplied by two, clock cycles to finish. This is dependent on the logic zeros and ones in the multiplier. When a multiplication has taken place the one bit output, Done is set to 1. In this multiplier the input and output bit widths can be specified according to the users needs. This particular model is also instantiated such that the width of the output is not necessarily the same as the combined width of the sum of the two input magnitude widths.

3.5.4 Design for Booths Algorithm in VHDL

I also toyed with the possibility of using a sequential multiplier that implemented booths algorithm. This hardware structure contained distinct similarities with the structure implemented in the previously examined shift and add algorithm. The three fundamental differences, however, were firstly the need for an extra flipflop to be placed at the LSB of the multiplying register. This is mainly to facilitate double bit inspection of the multiplier. Secondly, this algorithm required the ability to subtract and lastly the flipflop that held the carry out from the adder in the standard shift and add approach is not needed as an add here can never cause any overflow. Once again when load is set the registers are loaded as before. After the two numbers to be multiplied are loaded into the appropriate registers the operation starts by examining two test bits of the multiplier in a simple case statement. In the event of these two bits being '10', the first 1 in a string of 1s has been found in the multiplicand. This then requires a subtraction of the multiplicand from the partial product. After this subtraction has taken place one or more shift operations take place until the multiplier bits stored in a register are equal to binary '01', t meaning that the position of the first '0' in the multiplier has been found. When this first '0' is found in the multiplier the multiplicand is added to the partial product, one or more arithmetic shift rights can occur until either the next '1' is located or the total number of shifts is equivalent to the length of the multiplier. After loading data again, if the two test multiplier bits are equal to '00', no addition or subtraction is required and the shifting begins, searching for the first '1' in the multiplier. No overflow will ever take place due to the fact that addition and subtraction operations alternate and the numbers being either added of subtracted have different signs. This condition will never yield overflow. Following addition or subtraction the arithmetic shift right occurs on the partial product, the multiplier and the flipflop. The arithmetic shift ensures that the most significant bit of the register of the partial products, before the shift, is copied into the most significant bit of the register, after the shift. This addition/subtraction process is repeated for the number of bits in the multiplier. The VHDL implementation uses a process which takes in variables to compute intermediate values of addition or subtraction

prior to the shifting process. A procedure is then used to perform the shifting process as it is identical for each of the case statements that examine the test bits.

3.6 Problems in Simulation - Type unsigned

The code for both these implementations synthesised perfectly. However when I tried to generate results for a testbench waveform I ran into difficulties. The code would not generate results although I was able to get a testbench module. Each time I tried to generate results or perform simulation for the testbench I got an error which read, "no feasible entries for subprogram write". After trying to find the reason for this apparent error I stumbled on a conclusion. All my inputs and outputs were of type unsigned. I soon came to the conclusion that VHDL can not simulate testbenches when the outputs are of type unsigned. I tried out other entities with all outputs of type unsigned and the same error occurred. However when I changed these to type std_logic_vector, testbenches were simulated correctly and results generated as expected. I tried to verify on numerous occasions that what I had established was in fact correct. I asked several other students doing VHDL and Verilog projects who didn't know if this was in fact true for all cases. I then asked past students who had previously completed a project in a hardware descriptive language. They both said that they thought that it was possible to generate a testbench using this type. There was little or no information in VHDL books that helped and information on websites was again minimal. So for a time I was undecided on this issue until I came across a question on a website from a student trying to simulate testbenches using all unsigned inputs and outputs and again encountering similar difficulties. The response to her query was in fact that it was not possible to simulated testbenches using this type but was possible for the remainder of types. To this point I still am uncertain as to whether or not this can be done.

```
library.IEEE; use IEEE.STD_Logic_1164.all; IEEE.Numeric_STD.all;
entity MULT_SEQUENT is
```

```
generic(WidthMultiplicand, WidthMultiplier, MaxCount: Natural);
   port(Clock, Reset, Load: in unsigned(WidthMultiplicand - 1 downto
   0); Multiplier: in unsigned(WidthMultiplier - 1 downto 0); Done:
   out std_logic; Product: out unsigned(WidthMultiplicand +
 WidthMultiplier - 2 downto 0))

 end entity MULT_SEQ;
```

3.7 Problems in Simulation - Types in VHDL

I disregarded the possibility of using unsigned outputs. Dealing firstly with the sequential shift and add algorithm, I changed the outputs to std_logic_vector, whilst leaving the inputs as unsigned. As I expected I got a type conversion error as I was foolishly trying to convert an unsigned input to an output of std_logic_vector. I then began researching type conversion in VHDL. I had previously thought that these two types convert automatically as they are closely linked types. Converting from one of these types to the other requires a type conversion in the form of casting or a type conversion function. VHDL does have automatic type conversions. It is possible to convert from std_logic_vector to std_ulogic_vector. Type casting can be used when two types have a common base, like unsigned and std_logic_vector.

