NTNU
Norwegian University of
Science and Technology

[SmallSat] FPGA Implementation of PCA Dimensionality Reduction Technique

Aysel Karimova

Embedded Computing Systems
Submission date: June 2018
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Milica Orlandic, IES

Norwegian University of Science and Technology
Department of Electronic Systems

This page [is] intentionally left blank.

Abstract

The aim of the thesis is to develop an efficient hardware implementation of the PCA (Principal Component Analysis) algorithm for dimensionality reduction in hyperspectral imaging. PCA includes the calculation stages as normalization, covariance matrix calculation and eigenvalue decomposition. An efficient hardware implementation of covariance matrix calculation has been designed in the specialisation project (Karimova, 2017) which uses high parallelism. The main focus is performing eigenpair decomposition on the covariance matrix and using BRAM (block RAM) as intermediate storage such that a large covariance matrix can be decomposed in a compact module. Fully parameterized FPGA implementation of eigen decomposition is developed and performed on the covariance matrix. Cyclic-by-row Jacobi method is used to find all the eigenvalues and eigenvectors concurrently. CORDIC algorithm is used with Jacobi for more efficient implementation. The Jacobi is designed in such a way that it can adapt to any FPGA regardless of the amount of available resources. This allows finding solutions to huge dimensional matrices in even small-sized FPGAs, e.g 100×100 by sacrificing a small amount of execution speed. The whole synthesizable PCA design is modelled using VHDL attempting to optimize the design. Zedboard Zynq-7000 series is used as the FPGA. The design is tested on MATLAB. All hardware modules are synthesized using Xilinx Vivado-2017.3 tool. In addition to this, the proposed design is compared to previous works done.

Preface

The thesis work is a part of the university satellite project called SmallSat at NTNU (Norwegian University of Science and Technology). The work is carried out under the supervision of Professor Kjetil Svarstad and Milica Orlandic. Special thanks to my co-supervisor Milica Orlandic for her great help and guidance throughout the project. I appreciate her motivating as well as demanding character which helped me to become smarter and learn more than one can expect to learn from a thesis work. It is incredible to see how much this thesis contributed to me in this short time. In the beginning of the thesis, I was new to VHDL language, but in the end, I ended up writing thousands of lines of useful code for an algorithm I had never heard about before. This thesis work allowed me to use my mathematical background together with the hardware and make the best out of the two.

June, 2018
Aysel Karimova

Table of Contents

List of Tables

List of Figures

Abbreviations and Acronyms

PCA	=	Principal Component Analysis
FPGA	=	Field-Programmable Gate Array
HSI	=	Hyperspectral Imager
VHDL	=	VHSIC Hardware Description Language
VHSIC	=	Very High Speed Integrated Circuit
MATLAB	=	matrix laboratory
CORDIC	=	COordinate Rotation Digital Computer
sym-Schur	=	symmetric Schur (Algorithm)
SSD	=	Symmetric Schur Decomposition
BRAM	=	Block Random Access Memory
LUT	=	Look-up Table
FF	=	Flip-flop
DSP	=	Digital Signal Processor

Chapter 1

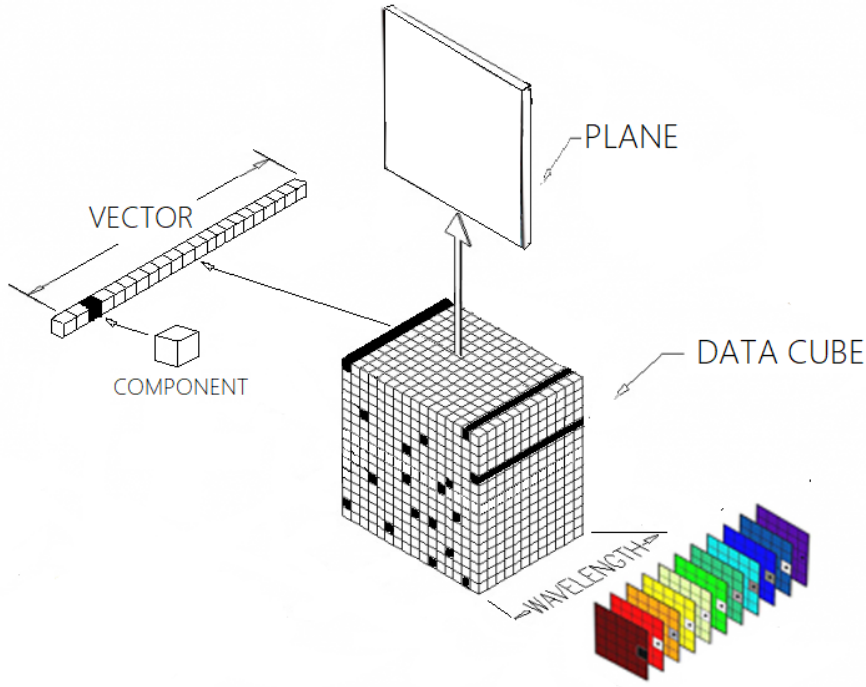
INTRODUCTION

1.1 Motivation

PCA (principal component analysis) algorithm, which is known as the oldest and quite popular multivariate analysis technique, is widely used in many applications to extract crucial information from a given inter-correlated dataset matrix. PCA is a very practical signal processing algorithm for dimensionality reduction and data compression. The key idea in PCA is to look for a lower dimensional representation for the given data. It deduces the most useful information from huge datasets. The more useful information, the more variance in the data. This, in turn, reduces the dimensionality of the input dataset. In order to achieve this, it generates new dataset variables through linear combination of the initial dataset. These variables are called principal components. The process is called the projection of the dataset variables onto principal components. There are several methods to deduce the principal components (also called eigenvectors). The first eigenvector is defined as the direction with the biggest variance. The mathematical operation, which computes a series of eigenvalues and eigenvectors, is called eigenanalysis. Eigenanalysis is performed on a square, symmetric matrix. There are several popular numerical techniques for eigenanalysis. Jacobi method is one of the earliest matrix algorithms and in recent decades it is still considered as one of the most powerful techniques. This technique is of current interest because it is amenable to parallel computation and because under certain circumstances it has superior accuracy according to (Golub and Van Loan (1996)). Only a few of the research works done until now reported hardware implementation of Jacobi algorithm. In this thesis, an efficient hardware implementation of Jacobi algorithm is achieved. To make it more efficient, CORDIC algorithm is used which utilises inexpensive operations to calculate complicated functions.

The application of the proposed design of Jacobi method in this thesis may not only be limited with PCA. Being fully parameterizable, it can also be used in various signal processing applications such as radio detection and ranging, other dimensionality reduction algorithms, machine learning algorithms, target detection, face recognition algorithms and in sparse linear systems or to solve linear system equations to name a few.

Figure 1.1 Visualisation of a data cube. Examples for plane, vector, component (pixel) are indicated. The vectors highlighted with black color are chosen randomly for covariance matrix computation.



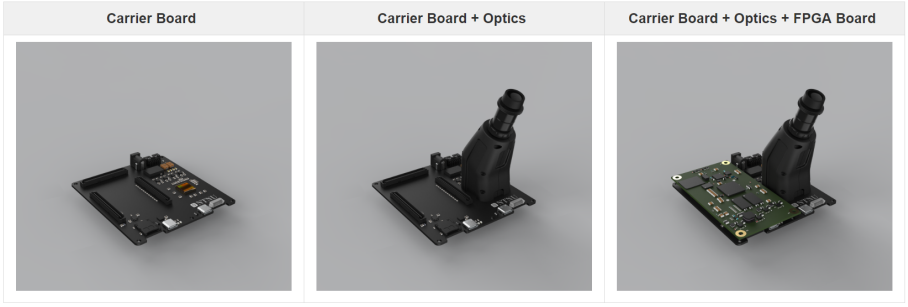
1.2 Background

This thesis work is a possible contribution to a cubesat design: **SmallSat** proposed in (Mariusz, E. Grøtte. et al) by NTNU. The work consists of the design of a small satellite (SmallSat) with a small push-broom HSI (Hyper-Spectral Imager). The HSI and SmallSat are under development and several versions of the cubesat and multi-agent system are expected to be operational in coming years. The main focus of the HSI is for both oceanographic measurements and synoptic in-situ field measurements. This is a novel approach and has a remarkable potential for reducing cost and improving data quality in oceanography. The mission objectives are given below as in the design documentation (Mariusz, E. Grøtte. et al):

1. Primary: To provide and support oceanography mapping through Hyper-Spectral Imager (HSI) payload and on-demand autonomous communications in an concert of robotic agents at higher latitudes in vicinity of the Arctic/Norwegian coast.

2. Secondary: To collect statistical data and to detect and characterise spatial extant extent of algal blooms, measure primary productivity using emittance from fluorescence generating micro-organisms, and other substances resulting from aquatic

Figure 1.2 Renders of V1R1 (Version 1, Revision 1) electronics design of the hyperspectral imager, fall 2017.



habitats and pollution to support environmental monitoring, climate research and marine resource management.

3. Tertiary: Investigate analytical reduced-order sizing relationships between identified performance characteristics and constituent SmallSat properties as related to oceanography.

4. Quaternary: Build strong competence on and strengthen prospect of nano- and micro-satellite systems as supporting intelligent agents in integrated autonomous robotic systems for dedicated marine and maritime applications.

5. Quinary: Describe the scientific methodology that will be adopted for the research, and coordinate the project plans with other ongoing research activities and NTNU and with engineering and scientific collaborators.

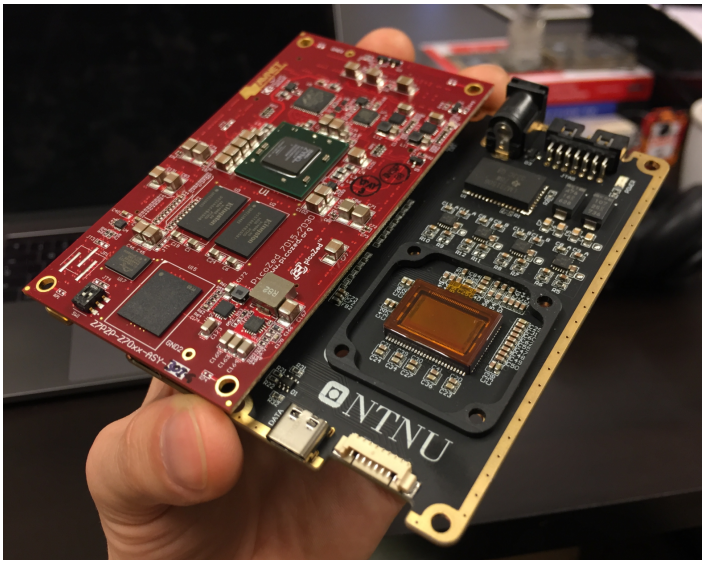
The role of the thesis on this project is dimensionality reduction of hyperspectral images captured by the push-broom hyperspectral camera. Minimised transmission and processing time and high throughput are important for better performance. The following information is retrieved from (Veisdal, 2018). The first prototype of the hyperspectral imager is shown in Fig. 1.2.

The image sensor used is a CMOSIS CMV2000. Additionally, the industrial version of the Avnet PicoZed board is used for the Smallsat project as the FPGA, its manufacturer number is AES-Z7PZ-7Z030-SOM-I-G and shown in Fig. 1.3.

For testing purposes, Xilinx ZedBoard is provided.

An example of a cubesat is visualised in Fig.1.1. Each pixel is given by a vector and each vector has 100 **components**. Each component belongs to one corresponding band or image plane (to be called **"plane"** throughout the report). All the components on one plane define the image size. The number of the planes correspond to the size of the matrix. It can be deduced that, the more planes, the bigger the module. Besides, from the perspective of the hyperspectral camera, the number of channels in the camera is equivalent to the number of the planes in the data cube. In the case of SmallSat, there are 100 planes in the data cube. One vector consists of the single components of each plane. In the hardware, the data is streamed through plane by plane.

Figure 1.3 Full assembly of the PCB, January 2018.



1.3 Organisation of the Thesis

This section describes the organisation of the rest of the thesis. Chapter 2 describes the state-of-the-art of PCA algorithm and computation steps by giving a brief overview of the covariance implementation. Previously reported related works are also mentioned. Chapter 3 describes the Jacobi Method used for the eigen decomposition along some historical and mathematical background. A fast and area-efficient method called CORDIC for calculation of trigonometric functions is also explained in this chapter. Chapter 4 presents the proposed hardware architecture of all the modules used in Jacobi implementation from a simple initial design to more efficient final design. Chapter 5 explains how Block RAM is used as an intermediate storage between covariance design and eigen decomposition design. Chapter 6 shows the synthesis results and metrics of the final design. Discussion and concluding remarks are also given in this chapter. Appendix consists of VHDL codes for the proposed final design.

Chapter 2

PCA ALGORITHM

2.1 Main Steps of computation

There are several steps in calculating PCA algorithm. They are summarized in Fig. 2.1. Before doing PCA, two assumptions should be made:

1. **Mean centering**: Mean is the average across each direction in the data. "Centering the data" means subtraction of the mean from every attribute. Although this step is not necessary, it can help to avoid difficulties. If mean subtraction is not performed, the first principal component might correspond similar to the mean of the data instead of the biggest variance. Even without the mean centering, the final result will be the same.

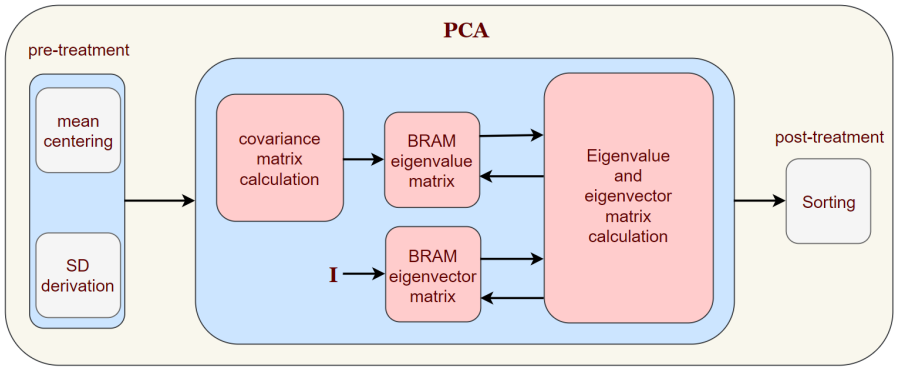
2. **Division by standard deviation**: If the variables have different units or weights in the data, it is necessary to normalise data to get a reasonable covariance analysis among all such variables. In order to make the data "unit-free", this assumption should be made. "Unit-free" data guarantees that the variance of the data in every dimension is 1. To normalize the data, every dimension in the data is divided by the corresponding standard deviation. The formula for standard deviation and mean centering applied together:

$$sd = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}} \tag{2.1}$$

The thesis work concerns the implementation of the main steps of PCA in hardware, which are:

1. **Calculation of the covariance matrix**: Covariance is the measure of the correlation between two or more variables. It indicates if the variables change together or in

Figure 2.1 The figure shows all possible steps of PCA algorithm. pre-treatment and post-treatment steps are not necessary to perform, but they might avoid difficulties. Red boxes are implemented/used in the thesis. Covariance matrix multiplication has been implemented before the thesis, its eigenvalues and eigenvectors are found using two BRAMs (Block RAM) as storage in the thesis work.



the opposite directions. Covariance is always calculated between two variables, so covariance matrix for n-dimensional data is made up of the covariances between every two variables. A visualisation of a covariance matrix for n dimensions can be shown as:

$$\begin{bmatrix} cov(0,0) & cov(0,1) & cov(0,2) & \dots & cov(0,n-1) \\ cov(1,0) & cov(1,1) & cov(1,2) & \dots & cov(1,n-1) \\ cov(2,0) & cov(2,1) & cov(2,2) & \dots & cov(2,n-1) \\ \dots & \dots & \dots & \dots & \dots \\ cov(n-1,0) & cov(n-1,1) & cov(n-1,2) & \dots & cov(n-1,n-1) \end{bmatrix}$$

Besides, as the covariance matrix is a square symmetric matrix, the entries on the diagonal of the covariance matrix consists of the covariance between a variable and itself, which is called variance. Variances are highlighted in the matrix above. Covariance calculation is done with the formula below assuming the data is already normalized:

$$cov(x_i, x_j) = \frac{\sum_{i=1}^{n} x_i x_j}{n-1} \tag{2.2}$$

2. **Calculation of the eigenvectors and eigenvalues**: PCA is a procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of linearly uncorrelated variables called principal components. PCA can be done by eigenvalue decomposition of a data covariance matrix. It finds a lower number of dimensions which explains the variance in the data best. If it is

applied to an object, PCA finds the projection of that object from the most informative viewpoint. It aims to reduce the dimensionality of the data, that is, to describe it with lower number of dimensions. These new dimensions are called principal components (PC in PCA). It follows that PCA aims to find the Principal Components. The first principal component corresponds to the first eigenvector which has the largest eigenvalue. The first eigenvector is the direction of the biggest variance from the covariance matrix in the data. All eigenvectors have their corresponding eigenvalues. Eigenvalues are the magnitude of the variance of a projection.

The next step could be **sorting** eigenvectors by their decreasing eigenvalues.

2.2 A brief overview of the specialisation project: Covariance Matrix Calculation

Specialisation project (Karimova, 2017) has been done before the master thesis and involves a detailed research prior to the PCA algorithm and the implementation of the first vital step of PCA: covariance matrix calculation. This section includes a general overview of what has been done as well as the functionality of the covariance matrix calculation on hardware. In the specialisation project, highly parallelised, optimised and fully parameterisable hardware design of covariance matrix calculation has been achieved. The thesis project performs eigen decomposition on this resulting matrix.

One of the main optimisations on covariance matrix calculation is subsampling which is proposed in Qian (2008) to reduce computational complexity of PCA. The paper proves that for the compression of the data, encoding only a subset of PCs is enough rather than full complement. This is called subsampling. It can be explained as follows: Instead of using all *M* pixels, a small subset of pixels of size $M^{'}$ is used where $M^{'} << M$. The heuristic in Makhoul and Gish (1985) dictates that $M^{'}$ should be at least *10N*:

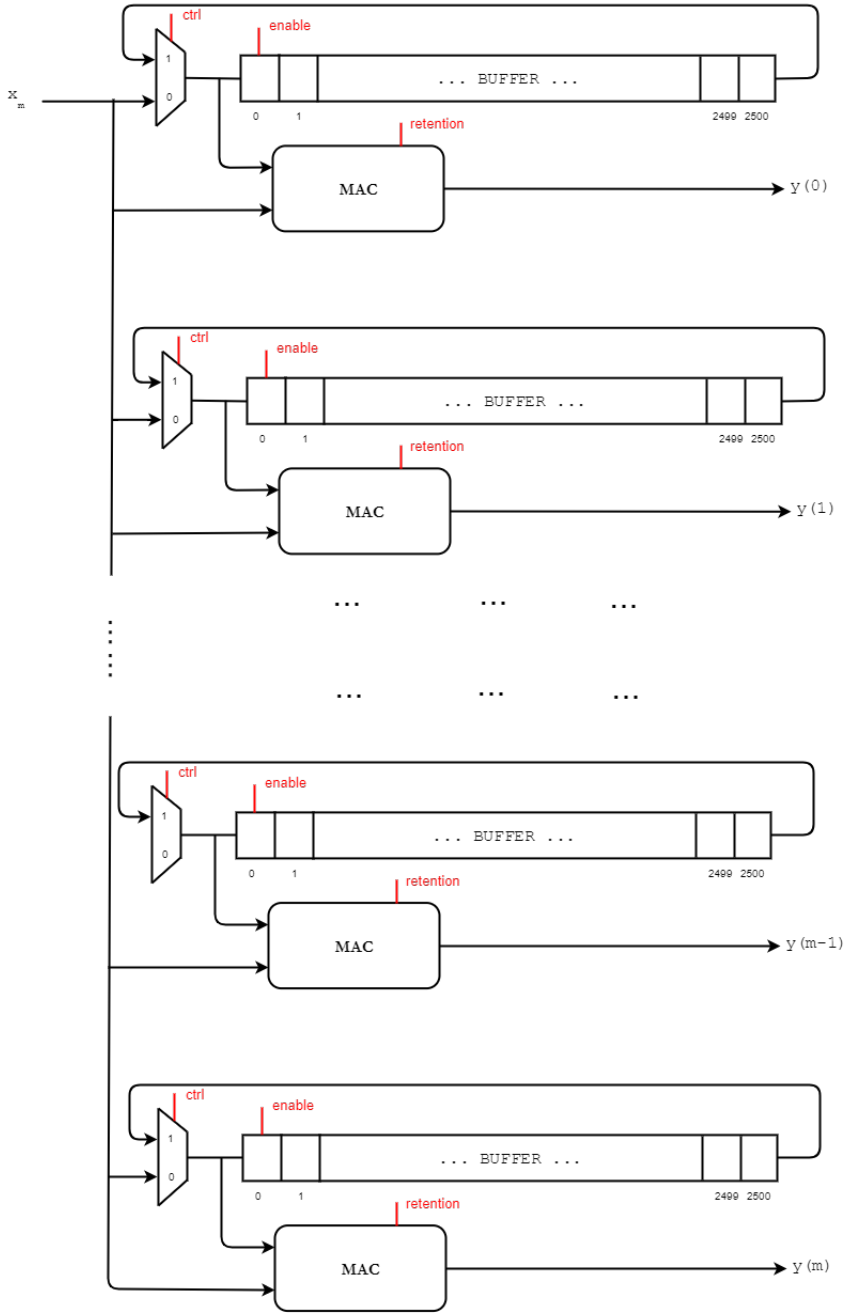
$$M^{'} \gtrapprox 10N$$

Besides, subsampling results in a transform which is just approximately the spectral PCA of the original image. The data size of the cube for the project is [500,500,100]. Therefore, these values are $M = 500^2 = 250000$ and $N = 100$. Consequently, taking $M^{'}$ greater than $10N$, we get $M^{'} \gtrapprox 1000$ and hence, finding 1% of subsampling, $M^{'} = 2500$ is the sufficient number of pixels in one plane. These pixels are chosen randomly.

For the 100 plane inputs, the target is to obtain the following covariance matrix:

$$\begin{bmatrix} cov(0,0) & cov(0,1) & cov(0,2) & \dots & cov(0,98) & cov(0,99) \\ cov(1,0) & cov(1,1) & cov(1,2) & \dots & cov(1,98) & cov(1,99) \\ cov(2,0) & cov(2,1) & cov(2,2) & \dots & cov(2,98) & cov(2,99) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ cov(98,0) & cov(98,1) & cov(98,2) & \dots & cov(98,98) & cov(98,99) \\ cov(99,0) & cov(99,1) & cov(99,2) & \dots & cov(99,98) & cov(99,99) \end{bmatrix}$$

Figure 2.2 The parallel computation of the covariances. **m** is the number of buffers which can be defined by the user.



Another optimisation is calculating only upper-triangle and duplicating the results around the main diagonal to the lower-diagonal. This reduces the calculation overhead significantly.

A novelty in this project is that the design is highly parallelised. The proposed design is shown in Fig. 2.2. Any number of buffers can be used for storing and streaming the input data while holding a condition that the number of the buffers divide the number of the planes for the best performance. Each buffer reduces the computation time significantly. One buffer is as big as one plane. All elements of one plane is streamed through to the first buffer while calculating the covariance of the first diagonal element of the matrix, *cov(0,0)*. When the first buffer is full, the **ctrl** signal for the first buffer is de-asserted and the next buffer is asserted. While the second plane is streamed through to the second buffer, it calculates the second diagonal element *cov(1,1)*. Simultaneously, the first buffer and the second buffer calculate the element *cov(0,1) = cov(1,0)*. All parallel calculations finish at the same time in spite of various starting times because each time there is one less calculation on upper-diagonal. The **enable_driver** signal, in turn, enables a number of buffers to work in parallel by asserting the corresponding bits. The input to the MAC unit is handled by a signal: **retention**. The **retention_driver** signal is assigned to *1* only if the first component is input to the buffer, it is set to *0* elsewhere. In this case, MAC performs a product of the input either by itself or with the last element of the buffer which has been rotated around. In all other cases, MAC has its typical functionality as multiply-accumulate.

