LETOURNEAU UNIVERSITY
SCHOOL OF ENGINEERING AND ENGINEERING TECHNOLOGY

FPGA IMPLEMENTATION AND HARDWARE OUTPUT COMPARISON OF FASTICA AND JADE BSS ALGORITHMS

by

Timothy S. Hong

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in the Discipline of Electrical Engineering

Thesis Advisor:

Dr. Joonwan Kim

LeTourneau University

2016

LETOURNEAU UNIVERSITY

SCHOOL OF ENGINEERING AND ENGINEERING TECHNOLOGY

FPGA IMPLEMENTATION AND HARDWARE OUTPUT COMPARISON OF FASTICA AND JADE BSS ALGORITHMS

by

Timothy S. Hong

A THESIS

APPROVED FOR THE DISCIPLINE OF

ELECTRICAL ENGINEERING

By Thesis Committee

_____, Chair

Dr. JoonWan Kim, PhD

_____

Dr. Paul Leiffer, PhD

_____

Dr. Marian Iordache, PhD

ABSTRACT

The problem of blind source separation (BSS) is a one which occurs in a variety of different applications. For example, electrical readings of the brain can be separated by type using a BSS algorithm. This can help to isolate specific signals and dramatically increase the effectiveness of signal analysis in such applications. There are a variety of BSS algorithms available to us today. The most popular three are the joint approximation and diagonalization (JADE) (Cardoso, 1993), Infomax (Bell, 1995), and FastICA (Hyvärinen, 1999) algorithms.

This thesis is concerned with the field programmable gate array (FPGA) hardware implementation and output comparison of two common BSS algorithms: FastICA and JADE. While these algorithms can currently be used in many practical applications with real sampled data, this approach is potentially slow and inefficient. A proper hardware implementation of a BSS algorithm leads to speed, efficiency, and convenience. While some BSS algorithms have been implemented previously on hardware platforms, there is still research to be done to confirm the effacacy of existing implementations or explore new implementations. Many of the hardware implementations simulate the algorithms at a low-level, but do not actually implement and test the hardware with live signals. Hardware implementations of both the JADE and FastICA algorithms, and their analysis are presented in the remainder of this thesis.

Contents

LIST OF FIGURES

LIST OF EQUATIONS

DEFINITIONS USED

CHAPTER 1

INTRODUCTION

1.1 Background

In almost any area of scientific research we commonly encounter cases in which we are unable to decipher sampled data due to signal noise or cross-interference from adjacent sources. Many methods have been employed to remove undesired interference such as adaptive noise cancellation (ANC), beamforming (elucidated in section 1.4), or blind source separation (BSS). A couple example applications include separation of a fetus's heartbeat from the mother's or separation of electromagnetic signals sampled through an array of antennas.

ANC removes noise from a signal based on second order information, beamforming allows one to isolate different signals based on directionality, and BSS separates out statistically independent components from a set of incoming signals. In this paper we will focus on the implementation of independent component analysis (ICA) algorithms and their application to BSS.

1.1.1 Blind Source Separation

BSS is a technique for separating sources given different linear mixtures of those sources without prior knowledge about how those sources were mixed. Each mixture is referred to as an observation, and the mixing process is modelled using a matrix transformation.

A great example to illustrate the mixing process is the cocktail party problem. Consider a cocktail party taking place in a large room. If we place several microphones at various locations around the room, each one will get a different combination of unique voices or sounds within the room. If the recording from one of these microphones was played back to a person, it is unlikely that the listener would be able to distinguish what any unique voice in the recording is saying because it is a mixture of so many different voices. The samples captured by one of these microphones is considered to be an observation while the set of sampled data is considered to be a set of observations.

Mathematically speaking, the incoming data is represented as an array where the columns correspond to data points and the rows correspond to different source signals. The mixing process then is carried out as a matrix multiplication between the incoming data matrix and the mixing model matrix, where the rows of the mixing matrix represent a different combination of the set of signal sources. Other details of BSS will be discussed in later chapters.

In general, BSS can extract as many sources or independent components as there are observations; if there are three observations (mixtures of sources), then a maximum of three sources can be extracted. Another limitation of BSS is that observations must be separated based on statistical information only, which can be less effective in contrast to beamforming which uses information about the data capture system.

Beneath BSS we have many different sub-categories or techniques. These include but are not limited to the following: principal component analysis (PCA), ICA, and stationary subspace analysis (SSA). PCA algorithms effectively orthogonalize, or de-correlate, the set of observations. ICA algorithms, which commonly use PCA as a pre-processing stage, separate the sources into additive sub-components. It should also be noted that ICA algorithms can separate out, at most, one Gaussian source, and the rest should be non-Gaussian. SSA algorithms separate the set of observations into stationary and non-stationary

sub-components.

1.1.2 Independent Component Analysis

As our focus is the implementation of ICA algorithms applied to BSS, we elucidate the relationship between the two. ICA takes the set of observed signals and calculates a transformation which maximizes the independence of each source. The transformation is output as a matrix (referred to as a separating matrix) which is the theoretical inverse of the matrix used to model the mixing process. The outcome of ICA is used to calculate the outcome of BSS. Figure 1.1 illustrates the relationship between BSS and ICA for the case of two observations.

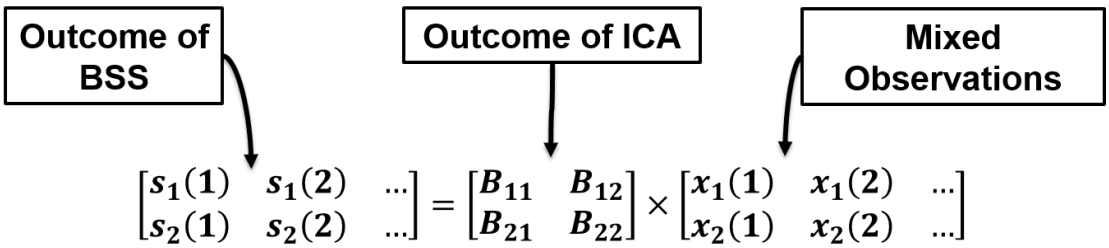

Figure 1.1: Relationship between ICA and BSS in terms of matrix math.

The class of ICA algorithms is itself vast. At the top level we can categorize algorithms as those which focus on convolutive mixtures of observations or those which focus on linear mixtures. For our research we focus on algorithms which assume a linear mixing model.

Among the linear-assumption ICA algorithms more than 15 different algorithms have been proposed. Of all the possible algorithms, three stand out as the most popular; the FastICA algorithm proposed by Hyvärinen in 1999 [1], the Joint Approximation and Diagonalization of Eigenmatrices (JADE) algorithm proposed by Cardoso and Souloumiac in 1993 [2], and the Infomax algorithm proposed by Bell and Sejnowski in 1995 [3]. Between the three, JADE

is best suited for a hardware implementation due the parallel nature of the algorithm. The other two are iterative in nature and show similarly reliable ICA results, but to limit the scope of this paper we select the FastICA algorithm to be used in a comparative study of the hardware implementations.

The FastICA algorithm was selected because of its popularity and because it exhibits high performance with very simple calculations, while the JADE algorithm was selected because, while it shows great performance in simulation, many have not attempted to implement it on hardware due to computational complexity. Many hardware implementations of the FastICA algorithm are available for comparison with our implementation, while not many implementations of the JADE algorithm are available which opens the grounds for a proof of feasibility implementation.

A proof of feasibility of Field Programmable Gate Array (FPGA) implementation of FastICA has been done for masters level thesis work in [4] and [5]. In this thesis a proof of feasibility of implementation of the JADE algorithm is presented along with another pipelined implementation of the FastICA algorithm. In contrast to Behera [4] and Taha [5] who stop their research at hardware simulation, both ICA algorithms presented in this thesis were physically implemented on an FPGA, and tested with real lab-generated analog signals.

1.2 Field Programmable Gate Arrays

An FPGA is a device which can be programmed with different digital circuits by developers after the chip has been manufactured. This is different from application specific integrated circuits (ASICs) in that an FPGA is not limited by the original design. Instead, an FPGA can be programmed for one function, and then re-programmed at a later time for another. FPGA devices are extremely fast, can be application-optimized, and are easily reprogrammed making them the perfect devices for high speed digital signal processing (DSP)

applications.

Another device similar to an FPGA is a Complex Programmable Logic Device (CPLD). A CPLD has a different internal makeup than an FPGA, but is used in a very similar way. Another major difference between the two is that CPLDs retain their configuration when the power is disconnected. FPGAs are volatile in that they lose their configured programming as soon as power is lost. The average FPGA can hold a significantly larger design than the average CPLD and can potentially operate at much higher speeds. The main reason we use an FPGA for our implementation instead of a CPLD is because of the advantages of larger available design space and faster operating speed.

FPGAs consist of three basic elements: logic cells, IO blocks, and interconnects. Logic cells hold the programmable logic. Each logic cell includes a manufacturer-specified number of look-up tables (LUT) and flip-flops. Sometimes FPGA resources are also reported in terms of logic slices or configurable logic blocks (CLBs) which consist of two logic cells and four logic cells respectively.

When a design which is written in a hardware description language such as Verilog or VHDL undergoes synthesis, a process analogous to compiling in higher level programming languages, a bit-stream is generated. This bit-stream can be programmed into the specified type of FPGA and can then be run in a real-world electrical system.

One attribute of note about FPGAs is that their programmed memory is volatile. This means that any design loaded into an FPGA is lost whenever the FPGA loses power. In many cases configuration memory is also included on the FPGA board which allows the design to be re-programmed upon restoration of power.

1.3 Motivation

FPGA designs are intrinsically parallel, allowing for complex computations to be carried out in a very short time. By designing a program

correctly, one can maximize the amount of calculation occurring in parallel, therefore maximizing the speed at which the processing is carried out. Because of the high computational cost of ICA algorithms, we selected an FPGA as the target hardware implementation platform to reap the benefits of high speed, parallel computation.

1.3.1 Application

As stated previously, one of the main motivations for this research is the many different areas in which BSS can be applicable. Aside from the two examples of separating a fetus and mother's heartbeat and separating electromagnetic signals captured by an array of antennas, there are a multitude of useful applications. Another example is the separation of different audio signals sampled from an array of microphones. Having a clean fetus's heartbeat help give medical professionals more accurately detect issues with the fetus's condition. Extracting unique electromagnetic signatures from samples collected by an array of antennas can improve the efficacy of radar systems, opening potential military applications. Separation of audio sources has application in many sub-areas such as in the music industry or in espionage.

These three examples are a dramatically insignificant portion of the possibilities. BSS can be used in any situation where the extraction of statistically independent patterns is useful. Another very interesting application of note is the applicability of BSS in compression algorithms. One can increase the periodicity of a set of signals to improve the efficacy of existing compression algorithms which intrinsically rely on repetitive patterns within the data which is being compressed. One such compression method that could benefit from pre-processing with BSS is the Lempel-Ziv-Welch algorithm.

Several practical examples also exist which demonstrate the use of BSS in a real-world system. In [6] the authors apply BSS techniques in cochlear implants to improve test subjects' ability to recognize speech in a noisy

environment. To test recognition, a set of phonetically balanced sentences from the IEEE database were used. A talker would read selected sentences aloud in the presence of another selected sentence read by another talker deemed the masker. Their experiments showed an improvement of speech recognition after processing using BSS.

A document titled "Applications of Independent Component Analysis" [7] presents a compilation of many different applications of ICA. Topics range from improvement of current cell-phone technology to use of ICA in decision tree implementation. There are a total of 20 different authors and five articles on different applications of ICA.

Another important note when considering the application of BSS is the number of incoming sources or observations that the algorithm will be processing. Some applications work with fewer than 5 signals while other applications may require an algorithm which can process as many as 40 or 50 signals. Implementations can be adjusted accordingly to maximize speed or minimize hardware resource usage.

1.3.2 BSS History

One of the earliest methods of BSS, proposed in 1986 by Jutten and Herault, used neural networks [8]. This method is called the Neuromimetic Method, and was further elucidated upon in 1991 by Jutten and Herault , and also in 1991 by Comon [9], in 1992 by Cichocki and Meszczynski [10], and in 1993 by Karhunen and Joutsensalo [11]. Neuromimetic Methods attempt to capture the powerful visual processing capabilities of neural networks.

In 1990, just four years after the proposed Neuromimetic Method, Gaeta and Lacoume proposed a Maximum Likelihood Estimation approach to the problem of source separation [12]. Another two years after that in 1992, Becker and Hinton [13], and Linsker [14], in two separate papers, proposed another method which was based on Information Theory. This method is similar to an

idea proposed by Horace Barlow in 1961. In his study of the nervous system of frogs he proposed that the purpose of the visual system is to reduce redundancy in the information entering a frogs eye. In a sense, the information theoretic method pursues this same idea with the exception that it can be applied to non-visual signals as well if necessary.

The concept of ICA was first proposed by Comon in 1994 [15]. While this was not the birth of the concept itself, it assigned a title to the existing concept. The JADE algorithm proposed by Cardoso and Souloumiac in 1993 [2] is a type of ICA algorithm although the title, ICA, had not necessarily come into common use. In 1995, Bell and Sejnowski proposed their Infomax algorithm [3]. Following that, in 1997, a generalized ICA learning rule was derived by Pearlmutter and Parra from the maximum likelihood estimation (MLE) [16]. Improving on this in 1996 and 1997, the learning rule was optimized for relative and natural gradient cases. Finally, in 1999, the wildly popular FastICA fixed-point algorithm was proposed by Hyvärinen [1].

The history of BSS in DSP applications goes back much further, even as far as the 1960s. However, the brief history above includes the major highlights leading towards the development of the two algorithms which we will be focusing on in the rest of this thesis: JADE and FastICA.

1.4 Related Fields

Some related fields of research include beamforming and ANC. Mathematically speaking, BSS achieves the same goals as beamforming, the only difference being that in BSS, information about the data capture system is not known nor is it used.

Beamforming aims to separate out independent sources from signal capture systems such as microphone or antenna arrays using information about the capture points. For example, in a two microphone system, if an audio signal is detected first at one microphone and then the other, we know that the source

is located closer to the first microphone. Beamforming is widely used in military applications to selectively tune into different electromagnetic sources.

In contrast to beamforming, BSS operates solely on the statistical information found within the observations. No information about the location of the capture points is used by the algorithm.

When applying BSS to noise cancellation, the algorithm may be in competition with ANC algorithms. The main difference between the two is that ANC de-correlates noise from signals, operating on second order statistics, while BSS achieves separation using fourth order statistics. In terms of separation quality, BSS will remove noise from a signal with much higher accuracy, but this is at the cost of computational time and complexity. To put this into perspective, an ANC algorithm can require more than five times fewer resources when implemented on an FPGA than a BSS algorithm designed for the same purpose. It should be noted however that BSS algorithms have the ability to separate out more than one signal simultaneously whereas ANC algorithms are limited to decorrelation of one signal from another, the output result being one 'cleaned' signal.

1.5 Research Objectives

This research has several main objectives as follows:

1. Investigate optimizations for software versions of the ICA algorithms.
2. Apply hardware-efficient architectures and optimizations in the FPGA implementation of both algorithms.
3. Implement algorithms on an Artix-7 XC7A100T FPGA, interface with an ADC/DAC pair, and test with real-time lab-generated analog inputs.
4. Develop a way to numerically compare software and hardware algorithms with each other.

In achieving these goals, we will be paving the way for future work to improve on the hardware implementations developed through this project. Improvements include implementation of the same algorithms using different contrast functions (functions which generate a measure of error between inputs) or expanding the implementations to handle a larger number of input signals. The latter is an area of particular focus as the complexity of processing increases drastically with a larger number of input sources thereby slowing down algorithm operation and requiring more hardware resources which may not be available in such large quantities in an appropriately priced FPGA.

1.6 Thesis Organization

This thesis is organized into seven chapters. The second chapter includes more detailed information on BSS, ICA, and both algorithms. Chapter three presents the software versions of both algorithms followed by hardware implementations in chapter four. Chapters five and six present simulation and hardware results respectively. Finally in chapter seven we end with a conclusion and a discussion about future work.

CHAPTER 2

BLIND SOURCE SEPARATION

2.1 The BSS System

The BSS system can be split into three processes and four data sets. We begin with the input data, followed by the input chain or mixing system which generates the sample observations. These observations are then centered and whitened to generate our whitened observations which are used in the separating process. It is at this stage that the ICA algorithms come into play to generate the final data set, the estimated sources based on the whitened observations. Figure 2.1 is a graphical representation of the BSS system. Each of the four data sets are plotted as graphs, and each process is depicted between the data sets.

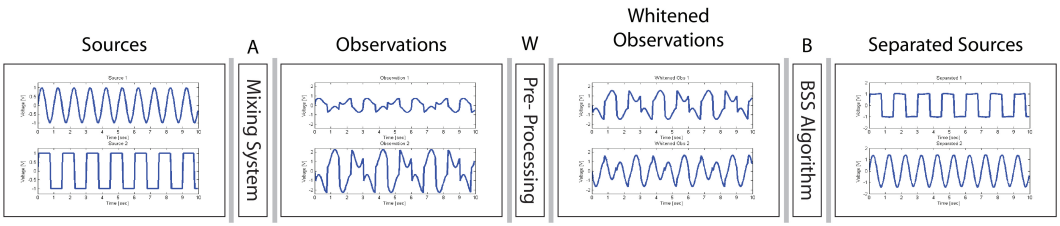

Figure 2.1: BSS System Overview

2.2 Matrix Math

Recall matrix multiplication. It is helpful in understanding the ICA algorithms if one is able to think easily in terms of matrix mathematics. The

following is an example of a matrix transformation:

$$x(t) = A \times s(t) \tag{2.4}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} s_1(1) & s_1(2) & \dots & s_1(T) \\ s_2(1) & s_2(2) & \dots & s_2(T) \end{bmatrix} = \begin{bmatrix} x_1(1) & x_1(2) & \dots & x_1(T) \\ x_2(1) & x_2(2) & \dots & x_2(T) \end{bmatrix} \tag{2.1}$$

The matrix A of dimension $[2 \times 2]$ transforms the matrix s of dimension $[2 \times T]$ into matrix x of dimension $[2 \times T]$. Mathematically speaking, each term in x is calculated based on the terms in A and s by the following equations:

$$x_1(t) = a_{11} \times s_1(t) + a_{12} \times s_2(t) \tag{2.2}$$

$$x_2(t) = a_{21} \times s_1(t) + a_{22} \times s_2(t) \tag{2.3}$$

For higher dimensions the same pattern is used. Recalling the relationship between matrices when carrying out matrix multiplications will help one understand both ICA algorithms and the model by which the BSS problem is described. Matrix multiplication is also another area for improvement in our hardware implementation. We are able to apply systolic arrays, homogeneous networks of interconnected processors, to reduce the amount of internal resources requred to perform expensive matrix calculations. Doing so removes the need to

2.3 The Mixing Model

The mixing model for which the JADE and FastICA algorithms were designed is a linear instantaneous mixture. This means we are assuming all sources are the same propagation time-distance away from the sampling apparatus, and that the mixed sample signals are each a different linear combination of the source signals. In a mathematical sense, the mixing model

can be written as follows:

$$x(t) = A \times s(t) \tag{2.4}$$

Where n is the number of sources, T is the number of samples to be processed, the $x(t)$ term is a $[n \times T]$ matrix holding the outputs of the mixing process, A is an $[n \times n]$ matrix representing the mixing process (referred to as the mixing matrix), and $s(t)$ is a $[n \times T]$ matrix containing the points of the original signals. In many BSS applications, the linear assumption holds true since the actual delay between signals is negligible. For example, in an audio signal separation case, the signals received by two or more different microphones will have negligible propagation delay, except in the unlikely case in which the microphones are physically separated by a significant distance.

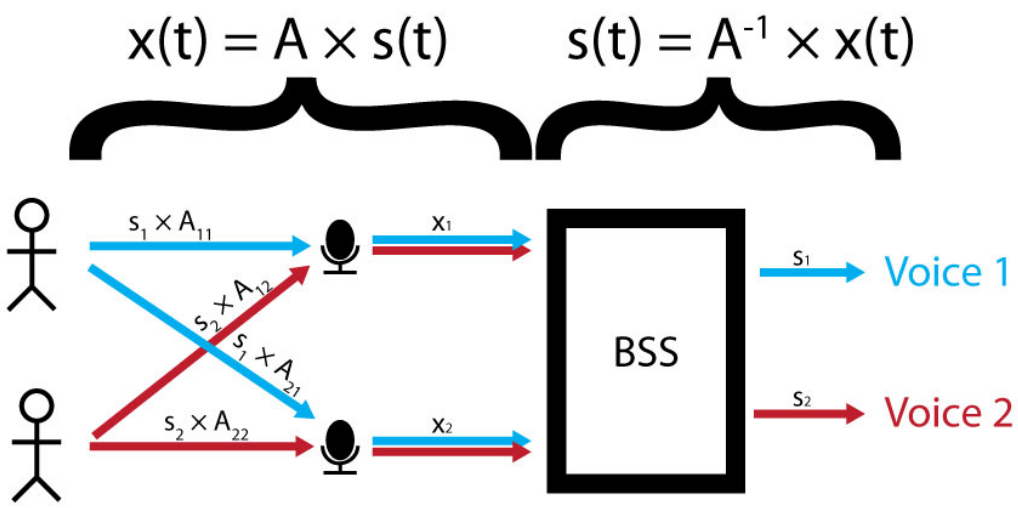

Figure 2.2: BSS Block Diagram

It may be easier to understand the mixing model given Figure 2.2. Person one and person two have their own unique voice referred to, in this case, as a source and denoted by s_1 and s_2 respectively. The elements of the mixing matrix A each represent a unique data path and are multiplied by the appropriate source to simulate the effects of each propagation path. Following this, each

post-propagation source is added to simulate how a microphone would capture both sources simultaneously. The sum of the post-propagation sources generates an observation denoted using x_i. Finally, the observations x_1 and x_2 are used in an ICA algorithm to estimate the inverse of the mixing matrix A^{-1} in the form of a separating matrix V which is used to separate out the original sources from the set of observations.

2.4 Pre-Processing (Centering and Whitening)

Depending on the contrast function being used by the ICA algorithm, it may be necessary to carry out some pre-processing on the dataset.
Pre-processing typically includes centering and whitening. Within the context of figure 2.2 the pre-processing happens occurs in the beginning part of the BSS block.

A signal is considered centered when the average of the signal over a given set of samples is equivalent to zero. Centering is carried out on each individual observation. To center the observation, simply subtract the observation sample average is subtracted from each observation sample. By doing this, the mathematical average of the signal will become equivalent to zero. In electrical terms this is equivalent to removing the DC component and leaving only the AC component.

Whitening a signal involves de-correlating the set of observations with respect to each other. The mathematical effect of the whitening processes is that the covariance matrix of the set of observations will be equivalent to the identity matrix. Whitening can be done by way of eigenvalue decomposition (EVD) on the sample covariance matrix. Transforming the set of observations by the resulting eigenvector matrix generates the whitened observations. The purpose of whitening in ICA is to ensure that each observation is treated equally by the ICA algorithm [17].

2.5 Contrast Functions and Mathematical Background

There are several different contrast functions which are used as bases for different ICA algorithms. The two of importance to us at this point are based on negentropy (used by FastICA) [1] and fourth-order statistics (used by JADE) [2].

Negentropy methods operate on the differential entropy of random variables as a measure of non-gaussianity. Given a set of variables of equivalent variance, the most gaussian variable will have the largest entropy. Therefore, a gaussian variable (generated by a gaussian source) will have a larger entropy than variables which contain information. The larger the entropy, the more random the variable, and the less likely it is that information is contained within the corresponding signal. FastICA takes the converse of the entropy (the negentropy) and tries to maximize that value. By maximizing the negentropy the gaussianity is minimized and the information is maximized resulting in a set of information-containing variables. These are our source signals.

Methods which are based on fourth-order statistics rely on approximate joint diagonalization techniques. Essentially, another contrast function is used on the fourth-order contrast function of choice. In the case of the JADE algorithm, cumulants are used as the contrast function. Cumulants are a measure of the interaction between variables. Another term for cumulants is the connected correlation. The fourth-order cumulant matrix is calculated first, then jointly approximately diagonalized. As a second-order cumulant matrix is of dimension $n \times n$, the fourth order cumulant matrix is of dimension $n \times n \times n \times n$. Therefore there are n^2 matrices of dimension $n \times n$ within the fourth-order cumulant matrix which are jointly diagonalized to generate the separating matrix (of dimension $n \times n$). This process is analogous to the whitening process. See Appendix A for more information about cumulants.

For the JADE algorithm used in the FPGA implementation, the Frobenius norm formulation was used as the contrast function. Future work may explore other contrast functions.

The negentropy method is applied to a fixed-point type algorithm by Hyvärinen [1]. A weight vector (the row or column within the separating matrix which corresponds to the source of interest) is selected to be iterated upon and a formula using any sort of moment-generating (non-quadratic, non-linear) function is applied at each iteration. Depending on certain conditions the weight vector will eventually converge to a solution which represents the coefficients required to extract a signal. The unique signal to be extracted from the set of observed signals is calculated via a weighted sum where the weights are the elements of the now-converged weight vector. This is in the case of the deflationary version of the FastICA algorithm as opposed to the symmetric one. The deflationary approach separates sources one-by-one, decorrelating the set of remaining sources as each unique source is separated. The symmetric approach separates sources simultaneously, similarly to the JADE algorithm.

2.6 BSS and ICA

While this thesis aims at implementing ICA algorithms, the purpose of those ICA algorithms is BSS. The relationship between the two is simple, but may be confusing since much of the literature refers to them interchangeably. In terms of definition, ICA aims to estimate the inverse of the system by which the observations were generated, while BSS is the technique by which the observations are decomposed back into the set of original sources. Mathematically speaking, the output of ICA is an estimated $n \times n$ matrix, while the output of BSS is a $n \times T$ matrix, where n is the number of sources, and T is the number of samples processed.

CHAPTER 3

ALGORITHMS AND SOFTWARE

3.1 FastICA

Proposed by Hyvärinen in 1999 in his paper titled "Fast and robust fixed-point algorithms for independent component analysis" [1], the FastICA algorithm quickly rose in popularity due to excellent performance characteristics and computational simplicity. FastICA is a fixed-point type algorithm meaning it is an iterative algorithm. In each iteration of the algorithm, a weight vector or matrix is updated based on Hyvärninen's proposed update equation. In the deflationary approach, vectors or columns of the separating matrix are calculated individually while in the symmetric approach all may be calculated simultaneously. For this thesis we focus on the deflationary approach.

As mentioned earlier, FastICA operates on a negentropy contrast function. The negentropy is calculated using any non-linear, non-quadratic function and the first and second derivatives of that function. According to Hyvärinen, a couple good functions to use are:

$$f(u) = \log(\cosh(u)) \tag{3.1}$$

or

$$f(u) = -\exp(-u^2/2) \tag{3.2}$$

The first and second derivatives of $f(u)$ are denoted $g(u)$ and $g\prime(u)$

respectively. For our specific test cases, we choose the following function:

$$f(u) = -e^{(-a_2 u^2/2)} \tag{3.3}$$

Taking the first and second derivatives of this function gives us:

$$g(u) = \frac{1}{a_2} e^{(-a_2 u^2/2)} \tag{3.4}$$

and

$$g\prime(u) = u \cdot e^{(-a_2 u^2/2)} \tag{3.5}$$

The FastICA algorithm can be broken into four basic steps. These steps involve a weight update and normalization function which are as follows:

$$w^+ = E\{x \cdot g(w^T x)\} - E\{g\prime(w^T x)\} \cdot w \tag{3.6}$$

and

$$w^* = \frac{w^+}{\|w^+\|} \tag{3.7}$$

The steps for the FastICA algorithm are outlined below:

First Initialize the weight vector w

Second $w^+ = E\{x \cdot g(w^T x)\} - E\{g\prime(w^T x)\} \cdot w$ 3.6

- $g(u) = \frac{1}{a_2} e^{(-a_2 u^2/2)}$ 3.4
- $g\prime(u) = u \cdot e^{(-a_2 u^2/2)}$ 3.5

Third $w^* = \frac{w^+}{\|w^+\|}$ 3.7

Fourth Check convergence and return to 2 if convergence conditions have not been met

To check for convergence we take the last weight vector w and compare it with the new estimate w^*. If it hasn't changed, or has only changed a negligible amount, then convergence has been reached.

3.2 JADE

The JADE algorithm was originally proposed by Cardoso and Souloumiac in 1993 in a paper titled "Blind Beamforming for non-Gaussian Signals" [2]. This algorithm operates on the sample fourth-order cumulant matrix of dimension $n \times n \times n \times n$, jointly approximately diagonalizing the set of $n \times n$ symmetric matrices of which the fourth-order cumulant matrix is comprised. To better understand how the algorithm works, it is useful to visualize the cumulant matrix. Figure 3.1 is a three-dimensional representation of the four-dimensional cumulant matrix of order $n = 2$. In this case the JADE algorithm carries out a diagonalization on the 2×2 sub-slices, or sub-matrices, in a manner that approximates the optimum diagonalization across all of the slices. More information about cumulants can be found in Appendix A.

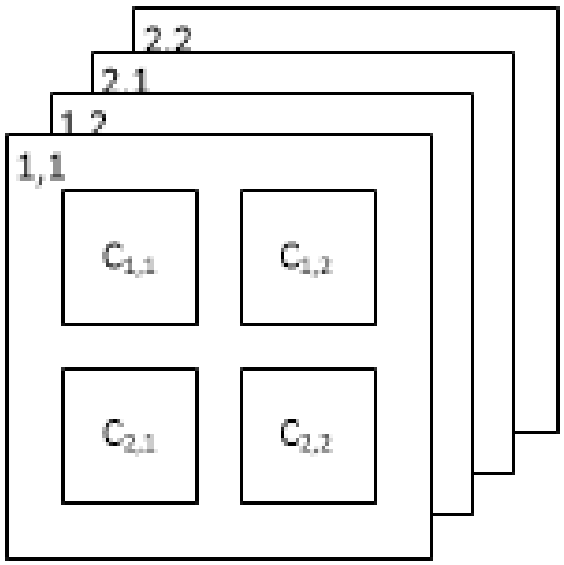

Figure 3.1: Fourth-order cumulant matrix for 2-observation case ($n = 2$)

The joint approximate diagonalization method used by the algorithm we will focus on in this thesis is based on the Frobenius norm formulation. The diagonalization process involves jointly approximately diagonalizing component 2×2 matrices in a Jacobi iteration pattern over the set of component matrices of dimension $n \times n$. In each case, the Frobenius norm formulation is used to diagonalize across all of the n^2 component matrices. The result of the joint approximate diagonalization is a matrix V referred to as the separating matrix which is of dimension $n \times n$ and which is a linear matrix transform by which the set of observations may be separated into a set of statistically independent components. This method of diagonalization is actually a type of QR factorization algorithm.

One result of the use of the Jacobi iteration scheme is that for a two-source, two-observation case the joint approximate diagonalization needs to run only one iteration for effective output. However, in increasingly larger cases, the diagonalization takes a correspondingly larger number of iterations to achieve convergence to an accurate diagonalization.

It should also be noted that this method requires the update of the cumulant matrix on which the diagonalization is operating as well as an update of the separating matrix V after each iteration. This adds to the computational load because each iteration is then burdened with extra overhead calculations.

3.3 Whitening

Whitening is a preprocessing step required by both the FastICA and JADE algorithms in which the set of n observations is orthogonalized with respect to each other. To accomplish this, we first calculate the covariance matrix of the set of observations. We then carry out eigenvalue decomposition (EVD) on the covariance matrix. The resulting eigenvector matrix is a linear matrix transform which whitens the set of signals from which the covariance matrix was calculated. The mathematical effect of this is that the covariance

matrix of the transformed set is equivalent to the identity matrix of dimension $n \times n$. This indicates that the set of transformed observations is de-correlated.

Whitening is another form of BSS; one which falls under the category of PCA. While PCA has been used to improve existing technologies, we still focus on researching ICA because the result of PCA does not necessarily return signals which resemble the original sources. Instead ICA algorithms, which maximize the non-Gaussianity of the separated sources must be used. Figure 3.2 shows the output of PCA and ICA both carried out on the same set of observations generated from a sinusoid and a square wave. In the far left columns are observation 1 and observation 2, in the middle column are the set of outputs after processing using PCA, and in the far right column are the separated sources after processing using ICA. Notice that the PCA output does not resemble a sine and square wave, but the ICA output does.