FROM	TO	FUNCTION
Std_Logic_Vector	unsigned	unsigned(Std_Logic_Vector)
Unsigned	integer	to_integer(unsigned)
Integer	unsigned	to_unsigned(integer, no_of_bits)
Unsigned	Std_Logic_Vector	Std_Logic_Vector

I tried endlessly to convert one type to another in VHDL using predefined functions that should have made my job pretty simple but that didn't. I would be unable to say for definite if this was down to me or problems with these functions in general that posed such difficulties. I also attempted writing my own type conversion

functions but I encountered similar setbacks. I was getting errors in synthesis saying that these conversions were in compatible with the packages, namely numeric_std. However other functions in the program depended on this package and when I changed it, I got a whole new set of errors, although it seemed to fix this particular one. I finally admitted defeat and what I had learnt on type conversion and casting now seemed fruitless. A whole new approach to the problem was necessary.

3.7.1 Libraries in VHDL

Back to square one, I made the decision to totally exclude all inputs, outputs, signals and variables of type unsigned from my instantiation in the hope that issues connected with them would now be resolved. I hoped that I could simply replace them with std_logic_vectors and began wondering why I hadn't just done that in the first place and saved myself needless intricacies. Back on track and with all variables converted, I once more synthesised my instantiation. The package that I included was the previously defined package for this entity, 'numeric_ std'. Once synthesised, this generated more errors for me in the form of 'shift-right can not have operands in this form'. I was swamped with similar error messages for the resize operator, which was primarily used to adjust the size of registers to hold growing bits in variables and signals. After running a different example from a different source(not my own work), using the same package and inputs/outputs of type std_logic_vector, when using the shift_right operator I got exactly the same error message in synthesis. However, if changed to an slr or sll operator with the operands rearranged to suit I found that the entity synthesised. It also simultated a testbench and generated the appropriate results accordingly. Having finally made a breakthrough I tried this approach on my previous code, exactly as was in the simpler entity only to find it didn't have a similar effect and once again left me with the same errors. On examining my new-found situation, I found that for pre-defined operations, the package numeric_std is used to define types of signed and unsigned and all arithmetic, comparison and logical operations for these types. Similarly for the package std_logic_arith it defines

arithmetic and comparison operations for types signed and unsigned.

3.7.2 Packages in VHDL

Whilst experimenting with packages, I changed the package from numeric to arithmetic whilst keeping all types as before. This time, after synthesis, error messages were different. They took the form of "Undefined symbol 'resize'" and "Undefined symbol 'shift-right'". This would in fact verify the deduction that these functions are not available to std_logic_vectors when using std_logic_arith. However this may not be the case for std_logic_vectors and the package numeric_std as the error indicates that these operations are compatible with this particular package but that they should simply take a different form. In the event of both of these packages being excluded, it becomes apparent that these operations are dependant on numeric_std as once again an 'undefined symbol' message is returned.

Reviewing these Packages

From the synthworks website, I ascertained most of the basics needed when dealing with packages and types in VHDL. As stated in the previous subsection, numeric_std is used for types signed and unsigned and all logic, arithmetic and comparison operators carried out on them. Std_logic_arith defines types signed and unsigned and arithmetic and comparison operators can be defined within the scope of these types . Std_logic_unsigned defines arithmetic and comparison operators for std_logic_vectors. From the aforementioned source and a couple of other sources, it is widely recommended that one uses numeric_std for all new designs. This package also maintains compatibility with the package std_logic_unsigned. It is also advised that for numeric operations, you can use std_logic_arith with std_logic_unsigned, but that the packages std_logic_arith and numeric_std should never be used together. When examining this I found that if these two packages are used together, the one named first is the one used and the other package is ignored. Other than this it has no effect on the entity. The higher one simply takes

precedence and the later will be unaffecting. When using arithmetic operations availing of the package unsigned in conjunction with the arithmetic package worked best. It was useful to later find out that this was due to the unsigned packages influence as opposed to any other which primarily didn't register. When replacing or eliminating the arithmetic package errors were endured, which presumably meant that there was a reliance on this package but that the unsigned package was priority. These packages were later used together in pursuit of alternatives.