A formula for the computation time of covariance module was generated in the specialisation project:

$$\begin{aligned} &computation\ time = \\ &c * (p + (p - 1 \times m) + (p - 2 \times m) + (p - 3 \times m) + \dots \\ &\qquad\qquad \dots + (p - (\frac{p}{m} - 1) \times m)) \end{aligned} \tag{2.3}$$

or

$$computation\ time = c * \Big(\sum_{i=0}^{\frac{p}{m}-1} p - i \times m \Big) \tag{2.4}$$

where **m** represents the number of buffers, **c** for components and **p** indicates the number of planes. Hence, for instance, the computation time for 100×100 matrix with 2500 components using 4 buffers can be calculated as:

$$\begin{aligned} &computation\ time = \\ &= 2500 \times (100 + (100 - 4) + (100 - 2 \times 4) + (100 - 3 \times 4) + \dots \\ &\qquad \dots + (100 - 24 \times 4)) = 2500 \times 1300 = \\ &\qquad\qquad = 3\ 250\ 000\ clock\ cycles \end{aligned} \tag{2.5}$$

2.3 Previous work

Until today, much research has been done concerning the use of PCA algorithm in signal processing, although only a few of them have implemented it on hardware. Although the

proposed high-throughput PCA architecture in a master thesis by (Uday, 2016) is quite practical for 4x4, 8x8, 16x16 matrices, it is not efficient for larger matrices. The design uses QR decomposition method which is similar to Jacobi, but QR does not have an advantage of being parallelized. The biggest size for input matrix shown in this work is 16x30 matrix which uses 56% of LUTs, 19% of the memory blocks and 88% (3250 DSP48 Slices) of the DSP48 modules on a Xilinx Virtex-7 while operating at 183 MHz. Additionally, the design implements matrix multiplication which uses 512 DSP48 blocks itself. The maximum input size can be 32x32 on the same FPGA due to limitations of the resource usage.

As Jacobi algorithm has a broad range of usage, it has been proposed to solve many other problems for various algorithms as well. In the master thesis project proposed by Jin (2014), MUSIC (Multiple Signal Classification) algorithm is implemented in ClaSH (CAES (Computer Architecture and Embedded Systems) Language for Synchronous Hardware). MUSIC algorithm is a classic subspace-based Direction of Arrival (DOA) estimation method which has similar computation principle as PCA such that implementing eigen-decomposition on the covariance matrix. The work uses Jacobi together with CORDIC for eigenvalue decomposition. The design is using Systolic array to implement a Parallel Jacobi. Another related work is proposed in Gonzales (2015). The work proposes the first FPGA implementation of HySime algorithm to estimate the number of endmembers in remotely sensed hyperspectral data. The FPGA used is Virtex-7 XC7VC690T. The algorithm has two stages in which in second stage eigenvectors of the correlation matrix have to be found for the signal subspace estimation.

Stopping criterion used is the comparison of the largest off-diagonal element with a provided value in each iteration which requires complex off-diagonal search. In order to get eigenvector matrix the product of each step matrices are used to reach eigenvalue matrix. Their way of obtaining eigenvector matrix is as follows:

$$P_k = Q_1 \cdot Q_2 \cdot Q_3 \cdot \cdots \cdot Q_k \tag{2.6}$$

Their procedure consists of the following: a parallel processing for Jacobi preselects the rotation with the largest off-diagonal element. Then the concerned rows and columns are zeroed out. Then the next biggest element is picked and all the others are zeroed out. All calculations are done in parallel. Then all the results are applied to the matrix in the order given. However, this method would be efficient only for really small matrices as it requires off-diagonal search in for each rotation. To carry out the operations of Jacobi method, a *multiplicator*, a *trigonometry* module, a *generator* and a *control unit* are used and for storing eigenvector and eigenvalue matrices Block RAMs are used to implement two FIFOs. However, trigonometry functions are implemented using Sym-Schur formulas as they are, which require expensive square root and division operations. The generator module is later modified to use *round-robin* algorithm which is a different naming for the parallel Jacobi method. Total hardware utilisation of 36.61% for this stage of the algorithm is reported. The number of BRAMs used is 650.

Another work is proposed in (Bravo, 2006) for solving eigenvalue problem according to the Jacobi's method and the work is applied in the PCA algorithm of artificial vision. The main contribution of their work is low execution time and minimal internal FPGA usage due to CORDIC. The drawback of their design is that the maximum size of the

matrix is limited to FPGA resources. The accuracy of the design results in 1.4% data error in worst case. Their tested maximum matrix size is 16×16 with 18 bits elements showing the results with 0.43% error and 19 iterations.

To the best knowledge of the author, most works done until now propose solutions for eigenanalysis are limited to the resource usage of the FPGAs. Therefore, the maximum possible matrix size can be performed on an FPGA is around 32x32.

This thesis work proposes a solution to this issue. The amount of resource usage in any FPGA is customisable in the design. The design can take any large matrix sizes while still using the similar amount of FPGA resources as small matrix sizes. Especially, for the SmallSat project, the size of the input matrix is 100x100 while the FPGA used for testing (ZedBoard Zynq-7000) has small number of available resources (220 DSP48s, 53116 LUTs, etc). That is why, the design is adapted to be used in small-sized FPGAs. If the design is used in FPGAs with more available resources, it will be faster by adjusting the parameter **n**. **n** defines how many computational slices will be used in the computation of two modules in the design: **off.vhd** and **eigenanalysis.vhd**.

Chapter 3

EIGENANALYSIS

This chapter first answers the questions "What is Jacobi method?" and "Why it is chosen?" before going into detail of how it works. It also introduces the terms "parallelisation" and "diagonalisation" and giving a short historical background. After that, the-state-of-the-art idea of Jacobi method and its types are presented followed by the mathematical background of CORDIC algorithm.

3.1 Preferred method

In the specialisation project (Karimova (2017)), various methods for eigenvalue problem have been analysed. After careful overview of iterative numerical methods namely, Gauss-Siedel (GS), Successive-over-Relaxation (SOR), QR-decomposion and Jacobi methods, it is proposed that the last method is the best starting point for implementation on this problem. In numerical linear algebra, Jacobi transformation method of matrix diagonalization (also known as Jacobi iterative algorithm) is used for finding eigenpairs. The work done by Demmel (1992) proves in the their work that "Jacobi's method (with a proper stopping criterion) computes small eigenvalues of symmetric positive definite matrices with a uniformly better accuracy bound than QR, divide and conquer, traditional bisection, or any algorithm which first involves tridiagonalizing the matrix". This work is supported by the research done by Mathias (1995). Jacobi method calculates not only eigenvalues, but also the corresponding eigenvectors. The method applies to only symmetric matrices (in which the matrix output from the covariance calculation in the first step of PCA is symmetric (Karimova (2017))). The main reason is that Jacobi methods for symmetric eigenvalue problem is amenable to parallel calculation since each rotation affects only two rows and two columns of the matrix. Parallelism is known as a powerful feature of hardware implementations which can significantly reduce the execution time by performing independent operations simultaneously. Besides, under certain circumstances Jacobi's methods has superior accuracy, thus they can be considered state-of-the-art methods for solving eigenvalue decomposition problems. As the resulting covariance matrix in this work has dimensions of 100×100, it is advantageous if the next step of PCA is fast enough to finish

the dimensionality reduction quickly. The other reason is that the Jacobi method is considered as the base for some other methods such as GS and SOR. Its implementation can easily be modified to aforementioned methods.

The calculation process of the eigenvalues and eigenvectors of a real symmetric matrix is known as diagonalization. A diagonal matrix is a matrix that all of its nonzero entries are on its diagonal. Note that it does not state that the entries on the diagonal are nonzero.

This algorithm is named after Carl Gustav Jacob Jacobi. He first proposed the method in 1846, whereas, it only became popular after 1950 with the development of computers (Wikipedia, 2017). The working principle of Jacobi algorithms consists of performing a sequence of orthogonal updates to a matrix $\mathbb{A}$ such as $\mathbb{A} \leftarrow \mathbb{R}^T \mathbb{A} \mathbb{R}$ provided that when each time $\mathbb{A}$ is updated, it will be "more diagonal" than its forerunner. Here, $\mathbb{R}$ is rotation operation. A number of iterations will eventually make the off-diagonal elements become zero or small enough so that they can be considered zero. It is worth to note that each step can undo some of the previously made zeros, but eventually the magnitude of the nonzero elements will be reduced. This is discussed more in the following sections.

3.2 The main idea of Jacobi

The Jacobi method aims to reduce the norm of the off-diagonal entries (Golub and Van Loan, 1996):

$$off(A) = sqrt(\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} a_{ij}^2) \tag{3.1}$$

where *off(A)* is called Frobenius norm.

In an N-dimensional space, this is achieved by the rotations of the form:

$$\mathbb{R}(p,q,\theta) = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix}$$

Here, c and s represents $\cos\theta$ and $\sin\theta$ respectively, where θ is the rotation angle. The rotation happens by an angle θ around the direction of the cross-product of the *i*th column and the *j*th column. This rotation form is called *Jacobi rotations*, also known as *Givens rotations*. The matrix $\mathbb{R}$ is an identical matrix, except its four elements c,s,c and -s.

For instance, in 3-D space, rotations for the three principal axes can be carried out with the following matrices:

$$\mathbb{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix} \quad \mathbb{R}_y = \begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c \end{bmatrix} \quad \mathbb{R}_z = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The simplified stages of Jacobi can be described as:

1. choosing an index pair (i,j) such that $1 \leq \text{i} < \text{j} \leq \text{n}$
2. calculating a (c,s) pair for the rotation matrix $\mathbb{R}$ such that the following matrix is diagonal:

$$\begin{bmatrix} a'_{pp} & a'_{pq} \\ a'_{qp} & a'_{qq} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \quad \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \quad \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

3. updating $\mathbb{A}$ with $\mathbb{A}' = \mathbb{R}^{\mathbb{T}}\mathbb{A}\mathbb{R}$ where $\mathbb{R} = \mathbb{R}(p, q, \theta)$.

After *k* iterations have been performed, the result is:

$$\mathbb{A}_{\mathbb{k}} = \mathbb{R}_{\mathbb{1}}^{\mathbb{T}} \cdot \cdots \cdot \mathbb{R}_{\mathbb{k}}^{\mathbb{T}} \cdot \mathbb{A} \cdot \mathbb{R}_{\mathbb{k}} \cdot \cdots \cdot \mathbb{R}_{\mathbb{1}} \tag{3.2}$$

where R_i are successive Jacobi rotations and the resulting matrix A_k is the eigenvalue matrix in which eigenvalues are in its diagonal.

After the eigenvalues are found, associating eigenvectors can be calculated by multiplying all the rotation matrices used to obtain matrix $\mathbb{A}$:

$$\mathbb{V}_{\mathbb{k}} = \mathbb{R}_{\mathbb{1}} \cdot \mathbb{R} \cdot \mathbb{R} \cdot \cdots \cdot \mathbb{R}_{\mathbb{k}} \tag{3.3}$$

To recapitulate, applying rotation matrix $\mathbb{R}(p, q, \theta)$ to symmetric matrix $\mathbb{A}$ means performing pre and post multiplications of $\mathbb{R}$ and $\mathbb{R}^{\mathbb{T}}$ to $\mathbb{A}$ respectively. Because $\mathbb{A}$ is diagonal, transposing $\mathbb{A}$ yields exactly the same matrix $\mathbb{A}$ = $\mathbb{A}^{\mathbb{T}}$. If the rotation is applied, the resulting $\mathbb{A}'$ will be as follows:

$$\mathbb{A}' = \mathbb{R}\mathbb{A}\mathbb{R}^{\mathbb{T}} = \begin{bmatrix} a'_{11} & \cdots & a'_{1p} & \cdots & a'_{1q} & \cdots & a'_{1n} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{p1} & \cdots & a'_{pp} & \cdots & a'_{pq} & \cdots & a'_{pn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{q1} & \cdots & a'_{qp} & \cdots & a'_{qq} & \cdots & a'_{qn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{n1} & \cdots & a'_{np} & \cdots & a'_{nq} & \cdots & a'_{nn} \end{bmatrix}$$

It is worth to note that $\mathbb{A}'^{\mathbb{T}} = (\mathbb{R}\mathbb{A}\mathbb{R}^{\mathbb{T}})^T = \mathbb{R}\mathbb{A}^{\mathbb{T}}\mathbb{R}^{\mathbb{T}} = \mathbb{R}\mathbb{A}\mathbb{R}^{\mathbb{T}} = \mathbb{A}'$ yields that $\mathbb{A}'$ is also symmetric (Wang, 2017).

In order to diagonalize the matrix, the rotation angle, which eliminates the off-diagonal element a_{pq}, must be found. a_{pq} is made zero while increasing the values of the elements in the diagonal. That is, the equation 3.4 is equal to zero:

$$0 = a_{pq} = (c^2 - s^2)a_{pq} + cs(a_{pp} - a_{qq}) \tag{3.4}$$

In case $a_{pq} = 0$, then (c,s)=(1,0) can be set. In all other cases, the above equation yields:

$$\frac{a_{qq} - a_{pp}}{a_{pq}} = \frac{c^2 - s^2}{cs} = \frac{1 - (s/c)^2}{s/c} = \frac{1 - t^2}{t} = 2\tau \tag{3.5}$$

The transitions used in above equation are as following:

$$t = \tan\theta = \frac{s}{c} = \frac{\sin}{\cos} \tag{3.6}$$

$$\tau = \frac{a_{qq} - a_{pp}}{2a_{pq}} = \frac{c^2 - s^2}{2cs} = \frac{\cos 2\theta}{\sin 2\theta} = \cot 2\theta \tag{3.7}$$

The obtained quadratic equation is solved for *t*:

$$1 - t^2 - 2\tau t = 0, \quad \text{i.e.} \quad t^2 + 2\tau t - 1 = 0, \quad t_{1,2} = -\tau \pm \sqrt{\tau^2 + 1} \tag{3.8}$$

Now, it is crucial to choose the smaller absolute value of the two roots, because the values of both $\sin\theta$ and $\cos\theta$ can not go over 1. Moreover, using the root *t*, results for *s* and *c* can now be further found:

$$c = \cos\theta = \frac{1}{\sqrt{1 + \tan^2}} = \frac{1}{\sqrt{1 + t^2}} \quad \text{and} \quad s = \sin\theta = \frac{\tan}{\sqrt{1 + \tan^2}} = \frac{t}{\sqrt{1 + t^2}} = tc \tag{3.9}$$

This way of decomposition is called **2-by-2 Symmetric Schur Decomposition** (Golub and Van Loan, 1996).

One of the most efficient optimisations in this project consists of algorithmic modification. As mentioned before, Jacobi rotations are pre and post multiplications of rotation matrix and its transpose to the main matrix. The visualisation of it has been given on a *5×5* matrix in (Wang, 2017) along its mathematical derivations:

$$RAR^T =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & -s & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & s & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} 1a_{11} & ca_{12}-sa_{14} & a_{13} & sa_{12}+ca_{14} & a_{15} \\ ca_{21}-sa_{41} & c^2a_{22}-cs(a_{24}+a_{42})+s^a_{44} & ca_{23}-sa_{43} & c^2a_{24}+cs(a_{22}-a_{44})-s^2a_{42} & ca_{25}-sa_{45} \\ a_{31} & ca_{32}-sa_{34} & a_{33} & sa_{32}+ca_{34} & a_{35} \\ sa_{21}+ca_{41} & c^2a_{42}+cs(a_{22}+a_{44})-s^a_{24} & sa_{23}+ca_{43} & s^2a_{22}+cs(a_{24}+a_{42})+c^2a_{44} & sa_{25}+ca_{45} \\ a_{51} & ca_{52}-sa_{54} & a_{53} & sa_{52}+ca_{54} & a_{55} \end{bmatrix}$$

It would be not so useful to perform multiplications on the elements which one of them is zero, since the result will always give zero. Therefore, the elements on the rows and columns other than the row **p** and column **q** do not change after an iteration. Hence, it is an efficient way just to update the altered elements only. This method will be implemented in hardware as well.

$$a'_{pq} = a'_{qp} = c^2a_{pq} - cs(a_{pq} + a_{qp}) - s^2a_{qp} = (c^2 - s^2)a_{pq} + cs(a_{pp} - a_{qq}) \quad (p \neq q) \tag{3.10}$$

$$a'_{pp} = c^2a_{pp} - cs(a_{pq} + a_{qp}) + s^2a_{qq} = c^2a_{pp} - 2csa_{pq} + s^2a_{qq} \tag{3.11}$$

$$a'_{qq} = s^2a_{pp} + cs(a_{pq} + a_{qp}) + c^2a_{qq} = s^2a_{pp} + 2csa_{pq} + c^2a_{qq} \tag{3.12}$$

$$a'_{pi} = a'_{ip} = ca_{pi} - sa_{qi} \quad (i \neq p, i \neq q) \tag{3.13}$$

$$a'_{qi} = a'_{iq} = sa_{pi} + ca_{qi} \quad (i \neq p, i \neq q) \tag{3.14}$$

$$a'_{ij} = a_{ij} \quad (i, j \neq p, q) \tag{3.15}$$

To the best of author's knowledge, this simplified method given on Wang (2017) has not been implemented many times on hardware before. Most of the previous works done on the hardware implementation of Jacobi usually uses matrix multiplication in each iteration.

To summarise: when performing Jacobi rotations around *(p,q)*, only rows and columns *p* and *q* are altered. It is not necessary to access and update the entire matrix.

This way of doing the calculations is also more efficient in terms of area and power on a hardware implementation when compared to matrix multiplication. It is also expected to get a reduced computational requirements by utilising this way.

Once a sufficient accuracy is reached in eigenvalue calculations, similar way has been later applied to finding eigenvectors of the matrix as well. This would, actually, make a huge improvement in the speed of the calculation. Eigenvector matrix calculation is made concurrent with eigenvalue matrix calculation. In the standard way, as eigenvector calculation is basically the multiplication of each new rotation matrix to the previous product. Thus, the new resulting matrix updates the previous product saved in each iteration. The calculation of eigenvectors in the work proposed in (Gonzales (2015)) is discussed in the **Related work** section. In this thesis, however, a simpler way of calculating eigenvector matrix is proposed. It is the same way as eigenvalue matrix calculation, that is updating the altered elements only. Roughly speaking, the main difference between the derivation of eigenvalue and eigenvector formulas is that the matrix to be rotated in the eigenvector calculation is the identity matrix. Updated elements can be obtained by two simple formulas:

$$v_{i,p} = cv_{i,p} - sv_{i,q} \tag{3.16}$$

$$v_{i,q} = sv_{i,p} + cv_{i,q} \tag{3.17}$$

where *i* is the row number, thus, only the two corresponding columns change while all the other elements stay the same. An interesting point here is that if the identity matrix is post-multiplied by the rotation matrix, the resulting eigenvectors will be ordered in columns, otherwise, if it is pre-multiplied by the transpose of the rotation matrix, the eigenvectors will be ordered in the rows of the resulting matrix.

When it comes to choosing the p and q indices in successive iterations, it varies for each form of Jacobi method. These forms are introduced in the following subsections.

3.2.1 Classical Jacobi Idea

From the standpoint of maximising the reduction of *off(A)* in Eq. 3.1, it would make sense to choose *(p,q)* pair so that a_{pq}^2 is maximal. That is exactly how Classical Jacobi chooses *(p,q)*. It follows that all off-diagonal elements must be sorted and compared in order to find the largest a_{pq}. Thus Classical Jacobi requires access to whole matrix. Then the found largest element is zeroed. Moreover, classical Jacobi converges in a linear rate. The key limitation with this algorithm is that the search for the optimal *(p,q)* has the complexity of $O(n^2)$. Hence, needed off-diagonal search makes it inefficient method to implement in hardware, although it might be preferable in software.

3.2.2 Cyclic Jacobi Idea

Cyclic Jacobi ignores the benefits of choosing the largest a_{pq} and rather cycles through each off-diagonal element. In other words, it does not require off-diagonal search. By picking a_{pq} sequentially, Cyclic Jacobi avoids time spent searching for the largest a_{pq}. That is why it is considerably faster than Jacobi's original algorithm. Cyclic Jacobi can be done in either row-by-row or column-by-column fashions. For instance, if $n = 4$, cycling for *cyclic-by-row* is as follows:

$$(p,q) = (1,2),(1,3),(1,4),(2,3),(2,4),(3,4),(1,2),\ldots \tag{3.18}$$

Cyclic Jacobi converges quadratically (Wilkinson (1962) and van Kempen (1966). To test this algorithm before implementing it in hardware, the following MATLAB code was written:

```
function [ A,V ] = jacobi_cyclic( A, tol )
n = size(A,1);
V = eye(n);
p = 1; q = 1;
%epsilon = tolerance*error_function
eps = tol*sqrt(sum(sum(A.^2)));
%initialise error variable for plotting later
error = zeros(1,1000);
i = 1;
%off_A is goal for error function
while (off_A(A) > eps)
```

```
    %choosing (p,q) for row-by-row cyclic
    [p,q] = next_pq( A,p,q );
    %calculation of sin and cos
    [ c,s ] = sym_schur_alt( A, p,q );
    %Construction of Rotation matrix
    R = eye(n);
    R(p,p) = c;     R(p,q) = s;
    R(q,p) = -s;     R(q,q) = c;
    %eigenvalues and eigenvectors matrix
    A = R' * A * R;
    V = V * R;
    %for plotting
    error(i) = off_A(A);
    i = i+1;
end
plot(error)
end
```

This code is for an initial general test of Jacobi function. As the stopping criterion, Frobenius form is used in Eq. 3.1. Indices are chosen based on the order in Cyclic-Jacobi and trigonometric calculations are done using symmetric-schur algorithm. Rotation matrices are created explicitly and then used in matrix multiplication to obtain eigenvalue and eigenvector matrices.

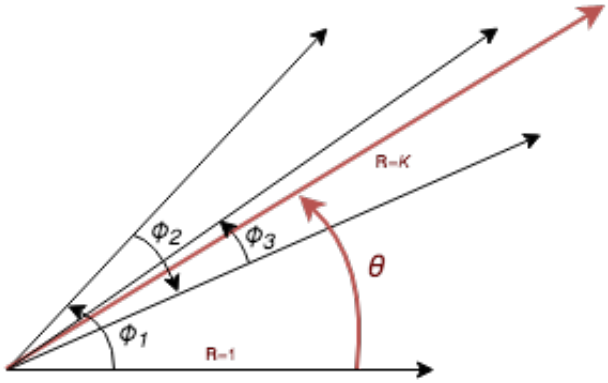
3.3 CORDIC

3.3.1 What is CORDIC?

This section introduces CORDIC and represents the step-by-step mathematical way of getting to the trigonometric functions needed in digital design of Jacobi algorithm.

