SYSTOLIC ARRAY PROCESSOR FOR EIGENVALUES AND EIGENVECTORS

by

Mohammad Saeed Tariq Chaudhary
B.E. Electronics Eng., NED University of Engineering, Pakistan, 1984

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of
Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

[redacted]
Dr. F. El-Guibaly, Supervisor, Dept. of Elect. and Comp. Eng.

[redacted]
Dr. W. D. Little, Departmental Member, Dept. of Elect. and Comp. Eng.

[redacted]
Dr. H. H. Kwok, Departmental Member, Dept. of Elect. and Comp. Eng.

[redacted]
Dr. M. Serra, Outside Member, Dept. of Computer Science

[redacted]
Dr. G. F. McLean, External Examiner, Dept. of Mech. Eng.

[redacted]
Dr. R. Vahldieck, Additional Member, Dept. of Elect. and Comp. Eng.

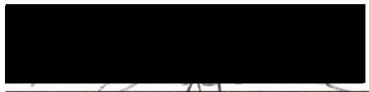
Supervisor: Dr. F. El-Guibaly

ABSTRACT

The CORDIC algorithm provides a rich set of operations necessary for digital signal processing. The algorithm is analyzed for the fixed- and floating-point data implementations. The analysis includes: required number of iterations, lookup table size and noise analysis of CORDIC. Based on the above considerations, a general purpose, floating-point CORDIC processing element (PE) is presented. Simulation results and hardware details of the PE are also presented. The PE is to be incorporated in a triangular systolic array. The triangular systolic array design for the spectral decomposition of a matrix based on the QR algorithm is proposed here. The proposed array is capable of evaluating the eigenvectors and the eigenvalues of a given matrix.

Examiners:

Dr. F. El-Guibaly, Supervisor, Dept. of Elect. and Comp. Eng.



Dr. W. D. Little, Departmental Member, Dept. of Elect. and Comp. Eng.

Dr. H. H. Kwok, Departmental Member, Dept. of Elect. and Comp. Eng.

Dr. M. Serra, Outside Member, Dept. of Computer Science

Dr. G. F. McLean, External Examiner, Dept. of Mech. Eng.

Dr. R. Vanldieck, Additional Member, Dept. of Elect. and Comp. Eng.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank most sincerely my supervisor, Dr. Fayez El-Guibaly for his guidance during the course of this project and his helpful advice for the preparation of this manuscript. Financial assistance received from Dr. Fayez El-Guibaly was greatly appreciated.

This work is dedicated to my teachers.

Chapter 1

Introduction

The growing need for real time digital signal processing (DSP) has resulted in constant push for the development of faster computing structures. DSP typically requires operating on large volumes of data in a minimum amount of time. The use of a general purpose computer is not optimal for real time DSP processing. Parallelism is a viable approach to satisfy DSP requirements. Systolic or wave-front array processing is a form of parallel computing with massive pipelining and reduced communication requirements.

The CORDIC algorithm (COrdinate Rotation DIgital Computer) provides the means of performing several elementary operations essential for DSP applications [1, 2, 3]. Volder [1] developed the CORDIC algorithm, which was later generalized elegantly by Walther [2]. Walther shows that the implementation of a wide range of transcendental functions can be fully represented by a single set of iterative equations. An asperity of the CORDIC algorithm is the scale factor K. To obtain the correct result, one must divide the output by K. Several solutions to the scale factor problem were proposed in [2, 3, 4].

Naseem and Fisher [5] transformed the original sequential CORDIC algorithm

into a parallel one. The modified CORDIC algorithm is based on a large ROM for lookup. However, for a given set of inputs, a basic iteration has to be performed to extract the rotation directions. The authors were not able to convert this iterative step into a parallel one.

The CORDIC algorithm has been proposed for several signal processing applications. Curtis [6] uses CORDIC to perform the laser trimming of hybrid microcircuit resistors. A software implementation of the algorithm was too slow, whereas the hardware implementation provides a speed advantage. Lee [7] describes a pipelined CORDIC processor for the computation of inverse kinematic position solution for robot arm motion. Despain [8] describes the pipelined processor array for implementing the fast Fourier transform. Ahmed describes the use of CORDIC algorithm in speech synthesis, adaptive equalization and digital filtering [9, 10, 11]. A bit-serial implementation of the CORDIC algorithm was proposed by Harber [12]. On the other hand Li and Jayakumar [13] presented a comparison between the serial and parallel implementation.

Haviland and Tuszynski designed a CORDIC arithmetic processing chip [4]. This chip is based on a parallel architecture. A single byte format is used in a fixed-point data format. Vaudin reported a processing element using the CORDIC algorithm based on a parallel hardware architecture [14].

A number of hardware implementations for the CORDIC algorithm were proposed or actually implemented. Ahmed [3] proposed a parallel implementation of three adders/subtractors and two shifters. The basic CORDIC butterfly operation is therefore performed simultaneously. Ahmed also suggested using an ALU that is composed of only one adder/subtractor and a shifter. He calls this design the parallel-serial implementation. These basic processing units are connected

in a pipelined fashion to perform the iterative CORDIC iterations. A pipelined CORDIC processor was proposed in [7]. In this application the closed-form joint-angle solution shows a limited amount of parallelism with a large amount of sequentialism in the flow of computation. This leads to a cost effective pipelined CORDIC architecture for computing the inverse kinematic position. In other words the architecture of the processor is application dependent. Other examples of pipelined CORDIC architectures were also reported in [15] and [16].

For the laser trimming application of hybrid microcircuit resistors [6]. The resistors are formed by the deposition of patterns of resistive inks onto a circuit substrate. Each resistor is then trimmed to the correct value by a computer controlled laser. The laser positioning system allows x, y coordinate translation, although any rotation adjustment must be done with software routines. This approach to rotation requires an excessive amount of computation time. A hardware approach was used to solve this problem. The author describes the design and implementation of a custom processor, which is built using off-the-shelf components, that provides rotational correction in laser trimming.

The basic issues for the implementation of the CORDIC algorithm are: table look up requirements, scale factor, number of iterations and the processor architecture. The goal of this thesis is to investigate each of the above mentioned factors, identify a suitable architecture and the implementation of the CORDIC algorithm.

Chapter 2 of this thesis presents the background information on the CORDIC algorithm, number of iterations required to perform the fixed-point and floating-point CORDIC operations and table lookup requirements. The result of the simulation of CORDIC hardware is also presented.

Chapter 3 deals with the CORDIC processing element (PE) design, architecture

and system hardware implementation. Design specifications were determined at system level as well as at the processing node level. The actual design of the processing node along with the pin description and operation is also presented in this chapter

Chapter 4 outlines the QR algorithm, mapping of the algorithm on to a systolic architecture, and its implementation by using the processing node designed in Chapter 3. This processing node form the bases of constructing such a systolic array. The technique for obtaining the eigenvalues and eigenvectors was also described.

Chapter 5 summerizes the major contribution of this thesis.

Chapter 2

The CORDIC Algorithm

2.1 Introduction

The acronym CORDIC stands for "Coordinate Rotation Digital Computer". Its basic computing modes are rotation and vectoring. The rotation mode computes the new coordinates of a vector after rotation of the axes by a given angle. The vectoring mode computes the angle and magnitude of the vector given its coordinates.

In 1959 Volder [1] developed the CORDIC algorithm, which was later unified by Walther [2]. These algorithms represent an efficient way to compute a variety of functions related to coordinate transformation such as trigonometric functions, vector rotation, multiplications and square root calculation. Given the triplet $(x,\ y,\ z)$, where x, y and z are real numbers, let x and y correspond to the coordinates of the end point of a vector $\vec{r}$ as shown in Fig. 2.1. The number z will correspond to a clockwise rotation of the vector or equivalently moving the axes in an anticlockwise direction. The magnitude of the vector (R) and angle relative

to the x-axis (θ) are defined, respectively, by:

$$R_c \triangleq \sqrt{x^2 + y^2} \tag{2.1}$$

$$\theta_c \triangleq \tan^{-1} y/x \tag{2.2}$$

where the subscript c represents the circular CORDIC case. CORDIC allows us to extract the value of R by rotating $\vec{r}$ such that it coincides with the x-axes. Referring to Fig. 2.1, this is equal to a clockwise rotation by an angle θ_c , the new values of x and y will be

$$x' = R_c \tag{2.3}$$

$$y' = 0 \tag{2.4}$$

CORDIC also allows us to rotate our vector by an arbitrary angle z. If we rotate $\vec{r}$ clockwise by angle z, we get

$$x'' = x\cos z + y\sin z \tag{2.5}$$

$$y'' = y\cos z - x\sin z \tag{2.6}$$

For this process a better representation, in the form of a block diagram is shown in Fig. 2.2. Where x, y, z are the three inputs, and the block itself is a CORDIC processor.

Referring to Fig. 2.3, the linear CORDIC rotation forces the end-point of the vector $\vec{r}$ to move along a vertical line. The magnitude and angle of $\vec{r}$ are defined by:

$$R_l \triangleq \sqrt{x^2} = x \tag{2.7}$$

$$\theta_l \triangleq \frac{y}{x} \tag{2.8}$$

If the vector $\vec{r}$ is rotated clockwise through an angle θ_l to coincide with the x-axes, the new value of x and y will be

$$x' = x \tag{2.9}$$

$$y' = 0 \tag{2.10}$$

If we rotate $\vec{r}$ clockwise by an angle z, we get

$$x'' = x \tag{2.11}$$

while from (2.8), we can write

$$\theta_l - z = \frac{y''}{x''} = \frac{y''}{x} \tag{2.12}$$

$$y'' = y - xz \tag{2.13}$$

Referring to Fig. 2.4, the hyperbolic CORDIC forces the end-point of the vector $\vec{r}$ to move along a hyperbola. The magnitude and angle of $\vec{r}$ are defined by:

$$R_h \triangleq \sqrt{x^2 - y^2} \tag{2.14}$$

$$\theta_h \triangleq \tanh^{-1} \frac{y}{x} \tag{2.15}$$

If $\vec{r}$ is rotated clockwise, through an angle θ_h to coincide with the x-axes then the value of x and y will be

$$x' = R_h \tag{2.16}$$

$$y' = 0 \tag{2.17}$$

If we rotate $\vec{r}$ clockwise by an angle z, we get

$$x'' = x\cosh z - y\sinh z \tag{2.18}$$

$$y'' = y\cosh z - x\sinh z \tag{2.19}$$

Walther [2] showed that the algorithms for these functions could be described by a single set of equations parameterized by a quantity m. For a given vector $\vec{r}$, as shown in Fig. 2.5, the magnitude R and angle θ are defined as

$$R = \sqrt{x^2 + my^2} \tag{2.20}$$

$$\theta = \frac{\tan^{-1}\sqrt{m}y/x}{\sqrt{m}} \tag{2.21}$$

where $m = 1$ for the circular case, $m = 0$ for the linear case and $m = -1$ for the hyperbolic case.

In all three cases, if z is a variable by which one can show the accumulated rotation of the vector then

$$z_{i+1} = z_i - \mu_i\theta_i \tag{2.22}$$

where

$$\mu_i = \begin{cases} +1 & \text{for counter-clockwise rotation of the axes} \\ -1 & \text{for clockwise rotation of the axes} \end{cases}$$

and

$$\theta_i \geq 0$$

If the initial vector $\vec{r} = (x_0, y_0)$ is rotated through a sequence of angles θ_i until $\vec{r}_n = (x_n, 0)$ then z_n will equal to the net rotation, provided it was initially equal

to zero. Therefore z is a useful quantity since after n rotations it accumulates the net rotation.

$$z_n = z_0 - \sum_{i=0}^{n-1} \mu_i \alpha_i \tag{2.23}$$

If the rotations of $\vec{r}$ are made in such a way, which forces y_n to zero i.e. $y_n \rightarrow 0$ then x_n and z_n result in one of the elementary functions. On the other hand more elementary functions are obtained when $z_n \rightarrow 0$. These relations are summarized in Fig. 2.6 for $m = 1$, $m = 0$ and $m = -1$, where each box depicts a function with three inputs on the left, which are the initial values of x, y, and z, while the outputs correspond to the final values of x_n, y_n, and z_n are shown on the right side. The quantity k is the scale factor as given in equation (2.40).

2.1.1 Derivation of Circular CORDIC Equations

For the circular case one can derive the initial equations, similar arguments are true for the linear and hyperbolic cases. Assuming vector $\vec{r}$ is to be rotated clockwise by a positive angle θ. The resulting components x' and y' of the rotated vector are given by the relations

$$x' = x\cos\theta + y\sin\theta \tag{2.24}$$

$$y' = y\cos\theta - x\sin\theta \tag{2.25}$$

In order to realize the above equations, four multiplications and two additions are required. The above equations can be simplified by dividing both sides by $\cos\theta$ to yield

$$x'' = \frac{x'}{\cos\theta} = x + y\tan\theta \tag{2.26}$$

$$y'' = \frac{y'}{\cos\theta} = y - x\tan\theta \tag{2.27}$$

For convenience the above equations can be written as

$$x'' = x_1 = x + y\tan\theta \tag{2.28}$$

$$y'' = y_1 = y - x\tan\theta \tag{2.29}$$

the values of x' and y' can be obtained from x'' and y'' by multiplying by $\cos\theta$.

The angle θ can be decomposed into an arbitrary number of angles such that

$$\theta = \mu_0\theta_0 + \mu_1\theta_1 + \cdots + \mu_{n-1}\theta_{n-1} \tag{2.30}$$

$$= \sum_{i=0}^{n-1} \mu_i\theta_i \tag{2.31}$$

where μ_i is equal to ± 1 and there are limits on the maximum value of θ [17].

$$|\theta_{max}| = \sum_{i=0}^{n-1} \theta_i \tag{2.32}$$

The accuracy or the resolution of rotations is determined by the smallest angle in the set θ_i.

Now the vector components at step i are given by

$$x_{i+1} = x_i + \mu_i y_i \tan\theta_i \tag{2.33}$$

$$y_{i+1} = y_i - \mu_i x_i \tan\theta_i \tag{2.34}$$

$$z_{i+1} = z_i - \mu_i\theta_i \tag{2.35}$$

Each θ_i is chosen as

$$\theta_i = \tan^{-1} 2^{-i} \tag{2.36}$$

where $i = 0, 1, 2, \cdots, n-1$

We can express the x_{i+1} and y_{i+1} components such that

$$x_{i+1} = x_i + \mu_i y_i 2^{-i} \tag{2.37}$$

$$y_{i+1} = y_i - \mu_i x_i 2^{-i} \tag{2.38}$$

$$z_{i+1} = z_i - \mu_i \tan^{-1}(2^{-i}) \tag{2.39}$$

The values of x_{i+1} and y_{i+1} may be obtained by two shift and add operations. The expressions for x_{i+1} and y_{i+1} are not perfect rotations, since the magnitude of the vector increases at each step by the value $\frac{1}{\cos\theta_i}$. The two choices of direction produces the same change in magnitude. This increase in magnitude may be considered as a constant, which is given as

$$K = \prod_{i=0}^{n-1} \frac{1}{\cos\theta_i} = \prod_{i=0}^{n-1} \sqrt{1 + m2^{-2i}} \tag{2.40}$$

Since $\sum_{i=0}^{\infty} \theta_i = \sum_{i=0}^{\infty} \tan^{-1} 2^{-i} = 100°$, a rotation of any angle of up to $\pm 100°$ may be achieved by adding or subtracting each θ_i in turn. A detailed discussion on the convergence properties and the range of convergence of the rotations is given in [3], [4]. To cover the whole range for θ an alternative method is considered in [18].

2.2 Number of Iterations

An important parameter in the CORDIC algorithm is the number of iterations required to achieve a final result. The number of iterations is expected to depend on the number of bits used to represent the data and whether fixed- or floating-point formats are employed [17]. In what follows we derive the number of iterations for the fixed- and floating-point cases.

2.2.1 Fixed-point Case

The most common fixed-point data formats are the integer and fractional formats. In integer format the radix point is fixed to the extreme right of the least significant bit, while in the fractional format it is located to the left of the most significant bit. For the fixed-point iterations, we assumed sign and magnitude notation with the radix point at the extreme left position, excluding the sign bit, as shown in Fig. 2.7(a). The number of bits in a word is assumed $n + 1$. The x variable at iteration i is given by:

$$x_{i+1} = x_i + \mu_i y_i 2^{-i} \tag{2.41}$$

The second term on the right hand side of the above equation is y shifted i times to the right. At iteration n, y is shifted n positions to the right. The value of $y2^{-n}$ would be zero and (2.41) becomes

$$x_{n+1} = x_n + 0 \tag{2.42}$$

Therefore, it is concluded that after n iterations the value of x will not change from its previous value. Further iterations will not affect the value of x. A similar argument applies to the y and z variables. Therefore, the total number of iterations is equal to n.

2.2.2 Floating-point Case

The floating-point data format is shown Fig. 2.7(b). For floating-point data, it is not immediately obvious how the number of iterations is related to the number of bits in the mantissa or exponent parts. The difficulties in analyzing this case stem from two reasons:

1. Addition or subtraction of two inputs makes the exponent of the result equal to the exponent of the larger input number.

2. At each iteration the CORDIC butterfly operation is performed. This operation will make the output exponent value for x and y approximately equal. This is true even if the input exponent values of x and y differ greatly.

There are three possibilities for the relative values of x and y. The value of x could be greater than, less than or equal to the value of y. Assume that in floating-point data format the mantissa part is equal to $m+1$ bits including the sign bit and the exponent part is equal to k bits. The value of the exponent parts for the x and y inputs are denoted by e_x and e_y, respectively. The value of e_x could be greater than, less than or equal to the value of e_y.

Figures 2.8 and 2.9 are graphical representations of the exponent magnitudes of the data at iteration i. The figures show only the relative magnitudes of the exponents, e, of the input data and output data. The horizontal axes of each figure represent the iteration number i and $i+1$. The vertical axes represent the value of the exponent of the input and output variables. We consider the basic CORDIC operation at iteration i when : (a) $i \leq m$ and (b) $i > m >$.

Case $\mathbf{i \leq m}$

This case deals with the situation when the amount of shift at iteration i is less than m. The amount of changes in the output value in relation to the input value is graphically shown in Fig. 2.8. Notice that the CORDIC butterfly operation starts by first performing a shift (which is exponent subtraction) followed by addition or subtraction of the mantissas. At the start of iteration i the difference between the two exponents at the input is equal to d_{in}. Figure 2.8(a) shows the situation when $d_{in} < m$. Since x is greater than y, after performing the CORDIC butterfly

operation, the value of e_x is not affected. The value of e_y is increased such that d_{out} lies between the limits set in (2.43).

$$min\{i, d_{in}\} \leq d_{out} \leq max\{i, d_{in}\} \quad ; d_{in} < m \tag{2.43}$$

Figure 2.8(b) shows the situation when the difference between two exponents d_{in} is more than m. Again the number having the larger exponent, e_x, emerges without much changes whereas the input with the smaller exponent, e_y, is changed to a greater extent and is given by (2.44).

$$d_{out} = i \quad ; d_{in} \geq m \tag{2.44}$$

From the above discussion it is concluded that the input number having the larger exponent emerge without much change. The input with the smaller exponent is changed to a greater extent. The amount of change in the output value of the exponent depends on i and the exponent difference d_{in}.

Case i > m

This case deals with the situation when the amount of shift at iteration i is greater than m. The amount of changes in the output value in relation to the input value is graphically shown in Fig. 2.9. Similar to the previous figures these figures shows, on the vertical axes, only the relative magnitudes of the exponents of the input data and output data and on horizontal axes the iterations. Figure 2.9(a) shows the case when $d_{in} < m$. The number with the greater exponent e_x will not change. The number with the lesser exponent e_y is also unchanged.

Figure 2.9(b) shows the case when $d_{in} > m$. The number with the greater exponent e_x is not affected by the CORDIC butterfly operation. The number with the smaller exponent e_y is increased such that the output value d_{out} lies between the limits in (2.43).

It is concluded that the input number having the larger exponent emerge without much change. The input with the smaller exponent is either unchanged or it changes very little as compare to the larger exponent value. The amount of change in the value of d_{out} depends upon the iteration number i and the difference between the two input exponents value d_{in}.

From the above discussion it is concluded that: When $i > m$ (case Fig. 2.9(b)), a CORDIC butterfly operation will not affect the larger of the input data, the smaller of the input data will become insignificant irrespective of its input value. Therefor, having $i > m$ does not affect the output result.

In conclusion, the number of CORDIC iterations for the floating-point case should be equal to the number of bits in the mantissa m. This result is similar to the fixed-point case. When the value of i is greater or equal to m, the value of the inputs having the bigger exponent, and z, will stop changing. Further iterations will have no effect, where as the value of the x and y inputs having the smaller exponent will be insignificant.

2.3 Table Lookup Requirements

The CORDIC algorithm requires a lookup table for the values of $\tan^{-1} 2^{-i}$ where $0 \leq i < n$ [17]. It is noticed that when θ is small then

$$\tan^{-1} \theta \simeq \theta - \frac{\theta^3}{3} \qquad |\theta| \ll 1 \tag{2.45}$$

thus we have

$$\tan^{-1} 2^{-i} \simeq 2^{-i} \tag{2.46}$$

with an error equal to $\frac{2^{-3i}}{3}$.

Therefore, it is possible not to store all the values of $\tan^{-1} 2^{-i}$ after a certain value of i.

For fixed-point arithmetic the size of the table k is decided by the inequality

$$\frac{2^{-3k}}{3} \leq 2^{-b} \tag{2.47}$$

$$2^{-3k} < 2^{-b} \tag{2.48}$$

$$3k > b \tag{2.49}$$

$$k > \frac{b}{3} \tag{2.50}$$

where b is the number of bits for the given format. Thus for 32-bits fixed-point data format 11 entries are sufficient.

For the floating-point numbers, the table size k is decisive from the amount of error at iteration k. At the kth iteration, the smallest angle that could be stored is $\theta_k = \tan^{-1} 2^{-k}$. Thus the error at this stage could be given by

$$\frac{1}{3}2^{-3k} \leq 2^{-b}(\tan^{-1} 2^{-k}) \tag{2.51}$$

$$\frac{1}{3}2^{-3k} < 2^{(-b+k)} \tag{2.52}$$

$$3k > b + k \tag{2.53}$$

$$k > \frac{b}{2} \tag{2.54}$$

where b is the number of bits for the given format. Thus for 32-bits floating-point data format when the mantissa is 24-bit wide, about 12 entries are sufficient.

2.4 Simulation of CORDIC algorithm

To simulate the CORDIC algorithm operation, only adder/subtractor shift register, accumulator and mux are required. To simulate each of the above mentioned hardware blocks, a software program was developed. The CORDIC hardware is simulated using 32-bit fixed-point format. The results are then compared with those obtained from simulation by using double precision floating-point format on the SUN work station. The difference between the two simulations results (relative error) is shown in Fig. 2.10. It is observed from the figure that the relative error is oscillating in nature and grows very fast as the iteration number tends to fixed-point data format bit size.

Similar simulation were run for 32-bit floating-point format. The result of this simulations were compared with the computer precision simulation results. The resultant relative error versus the iterations is shown in Fig. 2.11. It is observed that the relative error is negligible in this case and independent of the number of iterations.

2.5 Noise performance of CORDIC

In this section we consider only the noise performance of the CORDIC algorithm for fixed-point data. For fixed-point data, noise is introduced during the right-shift operation due to truncation. No noise is generated during addition or subtraction operations.

Assuming the binary point to lie at the extreme right for our data, then at

iteration i, the amount of noise introduced is given by

$$\epsilon_i = 1 - 2^{-i} \tag{2.55}$$

At iteration i, we can write

$$x_{i+1} = x_i + \mu_i 2^{-i} y_i \tag{2.56}$$

$$y_{i+1} = y_i - \mu_i 2^{-i} x_i \tag{2.57}$$

When noise is present, we assume our input data at iteration i to be given in the form

$$x_i + \epsilon_i^x \tag{2.58}$$

$$y_i + \epsilon_i^y \tag{2.59}$$

where ϵ_i^x is the total error in the variable x at iteration i and ϵ_i^y is similarly defined. Substituting the above equations in (2.56) and (2.57) we get

$$x_{i+1} + \epsilon_{i+1}^x = x_i + \epsilon_i^x + \epsilon_i + \mu_i 2^{-i} \left(y_i + \epsilon_i^y\right) \tag{2.60}$$

$$y_{i+1} + \epsilon_{i+1}^y = y_i + \epsilon_i^y + \epsilon_i - \mu_i 2^{-i} \left(x_i + \epsilon_i^x\right) \tag{2.61}$$

We are now able to write an iteration for the amount of noise introduced at each iteration as

$$\epsilon_{i+1}^x = \epsilon_i^x + \epsilon_i + \mu_i 2^{-i} \epsilon_i^y \tag{2.62}$$

$$\epsilon_{i+1}^y = \epsilon_i^y + \epsilon_i - \mu_i 2^{-i} \epsilon_i^x \tag{2.63}$$

where the values of ϵ_0^x and ϵ_0^y are equal to zero.