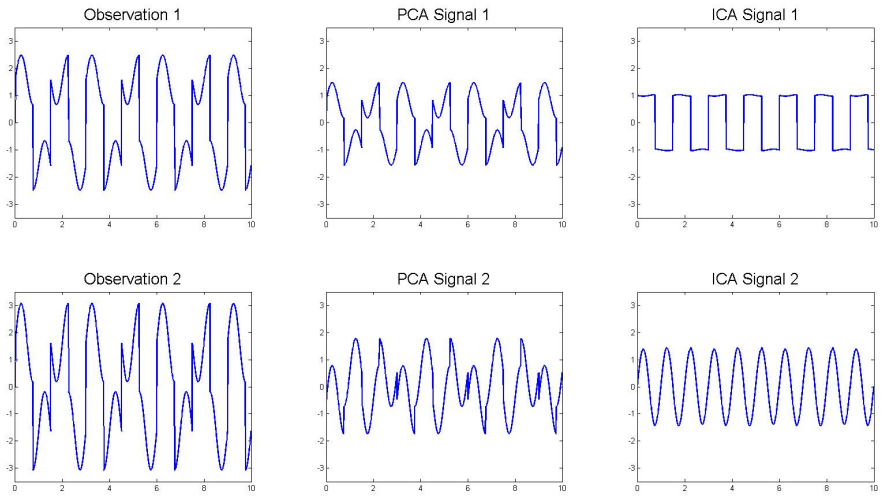

Figure 3.2: PCA vs. ICA

For ICA, whitening is paramount because the process ensures that all the observations are treated equally by the ICA algorithm. If they are treated unequally the result is a set of signals that do not resemble the original sources. For purposes of comparison we use the same whitening algorithm as a

preprocessing stage for both the FastICA and JADE algorithms.

3.4 Number Representation

There are two basic ways to represent numbers in a computer; fixed and floating point notation. Most personal computers are currently designed to use floating point notation because of the exceptional dynamic range. This is, of course, at the cost of uneven precision at different numerical ranges. Fixed point notation is commonly used in FPGA designs because when designed properly a fixed point system will run at a faster speed and require fewer resources. The following are some examples of fixed point representation of example decimal numbers:

- $54.75_{10} = 110110.11_2$
- $19.625_{10} = 1011.101_2$
- $3.1415_{10} = 11.0010010000111\ldots_2$

Recall that in binary, each digit represents a value equivalent to 2^i where i is the digit's position relative to the decimal point, and where the decimal point is located between digit positions $i = 0$ and $i = -1$. If one requires the use of large numbers, then many digits to the left of the decimal point should be used. Alternatively, if one requires the use of very small precision numbers, then many digits to the right of the decimal point should be used. The effective dynamic range and precision of floating point numbers is limited by the most significant bit and least significant bit respectively.

Floating point representation is more complicated in that the number is actually encoded further to allow for storage of a wider range of numbers with fewer bits. Floating point consists of a sign bit, an exponent, and a mantissa (also known as a significand). As an example we will consider the IEEE 754 single precision format.

The exponent and mantissa are defined with widths of 8-bits and 24-bits respectively. The 8 exponent bits store the exponent in excess-127 offset binary form meaning that the value of the exponent is calculated by subtracting 127 from the number represented by the 8-bits. This allows the exponent to hold any value between -128 and 127 which allows for a larger dynamic range centered about one. The mantissa is stored with a hidden most significant bit so that only 23 bits are actually required for storage. The decimal point of the mantissa is located between the hidden bit and the leftmost bit of the stored mantissa. Figure 3.3 depicts a single precision number and its equivalent decimal value to the right. Bits are ordered from 31 to 0 inclusively so that 32 bits in total are used. Single precision numbers are popular because they naturally fit with instruction widths of 32-bit computer architectures.

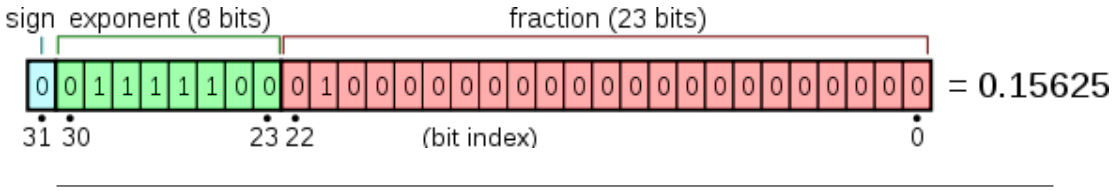

Figure 3.3: Floating Point Numbers

For our software algorithms we use the IEEE 754 single precision standard format because mathematical functions are already designed for 32-bits. The same notation is used in our hardware algorithms as well for the purpose of comparison.

CHAPTER 4

HARDWARE IMPLEMENTATION

4.1 Platform

When selecting a hardware platform on which to develop a high speed algorithm a few options rise to the top as the fastest and most cost-effective: Digital Signal Processors (DSPs), Field Programmable Gate Arrays / Application Specific Integrated Circuits (FPGAs / ASICs), and newer high speed computer architectures. For this project an FPGA was selected because of its intrinsically concurrent architecture along with the advantage of cost-effectiveness.

Specifically, the Artix-7 XC7A100T FPGA (Picture in Figure 4.1) by Xilinx was used via a Nexsys 4 FPGA development board from Digilent. The Nexsys4 FPGA development platform is packaged with many different peripherals, all built into the board itself and tied into dedicated pins on the FPGA, but for convenience and to handle high speed analog to digital conversion operations, a dedicated ADC/DAC combination was used. The ADC is packaged as the Pmod AD1 by Digilent which uses an AD7476 12-bit ADC, while the DAC is packaged as the Pmod DA2, also by Digilent, and uses a DAC121S101 12-bit DAC.

Analog circuitry (Picture in Figure 4.2) was also built to interface lab equipment such as signal generators, oscilloscopes, or microphones with the ADC and DAC. Simple low-noise operational amplifier circuits handled signal offset,

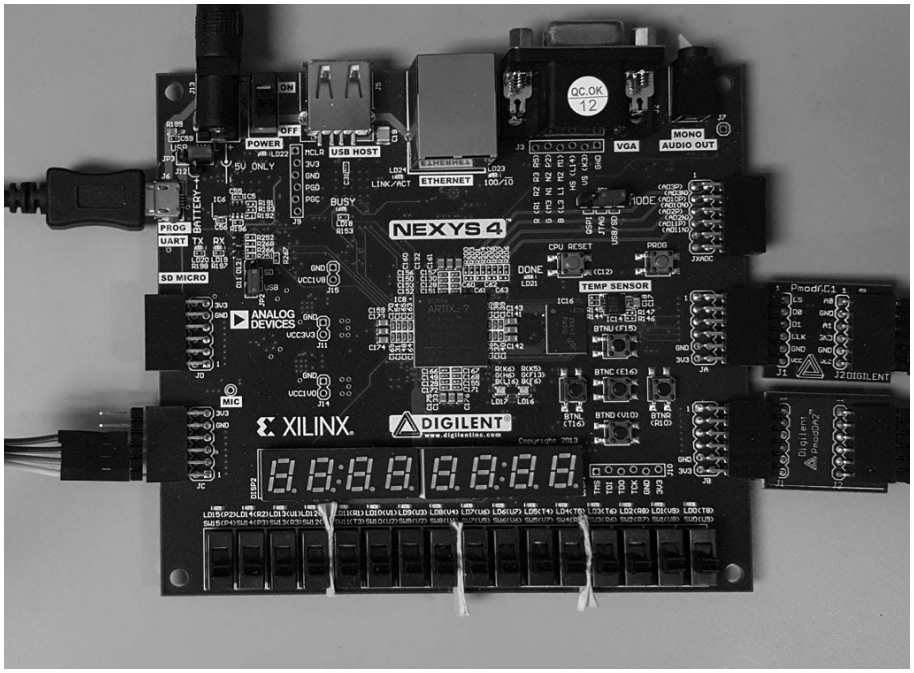

Figure 4.1: The Artix 7 FPGA board that was used

which entailed centering our incoming signals in the range of 0 to 3.3 Volts. Mixing circuits were also added to allow variable analog mixing of the incoming signals if necessary.

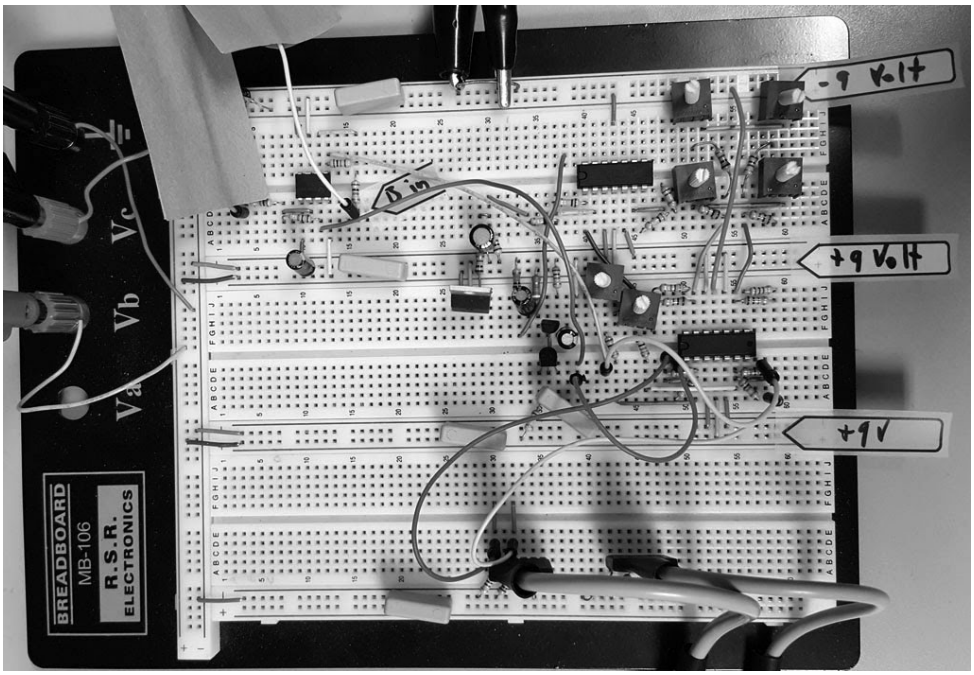

Figure 4.2: Circuit hardware used to generate mixed signals

4.1.1 Hardware Considerations

Three main issues arise when considering a hardware implementation of any algorithm. The first is how to take full advantage of the concurrent capabilities of an FPGA. This can be done by careful design of the hardware system. One such design pattern is a pipelined design in which the different hardware modules execute concurrently such that data can flow continuously through the entire system. The second is the amount of hardware space used by the algorithm chosen. This can be reduced by converting sections of the algorithm to fixed point or changing basic block strategies. For example, the attributes of a multiplier module/block may be adjusted to select a tradeoff between speed and resource usage. The third is the accuracy of numerical calculations. Many advanced algorithms require the calculation of higher order functions such as the exponential function or hyperbolic tangent (as in the case of the FastICA algorithm). These can be calculated using equivalent functions or approximations.

4.2 Algorithm Development

In the early stages of research, the development of algorithms entailed coding using the ieee_proposed numerical library, which is to be included officially in the VHDL-2008 standard, for access to fixed and floating point number functions. While the library is excellent for simulation purposes, it requires certain VHDL variable types which causes it to be impractical for real-life implementation. This is due to the fact that many large companies such as Xilinx do not fully support VHDL-2008 in their software products yet. Due to this fact, it became necessary to switch to a block coding scheme. This entailed the use of the Xilinx IP cores or custom modules designed for sub-sections of the algorithms, and resulted in a procedure for developing algorithms on FPGA platforms. The procedure is as follows:

1. Replace high order functions within the algorithm with low-order versions. This brings the algorithm closer to a block diagram level.

2. Draw out a block diagram version of the algorithm as well as sub-functions of the algorithm to be used as sub-modules in the final design.

3. Calculate delay on each line as necessary for pipelining.

4. Test sub-modules with pre-calculated input and output sets.

5. Integrate sub-modules into final design and do system tests.

6. Debug as necessary.

The procedure described, while being very generic, is a useful process in the development of complex algorithms on FPGAs. Without having a pre-defined procedure, the development process for larger, more complex algorithms can become stagnant and lose momentum very quickly. The advantages of having a pre-defined procedure include improving communication and progress tracking between members of a team and the speeding up of development time.

4.3 FastICA

We begin implementing the FastICA algorithm by examining the procedures within the different steps of the algorithm. Four procedures are identified: estimation of the new weight vector, decorrelation, normalization, and check for convergence. All these procedures are repeated in each iteration so the hardware designed for one iteration can be re-used by adjusting the inputs to the processing chain. The re-use of hardware can be thought of similarly to a basic funnel. One can pour water through it and later re-use it for oil or any other sort of fluid. The structure of the processing chain, analogous to the funnel, remains the same while the data being applied at the inputs of the chain, analogous to the fluid flowing through the funnel, changes at each iteration.

4.3.1 Estimation

Estimation of a new weight vector is based off the update equation proposed by Hyvärninen [1]. This update equation uses the first and second derivatives of a non-quadratic, non-linear function denoted $g(u)$ and $g\prime(u)$ respectively. The update equation is defined as:

$$w^+ = E\{x \cdot g(w^T x)\} - E\{g\prime(w^T x)\} \cdot w \tag{3.6}$$

For our hardware implementation, we use the function $f(u) = -e^{(-u^2/2)}$. The first and second derivatives of this function are defined as:

$$g(u) = \frac{1}{a_2} e^{(-a_2 u^2/2)} \tag{3.4}$$

$$g\prime(u) = u \cdot e^{(-a_2 u^2/2)} \tag{3.5}$$

This function is selected because the exponential function recurs in the derivatives. Because of this, both $g(u)$ and $g\prime(u)$ employ the same term $e^{((-u^2)/2)}$. This allows us to implement only one function, the exponential function, and use that same function for both $g(u)$ and $g\prime(u)$.

Within the estimation procedure, we develop a basic sub-module designed to calculate both $g(u)$ and $g\prime(u)$. A block diagram of this module can be found in Appendix C. This module must iterate over the entire dataset of T samples during each iteration. Because of this, the FastICA algorithm tends to be slow in terms of data throughput especially if the number of samples is large. The slower speed of separation is of course in stark contrast with the low amount of resources required.

There are two different iterations occurring; one which is carried out over the samples during each of the other iteration which occurs for each weight vector. The first iteration processing chain is implemented using the specially designed function block and accumulators to hold the running calculation of the

expected values $[E\{xg(w^Tx)\}]$ and $[E\{g\prime(w^Tx)\}]$. After the first iteration has completed processing over the entire set of samples, the updated weight can be calculated and the higher level iteration can continue to the decorrelation, normalization, and convergence check procedures.

4.3.2 Decorrelation

Decorrelation of the estimated weight vector is necessary to ensure that the same component source is not extracted from the set of observations more than once. Each new estimate is decorrelated with respect to every other completed estimate using the following function.

$$w_p^+ = w_p^+ - \sum_{j=1}^{p-1} w_j (w_p^+)^T w_j \tag{4.1}$$

This is a Gram-Schmidt-like decorrelation scheme. We take the weight vector w_p^+ which corresponds to the current signal being estimated, and remove from it any information which is also contained within previously estimated weight vectors. To carry this operation out, we take each previous weight vector w_j, project our estimate of interest w_p^+ along that vector, and subtract the result from our estimate of interest. The decorrelation hardware is another short iteration pattern which accumulates the sum term and then subtracts it from the estimate using a hardware subtractor. This process is relatively simple and does not cost much time or many hardware resources.

4.3.3 Normalization

Normalization is also a relatively simple process. This is implemented using fixed size set of function blocks, although re-use of hardware can also be applied here if necessary. The normalization formula is written as follows:

$$w^* = \frac{w^+}{\|w^+\|} \tag{3.7}$$

The main issue with this stage is that it involves a reciprocal square root operation which is very expensive in terms of processing time and hardware resources. The sum of the squares of the elements of our estimated weight vector w^+ is calculated. That value is then processed through a module designed to calculate the reciprocal square root to give us the quantity $[\frac{1}{\|w^+\|}]$ which can then be multiplied with our weight vector w^+ to obtain a normalized vector estimate w^*. This new normalized estimate is compared with the last version of the weight vector from which the estimate was calculated to check for convergence.

4.3.4 Convergence and Control

Convergence of an iteration is achieved when the weight vector estimate has not changed since the last iteration. We apply a threshold limit on the sum of the absolute differences between corresponding elements of the two vectors being compared. If the sum is less than the threshold, then convergence has been reached, otherwise we must repeat the iteration to gain a better estimate. The hardware design of this system involves subtractors, adders, absolute value modules, and a comparison module.

After each weight vector has converged, the module-level controller switches the multiplexed inputs for the next vector. When all of the weight vectors have converged, the control system sends a signal to the system-level controller which uses the newly calculated weight vectors to separate out the source estimates from the whitened observations.

4.4 JADE

In the case of the JADE algorithm, processing can be decomposed into two basic stages. First is the calculation of the fourth-order cumulant matrix from the set of whitened observations. Second is the joint approximate diagonalization of the set of symmetric matrices of which the fourth-order cumulant matrix is composed. The result of the diagonalization process is a

matrix which is used as the separating matrix.

4.4.1 Cumulants Calculation

For calculation of the fourth-order cumulants, we use a simple iteration over each of the four dimensions of size n. For the two-observation case we hard-code the cumulant calculation for speed and to remove all redundant calculations. This is alright since the number of unique cumulants in this case is only 5.

It should be noted that the number of unique cumulants increases drastically with the number of observations n. One can calculate the number of unique cumulants in a given fourth-order cumulant matrix using the following formula:

$$\frac{(3+n)!}{24 \times (n-1)!} \tag{4.2}$$

Another more intuitive way to calculate the number of unique cumulants is the following summation:

$$\sum_{i=1}^{n} \sum_{j=1}^{n-i+1} i \times j \tag{4.3}$$

This method of calculating the number of unique cumulants is more intuitive because it calculates based on all the unique combinations of our n observations. If the number of unique cumulants is plotted over different values of n, the result is as in the graph below. Appendix A includes information about how the unique cumulants are distributed in the fourth-order $n \times n \times n \times n$ matrix.

It is clear that the number of unique cumulants rises too fast for us to naively hard-code the entire cumulant calculation block for any value of n, the number of input observations. For example, the 10-observation case results in 715 unique cumulants to be calculated. This will clearly require an entirely

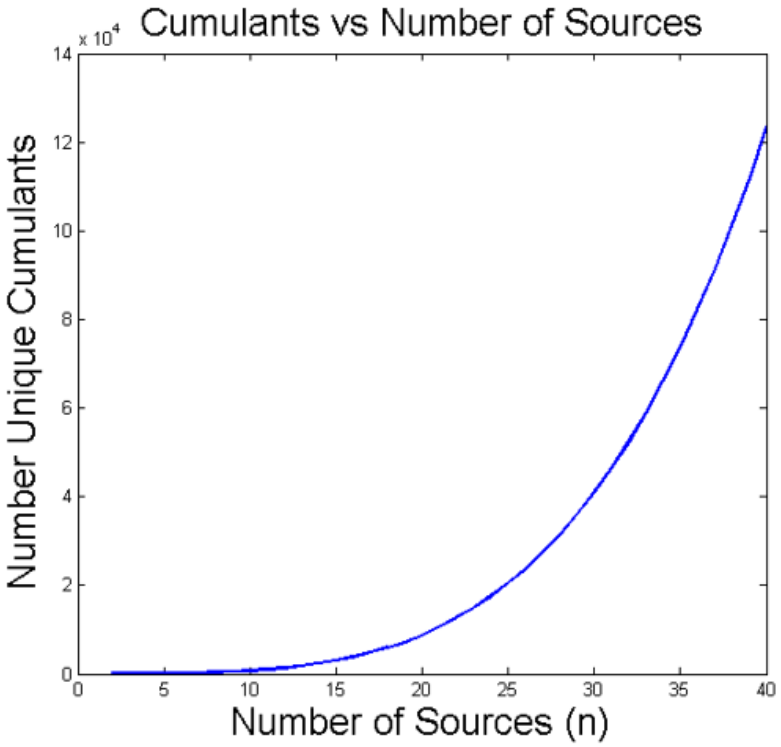

Figure 4.3: Unique Cumulants given Observations

inefficient amount of hardware resources. Because of this, it is necessary to alter the basic structure of the cumulants calculation code for larger and larger numbers of sources. It is possible to re-use the hardware used for calculating different cumulants in repeating passes. Although it will take much longer to calculate this, the effect will be to reduce hardware space requirements, making it possible to implement the larger algorithms on a hardware platform.

One compromise of hardware space to calculation speed requires $\frac{(n+1)n}{2}$ different cumulant calculators. These are iterated over $\frac{(n+1)n}{2}$ times. For the 10-observation case this means we need 55 different cumulant calculators going through 55 iterations instead of 710 calculators going through one iteration. This is a significant improvement, however it is still quite a large number of required resources. Another compromise of hardware space to calculation speed would use up to n cumulant calculators iterated over $\frac{(n+1)\,n^2}{2}$ times. For the 10-observation case this means we need 10 cumulant calculators going through 550 iterations.

These are both good compromises between hardware space and processing speed. Depending on the resources available and the requirements of the system, one can modify the cumulant calculation stage to adjust required resources and processing speed.

Future research may serve to make the cumulant calculation massively symmetrical as many of the calculations do share multiplications. Once the set of symmetric fourth-order cumulant matrices is calculated we move onto the actual joint approximate diagonalization stage.

4.4.2 Joint Approximate Diagonalization

The joint approximate diagonalization is done using a Frobenius norm formulation between the symmetric components of the fourth-order cumulant matrix. In the JADE algorithm proposed by Cardoso [2], a two-dimensional sub-component joint diagonalization process using a Jacobi iteration scheme is used. In terms of hardware resources required, this is highly efficient as only a two-dimensional processor needs to be implemented. This two-dimension processor is then fed different inputs to eventually complete the diagonalization process. At each stage, the set of cumulant matrices is updated by the calculated rotation matrices.

4.4.3 Angle Calculation

Each two-dimensional sub-matrix has a matrix angle associated with it. This is used in the calculation of rotation matrices for use in the diagonalization process. For example, if one were trying to rotate a two-dimensional matrix by an angle θ, they would transform it using the following rotation matrix:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{4.4}$$

This type of rotation is called a Givens rotation. The effective Givens

angle is calculated based off of a two-dimensional sub-matrix taken from the data set which is selected based on the Jacobi rotation pattern.

4.4.4 Data Update Stage

After each angle is calculated the entire set of cumulant matrices is updated. For each column, the data is rotated clockwise and each row is rotated counter-clockwise by the calculated angle. This is one of the most costly processes in the JADE algorithm as it requires multiplication across so much different data simultaneously.

4.4.5 Systolic Array Option

In the case of the hardware algorithm, a systolic array version of the joint approximate diagonalization algorithm can be used. This gives the hardware excellent performance with high data throughput by maximizing the amount of calculation carried out in parallel. A systolic array Singular Value Decomposition (SVD) algorithm is proposed by Brent, Luk and Van Horn in [18]. This is used as the basis of the systolic array version of the JADE algorithm.

A systolic array of processors consists of diagonal processors and off-diagonal processors. The diagonal processors carry out angle calculation along the diagonal elements of the matrix being processed. The angles are then transmitted to the off-diagonal processors which rotate the other elements of the data matrix by the calculated angles.

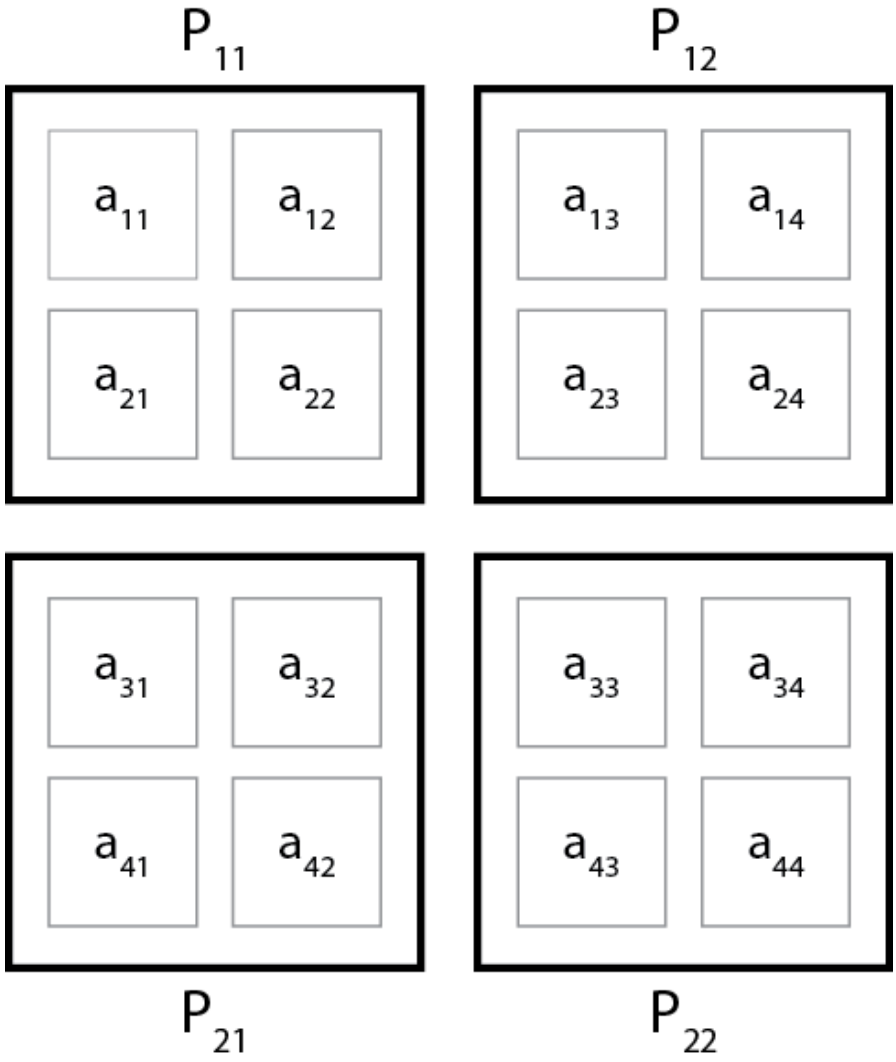

Figure 4.4: Systolic Array of Processors for $n = 4$

Figure 4.4 shows an example systolic array for a matrix of dimension $n = 4$. P_{11} and P_{22} are both diagonal processors which carry out two-dimensional local SVD while P_{12} and P_{21} are both off-diagonal processors. The data within each processor is rotated by the rotation angles calculated by the corresponding row and column pair. For example, the data in P_{11} would be rotated by the angle calculated by P_{11} while the data in P_{12} would be rotated by both the angle calculated by P_{11} and the angle calculated by P_{22}. The transformation is mathematically represented as follows.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \tag{4.5}$$

$$A_{new} = \begin{bmatrix} cos(\theta_h) & -sin(\theta_h) \\ sin(\theta_h) & cos(\theta_h) \end{bmatrix} \times A \times \begin{bmatrix} cos(\theta_v) & sin(\theta_v) \\ -sin(\theta_v) & cos(\theta_v) \end{bmatrix} \tag{4.6}$$

Where θ_h is the angle calculated by the diagonal processor on the same row as the processor carrying out the rotation and θ_v is the angle calculated by the diagonal processor on the same column as that processor. In the case of the diagonal processors, both angles are the same ($\theta_h == \theta_v$). Systolic data rotations are carried out after this transformation.

Typical convergence of this algorithm depends on a selected threshold value. When the calculated angle is less than the threshold value, the diagonal processors automatically assign the rotation angle as zero and automatically zero the off-diagonal elements local to the processor. In this way, it is ensured that eventually the off-diagonal elements will reach zero, and that the diagonal elements converge within tolerance to the singular values.

After each iteration, which includes angle calculation and the rotational transformations, the data in each processor is passed in a systolic rotation

pattern based on the Jacobi rotation pattern. Each processor has data flowing in and out. By connecting the inputs and outputs of each processor correctly, it is easy to setup a systolic array for the purpose of joint approximate diagonalization. One can also expand the matrix dimension being processed easily by adding new processors along both dimensions and connecting them appropriately.

The most intuitive way to understand this rotation pattern is by looking at row and column rotations separately. The following diagram illustrates the rotation pattern along a row or column for a matrix of dimension $n = 8$.

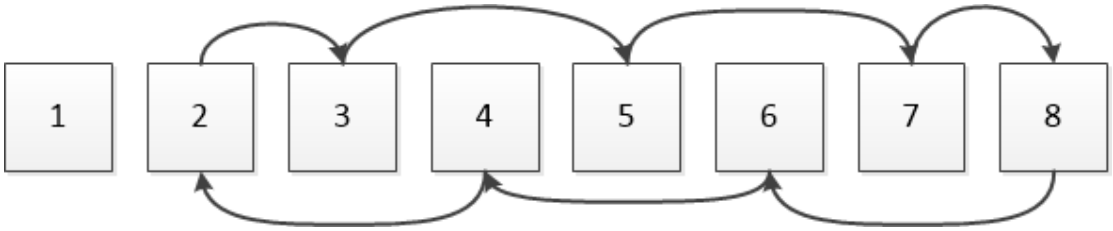

Figure 4.5: Jacobi rotation scheme for both columns and rows.

The matrix elements are separated in pairs as they would be in the systolic array for processing. After each rotation stage, the elements are moved between the systolic processors. An intuitive way to look at this process might be to carry out this rotation on each row first, then on each column. In this way the entire rotation process is broken down into many simpler movements. Optimizations can be and have been made by the original authors to where the data is instead directly transmitted in a diagonal pattern. The effect is the same, however, the hardware setup using diagonal data flow may potentially be more efficient. Ultimately however, when assigning inputs and outputs to the systolic processors, the data will incidentally move diagonally. More detail about the systolic array data algorithms can be found in Appendix A, and in [19] and [18]. Also included in Appendix A are example figures illustrating the rotation process for matrices of dimension $n = 4$ and $n = 8$.

4.5 Pre-Processing

Both the JADE and FastICA algorithms share a pre-processing, whitening and centering stage. Centering was implemented through the external hardware by removing all DC components and then adding back an exact DC component. Whitening was carried out using EVD on the covariance matrix calculated from the set of sampled observations.

4.5.1 Centering

Centering can be done multiple ways. The first step is to remove any DC components currently within the incoming analog signals. This is easily done by running our inputs through a $1\mu F$ capacitor. This capacitor is selected and used with the assumption that our test signals will be of a frequency composition that will not be lost due to capacitor attenuation. With a purely AC signal, we add 1.65 volts back into the observations to center our test signals on the range of 0 to 3.3 volts. This was done two different ways in different revisions of the circuit: using simple circuit components, and an operational amplifier based circuit. A simple example circuit is depicted in Figure 4.6. For the operational amplifier circuit a simple op-amp adder circuit is implemented with one of the inputs fixed to 1.65 volts.

After sampling via the ADC, the numerical value 2048 was subtracted from the now-digital signals. This is because our signals are digitalized to 12 bits, or on a range of 0 to 4095. The subtraction results in a digital sample in the range of -2048 to 2047, which is converted to a floating point representation of the sample, and scaled back to voltage by multiplying our set of signals by $\frac{3.3}{4095}$ for processing within our ICA algorithm.

4.5.2 Whitening

Given our centered, floating point, and voltage-scaled values, we can begin our whitening stage. First, the incoming samples are stored into internal

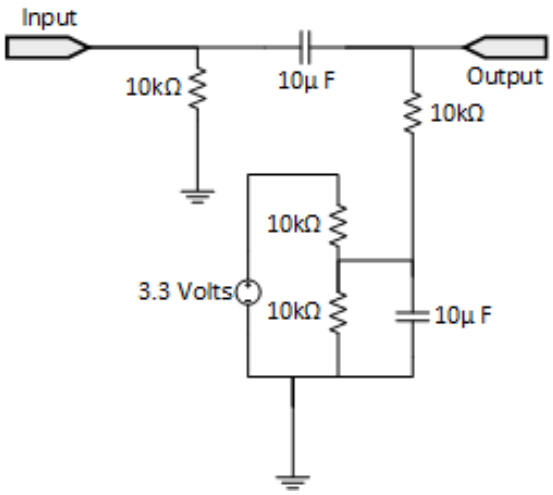

Figure 4.6: Example low-noise DC offset circuit

FPGA block RAM. Concurrently to the storage process, the covariance matrix is calculated. The covariance matrix is then sent into a systolic array version of an eigenvalue decomposition (EVD) processor. This processor works virtually the same way as the previously described systolic option for the JADE algorithm. The only difference between the two algorithms is in the diagonal processors. The angle calculation formulae are slightly different because the end goal of the whitening process is an EVD while the end goal of the JADE algorithm is actually a SVD.

