

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

# U·M·I

University Microfilms International  
A Bell & Howell Information Company  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
313/761-4700 800/521-0600



Order Number 1345319

**Architectural, numerical and implementation issues in the VLSI  
design of an integrated CORDIC-SVD processor**

Kota, Kishore, M.S.

Rice University, 1991

**U·M·I**  
300 N. Zeeb Rd.  
Ann Arbor, MI 48106



RICE UNIVERSITY

**Architectural, Numerical and Implementation  
Issues in the VLSI Design of an Integrated  
CORDIC-SVD Processor**

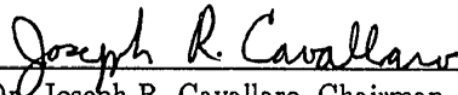
by

**Kishore Kota**

A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

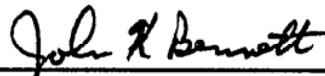
**Master of Science**

APPROVED, THESIS COMMITTEE:



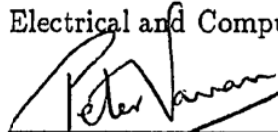
---

Dr. Joseph R. Cavallaro, Chairman  
Assistant Professor  
Electrical and Computer Engineering



---

Dr. John K. Bennett,  
Assistant Professor  
Electrical and Computer Engineering



---

Dr. Peter J. Varman  
Associate Professor  
Electrical and Computer Engineering

Houston, Texas

April, 1991

# Architectural, Numerical and Implementation Issues in the VLSI Design of an Integrated CORDIC-SVD Processor

Kishore Kota

## Abstract

This thesis describes the design of a systolic array for computing the Singular Value Decomposition (SVD) based on the Brent, Luk, Van Loan array. The use of COordinate Rotation DIgital Computer (CORDIC) arithmetic results in an efficient VLSI implementation of the processor that forms the basic unit of the array. A six-chip custom VLSI chip set for the processor was initially designed, fabricated in a  $2.0\mu$  CMOS n-well process, and tested. The CORDIC Array Process Element (CAPE), a single chip implementation, incorporates several enhancements based on a detailed error analysis of fixed-point CORDIC. The analysis indicates a need to normalize input values for inverse tangent computations. This scheme was implemented using a novel method that has  $O(n^{1.5})$  hardware complexity. Use of previous techniques to implement such a normalization would require  $O(n^2)$  hardware. Enhanced architectures, which reduce idle time in the array either through pipelining or by improving on a broadcast technique, are also presented.

## Acknowledgments

Special thanks are due to Dr. Joe Cavallaro for being the motivation and a constant driving force behind this work. I am especially thankful to him for being a good friend and just being there when I needed him most. I would also like to thank Dr. John Bennett and Dr. Peter Varman for serving on the committee and having the patience to deal with me.

My friend and constant companion “Ψχσ” Hemkumar deserves special praise for all those enlightening discussions and valuable comments, which on more than one occasion prevented me from taking the wrong decisions. Thanks are due to my roommate, Raghu, for his constant, but often futile efforts at waking me up. But for him I would have slept through the entire semester without a care. Thanks to the *Common Pool* for allowing me to work through even the most difficult times without compromising on the food. Special thanks to my office-mates Jay Greenwood and Jim Carson. The office would not be half as lively without them.

Finally I am eternally indebted to my mother and father. Even when they are halfway across the world, nothing is possible without their blessings.

To my Grandfather,

For his one statement that has been the driving force through the years:

*"... hope for the best, but be prepared for the worst ..."*



# Contents

Abstract	ii
Acknowledgments	iii
List of Tables	viii
List of Illustrations	ix
<b>1 Introduction</b>	<b>1</b>
1.1 Systolic Arrays for SVD	2
1.2 Contributions of the thesis	3
1.3 Overview of the thesis	3
<b>2 SVD Array Architecture</b>	<b>5</b>
2.1 Introduction	5
2.2 The SVD-Jacobi Method	6
2.3 Direct 2-Angle Method	8
2.4 Architecture for $2 \times 2$ SVD	12
<b>3 CORDIC Techniques</b>	<b>14</b>
3.1 Introduction	14
3.2 CORDIC Algorithms	15
3.3 CORDIC Operation in the Circular Mode	16
3.3.1 CORDIC Z-Reduction	19

3.3.2	CORDIC Y-Reduction . . . . .	20
3.3.3	Modified Y-Reduction for the SVD processor . . . . .	22
3.4	Scale Factor Correction . . . . .	23
3.5	Error Analysis of CORDIC Iterations . . . . .	23
3.6	Error Analysis of CORDIC Z-Reduction . . . . .	25
3.6.1	Error Introduced by X and Y Iterations . . . . .	26
3.6.2	Error Introduced by Z Iterations . . . . .	28
3.6.3	Effects of perturbations in $\theta$ . . . . .	29
3.7	Error Analysis of CORDIC Y-Reduction . . . . .	30
3.7.1	Error Introduced by X and Y Iterations . . . . .	31
3.7.2	Error Introduced by Z Iterations . . . . .	32
3.7.3	Perturbation in the Inverse Tangent . . . . .	32
3.8	Normalization for CORDIC Y-Reduction . . . . .	33
3.8.1	Single Cycle Normalization . . . . .	35
3.8.2	Partial Normalization . . . . .	36
3.8.3	Choice of a Minimal Set of Shifts . . . . .	39
3.8.4	Cost of Partial Normalization Implementation . . . . .	43
3.9	Summary . . . . .	43
<b>4</b>	<b>The CORDIC-SVD Processor Architecture</b>	<b>46</b>
4.1	Introduction . . . . .	46
4.2	Architecture of the Prototype . . . . .	46
4.2.1	Design of the XY-Chip . . . . .	48
4.2.2	Design of the Z-Chip . . . . .	50
4.2.3	Design of the Intra-Chip . . . . .	50
4.2.4	Design of the Inter-Chip . . . . .	54

4.3	The CORDIC Array Processor Element . . . . .	55
4.4	Issues in Loading the Array . . . . .	60
4.4.1	Data Loading in CAPE . . . . .	61
<b>5</b>	<b>Architectures to Improve Processor Utilization</b>	<b>64</b>
5.1	Introduction . . . . .	64
5.2	Architecture for Systolic Starting . . . . .	65
5.3	Architecture for fast SVD . . . . .	70
5.4	Architecture for Collection of U and V Matrices . . . . .	73
<b>6</b>	<b>VLSI Implementation Issues</b>	<b>75</b>
6.1	Design Methodology . . . . .	75
6.2	Reducing Skews in Control Signals . . . . .	78
6.3	Design of an Adder . . . . .	79
6.4	Testing . . . . .	80
<b>7</b>	<b>Conclusions</b>	<b>85</b>
7.1	Summary . . . . .	85
7.2	Future Work . . . . .	87
	<b>Bibliography</b>	<b>88</b>

## Tables

2.1	Parallel Ordering . . . . .	8
3.1	CORDIC Functionality in the Circular Mode . . . . .	19
3.2	Total Shifts Using a Novel Normalization Scheme . . . . .	41
3.3	Summary and Comparison of Error Analysis . . . . .	44
6.1	A list of some of the problems in the 6-chip prototype . . . . .	81
6.2	Breakdown of clock cycles for the various sub-tasks in the 6-chip prototype . . . . .	83
6.3	Breakdown of clock cycles for the various sub-tasks in CAPE . . . . .	83

## Illustrations

2.1	The Brent-Luk-Van Loan SVD Array . . . . .	9
2.2	Snapshots of Brent-Luk-Van Loan SVD Array . . . . .	11
3.1	Rotation of a vector . . . . .	17
3.2	Implementation of the CORDIC Iterations in Hardware . . . . .	18
3.3	Rotation of a vector in Y-Reduction . . . . .	21
3.4	Error in Y-Reduction Without Normalization . . . . .	34
3.5	Algorithm to Invoke Modified CORDIC Iterations . . . . .	40
3.6	Implementation of the CORDIC Iterations in Hardware . . . . .	41
3.7	Error in Y-Reduction With Partial Normalization . . . . .	42
4.1	Block Diagram of the 6-chip Prototype CORDIC-SVD Processor . . . . .	47
4.2	Die Photograph of the XY-Chip . . . . .	49
4.3	Die Photograph of the Z-Chip . . . . .	51
4.4	Block Diagram of the Intra-Chip . . . . .	52
4.5	Die Photograph of the Intra-Chip . . . . .	53
4.6	Block Diagram of Inter-Chip . . . . .	55
4.7	Die Photograph of the Inter-Chip . . . . .	56
4.8	Internal Architecture of CAPE . . . . .	58

4.9	A Combined DSP-VLSI Array for Robotics . . . . .	62
5.1	Interconnection of Control Signals for Systolic Starting . . . . .	66
5.2	Interconnection of Data Lines for Systolic Starting . . . . .	67
5.3	Snapshots of a Systolic Starting scheme . . . . .	69
5.4	Hardware Required for Fast SVD . . . . .	71
5.5	Snapshots of an Array Computing $U$ and $V$ . . . . .	73
6.1	Plot of the completed Single Chip Implementation . . . . .	77
6.2	Clock Generation with Large Skew . . . . .	78
6.3	Clock Generation with Small Skew . . . . .	78

# Chapter 1

## Introduction

Rapid advances made in VLSI and WSI technologies have led the way to an entirely different approach to computer design for real-time applications, using special-purpose architectures with custom chips. By migrating some of the highly compute-intensive tasks to special-purpose hardware, performance which is typically in the realm of supercomputers, can be obtained at only a fraction of the cost. High-level Computer-Aided Design (CAD) tools for VLSI silicon compilation, allow fast prototyping of custom processors by reducing the tedium of VLSI design. Special-purpose architectures are not burdened by the problems associated with general computers and can map the algorithmic needs of a problem to hardware. Extensive parallelism, pipelining and use of special arithmetic techniques tailored for the specific application lead to designs very different from conventional computers.

Systolic and wavefront arrays form a class of special-purpose architectures that hold considerable promise for real-time computations. Systolic arrays are characterized by “multiple use of each data item, extensive concurrency, a few types of simple cells and a simple and regular data and control flow between neighbors” [21]. Systolic arrays typically consist of only a few types of simple processors, allowing easy prototyping. The near-neighbor communication associated with systolic arrays results in short interconnections that allow higher operating speeds even in large arrays. Thus, systolic arrays offer the potential for scaling to very large arrays. Numerical problems

involving large matrices benefit from this property of systolic arrays.

Many real-time signal processing, image processing and robotics applications require fast computations involving large matrices. The high throughput required in these applications can in general, be obtained only through special-purpose architectures. A computationally complex numerical problem, which has evoked a lot of interest is the Singular Value Decomposition (SVD) [15]. It is generally acknowledged that the SVD is the only generally reliable method for determining the rank of a matrix numerically. Solutions to the complex problems encountered in real-time processing, which use the SVD, exhibit a high degree of numerical stability. SVD techniques handle rank deficiency and ill-conditioning of matrices elegantly, obviating the need for special handling of these cases. The SVD is a very useful tool, for example, in analyzing data matrices from sensor arrays for adaptive beamforming [30], and low rank approximations to matrices in image enhancement [2]. The wide variety of applications for the SVD coupled with its computational complexity justify dedicating hardware to this computation.

### **1.1 Systolic Arrays for SVD**

The SVD problem has evoked a lot of attention in the past decade. Numerous architectures and algorithms were proposed for computing the SVD in an efficient manner. The Hestenes' method using Jacobi transformations has been a popular choice among researchers, due to the concurrency it allows in the computation of the SVD. Numerous algorithms and architectures for the SVD were proposed by Luk [24, 26, 25, 23]. Brent, Luk and Van Loan [5] presented an expandable square array of simple processors to compute the SVD. A processor architecture for the Brent, Luk and Van Loan array, using CORDIC techniques, was presented by Cavallaro [8].



## 1.2 Contributions of the thesis

A detailed study of the numerical accuracy of fixed-point CORDIC modules is provided in this thesis. This analysis indicates a need to normalize the values for CORDIC Y-reduction in order to achieve meaningful results. A novel normalization scheme for fixed-point CORDIC Y-reduction that reduces the hardware complexity is developed.

This thesis is chiefly concerned with the issues relating to the VLSI implementation of the CORDIC-SVD array proposed by Cavallaro. Many of the lower-level architectural issues that are not relevant at the higher levels assume a new significance when trying to implement the array. Elegant solutions have been found for all the problems encountered in the design process.

An integrated VLSI chip to implement the  $2 \times 2$  SVD processor element, which serves as the basic unit of the SVD array, has been designed. A new internal architecture has been developed within the constraints imposed by VLSI.

Numerous array architectures that improve on the previous architectures were developed as part of this research. These schemes can be used in future implementations as improvements to the current design.

## 1.3 Overview of the thesis

Chapter 2 presents the SVD algorithm and the array proposed by Brent, Luk and Van Loan [5]. A discussion of the CORDIC-SVD processor is also provided. Chapter 3 reviews the CORDIC arithmetic technique and then presents an improved analysis of the numerical accuracy of fixed-point CORDIC. Chapter 4 presents the improvements made to the architecture of the single chip VLSI implementation of the SVD processor over the various sub-units that constitute the prototype SVD processor. Chapter 5 presents improvements to the array architecture and the impact of these

on the processor design. These improvements include schemes for systolic starting and architectures that reduce idle time in the array, to achieve higher throughput.

## Chapter 2

### SVD Array Architecture

#### 2.1 Introduction

This chapter introduces the principles that govern the efficient computation of the Singular Value Decomposition (SVD) in a parallel array. A major portion of the work in this thesis is based on the Brent, Luk and Van Loan [5] array for computing the SVD. Later sections of this chapter provide insight into the operation of this array.

The SVD has proved to be an important matrix decomposition with theoretical and practical significance. The canonical representation of a matrix provided by the SVD is very useful in determining its properties. The SVD [15] of a  $p \times p$  matrix  $A$  is defined as

$$A = U\Sigma V^T, \quad (2.1)$$

where,

$U$  and  $V$  are *orthogonal* matrices of dimension  $p \times p$ , and

$\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_p)$  is a diagonal matrix of *singular values*.

The columns of  $U$  and  $V$  are called the *left* and the *right* singular vectors respectively.

The SVD is a compute intensive operation requiring  $O(p^3)$  time on a sequential computer. For example, the SVD of an  $8 \times 8$  matrix on a SUN 3/60 requires about 1 second of CPU time using LINPACK [12], a library of linear algebra routines. A typical real-time application, the inverse kinematics engine of a robot, requires the

computation of the SVD of an  $8 \times 8$  matrix every  $400\mu\text{sec}$ . This is precisely the class of applications where the use of dedicated arrays is attractive. With a square array of processors it is possible to reduce the time complexity for the SVD to  $O(p \log p)$ .

## 2.2 The SVD-Jacobi Method

The Jacobi method to compute the SVD is a popular algorithm for parallel implementation, due to the concurrency it introduces to the problem. The algorithm is based on the use of orthogonal transformations, called *Jacobi Rotations*<sup>1</sup> to selectively reduce a given matrix element to zero. A Jacobi rotation by an angle  $\theta$  in the  $ij$  plane, is a  $p \times p$  matrix,  $J(i, j, \theta)$ , where,

$$\begin{aligned} j_{aa} &= 1 & \forall (a \neq i, j), \\ j_{ii} &= \cos \theta, & j_{ij} = \sin \theta, \\ j_{ji} &= -\sin \theta, & j_{jj} = \cos \theta, \\ & & \text{and all other } j_{ab} = 0. \end{aligned} \tag{2.2}$$

Intuitively, a Jacobi rotation is an identity matrix with four of its elements replaced by a vector rotation matrix as shown below:

$$J(i, j, \theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos \theta & \cdots & \sin \theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\sin \theta & \cdots & \cos \theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} i \\ j \end{matrix} \tag{2.3}$$

The parameters  $i$  and  $j$  select the matrix element that is to be reduced to zero, while the parameter  $\theta$  is computed using a few matrix elements. Since very few

---

<sup>1</sup>also called *Givens Rotations* or simply *plane rotations*

matrix elements are required to compute the entire transformation, local computation is possible by storing all the required data in the registers of a single processor. In addition, pre-multiplication with a Jacobi rotation matrix, called *left-sided rotation*, modifies only the  $i^{\text{th}}$  and  $j^{\text{th}}$  rows, while a right-sided rotation affects only the  $i^{\text{th}}$  and  $j^{\text{th}}$  columns. This introduces concurrency, since computations which use the unmodified data elements can be performed in parallel.

The SVD algorithm consists of a sequence of orthogonal <sup>2</sup> *Jacobi transformations*, chosen to diagonalize the given matrix. The algorithm to compute the SVD is iterative and can be written as

$$A_0 = A, \quad (2.4)$$

$$A_{k+1} = U_k^T A_k V_k = J_k(i, j, \theta_l)^T A_k J_k(i, j, \theta_r). \quad (2.5)$$

Appropriate choice of the 2-sided rotations can force  $A_k$  to converge to the diagonal matrix of singular values  $\Sigma$ . The matrix of left singular vectors is obtained as a product of the left rotations  $U_k$ , while the matrix of right singular vectors is obtained as a product of the right-sided rotations. At each iteration  $k$ , a left rotation of  $\theta_l$  and a right rotation of  $\theta_r$  in the plane  $ij$  are chosen to reduce the two off-diagonal elements  $a_{ij}$  and  $a_{ji}$  of  $A_k$  to zero. A sweep consists of  $n(n-1)/2$  iterations that reduce every pair of off-diagonal elements to zero. Several sweeps are required to actually reduce the initial matrix to a diagonal matrix. The sequence, in which the off-diagonal pairs are annihilated within each sweep is chosen apriori according to some fixed ordering. Forsythe-Henrici [14] proved that the algorithm converges for the *cyclic-by-rows* and *cyclic-by-columns* ordering. However, these orderings do not allow decoupling of successive iterations, thereby restricting the parallelism achievable. Brent and Luk [4] introduced the *parallel* ordering that does not suffer from this drawback.

---

<sup>2</sup>A matrix is orthogonal when its transpose equals the inverse

(1,2)	(3,4)	(5,6)	(7,8)
(1,4)	(2,6)	(3,8)	(5,7)
(1,6)	(4,8)	(2,7)	(3,5)
(1,8)	(6,7)	(4,5)	(2,3)
(1,7)	(8,5)	(6,3)	(4,2)
(1,5)	(7,3)	(8,2)	(6,4)
(1,3)	(5,2)	(7,4)	(8,6)

**Table 2.1: Parallel Ordering data exchange with 8 elements. This is the ordering used when eight chess players have to play a round-robin tournament, each player playing one game a day.**

An iteration that annihilates  $a_{11}$  and  $a_{22}$  does not affect any of the terms required for annihilating  $a_{33}$  and  $a_{44}$ , allowing the two sets of rotation angles to be computed concurrently. Thus parallel ordering allows decoupling of the computation of the Jacobi transformations of all the iterations in a single row of the ordering (Table 2.1). Brent, Luk and Van Loan [5] proposed a square array of processors based on this scheme. The ordering has an additional property that it requires only near-neighbor communication in an array and is hence amenable to systolic implementation.

### 2.3 Direct 2-Angle Method

The Brent, Luk and Van Loan systolic array [5] consists of a  $p/2 \times p/2$  array of processors (Figure 2.1) to compute the SVD of a  $p \times p$  matrix. The matrix dimension,  $p$ , is assumed to be even. Each processor stores a  $2 \times 2$  submatrix and has the ability to compute the SVD of a  $2 \times 2$  matrix. The special structure of the Jacobi rotation matrix allows computing the angles for a  $p \times p$  two-sided rotation in terms of a basic  $2 \times 2$  rotation. The application of the transformation on a  $p \times p$  matrix can also be expressed as a set of  $2 \times 2$  transformations. Thus the  $2 \times 2$  SVD forms the basic step for the  $p \times p$  SVD.

