NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents

PERFORMANCE ANALYSIS AND DESIGN OF AREA EFFICIENT FAULT-TOLERANT SYSTOLIC ARRAYS

Michael O. Esonu

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada.

September, 1991

ISBN 0-315-73666-6

ABSTRACT

Performance Analysis and Design of Area Efficient Fault-Tolerant Systolic Arrays

This thesis is concerned with the development of a systematic methodology to design optimal or near optimal fault-tolerant systolic array architectures. The space-time approach of mapping algorithms into systolic arrays, has been adopted in the design of fault-tolerant systolic arrays. The research problems considered in this thesis include: the formulation of a methodology to obtain the desired transformed dependency matrix (TDM) before generating its corresponding transformation matrix (T); the derivation of more realistic optimality criteria to design systolic arrays; the identification of a unifying performance index to measure the overall systolic array performance; and, the design of area efficient concurrent error detection and correction systolic arrays.

In the existing approaches of mapping algorithms into systolic arrays, the structure of the systolic array is not known until a valid transformation matrix is generated, and then used to select the systolic matrix. Although, the systolic array obtained with this transformation matrix represents a feasible design, in most cases, it does not satisfy some important VLSI requirements. The methodology for generating the transformed dependency matrix presented in this thesis, allows the desired systolic structure to be selected first hand. This approach is computationally efficient in that, it avoids the derivation of the TDM's that do not satisfy the VLSI requirement, thereby, eliminating the generation of their corresponding transformation matrices.

We have identified the optimality criteria that take into account the architectural and technological parameters of the systolic array. The relationship between the cost measures and the coefficients of the TDM has been established. By changing the values of the

coefficients of the TDM, the values of the cost function parameters are affected. This enables us to investigate the practical design issues at the design stage rather than after the implementation of the array. Also, a Compound Objective Function has been formulated to measure the overall performance of the systolic array. This is a unified performance index which takes into consideration the effects of all the optimization factors, in the design of the array. This approach eliminates the problem of sacrificing other factors of the optimization criteria when only one factor is optimized.

Furthermore, approaches to design optimal Concurrent Error Detection (CED) and Fault-Tolerant (FT) systolic arrays have been developed. The basic concept of the approaches is the introduction of redundant computations at the algorithmic level, such that when these algorithms are mapped into VLSI systolic array, the inherent hardware availability allow concurrent error detection and correction in the systolic arrays. Several fault-tolerance techniques have been presented based on the proposed FT approach. The CED and FT approaches are area efficient. They also overcome most of the drawbacks of other existing techniques.

TO MY PARENTS

COMFORT AND SUNDAY

ACKNOWLEDGEMENTS

I sincerely express my gratitude and strong appreciation to Professor A. J. Al-Khalili, for his continued guidance, suggestions, encouragement and assistance throughout the course of this research. Besides being a great teacher, Professor A. J. Al-Khalili is also a nice man. It has been a great pleasant experience working in association with him and I thank him very much for that.

Also, I am very grateful to my co-supervisor, Dr. S. Hariri for his insight, guidance, encouragement and various diligent contributions he made towards the progress of this research. I would like to thank Dr. H. F. Li for suggesting the research topic and for the initial discussions and help given to me on the topic of this thesis.

Last but not the least, my special thanks go to my wife for her support and also her help in the final preparation of this thesis. With great pride, I acknowledge the support and inspiration of my parents, my brother, sisters and relatives back home. Also, I would like to thank all my friends and well - wishers in Montreal for their encouragement and the special moments that we enjoyed together.

This research was supported by the Natural Sciences and Engineering Research Council of Canada under a Grant awarded to Professor A. J. Al-Khalili.

Table of Contents

LIST OF VARIABLES

T: Transformation matrix

D: Dependency matrix

Λ: Transformed dependency matrix

$L^n[\bar{J}]$: Index set of an algorithm

Z: Transformed index set

$\bar{d}_i$: Dependence vectors

δ_i: Transformed dependence vectors

R: Ordering imposed by the data dependencies on set L^n

R_T: Ordering imposed by the transformed data dependencies on set L^n

Ω: Matrix added to D to generate new Δ

UL: Upper limit on the generation of the new TDM

FT: Fault - tolerance variable

W: Constant (a coefficient in the objective function)

D_{int} :Delay of an interconnection line with unit length

R_{int} :Number of data routing steps

τ_L: Delay of the interconnection line with the longest path

F_s: Clock frequency of a single PE

C: Total number of clock cycles required for the computation of an algorithm

T_c: Total execution time of the algorithm

d_u: Number of delay units in a processor

U_d: Total number of delay units in the array

A_d: Area of a delay unit

AD: Total area of the delay units

m_{PE}:Total number of processing elements in the array

m_{io}: Total number of I/O's in the array

A_{PE}: Area of a processing element

AP: Total area of the processing elements

u_i: Data routing steps in the horizontal direction

v_i: Data routing steps in the vertical direction

K: Indicates the degree of complexity of the interconnection pattern

A_L: Area of a unit length interconnection line

$A_{int.}$:Area required for routing the interconnection lines

A_{si}: Total silicon area required to implement the algorithm

SP: Speedup per processing element

COF: Compound objective function

c_i: Modulating constants of the COF function.

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 SYSTOLIC ARRAYS

A systolic system consists of a set of interconnected simple processing elements (PE's) or cells, each capable of performing some simple or complex operations [1,2,3]. Because simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic system are typically interconnected to form a systolic pipeline, array or tree [1]. Information in a systolic system flows between cells in a pipeline fashion, and communication with the outside world occurs only at boundary cells. For example, in a systolic array, only the cells on the array boundaries may be I/O ports for the system.

The systolic array features the important properties of modularity, regularity, local interconnection, a high degree of pipelining and highly synchronized multiprocessing [4]. The data movements in a systolic array are often described in terms of the snapshots of the activities [4]. The systolic array design differs from the conventional Von Neuman computer [5] in its highly pipelined computations. By replacing a single processing element with an array of PE's (cells), higher computation throughput can be achieved without increasing memory bandwidth [1,2]. The memory pumps data through the array of cells. More precisely, once a data item is brought out from the memory, it can be used effectively at each cell it passes while being pumped from cell to cell along the array. This is especially appealing for a wide class of compute-bound computations, where multiple operations are performed on each data item in a repetitive manner [4]. This avoids the classic memory access bottleneck problem commonly incurred in Von Neuman architectures [5].

Being able to use each input data item a number of times is just one of the many advantages of the systolic approach. Other advantages include, modular expandability, simple and regular data and control flows, and the use of simple and uniform cells. Simple and regular interconnections lead to cheap implementations and high densities. High density on the other hand, implies both high performance and low overhead for support components. Due to these reasons, multiprocessor structures which have simple and regular communication paths have been considered interesting architectures for modern signal and image processing applications. Also, the use of pipelining as a general method for applying these structures is logical and attractive. By pipelining, computation may proceed concurrently with input and output operations, consequently, minimizing the overall execution time. Systolic arrays thus take advantages of the concepts of pipelining, parallelism and regular interconnection structures [3].

Unlike the closed-loop circulatory system of the body, from which this type of computer architecture derives its name, a systolic computing system usually has ports into which inputs flow and ports where the results of the systolic computations are retrieved. In a systolic system input and output can occur with every pulsation. This makes systolic arrays attractive as specialized peripheral processors attached to the data channel of a host computer. A systolic system may also possess a real-time data stream or be a component in a large special purpose-system [1].

A unique property of systolic arrays is that system performance is proportional to the number of cells. So systolic arrays in general have large numbers of cells, as required by the special purpose applications for which they are designed. Because systolic arrays are used to implement special purpose systems whose development costs cannot be amortized over a large number of units, it is useful to have some flexible means of assembling many different types and sizes of systolic arrays from a small number of building blocks. Such a tool must provide programmability both within individual cells and at the interconnection level.

Many projects developing systolic arrays for special purpose applications are described in the literature [6-8]. Some of the applications of systolic arrays include linear algebra computations such as matrix multiplication and various signal processing algorithms like digital filtering, convolution, Fast Fourier Transforms (FFT). Other application areas include robot control, medical image processing, computer vision, nuclear physics, structure analysis, sonar, radar, seismic, weather computation and so on.

1.2 BASIC PROBLEMS IN SYSTOLIC ARRAY DESIGN

An important problem associated with the design of a systolic array is the mapping of algorithms into systolic architectures. Many known designs of systolic architectures are based on heuristic approaches. However, there has been a considerable effort in the development of systematic methods for synthesizing systolic arrays based on algorithm-oriented analyses [9-28]. Systolic arrays derive maximal concurrency by using both pipelining and parallel processing. The features of many algorithms used in signal and image processing include localized operations, intensive computation and matrix operations. In order to facilitate the design of special purpose systolic arrays, the common features of this algorithm should be exploited. Therefore, another important issue with the design of systolic arrays is how to fully express the inherent concurrency in these special purpose algorithms.

