

---

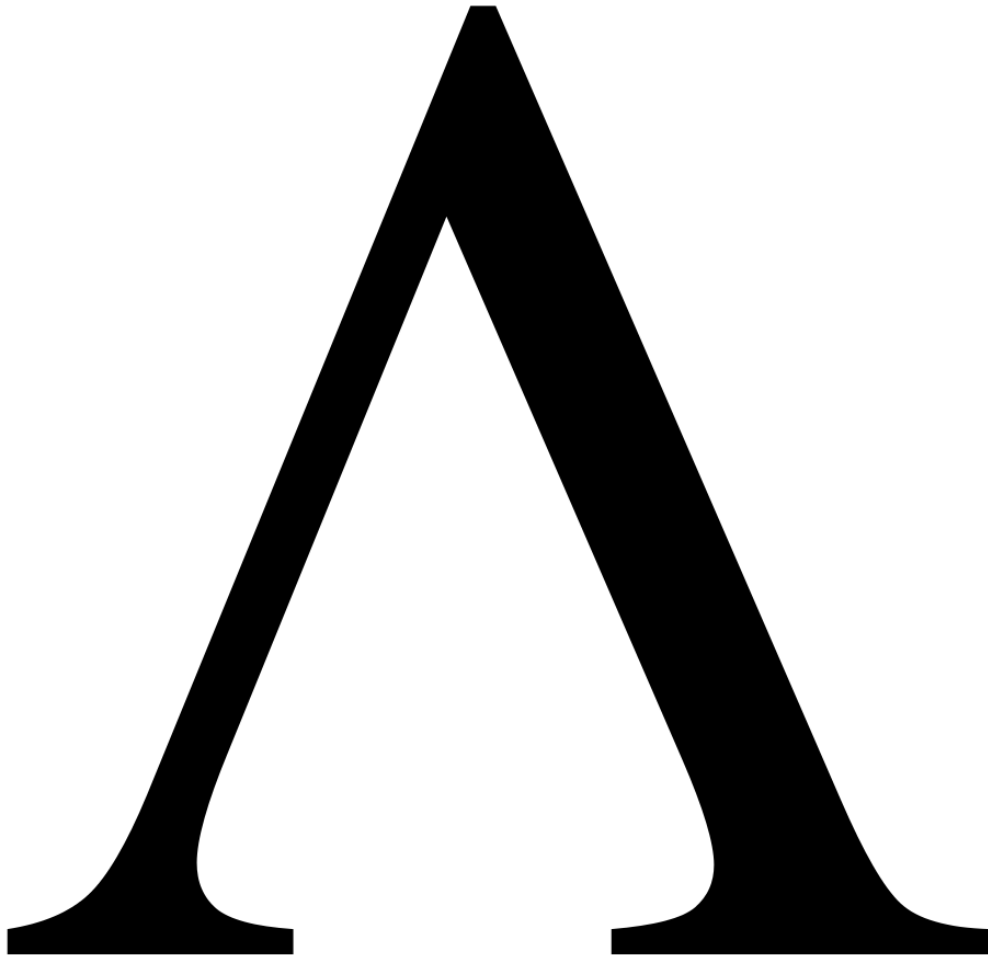
---

# Feasibility Study on a Custom Architecture for the Jacobi Eigenvalue Algorithm

- P8 -

---

---



Project Report  
Group 870

Aalborg University  
Department of Electronic Systems  
Fredrik Bajers Vej 7B  
DK-9220 Aalborg

Copyright © Aalborg University 2020

This report was written in LaTeX and has been shared online in-between the group members using Overleaf licensed to all students at Aalborg University.



# AALBORG UNIVERSITY

## STUDENT REPORT

### Department of Electronic Systems

Fredrik Bajers Vej 7

DK-9220 Aalborg Ø

<http://es.aau.dk>

**Title:**

Feasibility Study on a Custom Architecture for the Jacobi Eigenvalue Algorithm

**Theme:**

Reconfigurable Systems

**Project Period:**

Spring Semester 2020

**Project Group:**

Group 870

**Participants:**

Bjørn Uttrup Dideriksen

Jakob Krarup Thomsen

James Peter Harris

Kristoffer Calundan Derosche

Magnus Bøgh Borregaard Christensen

**Supervisor:**

Peter Koch

**Report Page Numbers: 92****Appendix Page Numbers: 3****Date of Completion:**

January 31, 2021

**Abstract:**

Within the field of signal processing, a common occurring problem is the symmetric eigenvalue problem. This problem is found in multiple applications, such as facial recognition algorithms, which require real time computation of the symmetric eigenvalue problem. One method for solving this eigenvalue problem is the Jacobi eigenvalue algorithm.

The report details the design of a parallelised Jacobi architecture through RTL description. The Jacobi eigenvalue algorithm is initially studied extensively, and details about its inherent parallelism are uncovered. It is found that the Jacobi eigenvalue algorithm can be implemented using iterative rotations of the matrix's row and column elements. In practice this is achieved using CORDIC modules in a systolic array, inspired by previous work. Through the aid of a data flow paradigm, the processing elements of this array are mapped to RTL in a 1 to 1 manner. The final architecture is largely combinatorial and employs a minimal control scheme.

The designed architecture has no major impact on the field of symmetric eigensolvers. However, a new method for finding the rotation direction using CORDIC is discussed, which would potentially allow for a greater amount of parallelism than previous work within the field.

# Preface

**Table 1:** Table of used mathematical notations.

Overview of Mathematical Notation	
Meaning	Notation
Matrix	Capital roman letters: $A, B, J$
Vector	Lower case roman letters: $v, u, z$
Matrix/Vector element indexing	Shown with subscripts: $A_{i,j}, v_i$
Transpose of matrix/vector	$A^T, v^T$
Scalars	Lower case greek letters: $\alpha, \beta, \lambda$
Index variables	Lower case roman letter: $i, p, q, n, k$
Iterations	$k^{\text{th}}$ iteration shown as: $A^{[k]}, v^{[k]}, \alpha^{[k]}$
Diagonal elements of a matrix	$\text{diag}(A)$
Function	Lower case roman with parentheses: $f(\lambda), p(v)$
Derivatives	Shown with apostrophe: $v', f'(\lambda)$

**Table 2:** Table of Definitions.

Overview of Special Matrices/Vectors/Scalars	
Meaning	Notation
$n$ eigenvalues of a matrix	$\lambda_1, \lambda_2, \dots, \lambda_n$ , where $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$
Diagonal matrix with eigenvalues elements sorted according to size	$\Lambda$
Matrix with eigenvectors as columns	$Q$
Jacobi Rotation Matrix	$J$
Identity Matrix	$I$

In addition, all vectors are column vectors

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Methodology</b>	<b>3</b>
2.1	$A^3$ -Model and Cost Function . . . . .	3
2.2	Gajski-Kuhn Y-chart . . . . .	5
<b>3</b>	<b>The Jacobi Eigenvalue Algorithm</b>	<b>9</b>
3.1	Similarity Transformation . . . . .	9
3.2	Jacobi Rotation . . . . .	9
3.3	Convergence . . . . .	14
3.4	Zeroing Strategies . . . . .	16
3.5	Stopping Criteria . . . . .	18
3.6	Parallelism . . . . .	19
3.7	Algorithm . . . . .	22
3.8	Numerical Example . . . . .	23
<b>4</b>	<b>Algorithm Analysis</b>	<b>25</b>
4.1	Computation of the Arctangent . . . . .	25
4.2	Row and Column Rotation . . . . .	27
4.3	CORDIC Algorithm . . . . .	27
4.4	Algorithm to be Implemented . . . . .	35
4.5	Avoiding Overflow . . . . .	36
4.6	Word length Analysis . . . . .	37
<b>5</b>	<b>Hardware Design</b>	<b>40</b>
5.1	Systolic Arrays . . . . .	40
5.2	Processing Elements . . . . .	41
5.3	Transfer of Matrix Elements . . . . .	46
5.4	Alternative Implementation . . . . .	50

<b>6</b>	<b>RTL Design</b>	<b>53</b>
6.1	Diagonal Processor . . . . .	54
6.2	Off-Diagonal Processor . . . . .	59
6.3	CORDIC RTL Derivation . . . . .	62
6.4	RTL for Systolic Array . . . . .	78
<b>7</b>	<b>Discussion</b>	<b>88</b>
<b>8</b>	<b>Conclusion</b>	<b>89</b>
	<b>Bibliography</b>	<b>90</b>
	<b>Appendix:</b>	<b>93</b>
<b>A</b>	<b>Derivation of equations equivalent to <math>JAJ^T</math></b>	<b>93</b>

# Chapter 1

## Introduction

The symmetric eigenvalue problem can be found in many fields, e.g. data analysis and signal processing. Principal Component Analysis (PCA) can for instance be used for facial recognition and relies on performing eigenvalue decomposition on the covariance matrix of the gathered data [1]. Since the covariance matrix is always symmetric, this is a symmetric eigenvalue problem. In signal processing, the Multiple Signal Classification (MUSIC) algorithm is used for frequency estimation and radio direction estimation [2]. The MUSIC algorithm also relies on eigenvalue decomposition of the covariance matrix of a signal. Both PCA and the MUSIC algorithm may be applied in real-time situations, imposing execution time constraints.

These motivating cases make the symmetric eigenvalue problem as presented in [3] a topic of interest.

The symmetric eigenvalue problem takes the general form of

$$Av = \lambda v \tag{1.1}$$

where  $A$  is a symmetric square matrix,  $v$  is an eigenvector and  $\lambda$  the corresponding eigenvalue.

An eigenpair of any square matrix  $A$  consists of a vector  $v$  and a scalar  $\lambda$ . These are usually referred to as eigen- or characteristic vectors/values. When discussing matrices, one may think of the linear transformation interpretation, which views matrices as transformations of a vector. In this interpretation it is easier to grasp the concept of an eigenpair. The eigenvector is defined as a non-zero vector that is only scaled when the transformation described by  $A$  is applied. This scaling factor  $\lambda$  is then called the eigenvalue.

While both applications presented above rely on eigenvalues and eigenvectors, this project scope is limited to focus only on computation of eigenvalues. This is done to ensure that all design aspects of the mapping of a mathematical algorithm to dedicated hardware can be achieved satisfactorily, within the allotted time.

One of the most widely taught methods for finding the eigenvalues of  $A$  is finding the roots of the characteristic polynomial

$$P_c(\lambda) = \det(\lambda I - A) \tag{1.2}$$

While both algebraic and numerical solvers for these polynomials exist, their solution is impractical for all but simple cases, where the dimensionality of the matrix is small [4]. Simultaneously while numerical polynomial root-finding tools exist, these are outside of the scope of this report.

An initial literature review led to the exploration of several different algorithms to solve the symmetric eigenvalue problem. These included a variety of methods from the QR-algorithm [5][6], the divide and conquer method [7][8][9], Multiple Relatively Robust Representations (MRRR) [10][11][12] as well as Laguerre's method [13]. Through this review the Jacobi algorithm was selected due to its potential for optimization and parallelization as well as its presence in the literature.

It is proposed that some custom hardware architectures may be best suited to this algorithm. This is due to the ability to tailor the hardware directly to the algorithm, while a CPU implementation favours hardware sharing, giving the benefit of general computational ability. Customising a hardware architecture to a specific algorithm will allow optimisation with respect to its inherent parallelism, which is impossible to the same extent in sequential hardware.

This report is a feasibility study investigating a custom hardware based solver for the real symmetric eigenvalue problem, exploring to what extent it is possible to optimise the design with respect to execution time in particular, but not at the excessive expense of numerical properties and area.

The work is primarily structured using the  $A^3$ -model, a model from Aalborg University used in the teaching of the *Reconfigurable and Low Energy Systems* course [14]. This model details the three stages of application, algorithm and architecture, and describes the interconnections between the stages. The Gajski-Kuhn Y-chart will also be used to formally structure the design process, dividing it into the three domains: behavioural, structural and physical.

This architecture will be optimised with respect to execution time, area, power consumption and numerical stability. This is done using a cost function, where different weighting is given to each of the four variables. An architecture that minimises this cost function is said to be optimal in this context.

The goal is to solve the symmetric eigenvalue problem using the Jacobi algorithm with an optimised hardware architecture. As such, the input for this hardware is a real symmetric matrix, and the output will be the corresponding eigenvalues.



# Chapter 2

## Methodology

This chapter outlines well-defined structures, priorities, and methods, in order to minimize the risk of the design process becoming erratic and needlessly complex. To mitigate this risk, this chapter will introduce structured design methodologies that will be used to guide the design process.

### 2.1 $A^3$ -Model and Cost Function

In this section a cost function will be introduced as a tool to guide the iterative design process of designing a hardware architecture suited to compute eigenvalues.

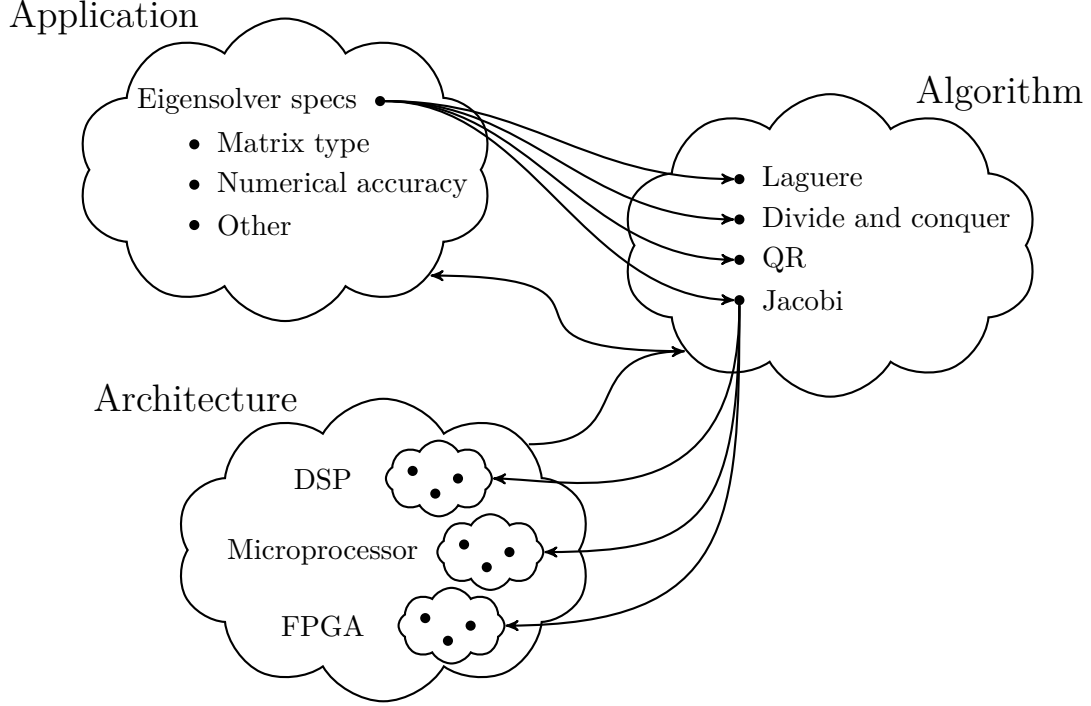
The necessity of such a tool is motivated by the concept that the design process from a specification to the final hardware architecture can be structured according to the  $A^3$ -model [14]. In this model the design is separated into three levels: Application, algorithm and architecture. This allows the addressing of the design process at three distinct levels. However, these are not independent, and choices made at each domain will affect the rest of the design process. The cost function becomes a helpful tool because the transitions between each level of the design process can be made in a multitude of ways. Specifically, a design specification made on the application level may materialise with many different algorithms on the algorithm level, which in turn each can be implemented in hardware using different architecture topologies on the architecture level. Even within a given hardware platform, e.g. FPGAs, multiple realisations of the design can fulfill the specifications. Thus, the transitions between each domain are described by one to many mappings. This mapping is illustrated by Figure 2.1. While these iterative one to many mappings provide designers with a lot of flexibility, the sheer number of possibilities can make it difficult to navigate the solution space and it is not necessarily obvious how choices made in one domain will affect the final hardware architecture.

The cost function encapsulates the properties we wish to achieve and therefore, in a way, becomes the link between the three levels by guiding the design decision on each level such that the final architecture has the "lowest cost". This is also helpful in the iterative process where it is necessary to reconsider earlier design decisions, which can be changed to lower the final cost.

Depending on the specification requirements, the cost function may consist of a variety of different variables. For example, four common variables when dealing with custom architectures could be:

- **Area** is the physical area needed to implement the hardware architecture and is usually directly correlated with the price of the solution.
- **Numerical Properties**, to be understood more broadly as the numerical aspects of the implementation such as sensitivity to fixed point implementation, accuracy and precision, and the numerical range of workable inputs/output values.
- **Power Consumption** is the power required to execute the algorithm.
- **Execution Time** is how long it takes to execute the algorithm.

Which variables are prioritised is determined by the design specification. For example, in the design of a hearing aid the designer is typically forced to pay close attention to all variables, while design of drone stabilisation could place more emphasis on numerical precision and execution time.



**Figure 2.1:**  $A^3$ -model showing the one to many mapping between each design domain [14].

The cost function should reflect that undesirable properties should make the cost increase. Large area and high power consumption as well as slow execution time is most often undesirable, thus one natural expression for the cost of a solution could simply be the sum of these properties. Note that the numerical properties parameter should reflect that negative performance makes the cost function increase. As such, high numerical accuracy can instead be interpreted as low numerical error. A further consideration in the construction of the cost function is that all variables may not be given the same importance depending on a given use case. Thus it would be beneficial to allow for this prioritisation in the cost function. To accommodate this, each variable in the cost function is weighted by a constant reflecting the importance of that specific variable in each design. An example of such a cost function is described by the following equation:

$$C(A, N, P, T) = \omega_A \cdot A + \omega_N \cdot N + \omega_P \cdot P + \omega_T \cdot T \quad (2.1)$$

where  $A$  is area,  $N$  is numerical properties,  $P$  is power consumption, and  $T$  is execution time. Additionally,  $\omega_A \dots \omega_T$  are the weights assigned to each respective variable.

While the above example is presented in a mathematical fashion, in practice the cost function is not defined/used as rigorously. Some costs, such as numerical properties, may prove difficult to quantify. Additionally, it is not necessarily clear exactly how weights should be selected. Due to these practical limitations the cost function is not directly calculated, but rather used to guide the overall design process.

The cost function of the particular design undertaken in this report is largely governed by the exploratory nature of a feasibility study, where no target cost or even specifications are available. Rather, it is an attempt to minimise the cost as much as possible. Therefore, in this scenario the weights assigned to each variable becomes extra important as the actual cost is not necessarily relevant. Instead, the weights show which variable a given improvement will lower the cost most. Equally important is the ratio of the weights, as this shows the extent it makes sense to reduce a single variable before the

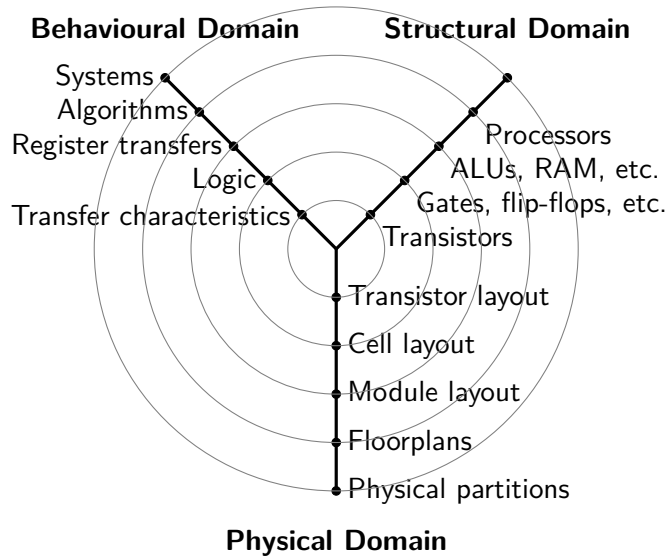
## 2.2. Gajski-Kuhn Y-chart

cost of the other variables shadow the reduction. The last mechanism quickly becomes relevant in a feasibility study as the variables are in many cases conflicting. For example, an architecture with little area available typically must sacrifice execution speed while an architecture with a large area can be hard to run with a low power consumption. This exemplifies the common phenomena that when one variable is squeezed another tends to blow up. Due to this relation, the ratio of the weights in the cost function becomes a good tool to specify the extent it makes sense to squeeze a variable and still have a reasonable or even realistic design.

In this design the primary variable of focus will be the execution time. I.e. exploiting the freedom granted by a custom architecture design to use as much inherent parallelism and optimisation of a known algorithm to reduce the time it takes to execute. However, it is desired to maintain respectable numerical precision such that the potential speed up does not come at the price of uncertain or even false results. The remaining two variables, power and area, are of less interest but with area weighted higher than power.

To summarise: the chosen cost function favours an architecture that allows for fast execution of an eigenvalue algorithm with good numerical precision at the expense of required area and power needed to support and execute the algorithm.

## 2.2 Gajski-Kuhn Y-chart



**Figure 2.2:** Gajski-Kuhn Y-chart with domains and abstraction layers shown [15].

This section acts as an introduction to the Gajski-Kuhn Y-chart. Since its rise in the 1980's the model has been used to design embedded systems. The model aims to narrow the productivity gap, that arose in the integrated circuit designs of the 1980's and remains a challenge today. The productivity gap is a label for the disconnection between the rise of hardware capabilities and the designer productivity. The model alleviates this by adding multiple levels of abstraction in the hardware as well as describing different aspects of each abstraction level.

The model was conceived by Daniel Gajski and Robert Kuhn in 1983 for the purpose of modelling the different aspects of Very Large Scale Implementation (VLSI) hardware design. The general model is shown in Figure 2.2, where the rings represent the abstraction levels and the axes represent domains.

The abstraction levels of the Y-chart are used to separate the development of the system into man-

## 2.2. Gajski-Kuhn Y-chart

ageable capacities. Starting from the outermost ring of the model, the abstraction level pertaining to the overall system.

At the system level, the outermost ring in Figure 2.2, we describe the external standard and custom processors needed for the system to function with the given requirements. This also includes fixing memory locations and communication paths.

At the algorithmic level, the second outermost ring in Figure 2.2, functional descriptions of the interacting subsystems are described either by use of pseudo code, transfer functions, difference equations or similar descriptions. This abstraction level usually entails a block diagram of abstract processors to handle sub-algorithms of the problem.

The register transfer level, the middle ring in Figure 2.2, describes the sub-algorithms from a more hardware near perspective using blocks such as ALUs, multipliers and registers. Investigating the algorithm, by describing precedence relations and signal flows, should help unveil the inherent parallelism. At this stage the scheduling and hardware allocation/sharing is also fixed by use of tools like the synchronous data flow graph and scheduling algorithms.

The logic level, the second innermost ring in Figure 2.2, further describes and implements the inner workings of higher level blocks such as the ALU, this could for example be done using Boolean equations which are implemented in the structural domain as various gates and flip-flops. At the centre of the model is the circuit level, defining transfer functions for the transistor networks and passive components that constitute higher abstraction block.

The model details three design aspects/domains needed to fully describe a hardware implementation. The first domain is behaviour in which the functionality and other specifications are determined. One way to represent the behavioural domain is to describe the system (with the proper abstraction level terminology) as a black box, as this will ensure that the engineer only accounts for the externally perceived behaviour. The behavioural domain should not contain any indication as to how the black box should be built.

The second domain is design structure or simply structure. In this domain the system is schematically designed using different methods depending on abstraction level. This could for example be a netlist of transistors, resistors and capacitors at the circuit abstraction level, or a block diagram showing connection of RTL components such as ALU's, multipliers and storage components.

The third domain is the physical design, which as the name suggests entails a description of the physical dimensions (including both sizes and positions) of components at each abstraction level. A physical design could for example be an integrated circuit or PCB design ready for production.

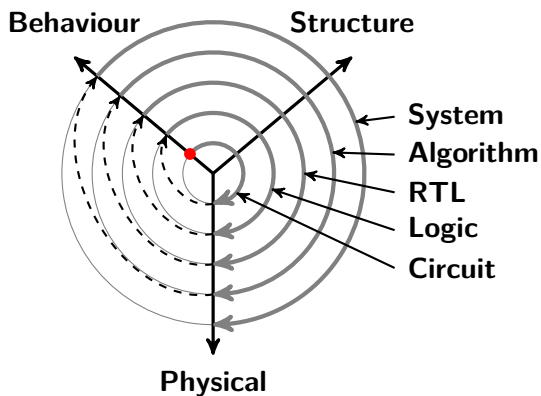


Figure 2.3: Y-chart with bottom-up methodology [16].

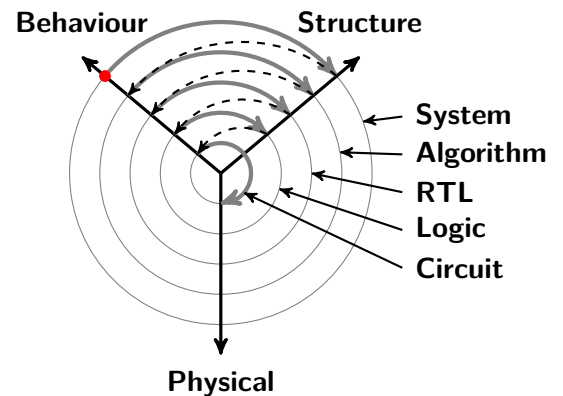
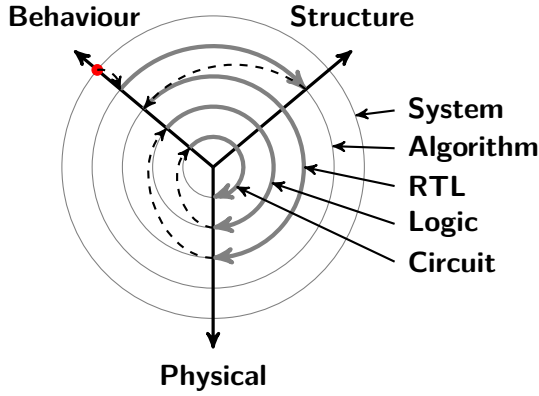
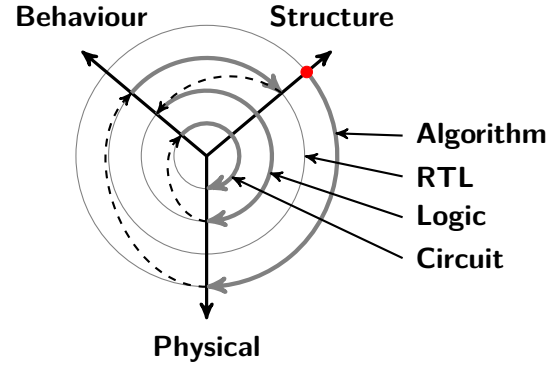


Figure 2.4: Y-chart with top-down methodology [16].

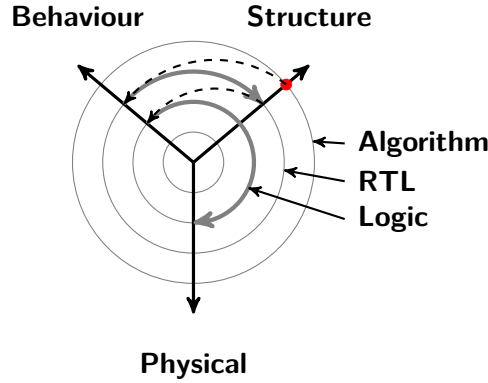
## 2.2. Gajski-Kuhn Y-chart



**Figure 2.5:** Y-chart with meet-in-the-middle methodology [16].



**Figure 2.6:** Y-chart with platform methodology [16].



**Figure 2.7:** Y-chart with FPGA methodology [16].

The model has multiple methodologies aimed at different hardware design processes, these methodologies include bottom-up, top-down, meet-in-the-middle, platform, and FPGA. The y-charts for different methodologies are illustrated in Figures 2.3 through 2.6, with the starting point shown as a red dot. The first methodology is the bottom-up shown in Figure 2.3. Starting from the lowest abstraction level the designer develops components for use in higher abstraction levels. The components designed at each abstraction level depend solely on the components from the lower abstraction levels. This method may be somewhat intuitive, like creating bricks before constructing a house, but comes at the cost of having to build large component libraries. The libraries can however be reused in later projects.

The top-down approach shown in Figure 2.4 describes functionality at the highest possible abstraction level and defines components at this level, which are then synthesised into functionality descriptions and structured components in the next-highest level. This process continues, all the while only describing behaviour and structure of the abstraction levels. When the lowest abstraction level is described and structured, the actual implementation begins. This approach has some severe problems regarding optimisation, since we usually know little to nothing about the component metrics at lower abstraction levels we cannot take these into account when optimising the cost function, e.g. propagation delays remain unknown until the very last step of the design process.

As Gajski states, most designers use a meet-in-the-middle methodology. This helps apply the best of both top-down and bottom-up. It allows the designer to use autonomous bottom-up tools for the well understood lower abstraction levels, while still providing manual synthesis of higher abstraction levels using the top-down approach. The meet-in-the-middle methodology offers a compromise of using predefined Intellectual Property (IP) blocks for common hardware, whether this be at the register

## 2.2. Gajski-Kuhn Y-chart

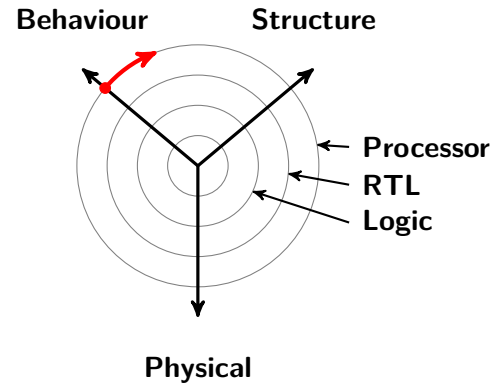
transfer- or logic level and customising hardware for strict requirements that the IP blocks do not fulfil.

The platform methodology is used when developing for an already existing platform, which may already have a set of components defined. In this method the designers may add custom blocks which are usually synthesised using tools at the algorithmic or RTL abstractions. This allows the designer to spend all efforts at the highest abstraction levels, i.e. narrowing the productivity gap significantly. The methodology that we will be using is the FPGA methodology, since this is a platform that allows for rapid and highly customisable architecture development. This is highly based on the top-down approach, where the designers map an algorithm to a platform followed by the synthesis of components to RTL. The RTL components are then defined by means of common libraries. Once all components are defined as RTL the designers will let the FPGA tools handle the low-level synthesis, consisting of mapping and connecting all logic components.

Throughout this report, the transitions between these levels and domains will be described, appearing in text boxes where relevant.

In the introduction we have already started defining the behaviour of our system. It has been decided to solve the symmetric eigenvalue problem using a customised hardware architecture. At this point the behavioural aspects of the system are defined. The system should accept a real symmetric matrix as input and output the eigenvalues for this matrix. Additionally it has been decided that this should be achieved with an architectural implementation of the Jacobi algorithm.

To put this in perspective of the Y-chart, we are now starting the journey from the behavioural to the structural domain in the processor abstraction level. The behaviour of the processor has been defined and we should now move on to the specifics of the algorithm, describing the blocks and steps needed in order to compute the Jacobi algorithm.

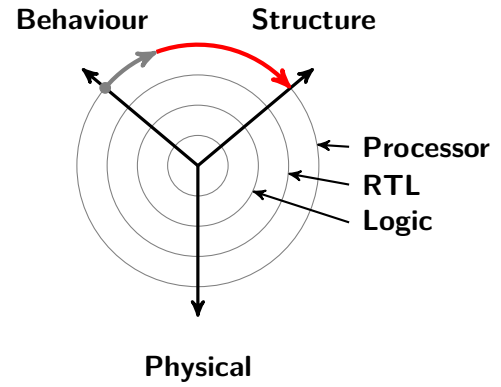


# Chapter 3

## The Jacobi Eigenvalue Algorithm

This chapter provides an overview of the Jacobi eigenvalue algorithm, detailing relevant derivations and examples when necessary. The chapter will also cover the considerations regarding parallelisation of the algorithm, stopping and zeroing strategies, and finally explain the algorithm for further development in pseudocode.

Throughout this chapter a thorough overview of the Jacobi algorithm is given, as we are now moving to the structural domain. From this overview it should become apparent which functionalities are needed in the further development of the architecture. The chapter culminates with an algorithmic description in form of pseudocodes and a blockdiagram. This leads us to the structural domain of the Y-chart.



### 3.1 Similarity Transformation

The Jacobi eigenvalue algorithm functions through iterative usage of similarity transformations. A similarity transform is of the form:

$$B = PAP^{-1} \quad (3.1)$$

Where it is said that matrix  $A$  and  $B$  are similar. A key property of similar matrices is that they have the same eigenvalues. Thus, if matrix  $P$  can be found such that  $B$  becomes diagonal, the eigenvalues of  $A$  will be equivalent to the diagonal entries of  $B$ . This is because the eigenvalues of a diagonal matrix are equal to its diagonal entries.

### 3.2 Jacobi Rotation

Iterative application of Jacobi rotations is the driving mechanism of the Jacobi algorithm. The idea of this iterative procedure is first conceptualised by the 2-dimensional case, following which it is generalised to  $n \times n$  matrices.

The  $2 \times 2$  Jacobi rotation matrix,  $J$ , is defined as [3, p. 189]:

$$J = \begin{bmatrix} \eta & -\sigma \\ \sigma & \eta \end{bmatrix} \quad (3.2)$$

### 3.2. Jacobi Rotation

where:

$$\eta = \cos(\theta) \quad (3.3)$$

$$\sigma = \sin(\theta) \quad (3.4)$$

It can be seen that  $J$  is an orthogonal matrix (i.e.  $JJ^T = I$ ):

$$JJ^T = \begin{bmatrix} \eta & -\sigma \\ \sigma & \eta \end{bmatrix} \begin{bmatrix} \eta & \sigma \\ -\sigma & \eta \end{bmatrix} \quad (3.5)$$

$$= \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad (3.6)$$

$$= \begin{bmatrix} \cos(\theta)^2 + \sin(\theta)^2 & -\sin(\theta)\cos(\theta) + \cos(\theta)\sin(\theta) \\ -\sin(\theta)\cos(\theta) + \cos(\theta)\sin(\theta) & \cos(\theta)^2 + (-\sin(\theta))^2 \end{bmatrix} \quad (3.7)$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (3.8)$$

The orthogonality is significant, as the orthogonality allows one to find the inverse in a computationally simple manner. This allows one to use  $J$  and  $J^T$  in a similarity transformation of a matrix  $A^{[0]}$ , to find a similar matrix,  $A^{[1]}$  [3, p. 189]:

$$A^{[1]} = JA^{[0]}J^{-1} \quad (3.9)$$

$$= JA^{[0]}J^T \quad (3.10)$$

The notation  $A^{[k]}$  implies that it is similar to  $A^{[0]}$ , and that it is derived from a Jacobi rotation of  $A^{[k-1]}$ .