In matrix form the above equations become

$$\begin{bmatrix} \epsilon_{i+1}^{x} \\ \epsilon_{i+1}^{y} \end{bmatrix} = \begin{bmatrix} 1 & \mu_i 2^{-i} \\ -\mu_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} \epsilon_{i}^{x} \\ \epsilon_{i}^{y} \end{bmatrix} + \begin{bmatrix} 1 - 2^{-i} \\ 1 - 2^{-i} \end{bmatrix} \tag{2.64}$$

A software simulation program was written to simulate the above noise equations. For the purposes of simulation, the value of μ_i at each iteration was taken equal to $+1$. The result of simulation of fixed point format is shown in Fig. 2.12. The x axis represents the number of iterations and the y-axis represents noise. The 'noise' is equal to the 'total error' normalized by the maximum 'truncation error' (which is unity in this analysis). Maple software package was used for simulation purpose [19]. As seen from the figure, the noise magnitude is approximately proportional to the number of iterations.

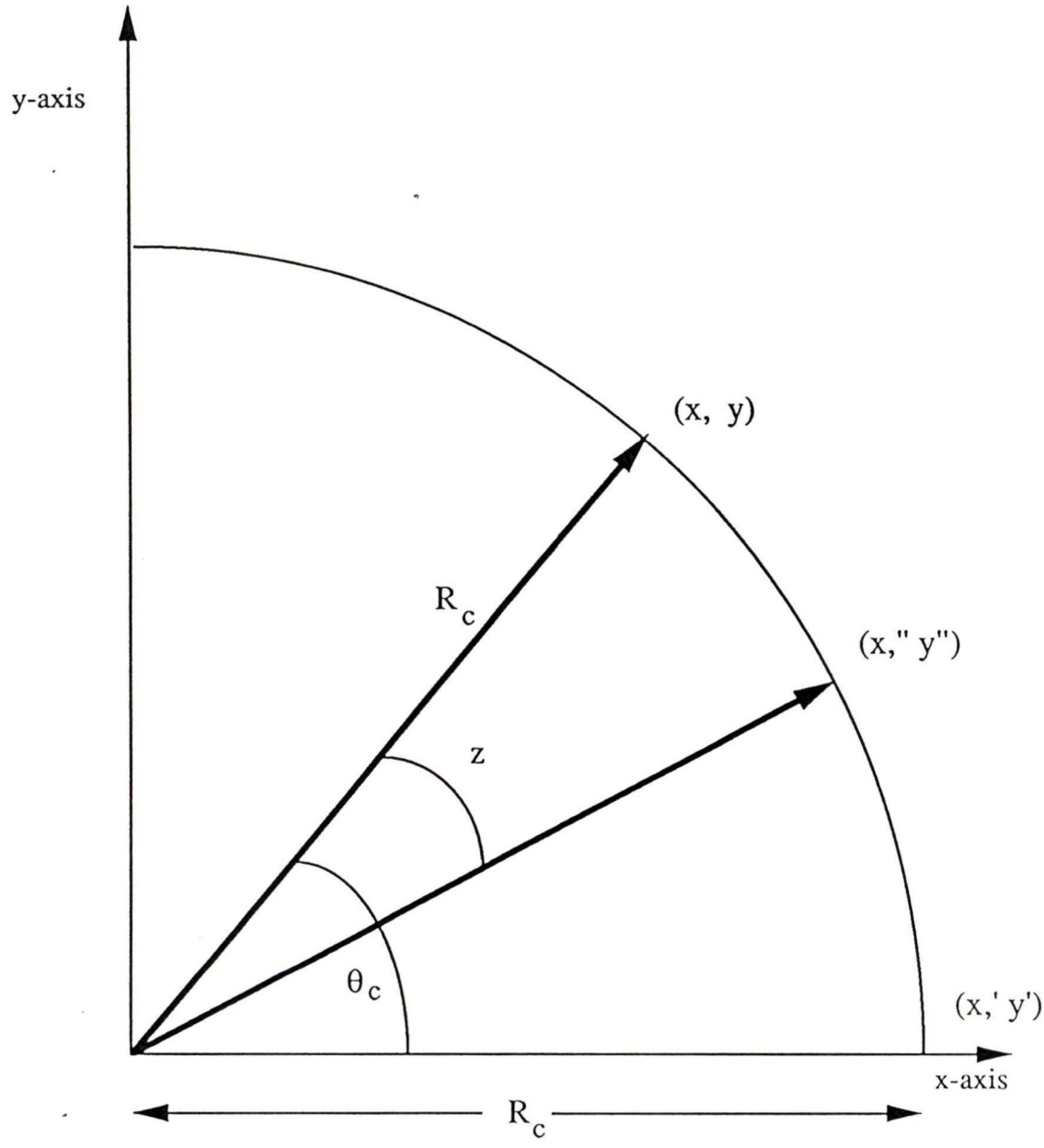
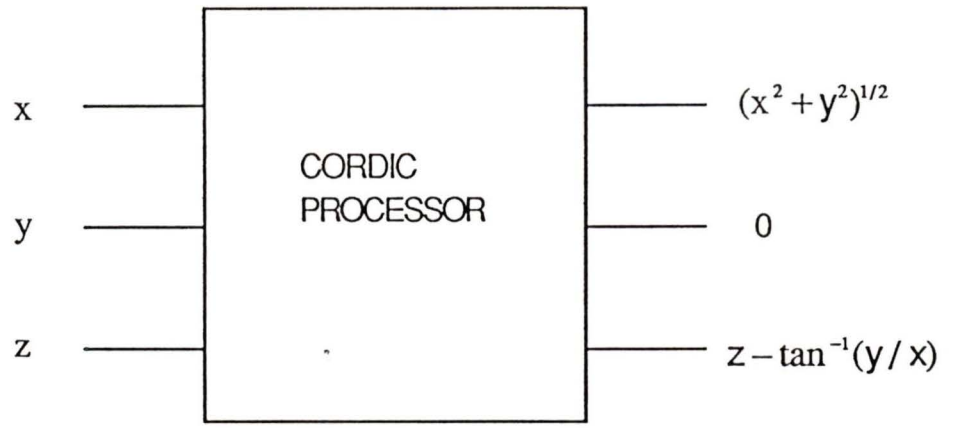
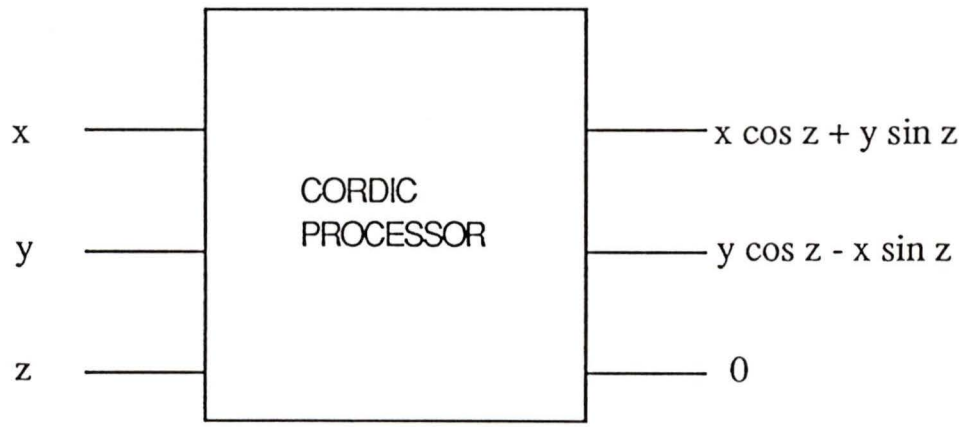


Figure 2.1: For the given triplet $(x,\ y,\ z)$, the coordinates of the end point of a vector, where **R** (circular case) is roated in a clock-wise direction.



(a)



(b)

Figure 2.2: Block diagram of CORDIC processor, (a) shows the vectoring mode and (b) shows the rotation mode, for the circular case.

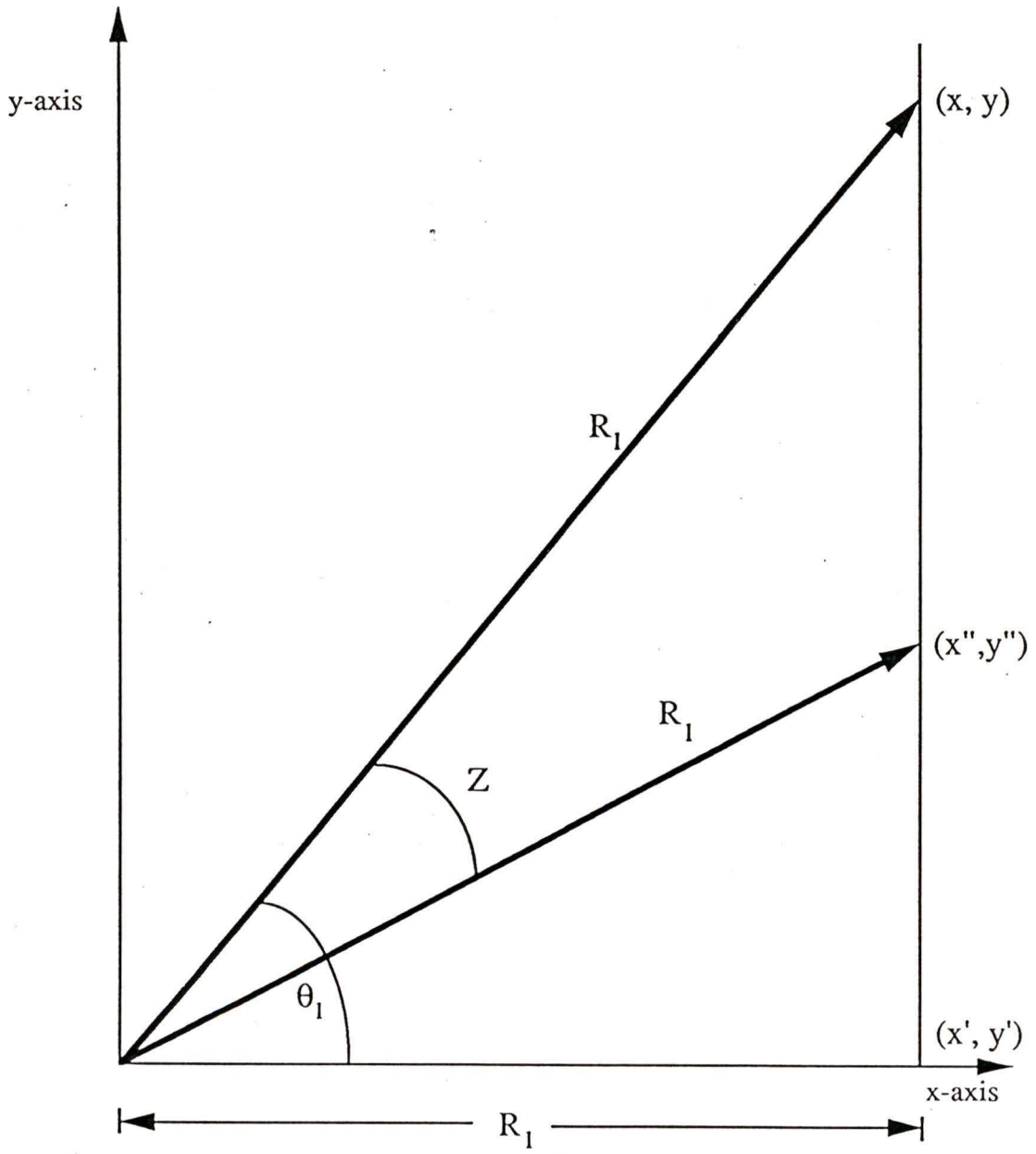


Figure 2.3: For the given triplet $(x,\ y,\ z)$, the coordinates of the end point of a vector, where $\mathbf{R}$ (linear case) is roated in a clock-wise direction.

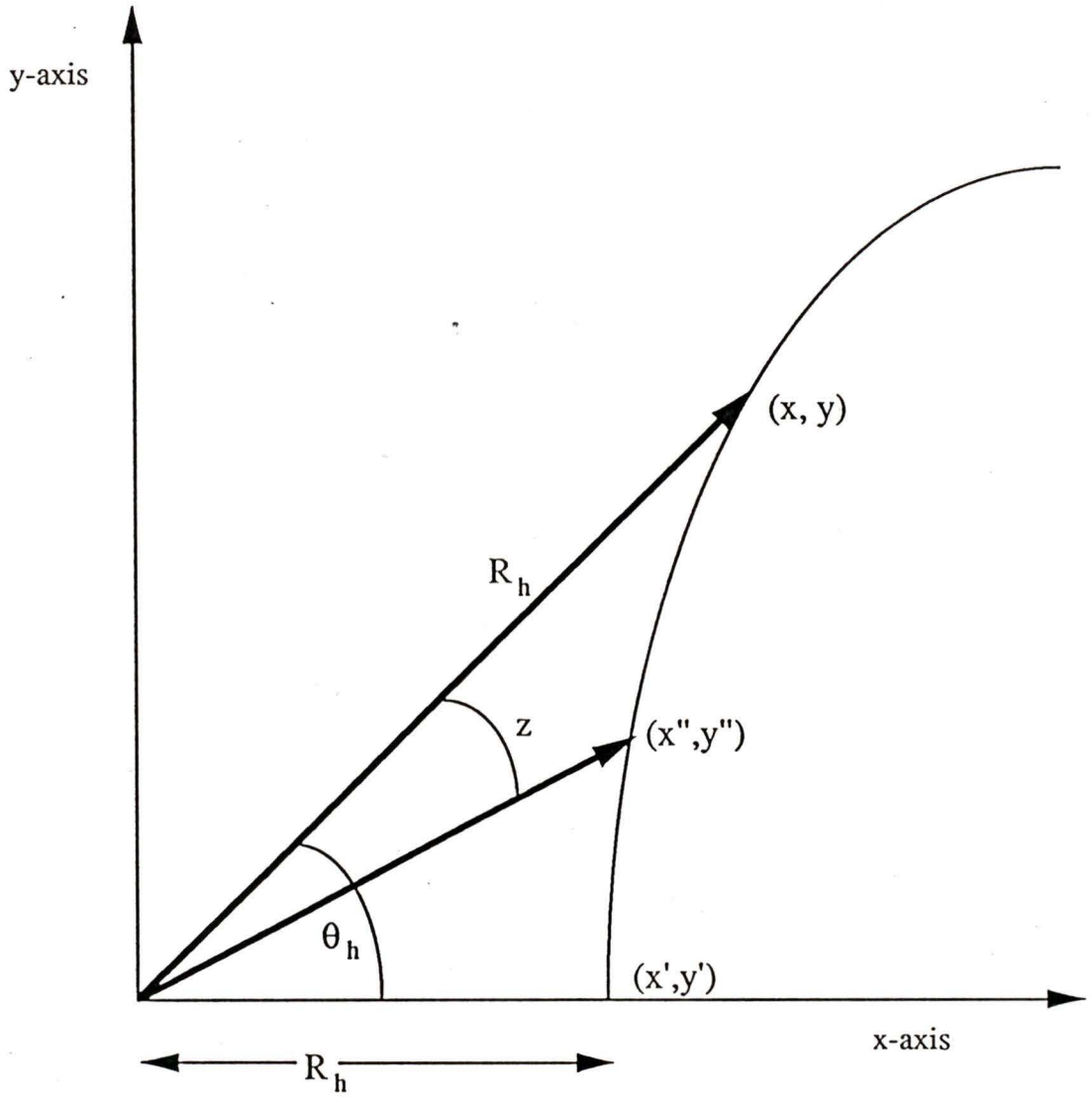


Figure 2.4: For hyperbolic rotation the triplet $(x,\ y,\ z)$ is similar to the circular and linear cases.

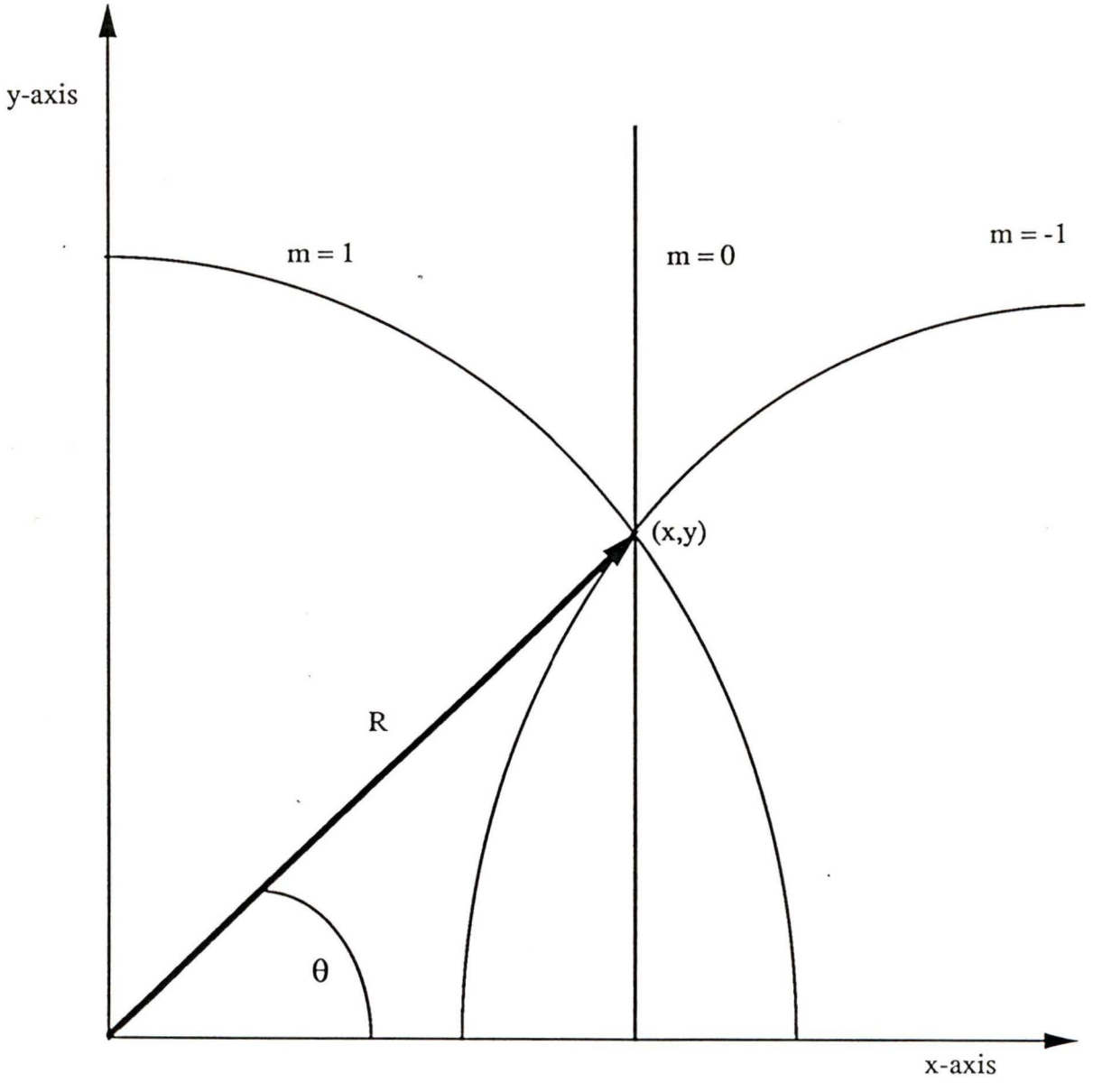


Figure 2.5: For a given vector $p(x, y)$, Walther unified algorithm for three functions (circular, linear and hyperbolic) parameterized by a quantity m.

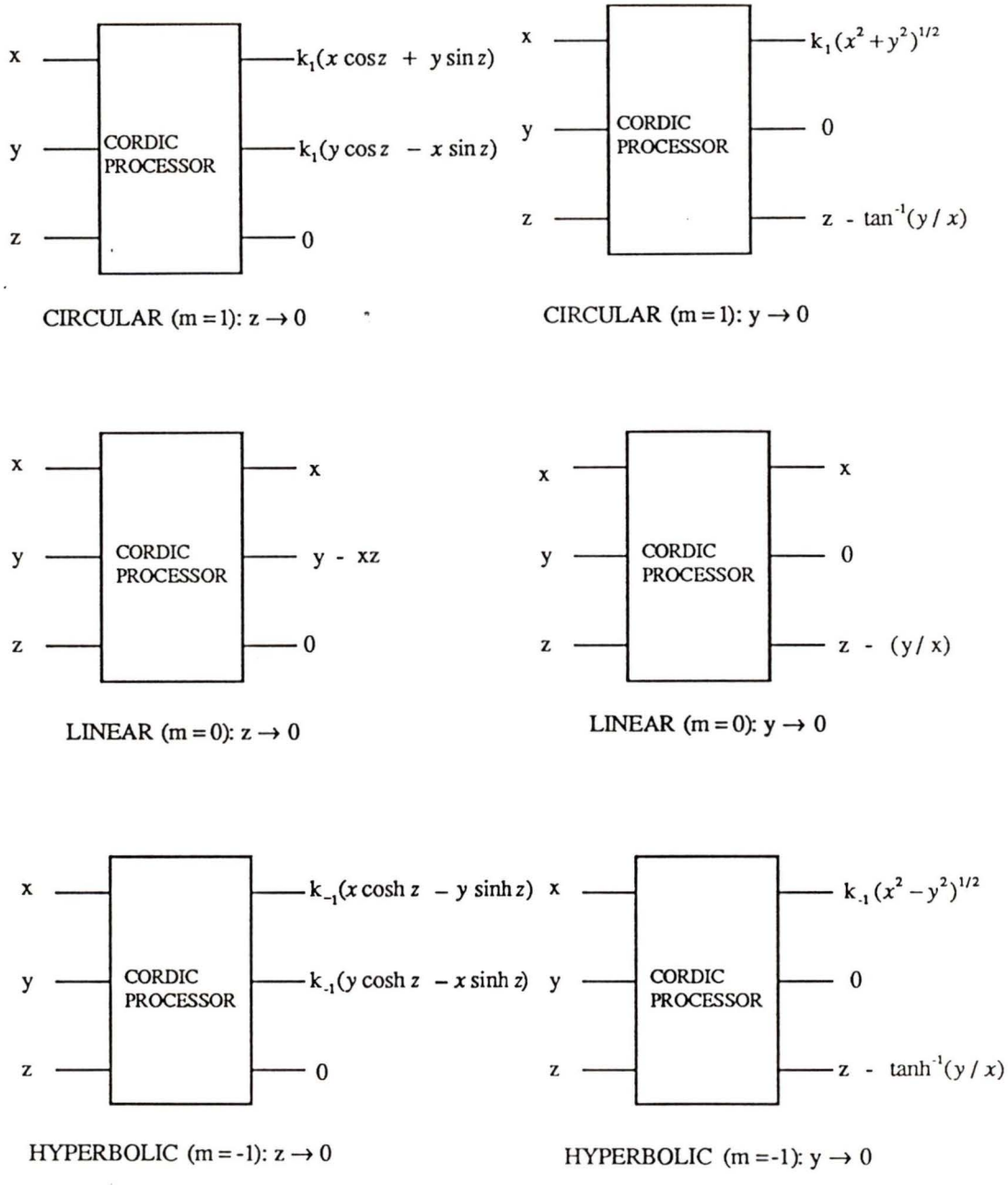
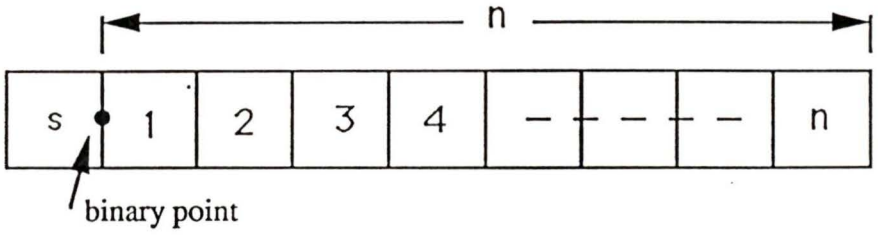
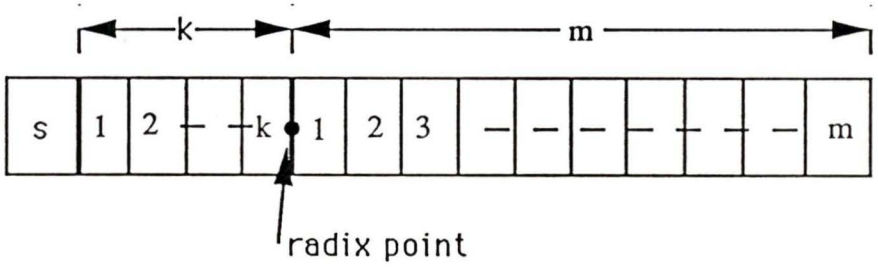


Figure 2.6: Block-diagram for all the special cases for CORDIC algorithm.



s: sign of fraction

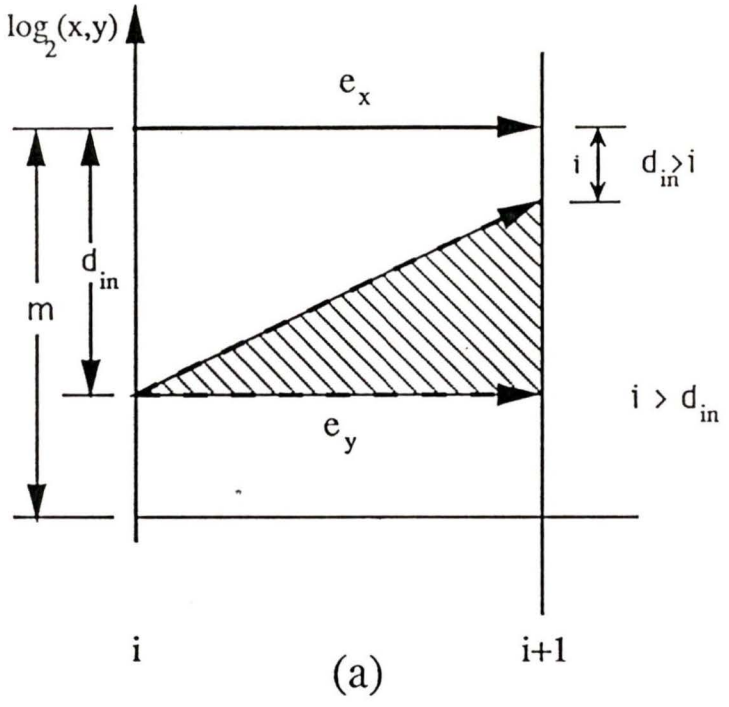
(a)