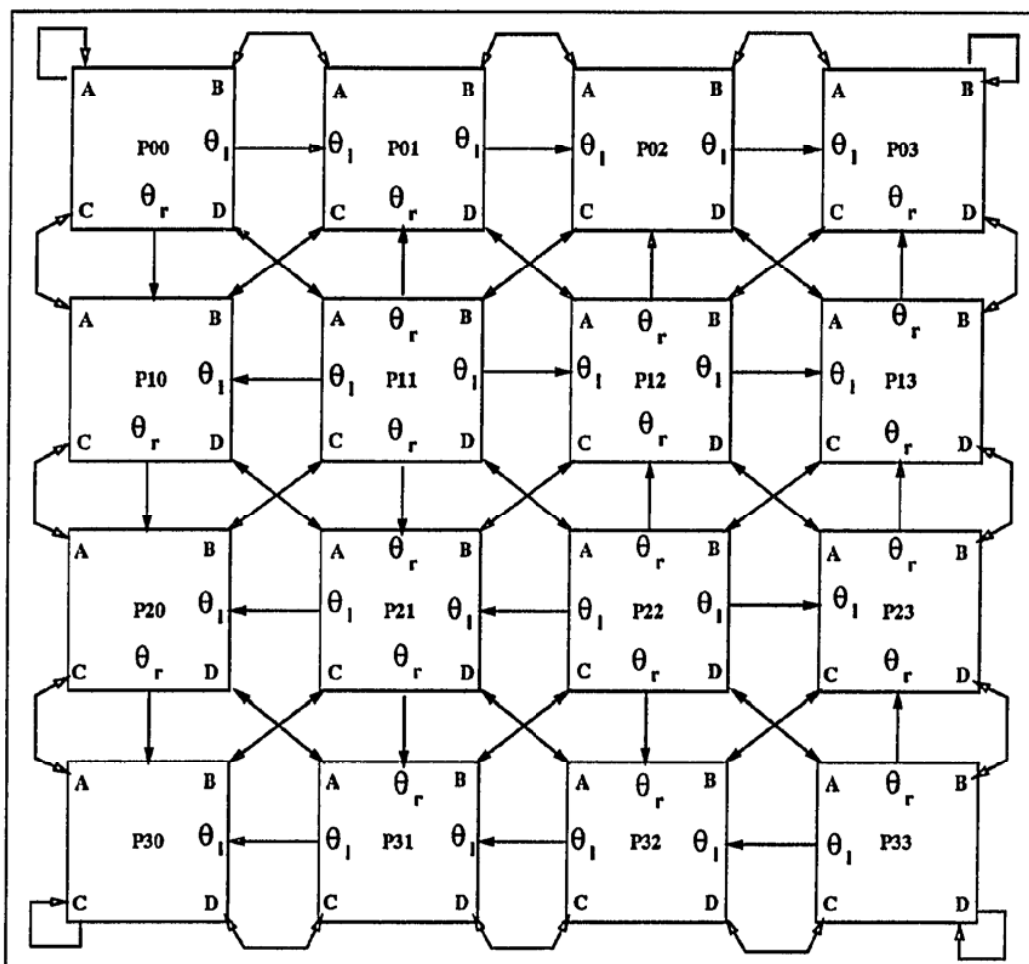


Figure 2.1: The Brent-Luk-Van Loan SVD Array for Computing the SVD of an  $8 \times 8$  Matrix

A  $2 \times 2$  SVD can be described as

$$R(\theta_l)^T \begin{bmatrix} a & b \\ c & d \end{bmatrix} R(\theta_r) = \begin{bmatrix} \psi_1 & 0 \\ 0 & \psi_2 \end{bmatrix}, \quad (2.6)$$

where  $\theta_l$  and  $\theta_r$  are the left and right rotation angles, respectively. The rotation matrix is

$$R(\theta) = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}, \quad (2.7)$$

and the input matrix is

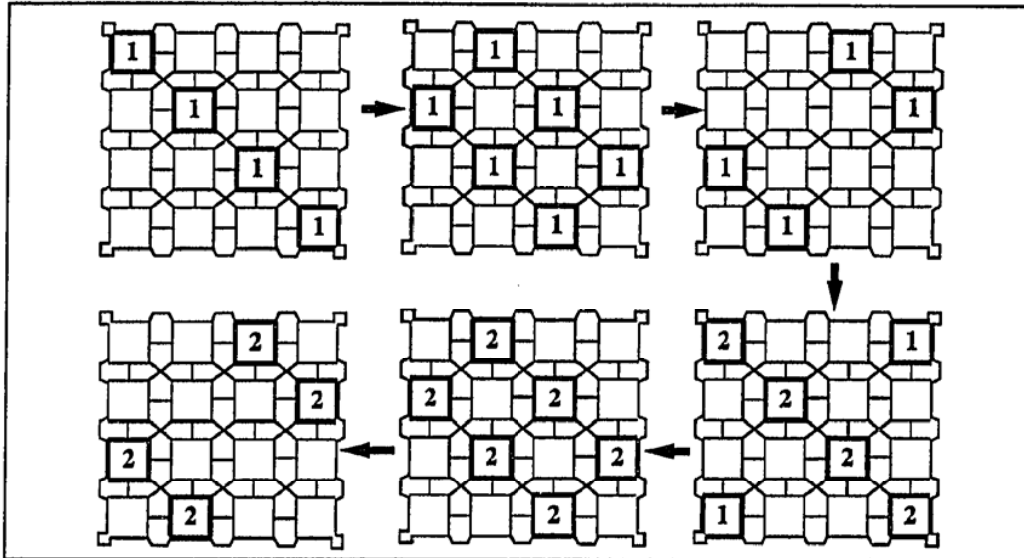
$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

The angles  $\theta_l$  and  $\theta_r$  necessary to diagonalize the matrix  $M$  can be shown to be the inverse tangents of the data elements of  $M$ :

$$\begin{aligned} \theta_r + \theta_l &= \tan^{-1} \left[ \frac{c+b}{d-a} \right], \\ \theta_r - \theta_l &= \tan^{-1} \left[ \frac{c-b}{d+a} \right] \end{aligned} \quad (2.8)$$

The left and right rotation angles in equation 2.5 can be computed using the equations 2.8 on a  $2 \times 2$  matrix formed by  $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ . It may be noted that the  $p/2$  diagonal processors store precisely the matrices required for  $p/2$  successive iterations, corresponding to a single row of parallel ordering in Table 2.1. All the transformations that annihilate the off-diagonal elements in the diagonal processors are independent and hence can be computed in parallel. An update of the entire array requires the left angles to be propagated from a diagonal processor to all the processors on the same row, and the right angle to be propagated to all the processors on the same column. This propagation is systolic. The update of the matrix is followed by a permutation of the rows and the columns of the matrix according to the parallel ordering. An 8-connected mesh is required to perform this data exchange. The permutation results in the next set of sub-matrices, required to compute the new set of angles, to be formed





**Figure 2.2: Snapshots of the Brent-Luk-Van Loan SVD Array Showing Diagonal Waves of Activity**

at the diagonal processors. Since the diagonal processors require data only from their diagonal neighbors, new computation can start as soon as these neighbors complete their portion of the computation. Thus, the updates of the matrix, corresponding to separate iterations in equation 2.5 are pipelined. This leads to diagonal waves of activity originating at the main diagonal and propagating outwards. This is shown in Figure 2.2. The various steps in the operation of this array are listed below:

- The  $p/2$  diagonal processors compute the required transformation angles using the  $2 \times 2$  sub-matrix stored in them.
- Each diagonal processor propagates the left angle, which it computed, to its east and west neighbor. Similarly the right rotation angles are propagated north and south.

- The non-diagonal processors receive the left and right angles and apply the 2-sided transformation to their sub-matrices. After the computation, they forward the left and right rotation angles to the processor next in the row or column.
- Processor P22 in Figure 2.1 receives its next set of data elements from processors P11, P33, P13 and P31. Hence, it has to wait till processors P31 and P13 finish their computations. Similarly, all the other diagonal processors need to wait for their diagonal neighbors to complete the transformations before exchanging data. Exchange of data in different parts of the array is staggered in time.
- After the data exchange, a new set of angles is computed by the diagonal processors and the same sequence of operations is repeated.

Each diagonal processor requires  $(p-1)$  iterations to complete a sweep. The number of sweeps required for convergence is  $O(\log p)$ . For most matrices up to a size of  $100 \times 100$ , the algorithm has been observed to converge within 10 sweeps. Since each iteration for a diagonal processor is completed in constant time, the total SVD requires  $O(p \log p)$ .

## 2.4 Architecture for $2 \times 2$ SVD

The array proposed by Brent, Luk and Van Loan was converted into an architecture suitable for VLSI implementation by Cavallaro and Luk [8]. By using CORDIC arithmetic, a simple architecture suitable for systolic implementation was developed. The basic functionality required in each processor is:

- Ability to compute inverse tangents is required for the computation of the 2-sided transformations from matrix data,

- Ability to perform 2-sided rotations of a  $2 \times 2$  matrix that is required when performing the transformation,
- Control to perform the parallel ordering data exchange.

CORDIC allows efficient implementation of the required arithmetic operations in hardware. Chapter 3 discusses the CORDIC algorithm in depth. The architecture of the individual processor element is discussed in Chapter 4.

## Chapter 3

### CORDIC Techniques

#### 3.1 Introduction

In a special-purpose VLSI processor it is not necessary to include a general purpose Arithmetic and Logic Unit (ALU). An efficient design with optimal area and time complexity can be obtained by using special arithmetic techniques that map the desired computations to simplified hardware. The CORDIC technique allows efficient implementation of functions, like vector rotations and inverse tangents, in hardware. These constitute the principal computations in the SVD algorithm. Additional ALU operations (addition, subtraction and divide-by-2) required for the SVD are more basic operations and can reuse the adders and shifters that constitute the CORDIC modules.

The Coordinate Rotation Digital Computer (CORDIC) technique was initially developed by Volder [35] as an algorithm to solve the trigonometric relationships that arise in navigation problems. Involving only a fixed sequence of additions or subtractions, and binary shifts, this scheme was used to quickly and systematically approximate the value of a trigonometric function or its inverse. This algorithm was later unified for several elementary functions by Walther [36]. The algorithm has been used extensively in a number of calculators [16] to perform multiplications, divisions, to calculate square roots, to evaluate sine, cosine, tangent, arctangent, sinh, cosh, tanh, arctanh, ln and exp functions, and to convert between binary and mixed radix num-

ber systems [10].

### 3.2 CORDIC Algorithms

A variety of functions are computed in CORDIC by approximating a given angle in terms of a sequence of fixed angles. The algorithm for such an approximation is iterative and always converges in a fixed number of iterations. The basic result of CORDIC concerns the convergence of the algorithm and is discussed in depth by Walther [36]. The convergence theorem states the conditions that affect convergence and essentially gives an algorithm for such an approximation. The theorem is restated here. Walther [36] gives the proof for this theorem.

**Theorem 3.1** (*Convergence of CORDIC*) Suppose

$$\varphi_0 \geq \varphi_1 \geq \varphi_2 \geq \cdots \geq \varphi_{n-1} > 0,$$

is a finite sequence of real numbers such that

$$\varphi_i \leq \sum_{j=i+1}^{n-1} \varphi_j + \varphi_{n-1}, \text{ for } 0 \leq i \leq n-1,$$

and suppose  $r$  is a real number such that

$$|r| \leq \sum_{j=0}^{n-1} \varphi_j$$

If  $s_0 = 0$ , and  $s_{i+1} = s_i + \delta_i \varphi_i$ , for  $0 \leq i \leq n-1$ , where

$$\delta_i = \begin{cases} 1, & \text{if } r \geq s_i \\ -1, & \text{if } r < s_i, \end{cases}$$

then,

$$|r - s_k| \leq \sum_{j=k}^{n-1} \varphi_j + \varphi_{n-1}, \text{ for } 0 \leq k \leq n-1,$$

and so in particular  $|r - s_n| < \varphi_{n-1}$ .

If  $\varphi_i$ s represent angles, the theorem provides a simple algorithm for approximating any angle  $r$ , in terms of a set of fixed angles  $\varphi_i$ , with a small error  $\varphi_{n-1}$ . The versatility of the CORDIC algorithm allows its use in several different modes: linear, circular and hyperbolic. The principal mode of interest in the CORDIC-SVD processor is the *circular* mode of CORDIC, due to its ability to compute vector rotations and inverse tangents efficiently.

### 3.3 CORDIC Operation in the Circular Mode

The anticlockwise rotation of a vector  $[\tilde{x}_i, \tilde{y}_i]^T$  through an angle  $\alpha_i$  (Figure 3.1) is given by

$$\begin{bmatrix} \tilde{x}_{i+1} \\ \tilde{y}_{i+1} \end{bmatrix} = \begin{bmatrix} \cos \alpha_i & \sin \alpha_i \\ -\sin \alpha_i & \cos \alpha_i \end{bmatrix} \begin{bmatrix} \tilde{x}_i \\ \tilde{y}_i \end{bmatrix}. \quad (3.1)$$

The same relation can be rewritten as

$$\begin{bmatrix} \tilde{x}_{i+1} \sec \alpha_i \\ \tilde{y}_{i+1} \sec \alpha_i \end{bmatrix} = \begin{bmatrix} 1 & \tan \alpha_i \\ -\tan \alpha_i & 1 \end{bmatrix} \begin{bmatrix} \tilde{x}_i \\ \tilde{y}_i \end{bmatrix}. \quad (3.2)$$

All computations in the circular mode of CORDIC involve a *fixed* number of iterations, which are similar to equation 3.2, but for a scaling,

$$x_{i+1} = x_i + \delta_i y_i \tan \alpha_i \quad (3.3)$$

$$y_{i+1} = y_i - \delta_i x_i \tan \alpha_i, \quad (3.4)$$

where  $x_i$  and  $y_i$  are states of variables  $x$  and  $y$  at the start of the  $i^{\text{th}}$  iteration,  $0 \leq i < n$ , and  $\delta_i \in \{-1, +1\}$ . The values  $x_n$  and  $y_n$  are the values obtained at the end of the CORDIC iterations and differ from the desired values  $\tilde{x}_n$  and  $\tilde{y}_n$  due to a scaling as shown in Figure 3.1. Along with the  $x$  and  $y$  iterations, iterations of the form

$$z_{i+1} = z_i + \delta_i \alpha_i \quad (3.5)$$

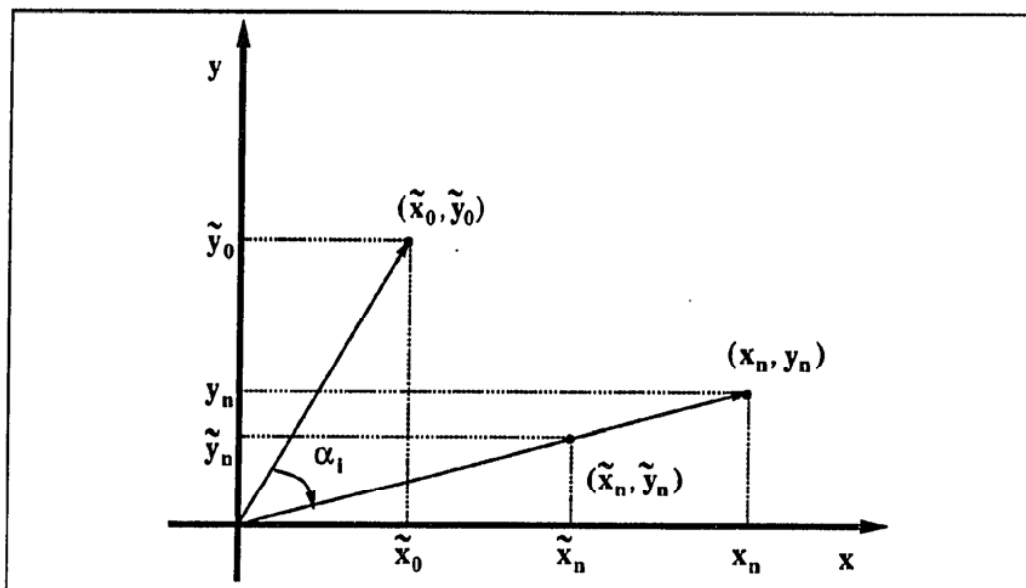


Figure 3.1: Rotation of a vector

are performed on a variable  $z$  to store different states of an angle. The angles  $\alpha_i$  are chosen to be  $\tan^{-1}(2^{-i})$ , which reduces equation 3.3 and equation 3.4 to

$$x_{i+1} = x_i + \delta_i y_i 2^{-i} \quad (3.6)$$

$$y_{i+1} = y_i - \delta_i x_i 2^{-i}. \quad (3.7)$$

These iterations, called the  $x$  and the  $y$  iterations, can be implemented as simple additions or subtractions, and shifts. The angles  $\alpha_i$  are stored in a ROM, allowing the  $z$  iterations, given by equation 3.5, to be implemented as additions or subtractions. The actual hardware required to implement these iterations in hardware is shown in Figure 3.2.

Rotation of a vector through any angle  $\theta$  is achieved by decomposing the angle as a sum of  $\alpha_i$ s, using the CORDIC convergence theorem, and rotating the initial vector through this sequence of angles. Different functions are implemented by a different choice of  $\delta_i$  at each iteration. If  $\delta_i$  is chosen to force the initial  $z_0$  to 0, it

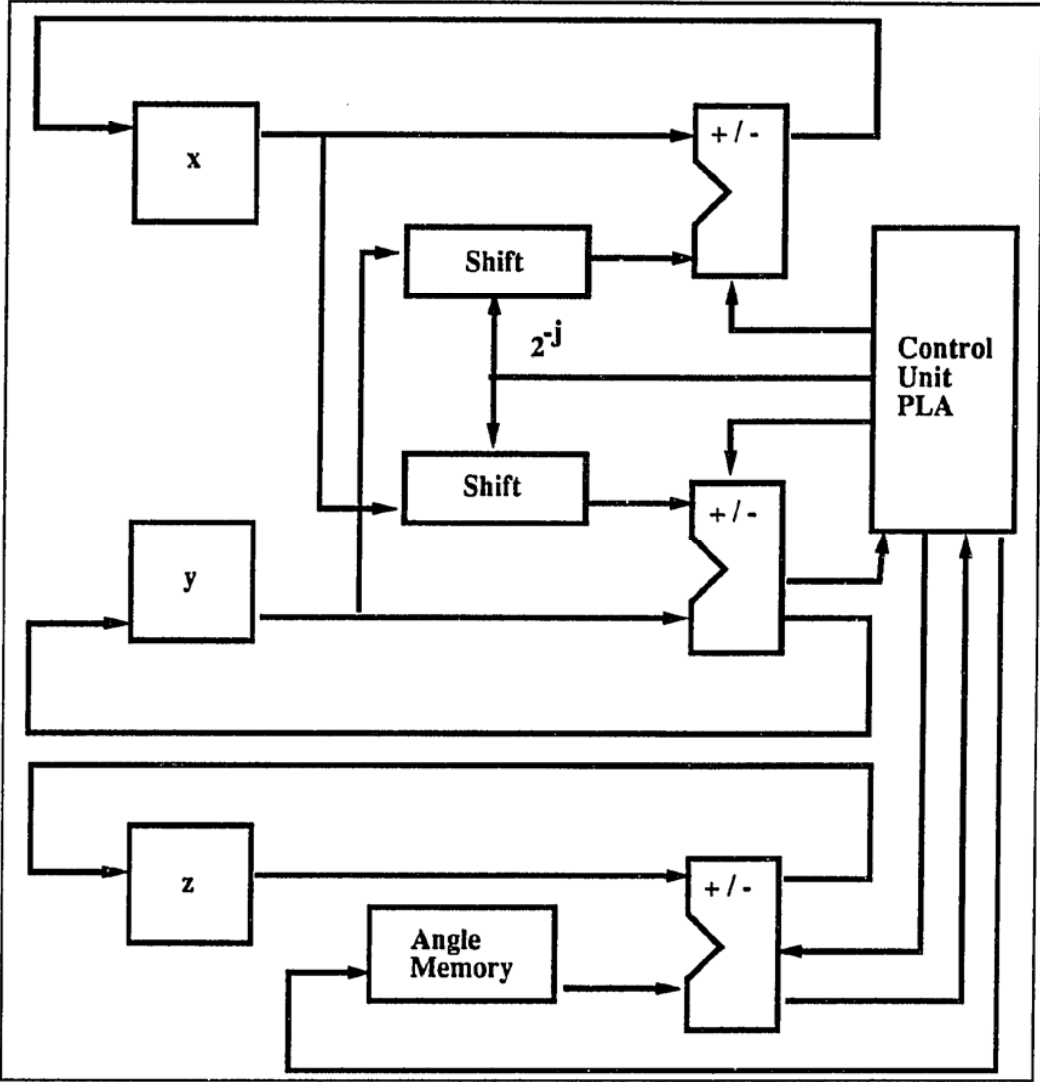


Figure 3.2: Implementation of the CORDIC Iterations in Hardware



Operation	Initial Values for Variables	Final values for variables	Functions Computed
Z-Reduction	$x_0 = \tilde{x}_0$ $y_0 = \tilde{y}_0$ $z_0 = \theta$	$x_n = (x_0 \cos \theta + y_0 \sin \theta)/K_n$ $y_n = (-x_0 \sin \theta + y_0 \cos \theta)/K_n$ $z_n = 0$	Vector Rotations, sine, cosine
Y-Reduction	$x_0 = \tilde{x}_0$ $y_0 = \tilde{y}_0$ $z_0 = \theta$	$x_n = \sqrt{x_0^2 + y_0^2}/K_n$ $y_n = 0$ $z_n = \theta = \tan^{-1} \left( \frac{y_0}{x_0} \right)$	Inverse Tangents

**Table 3.1: CORDIC Functionality in the Circular Mode**

is termed *Rotation* or *Z-reduction*. This is used to perform efficient vector rotations given an initial angle. A choice of  $\delta_i$ s to reduce the initial  $y_0$  to 0 is called *Vectoring* or *Y-reduction*. Iterations of this form allow the computation of the inverse tangent function. Table 3.1 summarizes the various operations performed in the CORDIC circular mode, as they pertain to the SVD processor.

### 3.3.1 CORDIC Z-Reduction

Given an initial vector  $[\tilde{x}_0, \tilde{y}_0]^T = [x_0, y_0]^T$  and an angle  $z_0 = \theta$ , through which to rotate, the rotated vector can be obtained by rotating the initial vector through a sequence of fixed angles  $\alpha_i$  given by  $\alpha_i = \tan^{-1}(2^{-i})$ . This results in the following iterative equations

$$\begin{aligned}
 x_{i+1} &= x_i + \delta_i y_i 2^{-i} \\
 y_{i+1} &= y_i - \delta_i x_i 2^{-i} \\
 z_{i+1} &= z_i - \delta_i \alpha_i,
 \end{aligned} \tag{3.8}$$

where,

$$\alpha_i = \tan^{-1}(2^{-i})$$

$$\delta_i = \begin{cases} 1 & \text{if } z_i \geq 0 \\ -1 & \text{if } z_i < 0. \end{cases}$$

Each iteration for  $x_i$  and  $y_i$  corresponds to either a clockwise or an anticlockwise rotation of  $[x_i, y_i]^T$  through the angle  $\alpha_i$ , and a scaling with  $\sec(\alpha_i)$ . The CORDIC iterations for  $z_i$  correspond to a decomposition of the initial angle in terms of a set of fixed angles,  $\alpha_i$ , as given by Theorem 3.1.

After  $n$  iterations, the vector  $[x_n, y_n]^T$  is a clockwise rotation of  $[x_0, y_0]^T$  through an angle  $\theta$  and a scaling by  $1/K_n$ . Figure 3.1 illustrates  $x_n$  and  $y_n$  scaled with respect to the desired values  $\tilde{x}_n$  and  $\tilde{y}_n$ . This can be expressed as

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_0/K_n \\ y_0/K_n \end{bmatrix}, \quad (3.9)$$

where  $K_n$ , the constant scale factor, is given by equation 3.10,

$$K_n = \prod_{i=0}^{n-1} \cos \alpha_i. \quad (3.10)$$

This scale factor is independent of the actual  $\delta_i$ s chosen at each step, and converges to  $\approx 0.607$  for large  $n$ . A post-processing step is required to eliminate the scale factor. An algorithm for scale-factor correction is described in Section 3.4. Z-reduction can be used to compute the cosine and sine of the initial angle  $\theta$ , by choosing special cases for the initial vector  $[x_0, y_0]^T$ .

### 3.3.2 CORDIC Y-Reduction

CORDIC is used in the Y-reduction mode for inverse tangent computations. The decomposition of an angle, according to Theorem 3.1, requires only the knowledge of the sign of the angle at that iteration and not the actual magnitude. In the

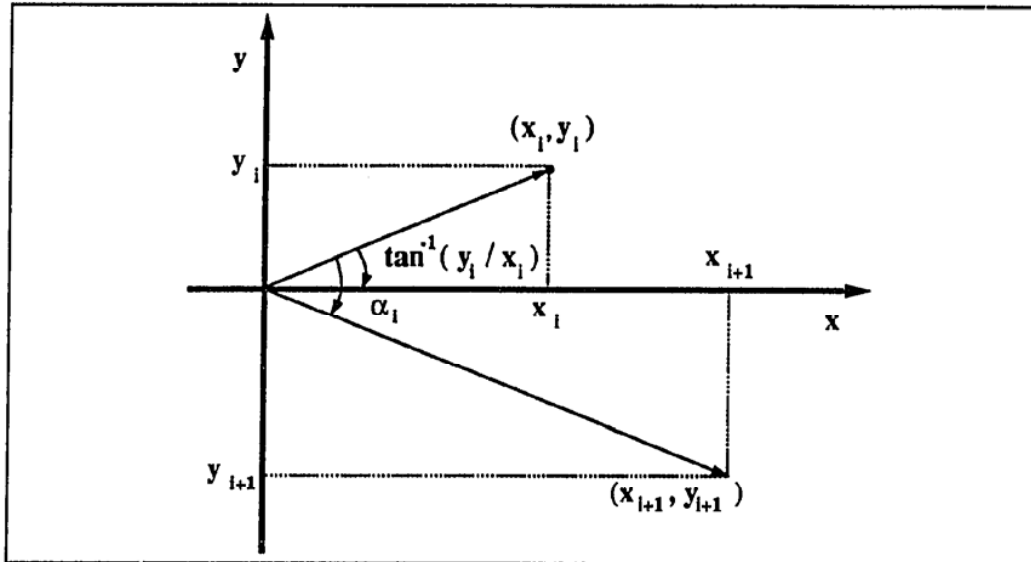


Figure 3.3: Rotation of a vector in Y-Reduction

computation of  $\tan^{-1}(y_0/x_0)$ , the same information is obtained from the  $y$  component of the vector at each iteration. As shown in Figure 3.3, if  $\theta_i > 0$ ,  $y_i > 0$  and vice versa. The function  $\tan^{-1}(y_0/x_0)$  is computed by rotating the initial vector through successively smaller angles  $\alpha_i$ , the direction of rotation at each step chosen to reduce the  $y$ -component of the vector to zero. The total angle, through which the vector has been rotated is accumulated in the  $z$ -register. The CORDIC convergence theorem forces the accumulated angle to within  $\tan^{-1}(2^{-n})$  of  $\tan^{-1}(y_0/x_0)$ . Using the initial values  $[\tilde{x}_0, \tilde{y}_0]^T = [x_0, y_0]^T$  and  $z_0 = 0$ , the following CORDIC iterations can be used to compute the value of  $\tan^{-1}(y_0/x_0)$ :

$$\begin{aligned}
 x_{i+1} &= x_i + \delta_i y_i 2^{-i} \\
 y_{i+1} &= y_i - \delta_i x_i 2^{-i} \\
 z_{i+1} &= z_i + \delta_i \alpha_i,
 \end{aligned}
 \tag{3.11}$$

where,

$$\alpha_i = \tan^{-1}(2^{-i}),$$

$$\delta_i = \begin{cases} 1 & \text{if } y_i \geq 0 \\ -1 & \text{if } y_i < 0. \end{cases} \quad (3.12)$$

After  $n$  iterations,  $x_n = \sqrt{x_0^2 + y_0^2}/K_n$ ,  $y_n \approx 0$  and  $z_n = \tan^{-1}(y_0/x_0)$ . If the radius  $\sqrt{x_0^2 + y_0^2}$  is desired, a scale factor correction step is necessary. The final vector  $(x_n, y_n)$  is the initial vector  $(x_0, y_0)$  rotated through  $\tan^{-1}(y_0/x_0)$  and scaled.