The above transformation can be expanded to derive an equation for each element of  $A^{[1]}$ .

$$A^{[1]} = JA^{[0]}J^T = \begin{bmatrix} \eta & -\sigma \\ \sigma & \eta \end{bmatrix} \begin{bmatrix} \alpha & \beta \\ \beta & \delta \end{bmatrix} \begin{bmatrix} \eta & \sigma \\ -\sigma & \eta \end{bmatrix} \quad (3.11)$$

$$= \begin{bmatrix} \eta\alpha - \sigma\beta & \eta\beta - \sigma\delta \\ \sigma\alpha + \eta\beta & \sigma\beta + \eta\delta \end{bmatrix} \begin{bmatrix} \eta & \sigma \\ -\sigma & \eta \end{bmatrix} \quad (3.12)$$

$$= \begin{bmatrix} \eta^2\alpha - \eta\sigma\beta - \eta\sigma\beta + \sigma\delta & \eta\sigma\alpha - \sigma^2\beta + \eta^2\beta - \eta\sigma\delta \\ \eta\sigma\alpha - \sigma^2\beta + \eta^2\beta - \eta\sigma\delta & \sigma^2\alpha + \eta\sigma\beta + \eta\sigma\beta + \eta^2\delta \end{bmatrix} \quad (3.13)$$

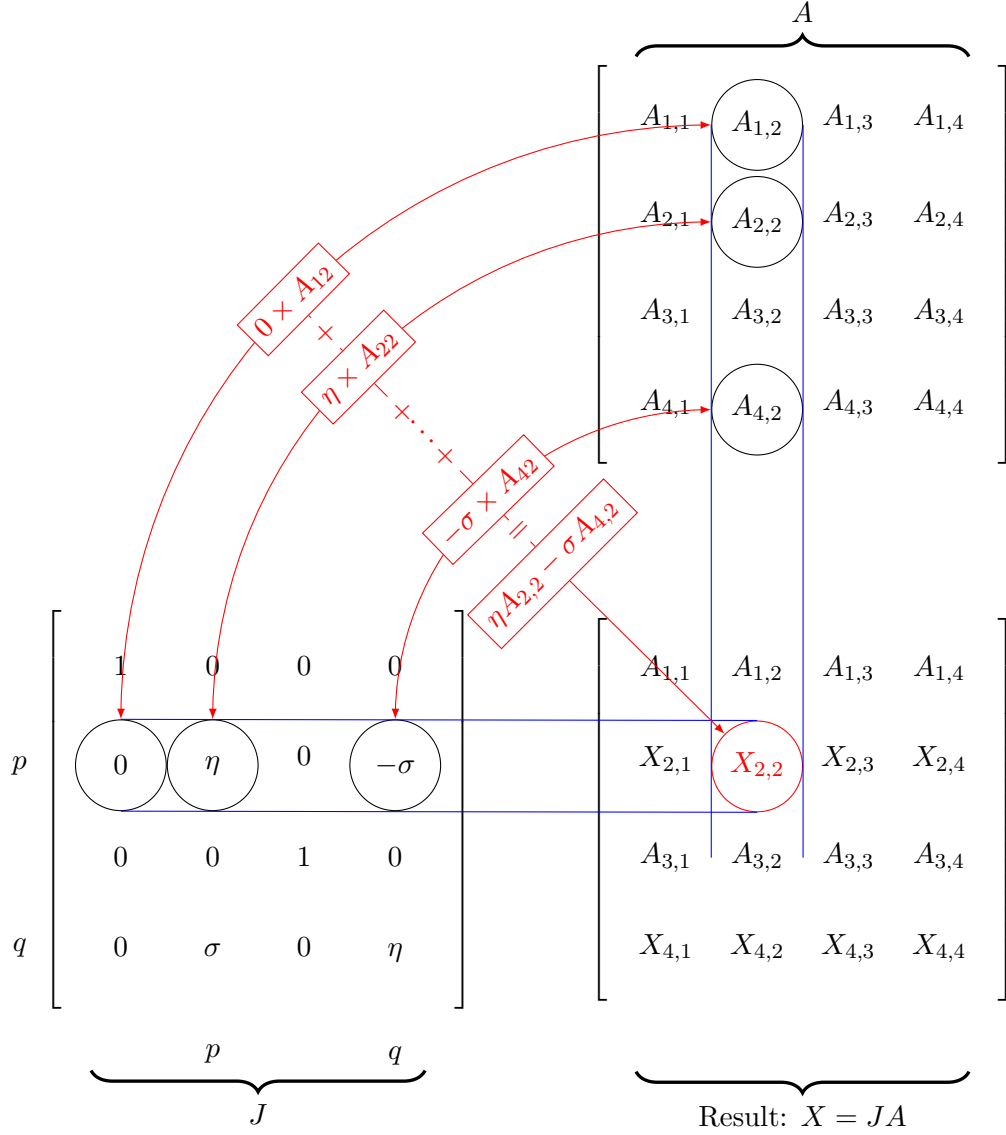
$$= \begin{bmatrix} \eta^2\alpha + \sigma^2\delta - 2\eta\sigma\beta & (\eta^2 - \sigma^2)\beta + (\alpha - \delta)\eta\sigma \\ (\eta^2 - \sigma^2)\beta + (\alpha - \delta)\eta\sigma & \sigma^2\alpha + \eta^2\delta + 2\eta\sigma\beta \end{bmatrix} \quad (3.14)$$

This matrix can be made diagonal by setting the non-diagonal entry equal to zero, and solving with





### 3.2. Jacobi Rotation



**Figure 3.1:** Illustration of the matrix multiplication used to find  $X = JA$ .  $n = 4, p = 2, q = 4$  [18].

With the aid of Figure 3.1, it should be clear that the elements of resulting matrix  $X$  are:

$$X_{2,i} = \eta A_{2,i} - \sigma A_{4,i} \quad (3.25)$$

$$X_{4,i} = \sigma A_{2,i} + \eta A_{4,i} \quad (3.26)$$

$$X_{i,j} = A_{i,j} \quad \text{for } i \neq 2, 4 \quad (3.27)$$

This hardly changes when generalised for arbitrary  $n, p$ , and  $q$ :

$$X_{p,i} = \eta A_{p,i} - \sigma A_{q,i} \quad (3.28)$$

$$X_{q,i} = \sigma A_{p,i} + \eta A_{q,i} \quad (3.29)$$

$$X_{i,j} = A_{i,j} \quad \text{for } i \neq p, q \quad (3.30)$$

This can conveniently be restated using linear algebra and a 2-dimensional rotation matrix:

$$\begin{bmatrix} X_{p,i} \\ X_{q,i} \end{bmatrix} = \begin{bmatrix} \eta & -\sigma \\ \sigma & \eta \end{bmatrix} \begin{bmatrix} A_{p,i} \\ A_{q,i} \end{bmatrix} \quad (3.31)$$

$$X_{i,j} = A_{i,j} \quad \text{for } i \neq p, q \quad (3.32)$$

### 3.2. Jacobi Rotation

As a rotation matrix rotates a vector, the  $JA$  operation is equivalent to rotating all the vectors formed by the elements of row  $p$  and  $q$  of  $A$ . The rotation angle is specified by  $\theta$ .

Following the calculation of  $X = JA$ , the  $XJ^T$  operation is performed. This is almost identical to the previous rotation of row elements, except the right-hand multiplication  $J^T$  acts on column  $p$  and  $q$  of matrix  $X$ :

$$A_{i,p}^{[k+1]} = \eta X_{i,p} - \sigma X_{i,q} \quad (3.33)$$

$$A_{i,q}^{[k+1]} = \sigma X_{i,p} + \eta X_{i,q} \quad (3.34)$$

$$A_{i,j}^{[k+1]} = X_{i,j} \quad \text{for } i \neq p, q \quad (3.35)$$

Again, this can also be shown using a rotation matrix:

$$\begin{bmatrix} A_{i,p}^{[k+1]} \\ A_{i,q}^{[k+1]} \end{bmatrix} = \begin{bmatrix} \eta & -\sigma \\ \sigma & \eta \end{bmatrix} \begin{bmatrix} X_{i,p} \\ X_{i,q} \end{bmatrix} \quad (3.36)$$

$$A^{[k+1]} = A_{i,j} \quad \text{for } i \neq p, q \quad (3.37)$$

Now that we have expression for both of the matrix multiplications in the similarity transformation, it is possible to combine them to find a set of equations for finding the elements of  $A^{[k+1]}$ , with respect to the elements of  $A^{[k]}$ . The exact derivations of this combination can be found in Appendix A. However, the result of this exercise is [3, p. 191]:

$$A_{p,q}^{[k+1]} = A_{q,p}^{[k+1]} = (\eta^2 - \sigma^2)A_{p,q}^{[k]} + (A_{p,p}^{[k]} - A_{q,q}^{[k]})\eta\sigma \quad (3.38)$$

$$A_{p,i}^{[k+1]} = A_{i,p}^{[k+1]} = \eta A_{p,i}^{[k]} - \sigma A_{q,i}^{[k]} \quad \text{for } \forall i \neq p, q \quad (3.39)$$

$$A_{q,i}^{[k+1]} = A_{i,q}^{[k+1]} = \eta A_{q,i}^{[k]} + \sigma A_{p,i}^{[k]} \quad \text{for } \forall i \neq p, q \quad (3.40)$$

$$A_{p,p}^{[k+1]} = \eta^2 A_{p,p}^{[k]} + \sigma^2 A_{q,q}^{[k]} - 2\eta\sigma A_{p,q}^{[k]} \quad (3.41)$$

$$A_{q,q}^{[k+1]} = \eta^2 A_{q,q}^{[k]} + \sigma^2 A_{p,p}^{[k]} + 2\eta\sigma A_{p,q}^{[k]} \quad (3.42)$$

$$A_{i,j}^{[k+1]} = A_{i,j}^{[k]} \quad \text{for } \forall i \neq p, q \quad \forall j \neq p, q \quad (3.43)$$

Based on these equations for the entries of  $A^{[k+1]}$ , it is evident that the similarity transformation only affects the entries in row and column  $p$ , as well as the entries in row and column  $q$ . The remaining entries of  $A^{[k+1]}$  are equal to the entries in  $A^{[k]}$ . As an example, Figure 3.2 illustrates which entries are affected if a Jacobi rotation with  $p = 1$  and  $q = 2$ , is applied on a  $4 \times 4$  matrix:

$$\begin{array}{cc} & \begin{matrix} p & q \end{matrix} \\ \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} & \begin{matrix} p \\ q \end{matrix} \end{array}$$

**Figure 3.2:** Illustration of the affected entries of a  $4 \times 4$  matrix, when applying a Jacobi rotation with  $p = 1$  and  $q = 2$ . The non-highlighted elements remain unchanged.

The reader should also note that the equations for finding  $A_{p,p}^{[k+1]}$ ,  $A_{q,q}^{[k+1]}$ ,  $A_{p,q}^{[k+1]}$ , and  $A_{q,p}^{[k+1]}$  are exactly the same as the 2D case in Equation (3.14). Thus Equation (3.22) (with  $\alpha, \beta, \delta$  substituted) can be

### 3.3. Convergence

used to find the angle used to zero entry  $A_{p,q}^{[k+1]}$  and  $A_{q,p}^{[k+1]}$ :

$$\theta = \frac{\arctan\left(\frac{2A_{p,q}^{[k]}}{A_{q,q}^{[k]} - A_{p,p}^{[k]}}\right)}{2} \quad (3.44)$$

One caveat of Equation (3.44) is that is valid for two different values of  $\theta$ . One in the range  $(-\frac{\pi}{4}, \frac{\pi}{4}]$ , and another in the disjoint interval  $[-\frac{\pi}{2}, -\frac{\pi}{4}] \cup (\frac{\pi}{4}, \frac{\pi}{2}]$  [3, p. 195]. Here it is customary to choose the  $\theta$  in the range  $[-\frac{\pi}{4}, \frac{\pi}{4}]$  [19, p. 429], as the other option will result in diagonal entries  $A_{p,p}$  and  $A_{q,q}$  switching place when a matrix  $A^{[k]}$  is close to diagonal. When this is the case, Equation (3.44) will (for distinct  $A_{p,p}^{[k]}$  and  $A_{q,q}^{[k]}$ ) tend to  $\theta = 0$  or  $\theta = \frac{\pi}{2}$ . When  $\theta = \frac{\pi}{2}$  is used, it is evident from Equation (3.41) and Equation (3.42) that the  $\sin(\theta)^2$  term will dominate, and thus the values of entry  $A_{p,p}^{[k]}$  and  $A_{q,q}^{[k]}$  are switched.

Based on the previous derivations, one might be fooled to conclude that the matrix can be diagonalised by zeroing each off-diagonal entry once. However, this is not the case, as each similarity transformation also impacts the row and column of the entry being zeroed, thus potentially unzeroing a previously zeroed entry.

While each Jacobi rotation does not guarantee to zero an entry permanently, it transfers part of the matrix norm from the off-diagonal entries to the diagonal entries, thereby getting it closer to a complete diagonalisation. This will be rigourously proven in the following section.

Thus, to effectively diagonalise the input matrix Jacobi rotations must be applied iteratively until the diagonal entries have converged to the eigenvalues [20, p. 266]:

$$\Lambda = A^{[k+1]} = J^{[k]} \dots J^{[1]} J^{[0]} A^{[0]} J^{[0]T} J^{[1]T} \dots J^{[k]T} \quad (3.45)$$

In the above, and going forward, assume that all  $J^{[i]}$  use a  $\theta^{[i]}$  and some  $p^{[i]}$  and  $q^{[i]}$  to zero entry  $A_{p,q}^{[i]}$  and  $A_{q,p}^{[i]}$ .

### 3.3 Convergence

We now prove that iterative use of Jacobi rotations causes matrix  $A$  to converge to diagonal form. As a precursor for this, key definitions used throughout the proof are presented.

The Frobenius norm of a real matrix is defined as [3, p. 15]:

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n A_{i,j}^2} \quad (3.46)$$

This matrix norm is invariant under orthogonal transformations [3, p. 15], i.e:

$$\|A\|_F = \|JA\|_F \quad (3.47)$$

and thus in extension:

$$\|A\|_F = \|JAJ^T\|_F \quad (3.48)$$

Furthermore, let  $E$  and  $D$  be matrices containing the off diagonal and diagonal components of  $A$ , respectively:

$$E = A - \text{diag}(A) \quad (3.49)$$

$$D = \text{diag}(A) \quad (3.50)$$

### 3.3. Convergence

Then, using Equation (3.38) to Equation (3.43), it can be shown that the Frobenius norm of the off diagonal entries decreases for every Jacobi rotation [3, p. 193]. In the following, note that a matrix element raised to a power of two, is expressed as,  $E_{i,j}^{[k+1]^2}$ :

$$\|E^{[k+1]}\|_F = \sqrt{\sum_{i=1}^n \sum_{j=1}^n E_{i,j}^{[k+1]^2}} \quad (3.51)$$

$$\|E^{[k+1]}\|_F^2 = \sum_{i=1}^n \sum_{j=1}^n E_{i,j}^{[k+1]^2} \quad (3.52)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{i=1}^n \left( E_{p,i}^{[k+1]^2} + E_{q,i}^{[k+1]^2} \right) \quad (3.53)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{\substack{i=1 \\ i \neq p,q}}^n \left( E_{p,i}^{[k+1]^2} + E_{q,i}^{[k+1]^2} \right) + \underbrace{2E_{p,q}^{[k+1]^2} + E_{p,p}^{[k+1]^2} + E_{q,q}^{[k+1]^2}}_{\text{these are all equal to 0}} \quad (3.54)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{\substack{i=1 \\ i \neq p,q}}^n \left( E_{p,i}^{[k+1]^2} + E_{q,i}^{[k+1]^2} \right) \quad (3.55)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{\substack{i=1 \\ i \neq p,q}}^n \left[ \left( \eta E_{p,i}^{[k]} - \sigma E_{q,i}^{[k]} \right)^2 + \left( \sigma E_{p,i}^{[k]} + \eta E_{q,i}^{[k]} \right)^2 \right] \quad (3.56)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{\substack{i=1 \\ i \neq p,q}}^n \left[ E_{p,i}^{[k]^2} (\eta^2 + \sigma^2) + E_{q,i}^{[k]^2} (\eta^2 + \sigma^2) \right] \quad (3.57)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{\substack{i=1 \\ i \neq p,q}}^n \left[ E_{p,i}^{[k]^2} (\cos(\theta)^2 + \sin(\theta)^2) + E_{q,i}^{[k]^2} (\cos(\theta)^2 + \sin(\theta)^2) \right] \quad (3.58)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{\substack{i=1 \\ i \neq p,q}}^n \left( E_{p,i}^{[k]^2} + E_{q,i}^{[k]^2} \right) \quad (3.59)$$

$$= \sum_{\substack{i=1 \\ i \neq p,q}}^n \sum_{\substack{j=1 \\ j \neq p,q}}^n E_{i,j}^{[k]^2} + \sum_{i=1}^n \left( E_{p,i}^{[k]^2} + E_{q,i}^{[k]^2} \right) - \underbrace{E_{p,p}^{[k]^2} - E_{q,q}^{[k]^2}}_{\text{equal to 0}} - 2E_{p,q}^{[k]^2} \quad (3.60)$$

$$= \sum_{i=1}^n \sum_{j=1}^n E_{i,j}^{[k]^2} - 2E_{p,q}^{[k]^2} \quad (3.61)$$

$$= \|E^{[k]}\|_F^2 - 2E_{p,q}^{[k]^2} \quad (3.62)$$

Given that  $\|A^{[k+1]}\|_F^2 = \|A^{[k]}\|_F^2$ , it follows that:

$$\|D^{[k+1]}\|_F^2 = \|D^{[k]}\|_F^2 + 2E_{p,q}^{[k]^2} \quad (3.63)$$

Thus a Jacobi rotation transfers  $2E_{p,q}^{[k]^2}$  from the off-diagonal to the diagonal entries. Following a sufficient number of rotations, the majority of the "weight" of the matrix should be located in the diagonal, and thus it should have converged to diagonal form.

## 3.4 Zeroing Strategies

The previous sections mostly focused on the mathematical details of how a Jacobi rotations can be used to zero an entry  $A_{p,q}$ . However, so far the details of which entry to zero in a given iteration has mostly been ignored. This issue is covered in this section, and two well known zeroing strategies are presented.

### 3.4.1 Classical

In Equation (3.62), it is given that the squared Frobenius norm decreases by  $2A_{p,q}^2$  every Jacobi rotation. Based on this it intuitively makes sense to zero the largest absolute entry of the current iteration of  $A$ . This selection strategy is referred to as "Classical Jacobi" [19, p. 428].

This approach allows one to determine a lower bound for how much the Frobenius norm of the off diagonal elements decrease for every Jacobi rotation. In other words, a lower bound for the rate of convergence to diagonal form can be established. Consider the case where all off-diagonal values are equal. Here the squared Frobenius norm of the off-diagonal entries is  $\|E\|_F^2 = n(n-1)E_{p,q}^2$  where  $n(n-1)$  is the number of off-diagonal entries of an  $n \times n$  matrix. Based on this, the following relationship is easily established [20, p. 267]:

$$\|E^{[k+1]}\|_F^2 = \|E^{[k]}\|_F^2 - 2E_{p,q}^{[k]2} \quad (3.64)$$

$$= \|E^{[k]}\|_F^2 - \frac{2}{n(n-1)}\|E^{[k]}\|_F^2 \quad (3.65)$$

$$= \left(1 - \frac{2}{n(n-1)}\right) \|E^{[k]}\|_F^2 \quad (3.66)$$

This is the minimum amount the Frobenius norm of the off-diagonal entries can change in a single iteration, and thus defines a lower bound for the convergence rate. If the off-diagonal entries are not equal, it follows that a Jacobi rotation will cause the Frobenius norm of the off-diagonal entries to decrease faster than specified by this lower bound. Thus for the general case it follows that (and extended to apply for multiple iterations) [20, p. 287]:

$$\|E^{[k+1]}\|_F^2 \leq \left(1 - \frac{2}{n(n-1)}\right) \|E^{[k]}\|_F^2 \leq \left(1 - \frac{2}{n(n-1)}\right)^k \|E^{[0]}\|_F^2 \quad (3.67)$$

In practise, off diagonal entries do not remain equal across multiple Jacobi rotations, and therefore convergence is usually much faster than what is established by the lower bound. In fact, following multiple iterations the convergence rate ultimately turns quadratic [19, p. 429].

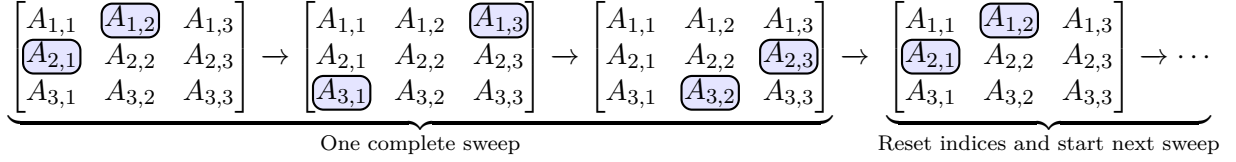
While the classical method is guaranteed to provide optimal convergence for the Jacobi algorithm, it is rarely used in practice. This is due to the  $n(n-1)/2$  operations required to find the largest off-diagonal entry of a symmetric matrix [17, p. 235].

### 3.4.2 Cyclical

In Cyclical Jacobi, the off-diagonal elements are zeroed in a cyclical manner. Once every element has been zeroed, the cycle repeats itself. The act of zeroing every element in a matrix once, is referred to as a sweep [20, p. 269], which is denoted as  $\tau$ . A sweep consists of  $\tau = \frac{n(n-1)}{2}$  Jacobi rotations.

### 3.4. Zeroing Strategies

One possible scheme would be to zero all the elements in a row before going to the next row. The exact order of this cycle would be [20, p. 269]:  $(1, 2), (1, 3), \dots, (1, n); (2, 3), (2, 4), \dots, (2, n); \dots; (n-1, n)$  and then return to  $(1, 2)$ . This cyclical approach is illustrated for a  $3 \times 3$  matrix in Figure 3.3.



**Figure 3.3:** Illustration of Cyclical sweep. The highlighted elements are the elements to be zeroed by the current Jacobi rotation.

As the cyclical method does not need to search for the next entry to zero, it is much simpler than the classical method. Furthermore, it is ideal for parallel computing. This aspect will be explored in Section 3.6.

However, unlike Classical Jacobi, Cyclical Jacobi does not guarantee optimal convergence. Despite this, the convergence is still initially linear, following which it turns quadratic. The intuition behind cyclical Jacobi's quadratic convergence is presented below.

Let  $A$  have distinct eigenvalues, and let  $\delta = \min(|\lambda_i - \lambda_j|)_{i \neq j}$  [21, p. 506]. Following multiple Jacobi rotations,  $\|E^{[k]}\|_F = \epsilon$ , where  $\epsilon \ll \delta$  [21, p. 506]. Then it is clear that in all subsequent rotations, the angle will be very small:

$$\theta = \frac{\arctan\left(\frac{2A_{p,q}^{[k]}}{A_{q,q}^{[k]} - A_{p,p}^{[k]}}\right)}{2} \leq \frac{\arctan\left(\frac{2\epsilon}{\delta}\right)}{2} \leq \frac{\epsilon}{\delta} \quad (3.68)$$

Using small angle approximations, it follows that for all subsequent rotations  $|\sigma| \leq \frac{\epsilon}{\delta}$  and  $\eta \approx 1$  [21, p. 506].

We now consider the beginning of a new cyclical Jacobi Sweep. At some rotation in the sweep, matrix element  $A_{i,j}$  ( $i \neq j$ ) is zeroed. In subsequent rotations targeting other entries in the same row/column as element  $A_{i,j}$ , element  $A_{i,j}$  will be modified according to Equation (3.39) and Equation (3.40):

$$\begin{aligned} A_{i,j}^{[k+u]} &= \eta A_{i,j}^{[k]} - \sigma A_{q,j}^{[k]} \\ A_{i,j}^{[k+u]} &= \eta A_{i,j}^{[k]} + \sigma A_{p,j}^{[k]} \end{aligned}$$

Since  $A_{i,j} \leq \|E\|_F \leq \epsilon$ ,  $|\sigma| \leq \frac{\epsilon}{\delta}$ , and  $\eta \approx 1$  it is clear that:

$$|A_{i,j}^{[k+u]}| \leq \left| A_{i,j}^{[k]} + \frac{\epsilon^2}{\delta} \right| \quad (3.69)$$

Thus, each subsequent rotations will at most change the value of entry  $A_{i,j}$  by  $\frac{\epsilon^2}{\delta}$ . Additionally, as  $A_{i,j}$  was initially zeroed it follows that at the end of the sweep, that entry  $A_{i,j}$  is bounded by [21, p. 506]:

$$|A_{i,j}| \leq \left| \gamma_{i,j} \frac{\epsilon^2}{\delta} \right| \quad (3.70)$$

### 3.5. Stopping Criteria

where  $\gamma_{i,j}$  is the number of times entry  $A_{i,j}$  has been modified in the sweep, following its initial zeroing.

The Frobenius norm at the end of the sweep is then bounded by:

$$\|E^{[k+\tau]}\|_F \leq \sqrt{\sum_{i=1}^m \sum_{j=1}^n \left(\gamma_{i,j} \frac{\epsilon^2}{\delta}\right)^2} = \sqrt{\sum_{i=1}^m \sum_{j=1}^n \left(\frac{\gamma_{i,j}}{\delta}\right)^2} \|E^{[k]}\|_F^2 \quad (3.71)$$

Or written in more general terms:

$$\|E^{[k+\tau]}\|_F \leq \mu \|E^{[k]}\|_F^2 \quad (3.72)$$

where  $\mu$  is some constant [19, p. 429].

This shows that following enough initial iterations, Cyclical Jacobi will converge quadratically after a complete sweep. The previous analysis is only valid for distinct eigenvalues. If eigenvalues are non-distinct, then  $\delta$  will be equal to zero. As most of the presented equation relies on division by  $\delta$ , the analysis is obviously void when  $\delta$  is zero. However it should be noted that proofs for the non-distinct case also exist [22].

## 3.5 Stopping Criteria

Since the Jacobi algorithm is iterative, it has to be stopped when a satisfactory solution is found. However, there exists little rigorous theory regarding this aspect of Jacobi [19, p. 429]. Instead the area is dominated by heuristic methods. Some of these methods are used due to their low computational cost, while others attempt to guarantee some minimum accuracy. Two commonly occurring methods are presented in this section.

### 3.5.1 Ratio of Frobenius Norms

One possibility for a stopping criteria would be to consider the frobenius norm of the off diagonal elements,  $\|E^{[i]}\|_F^2$ . As this value tends to zero as the matrix converges to a diagonal form, one could conceivably terminate the algorithm when  $\|E^{[i]}\|_F^2$  is less than some predetermined threshold,  $\epsilon_{stop}$ . However, this approach only considers the off diagonal elements, and does not take the magnitude of the eigenvalues into account. Consider a matrix with very large eigenvalues. Here  $\|E^{[i]}\|_F^2$  need only be a few orders of magnitudes smaller than  $\|D^{[i]}\|_F^2$ , to indicate "reasonable" convergence to diagonal form. If on the other hand a matrix has very small eigenvalues,  $\|E^{[i]}\|_F^2$  must be much smaller, than in the large eigenvalue case, to reflect a similar degree of convergence. Thus  $\|E^{[i]}\|_F^2$  is not by itself a good indicator of accuracy, rather one would like to consider the ratio between  $\|E^{[i]}\|_F^2$  and  $\|D^{[i]}\|_F^2$  [23, p. 195]:

$$\frac{\|E^{[i]}\|_F^2}{\|D^{[i]}\|_F^2} \leq \epsilon_{stop}. \quad (3.73)$$

While this method can dynamically adapt to the input matrix, it is very computationally expensive, as both  $\|E^{[i]}\|_F^2$  and  $\|D^{[i]}\|_F^2$  must be calculated periodically.



### 3.5.2 Fixed Iteration Count

Perhaps the most obvious stopping criteria is to let the algorithm perform a fixed number of iteration before it terminates. Unfortunately, while it is a computationally simple method, as this is a static approach it is not possible to guarantee a specified accuracy. Thus for certain ill-conditioned matrices, the algorithm will terminate before the minimum required accuracy is reached. On the other hand, for well conditioned matrices where the eigenvalues are easily determined, the algorithm can perform an excessive number of iterations.

However, in practice it has been observed that the number of iterations required to reach a certain accuracy is logarithmically proportional to the matrix size. This gives rise to the following equation [24]:

$$\text{Fixed iterations} = \lceil \epsilon_{stop} \tau \log_2(n) \rceil \quad (3.74)$$

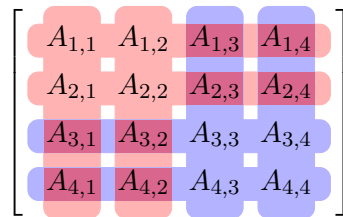
where  $\epsilon_{stop}$  is some constant proportional to the accuracy.

Since this stopping criteria only requires one to count the number of elapsed iterations, it is computationally inexpensive to evaluate. As this is a clear advantage when trying to maximise computational speed, going forward this is the stopping criteria that will be used.

## 3.6 Parallelism

The Jacobi eigenvalue algorithm may be parallelised by simultaneously computing different row-column pairs of the matrix. In doing so, it is assured that only one pair of diagonal entries is affected by one rotation at each step, and multiple off-diagonal entries may be zeroed in the same step [25].

Suppose a  $4 \times 4$  matrix as in Figure 3.4, which will be used below as a brief example to show the fundamental principle.



**Figure 3.4:** Potential parallelism in a  $4 \times 4$  matrix.

In this case, the off-diagonals at  $A_{1,2}$  and  $A_{2,1}$  are zeroed by the rotation highlighted in red, and  $A_{3,4}$  and  $A_{4,3}$  are zeroed by the rotation highlighted in in blue. At the points of intersection in purple, both rotations affect the entries. This involves making the rotations consecutively, or multiplying the rotations prior to multiplying with the entries in  $A$ . This is equivalent to

$$A^{[2]} = J^{[1]} J^{[0]} A^{[0]} J^{[0]T} J^{[1]T} \quad (3.75)$$

As the values of  $\theta$  can be calculated independently for each row-column pair using Equation (3.44) given a feasible set of pairs, it is possible to construct a full Jacobi rotation matrix immediately.

### 3.6. Parallelism

This means that for the  $4 \times 4$  example described in Figure 3.4, the rotation matrix as described by Equation (3.2) has the form

$$J(\theta_1, \theta_2) = \begin{bmatrix} c(\theta_1) & -s(\theta_1) & 0 & 0 \\ s(\theta_1) & c(\theta_1) & 0 & 0 \\ 0 & 0 & c(\theta_2) & -s(\theta_2) \\ 0 & 0 & s(\theta_2) & c(\theta_2) \end{bmatrix} \quad (3.76)$$

From this example, we can see that we are able to zero a maximum of 2 symmetric off-diagonal pairs per step.

There is also the question of deciding which order the row-column pairs should be zeroed in. Above, we can see that there remain 4 more off-diagonal pairs that need to be zeroed before the sweep is complete. Given that we may only take 2 pairs at a time in this case, it is easy to see that we need an additional 2 parallel rotations to complete the sweep. This collection of row-column pairs that includes all rotations in a sweep is referred to as a list of pairs.

Now generalising this process for  $n \times n$  matrices, the upper limit of  $\frac{n}{2}$  zeroes is possible to maintain by appending the matrix with zeroes and a one on the diagonal, for the cases where this adjustment is not made, the limit becomes  $\lfloor \frac{n}{2} \rfloor$  [26]. As an example, where  $A$  is instead a  $3 \times 3$  matrix, it is appended as below, allowing the same  $4 \times 4$  rotation matrix as in Equation (3.76) to be used

$$\hat{A} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & 0 \\ A_{2,1} & A_{2,2} & A_{2,3} & 0 \\ A_{3,1} & A_{3,2} & A_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.77)$$

This appended column is then ignored when extracting the final eigenvalues, as a matrix can not have more eigenvalues than it does dimensions. Although this does present some computational inefficiency as the same operations are required on a row-column pair that is ultimately ignored, the ratio of useful to ignored computations tends to infinity, as the odd size of the input matrices tends to infinity.

There are two main approaches to ordering for a matrix with odd dimensions. These are appending the matrix with a row and column from the Identity matrix [19], or leaving one row-column pair unchanged in each step [25][27]. The appendment approach presents the most possibility for parallelism, as it is then possible to make a complete set of pairs, however at the expense of some computational efficiency, as the results of the added row-column pair are not of significance.

This approach will increase the amount of row-column pairs zeroed per iteration at the expense of some relatively small losses in efficiency, which is in line with the cost function presented in Section 2.1, therefore this is the method that will be used for this project. Because of this, we now consider only matrices of even dimensionality in the design process, as any matrix can be converted to this form by the above method.

There are two necessary conditions to ensure that the maximum possible number of row-column pairs are zeroed in each rotation. This minimises the amount of rotations per sweep, by ensuring non-conflicting rotations are computed and applied in parallel. These are [25]

- Each rotation should be constructed so as to zero  $\frac{n}{2}$  off-diagonal elements
- Each sweep should zero each off-diagonal only once.