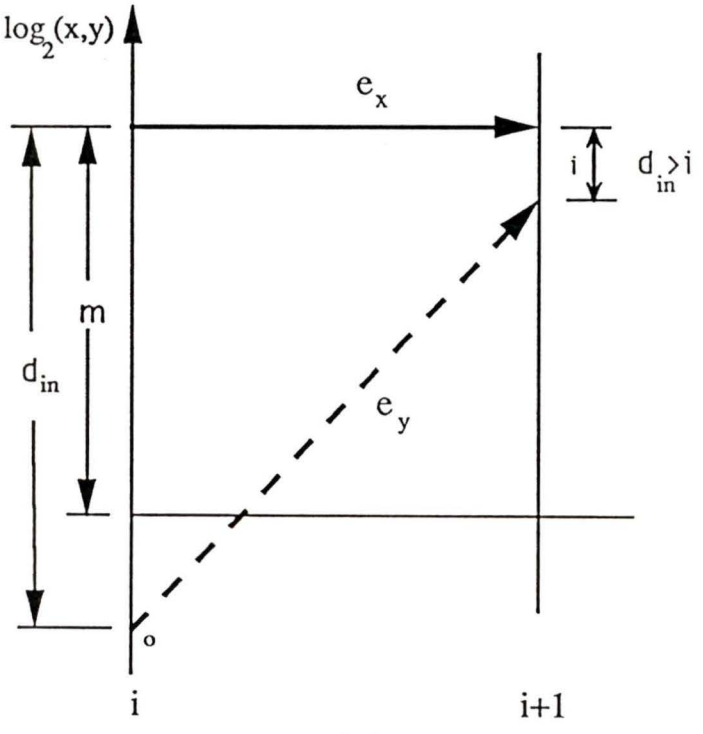
s : sign of the fraction
k-bit exponent biased by 127
m-bit fraction

(b)

Figure 2.7: The fixed-point and floating-data formats

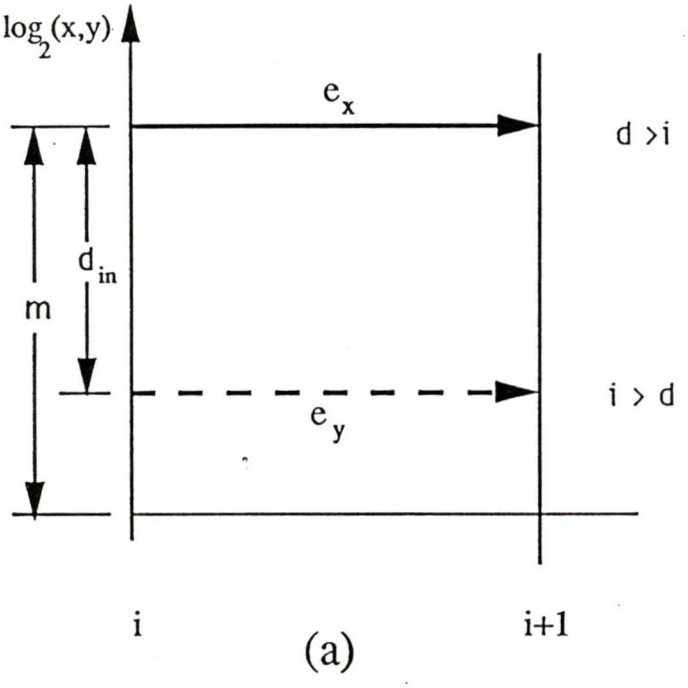


(a)

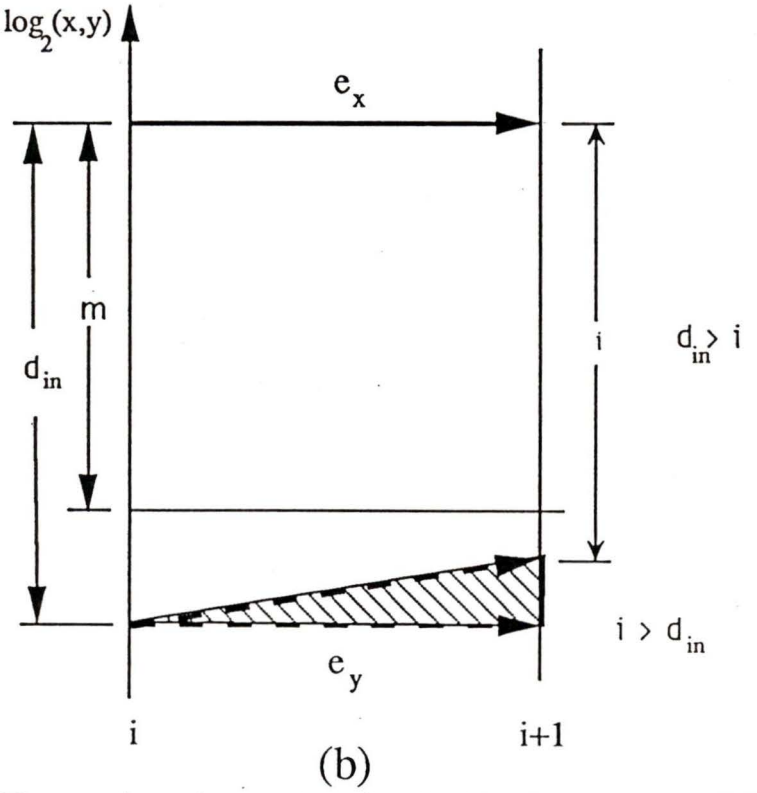


(b)

Figure 2.8: The number of iterations for the floating-point CORDIC processor. Figure (a) and (b) shows the number of iterations when $i \leq m$.



(a)



(b)

Figure 2.9: The number of iterations for the floating-point CORDIC processor. Figure (a) and (b) shows the number of iterations when $i > m$.

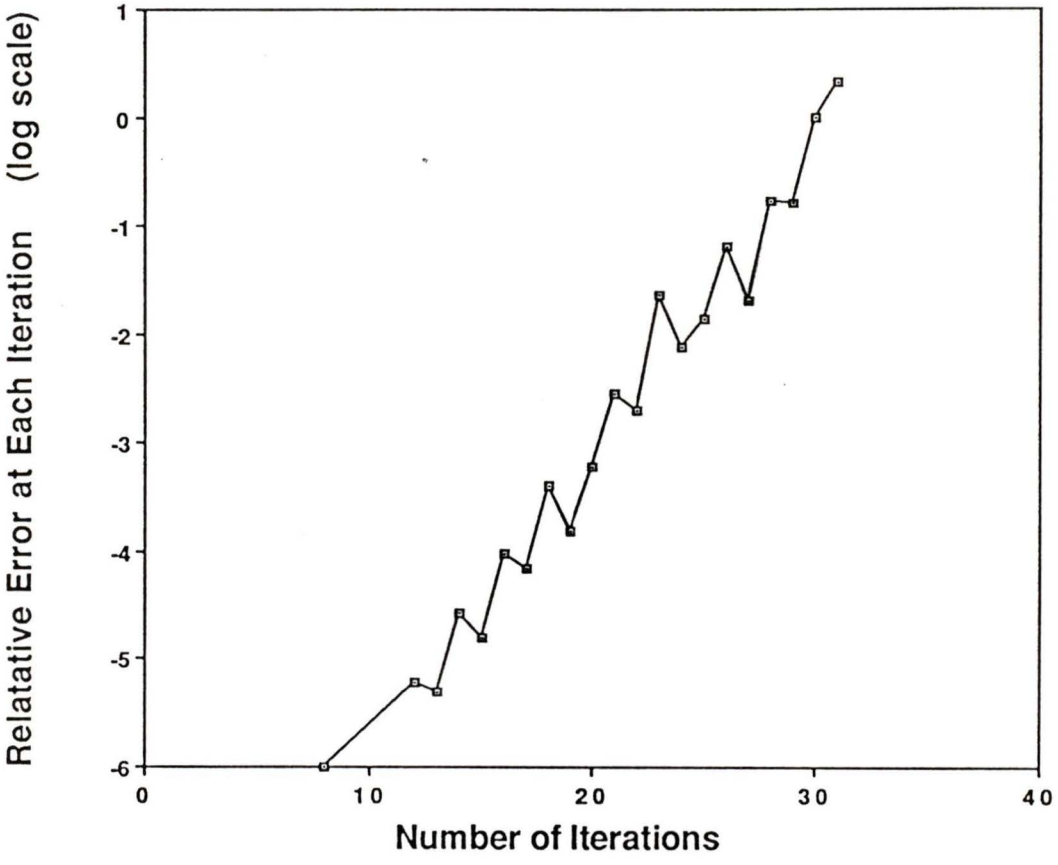


Figure 2.10: Number of iterations verses the relative error for the 32-bit fixed-point data format.

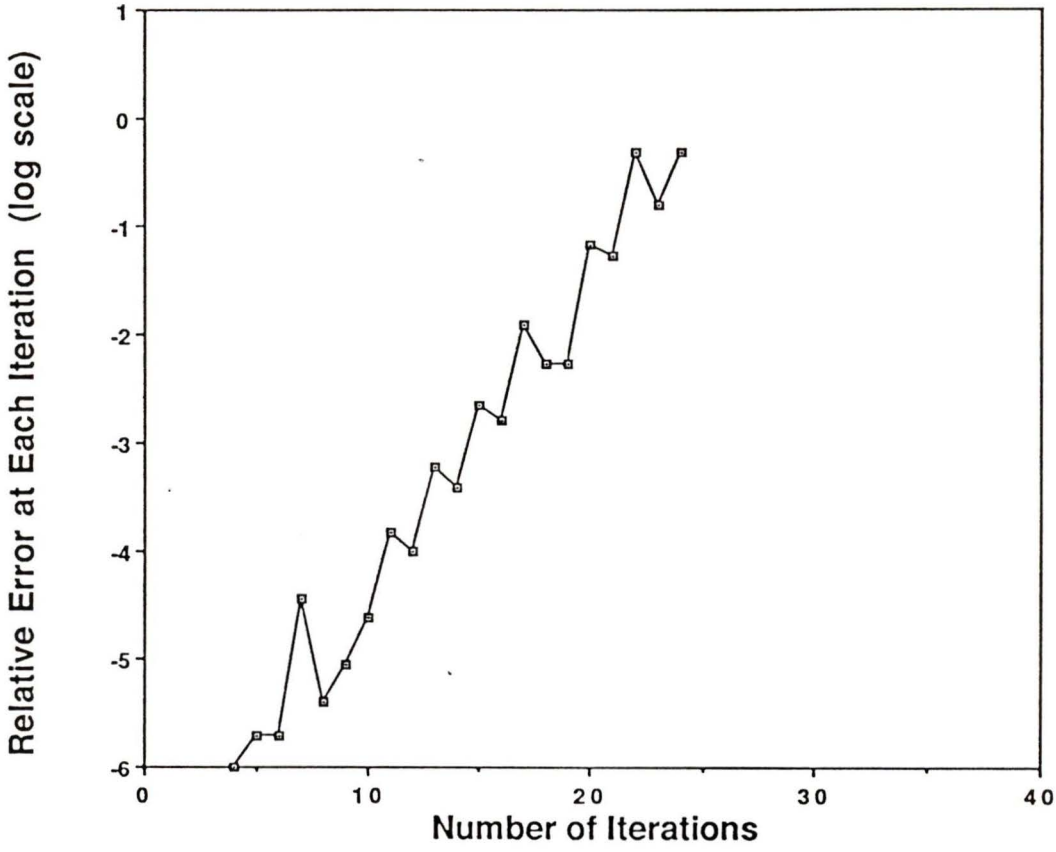


Figure 2.11: Number of iterations verses the relative error for the 32-bit floating-point data format.

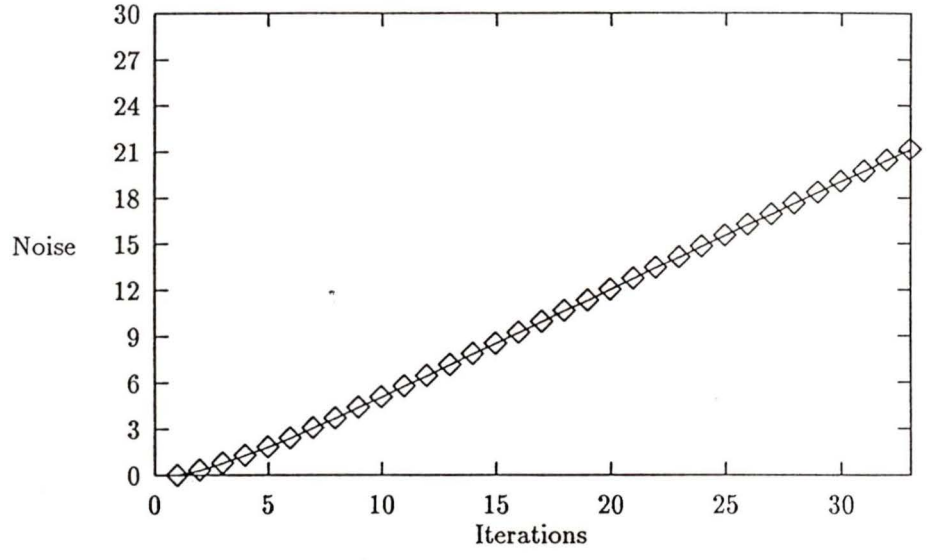


Figure 2.12: Simulation result for noise analysis of fixed-point CORDIC algorithm where the Noise is equal to the "total error" normalized by the "truncation error".

Chapter 3

Processor Architecture and Implementation

3.1 Introduction

There are many design options associated with the hardware design of any data processing system. Some of these design issues are (a) testability strategy (b) the design of each PE, (c) choice of data communication protocol, (d) memory allocation and (e) control strategy. In this Chapter we discuss our design of a systolic array processor or network for matrix algebra operations.

In recent years various kinds of PEs have been developed. For example CAP (cellular Array Processor) [20] is a multi-bit SIMD processor with a fault-tolerant design that allows the use of 16 active processors on a chip, whereas the actual number of processors is 20. Thus, up to four of the processors can fail and the failure will be transparent to the user. Each processing element has a $4K \times 1$, on-chip memory. An external memory can also be multiplexed with this on chip memory. The major functional blocks of each CAP processor are ALU, memory/common bus and I/O. Each block is controlled independently so that different functions

may run concurrently.

NCR's GAPP (Geometric Arithmetic Parallel Processor) chip [21] is composed of a 6×12 arrangement of single-bit processing elements (PE's). Each of the 72 PE's contains an ALU, 128 bits of RAM and bidirectional communication lines connected to its four nearest neighbors. Each instruction is broadcast to all the PEs, making the array perform like a SIMD machine. Many GAPP chips can be cascaded on a board to build an arbitrarily large array of processors in 6×12 increments. The GAPP array can be programmed using low level, macro assembly type language or by using NCRs GAL language. GAL (GAPP algorithm language) is a high-level language designed to enhance the development of GAPP programs.

The INMOS Transputer [22] processor is designed to be the processing element in a mesh connected array, which provides a very high performance building block for parallel computing systems. The IMS T800 has a floating point unit, a central processing unit, 4KB on-chip memory and four serial links to communicate with its four nearest neighbors. In addition to these links, there is a memory interface and an 8-bit bidirectional peripheral interface. The memory interface unit can be used to extend the on-board memory. Occam language is used to program the transputer array.

The MPP (massively pipelined processor) [23] is composed of 128×128 bit-serial processors configured in a square array. The PEs of the MPP have a number of interesting features, including a variable-length programmable shift register, a full adder, multiplexed neighbor inputs and a data structure allowing overlapped array and I/O operations. Eight such processors are integrated on a custom circuit interfaced to high-speed, high-density $1K \times 4$ bits RAM. Because of the high speed of the custom circuits, memory access time dominates operations. The MPP is a

good example of large-scale SIMD computers designed to process satellite images at high rate.

An example of a systolic array processor is the Warp machine [24]. The Warp array is a programmable linear systolic array with identical cells called Warp cells, synchronized by a single global clock issued by the interface unit. During computation, data, addresses and controls all flow between neighboring cells and there is no global communications at all. Each Warp cell has its own program memory and sequence controller. The data path of a cell consists of two floating point processors: one multiplier and one ALU, a 64K word memory, a sequence for X and Y communication channels, a register file to buffer data for each arithmetic unit. A boundary processor may be attached to one end of the Warp array. The boundary processor has bidirectional links to the interface unit, the first cell and to the last cell in the array. This ring connection makes it possible to recirculate data around the Warp array without involving the host. The Warp processor array may be configured to operate as a SIMD os as a MIMD machine.

The Connection machine from Thinking Machine Corporation [25] is a binary array processor consisting of a matrix of identical PEs operating in SIMD mode. It has a large number of very small processor/memory cells connected by a programmable communication network. Each cell on the connection machine contains a $4K \times 1$ bits memory and a simple serial arithmetic logic unit. Sixteen cells are connected in a 4×4 grid with a router unit on a custom CMOS chip. The router is responsible for routing messages between the chips. Within the chips each processing cell can communicate directly with its four neighbors without involving the router. All processors execute instructions from a single stream (SIMD mode) generated by a microcontroller under the direction of a host computer.

3.2 System Requirements

Digital signal processing and scientific applications demand high performance capability, for-example in matrix algebra operations [26], [27] a large dynamic range and sufficient accuracy are required to reduce computational noise. In image processing applications a huge amount of data has to be processed in a short amount of time. The basic operations used in the above applications are not confined to the simple multiply/accumulate operations.

Most signal processing algorithms have parallelism as an inherent property. This can be used to advantage in designing a processor array. However the processing elements (PEs) in these arrays have to be capable of performing several elementary functions in floating-point format.

From the above two broad requirements, it was decided to have the PEs in our processing array capable of implementing the CORDIC algorithm in floating-point format. The versatility of the CORDIC algorithm [Chapter 2] allows trigonometric operations as well as multiplication and division to be performed. Typical processor array structures are shown in Fig. 3.1. The linear systolic array of Fig. 3.1 (a) is useful for several applications such as convolution [28], filtering and matrix-vector multiplication [29]. The triangular systolic array of Fig. 3.1 (b), a special case of the mesh connected square array, is useful for solving system of linear equations and for matrix decomposition.

From the above discussion the system specifications are as follows:

1. The PE must be programmable and the PE interconnection must be configurable.

2. The PE must be capable of working in 32-bit, floating-point format.

3. The PE must be capable of doing the basic CORDIC algorithm operations.

4. The PE must be testable.

5. The PE must be capable of communicating with its nearest neighbors in a bit-serial fashion to conserve the number of chip pins.

6. In order to reduce processor communication and control complexity, each PE should be designed to operate as independently as possible.

3.3 Design Approach

In addition to the system design considerations discused in Section 3.2, VLSI system design options are affected by more design choices. These extra design choices are system complexity, chip area requirements, pin-out and CAD tools availability. In order to cope with the complexity of the system, a highly structured design methodology is needed. Figure 3.2 shows the design steps necessary for VLSI implementation of a design. Our design process is comprised of three concurrent design operations, behavioral design, structural design and physical design. Behavioral design refers to the decomposition of the initial design specification into simpler sub-designs and sub-specifications. This decomposition process is often referred to as top-down design technique. Using the top-down and bottom-up approaches, objectives at higher level become the design specification for the lower-level design. The structural design is defined as the realization of a logic block or structure through the interconnection of more primitive structures. At some point in the project, a mapping from the behavioral design onto the structural design is

necessary. This mapping will be referred to as the logic design process. Behavioral and timing specifications can be verified using computer simulation at any stage of the design depending on the CAD system available. The VTI CAD package available from VLSI Technology Inc., was used to design our CORDIC processor. This CAD package gives us the ability to drive simulators and to generate complete test programs. We use VTI test language (VTL) to create a modular description of the physical characteristic, timing, stimuli patterns and expected response values of the device under development. The physical design operation refers to the actual realization of a structure in a given technology. This physical design utilizes a bottom-up design approach by combining lower-level physical design to realize higher-level ones.

3.4 Processor Level Design

An icon for the node is shown in Fig. 3.3. A detailed block diagram of this icon for the floating point CORDIC node as derived from the above mentioned specifications is shown in Fig. 3.4. This section specifies the logic functions, the signal connections (I/O and control lines), the component test modes, and the timing diagrams. As shown in Fig. 3.1 the PE is to be incorporated into a mesh-connected array capable of performing the matrix algebra algorithms essential for DSP applications. The pin-out for each PE is defined in Table 3.1 and explained below.

Pin Name	*Type*	*Function*
SIO 0	I/O	Serial data North side
SIO 1	I/O	Gated clock North side
SIO 2	I/O	Port status North side
SIO 3	I/O	Serial data West side
SIO 4	I/O	Gated clock West side
SIO 5	I/O	Port status West side
SIO 6	I/O	Serial data South side
SIO 7	I/O	Gated clock South side
SIO 8	I/O	Port status South side
SIO 9	I/O	Serial data East side
SIO 10	I/O	Gated clock East side
SIO 11	I/O	Port status East side
PIO [15:0]	I/O	Parallel port for testing
address[2:0]	Input	address bus for PIO block
control bus[9:0]	Input	control bus to read and write
		the address/data into appropriate registers
Clock	Input	external clock input in testing mode only
Reset	Input	Force PE to a known state
SCK	Input	System clock signal
Test	Input	Place in test mode

Table 3.1: Function and description of the PE pins.

3.4.1 Pin Description

SIO[11:0]

The serial I/O pins (north, west, south and east) allow the PE to communicate synchronously with its four nearest neighbors. Each serial data pin may be configured as either input or output to transmit or receive program or data information.

PIO[16:0]

The 16-bit wide parallel port is used to feed in the test vectors and to receive test results. The test vectors, are generated externally and test the different internal blocks. The PIO bus is also used for loading the initialization program for each PE in the mesh connected array.

address[2:0]

In the test mode the PE internal controller will be disabled. Address[2:0] is used to select one of the eight internal registers of the PIO block. Out of these sixteen registers, two registers are reserved for PE internal address bus, two for data bus and four for the control bus.

control bus[9:0]

The control bus is used to control the operation of PIO block.

Clock

This clock input is used to load the input data to the PIO block during the test mode operation.

Reset

This input is a direct hardware reset for the processor. When reset is asserted, the processor is initialized and placed into a known state. Reset is typically applied after power up when the PE is in a random state. Asserting the Reset causes the processor to terminate execution and forces the program counter to zero. Reset also will affect all the registers and status bits. After the reset, the processor execution begins at location H00, which normally contains a jump statement to the first instruction in the initialization routine.

Clock (SCK)

This is where the system or global clock (SCK) is input to the PE. The processor will generate the local clock ϕ to control the operation of I/O block, datapath and control sections.

Test

The test pin places the PE in test mode. the testing strategy is explained in section 3.5.

3.4.2 Functional block diagram

The functional block diagram of the PE shown in Fig. 3.4 outlines the major blocks and datapaths within the processor. The PE is a 32-bit floating-point, multiple bus processor designed specifically for real time matrix algebra operations for DSP. Further details of each block and the buses are provided below.

Address Bus

The address bus allows the control unit to provide access control over all major blocks.

Data bus

Local data movement within the PE occurs over one 32-bit bidirectional bus. External data movement occurs through one of the four serial Input/Output Blocks (SIO) or the parallel I/O block (PIO).

Control bus

The control bus is used to carry the control signals generated by the control section of the PE.

Flag bus

The flags generated by the different sections of the PE will be sent back to control section on the flag bus.

Pin Name	*Type*	*Function*
PIO[16:0]	I/O	parallel I/O bus
address bus[2:0]	Input	address bus
control bus[9:0]	Input	control bus
clock	Input	external clock signal

Table 3.2: Function and description of inputs and outputs to the PIO block.

Datapath Block

The datapath is a 32-bit floating-point unit capable of performing the CORDIC algorithm operations. It consists of two add/subtract units. One add/subtract units handles the mantissa part of the data and the other unit handles the exponent part of the data. Further details of the datapath are discussed in section 3.6.

Parallel Input/Output Block (PIO)

Table 3.2 summarizes the function of each pin of the PIO icon. The parallel input/output block is provided to facilitate testing of the PE. When the test pin is asserted, the PE will go into test mode. In the test mode all the internal blocks will go into a tristate mode. The parallel I/O block is used to provide external control of the PE through the PIO[16:0] bus. The PIO block consists of two 16-bits internal buffer registers for each internal data bus and address bus. Where as the internal control bus has four 16-bits buffer register. In one clock cycle the even data is latched on to the buffer register from the PIO[16:0] and in next clock cycle the odd data is latched.