In order to reduce the complexity of *Givens Rotations*, CORDIC algorithm is widely used. CORDIC is an acronym for COordinate Rotation Digital Computer. It is an iterative algorithm to rotate vectors. The usage of CORDIC helps to avoid computationally complex or expensive operations i.e, square root, division in the formulas 3.9. It is an algorithm to convert between polar and cartesian coordinates using shift, add and subtract operations. Using only mentioned operations, it can perform rotations by any given angle. It was proposed by Jack Volder in an article published in 1959. The aim for developing this algorithm, which can be implemented with digital logic, was to make complex calculations such as sine, cosine, arctangent, hyperbolic, linear, logarithmic, etc. more quickly and accurately. Later, the formula has been expanded, thus the use of it became more popular. CORDIC is an irreplaceable part of calculators especially for finding the value of trigonometric functions, square root and even division operations which are directly computable by CORDIC (Alan, 2008). CORDIC performs a rotation by a sequence of micro-rotations of an elementary angle where the original rotation angle θ can be expressed by the sum of

Figure 3.1 The figure is to support Eq. 3.19 showing how CORDIC uses micro-rotations to reach the original angle



all elementary angles ϕ_i:

$$\theta = \sum_{i=0}^{\infty} \phi_i \tag{3.19}$$

An example for micro-rotations is visualised in Fig. 3.1.

3.3.2 Mathematical derivation of CORDIC

The flow of this section is made as a general summary of CORDIC based mainly on the survey by (Andraka, Copyright 1998), a master thesis by Uday (2016) and video tutorial by (Weedman, 2013). The goal here is to explain how to produce the equations for CORDIC that uses simple operations such as addition, subtraction and shifts instead of the trigonometric functions and square roots. In order words, recreating $\sin$ and $\cos$ functions without multiplications or some sort of trigonometric functions.

Fig. 3.2 presents two vectors of the same magnitude. The second vector is obtained by rotating the first one by an angle called ϕ. When rotating the one located at $V_0 = [X_0 Y_0]$, new coordinates $V_n = [X_n Y_n]$ are obtained.

The new coordinates relative to initial point and the angle is computed using the formulas as below:

$$X_n = X_0 \cos(\phi) - Y_0 \sin(\phi) \tag{3.20}$$
$$Y_n = Y_0 \cos(\phi) + X_0 \sin(\phi) \tag{3.21}$$

By simplifying this formula, the next one can be obtained, which still involves some trigonometry:

$$X_n = \cos(\phi)[X_0 - Y_0 \tan(\phi)] \tag{3.22}$$
$$Y_n = \cos(\phi)[Y_0 + X_0 \tan(\phi)] \tag{3.23}$$
$$\tag{3.24}$$

Figure 3.2 Rotation angle

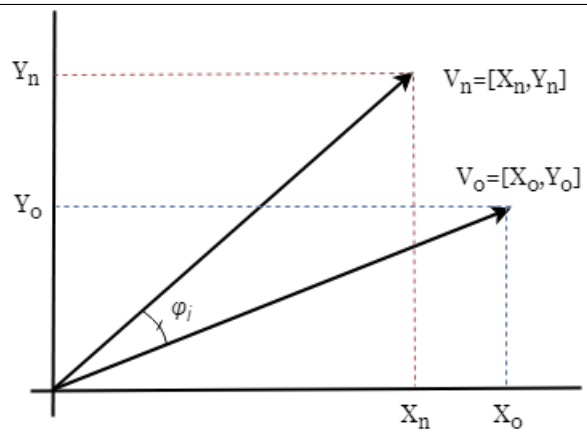


Table 3.1: Table for the given example showing $\arctan$ values for specific $\tan(\phi)$ functions.

i	$\tan(\phi) = 2^{-i}$	$\phi = \arctan(2^{-i})$
0	1	45
1	1/2	26.565
2	1/4	14.036
3	1/8	7.125
4	1/16	3.576
5	1/32	1.79
6	1/64	0.895
7	1/128	0.448
8	1/256	0.224
9	1/512	0.112

However, now an assumption can be made; the rotation angle is restricted to $tan(\phi) = \pm 2^{-i}$ where $0 < i < \infty$. Furthermore, the multiplication by the tangent term can be reduced to simple shift operation. Then, corresponding values will be $\tan(\phi) = \pm 2^0, 2^{-1}, 2^{-2}, 2^{-3}, \cdots = \pm 1, \frac{1}{2}, \frac{1}{4} \ldots$
Table 3.1 is created based on these values with their corresponding angles.

Because of the angle restriction assumption of tangent in the equation for $\cos$, $\tan$ can be substituted as below:

$$\cos(\phi) = \frac{1}{\sqrt{1 + \tan(\phi)}} = \frac{1}{\sqrt{1 + (\pm 2^{-i})^2}} = K_i \tag{3.25}$$

After substituting a new variable K_i in place of $\cos(\phi)$, the next step is defining the formulas based on the direction of rotation. The formulas has been changed as below to reflect if the rotation is made by a positive or negative angle. The value of d_i is plus and minus for positive and negative angles, respectively. The maximum rotation angle in either direction

Table 3.2: The extended table for the given example showing a series of corresponding $\arctan$ values for $\tan(\phi)$ functions. The resulting rotation and mini-rotation angles computed by CORDIC. The example is discussed in the text.

i	$\tan(\phi) = 2^{-i}$	$\phi = \arctan(2^{-i})$	Z_i	Rotation ϕ	Resulting angle
0	1	45	20	-45	-25
1	1/2	26.565	-25	26.565	1.565
2	1/4	14.036	1.565	-14.036	-12.47
3	1/8	7.125	-12.47	7.125	-5.346
4	1/16	3.576	-5.346	3.576	-1.77
5	1/32	1.79	-1.77	1.79	0.020
6	1/64	0.895	0.020	-0.895	-0.875
7	1/128	0.448	-0.875	0.448	-0.427
...	...	...	...	...	
...	...	...	...	...	
19	...	...	-0.000195		

is assumed to be 45 degrees or less.

$$X_n = K_i[X_0 - Y_0 d_i 2^{-i}] \tag{3.26}$$

$$Y_n = K_i[Y_0 + X_0 d_i 2^{-i}] \tag{3.27}$$

Using these last two equations, an algorithm was created to compute any given rotation angle, not only the fixed angles given in Table above. The algorithm implies that the desired angle of rotation is obtainable by performing a series of successively smaller elementary rotations. This is done in the scope of $\tan(\phi) = 2^{-i}$ where *i* gets the values from 1 to *n-1*. An example of this algorithm in the table below is taken from (Weedman, 2013). A desired rotation angle is taken 20 degrees in the example which is also the starting point. The core of the algorithm can be expressed as follows:

For each iteration if $Z_i > 0$, the current iteration angle is subtracted from Z_i, otherwise the current iteration angle is added to Z_i until $Z_i > 0$ approaches to zero or close enough to zero. Hence, appropriate X_n and Y_n calculations are made. The procedure of the algorithm is presented in Table 3.2.

It is worth to note that the sign of the angle is not important to K_i, because $\cos(-\phi) = \cos(\phi)$. Thus, in each iteration, K_i will be the same value not depending on the direction that rotation happens. Therefore, the only thing to be decided is which direction to rotate.

$$K_i = \frac{1}{\sqrt{1 + (\pm 2^{-i})^2}} = \frac{1}{\sqrt{1 + (2^{-i})^2}} \tag{3.28}$$

Furthermore, it is crucial for K_i to be a constant for each iteration and eventually, K_i term is needed to be eliminated from these equations at some point. The index *i* is chosen to

represent the current value, *i+1* will be the next value in the iterative process.

$$X_{i+1} = K_i[X_i - Y_i d_i 2^{-i}] \tag{3.29}$$
$$Y_{i+1} = K_i[Y_i + X_i d_i 2^{-i}] \tag{3.30}$$
$$Z_{i+1} = Z_i - d_i \tan^{-1}(2^{-i})] \tag{3.31}$$

Eliminating the scale constant from the iterative equations would, finally, yield an algorithm for vector rotation with only shifts and adds. Instead, the product of the K_i's can be applied later in the system or can be treated as a part of a system processing gain. As the number of iterations reaches infinity, this product approaches to a constant number 0.6073. Consequently, the rotation has a gain, A_n which is approximately 1.647, omitting the K_i term. The exact value of gain is computed with the following equation which depends on the number of equations:

$$A_n = \prod_{i=0}^{n-1} \frac{1}{K_i} = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \tag{3.32}$$

Cases of K_i term for first three iterations are calculated and approximation for infinite number of iteration becomes close to 1:

$$K_0 = 0.7071 \tag{3.33}$$
$$K_1 = 0.8944 \tag{3.34}$$
$$K_2 = 0.9412 \tag{3.35}$$
$$K_\infty \cong 1 \tag{3.36}$$

Not depending on the starting angle, K_i is always constant, hence, it can be removed from the final formulas of CORDIC. Finally, the following equations require only subtraction, addition and shift along with arctangent term. For this term, a Look-up table can be generated.

$$X_{i+1} = [X_i - Y_i d_i 2^{-i}] \tag{3.37}$$
$$Y_{i+1} = [Y_i + X_i d_i 2^{-i}] \tag{3.38}$$
$$Z_{i+1} = Z_i - d_i \arctan(2^{-i})] \tag{3.39}$$

Whereas, leaving out the K_i term, each result in the iteration is larger than it should be by a factor of $\sqrt{1 + 2^{-2i}}$. It is for this reason that the A_n term for the system gain is included.

The CORDIC rotator has 2 modes of operation. The first mode is called **rotation mode** by (Deprettere and Udo, 1984), which rotates the input vector by a user-specified angle (Uday, 2016). For rotation mode, the CORDIC equations for *i*-th iteration is:

$$X_{i+1} = [X_i - Y_i d_i 2^{-i}] \tag{3.40}$$
$$Y_{i+1} = [Y_i + X_i d_i 2^{-i}] \tag{3.41}$$
$$Z_{i+1} = Z_i - d_i \arctan(2^{-i})] \tag{3.42}$$

where

$$d_i = \begin{cases} -1, & \text{if } z_i < 0. \\ +1, & \text{otherwise.} \end{cases} \tag{3.43}$$

The resulting equations after a number of iterations become:

$$X_n = A_n[X_0 \cos(Z_0) - Y_0 \sin(Z_0)] \tag{3.44}$$
$$Y_n = A_n[Y_0 \cos(Z_0) - X_0 \sin(Z_0)] \tag{3.45}$$
$$Z_n \approx 0 \tag{3.46}$$

This algorithm can be created in digital logic to calculate vector rotation. Additionally, by setting $Y_0 = 0$ and $X_0 = \frac{1}{A_n}$, a function can be obtained which produces the following:

$$X_n = \cos(Z_0) \tag{3.47}$$
$$Y_n = \sin(Z_0) \tag{3.48}$$

This is the way that can produce sine and cosine functions in digital hardware or software. The other one is called **vectoring mode** which rotates the input vector to the *x* axis while recording the angle required to make the rotation (Uday, 2016). The CORDIC equations for this mode is as follows:

$$X_{i+1} = [X_i - Y_i d_i 2^{-i}] \tag{3.49}$$
$$Y_{i+1} = [Y_i + X_i d_i 2^{-i}] \tag{3.50}$$
$$Z_{i+1} = Z_i - d_i \arctan(2^{-i})] \tag{3.51}$$

where

$$d_i = \begin{cases} -1, & \text{if } y_i < 0. \\ +1, & \text{otherwise.} \end{cases} \tag{3.52}$$

The resulting equations after a certain number of iterations will be:

$$X_n = A_n \times \sqrt{X_0^2 + Y_0^2} \tag{3.53}$$
$$Y_n = 0 \tag{3.54}$$
$$Z_n = Z_0 + \tan^{-1}\left(\frac{Y_0}{X_0}\right) \tag{3.55}$$

Additionally, if the input Z_0 is initialised to zero, then $Z_n = \tan^{-1}(\frac{Y_0}{X_0})$, that is, $\arctan$ function can be obtained in Z_n after a fixed number of iterations.

The equations that are discussed until this point are limited to the rotations in $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$. In other words, the maximum value for rotation angle can be $\theta \approx 99.88^\circ \approx 1.7433$ *radian*. Thus, it requires pre-rotation or correction iteration:

$$X_0' = -dY_0 \tag{3.56}$$
$$Y_0' = dX_0 \tag{3.57}$$
$$Z_0' = Z_0 - d\frac{\pi}{2} \tag{3.58}$$

where

$$d_i = \begin{cases} -1, & \text{if } y_i < 0. \\ +1, & \text{otherwise.} \end{cases} \tag{3.59}$$

It is worth to note that there is no growth or gain for initial rotation.

Based on the formulas 3.10-3.15 and 3.16-3.17, the algorithm was then improved to only update affected elements. The algorithm is written in MATLAB. For the CORDIC, MATLAB's built-in functions are used in the code. Explanations of each formula are provided as commented lines in the code.

```
%a function prints the final results for eigenvalue matrix
%eigenvector matrix and iteration number
function [A_out, V_out, count ] = cyclicjacobi2( A )
count = 0;
%n is the size of the matrix
n = size(A);
%initial eigenvector matrix is identity matrix
V = eye(n);
%initial indices - fixed for p and q
p = 1; q = 2;
%assigning to the outputs
A_out = A;
V_out = V;
%until the norm - error function is small enough, do:
while off_A( A ) > 0.00001
    % x is numerator, y is denominator for arctan(y/x)
    x = A(q,q)-A(p,p);
    y = 2*A(p,q);
    %built-in cordic function to calculate arctan
    theta = cordicatan2(y, x);
    %the value of angle equals half arctan
    angle = theta /2;
    %built-in cordic function for generating sin and cos
    [s,c] = cordicsincos(angle, 32);
    %update formulas for diagonal App, Aqq, and off-diag Apq
    A_out(p,p)=(c^2)*A(p,p)-2*c*s*A(p,q)+(s^2)*A(q,q);
    A_out(q,q)=(s^2)*A(p,p)+2*c*s*A(p,q)+(c^2)*A(q,q);
    A_out(p,q)=(c^2-s^2)*A(p,q)+c*s*(A(p,p)-A(q,q));
    A_out(q,p)=(c^2-s^2)*A(p,q)+c*s*(A(p,p)-A(q,q));
    %for all elements in columns p and q:
    for i = 1:n
        %eigenvectors ordered in columns
        V_out(i,p)=c*V(i,p)-s*V(i,q);
        V_out(i,q)=s*V(i,p)+c*V(i,q);
        %updates for eigenvalues
```

```
        if(i ~= p && i ~= q)
            A_out(p,i)=c*A(p,i)-s*A(q,i);
            A_out(q,i)=s*A(p,i)+c*A(q,i);
            A_out(i,p)=c*A(i,p)-s*A(i,q);
            A_out(i,q)=s*A(i,p)+c*A(i,q);
        end
    end
    A = A_out;
    V = V_out;
    %choosing next indices after each iteration
    [p,q] = next_pq( A,p,q );
    %iteration increment
    count = count + 1;
    %print current eigenvalue and eigenvector matrices
    %for each iteration
    disp('current eigenvalue matrix is = '), disp(A_out);
    disp('current eigenvector matrix is = '), disp(V_out);
end
end
```

Chapter 4

HARDWARE IMPLEMENTATION

This chapter presents the steps taken in the hardware implementation of Jacobi method starting from a simple implementation with the goal to make it work accurately to an improved and more efficient design implementation with a number of optimisation techniques implemented. Each design is discussed in detail and compared with previous approach. Developed MATLAB code and block figures are given throughout the explanation of each submodule and VHDL code for the final design with all the modules are given in the Appendix. A random 3×3 example matrix used in most examples for simpler explanations of Jacobi modules:

$$A = \begin{bmatrix} 4000 & 2000 & 3000 \\ 2000 & 10000 & 2000 \\ 2000 & 3000 & 6000 \end{bmatrix}$$

4.1 Cyclic Jacobi Implementation on VHDL

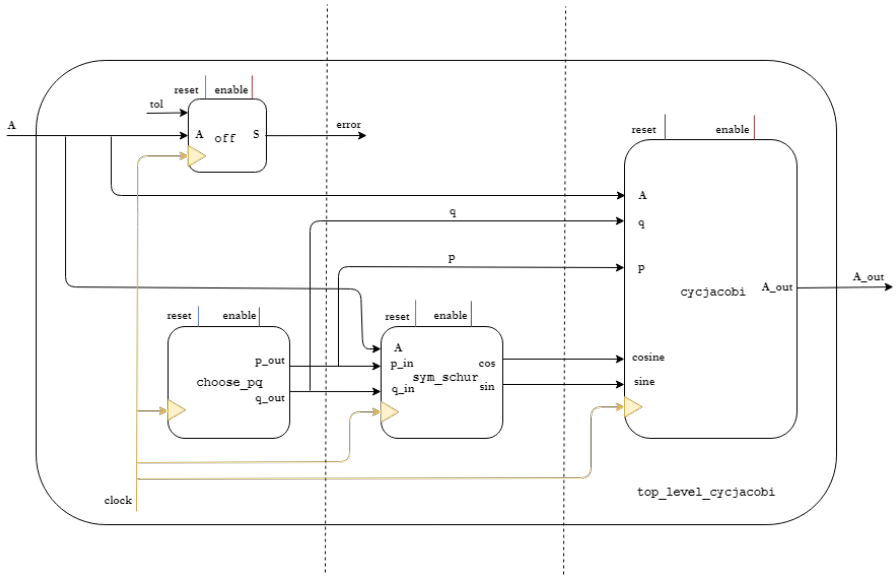
The block diagram for the initial implementation of the design is shown in Figure 4.1. The block diagram is updated through the report when there are significant new changes in the architecture.

The design consists of several modules. Their objectives are explained below:

The module off.vhd:

This module calculates the stopping criterion for the convergence. It computes the error function in Eq. 3.1. A problem with the Jacobi's algorithm is that it can get stuck in an infinite loop if all the off-diagonal entries are expected to become exactly zero. That is why, a margin of error has to be set. It is defined as a stopping point for when the matrix is close enough to being diagonal. The stopping rule used is called Frobenius norm which

Figure 4.1 Block diagram for a simple Jacobi Method. This is the initial working design that calculates only eigenvalues. Because sym-schur algorithm uses **real_math** library in order to do **sqrt** operations, this design is not synthesizable. Indices p and q are chosen by cyclic-by-row order.



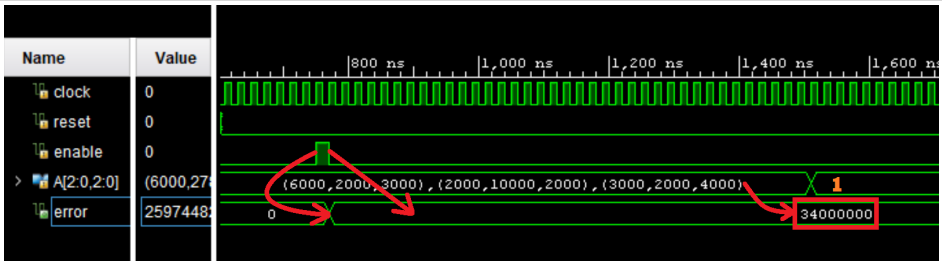
is given in Chapter 3. Its working principle is given as follows: when the program reaches a point where the square of all the off-diagonal entries added up is less than the given tolerance, it will stop and publish the final output. The signal named *tolerance* can be either given as a fixed input or as a parameter that can be changed by the user. In the initial design, this number is given as an input which can be set to any desired precision of the user wants their zero to be. The initial module takes the input matrix A and a predefined *tol* - tolerance value and calculates the result of the Frobenius norm formula named as *S_off*. Figure 4.2 shows an example with a random 3×3 matrix input. For the sake of simplicity, the square root function is eliminated and the error function is calculated as the sums of squares of off-diagonal elements as it does not affect the flow of the program.

The code for the error function written in MATLAB is as below:

```
function [ sqrt_S ] = off_A( A )
n = size(A,1);
S = 0.0;
for i=1:n
    for j=1:n
        if (j ~= i)
        S=S+A(i,j)^2;
        end
    end
end
```

```
sqrt_S = sqrt(S);
end
```

Figure 4.2 Simulation waveforms present the change in error value. It is implemented in combinatorial logic. When enable is set, the error function for the current matrix is calculated and its value is assigned in the following cycle.



Although this stopping criterion results in high accuracy, from the efficiency point of view it has pitfalls. As discussed, it needs to process the whole upper-diagonal matrix each time it is calculated. This creates problem in resource usage and input access for huge matrices. After having a working design, new ways are researched. One of the methods found leads to a similar consequences which has the inequality as follows:

$$\text{if} \quad |a_{ij}| \leq tol \cdot \underset{kl}{max} |a_{kl}| \,, \text{set} \quad a_{ij} = 0 \tag{4.1}$$

As seen, this is not much different than the previous method although it has been used in many related works before. (Gonzales, 2015) uses this method in a slightly different way. The stopping criterion used in their work consists of comparing the largest off-diagonal element to a provided value. However, (Veselic, 1990), (Stewart, 1983) have suggested a new method, also (Demmel, 1992) has given an explanation of its benefits. This method is as below:

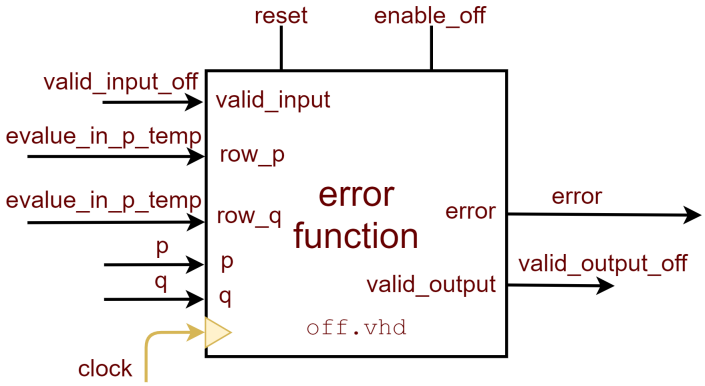
$$\text{if} \quad |a_{ij}| \leq tol \cdot (a_{ii}a_{jj})^{\frac{1}{2}} , \text{set} \quad a_{ij} = 0 \tag{4.2}$$

This method needs only two particular diagonal elements. The functionality of the method is first tested in MATLAB before implementing on hardware. The while loop in the MATLAB code given in Section 3.3 is replaced by the following loop in order to obtain the expected functionality:

```
while A(p,q)^2 > (0.00001)^2 *(A(p,p)*A(q,q))
...
end
```

As squaring both sides of an inequality does not affect the result, square root calculation is avoided by squaring both sides. Although this method looks promising for Classical Jacobi, it can be proved to be not efficient at all for Cyclic Jacobi. Because Cyclic Jacobi selects elements in a predefined order, while this error function requires that the largest

Figure 4.3 The designed block diagram for the calculation of the error function. The figure illustrates the inputs and outputs with the names as inside the module as well as in the top level.



A_{ij} is picked. Therefore, it can be assumed that if the first off-diagonal element is zero, the stopping criterion will terminate the calculation before it is done. So this formula does not work well with Cyclic Jacobi.

One of the main novelties in this thesis is a new method for calculating the error function in a more efficient way. This could be the simplest method, whereas, to the best of author's knowledge, it has not been discovered in previous works. The method calculates the error function as in Eq. 3.1 but using only the off-diagonal elements in two rows: p and q (it can also be the same columns or the sum of rows and columns) instead of the off-diagonal elements in whole matrix. The *tolerance* value is included to the design as a variable parameter for flexibility. The block diagram showing inputs and outputs of the final module is given in Fig. 4.3.