### 3.6. Parallelism

For cyclical methods, a list of pairs meeting these conditions means that the maximum number of rotations are done concurrently, increasing speed of computation. Proofs of the mathematical equivalence of some orderings can be seen in [27].

The comparative complexity of generating these parallel pairs for the case of a classical Jacobi algorithm is an important motivation in the selection of cyclical instead of classical implementations. In Section 3.4.1, it was shown that selecting the largest off-diagonal was necessary to provide a maximum reduction in the Frobenius norm of the off-diagonal. Now this concept can be extended to find the set of non-conflicting parallel pairs that maximise this reduction. This is mimicing to the travelling salesman problem, in that each "city" in this case represented by row-column entries must be "visited", or included in the listing once, forming a list of  $\frac{n}{2}$  parallel pairs for that iteration. Similarly, the index of the largest off-diagonal is not necessarily included in the set, but it is rather the combination as a whole. While combinatorial optimisation algorithms such as simulated annealing [28, p. 274] may be able to be used to solve such problems, these have significantly more time, memory and hardware complexity than cyclical parallel indexing methods, therefore a classical Jacobi algorithm will not be considered.

Since in a parallelised cyclical implementation, each off-diagonal should only be zeroed once per sweep, the generation of the indices must ensure this. The algorithm presented in [19, p. 432], referred to as "chess tournament", works by shuffling the indices in a controlled manner until all unique non-conflicting sets are generated. The algorithm mimics an organisation technique employed in e.g. a tournament setting, where all players must play all other players exactly once. Note that this means that the algorithm will only work for even values of  $n$ . The algorithm has two arrays of "contestants",  $p$  and  $q$ , which are indexed by  $k$ . The "contestants" in  $p^{[k]}$  are matched with the ones in  $q^{[k]}$ . When used to generate the row and column combinations for a Jacobi sweep, the values of  $p$  and  $q$  are simply the indices of the matrix. The algorithm is scalable to any value of  $n$ , provided enough hardware is available. Table 3.1 shows the initial state of this algorithm with  $n = 8$ , as well as one cycle of the algorithm.

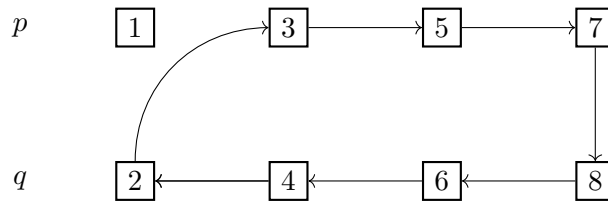
**Table 3.1:** Table showing the initial index combinations (left) and after one cycle (right).

<b>k</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>
<i>p</i>	1	3	5	7
<i>q</i>	2	4	6	8

→

<b>k</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>
<i>p</i>	1	2	3	5
<i>q</i>	4	6	8	7

Once one set of parallel rotations have been carried out, the algorithm shuffles the arrays in a manner that ensure all combinations are generated once and only once. In essence, the algorithm will cycle all entries in a circular manner in a pattern as shown in Figure 3.5.



**Figure 3.5:** Figure showing the cyclical movement of the music algorithm with  $n = 8$ .

### 3.7 Algorithm

Now that the mathematical details of the Jacobi eigenvalue algorithm have been explored, we unify the steps and present an algorithm inspired from the previous derivations. The pseudocode description of a purely sequential variant is shown in Algorithm 1.

---

**Algorithm 1** Sequential Cyclical Jacobi Algorithm

---

```

1: Accept a symmetric matrix  $A$  as input
2: while stopping criteria is not fulfilled do
3:    $(p, q) = \text{next index}$ 
4:    $\theta = \frac{1}{2} \arctan\left(\frac{2A_{p,q}}{A_{q,q} - A_{p,p}}\right)$ 
5:    $J = I$ 
6:    $J_{p,p} = \cos(\theta)$ ,  $J_{q,q} = \cos(\theta)$ ,  $J_{p,q} = -\sin(\theta)$ ,  $J_{q,p} = \sin(\theta)$ 
7:    $A = JAJ^T$ 
8: end while
9:  $\Lambda = \text{diag}(A)$ 

```

---

Specifically, it describes the cyclical variant of the Jacobi algorithm. While the classical approach would provide faster convergence properties, the cyclical variant is used. As described in Section 3.4.1, the search for the highest off-diagonal value requires  $\frac{1}{2}n(n-1)$  operations. For a parallel implementation, this has to be performed up to  $\lfloor \frac{n}{2} \rfloor$  times, for maximum parallelisation, while also ensuring no duplicates or interfering rows and columns are indexed (the parallel set  $\{(1, 2), (2, 4)\}$ , for instance). Upholding these criteria, can result in indexing values which are not the highest possible. This will result in a worse convergence, which will theoretically still be faster than pure cyclical. The search for these values will however take a considerable amount of time and area in the final hardware implementation. As such, cyclical Jacobi lends itself to a parallel implementation far more than classical Jacobi.

An algorithm taking advantage of this parallelism is described in Algorithm 2. To complement the pseudocode, Figure 3.6 visualises the distinct steps in the algorithm. Algorithm 2 is derived from Algorithm 1 by exploiting the inherent parallelism in the Jacobi algorithm as described in Section 3.6. This inherent parallelism only applies when performing multiple rotations on unique, non-conflicting sets of rows and columns. The Jacobi rotation matrix  $J$  can be constructed for multiple pairs of  $p$  and  $q$ , before being applied to the input matrix  $A$ . The parallel for loop from line 5 to 8 shows the construction of the  $J$  matrix. Once  $J$  has been constructed for one whole set of indices, it is applied to  $A$  in line 9. A new set of feasible indices is generated in line 4 and another  $J$  matrix is constructed.

---

**Algorithm 2** Parallel Cyclical Jacobi Algorithm

---

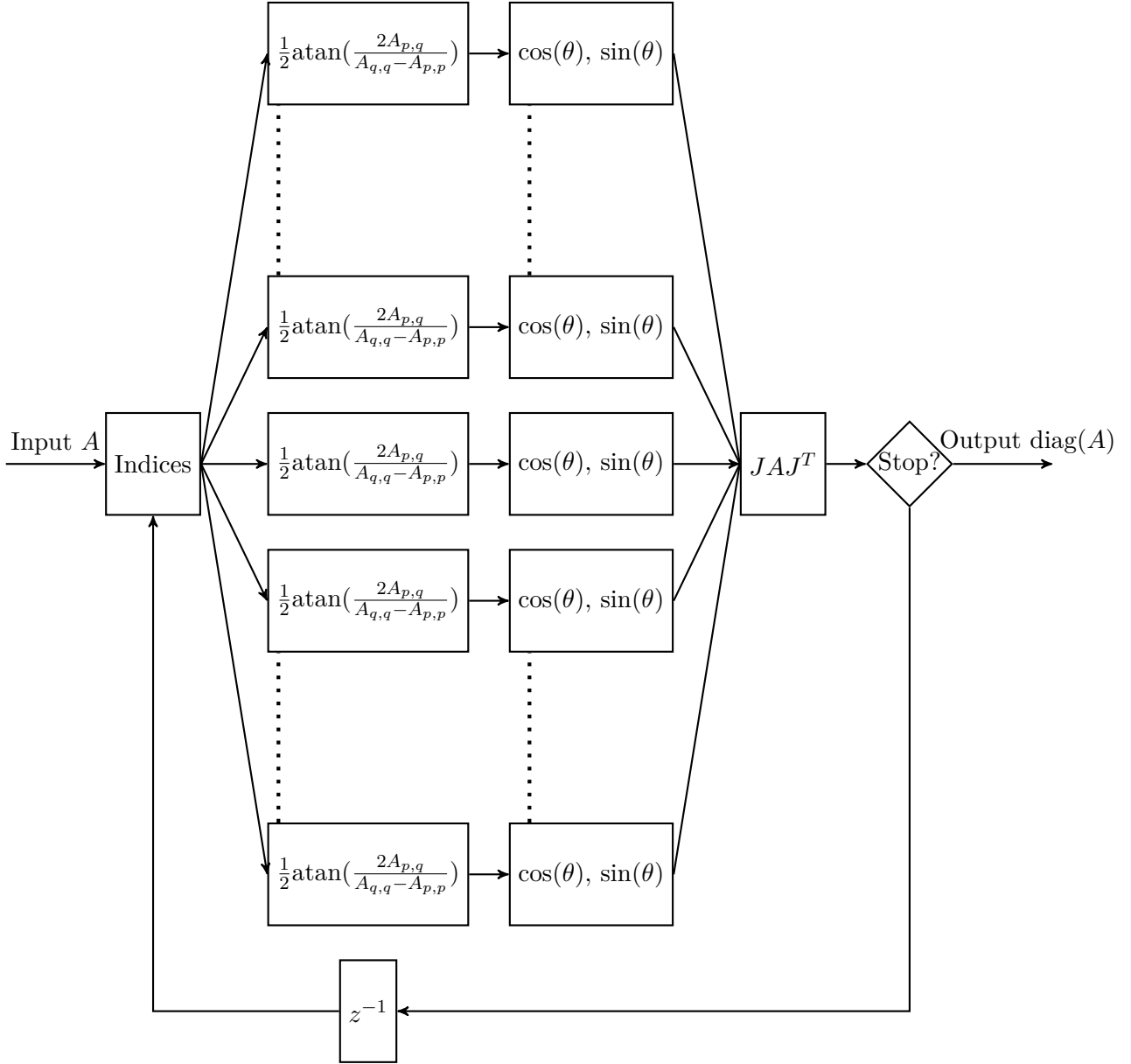
```

1: Accept a symmetric matrix  $A$  as input
2: while stopping criteria is not fulfilled do
3:    $J = I$ 
4:    $M = \text{new set of feasible indices } (p, q)$ 
5:   for each  $(p, q) \in M$  do in parallel
6:      $\theta = \frac{1}{2} \arctan\left(\frac{2A_{p,q}}{A_{q,q} - A_{p,p}}\right)$ 
7:      $J_{p,p} = \cos(\theta)$ ,  $J_{q,q} = \cos(\theta)$ ,  $J_{p,q} = -\sin(\theta)$ ,  $J_{q,p} = \sin(\theta)$ 
8:   end for
9:    $A = JAJ^T$ 
10: end while
11:  $\Lambda = \text{diag}(A)$ 

```

---

### 3.8. Numerical Example



**Figure 3.6:** Block diagram associated with the algorithm described by Algorithm 2.

### 3.8 Numerical Example

In this section, a numerical example of the parallel cyclical Jacobi eigenvalue method is shown, applied to a  $4 \times 4$  matrix. In this case, it converges to diagonal form in one sweep. Intermediary calculations are also shown on the right hand side, including the pairs rotations will be performed on  $(p_n, q_n)$ , the values of  $\cos(\theta_n)$  and  $\sin(\theta_n)$  used in the rotation matrix, and the Frobenius norm of the off diagonal entries.

### 3.8. Numerical Example

$$\begin{aligned}
A^{[0]} &= \begin{bmatrix} 1.112 & \boxed{1.061} & 1.407 & 1.235 \\ \boxed{1.061} & 1.243 & 1.403 & 1.292 \\ 1.407 & 1.403 & 2.119 & \boxed{1.861} \\ 1.235 & 1.292 & \boxed{1.861} & 1.712 \end{bmatrix} & \begin{aligned} p_1, q_1 &= (1, 2), p_2, q_2 = (3, 4) \\ \cos(\theta_1) &= 0.729, \sin(\theta_1) = 0.685 \\ \cos(\theta_2) &= 0.745, \sin(\theta_2) = -0.668 \\ \|E^{[0]}\|_F &= 4.844 \end{aligned} \\
A^{[1]} &= \begin{bmatrix} 0.114 & 0 & 0.057 & \boxed{-0.031} \\ 0 & 2.240 & \boxed{2.671} & 0.004 \\ 0.057 & \boxed{2.671} & 3.787 & 0 \\ \boxed{-0.031} & 0.004 & 0 & 0.043 \end{bmatrix} & \begin{aligned} p_1, q_1 &= (1, 4), p_2, q_2 = (2, 3) \\ \cos(\theta_1) &= 0.934, \sin(\theta_1) = 0.357 \\ \cos(\theta_2) &= 0.799, \sin(\theta_2) = 0.601 \\ \|E^{[1]}\|_F &= 3.780 \end{aligned} \\
A^{[2]} &= \begin{bmatrix} 0.126 & -0.033 & \boxed{0.041} & 0 \\ -0.033 & 0.232 & 0 & \boxed{-0.008} \\ \boxed{0.041} & 0 & 5.795 & 0.019 \\ 0 & \boxed{-0.008} & 0.019 & 0.031 \end{bmatrix} & \begin{aligned} p_1, q_1 &= (1, 3), p_2, q_2 = (4, 2) \\ \cos(\theta_1) &= 1.000, \sin(\theta_1) = 0.007 \\ \cos(\theta_2) &= 0.999, \sin(\theta_2) = -0.043 \\ \|E^{[2]}\|_F &= 0.0817 \end{aligned} \\
A^{[3]} &= \begin{bmatrix} 0.126 & \boxed{-0.033} & 0 & -0.001 \\ \boxed{-0.033} & 0.233 & -0.001 & 0 \\ 0 & -0.001 & 5.795 & \boxed{0.019} \\ -0.001 & 0 & \boxed{0.019} & 0.030 \end{bmatrix} & \begin{aligned} p_1, q_1 &= (1, 2), p_2, q_2 = (3, 4) \\ \cos(\theta_1) &= 0.961, \sin(\theta_1) = -0.278 \\ \cos(\theta_2) &= 1.000, \sin(\theta_2) = -0.003 \\ \|E^{[3]}\|_F &= 0.054 \end{aligned} \\
A^{[4]} &= \begin{bmatrix} 0.116 & 0 & 0 & \boxed{-0.001} \\ 0 & 0.242 & \boxed{-0.001} & 0 \\ 0 & \boxed{-0.001} & 5.795 & 0 \\ \boxed{-0.001} & 0 & 0 & 0.030 \end{bmatrix} & \begin{aligned} p_1, q_1 &= (1, 4), p_2, q_2 = (2, 3) \\ \cos(\theta_1) &= 1.000, \sin(\theta_1) = 0.018 \\ \cos(\theta_2) &= 1.000, \sin(\theta_2) = 0.000 \\ \|E^{[4]}\|_F &= 0.002 \end{aligned} \\
A^{[5]} &= \begin{bmatrix} 0.116 & 0 & 0 & 0 \\ 0 & 0.242 & 0 & 0 \\ 0 & 0 & 5.795 & 0 \\ 0 & 0 & 0 & 0.030 \end{bmatrix} & \begin{aligned} \text{Matrix diagonalisation complete} \\ \|E^{[5]}\|_F &= 0 \end{aligned}
\end{aligned}$$

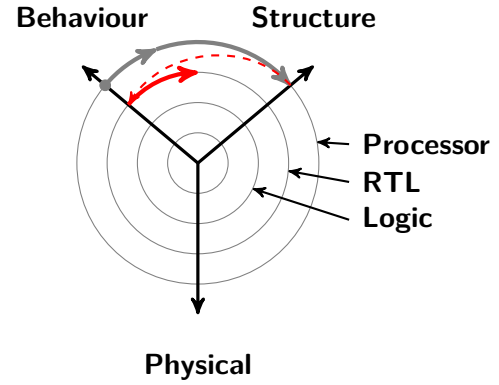
# Chapter 4

## Algorithm Analysis

In Chapter 3, the mathematical aspects of the Jacobi eigenvalue algorithm were discussed. In this chapter, more practical factors regarding its implementation are discussed, focusing more towards its implementation in hardware. These include choices and considerations regarding the sub-algorithms used as part of the computation, where variations of Algorithm 2 that can simplify its implementation are explored, culminating in Algorithm 3.

Finally, how the word length affects the accuracy of the eigenvalues is investigated, providing a connection between a strictly theoretical and a more practical approach to hardware design.

In the following sections we expand upon our description of the structural domain and derive the functionality and behavioural requirements for the sub-algorithms. These sub-algorithms will form the basis for the design of the RTL components. Specifically, we first introduce the computation of the inverse tangent. Afterwards we look into the rotation of rows and columns in the matrix. For both of these problems, it is decided that a CORDIC block will be used. The CORDIC is described behaviourally, with some considerations as to the structure of it.



### 4.1 Computation of the Arctangent

Computing the arctangent is required for finding the rotation angle as derived in Equation (3.44). There are multiple methods for performing this computation, which will be briefly covered before selecting one for use in the final design. The selection will be based on the cost function mentioned in Section 2.1, with computational speed being of particular significance.

A simple method of "computing" the arctangent, is to store a pre-calculated set of outputs of the arctangent function in a look-up table (LUT). This enables quick access to a desired value; however, the accuracy of the result depends entirely on the resolution of the stored values. A higher resolution or a longer word length of the output will require more memory to store either more values, or more bits per value respectively, which results in less area available for other computational operators. Similarly, to avoid evaluating the division  $(\frac{2A_{p,q}}{A_{q,q}-A_{p,p}})$ , a LUT will be required to have a dimension for both the numerator and denominator, further increasing its size.

Another method of computing the arctangent is to approximate the function within an interval. The accuracy of the result depends on the approximation, with better accuracy obtained with more mathematically complex approximations. A study on such approximations has been conducted in [29], which details both simple and more complex approximations of the arctangent function.

A method not mentioned directly in [29] involves the approximation of the arctangent function by the

#### 4.1. Computation of the Arctangent

Maclaurin series. The Maclaurin series is defined as

$$f(x) \approx f(0) + \sum_{n=0}^N \frac{f^{(n)}(0)}{n!} x^n, \quad (4.1)$$

where  $f(x)$  is the function to be approximated,  $N$  is the order of the approximation, and superscript  $(n)$  denotes the  $n$ th derivative of a function. The Maclaurin expansion of the arctangent function is [30]

$$\arctan(x) \approx x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots \quad (4.2)$$

The computational complexity of this approximation is rather high, considering the requirement for computing  $x^n$ , which requires at least  $n$  multiplications. This same argumentation also follows to the other polynomial approximations presented in [29], but as division by a variable is included in the rational approximations they used, its computational complexity must also be considered.

In addition to the computational complexity of the approximations of  $\arctan(x)$ , the computational complexity of finding  $x$  in the Jacobi algorithm is also necessary to consider. From Equation (3.44) it can be seen that

$$x = \frac{2A_{p,q}^{[k]}}{A_{q,q}^{[k]} - A_{p,p}^{[k]}}, \quad (4.3)$$

which requires a division by a variable. This is the same problem encountered in rational approximations, as stated above, and so the same argumentation and line of reasoning applies. While this can be implemented in hardware, it has high computational complexity due to the iterative nature of the algorithms used. [31, p. 109].

The Coordinate Rotation Digital Computer (CORDIC) algorithm [32] can be used to eliminate the need for this division, when used in a vectoring configuration [33, p.135][34]. This is due to the manner in which CORDIC operates in vectoring mode, iteratively rotating an input vector onto the  $x_1$ -axis by use of a process similar to binary search. Once the input vector lies on the  $x_1$ -axis, the sum of the rotations is then the result of  $\arctan\left(\frac{x_2}{x_1}\right)$ . This will be explored further in Section 4.3, and an example of its operation given in Section 4.3.4. In this case, the input vector is

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} A_{q,q}^{[k]} - A_{p,p}^{[k]} \\ 2A_{p,q}^{[k]} \end{bmatrix}. \quad (4.4)$$

In addition to eliminating the need for division, the CORDIC algorithm can be implemented with only add and shift operations. As such, the CORDIC algorithm is chosen for use in the overall design. The CORDIC algorithm will be explained in detail mathematically, algorithmically, and graphically in Section 4.3.



## 4.2 Row and Column Rotation

Once the rotation angles have been determined, the rows and columns of input matrix,  $A$ , must be rotated. Based on the mathematics presented in Chapter 3, the most obvious approach would be to form the sine and cosine of the determined angles, and then use these to explicitly calculate the two matrix multiplication in the similarity transformation  $JAJ^T$ . While this may sound like a daunting task, it must be emphasised that as most of the elements in the Jacobi rotation matrix are set equal to zero,  $J$  is a sparse matrix. This suggests that it is not necessary to perform full matrix multiplication, but that the sparse nature of  $J$  can be used to determine a set of equivalent and computationally simpler expressions for the resulting matrix formed by the  $JAJ^T$  operations. These equations have already been presented in Equation (3.38) to Equation (3.43) in Chapter 3. For the sake of readability, these equations are restated below:

$$A_{p,q}^{[k+1]} = A_{q,p}^{[k+1]} = (\eta^2 - \sigma^2)A_{p,q}^{[k]} + (A_{p,p}^{[k]} - A_{q,q}^{[k]})\eta\sigma \quad (4.5)$$

$$A_{p,i}^{[k+1]} = A_{i,p}^{[k+1]} = \eta A_{p,i}^{[k]} - \sigma A_{q,i}^{[k]} \quad \text{for } \forall i \neq p, q \quad (4.6)$$

$$A_{q,i}^{[k+1]} = A_{i,q}^{[k+1]} = \eta A_{q,i}^{[k]} + \sigma A_{p,i}^{[k]} \quad \text{for } \forall i \neq p, q \quad (4.7)$$

$$A_{p,p}^{[k+1]} = \eta^2 A_{p,p}^{[k]} + \sigma^2 A_{q,q}^{[k]} - 2\eta\sigma A_{p,q}^{[k]} \quad (4.8)$$

$$A_{q,q}^{[k+1]} = \eta^2 A_{q,q}^{[k]} + \sigma^2 A_{p,p}^{[k]} + 2\eta\sigma A_{p,q}^{[k]} \quad (4.9)$$

$$A_{i,j}^{[k+1]} = A_{i,j}^{[k]} \quad \text{for } \forall i \neq p, q \quad \forall j \neq p, q \quad (4.10)$$

For every entry which is to be zeroed, circuitry is needed to find the sine and cosine of an angle, following which this  $JAJ^T$  must be calculated. While the sparsity does allow for relatively efficient computation of the similarity transformation, it is observed that the number of hardware multipliers required for a sequential implementation will be proportional to  $2n$ .

Considering now that these equations Equation (4.5) to Equation (4.10) only account for zeroing one index at a time, From Section 3.6 it is evident that it is possible to zero up to  $\frac{n}{2}$  entries concurrently. Thus, the total number of required multiplications, and hence multipliers for full parallelism, will be proportional to  $n^2$ . While this might be acceptable in certain hardware solutions, the large number of multipliers will consume a significant amount of chip area.

Another approach involves recalling from Section 3.2 that the similarity transformation  $JAJ^T$ , is equivalent to first rotating all the vectors  $[A_{p,i}, A_{q,i}]^T$ ,  $i \in [1, n]$ , by an angle specified by  $\theta$ . Following the rotation of the row elements, the same procedure is performed on the elements in column  $p$  and  $q$ . This concept can be seen in Equation (3.24), where an accompanying example is given below. The advantages of this method are that the vector rotation can be performed using CORDIC and thus does not require any multipliers. Additionally, it is not necessary to explicitly calculate the sine and cosine of the angle. Thereby providing the possibility for additional area and time saving. Additionally, the use of CORDIC for both calculation of the arctangent and application of the similarity transform significantly simplifies the design process, while also opening new potential for hardware sharing. Due to the possibility to omit many multipliers, the possibility to avoid direct computation of sine and cosine, and the potential simplifications and hardware sharing, this is the approach which will be used.

## 4.3 CORDIC Algorithm

This section will give an insight into the CORDIC algorithm in both rotation and vectoring modes, showing relevant derivations and examples. Later in Chapter 6, this subalgorithm will be mapped

### 4.3. CORDIC Algorithm

into relevant hardware while considering the cost function mentioned in Section 2.1.

#### 4.3.1 Mathematical Overview of CORDIC Algorithm

Rotating a vector by  $\theta$  in Cartesian coordinates can be achieved by use of matrix operations. A two-dimensional rotation matrix is of the form

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}. \quad (4.11)$$

Given a vector  $x = [x_1, x_2]^T$ , the rotated vector  $y = [y_1, y_2]^T$  is found by

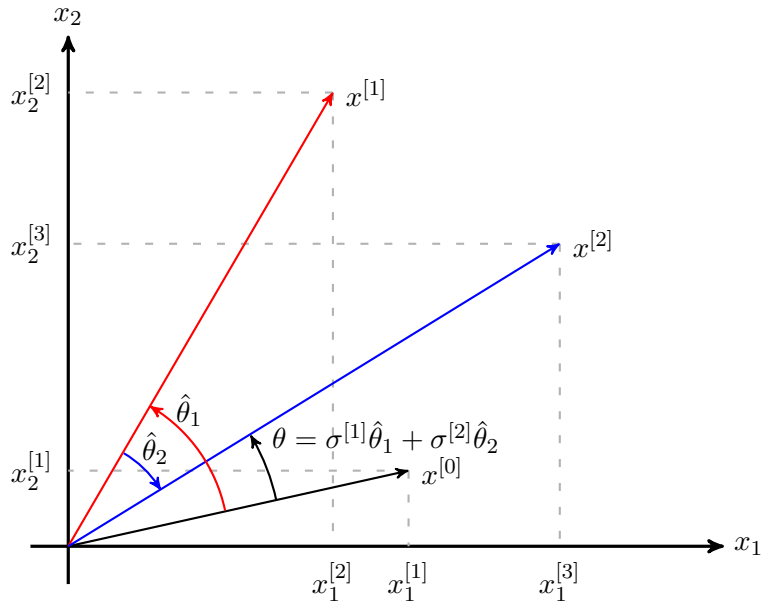
$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad (4.12)$$

or expressed without matrix notation

$$y_1 = x_1 \cdot \cos(\theta) - x_2 \cdot \sin(\theta) \quad (4.13)$$

$$y_2 = x_2 \cdot \cos(\theta) + x_1 \cdot \sin(\theta). \quad (4.14)$$

When CORDIC is used to rotate a vector by an input angle, it is referred to as *rotation mode* CORDIC. The foundation of the CORDIC algorithm is to rotate the vector in an iterative manner using predefined angles, until the desired rotation angle is reached. Before a formal description of the CORDIC algorithm, the intuition behind it is explained.



**Figure 4.1:** The principle of iterative rotations, showing how  $x$  is rotated by an angle  $\theta$ . First,  $x$  is rotated by  $\hat{\theta}_1$  resulting in  $x^{[1]}$  during the first iteration shown in red. As this is greater than  $\theta$ , it is then rotated in the opposite direction by  $\hat{\theta}_2$  as shown in blue, forming  $x^{[2]}$ , which is the desired angle of rotation.

### 4.3. CORDIC Algorithm

Rotations can be achieved through multiple smaller vector rotations by use of a binary search. This is exemplified in Figure 4.1. By accounting for the symmetry of the sine and cosine functions, it is possible to limit the domain of this rotation algorithm to  $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ . This means that the first guess the algorithm makes is to rotate by  $\frac{\pi}{4}$  radians. The next iterations rotation direction is now determined by the difference between desired angle and rotated angle. The next rotation is applied with half the rotation angle of the previous. The angles are defined as

$$\hat{\theta}^{[i]} = \frac{\pi}{4} \cdot 2^{-i} = \left[ \frac{\pi}{4}, \frac{\pi}{8}, \frac{\pi}{16}, \dots \right], \quad (4.15)$$

As an example, to iteratively rotate a point  $56^\circ$ , requires a rotation of  $\frac{\pi}{4} + \frac{\pi}{8} - \frac{\pi}{16} = \frac{5\pi}{16} = 56.25^\circ$ . This means that after three iterations the error is only  $0.25^\circ \approx 0.0044$  radians, which is somewhat close to the true value, considering the required operations. However, it must be noted that the error is not monotonically decreasing, meaning that there is the possibility of the error increasing with further iterations, as it will in fact in the case of the aforementioned example. The next iteration of the example would give  $\frac{\pi}{4} + \frac{\pi}{8} - \frac{\pi}{16} - \frac{\pi}{32} = \frac{9\pi}{32} = 50.625^\circ$  which is an error of  $5.375^\circ$ . The following iterations would however reduce this error, and so a balance between accuracy and speed must be decided upon when formulating a stopping criteria, with more iterations generally giving higher accuracy, but at the expense of time.

Now, the steps for one iteration of the algorithm can now be conceived from Equation (4.13), (4.14) and (4.15) as

$$x_1^{[i+1]} = x_1^{[i]} \cdot \cos(\hat{\theta}^{[i]}) - \sigma^{[i]} \cdot x_2^{[i]} \cdot \sin(\hat{\theta}^{[i]}) \quad (4.16)$$

$$x_2^{[i+1]} = x_2^{[i]} \cdot \cos(\hat{\theta}^{[i]}) + \sigma^{[i]} \cdot x_1^{[i]} \cdot \sin(\hat{\theta}^{[i]}), \quad (4.17)$$

where  $\sigma^{[i]}$  is used to specify clockwise or counter-clockwise rotation, for iteration  $i$ .

Note that these values of  $\cos(\hat{\theta}^{[i]})$  and  $\sin(\hat{\theta}^{[i]})$  can be pre-calculated at compile time and stored in a look-up table as

$$\cos_{lut}(i) = \cos(\hat{\theta}^{[i]}) \quad (4.18)$$

$$\sin_{lut}(i) = \sin(\hat{\theta}^{[i]}),$$

where  $i$  denotes the current iteration number and  $\cos_{lut}(i)$  and  $\sin_{lut}(i)$  denotes a saved look-up table entry indexed by  $i$ . When these are combined with...

$$x_1^{[i+1]} = x_1^{[i]} \cdot \cos_{lut}(i) - \sigma^{[i]} \cdot x_2^{[i]} \cdot \sin_{lut}(i) \quad (4.19)$$

$$x_2^{[i+1]} = x_2^{[i]} \cdot \cos_{lut}(i) + \sigma^{[i]} \cdot x_1^{[i]} \cdot \sin_{lut}(i), \quad (4.20)$$

To keep track of how close to the desired rotation the algorithm is,  $\theta$  is updated as

$$\theta^{[i+1]} = \theta^{[i]} - \sigma^{[i]} \hat{\theta}^{[i]} \quad (4.21)$$

where  $\theta^{[0]}$  is equal to the desired rotation angle.

### 4.3. CORDIC Algorithm

Finally, to actually determine whether to rotate clockwise or counterclockwise,  $\sigma$  is updated based on the current value of  $\theta^{[i]}$

$$\sigma^{[i]} = \begin{cases} 1, & \theta^{[i]} \geq 0 \\ -1, & \theta^{[i]} < 0. \end{cases} \quad (4.22)$$

The algorithm terminates when  $\theta^{[i+1]}$  is sufficiently close to zero, or a maximum number of iterations,  $n$ , is reached.

These algorithm steps are simple multiplications and additions/subtractions, which while not ideal in its current form, the current rotation algorithm can be further developed to simplify these hardware requirements. In this further development below, this above intuition is extended to derive the CORDIC algorithm.

To reduce the number of required multiplications, the rotation matrix can be modified using the identities [33, p. 135]

$$\cos(\theta) = \frac{1}{\sqrt{1 + \tan^2(\theta)}} \quad (4.23)$$

and

$$\sin(\theta) = \frac{\tan(\theta)}{\sqrt{1 + \tan^2(\theta)}}. \quad (4.24)$$

The new rotation matrix becomes

$$\frac{1}{\sqrt{1 + \tan^2(\theta)}} \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix}, \quad (4.25)$$

for each iteration. At a first glance the complexity has increased, however this can be simplified yet further by considering the scalar that has been factorised out of the matrix. With the values for the arctangent at each iteration being defined in Equation (4.29), it is possible to also calculate a value for  $\frac{1}{\sqrt{1 + \tan^2(\theta)}}$  in compile time. Furthermore with successive rotations, although the angle changes, this is always a scalar outside of the rotation matrix. This allows it to be computed as a product, using the values from Equation (4.29) to create an additional look-up table [33, p. 137]

$$k_{lut}(i) = \prod_{j=0}^i \frac{1}{\sqrt{1 + \tan^2(\hat{\theta}^{[j]})}} \quad (4.26)$$

The relevant one for the given number of iterations of CORDIC is then multiplied only once after the final rotation is performed. Finding the rotated vector for each iteration then becomes

$$x_1^{[i+1]} = x_1^{[i]} - \sigma^{[i]} \cdot x_2^{[i]} \cdot \tan_{lut}(i) \quad (4.27)$$

$$x_2^{[i+1]} = x_2^{[i]} + \sigma^{[i]} \cdot x_1^{[i]} \cdot \tan_{lut}(i), \quad (4.28)$$

### 4.3. CORDIC Algorithm

where  $\tan_{lut}(i)$  is a look-up table for the values of  $\tan(\hat{\theta}^{[i]})$ . These algorithm steps only require two multiplications per iteration as well as the k-factor multiplication at the end.

Next, to further reduce the number of multiplications necessary, the multiplication by  $\tan_{lut}(i)$  is replaced with a shift operation. To do so, the predefined rotation angles are changed to ensure that

$$\tan_{lut}(i) = \tan(\hat{\theta}^{[i]}) = 2^{-i}, \quad i \in [1, n]. \quad (4.29)$$

This means that the predefined rotation angles (in radians) become

$$\arctan(2^{-i}) = \hat{\theta}^{[i]} = [0.78540, 0.46365, 0.24498, 0.12435, \dots, \arctan(2^{-(n-1)})]. \quad (4.30)$$

The update steps for the vector coordinates then becomes

$$x_1^{[i+1]} = x_1^{[i]} - \sigma^{[i]} \cdot x_2^{[i]} \gg i \quad (4.31)$$

$$x_2^{[i+1]} = x_2^{[i]} + \sigma^{[i]} \cdot x_1^{[i]} \gg i \quad (4.32)$$

$$\sigma^{[i+1]} = \begin{cases} 1, & \theta^{[i+1]} \geq 0 \\ -1, & \theta^{[i+1]} < 0. \end{cases} \quad (4.33)$$

Where  $\gg$  denotes a shift operation. The k-factors also have to be changed to accommodate the shift operation. This can simply be expressed as

$$k_{lut}(i) = \prod_{j=0}^i \frac{1}{\sqrt{1 + 2^{-2j}}}. \quad (4.34)$$

This allows the CORDIC algorithm to rotate any vector up to  $\sum_{i=0}^n \hat{\theta}^{[i]} \approx 1.74$  radians, although this is by convention approximated to the domain  $\pm \frac{\pi}{2}$ . However, recall from Chapter 3 that the maximum rotation angle required in the Jacobi algorithm is  $\pm \frac{\pi}{4}$  radians. This is half as much as what the previously presented CORDIC is capable of. Thus, an exact implementation of the previous derivation would result in an over-engineered solution. While this does not alter the validity of the results, it does result in redundant computation. To optimise the method with respect to the smaller angle constraints, it is observed that initial rotation can be omitted. With this modification the maximum rotation range is reduced to  $\sum_{i=1}^n \hat{\theta}^{[i]} \approx 0.95$  radians. As this is greater than the required  $\frac{\pi}{4}$  radians, this altered CORDIC is sufficient for our purposes. Note that the omission of the initial rotation requires the k-factors to be altered to

$$k_{lut}(i) = \prod_{j=1}^i \frac{1}{\sqrt{1 + 2^{-2j}}}. \quad (4.35)$$

This concludes the description of CORDIC in rotation mode. Although the iterative nature of this vector rotation does not seem efficient at a glance, it compares favourably to the sparse matrix multiplication method found in Section 4.2, as the rotation does not require any variable multiplications. Additionally, the rotations can be applied directly to the rows and columns of  $A$ , rather than first calculating the sine and cosine.