Algorithm expression is a basic tool for a proper description of an algorithm for parallel processing. There are two approaches to derive parallel algorithm expressions [3,4,29]. These are vectorization of sequential algorithm expressions and direct parallel algorithm expressions, such as single assignment code, parallel codes, recursive equations, dependence graphs and so on. In the first approach, a vectorizing compiler processes a source code written in a sequential language and, where possible, generates parallel machine instructions. This approach is not sufficiently effective in extracting the inherent concurrent processing. It is advantageous to use parallel expressions to describe

algorithms.

The most important parallel expression of algorithms is the Dependence Graph (DG). In order to achieve the maximal parallelism in an algorithm, the data dependencies in the computations must be carefully studied. There is always a certain degree of dependency which dictates the sequence of computations. The DG is a graphical representation of these data dependencies. Once the data dependencies are explicitly expressed, a systolic array processor implementation can then be derived by mapping the DG's onto processor arrays.

There are two methodologies for mapping algorithms into systolic arrays. These are the canonical mapping methodology for mapping homogeneous DG's onto processor arrays; and the generalized mapping methodology for mapping heterogeneous DG's onto processor arrays. An enormous number of algorithms have the useful properties of being totally regular and localized [4]. The canonical mapping methodology is suitable to treat such class of algorithms, which can be expressed by shift-invariant dependence graphs. This mapping methodology consists of three design stages: (1) DG Design, (2) mapping the DG onto a signal flow graph (SFG) and (3) deriving a systolic array from the SFG. The generalized mapping methodology allows the treatment of a broader class of algorithms that are not completely regular (i.e., not totally shift-invariant) but exhibit a certain degree of regularity. This thesis concentrates only on the canonical mapping methodology.

Systematic approaches to derive a systolic array processor implementation by using such regular DG's have been proposed in the literature [9-28]. Karp, Miller and Winograd [9] proposed the *systems of uniform recurrence equations*. In their systems of uniform recurrence equations, they explored the idea of local and regular Dependence Graph (DG). They used an index space display to show the complete dependency of *locally recursive algorithms*. The idea of uniform recurrence equations was applied later on by Quinton [10,11] to the design of systolic arrays. Gachet *et al.* [12] described the metho-

dology underlying the *DIASTOL* system, whose aim is to allow systolic chips to be designed automatically. This methodology, called dependence mapping, is also based on the ability of someone to describe a problem as a system of *uniform recurrence equations*, then mapping the problem on a systolic array. Rao [13] defines a class of algorithms, namely the *regular iterative algorithms* similar to the systems of uniform recurrence equations defined in [9,11]. It is shown that a subclass of the regular iterative algorithms has the characteristics of the systolic algorithms and the corresponding systolic architectures may be systematically derived. The notion of *locally recursive algorithms* proposed by Kung [14], stresses the locality of spatial and time indices in a recursive algorithm and therefore is expressible in terms of a spatially local DG or Signal Flow Graph (SFG). Many research explorations on this issue are discussed in [15,16,17,18]. A more detailed review can be found in [13,19].

Capello and Steiglitz [20] introduced a geometric interpretation of the linear transformation on index space, which provides an insightful look into how several systolic designs for the same algorithm relate to each other. Along the same line as the approach proposed in [20], several researchers [21-28] address the issue of mapping cyclic (loop) algorithms into systolic arrays. The cyclic algorithms are specified in a high-level language, such as FORTRAN, in the form of DO loops. The approach is based on the space-time mapping of different cyclic algorithms into systolic array architectures. The mapping procedure is based on linear transformation of index sets and data dependence vectors. Moldovan extended the mapping to the partitioning problem [23]. He presented a technique for partitioning and mapping algorithms into VLSI systolic arrays. Algorithm partitioning is essential when the size of a computational problem is larger than the size of the VLSI array intended for that problem. His approach to the partitioning problem is to divide the algorithm index set into bands and to map these bands into the processor space. Finally, he presented a six step procedure for the partitioning and mapping technique. The main difference between the mapping approaches proposed in [9-20] and

those proposed in [22-28] is that, the latter approaches started from a program using imperative languages such as FORTRAN rather than from equations.

In this thesis, we concentrate on the approach of space-time mapping of different cyclic algorithms into systolic array architectures [21-28]. For this approach, in order for a computation structure to be implemented in a VLSI systolic array, the conceptual sites $\{\bar{J}^n = (J^1, J^2, J^3,..., J^n)\}$ must be mapped into $Z^n = \{(\hat{J}^1, \hat{J}^2, \hat{J}^3,..., \hat{J}^n)\}$, where $\bar{J}^n$ denotes the index set of the algorithm and J^n is the nth - dimensional set of an algorithm. Although it is considered that VLSI arrays are $(n-1)$ - dimensional, practical arrays have pure planar layout. Therefore, for a two - dimensional systolic array, the conceptual sites $\{\bar{J}^n = (i,j,k)\}$ must be mapped into $Z^3 = \{(t,x,y)\}$ where t specifies the time and (x,y) represents the 2-dimensional physical coordinates of the place in the VLSI systolic array where the node is computed. If a computational structure is characterized by a constant dependency matrix which consists of a set of vectors, $D = (\bar{d}_1,\bar{d}_2, \ldots ,\bar{d}_k)$. Then the structure may be mapped into a time-space representation in Z^3 with new dependency as a systolic matrix $\Delta = (\bar{\delta}_1,\bar{\delta}_2, \ldots ,\bar{\delta}_k)$, where k is the number of columns of the dependence vectors. This is done by means of matrix multiplication, $TD = \Delta$, where the matrix T is a valid transformation matrix [23-25,27].

The transformation T is chosen such that the ordering of the execution of the algorithm is preserved. Necessary and sufficient conditions for the existence of valid transformations are given for algorithms with constant data dependence [20-22,24]. Many transformation matrices T can be found for a given constant dependency matrix, and each transformation leads to a different array. This flexibility gives the systolic array designer the possibility to choose between a large number of arrays with different characteristics [27]. In order to obtain an optimal design, given the cost function or the optimality criteria, a heuristic procedure is used to search for the best one among many feasible transformations. The most suitable optimality criteria are hard to pinpoint and optimizing one factor may sacrifice other factors [4]. There are many factors in determining the

optimization criteria for the design of systolic arrays. The choice of the optimality criteria is, in general, application dependent.

Several works on how to design optimal systolic array architectures have been proposed in the literatures [4,12,13,15,23,24,30-42]. The works done by Kung [4], Delosme [15], Moldovan [23], Miranker [24], Wong [30], O'keefe [31] and Fortes [32] are based on how to minimize the *Computation time* of a systolic array. The *Computation time* (denoted by C) is the time interval between the first computation and the last computation of a problem instance by the processor array. They defined how the computation time of a systolic array is computed given the time schedule vector of the algorithm. They formulated that the range of time steps generated by points in the index set should be a set of consecutive integers. Therefore, the computation time is equal to the difference between the largest and the smallest time steps in the range set. Hence, in order to obtain the systolic array which is optimal with respect to the computation time, in most cases, all the allowable time schedule vectors are enumerated and the one that gives the minimal computation time solution is selected.

Rao [13] worked on minimizing the *Pipelining period* (α) and the *Block pipelining period* (β) *of a systolic array. The Pipelining period* is the time interval between two successive computations in a processor. In other words, the processor is busy for one out of every α time intervals. It should be noted that α is the reciprocal of the pipeline rate. On the other hand, the *Block pipelining period* is the time interval between the initiations of two successive problem instances by the processor array. He formulated an integer programming problem to find the schedule vector which minimizes α, given the projection vector. The block pipelining period can be calculated using the reservation table, which specifies for all PE's in the array, the time steps a certain PE is busy during the computation of one problem instance. Based on the table, the *time span* for each PE can be calculated, which indicates the difference between the first time step and the last time step during which a PE is busy in the reservation table. Therefore, the block pipelining period is

simply the largest time span of any PE in the array. Thus, a systolic array is optimal with respect to α and β if it has the minimum α and/or β, among the numerous arrays.