After processing, the result is a set of eigenvalues and the corresponding eigenvectors. Each eigenvalue corresponds to a principal component and each eigenvector acts as a linear transformation to extract that principle component from the set of observations. Put together in matrix form, the eigenvectors are the whitening matrix. The set of observations, which are currently stored in block RAM, are transformed with this whitening matrix and then stored in a second block of RAM. The reason for using multiple sets of block RAM is that RAM is abundant within the FPGA and using block RAM at strategic points allows for a large-scale pipelining scheme which greatly increases the amount of processing carried out concurrently.

4.6 Number Representation

When implementing any algorithm on an FPGA one must consider how the numbers will be represented. In this respect there are two possible choices; fixed or floating point. The advantage of fixed point is low hardware resource usage, easy programming, and fast operation. The advantage of floating point is fixed hardware usage per data and large dynamic range. In VHDL one can implement these number systems using std_logic_vectors, bit_vectors, signed, or unsigned types. For our implementation we generally use std_logic_vectors due to their generic and versatile nature.

4.6.1 Fixed Point

Implementing fixed point hardware is relatively straightforward and simple. The addition, subtraction, and multiplication circuits can be implemented using direct hardware algorithms. Some slightly more advanced functions must be used for division and an approximation method is used to calculate the square root. Some cases require scaling of values which is implemented simply by shifting our std_logic_vectors to the left or right.

4.6.2 Floating Point

Implementing hardware for floating point math involves much more overhead processing and logic than fixed point. In the naïve approach the numbers on which the operation is being carried out are converted first to fixed point before carrying out mathematical operations. However, in many cases this is not necessary. For example, if a very large number were added to significantly small number, the result would be virtually equivalent to the larger number because the small number is lost in precision error.

4.7 Hardware Math

4.7.1 Fixed Point Math

One important consideration with the different mathematical functions is that the size of the result of one of these operations can be different from the size of the input vectors. For example, if we were adding two three-bit binary numbers together, the result could potentially be a four-bit vector because of the carry bit. Another example is the multiplication case. If we were to multiply two three-bit binary numbers together, the result could be up to six bits wide.

Figure 4.7 is an example adder/subtractor circuit; notice that the result is one bit larger than the input vectors given the reasonable assumption that the input carry bit is assumed to be a logical zero. This hardware adder is used regardless of where the binary point is located along the binary vector. As long as the two input vectors are aligned in terms of their respective binary points, the result will be correct.

Figure 4.8 is an example hardware multiplier circuit; notice that the result is the same width as the sum of the widths of the input vectors. In this hardware algorithm as well, the inputs need not be aligned, but the binary point on the result vector should be calculated based on the binary points of the input vectors. However, in many cases it is convenient to align the input vectors to remove the overhead of having to calculate a new binary point.

The other basic math functions such as division and square root are created using sub-modules. Each of these modules accepts the appropriate inputs and outputs the calculated output. Putting these into sub-modules is necessary because the division and square root operators do not have direct hardware circuit implementations. The calculations for these functions must be carried out over several clock cycles.

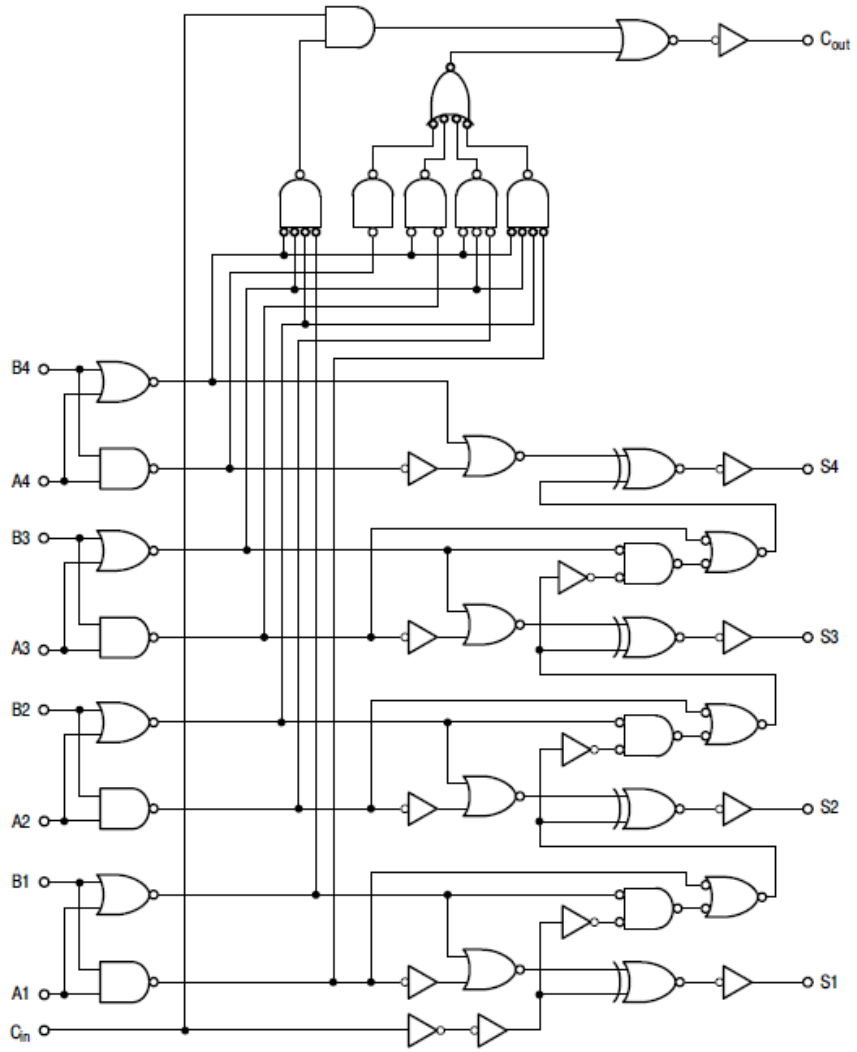

Figure 4.7: 4-bit Hardware Adder Subsystem Diagram [20]

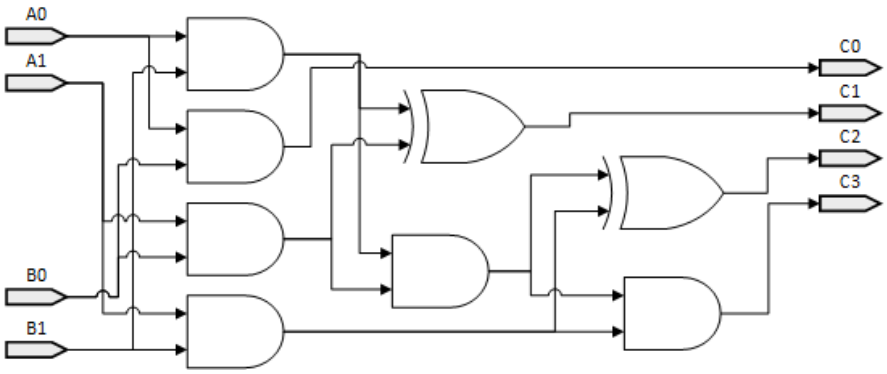

Figure 4.8: 2-bit Hardware Multiplier Diagram

4.7.2 Floating Point Math

In contrast to implementation of fixed point math, floating point math requires a fixed number of bits to represent each number. For example, in the case of 32-bit floating point format, any two-number mathematical function would accept two 32-bit numbers and return a 32-bit number. Such is also the case if a different floating point format were used as well.

Another major difference from implementing fixed point math functions is the relative complexity of the operations being carried out. In a floating point adder, the numbers must first be converted to fixed point format and then added using a large width adder. This is one major point at which an optimization can be made. As far as the implementation of the multiplier, both significands are multiplied, including the hidden most significant bit, and the exponents are added together after accounting for the excess-binary coding scheme. The sign bits can be run through an xor logic gate to generate the resulting sign.

While the multiplier is evidently relatively simple to implement in floating point, the adder/subtractor is slightly more complicated due to the variable width nature of operations. This can be accounted for by comparing the exponents before-hand. For example, if one were subtracting a very small number from a very large number, the small number may only affect the lower bits of the large number. This can be implemented using a bit inversion scheme. Considering a similar example using IEEE 754 single precision format, if the difference between the exponents is larger than 24 bits, then we know that the smaller number is located far to the right of our 24 most significant bits. If we were to convert both numbers to a fixed point format first, only the most significant bit of the smaller number's significand would affect our result. We take the significand of the larger number and look for the least significant non-zero bit. This bit will become zero and any less significant bits will become ones to account for the cascaded borrowed bit.

To further illustrate this example, let us subtract

$6.067912750040705 \times 10^{-8}$ from 2.0178651809692383. These numbers in floating point format are 00110011100000100100111010110100 and 01000000000000010010010010110100 respectively. First we consider the signs; since both are positive we know which order of operations we are considering. Next we take the difference of the exponents. The exponent for the larger number is 10000000_b and the smaller exponent is 01100111_b. In terms of decimal notation these are 128 and 103 respectively. Taking the difference $128 - 103$ results in a difference of 25, which is larger than the width of the significands. Therefore we can apply the bit-shift operations instead of wasting hardware space to hold two 49-bit numbers. In applying the bit-shift operations on the significands, we only need look at the significand for the larger number which is equivalent to 10000001001001001010110100 including the hidden bit. The least significant '1'-bit is 3 bits from the right end. After applying the bit-shift scheme the resulting significand should be 10000001001001001010110011. We return this significand along with the exponent of the larger number as the answer. To verify the result, we convert our number back into decimal notation which is equivalent to 2.017864942550659. This is correct if compared with the result of carrying out the subtraction in decimal notation. One can verify this using many online floating point adder/subtractor resources. Alternatively one can take the numbers and perform a typecasting along with the operations to see the results of the subtraction.

The addition and subtraction cases with both positive and negative numbers can be summarized by Table 4.1. The first two colums are the respective signs of the two input operands, $e_3 = e_1 - e_2$, and m_1 and m_2 are the respective significands or mantissas. In $e_3 = e_1 - e_2$, e_1 and e_2 represent the respective exponents of the operands. By taking the difference between them, we can decide which operation should be carried out on the significands to produce our result significand m_3.

The most amount of computational savings are found in the $op(m_x)$

Sign #1	Sign #2	$e_3 \geq 24$	$e_3 \leq 24$	$0 \leq e_3 < 24$	$-24 < e_3 < 0$
$+$	$+$	$+\,(m_1)$	$+\,(m_2)$	$+\,(m_1 + m_2)$	$+\,(m_1 + m_2)$
$-$	$+$	$-\,op(m_1)$	$+\,op(m_2)$	$-\,(m_1 - m_2)$	$+\,(m_2 - m_1)$
$+$	$-$	$+\,op(m_1)$	$-\,op(m_2)$	$+\,(m_1 - m_2)$	$-\,(m_2 - m_1)$
$-$	$-$	$-\,(m_1)$	$-\,(m_2)$	$-\,(m_1 + m_2)$	$-\,(m_1 + m_2)$

Table 4.1: Floating Point Adder Operations

operators. This is a simple bit shift operation whereby we shift the bits of the input operand beginning from the least significant bit until we reach the least significant '1'-bit. By doing this we effectively limit the total width fixed-point of adder/subtractor required to calculate the result significand.

For more complex functions such as division and square root calculations we use compound hardware. For division, a successive subtraction can be implemented in a partially cascading format to obtain the results of division in a timely manner. This is equivalent to long division.

For the square root function we use a simple approximation scheme. A popular algorithm for this is the Babylonian method which is a reduced case of Newton's method. The hardware aims to solve the problem $f(x) = x^2 - S = 0$ where x is our solution and S is the number of which we are finding a square root. Ultimately this reduces to numerical form as $x_{n+1} = \frac{1}{2}(x_n + \frac{S}{x_n})$. Each update of x_n by the formula results in a more accurate approximation for $\sqrt{S}$. We can also take series approximation methods centered at different points to improve convergence and accuracy characteristics of our algorithm.

For our hardware implementation, while these would be ways to optimize our code, we opted to use Xilinx IP cores for the mathematical operations. While it is important to understand the optimizations that go into hardware implementation of floating point mathematical operators, it is not absolutely necessary for the research detailed in this thesis, because a working

implementation is more important than an optimal one for proving feasibility of implementation.

4.8 Matrix Math

The basis of much of the math which is used to describe the FastICA and JADE algorithms is matrix math. We use matrix-based mixing and separating models as well as matrix-based models for the processing carried out in each algorithm. To implement matrix multiplication we have two basic structure options: systolic and direct.

4.8.1 Systolic Architecture

In the basic systolic architecture a set of n^2 systolic processors is required, each consisting of a multiplier and an accumulator. In the $n = 4$ case, the processors are arranged as depicted in Figure 4.9. The inputs to the columns and rows of this systolic array are delayed by the column and row number respectively.

At each calculation stage, each processor accumulates another term of the total sum of a given element of the result matrix. The values being used for calculation in each processor are passed either horizontally or vertically to adjacent processors after each calculation stage following the arrows shown in Figure 4.9. A detailed stage-by-stage example of a systolic multiplication can be found in Appendix C Section 2.

For a matrix transformation case it is necessary to read the results at each processor in real time and store them as they become available so that the next element, rotationally at that position, can begin processing. The rotation scheme repeats over n columns. For each element in the data matrix, the matrix which is to be transformed by our transformation matrix, the element is delayed by $delay = floor((c-1)/n) \times n + r + ((c-1) \mod n)$ where c is the column

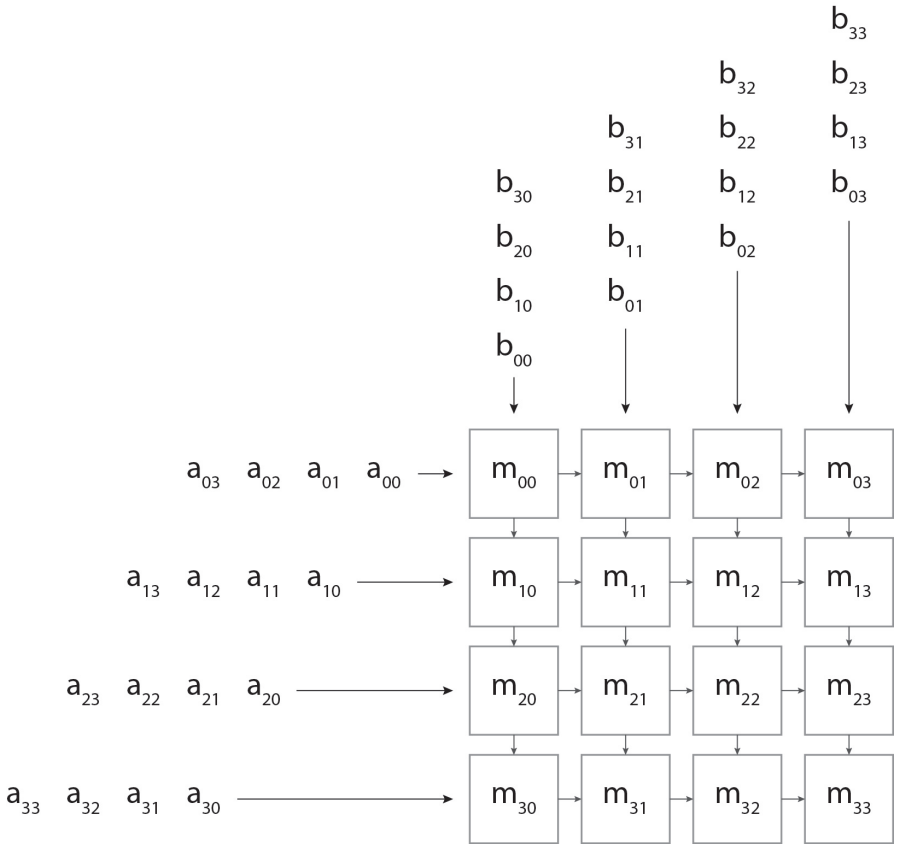

Figure 4.9: Systolic Array Multiplier

number starting at n and r is the row number counting from the bottom starting at 1, given that the data is of dimension $n \times T$ where n is the number of observations and T is the number of samples in the set. The elements of the transformation matrix are rotated counter-clockwise horizontally while the elements of the data matrix are passed down vertically with the delay given by the equation above. It may be easier to conceptualize this in terms of a graphical representation. Figure 4.10 depicts this rotation pattern.

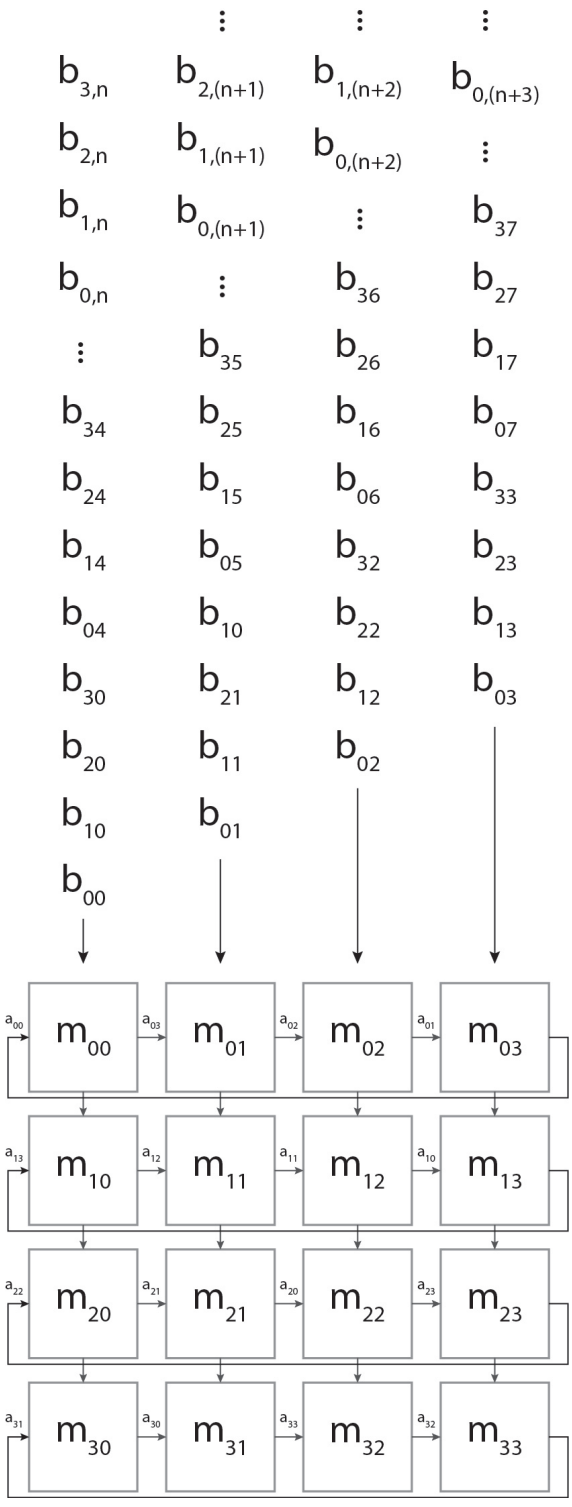

Figure 4.10: Systolic Array Transformation

4.8.2 Direct Calculation Architecture

To implement direct calculation architecture a set of n^2 multipliers and either n^2 or $n \times (n-1)$ adders is required. Figures 4.11 and 4.12 show hardware diagrams of row calculators for processors of dimension $n = 3$ and $n = 6$ respectively. There are n number of row calculators in any given set. Each row calculator receives a different row of the transformation matrix A and each subsequent column of the data matrix X. The output from a given row calculator corresponds to the value Y at row j, column i in the output matrix.

To create a matrix multiplier from the row calculators, the values in a row of the mixing matrix is assigned to a row calculator. The columns of the source matrix X are then fed into the set of row calculators to generate the columns of the output matrix Y.

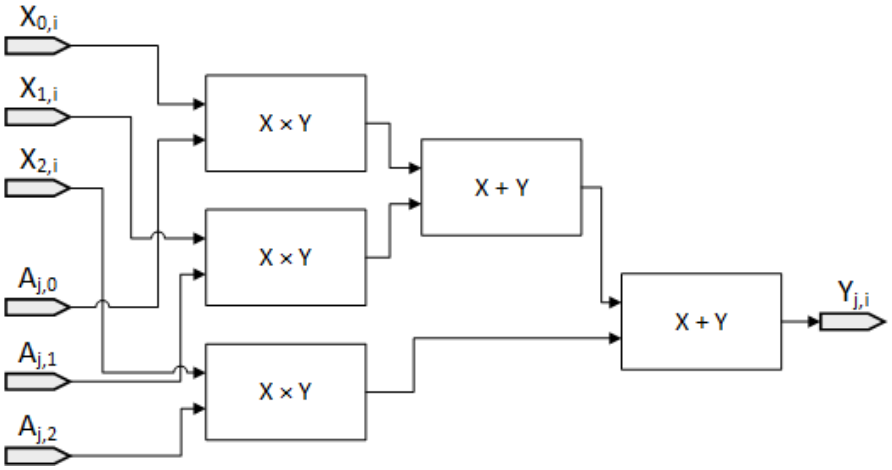

Figure 4.11: Direct Row Calculator $n = 3$

4.8.3 Comparison

To compare different hardware architectures we consider speed and latency, both of which are given in terms of the number of clock cycles required to achieve solution and output. In comparison with the systolic architecture the direct calculation architecture actually shows better performance overall. A

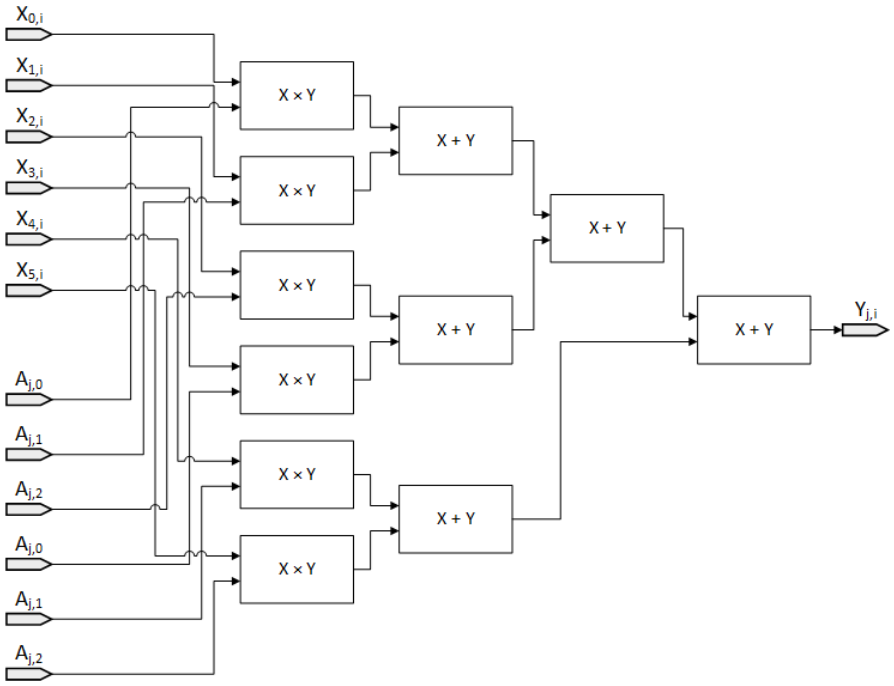

Figure 4.12: Direct Row Calculator $n = 6$

similar amount of hardware resources are required, and equivalently fast calculations are achieved at the cost of extra latency. This conclusion is drawn from a detailed analysis of the hardware diagrams based on the reported latency and speed reported after implementation of the two architectures in Xilinx ISE.

However, the latency increases at a log_2 rate so is therefore not a great factor. Another point of note about latency is that the length of samples T is usually very large so the latency is very small compared to overall calculation time.

As an example we consider the extreme case where $n = 40$. The latency factor for the systolic architecture is calculated in the worst case scenario as follows.

$$mult + add \times ceil(log_2(n)) = 8 + 11 \times ceil(log_2(40)) = 8 + 11 \times 6 = 74$$

In this equation, *mult* is the latency of a multiplier block (2 to 8 clock cycles), *add* is the latency of an adder block (2 to 11 clock cycles), and latency values are in terms of number of clock cycles. When referring to block speeds, we

will refer to the high latency versions of blocks as slow-block and the low latency versions of blocks as fast-block. Compare the direct architecture latency with the latency of the systolic architecture processors which is $mult + accum$ where $accum$ is the latency of the accumulator. The accumulator core has a maximum latency of 22, leading to a total worst case latency of $8 + 22 = 30$ for the systolic architecture. As far as overall calculation speed goes we use the case of $T = 500$. For the direct calculation architecture overall system speed is $speed_{direct} = T + mult + add \times ceil(log_2(n))$, and for the systolic architecture the overall system speed is $speed_{systo} = T + 3 \times n - 2 + mult + accum$. For the case of $n = 40$ and $T = 500$, the worst case, or slow-block speeds $speed_{direct} = 574$ and $speed_{systo} = 640$. The direct calculation architecture is faster even in the slow-block case. In the fastest-case speed $speed_{direct} = 514$ and $speed_{systo} = 622$. In either case the direct calculation architecture is faster.

Let us consider another case for comparison where $n = 3$ instead of 40. Slow-block speeds are $speed_{direct} = 530$ and $speed_{systo} = 537$, and fast-block speeds are $speed_{direct} = 506$ and $speed_{systo} = 511$. Even at a low number of observations, the direct calculation architecture shows faster calculation speeds than the systolic architecture. Therefore, for our implementations, any generic matrix multiplications and transformations are implemented using the direct calculation architecture.

It should be noted that it is possible to use the systolic array in a manner which will decrease required hardware space by stacking columns. For instance the width of a $n = 4$ array might be reduced to 2 and columns 2 and 3 would then be stacked on top of 0 and 1 in a manner similar to the matrix transformer in Figure 4.10. Further detail and optimization can be found in US patent 8,924,455 by Xilinx [21] which presents different configurations of systolic array arithmetic operators. In particular, optimum versions of systolic array multipliers are discussed. In some configurations, buffers are used to optimize the hardware space used, while other configurations are optimized for speed of calculation.

4.9 Pipelining

Pipelining in hardware is a design technique in which the hardware is arranged and connected in such a way that computations are carried out in successive steps without the need for an intermediate controller system. A pipelined system on an FPGA is extremely fast as the latency is only a factor at the beginning and end of the processor. Compare the data flowing through the hardware to water flowing through a system of pipes. When you first turn on the water, it takes some time for the water to appear at the pipe system outputs, but from that point on the water continues to flow correctly out of the outputs as long as the input water continues. Data flowing through a pipelined hardware system behaves in much the same way.

For the FastICA and JADE algorithms, it is impossible to properly pipeline the entire system without the use of some sort of concurrent processing scheme, modifying the algorithms, or massive use of hardware space. Consider the whitener which is common to both ICA algorithms. The whitening stage carries out EVD on the covariance matrix to calculate the whitening matrix. In this case it is necessary to calculate the covariance matrix before performing the EVD, and then it is necessary to transform the set of observations by the whitening matrix. Calculating the covariance matrix and transforming the set of observations both require iterating over the entire set of samples, but the covariance matrix iteration must occur before the transformation iteration.

A solution to this issue is to add another dimension to the data movement. In the basic pipelining case, the data is moving in one dimension, which is normally time. For example, for each sample, some sort of operation will be carried out to produce an output at the same rate as samples are captured and input to the system. In essence, each sample is moving along the pipeline over time. To add another dimension of movement, each set of samples is moved in time across different stages of the system. At each stage which requires iteration over the entire set of samples, we use internal FPGA block

RAM to store a version of the sample set. Each processing chain will iterate over this different version of the data concurrently with other processing chains. While this method costs more in terms of RAM, it is not a concern in the general case as there is an ample amount of on-chip RAM as well as extra on-board RAM in the unlikely event that the on-chip RAM is utilized.

4.9.1 RAM Swap Scheme

The RAM swap scheme is used when implementing a concurrent processing system. First we analyze the entire system to identify points at which processing chains can be established. Figure 4.13 shows a simplified block diagram which describes our system. From this diagram we can distinguish four core processing chains: input, whitening, ICA, and output. It is possible to further subdivide the ICA processing chain into several more depending on the algorithm being implemented, but at this point we will consider that as one chain with a variable latency factor to account for differences in processing time. For the input and output processing chains there is virtually no latency, factor and the speed is limited only by the maximum operating speed of the ADC and DAC respectively. For the whitening and ICA chains there are two basic latency factors. First is the amount of time required to calculate the transformation matrix, either the whitening or separating matrix. Second is the latency of the transformation operation which is insignificant since this involves only one multiplier stage and $log_2(n)$ adder stages.

Between each processing chain is a requirement of some form of intermediate data storage buffer. By using a dual-buffered system, we can maximize the concurrency of high latency processing, especially in the case of the whitening and ICA chains. Another positive benefit of a dual-buffered system is that the input and output chains can run continuously, since they are usually limited significantly in terms of speed relative to the other processing chains. The buffers are implemented using internal FPGA block RAM and are

located in the system at points marked RAM in Figure 4.13.

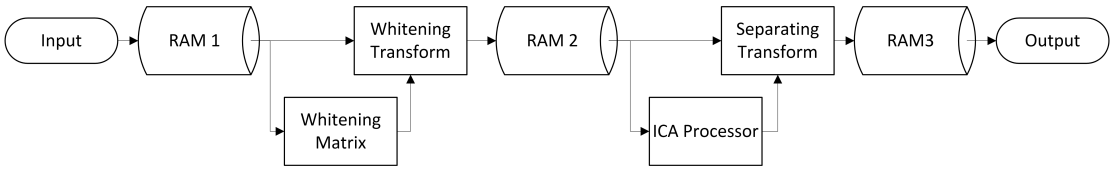

Figure 4.13: BSS System Block Diagram

To analyze the timing of different processing chains relative to each other and the availability of data, we consider their relative expected processing speeds and apply certain design constraints. One such constraint to limit the amount of analysis required is to assert that the input and output processing chains must run continuously. In other words, the input chain must always have RAM available to store incoming data and the output chain must always have data available for outputting. Using a timing diagram with dependencies marked, we can check the feasibility and expected behavior of our system with regard to times at which different processing chains will run relative to each other.