### 4.3. CORDIC Algorithm

#### 4.3.2 Computing the Arctangent Using CORDIC

This subsection considers the use of CORDIC to iteratively rotate a given vector  $x$  until  $x_2 = 0$ , and return the angle used for this rotation. When concerned only about the angle rotated through, a scaling operation as in Equation (4.26) is not necessary. This is referred to as *vectoring mode* [33, p. 138]. As mentioned in Section 4.1, CORDIC can be used for calculating the inverse tangent function. This is done by setting the input vector to

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} A_{q,q}^{[k]} - A_{p,p}^{[k]} \\ 2A_{p,q}^{[k]} \end{bmatrix},$$

as stated in Section 4.1 in Equation (4.4), and rotating this vector until  $x_2 \approx 0$ .

The reason this is possible is due to the definition of the tangent function and how CORDIC works. The definition of the tangent function is

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)} \quad (4.36)$$

or

$$\tan(\theta) = \frac{\text{opposite}}{\text{adjacent}}, \quad (4.37)$$

for right-angled triangles.

For any vector, the opposite and adjacent lengths are merely the Cartesian coordinates for the vector:

$$\text{adjacent} = x_1 \quad (4.38)$$

$$\text{opposite} = x_2, \quad (4.39)$$

which means the tangent can be defined as

$$\tan(\theta) = \frac{x_2}{x_1}. \quad (4.40)$$

Taking the inverse tangent on both sides gives

$$\arctan(\tan(\theta)) = \arctan\left(\frac{x_2}{x_1}\right) \Leftrightarrow \theta = \arctan\left(\frac{x_2}{x_1}\right). \quad (4.41)$$

Since CORDIC will iteratively rotate the vector  $x$  until  $x_2 \approx 0$ , the division will never be computed explicitly.

The rotation direction will be determined by the quadrant that the vector is in. Thus it is dependant of the sign of  $x_1$  and  $x_2$ . Specifically,  $\sigma^{[i]}$  is found by

$$\sigma^{[i]} = \begin{cases} -1, & x_1^{[i]} x_2^{[i]} \geq 0 \\ 1, & x_1^{[i]} x_2^{[i]} < 0 \end{cases}, \quad (4.42)$$

### 4.3. CORDIC Algorithm

since the rotation direction is towards the x-axis.

The result for Equation (3.44), restated here for convenience as

$$\theta = \frac{\arctan\left(\frac{2A_{p,q}^{[k]}}{A_{q,q}^{[k]} - A_{p,p}^{[k]}}\right)}{2} \quad (4.43)$$

then becomes

$$\theta = \frac{1}{2} \sum_{i=0}^n -\sigma^{[i]} \hat{\theta}^{[i]}, \quad (4.44)$$

where  $n$  is the number of CORDIC rotations applied until  $x_2 \approx 0$ , where the iterative part of the algorithm is terminated. As already discussed, multiplying by  $k_{lut}(n)$  is not necessary, since the length of the vector after rotation is not of any consequence, only the angle it has been rotated through.

#### 4.3.3 CORDIC Accuracy

As the CORDIC algorithm does not give an exact solution but instead an approximation, it is important that the accuracy of this approximation corresponds with the accuracy of the system as a whole, in order not to introduce any undue sources of error. In [35], it is stated that both the internal word length and number of iterations are of importance when determining the accuracy of the approximation. An excerpt from their findings based on CORDIC in rotation mode is presented below in Table 4.1. This is then used to determine an appropriate internal word length and number of iterations for the desired word length of the architecture. This architectural word length is further discussed in Section 4.6. For the purposes of guiding further design, the predicted value is sufficient as a worst-case estimate as it is always lower than the simulated value.

Further to these results, it has been observed around and above Equation (4.35) that one iteration can be omitted due to a domain reduction. This is not shown in Table 4.1, but is accounted for when setting these parameters in Section 4.6

As a full error analysis of CORDIC in vectoring mode is out of the scope of this report, it is assumed for sake of simplicity that the results presented in [35] for CORDIC in rotation mode are also valid for CORDIC in vectoring mode.

**Table 4.1:** Predicted number of significant bits for number of iterations  $n$  and internal word length  $b$  [35].

$\begin{matrix} \backslash & b \\ n \end{matrix}$	19	20	21	22	23
17	14.21	14.72	15.05	15.26	15.37
18	14.49	15.17	15.68	16.03	16.25
19	14.62	15.44	16.13	16.65	17.01
20	14.66	15.56	16.38	17.08	17.62
21	14.65	15.60	16.51	17.33	18.04
22	14.61	15.59	16.54	17.45	18.29
23	14.57	15.55	16.53	17.49	18.40

### 4.3. CORDIC Algorithm

#### 4.3.4 Numerical CORDIC Example

To further cement the behaviour of the CORDIC algorithm, a short numerical example is given. In this example, the Jacobi rotation angle is desired for the matrix

$$A = \begin{bmatrix} -0.25 & 1.5 & 2 \\ 1.5 & 0.75 & 7 \\ 2 & 7 & 12 \end{bmatrix}, \quad (4.45)$$

where  $p = 1$  and  $q = 2$ .

When calculating the arctangent of a vector  $[x_1, x_2]^T$  using CORDIC, the first step is to set the input vector to

$$\begin{bmatrix} x_1^{[0]} \\ x_2^{[0]} \end{bmatrix} = \begin{bmatrix} A_{q,q}^{[k]} - A_{p,p}^{[k]} \\ 2A_{p,q}^{[k]} \end{bmatrix} = \begin{bmatrix} 0.75 - (-0.25) \\ 2 \cdot 1.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}. \quad (4.46)$$

Next, sigma is set according to Equation (4.42), which in this case means  $\sigma^{[0]} = -1$ , since  $x_2^{[0]} \geq 0$ . Using Equation (4.31) and Equation (4.32), the first CORDIC iteration is

$$x_1^{[1]} = x_1^{[0]} - \sigma^{[0]} \cdot x_2^{[0]} \gg 0 = 1 - (-1 \cdot 3 \gg 0) = 4 \quad (4.47)$$

$$x_2^{[1]} = x_2^{[0]} + \sigma^{[0]} \cdot x_1^{[0]} \gg 0 = 3 + (-1 \cdot 1 \gg 0) = 2. \quad (4.48)$$

Before the next iteration can be carried out,  $\sigma^{[1]} = -1$ , since  $x_2^{[1]} \geq 0$ . The next iteration is then

$$x_1^{[2]} = x_1^{[1]} - \sigma^{[1]} \cdot x_2^{[1]} \gg 1 = 4 - (-1 \cdot 2 \gg 1) = 5 \quad (4.49)$$

$$x_2^{[2]} = x_2^{[1]} + \sigma^{[1]} \cdot x_1^{[1]} \gg 1 = 2 + (-1 \cdot 4 \gg 1) = 0. \quad (4.50)$$

Since  $x_2^{[2]} = 0$ , the algorithm computes  $\theta$  according to Equation (4.44), with  $n = 2$ , and terminates. The results (in radians) becomes

$$\theta = \frac{1}{2} \sum_{i=0}^n -\sigma^{[i]} \hat{\theta}^{[i]} = \frac{1}{2} (0.78540 + 0.46365) = 0.62453 \quad (4.51)$$

Note that further iterations of the algorithm are not necessary, as the vector has already been rotated to  $x_2 = 0$ . In a similar way to stopping conditions for CORDIC in rotation mode, CORDIC in vectoring mode can similarly be stopped when it is sufficiently close as deemed by the user.

In this particular case, the exact value computed by numpy is 0.62452, leaving an absolute error of  $1 \times 10^{-5}$ . However if a further iteration is applied, by Equation (4.42), with  $\sigma = -1$ , leading to

$$x_1^{[3]} = x_1^{[2]} - \sigma^{[2]} \cdot x_2^{[2]} \gg 1 = 5 - (-1 \cdot 0 \gg 2) = 5 \quad (4.52)$$

$$x_2^{[3]} = x_2^{[2]} + \sigma^{[2]} \cdot x_1^{[2]} \gg 1 = 0 + (-1 \cdot 5 \gg 2) = -1.25 \quad (4.53)$$



#### 4.4. Algorithm to be Implemented

With  $\theta$  updated for  $n = 3$  as

$$\theta = \frac{1}{2} \sum_{i=0}^n -\sigma^{[i]} \hat{\theta}^{[i]} = \frac{1}{2} (0.78540 + 0.46365 + 0.24498) = 0.74702 \quad (4.54)$$

leaving an absolute error of 0.12246. This result further shows how the error does not decrease monotonically. By further iterations, this comparatively large error will be significantly reduced, but it will never be zero again. This, and similar scenarios can however be eliminated by stopping the algorithm when  $x_2$  is sufficiently close to zero.

## 4.4 Algorithm to be Implemented

The previous sections exclusively considered the algorithmic implementations of the Jacobi algorithms mathematical operators. The choice of stopping criteria has also been determined in Section 3.5, where a fixed number of sweeps dependent on the matrix size was selected.

The mathematical aspects of the Jacobi Eigenvalue algorithm and its respective sub-algorithms have now been thoroughly explored, enabling further work towards a hardware implementation. Algorithm 2 is then updated below to account for the changes that have been made so far.

---

#### Algorithm 3 Parallel Cyclical Jacobi Algorithm

---

```

1: Accept a symmetric matrix  $A$  as input
2: counter = 0
3: while counter <  $\left\lceil \epsilon_{stop} \frac{n(n-1)}{2} \log_2(n) \right\rceil$  do
4:    $J = I$ 
5:    $M =$  new set of feasible indexes  $(p, q)$ 
6:   for each  $(p, q) \in M$  do in parallel
7:      $\theta = \frac{1}{2} \arctan\left(\frac{2A_{p,q}}{A_{q,q} - A_{p,p}}\right)$ 
8:     for all  $i$  from 1 to  $n$  do in parallel
9:       Rotate  $\begin{bmatrix} A_{p,i} \\ A_{q,i} \end{bmatrix}$  by  $\theta$ 
10:    end for
11:    for all  $i$  from 1 to  $n$  do in parallel
12:      Rotate  $\begin{bmatrix} A_{i,p} \\ A_{i,q} \end{bmatrix}$  by  $\theta$ 
13:    end for
14:  end for
15:  counter +=  $\lfloor \frac{n}{2} \rfloor$ 
16: end while
17:  $\Lambda = \text{diag}(A)$ 

```

---

The algorithm still lacks details regarding the generation of the parallel  $(p, q)$  pairs, however it is possible to implicitly find these through data transfers. Therefore an explicit hardware implementation of such an algorithm is not required, and thus further elaboration on this topic has been delegated to Chapter 5.

## 4.5 Avoiding Overflow

As the architectural design will utilise a fixed-point word length, it is important to consider the possibility of overflow and how to avoid it. To determine when overflow can occur, the algorithm as a whole as well as each of its individual operations must be analysed.

The key to this analysis is to recall that similarity transformations preserves the Frobenius norm of the input matrix. Thus, in cases where the input matrix only has one eigenvalue, one of its diagonal entries will converge to the Frobenius norm of the original input matrix. However, this value only represents the maximum possible value an element of  $A$  can take on following one Jacobi Rotation. One must also consider how the intermediary operations affect the magnitude of each element of  $A$ . Luckily this is simplified by the fact that the architecture only uses one distinct hardware block for the mathematical operations - a CORDIC implementation. During execution, a CORDIC operation will scale the input values. The upper bound for this scaling can be found by evaluating the reciprocal of Equation (4.34) as an infinite product

$$k_{lut} = \prod_{j=0}^{\infty} \frac{\sqrt{1 + 2^{-2j}}}{1} \quad (4.55)$$

$$\approx 1.647 \quad (4.56)$$

Based on this scaling factor, and the information regarding the largest possible value of a matrix elements, it is possible to deduce a lower bound for the maximum absolute value that the memory must be able to accommodate to ensure that overflow does not occur:

$$\text{Minimum value required to be representable by memory : } 1.647\|A\|_F \quad (4.57)$$

In it of itself this bound is not very useful as it required knowledge of  $\|A\|_F$ . However, if any information regarding the entries of  $A$  is known, then it is likely possible to deduce some kind of upper bound for the memory requirements.

For example, consider if  $A$  is a  $20 \times 20$  matrix and it is known that each entry can take on a value between -1 and 1. In this case the maximum possible Frobenius norm of  $A$  is equal to 20. Using the above relation, it is then clear that the internal memory of the architecture must be able to store values of at least  $1.646 \times 20 = 33$ . This implies that at least 6 integer bits are required.

As this report does not target any specific application, for the sake of convenience, it is chosen that the memory of the designed architecture should be able to represent numbers in the range  $[-1, 1 - 2^\kappa]$ , where  $\kappa$  is the number of fractional bits. In any subsequent tests, the input matrices will be scaled appropriately to ensure that overflow does not occur.

However, this limited range makes it impossible to represent  $\frac{\pi}{2}$ , thereby risking overflow when calculating an angle using  $\arctan\left(\frac{2A_{p,q}^{[k]}}{A_{q,q}^{[k]} - A_{p,p}^{[k]}}\right)$ . To overcome this, the angular range  $[-\frac{\pi}{2}, \frac{\pi}{2}]$  is mapped to  $[-1 - 2^\kappa, 1 - 2^\kappa]$ . In practice this is simply done by determining the ratio between  $\frac{\pi}{2}$  and  $1 - 2^\kappa$ , and then scaling the predetermined angles,  $\theta^{[i]}$ , by this ratio. For example, if  $\kappa = 15$ , then the ratio is approximately  $\frac{2}{\pi} \approx 0.64$ . Based on this the new numerical representation of the predetermined angles are found as:

$$\hat{\theta}_{\text{scaled}}^{[i]} = 0.64 \arctan(2^{-i}) = [0.495, 0.292, 0.154, 0.078, \dots, 0.64 \arctan(2^{-(n-1)})]. \quad (4.58)$$

#### 4.6. Word length Analysis

It may appear to have been more convenient to simply add an integer bit to increase the representable range to  $[-2, 2 - 2^\kappa]$ , thereby allowing  $\frac{\pi}{2}$  to be directly representable. While this is a possible avenue, the scaling approach is preferable, as it allows for a more efficient usage of the numerical representation space. If no scaling is used, then the numbers between  $[-2, -\frac{\pi}{2}]$  and  $[\frac{\pi}{2}, 2 - 2^\kappa]$  will never be used for the purpose of determining an angle. However, in the scaling approach, all representable values may be used. This is advantageous as it improves the effective numerical resolution of the CORDIC computations.

### 4.6 Word length Analysis

Having decided to implement the cyclic Jacobi eigenvalue solver with a stopping criterion based on a fixed number of iterations, it is important to analyse the numerical aspects of a fixed point implementation. This is necessary to determine the required number of fractional and integer bits that should be used. For this purpose, Algorithm 1 has been implemented in Python 3.8 using a fixed-point computation library. While this algorithm does not follow the exact same steps as the one proposed for the architectural implementation, it is assumed to be representative for the purposes of assessing the relationship between fractional word length and numerical accuracy.

The most rigorous method to evaluate how the number of fractional bits affects the accuracy of the calculated eigenvalues would be to conduct a detailed theoretical analysis. However, such a comprehensive analysis is beyond the scope of this report. Therefore, it has instead been decided to use a more empirical approach based on Monte-Carlo simulations.

100 real symmetric  $n \times n$  matrices are randomly generated. The eigenvalues of these matrices are then repeatedly found using Algorithm 1, configured for a range of fractional bit lengths. To ensure that overflow does not occur, this implementation is configured with infinite integer bits. Additionally, nominally true eigenvalues of the matrices are found using the double-precision floating-point eigenvalue solver from NumPy [36]. While it is important to be cognisant of the limitations of such a solver, the eigenvalues found using double precision floating-point arithmetic are defined as exact values for the purposes of this analysis.

Both absolute and relative error will be used as metrics for this analysis. These are defined as being the absolute value of the residual, and the absolute value of the residual relative to the size of the eigenvalue respectively.

$$\text{absolute error} = |\lambda_i - \hat{\lambda}_i| \quad (4.59)$$

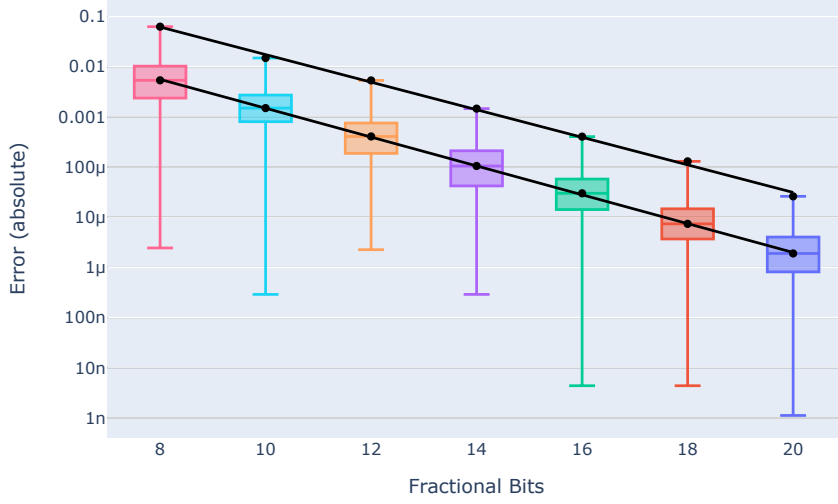
$$\text{relative error} = \left| \frac{\lambda_i - \hat{\lambda}_i}{\lambda_i} \right| \quad (4.60)$$

In Equation (4.59) and Equation (4.60)  $\lambda_i$  corresponds to the eigenvalues found by NumPy, and  $\hat{\lambda}_i$  is the value found by the implementation of the Jacobi eigenvalue algorithm.

The absolute and relative errors for each bit depth can be seen in Figure 4.2 and Figure 4.3 respectively. These box-and-whisker plots show the upper and lower quartiles of the error, as well as the median as the box, and minimum and maximum values of error at the whiskers.

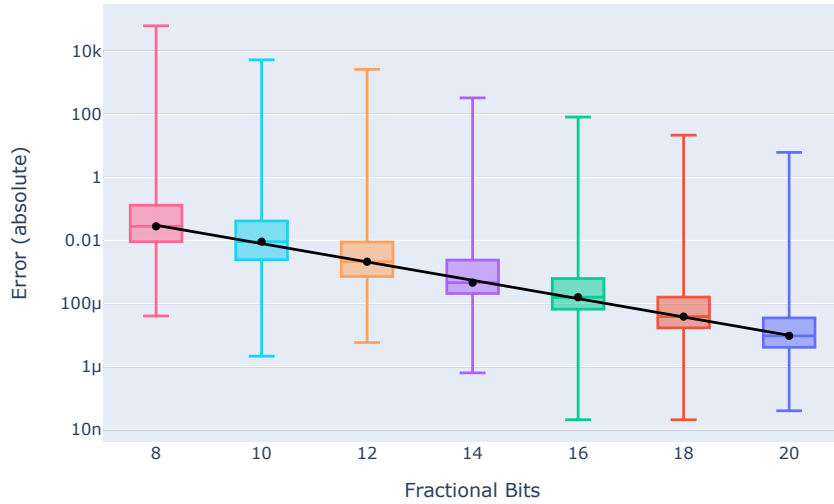
#### 4.6. Word length Analysis

Abs error of eigenvalues of 5x5 matrices using cyclic and 200 iters



**Figure 4.2:** Box plot of the absolute eigenvalue error versus number of fractional bits. Median Fit:  $0.1054e^{-0.6617x}$ ,  $r^2 = 0.9998$ , Upper bound fit:  $2.272e^{-0.6323x}$ ,  $r^2 = 0.9979$ .

Rel error of eigenvalues of 5x5 matrices using cyclic and 200 iters



**Figure 4.3:** Box plot of the relative eigenvalue error versus number of fractional bits. Median fit:  $1.869e^{-0.6699x}$ ,  $r^2 = 0.9986$ .

To supplement the box plots, Figure 4.2 also contains lines of best fit for both the median and worst case error, found using exponential curve fitting. In Figure 4.3 only the median fit is plotted, as the upper bound of the relative error does not convey any additional information, as it mirrors the changes in the median. These large relative errors are not surprising and can occur when an eigenvalue as calculated by the trusted reference is orders of magnitudes smaller than what it is possible to represent

#### 4.6. Word length Analysis

with the fractional word length in use.

The figures show strong evidence for an exponential relationship between the fractional word length and the obtained accuracy, as when plotted on a log-linear scale, error decreases exponentially as word length increases, with all datapoints being correlated by a high  $r^2$  value. From the lines of best fit, it can be inferred that the error almost halves for every additional fractional bit. This inference is in line with the broader concept that the minimum representable value of a fixed-point number halves for every additional fractional bit.

This analysis can be combined with a given application's accuracy requirements, to help guide the selection of the architectures word length. However, as this report has no specified accuracy requirements the word length is selected to be 16 bits, in line with many standard fixed-point DSPs. This will allow for another source of comparison and evaluation of a completed architecture.

It is then possible to conclude on the number of iterations and the internal word length of all the CORDIC functional units. based on the analysis presented in Section 4.3.3, it is seen that number of iterations  $n = 18$  and internal word length  $b = 21$  yields a predicted value of 16.13 accurate bits, when recalling that one iteration is saved by a restriction of the domain. These values of  $n$  and  $p$  will then be used later in Chapter 6 in the design of the CORDIC functional units.

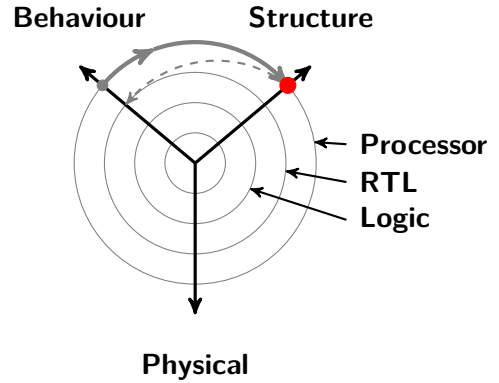
In this chapter, the CORDIC algorithm and its relevance for this application has been discussed, alongside the choice of stopping criteria. The errors associated with the word length have also been analysed, and 16 fractional bits have been selected for implementation. This discussion around sub-algorithms has resulted in the synthesis of Algorithm 3, which will be mapped into hardware in the next chapter.

# Chapter 5

## Hardware Design

The previous chapters have exclusively focused on the computational factors of the Jacobi algorithm. However, as part of the design of a hardware architecture, it is also necessary to consider the memory design and data transfer aspects. When allowing for fast parallel computations, it is critical that the architecture supports fast transfer of data to the relevant functional units. Failure to achieve this will result in inefficient computations as functional units must stay idle awaiting the availability of data. This chapter presents and argues for the use of systolic arrays to overcome such memory bottlenecks. The chapter culminates in high abstraction description of the architecture that is to be implemented.

With our newfound knowledge about the behaviour of the Jacobi algorithm and its functionality, we take a step back and expand our description of the structural domain at the processor level, to consider data transfer aspects. The concept of systolic arrays is explored throughout Section 5.1, to allow a large amount of computations to be performed simultaneously without being hindered by data transfer constraints. This can be considered an iteration of the  $A^3$ -model, between the algorithmic and architectural domain.

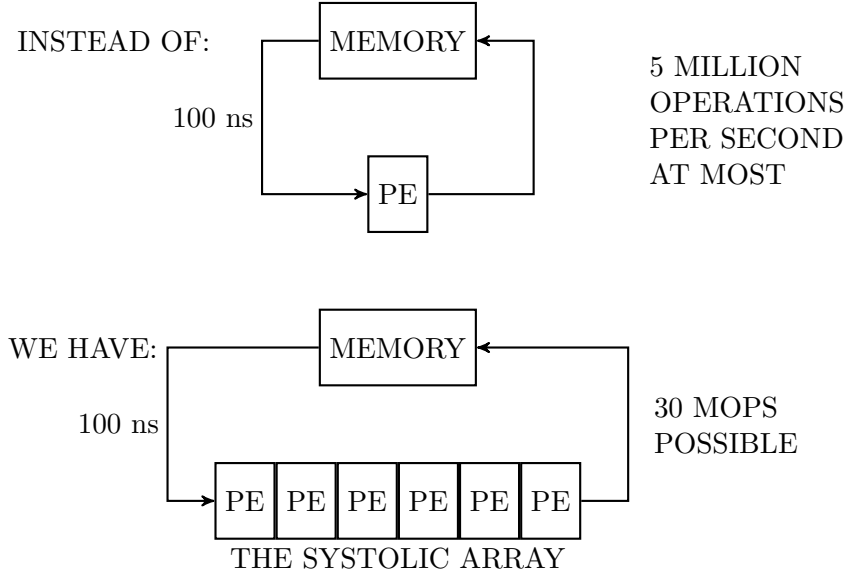


### 5.1 Systolic Arrays

A systolic array [37] is a type of architecture that seeks to reduce the dependency on I/O busses, and exploit the algorithm for maximum concurrent execution of one or more steps. This is useful for matrix computations as these often contain independent multiplications and additions. Systolic arrays aim to avoid the need for various processes independently and routinely reading from and writing to memory, a process which might cause the system to be bounded by the I/O speed as opposed to the computation speed. In this section, an overview of systolic arrays is given, and their relevance for application in the Jacobi eigenvalue solver is explored.

A systolic array consists of interconnected processing elements as well as an overarching control scheme. The control scheme will be detailed later in this chapter. The principal is that by allowing the processing elements to communicate with each other, the need for I/O access can be drastically reduced as exemplified in Figure 5.1. By allowing each processing element its own dedicated memory as well as dedicated data paths to specific neighbouring processing element, the data can be passed between the processing elements without relying on common data busses. This is, in effect, a form of pipelining with pre-defined passing [38]. If the processing elements throughput is limited by the I/O transfer rates on the common data busses, we should be able to optimise the data flow between processing elements to allow for higher throughput. All of these processing elements are essentially performing the same task over and over which is designed to complete in a fixed number of clock cycles. It is worth noting that not all processing elements are necessarily computing the same operation.

## 5.2. Processing Elements



**Figure 5.1:** The basic principle of a systolic system [37]. In this example the systems have a max memory bandwidth of 10 MB/s and each processing elements consumes/produces two bytes. Even if it is assumed that the processing elements allow for instantaneous execution, the top system is limited to a maximum of five million instruction per seconds by virtue of the memory bandwidth. However, if multiple operations are performed on the same data, the total throughput can be increased. This is illustrated in the bottom system.

Recalling the Jacobi Algorithm description in Algorithm 3, this algorithm simultaneously performs operations on all elements of a matrix, a systolic array is potentially a beneficial topology to select.

Several different implementations of the Jacobi algorithm have been developed using this technique [39][40][41][42][43]. Most of these are inspired by methods introduced in [24]. This previous work suggests that systolic array processing is a valid direction for this project.

Many structured methodologies for mapping algorithms to systolic arrays have been developed [44]. Many of these methods rely on similar principles. The process begins by formulating a high abstraction description of the algorithm, such as a set of equations or pseudo code [44]. As these descriptions are not directly useful to derive a systolic array, the description is transformed into a more appropriate form [44]. Examples of transformation include the creation of signal flow graph [45], or the creation of mathematical description where the algorithm is described in terms of data dependency vectors [46]. Such transformation can help uncover inherent parallelism and find patterns in the data communication requirements.

While these methods provide rigorous frameworks for generalised systolic array design, in this report a more ad-hoc approach to the presentation of a systolic array for the Jacobi algorithm is used. This style is chosen as the time frame of this project did not allow for detailed insight into the structured methodologies.

## 5.2 Processing Elements

As the systolic array is supposed to execute the behaviour described by Algorithm 3, it is logical to start the design by analysing the algorithms operations and data dependencies.

For the sake of improving readability, Algorithm 3 is restated:

---

Parallel Cyclical Jacobi Algorithm

---

```

1: Accept a symmetric matrix  $A$  as input
2: counter = 0
3: while counter <  $\left\lceil \epsilon_{stop} \frac{n(n-1)}{2} \log_2(n) \right\rceil$  do
4:    $J = I$ 
5:    $M =$  new set of feasible indexes  $(p, q)$ 
6:   for each  $(p, q) \in M$  do in parallel
7:      $\theta = \frac{1}{2} \arctan\left(\frac{2A_{p,q}}{A_{q,q} - A_{p,p}}\right)$ 
8:     for all  $i$  from 1 to  $n$  do in parallel
9:       Rotate  $\begin{bmatrix} A_{p,i} \\ A_{q,i} \end{bmatrix}$  by  $\theta$ 
10:    end for
11:    for all  $i$  from 1 to  $n$  do in parallel
12:      Rotate  $\begin{bmatrix} A_{i,p} \\ A_{i,q} \end{bmatrix}$  by  $\theta$ 
13:    end for
14:  end for
15:  counter +=  $\lfloor \frac{n}{2} \rfloor$ 
16: end while
17:  $\Lambda = \text{diag}(A)$ 

```

---

The algorithms three `for` loops contains three distinct operations. First a set of angles,  $\theta^{[1]}$  to  $\theta^{\lfloor \frac{n}{2} \rfloor}$ , is found according to Line 7. Following this, Line 9 defines that each  $\theta^{[i]}$  is used to rotate  $n$  unique two-dimensional vectors formed by elements along the  $p_i^{\text{th}}$  and  $q_i^{\text{th}}$  rows. Finally, Line 12 shows that each  $\theta^{[i]}$  is again used to rotate  $n$  unique vectors, though these are formed by elements along the  $p_i^{\text{th}}$  and  $q_i^{\text{th}}$  columns. All these operations are dependent on the current set,  $M$ , of  $\frac{n}{2}$  unique  $(p, q)$  pairs.

When the computations in the three `for` loops are complete, they should be repeated for a new set of  $(p, q)$  pairs. Based on this update, the elements used to determine the angles and the vectors of the row/column rotation are also changed to reflect the new content of set  $M$ . In practice this update implies that the hardware units performing the operations must somehow receive new matrix elements. Instead of explicitly calculating new  $(p, q)$  pairs and broadcasting the new vector elements from memory to each of the processing elements, the idea of a systolic array is to identify a cyclic pattern in the data transfer between processing elements. This will implicitly achieve an identical effect. However, before such a pattern can be found, more knowledge regarding the processing elements is required. Specifically, the types of processors, and which operations these must perform must be determined. Additionally, it must be specified how many and which matrix elements each processing element requires to complete its computations. These requirements are found by independently considering the algorithms three mathematical operations. Each of the  $\frac{n}{2}$  operations for calculating the angles only requires three matrix elements, and thus only a subset of the all the matrix elements are required in these operations. In contrast, as each of the  $\frac{n^2}{2}$  row and column rotations requires two matrix elements to be computed, the computation of all row and column rotations will require the use of all matrix elements.

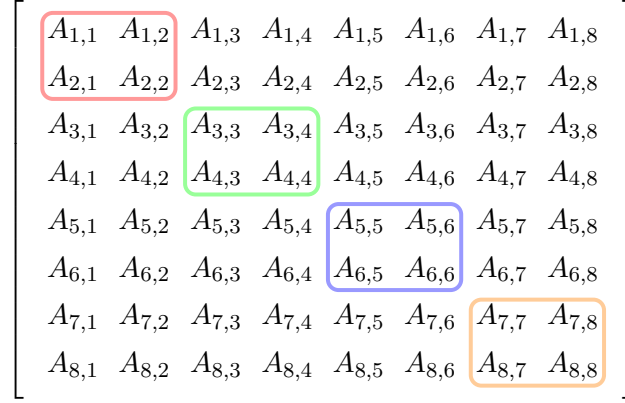
Consider the case where  $n = 8$  and the  $(p, q)$  pairs are set equal the initial state of the chess tournament algorithm in Section 3.6:

$$\text{set of } (p, q) \text{ pairs: } M = \{(1, 2), (3, 4), (5, 6), (7, 8)\}$$

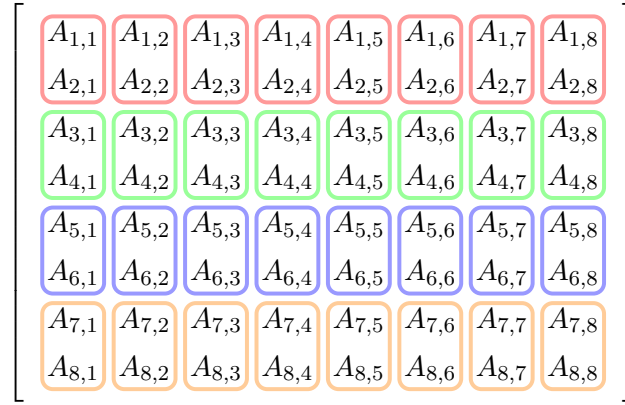
In this case it is simple to visualise which elements each of the three operations require as input to execute:



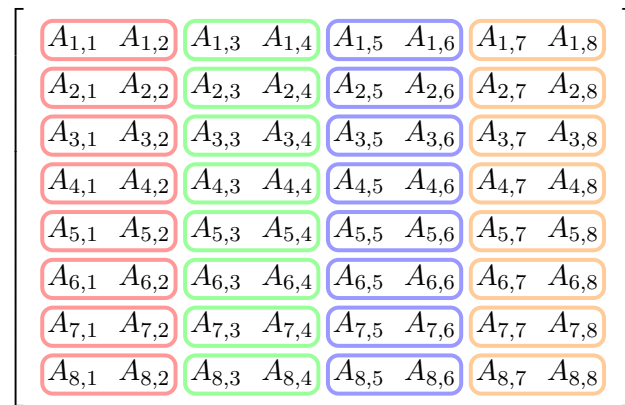
## 5.2. Processing Elements



**Figure 5.2:** Representation of the elements required for finding rotation angles. Each bounded box contains the four matrix elements required to determine the angle (strictly speaking only three elements are required, but either  $A_{p,q}$  or  $A_{q,p}$  may be used).