In addition to minimizing the execution or computation time, Fortes [32] proposed a heuristic approach for optimizing the hardware cost. The *array size*, which is defined as the number of processors in the array, obviously determines the basic hardware cost. Therefore, a systolic array which has the minimum number of processors gives the optimal solution with respect to this cost function. Gachet *et al.* [12] described a methodology, called the dependence mapping, which is based on the ability for someone to describe a problem as a system of uniform recurrence equations, then mapping the problem on a systolic array. In their design approach, they were interested in those transformations (projections) that will give fewer number of cells in the systolic design. Hence, their cost function is to minimize the number of processors in the array. Leiserson [33] mentioned that it is desirable to minimize hardware cost by using minimal delays in a systolic array, while preserving optimal α. He proposed a procedure to optimize the total number of delays in a systolic array, which is based on delay transfer through nodes. This is well known systolization scheme or the retiming of a synchronous circuit. Thus, by minimizing the delay elements in a systolic array, an optimal solution can be obtained with respect to this criterion.

A mapping technique to design systolic structures having limited I/O requirement has been proposed in [34-37]. The authors argued that even though many mapping techniques have been proposed for the design of systolic arrays, due to limited I/O access, many of the algorithm mapping techniques cannot be directly applied or result in complex designs. Some of these designs have complicated controls and non-uniform I/O patterns [35,36]. For example, many proposed designs in the literature requires $O(N)$ I/O bandwidth for problems of size N^2, which may be hard to realize in practice. In order to cope with this problem, they proposed a design methodology which is based on the linearization of arrays. Systolic design use linear connected arrays with data and control

signals pumped at either end [35,36,37]. Their approach requires $O(\sqrt{N})$ I/O bandwidth for N x N systolic matrix multiplication algorithm.

Li *et al.* [38] proposed a *parameter* method of designing optimal systolic arrays using the parameters: **velocities of data flows, spatial distributions of data and the periods of computation.** By relating these parameters in constraint equations that govern the correctness of the design, the design is formulated as an optimization problem. The velocity of datum x is defined as the directional distance passed by x during a clock cycle. The directional distance of x comprises of the number of PE's and buffers traversed by x in a given number of clock cycles. For instance, if x propagates through i PE's and $j-i$ buffers in j clock cycles, and if $i=1$, then there are $j-1$ buffers between two neighboring PE's in the pipelining direction of x. For the data distribution, it is considered that the input elements of a systolic array, along a row or column are arranged in a straight line and equally spaced as they pass through the systolic array, and the relative positions of the elements are iteration independent. The number of such distinct lines is referred to the number of streams of data flow. Their definition of period is the same as pipelining period α. The performance of a systolic design is expressed in terms of these parameters. The number of PE's required, denoted by m_{PE}, depends on the directions in which the inputs are moving. The completion time (T) can be expressed as a function of the PE configuration and velocity. The design problem is then formulated to minimize m_{PE} x T^2, or m_{PE} x T, or T. Hence, the optimal solution is the systolic design that is minimized with respect to any of these criteria.

In Ko *et al.* [39], algorithms are specified in terms of data dependency, and implementations are specified in terms of data propagation and sequence behavior. By establishing a relation between data propagation and sequence, an optimal mapping strategy is formulated as a problem of finding an integer solution of a set of linear equations. The optimal mapping strategy is as follows: (i) Find the time mapping function which gives the fastest output propagation. (ii) Find the space mapping function which gives the

highest throughput and minimum computation delay. (iii) Find the propagation behavior. (iv) Find the sequence behavior. Therefore, the mapping function which gives the fastest output propagation and highest throughput gives the optimal systolic array solution. One problem with this approach is that, in order to find the time and space mapping functions, the designer must choose the desired propagation and sequence vectors that will maximize throughput and give the fastest output propagation. This means that the designer must have prior knowledge on how to select the vectors that possess these desired properties. Therefore, the difficulty of this approach lies in determining the propagation and sequence vectors that give the optimal mapping function. Another problem with the approach is that, they assume that the solution matrix obtained as a mapping function consists of only integer elements. However, in some cases, there is no integer solution for the desired propagation and sequence. In this case, they choose the integer solution that closely approximates the desired propagation and sequence, by rounding the non-integer solution to the nearest integers. Then this means that the mapping function obtained in this case will yield a sub-optimal systolic array instead of the desired optimal array.

In the above approaches, the criterion used for optimum architecture selection is a space - time cost function formed by the total number of processing elements and the required clock cycles for a task. These are purely architectural arguments, which rely on the assumption that the maximum clock frequency is independent of processing element count. In a monolithically integrated circuit, this is not necessarily true [41,42]. There is an interplay between heat dissipation, signal propagation and chip area which must be satisfied if a system has to function in a synchronous fashion without overheating. This interplay sets physical limits on clock frequency [40]. Lee *et al.* [40] compared systolic architectures for matrix multiplication, in terms of the maximum speedup which can be achieved with increased processor count in a monolithically integrated circuit. The comparison process integrates the architectural characteristics and the technological parameters. The optimum systolic architecture is found for different physical limiting factors

including **switch delay, power dissipation, I/O bandwidth and clock skew.**

1.3 ERROR DETECTION AND FAULT-TOLERANCE

Since systolic arrays in general have larger number of cells, it is inevitable that some cells will fail in such large arrays. Therefore, systolic arrays must be designed to function correctly in the presence of failure. In other words, they must be fault-tolerant. Fault-tolerant techniques are characterized by the inclusion of redundant functional elements, both logic circuit and interconnections in the design and the ability to modify the interconnection structure. This results in yield enhancement. When a defect is found during production testing on a wafer containing systolic cells, the defective cell can be substituted by a working spare cell on the same wafer. This restructuring is done at the fabrication facility before shipping the wafer to the field.

An equally important use of fault-tolerance through redundancy results in enhanced reliability of the VLSI wafers after shipping. Many techniques have been proposed over the years to achieve fault-tolerance in the field [43]. Any fault-tolerance technique is designed to tolerate a given class of faults within a system. A fault can be treated at any level within the system. This can be from a very low level such as the transistor level, to a higher level such as the functional module level. Most fault-tolerance techniques have been designed to tolerate faults in some module within a system. Such a *module-level fault model* is ideal for VLSI, where a physical failure can cause some portion of the chip to be faulty [43]. A module fault is assumed to result in arbitrary errors at the output of the module. It is these errors that must be prevented from appearing at the output of the system, after the computation is completed.

Several steps are involved to achieve fault-tolerance [43]. These include, detection of an error at some module output; correction of the error; location of the faulty module; and reconfiguration of the system to bypass the faulty module. There are several ways to

detect and correct errors in systolic arrays. An example would be to check the computation in order to detect an error, and once it is detected, the computation can be rolled back to a previously error-free state to correct the error. However, if there is sufficient redundancy in the computation, an error due to a module failure can be masked by correct values from other modules. This technique is employed in the design of highly redundant systems for space applications. In the reconfiguration approach, the faulty module must be located first before the system can be restructured. Reconfiguration approach is a candidate for application environments in which permanent failures are the dominant concern.

There are three redundancy approaches to fault-tolerance, namely, space, time and algorithmic redundancies. The space redundancy can be static, hybrid or dynamic. In static redundancy approach, which is also known as masking redundancy, N copies (where N is odd) of a module and a majority voter are used to mask the errors from failed modules. This scheme is also known as N-Modular Redundancy (NMR) and a popular version for N = 3 is called the Triple Modular Redundancy (TMR). The static technique can be combined with a set of spares through the use of a disagreement detector and a switching unit to produce a hybrid redundant system [44]. In the dynamic redundancy approach, the faulty modules are identified and the system is reconfigured by replacing faulty modules with spares. Time redundancy approach involves recomputing the same computation twice in the same module or in adjacent modules at two different but close enough time periods and then comparing the results. If they do not match, a roll-back procedure can be used to correct the errors. Algorithmic redundancy is based on data-encoding approach. The input data to the algorithm are encoded at the system level in the form of error-correcting or error-detecting codes. The original algorithm is redesigned to operate on these encoded data and to produce encoded output data. The redundancy in the encoding would enable the correct data to be recovered.

Many classical fault-tolerance techniques employ the masking redundancy approach

[5,44-48]. In this approach at least three modules are necessary in a voting system. For a multiple processing unit systolic array, this technique can be applied by triplicating each of the systolic cells and having a voter at the output of the cell, or by triplicating the entire array with one voter. The overhead for fault-tolerance is at least 200%. The technique is very general and can be applied at any level in a highly parallel system. However, due to the high hardware overhead, the cost of the fault-tolerant system is high. It has been demonstrated that a lower cost fault-tolerance technique is to design the system to detect the errors during normal operation. This will then be followed by steps to identify the faulty unit and provide correction to the errors. It has also been shown that the combination of space and time redundancy can lead to a very attractive form of fault-tolerance [43].