Figures 4.14 and 4.15 show two different versions of the timing diagram for our system. In terms of our processing chains, input and output have their own timing columns, whitening consists of the whitener and corresponding transformation, and ICA consists of the ICA and corresponding transformation. The black arrows represent process or sub-module run-times while the gray arrows represent dependencies. The process pointed to by a gray arrow cannot begin operation until the process from which the arrow originates has completed its calculations. Figure 4.14 is the synchronous diagram in which each chain is synchronized relative to a fixed-time input chain. On the right in figure 4.15 is an asynchronous run-when-ready diagram which we call the greedy diagram because it runs under the premise of running as soon as possible, being greedy in terms of time. Both diagrams begin with the assumption that the system has just been started. At first, only the input chain runs, then when that initial data

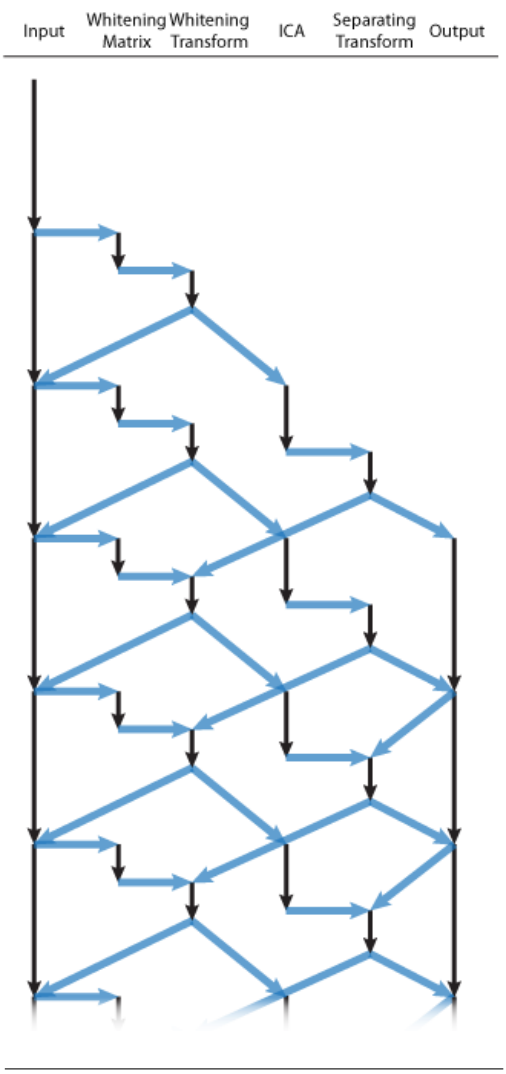

Figure 4.14: Synchronous Timing with Dependencies

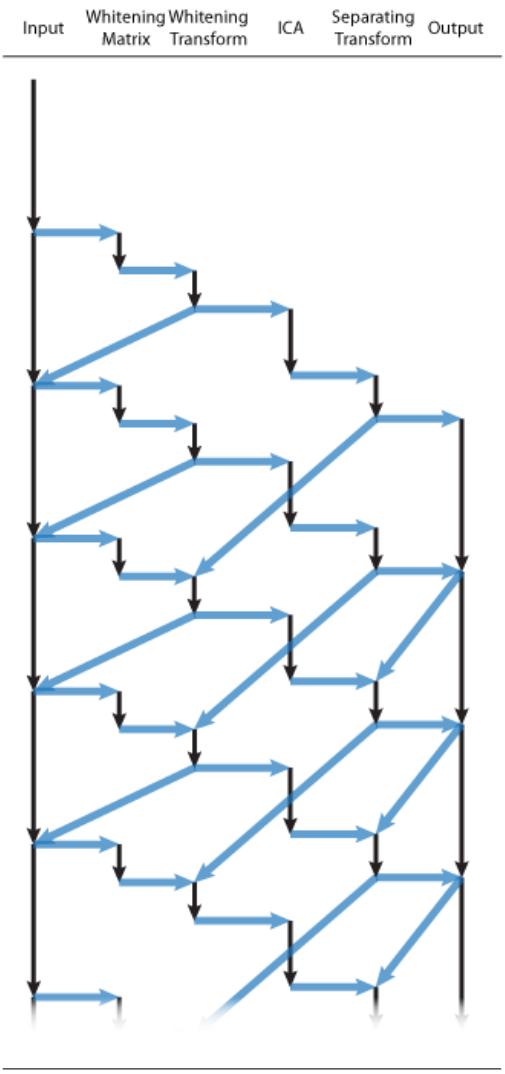

Figure 4.15: Greedy Timing with Dependencies

is captured, it begins to cascade through the system along each subsequent chain. At the same time, the RAM banks are swapped at each data junction so that the input chain begins loading new data into the RAM which is not being used by the subsequent chain at the moment. Doing so ensures continuous data flow (pipelined) while lowering overall processing time. In the synchronous diagram, the chains are set to begin their initial operation when data is available and synchronous to the input chain. This type of diagram is useful to understand all of the dependencies within the entire system, and to compare the relative running times of the different chains. In the asynchronous diagram each chain begins operation as soon as data is available and dependency conditions are met. This type of diagram is useful to understand the most optimal system operation, and the operation that is likely when the system is actually implemented. In both diagrams, the timing is possible as long as the dependency arrows are directed either horizontally or downwards.

What we see from the synchronous diagram is that the whitening and ICA chains each must have a run-time which is less than or equal to the input chain. This will ensure that all chains can run concurrently to each other, and that the data flow is continuous. For our specific system, the on-board clock runs at $100Mhz$ while the ADC and DAC are limited to run at $10Mhz$. The ADC and DAC are also 12-bit based on a 16-count cycle with plus one count for resetting operation between ADC captures. This means that our whitening and ICA processing chains are effectively running at 170 times the speed of the input and output chains which in turn means that, except in extreme cases, we will meet the condition that the data flow will be continuous. This effectively makes the overall system a pipelined one.

The asynchronous diagram is the same diagram as the synchronous with later chains shifted earlier along the timeline as far as possible while still ensuring proper operation. The exact same dependencies are added to this diagram to check feasibility and see how the overall system will operate. This

diagram tells us the expected hardware system latency.

While it is possible to run certain chains across the same RAM blocks concurrently by phasing the timing, doing so saves minimal latency and increases system complexity significantly. This may be an area of future research, especially if RAM becomes scarce, because a phased operation system of this manner would use a little over half the amount of RAM used in the dual-buffered system.

Another benefit of the concurrently buffered system is that each chain can be tested as another sub-module. Because of this, the parent system is less complex than a system in which chain sub-components and sub-chain control logic are at a higher position in the module hierarchy. Using a RAM swap scheme, we ensure overall pipelined behavior of the BSS system and reduce development complexity.

4.10 IP Cores

Previously we discussed hardware implementations of different mathematical operators. For our implementation we opt to use Xilinx IP cores. This gives the advantages of versatility and access to more complex functions without the need for writing them ourselves. Using the provided IP cores also allows us to adjust system properties such as latency on the fly. IP cores can be pre-compiled to speed up synthesis of the entire hardware design. We use the floating-point IP core which has many different selectable functions built into it. These are selectable when configuring the IP cores.

Math operations in our design are carried out using IEEE 754 single precision 32-bit floating point number format. The floating point IP core provides us with all of the necessary basis math functions including absolute value, add/subtract, compare, divide, exponential, logarithm, multiply, reciprocal, reciprocal square root, and square root. Also included is an accumulator, fused multiply-adders, and fixed/floating point converters. These

allow conversion between fixed and floating point as well as conversion to different floating point formats.

For our hardware modules, we use the adder, subtractor, comparison, divider, multiplier, square root, reciprocal square root, number format converter, and accumulator. Each module can be adjusted in terms of latency to adjust maximum operating speed and hardware resource usage. In many modules a table is provided showing the expected resource usage at different latency values.

4.10.1 Data Flow

We can also adjust the control signals going into and out of our different IP core blocks. This allows us to control how data flows. The specific floating-point IP core block uses the AXI4-Stream control signal format which is a commonly used, Xilinx-defined interconnect interface description. At the most basic level, this gives us a data array output under a signal labeled 't_data' and a control signal labelled 't_valid' which indicates when the signal currently being asserted on 't_data' is technically valid data. Connecting the output 't_valid' signal of one block to the corresponding input on another block will cause the child block, the block to which data is flowing in terms of pipeline hierarchy, to process the incoming data as soon as it is available.

4.10.2 Accumulator

The accumulator is one of the most important pieces of the hardware processor blocks. The accumulator provided in the floating-point IP core accumulates numbers in fixed point format, but outputs in terms of floating point. In the customization wizard we can adjust the internal precision of the fixed point part to change resource usage and accuracy. For our hardware design, ranges are selected to ensure that precision error due to the format conversion is an insignificant error factor.

4.10.3 Fixed/Float Converter

Another key function that we use is the fixed/floating point converter. In the customization wizard we define what the input and output formats should. For fixed point we define the most significant bit (MSB) and least significant bit (LSB) in terms of the std_logic_vector indices. For floating point we define the widths of the exponent and significand to adjust dynamic range and precision respectively.

4.11 Hardware Resources

When it comes to FPGA hardware implementations there are several different ways to compare different implementations of different algorithms. The two most important are speed and amount of resources used. Another important factor is latency, but this is less of a concern in applications which do not require a high rate of data through-put.

4.11.1 Speed and Latency

Speed and latency in FPGA implementations are measured in terms of clock cycles. For instance, a floating point adder synthesized using a Xilinx IP core block will result in different speed and latency values. Increasing the latency allows the system to run with a higher internal clock speed, but results in a longer delay between data entering the inputs to a correct result being output. For the adder block with non-blocking control configuration, latency values can be anywhere between 2 and 11 clock cycles. Any value selected will still allow for a throughput of one data point per clock cycle, but will vary the maximum clock speed and the amount of hardware resources the core uses.

If two adders were stacked together to add three numbers in total, the latency would have to be the sum of the latency of the two adders. The data throughput rate is still one data point per clock cycle, but the latency will now

be anywhere between 4 and 22 clock cycles. The same principle applies when connected any other function blocks together. If they are arranged in parallel, the highest latency is the system latency, and if they are arranged in series, the system latency is the sum of the latencies of all the components. This is important to understand for pipelining.

4.11.2 Resources

Hardware resources are measured in either Flip-Flops (FF) and Look-up Tables (LUT), logic slices, or logic cells. Flip-Flops in the case of the 7-series Xilinx FPGAs are all D-type Flip-Flops. When implementing using Xilinx Vivado, part of the synthesis and implementation reports is the number of FFs or LUTs used by each distinct type of block.

Other more specialized hardware resources are also used as comparison measures. One such resource is a dedicated, high speed hardware multiplier called a digital signal processing slice (DSP slice). These can be a measure of speed in implementations of systems which are very similar to each other. Internal FPGA RAM is also another resource that can be a comparison measure. There are two types of FPGA RAM; block and distributed. For this project, block RAM is used as the main data storage structure, but distributed RAM can be used as well in cases where speed is not as important. With the Nexsys 4 board we also have on-board RAM available in case internal FPGA RAM is all used up, but this changes the control structure and potentially results in slower operation depending on the maximum operating speed of the RAM.

CHAPTER 5

SOFTWARE SIMULATIONS

5.1 System Overview

The software algorithm system can be split up in a similar manner as the overall BSS system overview in Figure 4.13. The main difference is in the RAM blocks and capture system; Instead of sampling using a Data Acquisition (DAQ) system, the observed signals are generated using MATLAB code. A random mixing matrix of dimension $n \times n$ is generated to model the mixing system; this is done randomly because we are under a blind assumption. Using the generated mixing matrix and our generated set of source signals, we simply multiply them using the inherent matrix multiplication ability of MATLAB to generate our set of observations. The generated observations are then used by the different BSS algorithms to estimate the original sources. The advantage of doing this in software is that one can easily compare the estimates to the actual originals.

5.2 Simulations

Simulations are carried out in MATLAB. The results of the algorithms can be compared with the generated inputs to calculate separation accuracy. This includes mean square error (MSE) between the original sources and their corresponding outputs and comparison of the randomly generated mixing matrix to the estimated separating matrix, and/or the signal-to-noise ratio (SNR) of separated signals in the case where randomly generated noise is added to the

inputs.

Both software algorithms will be compared with each other, and ultimately will be used in verification of the hardware algorithms. If the hardware algorithms have the same output as the software algorithms, then we can assume that they are correctly coded and accurate to a degree. A numerical comparison can also be carried out between hardware and software algorithms by extracting sample data from the hardware system directly.

All four algorithms, two hardware and two software, will also be compared for computational speed. While it may be difficult to compare software algorithm performance with hardware performance, the comparison will still generate useful information; we may be able to see that one is better than the other, but not necessarily be able to see by how much an algorithm surpasses another.

5.3 Algorithm Comparison

5.3.1 Computational Speed

When comparing computational speed of both software and hardware algorithms, it is necessary to note the specifications of both systems. Computational speed of software algorithms can be measured using timing functions. For example one would measure the time at which the algorithm began and then the time at which it finished, then take the difference between the two times to get an estimate of the total time required for the algorithm to complete one batch of processing. This would typically be calculated in microseconds. To measure the computational speed of hardware algorithms, one must count the number of hardware clock cycles required to process one batch of data, then multiply that number by the clock speed of the hardware system. This can be done using the hardware design diagram by counting the pre-determined run-time of sub-elements.

5.3.2 Mean Square Error

Mean square error is useful to see the accuracy of the estimated output signals given the original input signals. Below is a graphical example:

Graphs titled Signal 1, Signal 2, Error, and Square Error in figure 5.1 will be referred to as graphs a, b, c, and d respectively. Graphs a and b include plots of two original sources (solid) and their corresponding estimates (dashed). Graph c shows the error for both signals and graph d shows the square error for both signals. It is important to note the scale on the error plots.

In this particular example we see that the error for each signal resembles the other signal. This indicates to us that the error in this case is due to an incomplete separation of the sources given mixtures of the originals. By using the MSE as a measure of error, we can measure the degree to which the ICA algorithm successfully completed its task.

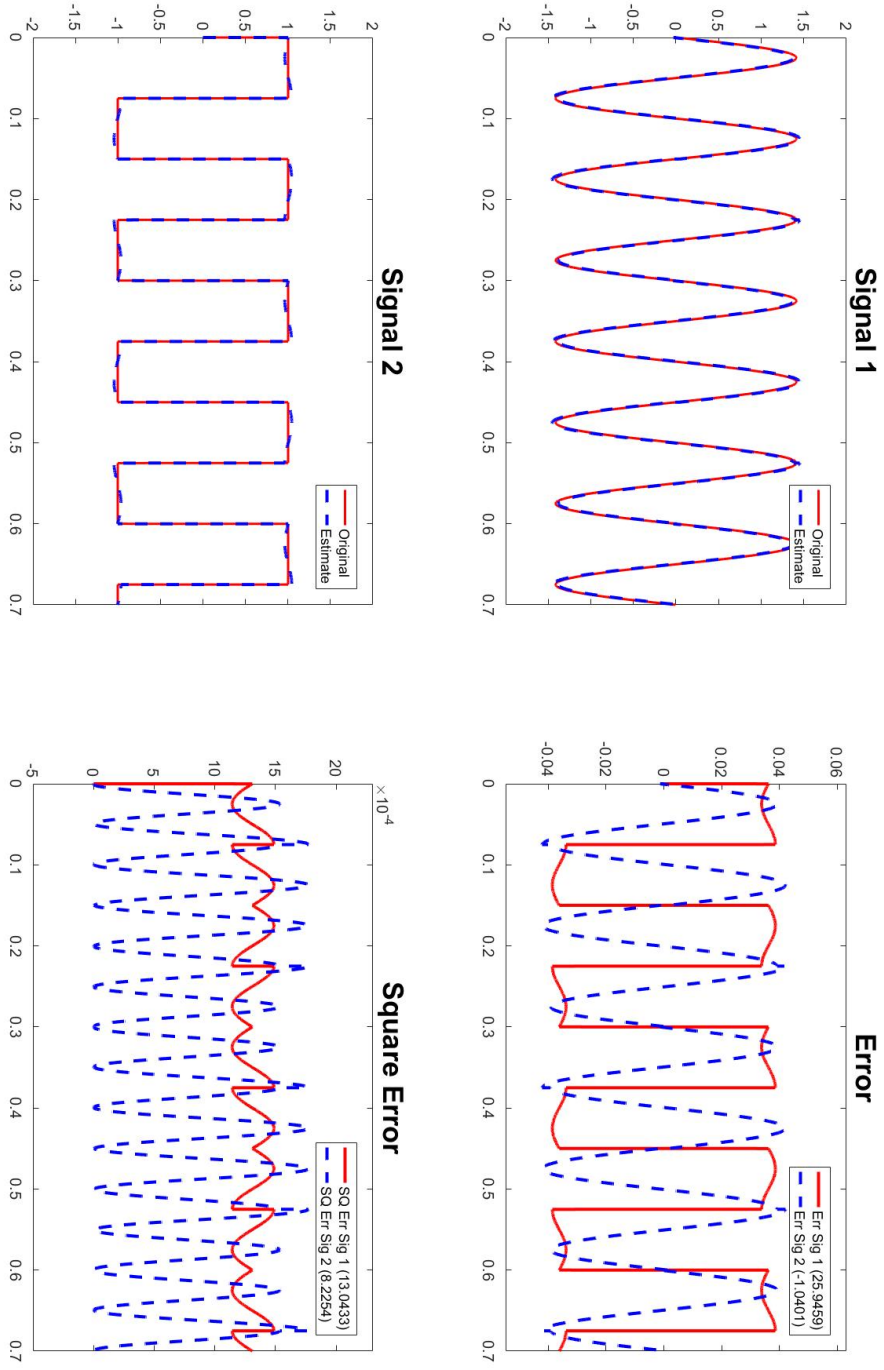

Figure 5.1: Mean Square Error - Graphical Example

5.3.3 Mixing Matrix

A comparison based on the accuracy of the separating matrix would tell us directly the accuracy of the ICA algorithms. However, when carrying out simulation comparisons the total error between calculated separating matrices and the randomly generated original matrix is outside the range of typical error between two randomly generated matrices. In other words, when we do an element-by-element subtraction between the estimated and original matrices and then take the sum of the squares of the result of the subtractions, we typically see a value which is within the range of typical values if we were to carry out the same operation on two randomly generated matrices. Therefore, comparison by mixing matrix using the sum of the square error is not a good measure of performance. In future work it may be possible to use a different measure of comparison between the two matrices.

5.3.4 Signal to Noise Ratio (SNR)

SNR is a useful measure by which to compare the two algorithms because by their nature, the ICA algorithms maximize the non-gassianity of our set of signals. The more non-gaussian a signal is, the smaller the SNR will be. SNR is calculated using the frequency spectrum. We know that the gaussian source will contribute a flat, evenly distributed spectrum, while the sine or square wave signals will give us spectrums which we can predict.

5.3.5 Simulation Results

5.4 Simulation Results

Both algorithms exhibit very good results with the simulation signals. Figure 5.2 shows the simulation output for the JADE algorithm. This is for two signals, a sine and square wave, and is organized into three main columns. The first two plots show the original sources, the next two plots in the center column

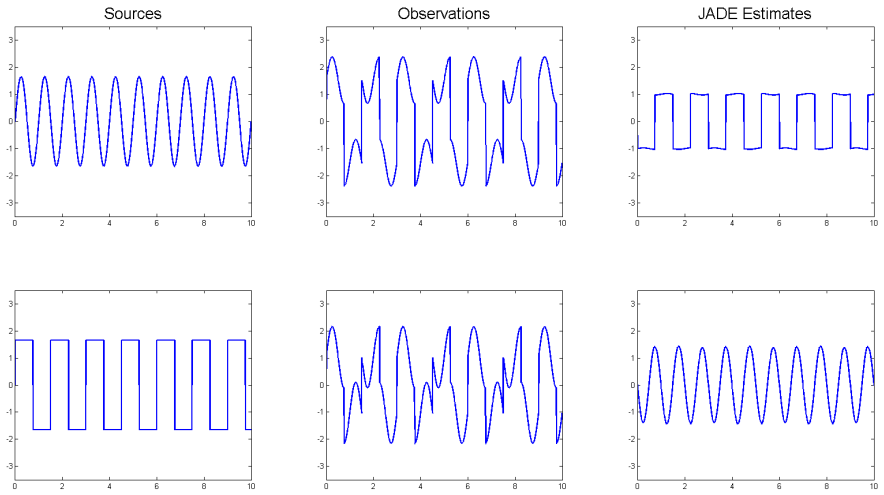

Figure 5.2: Example JADE Simulation Output for a sinusoid and square wave

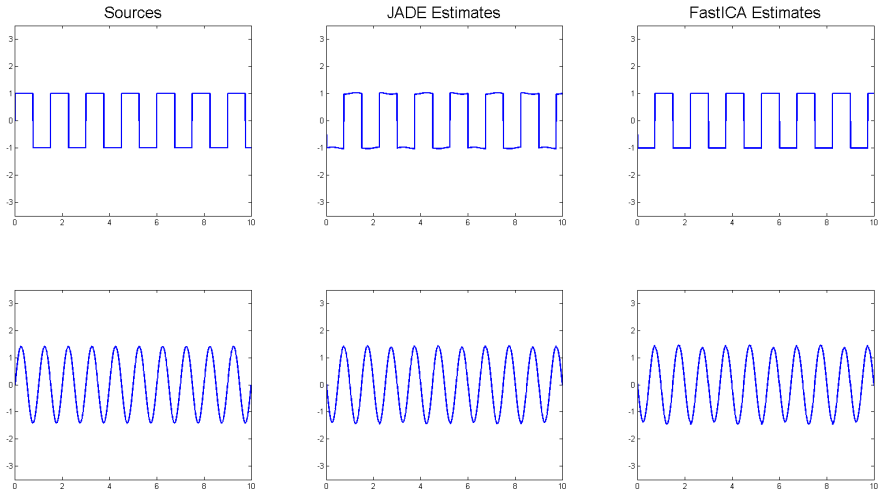

Figure 5.3: Example Comparison of JADE and FastICA

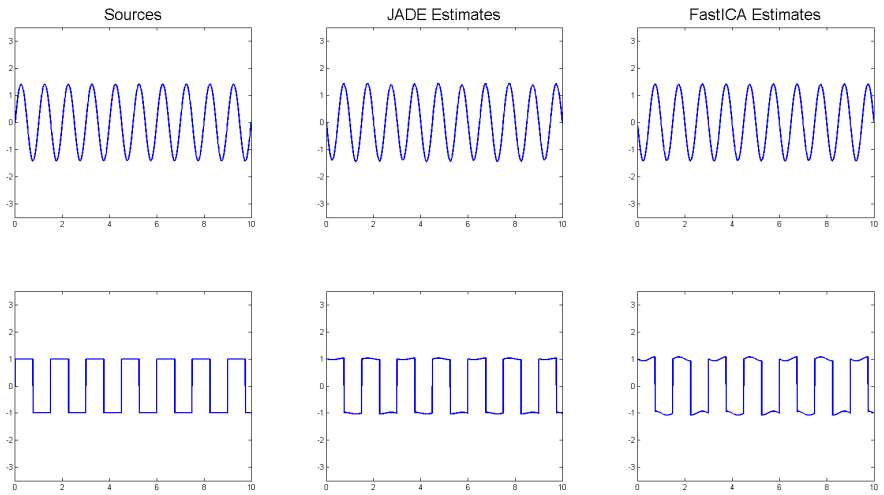

Figure 5.4: Example Comparison of JADE and FastICA

show the two observations generated by the mixing of the two original sources, and the last two plots show the signal estimates which the JADE algorithm produced.

Figures 5.3 and 5.4 show signal estimates produced by both the JADE and FastICA algorithms given the same set of input observations. The general trend seems to be that FastICA is better than JADE, but only if the sources are separated in a specific order. For example, if the sine wave was separated first and the square wave second, the JADE algorithm would typically have better results. However, if the square wave is separated first, the FastICA algorithm would typically show better results. In the average case, JADE performs with slightly more MSE, but with less variation between simulations.

CHAPTER 6

HARDWARE RESULTS

In this chapter we will aim to verify that our hardware algorithms performed accurately compared to our software simulations, and to compare the two hardware algorithms with each other based on several different metrics. Verification of our algorithm will be by comparison of BSS outputs given the same set of observed inputs. Comparison metrics include separation accuracy based on MSE, speed of operation, and hardware spaced used (in the case of hardware-to-hardware comparison).

6.1 Verification by MSE

For verification of the hardware design, we take the data stored on the internal FPGA Block RAM and save it to a location on the computer. This data is then parsed to a form which can be used by the simulation algorithms. After simulations on the same set of inputs as the hardware, the outputs are compared for similarity and separation accuracy. If both hardware and software algorithms output similar results we have verified that the hardware algorithm operates at a comparable accuracy to the simulation algorithms, and therefore verified that the hardware algorithms are, in fact, separating signals correctly.

Another way to verify the FPGA design is to simulate the hardware design. This is clearly the more desired option in many cases as a hardware test setup is not necessary. With a test bench, we can take our hardware design and apply a set of pre-calculated inputs, such as a simulated data stream, to see

what the resulting output is. This method of testing very popular and is used in many different peer reviewed articles as verification of hardware design. Test bench simulation is an essential part of hardware design development and was used extensively in the development process; for example, to test complex sub-modules before integration into the main design.

Verification examples can be found in Appendix B. Figure 6.1 shows a two-signal test case for the JADE algorithm. On the left are the set of observed signals, in the middle are the simulation algorithm source signal estimates, and on the right are the estimates from the hardware algorithm. Figure 6.2 shows the same for the FastICA algorithm. The MSE between software and hardware for the verification in Figure 6.1 is 0.0081 for source 1 and 0.0078 for source 2. The MSE for the verification in Figure 6.2 is 0.0013 for source 1 and 0.0014 for source 2. Because these values are technically in terms of voltage, translating them to a real-world system shows negligible error. Based on accuracy of separation, we have verified that both hardware algorithms work with similar accuracy to their software counterparts, and therefore are eligible for further testing and comparison based on other metrics.

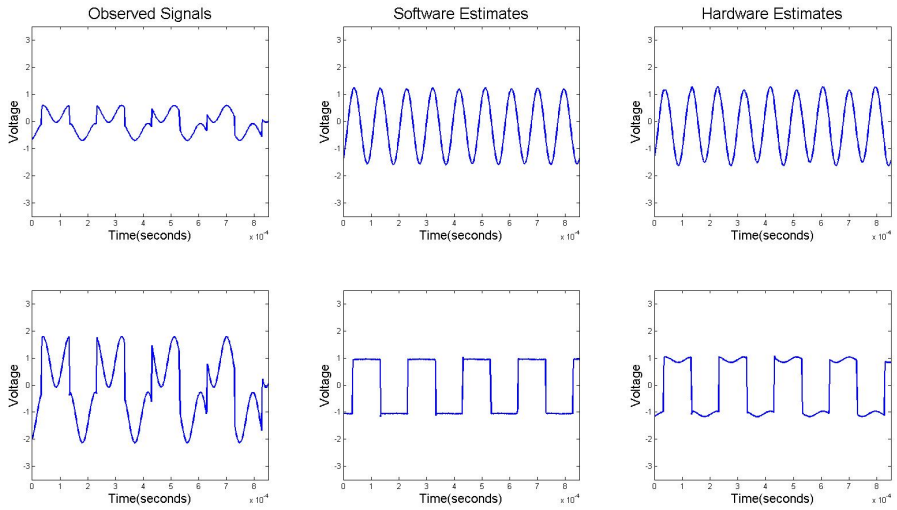

Figure 6.1: JADE Hardware Verification Exmaple

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.0058178	0.00027175	0.00358
Source 2	0.0058367	2.3029×10^{-05}	0.0051219

Table 6.1: JADE Hardware Verification Example MSE Values

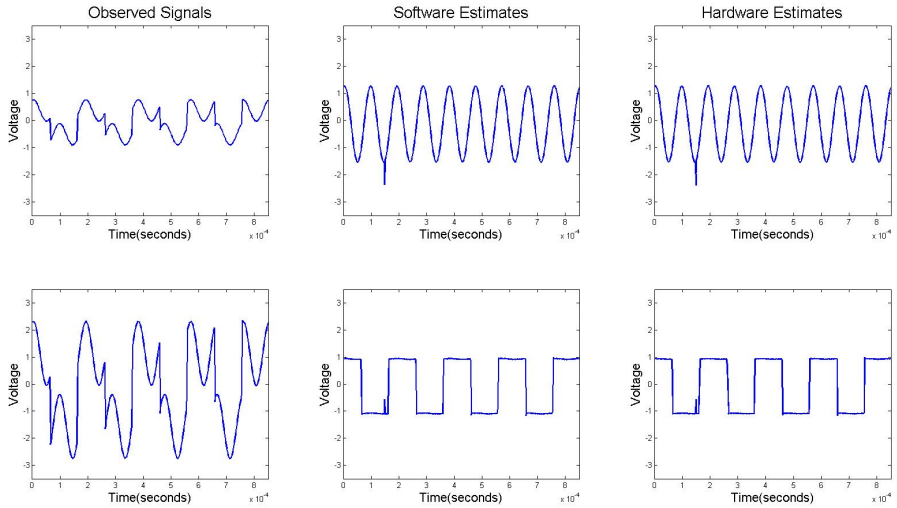

Figure 6.2: FastICA Hardware Verification Example

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	1.934×10^{-05}	4.242×10^{-06}	4.1697×10^{-05}
Source 2	1.9545×10^{-05}	0.00011596	4.0289×10^{-05}

Table 6.2: FastICA Hardware Verification Example MSE

6.2 Speed of Separation

Speed of separation is measured differently for the hardware and software algorithms. For the software algorithms, internal computer clocks are used, and for the hardware algorithms, speed of separation can be measured by directly counting the number of clock cycles in the designed system. Another consideration in measuring the speed of separation of software algorithms is that the system may not have consistent run-time as the algorithm is running on top of an operating system. As such, it is necessary to take the average of many different runs of the same algorithm on the same set of generated inputs.

Another consideration of speed of separation is the definition of systems. For the hardware algorithms, we may include the ADC and DAC only if we are considering our DAQ system in the software algorithms. We should consider comparison of the DAQ systems across both hardware and software separately from the actual algorithms themselves. In extension, it may also be desireable to compare the ICA algorithms separately from the shared whitening stage. By considering the ICA algorithms by themselves, minimizing common processing overlap, we maximize the difference between the performance of the algorithms.

To compare the ICA algorithms directly in the software algorithms we start a timer right before the ICA algorithm begins and stop it right after it completes. This ignores the time for whitening and other overhead tasks and solely focuses on the ICA algorithm. Likewise for the hardware algorithms, we simply count only the clock cycles required in the ICA algorithm sub-module.

After testing with the JADE algorithm, we saw separation speeds in simulation of between 2 and 4 milliseconds while the hardware system has a total counted latency of 0.8 milliseconds.

generated inputs, and the other with hardware generated inputs. Detailed results can be found in Appendix B Section 2. Below in [figure-. . .] is a summary of the results. [insert stuff about computer specifications and operating system, and about hardware system].

Software inputs are generated using custom MATLAB functions, and stored within text files for use in the non-MATLAB algorithms. For the hardware algorithms, the inputs are generated using signal generators, and mixtures are generated using simple operational amplifier-based mixing circuits. These are the stored within the FPGA on Block RAM, and extracted using an Arduino. The extraction process is as follows:

1. A switch on the Nexsys 4 board is set causing the system to store the latest dataset on the different sets of Block RAM and enter into a paused state.

2. The Arduino then clocks the contents of the RAM out via digital IO pins and sends the newly captured data to the computer via serial interface.

3. A custom program on the computer listens for data on the serial interface and stores the incoming data in a text file at a specified location.

The extracted data is a set of binary strings representing numbers in IEEE 754 single precision float format which must be converted by our software algorithms and test benches before use in simulations. By storing the data as a binary string (‘1’ and ‘0’ only), we ensure that precision is not lost between the hardware and software algorithms, removing the factor of conversion error when comparing software and hardware algorithms.

For the MATLAB versions of the algorithms, the software test signals are generated within our algorithm scripts while the hardware test signals are imported via text document. Binary strings are converted using typecasting functions. For the hardware algorithms, we use the analog generated inputs directly, and test the hardware algorithm with software inputs using a test bench. While it is possible to reverse the extraction process and insert the software-generated signals into the hardware algorithm, it is unnecessary and extraneous as the behavior of the hardware algorithm has already been verified; instead, we use a test bench to simulate the hardware behavior.

6.3 Hardware Space

Hardware space comparison is only for comparison of the two hardware algorithms with each other. Metrics to consider when performing a hardware comparison include number of Look-Up Tables (LUT) used, number of Flip-Flops (FF), amount of Block RAM (BRAM) used, and number of DSP slices used. Our selected FPGA, the Artix-7 (XC7A100T), has 15,850 logic slices, each with four 6-input LUTs and 8 FFs. By measuring the number of LUTs and FFs used in the design, we resolve to a general idea of how many logic slices are used.

Our first hardware space comparison is for the 2-signal cases of the hardware JADE and FastICA algorithms. Based on Xilinx Vivado synthesis and implementation results, we list out the metrics in the following table.

	JADE	FastICA
DSP Slices	133(55%)	277(95%)
LUT	42,851(68%)	31,571(50%)
FF	65,526(52%)	49,972(39%)
BRAM (bits)	108,000(2.2%)	108,000(2.2%)

Table 6.3: Hardware Space Results

While the FastICA algorithm uses fewer LUTs and FFs, we see that the JADE algorithm used fewer DSP slices. All of the metrics are important to an FPGA design, but DSP slices improve speed of design greatly. As this is a much more scarce resource, we give it more weight. Overall, it appears that the JADE algorithm uses fewer resources to separate two sources from a set of two observations.

The next step at this point is to compare the two algorithms given a higher number of incoming observations. As stated previously, the size of the fourth order cumulant matrix used in the JADE algorithm increases drastically

with a higher number of input sources. Because of this, the JADE algorithm will have to be re-designed to maximize re-use of hardware structures. The JADE algorithm will generally take up more hardware space than the FastICA algorithm which inherently re-uses hardware structures. After re-design, however, the JADE algorithm will have very low hardware usage at the cost of larger computation time.

The amount of hardware space used can be adjusted for each algorithm by serializing the JADE algorithm or parallelizing the FastICA algorithm. Serializing the JADE algorithm more decreases the hardware used by repeatedly using the same hardware structures. Parallelizing the FastICA algorithm increases hardware used, but improves processing speed greatly. For example the FastICA algorithm can be set up in such a way that one structure separates two sources directly. That structure can then be re-used to estimate sources two at a time as opposed to one at a time. The hardware space increases, but the overall processing time decreases.