**Figure 5.3:** Representation of the elements required for each row rotation. Each bounded box contains two matrix elements and represents the vector to be rotated. The rotation angle of each box is derived from the elements in Figure 5.2 bounded by the box of the same colour.



**Figure 5.4:** Representation of the elements required for each column rotation. Each bounded box contains two matrix elements and represents the vector to be rotated. The rotation angle of each box is derived from the elements in Figure 5.2 bounded by the box of the same colour.

To minimise the amount of registers and their associated logic in the processing elements, it is undesirable to maintain copies of data. Therefore, the processing element should be designed such that no processors are working using the same data. To achieve this, each processing element must be able to

## 5.2. Processing Elements

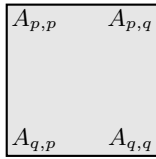
execute all the operations a given matrix element is used in. Figure 5.5 combines Figure 5.2, 5.3 and 5.4 to visualises which operations require the same elements as input.



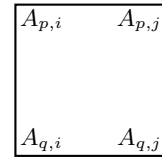
**Figure 5.5:** Combination of Figure 5.2, 5.3 and 5.4.

The figures above shows a clear segmentation of the matrix elements into separate  $2 \times 2$  sub matrices. These segment provide the basis for the definition of the processing elements.

In Figure 5.5, there are two different types of segmentation. One type along the diagonal of the figure all as a combination the same colour, and another in the off-diagonal as a combination of different colours. The diagonal segment defines the diagonal processors. These processors first calculate an angle and then proceeds to rotate its row and then column elements by this angle [47]. Similarly, the off-diagonal segments form the off-diagonal processors. These are slightly simpler and are only able to rotate its row and column elements by some specified row and column rotation angle [47]. Figure 5.6 and Figure 5.7 defines a symbolic representation of the two processors:



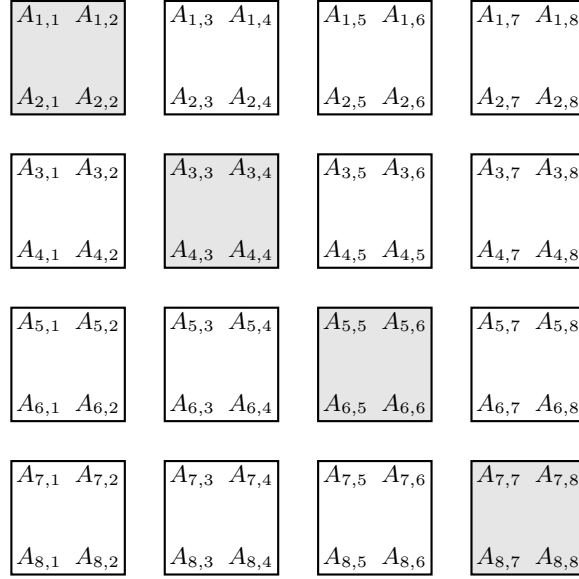
**Figure 5.6:** Diagonal processing element symbol.



**Figure 5.7:** Off-diagonal processing element symbol.

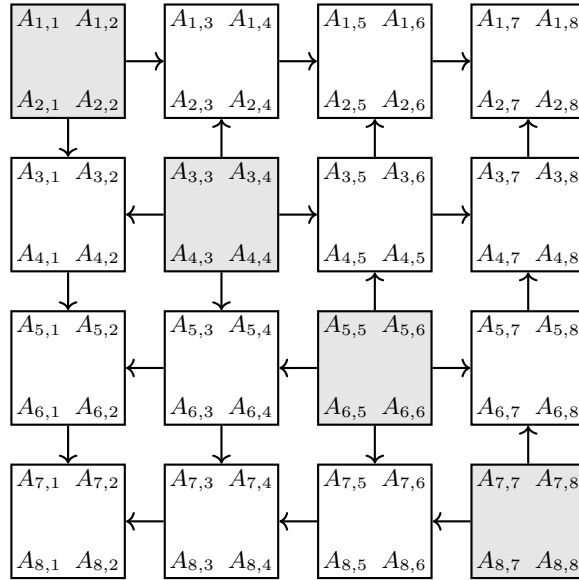
Figure 5.8 visualises the processor array when the segments of Figure 5.5 are replaced by their associated processing element:

## 5.2. Processing Elements



**Figure 5.8:** Illustration of the elements that each require for processing.

So far only the processing element's reliance on the matrix elements have been considered. However, to successfully execute row and column rotations the rotation angle must be known. As the diagonal processing elements determine these angles, the elements must broadcast the angles to the relevant off diagonal elements. In the current example configuration in Figure 5.8, this can be done by connecting each diagonal processing elements to the off-diagonal elements of its own row/column [47] [48]. These row/column processing busses are then used to transport the rotation angles to the off-diagonal processing elements. These busses are visualised in Figure 5.9.



**Figure 5.9:** Array including data bus for broadcasting the calculated angle from the diagonal to the off-diagonal processing elements.

### 5.3 Transfer of Matrix Elements

So far, the design of the systolic array has mostly been based on the initial state of the  $(p, q)$  pairs. This has been convenient, as the data used in the computations is physically located next to each other when it is displayed in standard matrix form. However, when other pairings are used the elements used in each of the computations are not by default located next to each other. For example, in the case where  $(p, q) = (1, 4)$ , the entries required to determine  $\theta$  are  $A_{1,1}$ ,  $A_{4,4}$ ,  $A_{1,4}/A_{4,1}$ . Unlike when  $(p, q) = (1, 2)$ , these elements are not located next to each other. This is illustrated by Figure 5.10:

$$\begin{bmatrix} \boxed{A_{1,1}} & A_{1,2} & A_{1,3} & \boxed{A_{1,4}} & \dots \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & \\ \boxed{A_{4,1}} & A_{4,2} & A_{4,3} & \boxed{A_{4,4}} & \\ \vdots & & & & \ddots \end{bmatrix}$$

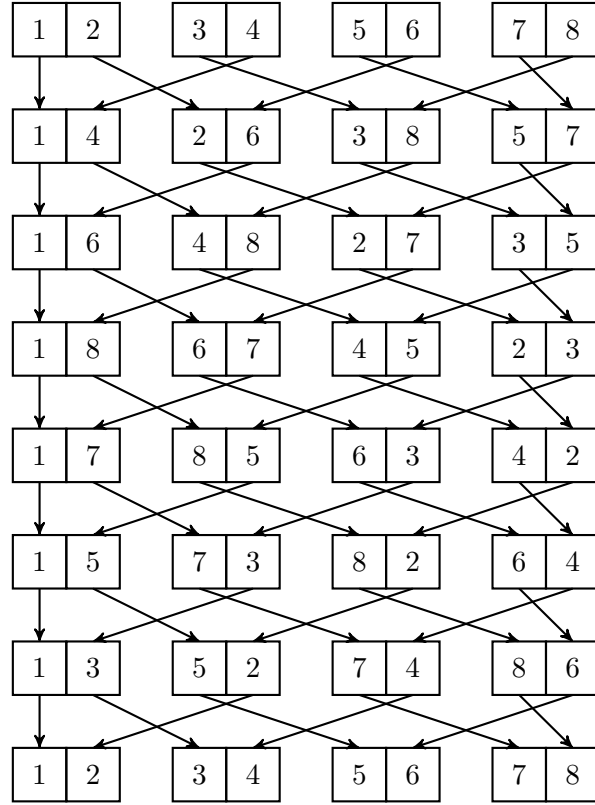
**Figure 5.10:** Illustration of elements required for determining  $\theta$  for the case where  $(p, q)$  equals  $(1, 2)$  and  $(1, 4)$ . The red box encapsulates the required elements in the case where  $(p, q) = (1, 2)$ . Similarly, the blue boxes contain the required elements for  $(p, q) = (1, 4)$ .

Due to this potential dispersion of the elements used in the same computations, the systolic array will somehow have to transfer the matrix elements in such a way that the processing elements are operating on the correct elements for the current  $(p, q)$  pairs. In the following it is shown how the previously presented array structure can be modified to facilitate such transfers.

Again, consider the case of  $n = 8$ . Using the chess tournament algorithm, the seven repeating  $(p, q)$  pairs composing a complete sweep, are found to be:

$$\begin{aligned} \text{Set 1: } & \{(1, 2), (3, 4), (5, 6), (7, 8)\} \\ \text{Set 2: } & \{(1, 4), (2, 6), (3, 8), (5, 7)\} \\ \text{Set 3: } & \{(1, 6), (4, 8), (2, 7), (3, 5)\} \\ \text{Set 4: } & \{(1, 8), (6, 7), (4, 5), (2, 3)\} \\ \text{Set 5: } & \{(1, 7), (8, 5), (6, 3), (4, 2)\} \\ \text{Set 6: } & \{(1, 5), (7, 3), (8, 2), (6, 4)\} \\ \text{Set 7: } & \{(1, 3), (5, 2), (7, 4), (8, 6)\} \\ \text{Set 1: } & \{(1, 2), (3, 4), (5, 6), (7, 8)\} \\ & \vdots \end{aligned}$$

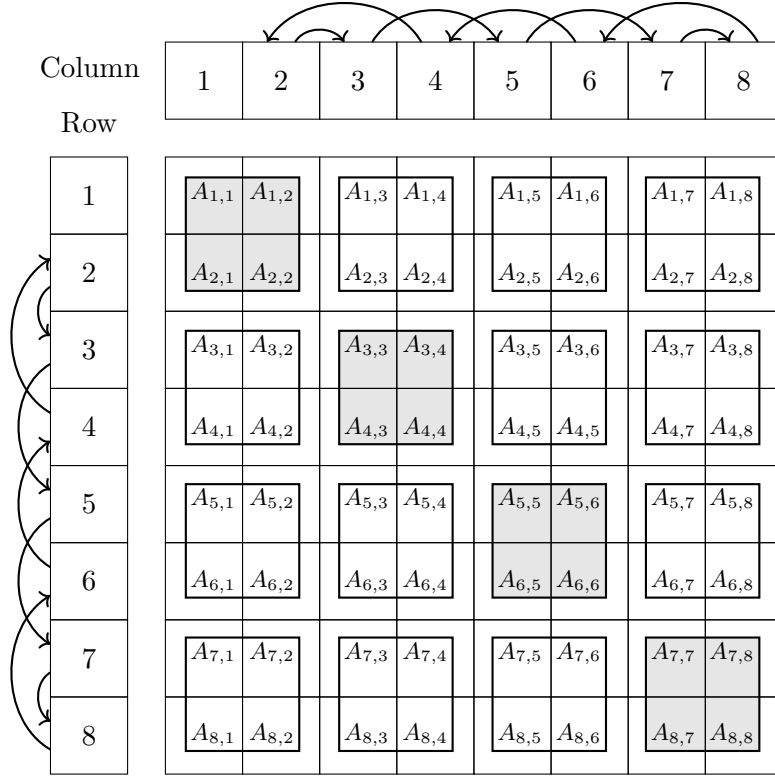
If a path of each index location in the set is traced, the following pattern emerges [48]:



**Figure 5.11:** Traced path of each index in the chess tournament algorithm for  $n = 8$  [48].

It follows that if a structure of processors exists for computing one of the pairing set, then if the row/column location of the elements of the input matrix are swapped according to the above pattern, the computation will be performed on the next pairing set of the chess tournament algorithm. One such structure has already been presented in Figure 5.8. Figure 5.12 visualises how every element in the array must be swapped to fulfil the pattern in Figure 5.11:

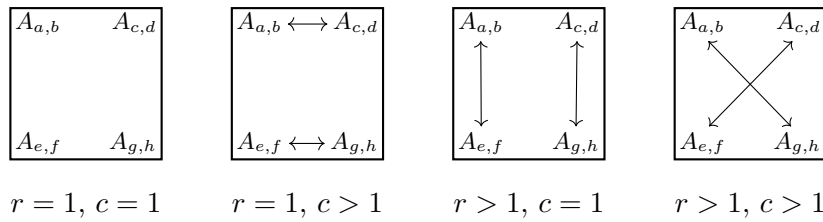
### 5.3. Transfer of Matrix Elements



**Figure 5.12:** Illustration of how each matrix elements should be switched to mirror the chess tournament pairings.

Based on the above illustration, it is evident that this swapping scheme will in some cases need to swap elements separated by another element in the row/column. However, as the processing elements each contain a  $2 \times 2$  sub-matrix, the processors only need to share data with its direct or diagonal neighbours.

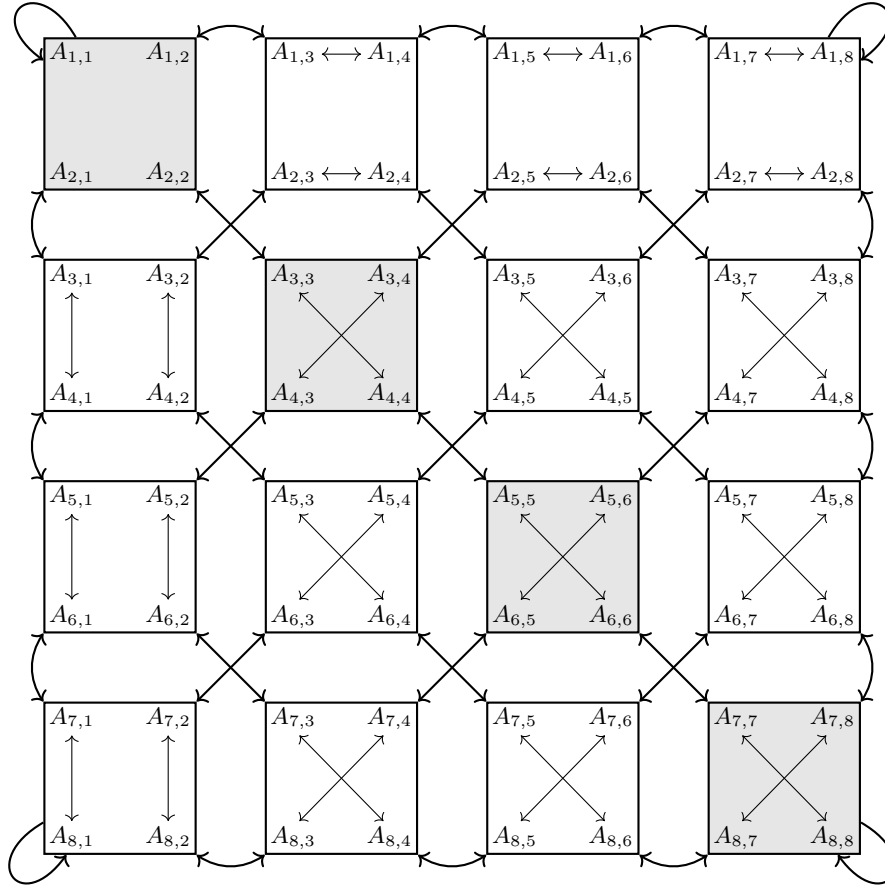
Specifically, the swapping scheme is achieved through an inner processing element data exchange, followed by an outer exchange where each element shares data [47]. The topology of the inner exchanges depend on the row and column location of the processing element [47]:



**Figure 5.13:** Inner exchange scheme of the different element locations. The elements in the processors are switched according to the arrows.  $r$  and  $c$  represent the row and column indices respectively [47].

After the inner exchanges, the following outer exchange scheme will ensure that the elements are switched according to Figure 5.11 and Figure 5.12 [47]:

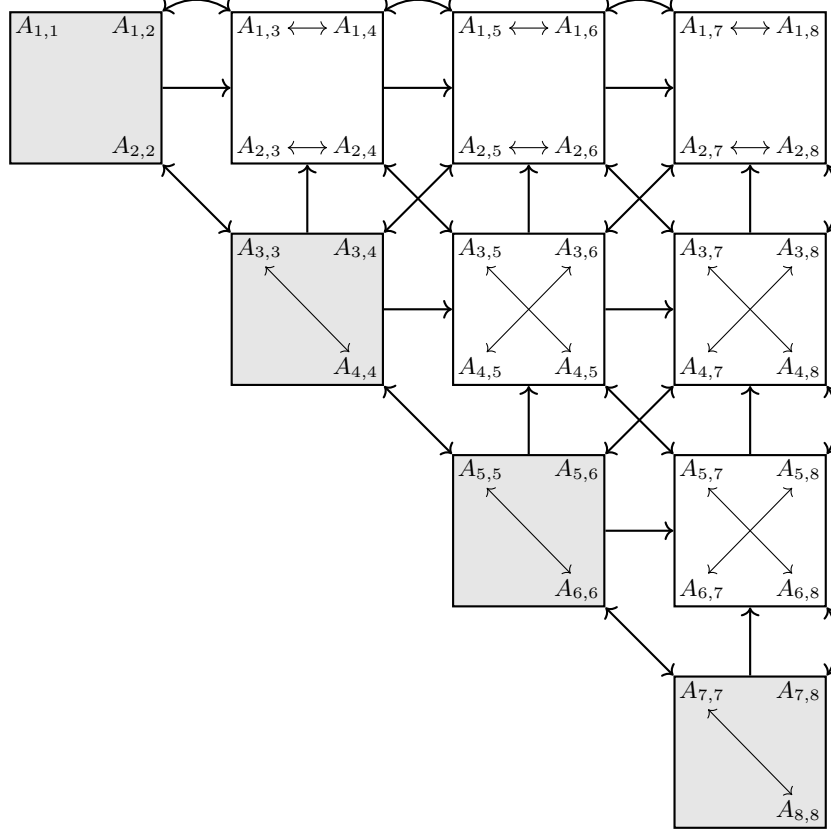
### 5.3. Transfer of Matrix Elements



**Figure 5.14:** Array with inner and outer exchanges visualised (Angle busses omitted to improve readability) [47].

This concludes the presentation of the systolic arrays data exchange scheme. However, there is still room for improvement regarding the structure of the systolic array. If the symmetric nature of the input matrix is considered, it is evident that the computations performed in processors of lower and upper triangular sections of the array are identical. Therefore, one of these groups of processors can be omitted without impacting the computation of the eigenvalues. With this modification, the final array structure becomes:

## 5.4. Alternative Implementation



**Figure 5.15:** Triangular array with all connections drawn.

## 5.4 Alternative Implementation

The method shown earlier in this chapter involves explicit calculation of the arctangent and transmission along the rows and columns. In this section, an alternative implementation developed in this project is presented bypassing the need to explicitly calculate the arctangent. This significantly simplifies the design process of the diagonal processors described in Section 5.2

To achieve this, we first consider the function used to determine  $\sigma$  in Equation (4.42) for the next iteration of CORDIC, and its relationship to the matrix entry being zeroed,  $A_{p,q}^{[k]}$ . In the case of CORDIC, this function is evaluated by taking the sign bit of  $x_2$ . This algorithm will use a similar process, considering instead the sign bit of  $A_{p,q}^{[k]}$ . Proving that this entry  $A_{p,q}^{[k]}$  exhibits sinusoidal behaviour is required for the use of CORDIC in this situation.

It is observed that rotating a 2-dimensional vector is analogous to applying some phase shift to its respective components. Without loss of generality, this is defined for unit vectors. A unit vector  $[x_1 \ x_2]^T$  has by definition a length of 1, and so it must lie on the unit circle  $x^2 + y^2 = 1$ . the components  $x_1, x_2$  of this vector can also be found in relation to the angle  $\theta$  between the vector and the positive  $x$ -axis as

$$x_1 = \cos(\theta) \quad (5.1)$$

$$x_2 = \sin(\theta) \quad (5.2)$$

A rotation without any scaling of this vector can be achieved by any constant phase shift  $\phi$  applied



#### 5.4. Alternative Implementation

to both angles, as

$$x_1 = \cos(\theta + \phi) \quad (5.3)$$

$$x_2 = \sin(\theta + \phi) \quad (5.4)$$

From Equation (4.5), re-written below with variables substituted for the benefit of the reader, we have

$$A_{p,q}^{[k+1]} = (\cos^2(\theta) - \sin^2(\theta))A_{p,q}^{[k]} + \sin(\theta) \cos(\theta)(A_{p,p}^{[k]} - A_{q,q}^{[k]}) \quad (5.5)$$

By applying the trigonometric identities for double angles

$$\sin \theta \cos \theta = \frac{\sin(2\theta)}{2} \quad \cos^2 \theta - \sin^2 \theta = \cos 2\theta$$

Equation (4.5) may now be re-written as

$$A_{p,q}^{[k+1]} = \cos 2\theta A_{p,q}^{[k]} + \frac{\sin(2\theta)}{2}(A_{p,p}^{[k]} - A_{q,q}^{[k]}) \quad (5.6)$$

Using the harmonic addition theorem [49], we will now show that this can be written as a single sinusoidal function. This result means that the sign of the entry  $A_{p,q}$  may be used to direct an iterative search algorithm to zero that particular off-diagonal entry. First, the desired form of our result is considered

$$A_{p,q}^{[k+1]} = \mu \cos(\theta + \phi) \quad (5.7)$$

expanding this using trigonometric addition properties results in

$$A_{p,q}^{[k+1]} = \mu \cdot \cos(\theta) \cos(\phi) - \mu \cdot \sin(\theta) \sin(\phi) \quad (5.8)$$

Then, letting  $\omega = 2\theta$  such that

$$A_{p,q}^{[k+1]} = \cos(\omega)A_{p,q}^{[k]} + \sin(\omega)\frac{(A_{p,p}^{[k]} - A_{q,q}^{[k]})}{2} \quad (5.9)$$

then allows for a method of comparing coefficients between equations to be employed in order to derive a relationship between  $A_{p,q}^{[k]}$ ,  $\frac{A_{p,p}^{[k]}A_{q,q}^{[k]}}{2}$  and  $\mu$ . This is initiated by taking

$$A_{p,q}^{[k]} = \mu \cos(\phi) \quad (5.10)$$

$$\frac{A_{p,p}^{[k]}A_{q,q}^{[k]}}{2} = \mu \sin(\phi) \quad (5.11)$$

Then from the tangent identity already mentioned in Equation (4.36), it follows that

$$\tan(\phi) = -\frac{\sin(\phi)}{\cos(\phi)} \quad (5.12)$$

$$= \frac{\left(\frac{A_{p,p}^{[k]}A_{q,q}^{[k]}}{2}\right)}{A_{p,q}^{[k]}} \quad (5.13)$$

where  $\mu$  appears as a common factor in both the numerator and denominator, and thus cancels. Taking the arctangent of this computation the gives the phase shift,  $\phi$ . Finding an expression for  $\mu$  then completes this proof. This comes from applying the Pythagorean identity to Equation (5.9), recalling that this equation evaluates to numbers that scale a cosine and sine term. this then leaves

$$\mu^2 = \left(\frac{A_{p,p}^{[k]}A_{q,q}^{[k]}}{2}\right)^2 + (A_{p,q}^{[k]})^2 \quad (5.14)$$

$$\mu = \sqrt{\left(\frac{A_{p,p}^{[k]}A_{q,q}^{[k]}}{2}\right)^2 + (A_{p,q}^{[k]})^2} \quad (5.15)$$

#### 5.4. Alternative Implementation

From this result, it is concluded that evaluating the sign of  $A_{p,q}^{[k]}$  is sufficient to determine rotation direction, thus eliminating the need to calculate the angle of rotation as a separate process as in Section 4.3.2.

As the zeroing of entry  $A_{p,q}^{[k]}$  also affect the rows and columns  $(p,q)$ . This is done in a similar manner to Figure 5.9, but the sign bit  $\sigma = \text{sign}(A_{p,q})^{[k]}$  is transmitted instead of the angle  $\theta$ .

This then enables a remarkably similar structure to the systolic array found in Section 5.1, while potentially speeding up the CORDIC iteration time by a factor of  $\frac{1}{3}$ , as one less CORDIC is required. The whole process can now be accomplished using CORDICs in rotation mode for all processors that are modified to accommodate an external source for  $\sigma$ . Transfer of data between processing elements has already been addressed in Section 5.3

# Chapter 6

## RTL Design

In this section the synthesis of the Register-Transfer Level (RTL) design, as well as the tools and methodologies used to obtain the synthesis will be described.

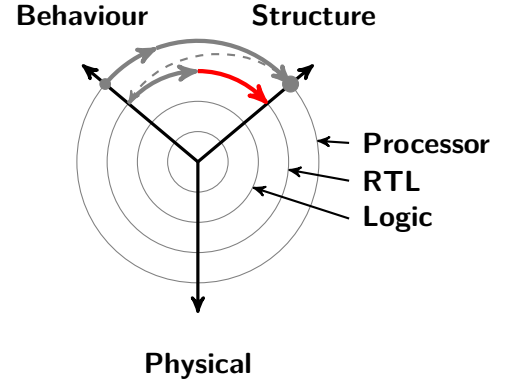
At this stage in the design process, the sequence of arithmetic operations necessary to calculate the eigenvalues of a real symmetric matrix using the Jacobi eigenvalue algorithm have been defined. In Chapter 5 it was decided to achieve this through a systolic array. As part of this, two processor elements are needed: A diagonal processor element and an off-diagonal processor element. The objective now, is to map these processing elements onto dedicated hardware capable of performing the required operations and create a schedule that ensures a valid sequence of execution.

There are many ways to approach such a task depending on the desired characteristics of the solution whether it is power consumption, area etc. In this design, execution time is the most wanted characteristic as defined by the cost function. Thus, it is desired to choose an approach that allows and exposes design choices that impact execution time. A data flow paradigm has been chosen as the basis for the algorithm representation. This paradigm aids in exposing inherent parallelism in an algorithm and thus the expectation is to improve execution time by exploiting parallelism. To construct an architecture that exhibits the best execution time within the chosen paradigm, an isomorphic hardware mapping is initially pursued. I.e. a "1 to 1" mapping where every operation is assigned a unique hardware element that performs said operation. The resulting architecture will likely use an excessive amount of resources and therefore occupy a large area. In spite of this excess, this approach allows the designed architecture to take advantage of all potential parallelism exposed by the data flow paradigm. This gives the fastest execution time conceivable within this paradigm.

When the isomorphic hardware mapping and accompanying schedule has been obtained, other techniques and paradigms could be considered to help balance the architecture towards a more resource friendly solution. Examples of such techniques could include hardware sharing or pipelining. The benefit of a "1 to 1" approach is that often the resulting architecture, although large, only requires simple control logic and it resembles its graphical representations. Additionally, it will often be a good benchmark as all found parallelism has been exploited. Thus, the effects of other techniques on execution speed is readily assessable. This assesment might not be as apparent if, for instance, hardware sharing and pipelining is introduced early in the design process.

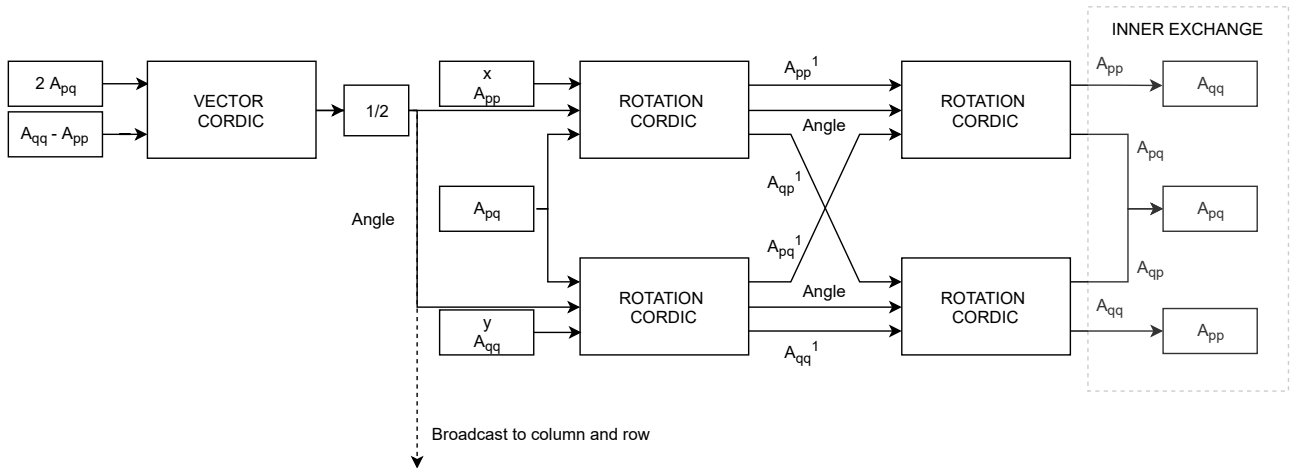
This chapter initially covers the design of the processing elements at a high abstraction level. To distinguish between the two types, their designs are treated separately. As CORDIC is an essential computational block of the two processing elements, its mapping to RTL is covered. Following this, the CORDIC RTL description is merged with the design of the processing elements to form an RTL description of these elements. Finally, multiple processors are combined to create a complete RTL description of the systolic array.

This chapter builds upon the previously defined system behaviour and structure described in Chapter 4 and Chapter 5. Ultimately, the chapter culminates in an RTL description of the architecture. The chapter initially considers a data flow based design of the systolic arrays processing elements. Following this, the constituent CORDICs of the processing elements are expanded upon. As execution speed is a priority, an unrolled CORDIC RTL design is derived. Next the RTL descriptions of the CORDICs are merged with the processing element designs. Finally, the systolic array is formed by combining multiple processors and designing control logic to realise the array's finite state machine. This then completes the arc from a behavioral to a structural description on the RTL abstraction level, thereby completing a first iteration of our design process.



## 6.1 Diagonal Processor

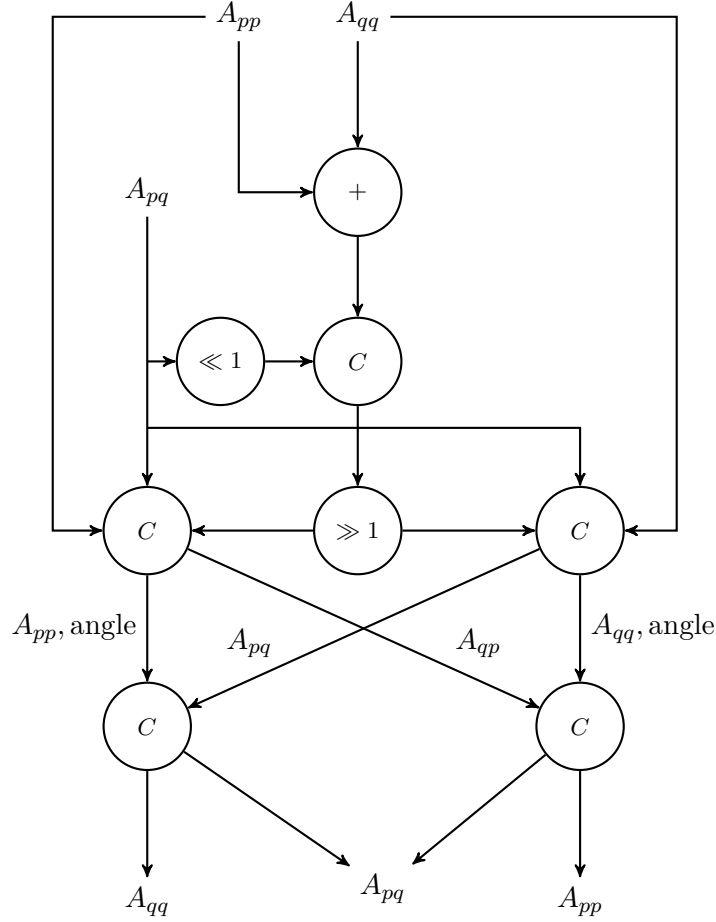
To calculate the eigenvalues of a  $n \times n$  matrix in the systolic array,  $\frac{n}{2}$  diagonal processors will be needed. They will all have to perform the same operation of finding the correct rotation angle for the Jacobi rotation and subsequently perform the Jacobi rotation. Therefore a general diagonal processor can be designed that will just require  $\frac{n}{2}$  instances in the RTL design. CORDIC has been chosen to calculate both the rotation angle as well as the Jacobi rotation. Based on the behavioural description of the diagonal processing element in Chapter 5, it is possible to construct a Large Grain Data Flow Graph (LGDFG) also known as a block diagram, as seen in Figure 6.1.



**Figure 6.1:** Large Grain Data Flow Graph (LGDFG) for diagonal processor.

From this it is evident that multiplication by 2 and  $\frac{1}{2}$ , a subtraction to calculate  $A_{qq} - A_{pp}$  along with CORDIC modules that can perform CORDIC in either vector- or rotation-mode is needed to implement the diagonal processor. By considering any CORDIC operation, whether it is vector- or rotation-mode, as the same operation the Data Flow Graph (DFG) seen in Figure 6.2 is constructed, where CORDIC is denoted as "C".