### 3.3.3 Modified Y-Reduction for the SVD processor

If  $\delta_i$  is chosen according to equation 3.12, the  $y$ -reduction iterations do not converge for negative values of  $x_i$ . This restricts the initial vectors to lie in the first and fourth quadrants. This is a restriction that cannot be ensured at all steps of the SVD algorithm. Although alternate algorithms utilize the final value of  $x_n$  [38], in this SVD processor only the inverse tangent is required. This allows scaling both  $x_0$  and  $y_0$  with a constant, without affecting the computed value. Thus a simple pre-processing step [8], which negates both  $x_0$  and  $y_0$ , if  $x_0$  is negative, can be used to map the vectors from third and second quadrants to the first and fourth quadrants respectively. A different method of choosing the  $\delta_i$ s extends the convergence of the CORDIC module and hence obviates the need for this pre-processing step. This alternate choice [38] of  $\delta_i$  is given by equation 3.13,

$$\delta_i = \begin{cases} 1 & \text{if } y_i x_i \geq 0 \\ -1 & \text{if } y_i x_i < 0. \end{cases} \quad (3.13)$$

This choice of  $\delta_i$  is implemented using a single exclusive-or gate to find the XOR of the sign-bits of  $x_i$  and  $y_i$ . This is considerably simpler than an implementation using pre-processing.

### 3.4 Scale Factor Correction

There have been numerous approaches to scale factor correction in the literature [36, 16, 1, 11]. Most approaches require special iterations of a form similar, but not identical, to the CORDIC iterations. It is important to reduce the number of these iterations to the minimum, since scale factor correction may be regarded as overhead. Most scale factor correction algorithms approximate the value of  $K_n$  as

$$\prod_{j \in J} (1 - 2^j),$$

where  $J$  is a set of values obtained empirically. The cardinality of the set  $J$  is typically at least  $n/4$  and in some cases as large as  $n$ . An alternate algorithm (equation 3.14), which is more regular and hence useful for different values of  $n$  is applicable after two complete CORDIC rotations. By the end of two CORDIC rotations, the variables  $x_n$  and  $y_n$  are scaled by  $K_n^2$ . The scheme [8] to eliminate the factor of  $K_n^2$  requires only  $\lceil n/4 \rceil$  scale factor correction iterations after two CORDIC rotations of  $n$  iterations each and, hence, is cheaper than the other scale factor correction algorithms,

$$2K_n^2 = \prod_{j \in J} (1 - 2^{-2j}), \quad (3.14)$$

where  $J = \{1, 3, 5, \dots, 2\lceil n/4 \rceil - 1\}$ . The scale factor iterations are given by

$$\begin{aligned} x_{i+1} &= x_i - x_i 2^{-j} \\ y_{i+1} &= y_i - y_i 2^{-j}, \end{aligned} \quad (3.15)$$

where,  $j = 2, 6, 10, \dots, 4\lceil n/4 \rceil - 2$ . At the end of these iterations, the factor of 2 in equation 3.14 is eliminated by a simple right shift.

### 3.5 Error Analysis of CORDIC Iterations

Any numerical scheme has to be resilient to truncation errors at every stage of computation in order to justify its implementation in a computer. Error analysis is the

acid test to determine whether a numerical algorithm is suitable for implementation. Many algorithms that are mathematically accurate, fail in the presence of truncation errors. While finite precision presents insurmountable problems to some algorithms, it results in only a bounded error in some others. A detailed study of these errors is necessary to interpret the results and guarantee a certain degree of accuracy.

Although CORDIC has been used previously, much of the analysis of its numerical accuracy has been *ad hoc*. Walther [36] claimed that a precision of  $n$  bits can be achieved if the internal representation used is  $n + \log_2 n$  bits. This analysis neglects any interaction between the various iterations and considers only the effect of the finite precision of the  $x$  and  $y$  data paths. Walther studied CORDIC for a floating-point implementation and pointed out that it is necessary to use a normalized representation to achieve this precision. Johnsson [18] attempted to obtain a closer bound on the error caused by  $x$  and  $y$  iterations, while still neglecting the interaction between the various iterations. This analysis showed that the internal representation of  $L$  bits guarantees an accuracy of  $n$  bits, if,

$$L > n + \log_2(2n - L - 3). \quad (3.16)$$

This is again  $n + \log_2 n$  bits internal representation to obtain the desired precision of  $n$  bits. Both the analyses neglect the effect of the  $z$  iterations on the the  $x$  and  $y$  iterations and *vice versa*.

A more detailed error analysis for fixed-point CORDIC modules is presented by Duryea [13]. He shows that the error for Z-reduction is bounded as:

$$|\tilde{x}_n - \hat{x}_n| \leq \frac{7}{6}n2^{-t_1} + 2^{-(n-1)} \quad (3.17)$$

where,  $n$  is the number of iterations and  $t_1$  is the bit precision of  $x$ ,  $y$  and  $z$ . In addition, he shows that the error in the computed value of inverse tangent is:

$$|\tilde{\theta}_n - \hat{\theta}_n| \leq (n + 1)2^{-t_1} + 2^{-(n-1)} \quad (3.18)$$

This error bound, however, does not account for the effect of unnormalized  $x_0$  and  $y_0$ , which is the main source of error in Y-reduction.

An exact analysis of the various errors involved in CORDIC is presented in Sections 3.6 and 3.7. For a finite number of iterations, CORDIC computes an approximate value of the desired functions even with infinite precision. Thus an error is inherently associated with the computations. This error is usually very small. Additional errors occur due to the finite-precision representation of the different variables. The analysis presented in this thesis uses the following notation to represent the different values of the variables  $x$ ,  $y$  and  $z$ :

- $\hat{x}_i, \hat{y}_i, \hat{z}_i$  represent the actual values obtained due to CORDIC iterations, assuming finite-precision arithmetic in a real-world implementation of CORDIC,
- $x_i, y_i, z_i$  represent the values that will be obtained if the *same* CORDIC iterations that were performed in the previous case, are performed on numbers represented with infinite-precision,
- $\tilde{x}_i, \tilde{y}_i, \tilde{z}_i$  represent the values that should be obtained for the desired function with infinite precision, as mathematically defined.

### 3.6 Error Analysis of CORDIC Z-Reduction

Let the initial values for  $x$ ,  $y$  and  $z$  be  $x_0, y_0, z_0 = \theta$ . Each CORDIC iteration for  $x$  and  $y$  is of the form,

$$\begin{aligned}x_{i+1} &= x_i + \delta_i y_i 2^{-i} \\ y_{i+1} &= y_i - \delta_i x_i 2^{-i}.\end{aligned}$$

Finally after  $n$  iterations,

$$x_n = (x_0 \cos \alpha + y_0 \sin \alpha) / K_n \quad (3.19)$$

$$y_n = (y_0 \cos \alpha - x_0 \sin \alpha) / K_n, \quad (3.20)$$

where,

$$\begin{aligned} K_n &= \prod_{i=0}^{n-1} \cos \alpha_i \\ \alpha_i &= \tan^{-1}(2^{-i}) \\ \alpha &= \sum_{i=0}^{n-1} \delta_i \alpha_i. \end{aligned} \quad (3.21)$$

The sequence of  $\delta_i$ s is exactly the same as what would be obtained in a real-world implementation of CORDIC, using the same initial conditions. The angle  $\alpha$  is an approximation of  $\theta$ . Since the  $x$  and  $y$  iterations implement a rotation through  $\tan^{-1}(2^{-i})$  by shifting the variables, they achieve a rotation through exactly  $\alpha_i$ . The finite-precision representation of  $x_i$  and  $y_i$ , however, causes a truncation error in their computation. Let the actual values of  $x_i$  and  $y_i$  obtained at the end of each iteration be  $\hat{x}_i, \hat{y}_i$ .

### 3.6.1 Error Introduced by X and Y Iterations

Suppose  $x$  and  $y$  are represented with  $t_1$  bits and  $x$  and  $y$  are interpreted as fractions, without any loss of generality. Since the numbers are truncated after the right shift, the addition and subtraction operations result in a truncation error not exceeding  $2^{-t_1}$ . Accordingly,

$$\hat{x}_1 = x_0 + \delta_0 y_0 2^{-0} = x_1$$

$$\hat{y}_1 = y_0 + \delta_0 x_0 2^{-0} = y_1.$$

The second iteration results in a truncation error due to the right shift,

$$\hat{x}_2 = x_1 + \delta_1 y_1 2^{-1} + \mu_{x1} = x_2 + \mu_{x1}$$

$$\hat{y}_2 = y_1 + \delta_1 x_1 2^{-1} + \mu_{y1} = y_2 + \mu_{y1},$$



where,

$$|\mu_{x1}|, |\mu_{y1}| < 2^{-t_1}.$$

Using these values in the next iteration results in,

$$\begin{aligned}\hat{x}_3 &= \hat{x}_2 + \delta_2 \hat{y}_2 2^{-2} + \mu_{x2} \\ &= x_2 + \delta_2 y_2 2^{-2} + (\mu_{x1} + \mu_{y1} \delta_2 2^{-2}) + \mu_{x2} \\ &= x_3 + (\mu_{x1} + \mu_{y1} \delta_2 2^{-2}) + \mu_{x2}.\end{aligned}$$

Continuing this for  $n$  iterations, we obtain

$$\hat{x}_n = x_n + \epsilon_x \quad (3.22)$$

$$\hat{y}_n = y_n + \epsilon_y, \quad (3.23)$$

where,

$$\begin{aligned}|\epsilon_x|, |\epsilon_y| &< 2^{-t_1} [1 + (1 + \delta_{(n-1)} 2^{-(n-1)}) + \\ &\quad (1 + \delta_{(n-1)} 2^{-(n-1)})(1 + \delta_{(n-2)} 2^{-(n-2)}) + \dots + \\ &\quad (1 + \delta_{(n-1)} 2^{-(n-1)})(1 + \delta_{(n-2)} 2^{-(n-2)}) \dots (1 + \delta_1 2^{-1})] \\ &< 2^{-t_1} [1 + (1 + 2^{-(n-1)}) + \\ &\quad (1 + 2^{-(n-1)})(1 + 2^{-(n-2)}) + \dots + \\ &\quad (1 + 2^{-(n-1)})(1 + 2^{-(n-2)}) \dots (1 + 2^{-1})] \\ &\approx 2^{-t_1}(n).\end{aligned}$$

The upper bound is obtained by considering the maximum value for the right hand side of the above relation, which occurs when all  $\delta_i = +1$ . The sum of the product terms is approximately  $n$ . Hence, the finite precision representation of the  $x$  and  $y$  causes an error in approximately  $\log n$  bits.

### 3.6.2 Error Introduced by Z Iterations

Let  $\beta_i$  be the finite-precision approximation of  $\alpha_i$ , then assuming fixed point implementation,

$$\alpha_i = \beta_i + \mu_{2i}, \quad |\mu_{2i}| < 2^{-t_2}, \quad (3.24)$$

where  $\beta_i$  is assumed to be represented by  $t_2$  bits. The value  $\beta_i$  is the actual value of  $\tan^{-1}(2^{-i})$  stored in the ROM. The error in representation of the angle  $\alpha_i$ , due to finite precision is  $\mu_{2i}$ .

Each iteration of CORDIC for the  $z$  variable is of the form

$$\hat{z}_{i+1} = \hat{z}_i - \delta_i \beta_i.$$

The sequence of  $\beta_i$ s satisfies the conditions required by Theorem 3.1, in spite of the errors, and hence causes the  $z$  iterations to converge. The absence of any shifting in the  $z$  iterations implies exact arithmetic, since no bits are truncated. The only deviation of  $\sum \delta_i \beta_i$  from the angle  $\theta$  is a consequence of the CORDIC approximation. Accordingly, from the convergence properties of CORDIC,

$$\begin{aligned} \hat{z}_n &\leq \beta_{n-1}, \\ \hat{z}_0 = \theta &= \sum_{i=0}^{n-1} \delta_i \beta_i + \gamma, \quad |\gamma| \leq \beta_{n-1} \end{aligned}$$

Theorem 3.1 governs the maximum error  $\gamma$  possible in this expression to be less than the smallest angle  $\beta_{n-1}$ . Coupling the finite-precision approximation error in the representation of  $\alpha_i$  with this relation, gives a bound on the deviation of the angle  $\alpha$  from the desired angle  $\theta$ . The angle  $\alpha$ , is the angle through which the vector  $[x_0, y_0]^T$  is actually rotated. Using equations 3.21 and 3.24:

$$\begin{aligned} \theta &= \sum_{i=0}^{n-1} \delta_i \alpha_i - \sum_{i=0}^{n-1} \delta_i \mu_{2i} + \gamma \\ &= \alpha - \sum_{i=0}^{n-1} \delta_i \mu_{2i} + \gamma. \end{aligned} \quad (3.25)$$

Substituting the bounds on  $\gamma$  and  $\mu_{2i}$  and choosing  $\delta_i = -1$ , the error in the angle  $|\alpha - \theta|$  is,

$$|\alpha - \theta| < \beta_{n-1} + n2^{-t_2} = \tan^{-1}(2^{-(n-1)}) + n2^{-t_2} \approx 2^{-(n-1)} + n2^{-t_2} \quad (3.26)$$

Equation 3.26 gives the error that could be caused in the  $z$  iterations in the worst case.

### 3.6.3 Effects of perturbations in $\theta$

Combining results from equations 3.19, 3.22, 3.25, a relation can be derived between what we obtain,  $[\hat{x}_n, \hat{y}_n]^T$ , and what we expect,  $[\tilde{x}_n, \tilde{y}_n]^T$ . The expected values can be expressed in terms of exact equations as,

$$\tilde{x}_n = (x_0 \cos \theta + y_0 \sin \theta)/K_n \quad (3.27)$$

$$\tilde{y}_n = (y_0 \cos \theta - x_0 \sin \theta)/K_n. \quad (3.28)$$

Assuming that the error  $\Delta\theta = |\alpha - \theta|$  is small, the error in  $\tilde{x}_n$  and  $\tilde{y}_n$  can be obtained through differentiation as

$$\frac{\partial \tilde{x}_n}{\partial \theta} = (-x_0 \sin \theta + y_0 \cos \theta)/K_n = \tilde{y}_n/K_n \quad (3.29)$$

$$\frac{\partial \tilde{y}_n}{\partial \theta} = (-y_0 \sin \theta - x_0 \cos \theta)/K_n = -\tilde{x}_n/K_n. \quad (3.30)$$

An approximation that can be made if the error in the angles is small is,

$$\Delta \tilde{x}_n \approx \frac{\partial \tilde{x}_n}{\partial \theta} \Delta \theta$$

$$\Delta \tilde{y}_n \approx \frac{\partial \tilde{y}_n}{\partial \theta} \Delta \theta.$$

Observing that the derivatives are always less than 1 yields,

$$|\tilde{x}_n - x_n| < \Delta \theta$$

$$|\tilde{y}_n - y_n| < \Delta\theta.$$

Substituting the maximum errors for  $\Delta\theta$ ,  $x_n$  and  $y_n$ , an upper bound on the actual observed error is obtained:

$$\begin{aligned} |\tilde{x}_n - \hat{x}_n|, |\tilde{y}_n - \hat{y}_n| &< (\beta_{n-1} + n2^{-t_2}) + \epsilon_x \\ &< (2^{-(n-1)} + n2^{-t_2}) + (n)2^{-t_1}. \end{aligned} \quad (3.31)$$

This error is bounded for all values of  $|x_0|$  and  $|y_0|$ . Hence, this error can be eliminated by increasing the data-width to include extra bits as *guard* bits. The error bound obtained here is nearly the same as that given by equation 3.17. However, equation 3.31 separates the effects of the  $x$ ,  $y$  iterations and the  $z$  iterations allowing a study of different components of the error.

For the prototype implementation,  $t_1 = 15$ ,  $t_2 = 15$  and  $n = 16$ . Thus the error is given by,

$$\begin{aligned} \text{Observed Error} &< (2^{-15} + (16)2^{-15}) + (16)2^{-15} \\ &< (33)2^{-15} \end{aligned}$$

The error predicted by this relation is pessimistic and may never occur in practice. It is reasonable to include  $\log_2 32 = 5$  guard bits to correct the error.

### 3.7 Error Analysis of CORDIC Y-Reduction

Let the initial values for  $x$ ,  $y$  and  $z$  be  $x_0$ ,  $y_0$ ,  $z_0 = 0$ . Each CORDIC iteration for  $x$  and  $y$  is of the form,

$$\begin{aligned} x_{i+1} &= x_i + \delta_i y_i 2^{-i} \\ y_{i+1} &= y_i - \delta_i x_i 2^{-i}. \end{aligned}$$

Finally after  $n$  iterations the equations yield,

$$x_n = (x_0 \cos \alpha + y_0 \sin \alpha) / K_n \quad (3.32)$$

$$y_n = (y_0 \cos \alpha - x_0 \sin \alpha) / K_n, \quad (3.33)$$

where,

$$\begin{aligned} K_n &= \prod_{i=0}^{n-1} \cos \alpha_i \\ \alpha_i &= \tan^{-1}(2^{-i}) \\ \alpha &= \sum_{i=0}^{n-1} \delta_i \alpha_i. \end{aligned} \quad (3.34)$$

The sequence of  $\delta_i$ s is exactly the same as what would be obtained in a real-world implementation of CORDIC, using the same initial conditions. The angle  $\alpha$  is an approximation of  $\theta$ , the inverse tangent accumulated in the  $z$ -variable. Since  $x$  and  $y$  iterations implement a rotation through  $\tan^{-1}(2^{-i})$  by shifting the variables, they achieve a rotation through exactly  $\alpha_i$ . The finite-precision representation of  $x_i$  and  $y_i$ , however, causes a truncation error in their computation. Let the actual values of  $x_i$  and  $y_i$  obtained at the end of each iteration be  $\hat{x}_i, \hat{y}_i$ .

### 3.7.1 Error Introduced by X and Y Iterations

This analysis is identical to that for Z-reduction in Section 3.6.1. The error bound is given by:

$$\hat{x}_n = x_n + \epsilon_x \quad (3.35)$$

$$\hat{y}_n = y_n + \epsilon_y \approx 0 \quad (3.36)$$

$$|\epsilon_x|, |\epsilon_y| < 2^{-t_1}(n). \quad (3.37)$$

### 3.7.2 Error Introduced by Z Iterations

Let  $\beta_i$  be the finite-precision approximation of  $\alpha_i$ ; then assuming fixed point implementation,

$$\alpha_i = \beta_i + \mu_{2i}, \quad |\mu_{2i}| < 2^{-t_2},$$

where  $\beta_i$  is assumed to be represented by  $t_2$  bits .

Each iteration of CORDIC for the  $z$  variable is of the following form:

$$\begin{aligned} \hat{z}_{i+1} &= \hat{z}_i + \delta_i \beta_i \\ \hat{z}_0 &= 0 \\ \hat{z}_n &= \sum_{i=0}^{n-1} \delta_i \beta_i \triangleq \theta \\ \theta &= \sum_{i=0}^{n-1} \delta_i \alpha_i - \sum_{i=0}^{n-1} \delta_i \mu_{2i} \\ &= \alpha - \sum_{i=0}^{n-1} \delta_i \mu_{2i}. \end{aligned} \tag{3.38}$$

The maximum deviation of the computed result  $\theta$ , from the angle through which the vector is rotated  $\alpha$  is given by,

$$|\alpha - \theta| < n2^{-t_2}. \tag{3.39}$$

### 3.7.3 Perturbation in the Inverse Tangent

If  $x$  and  $y$  could be represented with infinite precision, then  $y_n$  is related to the initial values as

$$y_n = (y_0 \cos \alpha - x_0 \sin \alpha) / K_n. \tag{3.40}$$

Let  $r$  and  $\varphi$  be defined as

$$\begin{aligned} r &= \sqrt{x_0^2 + y_0^2} \\ \tan \varphi &= \frac{y_0}{x_0}. \end{aligned}$$

Equation 3.40 can now be rewritten as:

$$\begin{aligned}
 K_n y_n &= r \sin \varphi \cos \alpha - r \sin \alpha \cos \varphi \\
 \frac{K_n y_n}{\sqrt{x_0^2 + y_0^2}} &= \sin(\varphi - \alpha) \\
 |\varphi - \alpha| &= \sin^{-1} \left( \frac{K_n y_n}{\sqrt{x_0^2 + y_0^2}} \right) \\
 |\varphi - \theta| &< \sin^{-1} \left( \frac{K_n y_n}{\sqrt{x_0^2 + y_0^2}} \right) + \sum_{i=0}^{n-1} \delta_i \mu_{2i}.
 \end{aligned}$$

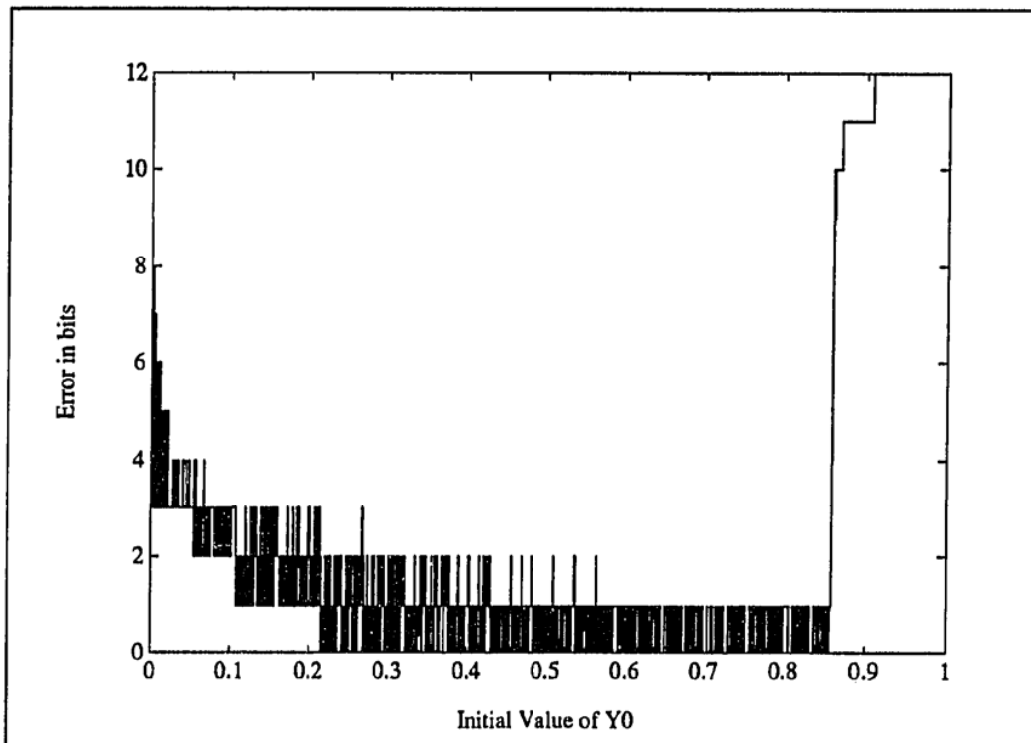
Substituting all the maximum errors from Equations 3.37 and 3.39, we get,

$$|\varphi - \theta| < \frac{2n K_n 2^{-t_1}}{\sqrt{x_0^2 + y_0^2}} + n 2^{-t_2}. \quad (3.41)$$

Here,  $\varphi$  is the true inverse tangent that is to be evaluated. The final value accumulated in the  $z$  variable,  $z_n = \theta$ , is the angle actually obtained. Thus relation 3.41 gives the numerical error in inverse tangent calculations. However, this relation does not provide a constant bound for all  $x_0$  and  $y_0$ . In the worst case,  $x_0$  and  $y_0$  are close to zero, resulting in a large error.