Influencing Nature of these Packages

Believing that a wealth of knowledge would have been obtained from the experiences with these packages, difficulties were attempted to be rectified once and for all. Needless to say this was easier said than done. Failure at every point when trying to adjust the shift and resize statements to work effectively for my design was met. To this point I do not know if the shift operator can be used with std_logic_vectors when using the numeric package. I only know that it is impossible with the arithmetic one. If it is not possible for logic vectors and only possible for type unsigned it is in fact useless if one can not generate testbench waveform outputs. However on either of these counts I would be open to correction and indeed any feedback or expertise that could shed light on my apparent inabilities. Regrettably not an awful lot of information was available to me. Any information that was on offer was sometimes ambiguous and often contradictory. Confusing would be an understatement. Anything else ascertained on this subject was through plain old trial and error. With more experience in the first place setbacks could have been avoided or once encountered eliminated. Examination of the need to use unsigneds in the first place brings its own conclusions and in hindsight this was perhaps completely unnecessary and logic vectors would have been a safer alternative. Lessons were learnt as frustration mounted and from here on in the easier option would inevitably be pursued as more confidence and ability with the constructs of the language was paramount to any progress.

3.8 Sequential Multiplier - The final Curtain

Rewriting the shift right operator to simply shift the vector one place to the right and concatenating it with a zero in its lsb and similarly coding a function for resize that placed all contents of a register into a register of a greater defined size made my entity synthesisable.

```
E_regA_regQ := shift_right(E & RegA & regQ), 1);
```

This is replaced by a small set of instructions. RegG was previously declared a signal one bit size larger.

```
E_regA_regQ := E & RegA & RegQ;
 RegG <= E_regA_regQ & '0';
```

Test bench waveform results were all undefined for certain inputs and for a time I believed it to be the readjusted code. However on closer examination I realized that it was because of the structure of the if statements and that because of the inputs I had given a relevant output could never be met because the program had no where to go. I further verified this by inserting fixed inputs into the testbench that I knew would result in output as it would comply with the first if statement and would not require any further testing in the program. The results were as expected. Due to the complexity and structure of the entity I was unable to change it. However this was the first time I had found a flaw in the code. Due to my frustration and ongoing struggles with this particular piece of code, I firmly decided to leave it where it was, in a book. I felt now that if I had better employed my time in writing entities of my own as opposed to using ones from source it would have been more rewarding. Code like this, on both counts, turned out to be worthless in terms of my project. I resigned myself for future exploits to design and code from scratch given these prior dilemmas and experiences.

3.9 Signed Combinational Multiplier - Alternative Approach

This combinational multiplier is also modelled on the shift and add algorithm and is the one I used in my implementation. Once more it contains an exclusive OR of the input sign bits when generating the output sign. The models structure is based on parallel adders. Partial products are computed according to the algorithm, zero if the associated bit in the multiplier is zero or a shifted version in the event of a one. The partial products are obtained using conditional signal assignments and don't infer logic but merely specify how the shifted multiplier input is connected with the adders. It is much less complex than the sequential multiplier but was easily implemented and adjusted to deal with fixed point numbers and integers alike. The input bit width can be changed at will, however all internal signals then have to be modified accordingly. This can be achieved without too much effort or indeed thought. As stated fixed point numbers were taken into account by a modification of the design that once again separates out the integer and fractional parts of the number, dealing with them separately and outputting an integer and fraction when the set bit is asserted. When the set bit is a zero integer multiplication simply takes place incurring a single integer output.

Chapter 4

Fourth Chapter

4.1 Dealing with two dimensional arrays - matrices in VHDL

This chapter entails examining two dimensional arrays in VHDL and finding the most feasible approach to matrix implementation.

4.2 Overview of Arrays

Like in most software object oriented languages, an array in vhdl is an indexed collection of objects all of the same type. Arrays may be one dimensional or indeed multidimensional. In VHDL, an array type may be constrained, in which the bounds for the index are determined when the type is first defined. Naturally an unconstrained array would imply that the type is defined and the index bounds are established subsequently. This is exactly what happens.

```
type myWord is array(15 downto 0)of std_logic;
```

Multi-dimensional arrays can be declared in the following way:

```
type MY_MATRIX3X2 is array(1 to 3, 1 to 2) of natural;
```

An unconstrained array takes the form of:

```
type VECTOR_INT is array(natural range <>) of integer;
```