The input address to the PIO block is latched on the falling edge of the input clock signal. A timing diagram for the read and write cycle of PIO block is shown in Fig. 3.6. The address[2:0] is used to load the address and data contents from

PIO[16:0] bus into the appropriate register inside the PIO block. Figure 3.6(a) shows the timing of loading the address and data into PIO register from the PIO bus, while (b) illustrate the timing of the data from the data register of 'PIO block' on to PIO bus. To perform each of the above defined operation and other decision making operation, inside the PIO block, are controlled through the control bus[9:0].

Serial Input/Output Block (SIO)

An icon for the SIO is shown in Fig. 3.7. Table 3.3 summarizes the function of each pin of the SIO icon. There are four I/O blocks corresponding to the four (north, west, south, east) communication directions. All the data transfer between the processor and outside world is being done through the serial I/O blocks. The serial data pin allows the PE to communicate synchronously with its nearest neighbors. A timing diagram of the port protocol for transmitting and receiving data is shown in Fig. 3.9. Each serial data pin may be configured as either input or output to receive or transmit the control or data, respectively.

Serial data transfer is synchronized by use of a gated clock. In this way the bit synchronization is accomplished. There are four gated clock pins for each north, west, south, east serial data pins. This pin is active when data is being transferred between this processor and its neighbor and the direction of this pin is the same as the serial data pin. The port status signal indicates the state of the I/O of the neighboring processor to which it is connected. This is a kind of handshaking by which the processor can know the state of the other processor's port, i.e. whether it is ready to accept the data or it is busy. The direction of PS signal is opposite to that of the serial data and Gated clock signals. Thus if the PE is sending data to a neighboring PE, it checks the state of the PE signal to see if the receiver is

Pin Name	*Type*	*Function*
SD	I/O	Serial data
CK	I/O	Gated clock
PS	I/O	Port status
D[31:0]	I/O	Parallel data
A[3:0]	Input	Address bus
Control Bus	Input	Control Bus
Flags	Output	Flag bus
ϕ	Input	Internal clock signal

Table 3.3: Function and description of inputs and outputs of the SIO block.

ready to receive the data. Once the data has been sent, the PE is again checked to make sure that the data was successfully received.

The flags generated by the I/O blocks are sent to the controller on the flag bus. By using these flags the controller will know the status of the port. The register containing this information is known as status control register (SCR). The SCR and other I/O block register are shown in Fig. 3.10. Similarly the flags generated by the header register are sent back to the controller on the flag bus.

The operation of each serial I/O block is similar to each other and independent of the operation of the datapath unit. Each I/O block provides a half duplex port for serial communication to other processors. A timing diagram for the serial I/O block read and write cycles is shown in Fig. 3.8.

Control Block

An icon for the controller is shown in Fig. 3.11. The controller generates the address of the next instruction, controls the datapath as well as data movement between the IOBs and the program/data RAM. Table 3.4 summarizes the function of each line at the periphery of the controller icon. The control block is a microcode

Pin Name	*Type*	*Function*
Data bus		data bus
Control bus	output	control bus
Flag bus	input	flag bus
Reset	Input	place the PE in a known state
Test	Input	place the PE in test mode
ϕ	Input	clock signal

Table 3.4: Function and description of inputs to the Control block.

based controller, implemented with the AMD 2910A.

Program/Data RAM block

The random access memory block stores the sequence of operations to be performed by the PE. It also stores data, as well as coefficients for the CORDIC algorithm. Data may be transferred between the datapath, the program/data RAM and the IOBs by the controller. A timing diagram for the RAM read and write cycle is shown in Fig. 3.12.

Microcode Control ROM Block

The control ROM is the memory where the controller microcode is stored.

Clock Generator

The local clock ϕ, used in the I/O block, controller and datapath is generated locally in the PE based on the system or global clock (SCK).

3.5 Testing Strategy

To test the hardware of each internal block of the PE, a test pin is provided along with external 3-bit address bus, 16-bit bidirectional bus and a 10-bit control bus. When the test pin is not asserted all connections between the PIO and the PE internal buses are tri-stated. The PE is operating in the normal mode. When the test pin is asserted the PE is in test mode. All connections between the PIO block and the PE internal buses are established. The connections between the control block and internal address and control buses are tri-stated. When the PE is in normal operation, the SCK signal provides the clocking information for the PE. When the PE is in test mode, the pin "clock" provides the clocking information of the PE.

The PIO block is used to load the PE with the test inputs and to extract the PE responses. A block diagram of the PIO block is shown in Fig. 3.13. The PIO[15:0] bus is used to load and retrieve the test vectors from the PIO registers. The address[2:0] bus is used to select one of the eight 16-bits internal registers of the PIO block. The control bus[9:0] determines direction of flow of data and the loading of the PIO registers.

To test any block of the PE the test vectors will be loaded through the PIO[15:0] bus to the eight internal PIO registers. Once all the information are loaded in these registers, the pin "clock" will activate the operation of the PE. The response of the PE will be captured through loading the PIO registers from the internal control bus, flag bus and data bus. Once the response is captured, the PIO bus is used to send the data to the outside world. This output response can be compared with the desired response.

Pin Name	*Type*	*Function*
D[31:0]	I/O	Data Bus
Address bus	Input	Address bus
Control Bus	Input	Control bus coming from the controller
Flags	Output	Flag output lines
DPS	Input	Datapath Select
ϕ	Input	Internal clock signal

Table 3.5: Function and description of datapath buses and I/O pins.

3.6 Datapath Block

The datapath icon is shown in Fig. 3.14. The functions of all I/O pins of the datapath are summarized in Table 3.6.

The major connections, with outside world, on the periphery of the data-path block are as follow:

Data Bus (D[31:0])

The data bus is used for the movement of bidirectional data from the datapath to the RAM block or from the RAM to the datapath block.

Address Bus

The address bus is used to select one of the four datapath registers as a destination for the data loaded from the program/data RAM.

Control Bus

The control bus is used to control the operation and mode of datapath block. After loading the registers, the controller will send the proper signals to perform

the required operation of the datapath.

Flag Bus

The flags generated by the datapath block is sent back to the controller on the flag bus. By using these flags the controller knows the status of the results of arithmetic operations.

Datapath Select (DPS)

This input pin is used to select the datapath block for the data movement between the datapath and program/data RAM.

3.6.1 Datapath Design

The specifications governing the design decisions for the datapath are as follows:

1. It should be optimized to handle floating-point data of 24-bits mantissa and 8-bits exponent parts.

2. It must be capable of supporting bit shifts, mantissa justification, and addition/subtraction operation.

3. It must be capable of recursive data flow.

Based on these specifications a datapath was designed. The datapath functional block diagram is shown in Fig. 3.15. The datapath design is based on two functional units, one each for the mantissa and the other for the exponential components of the floating-point data. The logic blocks in Fig. 3.15 were designed using the VTI tool-set. The datapath is capable of performing any basic operation within one

clock cycle. For example, within one clock cycle, one could add or subtract two operands and perform a one-bit shift operation on the result. The output of the datapath is connected to the data bus by a noninverting tri-state buffer.

The operation of the datapath in brief is as follow: On the falling edge of the clock, the appropriate control signals from the controller are stable. The data from the data bus will be loaded to one of the scratchpad registers. A timing diagram for loading the input registers from the data bus and operation of the datapath is shown in Fig. 3.16. In one clock cycle the first operand will be loaded to an input register. In the following clock cycle the second operand will be loaded to another scratchpad register. Upon completion of an operation, the output can be loaded back to the registers by enabling the tristate gates at the next clock cycle. Alternatively, the resulting data can be written into the RAM using the data bus.

3.6.2 Add/Sub Units

The add/sub unit has inputs a and b and carry input CIN. The control input "add/sub" is low, the cell performs the addition operation. When the control input is high, the cell performs the subtraction operation. As described earlier, there are two add/sub units used in datapath unit. One is used for the mantissa and other is used for the exponential part of the data. Both add/sub units are similar in operation but the size is different. Mantissa add/sub units is 24 bits wide, whereas exponent add/sub unit is 8-bits wide. Each Add/sub has four scratchpad registers. These registers latch the input data at the rising edge of the clock. The input multiplexers route the data from the scratchpad register to the a and b inputs of each add/sub unit. Both multiplexers also have an inhibit capability; i.e., no data is passed, this is equal to a "zero" source operand. This mode is used

as zero operand to the one of the input of add/sub unit. Referring to Fig. 3.15 the multiplexer at the input a of add/sub has the output of all the four registers connected as inputs. Likewise, the multiplexer at input b has the input of all of the registers. This multiplexer scheme gives the capability of selecting one of the four inputs as a source operands to the add/sub unit.

The hardware for the mantissa part on a cell level bases was designed and simulated using the VTI tool-set. A diagram of this mantissa block is shown in Fig. 3.17. Each cell shown in the diagram is 24-bits wide. All the "data-in" inputs are directly connected to the data bus. Referring to the Fig. 3.18, the loading of the input latches is done by enable LL[3:0] line, depending upon the latch to be loaded.

3.6.3 Shifter

At the output of the add/sub cell a zero detect circuit and an up/down one-bit shifter with control is connected. The shifter unit is a single bit shifter cell with the two control lines, control[1:0]. The control[1] input, when high, causes data to shift up, when low, causes data to shift down. The control[0] input disables the shift when low, so no shift occurs. The control[0] input overrides the control [1] input. This shifter unit is connected to the output of the mantissa stage only. The exponential part has no shifter at the output. The zero detect unit at the output of the add/sub unit, will set a flag if output of the add/sub unit is zero. This flag along with the "carry out" and "over flow" flags will be checked by the controller for the conditional operations.

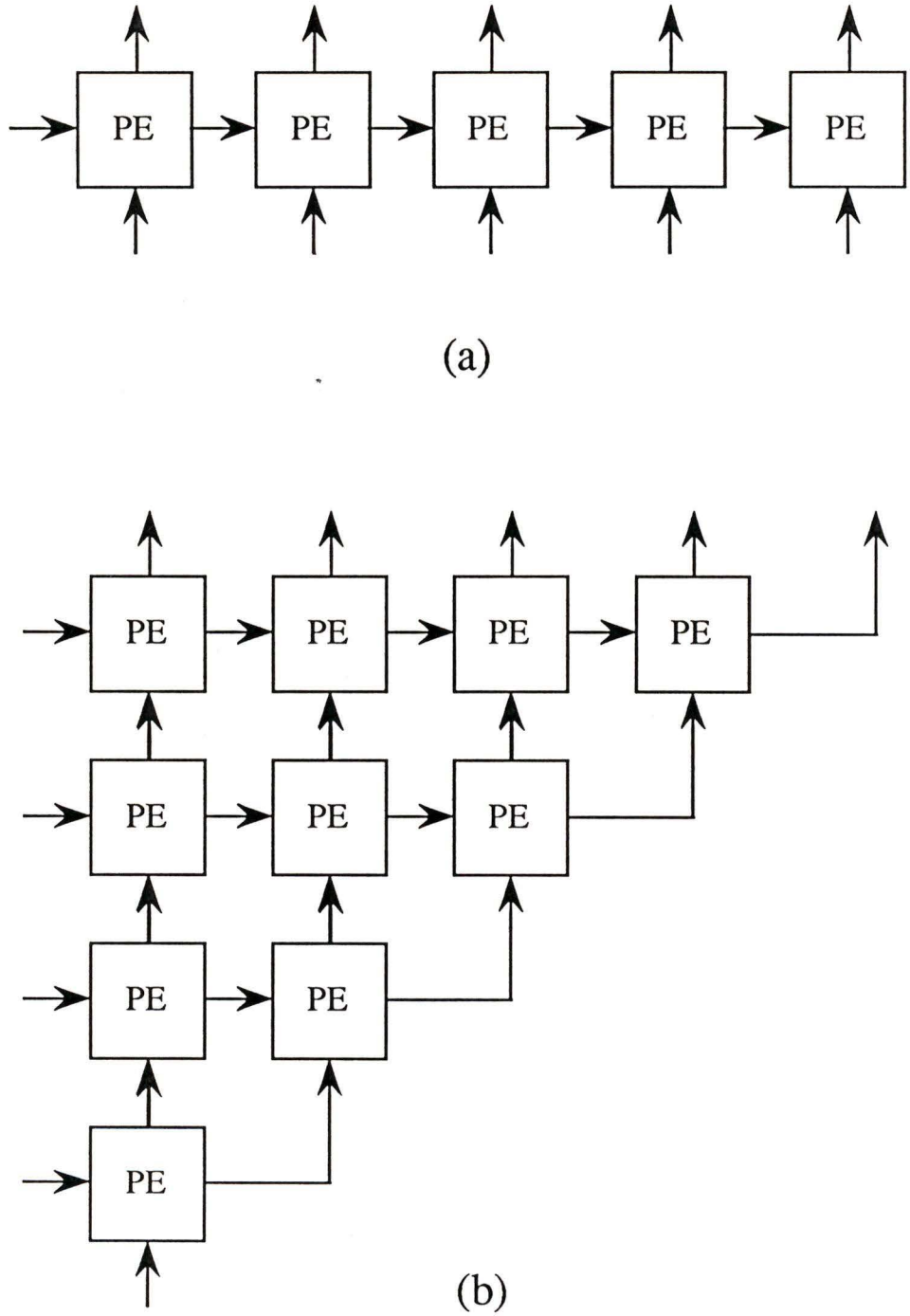


Figure 3.1: Mesh connected processor arrays. (a) shows a linearly connected and (b) triangularly connected processing element (PE), where each PE is a floating-point CORDIC processor.

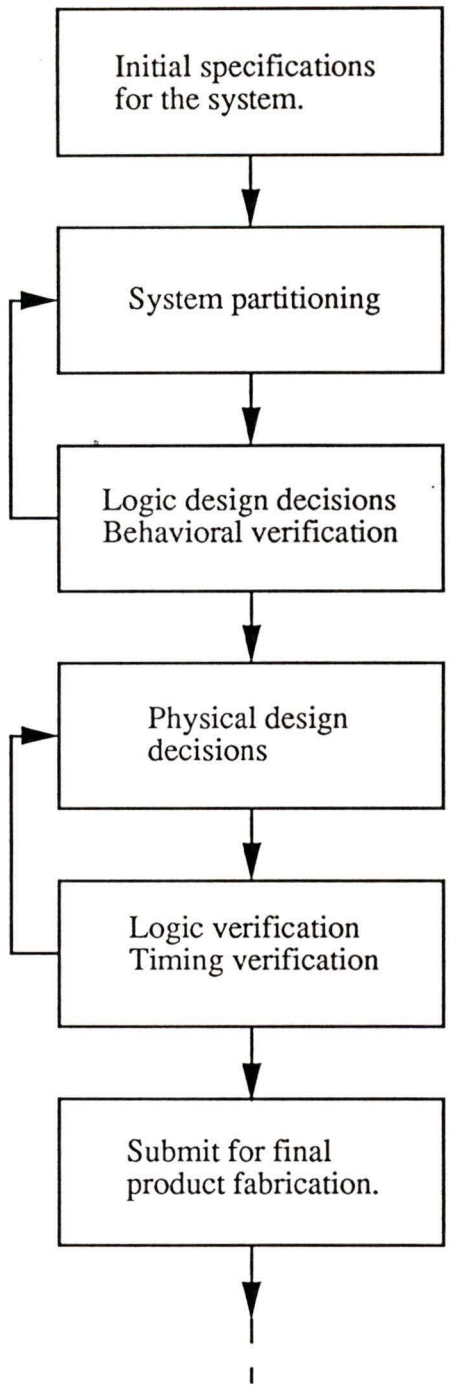


Figure 3.2: The flow diagram of design decisions and modifications associated with the system design.

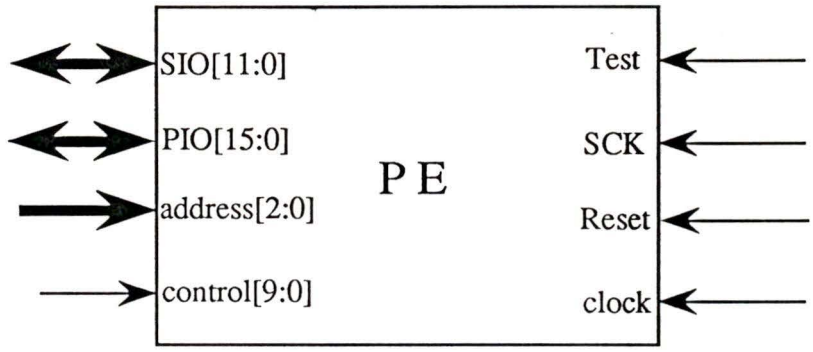


Figure 3.3: A processing element (PE) icon, showing different buses and control lines at its periphery.

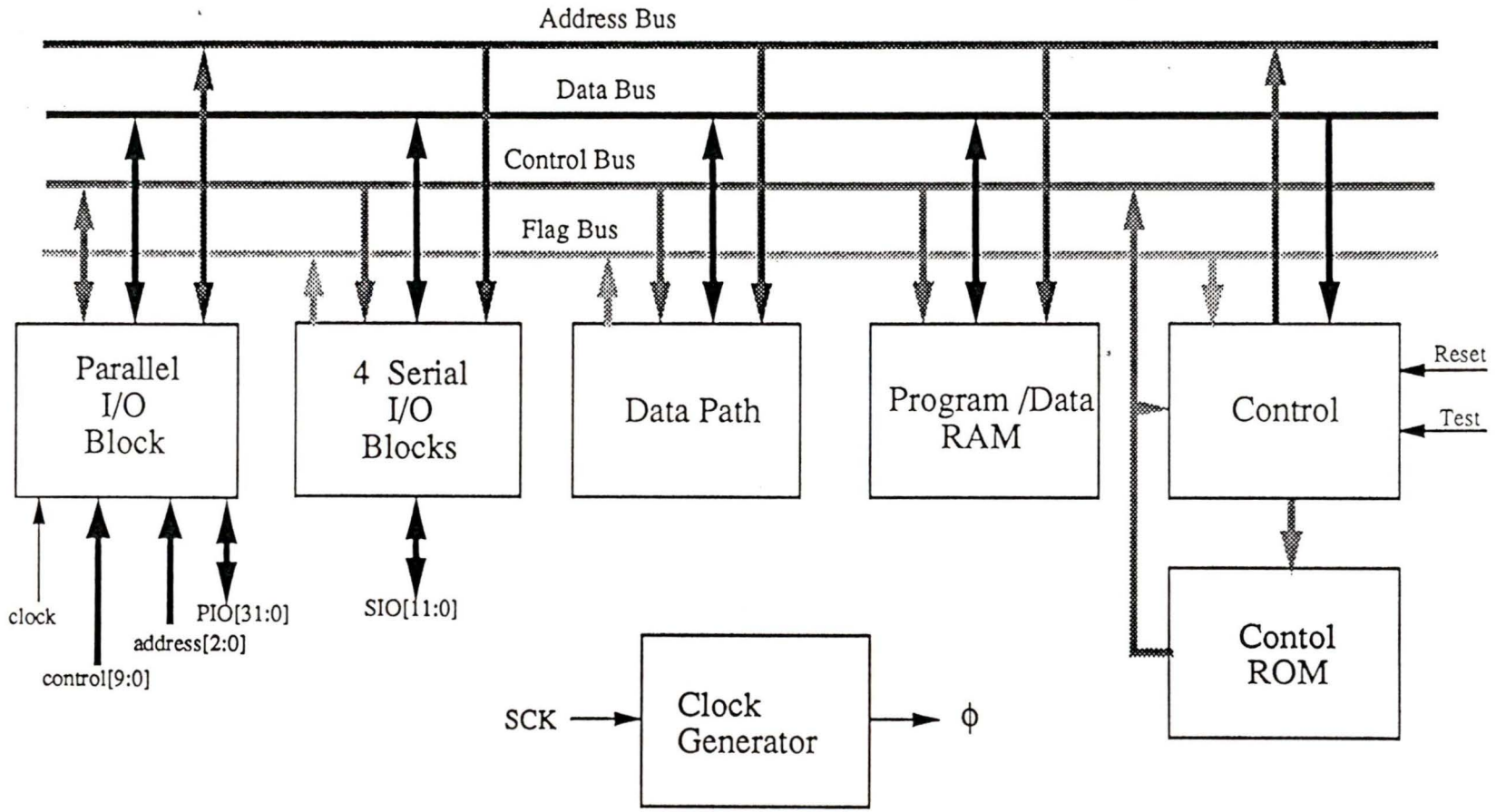


Figure 3.4: A proposed block diagram for floating-point CORDIC processing element (PE).

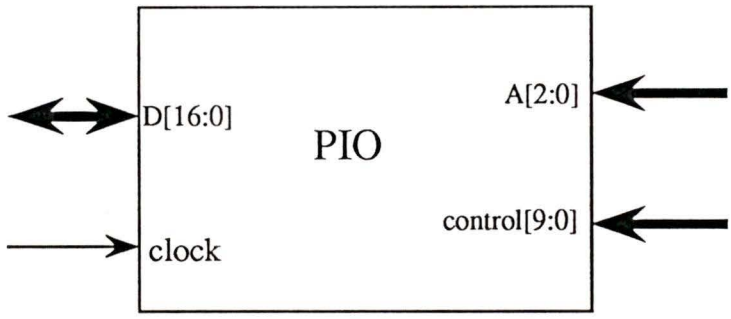
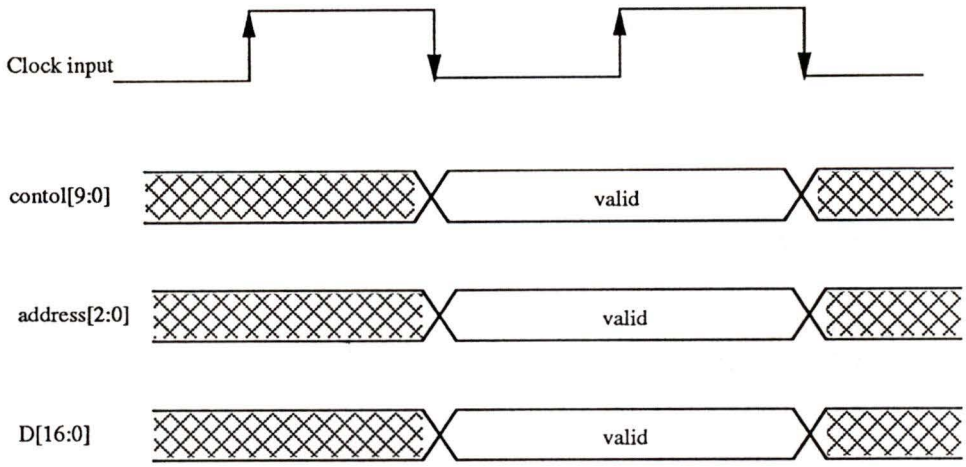
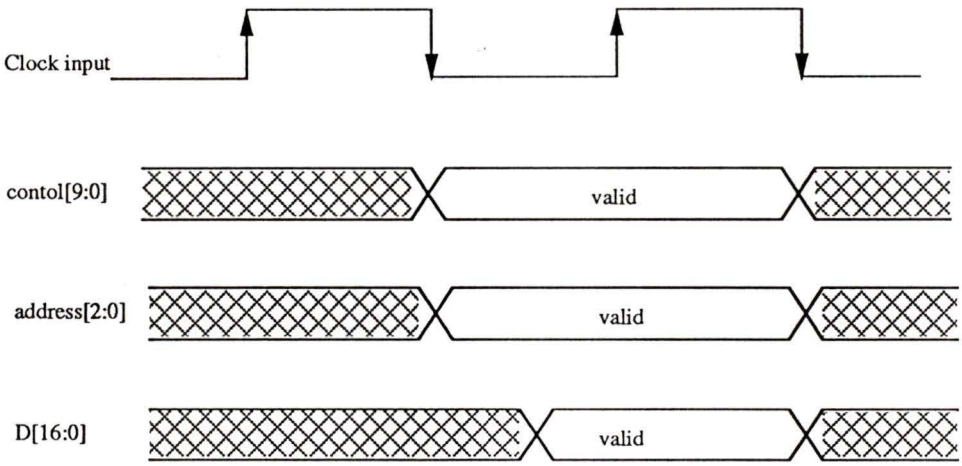


Figure 3.5: The PIO icon, showing different buses and control lines at its periphery.



(a)



(b)

Figure 3.6: A timing diagram for the read and write cycle of the PIO block.