CHAPTER 7

CONCLUSION

The main purpose of this research was to provide a proof of feasibility for the JADE algorithm and compare it with another hardware implementation of the FastICA algorithm. Another goal was to explore optimizations to both the software and hardware algorithms. Verification that these two algorithms work correctly was done both visually and numerically.

Feasibility of implementation was used as a topic for masters theses in [5] and [4]. In both cases, the author presents hardware simulation results, but does not actually implement and test the implementations on a real system. In contrast, this thesis has verified a real hardware implementation of two different algorithms and shown comparison between the different algorithms.

A systolic architecture implementation of the JADE algorithm is proposed and simulated. This architecture provides faster results at the cost of higher resources used. Overall, the systolic architecture maintains a good performance to resource ratio, increasing algorithm speed very cost-effectively. In addition, the whitening block is also implemented using a systolic architecture which benefits both algorithms by reducing the overhead of the pre-processing stage.

Both hardware algorithms were run on an Artix-7 XC7A100TCSG324-1 FPGA using the pmod AD1 and DA2 from Digilent. Hardware internal to the FPGA is run at 100 Mhz while the input and output through the ADC and DAC peripheral modules (Pmod) are limited to approximately 590 ksps due to hardware limitations of the ADC and DAC chips themselves. Overall system

speed is limited by the ADC and DAC maximum speed. Given this limitation the system is still potentially highly effective for audio processing applications or other low-frequency applications where the sources of interest have a frequency lower than 290 khz.

A pipelined architecture is also used in the implementation of both algorithms such that the flow of data is constant. In doing so a design technique is also presented in which a hardware design can be broken up into different segments referred to as processing chains to maximize the amount of parallel computation. This effectively reduces the overall system latency.

7.1 Future Work

In general, future work will be aimed at improving the algorithms and exploring new applications. Four-signal hardware algorithms should be implemented, and optimizations should be tested in both the hardware as well as the software algorithms. The software algorithms should also be implemented using a dedicated, high speed computer interfaced with a DAQ to see how a practical software implementation would compare with the hardware implementations. MATLAB and C/C++ simulations should also be run on that same computer to measure the speed of just simulation without the overhead of a DAQ.

Different methods of diagonalization can be explored for the JADE algorithm. For example, it may potentially be more cost-effective in terms of hardware resources to use COordinate Rotation DIgital Computer (CORDIC) processors in the systolic array sub-modules instead of the algebraic equivalents.

A sorting method should also be included for the post-whitening stage to ensure a consistent order of outputs and reduce permutation ambiguity. This would re-order our eigenvectors based on the eigenvalues.

The existing pipelining scheme can also be changed to adjust the amount of parallel computation and required RAM. Depending on the application, speed

or resources may become scarce, necessitating these types of adjustments.

Both algorithms should also be tested using live-captured signals such as audio channels generated by two different speakers in the lab, captured using an array of microphones. Software simulations should also be carried out to explore the feasibility of application in alternative areas such as improvement of compression algorithms. If the software shows feasibility, then the hardware can ultimately be used for high speed DSP applications such as a high speed data compressor which could potentially increase data transfer rates between satellites or improve the speed of existing internet frameworks. With an FPGA implementation, a high speed data compressor can be very useful.

Ultimately, these algorithms can be used for a wide variety of different applications, and the design techniques developed for the implementation of these algorithms can be used to implement other useful algorithms on FPGAs.

APPENDIX A

1 Cumulants

Unrolling the 4-dimensional cumulant matrix into three dimensions yields a representation that looks like the following. Figure 3.1 from Chapter 3 Section 2 shows a 2-signal case and A.1 below shows a 4-signal case.

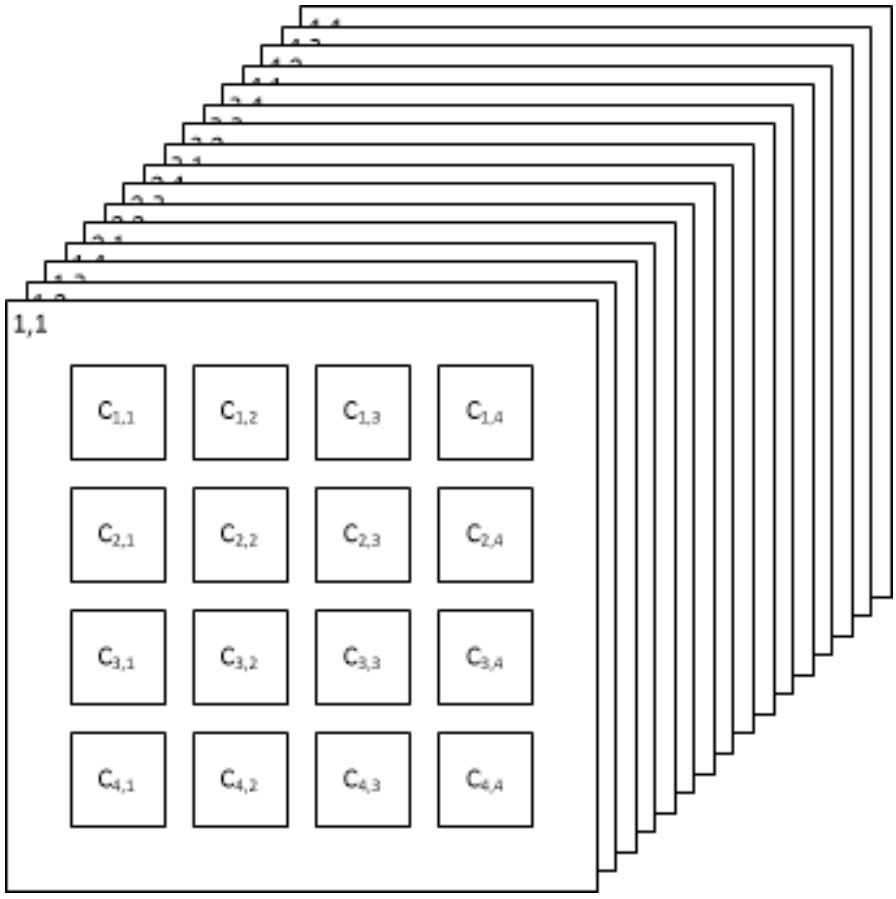

Figure A.1: Fourth-order cumulant matrix for 4-observation case ($n = 4$)

Unrolling the 4-dimensional cumulant matrix into two dimensions yields a representation that looks like the following. Figure A.2 shows a 2-signal case and A.3 shows a 4-signal case.

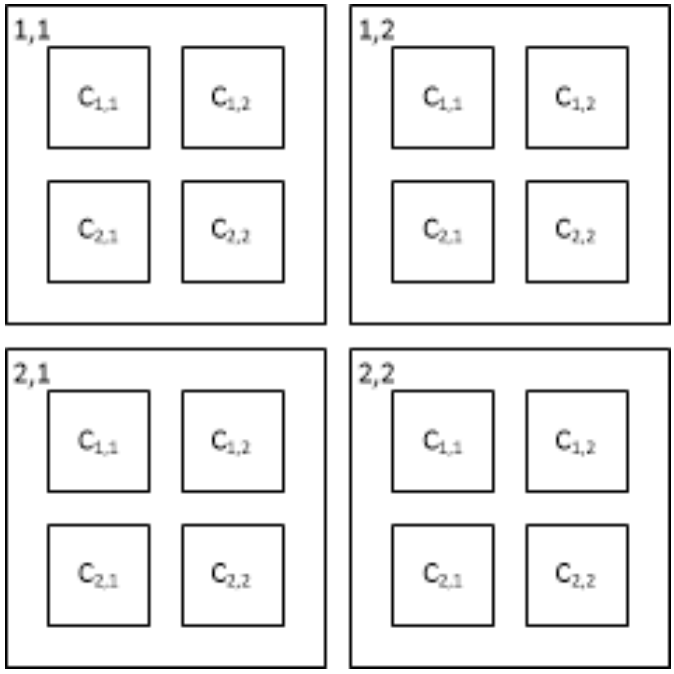

Figure A.2: Fourth-order cumulant matrix for 2-observation case ($n = 2$)

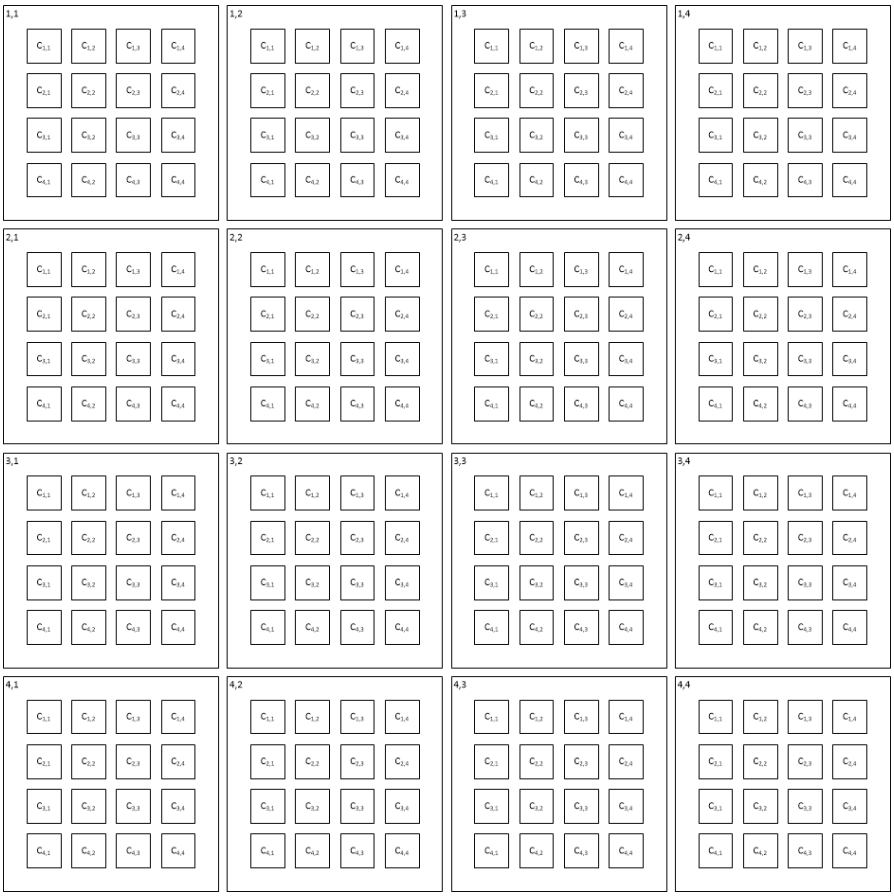

Figure A.3: Fourth-order cumulant matrix for 4-observation case ($n = 4$)

The super-symmetric nature of the fourth-order cumulant matrices means there are many repeated cumulants. Figures A.4 and A.5 are good ways to visualize the quantity and location of unique cumulants for the 2 and 4-observation cases respectively. For higher numbers of observations, it is easy to imagine what the equivalent figures would look like.

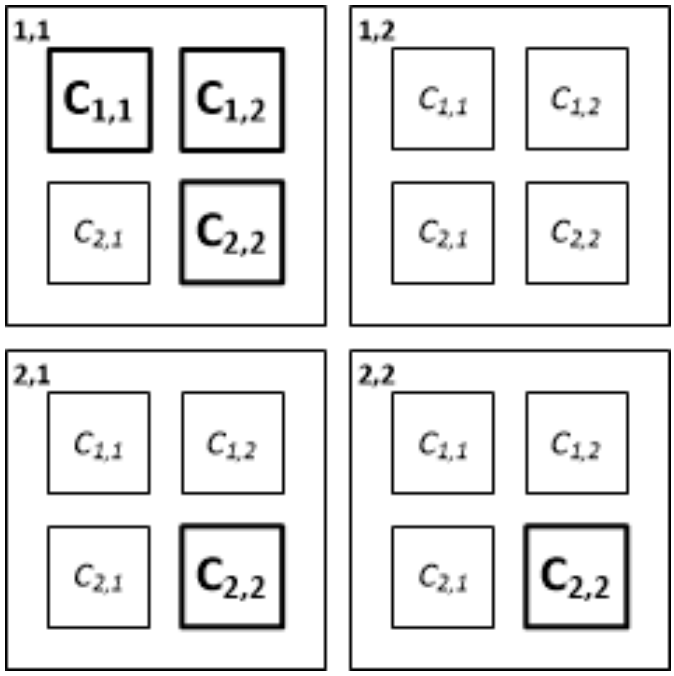

Figure A.4: Unique cumulants (bold) in the fourth-order cumulant matrix for 2-observation case ($n = 2$)

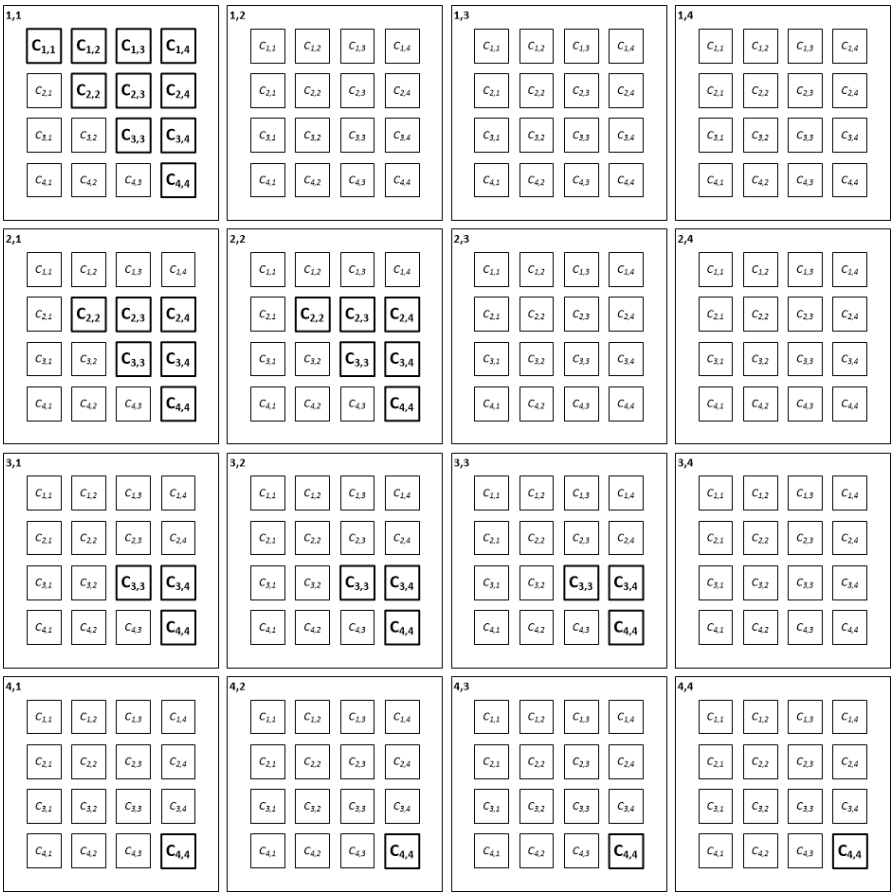

Figure A.5: Unique cumulants (bold) in the fourth-order cumulant matrix for 4-observation case ($n = 4$)

2 Jacobi/Givens Rotations

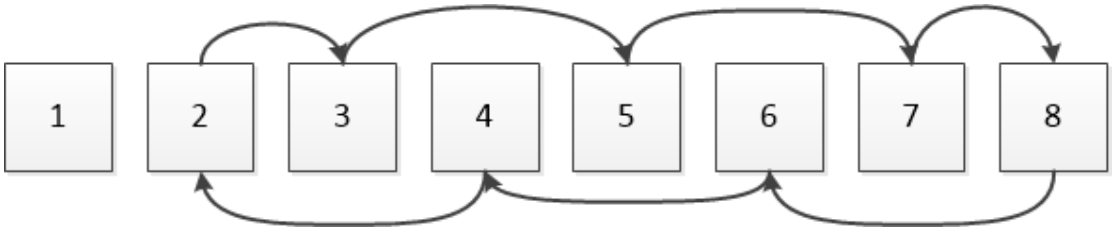

Figure A.6: Jacobi rotation pattern on a single row.

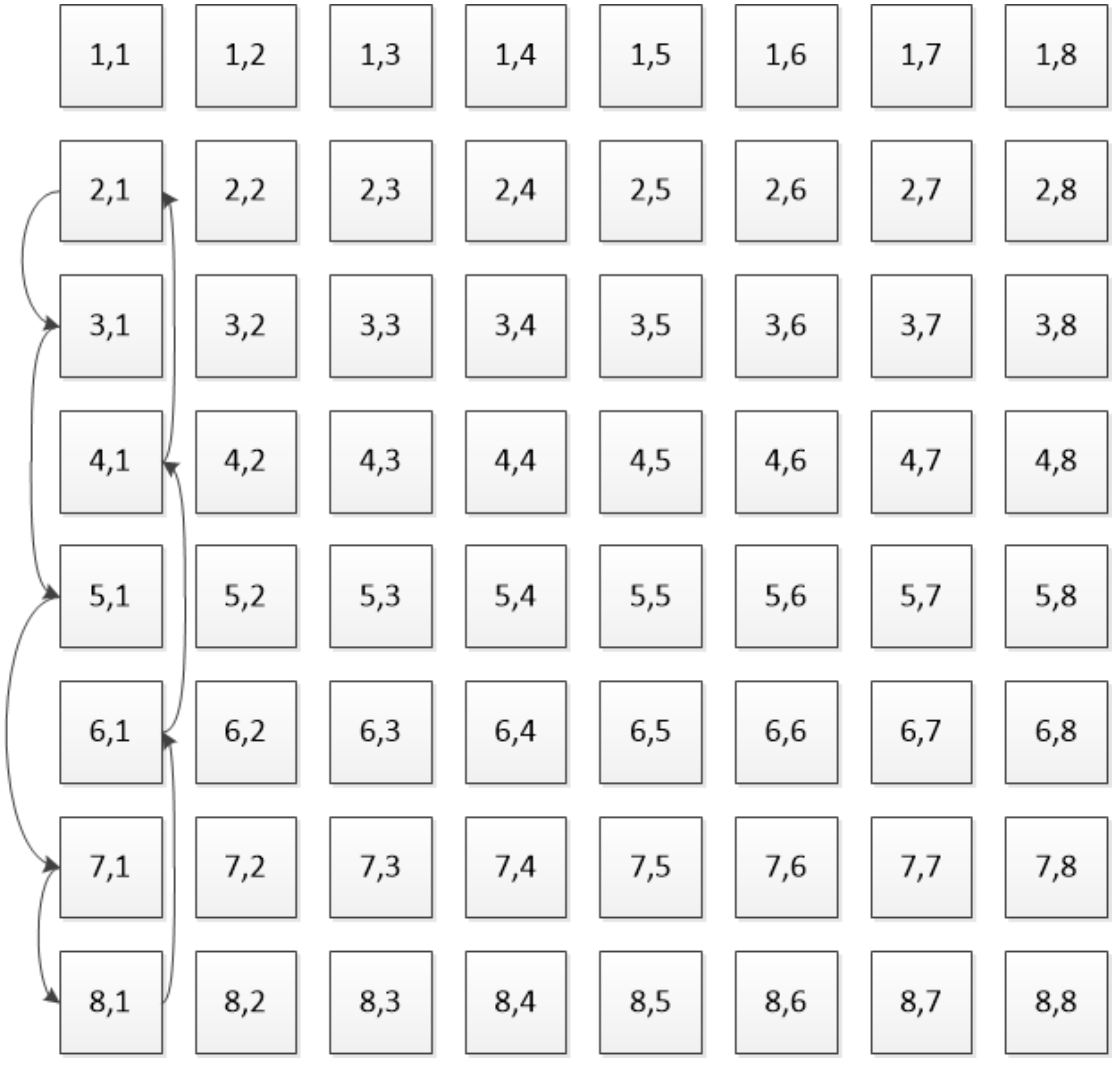

Figure A.7: Jacobi rotation pattern for rows and columns.

Figure A.8 shows the Jacobi rotation pattern for a 4×4 matrix. Notice how elements are rotating in sets of three for the 4×4 dimensional case: {(1,2),(1,3),(1,4)}, {(2,1),(3,1),(4,1)}, {(2,2),(3,3),(4,4)}, {(2,3),(3,4),(4,2)}, and {(3,2),(4,3),(2,4)}.

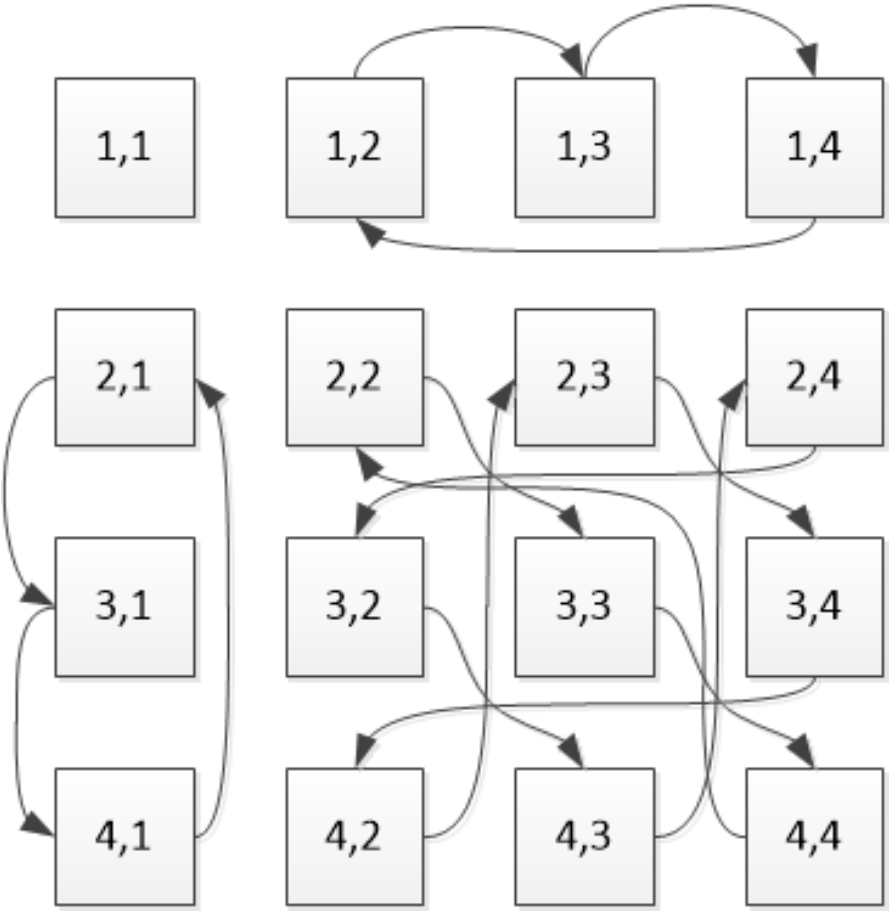

Figure A.8: Jacobi rotation pattern for a 4×4 matrix.

Figures A.9 through A.12 show one full cycle of rotations for a 4×4 matrix.

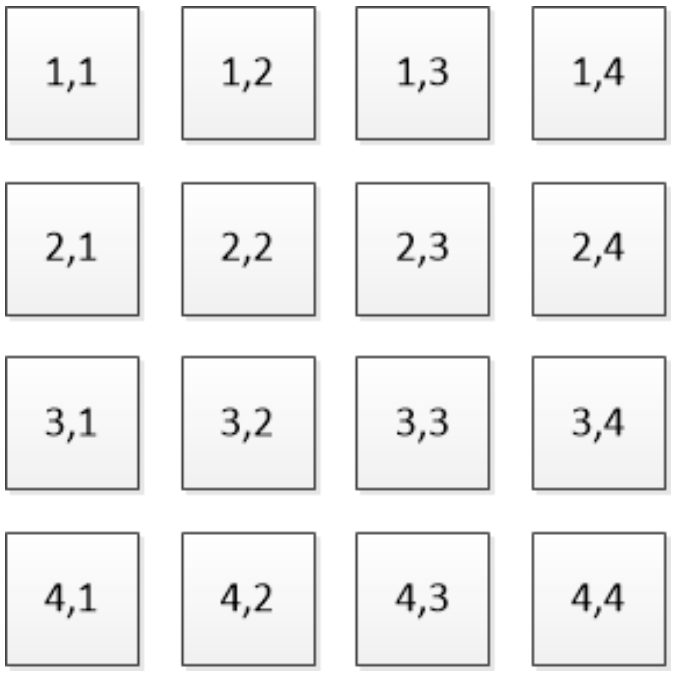

Figure A.9: Jacobi rotation pattern for 4×4 matrix at rotations $(r = 0)$.

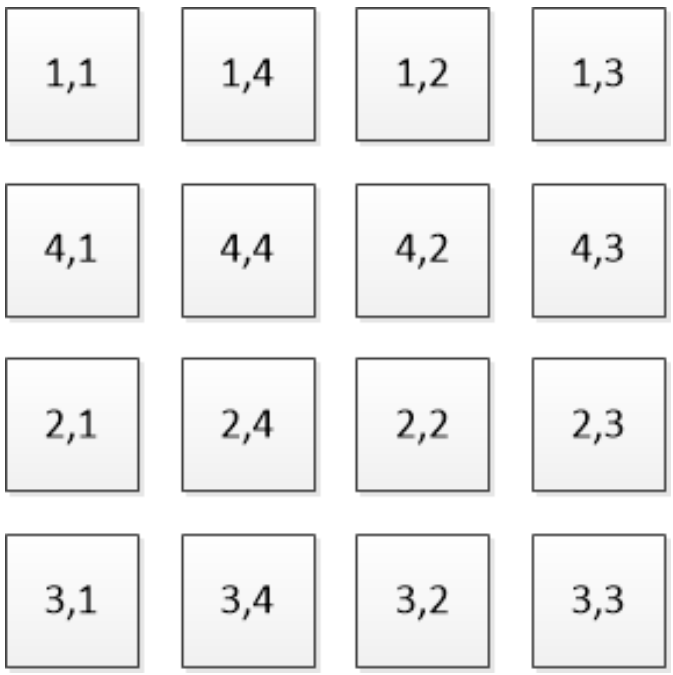

Figure A.10: Jacobi rotation pattern for 4×4 matrix at $(r = 1)$.

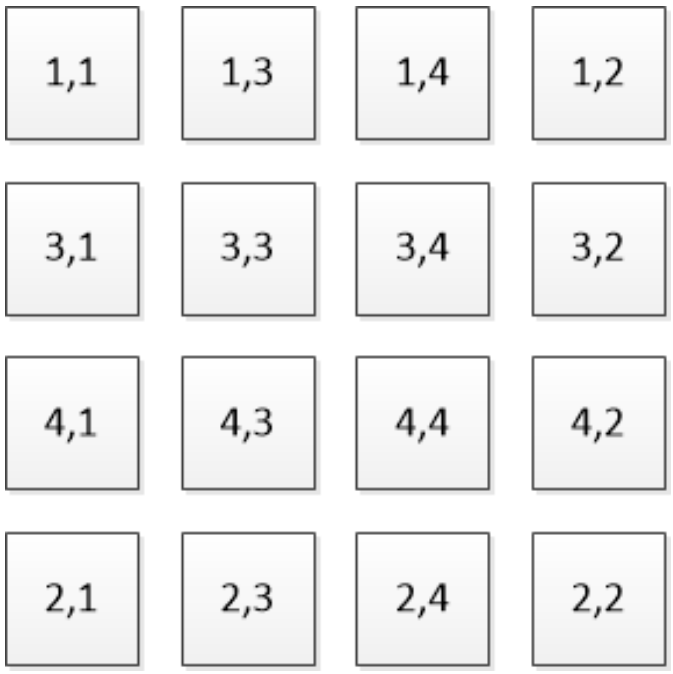

Figure A.11: Jacobi rotation pattern for 4×4 matrix at $(r = 2)$.

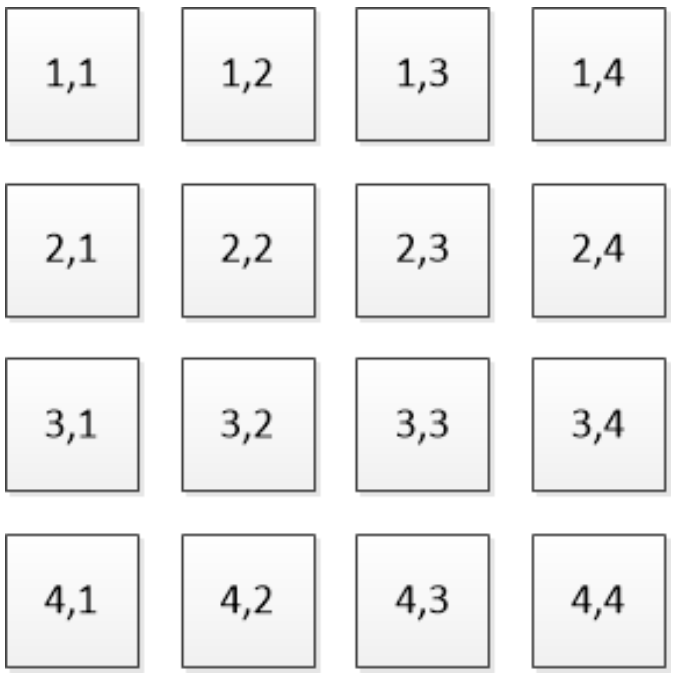

Figure A.12: Jacobi rotation pattern for 4×4 matrix at $(r = 3)$.

Figures A.13 through A.20 show one full cycle of rotations for a 8×8 matrix.

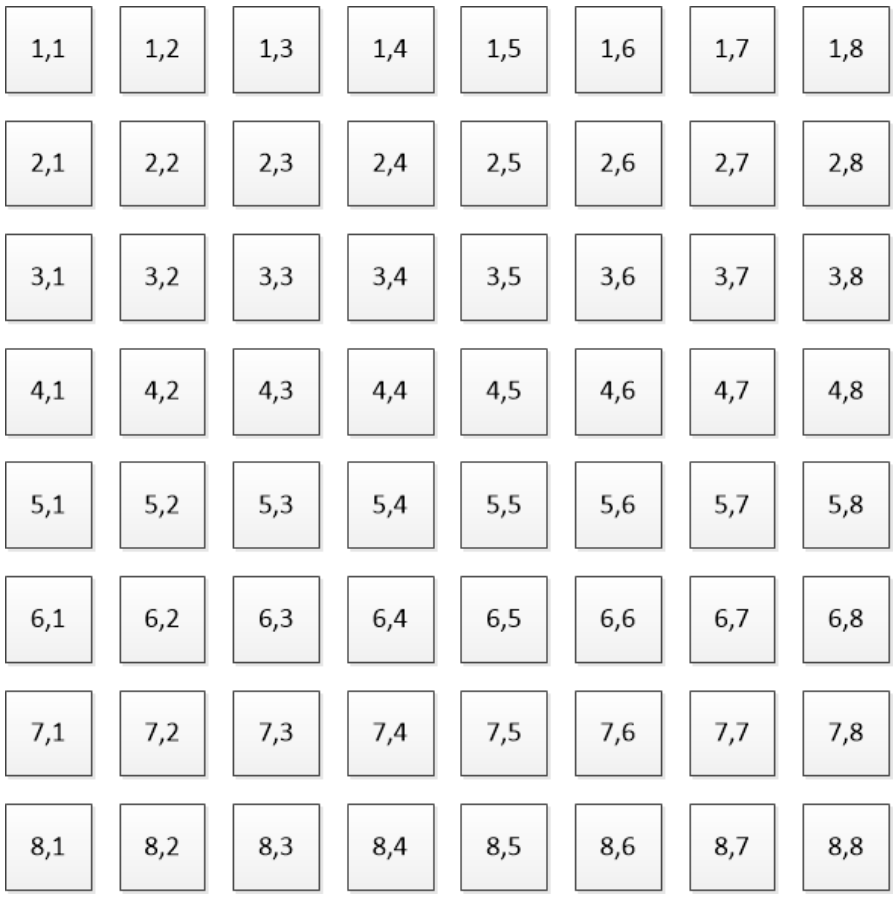

Figure A.13: Jacobi rotation pattern for 8×8 matrix at $(r = 0)$.