Many Concurrent Error Detection (CED) schemes for systolic arrays have been proposed in the literature [4,49-61]. The CED scheme proposed in [50] is called Comparison with Concurrent Redundant Computation (CCRC). In this scheme, each computation and its redundant counterpart are performed in two adjacent cells simultaneously and then the results are compared. This approach is applicable to a class of systolic arrays in which the data as well as the (sub) results keep moving from cell to cell during computation. A similar approach to that proposed in [50] was proposed in [4,51,52], but the technique in [4,51,52] is applicable to unidirectional data flow linear systolic arrays. The technique in [53] employs Algorithm-based CED approach. The strategy used in the technique is to encode input matrices and to check the encoded matrices of the output of computations to determine whether the outputs are reliable.

The approach in [54] is based on concurrent redundant computation which is similar to the approaches in [4,50-52]. However, the scheme in [54] is restricted to a class of systolic arrays where the partial results most stay in the cells. In [55], the technique combines systolic array circuit architectures with residue number system (RNS) computations. Two independent residue calculations are performed and the results are compared for

discrepancy. A CED approach for implementing algorithm-based fault-tolerance in parallel processing arrays is proposed in [56]. It is based on the notion of diagnostic invariance. Certain characteristics of the input data called diagnostics, which match the characteristics of the output data are determined. A mis-match between the characteristics indicates the presence of faults in the systolic array.

The CED schemes in [49,57-59], use the approach of Recomputation with Shifted Operands (RESO) to test systolic array by repeating every computation with shifted operands so that each cell in the arithmetic unit operates on a different set of bits. It can be shown that, with appropriate shifts and design of the arithmetic unit, a faulty cell will cause the two results to be different, achieving error detection. The CED scheme proposed in [60] is based on the use of logarithmic coding (addition theorem of logarithms) to detect errors in systolic arrays. Two independent matrix computations are performed using the multiplication and anti-logarithm approaches, respectively. A discrepancy between the two results indicates that a set of the output results is erroneous and hence the fault can be detected. The relationship between concurrent error detection via Concurrent Redundant Computation (CRC) and space-time transformation has been investigated in [61].

These CED techniques can be employed to achieve error detection using a small amount of hardware overhead. The error correction can be done using time redundancy, that is using further clock cycles. This balance between the hardware and time redundancy achieves the best possible utilization of the system modules, since normal performance is not sacrificed for error detection, and the overhead for error correction is needed only after an error has occurred [43].

Several fault-tolerance schemes, which correct errors with normal operation of the systolic array systems, have been proposed in the literature [4,43,46,52-56,62,63]. These schemes employ concurrent error detection techniques to identify errors in the computation during normal operation. Then further procedures are taken to provide correction of

the errors. The technique proposed in [46] uses the idle processing element in a systolic array to achieve error correction. It is based on TMR approach, whereby three adjacent cells in a linear bidirectional systolic array are used to produce three copies of the computed results. These results can be used to correct the errors and also identify the faulty unit. A technique to detect and correct errors using time redundancy is proposed in [4,52]. The approach is called the time redundancy with interleaving for fault-tolerance. The strategy is to perform two computations in adjacent cells at two different time periods and then compare the results. If an error is detected in the computations, a roll-back procedure is performed to correct the errors.

The technique proposed in [54] is a dual redundancy approach whereby identical sequences of inputs are entered into two adjacent processing elements. The redundant computations of two adjacent output results, y_n and y_{n+1}, are performed by one cell. If a fault occurs and it results in discrepancies in two adjacent output results, the fault can be localized to a particular cell. The outputs from that cell can be ignored and the corresponding redundant output from two adjacent cells can be accepted as correct. The detection and correction of errors arising from faults in the array can be performed either by software in the host processor, or by hardware following the system output. The scheme proposed in [55] combines systolic array architecture with residue number system computations. Independent computations are performed in modulo-controlled processing channels. Two channels in the processing model can be made redundant to form what is in effect a triply redundant array. With the capability to correct any one erroneous residue, the system becomes tolerant to any pattern of faulty cells that has no more than one faulty cell in the processing block. The approach in [56] integrates error detection with the execution of the algorithm itself. As mentioned before, this scheme is based on the determination of certain characteristics of the input data called diagnostics, which match the characteristics of the output data. If a fault occurs in the array, there will be a mismatch between those two characteristics. After the error is detected further steps are

taken to locate the faulty cell and to correct the error.

Algorithm-based fault-tolerance technique has been proposed in [43,53,62,63] to detect and correct errors in matrix operations performed by systolic arrays. The technique utilizes a matrix encoding scheme in which the data is encoded at a higher level to protect against errors affecting a faulty module in the systolic array. The encoding is done by considering the set of input data to the algorithm and encoding this set. The original algorithm must then be redesigned to operate on the encoded data and to produce encoded output data. The redundancy in the encoding enables the correct data to be recovered or, at least, recognizes that the data are erroneous.

Design strategies for fault-tolerance in systolic arrays are highly dependent on the application environments [43]. For environments in which transient errors and intermittent failures are dominant, the techniques proposed in [4,5,43-63], may be most effective for detecting and tolerating the faults. However, environments where permanent failures are the dominant concern, may best be served by reconfiguration techniques. In a reconfiguration approach, failures are tolerated by replacing the faulty processors and interconnections with fault-free spares or bypassing the faulty elements and reducing the size of the computational structure. Design for reconfiguration can be utilized to tolerate failures of a systolic array at the point of application, as well as enhancement of yield through toleration of manufacturing defects in a chip or wafer [43,64,65]. Such restructuring can be performed only once to adapt the wafer to a specific application after manufacturing. Dynamic restructuring can be performed as many times as required and provides dynamically reconfigurable architectures in the field. A number of different techniques exist for modifying the interconnection structure of VLSI cells [66,67], once the faulty cell is located. Such techniques are particularly important in wafer scale integration. Wafer scale systems are composed of large numbers of functional simple processors in the wafer surface connected by on-wafer wiring. The entire wafer is packaged, and a complete systolic array is formed from one or more such packages.

In order to reconfigure the processors' interconnection flexibly, restructurable wiring is laid out between the processors. After testing the processors, the connection between the functional processors are established by routing the wiring around the faulty processors. Several techniques exist for performing such reconfiguration. For instance, a completely unconnected wiring pattern can be fabricated on the wafer using double layer metalization. The vertical wires between modules are patterned in one level of metal and the horizontal wires are patterned on the second metal layer. The connections between the metal layers are established by layer programming [66], thereby, defining the interconnections between the processors. Another technique involves the programmable switches that can be inserted into the wiring. Each switch stores a switch setting in its local memory specifying a connection between two or more incident wires [67]. Switch setting must be defined externally, depending on the testing of the processors and the location of the faulty ones, in order to establish a specific interconnection pattern.

Successful reconfiguration relies on the following elements [43]; (i) testing or concurrent error detection techniques for diagnosis of failures, (ii) a design strategy for placement of spare computational elements, (iii) appropriate interconnection and switching techniques for incorporation of spares [66,67], and (iv) reconfiguration algorithms, which optimally allocate spares in the presence of multiple faults.

An understanding of the design objectives appropriate for reconfigurable systolic architectures provides a basis for the evaluation of different reconfiguration techniques. One of the objectives is the efficient utilization of spares. The algorithms that allocate spares in the presence of failures should also be of reasonable complexity. Other objectives for a targeted VLSI implementation include manageable VLSI layout complexity for large numbers of processors, moderate interconnect requirements, and a bounded number of pins per chip for multiple-chip implementations. Also, reconfiguration should not result in significant performance degradation such as may be due to clock skew from long interconnects in the presence of failures [43]. All of these objectives must be evaluated

through analysis that includes reliability or yield studies, and performance evaluation.

Several approaches to reconfigure systolic arrays have been proposed in the literature [36,37,43,53,64-80]. For instance in [68], the proposed systolic fault-tolerance scheme maintains the original data flow pattern by bypassing defective cells with few registers. In another example, the approach proposed in [43] for cube-connected-cycles-based networks maintain a fixed structure through switching mechanisms that replace faulty elements with spare elements. The advantage of a fixed structure is that performance of the network is maintained throughout the life of the system. However, spare elements are usually inactive during normal operation, which means that the utilization of all good PE's is not optimal. The difference between all the reconfiguration approaches lies in the reconfiguration strategy employed by each approach.

1.4 MOTIVATIONS, OBJECTIVES AND CONTRIBUTIONS

In the space-time approach of mapping algorithms into systolic array architectures, several transformation matrices T are generated for a given constant dependency matrix (D). In the previous approaches [21-28] that employ this mapping technique, a valid transformation matrix T is first obtained and then used to select the transformed dependency matrix (TDM (Δ)) by performing the matrix multiplication $TD = \Delta$. This procedure has to be repeated many times until an optimal or near optimal design is obtained. The structure of the TDM(Δ) is not known until all the operations to generate T are performed and Δ is selected. Although, the resulting systolic array obtained with the selected transformation matrix represents a feasible design, in most cases it does not satisfy some important VLSI requirements. These requirements include improving the fault-tolerant capability of the array, minimizing the silicon area and delay, minimizing the VLSI routing complexity, improving the speedup of computations, maximizing throughput and obtaining the fastest propagation of output variables.