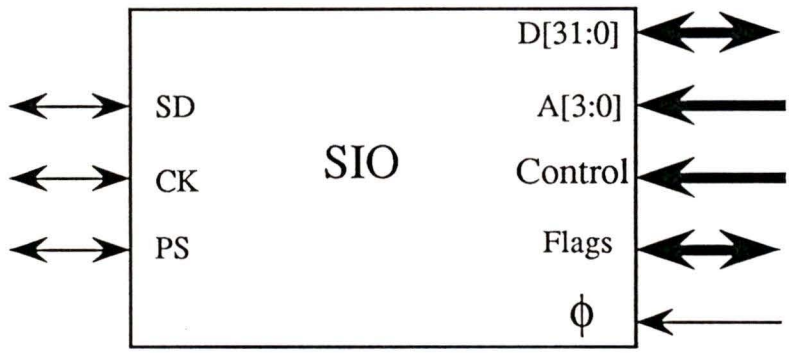
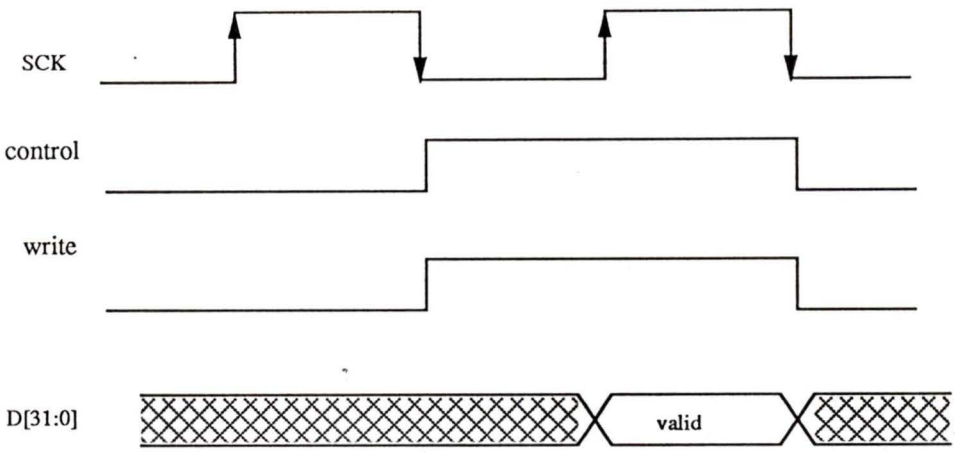
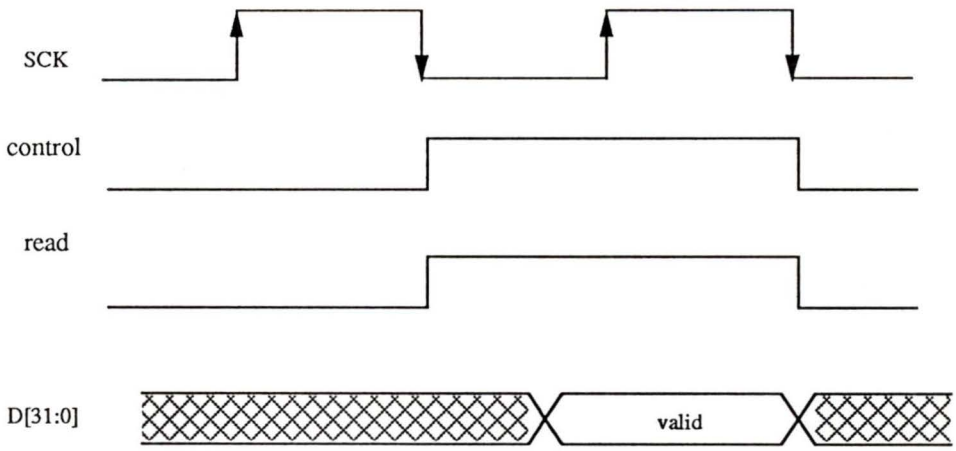


Figure 3.7: The SIO icon, showing different buses and control lines at its periphery.



(a)



(b)

Figure 3.8: A timing diagram for the SIO block read and write cycles.

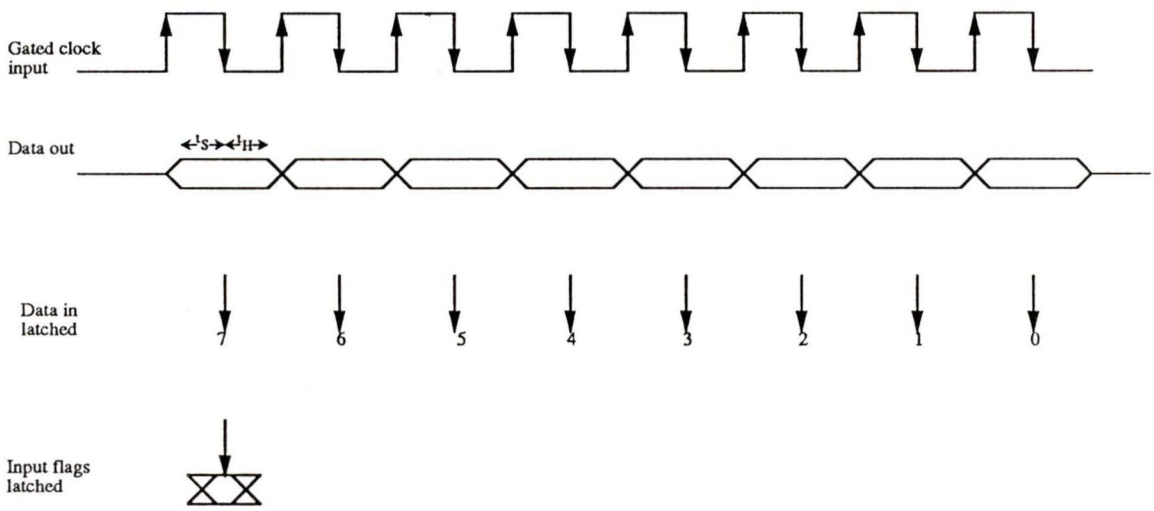


Figure 3.9: Timing diagram of port protocol for transmitting and receiving data (8-bit example).

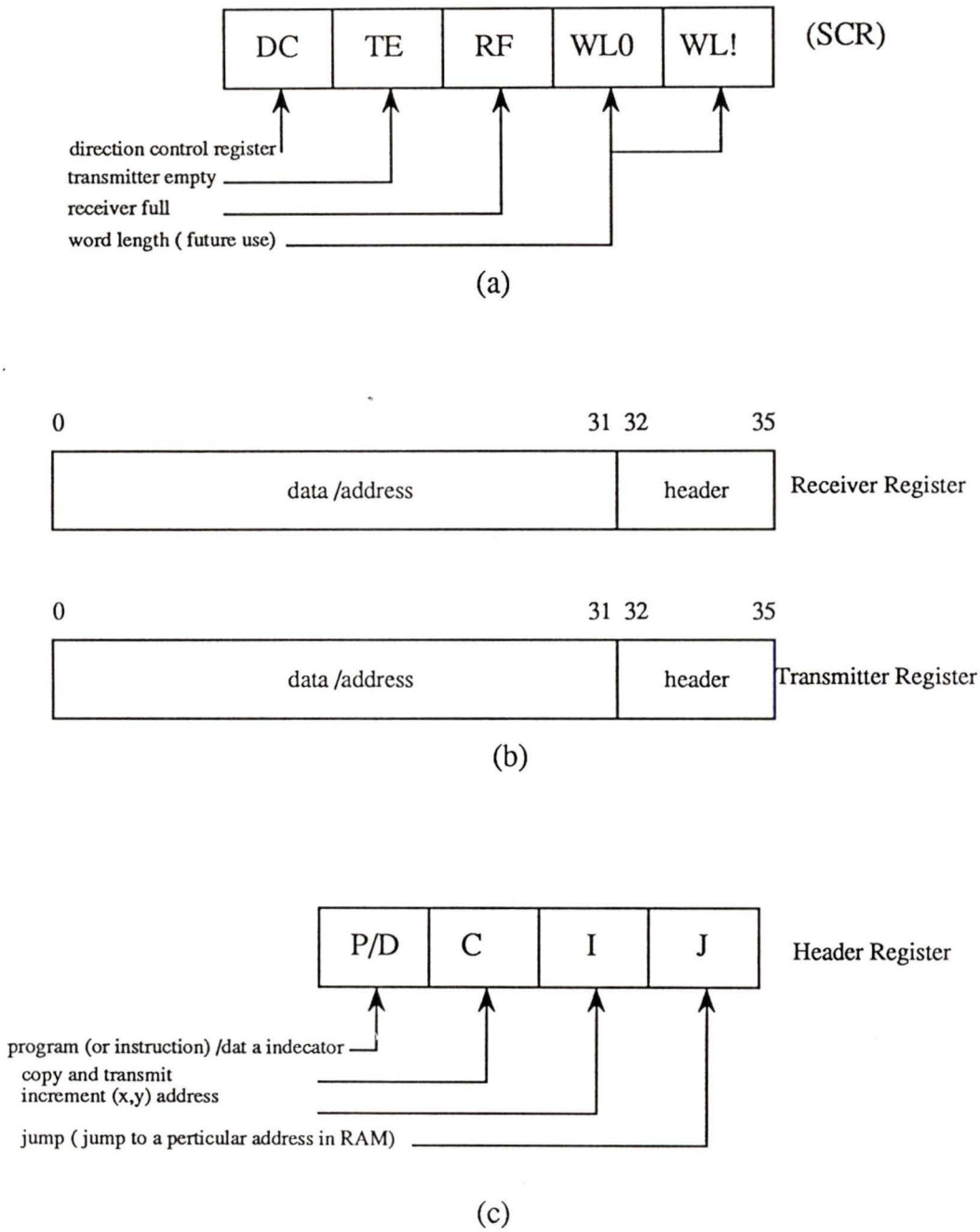


Figure 3.10: The status control register (SCR) and the function of each bit in SCR.

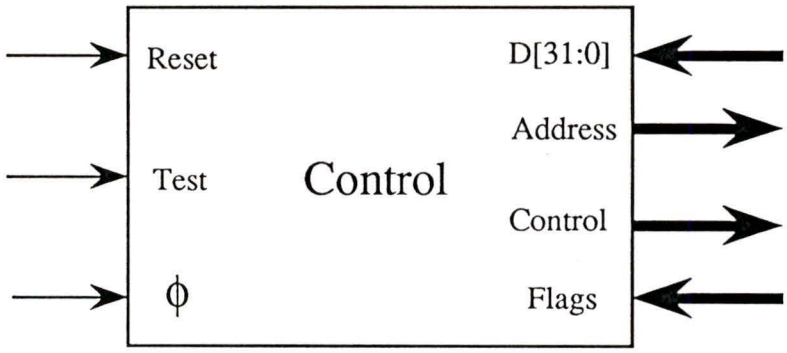
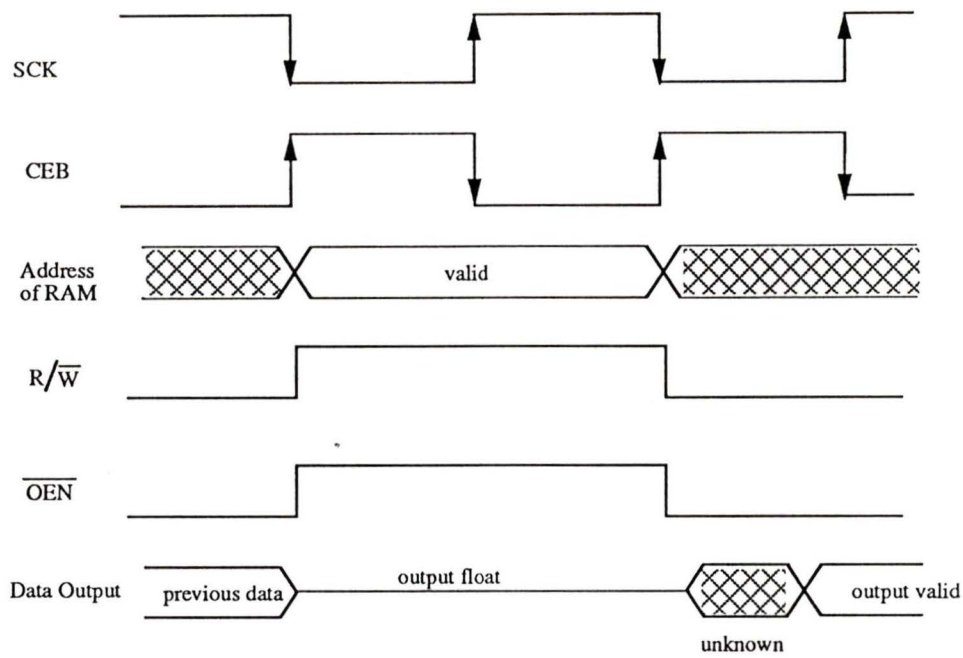
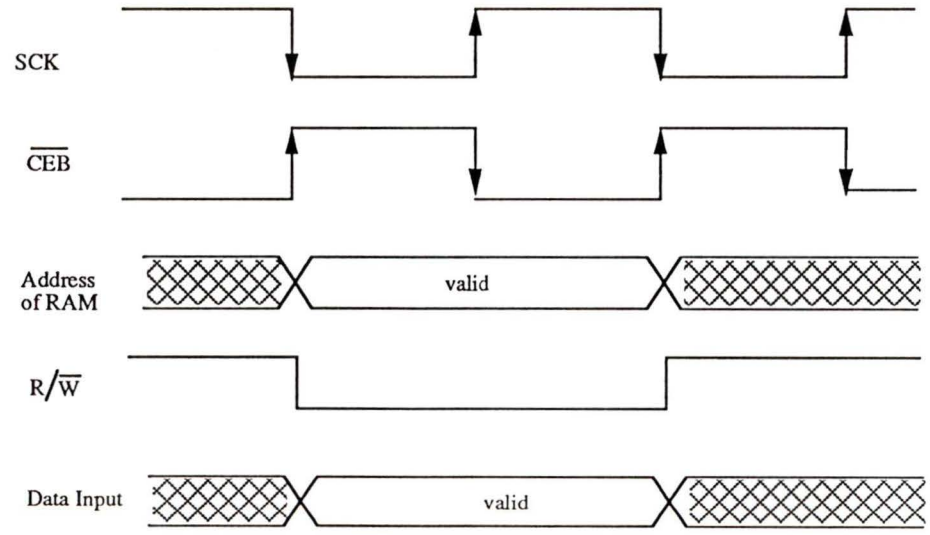


Figure 3.11: The controller icon, showing different buses and control lines at its periphery.



(a)



(b)

Figure 3.12: Timing diagram for the RAM (a) read and (b) write cycles.

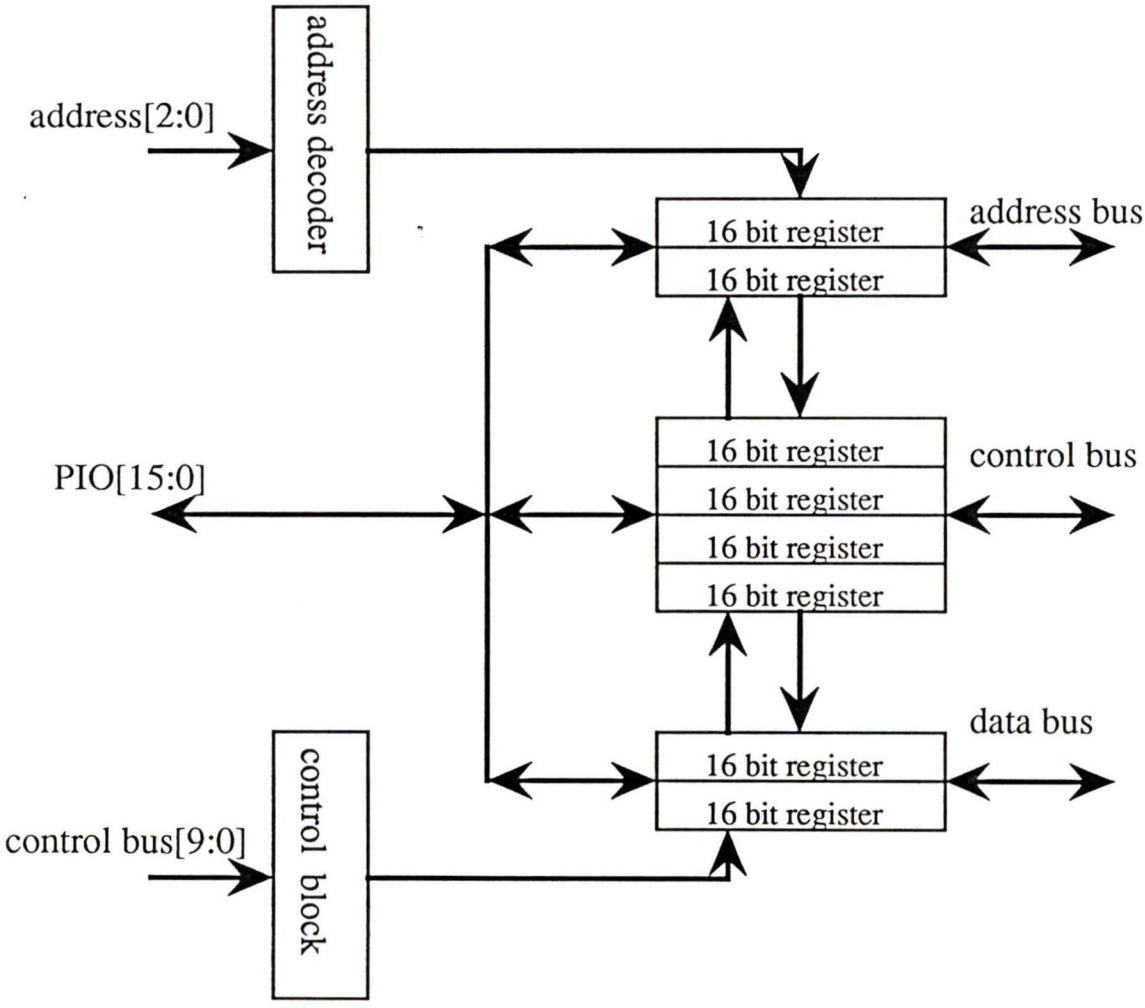


Figure 3.13: A proposed block diagram for PIO block to facilitate the test of the PE.

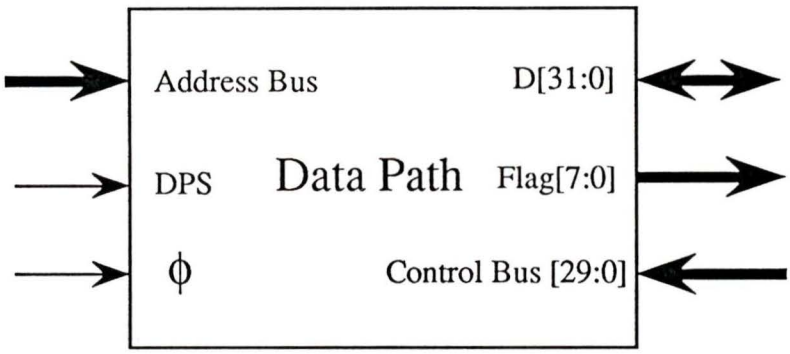


Figure 3.14: The datapath icon, showing different buses and control lines at its periphery.

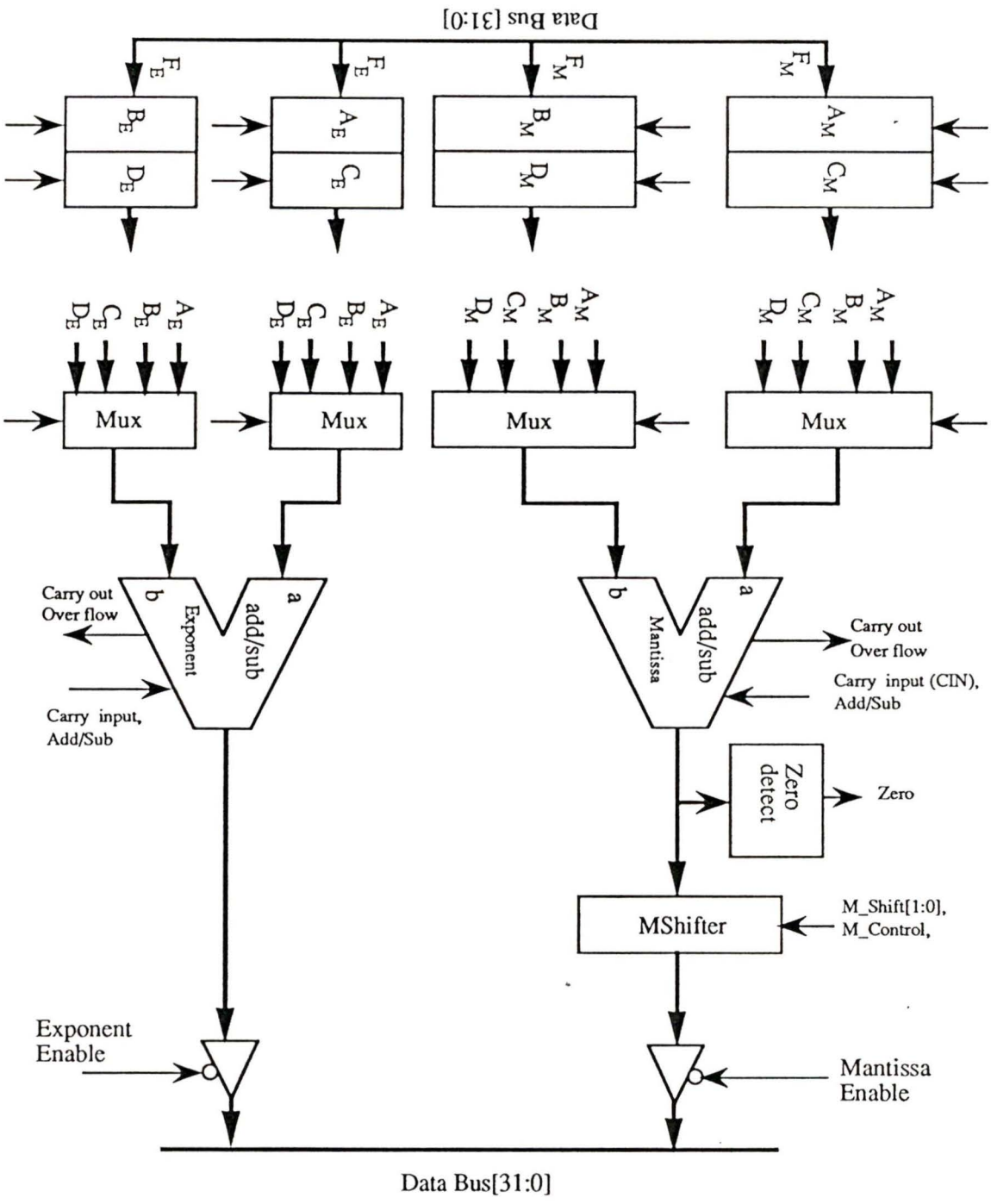


Figure 3.15: A proposed block diagram for floating-point datapath.

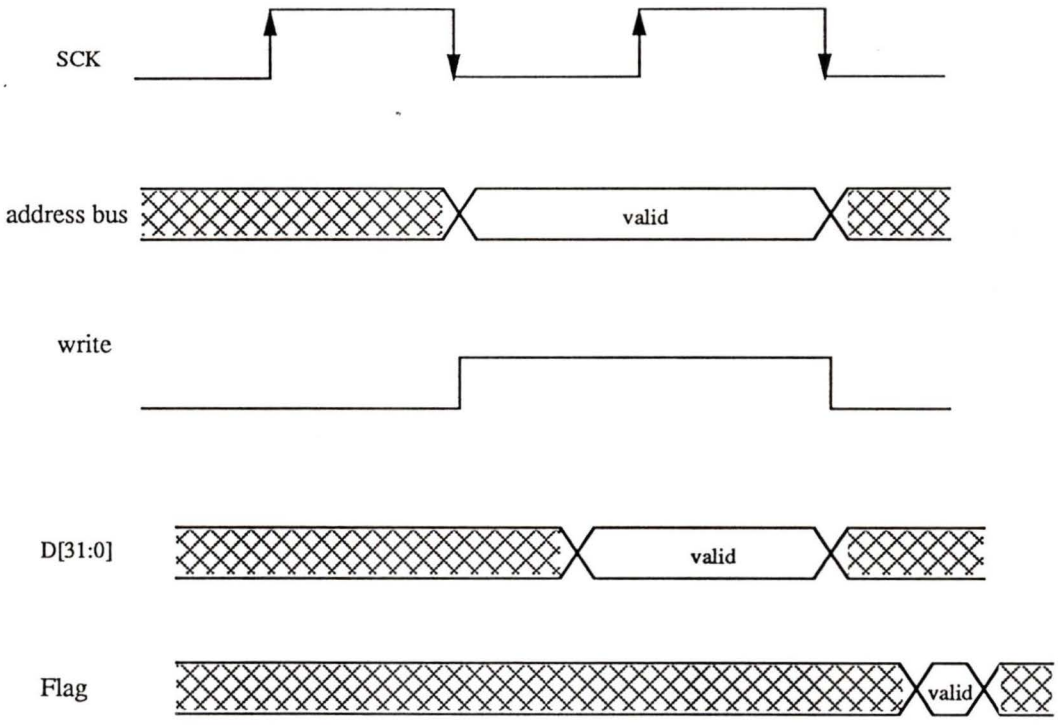


Figure 3.16: Timing diagram for latching the input data at datapath registers.