This symbol is generally assumed a 'place-holder', which is filled in when the array is used for the first time. There are two predefined types in VHDL, both of which are unconstrained. They are:

```
TYPE string is array (positive range<>) of character;
```

```
TYPE bit-vector is array(natural range<>) of  bit;
```

Any element of an array can be reference by indexing to the position of that particular element in an array. Supposing that X and Y are arrays of dimension one and two respectively, then X(1) will refer to the first element in the X array and Y(3,2) will reference the element in the third row and second column of X. When writing a literal value of an array type, it is necessary to use an array aggregate, which is a list of the values of the elements. If an array is declared in the form of:

```
TYPE myArray is array(1 to 5) of character;
```

and it is necessary to write a value of this type containing the elements 'D', 'A', 'N', 'C', 'E'. It is possible to write an aggregate with positional association as below: ('D','A', 'N', 'C', 'E')

In VHDL, a multidimensional array is implemented as an array of arrays, and matrices fit the multidimensional array specification perfectly.

4.2.1 Addition of Arrays - Attempt One

The first approach to adding matrices using the previously defined adder as discussed in section 3.2.2 was to define an array outside the actual entity in a package. This specified two constrained arrays, one that would be used for the input matrices and the other for the resulting matrix to be outputted. The reason I needed two different matrices was because although the dimensions of both arrays would be similar, the logic vector for the output matrix would be bigger than the logic vectors for the input elements of the input matrices. I had little experience with packages until I happened upon an example of one used for holding one dimensional matrices. It seemed an easily understandable, straightforward approach to what I was trying to achieve. Packages can be created similar to normal VHDL modules, and modules that they depend on will in turn be linked to them in the hierarchical manner in which all entities relating to one another are depicted in Xilinx. Single types, arrays and records can all be defined using packages. Defining two arrays in a package is as follows:

```
PACKAGE array_example is
    TYPE arrayOne IS ARRAY(2 downto 0, 2 downto 0) OF std_logic_vector(8 downt
    TYPE arrayTwo IS ARRAY(2 downto 0, 2 downto 0) OF std_logic_vector(9 downt
END array_example;
```

When using this in another entity it must be included with the other packages normally included so that it can be accessed in this entity

```
USE work.array_example.All;
```

In the input port, as opposed to declaring many inputs to fill up both arrays as I had expected to do, it was now possible to input the arrays of type arrayOne and arrayTwo instead.

```
PORT(matrixA: IN arrayOne; matrixB: IN arrayOne; resultMatrix: out
arrayTwo);
```

In the body of the entity was a process containing the two input matrices which would be subsequently indexed through using a for loop and the corresponding elements would be added together using portmaps. The output of the adder would then carry the results to each position in the resulting matrix.

```
MA1: adder port map(adder1 => matrixA(i), adder2 => matrixB(i),
adderOut => resultMatrix(i));
```

After a little work the code synthesised and the testbench waveform was generated. However when I tried to input numbers into the testbench that contained more than one bit, a warning was returned which read, "string error" "bit width is more than 1"

On entering a 1 or a 0 into the testbench no warning was returned, however anything bigger caused the above warning to be outputted. This was of course little use to me as the problem could not be located as all of the logic vectors were nine and ten bits wide respectively. Once again, having hit a brick wall, it was back to the drawing board. Rectifying this error by checking it out on the internet was impossible and finding a means of resolution was difficult. However in a bid to prove that this was not to do with my code, an example from a VHDL textbook , which my code was closely modelled on was synthesised. This code used a package to declare two one dimensional arrays of type bit and with 32 and 8 bit array elements respectively. A function was performed on these arrays which is irrelevant to the problem description. When the bits were changed to anything more than a single bit, the error once more occurred in the testbench. A couple of other examples gave a similar error and I was unable to sort it out. Disillusioned, I disregarded the possibility of using packages to define arrays and set about to find an alternative way to do this.

4.2.2 Addition of Arrays - Attempt Two

After this it was decided that I would declare all inputs and outputs of the array in the input port. This meant I would now be using

arrays of fixed size and I decided on firstly using three by three matrices until I was in a better position to be changing the size when required. This was very easy to do. After all inputs were declared, two array types were defined in the architecture of the entity. Then three matrices of these types were declared as signals.

```
TYPE arrayOne IS ARRAY(2 downto 0, 2 downto 0) OF
std_logic_vector(8 downto 0);
TYPE arrayTwo IS ARRAY(2 downto 0, 2
downto 0) OF std_logic_vector(9 downto 0);

Signal matrixA: arrayOne; Signal matrixB: arrayOne; Signal
ResultMatrix: arrayTwo;
```

Each of the inputs were placed into each element of the arrays, these elements were then added together and results were placed in the resulting array, which were in turn outputted. Although a very simple approach, this was one that worked.