In view of this problem, one of the objectives of this thesis, is to find an approach that will allow us to select Δ first, and examine it to see if it meets the desired VLSI requirements. The desired systolic structure will be selected first hand and then its corresponding transformation matrix T will be derived. We want to avoid deriving T for such TDM, if we know that corresponding TDM may not satisfy our requirements. In this way, we can eliminate those TDM's without the need to generate their corresponding transformations.

In order to achieve the above objective, one of the contributions of this thesis, is to formulate a methodology to obtain the desired transformed dependency matrix (TDM) directly from the original dependency matrix (DM). Rather than producing several transformation matrices and selecting the one that gives optimal Δ, we generate the new transformed dependency matrix directly from the dependency matrix, by adding another matrix Ω to the original dependency matrix (i.e. $\Delta = D + \Omega$). The validity of this approach lies on showing that it is possible in general, to add any valid matrix Ω to D and (i) obtain a Δ that does not violate the dependency constraints of the given algorithm and (ii) find non-singular integer T. This approach gives us the option to first, examine and then select the systolic structure (Δ) that meets our VLSI requirements.

Since several TDM's that meet the desired requirements can be obtained, then it will be imperative to select the TDM that gives the optimal systolic array based on certain measure of cost or criteria. A literature survey [12,13,15,23,24,30-40], of several approaches of designing optimal systolic arrays, is described above. In these approaches, the cost function is based mostly on architectural features of the systolic array, such as the total number of processing elements in the array, total computation time of the array, block pipelining period and so on. They fail to take into account those realistic optimality criteria such as the cell's complexity, the number of interprocessor connections, VLSI routing complexity and other practical design considerations. Therefore, the second objective of the thesis is to identify a more realistic optimality criteria to design systolic arrays.

In order to achieve this objective, we have formulated a more realistic and suitable optimality criteria that take into account both the architectural features and technological parameter of the systolic array. For instance, the area of a systolic array does not consist of only the number of processing elements in the array, as employed in the previous approaches. The area is a complex function of the number of PE's, the number of buffers, the interconnection pattern and the available technology. The cost functions include the speedup, cell's complexity, number of interprocessor connections, VLSI routing complexity, the propagation delay, fault-tolerant capability of the array and other practical design issues. We establish a relationship between the coefficients of the TDM (systolic matrix) and the factors of the cost function. This enables us to investigate the practical design issues at the design stage of the systolic array rather than after the implementation of the architecture.

Also, in the previous approaches [12,13,15,23,24,30-40], the optimality of systolic arrays is determined by optimizing only one cost function. This cost function would be either the computation time, or the number of processing elements, or maximum speed or a cost function formed by combining the number of processors (m_{PE}) and the required number of clock cycles (T_c) of the systolic array. It is important to note here that $m_{PE} \times T_c$ or $m_{PE} \times T_c^2$ is regarded as one optimization factor rather than two individual cost functions comprising of m_{PE} and T_c. Since optimizing only one cost function may sacrifice other factors, it becomes imperative to identify a "unifying performance index" to measure the "overall array performance". In view of this problem, we have proposed a compound objective function (COF), which is composed of several cost functions, to measure the cost of each TDM obtained directly from the original dependency matrix. The COF is a unified performance index to measure the overall performance of the systolic array architecture. It includes cost functions like the fault-tolerant capability of the array, the silicon area, the throughput, the propagation delay, the product of the silicon area and total execution time, and the speedup of computation for the systolic array. By

taking all these factors into consideration when selecting the optimal systolic array for a given algorithm, we can avoid sacrificing any of the practical design issues, during the design stage of the systolic array. The systolic array design is therefore formulated into an optimization problem of finding the TDM with the minimum cost function. The optimization algorithm is systematic and also computationally efficient.

As mentioned in the previous section, systolic arrays should be designed to function correctly in the presence of failures. This increased reliability can be achieved through fault-tolerance. Many techniques have been proposed to achieve fault-tolerance in systolic arrays [4,5,36,37,43-80]. For the reconfiguration approaches [36,37,43,64-80], in general, the problem of how to tolerate the defects once they are located is addressed in each approach. Concurrent error correction cannot be achieved using these techniques. Also, the fault detection problem, requiring a totally different set of techniques such as voting and self testing is not discussed. In most cases, such fault detection techniques are specific to a given architecture and may not be applicable to another architecture. Additionally, the techniques may be expensive in terms of area and time overheads and as such cannot be applied to real-time systems. Since transient errors are becoming more frequent due to low supply voltages and decreased signal-to-noise ratios on VLSI chips [81], techniques are needed to tolerate faults concurrently with normal operation of the system. Concurrent error detection and correction techniques have been presented in the literatures [4,43,46,49-63]. The drawbacks of the CED schemes [4,5,43,49-61] include: there is a reduction in throughput by 50% [4,5,50-52,54], they allow the detection of only single faults, multiple fault detection requires high hardware overhead. Some of the techniques are limited to only those systolic implementations in which the data as well as (sub) results keep moving [4,5,50-52]. While some can only be applied to systolic arrays where data are stored in the cells [54,60].

In the case of error correction, the techniques proposed in [5,44-48] require, at least, hardware overhead of 200% without affecting the cost of the voter. Also the schemes in

[53,55,56,60,62,63] require high overhead to achieve fault-tolerance. The schemes in [4,46,52] are restricted to a class of systolic arrays where the data as well as the (sub) results move from one cell to another. In [54], partial results must stay in the cells for the scheme to work. There is a halving of the maximum effective output rate from the techniques proposed in [4,46,52,54]. The schemes in [4,52,53,62,63] are only effective for transient faults. They are not effective for permanent faults. Error correction requires high time overhead in [53,56,62,63]. In [46,54,55,56], the schemes cannot detect nor correct faults in the data paths of input/output registers. The correction latency is very long for the schemes in [53,55,56,62,63]. Also the schemes in [53,55,61-63] are vulnerable to false alarms brought about by round-off errors. The fault-tolerant design approaches in [46,56] are not systematic. The structure of the FT systolic array resulting from the design schemes in [53,56,62,63] are no longer regular and granular.

One of the major contribution of this thesis is the development of a methodology for designing reliable systolic arrays. we have proposed a novel approach to achieve concurrent error detection in systolic arrays. The methodology employs the space-time mapping techniques to design error detectable systolic arrays. In the proposed approach, redundant computations are introduced at the algorithmic level such that when the algorithm is mapped into systolic array, the redundancy in the array allows errors to be detected in the array. Finally, in addition to the proposed CED technique, another major contribution of the thesis is the development of a novel approach to design fault-tolerant systolic array architectures. The CED technique has been extended to design efficient fault-tolerant systolic arrays. Both the CED and FT approaches are systematic, the methodologies are cost-effective in terms of the hardware and time overhead. The proposed CED and FT schemes overcome most of the drawbacks of the existing schemes proposed in the literatures.

1.5 OUTLINE OF THE THESIS.

This thesis presents a new methodology for evaluating the performance and also the design of reliable systolic array architectures. The layout of the thesis is as follows: Chapter II contains the preliminary research on the design of systolic array architectures. Chapter III is concerned with the performance analysis and the design of optimal systolic arrays, which is based on the formulated optimization algorithm to measure the cost of the generated TDM's. Chapter IV discusses the systematic approach for designing fault-tolerant systolic array architectures. Some techniques to design area efficient computing structures for concurrent error detection and fault-tolerance in systolic arrays, are presented in Chapters V and VI respectively. In particular, the main contents of each chapter are briefly described as follows:

Chapter II : Preliminaries.

This chapter presents a comprehensive overview and some basic concepts for designing systolic arrays. Also the different methodologies for mapping algorithms into systolic arrays are described in this chapter.

Chapter III : Performance Analysis and Design of Optimal Systolic Arrays.

This chapter discusses the proposed methodology for obtaining the desired TDM directly from the original Dependency Matrix (DM). It also describes the formulation of a unifying performance index to measure the overall performance of the systolic arrays. The optimization algorithm that maps any algorithm into optimal systolic array is also introduced in this chapter.

Chapter IV : Systematic Approach for Designing Fault-Tolerant Systolic Array Architectures.