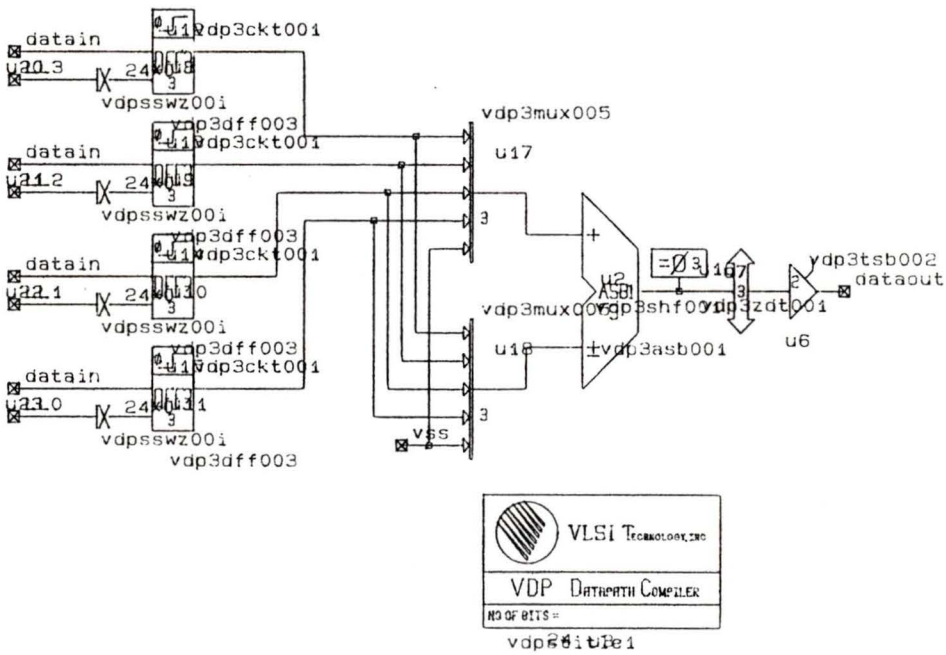


Figure 3.17 A 24-bit cell based block diagram for mantissa part of the data path.

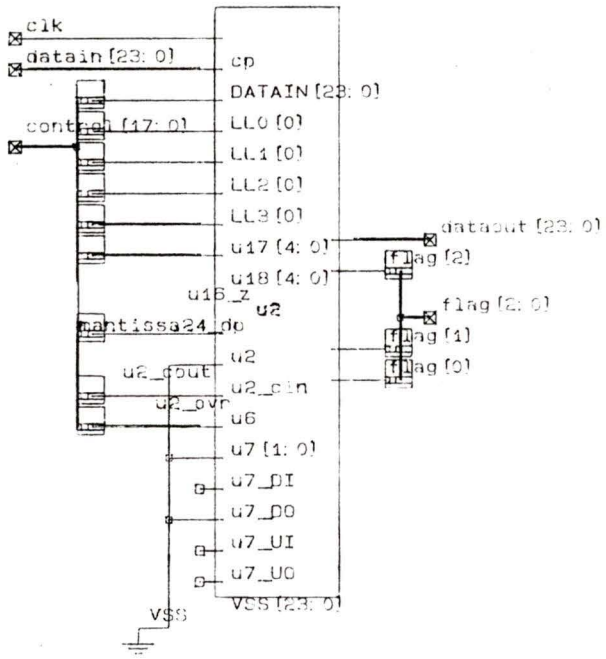


Figure 3.18: An icon showing the input, output and control connections for the mantissa block.

Chapter 4

An Application of CORDIC Processor Arrays

4.1 Introduction

The spectral decomposition of a matrix (i.e. evaluating its eigenvalues and eigenvectors) is of central importance in many scientific and engineering applications. Beamforming is a typical DSP algorithm that is very much suited for implementation using CORDIC processor arrays. A CORDIC processor is needed because the required arithmetic operations include multiplication, division and trigonometric functions. An array of such processors is required to speed up the response of the system. This type of parallel processor would be very beneficial to adaptive beamforming applications when the number of beam elements is large and speed of adaptation is essential. In beamforming the signals to or from the antenna elements are weighted. The optimal solution for the "weight vector" determines the best response of the antenna array. The optimal weight vector may be found using the QR decomposition method [28].

The QR decomposition of a matrix can be obtained by using either the Gram-

Schmit orthogonalization procedure or an orthogonal transformation (Givens rotations). The method we are using is based on Givens rotations to find the eigenvalues and eigenvectors of a matrix. Matrix triangularization is an essential step for the QR algorithm. The process of triangularizing a matrix is usually done using Givens rotations since this is a stable technique. By using this technique one can transform the matrix into an upper or lower triangular matrix. The CORDIC algorithm explained in Chapter 2 in conjunction with a proper systolic array explained in Chapter 3, can perform the triangularization operation and the QR algorithm.

Systolic arrays have been proposed as a suitable approach for implementing specialized processors. Systolic arrays consist of several processing elements connected in a nearest-neighbor fashion. Using massive pipelining, systolic arrays achieve high performance while keeping low I/O requirements. The regularity of signal processing algorithms is reflected in the physical structure of the systolic array.

The literature abounds with systolic array architectures to solve particular matrix problems. For example, reference [30] deals with the solution of sparse linear systems. Reference [31] deals with matrix transposition. References [32] and [33] deal with matrix triangularization. References [34], [35] and [36] deal with the eigenvalue extraction of a matrix. Reference [37] deals with the singular value decomposition of a matrix. Reference [38] deals with the least squares estimation problem. References [39], [40] and [41] deal with Kalman filtering problems. References [29] and [42] deal with matrix multiplication.

A systolic design for the spectral decomposition of a matrix based on the QR algorithm is proposed here. The basic feature of this technique is the evaluation of eigenvectors along with the eigenvalues. Only one type of processing elements

are used in the systolic array structure. The same systolic array is also suitable for solving systems of linear equations, least square estimation and matrix-matrix multiplication. In addition, the array can be used to perform other useful, low-level operations such as: plane rotations, pre-multiplication and post-multiplication of a given matrix by an orthogonal transformation.

It is crucial in the design of a systolic array to be able to handle matrices larger than the size of the hardware. A solution to this problem is partitioning. A large matrix is broken up into smaller ones matching the size of the systolic array. The partitioning strategy depends on the nature of the problem. The QR algorithm requires global communication of the data. This is handled by multiple passes of the partitioned matrix. Of course, performing the QR algorithm using hardware that does not match the size of the matrix incurs a performance penalty. The technique we use here could be termed systolic array emulation [35], [43]. In this technique, the systolic array represents a small segment of the larger virtual systolic array that matches the problem size.

4.2 Preliminaries

Givens Rotation

The Givens rotation technique is used to annihilate a selected element of a matrix. The Givens rotation matrix G_{pq} $(p > q)$ is shown in Fig. 4.1. The elements of the rotation matrix are the same as those of the identity matrix with the exception of the (p,p), (p,q), (q,p) and (q,q) elements, which are $\cos\theta$, $\sin\theta$, $-\sin\theta$ and $\cos\theta$, respectively. It is easy to prove, that the rotation matrix G_{pq} satisfies the relation

$$G_{pq}^T G_{pq} = 1 \tag{4.1}$$

which is the defining property of an orthogonal matrix. Premultiplication of a matrix A, with the rotation matrix G_{pq} affects only rows p and q of matrix A.

$$A' = G_{pq}\ A \tag{4.2}$$

$$a'_{ij} = a_{ij}, \qquad i \neq p, q \tag{4.3}$$

$$a'_{pj} = a_{pj}\ \cos\theta\ + a_{qj}\ \sin\theta \tag{4.4}$$

$$a'_{qj} = -a_{pj}\ \sin\theta + a_{qj}\ \cos\theta \tag{4.5}$$

According to (4.5), the element a'_{qj} is eliminated if the angle θ is chosen such that

$$\tan\theta = \frac{a_{qj}}{a_{pj}} \tag{4.6}$$

Postmultiplication of a matrix A, with the rotation matrix G_{pq} affects columns p and q of matrix A.

$$A' = A\ G_{pq} \tag{4.7}$$

$$a'_{ij} = a_{ij} \qquad j \neq p, q \tag{4.8}$$

$$a'_{ip} = a_{ip}\ \cos\theta - a_{iq}\sin\theta \tag{4.9}$$

$$a'_{iq} = a_{ip}\sin\theta + a_{iq}\cos\theta \tag{4.10}$$

According to (4.10), the element a'_{iq} is eliminated if the angle θ is chosen such that

$$\tan\theta = -\frac{a_{iq}}{a_{ip}} \tag{4.11}$$

The process of obtaining the angle θ is called generating the rotation, and the process of obtaining the new rows in (4.4), (4.5) and columns in (4.9), (4.10) is called applying the rotation.

Matrix Triangularization

Triangularization of a matrix into an upper triangular matrix involves reducing all the elements below the main diagonal to zero. Gaussian elimination can be used to factorize a matrix A to $A = LU$ where L is a unit lower triangular and U is upper triangular matrix. This technique requires pivoting however. To guarantee stable solutions, row reordering might be necessary. This is not very attractive from a systolic implementation point of view. On the other hand, the QR algorithm based on Givens' rotation [44] is stable and does not require pivoting [45]. For the QR algorithm

$$A = QR \tag{4.12}$$

where Q is an orthogonal matrix and R an upper triangular matrix. The matrix Q^T is given by

$$Q^T = Q_{n-1}Q_{n-2}\cdots Q_1 \tag{4.13}$$

where Q_p is the matrix to eliminate the elements $(p+1,p),(p+2,p),\cdots,(n-1,p),(n,p)$ of the matrix A.

$$Q_p = G_{p+1,p}G_{p+2,p}\cdots G_{n,p} \tag{4.14}$$

The upper triangular matrix R is obtained by pre-multiplying matrix A by a series of Givens' rotation matrices G, to produce the upper triangular matrix A'. For example, the first column is reduced to the vector $[a'_{11}, 0, \cdots, 0]^T$ by performing $n-1$ rotations using Givens matrices G_{i1} with $i = 2, ..., n$. The first four steps for zeroing the subdiagonal elements of the first column for a 5×5 matrix are shown in Fig. 4.2. Note that newly created zeros will not disturb the previously created zeros. For example in the first column, the zero in position (2,1) is not

disturbed by the transformation G_{j1}; where $j > 2$. The subdiagonal elements in the second column are zeroed by successive premultiplication with the matrices G_{i2}, $i = 3, 4, \cdots, n$. The newly created zeros will not disturb the zeros in the first column since the elements involved are already zero. Triangularizing a matrix using this technique is direct, rather than iterative, because the triangularization is achieved in a known number of steps.

Similarity Transformation

If there exists a nonsingular matrix P such that $P^{-1}AP = B$, then B is said to be similar to A. Similar matrices have the same characteristic polynomial and the same eigenvalues.

Suppose B is similar to A:

$$B = P^{-1}AP \tag{4.15}$$

if $\mathbf{x}$ is an eigevactor of A associated with the eigenvalue λ then the $P^{-1}\mathbf{x}$ is an eigenvactor of B associated with the eigenvalue λ.

4.3 The QR Algorithm

The QR algorithm is used to find the eigenvalues and eigenvectors of symmetric or non-symmetric matrices. This method is based on triangular decomposition of a matrix which can be achieved by applying a series of Givens rotations [46]. A sequence of matrices $A_1, A_2, \cdots$ can be computed which converge to an upper triangular matrix. The first stage in the QR factorization is the decomposition of the matrix $A = A_1$, i.e.

$$A_1 = Q_1 R_1 \tag{4.16}$$

where A_1 is the original matrix, R_1 is an upper triangular matrix and Q_1 is a orthogonal transformation matrix, given by

$$\begin{aligned} Q_1^T &= \left(G_{n,(n-1)}\right)\left(G_{n,(n-2)}G_{(n-1),(n-2)}\right)\cdots \\ &\quad (G_{n,2}\cdots G_{4,2}G_{3,2})\,(G_{n,1}\cdots G_{3,1}G_{2,1}) \end{aligned} \tag{4.17}$$

where G_{qp} is Givens rotation matrix to eliminate matrix element a_{qp}; $(q > p)$. A new matrix A_2 is formed according to

$$A_2 = R_1 Q_1 \tag{4.18}$$

Using 4.16 and 4.18 it can be easily proven that A_1 and A_2 are similar.

The whole process is then repeated and the general equations for the QR algorithm are:

$$A_s = Q_s R_s \tag{4.19}$$

$$A_{s+1} = R_s Q_s \tag{4.20}$$

where $s = 1, 2, \cdots$. In (4.19) each complete step is a similarity transformation because

$$A_{s+1} = R_s Q_s \tag{4.21}$$

$$= Q_s^{-1} A_s Q_s \tag{4.22}$$

So all the matrices have the same eigenvalues.

Eigenvalues and Eigenvectors

The eigenvalues and eigenvectors of a given matrix are defined by

$$A\mathbf{x} = \lambda\mathbf{x} \tag{4.23}$$

where A is a $n \times n$ matrix, λ is an eigenvalue and $\mathbf{x}$ an eigenvector. If the eigenvalues of matrix A are distinct then it has linearly independent eigenvectors. If we set

$$X = (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n) \tag{4.24}$$

and

$$\Lambda = diag(\lambda_1, \lambda_2, \cdots, \lambda_n) \tag{4.25}$$

It is easily verified that the (4.23) is equivalent to the single matrix equation

$$AX = X\Lambda \tag{4.26}$$

$$= [\mathbf{x}_1 \mathbf{x}_2 \cdots \mathbf{x}_n] \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \lambda_n \end{bmatrix} \tag{4.27}$$

or

$$A = X\Lambda X^{-1} \tag{4.28}$$

where X is the eigenmatrix for matrix A.

After N iterations A_N converges to an upper triangular matrix

$$A_N = \begin{bmatrix} \lambda_1 & x & x & x \\ 0 & \lambda_2 & x & x \\ 0 & 0 & \ddots & x \\ 0 & 0 & 0 & \lambda_n \end{bmatrix} \tag{4.29}$$

where $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$ are the eigenvalues of A.

A_N is related to A_1 by:

$$A_N = Q_{N-1}^T A_{N-1} Q_{N-1} \tag{4.30}$$

$$= Q_{N-1}^T (Q_{N-2}^T A_{N-1} Q_{N-2}) Q_{N-1} \tag{4.31}$$

$$\vdots \tag{4.32}$$

$$= Q_{N-1}^T Q_{N-2}^T \cdots Q_2^T Q_1^T A_1 Q_1 Q_2 \cdots Q_{N-2} Q_{N-1}$$

$$= P^{-1} A_1 P \tag{4.33}$$

Where the similarity transformation matrix P is given by

$$P = Q_1 Q_2 \cdots Q_{N-2} Q_{N-1} \tag{4.34}$$

which is a similarity transformation.

Eigenvector Evaluation

If matrix A_1 has an eigenvalue λ_i corresponding to the eigenvector $\mathbf{x}_i$, then $P^{-1}AP$ has the same eigenvalue as A_1 but with eigenvectors $P^{-1}\mathbf{x}_i$ [44].

If B_N is the eigenmatrix for matrix A_N, then B_N is given by

$$B_N = \left[\begin{array}{cccc} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_n \end{array} \right] = \left[\begin{array}{cccc} 1 & b_{12} & \ldots & b_{1n} \\ 0 & 1 & \ldots & b_{2n} \\ \vdots & & & \vdots \\ 0 & 0 & \ldots & 1 \end{array} \right] \tag{4.35}$$

where $b_{ii} = 1; \forall \ 1 \leq i \leq n$ and $b_{ij} = 0; \ i > j$

Each of the elements b_{ij} of the eigenvector $\mathbf{b}_i$ is evaluated according to the equation

$$b_{ij} = -\frac{1}{\lambda_i - \lambda_j} \sum_{k=i+1}^{j} a_{ik} b_{kj} \tag{4.36}$$

The eigenmatrix X_1 of matrix A_1 is related to B_N according to the equation [44]:

$$X_1 = P_N B_N = Q_1 Q_2 \ldots\ldots Q_{N-1} B_N \tag{4.37}$$

4.4 System Design and Operation

Our objectives in this section are to derive a systolic architecture to implement the QR algorithm as summarized in (4.19) and (4.20). We also want to be able to use the same architecture to extract the eigenvectors of a given matrix.

The QR algorithm can be written in a sequential program. The sequential program for QR algorithm is as follows:

For k from 1 to N

  For i from k to $N-1$

    $\theta \leftarrow \tan^{-1}(\frac{a_{i+1,k}}{a_{i,k}})$

    For j from k to N

      $temp1 \leftarrow a_{i,j}\cos\theta + a_{i+1,j}\sin\theta$

      $temp2 \leftarrow -a_{i,j}\sin\theta + a_{i+1,j}\cos\theta$

      $a_{i,j} \leftarrow temp1$

      $a_{i+1,j} \leftarrow temp2$

    end For

  end For

end For

The signal flowgraph for the above QR algorithm is shown in Fig. 4.3 [28]. In Fig. 4.3 the scheduling of node activities is set by the scheduling vector **s**. We chose the scheduling vector $\mathbf{s} = [1 \;\; 1 \;\; 1]$. The choice of the orientation of **s** is such that

$$\mathbf{s}.\mathbf{e} > 0 \tag{4.38}$$

where **e** is any arc in the graph [28]. Inequality (4.38) ensures that casuality is enforced. For example, if node p depends on node q, then the time step assigned

for p cannot be less than the time step assigned for q.

The assignment of the nodes in Fig. 4.3 to the processors in the systolic array is set by the projection vector **d**. We chose the projection vector $\mathbf{d} = [0 \quad 1 \quad 0]$. The choice of the orientation of **d** is such that

$$\mathbf{d}.\mathbf{s} > 0 \tag{4.39}$$

Inequality (4.39) ensures that the nodes on an equitemporal hyperplane are not projected to the same processing element. It is clear from the signal flowgraph that after projecting in the direction **d** a triangular array is obtained.

The QR algorithm is implemented on the triangular array of Fig. 4.4. The array effects one step in the QR iterations. At each step in the QR iteration an input matrix is triangularized by premultiplication with a succession of Givens rotations. Each processor in the array performs one Givens rotation operation on the input matrix.

In Fig. 4.4 the numbers labeling the processors represent the matrix element to be eliminated. Input data elements are fed from the left side of the array while the resulting output matrix leaves the array from the top. Each processor in the systolic array first generates the angle of rotation and then performs the rotation on the pertinent rows of the matrix.

Note that the Givens rotation angles are extracted at each processor when the relevant elements arrive. For example, the processor labeled 21 in Fig. 4.5(a) extracts the angle θ_{21} upon receiving the elements a_{11} and a_{21}. After θ_{21} is obtained, all the elements of the first and second rows are rotated by this angle. The timing scheme of Figs. 4.4 and 4.5 ensures the correct sequence of operations. Figure 4.5(a) shows the sequence of arrival of elements to the first column of the systolic

array. At each time instant two elements are received and two outputs are produced. The rows are delayed two processing cycles relative to each other. There is a one cycle delay between the elements of the same row. Figure 4.5(b) shows the sequence of arrival of elements to the second column of the array. The sequence of element arrival to the second column is identical to that of the first column.

It should be noted from Fig.4.4 that feeding in a matrix in a row-wise fashion results in an output matrix equal to the input matrix premultiplied by the transpose of the transformation matrix stored in the array. The output matrix is obtained one column at a time.

The proposed triangular array performs useful operations in addition to triangularizing a matrix. These extra operations allow us to perform the spectral decomposition of the matrix as will be explained below. If a matrix A is fed to the array as in Fig. 4.4, the output matrix is $R = Q^T A$ according to (4.16). Where Q^T is defined in (4.17). Feeding in a unit matrix to the systolic array allows us to obtain the orthogonal transformation matrix Q^T.

After the first iteration, the systolic array output is matrix R_1, where R_1 is given form (4.17) as

$$R_1 = Q_1^T A_1 \tag{4.40}$$

At iteration k the systolic array has to perform the following set of operations in sequence:

1. Premultiply the input matrix by the transpose of the stored orthogonal matrix according to

$$R_k = Q_k^T A_k \tag{4.41}$$

This is achieved by feeding the rows of the input matrix in the fashion de-

picted in Fig. 4.4.

2. Postmultiply matrix P_{k-1} by the stored orthogonal matrix according to

$$P_k = P_{k-1}Q_k \tag{4.42}$$

This is achieved by feeding the input matrix in a manner similar to Fig. 4.4 except the input matrix is transposed first. Each processor receives one column of the input matrix.

3. Postmultiply matrix R_k by the stored orthogonal matrix according to

$$A_{k+1} = R_kQ_k \tag{4.43}$$

This is achieved by feeding the input matrix in a manner similar to Fig. 4.4 except the input matrix is transposed first. Each processor receives one column of the input matrix.

The above sequence of events is explained in Fig. 4.6. During the first iteration an orthogonal matrix Q_1 is generated. The output matrix R_1 is obtained according to (4.41).

There is a need to obtained the transformation matrix, Q_1. Since, Q_1 is at present inside the array, a unit matrix is used to flush it out. This is done by feeding a unit matrix. The output matrix which is Q_1 is stored for future use. The matrix present in the systolic array remains as Q_1 and is unaffected by the unit matrix multiplication. The matrix R_1 which was obtained in the previous step is fed column-wise into the systolic array containing the transformation matrix Q_1. Accordingly matrix A_2 is obtained.

Step 2 is similar to step 1 except for the fact that the input matrix is now A_2 instead of A_1. As in step 1, matrix A_2 is fed row-wise into the array to yield

the new matrix R_2. The previously obtained transformation matrix, Q_1 is now fed column-wise into the array to yield a product matrix Q_1Q_2 which is stored for use in future steps. Matrix A_3 is generated by feeding matrix R_2 in column-wise fashion into the array.

The above mentioned steps are repeated till the criterion for convergence is satisfied. At iteration N the diagonal elements of matrix R_N are the eigenvalues of matrix A_1.

It is however, to be noted that due to the intermediate procedure of multiplying and storing all the orthogonal transformation matrices, $Q_i's$, we are finally left with the product $P_N = Q_1Q_2.....Q_{N-1}$ after N iterations. This product matrix is very helpful in estimating the eigenvectors corresponding to the eigenvalues obtained. This is evident from (4.37). Having found P_N, matrix A_N is fed and stored in the systolic array. The elements of A_N are stored as shown in Fig. 4.7. A signal flowgraph for obtaining the eigenvectors is shown in Fig. 4.8. The elements of the eigenmatrix B_N associated with A_N are evaluated according to (4.36). Once the values of all the b_{ij}'s are calculated, the value of X_1 is obtained from (4.37). Figure 4.9 shows the signal flowgraph for obtaining the matrix X_1.

For triangularization of matrices having dimensions larger than the size of the systolic array, the following procedure is used and is also graphically shown in Fig. 4.10. Assuming the size of the array to be m on a side. Such an array can triangularize a matrix of maximum dimension $(m+1) \times (m+1)$. To operate on larger matrices, the matrix is divided into m horizontal strips. These strips are fed in the order shown in Fig. 4.10. It should be noted that the last row in each strip is the nth row. In this way the zeros are created in the columns below the main diagonal.

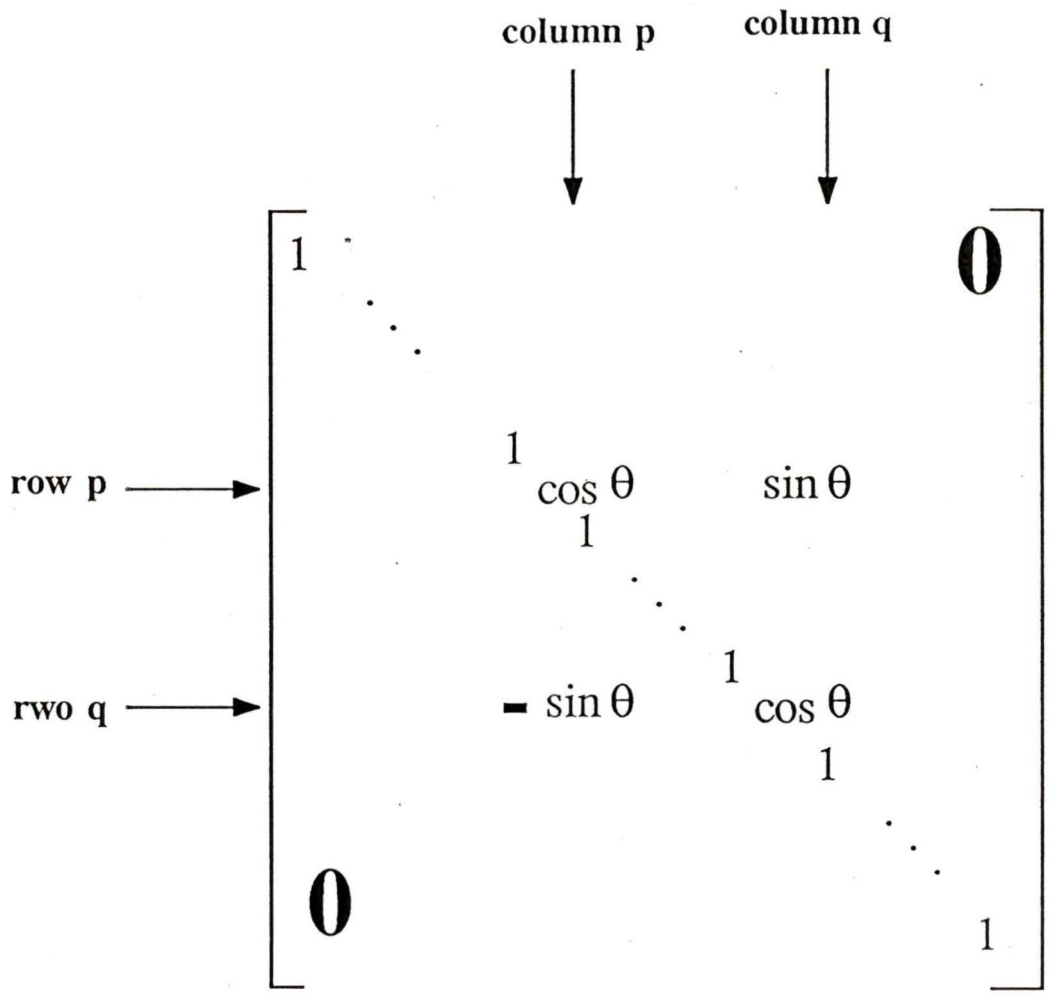


Figure 4.1: A Givens rotation matrix with nonzero off-diagonal elements at positions (p,p), (p,q), (q,p) and (q,q).