### 3.8 Normalization for CORDIC Y-Reduction

Equation 3.31 gives an upper bound on the error introduced by Z-reduction. This upper bound allows elimination of all the effects of the error by increasing the number of digits in the internal representation of the  $x$  and  $y$ . A similar upper bound does not exist for Y-reduction. This is reflected by relation 3.41, where small values of  $x_0$  and  $y_0$  result in a very large right-hand-side in the relation. Figure 3.4 shows the large errors observed in a prototype fixed-point CORDIC module when both  $x_0$  and  $y_0$  are small. An identical analysis of Y-reduction using floating-point data-paths shows that the error remains bounded for all possible inputs. Floating-point



**Figure 3.4:** Error in the computed value of  $\tan^{-1}(1) = \tan^{-1}(y_0/y_0)$  over the entire range of  $y_0$  in a CORDIC module without normalization. The error has been quantized into the number of bits in error. A large error is observed for small values of  $y_0$ . For very large  $y_0$ , the large error is due to overflow. In the SVD problem, by choosing small values for the initial matrix, the values at all subsequent iterations can be kept sufficiently small as to avoid overflow. This experiment has been performed on a datapath that is 16 bits wide.



arithmetic implies normalization at every stage of computation, which prevents a loss of significant digits during Y-reduction iterations. The same effect can be achieved in fixed-point CORDIC by normalizing the initial values,  $x_0$  and  $y_0$ , before performing the CORDIC iterations.

Normalization involves increasing the absolute magnitude of  $x_0$  and  $y_0$  by multiplying them with the maximum power of 2 that does not cause the subsequent CORDIC iterations to result in an overflow. Since both  $x$  and  $y$  are shifted by the same amount, the inverse tangent is not affected and no post-processing is required. If  $x$  and  $y$  are represented by  $t_1$  bits, normalization tries to increase  $x$  and  $y$  until the third most significant bit is a 1. In the worst case, the two most significant bits are reserved to avoid overflow, since every step of Y-reduction, results in an increase in the magnitude of  $x_i$ . In the worst case,  $x_0 = y_0$ , which results in  $x_n = \sqrt{2}/K_n x_0 \approx 1.6468\sqrt{2}$ .

### 3.8.1 Single Cycle Normalization

If normalization is performed as a pre-processing step, it is necessary to reduce the number of cycles required. A single cycle normalization pre-processing step would require the following components:

**Barrel Shifter:** To handle any shift up to  $(t_1 - 2)$  in a single cycle, a barrel shifter is required. The area complexity of such a shifter is  $O(t_1^2)$ . A shifter with a few selected shifts would suffice if more cycles are allowed for normalization.

**Leading Zero Encoder:** The normalization shifts are determined by the number of leading-zeros in both  $x_0$  and  $y_0$ ; the register with fewer leading zeros determining the shift. For a single cycle implementation, this takes the form of a  $(t_1 - 2)$  bit leading-zero encoder, since  $(t_1 - 2)$  is the maximum shift that will be ever be encountered. The area complexity of this encoder is  $O(t_1^2)$ . Implementations

that allow more cycles for normalization would require smaller leading-zero encoders.

The hardware complexity of implementing normalization in a single-step is  $O(t_1^2)$ . An implementation with no associated time penalty and a low area complexity is described in the next section.

### 3.8.2 Partial Normalization

The large error in inverse tangent computation occurs due to a loss of significant digits caused by the right shift associated with each iteration. If both  $x_0$  and  $y_0$  are small, then the maximum value of  $x_n$  is also small. Thus after only a few iterations of the form,

$$x_{i+1} = x_i + \delta_i y_i 2^{-i} \quad (3.42)$$

$$y_{i+1} = y_i - \delta_i x_i 2^{-i}, \quad (3.43)$$

the right shift,  $i$ , results in a complete loss of significant bits. Hence, the values of  $x_i$  and  $y_i$  remain unaffected by further iterations. Pre-normalization avoids this situation by artificially increasing the magnitudes of  $x_0$  and  $y_0$ .

The following scheme, called *Partial Normalization*, provides a low overhead solution to the normalization problem and is applicable to fixed-point CORDIC modules. Partial normalization reduces the hardware complexity by making normalization a part of the CORDIC iterations. Some of the initial CORDIC iterations are modified to include a small left shift of the results:

$$x_{i+1} = (x_i + \delta_i y_i 2^{-i}) 2^j \quad (3.44)$$

$$y_{i+1} = (y_i - \delta_i x_i 2^{-i}) 2^j. \quad (3.45)$$

This small shift introduces zeros in the low order bits, which are then shifted out in further CORDIC iterations. By keeping this shift a small integer, the hardware complexity of the shifter can be reduced. However, since the right shift,  $i$ , increases at each iteration, the magnitude of the left shift,  $j$ , should be large enough to prevent any loss of bits before normalization is complete. Using this technique, a small left shift can be used to simulate the effect of pre-normalization. The following theorem quantifies this idea.

**Theorem 3.2** Let  $t' = \arctan(y_0 2^k / x_0 2^k)$ , be the value calculated by preshifting  $x_0$  and  $y_0$  and using an unmodified CORDIC unit, where  $k$  is an integral multiple of a constant  $j$ . Let  $t = \arctan(y_0 / x_0)$  be the value calculated by using a modified CORDIC unit and unnormalized initial values. The modified CORDIC unit initially performs  $k/j$  iterations of the form

$$\begin{aligned} x_{i+1} &= (x_i + \delta_i y_i 2^{-i}) 2^j \\ y_{i+1} &= (y_i - \delta_i x_i 2^{-i}) 2^j, \end{aligned}$$

where  $i$  is the iteration index, followed by  $(n - k/j)$  unmodified CORDIC iterations. The two inverse tangents computed,  $t$  and  $t'$ , will be identical if  $k < K$ , where  $K$  is an upper bound given by

$$K = 2j^2$$

**Proof** Let  $x'_i$  and  $y'_i$  be the intermediate values obtained in the computation of  $t'$  and  $x_i$  and  $y_i$  be the intermediate values obtained in the computation of  $t$ . Since a left shift of  $j$  is introduced in each modified CORDIC iteration, the desired shift of  $k$  requires  $k/j$  cycles to achieve. The two results are guaranteed to be identical if the following conditions hold:

1. At every iteration  $i < k/j$ ,  $x_i$  and  $y_i$  are multiples of  $2^{m_i}$  where  $m_i \geq i$ . This condition prevents any loss of bits due to the right shift of  $i$  in a CORDIC iteration  $i$ , when  $i < k/j$ ,
2. In the computation of  $t'$ , a similar condition should hold for  $x'_i$  and  $y'_i$  for at least  $k/j$  cycles.

The proof for this theorem first finds the maximum total shift,  $K$ , which can be achieved with modified iterations given a constant  $j$ , and then shows that for any  $k$  that is a multiple of  $j$  and less than  $K$ , condition 2 holds.

For the modified CORDIC module, at any iteration  $0 \leq i < k/j$ ,  $x_i$  and  $y_i$  are multiples of  $2^{m_i}$  where

$$2^{m_i} = \frac{(2^j)^i}{2^{0+1+2+\dots+(i-1)}}$$

Imposing the condition required to avoid any loss of bits,

$$\begin{aligned} m_i &\geq i \\ \frac{(2^j)^i}{2^{i(i-1)/2}} &\geq 2^i \\ ji - \frac{i(i-1)}{2} &\geq i \\ i &\leq 2j - 1. \end{aligned}$$

Thus, the maximum number of iterations, for which no bits are lost is given by  $2j$ .

The maximum achievable total shift is  $K = (2j)j = 2j^2$ .

If  $k < 2j^2$  and is a multiple of  $j$ , then, in the computation of  $t'$ , at the  $k/j$ th iteration (*viz*,  $i = k/j - 1$ ),  $x'$  and  $y'$  are multiples of  $2^q$  where

$$q = k - \left[ 0 + 1 + 2 + 3 + \dots + \left( \frac{k}{j} - 2 \right) \right].$$

Thus, no bits will be lost in the computation of  $t'$  within the first  $k/j$  iterations, if the condition  $q \geq k/j - 1$  is true. Solving this inequality,

$$\begin{aligned} q &\geq k/j - 1 \\ k - \frac{(k/j - 1)(k/j - 2)}{2} &\geq k/j - 1 \\ k &\leq 2j^2. \end{aligned}$$

This is exactly the condition  $k \leq K$ ; hence condition 2 holds.

Thus  $K = 2j^2$  is the upper bound such that if  $k \leq K$ ,  $t' = t$ . □

This theorem shows that if a constant shift of  $j$  is introduced in the CORDIC iterations, then any overall shift that is a multiple of  $j$  but less than  $2j^2$  can be achieved. In practice, the parameter  $j$  can be made a variable, allowing one of several shifts to be chosen at each iteration. An appropriate choice of these shifts can achieve any desired total shift.

Suppose  $j$  in equation 3.44 and 3.45 can be chosen from a total of  $J$  shifts,  $\text{shift}[J-1] > \text{shift}[J-2] > \dots > 1$ , if  $k$  is the desired shift, then the algorithm given in Figure 3.5, gives a choice of shifts at each iteration to normalize the input. This control can be implemented using combinational logic; a pair of leading-zero encoders and a comparator to select the smaller shift from the output of the encoders.

### 3.8.3 Choice of a Minimal Set of Shifts

The following is an example of a CORDIC unit with a data-width  $t_1 = 15$  bits and  $n = 16$  iterations.

- An overall shift of 1 can be achieved only by making  $j = 1$ . Thus a shift of 1 is necessary in any implementation.

---

```

begin
  for  $i := 0$  to  $n$  do
    begin
      Choose maximum  $xindex$  such that  $x_i 2^{\text{shift}[xindex]} < 0.25$ ;
      Choose maximum  $yindex$  such that  $y_i 2^{\text{shift}[yindex]} < 0.25$ ;
       $index := \min(xindex, yindex)$ ;
      Perform a modified CORDIC Iteration with  $j := index$ ;
    end
  end
end

```

---

**Figure 3.5:** Modified CORDIC Iterations, invoked during Y-reduction when the input is not normalized

- Multiple iterations with  $j = 1$  can achieve any shift up to a maximum of  $2j^2 = 2$ . A shift of 3, however, cannot be implemented as three iterations with  $j = 1$ , since this will result in a loss of significant bits. Hence, a shift of 3 requires a single iteration with  $j = 3$  that necessitates the inclusion of the shift  $j = 3$ .
- The shift  $j = 3$  can achieve any shift up to a maximum  $2j^2 = 18$  as given by Table 3.2. Since the maximum shift required is 13, the normalization shifter does not have to implement any shift other than 0, 1 and 3. Any shift that is not a multiple of 3 is performed as shifts of 3 for  $\lfloor k/3 \rfloor$  iterations, followed by iterations with  $j = 1$  to achieve the remaining shift.

Figure 3.6 shows a hardware implementation of the CORDIC unit that includes the above normalization scheme. The number of bits in error using such a scheme is bounded even for small initial values, as shown in Figure 3.7.

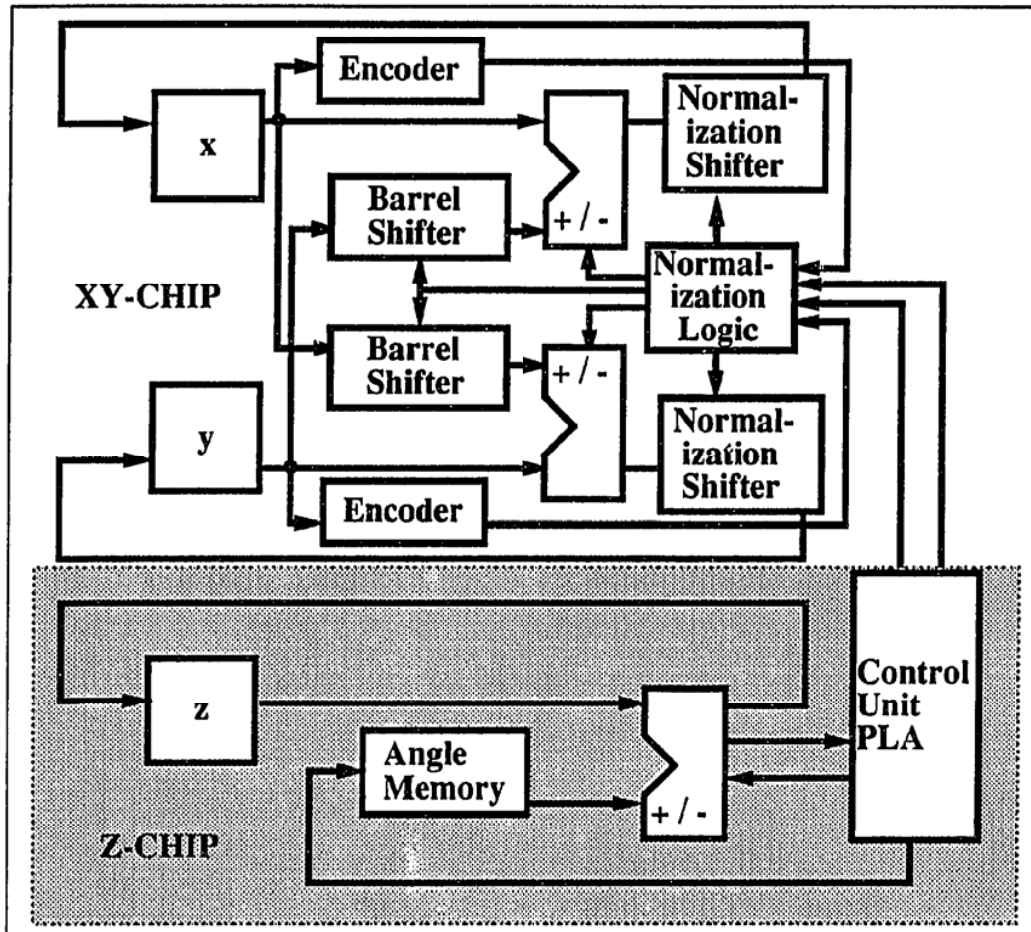


Figure 3.6: Implementation of the CORDIC Iterations in Hardware

Shift introduced in each CORDIC iteration $j$	Maximum shift that can be achieved $k$
1	2
2	8
3	18
4	32

Table 3.2: Total shifts that can be achieved with a small shift in each iteration, as part of a novel normalization scheme

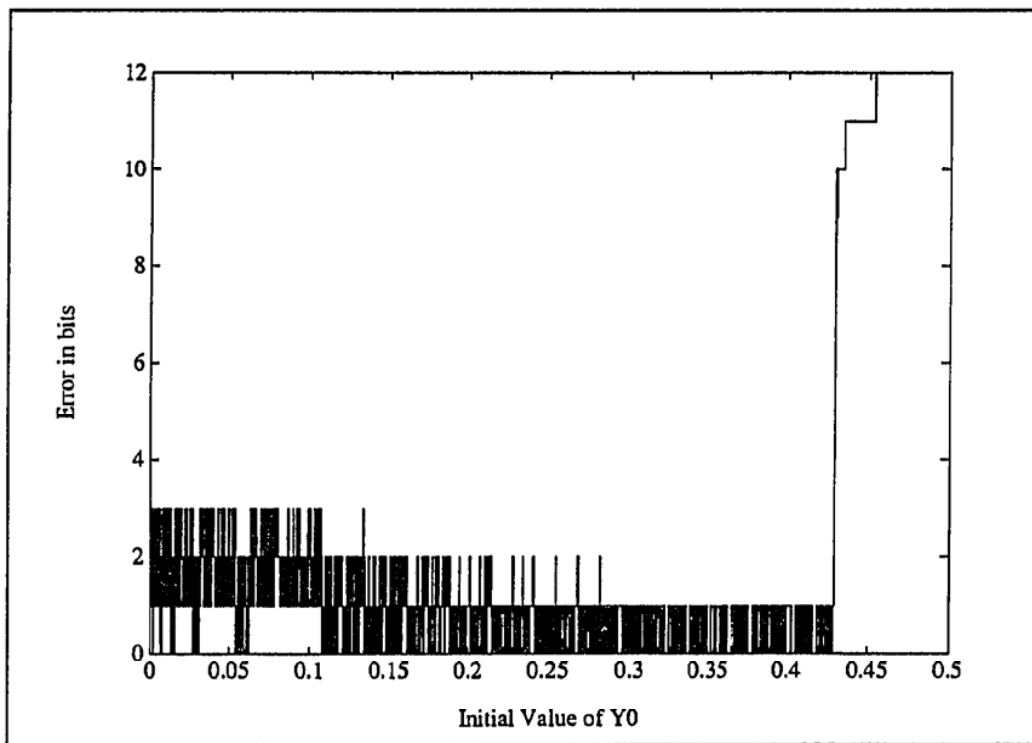


Figure 3.7: Error in the computed value of  $\tan^{-1}(y_0/y_0)$  over the entire range of  $y_0$  in a CORDIC module that implements the partial normalization scheme



### 3.8.4 Cost of Partial Normalization Implementation

For a CORDIC implementation with the data width of the  $x$  and  $y$  datapaths  $t_1$  bits, a shifter with a maximum shift  $\sqrt{t_1/2}$  is required. The shifts at each iteration are obtained from leading-zero encoders that encode the  $\sqrt{t_1/2}$  most significant bits of the  $x$  and  $y$  variables. The hardware costs involved in these operations are:

**Shifter:** The area complexity [34] of the shifter grows as

$$O(t_1 \times \text{Maximum Shift required}) = O(t_1 \times \sqrt{t_1}) = O(t_1^{1.5}).$$

**Control Logic:** The size of the control logic grows as  $O((\text{Maximum Shift required})^2)$   
 $= O(\sqrt{t_1}^2) = O(t_1)$ .

**Time Penalty:** The shifting is performed as part of the CORDIC iterations. Thus, the only time penalty is the decrease in the clock rate due to the extra propagation delay caused by the presence of the shifter in the CORDIC data path. This effect can be neglected for most cases. Hence, this scheme does not require any extra clock cycles.

## 3.9 Summary

CORDIC is an efficient technique to implement inverse tangent and vector rotation computations in hardware. Rotation of a vector is implemented as several iterations of rotations through a set of fixed angles given by  $\alpha_i = \tan^{-1}(2^{-i})$ , where  $i = 0, 1, 2, \dots, (n - 1)$ , is the iteration count. These rotations reduce to the following iterative relations:

$$x_{i+1} = x_i + \delta_i y_i 2^{-i}$$

$$y_{i+1} = y_i - \delta_i x_i 2^{-i}$$

$$z_{i+1} = z_i + \delta_i \alpha_i$$

	Z-Reduction Maximum Error in computed value of vector rotation (Number of bits)	Y-Reduction Maximum Error in computed value of $\tan^{-1}(y_0/x_0)$ (Number of bits)
Walther	$\log n$	—
Johnsson	$(L - n)$ , where $L > n + \log_2(2n - L - 3)$	—
Duryea	$\log \left[ \frac{7}{6}n2^{-t_1} + 2^{-(n-1)} \right]$	$\log \left[ (n + 1)2^{-t_1} + 2^{-(n-1)} \right]$
Section 3.5	$\log_2 \left[ (2^{-(n-1)} + n2^{-t_2}) + (n)2^{-t_1} \right]$	$\log_2 \left[ \frac{2nK_n2^{-t_1}}{\sqrt{x_0^2 + y_0^2}} + n2^{-t_2} \right]$

**Table 3.3: Summary and Comparison of the maximum errors obtained from previous methods**

These relations can be implemented using simple structures: adders, shifters and a small ROM table of angles. The direction of rotation,  $\delta_i$ , of each iteration is determined by the data and the function that is to be evaluated. If  $\delta_i$ s are chosen to reduce the initial  $z_0$  to zero, it is termed *Z-reduction*, and is used to compute vector rotations. On the other hand, a choice which reduces the initial  $y_0$  to zero, is called *Y-reduction* and is used to compute inverse tangents. A detailed error analysis takes into account the errors due to the finite-precision representation in both the  $x$ ,  $y$  iterations and the  $z$  iterations. Table 3.3 gives a summary of the error bounds given by present and previous methods. The analysis indicates that normalization of the input values is required for Y-reduction to bound the errors. Hardware complexity

of a normalization scheme would be  $O(n^2)$  using generic techniques. If the required total left shift is performed as several iterations of small shifts, however, it is possible to perform normalization as a part of the CORDIC iterations. The modified CORDIC iterations are:

$$x_{i+1} = (x_i + \delta_i y_i 2^{-i}) 2^j \quad (3.46)$$

$$y_{i+1} = (y_i - \delta_i x_i 2^{-i}) 2^j. \quad (3.47)$$

A constant shift of  $j$  can be used to perform any left shift up to  $2j^2$  through several iterations. A shifter with multiple values of  $j$  is required in an actual implementation. Such a *partial normalization* scheme reduces the hardware complexity to  $O(n^{1.5})$ , with no associated time penalty.