This chapter introduces a systematic approach for designing fault-tolerant systolic arrays. In particular, a new approach of designing fault-tolerant algorithms is described. It shows how redundancy can be incorporated at the algorithmic level such that when the algo-

rithm is mapped into an architecture, the resulting systolic array will be inherently fault-tolerant.

Chapter V : Area Efficient Computing structures for Concurrent Error Detection in Systolic Architectures.

Based on the new approach of designing fault-tolerant algorithms introduced in chapter IV, an area efficient technique for achieving Concurrent Error Detection in systolic array is proposed in this chapter.

Chapter VI : Area Efficient Fault-Tolerant Computing Structures for Systolic Arrays.

In this chapter, the Concurrent Error Detection technique proposed in Chapter V is extended to design area efficient fault-tolerant systolic arrays. The technique employed here is termed area-efficient in that it utilizes less area than the technique for achieving fault-tolerance, described in Chapter IV.

Chapter VII : Conclusions and Future Work

Basic contributions of the research described in this thesis are summarized in this chapter. Also, possible extensions of the results to specific, or perhaps new problems are discussed. A brief section on topics that have not been treated in this thesis is included as suggestions for contributions to this area of research.

1.6 REFERENCES

[1] A.L. DcCegama, The Technology of Parallel Processing. Parallel Processing Architectures and VLSI Hardware, Vol. I, *Prentice Hall*, Englewood Cliffs, New Jersey, 1989.

[2] H.T. Kung, "Why Systolic Architectures?," *IEEE Computer*, Vol. C-31, pp. 37-46, Jan, 1982.

[3] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)," *In Sparse Matrix Symposium*, pp. 256-282, SIAM, 1978.

[4] S.Y. Kung, VLSI Array Processors, *Prentice Hall*, 1988.

[5] J. V. Neuman, "Probabilistic Logics and Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, No. 34, pp. 43-99, Princeton, NJ · Princeton University Press.

[6] M. Annavatone et al., "Architecture of Warp," *1987 IEEE Conference Proceedings on Computer Architecture*, pp. 264-267, 1987.

[7] B. Bruegge *et al.*, "Programming Warp," *1987 IEEE Conference Proceedings on Computer Architecture*, pp. 268-271, 1987.

[8] M. Annavatone *et al.*, "Applications and Algorithm Partition on Warp," *1987 IEEE Conference Proceedings on Computer Architecture*, pp 272-275, 1987.

[9] R.M. Karp, R.E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of ACM*, 14(3), pp. 563-590, July, 1967.

[10] P. Quinton, "The Systematic Design of Systolic Arrays," *IRISA Research Report*, No. 193, March 1983.

[11] P. Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *In Proceedings of 11th Annual Symposium on Computer Architecture*, pp 208-214, 1984.

[12] P. Gachet, B. Joinnault and P. Quinton, "Synthesizing Systolic Arrays using DIASTOL," Systolic Arrays (edited by W. Moore, A. McCabe and R. Urquhart), pp.25-36, *Adam Hilger Ltd.*, Bristol, UK, 1987.

[13] S.K. Rao, "Regular Iterative Algorithms and their Implementation on Processor Arrays," Ph.D thesis, Stanford University, Stanford, California, 1985.

[14] S.Y. Kung, "From Transversal Filter to VLSI Wavefront Array", *In Proc. Int'l Conf. on VLSI 1983, IFIP*, Trondheim, Norway, 1983.

[15] J-M. Delosme and I.C.F. Ispen, "Efficient Systolic Arrays for the Solution of Toeplitz Systems: An illustration of a Methodology for the construction of Systolic Architectures in VLSI," *Int'l Workshop on Systolic Arrays*, University of Oxford, pp. F2, July, 1986. Also in Systolic Arrays: edited by W. Moore, A. McCabe and R. Urquhart, pp. 27-46, 1987.

[16] M.C. Chen, "A Synthesis Method for Systolic Designs," *Technical Report 334*, Yale University, March, 1985.

[17] M.C. Chen, "Synthesizing Systolic Designs," *Technical Report 374*, Yale University, March, 1985.

[18] M.C. Chen, "Synthesizing VLSI Architectures: Dynamic Programming Solver," *In Int. Conf. in Parallel Processing*, pp. 776-784, Chicago, IL, August, 1986.

[19] J.A.B. Fortes, K.S. Fu and B.W. Wah, "Systematic Approaches to the Design of Algorithmic Specified Systolic Arrays," *In Proc. IEEE ICASSP '85*, pp. 300-303, Tampa, Florida, March, 1985.

[20] P.R. Capello and K. Steiglitz, "Unifying VLSI array Designs with Geometric Transformations," *In Int'l Conf. on Parallel Processing*, 1983.

[21] U. Weiser and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," in *VLSI Systems and Computations*, Rockville, Maryland: Computer Science Press, 1981.

[22] D. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Trans. on Computers*, vol C-31, No. 11, November 1982.

[23] D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proc. IEEE*, Vol. 71, No. 1, January 1983.

[24] W. L. Miranker and A. Winkler, "Spacetime Representations of Computational Structures," *Journal of Computing*, Vol. 32, 1984.

[25] H. F. Li et al., "A Systematic Approach for Mapping Algorithms into Systolic Arrays," Technical Report, Dept. of Comp. Sc., Concordia University, Montreal, Quebec, Canada.

[26] D. I. Moldovan, "Tradeoffs Between Time and Space Characteristics in the Design of Systolic Arrays," *Proc. of ISCAS*, 1985.

[27] D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays'" *IEEE Trans. Comput.*, Vol. C-35, No. 1, pp. 1-12, January 1986.

[28] D. I. Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays," *IEEE Trans. Comput.-Aided Design*, Vol. CAD-6 January 1987.

[29] M.C. Chen, "Space-Time Algorithm: Semantics and Methodology," Ph.D thesis, Computer Science Department, California Institute of Technology, 1983.

[30] Y. Wong and J-M Delosme, "Optimal Systolic Implementations of N - dimensional Recurrences," *ICCD*, pp. 618-621, 1985.

[31] M. T. O'Keefe and J. A. B. Fortes, "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," *In International Conference on Parallel Processing*, pp. 672-675, Chicago, IL, August, 1986.

[32] J. A. B. Fortes, "Algorithm Transformations for Parallel Processing and VLSI Architecture Design," Ph.D. dissertation, Univ. Southern California, Los Angeles, CA., Dec., 1983.

[33] C. E. Leiserson, F. M. Rose and J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *in Proceedings, Caltech VLSI Conference, Pasadena, CA, 1983.*

[34] C. S. Raghavendra, V. K. Prasanna Kumar and A. Varma, "On systolic processing with bounded I/O bandwidth," *in Proc. ICCD*, 1985.

[35] I. V. Ramakrishnan and P. J. Varman, "Synthesis of an optimal family of matrix multiplication algorithms on linear arrays," *Tech. Rep.*, Univ. of Maryland, Dept. Comput. Sci., *Proc. ICPP*, 1985.

[36] P. J. Varman and I. V. Ramakrishnan, "Optimal matrix multiplication on fault-tolerant VLSI array," *Proc. ICALP*, 1985.

[37] V. K. P. Kumar and Y-C Tsai, "On Mapping Algorithms to Linear and Fault-tolerant Systolic Arrays," *IEEE Trans. on Comput.*, Vol. 38, No. 3, pp. 470-478, March, 1989.

[38] G - J Li and B. W. Wah, "The Design of Optimal Systolic Arrays," *IEEE Trans. Comput.*, Vol. C-34, No. 1, pp 66-77, January 1985.

[39] C. K. Ko and O. Wing, "Mapping Strategy for Automatic Design of Systolic Arrays," *in Proc. 1988 International Conf. on Systolic Arrays*, pp. 285-294, 1988.

[40] H. B. Lee and R. O. Grondin, "A Comparison of Systolic Architectures for Matrix Multiplication," *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 1, pp. 285-289, February 1988.

[41] R. W. Keyes, "Physical Limits in Digital Electronics," *Proc. IEEE*, Vol. 63, No. 5, pp. 740-767, May, 1975.

[42] R. O. Grondin, W. Porod and D. K. Ferry, "Delay Time and Signal Propagation in Large - Scale Integrated Circuits," *IEEE J. Solid - State Circuits, Vol. SC-19, No. 2, pp.263-263, April, 1984.*

[43] J. A. Abraham, P. Banerjee, C-Y Chen, W. K. Fuchs, S-Y Kuo and N. Reddy, "Fault - Tolerance techniques for systolic arrays," *IEEE Computer*, pp. 65-74, July

1987.

[44] P. K. Lala, *Fault Tolerant and Fault Testable Hardware Design*, Prentice Hall International, 1985.