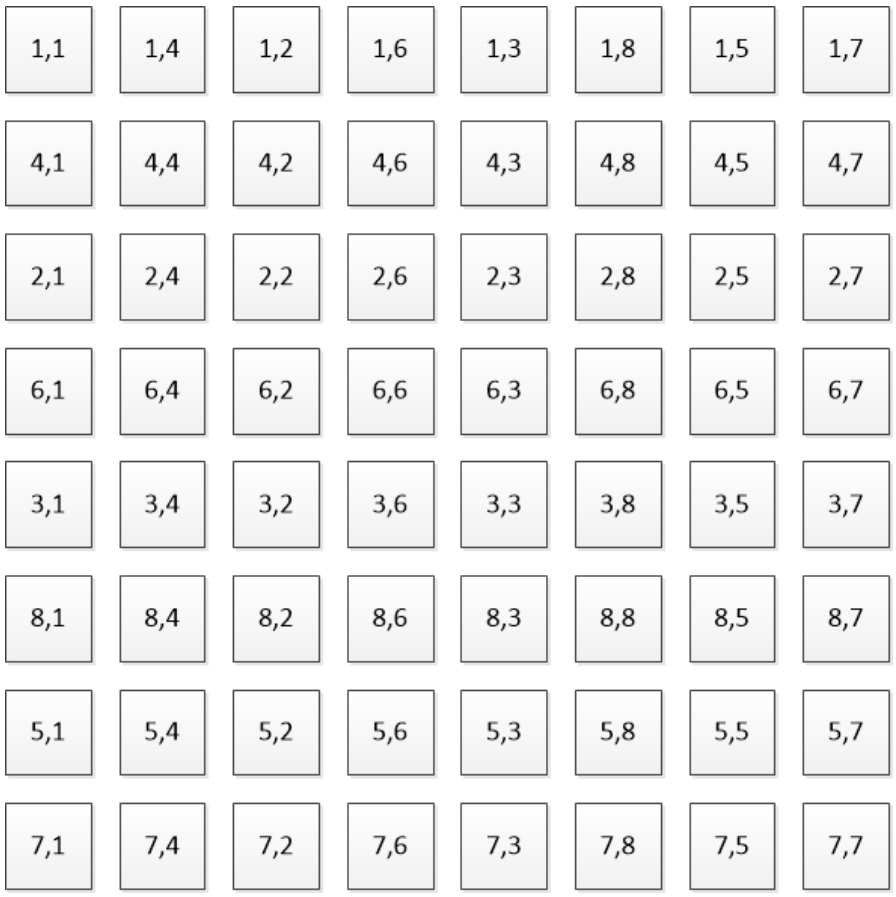

Figure A.14: Jacobi rotation pattern for 8×8 matrix at $(r = 1)$.

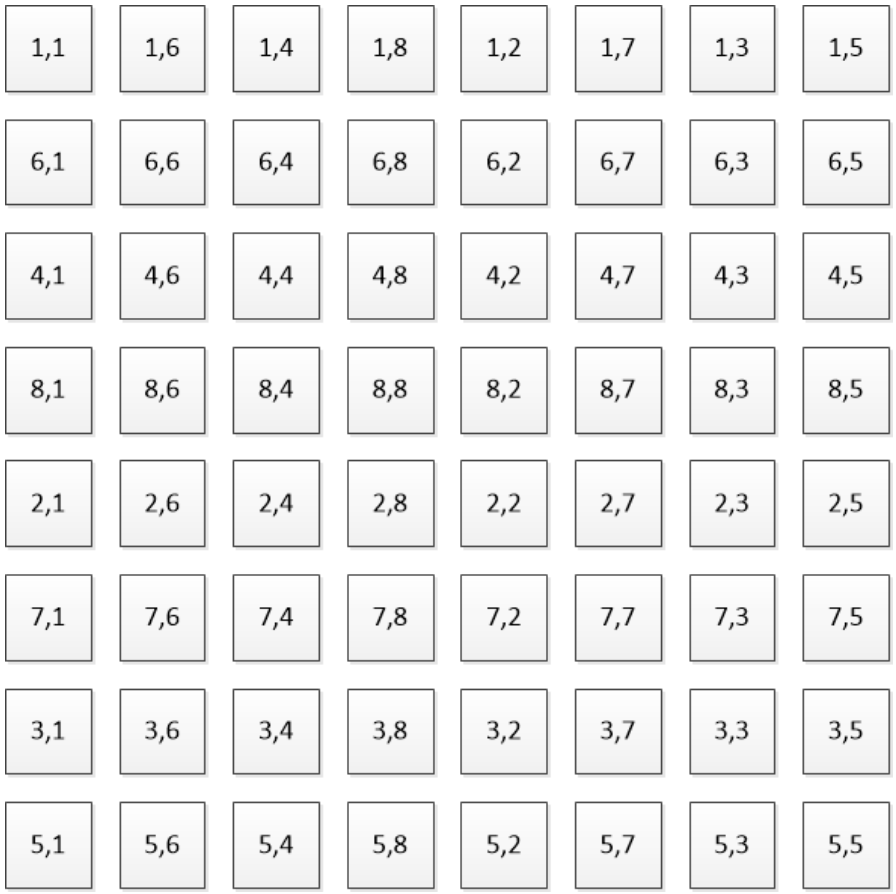

Figure A.15: Jacobi rotation pattern for 8×8 matrix at $(r = 2)$.

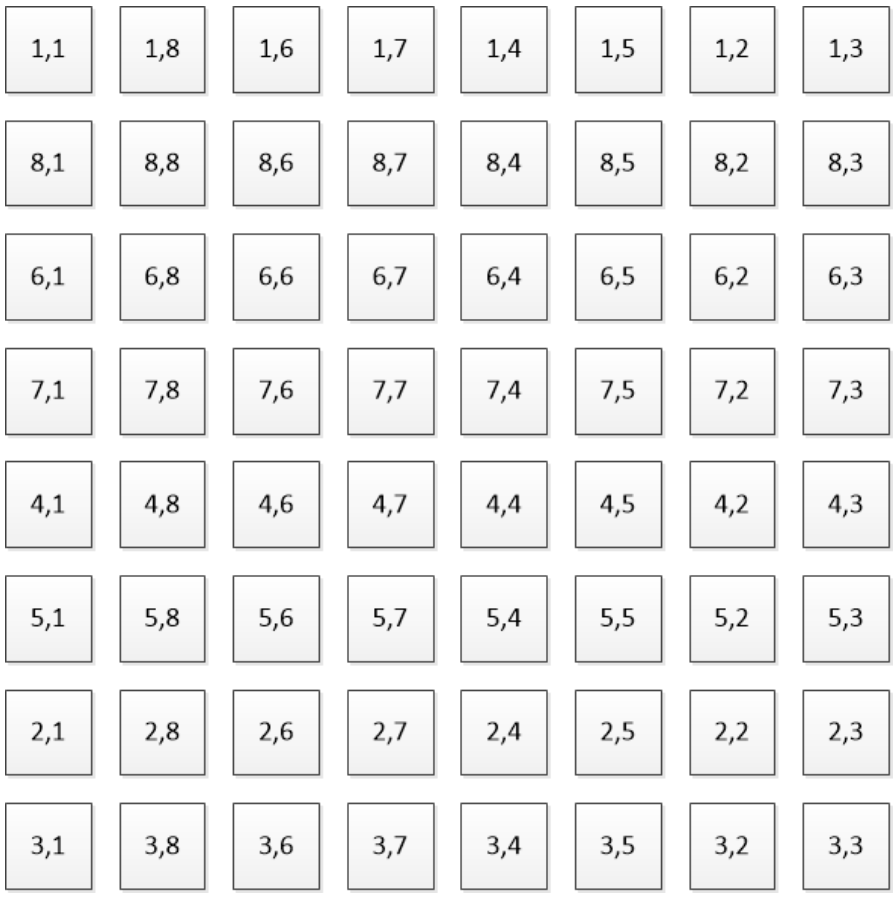

Figure A.16: Jacobi rotation pattern for 8×8 matrix at $(r = 3)$.

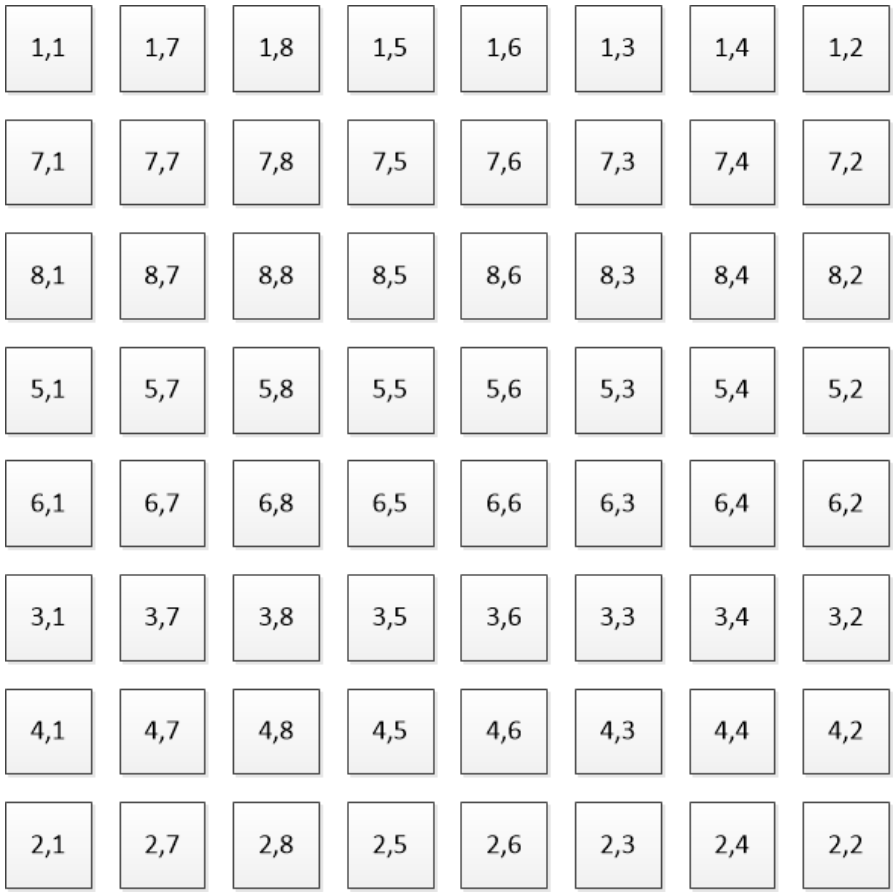

Figure A.17: Jacobi rotation pattern for 8×8 matrix at $(r = 4)$.

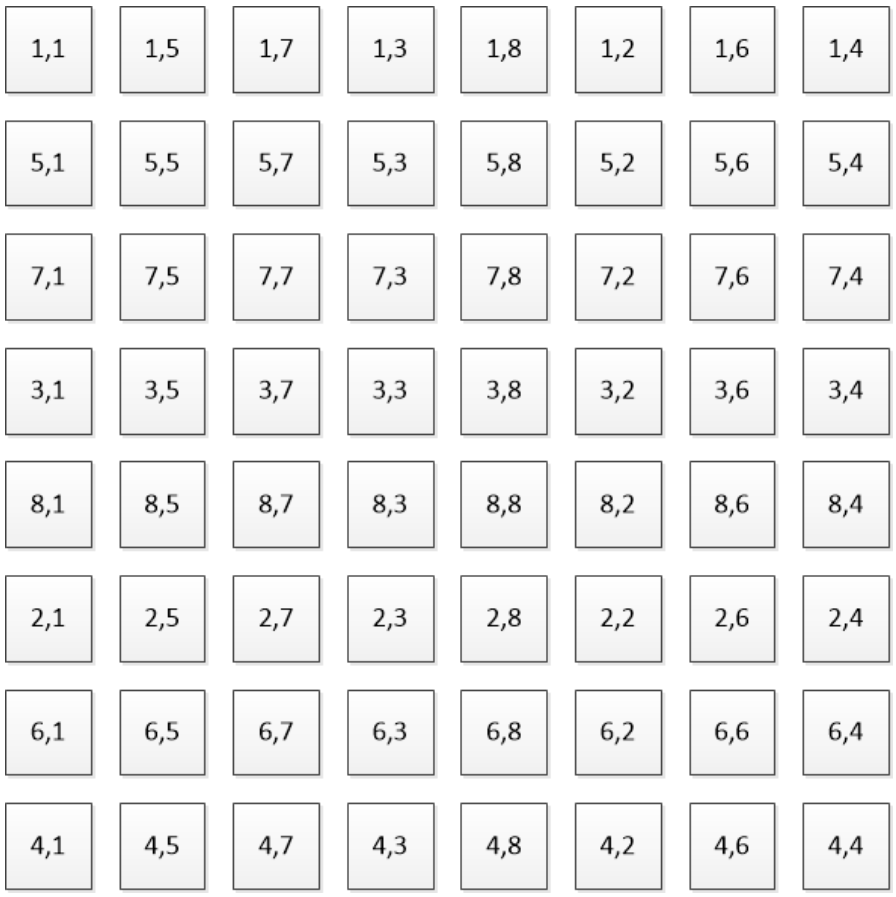

Figure A.18: Jacobi rotation pattern for 8×8 matrix at $(r = 5)$.

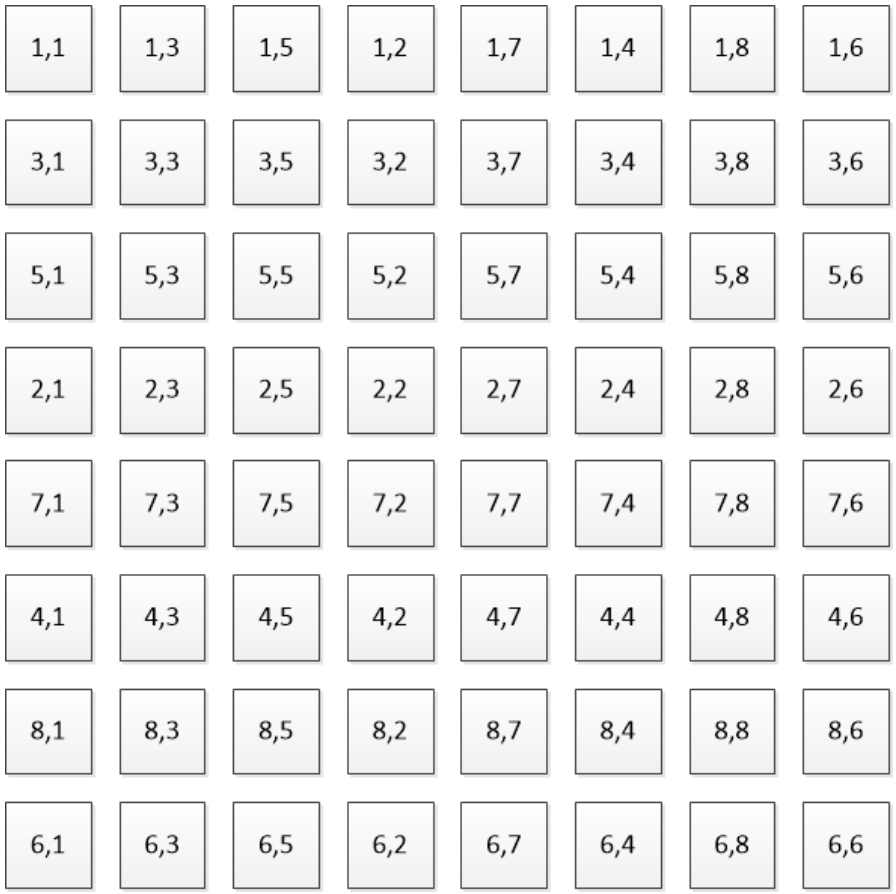

Figure A.19: Jacobi rotation pattern for 8×8 matrix at $(r = 6)$.

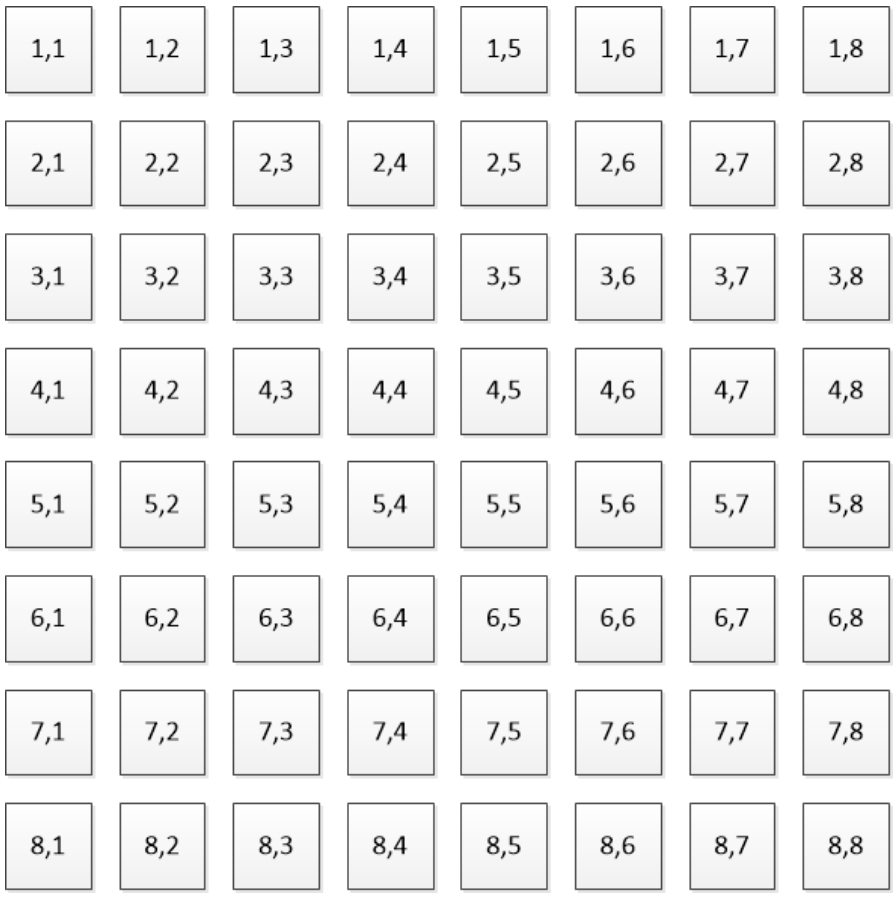

Figure A.20: Jacobi rotation pattern for 8×8 matrix at $(r = 7)$.

APPENDIX B

1 Verification of FastICA Hardware

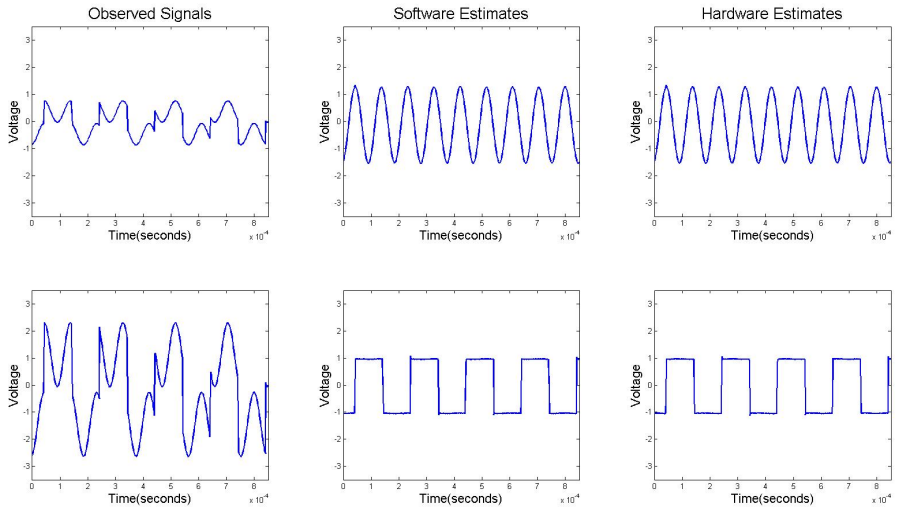

Figure B.1: FastICA Hardware Verification #1

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.00010567	5.0586×10^{-05}	1.0038×10^{-05}
Source 2	0.000106	2.7341×10^{-05}	2.568×10^{-05}

Table B.1: FastICA Hardware Verification #1 MSE

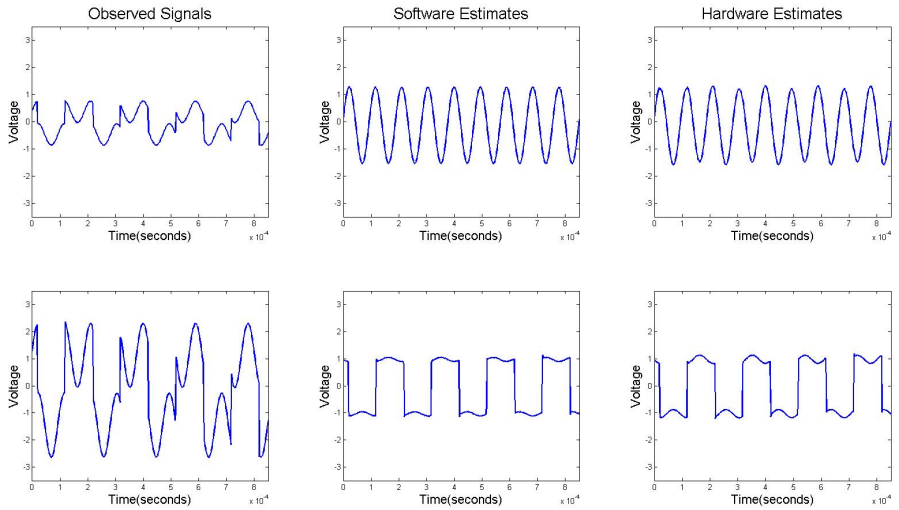

Figure B.2: FastICA Hardware Verification #2

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.0028006	5.3028×10^{-06}	0.0029212
Source 2	0.0027958	0.0025614	0.011432

Table B.2: FastICA Hardware Verification #2 MSE

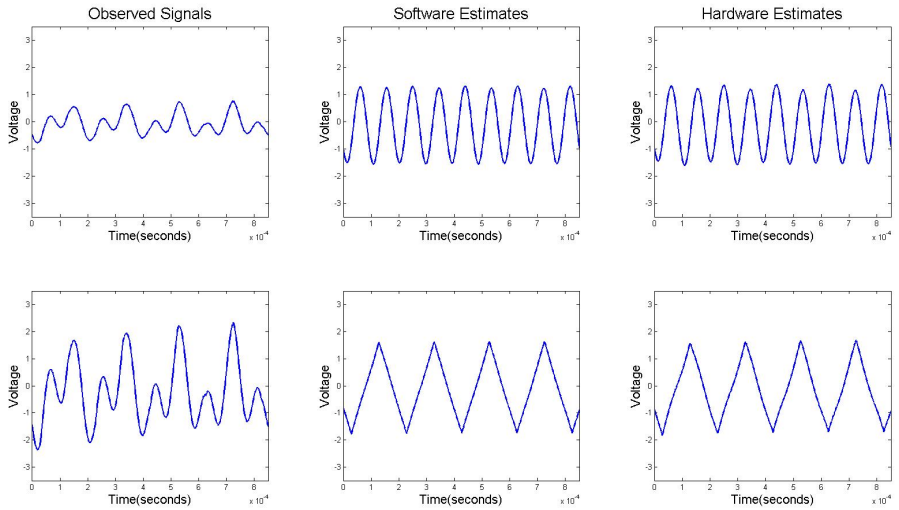

Figure B.3: FastICA Hardware Verification #3

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.0019009	0.0006417	0.0047579
Source 2	0.0019013	0.00040263	0.0040492

Table B.3: FastICA Hardware Verification #3 MSE

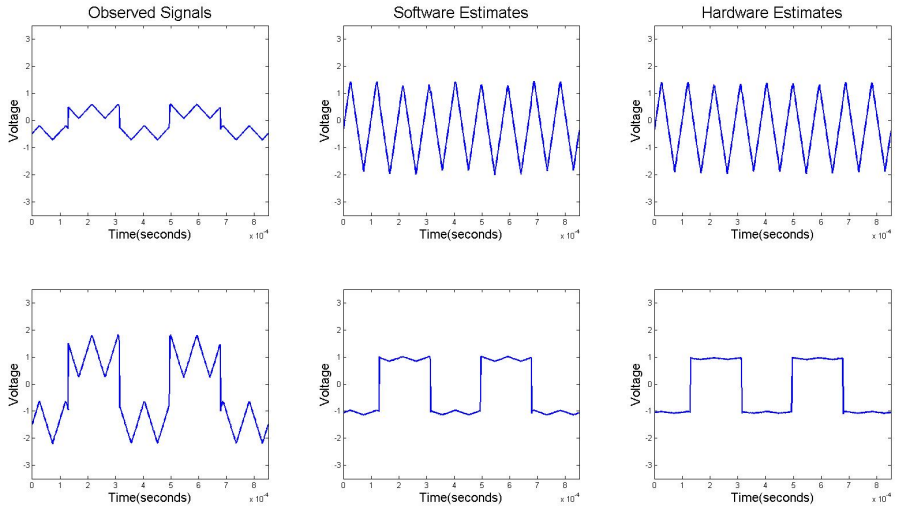

Figure B.4: FastICA Hardware Verification #4

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.0011502	0.0050637	0.0013761
Source 2	0.0011503	0.0026795	0.00031914

Table B.4: FastICA Hardware Verification #4 MSE

2 Verification of JADE Hardware

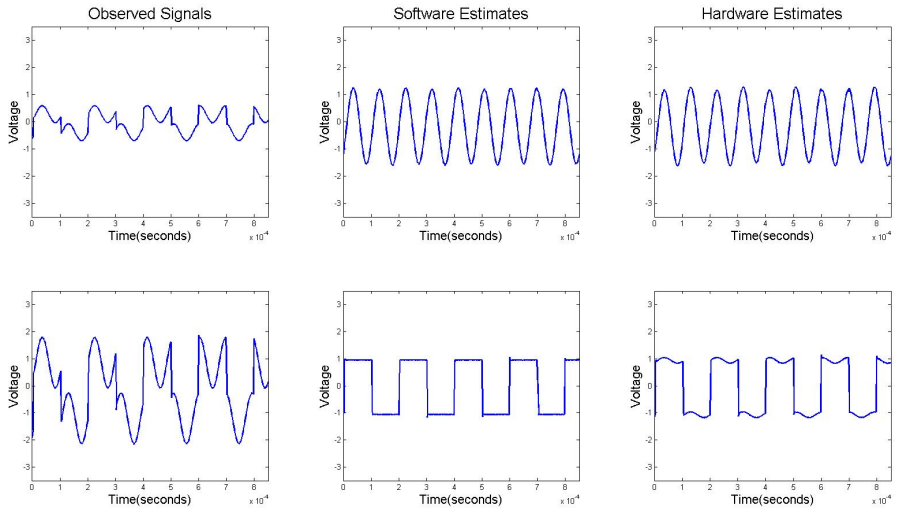

Figure B.5: JADE Hardware Verification #1

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.005509	0.00048556	0.0027251
Source 2	0.0055224	3.0565×10^{-08}	0.0054948

Table B.5: JADE Hardware Verification #1 MSE

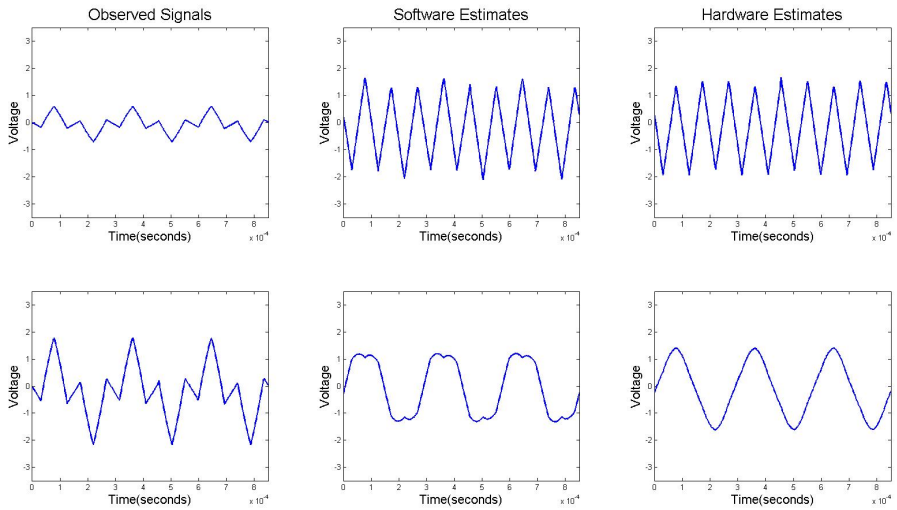

Figure B.6: JADE Hardware Verification #2

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.05921	0.024183	0.0077702
Source 2	0.05929	0.032557	0.0040586

Table B.6: JADE Hardware Verification #2 MSE

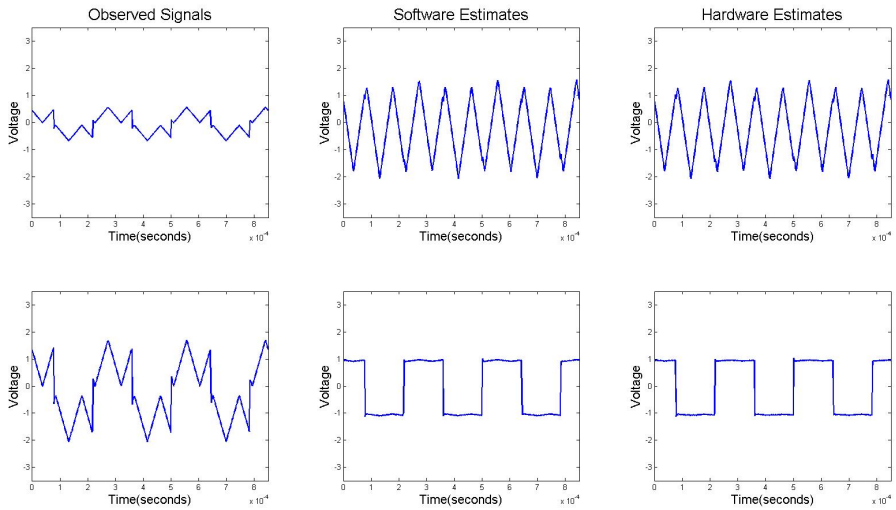

Figure B.7: JADE Hardware Verification #3

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	9.15×10^{-05}	0.016062	0.0003469
Source 2	9.1629×10^{-05}	0.018567	8.1973×10^{-05}

Table B.7: JADE Hardware Verification #3 MSE

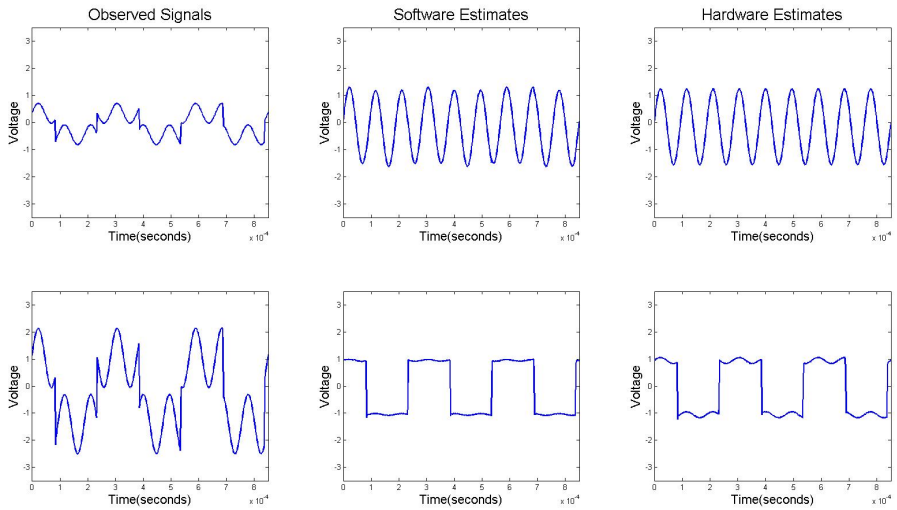

Figure B.8: JADE Hardware Verification #4

	Software vs Hardware MSE	Software MSE	Hardware MSE
Source 1	0.0031297	0.0031885	2.9628×10^{-07}
Source 2	0.0031253	0.00052936	0.0062181