## Chapter 4

# The CORDIC-SVD Processor Architecture

### 4.1 Introduction

The SVD algorithm discussed in Chapter 2 and the CORDIC algorithm described in Chapter 3 form the basis for the design of the CORDIC-SVD processor. The processor has been designed in two phases. Initially, a six chip prototype of the processor was designed. Once the basic blocks were identified, they were designed, fabricated and tested independently by different design groups. The prototype was built of TinyChips, fabricated by MOSIS, which are cost effective and serve as a proof of concept. Implementation of the processor as a chip set provided controllability and observability that was essential at that stage of design. This prototype served as a means of exhaustively testing every aspect of the design, which was not possible with simulation.

The second phase involved designing a single chip version of the processor. This chip utilized many of the basic blocks from the six-chip prototype, in an enhanced architecture. The single chip version utilized better layout techniques using higher level design tools. Some of the low level VLSI layout issues are discussed in Chapter 6.

### 4.2 Architecture of the Prototype

The basic structure of the CORDIC SVD processor was discussed by Cavallaro [6].

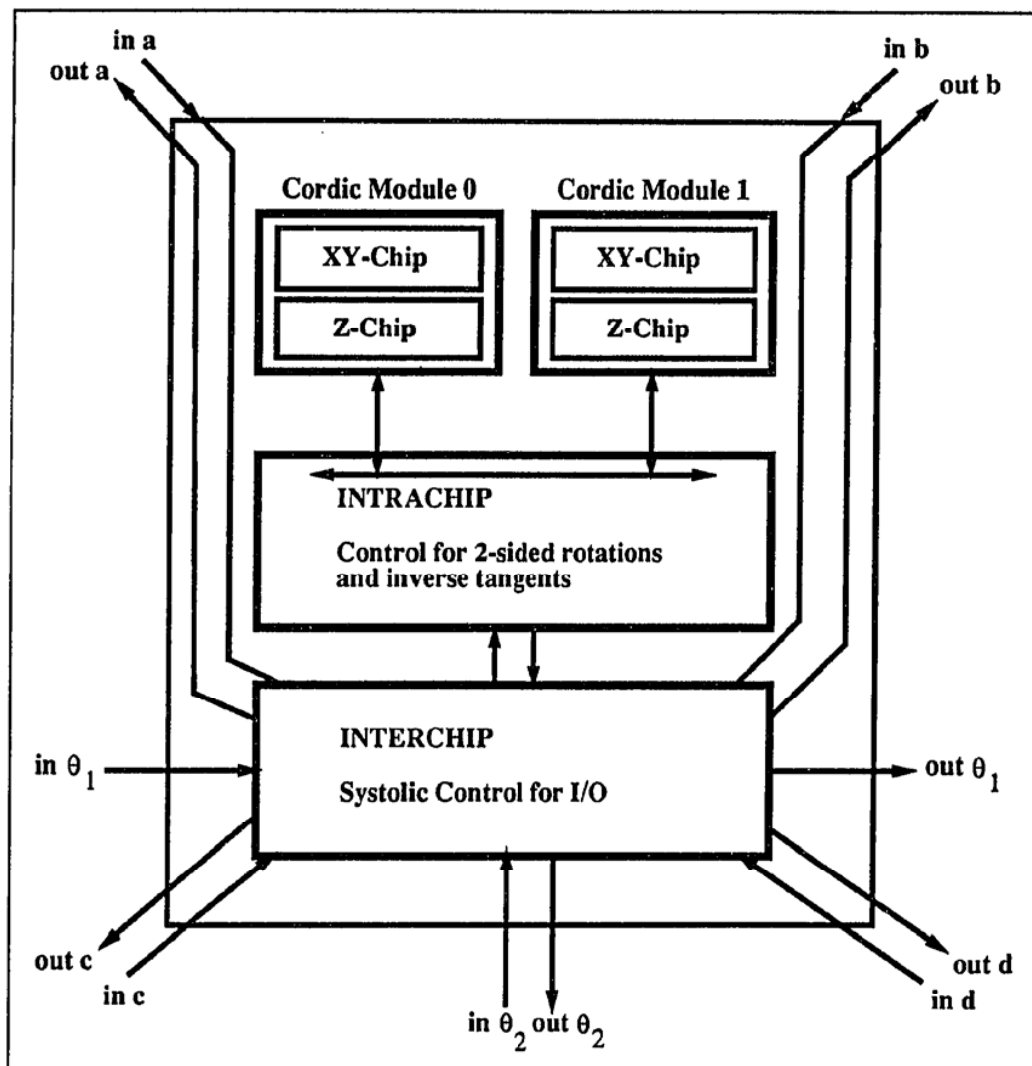


Figure 4.1: Block Diagram of the 6-chip Prototype CORDIC-SVD Processor

This architecture, shown in Figure 4.1, consists of four distinct modules that have been implemented in four different TinyChips. Two of the chips, the *xy-chip* and the *z-chip*, together form one CORDIC module. Maximal parallelism is achieved in the computation of the  $2 \times 2$  SVD by including two CORDIC modules in the processor. The control and interconnection for the SVD processor was implemented in two separate chips. The *intracontroller* chip is used to provide all the internal control necessary to implement the computation of the  $2 \times 2$  SVD. The systolic array control is provided by the *intercontroller* chip. A hierarchical control mechanism provides a way to effectively shield the low level control details from the higher level controllers. At each level the controllers provide a set of macros to the controller next in the hierarchy. In addition, this separation of control allows a degree of concurrency not possible with a single controller. The next few sections describe the individual chips in greater detail.

#### 4.2.1 Design of the XY-Chip

Figure 3.6 on page 41 shows the basic blocks required in an implementation of a fixed point CORDIC processor. The *xy-chip* implements the  $x$  and  $y$  data paths of a single CORDIC module. Each path consists of a 16-bit register, a 16-bit barrel shifter implementing all right shifts up to 15, a 16-bit ripple-carry adder with associated temporary latches at the inputs, and a data switch to implement the cross-coupling implied by the  $x$  and  $y$  CORDIC iterations (Equation 3.4 on page 16). Either CORDIC iterations or scale factor correction iterations are possible on the data stored in the registers. The absence of any on-chip control allows an external controller complete flexibility of the number of iterations as well as the nature of each iteration. Read and write access to the  $x$  and  $y$  registers is possible independent of computation. Several

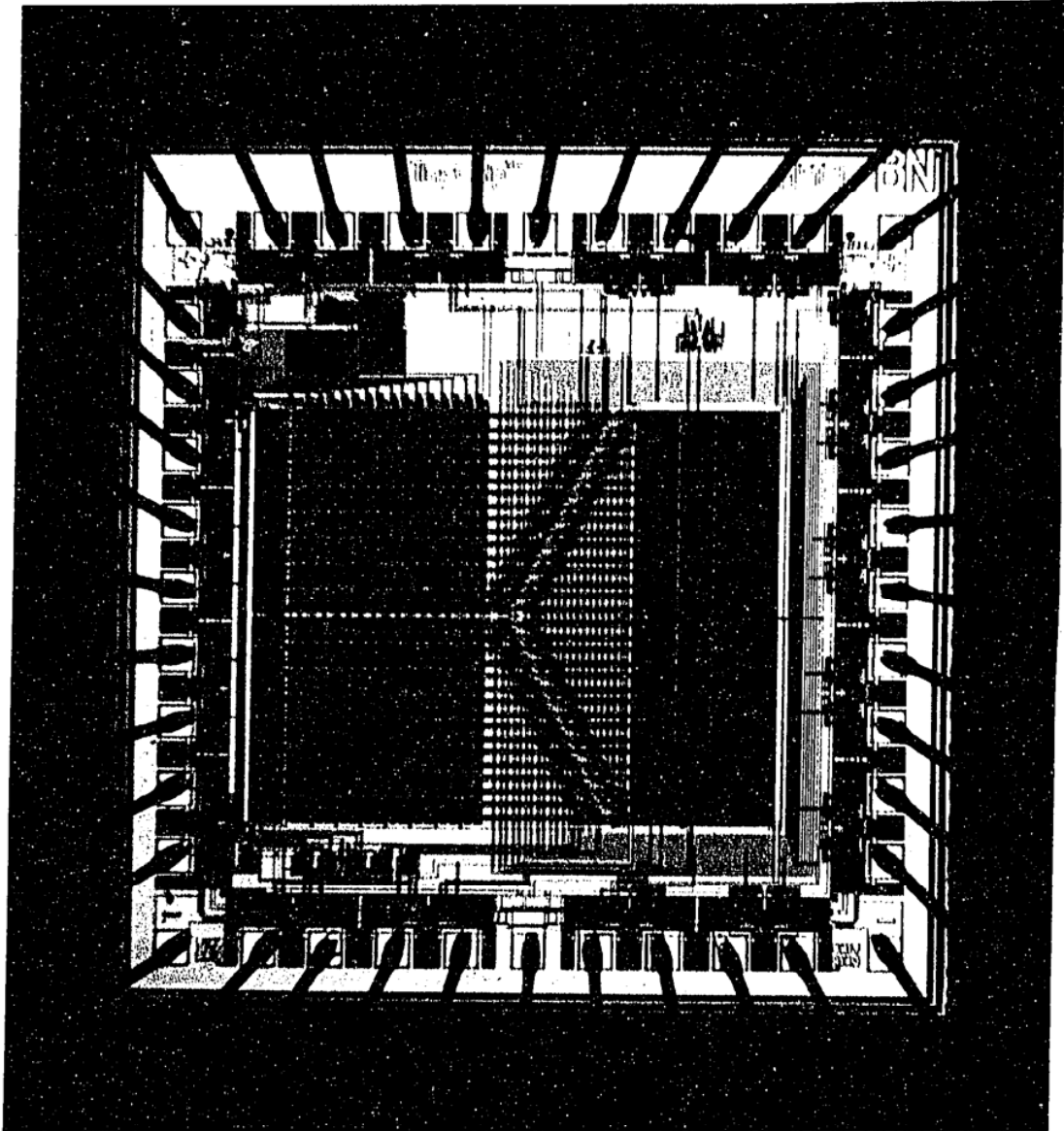


Figure 4.2: Die Photograph of the XY-Chip

versions of the *xy*-chip have been fabricated [7]. Figure 4.2 shows a die photograph of the *xy*-chip.

#### 4.2.2 Design of the Z-Chip

The *z*-chip complements the *xy*-chip to form a complete 16-bit fixed point implementation of the CORDIC module. It implements the *z* data path that consists of a 16-bit *z*-register, a carry-lookahead adder and a ROM table of 16 angles. It also implements a controller to provide all the control signals for the *xy* datapaths in the *xy*-chip and the *z* datapath in the *z*-chip. This controller allows use of the CORDIC module in five modes: Z-reduction, Y-reduction, scale factor correction, single add and subtract, and divide-by-two. These are the only ALU operations required in the computation of the SVD, which allowed fine tuning the controller to perform these operations. The design of this chip was completed by Szeto [33] Figure 4.3 shows a die photograph of the *z*-chip.

#### 4.2.3 Design of the Intra-Chip

The intra-chip controls data movement between a register bank and two CORDIC modules, and sequences the ALU primitives provided by the *z*-chip on the matrix data to implement a  $2 \times 2$  SVD. It allows operation in three modes corresponding to processor behavior in three different positions in the array. If the processor is on the main diagonal of the array, it computes the left and right angles and then uses these angles to diagonalize a  $2 \times 2$  submatrix. The processors P00, P11, P22 and P33 in Figure 2.1 on page 9 are examples of such a processor. Any off-diagonal processor only transforms a  $2 \times 2$  submatrix with the angles received from the previous processors. However, if a processor is diagonally a multiple of three away from the main diagonal, for example processor P03, in addition to the transformation, a delay equal to the



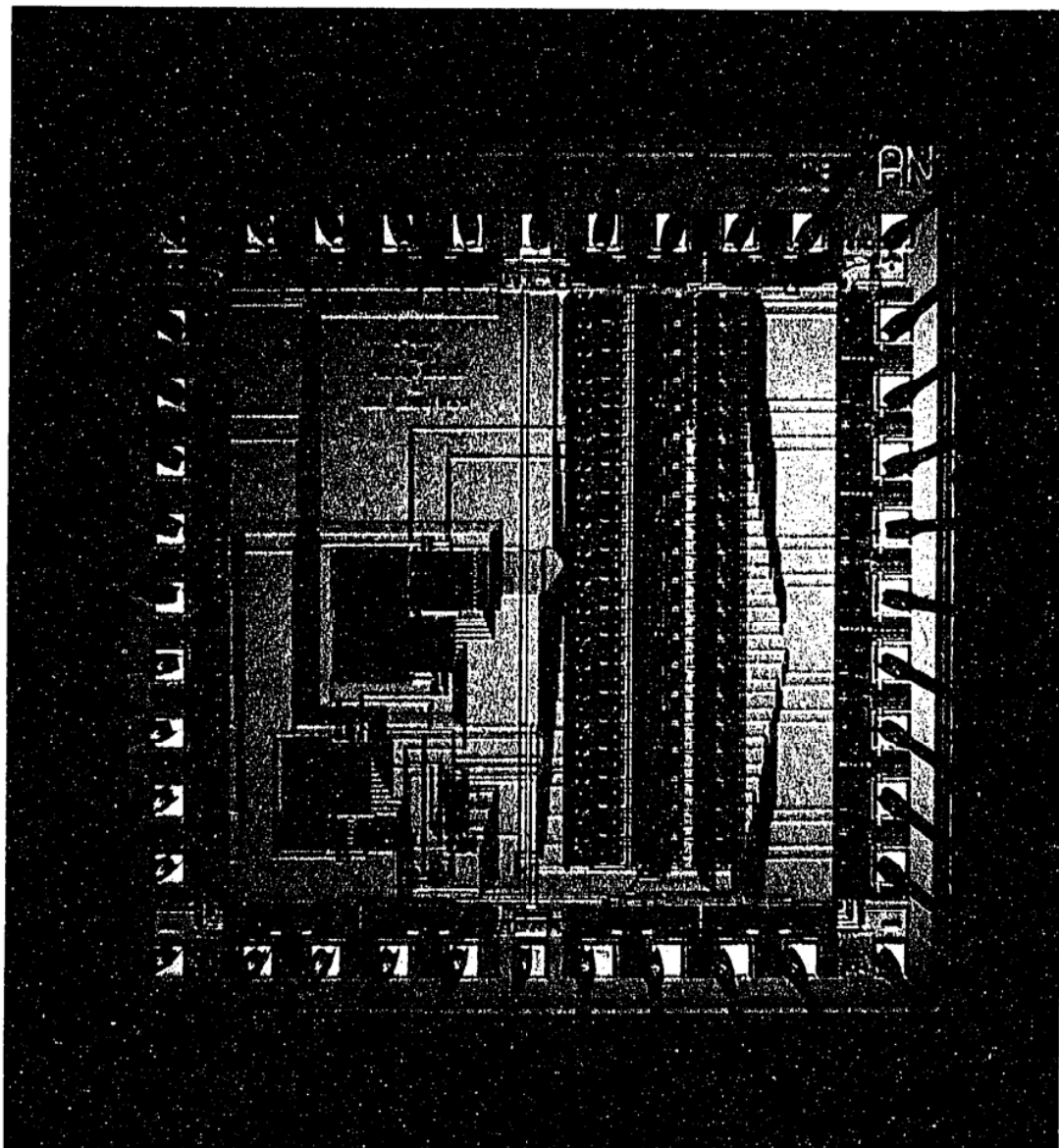


Figure 4.3: Die Photograph of the Z-Chip

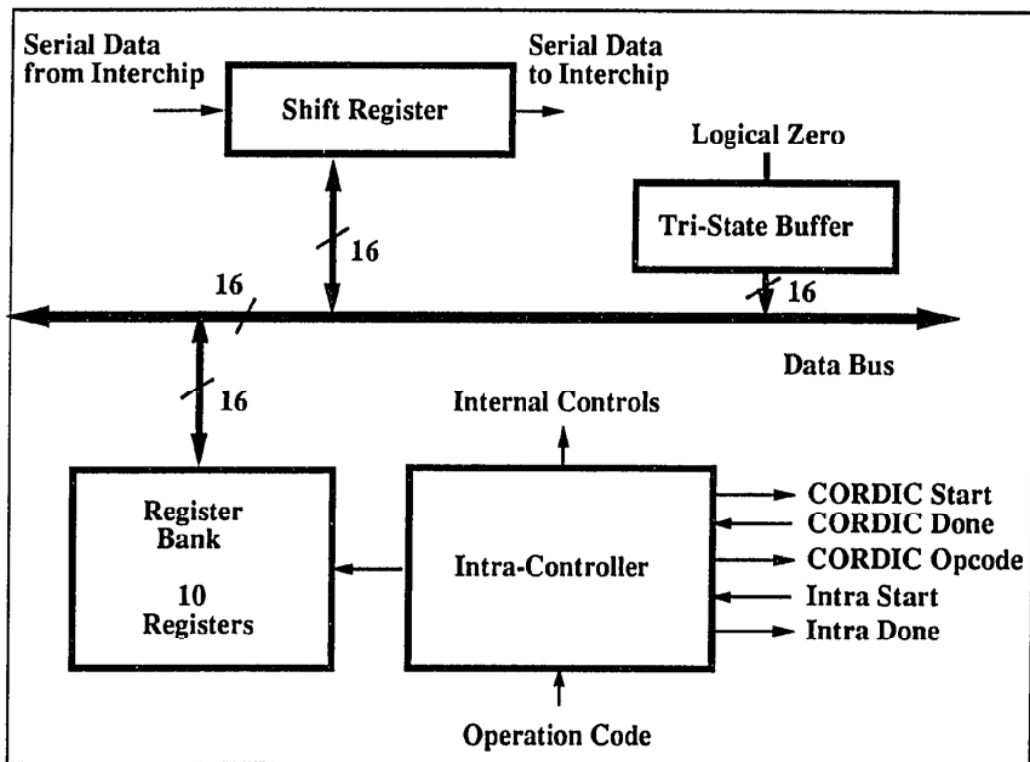


Figure 4.4: Block Diagram of the Intra-Chip

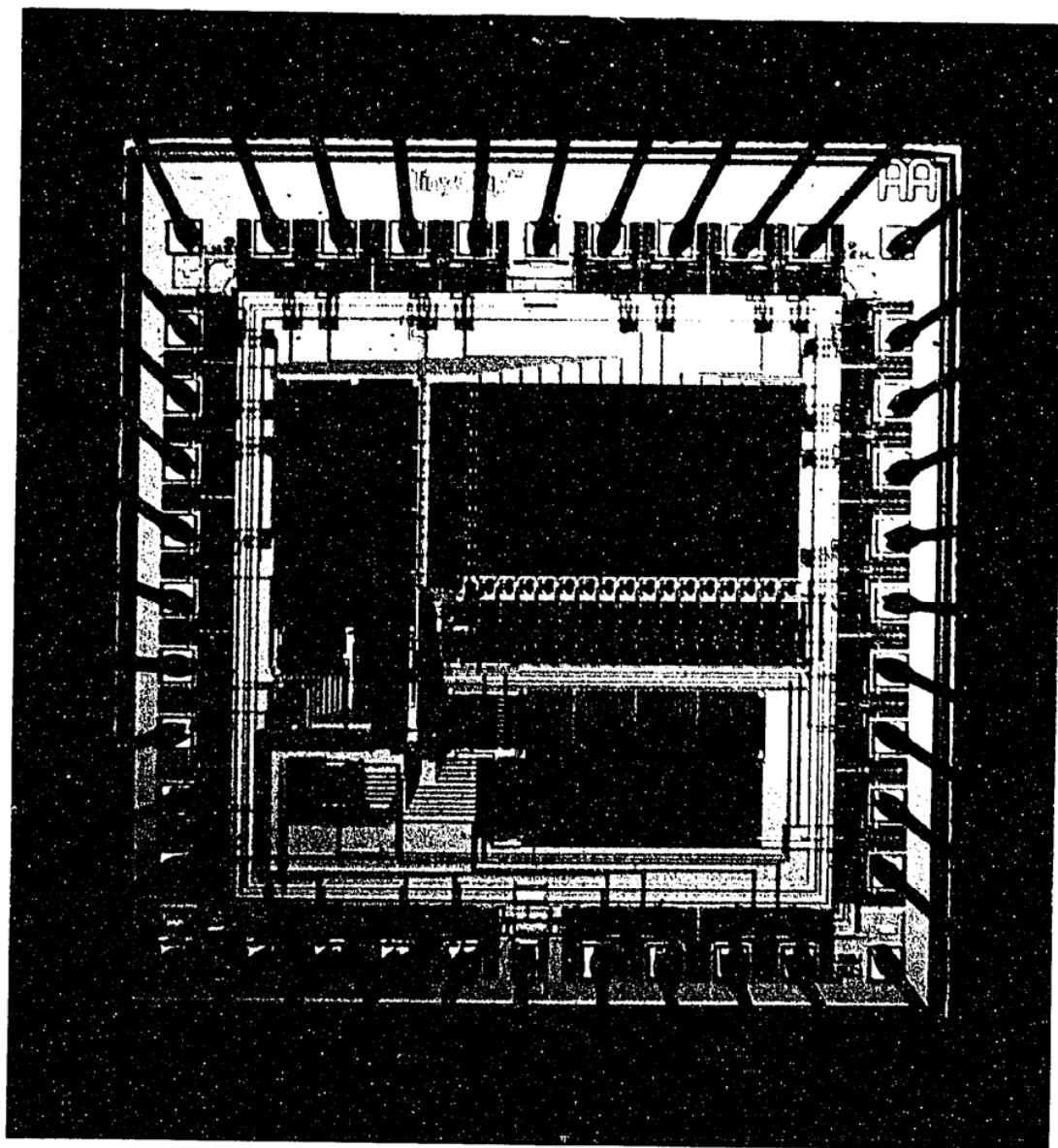


Figure 4.5: Die Photograph of the Intra-Chip

time it takes to compute the angles is included. The delay is necessary to maintain synchronization between all the processors due to the wavefront nature of the array. These three modes allow the same processor to be used in any position of the array in spite of the required heterogeneity. The aforementioned storage is a register bank consisting of ten registers implemented in the intra-chip. A 16-bit bus interconnects these registers and the two CORDIC modules. A block diagram of the intra-chip is shown in Figure 4.4. Figure 4.5 shows a die photograph of the intra-chip.

#### 4.2.4 Design of the Inter-Chip

The inter-chip contains the controller that exercises the highest level of control in the processor. As a communication agent, it implements the systolic array control for the SVD processor. The parallel ordering data exchange [5], which provides a new pair of data elements at the main diagonal to be annihilated, is implemented in this chip. A set of six shift registers, as shown in Figure 4.6, is used as communication buffers to store a copy of the data to be exchanged with the other processors while the intra-chip operates concurrently on its own private copy of the data.