This can be proved to be correct by the fact that in each iteration altered off-diagonal elements are only in the rows and columns under calculation. The assumption is first verified to work as expected in MATLAB as below:

```
function [ stop_crit ] = off_new( A, p, q )
x = A(p,:);
n = size(x,2);
y = A(q,:);
m = size(y,2);
S = 0.0;
for i=1:n
    for j=1:m
        if (j ~= i)
        S=S+A(i,j)^2;
        end
    end
end
```

```
stop_crit = sqrt(S);
end
```

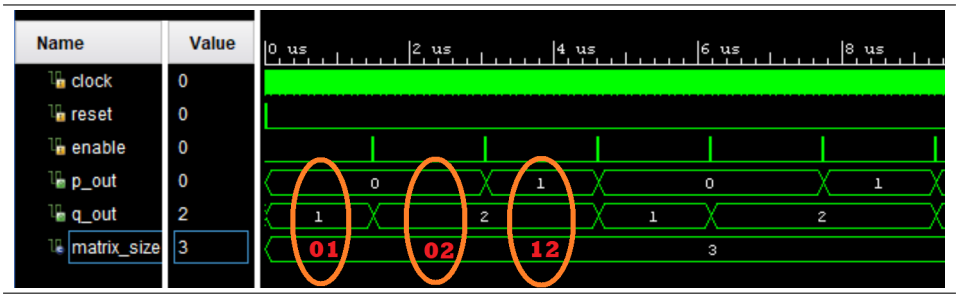
This method decreases the input-output overhead significantly along with the computation overhead. It can also be noted that, there is no need to pull additional elements as the two rows are already pulled for using on update formulas. The same rows are used to calculate error function, too.

However, the resource usage is still an issue for big matrices, e.g. 100×100. Considering that, in Zedboard FPGA where the number of available resources is small and **off** submodule uses a great deal of hardware in each iteration, it should be pipelined. The submodule is pipelined and the number or computational slices is made parameterizable. The modified submodule consists of two states, namely: **st_busy** and **st_idle**. This allows the module to use user-defined number of computational slices for the calculations of the MACs (Multiply-Accumulates). Computational slices, ideally, synthesize into DSP blocks. Thus, increased speed of the computation and reuse of DSP blocks are the effective outcomes from this pipelining. More information is given and the compared synthesis results are reported in **Results** section.

The module choose_pq.vhd:

This module is made for cyclic-by-row Jacobi algorithm. The basic idea here is that in each clock cycle, the indices p and q are updated and the next element of the matrix is chosen. When the calculation of all the elements in the row finished, indices are updated to start from the upcoming upper-diagonal element. Fig. 4.4 presents its functionality on an example.

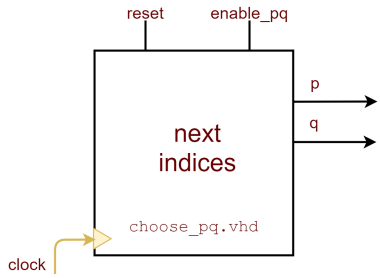
Figure 4.4 Simulation waveforms show an example of index transition in a 3×3 matrix. Here, two buffers are used. *p_out* is the row, *q_out* is the column number. Each time the submodule is enabled, next element is taken. The order of the indices for the elements to be calculated is circled and emphasised in the figure.



The working fashion was explained in the Chapter 3 in more detail. The implemented block diagram in hardware for this module is provided in Fig. 4.5. The MATLAB function for choosing (p,q) pair in cyclic mode is written:

```
function [p,q] = next_pq( A,p,q )
n=size(A,1);
```

Figure 4.5 The block diagram for **choose_pq** module is shown with its inputs and outputs. The module is triggered in the top level by the *enable_pq* signal.




```
    if(p == n-1 && q == n)
        p = 1;
        q = 2;
    else
        if(q < n)
            q = q+1;
        else
            if(q == n)
                p = p+1;
                q = p+1;
            end
        end
    end
end
```

The modules for calculating trigonometry:

The initial implementation of this submodule is symmetric Schur algorithm, the module name is **sym_schur.vhd**. The derivation of the formulas for Symmetric Schur Decomposition (SSD) are given in Eq. 3.4 - Eq. 3.9. Since this module is intended to be replaced by CORDIC for the calculation of trigonometric functions which Jacobi requires, it is made only to begin with, that is why the efficiency is not taken into account. As formulas for SSD suggest, this method requires square root calculations, thus in the design *sqrt* function of the **math_real** library is used. However, this library is not always synthesis-friendly. MATLAB code for testing has been written and the implemented VHDL code is given in Appendix:

```
function [ c,s ] = sym_schur( A, p,q )
if(A(p,q) ~= 0)
    tau = (A(q,q)-A(p,p))/(2*A(p,q));
    if (tau >= 0)
```

```
        t = -tau+(sqrt(tau^2+1));
    else
        t = -tau-(sqrt(tau^2+1));
    end
    c = 1/sqrt(1 + t^2);
    s = t*c;
else
    c = 1;
    s = 0;
end
```

The module takes the indices p and q from the previous module for choosing the indices which are going to be rotated and the input matrix, consequently produces cosine and sine trigonometric functions.

4.1.1 Use of CORDIC to accomplish a more efficient Jacobi

First of all, the question is: Why to use CORDIC in hardware implementation? The recapitulation of the reasons provided below answers this question:

- Commonly used for FPGAs
- Complexity comparable to division
- Execution time comparable to division operation
- Low number of gates required, thus, cost effective
- Requires only shifter, adder/subtractor and a small lookup table all of which are considered to be cheap components

Although it is complex to grasp the idea of CORDIC as well as to implement, it makes the computations much faster and more efficient. In the proposed design of main CORDIC module, obviously, the goal is the same as in symmetric Schur algorithm. It is considered to use the CORDIC modules as calculators, then based on the plugged coordinates, the CORDIC with the vectoring mode must generate arctangent of the input values. Starting with the same formulas given in equations 3.4, then 3.5, 3.7 which are intended to make the off-diagonal elements zero and using the condition 3.6 the relation between the arctangent function and the rotation angle can be obtained by changing the direction of the derivations slightly:

$$\tau = \frac{a_{qq} - a_{pp}}{2a_{pq}} = \frac{c^2 - s^2}{2cs} = \frac{\cos 2\theta}{\sin 2\theta} = \frac{1}{\tan 2\theta} \tag{4.3}$$

$$\tan 2\theta = \frac{2a_{pq}}{a_{qq} - a_{pp}} \tag{4.4}$$

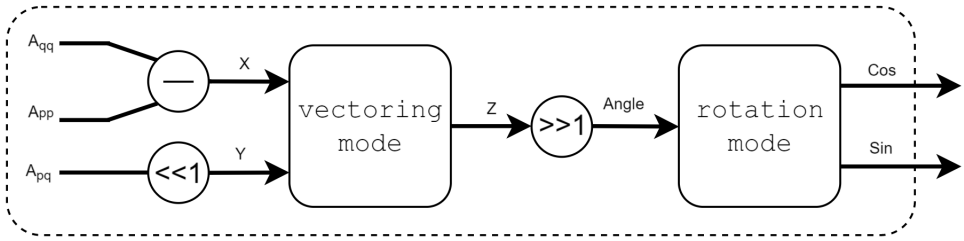
$$2\theta = \tan^{-1}\left(\frac{2a_{pq}}{a_{qq} - a_{pp}}\right) = \arctan\left(\frac{2a_{pq}}{a_{qq} - a_{pp}}\right) \tag{4.5}$$

$$\theta = \frac{1}{2}\arctan\left(\frac{2a_{pq}}{a_{qq} - a_{pp}}\right) \tag{4.6}$$

Eq. 4.6 which shows the relationship between the angle and $\arctan$ is used as a transition between the vectoring mode and the rotation mode of CORDIC. It can be seen that the numerator corresponds to $Y = 2 * a_{pq}$ and denominator is $X = a_{qq} - a_{pp}$ where X and Y are the inputs to the vectoring mode block. The output *arctan* is divided by two, in other words, shifted one bit to right to acquire the angle θ. The obtained angle is plugged into the rotation mode and used in the calculation of $\sin(\theta)$ and $\cos(\theta)$ functions.

The proposed CORDIC design for this thesis is presented in Figure 4.6. The CORDIC modules for *arctan* and *sin-cos* used in this work is made by (Thibedeau, 2014) and freely available for use with the permission. The testbenches have been created for both modules. After careful testing, these modules are verified to be functioning in expected way, both separately and together. The results in the hardware are the same as the results with MATLAB built-in functions: **theta = cordicatan2(y,x,niters)** and **[y, x] = cordicsincos(theta,niters)**. As a next step, **sym_schur** module is substituted by CORDIC modules.

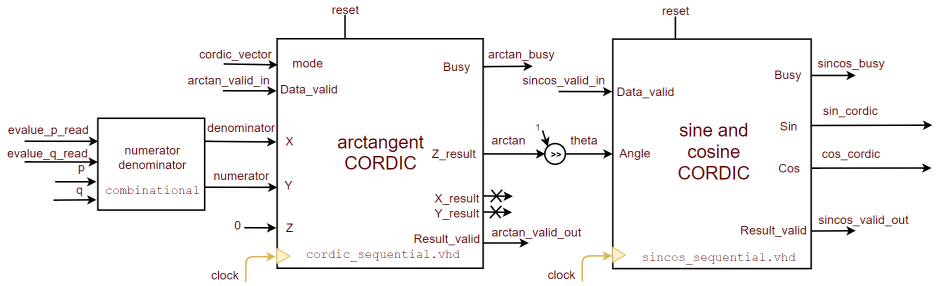
Figure 4.6 Proposed CORDIC design.



The block diagram of the CORDIC architecture used in VHDL is given in Fig. 4.7. The first module called **cordic_sequential** is for the vectoring mode and the rotation mode is called **sincos_sequential**. These iterative modules are designed to use minimal hardware. The output signal *Z_result* is called *arctan* in the top-level and after the division by 2, the result of it is assigned to an intermediate signal called *theta*. In turn, the input signal to the **sincos module** called *Angle* takes takes the value of *theta* which is the rotation angle needed.

Additionally, the scaling methods used in CORDIC modules must be taken into account. It is worth to note that the provided unit for the angle parameter should be in brads (binary-radians): $2 * \pi$ radians = 2^{size} brads for "size" number of bits, 32 bits in this case. Hence, to see the exact angle generated, the output should be "de-scaled". $\sin(\theta)$ and $\cos(\theta)$ are scaled by multiplying by $2^{fractional\_bits}$. The fraction bits are parameterizable that can be defined by the user. The number of fractional bits should be as high as possible. The more precision, the higher resolution. In order to see the actual values of the trigonometric functions, the outputs can be "de-scaled" by dividing by $2^{fractional\_bits}$.

Figure 4.7 CORDIC blocks used in hardware.



The module cycjacobi.vhd:

In this module, the Eqs. 3.10 - 3.15 have been implemented in VHDL. The reasons why this method looks more promising than matrix multiplication have been discussed in Chapter 3. The module takes the input matrix, the indices from **choose_pq** and cosine and sine from CORDIC (initially from **sym_schur**), thus forms the output matrix with the new elements for one iteration. The method is equivalent to performing a matrix multiplication, but more efficient in hardware. The formulas are applied to the inputs and the outcomes are assigned to temporary signals. The temporary signal is continuously assigned to the output matrix. As *cos* and *sin* are scaled in top-level module before input into this module, they need to be scaled back. In hardware, in the equations for A_{pp} and A_{qq}, the overflow must be taken into account carefully while "de-scaling". The reason is that these equations contain multiplication of three numbers where the bit size can not fit to the size of integer data type. The corresponding formulas are written in MATLAB as:

```
A_out(p,p)=(c^2)*A(p,p)-2*c*s*A(p,q)+(s^2)*A(q,q);
A_out(q,q)=(s^2)*A(p,p)+2*c*s*A(p,q)+(c^2)*A(q,q);
A_out(p,q)=(c^2-s^2)*A(p,q)+c*s*(A(p,p)-A(q,q));
A_out(q,p)=(c^2-s^2)*A(p,q)+c*s*(A(p,p)-A(q,q));
for i = 1:n
    if(i ~= p && i ~= q)
        A_out(p,i)=c*A(p,i)-s*A(q,i);
        A_out(q,i)=s*A(p,i)+c*A(q,i);
        A_out(i,p)=c*A(i,p)-s*A(i,q);
        A_out(i,q)=s*A(i,p)+c*A(i,q);
    end
end
```

Fig. 4.8 indicates the working behaviour of the **cycjacobi** module and Fig. 4.9 shows the resulting eigenvalue matrix. A 3×3 matrix is given as an example. Initially, when reset is low, *A_in* input matrix is assigned to an intermediate signal called *A_out_temp* which is a temporary signal keeping the value of the output. Then the formulas make modifications on the value of output matrix. In the top level, the output matrix from the previous iteration *A_out* is stored back to *A_in*. As new input matrix *A_in* is updated, it can be used for the

Figure 4.8 The provided simulation waveforms show an example of a circulation of eigenvalue calculation. The general flow is: *A_out_temp* signal takes the value of *A_in*, applies the updates formulas on it. *A_out_temp* is assigned to output signal *A_out* in combinational logic. Explicitly, in the top level, *A_out* takes the value of *A_in* for the next iteration.

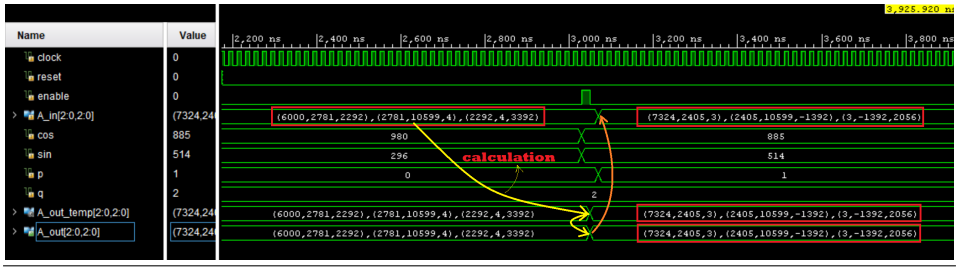


Figure 4.9 The waves provide an example of the final result for a random matrix. The result is compared to be the same with MATLAB results of the same matrix. The inputs e.g trigonometric functions and indices are used to do the calculations.

following iteration.

The module is modified after the Block RAM is set up to read and write the elements. Instead of all the elements in the matrix, only the concerning two rows are read from the BRAM and updated results are written back to the corresponding rows and columns. For this functionality, *A_in* signal is substituted with *evalue_p_in* and *evalue_q_in* for reading the columns p and q. Similarly, *A_out* is replaced by *evalue_p_out* and *evalue_q_out*. Hence, the update formulas are performed on two particular rows in each iteration.

The module eigvector.vhd:

After getting sufficient accuracy in eigenvalues by using the similar method of matrix updating, new formulas for eigenvectors have been made as discussed in Chapter 3. These formulas for eigenvector matrix calculation have been tested in MATLAB as in Section 3.2.2. The equations 3.16 and 3.17 are added to MATLAB code:

```
for i = 1:n
    V_out(i,p)=c*V(i,p)-s*V(i,q);
    V_out(i,q)=s*V(i,p)+c*V(i,q);
    ...
```

```
end
```

After that, they are implemented in hardware in a very similar way. A separate module is created for eigenvector computation and it has been added to the design after testing with a separate testbench. As shown in Fig. 4.10, the initial matrix is identity matrix which is updated to the product of the corresponding rotation matrices in each iteration. The resulting eigenvector matrix is presented in Fig. 4.11.

Figure 4.10 The simulation result given is for calculation of the eigenvectors for a 3×3 matrix. Initial identity matrix and the first rotation are highlighted in the figure. Identity matrix (**I**) is scaled and then updated to the first rotation matrix (**R1**) when *enable* is set. For the calculation, input signals for *cos* and *sin* and indices are used as in the formulas 3.16 and 3.17.

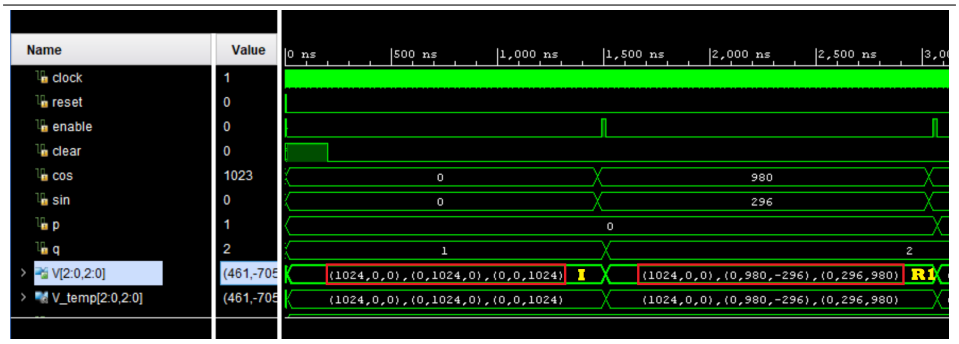
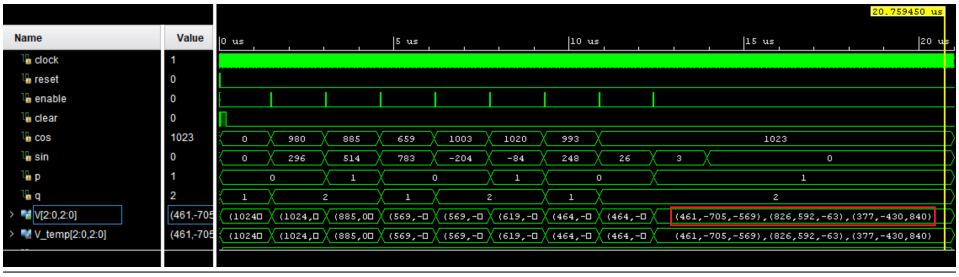


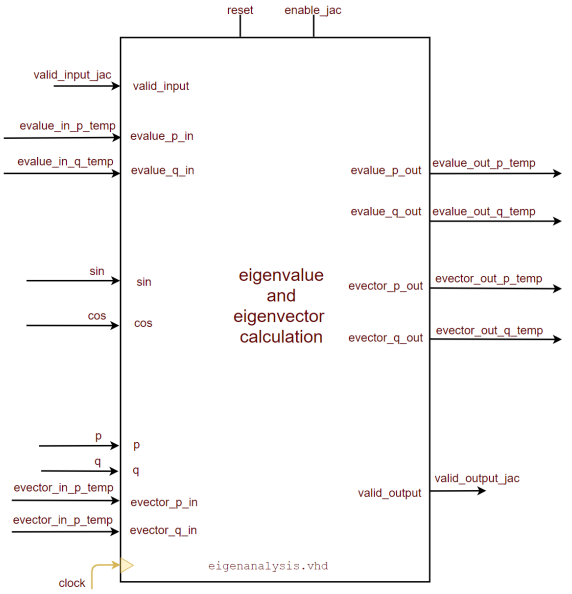
Figure 4.11 Simulation waves for a full calculation of eigenvectors. The resulting eigenvector matrix is highlighted on the figure and is verified with MATLAB results to be as expected.



The same changes as eigenvalue calculations have been made to eigenvector calculations as well. Instead of performing the calculations on whole matrix, they are updating only two concerning rows in each iteration. Additionally, the Identity matrix is scaled by 10000 for simplicity instead of 1024.

After functionality was confirmed on the individual submodules, they were integrated into a single module called **eigenanalysis**.

Figure 4.12 The implemented hardware design for eigenanalysis.



The module eigenanalysis.vhd:

This module performs the eigenvalue and eigenvector calculations. As it is the main part of the Jacobi, it requires so much hardware. In order to control this, the module is pipelined. Two states are created: **st_busy** and **st_idle**. In reset, state is in **st_idle**. In **st_busy**, update formulas are performed on two input rows for the eigenvalue and two rows for the eigenvectors. Two separate BRAMs are used to store the eigenvalue matrix and eigenvector matrix. Initially, identity matrix is created in one of the BRAMs, the other BRAM contains the covariance matrix. The corresponding two rows from each BRAM are read to the **eigenanalysis** module in each iteration. They are updated, then they are written back to BRAM. Pipelining also allows to define the number of the computational slices that are intended to be used. Similarly to **off** module, a new variable is included: **n**. Hence, the number of computational slices are to be used is highly parameterizable. The block diagram showing the inputs and outputs of the module is presented in Fig. 4.12. Besides, Fig. 4.14 shows the connections of the module with other modules in the design. The synthesis results for different values of **n** are compared in **Results** section.

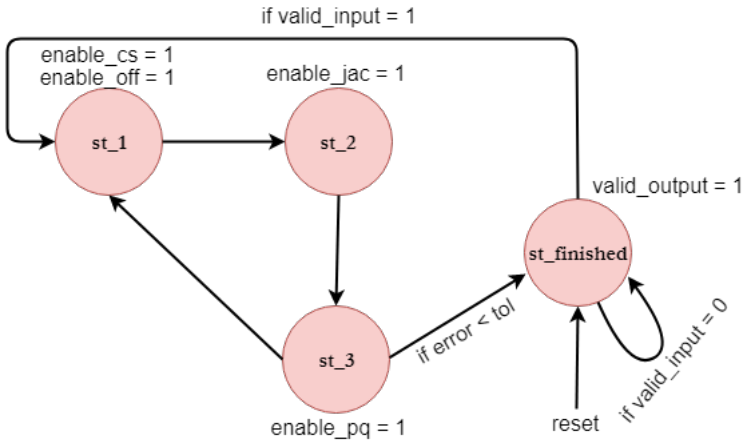
4.1.2 Initial design

Using sym-Schur:

The top level module is used to instantiate and declare the connections among all four modules. Beside encapsulating of all module connections, it also includes a state machine to control the transitions by the use of enables signals created for each module. The state

machine illustration is given in Figure 4.13.

Figure 4.13 The implemented state machine for cyclic-by-row Jacobi in top level of the design.



The state is initially in the reset state which is called **st_finished**. When there is data coming in, the *valid_input* signal is triggered and gets the new data *A_in* and transition to the state **st_1** happens. In this state, the *enable_off* and *enable_cs* signals trigger the computation of the error function and cosine and sine functions. The second transition occurs to **st_2** where **cycjacobi** module is enabled by *enable_jac* and, in turn, Jacobi formulas update the matrix for the current iteration. Finally, **st_3** chooses the next indices for the next iteration. The code snippet of these continuous assignments from the design is as follows:

```
enable_pq <= '1' when (state = st_3) else '0';
enable_cs <= '1' when (state = st_1) else '0';
enable_jac <= '1' when (state = st_2) else '0';
enable_off <= '1' when (state = st_1) else '0';
```

4.1.3 Final design

Using CORDIC

The block diagram for the proposed design cyclic-by-row Jacobi after including CORDIC is shown in Figure 4.14. The top-level module of the design is modified after substituting CORDIC in place of symmetric-Schur. The components instantiation of the Symmetric Schur transformation have been replaced with CORDIC. The flow of the state machine have been altered as in Fig.4.15. The state transitions are indicated with directional arrows. The conditions that make the transitions happen are noted above the arrows.