Table B.8: JADE Hardware Verification #4 MSE

APPENDIX C

1 Hardware Block Diagrams

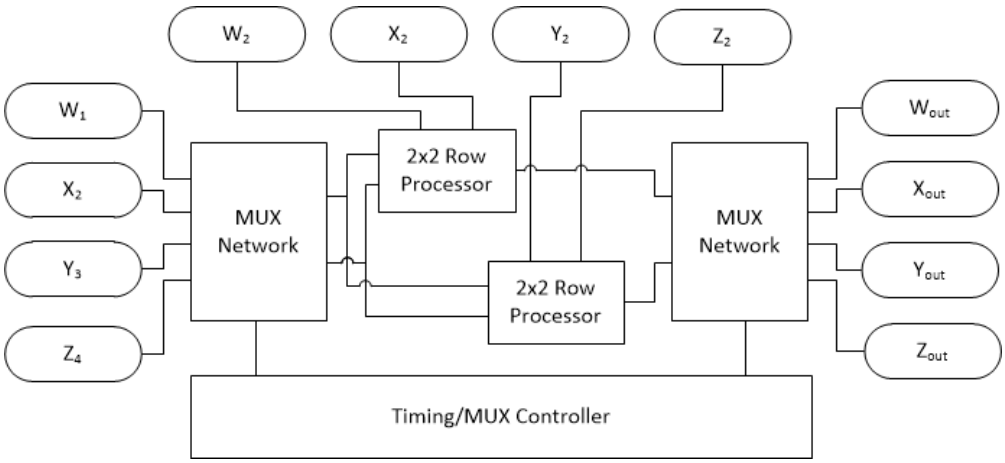

Figure C.1: A 2×2 matrix multiplier

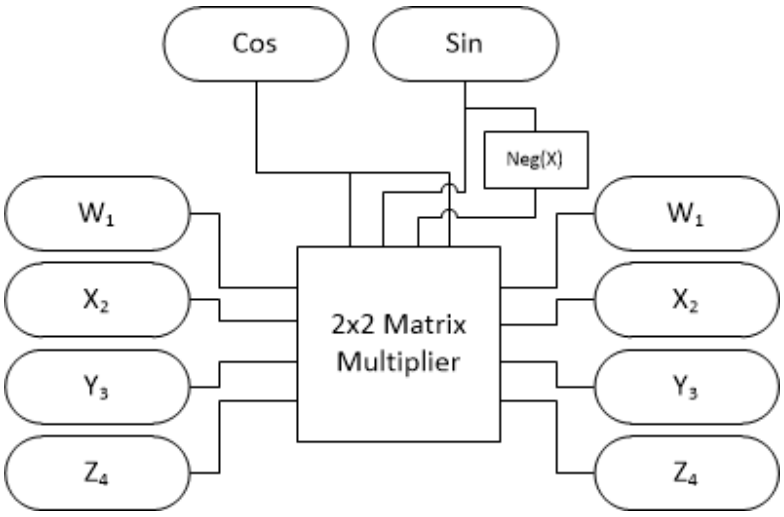

Figure C.2: A 2×2 matrix rotation/transformation module

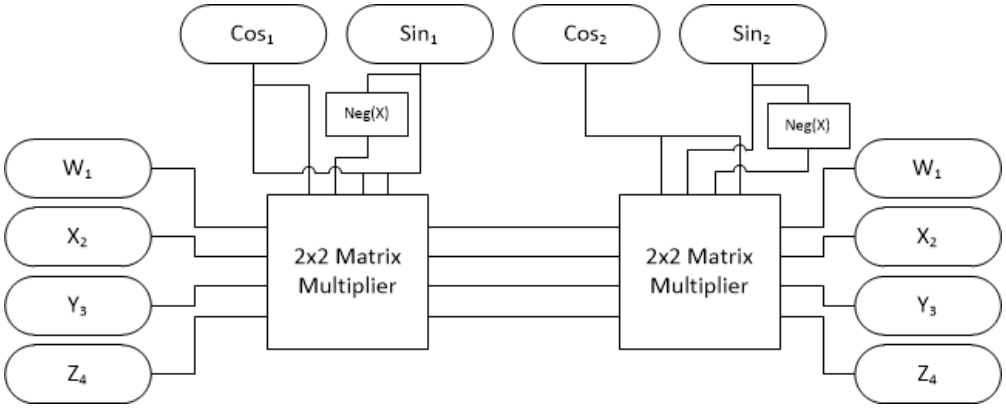

Figure C.3: A 2×2 double matrix rotation module $(S_1 \times A \times S_2)$

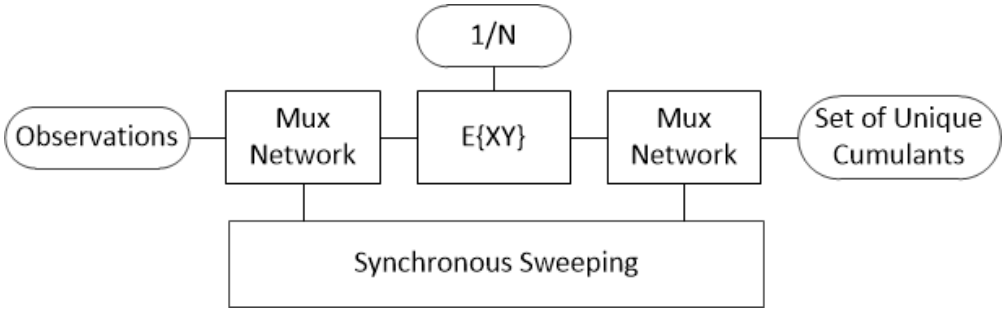

Figure C.4: Second order cumlants calculator.

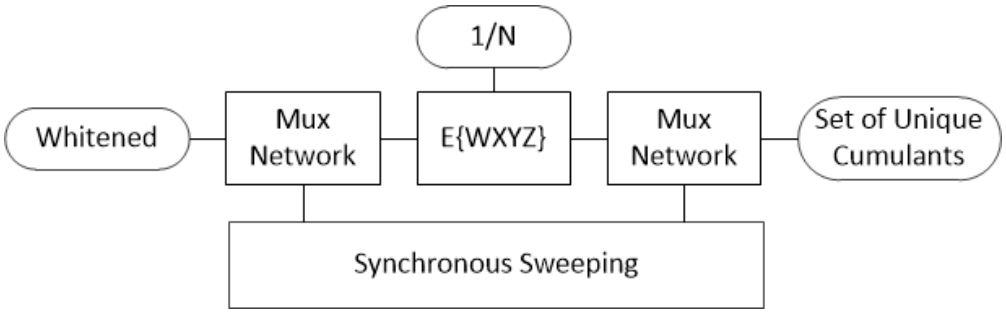

Figure C.5: Fourth order cumlants calculator.

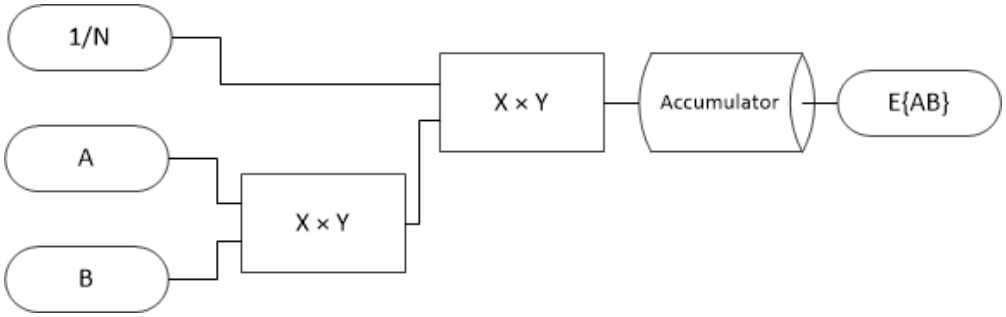

Figure C.6: Calculated expected value of two signals. Used in the cumulant calculations

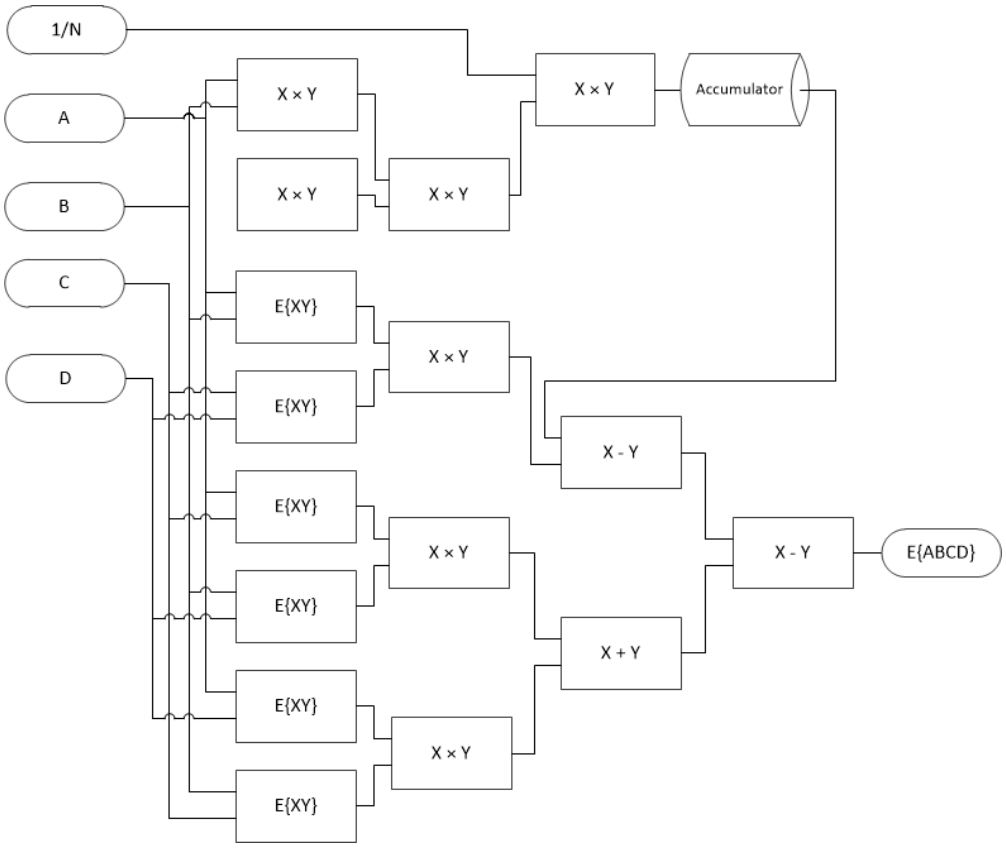

Figure C.7: Calculated expected value of four signals. Used in calculating Fourth order cumulants

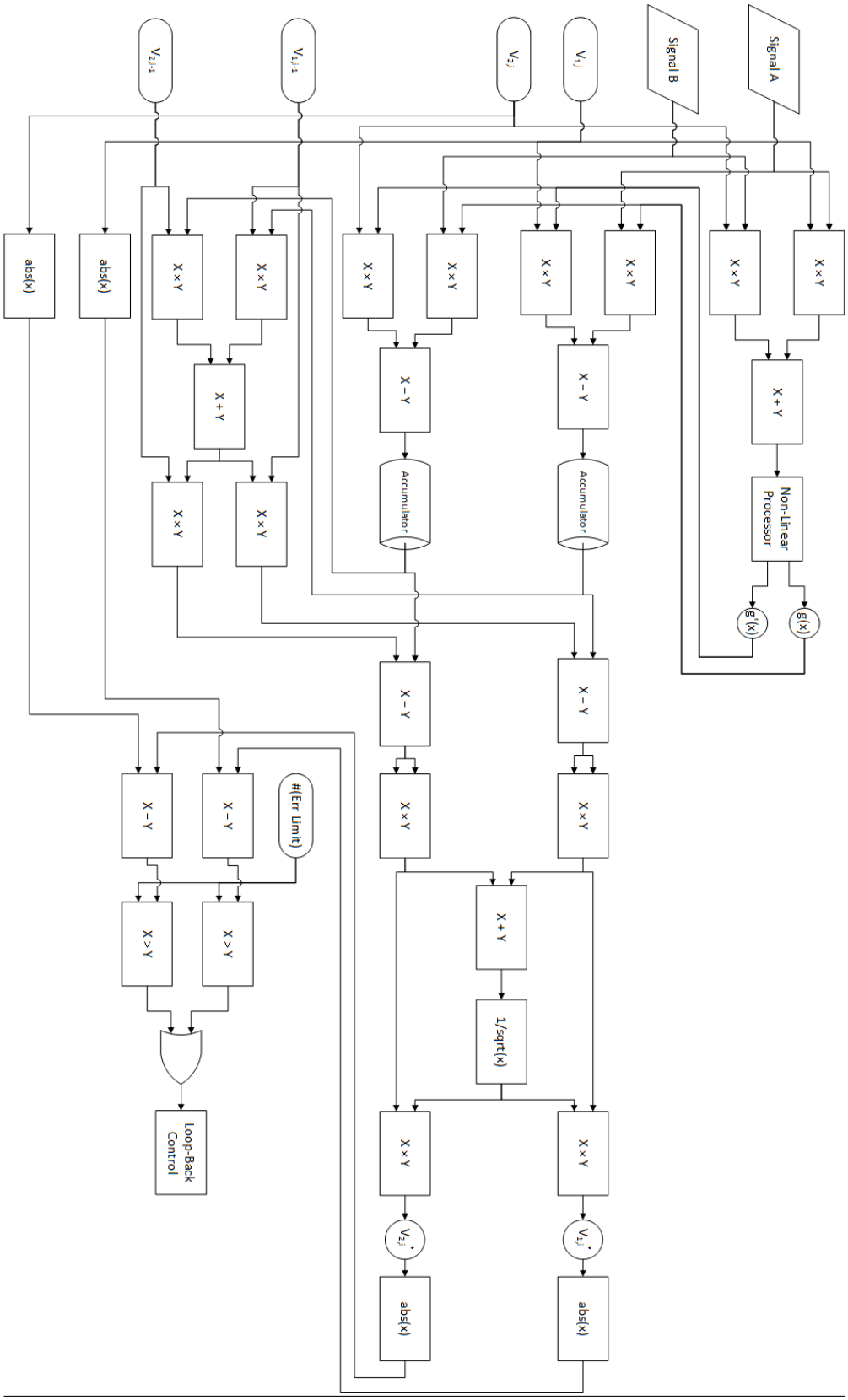

Figure C.8: FastICA Algorithm Block Diagram

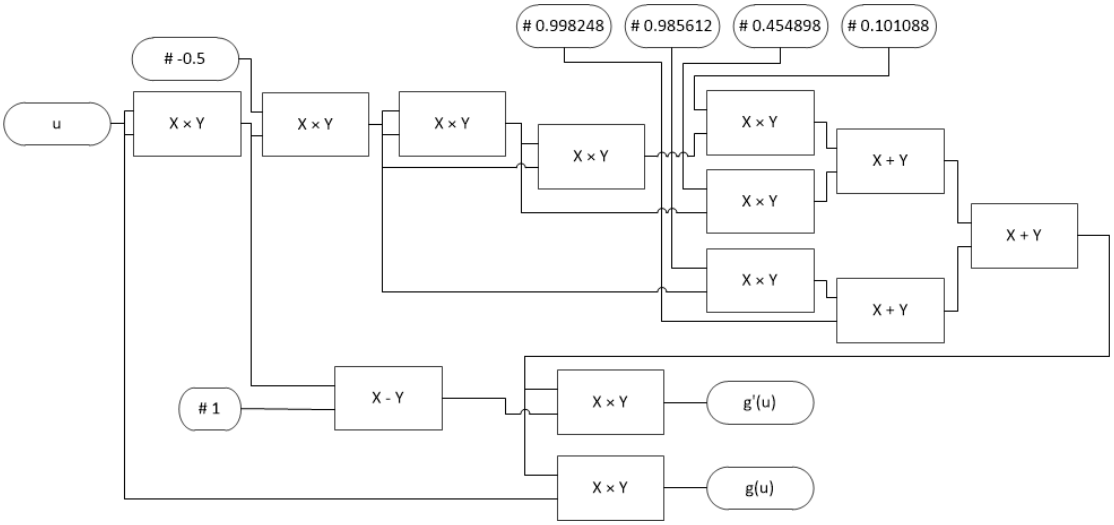

Figure C.9: Non-Linear function calculator for FastICA

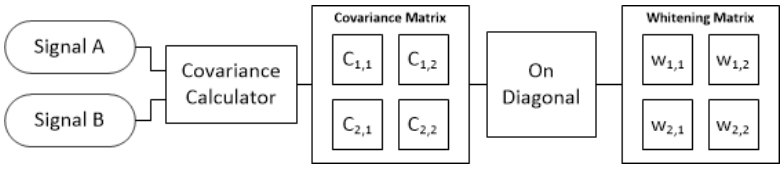

Figure C.10: Whitener for 2-signal

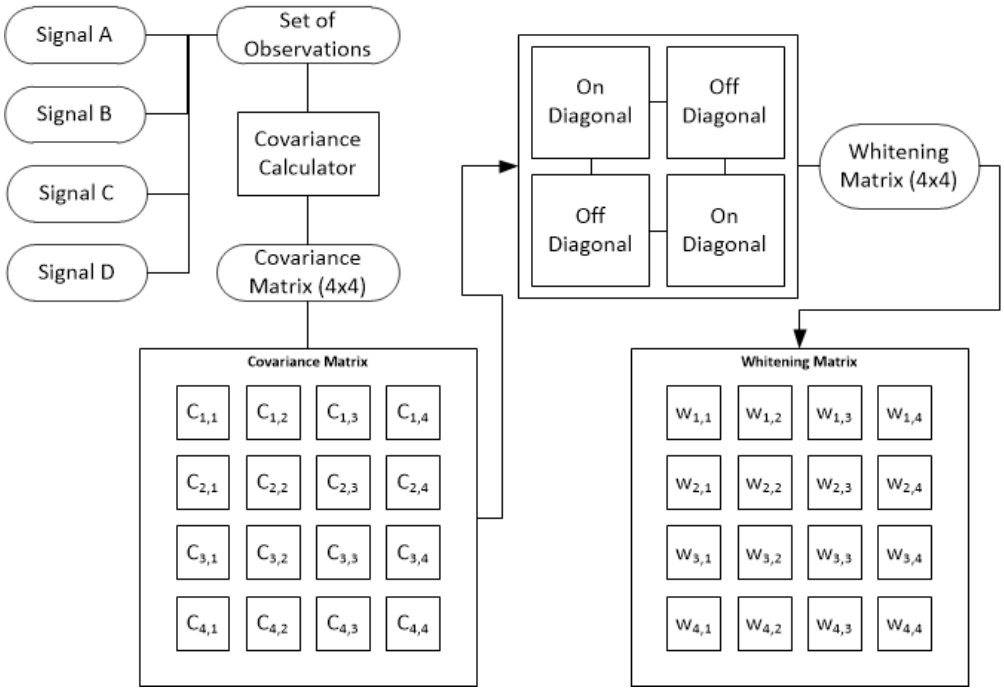

Figure C.11: Whitener for 4-signal case

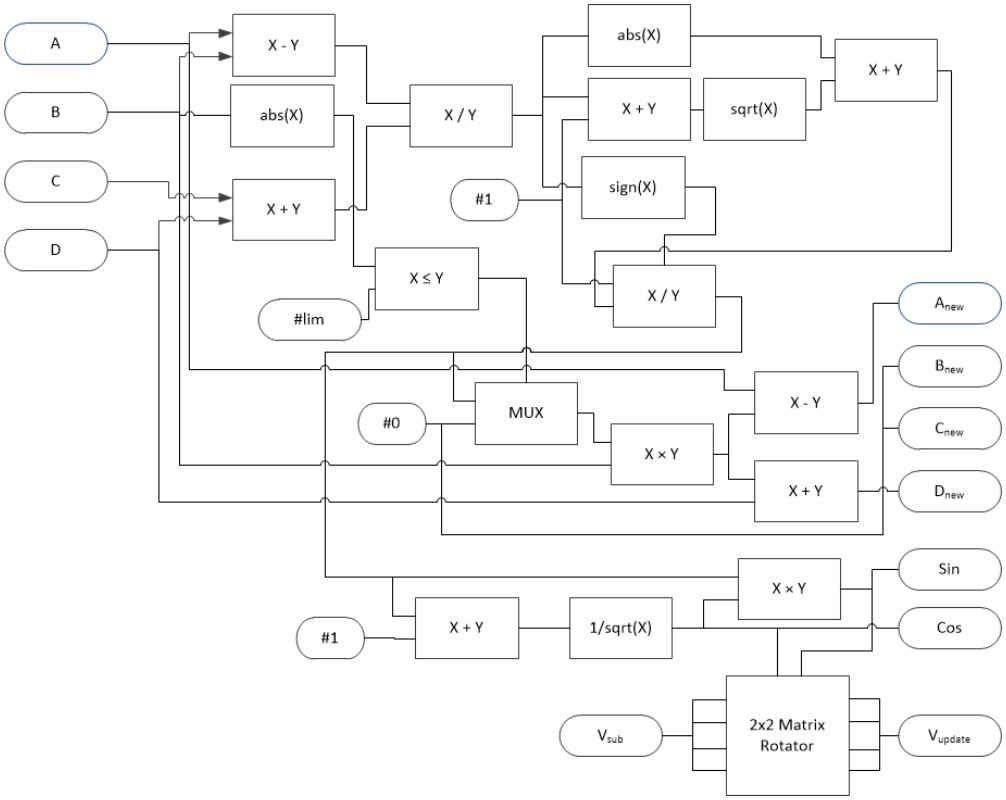

Figure C.12: On-Diagonal processor for systolic array Whitener

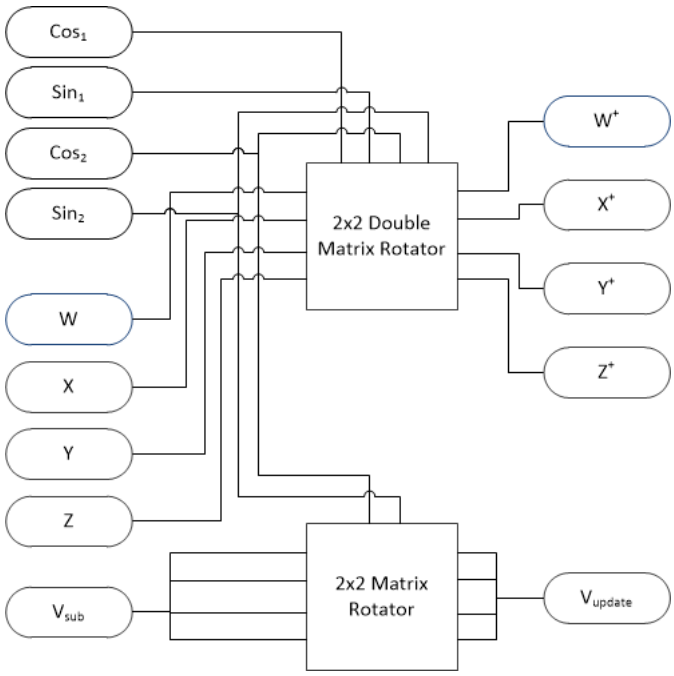

Figure C.13: Off-Diagonal processor for systolic array Whitener

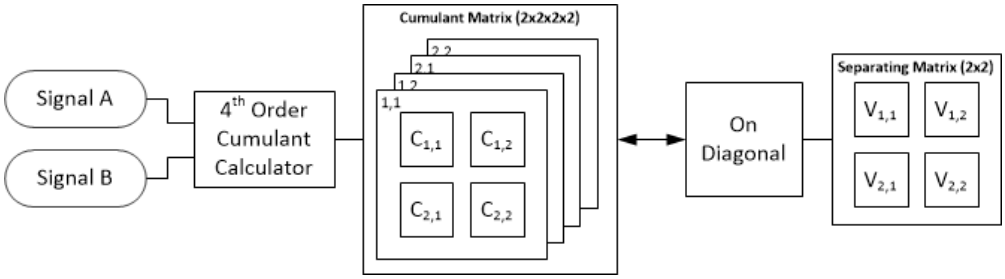

Figure C.14: 2-Signal JADE algorithm block diagram.

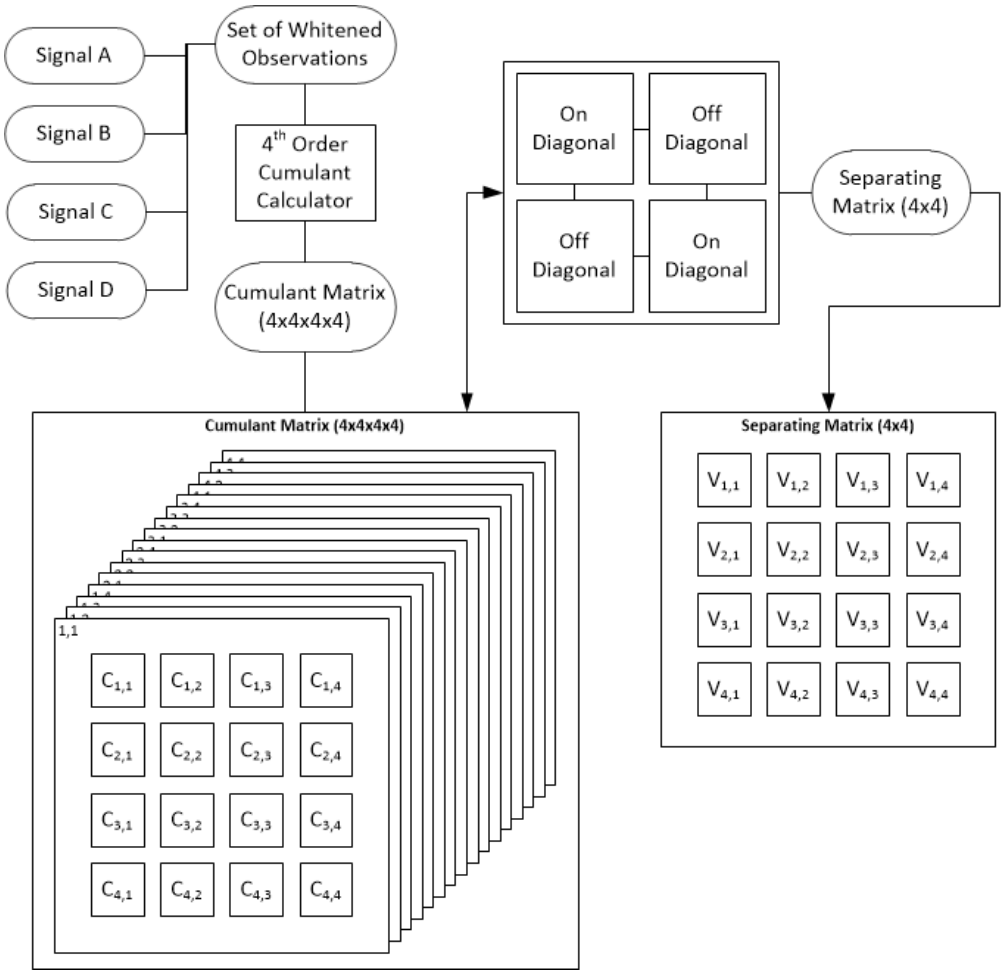

Figure C.15: 4-Signal JADE algorithm block diagram.

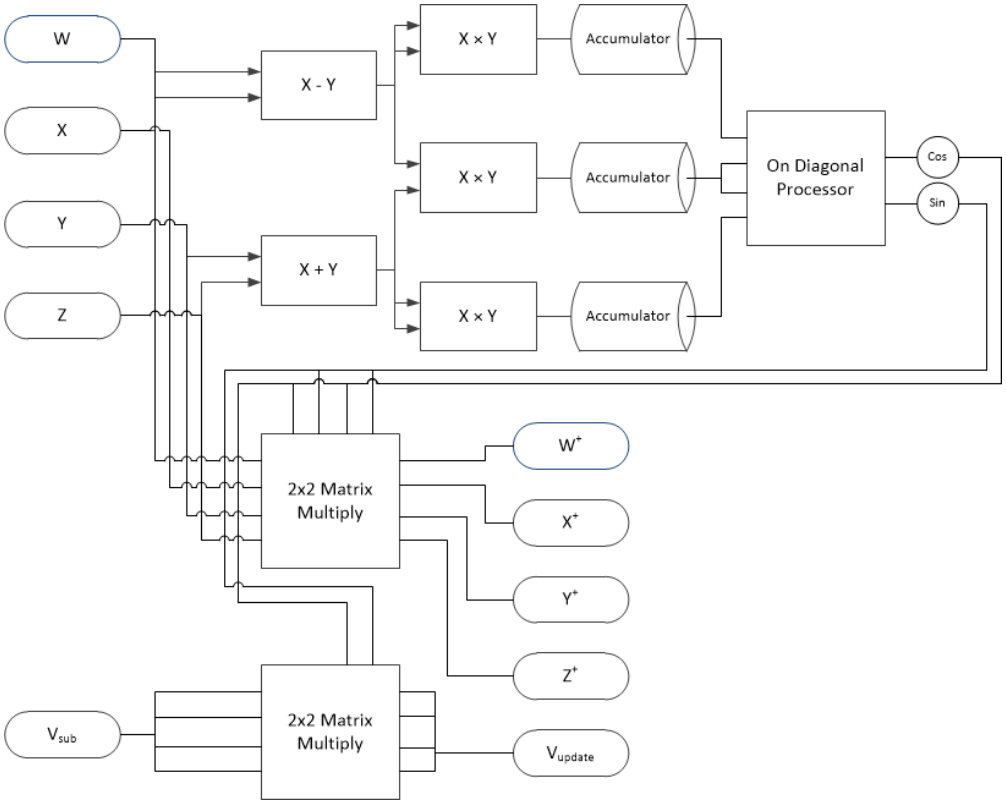

Figure C.16: On-Diagonal processor for systolic JADE algorithm.

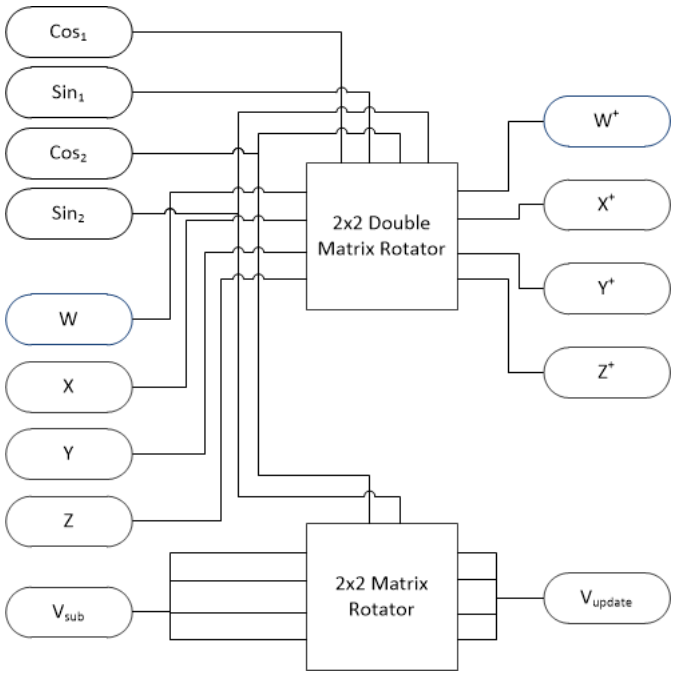

Figure C.17: Off-Diagonal processor for systolic JADE algorithm.

2 Systolic Array Matrix Multipliers

The following several diagrams depict the systolic array multiplier through the first time iterations.