Interprocessor communication is through a set of eight serial links. Four of these links are hardwired to exchange data with the diagonal neighbors. Four other links serve to systolically propagate the left angles horizontally and the right angles vertically. Since the SVD algorithm is compute bound, the bandwidth provided by the serial links is sufficient for the data interchange. In addition, pin limitations prohibit implementation of parallel ports to communicate with neighboring processors. Data is exchanged synchronously between processors. Data is shifted out to the neighboring processor and simultaneously shifted in from it. Synchronization in the array is of utmost importance since no other form of synchronization is provided. This is in accordance with the definition of systolic arrays [21] that emphasizes exploiting the

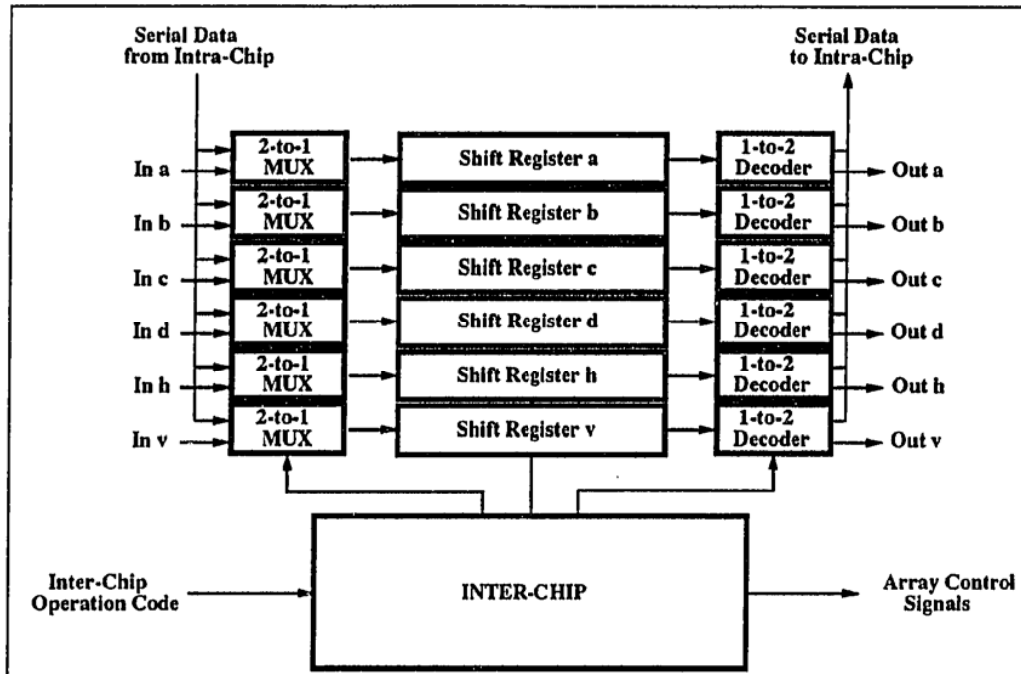


Figure 4.6: Block Diagram of Inter-Chip

special-purpose nature of the array to eliminate the overhead of synchronization. In the prototype, pin limitations forced a serial link for communication of data with the intra-chip. The inter-chip controller also allows initialization of the array by loading the matrix data. The inter-chip was designed by Szeto [3] Figure 4.7 shows a die photograph of the inter-chip.

### 4.3 The CORDIC Array Processor Element

The CORDIC array processor element (CAPE) is a single chip implementation of the CORDIC SVD processor [17]. This chip incorporates elegant solutions to the problems encountered in the six-chip prototype. CAPE also includes additional details necessary to construct a working systolic array. CAPE essentially consists of the same basic cells

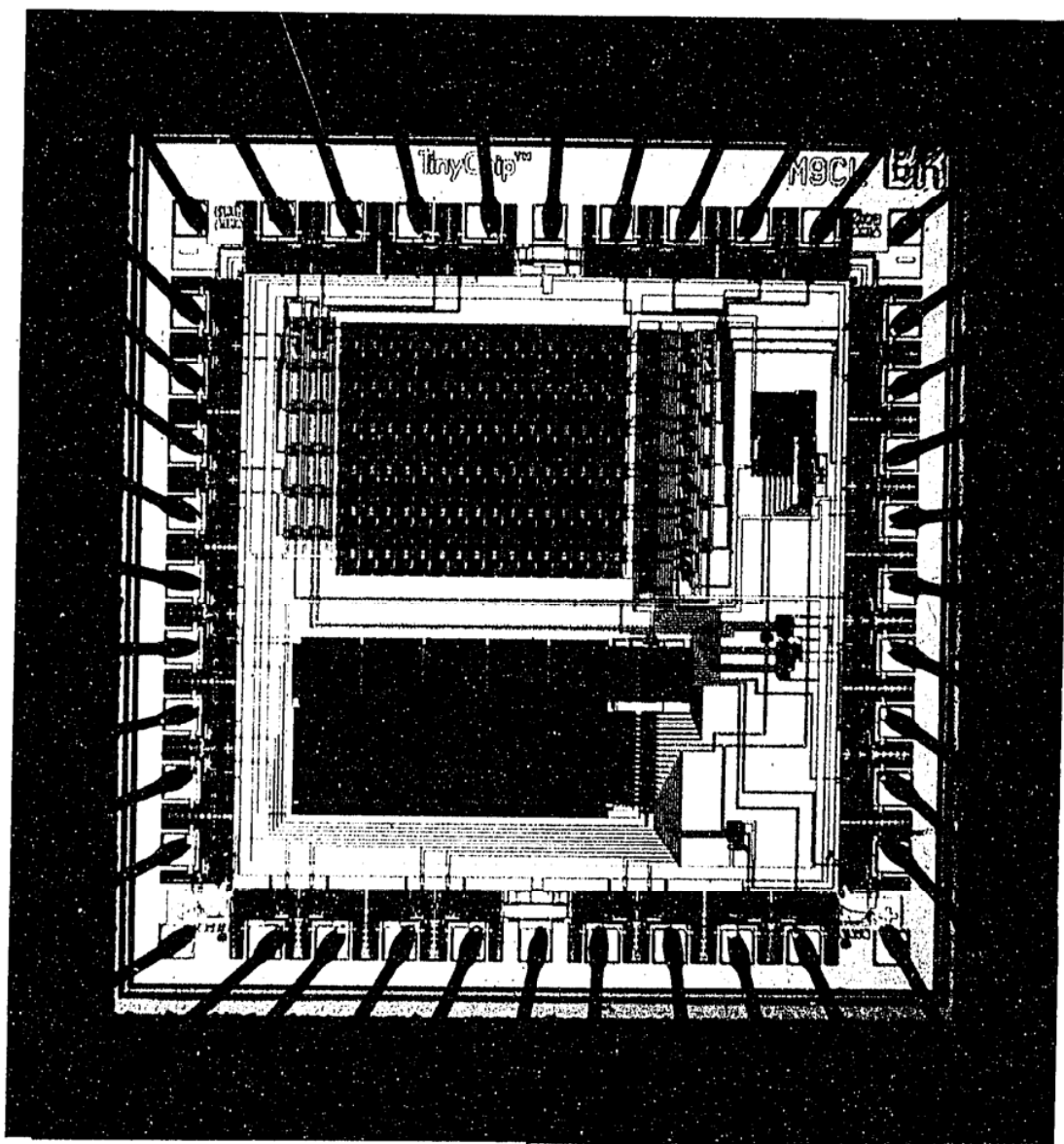


Figure 4.7: Die Photograph of the Inter-Chip

as the prototype. Many of the architectural enhancements were made to reduce the communication required within the chip. A completely new loading scheme was added to facilitate easy interface with a variety of hosts. The next few paragraphs describe the details of all these improvements.

A study of the prototype CORDIC units indicated that better numerical accuracy could be achieved with some modifications to the modules as described in Chapter 3. To improve the numerical accuracy of the inverse tangent calculations, a normalization shifter was included in each of the  $x$  and  $y$  data paths. These shifters are activated during Y-reduction by two leading zero encoders that monitor the magnitude of the  $x$  and  $y$  data. Simulations indicate that this scheme provides several extra bits of accuracy in many cases.

The CORDIC modules in the prototype converge only for vectors in the first and fourth quadrants while operating in the Y-reduction mode. The modified scheme uses the exclusive-or of the sign bits of the  $x$  and  $y$  registers to control the next iteration. This effectively increases the range of convergence for the CORDIC modules. In addition, the above scheme adds symmetry to the  $x$  and  $y$  data paths since they cannot be distinguished externally. This allows the same CORDIC module to rotate a vector in either clockwise or anticlockwise direction. Some data movement is rendered unnecessary by the above method, for example, if the data to be used in the next computation already resides in the  $x$  and  $y$  registers in the wrong order, since the definition of the  $x$  and  $y$  registers can be interchanged, no data movement is required.

The  $z$ -chip was modified to allow mapping the  $x$ ,  $y$  and  $z$  registers as part of the register bank. This allows reuse of the data left in the registers by a previous computation, which is important when attempting to cut down the communication overhead. A study of the prototype SVD processor showed that there was a large overhead incurred in moving data to the appropriate registers between computations. A double

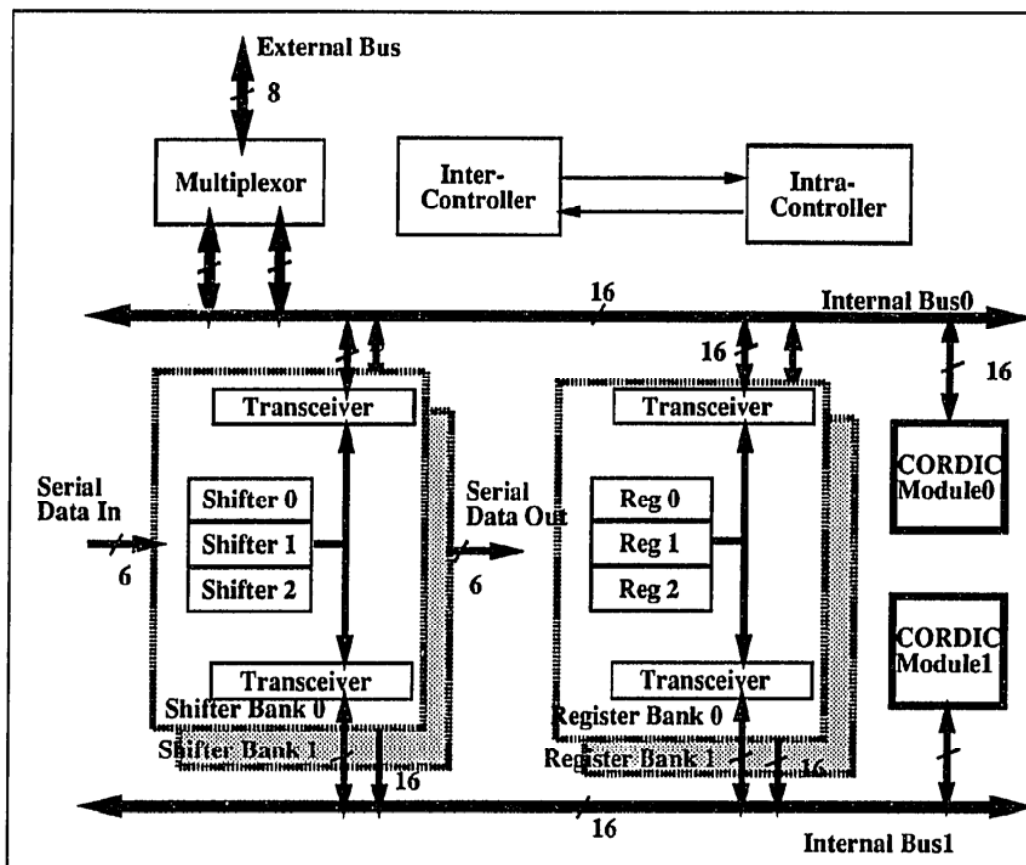


Figure 4.8: Internal Architecture of CAPE

bus architecture, as shown in Figure 4.8, was used to minimize this communication. A separate CORDIC module is hardwired to communicate on each bus. The set of six registers is split into two banks that can communicate with either bus allowing simultaneous access to two different registers. The number of registers required in the architecture was reduced from ten to six due to mapping of the CORDIC registers as general purpose registers. The double bus allows concurrent data movement to the two CORDIC modules, which reduces communication cycles by half. Using the double bus architecture did not cause an increase in the area of the chip. This was a



result of the reduction in the size of the intra-chip controller to half its original size. This corresponds to a reduction in the number of states due to greater concurrency provided by the architecture.

Special cases that occur in the SVD computation require extra hardware. Two zero comparators were included to detect the case  $\tan^{-1}(0/0)$ . This is an invalid computation on the CORDIC modules that return a code equivalent to the floating-point *NaN*. The SVD algorithm allows such computed angles to be replaced by zero. The zero comparators compare the data on the bus with zero and return a control signal to the intra-chip controller to handle this special case. A literal constant zero can be driven onto either bus and hence into any register. This allows forcing a register to store a zero to handle special cases.

The inter-chip has been modified to handle an improved array loading scheme. The prototype did not make any special provision to efficiently load the array. Since CAPE is to be used in a systolic array in a real system, the interface with the host has been refined. A parallel port has been included along with the requisite handshake signals. The controller implements an asynchronous loading scheme that allows interfacing with a host running at an independent clock rate.

The prototype inter-chip was pin limited and hence forced to communicate with the intra-chip through a serial link. This restriction is no longer valid in the single chip implementation. Consequently, this serial link was replaced with an internal parallel bus that reduces the time to exchange data items between the intra-chip and the inter-chip. The rest of the inter-chip was redesigned with only minor changes.

#### 4.4 Issues in Loading the Array

Data loading is a key issue in the design of a systolic array. Systolic architectures typically include data loading as part of the algorithm. Data is continually loaded as new results are unloaded. A systolic loading scheme increases the throughput of the array by pipelining several different computations. An important problem in the design of any pipeline is to keep the pipe full at all times, since it affects the throughput. The time taken to fill or empty a pipeline is analogous to the non-parallelizable section of a program, which limits the speedup that can be achieved just by increasing the number of processors [32]. In a systolic array, computation typically begins in a subset of the processors and then propagates to other processors. The goal of a loading scheme is to overlap communication and computation to the largest degree possible by loading the processors in the order that the data is required.

In the SVD array, as illustrated by Figure 2.2 on page 11, computation starts at the main diagonal and propagates as diagonal waves of activity towards the corners of the array. Consequently, a systolic loading scheme should load the main diagonal as early as possible and continue with the other super and sub-diagonals at a rate that is not slower than the rate of computation. The computation of the SVD in the array takes  $O(p \log p)$  time, where each sweep takes  $O(p)$  time. Since all the array data is required in one sweep, the loading algorithm has to be at least  $O(p)$ . Since, the matrix data itself does not propagate systolically as part of the SVD algorithm, currently, the loading algorithm and the SVD algorithm are independent.

In a square array, it is necessary to load a total of  $O(p^2)$  data elements. Thus any  $O(p)$  loading algorithm requires  $O(p)$  entry points. Hence, it is not possible to build a *scalable* SVD array with a constant number of entry points. This is not a problem, however, in some real-time processing applications. Some applications

combine analog sensors with each processor, such that data is loaded directly from the sensor to the processor, providing  $O(n^2)$  entry points. In such an application, loading is not a problem.

The simplest loading scheme involves loading the array a column at a time. Assuming that the data is loaded from the left edge, the data for the right edge is loaded first, and as new columns are loaded, the previous columns are systolically propagated right. This is an  $O(p)$  loading algorithm. However, the main diagonal is not loaded till the entire array has been loaded and hence does not allow any overlap of communication with computation.

A systolic loading scheme that starts at the diagonal, needs to load the main diagonal, followed by the next super and sub diagonals and so on. This scheme requires each processor to know exactly where it is located in the array, so it can propagate the correct number of elements to the right and left. The algorithm requires  $O(p)$  time to load. However, this loading scheme requires a complex control to load and unload the array. The main drawback is that it requires multiple parallel ports in each processor, which is a problem due to pin limitations.

#### 4.4.1 Data Loading in CAPE

In CAPE multiple buses are used to load the array (Figure 4.9) in time  $O(p)$ . All the processors on a row are connected by a bus. Only one processor is enabled permanently, while the others are daisy-chained. After loading an appropriate number of words, the currently enabled processor enables the next processor in the chain. The array consists of  $p/2$  entry points, one at each of the diagonal processors. The host has to provide the data in the correct order, which depends on how the processors have been chained.

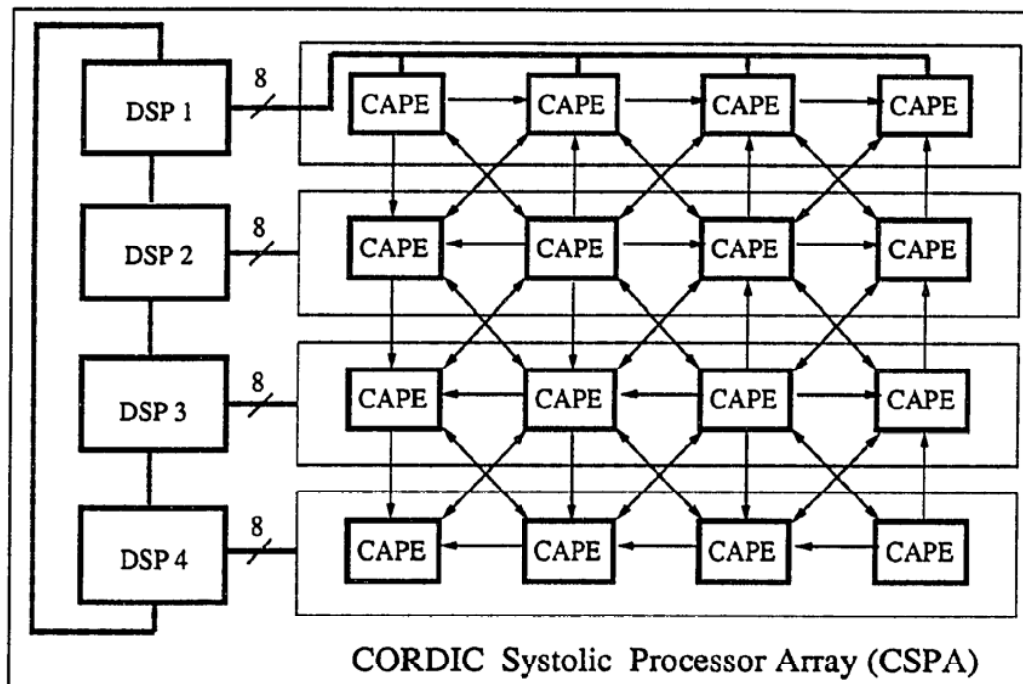


Figure 4.9: A square array of CAPE chips connected to a host, a linear array of DSP processors, for Robotics

In large arrays global connections will limit the clock rate due to long paths. This problem has been avoided in CAPE by using an asynchronous handshake to load data into the processors. This decouples the speed of loading from the speed of computation. The array can run at a clock independent of the speed at which the array can be loaded. Since the processors have a fixed latency to acknowledge, it is possible to load the diagonal processor and start computation before the entire array has been loaded. Asynchronous loading allows the array to be interfaced with a host running at an independent clock speed. For very large arrays, the speed at which the array can be loaded reduces. When such a limit is reached, the speed of computation can be decreased to achieve a graceful degradation as the size of the array increases.

This chapter described the design of the CORDIC-SVD processor. Several enhancements can be made to the SVD array, with minor modifications to the design of the processor. Some of these alternative array designs are described in the next chapter.

## Chapter 5

### Architectures to Improve Processor Utilization

#### 5.1 Introduction

In a system with a large number of processors, keeping them busy at all times presents a formidable task, leading to intervals when processors wait for some other event to occur in the system. Sometimes, however, this idle time can be used for tasks normally regarded as overhead. An example of this is the fault-tolerance reconfiguration scheme [9] proposed by Cavallaro, where the idle processors backup the functionality of a neighboring faulty processor. Use of this time for fault-detection is yet another example. However, if a higher throughput is desired, it would be possible to reduce the idle time by pipelining several different computations. Idle time is of concern in the CORDIC-SVD array, since at any given time two-thirds of the processors are idle.

This chapter focuses on methods to eliminate the idle time or to utilize it to perform some useful computations. An array, in which the angles from the diagonal processors are broadcast along a row or column, can attain a speedup close to 100% over the current design. A variation of this scheme that avoids the broadcast and yet achieves the same speedup is discussed in Section 5.3. This architecture can be useful to achieve a higher throughput if only the singular values need to be computed by the array.

Many applications require the complete SVD decomposition including the left and the right singular vectors. A modification to the current array design allows the

complete decomposition to be computed in the same time that it takes to compute the singular values. The impact of this algorithm on the design of the processor is discussed in Section 5.4.

A systolic array, by definition, does not require any globally synchronous control signals, except perhaps the clock. Many arrays require a globally connected *start* signal to achieve synchronization between a large subset of the processors. Section 5.2 discusses an architectural alternative that avoids such a signal, allowing greater scalability.

## 5.2 Architecture for Systolic Starting

The CORDIC-SVD processor has been designed assuming a global *start* signal to synchronize all the processors on the main diagonal. This type of a signal is omnipresent in most systolic array designs to facilitate easy understanding of the timing issues in the array. In most arrays, the processors retain a certain degree of simplicity that allows elimination of these signals by skewing the computations among processors, after the initial design phase. The SVD processor is no exception and lends itself to this sort of skewing, permitting a system *start* signal that has a constant fan-out independent of the size of the array. A *start* to the diagonal processor P00 in Figure 5.1 is propagated diagonally, with a clock cycle delay at each processor. Consequently, the diagonal processors are no longer in unison. This affects the timing requirements of the rest of the array. Special care is required while interconnecting the control and data-paths of the various processors, which were hitherto relatively easy to comprehend. Delays have to be inserted at the appropriate places to solve timing problems.