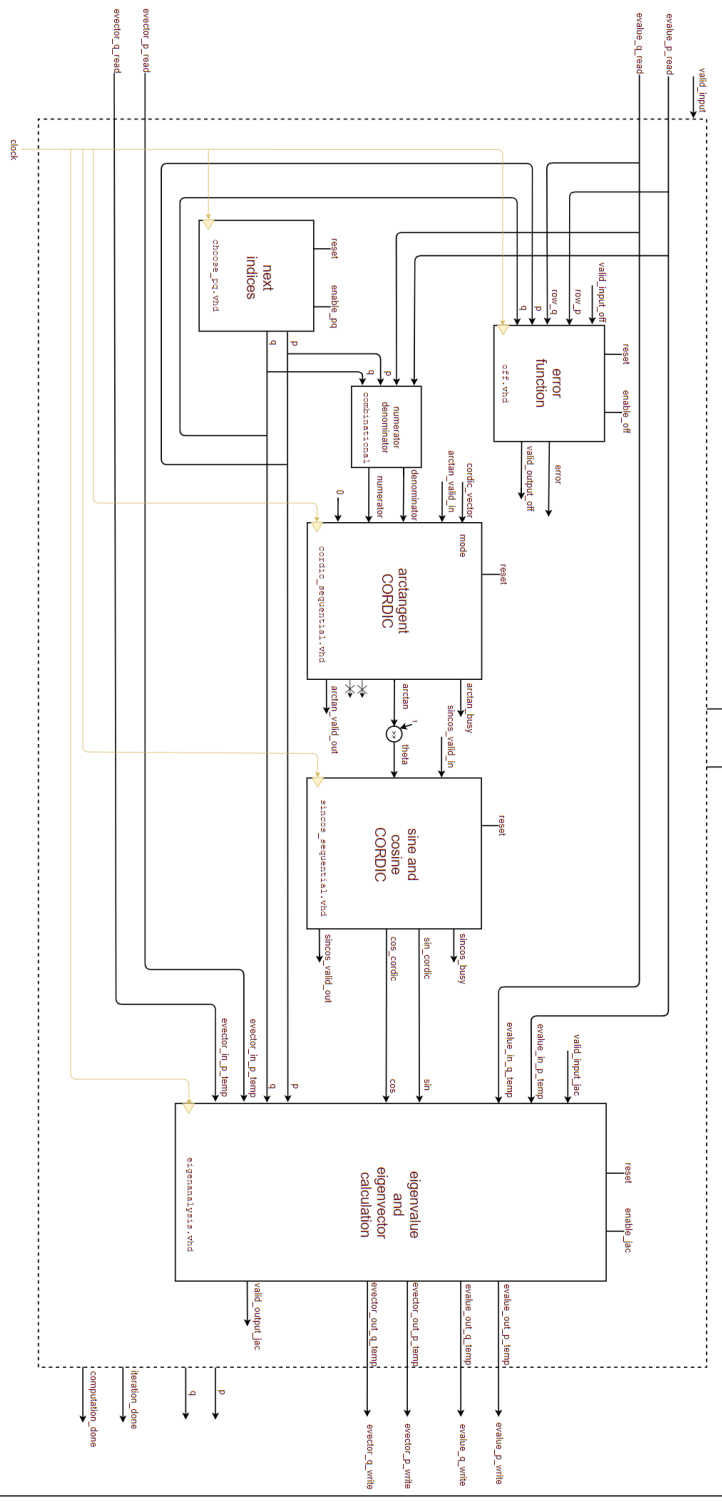
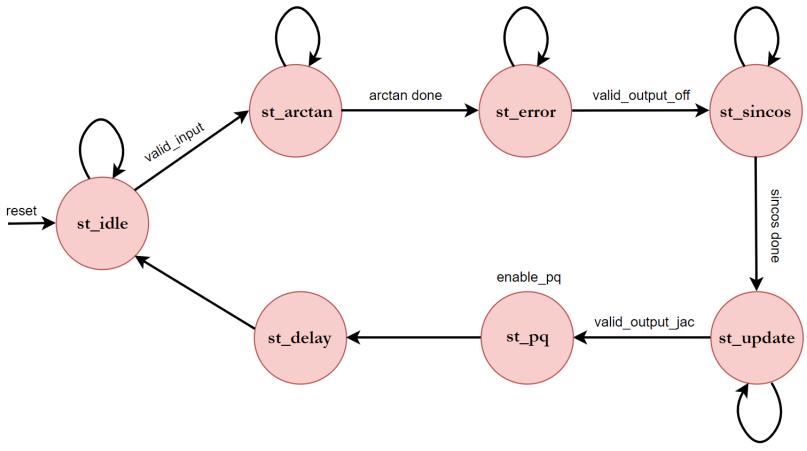


Figure 4.14: The illustration shows the proposed architecture for the Cyclic Jacobi design using CORDIC.

states	operations
st_idle	State is in idle in the beginning until there is a valid input. *Read* data signals from outside world (BRAM) are assigned to the intermediate signals of the Jacobi module.
st_arctan	An iteration starts here. CORDIC performs vectoring mode and produces *arctan* value. The rotation angle θ is calculated in a combinatorial logic.
st_error	The submodule is triggered by *valid_input_off*. Error function is calculated. When there is a valid output: *valid_output_off*, the state changes. *enable_off* stays asserted until one iteration finishes. This state also provides the required delay between two CORDIC modules.
st_sincos	CORDIC performs rotation mode. $\sin\theta$ and $\cos\theta$ are calculated. The next state is triggered by *valid_input_jac*.
st_update	Formulas for eigenvalues and eigenvectors are performed and the input rows are updated. When *valid_output_jac* is asserted, the state transitions.
st_pq	An iteration finishes here. New indices are chosen for the next iteration by asserting *enable_pq*. Error function is compared with the given tolerance. If error is small enough, computation finishes here.
st_delay	Updated intermediate signals are assigned to outputs for *write* operation back to BRAM.

Table 4.1: Cyclic Jacobi state machine with explanation of the functionality of each state.

Figure 4.15 The illustration shows the state machine chart presenting the state transitions of cyclic-by-row Jacobi algorithm using CORDIC. This state machine is implemented in the top level, in the module called **top_level_cycjacobi.vhd**.



There are 7 states in the design. Their explanations are given in detail in the table 4.1. First of all, the state is in **st_idle** while reset. Here, the inputs are assigned to temporary variables. In the final design, the inputs are the corresponding rows for the eigenvalue matrix and columns for the eigenvector matrix calculations. The condition is checked if the CORDIC module has not started yet, that means the calculation has not started either, so the signals *computation_done* and *iteration_done* are reset and the state transitions to the beginning of the first iteration which is state **st_arctan**. Here, CORDIC module calculates the **arctan** and combinational assignment calculates the rotation angle out of it. As discussed before, the output *arctan* from the vectoring mode is divided by two in a combinatorial assignment and input to the rotation mode. Besides, the error function module is triggered using its matching valid signal. Then the state moves to **st_error**. The state **st_error** is put between the CORDIC modules, which allows the calculation of arctangent by creating some delay between the CORDIC modules. In this state, error function is computed so the valid signal can be de-asserted. When it finishes, **st_sincos** is the next. This state allows the CORDIC calculate the *sin* and *cos* functions. Before going to the next state, valid signal for the eigenpair computation module is asserted. In state **st_update**, the update formulas are performed for both eigenvalue and eigenvectors. When the outputs are ready, the corresponding module asserts *valid_output* signal. Finally, new indices are chosen for the next iteration in state **st_pq**. The error function is also compared with the given tolerance value here. If the error function is small enough, then *computation_done* signal is assigned and the computation is completely finished. State transitions to **st_delay**. This state is either in the end of one iteration or the whole computation, hence the altered elements are assigned to the output signals in this state. Moreover, *iteration_done* is set. In the end, state transits back to **st_idle** and the process restarts all over for the next iteration or the next calculation.

Chapter 5

BLOCK RAM AS INTERMEDIATE STORAGE

Writing to and Reading from the Block RAM

One of the main challenges in hardware implementation of matrix operations is streaming the data. As the real symmetric matrix is quite huge, it needs to be stored in memory and filled into and pulled from the memory to perform operations on it. Firstly, the output matrix from the covariance module needs to be stored in memory so that Jacobi design can pull concerned rows and columns for the calculations. In order to read the output matrix from the covariance design, the design made in (Karimova (2017)) needs to be modified so that the rows and columns can be accessible in the outputs of the design. Because the rows and columns are to used in the address mapping for the memory. The block diagram for the proposed architecture of PCA is given in Fig. 5.1. It shows all the main connections between covariance module and Jacobi through two BRAMs.

5.1 Block RAM design

A Single-port Block RAM is designed. Aids used are Xilinx User Guides: Libraries (2017), 7series (2016), BRAM (2011) and Designing with IP (2018). There is one common enable signal for write and read called *we*. When *we* is asserted, BRAM performs write operation, otherwise, it always reads. Both operations use the same clock. The block RAM is in "read-first" mode. The RAM is defined to be an array of *std_logic_vector* data type. Size, width of BRAM and width of address signals are generic constants that are parameterisable. For simplicity, small numbers and small matrix are used for explanation.

5.1.1 Write operation from covariance design to memory

First of all, two new output signals for tracking row and column values are added to the covariance design. The size of *row* and *column* signals depend on how many buffers are

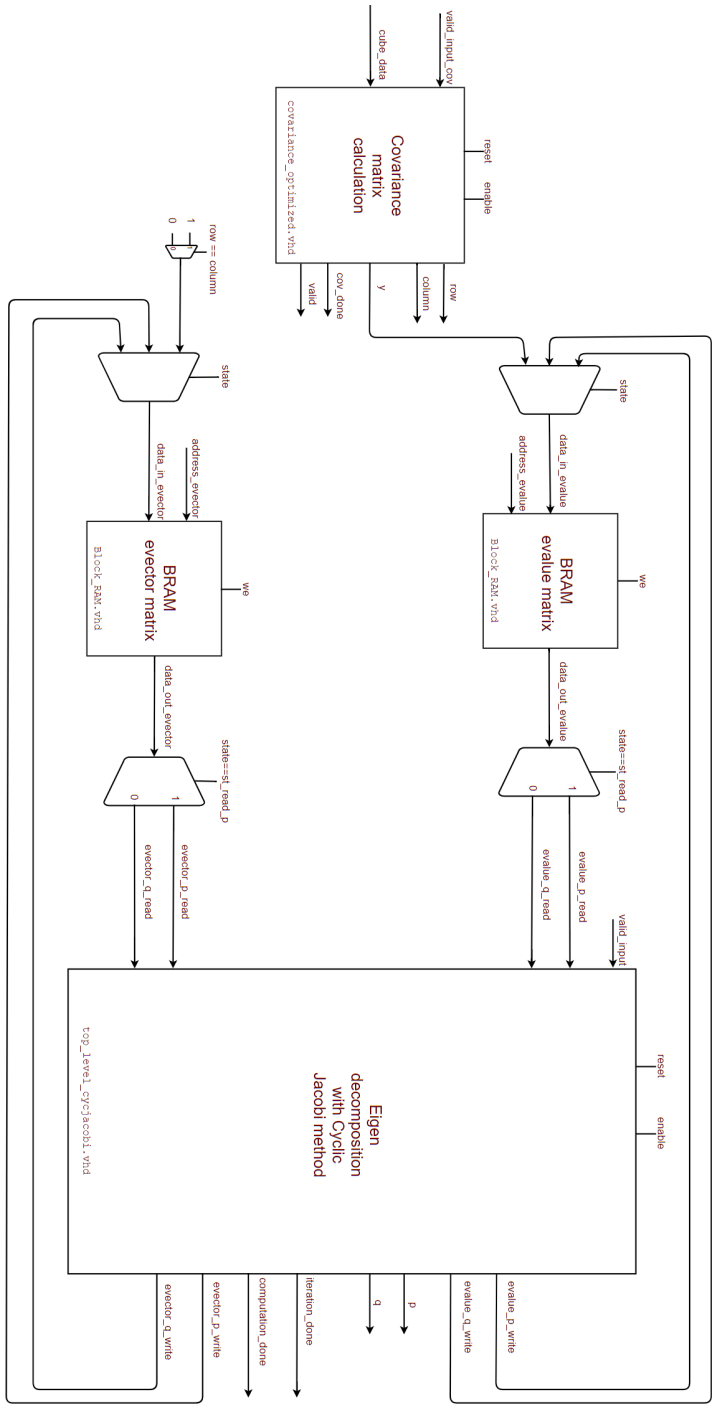
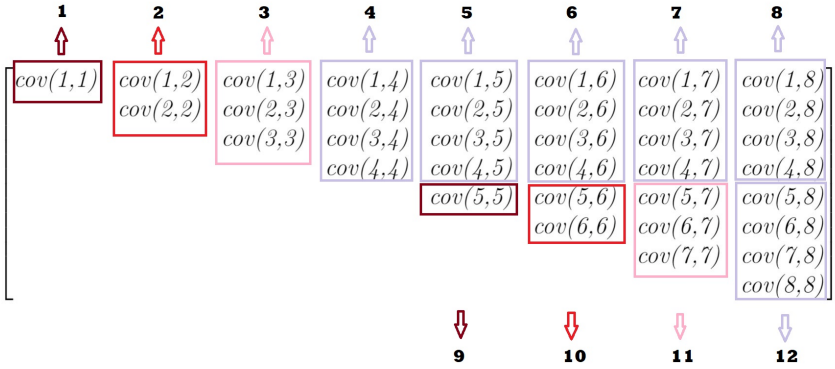


Figure 5.1: The proposed architecture for the top level design with Block RAM.

used in the design. As the covariance is designed in a special way that it is generic for any application and even the number of buffers can be specified by the user. In each step the number of output elements depends on the number of parallel buffers used. Thus, size of these signals must be as long as the buffers used in the design, e.g. row and column numbers for each element of the output matrix. The objective is to store the matrix in the BRAM in a column-by-column fashion. For instance, for a 8×8 matrix with 4 buffers, the order of reading the outputs are as in Fig. 5.2. Fig. 5.3 shows how the rows are written to BRAM with another example on 5×5 matrix.

Figure 5.2 The figure shows the order of reading covariance outputs.



Additionally, the upper-diagonal elements are duplicated to corresponding lower-diagonal. This functionality is achieved by writing the same input to two different addresses in two separate states. For the address mappings the following formulas are derived:

$$u = row * m + column. \tag{5.1}$$

$$l = column * m + row. \tag{5.2}$$

where *m* is the matrix size, *u* and *l* are the address values for upper and lower diagonals respectively.

The output control signal *valid* from the covariance design announces that the output is ready and it is reset to zero just after one cycle. Its value needs to be captured and stored while it is high. Therefore, a *valid_latched* signal is included. It is worth to note that, there is no extra time needed to write the covariance outputs to the BRAM. As soon as the covariance elements are calculated, they are directly written to the corresponding addresses in the BRAM. This is achieved by assigning the corresponding buffer index of the *valid_latched* signal to the *we* - write enable signal of the BRAM. As the valid output stays only for one clock cycle, another signal is needed to store its value for the use by the BRAM. So, *y_latched* signal is created. The same situation holds for row and column signals. The given simulation waves in Fig. 5.4 illustrates this functionality clearly. Latched signal transitions are highlighted with arrows. It can also be viewed from this figure that latched element (1240 in decimal) is stored in BRAM in the address calculated (0 in this case).

Figure 5.3 The figure shows the order of the write process of the covariance matrix to the BRAM on another example.

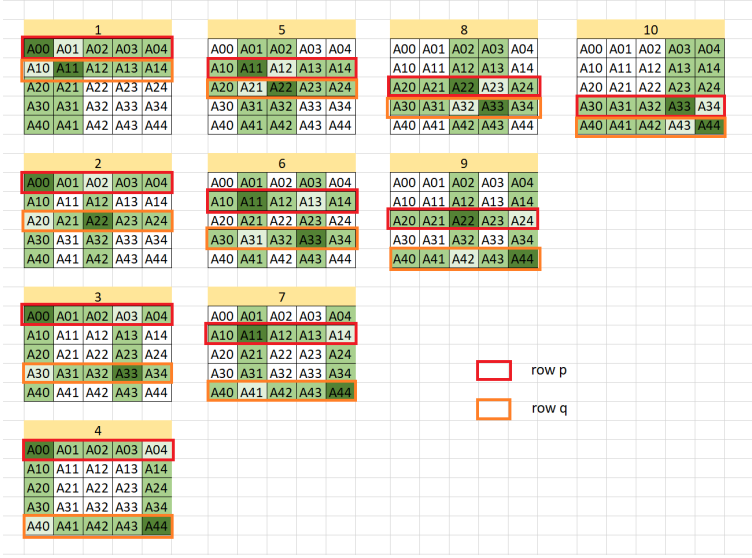
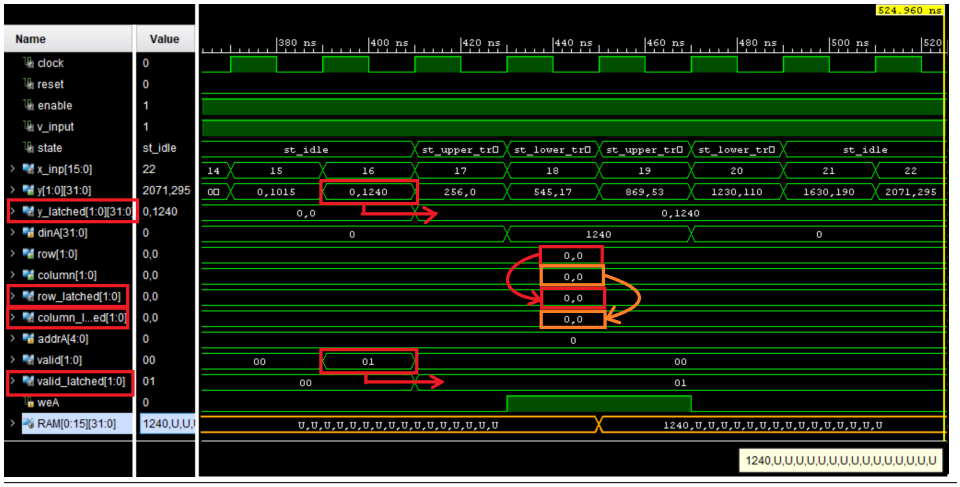
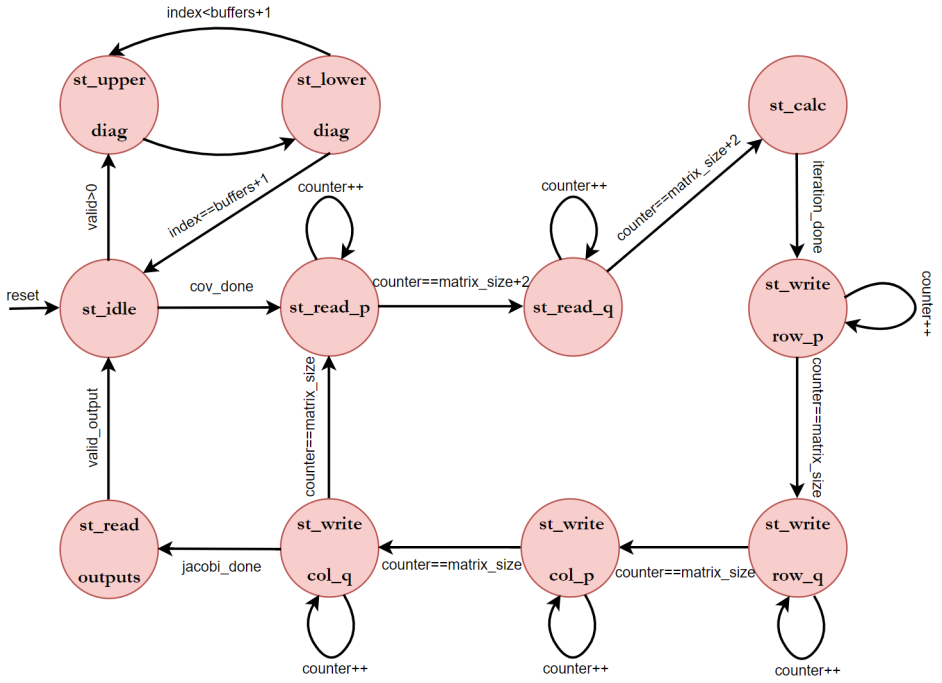


Figure 5.4 Writing covariance output to RAM. The waves show how a output from the covariance module is latched with a new signal and written to RAM in its corresponding address.



The state machine for the top level is given in Fig. 5.5 presenting the state transitions and the behaviour of control signals. Moreover, the Table 5.1 summarizes the functionality of each state in the top level.

Figure 5.5 State machine for the main module of the design. The flow of writing to and reading from BRAM operations.



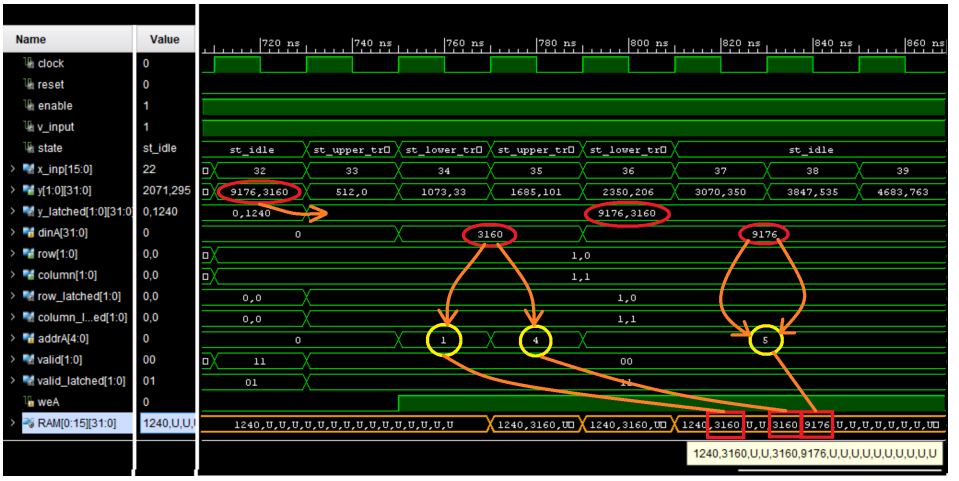
The sequential design involves the first three states of the state machine for covariance writing, namely **st_upper_triangle**, **st_lower_triangle** and **st_idle**. While *enable* is asserted, each state transition occurs in every clock cycle. Initially, the process is in the **st_idle** state. When there is a valid output from the covariance, that is a new element of the matrix is computed, all required values are latched. This behaviour is indicated with arrows in Fig. 5.4 for all four signals mentioned above. It must be noted that, the input stream to the covariance module is randomly generated values only for easy testing purposes.

Fig. 5.6 is given as an example of having two buffers, which means there are two valid outputs at a time. In this case, the outputs are 3160 and 9176 which are a_{01} and a_{11}, respectively. When the output array is latched, in the next clock cycle, state transits to **st_upper_triangle**. First, address is calculated with the formula in 5.2, in this example it is 1. The data is put to the RAM by enabling *we* - write enable signal for the particular index. Then the state goes to **st_lower_triangle** where it puts another copy of the same data in its corresponding address in the lower diagonal of the symmetric matrix. The address is calculated with the formula given in Eq. 5.2 and it is 4. Before the next transition, the index is incremented if it has not yet reached to the number of buffers. If not, then the

state	functionality
st_upper_diagonal	Upper diagonal elements of the covariance matrix are written to Block RAM. One entry is written at a time. Identity matrix is constructed in another Block RAM at the same time.
st_lower_diagonal	Lower diagonal elements of the covariance matrix are written to Block RAM. One entry is written at a time. Identity matrix is constructed in another Block RAM at the same time.
st_idle	The outputs from the covariance matrix are captured here for writing. When writing of covariance matrix to BRAM is finished, state is transits to the reading states to read the corresponding rows from BRAM to Jacobi design.
st_read_p	Row p is read to Jacobi for eigenvalue matrix calculation. Column p is read to Jacobi for eigenvector matrix calculation. When the whole row and whole column are read, state goes to read the next row and column.
st_read_q	Row q is read to Jacobi for eigenvalue matrix calculation. Column q is read to Jacobi for eigenvector matrix calculation. Reading continues until it finished all elements in the corresponding row and column.
st_calc	Jacobi starts calculations of eigenvalues and eigenvectors. One iteration is done and Jacobi module asserts *iteration_done* signal. Then BRAM allows Jacobi to write updated rows and columns back.
st_write_row_p	Updated row p is written to row p in eigenvalue matrix in BRAM.
st_write_row_q	Updated row q is written to row q in eigenvalue matrix in BRAM.
st_write_column_p	Updated row p is duplicated to column p in eigenvalue matrix in BRAM. Updated column p in eigenvector matrix is written to column p in eigenvector matrix in BRAM.
st_write_column_q	Updated row q is duplicated to column q in eigenvalue matrix in BRAM. Updated column q in eigenvector matrix is written to the same column in BRAM. If one iteration has finished but the whole computation still continues, the state will switch to **st_read_p** and repeat the steps for next iteration. If the Jacobi has finished the whole computation, that is if it asserts *jacobi_done* signal, the state will go to **st_read_outputs**.
st_read_outputs	The computation is done. Eigenvalue and eigenvector matrices are in the BRAM. Depending on *output_select* signal, the corresponding eigenvalue and its eigenvectors are read from BRAM to the outside world.