[45] T. Anderson and P. A. Lee, *FAULT TOLERANCE - Principles and Practice*, Prentice Hall, 1981.

[46] J-H Kim and S.M. Reddy, "A Fault-Tolerant Systolic Array Design using TMR Method," *1985 ICCD*, pp. 769-773.

[47] M. O. Esonu, S. Hariri and A. J. Al-Khalili, "A Systematic Approach for Designing Fault - Tolerant Systolic Architectures," in *Proc. 1989 Joint Tech. Conf. on Circuits/Systems, Comput. and Communications*, Sapporo, Japan, June 25-27, 1989.

[48] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "Variation on the Theme for Designing Fault-Tolerant Systolic Array Architectures," *Pacific RIM Conference on Communications*, Victoria, B.C., May, 1991.

[49] J. H. Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. Comput.*, Vol C-31, pp. 589-595, 1982.

[50] R. K. Gulati and S. M. Reddy, "Concurrent Error Detection in VLSI Array Structures," *Proc. IEEE Intl. Conf. on Computer Design*, pp. 488-491, 1986.

[51] C-C Wu and T-S Wu, "Concurrent Error Correction in Unidirectional Linear Arithmetic Arrays," *Proc. 17-th Intl. Symp. on Fault-Tolerant Computing*, pp. 136-141, 1987.

[52] E. S. Manolakos, "Transient Fault Recovery Techniques for the VLSI Processor Arrays," Ph.D thesis, University of Southern California, May, 1989.

[53] K. H. Huang and J. A. Abraham, "Algorithm-based fault-tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, pp. 518-528, June 1984.

[54] R. J. Cosentino, "Concurrent Error Correction in Systolic Architectures," *Proc. IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 1, pp. 117-125, January 1988.

[55] R. J. Cosentino, "Fault Tolerance in a Systolic Residue Arithmetic Processor Array," *IEEE Trans. on Comput.*, vol. 37, No. 7, pp. 886-890, July, 1988.

[56] H. Lev-Ari and B. Friedlander, "On the Systematic Design of Fault-Tolerant Processor Arrays with Application to Digital Filtering," *VLSI Signal Processing III*, pp. 483-493, 1988.

[57] S-W Chan and C-L Wey, "The Design of Concurrent Error Diagnosable Systolic Arrays for Band Matrix Multiplication," *Proc. IEEE Trans. on Computer-Aided Design*, Vol.7, No.1, pp. 21-37, January 1988.

[58] J. H. Patel and L. Y. Fung, "Concurrent Error Detection in Multiply and Divide Arrays," *IEEE Trans. Comput.*, vol. C-32, No. 4, pp. 417-422, April 1983.

[59] W-T Cheng and J. H. Patel, "Concurrent Error Detection in Iterative Logic Arrays," *FTCS*, pp. 10-15, June 1984.

[60] S. R. Gupta and M. A. Bayoumi, "Concurrent Error Detection In Systolic Arrays For Real-Time DSP Applications," *VLSI Signal Processing III*, edited by Robert W. Brodersen and Howard S. Moscovitz, IEEE Press, 1988.

[61] H. F. Li, C. N. Zhang and R. Jayakumar, "Latency of Computational Data Flow and Concurrent Error Detection in Systolic Arrays," *Canadian Conf. on Very Large Scale Integration (CCVLSI)*, pp. 251-258, 1989. Canada.

[62] K-H Huang and J. A. Abraham, "Fault-Tolerant Algorithms and their Application to Solving Laplace Equations," *IEEE Int'l Conf. Parallel Processing*, pp. 117-122, August, 1984.

[63] J-Y Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE*, vol.74, No. 5, pp. 732-741, May, 1986.

[64] I. Koren and M. A. Breuer, "On Area and Yield Considerations for Fault-Tolerant VLSI Processor Arrays," *IEEE Trans. on Comput.*, Vol. C-33, No. 1, pp. 21-27, Jan., 1984.

[65] J-H Kim and S. M. Reddy, "On the Design of Fault-Tolerant Two-Dimensional Systolic Arrays for Yield Enhancement," *IEEE Trans. Comput.*, Vol. 38, No. 4, April 1989.

[66] J. I. Raffel *et al.*, "A Demonstration of Very Large Area Integration Using Lesser Destructuring," *IEEE International Symposium on Circuits and Systems*, May 1983.

[67] P. E. Blonkenship, "Restructurable VLSI Program," *Semiannual Technical Summary*, ESD_TR_81_153, MIT Lincoln Lab, March 1989.

[68] H. T. Kung and M. S. Lam, "Fault-Tolerance and Two Level Pipelining in VLSI Systolic Arrays," *MIT Conference on ADV Research in VLSI*, pp. 74-83, Jan. 1984.

[69] F. T. Leighton and C. E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Trans. Computers*, Vol. C-34, pp. 448-461, May 1985.

[70] P. J. Varman and I. V. Ramakrishnan, "A Fault-Tolerant VLSI Matrix Multiplier," *ICPP*, pp. 351-357, August, 1986.

[71] T. Ishikawa, S. Momoi, S. Shimada, Y. Ogawa, "Hierarchical Array Processor (HAP) Featuring High Reliability and High System Performance," *ICPP*, pp. 293-300, August, 1986.

[72] F. Lombardi, R. Negrini, M. G. Sami, and R. Stefanelli, "Reconfiguration of VLSI Arrays: A Covering Approach," *FTCS*, pp. 251-256, July, 1987.

[73] D. L. Landis, W. A. Check and D. C. Muha, "Influence of Built-In Self-Test on the Performance of Fault-Tolerant VLSI Multiprocessors," *ICPP*, pp. 114-116, August, 1987.

[74] H. F. Li, R. Jayakumar and C. Lam, "Restructuring for Fault-Tolerant Systolic Arrays," *IEEE Trans. on Comp.*, vol. 38, No. 2, pp. 307-311, Feb., 1989.

[75] D. S. Fussell and P. J. Varman, "Designing Systolic Algorithms For Fault-Tolerance," *Proc. of the IEEE Int'l Conf. on Comp. Design: VLSI in Comp.*, pp. 617-622, 1984.

[76] Y-H Choi, S. H . Han and M. Malek, "Fault Diagnosis of Reconfigurable systolic arrays," *ICCD '84*, pp. 451-455, 1984.

[77] M. S. Lee and G Frieder, "Massively Fault-tolerant Cellular Array," *IEEE Int'l Conf. Parallel Processing*, pp. 343-350, 1986.

[78] J. H. Hwang and C. S. Raghavendra, "VLSI Implementation of Fault-Tolerant Systolic Arrays," *ICCD '86*, pp. 110-113, 1986.

[79] J. H. Kim and S. M. Reddy, "On Easily Testable and Reconfigurable Two-Dimensional Systolic Arrays," *ICPP '87*, pp. 101-109, 1987.

[80] L. A. Shombert and D. P. Siewiorek, "Using Redundancy for Concurrent Testing and Repairing of Systolic Arrays," *FTCS*, pp. 244-249, July 1987.

[81] D. F. Barbe, "VHSIC Systems and Technology," *Computer*, pp. 13-22, February, 1981.

CHAPTER II

PRELIMINARIES

This chapter gives a comprehensive overview of systolic array architectures and presents some basic concepts for designing systolic arrays. The chapter consists of three main sections: Section 2.1 discusses the basic principle of systolic architecture and gives a coherent definition of systolic arrays. It also discusses the justification of why systolic arrays are preferred architectures for executing many algorithms, by presenting the major factors of adopting systolic arrays for special - purpose processing architectures. Furthermore, the section describes the basic components of systolic array structures and then concludes by giving the applications of systolic arrays.

Section 2.2 discusses methods of mapping algorithms into systolic arrays. It reviews the different mapping methodologies and also, introduces the notations and defines relevant terminologies associated with the Space - Time mapping procedure. Section 2.3 describes in detail one of the mapping methodologies, which many research works on systolic designs are based on. This is a systematic method of mapping algorithms into systolic arrays based on the linear transformations of index set and data dependencies. In order to illustrate this approach, a systolic design example for matrix multiplication algorithm is presented.

2.1 SYSTOLIC ARRAY ARCHITECTURES : AN OVERVIEW

Systolic processors are a class of pipelined array architectures [1,2]. As described by Kung and Leiserson [1], *A systolic system is a network of processors which rythmically compute and pass data through the system.* Some basic simple processing elements (PE's) can be locally connected together to perform some simple signal processing

operations and/or other related operations. Systolic array features the important properties of *modularity, regularity, local interconnection, a high degree of pipelining and highly synchronized multiprocessing*. The data movements in a systolic array are often described in terms of the *snapshots* of the activities [1,2]. Information in a systolic system flows between cells in a pipeline fashion, and communication with the outside world occurs only at the boundary cells. Only those cells on the array boundaries may be I/O ports for the systolic system.