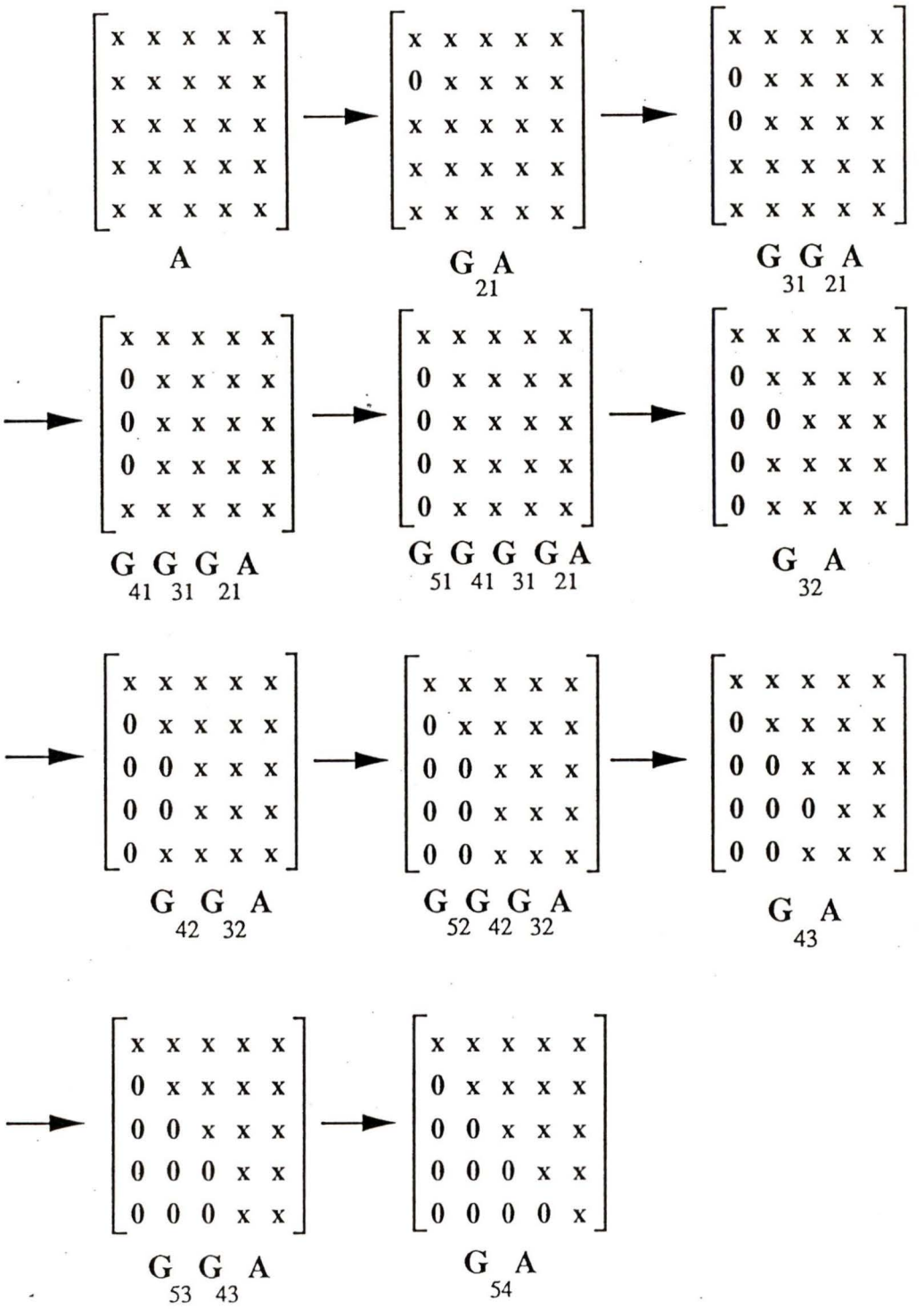


Figure 4.2: Premultiplication of a matrix A by a succession of Givens rotation matrices G, to make it upper triangular matrix.

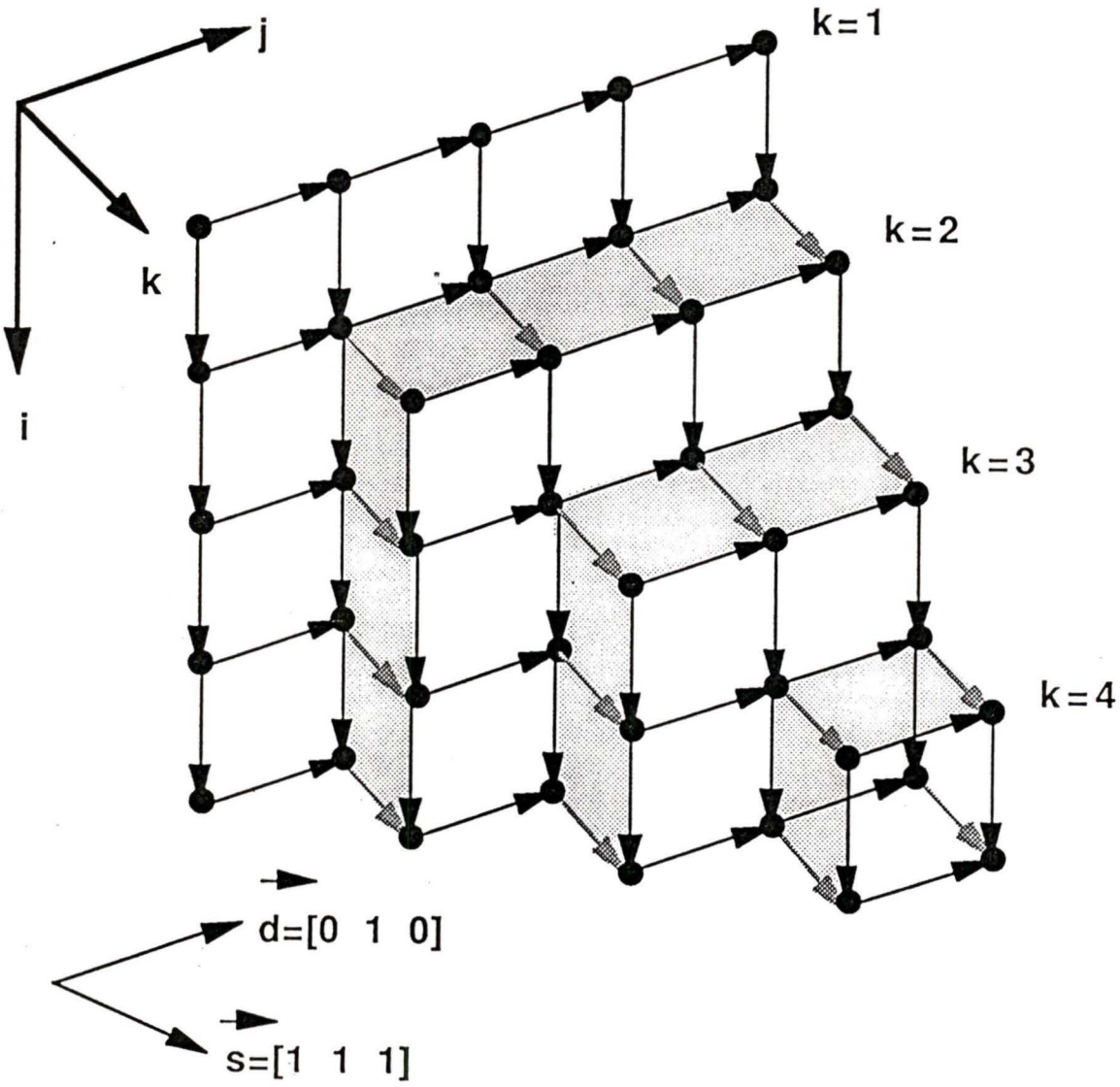


Figure 4.3: A three dimensional signal flow graph for the QR algorithm.

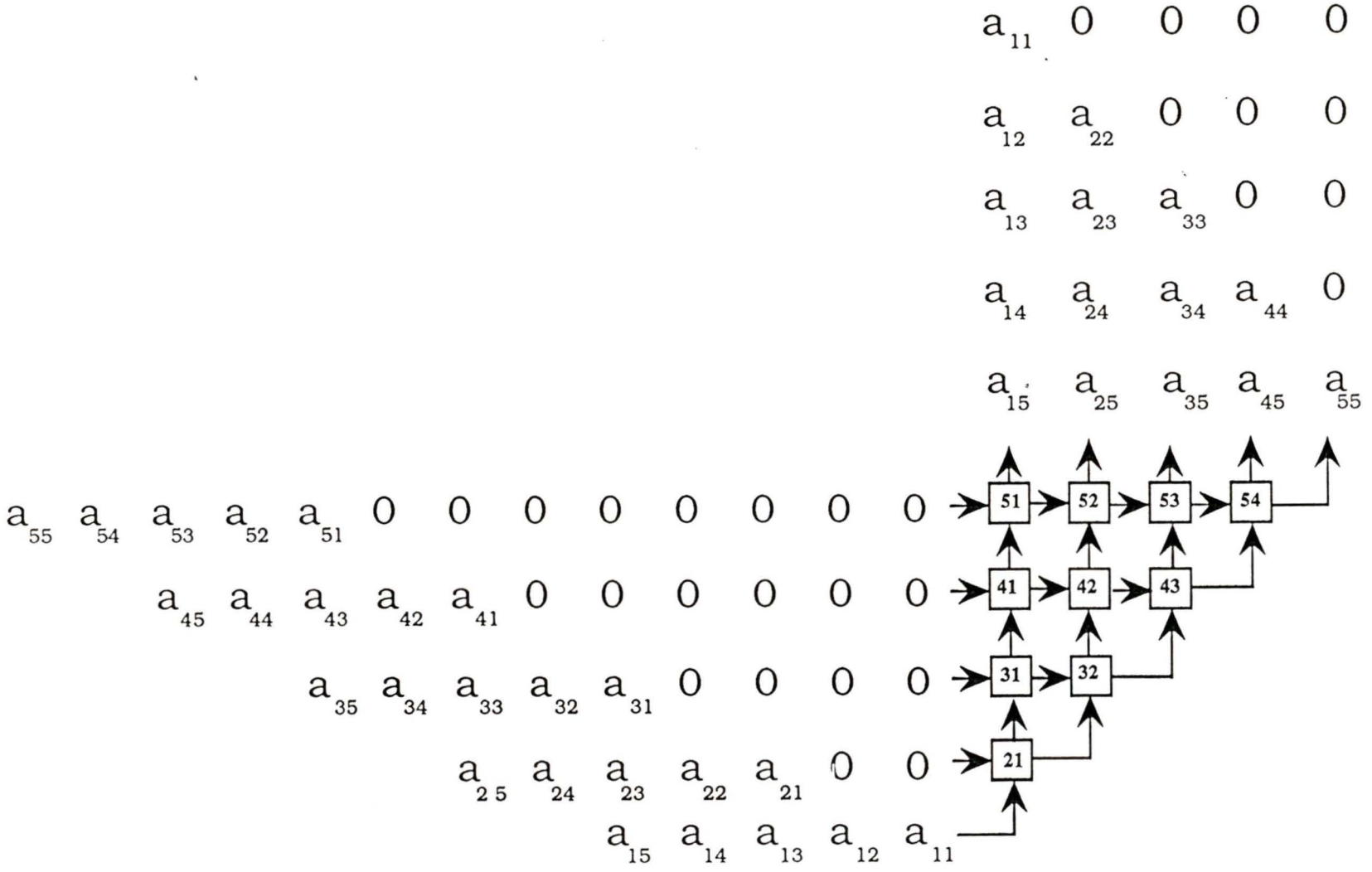


Figure 4.4: A systolic array for the triangularization of a 5×5 matrix by premultiplication with a succession of Given rotations.

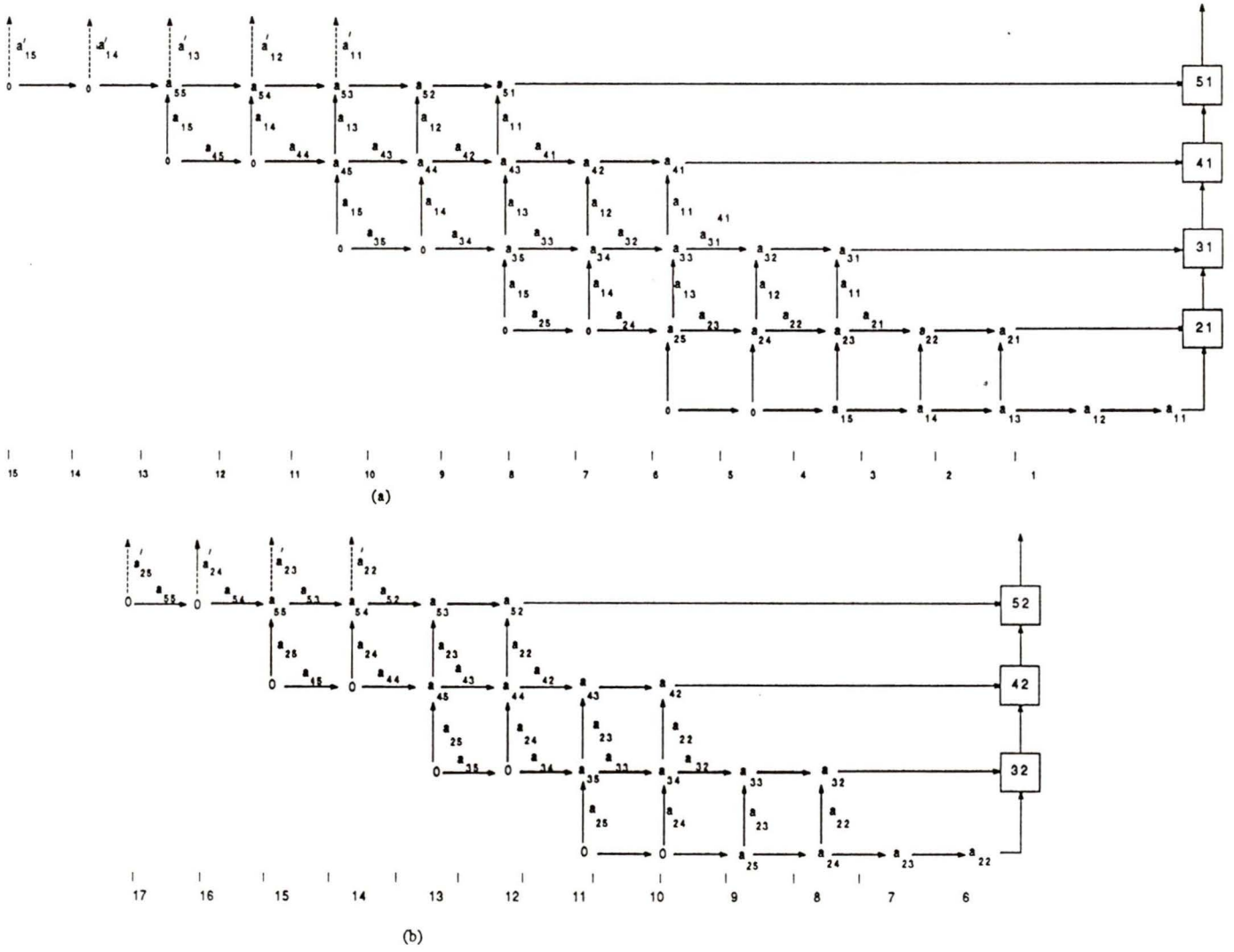


Figure 4.5: Space-time chart for the processors: of the first two columns of the array of Fig. 4.4. (a) The processors 21, $\cdots$, 51 of the first column. (b) The processors 32, $\cdots$, 52 of the second column.

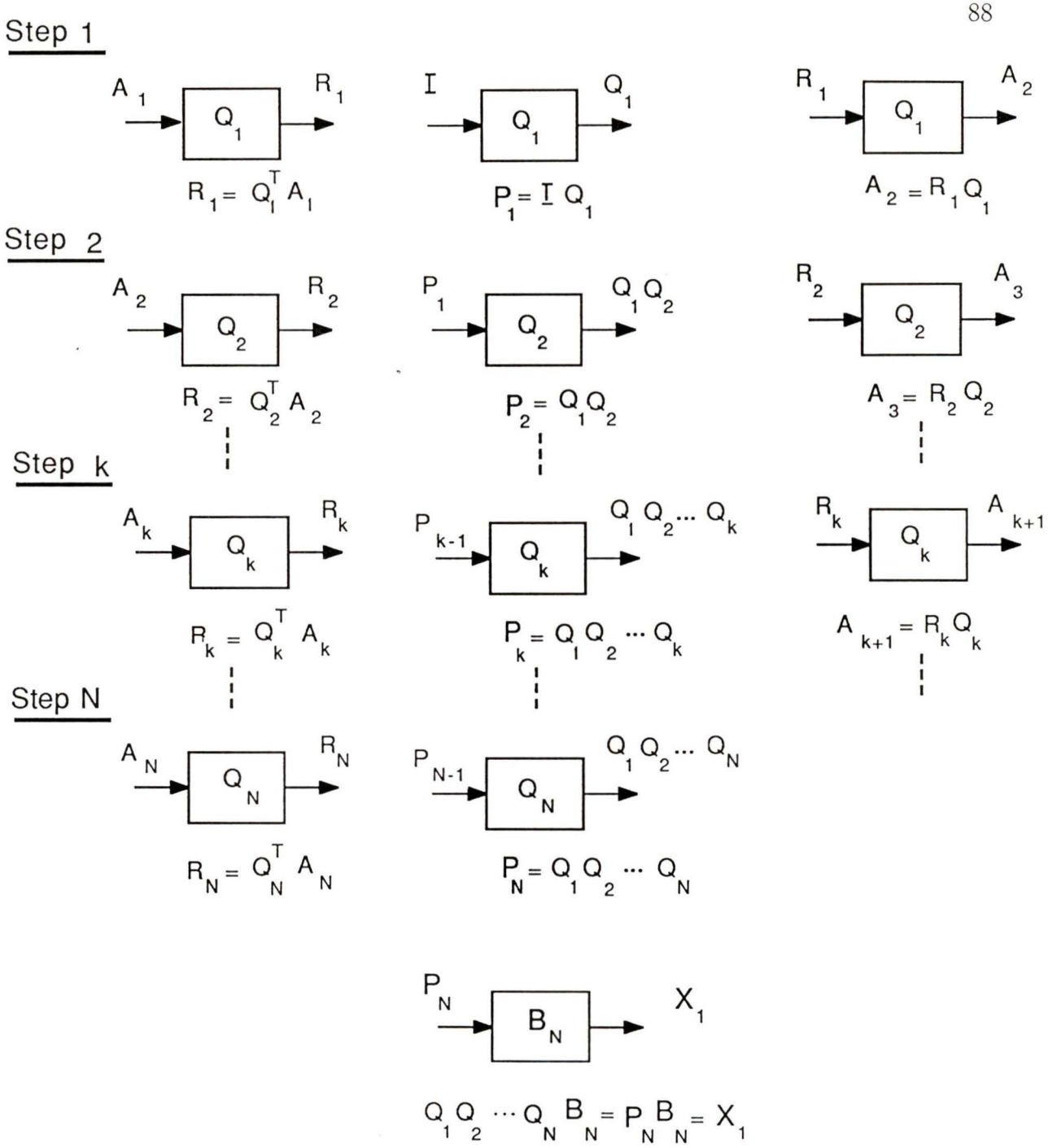


Figure 4.6: The operation of the system to find the eigenvalues and eigenvectors of a given matrix A.

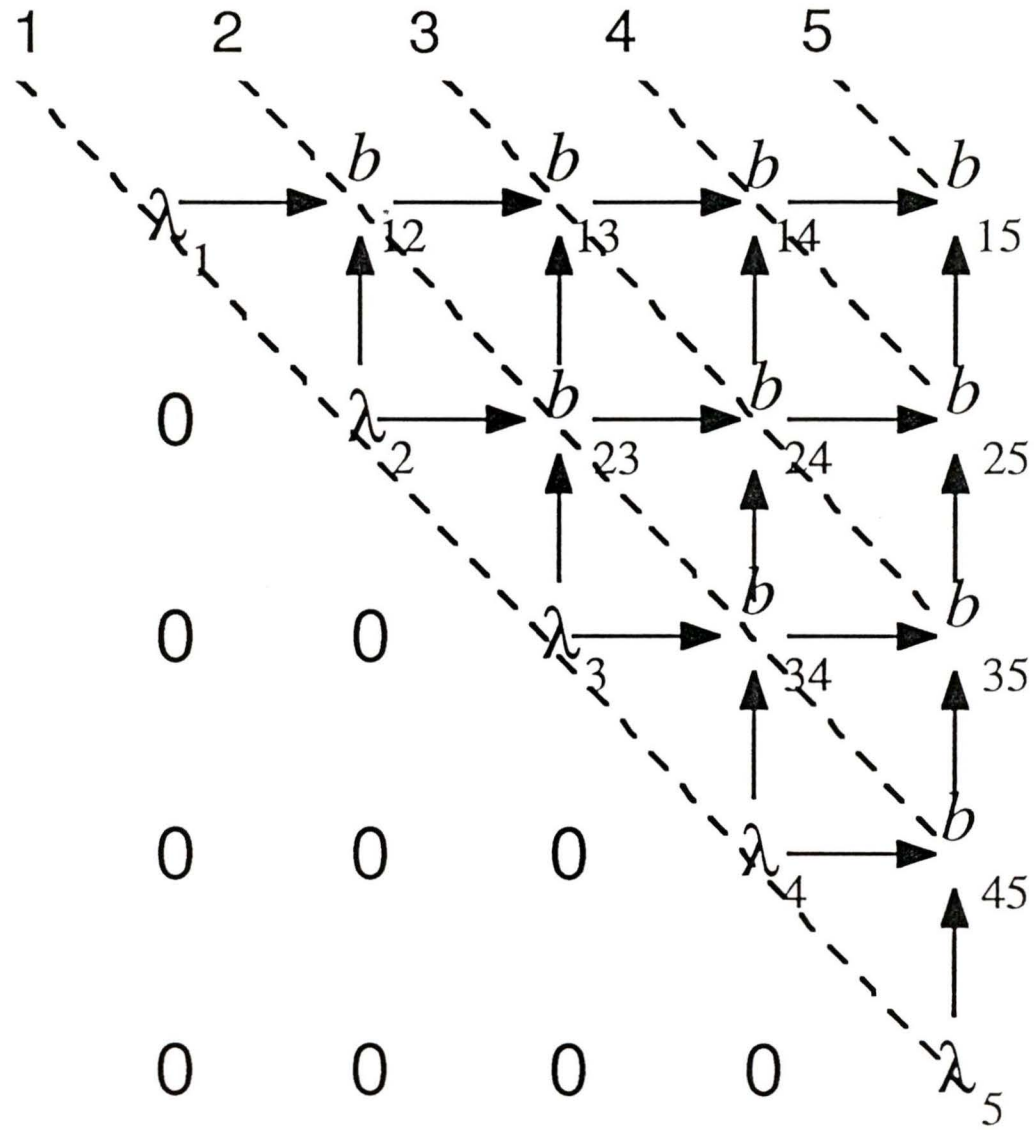


Figure 4.7: It is suppose that after the final iteration the upper triangular matrix is left inside the array. In a similar way the triangular matrix can be fed externally as shown in the above figure.