Table 5.1: The states of the state machine in top level design for reading from and writing to Block RAM.

Figure 5.6 Writing covariance output to RAM. The waves indicate how the states work and the same value is written to two different addresses, those are the upper and lower diagonal of a symmetric matrix.



transition happens back to **st_upper_triangle** again for the element in next index. These steps repeat until all the elements are stored in their matching addresses both in upper and lower diagonals of the matrix. As seen, the elements on the diagonal gets the same address number.

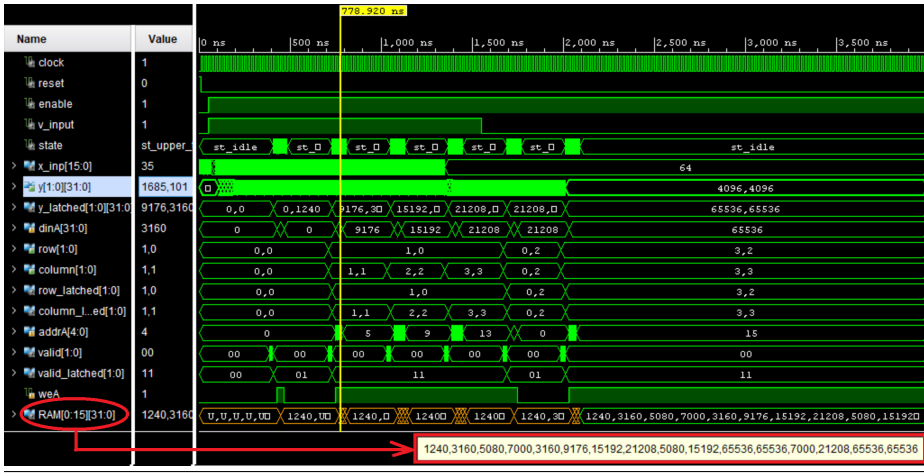
After all the elements finish storing, state becomes idle and RAM is full. An example of a finished process is provided in Fig. 5.7.

As the covariance matrix is fully stored in BRAM, it is ready for eigenvalue decomposition. To optimise the timing, both eigenvalue and eigenvector matrices are calculated concurrently. Therefore, another BRAM instantiation is created to hold eigenvector matrix. As discussed before, initial matrix is Identity matrix for eigenvector calculations. So that, while covariance matrix is being stored in BRAM, identity matrix is generated in another BRAM. This process also happens in the first two states: **st_upper_diagonal** and **st_lower_diagonal**. When initial matrices are constructed, a state transition happens to **st_idle**. The covariance module asserts *cov_done* signal, so the top level checks that covariance matrix is finished calculation and Jacobi module does not have valid output, read operation can begin. The *we* signal of the BRAM is de-asserted here, so BRAM is in reading operation mode.

There are two states for reading: **st_read_p** reads the row *p*, **st_read_q** reads the row **q** from each eigenvalue and eigenvector matrices to Jacobi module. The simulation waves given in Fig. 5.8 show the behaviour of the signals in **st_read_p** and **st_read_q** states. The process of reading the correct rows from the BRAM is highlighted using different colors for different states and arrows. Besides, the write enable signal of the BRAM: *we* is low, so it allows reading.

In the figure, it is also observed that the Jacobi stays in idle while the reading is happening. After row **p** finishes the reading of one whole row from the particular addresses,

Figure 5.7 Writing covariance output to RAM. The simulation waves show a finished write operation. Here, a 4×4 matrix is used for simplification. All the elements of the covariance matrix is located in their corresponding addresses.



row **q** starts. When both rows are read to Jacobi module, it starts doing Jacobi rotations on the concerned rows. The calculations happen in **st_calc**. The simulation waves for this state are shown in Fig. 5.9.

As can be seen, while the top level is in **st_calc**, Jacobi executes all its states in an order. When one whole iteration finishes, Jacobi module assigns *iteration_done* signal, that is in its **st_delay** state. In top level, the condition *iteration_done* = *'1'* is checked in the **st_calc** state. If the iteration is finished, that is the rows are updated, top level starts writing altered rows back to BRAM to their matching addresses. As the matrix is a symmetric matrix, row *p* and row *q* are exactly the same as column *p* and column *q*, respectively. Thus, redundant calculation of the addresses for the columns are eliminated and the altered rows are written back to the same rows and also duplicated to the equivalent

Figure 5.8 The simulation waves illustrating the *read* operation from BRAM to Jacobi module. There are two states to read different rows.

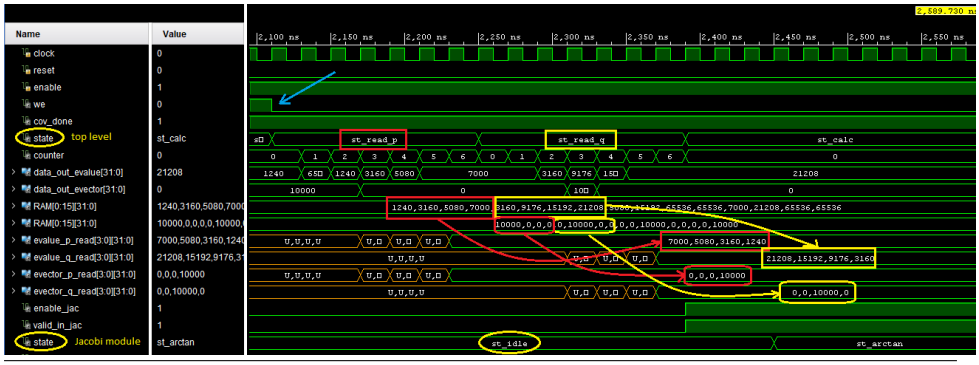
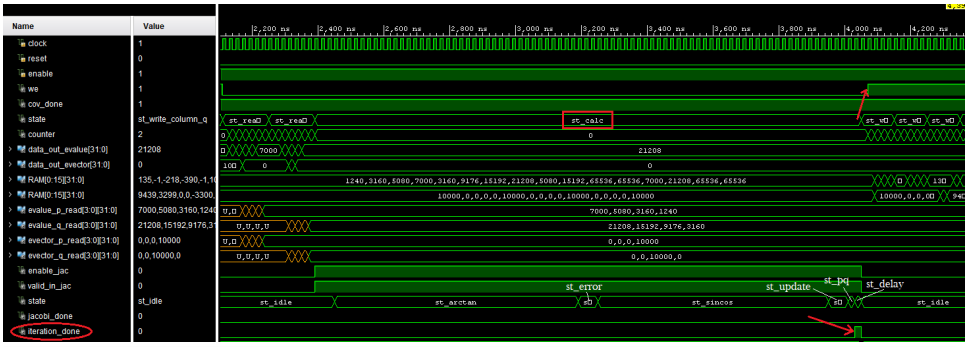
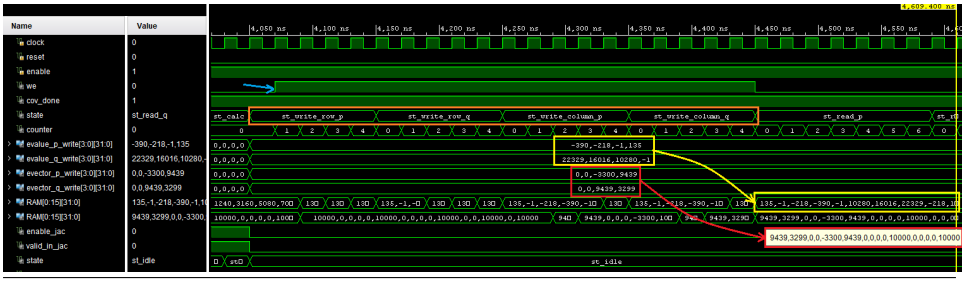


Figure 5.9 The waves present the **st_calc** state. While Jacobi calculations are done, the top level stays in **st_calc** until one iteration finishes.



columns. The write-back operation for eigenvalues is done in 4 states: **st_write_row_p**, **st_write_row_q**, **st_write_column_p**, **st_write_column_q**. Write enable signal *we* for the BRAM goes high and stays asserted until the 4 states finish writing the updated values back to their corresponding addresses. This behaviour of writing operation can be observed in the waves given in Fig. 5.10.

Figure 5.10 The simulation waveforms for the *write* operation are shown.



After each *write* state finishes, state transition happens to the next one in the given order. The write-back operation for the eigenvectors are performed at the same time as eigenvalues when they are written back to columns, e.g. in states **st_write_column_p** and **st_write_column_q**. When the whole writing operation has finished, but the computation still continues (because the error function is not sufficiently small yet), state loops back to **st_read_p** for the following iteration and the steps are repeated. If the computation is done, Jacobi asserts *computation_done* (called *jacobi_done* in main module), the state transitions to the last state called **st_read_outputs** from **st_write_column_q**. In **st_read_outputs**, the valid output results are read from the BRAM to the outside world. The order number of which eigenpair is going to be read is chosen by *output_select* signal. Because of the **IO constraints** of the FPGA, eigenvectors in one row are read out one by one. Consequently, one row of eigenvectors and their matching eigenvalue is read out in each selection. The clock cycles needed for each state is discussed in Chapter 6.

Chapter 6

CONCLUSION

6.1 Results

6.1.1 Timing analysis

In order to calculate the computation time for the whole PCA design, the time used for the covariance calculation, *read* and *write* operations and Jacobi design must be summed up together. The computation time for the covariance module has been already calculated in the specialization project with the formulas 2.3 and 2.4 given in Chapter 1. For the 100×100 matrix with 2500 components and using 4 buffers, the needed number of clock cycles is 3 250 000. Obviously, it can be made faster by using more buffers. As the covariance matrix is written to the BRAM while it is calculating, there is no additional time needed for its *write* operation.

The results for each state of the Jacobi design is summarised in the Table 6.1:

states	st_arctan	st_error	st_sincos	st_update	st_pq	st_delay	**total**
#cycles	37	3	35	3	1	1	**80**

Table 6.1: Number of clock cycles each state in the Jacobi design utilises in one iteration.

The time required for each operation for the calculations and storing is presented in Table 6.2. The result is for one complete iteration. There are two states for reading and four states for writing.

states of top module	read	calculation	write
# of clock cycles	2 * (matrix_size + 3)	81	4 * (matrix_size + 1)

Table 6.2: Number of clock cycles for Jacobi (1 cycle is added because of the transition to **st_calc** state) along reading and writing in one iteration.

It follows that for one iteration of 100×100 matrix, the required number of clock cycles

is:

$$2 * (100 + 3) + 81 + 4 * (100 + 1) = 691 \tag{6.1}$$

The number of iterations depends on the input data. Therefore a formula for 100×100 matrix is generated as below:

$$\text{computation time} = 3250000 + \#iterations * 691 \tag{6.2}$$

Additionally, there is more time required to read the outputs and it depends on how many eigenvectors the user wants to read out.

Optimisation: Pipelining

(Kiusalaas, 2010) mentions that the utility of Jacobi method is limited to 20×20 matrices due to the rapid increase in the computational effort regards to the matrix size. This condition holds for the hardware as well. Especially, when the utilised FPGA has a small number of available resources, it turns out to not to fit to the hardware for bigger matrices. The usage of the computational slices depending on the matrix size is roughly visualised in Fig. 6.1(a). In this design, submodules **off** and **eigenanalysis** are pipelined. The number of computational blocks is parameterizable. Ideally, these computational slices synthesize to DSPs. In this case, the intended number of computations use the same DSP. Fig. 6.1(b) illustrates the novel behaviour. This, in turn, makes the design more flexible for any FPGA. It makes it also possible to just decrease the parameter value if the other parts of the design needs more DSPs. The slices are designed to fit in DSPs.

matrix size	n_{off}	n_{eig}	**LUT**	**FF**	**DSP**
4	1	1	7023 (13%)	5533 (5%)	58 (26%)
8	1	1	8331 (16%)	7610 (7%)	58 (26%)
16	1	1	10763 (20%)	11719 (11%)	58 (26%)

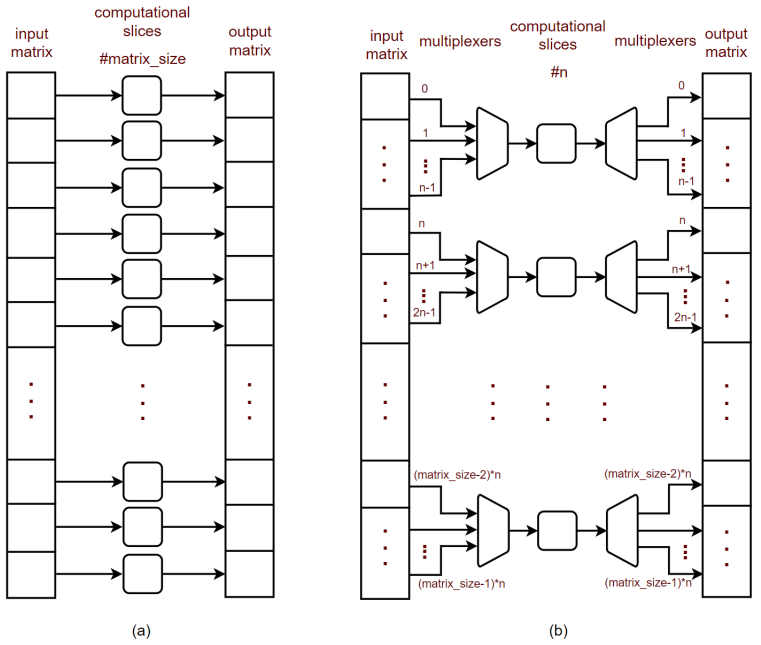
Table 6.3: Resource usage in Zilinx Zedboard for 1 computational slices both in **off** and **eigenanalysis** submodules. For all of them, 2 buffers are used in covariance. The table proves that the matrix size does not affect the number of DSP block utilization.

matrix size	n_{off}	n_{eig}	**LUT**	**FF**	**DSP**
4	4	4	7607 (13%)	5505 (5%)	130 (59%)
8	4	4	10213 (16%)	7658 (7%)	130 (59%)
16	4	4	12565 (20%)	14156 (11%)	130 (59%)

Table 6.4: Resource usage in Zilinx Zedboard for 4 computational slices both in **off** and **eigenanalysis** submodules. For all of them, 2 buffers are used in covariance. The table proves that the matrix size does not affect the number of DSP block utilization.

Along with the pipelining of both submodules, a new variable has been added to both. This new variable *n* defines how many computational slices will be shared among the computations. It is important to find the ideal value for the *n*. This optimum point can be

Figure 6.1 (a) Typical behaviour of the computation. (b) Behaviour after pipelining.



#buffers	LUT	FF	LUTRAM	DSP
2	2952 (6%)	2732 (3%)	2718 (16%)	2 (1%)
4	5863 (11%)	5433 (5%)	5436 (31%)	4 (2%)
10	14626 (27%)	13560 (14%)	13590 (78%)	10 (5%)

Table 6.5: Resource usage for the covariance module with different numbers of buffers used. The matrix size is 100×100.

defined as maximum number of available DSPs should be used. It is worth to note that the matrix size must be divisible by **n**, it is crucial to make sure it scales well with the number of planes. It can be seen from the tables 6.3 and 6.4 that DSP usage is constant for any matrix size. The number of LUTs and FFs increase slightly because of the rest of the design with larger matrices. To summarize, choosing a bigger *n* yields faster computation, while a smaller *n* results in lower DPS occupation.

The usage of DSP and other resources in the covariance module alone for 100×100 matrix is given in Table 6.5. The number of DSPs increase linearly with the number of buffers used. Moreover, the Table 6.6 presents the resource usage for the remaining modules. They each make up nearly 1% of the whole resource usage.

File Name	LUT	FF	DSP
cordic_vector	356 (1%)	104 (1%)	0
cordic_rotation	325 (1%)	204 (1%)	0
choose_pq	23 (1%)	14 (1%)	0

Table 6.6: Synthesis results - Resource usage for separate modules. The matrix size is 100×100.

6.2 Discussion

A hardware implementation of PCA based on the Jacobi method was designed and implemented in FPGA. A fully parameterizable and efficient Jacobi method was designed for eigenpair decomposition. The module calculates both eigenvalues and eigenvectors. In the proposed design of Jacobi method, fixed point number representation is used. By using BRAM as intermediate storage, covariance matrix modules are connected with the Jacobi modules. The use of CORDIC for trigonometric as well as square root calculations made the design more efficient by replacing expensive operations with inexpensive hardware. The design uses minimal hardware for big matrices (e.g. 100×100). New ways of implementation have been researched and found. Expensive matrix multiplication is eliminated by using update formulas for both eigenvalue and eigenvector matrix calculations. The stopping criterion is one of the crucial parts of the design for hardware. A novel and efficient approach for calculating error function is proposed. Another remarkable novel optimisation in the design achieves its high flexibility of usage in various FPGA sizes. Along with the pipelining in two biggest modules of the Jacobi, parameterizable number of computational slice usage results in substantial improvement in resource usage as well as the speed.

6.3 Future work

The proposed design can be used in many other signal processing and machine learning applications. The PCA design can be used together with other compression and dimensionality reduction algorithms for better performance. Cyclic Jacobi may be replaced by Parallel Jacobi for an FPGA with more resources available where the speed is the first priority. Parallel Jacobi would use more hardware resources, but it would be much faster than other methods of Jacobi. If the data variables are in different units, normalization step must be done to eliminate the difficulties: mean centering and division by standard deviation. More optimisations can be made to increase the throughput and decrease the power consumption. The design may be made to use less clock cycles for the calculation. Although the implemented design is efficient, there is still more room for the paralellisation and improvements.

Bibliography

7series, X., 2016. 7 series FPGAs Memory Resources User Guide. Xilinx All Programmable.

Alan, S., 2008. Cordic: How hand calculators calculate.

Andraka, R. A. C. G. I., Copyright 1998. "a survey of cordic algortihms for fpga based computers".

BRAM, X., 2011. Spartan-6 FPGA Block RAM Resources. Xilinx All Programmable.

Bravo, Ignacio. Jimenez, P. e. a., 2006. Implementation in fpgas of jacobi method to solve the eigenvalue and eigenvector problem.

Demmel, J. Veselic, K., 1992. Jacobi's method is more accurate than QR.). SIAM J. Matrix Anal. Appl. 13(4).

Deprettere, E., Udo, R., 1984. Pipelined cordic architecture for fast vlsi filtering and array processing, 41.A.6.1 – 41.A.6.4.

Designing with IP, X., 2018. Vivado Design Suite User Guide. Xilinx All Programmable.

Golub, G. H., Van Loan, C. F., 1996. Matrix Computations (3rd ed.). The Johns Hopkins University press.

Gonzales, C. Lopez, S. M. D. S. R., 2015. "fpga implementation of the hysime algorithm for the determination of the number of endmembers in hyperspectral data".

Jin, X., 2014. Implementation of the music algorithm in clash.

Karimova, A., 2017. Spealisation project: High-throughput fpga implementation of pca algorithm for dimensionality reduction in hyperspectral imaging, covariance calculation.

Kiusalaas, J., 2010. Numerical Methods in Engineering with Python. Cambridge University Press.

Libraries, X., 2017. Vivado Design Suite 7 series FPGA and Zynq-7000 all programmable SoC Libraries Guide. Xilinx All Programmable.

Makhoul, J., S. R., Gish, H., 1985. Vector quantization in speech coding, 1551–1588.

Mathias, R., 1995. Accurate eigensystem computations by Jacobi methods.). SIMAX, Vol 16, No 3.

Qian, Du. James Fowler, E., 2008. "low-complexity principal component analysis for hyperspectral image compression".

Stewart, G. W., 1983. "a method for computing for computing the generalzed singular value decomposition.".

Thibedeau, K., 2014. Vhdl extras.
URL http://github.com/kevinpt/vhdl-work

Uday, A. K., 2016. "reconfigurable hardware implementation for the principal component analysis".

Veisdal, J., 2018. Hardware hsi design.
URL https://www.ntnu.no/wiki/display/NSSL/Electronics+Design

Veselic, K. Hari, V., 1990. "a note on a one-sided jacobi algorithm.".

Wang, R., 2017. Jacobi eigenvalue algorithm for symmetric real matrices, 1–5.
URL http://fourier.eng.hmc.edu/e176/lectures/ch1/node1.html

Weedman, K., 2013. The tutorial: Cordic design & simulation in verilog.
URL https://www.youtube.com/watch?v=TJe4RUYiOIg&t=1676s

Wikipedia, C., 2017. Jacobi eigenvalue algorithm.
URL https://en.wikipedia.org/w/index.php?title=Jacobi_eigenvalue_algorithm&oldid=778088775

Appendix

Top level design for the proposed PCA architecture.