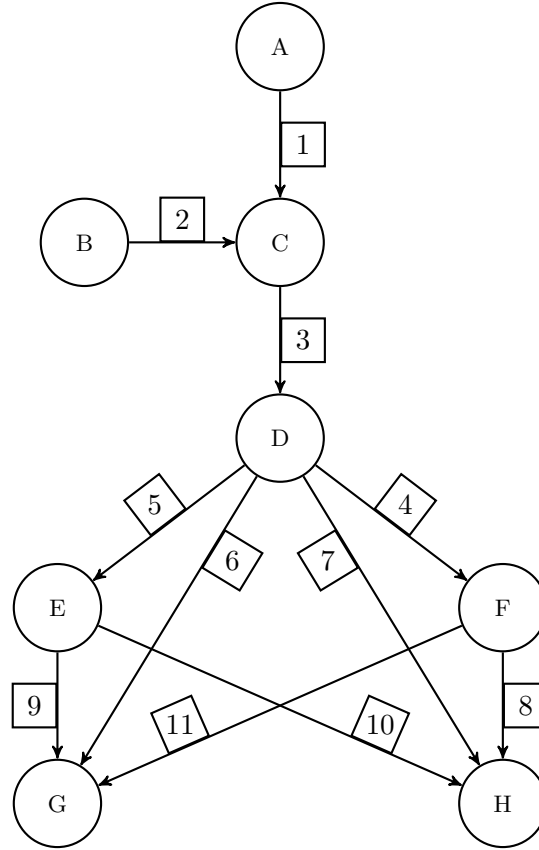
### 6.1. Diagonal Processor



**Figure 6.2:** Data Flow Graph (DFG) for diagonal processor.

To get towards a schedule that exploits inherent parallelism in this algorithm, it is desirable to obtain an acyclic precedence graph. This graph is useful as it orders the intra-iteration dependencies among operations such that it becomes evident which operations can be run in parallel. However, before this graph is obtained, the notion on synchronous data flow is introduced in this analysis [50].

While the notion of synchronous data flow is primarily intended for the analysis of algorithms with multiple sample rates [50], which is not the case for the behaviour considered here, it provides a tool that can be used to partially validate the constructed DFG [50]. In this context, the word "synchronous" refers to the ability to specify *a priori* the number of samples consumed/produced by each operation or node in the DFG. Conversely, asynchronous means that this cannot be determined prior to execution [50]. A possible instance of asynchronous data flow could be data-dependent execution in which the appropriate operations can first be determined once something can be inferred about the available data. For example, if-statements, where appropriate action can only be taken once the conditional has been evaluated and thus remains unknown before execution. In this processing element, the samples consumed and produced by every node can be specified *a priori* and as they will all be unity, this system does not contain multiple sample rates. Nonetheless, by constructing the Synchronous Data Flow Graph (SDFG) from the DFG, as seen in Figure 6.3, it becomes easier to derive feasible schedules and precedence relations.



**Figure 6.3:** Synchronous Data Flow Graph (SDFG) for diagonal processor.

In the SDFG each arc and node is labeled by a number and letter respectively. This SDFG can then be converted to a topology matrix. The arcs and nodes represent the rows and columns of the matrix respectively. Generally, the arcs of SDFGs may also contain delay elements, however in this case none are required.

The topology matrix can be constructed such that index  $(i, j)$  expresses how many samples node  $j$  consumes or produces on arc  $i$  where consuming samples are represented by a negative number. The rank of this matrix shows whether or not the constructed graph exhibits sample-rate inconsistencies and thus in extension if a schedule can exist for the algorithm. Since there are not multiple sample-rates in this algorithm it should be surprising to encounter sample-rate inconsistency so in this case it will largely be used as a cross-check to see if any grave mistakes have been made. The topology matrix for the constructed SDFG can be seen in (6.1).

### 6.1. Diagonal Processor

$$\Gamma = \begin{array}{c|cccccccc} & \text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} & \text{G} & \text{H} \\ \hline 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 8 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 9 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 10 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 11 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \end{array} \quad (6.1)$$

From graph theory it can be found that the rank of  $\Gamma$  is limited to either  $s$  or  $s - 1$ , where  $s$  is the number of nodes [50]. However, for sample rate consistency it is required that the rank of the topology matrix,  $\Gamma$ , is equal to  $s - 1$ :

$$\text{Rank}(\Gamma) = s - 1 \quad (6.2)$$

In this case  $s$  is equal to 8 and it can be verified that the rank of the constructed topology matrix is indeed 7:

$$\text{Rank}(\Gamma) = 7 = 8 - 1 = s - 1 \quad (6.3)$$

this rank requirement is motivated by considering the interpretation of multiplying the topology matrix with any vector  $q$ , typically called an invocation vector [50]. This multiplication will result in a vector representing how many samples are available on each arc after all the nodes have been run as many times as specified by the invocation vector,  $q$ . From this it is concluded that the invocation vector can only contain integer values as it is not possible to execute a node partially, and that the invocation vector can not contain negative values because a node can not consume samples from an arc where it is modeled to produce samples. It is also natural to require that at some point all data that has been produced must also be consumed. Otherwise a non-terminating algorithm would either generate infinitely many samples or require infinitely many samples and both outcomes are infeasible. Thus it is required that an invocation vector,  $\hat{q}$ , must exist such that [50]:

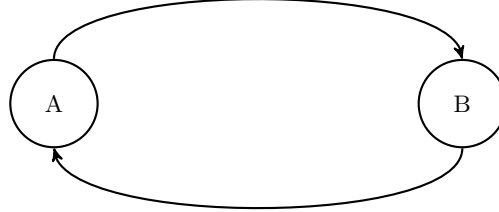
$$\vec{0} = \Gamma \hat{q} \quad (6.4)$$

It is from Equation (6.4) that it is concluded, that for a feasible solution to exist the rank of the topology matrix can not be equal to the number of nodes. Because, if the topology matrix,  $\Gamma$ , had full rank then the dimensionality of its nullspace must be zero and thus only the zero vector,  $\vec{0}$ , satisfies Equation (6.4). As such, the only way an algorithm with said topology matrix could be feasible is if none of the nodes are ever executed. Thus it can be concluded that for any feasible algorithm the rank of the associated topology matrix must be different from the number of nodes:

$$\text{Rank}(\Gamma) \neq s \quad (6.5)$$

## 6.1. Diagonal Processor

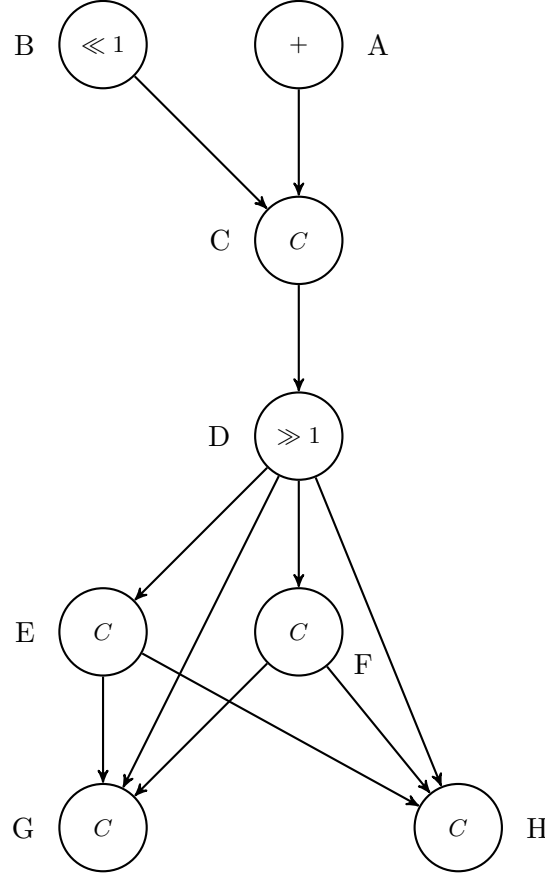
This is not a complete proof that the rank of the topology matrix must equal  $s - 1$  and nor does it show whether it can be expected that a non-negative integer vector exists in the nullspace of  $\Gamma$ , given that it has rank equal to  $s - 1$ , it can be used to construct a schedule for the algorithm. These proofs are out of scope for this report. By considering Figure 6.4 it is hopefully evident that it is possible to create algorithms which are sample-rate consistent, and thus have topology matrix with rank equal to  $s - 1$ , for which no schedule exists.



**Figure 6.4:** Algorithm with non-defective topology matrix which can not be scheduled.

Therefore Equation (6.2) is a necessary condition for a schedule to exist for an algorithm but not a sufficient condition. There exists a type of algorithms referred to as Class S algorithms (S for sequential) which are guaranteed to find a sequential schedule, if such a schedule exists, for algorithms with a non-defective topology matrix [50]. Similar algorithms also exist that can find a parallel schedule [50], however, to derive such schedules, additional information must be inferred from the constructed graphs. This need can also be exemplified with Figure 6.4 in which a sample-rate consistent algorithm is shown and it can readily be found that any invocation vector,  $q$ , of the form  $[n \ n]^T$ , where  $n$  is a non-negative number, satisfies Equation (6.4). Therefore this algorithm should perform as intended if each node is executed  $n$  times. Yet, the algorithm in Figure 6.4 can never execute any of the nodes meaningfully as the loop imposes a deadlock since node A can not execute before node B and similarly for node B it must wait for node A to execute. To express this ordering between nodes, a precedence graph can be constructed. The precedence graph is obtained by removing all arcs with delays. By doing so, a graph that expresses dependencies among nodes is obtained. If this is done to Figure 6.4, the same graph is obtained since there are no delays and from that it can be concluded that for a schedule to exist the precedence graph must be acyclic i.e. contain no loops as that results in deadlock. Similarly for this to be fulfilled it can also be required that any loop in the SDFG must contain one or more delays. In the design of the diagonal processor, the acyclic precedence graph is also identical to the SDFG as seen on Figure 6.5. This is because the SDFG contain no delays, like that of Figure 6.4, however it does not contain any loops, unlike Figure 6.4, and thus deadlock does not occur. As a result the precedence graph is acyclic and a schedule may be found.





**Figure 6.5:** Acyclic precedence graph for diagonal processor. The operation of each node is displayed inside the nodes. Additionally, the label of each nodes corresponding SDFG node is also shown.

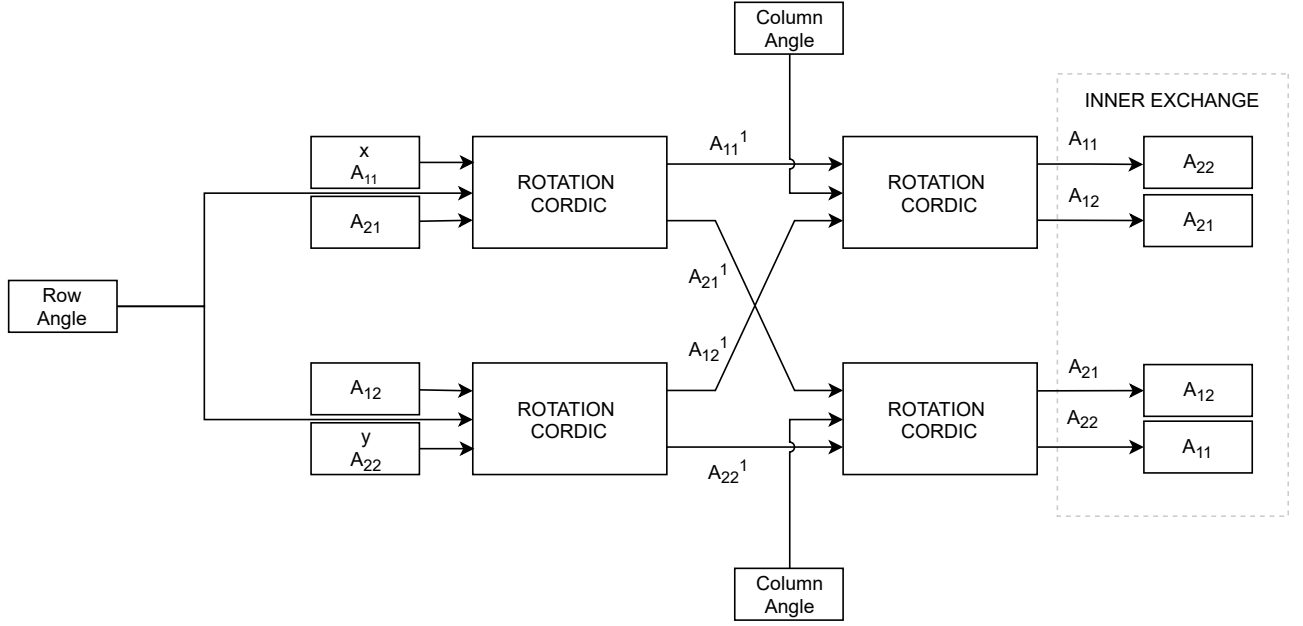
From this precedence graph it is possible to construct a valid schedule by only scheduling a node after all other nodes with incoming arcs have been scheduled. Nodes without incoming arcs may be scheduled at any time. Structured methods for finding such schedules exist [50]. However, the complexity of this scheduling problem is very manageable as maximum concurrency is desired. That is to say that all nodes that can be scheduled at a given time will be scheduled. Additionally, since an isomorphic mapping with no hardware constraints is pursued, any node can also be scheduled without hardware conflicts. As the SDFG contains no loops, it is possible to form a purely combinatorial circuit, where the input signal continuously propagates through all the nodes without being bound by a clock. This is advantageous as it allows for the fastest possible computation time, since the only limiting factor is the propagation delay of the hardware. With these considerations in mind, the final Periodic Admissible Parallel Schedule (PAPS) will be identical to the precedence graph in Figure 6.5.

## 6.2 Off-Diagonal Processor

In this section, the RTL for the off-diagonal processor will be derived. The structure of the section will mostly follow that of the diagonal processor section, since the same approach is used to derive both.

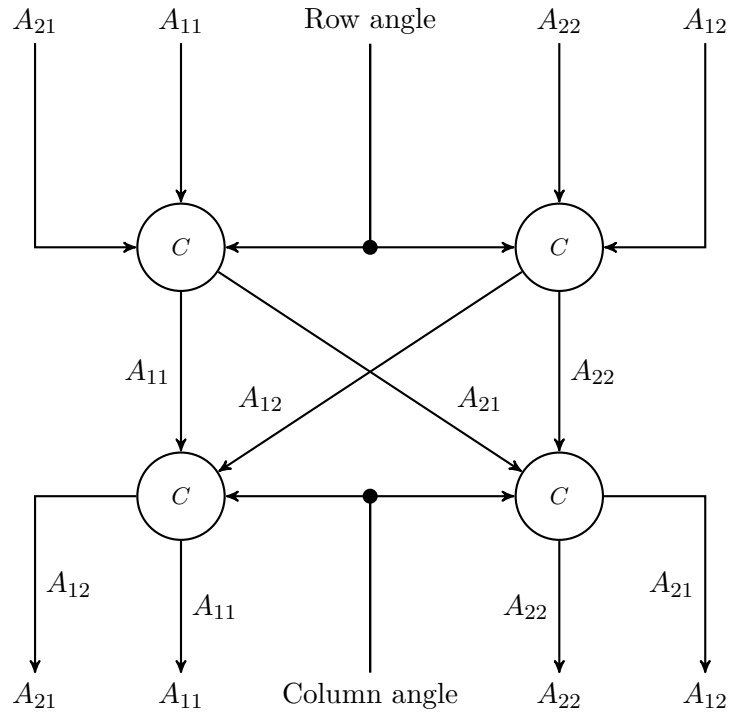
The off-diagonal processor has to perform the same operations as the diagonal processor except that there is no need to calculate the rotation angle. Thus the two processors are almost identical except the off-diagonal processor has no vector-mode CORDIC as seen in Figure 6.6.

## 6.2. Off-Diagonal Processor



**Figure 6.6:** Large Grain Data Flow Graph (LGDFG) for off-diagonal processor.

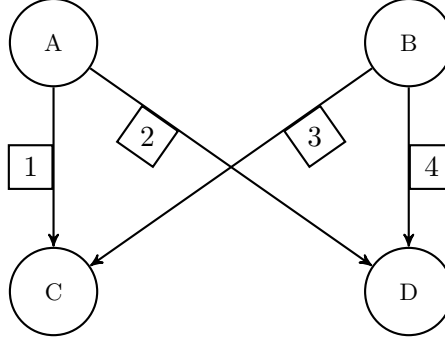
As it can be seen from Figure 6.6, the off-diagonal processor is composed only of CORDIC modules in rotation-mode. From Figure 6.6 the DFG can be constructed:



**Figure 6.7:** Data Flow Graph (DFG) for off-diagonal processor.

Given the DFG the SDFG can be obtained as seen in Figure 6.8.

## 6.2. Off-Diagonal Processor



**Figure 6.8:** Synchronous Data Flow Graph (SDFG) for off-diagonal processor.

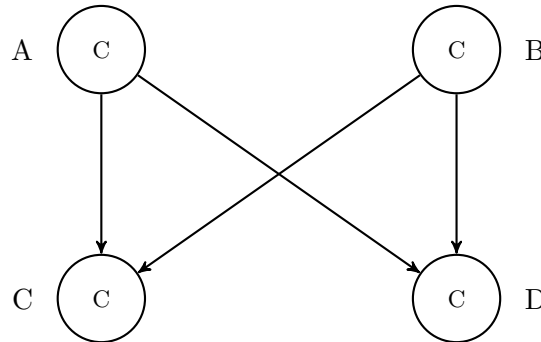
As with the diagonal processor this is not a multiple sample-rate algorithm and therefore sample-rate inconsistency should be very surprising, however as a cross-check, the topology matrix is formed and the rank checked.

$$\Gamma = \begin{matrix} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \end{matrix} \quad (6.6)$$

Since the rank of the topology matrix is equal to 3 and the amount of nodes equal 4, then

$$\text{Rank}(\Gamma) = 3 = 4 - 1 = s - 1 \quad (6.7)$$

Therefore the algorithm is sample-rate consistent and the acyclic precedence graph may be constructed to find a valid schedule. The acyclic precedence graph is seen on Figure 6.9.



**Figure 6.9:** Acyclic precedence graph for off-diagonal processor.

Finding a PAPS that exploits all the intra-iteration parallelism presented by the acyclic precedence graph is done in the same manner as for the diagonal processing element. Thus this PAPS is also equivalent to its precedence graph in Figure 6.9.

## 6.3 CORDIC RTL Derivation

The architecture only contains one type of functional unit - CORDIC. As described in Chapter 4, two variations of CORDIC are required. One in rotation mode used to rotate vectors, and another in vectoring mode is required to determine the rotation angles. Since the two variations share much of the same mathematical background, their RTL mapping is presented in tandem. However, this design process is initiated with CORDIC in rotation mode.

Recall the steps of the CORDIC algorithm in rotation mode from Section 4.3.1. Before execution (compile time), the  $k$ -factor as well as a set of angles are generated. To initiate execution, the algorithm requires three input variables: a rotation angle and the two components of a vector. Once these have been inserted, the algorithm rotates the vector based on the initial state of the input angle. Mathematically this is shown as

$$\sigma^{[i]} = \begin{cases} 1, & \theta^{[i]} \geq 0 \\ -1, & \theta^{[i]} < 0 \end{cases} \quad (6.8)$$

$$x_1^{[i+1]} = x_1^{[i]} - \sigma^{[i]} \cdot x_2^{[i]} \gg i \quad (6.9)$$

$$x_2^{[i+1]} = x_2^{[i]} + \sigma^{[i]} \cdot x_1^{[i]} \gg i \quad (6.10)$$

$$(6.11)$$

As sigma is either equal to -1 or 1, the expression for the vector rotation can be restated without explicit use of multiplication

$$x_1^{[i+1]} = \begin{cases} x_1^{[i]} - x_2^{[i]} \gg i, & \sigma^{[i]} = 1 \\ x_1^{[i]} + x_2^{[i]} \gg i, & \sigma^{[i]} = -1 \end{cases} \quad (6.12)$$

$$x_2^{[i+1]} = \begin{cases} x_2^{[i]} + x_1^{[i]} \gg i, & \sigma^{[i]} = 1 \\ x_2^{[i]} - x_1^{[i]} \gg i, & \sigma^{[i]} = -1 \end{cases} \quad (6.13)$$

To complete the CORDIC iteration, the angle is also updated

$$\theta^{[i+1]} = \begin{cases} \theta^{[i]} - \hat{\theta}^{[i]}, & \sigma^{[i]} = 1 \\ \theta^{[i]} + \hat{\theta}^{[i]}, & \sigma^{[i]} = -1 \end{cases} \quad (6.14)$$

To perform another iteration, the above steps are simply repeated. Once the desired number of iterations have executed, the final values of the rotated vector must be scaled by the pre-generated  $k$ -factor.

In a software paradigm, it would be natural to describe the repetitious nature of the algorithm in terms of a loop. This is portrayed in the pseudo code description in Algorithm 4. Note that the pseudocode description omits the first CORDIC rotation of  $\pm \frac{\pi}{4}$ , as described in Section 4.3.1.

---

**Algorithm 4** CORDIC in Rotation Mode (Rotation range limited to  $\pm \frac{\pi}{4}$ )

---

**Compile time instructions:**

- 1:  $n$  is set as the number of iterations to run CORDIC
- 2:  $\hat{\theta} = [\arctan(2^{-1}), \arctan(2^{-2}), \dots, \arctan(2^{-n})]$  array elements are indexed by  $\hat{\theta}^{[1]}$  to  $\hat{\theta}^{[n]}$
- 3:  $k = \prod_{j=1}^n \frac{1}{\sqrt{(1+2^{-2j})}}$

**Run time instructions:**

- 1:  $x_1$  is set as the first entry of the vector
  - 2:  $x_2$  is set as the second entry of the vector
  - 3:  $\theta$  is set as the angle to rotate the vector by
  - 4: **for**  $i = 1 \dots n$  **do**
  - 5:    $\sigma = \begin{cases} 1, & \theta \geq 0 \\ -1, & \theta < 0 \end{cases}$
  - 6:    $x_{1s} = x_1 \gg i$
  - 7:    $x_{2s} = x_2 \gg i$
  - 8:    $x_1 = \begin{cases} x_1 - x_{2s}, & \sigma = 1 \\ x_1 + x_{2s}, & \sigma = -1 \end{cases}$
  - 9:    $x_2 = \begin{cases} x_2 + x_{1s}, & \sigma = 1 \\ x_2 - x_{1s}, & \sigma = -1 \end{cases}$
  - 10:    $\theta = \begin{cases} \theta - \hat{\theta}^{[i]}, & \sigma = 1 \\ \theta + \hat{\theta}^{[i]}, & \sigma = -1 \end{cases}$
  - 11: **end for**
  - 12:  $x_1 = kx_1$
  - 13:  $x_2 = kx_2$
- 

Another possibility is to omit the loop, and explicitly state the instructions in each iteration. This is equivalent to unrolling the loop in Algorithm 4. A pseudocode representation of this can be seen in Algorithm 5. To improve the readability of the pseudo code, it defines a macro for a single iteration, and then calls this macro the required number of times.

**Algorithm 5** CORDIC in Rotation Mode (Rotation range limited to  $\pm \frac{\pi}{4}$ ) - Unrolled**Compile time instructions:**

- 1:  $n$  is set as the number of iterations to run CORDIC
- 2:  $a = [\arctan(2^{-1}), \arctan(2^{-2}), \dots, \arctan(2^{-n})]$  array elements are indexed by  $a_1$  to  $a_n$
- 3:  $k = \prod_{j=1}^n \frac{1}{\sqrt{(1+2^{-2j})}}$

**Macro definition:**

- 1: **macro:** CORDIC iteration( $i$ ) :
- 2:  $\sigma = \begin{cases} 1, & \theta \geq 0 \\ -1, & \theta < 0 \end{cases}$
- 3:  $x_{1s} = x_1 \gg i$
- 4:  $x_{2s} = x_2 \gg i$
- 5:  $x_1 = \begin{cases} x_1 - x_{2s}, & \sigma = 1 \\ x_1 + x_{2s}, & \sigma = -1 \end{cases}$
- 6:  $x_2 = \begin{cases} x_2 + x_{1s}, & \sigma = 1 \\ x_2 - x_{1s}, & \sigma = -1 \end{cases}$
- 7:  $\theta = \begin{cases} \theta - \hat{\theta}^{[i]}, & \sigma = 1 \\ \theta + \hat{\theta}^{[i]}, & \sigma = -1 \end{cases}$
- 8: **end macro**

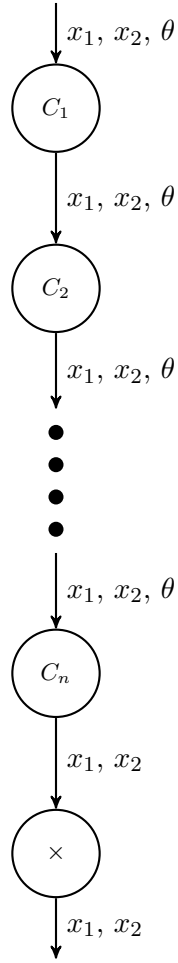
**Run time instructions:**

- 1:  $x_1$  is set as the first entry of the vector
- 2:  $x_2$  is set as the second entry of the vector
- 3:  $\theta$  is set as the angle to rotate the vector by
- 4: CORDIC iteration(1)
- 5: CORDIC iteration(2)
- $\vdots$
- 6: CORDIC iteration( $n$ )
- 7:  $x_1 = kx_1$
- 8:  $x_2 = kx_2$

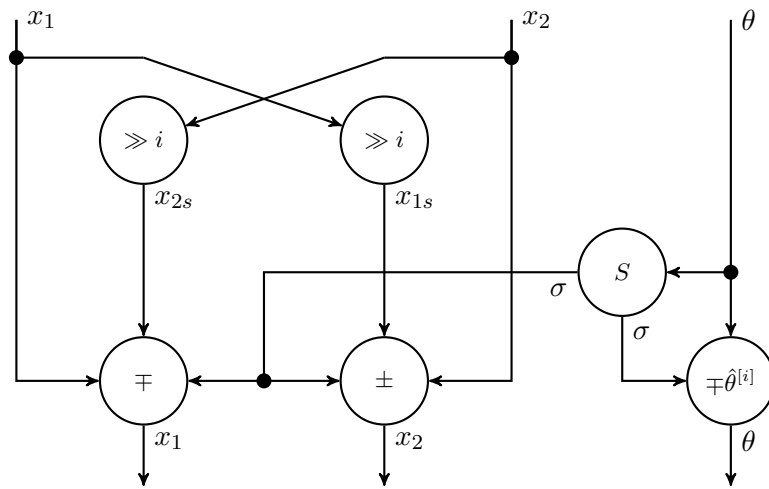
As the description in Algorithm 5 explicitly maps all the mathematical operation of each CORDIC iteration, it is more representative of the desired "1 to 1" mapping, than the looping alternative described in Algorithm 4. Thus, Algorithm 5 will form the basis for the RTL design.

To initiate the design, the pseudocode description is used to construct a DFG of the algorithm. Like in Algorithm 5, it possible to abstract the inner working of each CORDIC iteration. Such a representation is shown in the LGDFG in Figure 6.10. Similarly to the macro definition in Algorithm 5, the CORDIC iterations nodes in the LGDFG are defined in Figure 6.11.

When each of the nodes in the LGDFG are expanded according to Figure 6.11, the resulting DFG is shown in Figure 6.12 To help decipher the DFGs, Table 6.1 defines the meaning of the different nodes used.



**Figure 6.10:** LGDFG of unrolled CORDIC iteration. Each  $C_i$  represents a CORDIC iteration.



**Figure 6.11:** DFG of a single CORDIC iteration.

### 6.3. CORDIC RTL Derivation

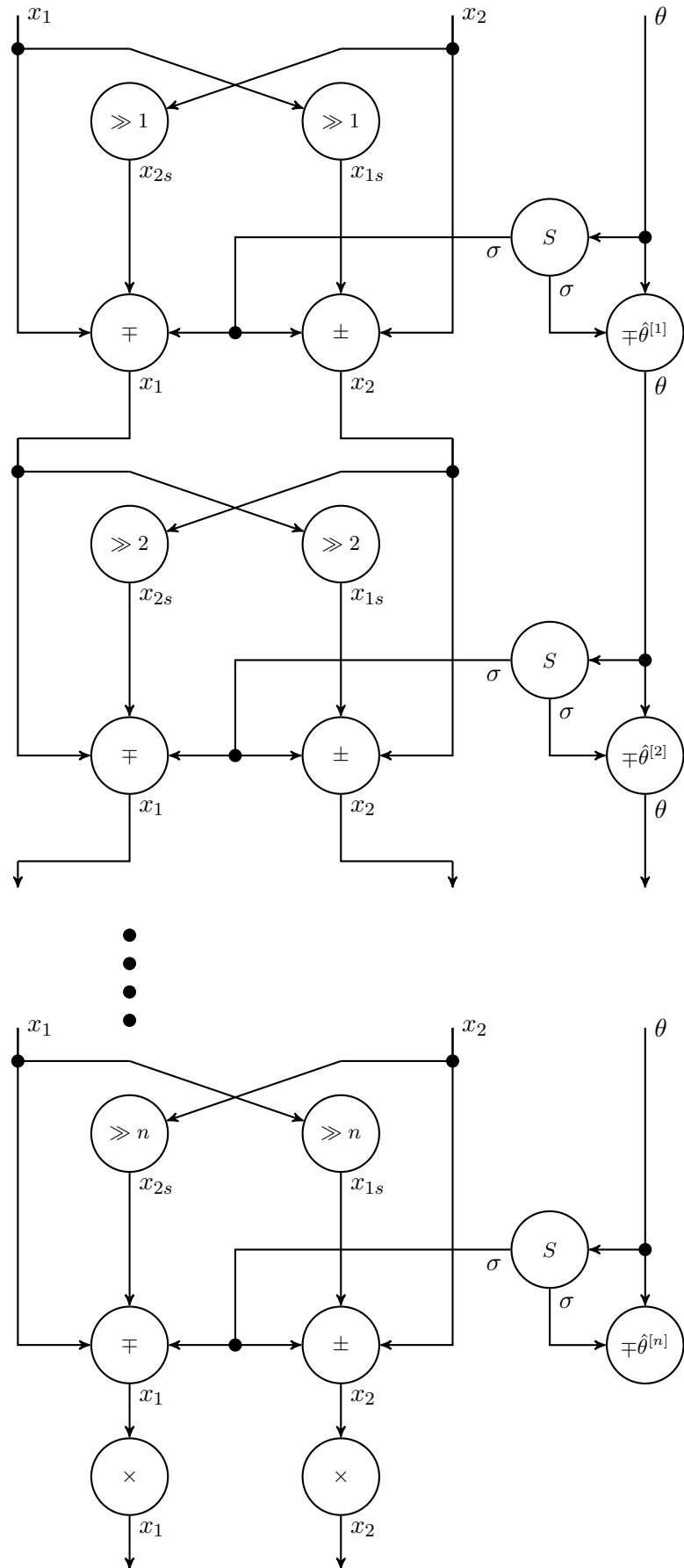


Figure 6.12: DFG of the CORDIC algorithm.



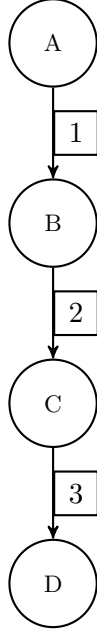
### 6.3. CORDIC RTL Derivation

**Table 6.1:** Symbol Definition of the nodes in Figure 6.11, 6.12, 6.19, and 6.16.

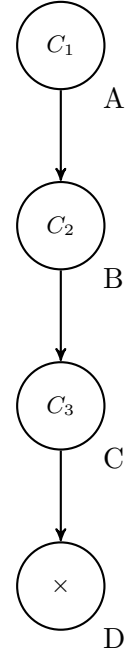
Symbol	Input(s)	Output
$\pm$	$x \in \mathbb{R}, \quad y \in \mathbb{R}, \quad \sigma \in -1, 1$	$\begin{cases} x + y, & \sigma = 1 \\ x - y, & \sigma = -1 \end{cases}$
$\mp$	$x \in \mathbb{R}, \quad y \in \mathbb{R}, \quad \sigma \in -1, 1$	$\begin{cases} x - y, & \sigma = 1 \\ x + y, & \sigma = -1 \end{cases}$
$\mp \hat{\theta}^{[i]}$	$x \in \mathbb{R}, \quad \sigma \in -1, 1$	$\begin{cases} x - \hat{\theta}^{[i]}, & \sigma = 1 \\ x + \hat{\theta}^{[i]}, & \sigma = -1 \end{cases}, \text{ where } \hat{\theta}^{[i]} \text{ is a predetermined angle}$
$>> n$	$x \in \mathbb{R},$	$2^{-n}x$ (bits of $x$ shifted right $n$ times)
$S$	$x \in \mathbb{R}$	$\begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$
$\times$	$x \in \mathbb{R}$	$x \times k$ , where $k$ is some predetermined constant
$D$	$x \in \mathbb{R}, \quad y \in \mathbb{R}$	$\begin{cases} -1, & xy \geq 0 \\ 1, & xy < 0 \end{cases}$

The next step in the design is to construct the algorithms SDFG and associated precedence graph. While it may be intuitive to base these graphs on the expanded DFG in Figure 6.12, doing so risks that the SDFG and DFG grow to an unwieldy size. Instead, the LGDFG is first considered in isolation. If a feasible schedule exists, then the SDFG and precedence graph for its individual nodes i.e the DFG Figure 6.11, are found.

As the size of the LGDFG and its associated SDFG is dictated by the number of iterations to perform, the following presentation limits the iteration count to  $n = 3$ . This is done to keep the figures to a manageable size. Due to the repetitive structure of the LGDFG, it is trivial to convert the LGDFG to an SDFG and its precedence graph. These are presented in Figure 6.13 and Figure 6.14 respectively.



**Figure 6.13:** SDFG of the LGDFG in Figure 6.11. Node A, B, and C each represent a CORDIC iteration. Node D is represents scaling by the k factor.



**Figure 6.14:** Precedence graph of Figure 6.13. Node A, B, and C each represent a CORDIC iteration. Node D is represents scaling by the k factor.

While it is obvious that the SDFG in Figure 6.13 is executable, for the sake of completion, its topology matrix is shown

$$\Gamma = \begin{matrix} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{vmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{vmatrix} \end{matrix} \quad (6.15)$$

The SDFG has 4 nodes, and as the rank of  $\Gamma$  is equal to 3, a feasible schedule may exist. Similarly, based on the structure of the SDFG it follows that there only exist one feasible sequential schedule

$$\text{schedule} = \{A \quad B \quad C \quad D\} \quad (6.16)$$

Additionally, as the nodes are connected in a purely sequential manner, there exists no PAPS. Since there only exists one feasible schedule, this definitively defines how the nodes are to be connected. It is critical to note that while the previous analysis were done for case where  $n = 3$ , it is clear that similar results can be derived for arbitrary  $n$ .