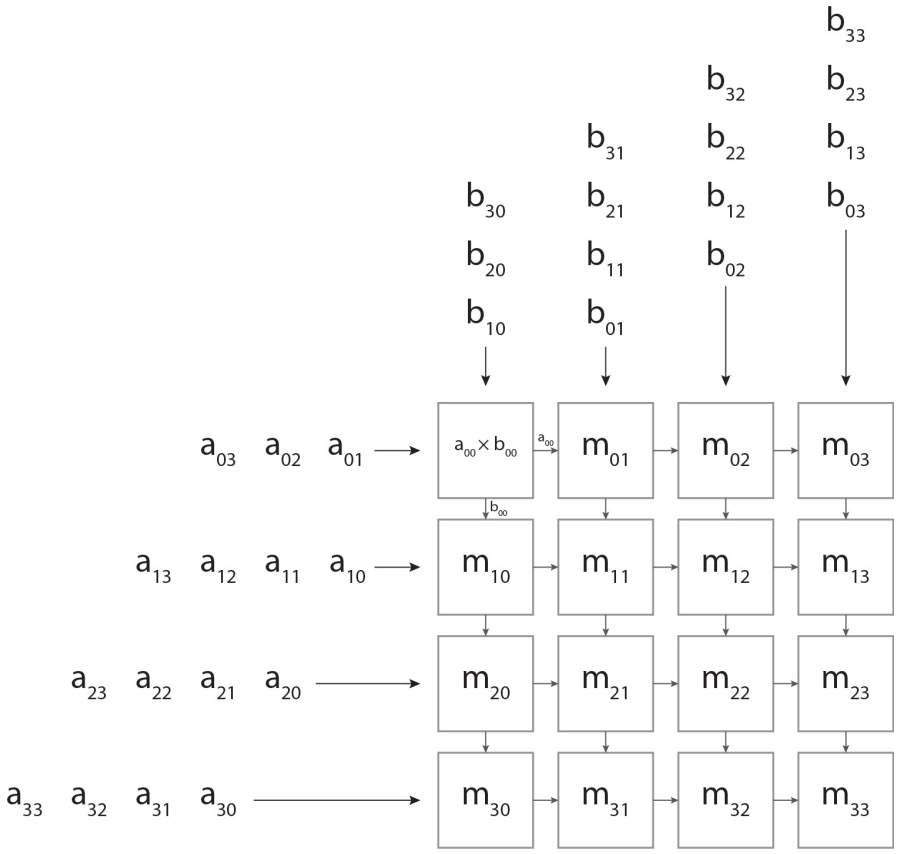

Figure C.18: Systolic Array Multiplier ($time = 1$)

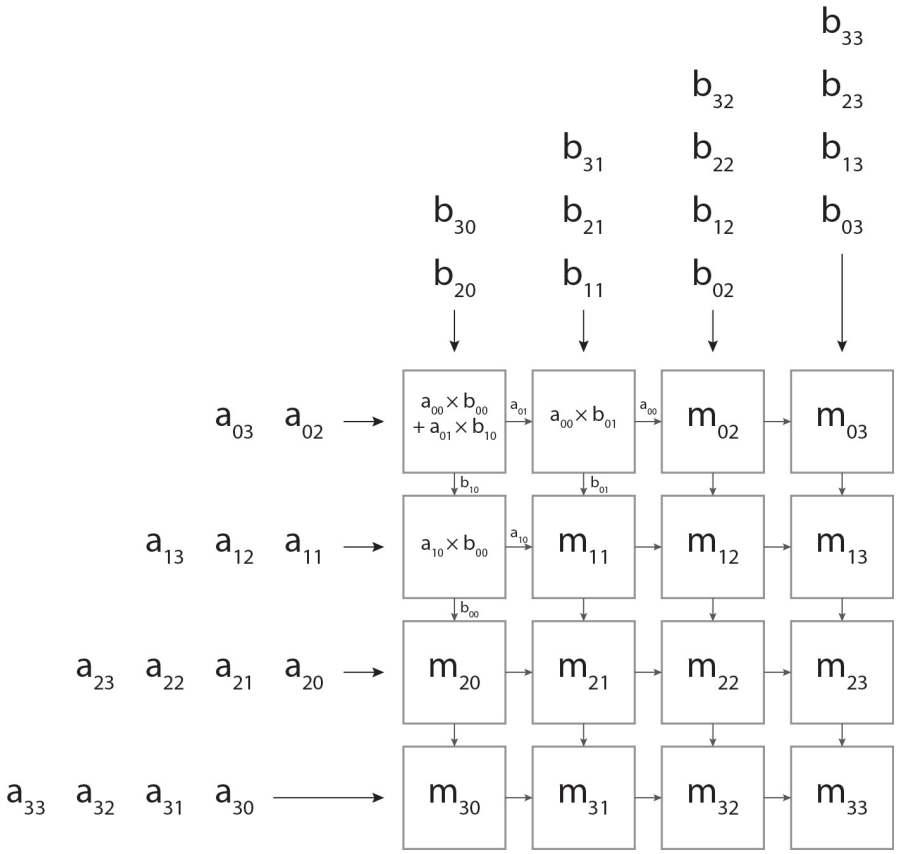

Figure C.19: Systolic Array Multiplier ($time = 2$)

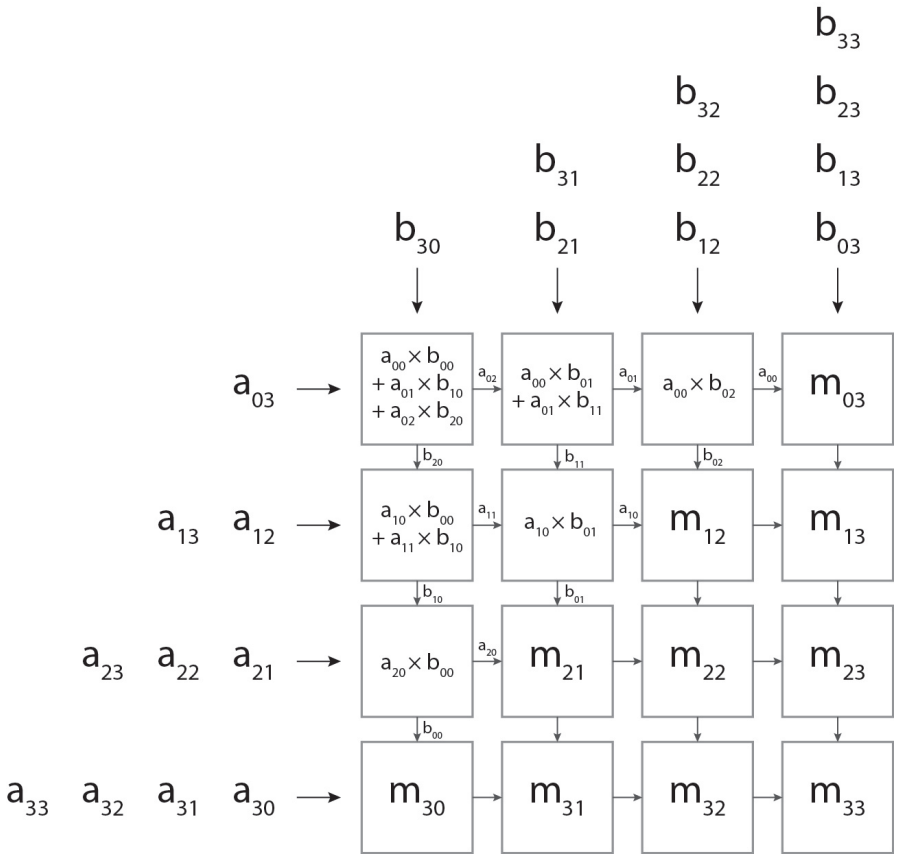

Figure C.20: Systolic Array Multiplier ($time = 3$)

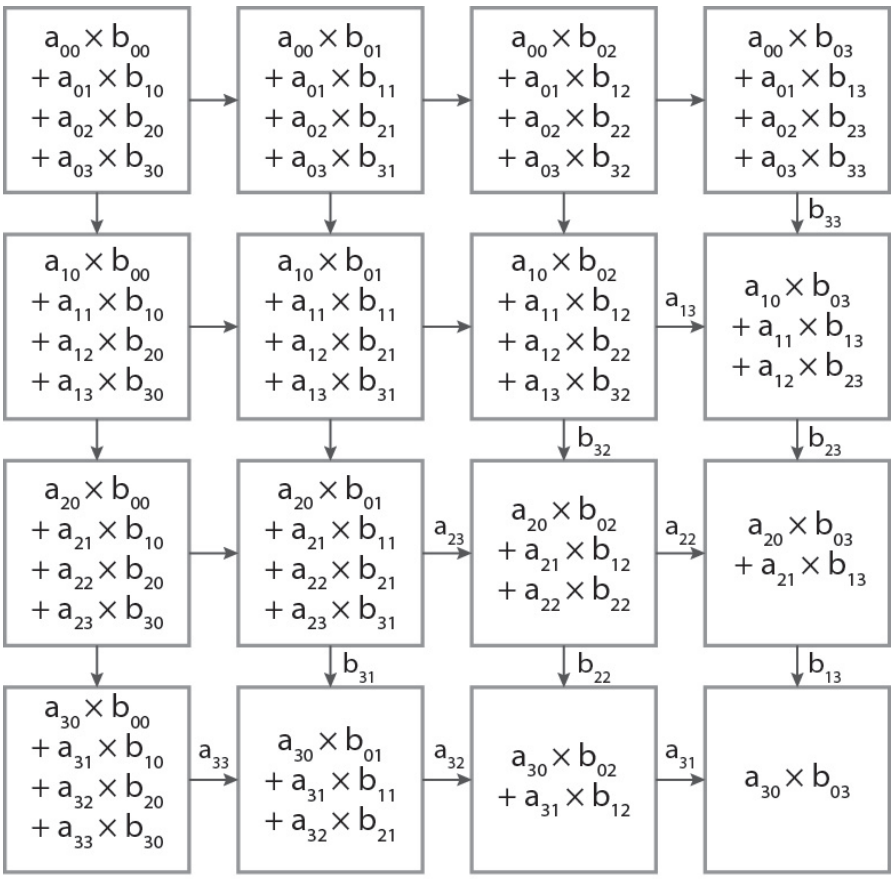

Figure C.21: Systolic Array Multiplier ($time = 7$)

APPENDIX D

1 Source Code for Transformer

```
----------------------------------------------------------------
-- Company: LeTourneau University Electrical and Computer
--    Engineering Department
-- Engineer: Timothy S. Hong
-- Target VHDL Version: IEEE STD 1076-2008
--
-- Create Date: Jul 14, 2015
-- Design Name: Transformer
-- Project Name: Support Library
-- Target Devices: Artix 7
-- Module Description: Performs 2x2 x 2xn matrix
--    multiplication
--
-- Version: 2.0 - Added components, entities, and connected
--    them all
-- Version: 1.0 - File Created
--
----------------------------------------------------------------

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

library xil_defaultlib;
use xil_defaultlib.custom.all;

entity transformer is
    port(
        aclk : in std_logic; -- System Clock (or Virtual)
        inA, inB : in ctrl_in; -- our input data streams
        w, x, y, z : in ctrl_in; -- input matrix weights
        outA, outB : out ctrl_out -- output data streams
        );
end transformer;

architecture action of transformer is

signal v_wa, v_xb, v_ya, v_zb : ctrl_out;

COMPONENT multiplier
  PORT (
    aclk : IN STD_LOGIC;
    s_axis_a_tvalid : IN STD_LOGIC;
```

```
    s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_b_tvalid : IN STD_LOGIC;
    s_axis_b_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_result_tvalid : OUT STD_LOGIC;
    m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;

COMPONENT adder
  PORT (
    aclk : IN STD_LOGIC;
    s_axis_a_tvalid : IN STD_LOGIC;
    s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_b_tvalid : IN STD_LOGIC;
    s_axis_b_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_result_tvalid : OUT STD_LOGIC;
    m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;

begin

wa : multiplier
  PORT MAP (
    aclk => aclk,
    s_axis_a_tvalid => w.tvalid,
    s_axis_a_tdata => w.tdata,
    s_axis_b_tvalid => inA.tvalid,
    s_axis_b_tdata => inA.tdata,
    m_axis_result_tvalid => v_wa.tvalid,
    m_axis_result_tdata => v_wa.tdata
  );

xb : multiplier
  PORT MAP (
    aclk => aclk,
    s_axis_a_tvalid => x.tvalid,
    s_axis_a_tdata => x.tdata,
    s_axis_b_tvalid => inB.tvalid,
    s_axis_b_tdata => inB.tdata,
    m_axis_result_tvalid => v_xb.tvalid,
    m_axis_result_tdata => v_xb.tdata
  );
```

```
ya : multiplier
  PORT MAP (
    aclk => aclk,
    s_axis_a_tvalid => y.tvalid,
    s_axis_a_tdata => y.tdata,
    s_axis_b_tvalid => inA.tvalid,
    s_axis_b_tdata => inA.tdata,
    m_axis_result_tvalid => v_ya.tvalid,
    m_axis_result_tdata => v_ya.tdata
  );

zb : multiplier
  PORT MAP (
    aclk => aclk,
    s_axis_a_tvalid => z.tvalid,
    s_axis_a_tdata => z.tdata,
    s_axis_b_tvalid => inB.tvalid,
    s_axis_b_tdata => inB.tdata,
    m_axis_result_tvalid => v_zb.tvalid,
    m_axis_result_tdata => v_zb.tdata
  );

addA : adder
  PORT MAP (
    aclk => aclk,
    s_axis_a_tvalid => v_wa.tvalid,
    s_axis_a_tdata => v_wa.tdata,
    s_axis_b_tvalid => v_xb.tvalid,
    s_axis_b_tdata => v_xb.tdata,
    m_axis_result_tvalid => outA.tvalid,
    m_axis_result_tdata => outA.tdata
  );

addB : adder
  PORT MAP (
    aclk => aclk,
    s_axis_a_tvalid => v_ya.tvalid,
    s_axis_a_tdata => v_ya.tdata,
    s_axis_b_tvalid => v_zb.tvalid,
    s_axis_b_tdata => v_zb.tdata,
    m_axis_result_tvalid => outB.tvalid,
    m_axis_result_tdata => outB.tdata
```

```
  );
end action;
```

2 Source Code for ADC

```
----------------------------------------------------------------------
-- Company: LeTourneau University Electrical and Computer
--    Engineering Department
-- Engineer: Timothy S. Hong
-- Target VHDL Version: IEEE STD 1076-2008
--
-- Create Date: Jun 16, 2015
-- Design Name: ADC
-- Project Name: Support Library
-- Target Devices: Artix 7
-- Module Description: Receives input from PMOD 12-bit ADC
--    and outputs 12-bit vectors
--
-- Revision: 1.1 - Used old module and updated the port types
--    to clean up overall code
-- Revision: 1.0 - File Created and basic functionality
--
----------------------------------------------------------------------

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

library xil_defaultlib;
use xil_defaultlib.custom.all;

entity adc is
    port(clk : in std_logic; -- System Clock (or Virtual)
       output : out DATAPAIR; -- Output to parent module
       data : in ADC_DATA; -- Incoming data from peripheral
       control : out ADC_CONTROL; -- Control signals for
                                  --   peripheral

       busy : out std_logic; -- Module Status Bit [aH]
       run : in std_logic -- Module Control Bit [aL]
        );
end adc;

architecture pull of adc is
    signal activate : std_logic; -- Internal Control Signal for
                                 --  Persistent Operation [aL]
    signal outBuff, outTemp : DATAPAIR; -- for latching output
begin
```

```
control.sck <= clk; -- Driven by System Clock
busy <= not activate; -- Status is driven by the internal signal
                      --  'activate'

outTemp <= outBuff when (activate = '1') else outTemp; -- latch
output <= outTemp;

sample : process(clk, run, activate)
    variable counter: integer range -1 to 16 := -1;
begin
    if rising_edge(clk) then -- run on rising edge
        if activate = '0' then
            counter := counter + 1;
            control.cs <= '0'; -- Activate the ADC peripheral

            if(counter < 4) then
                -- don't do anything
            elsif(counter < 16) then
                outBuff.A(15-counter) <= data.A; -- Clock in
                                                 --  data bit
                outBuff.B(15-counter) <= data.B; -- Clock in
                                                 --  data bit
            elsif(counter = 16) then
                counter := -1; -- Reset Counter

                activate <= '1'; -- Internal module signal to
                                 --  return to poll state
            end if;
        else -- if activate = '1'
            control.cs <= '1';

            if run = '0' then -- Check for trigger run
                activate <= '0'; -- run ADC
                counter := -1;
            else
                activate <= '1'; -- persist disable ADC
            end if;
        end if;
    end if; -- End if rising_edge(clk)
end process; -- END sample

end pull;
```

3 Source Code for DAC

```
----------------------------------------------------------------
-- Company: LeTourneau University Electrical and Computer
--    Engineering Department
-- Engineer: Timothy S. Hong
-- Target VHDL Version: IEEE STD 1076-2008
--
-- Create Date: Jun 16, 2015
-- Design Name: DAC
-- Project Name: Support Library
-- Target Devices: Artix 7
-- Module Description: Receives 12-bit vectors and writes them
--    to DAC
--
-- Revision: 1.1 - Used old module and updated the port types
--    to clean up overall code
-- Revision: 1.0 - File Created
--
----------------------------------------------------------------

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

library xil_defaultlib;
use xil_defaultlib.custom.all;

entity dac is
    port(clk : in std_logic; -- System Clock (or Virtual)
       input : in DATAPAIR; -- Incoming data fram parent
                            --  module
       data : out DAC_DATA; -- Output data to peripheral
       control : out DAC_CONTROL; -- Control signals for
                                  --  peripheral

       busy : out std_logic; -- Module Status Bit [aH]
       run : in std_logic -- Module Control Bit [aL]
        );
end dac;

architecture push of dac is
    signal activate : std_logic; -- Internal Control Signal
                                 --  for Persistent
                                 --  Operation [aL]
```

```
    signal inBuff : DATAPAIR;
begin

control.sck <= clk; -- Drive with system clock

-- Should work because activate is switched to '1' at least
--  one clock cycle every run
inBuff.A <= input.A when(activate = '1')
        else inBuff.A; -- Latch inputs
inBuff.B <= input.B when(activate = '1')
        else inBuff.B; -- (this ensures the data doesn't
                       --  change while running)

busy <= not activate; -- Status is driven by the internal
                      --  signal 'activate'

output : process(clk, run, activate)
    variable counter : integer range -1 to 16 := -1;
begin
    if rising_edge(clk) then
        if activate = '0' then
            counter := counter + 1;

            if(counter = 0) then
                control.sync <= '0'; -- Trigger DAC output
                                     --  operation
            elsif(counter < 4) then -- Zero Bits + Mode
                                    --  Config Bits
                data.A <= '0';
                data.B <= '0';
            elsif(counter < 16) then
                data.A <= inBuff.A(15-counter);
                data.B <= inBuff.B(15-counter);
            elsif(counter = 16) then
                counter := -1; -- reset counter

                activate <= '1'; -- Internal module signal
                                 --  to return to poll state
            end if;
        else -- if activate = '1'
            control.sync <= '1';

            if run = '0' then
```

```
                activate <= '0';
                counter := -1;
            else
                activate <= '1'; -- Persistent Disable
            end if;
        end if;
    end if; -- End if rising_edge(clk)
end process; -- END output

end push;
```

BIBLIOGRAPHY

[1] A. Hyvärinen, "Fast and robust fixed-point algorithms for independent component analysis," *IEEE Transactions on Neural Networks*, vol. 10, pp. 626–634, 1999.

[2] J.-F. Cardoso and A. Souloumiac, "Blind beamforming for non-Gaussian signals," *IEE Proceedings F (Radar and Signal Processing)*, vol. 140, no. 6, pp. 362–370, 1993. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/ip-f-2.1993.0054

[3] A. J. Bell and T. J. Sejnowski, "An Information-Maximization Approach to Blind Separation and Blind Deconvolution," *Neural Computation*, vol. 7, no. 6, pp. 1129–1159, 1995. [Online]. Available: http://www.mitpressjournals.org/doi/abs/10.1162/neco.1995.7.6.1129{#}.VbJaKPmqqko

[4] S. K. Behera, "FAST ICA FOR BLIND SOURCE SEPARATION AND ITS IMPLEMENTATION," Ph.D. dissertation, National Institute of Technology in Rourkela, 2009.

[5] A.-l. Taha, "FPGA Implementation of Blind Source Separation using FastICA," Ph.D. dissertation, University of Windsor, 2010.

[6] K. Kokkinakis and P. C. Loizou, "Multi-microphone adaptive noise reduction strategies for coordinated stimulation in bilateral cochlear implant devices." *The Journal of the Acoustical Society of America*, vol. 127, no. 5, pp. 3136–44, 2010. [Online]. Available:

http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2882668{&}tool=pmcentrez{&}rendertype=abstract

[7] A. Hyvärinen and E. Oja, "Independent component analysis: algorithms and applications," *Neural networks*, vol. 13, no. 4, pp. 411–430, 2000.

[8] J. Herault and C. Jutten, "Space or time adaptive signal processing by neural network models," in *Neural networks for computing*, vol. 151, no. 1. AIP Publishing, 1986, pp. 206–211.

[9] J.-F. Cardoso and P. Comon, "Tensor-based independent component analysis," *Proc. EUSIPCO*, pp. 673–676, 1990.

[10] A. Cichocki and L. Moszczynski, "New learning algorithm for blind separation of sources," *Electronics Letters*, vol. 28, no. 21, pp. 1986–1987, Oct 1992.

[11] J. Karhunen and J. Joutsensalo, "Learning of robust principal component subspace," in *Neural Networks, 1993. IJCNN '93-Nagoya. Proceedings of 1993 International Joint Conference on*, vol. 3, Oct 1993, pp. 2409–2412 vol.3.

[12] J. Lacoume and M. Gaeta, "The general source separation problem," in *Spectrum Estimation and Modeling, 1990., Fifth ASSP Workshop on*, Oct 1990, pp. 154–158.

[13] S. Becker and G. E. Hinton, "Self-organizing neural network that discovers surfaces in random-dot stereograms." *Nature*, vol. 355, no. 6356, pp. 161–163, 1992.

[14] R. Linsker, "Local Synaptic Learning Rules Suffice to Maximize Mutual Information in a Linear Network," *Neural Computation*, vol. 4, no. 5, pp. 691–702, 1992.

[15] P. Comon, "Independent component analysis, A new concept?" *Signal Processing*, vol. 36, no. 3, pp. 287–314, 1994.

[16] B. A. Pearlmutter, L. C. Parra, and L. Jolla, "A Context-Sensitive Generalization of ICA," in *1996 International Conference on Neural Information Processing*, 1997, pp. 613–619.

[17] A. Kessy, A. Lewin, and K. Strimmer, "Optimal whitening and decorrelation," *Statistics*, p. 12, 2015. [Online]. Available: http://arxiv.org/abs/1512.00809

[18] R. P. Brent, F. T. Luk, and C. Van Loan, "Computation of the singular value decomposition using mesh-connected processors," Cornell University, Ithaca, Tech. Rep., 1982. [Online]. Available: http://www.ecommons.cornell.edu/handle/1813/6367

[19] R. P. Brent and F. T. Luk, "A Systolic Architechture for Almost Linear-Time Solution of the Symmetric Eigenvalue Problem," Cornell University, Ithaca, Tech. Rep., 1982. [Online]. Available: http://techreports.library.cornell.edu: 8081/Dienst/UI/1.0/Display/cul.cs/TR82-525

[20] *MC14008B: 4-Bit Full Adder*, ON Semiconductors, 7 2014, rev. 8.

[21] K. Barman, P. Dighe, and R. Rao, "Multiplication of matrices using systolic arrays," Dec. 30 2014, uS Patent 8,924,455. [Online]. Available: https://www.google.com/patents/US8924455

[22] A. Delorme, "Ica (independent component analysis) for dummies." [Online]. Available: https://sccn.uscsd.edu/~arno/indexica.html

[23] J. E. Rodríguez, "Applications of blind source separation to the magnetoencephalogram background activity in alzheimer's disease," Ph.D. dissertation, Universidad de Valladolid, 2010.

[24] J. Morton and L. Lim, "Principal cumulant component analysis," *preprint*, pp. 1–10, 2009. [Online]. Available: http://www.jasonmorton.com/morton/publications/pcca.pdf

[25] H. Zhang, G. Wang, P. Cai, Z. Wu, and S. Ding, "A fast blind source separation algorithm based on the temporal structure of signals," *Neurocomputing*, vol. 139, pp. 261–271, sep 2014. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0925231214004354

[26] C. Li and G. Liao, "A reference-based blind source extraction algorithm," *Neural Computing and Applications*, vol. 19, no. 2, pp. 299–303, sep 2009. [Online]. Available: http://link.springer.com/10.1007/s00521-009-0303-x

[27] J. Karhunen and P. Pajunen, "Blind source separation using least-squares type adaptive algorithms," *Acoustics, Speech, and Signal . . .*, no. 2, 1997. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=595514

[28] D. Langlois, S. Chartier, and D. Gosselin, "An Introduction to Independent Component Analysis : InfoMax and FastICA Algorithms," *Tutorials in Quantitative Methods for Psychology*, vol. 6, no. 1, pp. 31–38, 2010. [Online]. Available: http://www.tqmp.org/Content/vol06-1/p031/p031.pdf

[29] T. W. Lee, M. Girolami, and T. J. Sejnowski, "Independent component analysis using an extended infomax algorithm for mixed subgaussian and supergaussian sources." *Neural computation*, vol. 11, pp. 417–441, 1999.

[30] A. S and a. Cichocki, "A New Learning Algorithm for Blind Signal Separation," *Advances*, p. 8, 1996.

[31] E. B. Davies, "Approximate Diagonalization," *SIAM J. Matrix Anal. Appl.*, vol. 29, no. 4, pp. 1051–1064, 2007.

[32] S. Dégerine and E. Kane, "A comparative study of approximate joint diagonalization algorithms for blind source separation in presence of

additive noise," *IEEE Transactions on Signal Processing*, vol. 55, no. 5, pp. 3022–3031, 2007.

[33] C. Févotte and F. J. Theis, *Orthonormal approximate joint block-diagonalization.* École nationale supérieure des télécommunications, 2007.

[34] G. Hori, "A new approach to joint diagonalization," *Proceedings of 2nd International Workshop on ICA and ...*, vol. 9, pp. 151–156, 2000. [Online]. Available: http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:No+Title{#}0http://users.ics.aalto.fi/ica2000/proceedings/0151.pdf

[35] X. Liu, J.-F. Cardoso, and R. B. Randall, "Very fast blind source separation by signal to noise ratio based stopping threshold for the SHIBBS/SJAD algorithm," *Mechanical Systems and Signal Processing*, vol. 24, no. 7, pp. 2096–2103, oct 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0888327010001007

[36] X. Liu and R. Randall, "A very fast large scale BSS algorithm by joint approximate diagonalization of simplified cumulant matrices," *Sciencepaper Online. http://www. paper. edu. cn/ ...*, no. 1, pp. 1–15, 2007. [Online]. Available: http://www.paper.edu.cn/en{_}releasepaper/downPaper/200712-326

[37] E. Moreau, "Comments on" CuBICA: Independent Component Analysis by Simultaneous third-and Fourth-Order Cumulant Diagonalization," *Signal Processing, IEEE Transactions on*, vol. 54, no. 12, pp. 4826–4828, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=4014386

[38] D. Pani, S. Argiolas, and L. Raffo, "A DSP algorithm and system for real-time fetal ECG extraction," *Computers in Cardiology*, vol. 35, pp. 1065–1068, 2008.

[39] D. Pham, "Joint approximate diagonalization of positive definite Hermitian matrices," *SIAM Journal on Matrix Analysis and Applications*, 2001. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/S089547980035689X

[40] F. Theis and Y. Inouye, "On the use of joint diagonalization in blind signal processing," *2006 IEEE International Symposium on Circuits and Systems*, no. 2, pp. 7–10, 2006.

[41] P. Tichavsky, "A fast approximate joint diagonalization algorithm using a criterion with a block diagonal weight matrix," *Acoustics, Speech and . . .*, pp. 3321–3324, 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=4518361

[42] A. Ziehe, "Blind source separation based on joint diagonalization of matrices with applications in biomedical signal processing," Ph.D. dissertation, Citeseer, 2005.

[43] A. Ziehe, P. Laskov, G. Nolte, and K.-R. MÃžller, "A fast algorithm for joint diagonalization with non-orthogonal transformations and its application to blind source separation," *Journal of Machine Learning Research*, vol. 5, no. Jul, pp. 777–800, 2004.

[44] J.-F. Cardoso, "Source separation using higher order moments," *Acoustics, Speech, and Signal Processing, 1989. . . .*, no. M, pp. 2109–2112, 1989. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=266878$\delimiter"026E30F$nhttp://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=266878

[45] ——, "Fourth-order cumulant structure forcing: application to blind array processing," in *Statistical Signal and Array Processing, 1992. Conference Proceedings., IEEE Sixth SP Workshop on.* IEEE, 1992, pp. 136–139.

[46] ——, "Blind signal separation: statistical principles," *Proceedings of the IEEE*, vol. 9, no. 10, pp. 1–16, 1998. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=720250

[47] ——, "High-order contrasts for independent component analysis," *Neural computation*, vol. 11, pp. 157–192, 1999. [Online]. Available: http://www.mitpressjournals.org/doi/abs/10.1162/089976699300016863

[48] ——, "Iterative techniques for blind source separation using only fourth-order cumulants," in *European Signal Processing Conference*, no. 33, 1992, pp. 1–4. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.1138$\delimiter"026E30F$nhttp://www.tsi.enst.fr/{~}cardoso/Papers.PDF/eusipco92.pdf

[49] ——, "Eigen-structure of the fourth-order cumulant tensor with application to the blind source separation problem," *International Conference on Acoustics, Speech, and Signal Processing*, 1990.

[50] ——, "Dependence, Correlation and Gaussianity in Independent Component Analysis," *Journal of Machine Learning Research*, vol. 4, pp. 1177–1203, 2003.

[51] ——, "Super-symmetric decomposition of the fourth-order cumulant tensor. Blind identification of more sources than sensors," *[Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing*, 1991.

[52] J.-F. Cardoso and A. Souloumiac, "Jacobi angles for simultaneous diagonalization," *SIAM journal on matrix analysis and applications*, pp.

7–9, 1996. [Online]. Available:
http://epubs.siam.org/doi/abs/10.1137/S0895479893259546

[53] D. Acharya and G. Panda, "A Review of Independent Component Analysis Techniques and their Applications," *IETE Technical Review*, vol. 25, no. 6, p. 320, 2008.

[54] L. E. A. Albera, "ICA-Based EEG denoising: a comparative analysis of fifteen methods," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 60, no. 3, pp. 407–418, 2012. [Online]. Available:
http://www.degruyter.com/view/j/bpasts.2012.60.issue-3/
v10175-012-0052-3/v10175-012-0052-3.xml

[55] E. Fykse, "Performance comparison of gpu, dsp and fpga implementations of image processing and computer vision algorithms in embedded systems," Ph.D. dissertation, Norges teknisk-naturvitenskapelige universitet, 2013.

[56] Y. Li, D. Powers, and J. Peach, "Comparison of blind source separation algorithms," *Advances in Neural Networks and Applications*, no. C, pp. 18–21, 2000. [Online]. Available:
http://www.infoeng.flinders.edu.au/papers/20000001.pdf

[57] V. Matic, W. Deburchgraeve, and S. Van Huffel, "Comparison of ICA Algorithms for ECG Artifact Removal from EEG Signals," *IEEE-EMBS Benelux Chapter Symposium*, pp. 137–140, 2009. [Online]. Available:
http://www.embs-chapter.be/

[58] G. Raja, P. K. Chaitanya, and R. Malmathanraj, "Performance Analysis of Independent Component Analysis Algorithms for Multi-user Detection of DS-CDMA." *International Journal of Computer Applications*, vol. 39, no. 11, pp. 34–37, 2012. [Online]. Available: http:
//search.ebscohost.com/login.aspx?direct=true{&}profile=ehost{&}scope=
site{&}authtype=crawler{&}jrnl=09758887{&}AN=74273209{&}h=

S65obqUn3szCroUi8HGqy7mcHpLTztwQRYaw3aI4HZmFeL8aByuFZzmOj/VrPt3RrUvbpp2w0buL/nBH9GbDOQ=={&}crl=c

[59] H. Rutishauser, "The Jacobi method for real symmetric matrices," *Numerische Mathematik*, vol. 9, no. 1, pp. 1–10, 1966.

[60] Y. Liu, C.-S. Bouganis, P. Y. Cheung, P. H. Leong, and S. J. Motley, "Hardware efficient architectures for eigenvalue computation," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1. IEEE, 2006, pp. 1–6.

[61] I. Bravo, P. Jiménez, M. Mazo, J. L. Lázaro, and A. Gardel, "Implementation in FPGAS of Jacobi method to solve the eigenvalue and eigenvector problem," *Proceedings - 2006 International Conference on Field Programmable Logic and Applications, FPL*, pp. 729–732, 2006.

[62] A. Hyvärinen and E. Oja, "Independent component analysis: algorithms and applications." *Neural networks : the official journal of the International Neural Network Society*, vol. 13, no. 1, pp. 411–30, 2000. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/10946390

[63] S. Becker, "JPMAX: Learning to recognize moving objects as a model fitting problem," *Advances in NIPS 7*, pp. 933–940, 1995.

[64] S. Becker and R. Zemel, "Unsupervised Learning with global objective functions," *The handbook of brain theory and neural . . .*, no. 905, pp. 1–17, 1995. [Online]. Available: http://www.researchgate.net/publication/2602101{_}Unsupervised{_}Learning{_}With{_}Global{_}Objective{_}Functions/file/d912f50fda919901f0.pdf