4.2.3 Addition of Arrays - Final Attempt

After a meeting with my supervisor, I was strongly advised to implement these arrays in VHDL using blockram. Previously unknown to me, once again research was inevitable. The Spartan 2E FPGAs have blocks of bits, 4096 bits per block, called block ram that can be configured as single port or dual port ram. Block ram can be used as data ram for a processor or if broken into smaller pieces can be used as register files. To my understanding, in dual port ram the inputs are written to memory using the address lines and the outputs are subsequently read. Depending on the size of the board being used the input bit widths have to be adjusted accordingly. Deciding not to make this factor a priority, I firstly set the data inputs to 20 bits. This meant that data for both the first element of matrix A and the first element of matrix B would come in on the first data line. The data outputs were set to 10 bits. This was done as below:

```
Generic(data_width1   : natural := 20;
    Addr_width1: natural := 18;
    Data_width2: natural : = 10;
    Addr_width2 : natural := 10);
```

The same amount of address lines as data input lines were required. An array was created to store the data elements. This took the form of:

```
Type mem_type is array(2 downto 0, 2 downto 0) of
std_logic_vector(data_width -1 downto 0);
```

On realising that the inputs and outputs were not the same size, difficulties were envisioned. Attempting dual ported ram would not be feasible unless two different arrays were created of different bit sizes or if inputs were made one bit bigger. In reality neither of these options would work due to the fact that the same memory array would be needed when writing and reading values. Due to this I opted for the easier option of using single port block ram where the inputs would be written to memory and outputted from here. This was done using a process, on the rising edge of the clock with the write signal asserted, all data inputs were written to memory. From here, exactly the same procedure as was described above to add the matrices using portmapping was carried out. The code was synthesised and testbench outputs verified that the implementation worked as expected. Some time was allocated to trying to get the design to work for dual port ram. However with time pushing on and so much other work needing to be done, I resigned myself to what I had and made a mental note to return to this in the event of me having any spare time towards the end. I hoped rather than believed this would be the case.

4.3 Matrix Multiplication using Block Ram

The next logical step in the project was to similarly use block ram in an implementation of matrix multiplication. Block ram was used in

exactly the same way as above to allocate memory for storage of each of the data inputs. The algorithm for matrix multiplication required the use of the multiplier entity to multiply the data inputs and then the adder entity to add these products. Internal signals were used to hold the results of the partial products after multiplication was achieved. These signals had to be continually changed to facilitate the growing bit widths, as data inputs were multiplied added to another product of multiplication and this in turn added to another product. This also meant that the bit widths of the adder had to be changed.

As the signals grew in size to store products it was essential that the sign was handled. If a zero was concatenated with a number, in order to make it bigger, and it was positive this would be fine. However in the event of it being negative and the msb being representative of its sign, this would seriously damage the process. For this a process was created with a series of if statements which tested whether the msb was a 1 or a zero. In the case of a one, this one was shifted to the most significant bit and a zero was put into this position, the rest of the logic vector remaining as it was. This meant that it would be still the same number and represent that number whether it was plus or minus. When generating expected outputs in the test bench it made it a little difficult to calculate whether the design worked as required as the growing bit widths meant that for any negative numbers, the output number was extremely high. However after checking all these results, clearly the design worked as expected.

Problems with Block Ram

One of the only difficulties encountered when using block ram was the size of the data inputs. When declared at the beginning of the entity as:

```
Generic(data_width1: natural:= 20);
```

This being used to effectively hold two data inputs it appeared to be fine to do this. However in synthesis the process ran forever and

never came to the point where it returned the output message 'synthesis completed'. On one occasion, when left, the process had not finished after forty five minutes so the process was then terminated as it looked as though it would never synthesise. If the data_width was subsequently changed to ten and then synthesised, obviously with the size of everything else changed too, then the process was very slow but it did eventually synthesise. Beyond this point if the bit width was changed the process became slower and slower to the point where it was uncertain whether or not it would ever finish. Obviously changing the data_width to ten was of little use to me as it was too small to hold two inputs. Afterwards, not knowing the cause of this and not really knowing what to do, I just allowed all data inputs to be independent of one another and were declared separately in the entity. This was indeed contradictory to one data input holding two matrix values but on feeling that it couldn't be helped this was the next best thing.