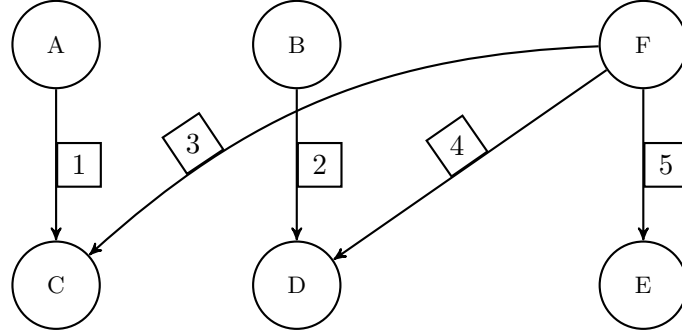
Finally, it is necessary to allocate hardware units based on the above schedule. As it is desired to find a "1 to 1" mapping, dedicated hardware is assigned to each of the nodes in the precedence graph. With this in mind, one scheduling option is to add pipeline registers between each of the nodes in the precedence graphs. This addition could increase the maximum throughput of the circuit by allowing multiple vector rotations to be performed concurrently. However, as the algorithm and architecture requires that one vector rotation is completed before the next one may begin, it is not possible to take advantage of this throughput benefit when doing a "1 to 1" mapping. Additionally,

### 6.3. CORDIC RTL Derivation

as it was previously decided to design processing element with a purely combinatorial data-path, it is not possible to implement a pipeline.

Now that it is defined how each of the CORDIC iterations are to interact, a closer look is taken at each iteration.

First, the DFG in Figure 6.11 is converted to the SDFG in Figure 6.15.

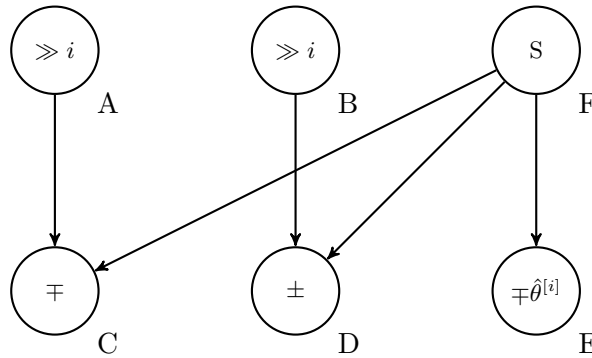


**Figure 6.15:** SDFG of the CORDIC algorithm.

This SDFG has the following topology matrix

$$\Gamma = \begin{matrix} & \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \end{matrix} \quad (6.17)$$

As, the rank of  $\Gamma$  is 5, a PASS exists. Therefore it is justifiable to proceed to find the SDFG's precedence relation. Such a relation is shown by the precedence graph in Figure 6.16. Again, note that it is identical to the previous SDFG.

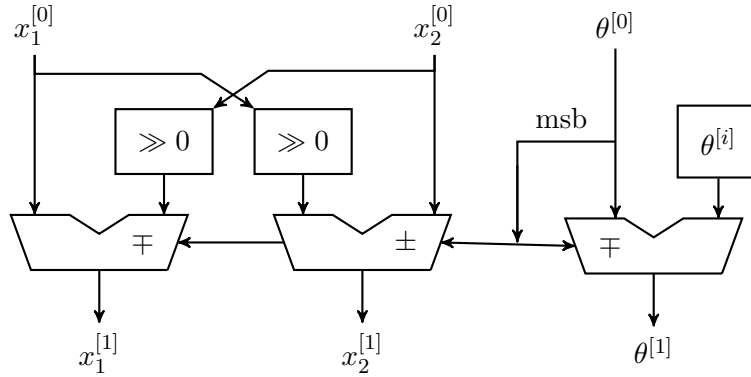


**Figure 6.16:** Precedence graph of the CORDIC algorithm.

Having established the precedence relations, the next step is to perform scheduling and allocation of the required functional units. As it is desired to both take advantage of all available parallelism, and create a "1 to 1" mapping, a dedicated functional unit is allocated for each of the instructions in the precedence graph shown in Figure 6.16. As a combinatorial circuit is desired, all the hardware units' input/output port are simply connected without any dependence on a clock. The RTL diagram of

### 6.3. CORDIC RTL Derivation

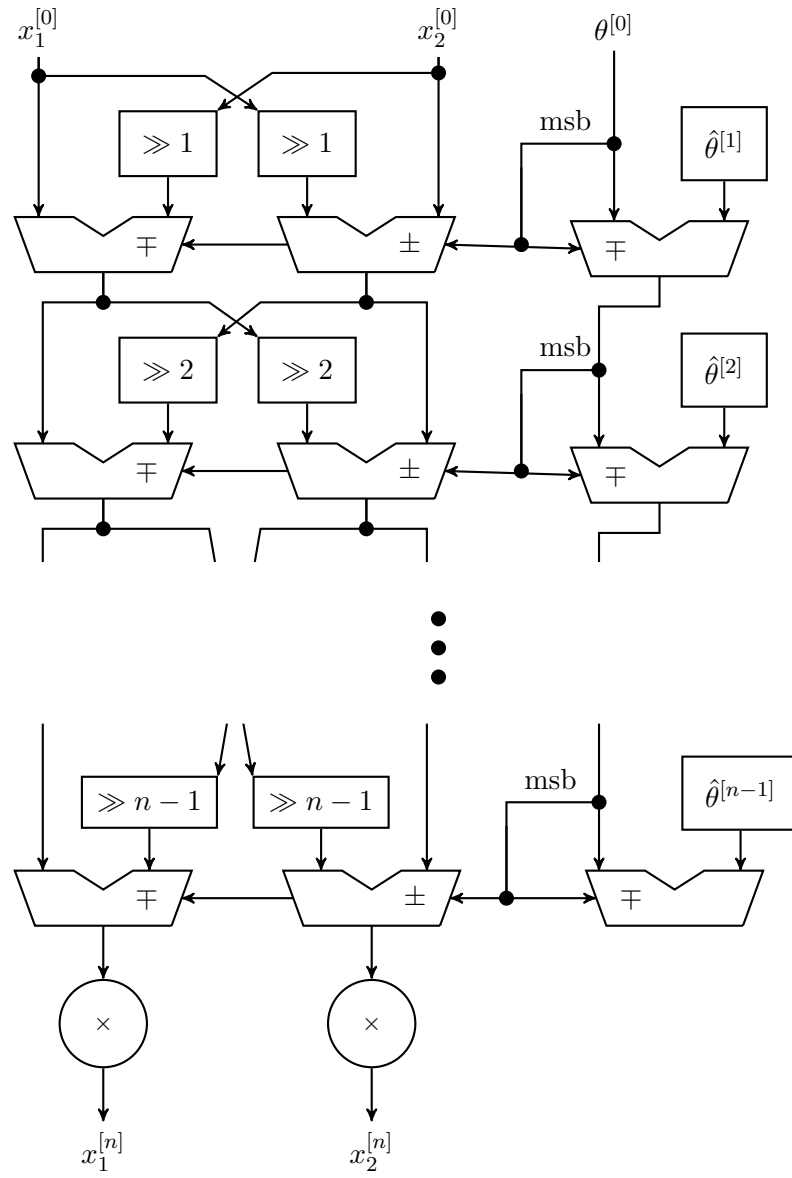
this topology is shown in Figure 6.17. An associated table describing the meaning of each RTL block is shown in Table 6.2. This topology has the advantage that it greatly simplifies the circuits control logic. In fact, as the input signal ripples through each functional unit, no external control circuitry is required.



**Figure 6.17:** RTL of an iteration of CORDIC in rotation mode.

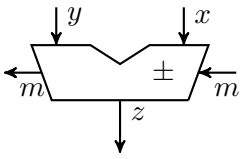
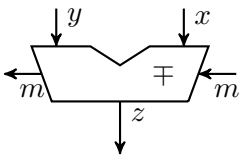
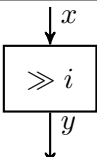
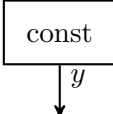

To obtain the final RTL description of the entire CORDIC algorithm, multiple instances of the single iteration CORDIC RTL are chained together. Note that the number of shifts and pre calculated angle differ for each of these iteration instances. Additionally, the final values of  $x_1$  and  $x_2$  must be routed into multipliers to scale them by the required  $k$  factor. With these details in place, the complete RTL description of the unrolled CORDIC algorithm in rotation mode is shown in Figure 6.18.

### 6.3. CORDIC RTL Derivation



**Figure 6.18:** RTL of CORDIC in rotation mode.

**Table 6.2:** RTL Symbol Definition.

Symbol	Input	Output
	<i>x</i> : a number <i>y</i> : a number <i>m</i> : a bit	$z = \begin{cases} x + y, & m = 1 \\ x - y, & m = 0 \end{cases}$
	<i>x</i> : a number <i>y</i> : a number <i>m</i> : a bit	$z = \begin{cases} x - y, & m = 1 \\ x + y, & m = 0 \end{cases}$
	<i>x</i> : a number	$y = x \gg i$
		$y = \text{const}$
	<i>x</i> : a number <i>y</i> : a number	$z = \text{sign bit } x \text{ XOR sign bit } y$

The next task is to derive an RTL description of CORDIC for calculating the arctangent. As this algorithm shares many of the same aspects as the CORDIC algorithm in rotation mode, the following derivation will not be as comprehensive.

Like in the case of CORDIC in rotation mode, as a first step, a description of the algorithm is presented. This is shown in Algorithm 6.

**Algorithm 6** CORDIC in Vectoring mode - Unrolled

---

**Compile time instructions:**

- 1:  $n$  is set as the number of iterations to run CORDIC
- 2:  $a = [\arctan(2^{-0}), \arctan(2^{-1}), \dots, \arctan(2^{-(n-1)})]$  array elements are indexed by  $a_1$  to  $a_n$

**Macro definition:**

- 1: **macro:** Atan CORDIC iteration( $i$ ) :

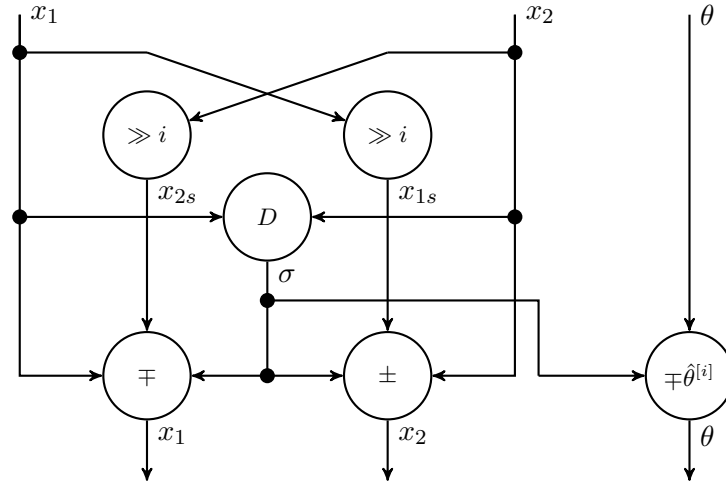
- 2:  $\sigma = \begin{cases} -1, & x_1 x_2 \geq 0 \\ 1, & x_1 x_2 < 0 \end{cases}$
- 3:  $x_{1s} = x_1 \gg i$
- 4:  $x_{2s} = x_2 \gg i$
- 5:  $x_1 = \begin{cases} x_1 - x_{2s}, & \sigma = 1 \\ x_1 + x_{2s}, & \sigma = -1 \end{cases}$
- 6:  $x_2 = \begin{cases} x_2 + x_{1s}, & \sigma = 1 \\ x_2 - x_{1s}, & \sigma = -1 \end{cases}$
- 7:  $\theta = \begin{cases} \theta - \hat{\theta}^{[i]}, & \sigma = 1 \\ \theta + \hat{\theta}^{[i]}, & \sigma = -1 \end{cases}$
- 8: **end macro**

**Run time instructions:**

- 1:  $x_1$  is set as the first entry of the vector
  - 2:  $x_2$  is set as the second entry of the vector
  - 3:  $\theta = 0$
  - 4: Atan CORDIC iteration(0)
  - 5: Atan CORDIC iteration(1)
  - ⋮
  - 6: Atan CORDIC iteration( $n - 1$ )
  - 7:  $\theta \gg 1$
- 

Based on this pseudo code description, the DFG's are formed. Like in the previous derivations, it makes sense to do these in two stages, one higher abstraction LGDFG and one lower abstraction DFG for a single iteration. However, as the control sequences in Algorithm 6 is almost identical to Algorithm 5 the LGDFG is omitted of Algorithm 6. Instead, a DFG of a single iteration is shown in Figure 6.19. To compensate for the lack of an LGDFG for Algorithm 6, a DFG of the entire algorithm is shown in Figure 6.20. Like in the previous CORDIC DFG's, node definitions are found in Table 6.1.

### 6.3. CORDIC RTL Derivation



**Figure 6.19:** DFG of a single iteration of CORDIC in vectoring mode.



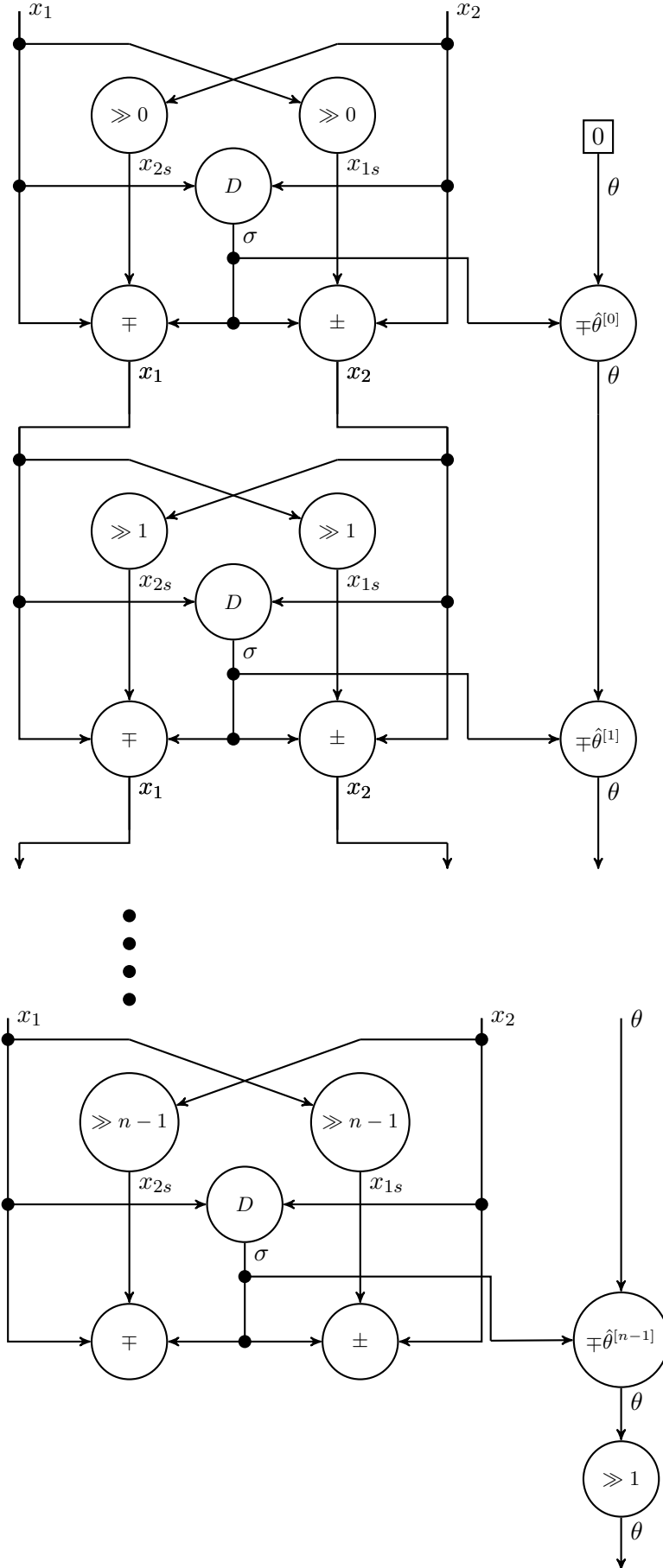
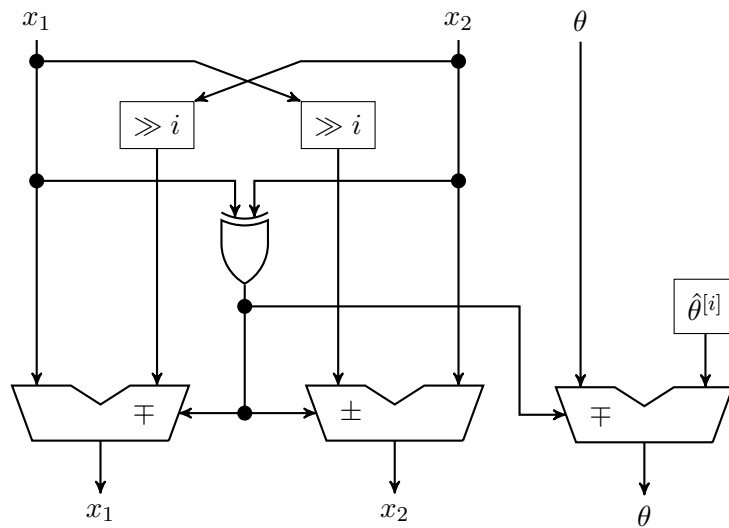


Figure 6.20: DFG of CORDIC in vectoring mode.

### 6.3. CORDIC RTL Derivation

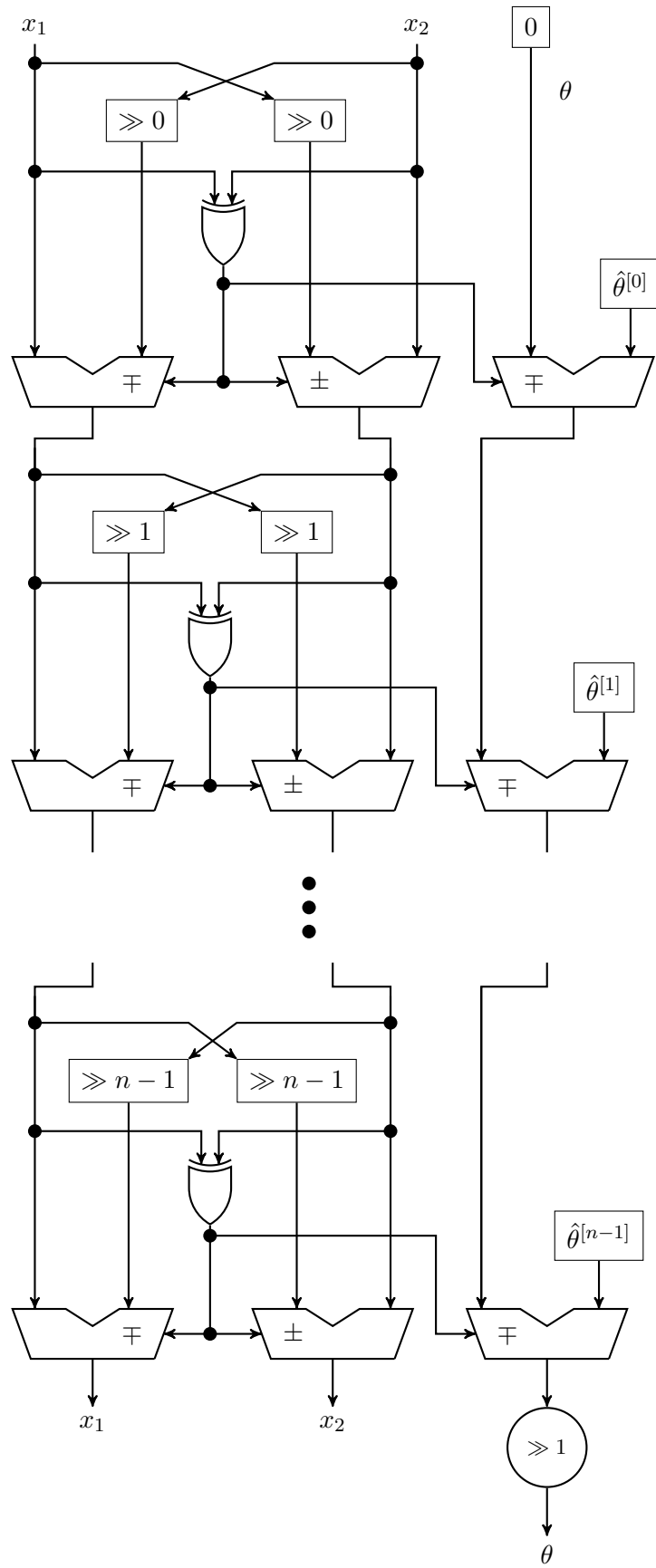
From the pseudocode and DFG descriptions, it is clear that this algorithm shares most of its computations with CORDIC in rotation mode. The three major differences between the two algorithms being the difference in how  $\sigma$  is found, the initial value of  $\theta$ , and the post processing steps where CORDIC in rotation mode scales  $x_1$  and  $x_2$  by a k-factor and the requirements of CORDIC in vectoring modes presented thusfar requires  $\theta$  to be halved. Due to these similarities it turns out that the SDFG, and therefore in extension the topology matrix and precedence graph, of an unrolled CORDIC iteration is identical for both the previous derived CORDIC in rotation mode, and the currently considered CORDIC in vectoring mode. As these graphs are mostly identical, the reader is referred back to Figure 6.15 and Figure 6.16. The only difference is that node F no longer represents operation  $S$ , but operation  $D$  as shown in Table 6.1.

Finally these graphs are used to derive the RTL. Like with the CORDIC in rotation mode, an RTL description that is purely combinatorial is desired. The RTL for a single iteration, and complete CORDIC is shown in Figure 6.21 and Figure 6.22 respectively.



**Figure 6.21:** RTL of an iteration of CORDIC in vectoring mode.

### 6.3. CORDIC RTL Derivation



**Figure 6.22:** RTL of CORDIC in vectoring mode.

## 6.4 RTL for Systolic Array

The design of the processing elements is now combined with the RTL of CORDIC to derive the RTL of the systolic array. However, first the RTL of the two processing elements are shown in Figure 6.23 and Figure 6.24. The RTL is constructed by connecting the derived CORDIC RTL in the order specified by the processing elements precedence graph.

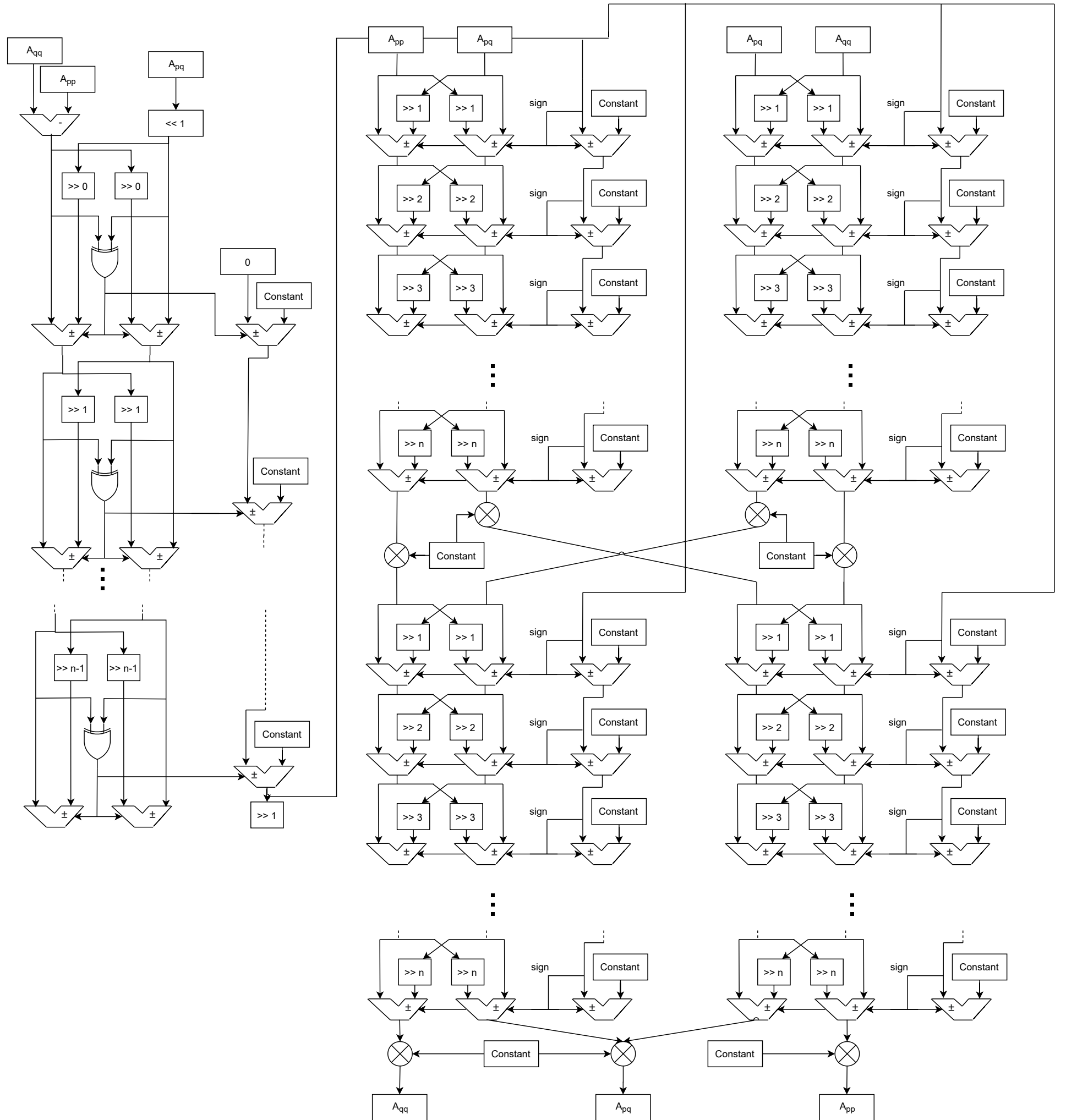


Figure 6.23: RTL for diagonal processor.

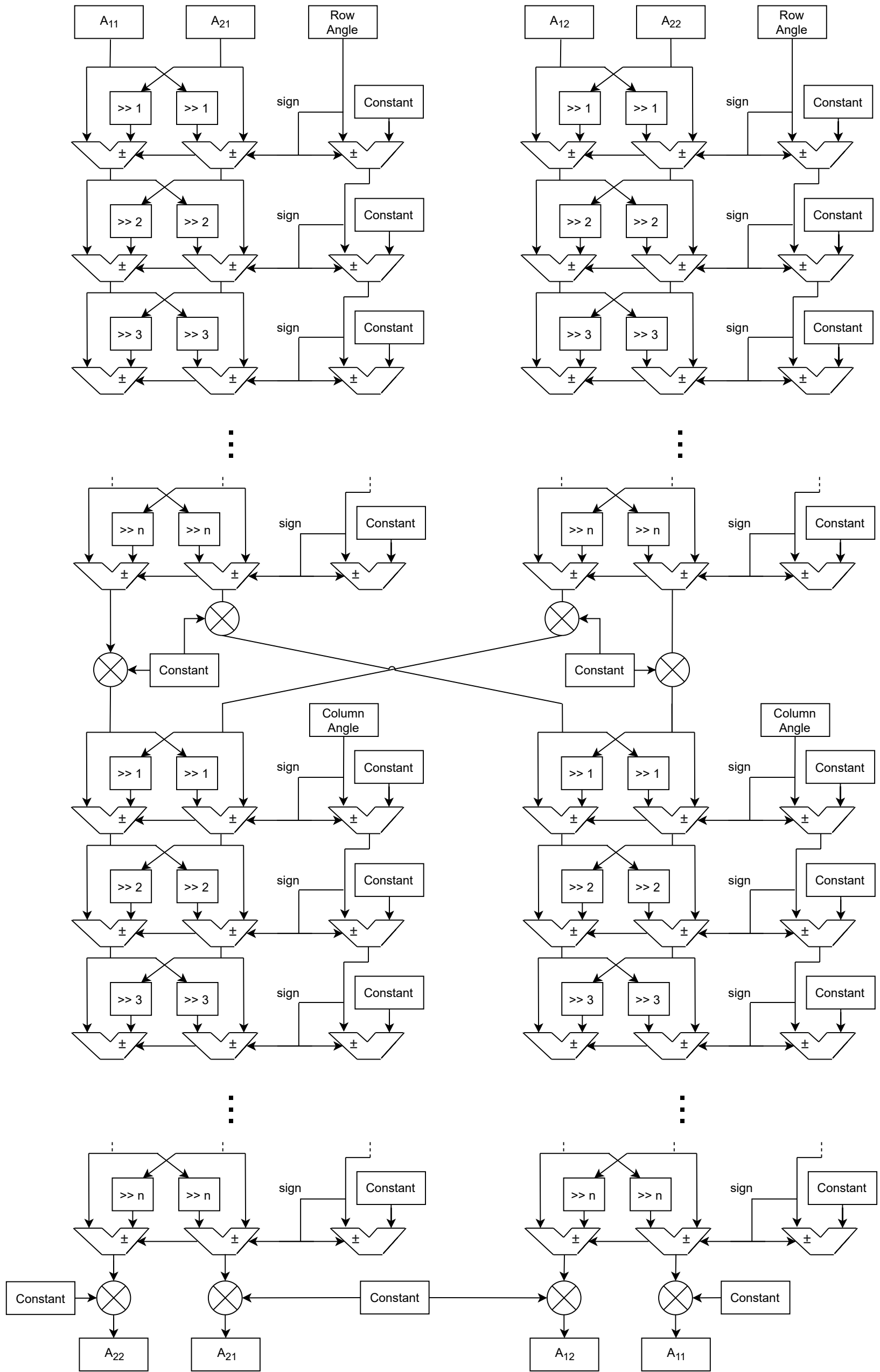
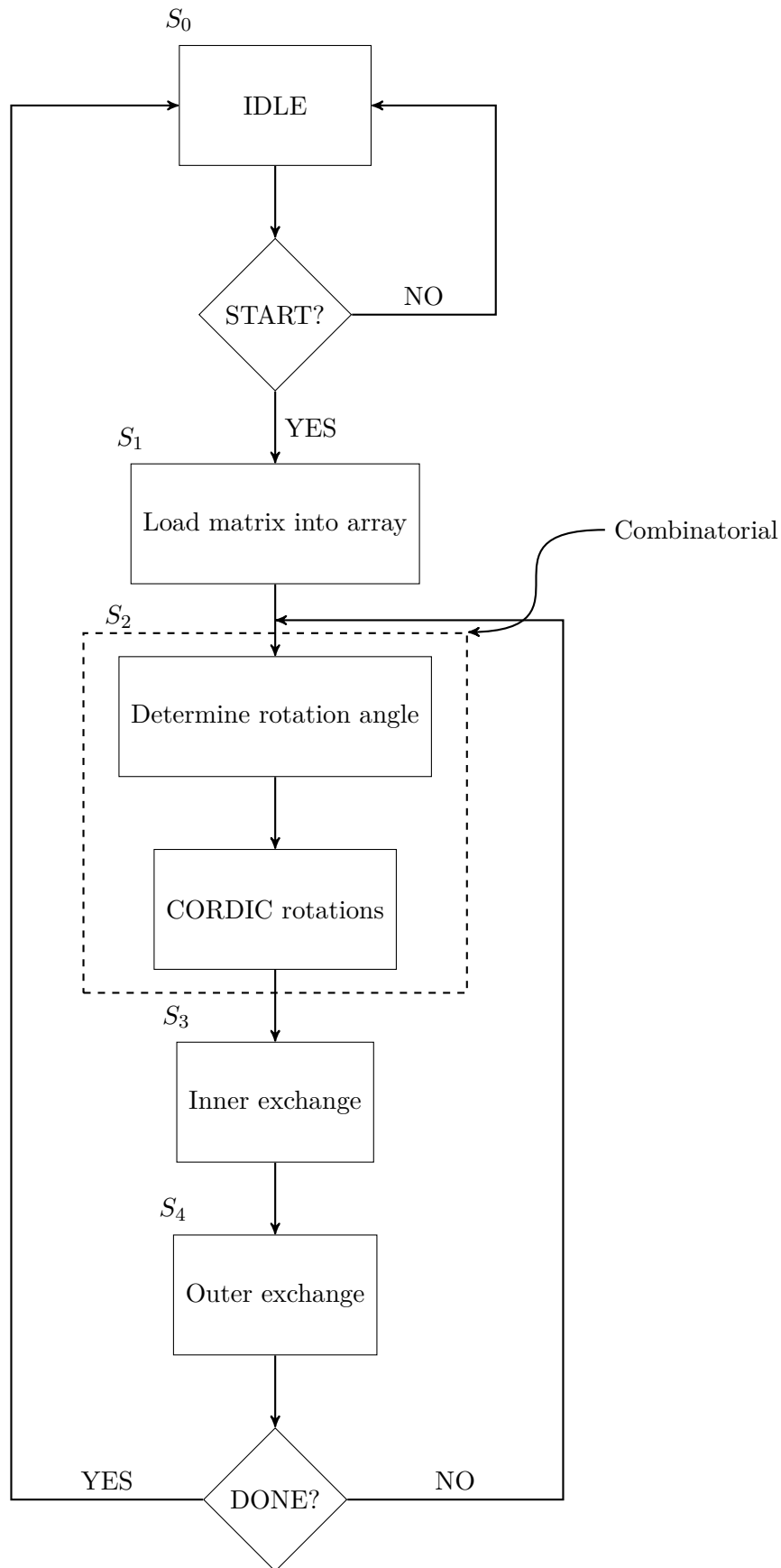


Figure 6.24: RTL of off-diagonal processor.

#### 6.4. RTL for Systolic Array

Having obtained the RTL for both the diagonal- and off-diagonal processors the final step is to form the systolic array using the two obtained processors. To complete the RTL of the systolic array it is also desired to create the control logic needed to control the systolic array. This results in a Finite State Machine with Datapath (FSMD).