Covariance module and Jacobi module connected through two Block RAMs. File name: toplevel_cov_bram_jac.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT master thesis-PCA Jacobi              --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS-2018                                --
-- Top level for PCA algorithm                    --
----------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_unsigned.all;
use work.covariance_pkg.all;
use work.array_pkg.all;

entity toplevel_cov_bram_jac is
  generic(
    matrix_size : integer := 100;
    BRAM_size   : integer := 10000;
    BRAM_width  : integer := 32;
    ADDR_width  : integer := 14;
    data_width  : positive:= 16;
    buffers     : positive:= 4
  );
  port(
    clock             : in  std_logic;
    reset             : in  std_logic;
    enable            : in  std_logic;
    cube_data         : in  std_logic_vector(data_width - 1 downto 0);
    valid_input_cov   : in  std_logic;
    output_select     : in  natural range 0 to matrix_size - 1;
    select_valid      : in  std_logic;
    valid_output      : out std_logic;
```

```
    eigenvalue          : out signed(31 downto 0);
    eigenvectors        : out signed(31 downto 0)
  );
end toplevel_cov_bram_jac;

architecture behav_top of toplevel_cov_bram_jac is

  type state_type is (st_upper_diagonal, st_lower_diagonal, st_idle,
                      st_read_p, st_read_q, st_calc, st_write_row_p,
                      st_write_row_q, st_write_column_p,
                      st_write_column_q, st_read_outputs);
  signal state : state_type;

  component covariance_optimised is
    generic(
      data_width  : positive := 16; --size of one component
      depth_array : positive := 2500;--number of components in 1 plane
      planes      : positive := 100; --number of planes, e.g. 100
      buffers     : positive := 4   --number of buffers
    );
    port(
      x             : in  std_logic_vector(data_width - 1 downto 0);
      clock, reset  : in  std_logic;
      enable        : in  std_logic;
      done          : out std_logic;
      valid_input   : in  std_logic;
      valid         : out std_logic_vector(buffers - 1 downto 0);
      column, row   : out output_row_column(buffers - 1 downto 0);
      y             : out output_t(buffers - 1 downto 0)
    );
  end component;

  component Block_RAM is
    generic(
      BRAM_size  : integer := 10000;
      BRAM_width : integer := 32;
      ADDR_width : integer := 14
    );
    port(
      clock   : in  std_logic;
      din     : in  std_logic_vector(BRAM_width-1 downto 0);
      we      : in  std_logic; -- write enable
      address : in  std_logic_vector(ADDR_width-1 downto 0);
      dout    : out std_logic_vector(BRAM_width-1 downto 0)
    );
  end component;
```

```
  component top_level_cycjacobi is
      generic(
      constant matrix_size : positive := 100;
      constant tolerance   : integer  := 100 --1000000
      );
      port(
      clock           : in  std_logic;
      reset           : in  std_logic;
      enable          : in  std_logic;
      --input data declarations: eigenvalue
      evalue_p_read   : in  row_column_array;
      evalue_q_read   : in  row_column_array;
      --input data for eigenvectors
      evector_p_read  : in row_column_array;
      evector_q_read  : in row_column_array;
      --control signal declarations
      valid_input     : in  std_logic;
      iteration_done  : out std_logic;
      computation_done: out std_logic;
      --output data declarations: eigenvalue
      evalue_p_write  : out row_column_array;
      evalue_q_write  : out row_column_array;
      --output data eigenvectors
      evector_p_write : out row_column_array;
      evector_q_write : out row_column_array;
      --output indices
      --old indices for write operations
      p_latched       : out natural range 0 to matrix_size - 1;
      q_latched       : out natural range 0 to matrix_size - 1;
      --new indices for read operations
      p_new           : out natural range 0 to matrix_size - 1;
      q_new           : out natural range 0 to matrix_size - 1
      );
  end component;

--intermediate signal declarations
 signal column, row          : output_row_column(buffers-1 downto 0);
 signal column_latched       : output_row_column(buffers-1 downto 0);
 signal row_latched          : output_row_column(buffers-1 downto 0);
 signal data_in_evalue       : std_logic_vector(BRAM_width-1 downto 0);
 signal data_in_evector      : std_logic_vector(BRAM_width-1 downto 0);
 signal address_evalue       : std_logic_vector(ADDR_width-1 downto 0);
 signal address_evector      : std_logic_vector(ADDR_width-1 downto 0);
 signal index, counter       : integer;
 signal we                   : std_logic;
 signal cov_done             : std_logic;
 signal enable_jac           : std_logic;
```

```
 signal valid_in_jac         : std_logic;
 signal jacobi_done          : std_logic;
 signal iteration_done       : std_logic;
 signal evalue_p_write       : row_column_array;
 signal evalue_p_read        : row_column_array;
 signal evalue_q_read        : row_column_array;
 signal evalue_q_write       : row_column_array;
 signal evector_q_read       : row_column_array;
 signal evector_p_read       : row_column_array;
 signal evector_p_write      : row_column_array;
 signal evector_q_write      : row_column_array;
 signal p, q, p_new, q_new   : natural range 0 to matrix_size-1;
 signal p_latched, q_latched: natural range 0 to matrix_size-1;
 signal y, y_latched         : output_t(buffers-1 downto 0);
 signal valid, valid_latched: std_logic_vector(buffers-1 downto 0);
 signal data_out_evalue      : std_logic_vector(BRAM_width-1 downto 0);
 signal data_out_evector     : std_logic_vector(BRAM_width-1 downto 0);


begin
  --port mappings
  cov_ins : covariance_optimised
    port map(
      x           => cube_data,
      clock       => clock,
      reset       => reset,
      enable      => enable,
      done        => cov_done,
      valid_input => valid_input_cov,
      valid       => valid,
      column      => column,
      row         => row,
      y           => y
    );

  --BRAM instantiation for eigenvalue matrix
  bram_ins_evalue : Block_RAM
    generic map(
      BRAM_size   => 10000,
      BRAM_width  => 32,
      ADDR_width  => 14
    )
    port map(
      clock   => clock,
      din     => data_in_evalue,
      we      => we,
      address => address_evalue,
```

```
    dout    => data_out_evalue
  );

--BRAM instantiation for eigenvector matrix
bram_ins_evector : Block_RAM
  generic map(
    BRAM_size   => 10000,
    BRAM_width  => 32,
    ADDR_width  => 14
  )
  port map(
    clock   => clock,
    din     => data_in_evector,
    we      => we,
    address => address_evector,
    dout    => data_out_evector
  );


jac_ins : top_level_cycjacobi
  generic map(
    matrix_size => 100,
    tolerance   => 100
  )
  port map(
    clock           => clock,
    reset           => reset,
    enable          => enable_jac,
    evalue_p_read   => evalue_p_read,
    evalue_q_read   => evalue_q_read,
    evector_p_read  => evector_p_read,
    evector_q_read  => evector_q_read,
    valid_input     => valid_in_jac,
    iteration_done  => iteration_done,
    computation_done=> jacobi_done,
    evalue_p_write  => evalue_p_write,
    evalue_q_write  => evalue_q_write,
    evector_p_write => evector_p_write,
    evector_q_write => evector_q_write,
    p_latched       => p_latched,
    q_latched       => q_latched,
    p_new           => p_new,
    q_new           => q_new
  );

-- combinational logic
enable_jac   <= '1' when (state = st_calc) else '0';
```

```
valid_in_jac <= '1' when (state = st_calc) else '0';

--sequential logic
process(clock, reset)
begin
  if reset = '1' then
    state          <= st_idle;  --default state is st_idle
    index          <= 0;
    data_in_evalue <= (others => '0');
    data_in_evector<= (others => '0');
    address_evalue <= (others => '0');
    address_evector<= (others => '0');
    eigenvalue     <= (others => '0');
    eigenvectors   <= (others => '0');
    valid_output   <= '0';
    we             <= '0';
    counter        <= 0;
    valid_latched  <= (others => '0');
    y_latched      <= (others => (others => '0'));
    column_latched <= (others => 0);
    row_latched    <= (others => 0);
  elsif clock'event and clock = '1' then
    if enable = '1' then
      case state is

        --write upper diagonal elements of
        --the covariance matrix to memory
        when st_upper_diagonal =>
        --calculation of the corresponding address in upper diagonal
          address_evalue <=
          std_logic_vector(to_unsigned(
          (row_latched(index) * matrix_size +
          column_latched(index)), address_evalue'length));

          address_evector <=
          std_logic_vector(to_unsigned(
          (row_latched(index) * matrix_size +
          column_latched(index)), address_evector'length));

        --getting the data into the memory
          data_in_evalue <= std_logic_vector(y_latched(index));
         --creating an identity matrix for eigenvectors
          if (row_latched(index) = column_latched(index)) then
            data_in_evector <= "00000000000000000010011100010000";
          else
            data_in_evector <= "00000000000000000000000000000000";
          end if;
```

```
        we <= valid_latched(index);

        state <= st_lower_diagonal;

      --duplicate the elements from upper diagonal
      --to lower diagonal in the symmetric matrix
      when st_lower_diagonal =>
      --calculation of the corresponding address in lower diagonal
        address_evalue <=
        std_logic_vector(to_unsigned(
        (column_latched(index) * matrix_size +
        row_latched(index)), address_evalue'length));

        address_evector <=
        std_logic_vector(to_unsigned(
        (column_latched(index) * matrix_size +
        row_latched(index)), address_evector'length));

      --getting the data into the memory
        data_in_evalue <= std_logic_vector(y_latched(index));
      --creating an identity matrix for eigenvectors
        if (row_latched(index) = column_latched(index)) then
          data_in_evector <= "00000000000000000010011100010000";
        else
          data_in_evector <= "00000000000000000000000000000000";
        end if;
        we <= valid_latched(index);

        if (index < buffers-1) then
          index <= index + 1;
          state <= st_upper_diagonal;
        else
          state <= st_idle;
        end if;

      --all the output values from the covariance module are kept
--        in new values
      --the reason is the outputs are gone after one clock cycle,
      --thus they must be captured
      when st_idle =>
        index <= 0;
        if (unsigned(valid) > 0) then
          y_latched <= y;           --capturing the output
          valid_latched <= valid;   --capturing the valid signal
          column_latched <= column; --capturing the column number
          row_latched <= row;       --capturing the row number
          state <= st_upper_diagonal;
```

```
  --if covariance is finished and
  --jacobi has not ended the computation, then
  --start reading the concerning rows.
  elsif ((cov_done = '1') and (jacobi_done <= '0')) then
    we <= '0';
    state <= st_read_p;
  else
    state <= st_idle;
  end if;

when st_read_p =>
--reading row p for eigenvalue and eigenvector calc.s
  if (counter < matrix_size) then
    address_evalue <= std_logic_vector(to_unsigned(
      (p_new * matrix_size + counter),
      address_evalue'length ));
    address_evector <= std_logic_vector(to_unsigned((p_new +
    counter * matrix_size), address_evalue'length ));

    counter <= counter + 1;
    --evalue_p_read(counter) <= signed(data_out_evalue);
  elsif (counter < matrix_size+2) then
    counter <= counter + 1;
  else
    counter <= 0;
    state <= st_read_q;
  end if;

  if (counter < matrix_size + 2) then
    if (counter >= 2) then
      --reading row p for eigenvector calc
      evalue_p_read(counter-2) <= signed(data_out_evalue);
      --reading column p for eigenvector calc
      evector_p_read(counter-2) <= signed(data_out_evector);
    end if;
  end if;


when st_read_q =>
  if (counter < matrix_size) then
    address_evalue <= std_logic_vector(to_unsigned(
    (q_new*matrix_size + counter),address_evalue'length));
    address_evector <= std_logic_vector(to_unsigned(
    (q_new + counter*matrix_size),address_evector'length));
    counter <= counter + 1;
  elsif (counter < matrix_size + 2) then
    counter <= counter + 1;
```

```
      else
        counter <= 0;
        state <= st_calc;
      end if;

      if (counter < matrix_size + 2) then
        if (counter >= 2) then
          --reading row p for eigenvector calc
          evalue_q_read(counter-2) <= signed(data_out_evalue);
          --reading column p for eigenvector calc
          evector_q_read(counter-2) <= signed(data_out_evector);
        end if;
      end if;

    when st_calc =>        --Jacobi calculates the eigenvalue
      if (iteration_done = '1') then     --and eigenvectors
        state <= st_write_row_p;
      else
        state <= st_calc;
      end if;

    when st_write_row_p =>
      if (counter < matrix_size) then
      --writes the updated row p back to bram
        address_evalue <= std_logic_vector(to_unsigned(
        (p_latched*matrix_size + counter),address_evalue'length));

        data_in_evalue <=
                      std_logic_vector(evalue_p_write(counter));

        we <= '1';
        counter <= counter + 1;
      else
        counter <= 0;
        state <= st_write_row_q;
      end if;

    when st_write_row_q =>
      if (counter < matrix_size) then
      --writes the updated row q back to bram
        address_evalue <= std_logic_vector(to_unsigned(
        (q_latched*matrix_size + counter),address_evalue'length));
        data_in_evalue <=
                      std_logic_vector(evalue_q_write(counter));
        we <= '1';
        counter <= counter + 1;
      else
```

```
    counter <= 0;
    state <= st_write_column_p;
  end if;

when st_write_column_p =>
  if (counter < matrix_size) then
  --writes the updated row p back to bram
    address_evalue <= std_logic_vector(to_unsigned(
    (p_latched + counter*matrix_size),address_evalue'length));
    address_evector <= std_logic_vector(to_unsigned(
    (p_latched + counter*matrix_size),address_evector'length));
    data_in_evalue <=
                  std_logic_vector(evalue_p_write(counter));
    data_in_evector <=
                  std_logic_vector(evector_p_write(counter));
    we <= '1';
    counter <= counter + 1;
  else
    counter <= 0;
    state <= st_write_column_q;
  end if;


when st_write_column_q =>
  if (counter < matrix_size) then
  --writes the updated row q back to bram
    address_evalue <= std_logic_vector(to_unsigned(
    (q_latched + counter*matrix_size),address_evalue'length));

    address_evector <= std_logic_vector(to_unsigned(
    (q_latched + counter*matrix_size),address_evector'length));

    we <= '1';
    data_in_evalue <=
                  std_logic_vector(evalue_q_write(counter));
    data_in_evector <=
                  std_logic_vector(evector_q_write(counter));
    counter <= counter + 1;
  else
    we <= '0';
    counter <= 0;
    if (jacobi_done = '1') then
      state <= st_read_outputs;
    else
      state <= st_read_p;
    end if;
  end if;
```

```
          --reading the results out
          when st_read_outputs =>
            if (counter < matrix_size) then
              if (select_valid = '1') then
                address_evalue <= std_logic_vector(to_unsigned(
                (output_select*(matrix_size+1)),address_evalue'length));

                address_evector <=
                std_logic_vector(to_unsigned(
                (output_select + counter * matrix_size),
                address_evector'length));

                eigenvalue <= signed(data_out_evalue);
                eigenvectors <= signed(data_out_evector);
                --when the first element is read, set valid_output
                valid_output <= '1';
              end if;
              counter <=  counter + 1;
            elsif (counter < matrix_size + 2) then
              counter <= counter + 1;
            else
              valid_output <= '0';
              counter <= 0;
              state <= st_idle;
            end if;

        end case;
      end if;
    end if;
  end process;
end behav_top;
```

The testbench module designed for 4×4 matrix. Module name: toplevel_cov_bram_jac_tb.vhd.

```
-------------------------------------------------------
-- Aysel Karimova                   --
-- SMALLSAT master thesis-PCA Jacobi        --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS-2018                  --
-- Testbench for PCA algorithm            --
-- This testbench is designed for 4x4 matrix    --
-------------------------------------------------------

library IEEE;
use IEEE.Std_logic_1164.all;
```

```
use IEEE.Numeric_Std.all;
use work.covariance_pkg.all;
use work.array_pkg.all;

entity toplevel_cov_bram_jac_tb is
end toplevel_cov_bram_jac_tb;

architecture sim_top of toplevel_cov_bram_jac_tb is
  constant matrix_size  : integer := 4;    --100
  constant BRAM_size    : integer := 16;   --10000
  constant BRAM_width   : integer := 32;   --32
  constant ADDR_width   : integer := 5;    --14
  constant data_width   : positive := 16; --16
  constant buffers      : positive := 2;   --4

  component toplevel_cov_bram_jac
  port(
    clock          : in  std_logic;
    reset          : in  std_logic;
    enable         : in  std_logic;
    cube_data      : in  std_logic_vector(data_width - 1 downto 0);
    valid_input_cov: in  std_logic;
    output_select  : in  natural range 0 to matrix_size - 1;
    select_valid   : in  std_logic;
    valid_output   : out std_logic;
    eigenvalue     : out signed(31 downto 0);
    eigenvectors   : out signed(31 downto 0)
  );
  end component;

  signal clock          : std_logic := '0';
  signal reset          : std_logic := '1';
  signal enable         : std_logic := '0';
  signal valid_input_cov: std_logic := '0';
  signal cube_data      : std_logic_vector(data_width-1 downto 0) :=
                                                    "0000000000000000";
  signal output_select  : natural range 0 to matrix_size-1;
  signal select_valid   : std_logic := '0';
  signal valid_output   : std_logic;
  signal eigenvalue     : signed(31 downto 0);
  signal eigenvectors   : signed(31 downto 0);

begin

  top : toplevel_cov_bram_jac
  port map(
    clock     => clock,
```

```
    reset      => reset,
    enable       => enable,
    valid_input_cov => valid_input_cov,
    cube_data    => cube_data,
    output_select => output_select,
    select_valid  => select_valid,
    valid_output  => valid_output,
    eigenvalue    => eigenvalue,
    eigenvectors  => eigenvectors
  );

  clock <= not clock after 10 ns;

    testing : process
      begin

      wait for 10 ns;
      reset <= '0';
      wait for 20 ns;
      enable <= '1';
      cube_data <= "0000000000000000";

      enable <= '0';
      wait until clock = '1';
      cube_data <= "0000000000000010";
      enable <= '1';

      valid_input_cov <= '1';

      for i in 0 to 64 loop
        wait until clock = '1';
        cube_data <= std_logic_vector(to_unsigned(i, 16));
      end loop;

      wait for 200 ns;
      valid_input_cov <= '0';

      wait for 200 ns;
      output_select <= 0;
      select_valid <= '1';

      if (valid_output = '1') then
        select_valid <= '0';
      end if;

      wait for 200 ns;
      output_select <= 1;
```

```
      select_valid <= '1';

      if (valid_output = '1') then
        select_valid <= '0';
      end if;


      assert false report "test ended" severity note;
      wait;
      end process testing;
end sim_top;
```

Proposed Jacobi design and its submodules.

Top level design for Cyclic Jacobi method. File name: top_level_cycjacobi.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT master thesis-PCA Jacobi              --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS-2018                                --
-- Top level module for Jacobi                    --
----------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library work;
use work.array_pkg.all;
use work.cordic.all;

entity top_level_cycjacobi is
  generic(
    constant matrix_size : positive := 100;
    constant tolerance   : integer  := 100 --1000000
  );
  port(
    clock           : in  std_logic;
    reset           : in  std_logic;
    enable          : in  std_logic;
    --input data declarations: eigenvalue
    evalue_p_read   : in  row_column_array;
    evalue_q_read   : in  row_column_array;
    --input data for eigenvectors
    evector_p_read  : in row_column_array;
    evector_q_read  : in row_column_array;
```

```
    --control signal declarations
    valid_input      : in  std_logic;
    iteration_done   : out std_logic;
    computation_done: out std_logic;
    --output data declarations: eigenvalue
    evalue_p_write   : out row_column_array;
    evalue_q_write   : out row_column_array;
    --output data eigenvectors
    evector_p_write : out row_column_array;
    evector_q_write : out row_column_array;
    --old p to calculate the write data
    p_latched        : out natural range 0 to matrix_size - 1;
    q_latched        : out natural range 0 to matrix_size - 1;
    --output updated indexes for reading operation
    p_new            : out natural range 0 to matrix_size - 1;
    q_new            : out natural range 0 to matrix_size - 1
  );
end top_level_cycjacobi;

architecture Behavioral of top_level_cycjacobi is

  type state_type is (st_error, st_update, st_pq, st_delay, st_idle,
                      st_arctan, st_sincos);
  signal state : state_type;

  --component instantiations
  component off
    port(
      valid_input    : in  std_logic;
      row_p          : in  row_column_array;
      row_q          : in  row_column_array;
      p              : in  natural range 0 to matrix_size - 1;
      q              : in  natural range 0 to matrix_size - 1;
      clock          : in  std_logic;
      reset          : in  std_logic;
      enable         : in  std_logic;
      error          : out signed(71 downto 0);
      valid_output   : out std_logic
    );
  end component;

  component cordic_sequential is
    generic(
      SIZE               : positive   := 32; --# Width of operands
      ITERATIONS         : positive   := 32; --# Number of iterations
      RESET_ACTIVE_LEVEL : std_ulogic := '1' --# Asynch. reset control
    );
```

```
  port(
    --# {{clocks|}}
    Clock         : in  std_ulogic; --# System clock
    Reset         : in  std_ulogic; --# Asynchronous reset

    --# {{control|}}
    Data_valid    : in  std_ulogic; --# Load new input data
    Busy          : out std_ulogic; --# Generating new result
    Result_valid  : out std_ulogic; --# Flag when result is valid
    Mode          : in  cordic_mode; --# Rotation or vector mode sel.

    --# {{data|}}
    X             : in  signed(SIZE - 1 downto 0); --# X coordinate
    Y             : in  signed(SIZE - 1 downto 0); --# Y coordinate
    Z             : in  signed(SIZE - 1 downto 0); --# Z coordinate
                                                   --(angle in brads)

    X_result      : out signed(SIZE - 1 downto 0); --# X result
    Y_result      : out signed(SIZE - 1 downto 0); --# Y result
    Z_result      : out signed(SIZE - 1 downto 0)  --# Z result
  );
end component;

component sincos_sequential is
  generic(
    SIZE                : positive    := 32; --# Width of operands
    ITERATIONS          : positive    := 32; --# Number of iterations
    FRAC_BITS           : positive    := 30; --# Total fractional bits
    MAGNITUDE           : real        := 1.0;--# Scale factor
    RESET_ACTIVE_LEVEL  : std_ulogic  := '1' --# Asynch. reset control
  );
  port(
    --# {{clocks|}}
    Clock         : in  std_ulogic; --# System clock
    Reset         : in  std_ulogic; --# Asynchronous reset

    --# {{control|}}
    Data_valid    : in  std_ulogic; --# Load new input data
    Busy          : out std_ulogic; --# Generating new result
    Result_valid  : out std_ulogic; --# Flag when result is valid

    Angle         : in  signed(SIZE - 1 downto 0); --# Angle in brads
                                     --(2**SIZE brads = 2*pi radians)

    --# {{data|}}
    Sin           : out signed(SIZE - 1 downto 0); --# Sine of Angle
    Cos           : out signed(SIZE - 1 downto 0) --# Cosine of Angle
```

```
  );
end component;

component choose_pq
  port(
    clock  : in  std_logic;
    reset  : in  std_logic;
    enable : in  std_logic;
    p_out  : out natural range 0 to matrix_size - 1;
    q_out  : out natural range 0 to matrix_size - 1
  );
end component;

component eigenanalysis
  port(
     clock          : in std_logic;
     reset          : in std_logic;
     enable         : in std_logic;
     valid_input    : in std_logic;
     evalue_p_in    : in row_column_array;
     evalue_q_in    : in row_column_array;
     evector_p_in   : in row_column_array;
     evector_q_in   : in row_column_array;
     cos            : in signed(31 downto 0);
     sin            : in signed(31 downto 0);
     p              : in natural range 0 to matrix_size-1;
     q              : in natural range 0 to matrix_size-1;
     evalue_p_out   : out row_column_array;
     evalue_q_out   : out row_column_array;
     evector_p_out  : out row_column_array;
     evector_q_out  : out row_column_array;
     valid_output   : out std_logic
     );
end component;

--intermediate signals
signal error                                  : signed(71 downto 0);
signal arctan_valid_in, sincos_valid_in       : std_logic;
signal enable_jac, enable_pq, enable_off      : std_logic;
signal valid_input_jac, valid_output_jac      : std_logic;
signal valid_input_off, valid_output_off      : std_logic;
signal evalue_in_p_temp, evalue_in_q_temp     : row_column_array;
signal evector_in_p_temp, evector_in_q_temp   : row_column_array;
signal evalue_out_p_temp, evalue_out_q_temp   : row_column_array;
signal evector_out_p_temp, evector_out_q_temp : row_column_array;
signal arctan_valid_out, sincos_valid_out     : std_ulogic;
signal denominator, numerator                 : signed(31 downto 0);
```

```
  signal theta, arctan                              : signed(31 downto 0);
  signal cos_cordic, sin_cordic, cos, sin           : signed(31 downto 0);
  signal X_result, Y_result                         : signed(31 downto 0);
  signal sincos_busy, arctan_busy                   : std_ulogic;
  signal p, q                        : natural range 0 to matrix_size-1;

begin

--module instantiations

  off_instance : off
    port map(
      valid_input   => valid_input_off,
      row_p         => evalue_in_p_temp,
      row_q         => evalue_in_q_temp,
      p             => p,
      q             => q,
      clock         => clock,
      reset         => reset,
      enable        => enable_off,
      error         => error,
      valid_output  => valid_output_off
    );

  cordicvector_instance : cordic_sequential
    generic map(
      SIZE       => 32,
      ITERATIONS => 32
    )
    port map(
      Clock        => clock,
      Reset        => reset,
      Data_valid   => arctan_valid_in,
      Busy         => arctan_busy,
      Result_valid => arctan_valid_out,
      Mode         => cordic_vector,
      X            => denominator,
      Y            => numerator,
      Z            => (others => '0'),
      X_result     => X_result,
      Y_result     => Y_result,
      Z_result     => arctan
    );

  cordicrotate_instance : sincos_sequential
    generic map(
      SIZE       => 32,
```

```
    ITERATIONS => 32,
    FRAC_BITS  => 30
  )
  port map(
    Clock        => clock,
    Reset        => reset,
    Data_valid   => sincos_valid_in,
    Busy         => sincos_busy,
    Result_valid => sincos_valid_out,
    Angle        => theta,
    Sin          => sin_cordic,
    Cos          => cos_cordic
  );

choose_pq_instance : choose_pq
  port map(
    clock  => clock,
    reset  => reset,
    enable => enable_pq,
    p_out  => p,
    q_out  => q
  );

 eigenanalysis_instance : eigenanalysis
  port map(
    clock         => clock,
    reset         => reset,
    enable        => enable_jac,
    valid_input   => valid_input_jac,
    evalue_p_in   => evalue_in_p_temp,
    evalue_q_in   => evalue_in_q_temp,
    evector_p_in  => evector_in_p_temp,
    evector_q_in  => evector_in_q_temp,
    cos           => cos,
    sin           => sin,
    p             => p,
    q             => q,
    evalue_p_out  => evalue_out_p_temp,
    evalue_q_out  => evalue_out_q_temp,
    evector_p_out => evector_out_p_temp,
    evector_q_out => evector_out_q_temp,
    valid_output  => valid_output_jac
  );

--combinational logic

p_new <= p;
```

```
q_new <= q;

enable_pq  <= '1' when (state = st_pq) else '0';
enable_jac <= '1'; -- when (state = st_update) else '0';
enable_off <= '1';-- when (state = st_error) else '0';

--the inputs for cordic vectoring:
--the values are the inputs for arctan to calculate the angle
--formula is angle = 1/2 * arctan (2*Apq / (Aqq - App))
denominator   <= (evalue_in_q_temp(q)-evalue_in_p_temp(p));
numerator     <= (evalue_in_p_temp(q)(30 downto 0) & '0');--2*A(p,q)
theta         <= (arctan(31) & arctan(31 downto 1)); -- divide by 2

--sequential logic
process(clock, reset)
begin
  if reset = '1' then
    state            <= st_idle;
    arctan_valid_in  <= '0';
    sincos_valid_in  <= '0';
    computation_done <= '0';
    iteration_done   <= '0';
    valid_input_off  <= '0';
    valid_input_jac  <= '0';
    cos              <= "00000000000000000000000000000000";
    sin              <= "00000000000000000000000000000000";
    evalue_p_write   <=(others=>"00000000000000000000000000000000");
    evalue_q_write   <=(others=>"00000000000000000000000000000000");
    evector_p_write  <=(others=>"00000000000000000000000000000000");
    evector_q_write  <=(others=>"00000000000000000000000000000000");
    evalue_in_p_temp <=(others=>"00000000000000000000000000000000");
    evalue_in_q_temp <=(others=>"00000000000000000000000000000000");
    evector_in_p_temp<=(others=>"00000000000000000000000000000000");
    evector_in_q_temp<=(others=>"00000000000000000000000000000000");
  elsif clock'event and clock = '1' then
    if enable = '1' then
      case state is

        when st_arctan =>   -- calculate cordic arctan
          arctan_valid_in <= '0';
          if (arctan_valid_out = '1') then
            sincos_valid_in <= '1';
            if (sincos_valid_out = '0') then
              valid_input_off <= '1';
              state <= st_error;
            end if;
          end if;
```

```
when st_error =>     -- calculate error function
  valid_input_off <= '0';
  if (valid_output_off = '1') then
      state <= st_sincos;
  end if;

when st_sincos =>    -- calculate (sin(theta), cos(theta))
  sincos_valid_in <= '0';
  if (sincos_valid_out = '1') then
    valid_input_jac <= '1';
    sin <= sin_cordic;
    cos <= cos_cordic;
    state <= st_update;
  else
    state <= st_sincos;
  end if;

when st_update =>    -- calculate jacobi formulas
   valid_input_jac <= '0';
   if (valid_output_jac = '1') then
      state <= st_pq;
   end if;

when st_pq =>    -- choose new p and q and update the input
  iteration_done <= '1';
  if (error < tolerance) then
      computation_done <= '1';
  end if;
  state <= st_delay;

when st_delay =>     -- delay
  evalue_p_write <= evalue_out_p_temp;
  evalue_q_write <= evalue_out_q_temp;
  evector_p_write <= evector_out_p_temp;
  evector_q_write <= evector_out_q_temp;
  iteration_done <= '0';
  arctan_valid_in <= '1';
  state <= st_idle;

when st_idle =>
  if valid_input = '1' then
    p_latched <= p;
    q_latched <= q;
    evalue_in_p_temp <= evalue_p_read;
    evalue_in_q_temp <= evalue_q_read;
    evector_in_p_temp <= evector_p_read;
```

```
            evector_in_q_temp <= evector_q_read;
            arctan_valid_in <= '1';
            if (arctan_valid_out = '0') then
              computation_done <= '0';
              iteration_done   <= '0';
              state <= st_arctan;
            end if;
          else
            state <= st_idle;
          end if;
        end case;
      end if;
    end if;
  end process;
end Behavioral;
```