Chapter 5

Fifth Chapter

5.1 Simplistic Transpose Design

The most non complex of all entities, the module basically took nine inputs and outputted them in order which is intrinsic to the transpose algorithm. Placing the elements into different array positions and then outputting these results was all that was needed in this implementation. The size of the matrix could be changed at will as is understandable from the simplistic nature of matrix transpose algorithm. Sizes were adjusted in accordance with what would be needed for successful implementation of this entity in the design for a matrix inverter. The time taken to complete this design was minimal in comparison with previous entities. Synthesis was performed and simulation results proved that this specification worked as expected.

5.2 Inverting a Matrix - Design for an Adjoint Matrix

As the size of the previous matrices were all three by three in dimension, the adjoint matrix presented itself as a very real possibility for successful matrix inversion. Having been advised by my supervisor to implement an inverter using Gaussian elimination, I researched this once again in great detail and examined the C++ code on matrix inversion that was presented to me. Naturally, Gaussian elimination would have been the most effective and indeed elegant option to pursue as it deals with matrices of any dimension. The adjoint matrix formula is limited to a specific size and is inefficient for matrix inversion when the dimensions grow in excess of four by four. However due to the apparent complexities that would be involved in carrying out Gauss Jordan elimination in VHDL I opted for the later and time permitting would perhaps review my choice. Once again I was dubious. Feeling that I was ill equipped for concatenation of an identity matrix with the original matrix in the very first step evidently posed some concerns. This coupled with a series of complex tests and adds of individual rows, whilst trying to derive an identity matrix on the LHS, further confirmed my reservations. A more confident VHDL programmer may have had the means and ability to successfully deal with this. Regrettably I was not at this stage of expertise.

As is indicative of the adjoint matrix formula, cofactor expansion needed successful coding. First off was the task of computing the matrix of minors. This involved computing cofactors(determinants of the matrix for ith row and jth column as in chapter 1). Availing of the multiplier entity along with a subtractor (already designed) this was achieved quite easily. To compute the matrix of minors, a sign change is then applied to the second, fourth, sixth and eight elements of the newly formed matrix of cofactors. This sign change was computed by making all elements one bit larger whilst dealing with the msb as the sign of the number as in matrix multiplication. Subsequently, the second, fourth, sixth and eight elements were exclusive ored with a one, changing their signs. This had the same effect as multiplying them by minus one. This being achieved

the values were transposed and this completed the first stage , the matrix of minors.

Next logical step was to compute the determinant of the whole matrix. Cofactor expansion along the third row seemed appropriate. This step again was straightforward in that matrix multiplication and subtraction, coupled with addition of these results, was all that was necessary. Not very complex as all were previously coded entities. This concluded this step.

The final step was were any difficulty would lie. All the values in the matrix of minors needed to be divided by the overall determinant achieved through cofactor expansion. This inevitably required a fractional divisor, as the values in the matrix are always going to be much smaller than the determinant value.

5.2.1 Fraction Divisor in VHDL

The combinational divider was modelled closely on a 10 bit divide by 5 bit combinational logic divider from 'HDL chip design. However this divider had to be changed to a fraction divider for the purposes of my design implementation. With this divisor instead of using consecutive sequences of shift, compare and subtract operations, it is more appropriate to use consecutive sequences of shift and add a twos complement number. Adding a twos complement number is equivalent to subtraction, however its convenience lies in that fact that a single adder can perform both the compare and subtract operations. The carry out from this will provide an indication as to which of the inputs is the larger of the two.

The first compare is of the upper bits of the dividend this size always being the same as the size of the divisor. This is followed by a chain of add, compare and shift operations. The VHDL code used for this implementation is a series of signal assignments and if statements. As the number of bits in the dividend grows with the dividend remaining the same there will be a significant growth in the number of if statements.

SIGNAL NAME	BINARY VALUE	OPERATION
A (dividend)	0111000010 (450)	
B (divisor)	10001 (17)	
2's comp B	01111	
Overflow	0	
Compare1[5:0]	101011	A[8:4] + 2's comp B
Quotient[4]	1	
PartRem1[4:0]	01011	Compare1[4:0]
PartRem1_Abit[4:0]	10110	Bring down dividend bit 3
Compare2[5:0]	100101	Compare1_Abit + 2's comp B
Quotient[3]	1	
PartRem2[4:0]	00101	compare2[4:0]
PartRem2_Abit[4:0]	01010	Bring down dividend 2
Compare3[5:0]	011001	Compare2_Abit + 2's comp B
Quotient[2]	0	
PartRem3[4:0]	01010	Compare2_Abit
PartRem3_Abit[4:0]	10101	Bring down dividend bit 1
Compare4[5:0]	100100	Compare3_Abit + 2's comp B
Quotient[1]	1	
PartRem4[4:0]	00100	PartRem3_Abit - B
PartRem4_Abit[4:0]	01000	Bring down dividend bit 0
Compare5[5:0]	010111	Compare4_Abit + 2's comp B
Quotient[0]	0	
Quotient[4:0]	11010(26)	
Remainder[4:0]	01000(8)	