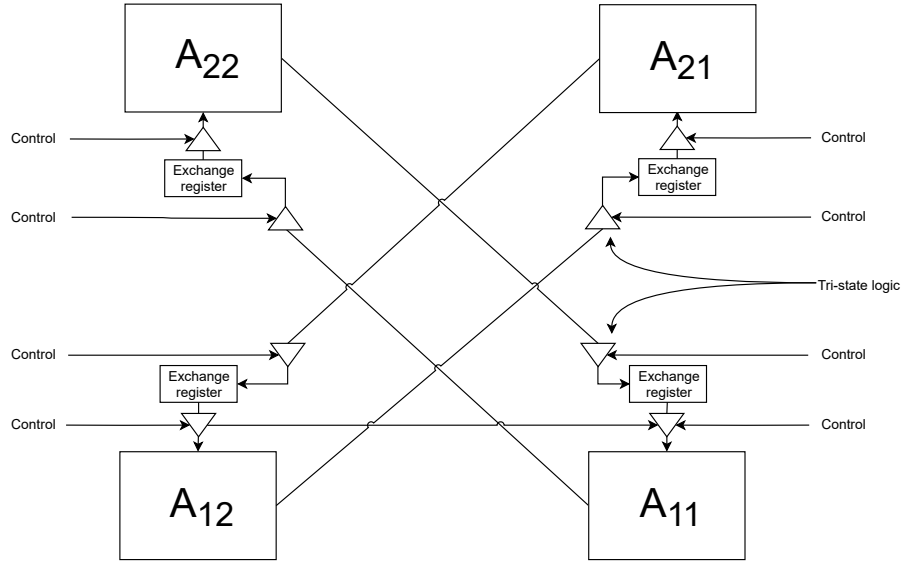
To create this FSMD a Control Data Flow Graph (CDFG) is made to clarify the overall control of the data flow. The CDFG can be seen on Figure 6.25.



**Figure 6.25:** Control Data Flow Graph (CDFG) for systolic array.



As indicated on Figure 6.25 with a dashed box, the process of finding the rotation angle and performing the CORDIC rotations has been made strictly combinatorial by using the unrolled CORDIC architecture and can therefore, from a control logic perspective, be regarded as one operation. The 4 operations are therefore treated as 3. This is also beneficial when considering the dynamics between the diagonal- and off-diagonal processors. Although the off-diagonal processors do not calculate any rotation angle, they receive the angle at the same time as the rotation-mode CORDICs in the diagonal processor and therefore, the two processors finish at the same time and can share the same control logic. The primary task when constructing the RTL for the systolic array is performing the inner- and outer exchanges, such that they fulfill the chess tournament algorithm. The inner exchange have already been incorporated in the processors by simply gating the outputs to the entries specified by Figure 5.13. This can be verified in the data flow graphs shown in the respective processor sections. It is worth noting that the registers containing the matrix entries are written to both in the inner exchange but also in the outer exchange. Therefore some sort of mechanism must ensure that the register is only driven by one at a time to avoid a bus conflict. This can be achieved in several ways using multiplexers, bus arbiters etc. In this design tri-state logic is used. If the tri-state enable port is pulled high, the buffer will allow the input to propagate through to the output and drive the bus. Otherwise it will be in a high-impedance mode, thus not driving the bus. The same principle is applied in the outer exchange. The outer exchange scheme is shown in Figure 5.15 and Figure 6.26 illustrates the RTL for the outer exchange where four entries has to be exchanged.



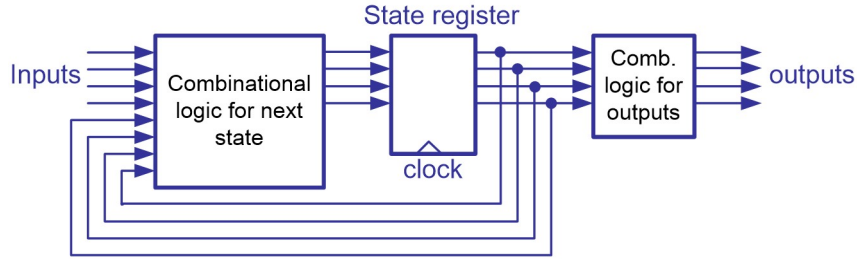
**Figure 6.26:** Outer exchange RTL for systolic array.

Since it is not possible to read and write from the same two registers at the same time, it is necessary to first gate each matrix entry to a temporary exchange register and then from said register to the new matrix entry and thus completing the outer exchange. To operate the systolic array three control signals are needed. One for the inner exchange and two for the outer exchange, such that after the CORDIC rotations have been made the outputs are propagated to the entries dictated by the inner exchange and then following that they are gated twice to perform the outer exchange. This is summarised in the following State-action table.

**Table 6.3:** State-action table for systolic array.

Present State	Condition	Next State	Datapath Action	Control Signals		
				tri <sub>1</sub>	tri <sub>2</sub>	tri <sub>3</sub>
$s_0$	start=0 start=1	$s_0$ $s_1$	IDLE			
$s_1$		$s_2$	Load Matrix			
$s_2$		$s_3$	CORDIC	0	0	1
$s_3$		$s_4$	Inner Exchange	1	0	0
$s_4$	done=0 done=1	$s_2$ $s_1$	Temporary Exchange	0	1	0

To implement the control flow of the FSMD, as expressed by Figure 6.25 and Table 6.3, a Moore machine will be used as opposed to a Mealy machine since the inputs are very simple (start and done) and no inputs from the datapath is needed for the control flow. The general form of a Moore machine is seen in Figure 6.27.

**Figure 6.27:** Illustration of a general Moore machine [51].

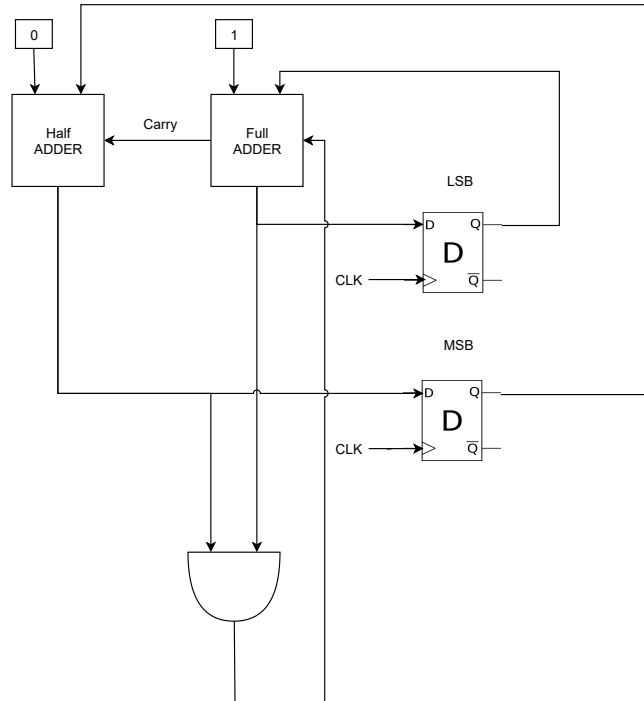
The Moore machine consists of some next-state logic that, based on the input and current state, expressed in the state register, determines the next state as described by the state-action table. Based on the current state, some output logic generates the necessary control signals, as specified in state-action table, needed to operate the datapath. In this design it is assumed that the loading and extraction of data from the systolic array is done separately, such that the only states present in the control flow is states  $s_2$ ,  $s_3$  and  $s_4$ . Since there are three states, the state register must be two bits wide to have a representation for each state. From this and the state-action table, a truth table for the next state logic can be made.

**Table 6.4:** Truth table for next-state logic.

MSB	LSB	Next State Logic	
		MSB	LSB
0	0	0	1
0	1	1	0
1	0	0	0

From this the following RTL is constructed.

#### 6.4. RTL for Systolic Array



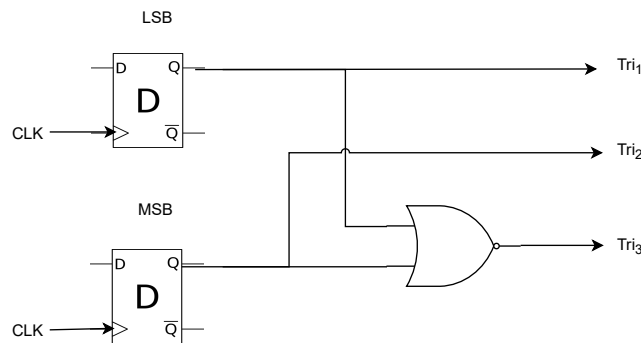
**Figure 6.28:** RTL for next-state logic.

The same may be done for the output logic:

**Table 6.5:** Truth table for the output logic.

MSB	LSB	Next State Logic		
		tri <sub>1</sub>	tri <sub>2</sub>	tri <sub>3</sub>
0	0	0	0	1
0	1	1	0	0
1	0	0	1	0

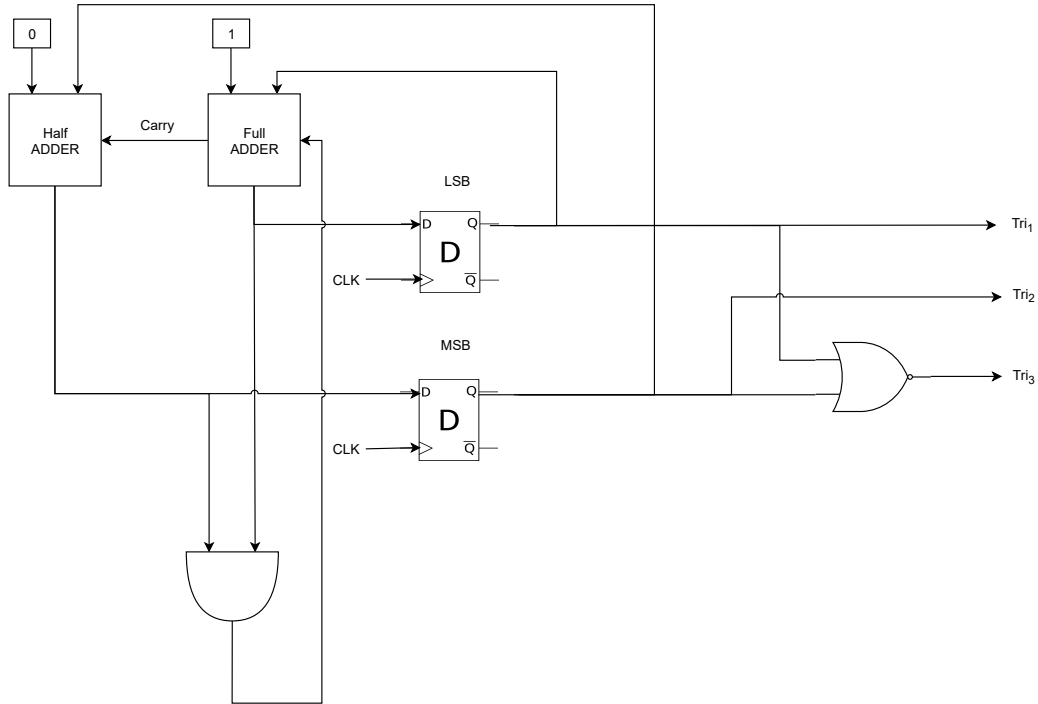
The RTL for the output logic can be seen on Figure 6.29.



**Figure 6.29:** RTL for the output logic.

The entire Moore machine can be seen on Figure 6.30.

#### 6.4. RTL for Systolic Array



**Figure 6.30:** RTL for the Moore machine.

Having obtained the finite state machine, the complete FSMMD may now be constructed using the designed components. That is, the diagonal- and off-diagonal processors and the Moore machine. By connecting the tri-state control signals, the final architecture becomes that shown on Figure 6.31.

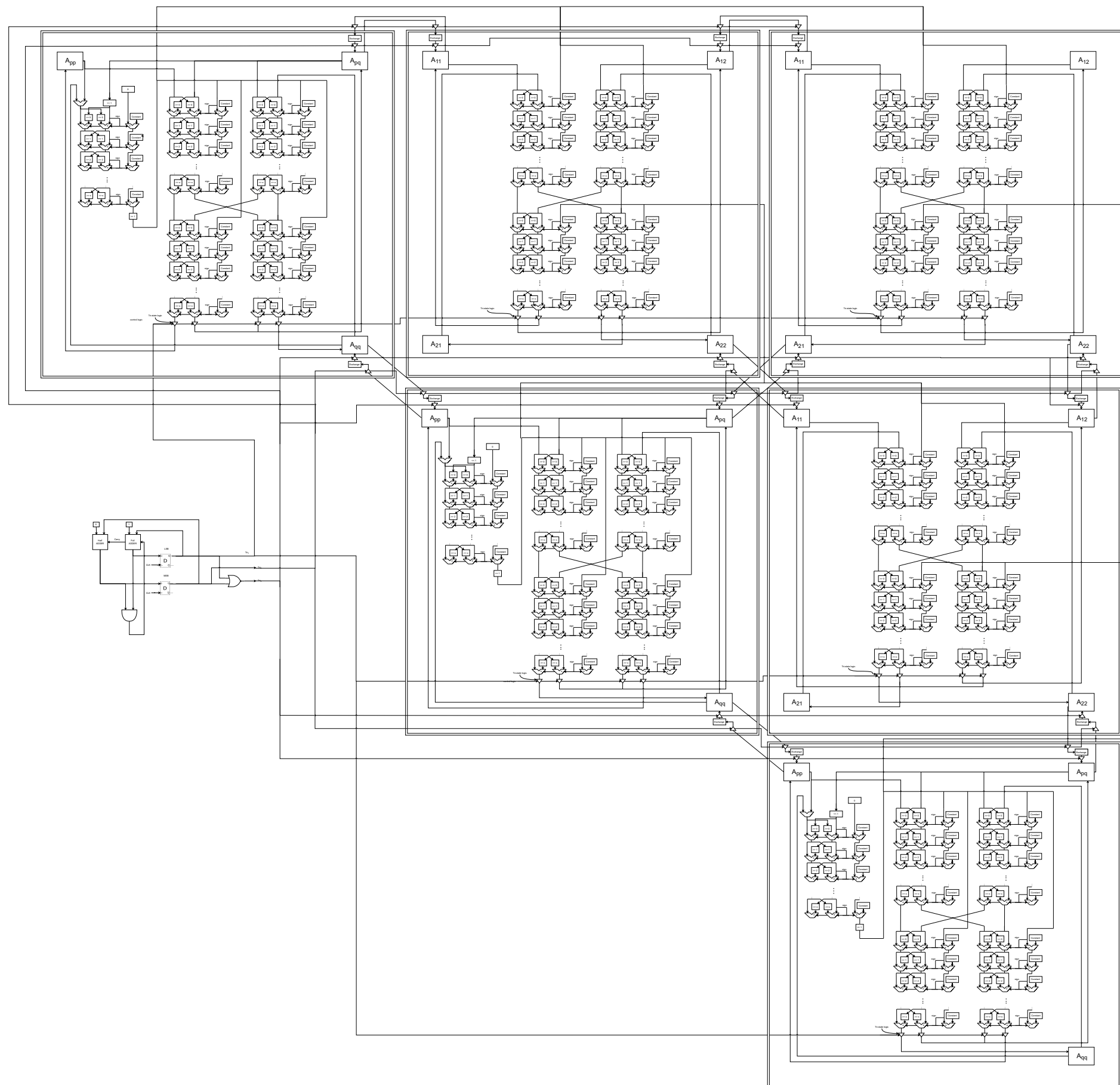


Figure 6.31: RTL for the systolic array.

# Chapter 7

## Discussion

Throughout the project, a custom architecture was developed to solve the symmetric eigenvalue problem. This architecture however is not at a finished stage yet. Some of the remaining tasks and consideration will be discussed in this section.

Firstly, the developed architecture was not implemented. As the main purpose of the semester project focuses on the design of the design of an architecture, rather than the implementation, this is in line with the objectives. The implementation of the architecture is a prerequisite for evaluating the performance. As such, the evaluation of the architecture cannot be documented. Another omission was that of the external I/O capabilities of the architecture, which includes the input of a matrix as well as the output of eigenvalues. Further work is required within the implementation discipline to compare this architecture to other designs.

Even though the architecture at this stage is only capable of computing the eigenvalues of a matrix, changes to include eigenvectors are possible. The decision to work with eigenvalues arose due to time constraints, however given more time the authors would have used the methods described in [43, p. 6] to modify the currently described architecture. This addition requires the use of iterative column rotations on a separate identity matrix. The matrix will converge towards having the eigenvectors as its columns.

In Section 5.4, a new method to determine the rotation direction of the CORDIC algorithm was proposed. This new method alleviates the need for explicit calculation of the arctangent. Instead the rotation direction can be read from the sign of the element of rotation. Using this in the implementation, it is hypothesised that the diagonal processor elements can be simplified to that of the off-diagonal, which results in less overall computation time as well as lower complexity.

The project employed multiple methodologies for the work structure. One of these tools is the  $A^3$ -model and its related cost-function as described in Section 2.1. These tools were used to guide the iterative design process by keeping the authors mindful regarding optimisation criteria through the different levels of the  $A^3$ -model. Before diving straight into the algorithm and architecture of the Jacobi algorithm a series of worksheets were written on the topic of the application specifications. This included a state of the art overview of the most common algorithms used in eigensolvers today as well as selection of different parameters, such as matrix type and output requirements. The Gajski-Kuhn Y-chart was used throughout the design process to narrow the productivity gap. The architectural design was carried out with a focus on data flow, describing the sub-algorithms using data flow and precedence graphs. These allow for better recognition and exploitation of inherent parallelism within the algorithms.

# Chapter 8

## Conclusion

In conclusion, a highly parallelised architecture for the Jacobi algorithm was developed albeit not implemented nor evaluated. The architecture was developed with the cost function in mind, with the main parameter being computational speed. The computation time has been reduced by utilizing inherent parallelism, coming at the significant cost of area and power, which the authors deemed less important in this feasibility study. A point of interest for further work could be the application of a different cost function focusing on reducing area and power consumption, at the expense of accuracy and speed.

The architecture exploits the parallelism of multiple non-conflicting simultaneous rotations to speed up the computation up by a factor  $\frac{n}{2}$ , as compared to a purely sequential approach. To improve the computation speed divisions were avoided. Instead the architecture uses a combinatorial implementation using only adders and predefined bit-shifts. It was discovered that CORDIC processor elements could be used as the only computation unit. Utilizing the CORDIC in a systolic array was used to allow the data transfer between the processing elements reducing the need for a shared data bus, which would otherwise likely become congested.

The authors achieved a one to one mapping of the algorithm which is highly parallelised but is not suited for all applications. Within the scope of this project, no further possible optimisations are identified, however with a different set of tools, the architecture might still be optimised.

During the late stages of work on the CORDIC processor elements of the systolic array, it was discovered that the explicit calculation of the arctangent term could be avoided entirely. As this saves the diagonal CORDIC elements from performing an entire cycle of vectoring mode, it is expected that the mean rotation iteration time can be reduced by approximately  $\frac{1}{3}$ . This change was not implemented or explored further.

Future work includes implementing the architecture to evaluate and compare with other implementations. Another task is to explore and derive an RTL implementation of the alternative method of finding and broadcasting rotation directions to off-diagonal processing elements. This would be done to evaluate its efficiency and possible further parallelism.

To summarise, the goal of creating a hardware architecture better suited for the Jacobi algorithm was achieved. Amongst the points to improve are: implementing and evaluating the architecture as well as removing the explicit calculation of arctangent in CORDIC rotations.

# Bibliography

- [1] X. Zhang and X. Ren, “Two dimensional principal component analysis based independent component analysis for face recognition,” *2011 International Conference on Multimedia Technology*, 2011.
- [2] R. Schmidt, “Multiple emitter location and signal parameter estimation,” *IEEE Transactions on Antennas and Propagation*, vol. 34, no. 3, p. 276–280, 1986.
- [3] B. N. Parlett, *The symmetric eigenvalue problem*. siam, 1998, vol. 20.
- [4] T. J. Lund, Erik & Condra, “Notes and excersises for numerical methods,” August 2018. [Online]. Available: <https://www.moodle.aau.dk/mod/resource/view.php?id=782936>
- [5] L. N. Trefethen and D. I. Bau, *Numerical linear algebra*. SIAM Society for Industrial and Applied Mathematics, 2000.
- [6] D. S. Watkins, “Francis’s algorithm,” *The American Mathematical Monthly*, vol. 118, no. 5, pp. 387–403, 2011. [Online]. Available: <http://www.jstor.org/stable/10.4169/amer.math.monthly.118.05.387>
- [7] J. J. Cuppen, “A divide and conquer method for the symmetric tridiagonal eigenproblem,” *Numerische Mathematik*, vol. 36, no. 2, pp. 177–195, 1980.
- [8] J. J. Dongarra and D. C. Sorensen, “A fully parallel algorithm for the symmetric eigenvalue problem,” *SIAM Journal on Scientific and Statistical Computing*, vol. 8, no. 2, pp. s139–s154, 1987.
- [9] M. Gu and S. C. Eisenstat, “A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem,” *SIAM Journal on Matrix Analysis and Applications*, vol. 16, no. 1, pp. 172–191, 1995.
- [10] I. Dhillon, B. Parlett, and C. Vömel, “The design and implementation of the mrrr algorithm,” *ACM Trans. Math. Softw.*, vol. 32, pp. 533–560, 12 2006.
- [11] M. Petschow, “Mrrr-based eigensolvers for multi-core processors and supercomputers,” 01 2014.
- [12] I. S. Dhillon and B. N. Parlett, “Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices,” *Linear Algebra and its Applications*, vol. 387, pp. 1 – 28, 2004. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S002437950300908X>
- [13] J. Badía and A. Vidal, “Parallel bisection algorithms for solving the symmetric tridiagonal eigenproblem,” *High Performance Algorithms for Structured Matrix Problems*, 1998.
- [14] P. Koch, “Reconfigurable & low energy systems lecture 1.”
- [15] I. Griffin, “Example: Gajski-kuhn y-chart,” Dec 2009. [Online]. Available: <http://texample.net/tikz/examples/gajski-kuhn-y-chart/>
- [16] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*, 1st ed. Springer Publishing Company, Incorporated, 2009.
- [17] J. W. Demmel, *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.
- [18] A. Matthes, “Matrix multiplication illustration.” [Online]. Available: <https://altermundus.fr/articles/art-matrix-multiplication.php>
- [19] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. USA: Johns Hopkins University Press, 1996.



- [20] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*. USA: Oxford University Press, Inc., 1988.
- [21] S. W. Watkins, D., *Fundamentals of Matrix Computations, 3rd Edition*. John Wiley and Sons, Inc., 2010.
- [22] H. P. Kempen, "On the quadratic convergence of the special cyclic jacobi method," *Numer. Math.*, vol. 9, no. 1, p. 19–22, Nov. 1966. [Online]. Available: <https://doi.org/10.1007/BF02165225>
- [23] M. Berry and A. Sameh, "An overview of parallel algorithms for the singular value and symmetric eigenvalue problems," *Journal of Computational and Applied Mathematics*, vol. 27, no. 1-2, p. 191–213, 1989.
- [24] R. P. Brent and F. T. Luk, *The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays*. Cornell University. Department of Computer Science, 1983.
- [25] A. H. Sameh, "On jacobi and jacobi-like algorithms for a parallel computer," *Mathematics of computation*, vol. 25, no. 115, pp. 579–590, 1971.
- [26] J. W. Demmel, M. T. Heath, and H. A. Van Der Vorst, "Parallel numerical linear algebra," *Acta numerica*, vol. 2, pp. 111–197, 1993.
- [27] F. T. Luk and H. Park, "On parallel jacobi orderings," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 1, pp. 18–26, 1989.
- [28] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, p. 268–308, Sep. 2003. [Online]. Available: <https://doi.org/10.1145/937503.937505>
- [29] S. Rajan, S. Wang, R. Inkol, and A. Joyal, "Efficient approximations for the arctangent function," *IEEE Signal Processing Magazine*, vol. 23, no. 3, p. 108–111, May 2006.
- [30] H. A. Medina, "A sequence of polynomials for approximating arctangent," *The American Mathematical Monthly*, vol. 113, no. 2, pp. 156–161, 2006.
- [31] J.-P. Deschamps, G. J. A. Bioul, and G. D. Sutter, *Synthesis of arithmetic circuits: FPGA, ASIC and embedded systems*. John Wiley, 2006.
- [32] J. Volder, "The cordic computing technique," in *Papers presented at the the March 3-5, 1959, western joint computer conference*, 1959, pp. 257–261.
- [33] P. Pirsch, *Architectures for digital signal processing*. John Wiley & Sons, Inc., 1998.
- [34] E. V. Bonet, *VHDL User's Forum in Europe*. Ed. Universidad de Cantabria, 1997.
- [35] Y. H. Hu, "The quantization effects of the cordic algorithm," *IEEE Transactions on signal processing*, vol. 40, no. 4, pp. 834–844, 1992.
- [36] T. E. Oliphant, "Numpy v1.18 manual: numpy.linalg.eig," 2 2020. [Online]. Available: <https://numpy.org/doc/1.18/reference/generated/numpy.linalg.eig.html>
- [37] H.-T. Kung, "Why systolic architectures?" *Computer*, no. 1, pp. 37–46, 1982.
- [38] J. A. Fortes and B. W. Wah, "Systolic arrays-from concept to implementation," *Computer;(United States)*, vol. 20, no. 7, 1987.
- [39] A. Ahmedsaid, A. Amira, and A. Bouridane, "Improved svd systolic array and implementation on fpga," in *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, 2003, pp. 35–42.

- [40] Tao Wang and Ping Wei, “Hardware efficient architectures of improved jacobi method to solve the eigen problem,” in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 6, April 2010, pp. V6–22–V6–25.
- [41] Yang Liu, C. . Bouganis, P. Y. K. Cheung, P. H. W. Leong, and S. J. Motley, “Hardware efficient architectures for eigenvalue computation,” in *Proceedings of the Design Automation Test in Europe Conference*, vol. 1, 2006, pp. 1–6.
- [42] J.-M. Delosme, “Bit-level systolic algorithms for real symmetric and hermitian eigenvalue problems,” *VLSI Signal Processing*, vol. 4, pp. 69–88, 02 1992.
- [43] M. U. Torun, O. Yilmaz, and A. N. Akansu, “Fpga, gpu, and cpu implementations of jacobi algorithm for eigenanalysis,” *Journal of Parallel and Distributed Computing*, vol. 96, pp. 172–180, 2016.
- [44] J. Fortes, K. Fu, and B. Wah, “Systematic approaches to the design of algorithmically specified systolic arrays,” in *ICASSP '85. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 10, 1985, pp. 300–303.
- [45] S. Kung, “Vlsi array processors,” *IEEE ASSP Magazine*, vol. 2, no. 3, pp. 4–22, 1985.
- [46] D. I. Moldovan, “On the design of algorithms for vlsi systolic arrays,” *Proceedings of the IEEE*, vol. 71, no. 1, pp. 113–120, 1983.
- [47] M. Torun, O. Yilmaz, and A. Akansu, “Fpga, gpu, and cpu implementations of jacobi algorithm for eigenanalysis,” *Journal of Parallel and Distributed Computing*, vol. 96, 05 2016.
- [48] R. P. Brent and F. T. Luk, “The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays,” *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 69–84, 1985.
- [49] E. W. Weisstein, “Harmonic addition theorem.” May 2020. [Online]. Available: <https://mathworld.wolfram.com/HarmonicAdditionTheorem.html>
- [50] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987.
- [51] Y. Yajun, “7. modeling at the fsmd level¶,” 2019. [Online]. Available: <https://sustechvhdl.readthedocs.io/lecture/chapter7.html>

# Appendix A

## Derivation of equations equivalent to $JAJ^T$

This appendix aims to derive a set of equations for the elements of the matrix resulting from the  $JAJ^T$  similarity transformation.

Recall from Section 3.2 that the similarity transformation can be split into two separate matrix multiplications:

$$X = JA^{[k]} \quad (\text{A.1})$$

$$A^{[k+1]} = XJ^T \quad (\text{A.2})$$

The first elements of  $X$  can then be expressed as:

$$X_{p,i} = \eta A_{p,i} - \sigma A_{q,i} \quad (\text{A.3})$$

$$X_{q,i} = \sigma A_{p,i} + \eta A_{q,i} \quad (\text{A.4})$$

$$X_{i,j} = A_{i,j} \quad \text{for } i \neq p, q \quad (\text{A.5})$$

Similarly the elements of  $A^{[k+1]}$  can be expressed in terms of the elements of  $X$  as:

$$A_{i,p}^{[k+1]} = \eta X_{i,p} - \sigma X_{i,q} \quad (\text{A.6})$$

$$A_{i,q}^{[k+1]} = \sigma X_{i,p} + \eta X_{i,q} \quad (\text{A.7})$$

$$A_{i,j}^{[k+1]} = X_{i,j} \quad \text{for } i \neq p, q \quad (\text{A.8})$$

Separately, these equations only allows one to express  $A^{[k+1]}$  in terms of  $X$ , and  $X$  to be expressed in terms of  $A^{[k]}$ . To directly express  $A^{[k+1]}$  in terms of  $A^{[k]}$ , the two sets of equations must be combined.

Based on the equations, we note that the both rotations,  $JA$  and  $A'J^T$ , will modify indicies  $(p, p)$ ,  $(q, q)$ ,  $(p, q)$ , and  $(q, p)$ . Thus these four entries will be changed twice.

From this, we can infer nine different possibilities:

1.  $(p, q)$
2.  $(q, p)$
3.  $(p, p)$
4.  $(q, q)$
5.  $(p, i)$  for  $i \neq p, q$
6.  $(i, p)$  for  $i \neq p, q$
7.  $(q, i)$  for  $i \neq p, q$
8.  $(i, q)$  for  $i \neq p, q$

9. The rest which remain unchanged

Only option 1-4 is modified by both rotations.

We now consider each of these possibilities and expand Equation (A.6) and Equation (A.7) by Equation (A.3) and Equation (A.4) to get:

$$A_{p,q}^{[k+1]} = \sigma X_{p,p} + \eta X_{p,q} \quad (\text{from (A.7) and (A.3)}) \quad (\text{A.9})$$

$$= \sigma(\eta A_{p,p} - \sigma A_{q,p}) + \eta(\eta A_{p,q} - \sigma A_{q,q}) \quad (\text{A.10})$$

$$= \eta\sigma A_{p,p} - \sigma^2 A_{q,p} + \eta^2 A_{p,q} - \eta\sigma A_{q,q} \quad (\text{A.11})$$

$$= (\eta^2 - \sigma^2) \underbrace{A_{q,p}}_{\text{Due to symmetry}} + (A_{p,p} - A_{q,q})\eta\sigma \quad (\text{A.12})$$

$$A_{q,p}^{[k+1]} = \eta X_{q,p} - \sigma X_{q,q} \quad (\text{from (A.6) and (A.4)}) \quad (\text{A.13})$$

$$= \eta(\sigma A_{p,p} + \eta A_{q,p}) - \sigma(\sigma A_{p,q} + \eta A_{q,q}) \quad (\text{A.14})$$

$$= \eta\sigma A_{p,p} + \eta^2 A_{q,p} - \eta\sigma A_{p,q} - \sigma^2 A_{q,q} \quad (\text{A.15})$$

$$= (\eta^2 - \sigma^2) \underbrace{A_{q,p}}_{\text{Due to symmetry}} + (A_{p,p} - A_{q,q})\eta\sigma \quad (\text{A.16})$$

$$A_{p,p}^{[k+1]} = \eta X_{p,p} - \sigma X_{p,q} \quad (\text{from (A.6) and (A.3)}) \quad (\text{A.17})$$

$$= \eta(\eta A_{p,p} - \sigma A_{q,p}) - \sigma(\eta A_{p,q} - \sigma A_{q,q}) \quad (\text{A.18})$$

$$= \eta^2 A_{p,p} - \eta\sigma A_{q,p} - \sigma\eta A_{p,q} + \sigma^2 A_{q,q} \quad (\text{A.19})$$

$$= \eta^2 A_{p,p} - 2\eta\sigma \underbrace{A_{q,p}}_{\text{Due to symmetry}} + \sigma^2 A_{q,q} \quad (\text{A.20})$$

$$A_{q,q}^{[k+1]} = \sigma X_{q,p} + \eta X_{q,q} \quad (\text{from (A.7) and (A.4)}) \quad (\text{A.21})$$

$$= \sigma(\sigma A_{p,p} + \eta A_{q,p}) + \eta(\sigma A_{p,q} + \eta A_{q,q}) \quad (\text{A.22})$$

$$= \sigma^2 A_{p,p} + \sigma\eta A_{q,p} + \eta\sigma A_{p,q} + \eta^2 A_{q,q} \quad (\text{A.23})$$

$$= \sigma^2 A_{p,p} + 2\sigma\eta \underbrace{A_{p,q}}_{\text{Due to symmetry}} + \eta^2 A_{q,q} \quad (\text{A.24})$$

$$A_{p,i}^{[k+1]} = \eta A_{p,i} - \sigma A_{q,i} \quad \forall i \neq p, q \quad (\text{from (A.3)}) \quad (\text{A.25})$$

$$A_{i,p}^{[k+1]} = \eta A_{i,p} - \sigma A_{i,q} \quad \forall i \neq p, q \quad (\text{from (A.6)}) \quad (\text{A.26})$$

$$A_{q,i}^{[k+1]} = \sigma A_{p,i} + \eta A_{q,i} \quad \forall i \neq p, q \quad (\text{from (A.4)}) \quad (\text{A.27})$$

$$A_{i,q}^{[k+1]} = \sigma A_{i,p} + \eta A_{i,q} \quad \forall i \neq p, q \quad (\text{from (A.7)}) \quad (\text{A.28})$$

Due to the symmetry of  $A$ , it is easy to see that the following entries are numerically equivalent:  
 $A''_{i,q} = A''_{q,i}$ ,  $A''_{i,p} = A''_{p,i}$ ,  $A''_{p,q} = A''_{q,q}$ .