Submodule for eigenvector and eigenvalue matrices calculation. File name: eigenanalysis.vhd

```
----------------------------------------------------------
-- Aysel Karimova                                       --
-- SMALLSAT master thesis-PCA Jacobi                    --
-- Supervisors: Kjetil Svarstad, Milica Orlandic        --
-- NTNU-EMECS-2018                                      --
-- Eigenpair decomposition for Jacobi                   --
----------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library work;
use work.array_pkg.all;

entity eigenanalysis is
  generic(
    constant matrix_size : positive := 100;
    constant n           : positive := 4
  );
  port(
     clock        : in std_logic;
     reset        : in std_logic;
     enable       : in std_logic;
     valid_input  : in std_logic;
     evalue_p_in  : in row_column_array;
     evalue_q_in  : in row_column_array;
     evector_p_in : in row_column_array;
     evector_q_in : in row_column_array;
```

```
    cos          : in signed(31 downto 0);
    sin          : in signed(31 downto 0);
    p            : in natural range 0 to matrix_size-1;
    q            : in natural range 0 to matrix_size-1;
    evalue_p_out : out row_column_array;
    evalue_q_out : out row_column_array;
    evector_p_out: out row_column_array;
    evector_q_out: out row_column_array;
    valid_output : out std_logic
  );
end eigenanalysis;

architecture Behavioral of eigenanalysis is

  type state_type is (st_idle, st_busy);
  signal state : state_type;

  --intermediate signals for storing the outputs
  signal evalue_p_temp  : row_column_array;
  signal evalue_q_temp  : row_column_array;
  signal evector_p_temp : row_column_array;
  signal evector_q_temp : row_column_array;
  signal counter        : integer;
begin
  --output assignments
  evalue_p_out  <= evalue_p_temp;
  evalue_q_out  <= evalue_q_temp;
  evector_p_out <= evector_p_temp;
  evector_q_out <= evector_q_temp;

  jacobi : process(clock, reset)
  --variable declarations
  variable var1, var2, var3, var4 : signed(47 downto 0);
  variable var5, var6             : signed(79 downto 0);
  variable var7, var8             : signed(63 downto 0);
  variable vara, varb             : signed(47 downto 0);

  begin

    if reset = '1' then
      evalue_p_temp <= (others => "00000000000000000000000000000000");
      evalue_q_temp <= (others => "00000000000000000000000000000000");
      evector_p_temp<= (others => "00000000000000000000000000000000");
      evector_q_temp<= (others => "00000000000000000000000000000000");
      valid_output  <= '0';
      state         <= st_idle;
      counter       <= 0;
```

```
elsif(clock'event and clock = '1') then
  if enable = '1' then
    case state is
      when st_busy =>
        for i in 0 to n-1 loop
          --update formulas for eigenvector matrix
          var1 := cos(31 downto 16) * evector_p_in(n*counter+i) -
                  sin(31 downto 16) * evector_q_in(n*counter+i);
          evector_p_temp(n*counter+i) <= var1(45 downto 14);

          var2 := sin(31 downto 16) * evector_p_in(n*counter+i) +
                  cos(31 downto 16) * evector_q_in(n*counter+i);
          evector_q_temp(n*counter+i) <= var2(45 downto 14);
          --update formulas for eigenvalue matrix
          if (n*counter+i /= p) and (n*counter+i /= q) then
            vara := cos(31 downto 16) * evalue_p_in(n*counter+i) -
                  sin(31 downto 16) * evalue_q_in(n*counter+i);
            evalue_p_temp(n*counter+i) <= vara(45 downto 14);

            varb := sin(31 downto 16) * evalue_p_in(n*counter+i) +
                  cos(31 downto 16) * evalue_q_in(n*counter+i);
            evalue_q_temp(n*counter+i) <= varb(45 downto 14);
          end if;
        end loop;

          if (counter < (matrix_size/n)-1) then
             counter <= counter + 1;
             state <= st_busy;
          else
    --eigenvalue formulas for App, Aqq and Apq
            var5 :=
              cos(31 downto 16)*cos(31 downto 16)*evalue_p_in(p)-
              2*cos(31 downto 16)*sin(31 downto 16)*evalue_p_in(q)
              + sin(31 downto 16)*sin(31 downto 16)*evalue_q_in(q);
            evalue_p_temp(p) <= var5(59 downto 28);

    --out(q,q) = sin*sin*A(p,p)+2*cos*sin*A(p,q)+cos*cos*A(q,q);
            var6 :=
              sin(31 downto 16)*sin(31 downto 16)*evalue_p_in(p) +
              2*cos(31 downto 16)*sin(31 downto 16)*evalue_p_in(q)
              + cos(31 downto 16)*cos(31 downto 16)*evalue_q_in(q);
            evalue_q_temp(q) <= var6(59 downto 28);

    --out(p,q) = (cos*cos-sin*sin)*A(p,q)+cos*sin*(A(p,p)-A(q,q));
            var7 :=
              (cos(31 downto 16)*cos(31 downto 16) -
              sin(31 downto 16)*sin(31 downto 16))*evalue_p_in(q)+
```

```
                cos(31 downto 16)*sin(31 downto 16)*(evalue_p_in(p)-
                evalue_q_in(q));
              evalue_q_temp(p) <= var7(59 downto 28);

      --out(q,p) = (cos*cos-sin*sin)*A(p,q)+cos*sin*(A(p,p)-A(q,q));
              var8 :=
                (cos(31 downto 16)*cos(31 downto 16) -
                sin(31 downto 16)*sin(31 downto 16))*evalue_p_in(q)+
                cos(31 downto 16)*sin(31 downto 16)*(evalue_p_in(p)-
                evalue_q_in(q));
              evalue_p_temp(q) <= var8(59 downto 28);

              counter <= 0;
              valid_output <= '1';
              state <= st_idle;
            end if;

          when st_idle =>
           if valid_input = '1' then
             valid_output <= '0';
             state <= st_busy;
           else
             state <= st_idle;
           end if;
       end case;
     end if;
   end if;
  end process;
end Behavioral;
```

Calculation of the error function. File name: off.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT master thesis-PCA Jacobi              --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS-2018                                --
-- Error function calculation for Jacobi          --
----------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
library work;
use work.array_pkg.all;
```

```
entity off is
  generic(
    constant matrix_size  : positive := 100;
    --n is the number of computational slices
    constant n            : positive := 4
  );
  port(
    valid_input : in std_logic;
    row_p       : in row_column_array;
    row_q       : in row_column_array;
    p           : in natural range 0 to matrix_size-1;
    q           : in natural range 0 to matrix_size-1;
    clock       : in std_logic;
    reset       : in std_logic;
    enable      : in std_logic;
    error       : out signed(71 downto 0);
    valid_output: out std_logic
  );
end entity;

architecture rtl_err of off is

  type state_type is (st_idle, st_busy);
  signal state : state_type;

  signal counter      : integer;
  signal error_signal : signed(71 downto 0);
begin
--output assignment
  error <= error_signal;

accum: process(clock, reset)
  variable error_variable   : signed(71 downto 0);
    begin
      if reset = '1' then
        error_signal <= (others => '0');
        valid_output <= '0';
        state <= st_idle;
        counter <= 0;
      else
        if clock'event and clock = '1' then
          if enable = '1' then

            case state is
            --pipelined calculation
               when st_busy =>
                error_variable := error_signal;
```

```
                for i in 0 to n-1 loop
                  if (n*counter+i /= p)  then
                  --multiply accumulate in Frobenius formula
                    error_variable := error_variable +
                          row_p(n*counter + i) * row_p(n*counter + i);
                  end if;
                  if (n*counter+i /= q) then
                    error_variable := error_variable +
                          row_q(n*counter + i) * row_q(n*counter + i);
                  end if;
                end loop;
                error_signal <= error_variable;
                if (counter < (matrix_size/n)-1) then
                   counter <= counter + 1;
                   state <= st_busy;
                else
                   counter <= 0;
                   valid_output <= '1';
                   state <= st_idle;
                end if;

               when st_idle =>
                if valid_input = '1' then
                  valid_output <= '0';
                  error_signal <= (others => '0');
                  state <= st_busy;
                else
                  state <= st_idle;
                end if;

            end case;

          end if;
        end if;
      end if;
    end process;
end architecture rtl_err;
```

Choosing the following indices for the Jacobi module. File name: choose_pq.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT master thesis-PCA jacobi              --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS  2018                               --
-- Choosing (p,q) for cyclic Jacobi               --
----------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity choose_pq is
  generic (
    constant matrix_size  : positive := 100
  );
  port (
    clock : in std_logic;
    reset : in std_logic;
    enable: in std_logic;
    p_out : out natural range 0 to matrix_size-1;
    q_out : out natural range 0 to matrix_size-1
  );
end choose_pq;

architecture rtl_pq of choose_pq is

  signal p_temp : natural range 0 to matrix_size-1;
  signal q_temp : natural range 0 to matrix_size-1;

begin

  p_out <= p_temp;
  q_out <= q_temp;

  path: process(reset, clock)
  begin
    if reset = '1' then
      p_temp <= 0;
      q_temp <= 1;
    else
      if clock'event and clock = '1' then
        if enable = '1' then
          if((p_temp = matrix_size-2)and(q_temp = matrix_size-1))then
            p_temp <= 0;
            q_temp <= 1;
          elsif(q_temp < matrix_size-1) then
            q_temp <= q_temp + 1;
          elsif(q_temp = matrix_size-1) then
            p_temp <= p_temp + 1;
            q_temp <= p_temp + 2;
          end if;
        end if;
      end if;
    end if;
```

```
  end process path;
end rtl_pq;
```

Inferred Block RAM. File name: Block_RAM.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT master thesis-PCA Jacobi              --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS-2018                                --
-- Block RAM inference for Jacobi                 --
----------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_unsigned.all;

entity Block_RAM is
  generic(
    BRAM_size  : integer := 10000;
    BRAM_width : integer := 32;
    ADDR_width : integer := 14
  );
  port(
    clock   : in  std_logic;
    din     : in  std_logic_vector(BRAM_width-1 downto 0);
    we      : in  std_logic; -- write enable
    address : in  std_logic_vector(ADDR_width-1 downto 0);
    dout    : out std_logic_vector(BRAM_width-1 downto 0)
  );
end Block_RAM;

architecture behav of Block_RAM is
  --RAM declaration
  type ram_type is array (0 to BRAM_size-1) of
                              std_logic_vector(BRAM_width-1 downto 0);
  signal RAM : ram_type;

  --Block RAM attribute
  attribute ram_style : string;
  attribute ram_style of RAM : signal is "block";

begin
  process(clock)
  begin
    if rising_edge(clock) then
      if we = '1' then
```

```
        --write operation
        RAM(conv_integer(address)) <= din;
      else
        --read operation otherwise
        dout <= RAM(conv_integer(address));
      end if;
    end if;
  end process;
end behav;
```

Package for the input-output array declaration for the Jacobi. File name: array_pkg.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT master thesis-PCA Jacobi              --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS-2018                                --
-- package for IO matrices for Jacobi             --
----------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package array_pkg is

type row_column_array is array(99 downto 0) of signed(31 downto 0);

end package;
```

Covariance modules. Implemented in the specialization Project, modified for the thesis project.

Datapath for the covariance calculation. File name: cov_datapath.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT specialisation project-PCA covariance --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS  Dec, 2017                          --
-- Updated for the master thesis, 2018            --
-- Covariance calculation - the datapath          --
----------------------------------------------------

library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cov_datapath is
  generic(
    constant width_array : positive := 16;
    constant depth_array : positive := 2500;   --components, pixels
    constant planes      : positive := 100;    --equals to matrix size
    constant buffers     : positive := 4       --number of buffers
  );
  port(
    x             : in  std_logic_vector(width_array-1 downto 0);
    clock, reset  : in  std_logic;
    retention     : in  std_logic;
    ctrl, enable  : in  std_logic;
    y             : out signed(31 downto 0));
end cov_datapath;

architecture behav of cov_datapath is
  type buffer_array is array (depth_array - 1 downto 0) of
                               std_logic_vector(width_array-1 downto 0);
  signal buffers_m : buffer_array;

  signal y_driver : signed(31 downto 0);
  signal mac_in   : signed(width_array-1 downto 0);
begin
  --continuous assignment
  mac_in <= signed(x) when ctrl = '1' else
                                   signed(buffers_m(depth_array - 1));
  y      <= y_driver;
  -- Clocked
  data_path : process(reset, clock)
  begin
    if reset = '1' then
      y_driver  <= (others => '0');
      buffers_m <= (others => (others => '0'));
    else
      if clock'event and clock = '1' then
        if enable = '1' then
          buffers_m <=
          buffers_m(depth_array-2 downto 0) & std_logic_vector(mac_in);
          if (retention = '1') then
            y_driver <= y_driver + signed(x)*mac_in;
          else
            y_driver <= signed(x)*mac_in;
          end if;
        end if;
```

```
      end if;
    end if;
  end process data_path;
end behav;
```

Covariance top level-control path. File name: covariance_optimized.vhd

```
----------------------------------------------------
-- Aysel Karimova                                 --
-- SMALLSAT specialisation project-PCA covariance --
-- Supervisors: Kjetil Svarstad, Milica Orlandic  --
-- NTNU-EMECS  Dec, 2017                          --
-- Updated for the master thesis, 2018            --
-- Covariance module calculation - top level      --
----------------------------------------------------

--Control path with the instantiation of data path
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use work.covariance_pkg.all;

entity covariance_optimised is
  generic(
    constant data_width  : positive := 16; --size of 1 input component
    constant depth_array : positive := 2500;--# of components 1 plane
    constant planes      : positive := 100; --# of planes, e.g.100
    constant buffers     : positive := 4 --# of buffers
  );
  port(
    x            : in  std_logic_vector(data_width - 1 downto 0);
    clock, reset : in  std_logic;
    enable       : in  std_logic;
    valid_input  : in  std_logic;
    done         : out std_logic;
    valid        : out std_logic_vector(buffers - 1 downto 0);
    column, row  : out output_row_column(buffers - 1 downto 0);
    y            : out output_t(buffers - 1 downto 0));
end covariance_optimised;

architecture behav of covariance_optimised is

  type state_type is (st_busy, st_idle);
  signal state : state_type;

  --intermediate signal declarations
```

```
   --components on each plane
  signal count_values  : natural range 0 to depth_array;
  --# of input planes; each = one row in the cov matrix
  signal count_rows    : natural range 0 to planes-1;
  signal valid_driver  : std_logic_vector(buffers - 1 downto 0);
  signal count_columns : natural range 0 to planes-1;

  signal column_temp    : output_row_column(buffers - 1 downto 0);
  signal row_temp       : output_row_column(buffers - 1 downto 0);
  signal enable_driver  : std_logic_vector(buffers - 1 downto 0);
  signal ctrl_driver    : std_logic_vector(buffers - 1 downto 0);
  signal retention_driver: std_logic;

  component cov_datapath
    port(
      x             : in  std_logic_vector(15 downto 0);
      clock, reset  : in  std_logic;
      retention     : in  std_logic;
      ctrl, enable  : in  std_logic;
      y             : out signed(31 downto 0));
  end component;

begin

  Datapath_generator : for i in 0 to buffers - 1 generate
  begin
    datapath_inst : cov_datapath
      port map(
        x         => x,
        clock     => clock,
        ctrl      => ctrl_driver(i),
        enable    => enable_driver(i),
        retention => retention_driver,
        reset     => reset,
        y         => y(i));
  end generate Datapath_generator;

  retention_driver <= '1' when (count_values /= 0) else '0';
  valid            <= valid_driver;

  column <= column_temp;
  row    <= row_temp;

  --sequential logic

  control_path : process(reset, clock)
  begin
```

```
if reset = '1' then
  state         <= st_idle;
  count_values  <= 0;
  count_rows    <= 0;
  count_columns <= 0;
  done          <= '0';
  column_temp   <= (others => 0);
  row_temp      <= (others => 0);
  valid_driver  <= (others => '0');
  enable_driver <= (others => '0');
  ctrl_driver   <= (others => '0');
elsif clock'event and clock = '1' then
  if enable = '1' then
    case state is
      when st_busy =>
        count_values <= count_values + 1;
        valid_driver <= (others => '0');
        if (count_values = depth_array - 1) then
          --shift ctrl one left
          ctrl_driver   <=
                      (ctrl_driver(buffers - 2 downto 0) & '0');
          --shift enable one left
          enable_driver <=
                      (enable_driver(buffers - 2 downto 0) & '1');
          -- All enabled data paths in previous plane
          --have now finished computation
          valid_driver  <= enable_driver;
          count_values  <= 0;
          -- to output rows and columns to use in bram
          for i in 0 to buffers - 1 loop
            if (enable_driver(i) = '1') then
              column_temp(i) <= count_columns;
              row_temp(i) <= count_rows + i;
            else
              column_temp(i) <= 0;
              row_temp(i) <= 0;
            end if;
          end loop;
          if (count_rows < planes-buffers and
                              count_columns=(planes-1)) then
            ctrl_driver   <=
                        std_logic_vector(to_unsigned(1, buffers));
            enable_driver <=
                        std_logic_vector(to_unsigned(1, buffers));
            count_columns <= count_rows + buffers;
            count_rows    <= count_rows + buffers;
          elsif (count_rows = planes-buffers and
```

```
                                        count_columns=(planes-1)) then
              done    <= '1';
              state   <= st_idle;
            else
              count_columns <= count_columns + 1;
            end if;
          end if;
        when st_idle =>
        --finished! Get ready for next computation
          count_values  <= 0;
          count_rows    <= 0;
          count_columns <= 0;
          enable_driver <= (others => '0');
          ctrl_driver   <= (others => '0');
          valid_driver  <= (others => '0');
          if valid_input = '1' then
            done         <= '0';
            enable_driver<= std_logic_vector(to_unsigned(1,buffers));
            ctrl_driver  <= std_logic_vector(to_unsigned(1,buffers));
            state <= st_busy;
          else
            state <= st_idle;
          end if;

      end case;
    end if;
  end if;
end process control_path;
end behav;
```

Package for the covariance module. File name: covariance_pkg.vhd

```
------------------------------------------------------
-- Aysel Karimova                                   --
-- SMALLSAT specialisation project-PCA covariance   --
-- Supervisors: Kjetil Svarstad, Milica Orlandic    --
-- NTNU-EMECS  Dec, 2017                            --
-- Updated for the master thesis, 2018              --
-- covariance module - package                      --
------------------------------------------------------

library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

package covariance_pkg is
```

```
  type output_t is array (natural range <>) of signed(31 downto 0);
  type output_row_column is array (natural range <>) of integer;
end package;
```