Changing the structure of this entity a little, a 20 bit divide by 10 bit divider took on the same form as the previous divider. When completed a 9 bit divide by 10 bit was attempted using the behaviour modelled in the 20 bit divide by 10 bit. The 9 bit input was multiplied by 100 and concatenated with zeros so it would fit the twenty bit wide stipulation. To multiply the number by 100 it was multiplied by 10 twice. This was achieved by shifting it to the left three times storing the result. The original was shift left once and subsequently added to the stored result. The procedure with this new result was repeated, effectively multiplying the original number by 100.

Because it was a fraction divisor there was a need for an extra bit which held the sign of the number. The input signs were tested in a process containing a series of if statements The output signs were then deduced accordingly. The output number was also concatenated with a 0 to represent the bit left of the decimal point. Testbench outputs verified its functionality.

5.3 Inverting a matrix

With this new divisor successfully implemented, matrix inversion seemed like just a step away. First the design was tested to see if the matrix of minors was working. This along with cofactor expansion for finding the matrix determinant was working. However when trying to divide by the determinant it became apparent that bit widths didn't match because the sign part of the fraction divisor had been overlooked. Having spent an awful lot of time on the divisor and encountering some difficulties, I felt that it would be more appropriate to change the bit width of the rest of the entity. With just a day to the demonstration, this seemed like an impossible task. After changing the entity and allowing the divisor to remain as was, I was in undated with a series of synthesis errors and was unable to figure out what they were. Disappointed and disillusioned, I decide to just demonstrate what I had done with the hope of perhaps getting it working after the demo. Needless to say once again this did not materialise, at large due to difficulties with an adequate divisor and ongoing commitments to other course material.Previously finding it difficult to locate appropriate material on this topic, did not help my cause.

Chapter 6

chapter six

6.1 Review of the project and personal thoughts

The initial task which was probably of most importance was background research. A lot of this proved to be informative and at times interesting. Without this key factor project progress would have been impossible and reference material, though at times sparse, enabled a form of resolution and even verification. It was paramount to achieving any kind of finished product and to say that it proved a helping hand is an understatement. Without resources readily available any support a student has would be severely diminished. My only reservation is that at stages I personally felt that I did not have as much material as I could have. From the early stages of the project, it was thought-provoking to see how the kalman filter actually worked. Without deep examination, one could not understand its capabilities and what I believe of it now is far removed from any initial reaction towards it.

Its implementation in VHDL although at times difficult and even straining was all in all a worthwhile experience. My familiarity with the language grew in leaps and bounds as the project progressed and in the later stages I felt I had really obtained a firm grasp of the key concepts of VHDL. For a relatively inexperienced program-

mer, it was refreshing to know that most things in VHDL can be done using simple statements, namely processes, if statements and loops. This strongly influenced my learning curve as the spectrum of what was needed to be known could often be kept to a minimum. I would now have a strong belief in what was attempted to be instilled in me from first year. This being, that the only way to learn anything in any programming language is to actually take the time to sit down and do it for yourself, without interruption. The hardware labs provided an environment to do just that, an environment that is not always available when dealing with other languages. It provided solitude, isolation and above all somewhere to think and to work. I wished I had taken this advise earlier on, and my relationship with and of course attitude towards the course would have been different. Ideally this habit should come naturally to a student but unfortunately for me it was a case of needs must. It was only toward the end of the project that I really appreciated how dedication and hard work were of utmost importance and anything else would come second. Arguably interest should remain high on ones agenda but where this is lacking dedication towards some form of interest could in circumstances compensate. Despite at times being set back or disappointed, sometimes this could have been mistaken for negativity, there were areas of the project that proved enjoyable. There is an immense sense of achievement as stages are completed, goals obtained and the finishing line in sight as opposed to a tiny spec far, far away. This of course also empowers the student and provides confidence for the next stage. As confidence builds the next stage is more do able. As things become easier, productivity grows and time is more well spent. It is only when resolution seems impossible that ones ability to apply themselves wane and that time is consumed needlessly. Also paths taken that need not be taken are also undeniably another factor that uses time and resource. However this is probably all part and parcel of the learning curve. Indecisiveness and reliability on trial and error understandably did not help my case but I suppose that this could be eliminated with practice.