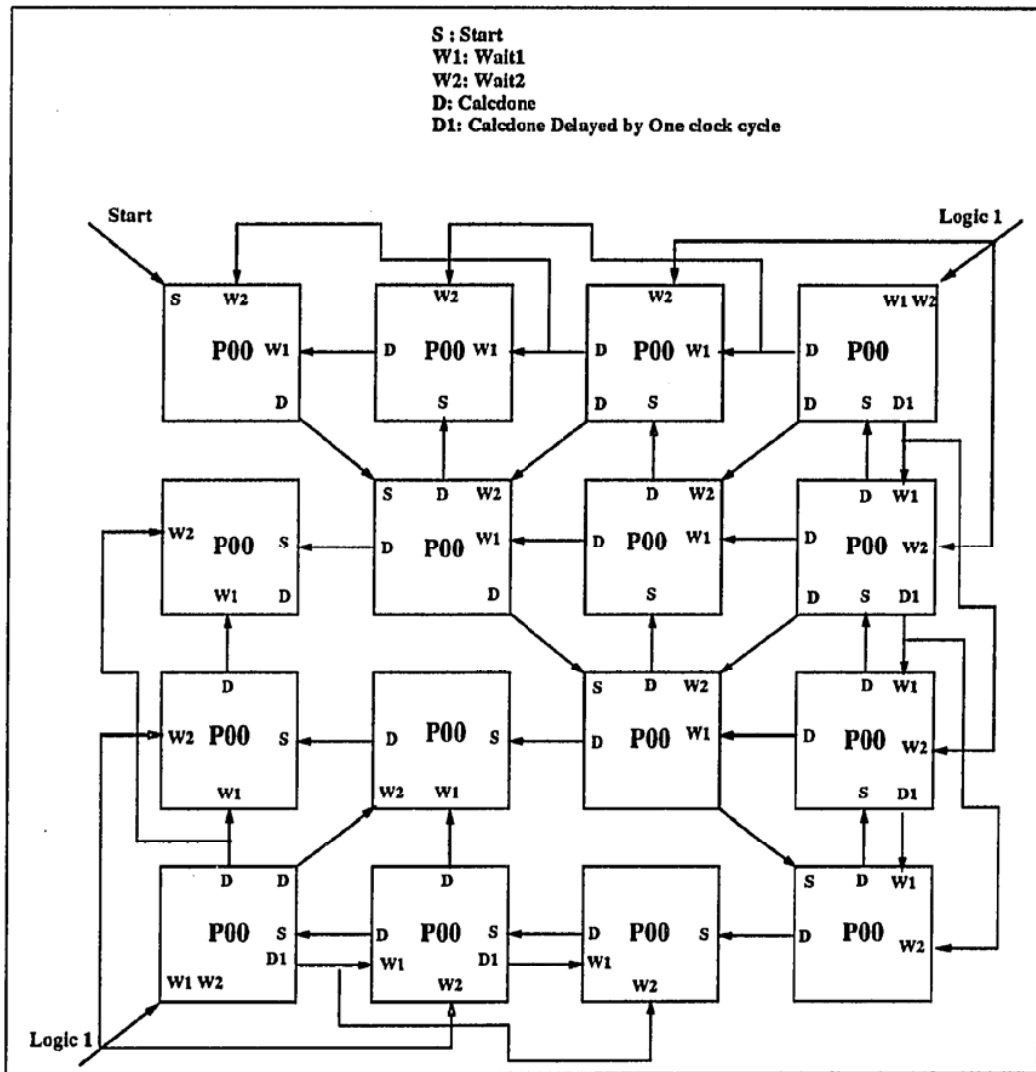


Figure 5.1: Interconnection of Control Signals for Systolic Starting



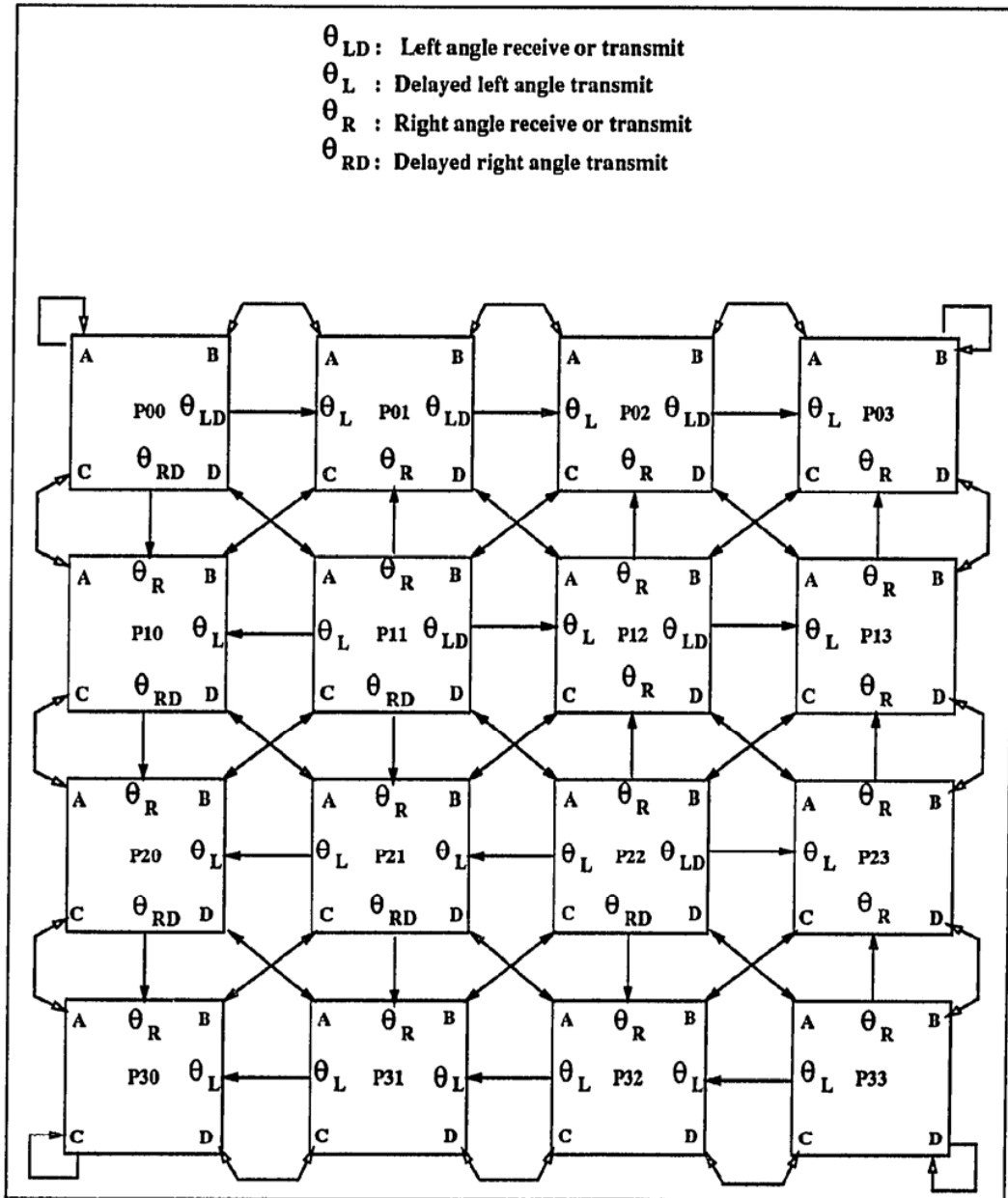


Figure 5.1 and 5.2 show an interconnection for the CORDIC-SVD processor that results in a consistent operation. The most important modification made to the SVD processor is the introduction of a clock cycle delay in a few selected signals and data links. In addition to a *normal done* signal, a signal delayed by one clock cycle, called the *delayed done*, is required to maintain coherence at the right and bottom edges of the array. Angle transmission from the diagonal processors to the *east* and *west* neighbors is skewed. The angle transmitted *east* is delayed one clock cycle to account for the skew in the computation along the diagonal. Such a delay is not required in the angle transmitted *west*. A similar delay is required in the angle transmitted *south* but not in the angle transmitted *north*. The only other change required in the processor is to delay the exchange of data with the *south-east* processor by one cycle. All other control signals and data exchanges in the current design remain unmodified. An interconnection of these modified processors, as shown in Figure 5.1 and 5.2, achieves the goal of eliminating the global *start* signal. Modification of the above scheme to eliminate global signals in other arrays requires a careful study of the timing dependencies, but, is nevertheless straightforward.

Figure 5.3 shows snapshots of array taken at various time instants and helps illustrate the inner working of the array. The processors that are highlighted, are the processors which are computing at the given time instant. The highlighted communication links indicate an exchange of data during that cycle. The communication is assumed to take only one clock cycle for simplicity, but the analysis can easily be extended for more cycles. Some of the salient features of this figure are:

- The *start* signal to processors P00 is provided at time instant  $t_1$ . This signal is propagated systolically down the diagonal resulting in processors P00, P11, P22, P33 starting computations at time  $t_1+1$ ,  $t_1+2$ ,  $t_1+3$  and  $t_1+4$  respectively.

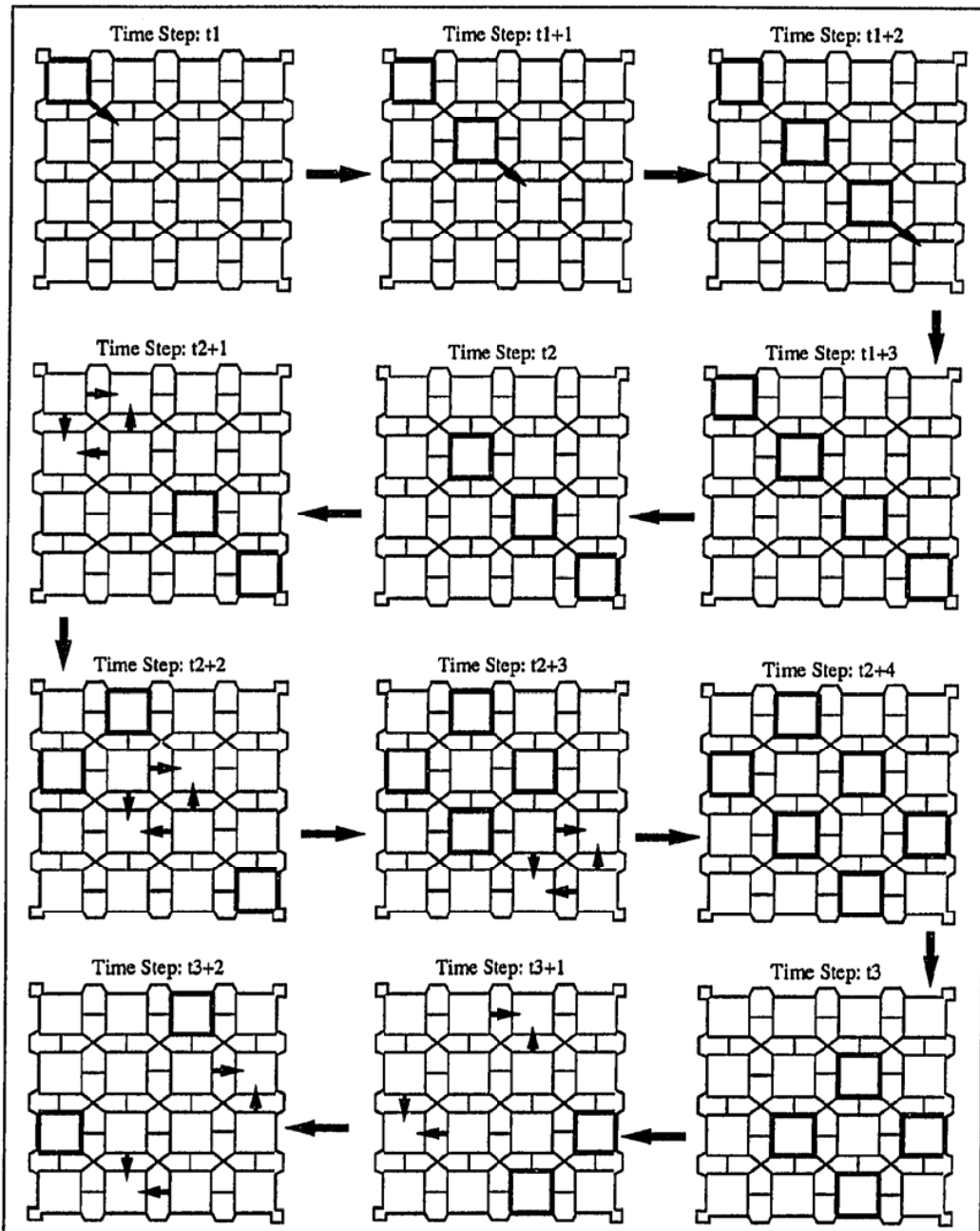


Figure 5.3: Snapshots of a Systolic Starting scheme

- Processor P00 completes its computations at time  $t_2$ . However, it delays the transmission of the angles to processors P01 and P10 by one clock cycle. Processor P11 stops computing at time  $t_2+1$  and immediately starts sending out the angles to processors P01 and P10 completing the rendezvous of the left and right angles at the processors P01 and P10. The same sequence of operations continues all along the diagonal. Thus processors P01, P10, P21, P12, P32, P23 start computing at time  $t_2+2$ ,  $t_2+2$ ,  $t_2+3$ ,  $t_2+3$ ,  $t_2+4$ ,  $t_2+4$ , respectively.
- Processors P01 and P10 finish first at  $t_3$ . The diagonal processor P00 on receiving the wait1 signal, exchanges matrix element  $a$ , but delays exchanging data element  $d$  by one clock cycle. This allows it to synchronize with processor P11, which receives its wait1 at  $t_3+1$  and shifts out its matrix data element  $a$ , which it exchanges with the data element  $d$  of processor P00. The same sequence continues down the diagonal.

Using a systolic starting scheme requires an extra  $n/2$  clock cycles to finish, as compared to the scheme with global *start*. The extra cycles form an insignificant portion of the total time, 0.02% for an  $8 \times 8$  matrix, and hence do not affect the performance of the array.

### 5.3 Architecture for fast SVD

Many applications utilizing the SVD require only the singular values of a matrix. Often, some parameter of the matrix, like the 2-norm or the Frobenius-norm [15], is desired and can be evaluated from the knowledge of the singular values alone. Rank determination and condition estimation problems, too, require only the singular

values. Hence, a speedup in the computation of the singular values alone, can be useful in such applications.

A simple mechanism for reducing the idle time in an array is to use broadcast. If the diagonal processors compute the angles, which are then broadcast along the row or column, the actual 2-sided rotations can then be computed by all the processors in unison, followed by the data exchange. Such a scheme would permit the diagonal processors to compute the angles every  $3\frac{1}{4}$  CORDIC cycles as opposed to  $7\frac{3}{4}$  CORDIC cycles in the current design. This results in an array that is twice as fast as the current design. In small arrays, of the order of  $10 \times 10$ , broadcast of angles is a viable solution. However, for scalability, this broadcast can be easily converted into a systolic scheme using the techniques described in Section 5.2.

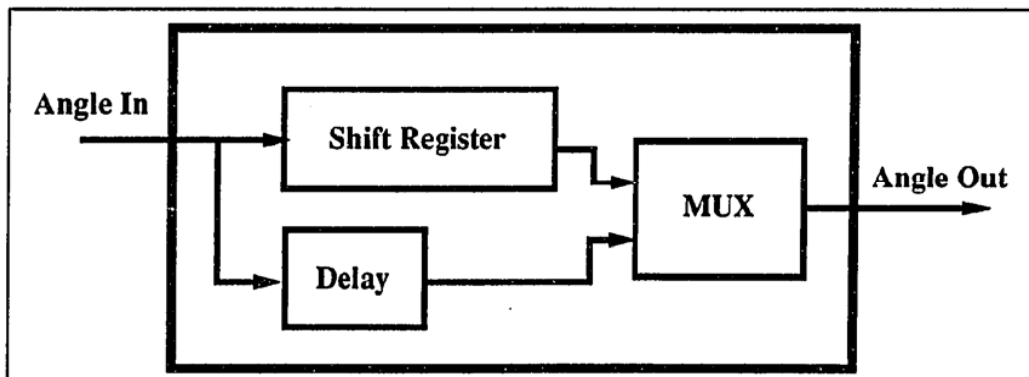


Figure 5.4: Hardware required to allow serial propagation of angles with only a clock cycle delay

The broadcast in the array is avoided by skewing the operations of adjacent processors along a row or a column by one clock cycle. A simple hardware “trick” shown in Figure 5.4 can be used to pass an angle down a row or a column with only a single clock cycle delay. Using this scheme, the array operation reduces to the following algorithm:

- The diagonal processors compute a new set of angles.
- The angles are propagated down the rows and columns with only a clock cycle delay between the angles received by adjacent processors. Thus every processor mimics the operation of the previous processor with a single clock cycle delay.
- After a processor finishes receiving the angles, it starts the 2-sided rotation. Thus all the processors compute the 2-sided rotation with a single clock cycle delay. This constant delay allows elimination of the *wait1* and *wait2* signals. In CAPE, the control signals, *wait1* and *wait2*, indicate the end of computation one diagonal and two diagonals away, respectively.
- Every processor that completes a rotation can start data exchange with the following rules:
  - After one clock cycle delay, the processor can begin exchange of data, which it should on receiving *wait1*,
  - After two clock cycles delay, the processor can begin exchange of data, which it should on receiving *wait2*.
- After the data exchange the entire process is started again.

A systolic starting scheme as described in the previous section can be used to eliminate the global *start* signal. This introduces additional skews in the array. Special care is required when exchanging data at the boundaries of the array. An extra bit of information is required by every processor to determine if it is on the boundary.

## 5.4 Architecture for Collection of $U$ and $V$ Matrices

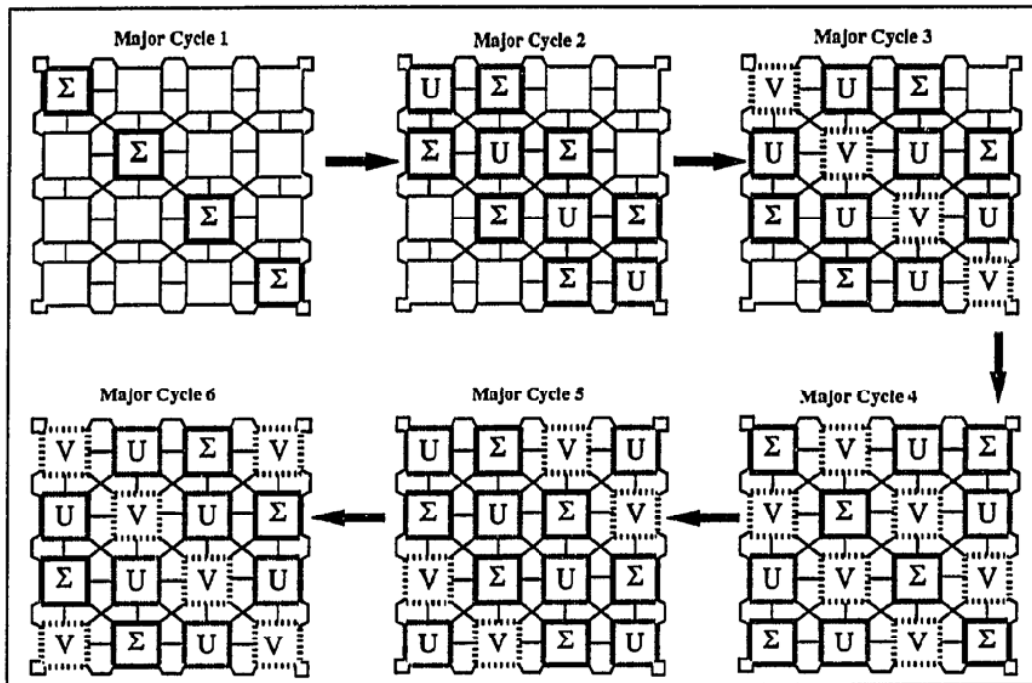


Figure 5.5: Snapshots of an Array Computing  $U$  and  $V$

In spite of the assertions made in Section 5.3, many applications do require the computation of the left and right singular vectors for further processing. An example of such an application is the inverse kinematics engine for a robot, the RICE-OBOT 1, which relies on the singular values and the vectors for singularity avoidance. A rather surprising result obtained while studying ways to utilize the idle time is that it is possible to compute the  $U$  and  $V$  matrices along with the singular values in the same array in about the same time that it takes to compute the singular values alone.

The Jacobi algorithm for computing the SVD applies a sequence of left and right orthogonal transformations to the original matrix to diagonalize it. The product of the left transformations is the final  $U$  shown in equation 2.5. Similarly, the product of all

the right transformations is the transpose of  $V$ . Thus computation of  $U$  corresponds to applying the left transformation alone to the  $U$  that has been accumulated. The computation of  $V$  is analogous; it involves updating the accumulated  $V$  with a right transformation. The matrices  $U$  and  $V$  can be stored as  $2 \times 2$  matrices in each processor, in extra banks of registers provided to store them.

If the CORDIC module does not provide a single-sided scale factor correction, it is necessary to compute the  $U$  and  $V$  updates as 2-sided transformations, in order to perform a 2-sided scale factor correction.

$$\begin{aligned} \begin{bmatrix} \cos \theta_l & \sin \theta_l \\ -\sin \theta_l & \cos \theta_l \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} &= \begin{bmatrix} \cos \theta_l & \sin \theta_l \\ -\sin \theta_l & \cos \theta_l \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta_l & \sin \theta_l \\ -\sin \theta_l & \cos \theta_l \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \begin{bmatrix} \cos 0 & \sin 0 \\ -\sin 0 & \cos 0 \end{bmatrix} \end{aligned}$$

The missing rotation can be simulated as a rotation with an angle zero, followed by a 2-sided scale factor correction. The idle time in the processors corresponds to two such identical operations in the other processors. Hence, the computation of  $U$  and  $V$  can be interleaved with the computation of the singular values. This architecture requires eight extra registers in each processor and modified intra and inter-controllers.

The data elements of the  $U$  and  $V$  matrices have to be exchanged between processors in a manner identical to the exchange of the data elements of the principal matrix. Figure 5.5 shows snapshots taken of this array and illustrates the manner, in which the three separate waves of computation are interleaved.



## Chapter 6

### VLSI Implementation Issues

#### 6.1 Design Methodology

A structured VLSI design approach following the Mead and Conway methodology [27, 37] was used in the design of all the chips for the CORDIC-SVD processor. Static complementary (CMOS) logic was used to simplify design. In addition, a formal two-phase clocking strategy was imposed [19, 20] to allow proper functionality independent of the propagation delays.

All the chips comprising the prototype were developed using the *Magic* [29] layout editor and other VLSI tools from the University of California, Berkeley. *Magic* allows a hierarchical design, which facilitates layout in a bit-slice manner. Programmable Logic Arrays (PLAs), used as controllers, were generated automatically from a high level description. All the other cells were custom designed. *Magic* provides limited routing facilities, which are sufficient for small chips. A large chip like CAPE, however, cannot utilize this facility. The amount of custom design involved in this approach makes design using *Magic* a time consuming operation.

The term *Silicon Compilation* has been coined for high level CAD tools that allow generation of VLSI chips almost completely from a high-level description, like VHDL. An alternate set of CAD programs, *Octtools*, allows this kind of a design. *Octtools* [31] take an object-oriented approach to VLSI design. This allows extensive functional simulation of an unplaced, unrouted chip, generated from a high level description.

Actual placement and routing of the chip can be postponed until the description is completely debugged. The different modules are generated from the high-level description of the cells, using libraries of standard elementary cells. A placement tool, based on the simulated annealing technique, finds an optimal placement of the modules to minimize a variety of costs, like wire length and chip area. Finally, a routing package physically routes these cells. Additional post-processing steps reduce the number of vias and compact the layout to satisfy the minimum spacing rules. Design using these tools can reduce the tedium of VLSI design considerably.

The single chip, CAPE, utilized the advanced placement and routing features provided by *Octtools*. Most of the modules were scavenged from the original prototype, and modified to improve performance, using the *Magic* editor. These modules were then converted to *Octtools* format to place and route the final chip. This routed chip was then converted back to *Magic* format for final placement and routing of the pads.

Functional simulation was performed using the switch-level simulators *esim* and *irsim*. Longest path delays were identified with *crystal*. Detailed simulation at the circuit level was performed for critical cells using *spice*. The circuit design of some of the cells was modified to allow functional simulation using the switch-level simulators. For instance, an extra inverter was included to buffer the output of each latch to prevent back-propagation and allow functional simulation. This simulation is essential to eliminate any design errors, since fabrication is a costly process.