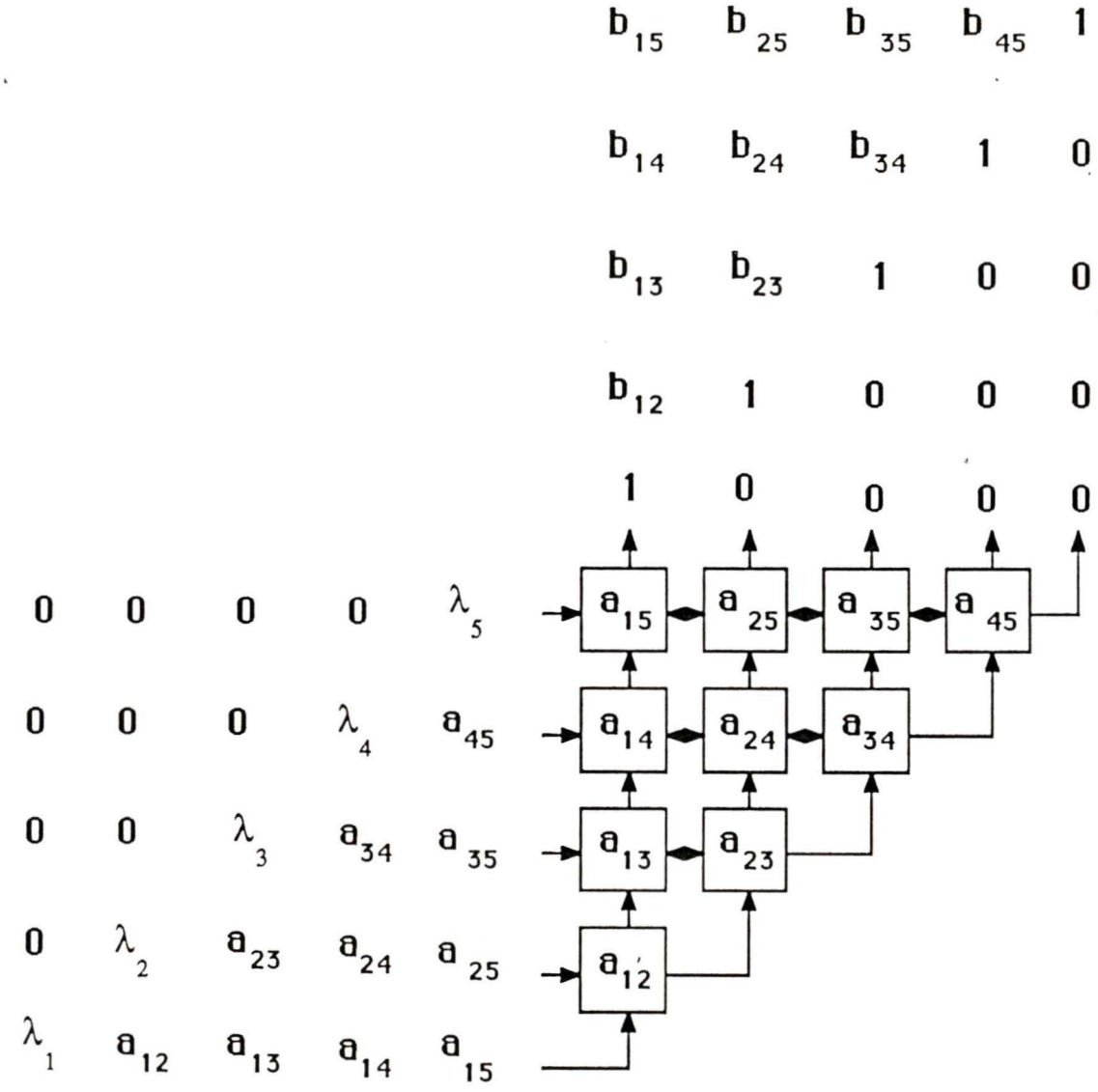


Figure 4.8: A signal flowgraph for the eigenvectors.

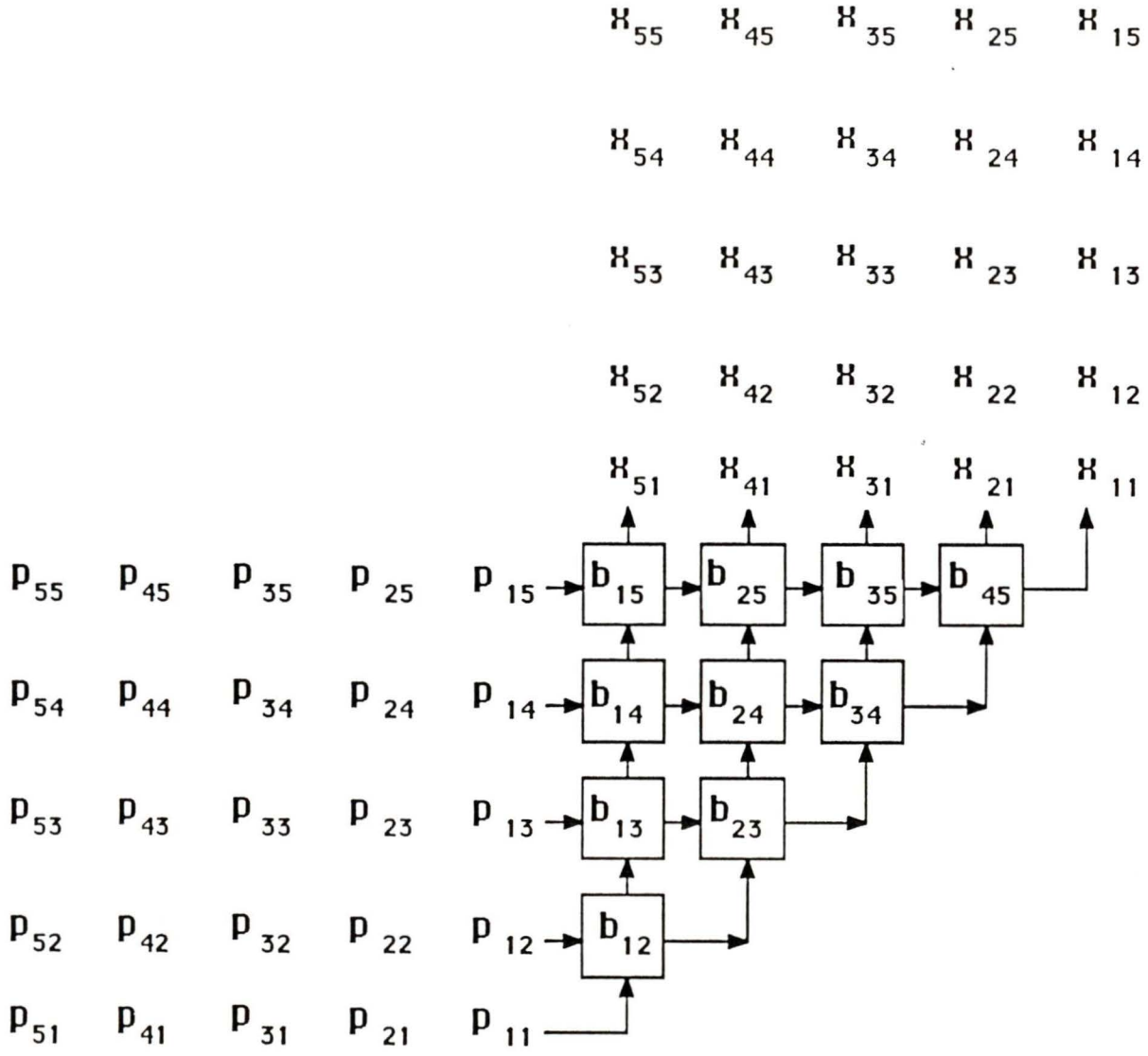


Figure 4.9: The final matrix X, is spelled out of the array.

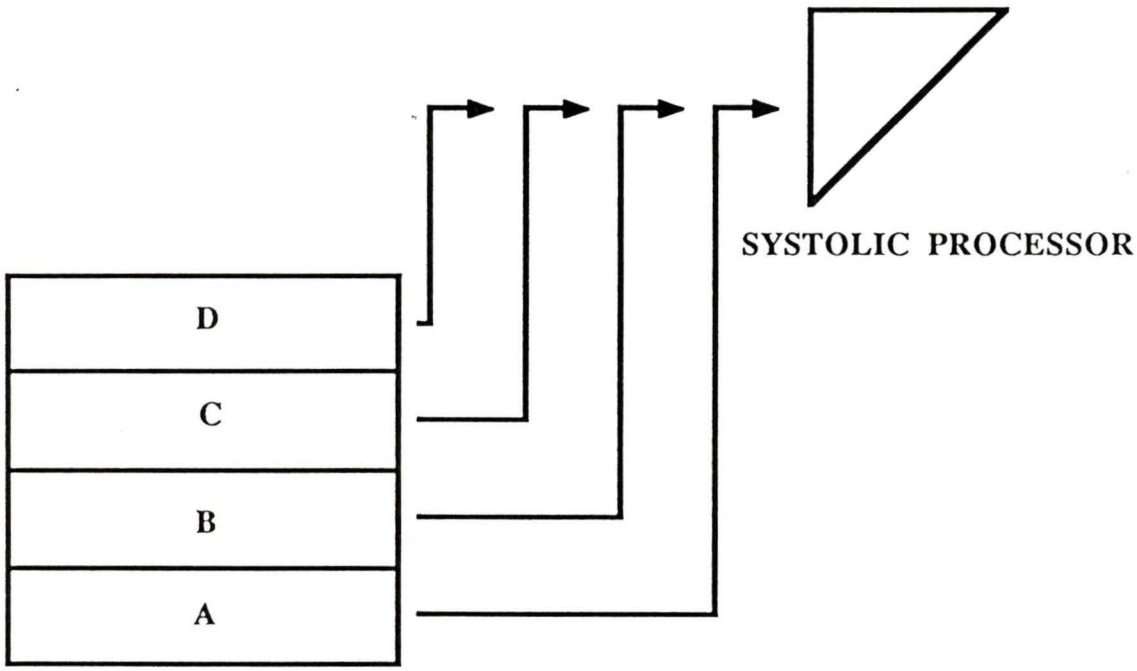


Figure 4.10: The partioning of a large square matrix to be triangularized on a systolic array of smaller dimension.

Chapter 5

Conclusions

The objectives of the research were to study the CORDIC algorithm, design a CORDIC processor suitable for parallel computing and to investigate a typical DSP application requiring such an array.

The major contributions of this thesis are as follow:

a. The CORDIC algorithm and its three basic modes of operations, circular, linear and hyperbolic were outlined. Considerations for the implementation of the CORDIC algorithm such as table look up requirements, the number of iterations for the fixed-point and floating-point cases and the noise performance were discussed. For table look up requirements, it was found that for 32-bits fixed-point data 11 table entries of the rotation angles are sufficient. Similarly, it was found that for 32-bits floating-point data, 12 table entries are sufficient. The number of iterations for the fixed-point CORDIC case is found to be equal to the number of bits in the data format. For the floating-point case the number of iterations is equal to the number of bits in the mantissa. The results of hardware simulation were also presented.

b. System requirements for the systolic processing element were identified. A

systematic approach to implement the system requirements was followed. Based on the system requirements, system specifications were identified. The VTI CAD package was used to design and test the datapath block of the CORDIC processor. The PE structure, at the block level, was explained. The pin connections and timing diagrams of the PE were also discussed.

The PE node consists of six major blocks. The operation of each block was discussed. The PE can communicate with its four nearest neighbors through four serial I/O ports. The direction of data traffic of the serial port is under program control. A parallel I/O block was also provided to facilitate testing of the PE.

c. A typical application requiring parallel processing was considered. Implementation of the QR matrix decomposition onto a systolic array was investigated. A signal flowgraph approach was used for the mapping of the QR algorithm onto the systolic array. The same systolic array structured allowed us to extract the eigenvalues and eigenvectors of matrix A.

Suggestions for future research

a. Investigate combining the CORDIC algorithm with other technique for faster evaluation of the elementary functions.

b. More efficient hardware design is required to enhance the processing speed and to increase integration level.

c. Several aspects of ystolic array design should be further investigated such as: Fault-tolerance, configurability, control strategy, communication protocol, and system -level integration.

Bibliography

[1] J. Volder, "The CORDIC trigonometric computing technique," in *IRE Trans. on Electronic Computers*, vol. EC-8, pp. 330–334, Sept. 1959.

[2] J. Walther, "A unified algorithm for elementary functions," in *Proc. of the 1971 Spring Joint Computer Conference*, pp. 379–385, 1971.

[3] H. Ahmed, *Signal Processing Algorithms and Architectures.* PhD thesis, Stanford University, 1982.

[4] G. Haviland and A. Tuzynski, "A CORDIC arithmetic processor chip," *IEEE Trans. Comput.*, vol. c-29, pp. 68–79, Feb. 1980.

[5] A. Naseem and P. Fisher, "A modified CORDIC algorithm," in *In. Proc. ICCD Conf.*, pp. 684–688, May 1984.

[6] T. Curtis, P. Allison, and J. Howard, "A CORDIC processor for laser trimming," *IEEE Micro*, vol. 6, pp. 61–71, July 1986.

[7] C. G. Lee and P. Chang, "A maximum pipelined CORDIC architecture for inverse kinematic position computation," *IEEE J. Robotics and Automation*, vol. RA-3, pp. 445–458, Oct. 1987.

[8] A. Despain, "Fourier transformation computers using CORDIC iterations," in *IEEE Trans. Comput.*, Oct. 1974.

[9] H. Ahmed and M. Morf, "Synthesis amd control of signal processing architectures based on rotations," in *First International Conferance on VLSI*, (Edinburgh), pp. 43–52, Aug. 1981.

[10] H. Ahmed, "Alternative arathmetic unit architectures for VLSI digital signal processors," in *VLSI and Modern Signal Processing* (S. Kung, H. Whitehouse, and T. Kailath, eds.), pp. 277–303, Englewood Cliffs: Prentice-Hall, 1985.

[11] H. Ahmed, J. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," in *IEEE Computer Magazine*, pp. 65–82, Jan. 1982.

[12] R. Harber, J. Li, and S. Bass, "Bit-serial CORDIC circuit for use in a VLSI compiler," in *In Proc. IEEE Int. Symp. Circuits and Systems*, (Portland, OR), pp. 154–157, 1989.

[13] H. Li and R. Jayakumar, "VLSI implementation of the CORDIC algorithm for trignometric functions," in *In Proc. Can. Conf. VLSI*, (Toronto), pp. 30–33, Oct. 1985.

[14] G. Vaudin and G. Nudd, "3um VLSI processing element using the CORDIC algorithm," *Electronics Letters*, vol. 23, pp. 1164–1166, Oct. 1987.

[15] E. Deperettere, P. Dewilde, and R. Udo, "Pipelined CORDIC architectures for fast VLSI filtering," in *In Proc. IEEE ICASSP*, (San Diego), pp. 41A.6.1–41A.6.4, 1984.

[16] T.-Y. Sung, Y.-H. Hu, and H. Yu, "Doubly pipelined CORDIC array for digital signal processing algorithm," in *In Proc. ICASSP'86*, (Tokyo), pp. 1169–1172, 1986.

[17] F. El-Guibaly, "Elec 649b," *Class notes*, 1989.

[18] Y. H. Hu and S. Naganathan, "Angle recording method for efficient implementation of the CORDIC algorithm," in *ISCAS*, Feb. 1989.

[19] B. Char, K. Geddes, G. Gonnet, M. Monagan, and S. Watt, *MAPLE.* Canada: WATCOM Publications, 1988.

[20] S. Morton, *A Fault Tolerant Bit Parallel Cellular Array Processor.* Reprent, ITT Advance Technology Center, 1984.

[21] R. Davis and D. Thomas, "Systolic array chip matches the pace of high speed processing," in *Electronic Design*, pp. 207–218, Oct. 1984.

[22] M. Homewood, D. May, D. Shepherd, and R. Shepherd, "The IMS T800 transputer," *IEEE Micro*, pp. 10–26, Oct. 1987.

[23] K. Batcher, "Design of massively parallel processor," *IEEE Trans. Comput.*, vol. c-28, pp. 836–840, 1980.

[24] E. Arnould, H. Kung, O. Menzilcioglu, and K. Sarocky, "A systolic array computer," in *In Proc. IEEE ICASSP'85*, (Tampa, FL), pp. 232–235, 1985.

[25] W. Hillis, *The Connection Machine.* Massachusetts Institute Technology Press, 1985.

[26] M. Tariq and F. El-Guibaly, "Systolic array for solving linear equations," in *Proc. Canadian Conference on Electrical and Computer Engineering*, (Montreal), Sept. 1989.

[27] M. Tariq and F. El-Guibaly, "Systolic array for successive over relaxation of linear equations," in *Proc. IEEE Pacific Rim Conference on Communication Computers and Signal Processing*, (Victoria), June 1989.

[28] S. Kung, *VLSI Array Processors.* Englewood Cliffs, NJ: Prentice Hall, 1988.

[29] C. Mead and L. Conway, *Introduction to VLSI Design.* Addison-Wesley, 1981.

[30] B. Codenotti and F. Romani, "A compact and modular VLSI design for the solution of general sparse linear systems," in *Integration*, vol. 5, pp. 77–86, 1987.

[31] D. O'Leary, "Systolic arrays for matrix transpose and other reorderings," in *IEEE Trans. Comput.*, vol. c-36, pp. 117–122, Jan. 1987.

[32] H. Barada and A. El-Awamy, "Systolic architecture for matrix triangularisation with partial pivoting," in *Proc. IEE*, vol. 135 Part E, pp. 208–213, 1988.

[33] F. E. Guibaly, "Matrix triangularization using givens rotations," in *Proc. Canadian Conference on Electrical and Computers Engineering*, (Vancouver), pp. 525–528, Nov. 1988.

[34] P. Ang and M. Morf, "Concurrent array processor for fast eigenvalue computation," in *Proc. ICASSP*, pp. 34A.2.1–34A.2.4, 1984.

[35] H. Chuang, L. Chen, and D. Qian, "A size-independent systolic array for matrix triangularization and eigenvalue computation," in *Circuits and Systems Signal Procecss.*, vol. 7, pp. 173–189, 1988.

[36] P. Eberlein, "On the schur decomposition of a matrix for parallel computation," in *IEEE Trans. Comput.*, vol. c-36, pp. 167–174, 1987.

[37] U. Schwiegelshohn and L. Thiele, "A systolic algorithm for cyclic-by-row SVD," in *Proc. ICASSP*, pp. 768–770, 1987.

[38] U. Schwiegelshohn and L. Thiele, "One- and two-dimensional systolic arrays for least-squares problems," in *Proc. ICASSP*, pp. 791–794, 1987.

[39] G. Papadourakis and F. Taylor, "Implementation of kalman filters using systolic arrays," in *Proc. ICASSP*, pp. 783–786, 1987.

[40] H. Yeh, "Systolic implementation on kalman filters," in *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 36, pp. 1514–1517, 1988.

[41] H.-G. Yeh, "Kalman filtering and systolic processors," in *Proc. ICASSP*, pp. 2139–2141, 1986.

[42] R. Linderman and W. Ku, "A three dimensional systolic array architecture for fast matrix multiplication," in *Proc. ICASSP*, pp. 34A.6.1–34A.6.4, 1984.

[43] H. Chuang and G. He, "Design of Problem-size independent systolic array systems," in *Prc. International Conference on Computer Design: VLSI in Computers*, pp. 152–157, 1984.

[44] I. Jacques and C. Judd, *Numerical Analysis.* New York: Chapman and Hall, 1987.

[45] H. Kung and C. Leiserson, "Highly concurrent systems," in *Introduction to VLSI Systems* (Mead and Conway, eds.), Addison Wesley, 1980.

[46] J. Wilkinson, *The Algebric Eigenvalue Problem.* Oxford: Clarendon Press, 1965.

Appendix A

Details of Datapath Design

To design the datapath (DP) block from 2-micron standard cells, we need the standard cell library (pvsc010d) and the datapath library (vdp010d) on our search path. The pvsc010d is the basic standard cell portable library. The vdp010d library contains all the specifications for the datapath netlist compiler: specification schematics, leaf cell netlists, the compiler code and the template.

To create a datapath, first we must draw a schematic that specifies the functional units needed and the connection between them. To do that use the VLSI Schematic Editor and the elements function in the vdp010d library.

Inside the VLSI Schematic Editor the datapath specification schematic is a normal schematic except:

- The schematic requires one instant peparameter which is the word width of our datapath, N.
- Each element represents a functional unit that operates on N-bit data. The schematic we place is actually empty, it is just an indication to the compiler.
- Each wire we draw represents an N-bit bus.

- Connectors created in the specification schematic imply N connectors in the final design.

- We don't specify control connectors, they are implied by what functional units we use.

On the basis of above mentioned discussion the details of our datapath specification schematic and how it looks like is shown in Fig. 3.17. More details about the datapath compiler and VDP library can be found in " Datapath Library " manual.

The datapath was designed and captured using the 2-micron CMOS VDP10 datapath library. The name of each cell, the area per bit and the power consumption of each cell is given in the following table.

	Name	Area/bit (k μm^2)	Power ($\mu W/MHz$)
1	Adder/subtractor VDP3ASB001	30.4	156
2	Shifter VDP3SHF001	$(11.5 + 1.12 * N + 1)$	156
3	Zero detect VDP3ZDT001	9.1	10
4	Mux VDP3MUX005	5.4	13.9
5	D latch VDP3DFF003	11.25	28.5
6	Bit swizzel VDPSSWZ001	NA	NA
7	Tri-state buffer VDP3TSB002	6.5	28.5

A.1 Datapath Verification

VTI test description language gives us the ability to create simulation files and to generate complete test programs. We used VTI test language (VTL) to create a modular description of clocking, input data, stimuli patterns and expected responses. A sample program used for the simulation of the datapath is shown below.

```
#*************************************************
#                 Datapath Test Program
#*************************************************

#cell2 * man_add2 tst * 2 any 0 v8r1.5
# "19:47:35 GMT" mstariq * .

PINDEF; # define the input and output pins of the datapath.

        BEGIN
                TESTPINS 70;  # total number of pins of datapath
                datain[23:0]    := 24:1 INPUT;
                control[17:0]   := 42:25 INPUT;
                flag[2:0]       := 45:43 OUTPUT;
                clk             := 46   INPUT;
                dataout[23:0]   := 70:47 output;

        END;

TIMEPARAM normal;  # setting the timing parameters
        BEGIN
                tsetup   := 20n, 30n, 40n; # min, nom, max
```

```
                trise    := 50n, 50n, 50n;
                tcontrol := 10n, 20n, 30n;
                time     := 100n,100n,100n;
        END;

EDGETIME; # setting the input clock timing
        BEGIN
                t0        := 0n;
                clockrise :=trise.nom;
                setup     :=trise.nom - tsetup.min;
                hold      :=trise.nom + tsetup.min;
                csetup    :=tsetup.min;
                t_cycle   :=time.nom;
        END;

CYCLE reset; # resetting the inputs
        BEGIN
                 clk            <- 0 @ t0;
#                datain           <- 0 @ t0;
#                control        <- 0 @ t0;
        END;

# defining the timing for the clock, input data and
the control signal.
CYCLE load1(aval, dval);
        BEGIN
                clk       <- 1 @ clockrise, 0 @ t_cycle;
                datain    <- dval @ setup, 0 @ hold;
```

```
                control     <- aval @ csetup;
        END;

MAIN; # the main module to test the datapath
        BEGIN
            DURATION t_cycle;
     VARIABLE llatch0, llatch1, llatch2, add, sub;
     llatch0 := 'H04000; #the generation of control signal to latch the data
     llatch1 := 'H08000;

     llatch2 := 'H10000;

     add     := 'H01044; # to add two input numbers

     sub     := 'H0104C; # to subtract two input numbers
            normal;
                vlsisim "vec control[17:0]";
                vlsisim "vec datain[23:0]";
                vlsisim "vec dataout[23:0]";
                vlsisim "vec flag[2:0]";
                vlsisim "watch (dis) control flag datain dataout";
                load1 (llatch0 ,'H00000F);
                load1 (llatch1 ,'H000001);
                load1 (add , 0         );
        END;
############ End of the test program ##############
```

The output report for the datapath of loading two input data at consecutive clock cycles and adding them is as follow:

```
#*******************************************************
#     The test output of the datapath          *
#*******************************************************

VLSItest> load man_add2
     Parsing VLSItest description file 'man_add'
VLSItest> simulate
  Redefining vector CONTROL
  Redefining vector DATAIN
  Redefining vector DATAOUT
  Redefining vector FLAG
  Time = 10:1   [1800 ns]
                      CONTROL  =  'B000100000000000000  [+0]
                       DATAIN  =  'B000000000000000000001111  [+0]
                         FLAG  =  'B100  [+61.1]
  Time = 11:1   [1900 ns]
                      CONTROL  =  'B001000000000000000  [+0]
                       DATAIN  =  'B000000000000000000000001  [+0]
  Time = 12:1   [2000 ns]
                      CONTROL  =  'B000001000001000100  [+0]
                       DATAIN  =  'B000000000000000000000001  [+0]
                         FLAG  =  'B000  [+45.2]
                      DATAOUT  =  'B000000000000000000010000  [+39]
VLSItest> quit
  Time = 13:1   [2100 ns]

#************ End of the test output ***************
```

To simulate the datapath the following procedure was used.

1. A VTI Test program is written using VTI Test as shown in section A.1.
2. Load the cell to be simulated in "VTI sim".
3. Inside the "VTI sim" window type "VTI Test" to enter VTI test mode.
4. Use load "tst program" to load the VTITest program. The syntax errors will be flagged. These errors must be fixed before proceeding further.
5. Use the command "output file name" to output the result of simulation.
6. "quit" will return to VTI sim any time.

VITA

Surname:	Chaudhary	Given Names:	M.S. Tariq
Place of Birth:	Rawalpindi, Pakistan	Date of Birth:	21 Sept. 1956

Educational Institutions Attended:

University of Victoria	1987 to 1990
NED University of Engineerring and Technology Karachi, Pakistan	1980 to 1984

Degrees Awarded:

BE Electronics	NED University of Engineering	1984

Publications

1. M. S. Tariq and F. El-Guibaly, "Systolic Array For Successive Over Relaxation of Linear Equations", Proc. IEEE Pacific Rim Conf. on Communication, Computers and Signal Processing, pp. 48-51, Victoria, June 1989.

2. M. S. Tariq, F. El-Guibaly, "Systolic arrays for solving linear equations", Proc. Canadian Conference on Electrical and Computers Engineering, Montreal, September 17-20 1989.

3. F. El-Guibaly and M. S. Tariq, "Spectral Decomposition of Matrices Using Systolic Arrays", Canadian Conference on VLSI, Vancouver, October 22-24 1989.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this dissertation for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this dissertation for financial gain shall not be allowed without my written permission.

Title of Dissertation: Systolic Array Processor for Eigenvalues and Eigenvectors

Author: 

(Signature)

M.S. TARIQ CHAUDHARY

(Name in Block Letters)

December 6, 1990

(Date)