6.2 Doing it all over again ....

At my demonstration, I was asked that all important question "what would I do differently if I was starting the project now?" I felt I hadn't answered the question as well as I could have. Obviously a lot of things could have been done differently but at times too many approaches seemed to be more of a problem than anything. Justifiably I would spend less time on research as full knowledge of the kalman filter although interesting was a little more than useless. Also given that I did not have time to put my project on the board, thorough research of FPGAs seemed a little wasted. Time like this could have been better spent on actual coding and at the very least getting a working inverter. Too much time was spent checking alternative code that had no reference to my project, just to see if a particular thing worked in VHDL. A lot of frustration came from the fact that at times things seem much harder to do in this hardware language than they would in an alternative software one. Also my limited knowledge on libraries, types and packages left me naturally disadvantaged. As time was an issue I would have set out a timetable for steps to be achieved. I would not have allowed myself to be as flexible as I was though. In hindsight looming deadlines for other coursework at times did not give me much scope. If I started tomorrow with the benefit of what I now know I would be more focused and have a definite plan in mind of the path the project would take. I would also not allow myself to become disheartened when things didn't work out as planned and with more time I would be able to step back, take a deep breath and regain my focus. In saying this continually crashing computers wouldn't help my cause. Needless to say timing was a major issue and everything considered there just seemed there wasn't enough of it. With the benefit of hindsight things could have been managed better and demonstration more successful.

6.3 Future Work

Scope for improvement - obviously, quite definitely. My work is only a tip of the iceberg in comparison to what could be done with this project. With adjustments across the board, namely with a variable size for matrices and bit widths, the kalman filter algorithm could be successfully carried out in VHDL. Also some work would be needed for an efficient inverter and fractional divisor. This all being achieved it would be time to implement the all important, aforementioned equations. There would only be a very small transition between having a working inverter and having a working kalman filter implementation. I strongly propose that another student take on this project and see what they can achieve. Undoubtedly the finished product would be well worth the hard work and I personally would take a great interest in its implementation in the later stages.

BIBLIOGRAPHY

(1.) Pellerin Taylor - VHDL Made Easy, 1997 Prentice Hall

(2.) Lipsett/Schaefer/Ussery - VHDL: Hardware Description and Design, Eleventh printing 1993

(3.) Weng Fook Lee - VHDL Coding and Synthesis with Synopsys, 2000 Academic Press

(4.) Morris M Mano - Logic and Computer Design Fundamentals, 3rd Edition

(5.) Wiley - VHDL Programming with Advanced Topics, Wiley Professional computing

(6.) Douglas E Ott, Thomas j Wilderotter - Introductory VHDL from Simulation to Synthesis, Klumei Academic Publishers

(7.) Smith, Douglas J - HDL chip design: A Practical Guide for Designing, Synthesising and Simulating ASICS and FPGAs using VHDL or Verilog, Madison, Al Doone Publications(1996).

(8.) Yu-chin hsu, Kevin F Tsai, Jessie T Liu, Eric S Lin - VHDL Modelling for Digital Design Synthesis, Klumer Academic Publishers

(9.) Ercegovac, Lang, Moreno - Introduction to Digital Systems

(10.)Kevin Skahill - VHDL for Programmable Logic, Addison Wesley

(11.) Charles E. Roth, Jr.- Digital System Design Using VHDL, PWS Publishing Company

(12.) Sudhakar Yalamanchili - VHDL Starter's Guide

(13.) Brian Hahn - Essential Matlab for Scientists and Engineers

(14.) Darren Redfern - The Matlab Handbook

(15.) Anton - Elementary Linear Algebra, John Wiley and Sons Inc

(16.) Hoffman/Kunze - Linear Algebra, Second Edition

(17.) Lang - Linear Algebra, 3rd edition

(18.) F.R. Gantmacher - The Theory of Matrices, Amer Mathematical Society

(19.) http://www.xilinx.com/bvdocs/appnotes/xapp204.pdf

(20.)http://www.eda.ei.tum.de/forschung/vhdl/

(21.) http://www.eda.org/comp.lang.vhdl/FAQ1.html

(22.)http://ieeexplore.ieee.org/

(23.)http://www.acc-eda.com/

(24.)http://toolbox.xilinx.com/

(25.)http://www.ashenden.com.au/