The six-chip prototype was designed as four TinyChips, fabricated on  $2200 \times 2200 \mu$  die using a  $2 \mu$  CMOS n-well process. The transistor count for the different chips varied between 4500 and 6000 transistors. The single chip version, CAPE, was designed on a  $6900 \times 5600 \mu$  die and contained about 26000 transistors. A plot of this chip is shown in Figure 6.1.

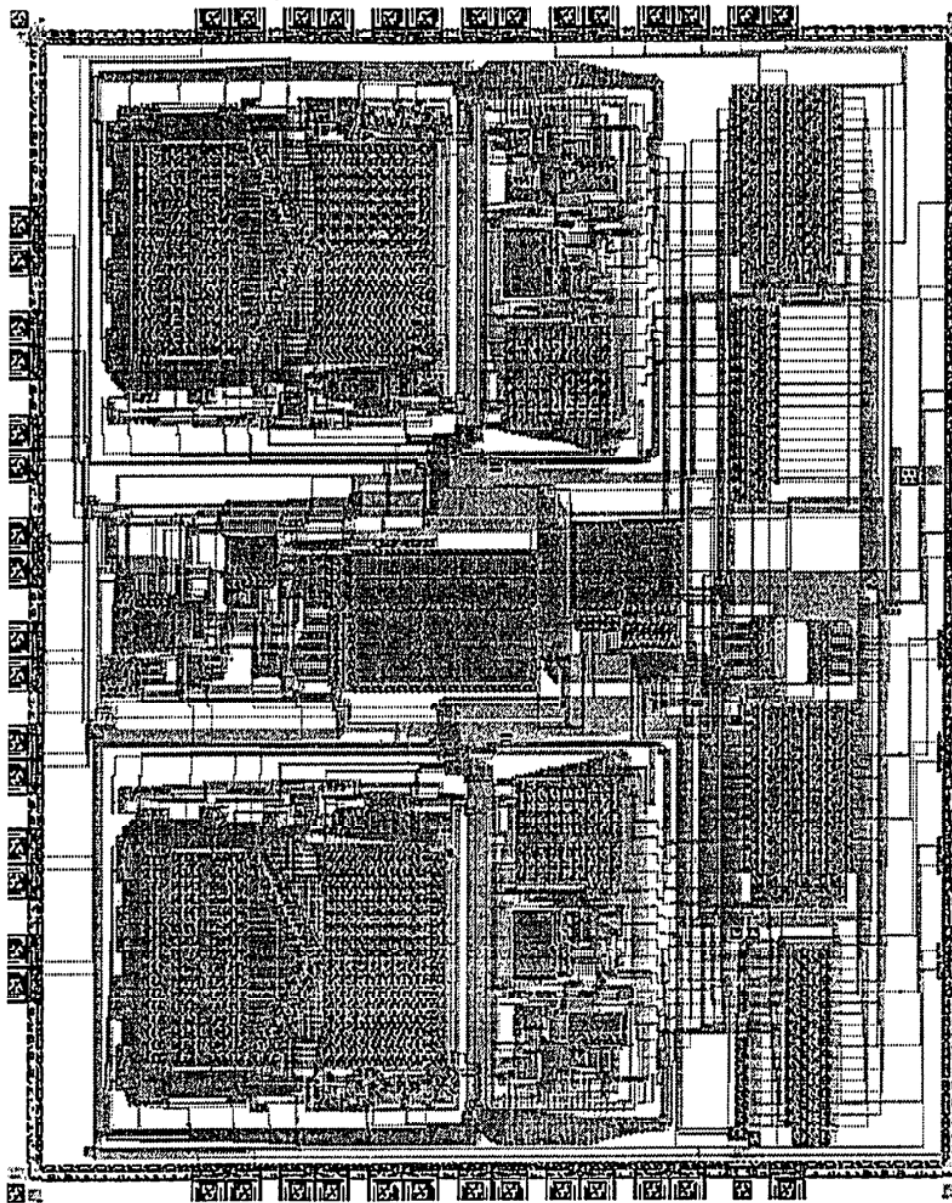
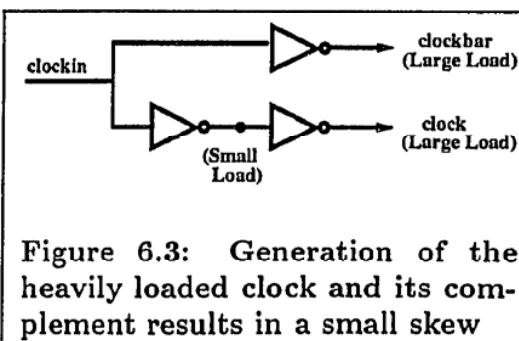
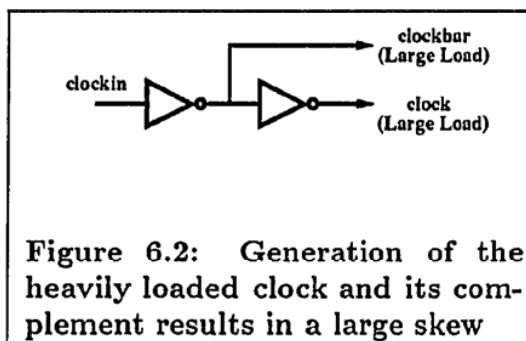


Figure 6.1: Plot of the completed Single Chip Implementation

## 6.2 Reducing Skews in Control Signals

The modules that constitute the chip, when described at the circuit level, require the logical complement of most of the control signals along with the original signals. Some signals, for instance the clock signals, have a heavy resistive and capacitive (RC) load on the signal lines, due to the large number of cells that require them. Hence the structures required to drive these signals require careful design to prevent large skews between the signals and their complements, which can cause incorrect operation. A circuit as shown in Figure 6.2, which consists of two inverters in tandem, can cause a large skew between the transitions of the signal and its complement. The large RC load results in a large switching time that affects the input of the second stage, which itself is heavily loaded, thus, resulting in a large skew. An alternate design as shown in Figure 6.3 avoids this situation by controlling the skew between the inputs of two inverters, which generate the signal and its complement. An additional inverter is required to invert one of these input signals. However, this inverter is connected to the input of only one inverter, resulting in a fast switching. Thus the skew in the inputs of the drivers is small. Assuming a large, but nearly equal, load on the signal and its complement, the final skews will be small. Static CMOS design, by definition, requires every structure to be associated with an identical complementary structure.



Though this nearly doubles the transistor count, it leads to a more symmetrical design that can be compact. It also leads to the desirable property of an almost identical RC load on any signal and its complement.

Circuits that need to drive a large load can be designed with wider transistors. This causes the transistors to switch faster since the resistance is reduced by a factor that is greater than the increase in the capacitance. This is true only up to a certain maximum size, after which the propagation delay increases again. These techniques were used to reduce the skews of numerous control signals.

### 6.3 Design of an Adder

To obtain a simple design with minimal area complexity, ripple carry adders were used in the CORDIC units. However, a poorly designed adder can result in a large propagation delay, affecting the overall clock rate. The adder used in the prototype chips was modified to reduce this propagation delay when used in CAPE.

In a ripple carry adder, the longest path occurs from the carry-in of the adder to the carry-out. Reducing the delay from one stage of the adder to the next can significantly improve the speed of operation of the adder. To reduce the propagation delay of the carry through a single stage, the minimum number of levels of logic should be used to generate the carry-out of that stage. Using wider transistors can further reduce this delay. Since PMOS transistors are twice as slow as NMOS, the logic should be designed to contain the minimal number of PMOS transistors in series. Reducing the load seen by the carry-out from a previous stage can further reduce propagation delay. This reduction can be effected by using pseudo-NMOS logic. The use of pseudo-NMOS logic increases power dissipation, but this is negligible compared to the power

dissipation of the entire chip. The longest delay in the adder used in CAPE was reduced by half by using these techniques.

## 6.4 Testing

PC-based high speed test equipment, Omnilab from Orion Instruments [28], was used in verifying operation of the chips. This tester allows testing up to a frequency of 17 MHz, which corresponds to a clock frequency of 4.25 MHz for the chip, due to the two-phase clocking strategy. All the six chips, which constitute the SVD processor, operate at the maximum test frequency.

Verifying correct functionality of the sub-units of the SVD processor was the primary goal of the testing process. The chips were tested with the same vectors, which were used for simulation. The testing exposed several errors in the design of the prototype. Table 6.1 lists the most significant of these errors. The errors are categorized as design errors or architecture problems. The design errors occurred due to the independent design of the various units. These were primarily errors in the interaction between different units, which could be verified only after fabrication. The architecture problems, on the other hand, required further research to determine a low-cost solution. The numerical accuracy problem encountered in Y-reduction belongs to the latter class of problems. The error observed in practice was larger than expected and required the detailed analysis described in Chapter 3. The analysis indicated that normalization could solve the accuracy problem.

A C-language library of routines was created to use the CORDIC module as a co-processor for an IBM PC. A parallel I/O card on an IBM PC-AT was used to communicate with the CORDIC processor. The design of the prototype requires initial matrix data to be loaded synchronously. After an initial start signal, a different

Design Errors	
Width of the Z-Datapath	Two bits are required before the binary point in order to represent all angles from $-\pi/2$ to $\pi/2$ . The representation of the angles stored in the ROM had to be changed accordingly.
Calculation of angles	<p>From equations 2.8, the sum and difference of the left and right angles is obtained. The computation of <math>\theta_l</math> and <math>\theta_r</math> can be executed in the following order:</p> <ol style="list-style-type: none"> <li>1. Compute <math>(\theta_l + \theta_r)</math> and <math>(\theta_l - \theta_r)</math></li> <li>2. Compute <math>(\theta_l + \theta_r)/2</math> and <math>(\theta_l - \theta_r)/2</math></li> <li>3. Compute <math>\theta_l</math> and <math>\theta_r</math></li> </ol> <p>In the prototype the shift was performed after <math>2\theta_l</math> and <math>2\theta_r</math> were computed from the sum and the difference, resulting in an overflow if the numbers are too large. This effectively reduced the range of all angles to <math>-\pi/4</math> to <math>\pi/4</math>. This problem was corrected in the single chip.</p>
Architectural Problems	
Convergence of Y-reduction	Negative values of $x_0$ prevent the Y-reduction iterations from converging. A different choice of $\delta_i$ s forces convergence.
Accuracy	A careful study was necessary to determine the number of significant bits that could be guaranteed by the processor. The error analysis is described in Chapter 3.
Computation of $\tan^{-1}(0/0)$	A careful examination of the SVD algorithm indicates that this problem can be corrected by replacing the angle with a zero angle. Zero comparators were used to detect this condition and correct it.

**Table 6.1: A list of some of the problems in the 6-chip prototype**

data element is expected on the bus on each successive clock cycle. This prohibits generation of the processor clock from the PC clock. The solution to the clock generation problem in the prototype was to generate the clock using the parallel ports. The program generates a clock for the processor by treating it as a control signal, requiring four separate writes to the parallel port for each clock cycle. This interface was helpful in performing an exhaustive test of the CORDIC processor. A program, which performs a simulation of the CORDIC module at the bit-level, was written to generate the expected results for the exhaustive tests. Discrepancies in the outputs were used to determine design errors.

Data loading in CAPE was modified to be asynchronous, solving the interface problem by allowing the array to operate at a rate independent of the PC clock. Asynchrony, however, introduces an additional problem. In an asynchronous system, a data transition at the same instant as a clock transition can result in *metastability*. In such a state the output voltages remain undefined for a prolonged period of time. One solution to the metastability problem is to design flip-flops that are resilient to this problem [22]. The same effect can be achieved externally. Typically, the TTL MSI flipflops in the F-series have a very small metastability window and hence decrease the probability of the occurrence of the problem. Using more of these chips in series can further reduce this probability. The latter approach was preferred in CAPE due to its simplicity.

Study of the prototype helped identify those factors that affect performance. Communication within the chip was identified as the main source of overhead. The extra clock cycles associated with this communication were eliminated in the single chip implementation by using an enhanced architecture. These enhancements were described in Chapter 4. Table 6.2 shows a breakdown of the various communication costs in a single computation cycle. A *computation cycle* in the table corresponds to the time



Total clock cycles between each wave of activity = 857		
Sub-task	Number of clock cycles	Percentage of the whole cycle
Computation	128	14.9%
Intra-chip to Inter-chip Communication	576	67.2%
Movement of data within the chip	105	12.2%
Systolic Communication	48	5.6%

**Table 6.2: Breakdown of clock cycles for the various sub-tasks in the 6-chip prototype**

Total clock cycles between each wave of activity = 260		
Sub-task	Number of clock cycles	Percentage of the whole cycle
Computation	128	49.2%
Intra-chip to Inter-chip Communication	27	10.4%
Movement of data within the chip	57	21.9%
Systolic Communication	48	18.4%

**Table 6.3: Breakdown of clock cycles for the various sub-tasks in CAPE**

required by three consecutive diagonals to complete computation. In the prototype, the principal communication cost was due to an unavoidable serial link connecting the intra-chip and inter-chip. This cost was reduced to a small fraction in the single chip design through the use of a parallel bus. Reducing inter-processor communication would require use of parallel ports to transmit data between processors. However, use of parallel ports in CAPE is not currently feasible due to pin limitations. The internal communication, however, was reduced to half through the use of a double bus architecture as described in Chapter 4.

## Chapter 7

### Conclusions

#### 7.1 Summary

This thesis presented the issues involved in the VLSI implementation of a systolic array to compute the SVD of a matrix, using CORDIC arithmetic techniques. Implementation of the CORDIC-SVD processor was carried out in two phases. Initially, a prototype of the processor, composed of 6 chips, was developed. This required the design of four custom MOSIS TinyChips. Testing of this prototype revealed several architectural problems. Elegant solutions to all these problems, which were presented in this thesis, were incorporated in CAPE, the single chip implementation.

The primary factor, which affected performance of the processor was the large overhead incurred in moving data between the modules within the chip. In CAPE, several changes were made to reduce this communication overhead. Since the CORDIC modules in the 6-chip prototype used dedicated registers to operate, there was a need to move the data to these registers for all arithmetic operations. In CAPE, these registers were made part of the register set, eliminating the intermediate communication step when data had to be moved from a general register to these dedicated registers. Another architectural improvement was a two-bus architecture, which was developed to reduce the communication by half. Additional modifications helped to further reduce the communication.

Another important problem encountered in the prototype was numerical accuracy of the fixed-point CORDIC modules. Previous analyses of CORDIC neglected the interaction between the  $x$ ,  $y$  iterations and the  $z$  iterations. Consequently, the effects of unnormalized  $x_0$  and  $y_0$  on the results of Y-reduction were not noticed. This thesis presented a methodology to perform a detailed analysis of numerical error in any mode of CORDIC. Examples which determine pessimistic bounds for Z-reduction and Y-reduction were also presented. Although a bound on the error was found for Z-reduction, no such bound could be found for Y-reduction. The analysis, however, showed that a normalization of the initial values for Y-reduction was required to force such a bound. A normalization scheme with low time penalty requires  $O(n^2)$  area to implement using conventional techniques. This area was reduced to  $O(n^{1.5})$  by performing the overall shift as small shifts, which are implemented as part of the CORDIC iterations. A detailed analysis of this novel method, which reduces the area complexity with no associated time penalty, was presented in this thesis.

Several improvements and new features were developed, which can be incorporated with small changes to the current processor design. A systolic starting scheme was developed to eliminate a global start signal. This scheme allows greater scalability and presents a method that can be used to achieve the same effect in other systolic arrays. An architecture, which results in a speedup of 100% over the current design was also developed. This scheme provides a way to achieve the performance gain obtained through broadcast, without the global connections that prevent scalability. The performance gain is achieved by eliminating idle time in the array. An alternate scheme to utilize the idle time in the array was also developed. This scheme pipelines the computation of the  $U$  and  $V$  matrices along with the singular values, to compute all the three matrices in the time it currently takes to compute the SVD alone.

## 7.2 Future Work

The error analysis of fixed-point CORDIC indicates that floating-point CORDIC would provide better numerical accuracy, since the normalization is already a part of floating-point arithmetic. Future implementations of CORDIC using floating-point units can achieve better numerical accuracy.

Most commercial chips implement a Test Access Port (TAP) to provide observability and controllability during manufacturing testing. A similar TAP is often used on printed circuit boards. IEEE standard 1149.1 defines the TAP protocol. It appears that an approach similar to TAP can be used to implement run time fault detection. Adherence to a standard would allow circuits with different types of components to interact and cooperate to detect faults in the system. Designing a complete systolic array that incorporates fault-tolerance can provide some insight to implementing fault-tolerance in general-purpose systems.

Since the SVD array requires  $O(p^2)$  processors, large arrays would benefit from higher levels of integration. Wafer-Scale Integration can provide the desired degree of integration. A problem that requires further study is the design of the array as an interconnection of modules, where each module is composed of several processors. The number of pins on a wafer package limit the number of interconnections between these modules.

Currently, WSI technologies require special post-processing steps, like laser repair, to disconnect the faulty processors on the wafer. Fault tolerance techniques developed for arrays of this nature can be used in WSI to eliminate the post-processing.

## Bibliography

- [1] H. M. Ahmed. *Signal Processing Algorithms and Architectures*. PhD thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, CA, June 1982.
- [2] H. C. Andrews and C. L. Patterson. Singular Value Decompositions and Digital Image Processing. *IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-24(1):26–53, February 1976.
- [3] J. Barnaby, B. On, and P. Szeto. A Systolic I/O Controller for the CORDIC-SVD Processor. Technical Report ELEC 560, Rice University, December 1989.
- [4] R. P. Brent and F. T. Luk. The Solution of Singular Value Problems Using Systolic Arrays. *Proc. SPIE Real-Time Signal Processing*, 495(VII):7–12, August 1984.
- [5] R. P. Brent, F. T. Luk, and C. F. Van Loan. Computation of the Singular Value Decomposition Using Mesh-Connected Processors. *Journal of VLSI and Computer Systems*, 1(3):242–270, 1985.
- [6] J. R. Cavallaro. *VLSI CORDIC Processor Architectures for the Singular Value Decomposition*. PhD thesis, School of Electrical Engineering, Cornell Univ., Ithaca, NY, August 1988.
- [7] J. R. Cavallaro, M. P. Keleher, R. H. Price, and G. S. Thomas. VLSI Implementation of a CORDIC SVD Processor. *Proc. 8th Biennial Univer-*

- sity/Government/Industry Microelectronics Symposium*, pages 256–260, June 1989.
- [8] J. R. Cavallaro and F. T. Luk. CORDIC Arithmetic for an SVD Processor. *Journal of Parallel and Distributed Computing*, 5(3):271–290, June 1988.
- [9] J. R. Cavallaro, C. D. Near, and M. Ü. Uyar. Fault-Tolerant VLSI Processor Array for the SVD. *IEEE Int. Conf. on Computer Design*, pages 176–180, October 1989.
- [10] D. Daggett. Decimal-Binary Conversions in CORDIC. *IRE Transactions on Electronic Computers*, EC-8(3):335–339, Sept. 1959.
- [11] J. M. Delosme. VLSI Implementation of Rotations in Pseudo-Euclidean Spaces. *IEEE Int. Conf. Acoustics, Speech and Signal Processing*, 2:927–930, April 1983.
- [12] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. Chapter 11: The Singular Value Decomposition. In *Linpac Users' Guide*, pages 11.1–11.23. SIAM, Philadelphia, PA, 1979.
- [13] R. A. Duryea. *Finite Precision Arithmetic in Singular Value Decomposition Processors*. PhD thesis, School of Electrical Engineering, Cornell Univ., Ithaca, NY, August 1987.
- [14] G. E. Forsythe and P. Henrici. The Cyclic Jacobi Method for Computing the Principal Values of a Complex Matrix. *Transactions of the American Mathematical Society*, 94(1):1–23, January 1960.
- [15] G. H. Golub and C. F. Van Loan. *Matrix Computations, Second Edition*. Johns Hopkins Univ. Press, Baltimore, MD, 1989.

- [16] G. L. Haviland and A. A. Tuszynski. A CORDIC Arithmetic Processor Chip. *IEEE Trans. Computers*, C-29(2):68–79, Feb. 1980.
- [17] N. D. Hemkumar, K. Kota, and J. R. Cavallaro. CAPE-VLSI Implementation of a Systolic Processor Array: Architecture, Design and Testing. In *Ninth Biennial University/Government/Industry Microelectronics Symposium*, June 1991 (to appear).
- [18] S. L. Johnsson and V. Krishnaswamy. Floating-Point CORDIC. Technical Report YALEU/DCS/RR-473, Dept. of Computer Science, Yale Univ., New Haven, CT, April 1986.
- [19] K. Karplus. Exclusion Constraints, A New Application of Graph Algorithms To VLSI Design. *Advanced Research in VLSI, Proc. 4th MIT Conference*, pages 123–139, April 1986.
- [20] K. Karplus. A Formal Model for MOS Clocking Disciplines. Technical Report TR 84-632, Dept. of Computer Science, Cornell Univ., Ithaca, NY, August 1984.
- [21] H. T. Kung. Why Systolic Architectures? *IEEE Computer*, 15(1):37–46, January 1982.
- [22] Lee-Sup Kim and Robert W. Dutton. Metastability of CMOS Latch/Flip-Flop. *IEEE Journal of Solid-State Circuits*, 25(4):942–951, August 1990.
- [23] F. T. Luk. Architectures for Computing Eigenvalues and SVDs. *Proc. SPIE Highly Parallel Signal Processing Architectures*, 614.
- [24] F. T. Luk. A Parallel Method for Computing the Generalized Singular Value Decomposition. *Journal of Parallel and Distributed Computing*, 2:250–260, 1985.



- [25] F. T. Luk. A Triangular Processor Array for Computing Singular Values. *Journal of Linear Algebra and Its Applications*, 77:259–273, 1986.
- [26] F. T. Luk. Computing the Singular Value Decomposition on the ILLIAC IV. *ACM Transactions on Mathematical Software*, 6(4):524–539, December 1980.
- [27] C. A. Mead and L. A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, MA, 1980.
- [28] Orion Instruments. *The OmniLab<sup>TM</sup> User's Manual*. Menlo Park, CA, June 1990.
- [29] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. Magic, a VLSI Layout System. *ACM/IEEE 21st Design Automation Conference*, pages 152–159, June 1984.
- [30] L. H. Sibul. Application of Singular Value Decomposition to Adaptive Beamforming. *IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, 2:33.11.1–33.11.4, March 1984.
- [31] R. Spickelmier. *Octtools Distribution, Release 4.0*. ERL/UC Berkeley Memorandum, August 1990.
- [32] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, July 1989.
- [33] P. Szeto. Design of a Z-controller for the CORDIC-SVD Processor. Technical Report ELEC 590, Rice University, May 1990.
- [34] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Rockville, MD, 1984.

- [35] J. Volder. The CORDIC Trigonometric Computing Technique. *IRE Trans. Electronic Computers*, EC-8(3):330–334, Sept. 1959.
- [36] J. S. Walther. A Unified Algorithm for Elementary Functions. *AFIPS Spring Joint Computer Conf.*, pages 379–385, 1971.
- [37] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley Publishing Company, Reading, MA, 1985.
- [38] B. Yang and J. F. Böhme. Reducing the computations of the SVD array given by Brent and Luk. *SPIE Advanced Algorithms and Architectures for Signal Processing*, 1152:92–102, 1989.