Computational tasks can be conceptually classified into two families: *compute - bound* computations and *I/O - bound* computations [2]. In compute - bound computations, the total number of computations is larger than the total number of input and output operations. While in the I/O - bound computations, the reverse is the case. For example, ordinary matrix multiplication is compute - bound, whereas adding two matrices is I/O - bound. Speeding up the I/O - bound computations requires an increase in memory bandwidth, which may be difficult to achieve given the present technologies. Speeding up a compute - bound computation, however, may often be accomplished by using systolic arrays.

The basic configuration of a systolic array is illustrated in Fig.2.1 . By replacing a single processor element with an array of PE's, higher computation throughput can be achieved without increasing memory bandwidth. The function of the memory in the Figure is to pump data through the array of cells. The essence of this approach is to ensure that, once a data item is brought into the array from the memory, it can be used effectively at each cell it passes through while being moved step by step from cell to cell along the array. This is especially appealing for a wide class of compute - bound computations, where multiple operations are performed on each data item in a repetitive manner.

Being able to use each input data item a number of times is just one of the many advantages of the systolic approach. Other advantages include, modular expandability, simple and regular data and control flows, and the use of simple and uniform cells.

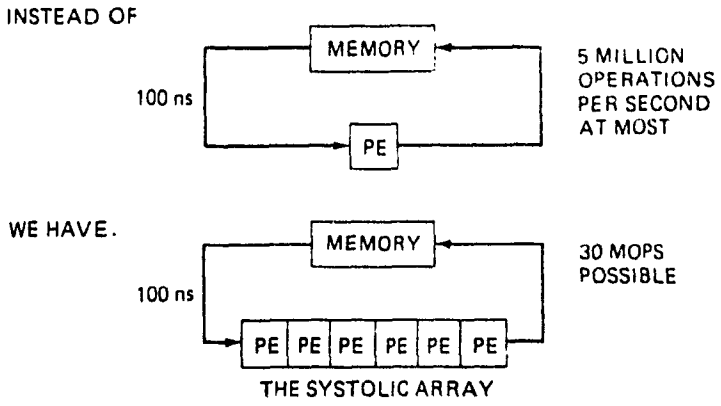


Figure 2.1 Basic principle of a systolic system. [3]

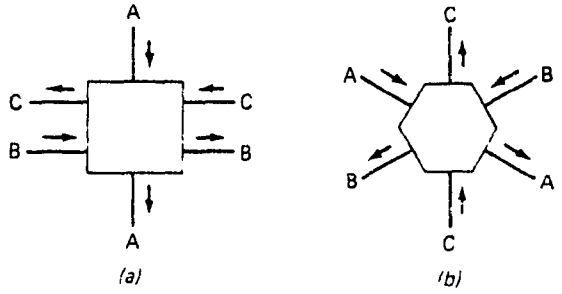


Figure 2.2 Inner product step processors. [3]

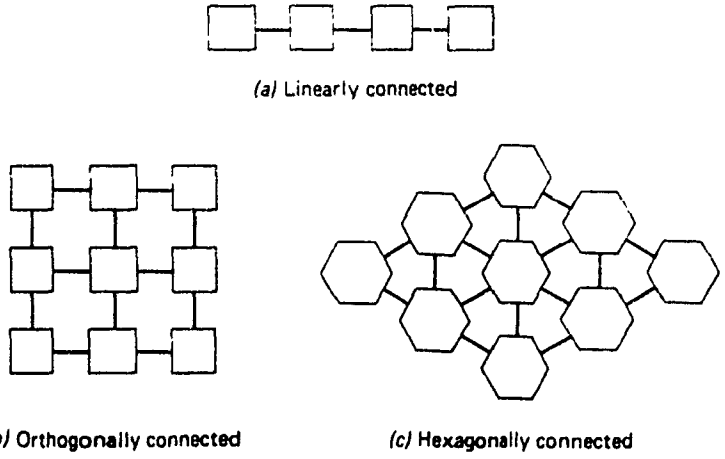


Figure 2.3 Systolic array systems. [3]

Simple and regular interconnections lead to cost-effective implementations and high densities. High density on the other hand, implies both high performance and low overhead for support components. Due to these reasons, multiprocessor structures which have simple and regular communication paths have been considered interesting architectures for modern signal and image processing applications. Also, the use of pipelining as a general method for applying these structures is logical and attractive. By pipelining, computation may proceed concurrently with input and output operations, consequently, minimizing the overall execution time. Systolic arrays thus take advantages of the concepts of pipelining, parallelism and regular interconnection structures [3].

2.1.1 Definition of Systolic Arrays

A number of definitions of systolic arrays are given in the literature [1,4,5], however, a coherent definition given in [6] is as follows:

Definition 2.1 : A systolic array is a computing network possessing the following features:

- *Synchrony* :- The data are rhythmically computed (timed by a global clock) and passed through the network.

- *Modularity and regularity* :- The array consists of modular processing units with homogeneous interconnections, the computing network can be extended easily.

- *Spatial locality and temporal locality* :- The array manifests a locally communicative interconnection structure, i.e. spatial locality. There is at least one unit-time delay alloted so that signal transaction from one cell to the next can be completed, i.e. temporal locality.

- *Pipelinability* :- The array exhibits a linear rate pipelinability, i.e., in terms of the processing rate, it should achieve an $O(m_{PE})$ speedup, where m_{PE} is the number of processing elements (PE's).

2.1.2 Properties of Systolic Architectures

2.1.2.1 Why Systolic Arrays are Preferred Architectures for Executing Many Algorithms

The major factors for adopting systolic arrays for special - purpose processing architectures are first, simple and regular design, second, concurrency and communication, and balancing computation with I/O [1,2]. For the first factor, with the advance in integrated circuit technology, the cost of the processing elements has decreased dramatically, however, the cost of design grows with the complexity of the system. By exploiting the VLSI technology and using a regular and simple design, tremendous savings in design cost can be achieved. In addition, simple and regular systems are likely to be modular and therefore adjustable to various performance goals.

Another important factor is concurrency which is very essential to achieve high-speed computing systems. The concurrency depends on the underlying algorithms employed by the system, especially for special - purpose systems. When a large number of processors work together, communication becomes very important and significant. Routing cost dominate the amount of power dissipation, time and area required to implement a computation [7], therefore, regular and local communication in systolic arrays offers a great advantage.

The I/O problem is especially severe when the computation of a large dimension problem is performed on a small array. It involves a partitioning problem, that is, the

computation must be decomposed. Since this is the case in practice, therefore, the questions that are critical to the practical design of an array processor system, are how the computation can be decomposed and how the buffer memory can be arranged to minimize I/O. A systolic array is typically used as an attached array processor, and it receives data and output results through a host computer. Therefore, I/O considerations have to be taken into account in the overall performance. The ultimate performance goal of an array processor system is a computation rate that balances the available I/O bandwidth with the host. With the relatively low bandwidth of current I/O devices, to achieve a faster computation rate, it is necessary to perform multiple operations per I/O access. Since in systolic arrays, multiple operations are performed on each data item in a repetitive manner as the data is pumped into the array from the memory, therefore, available I/O bandwidth is balanced with the computation rate. As a result, systolic arrays are preferred architectures for performing compute - bound computations.

2.1.3 Components of Systolic Array Structures

The single operation common to the computations of most signal processing algorithms is the so - called inner product step. For example, $C \leftarrow C + A \times B$. Figure 2.2 shows two types of geometries of a processor (cell) that can be used for such computations. In each type, the simple processor has three registers R_A, R_B and R_C, and each register has two connections, one for input and the other for the output. The type *a* geometry in Fig.2.2 is used for the computations of the algorithms such as the matrix - vector multiplication and the LU - decomposition. While the type *b* geometry can be used for the computation of matrix - matrix multiplication algorithm. Each of the processors is capable of performing the inner product step and they are called the *inner product step processors*.

The operation performed by the inner product processor is described as follows: In each time interval, the processor shifts the data on its input lines denoted by A, B and C

into R_A, R_B and R_C, respectively. The processor computes $R_C \leftarrow R_C + R_A \times R_B$, and makes the input values for R_A and R_B together with the new value of R_C available as outputs on the lines denoted by A, B and C, respectively. The inputs and the functional block in the processor are clocked in such a way that when one processor is connected to another, the change in one processor's output will not interfere with the input to another during this time interval.

A systolic array is typically composed of many inner product step processors connected as a mesh (using type a geometry) in which all connections to a processor are to neighboring processors. A *hexagonally* systolic array structure is formed using the type b inner product processors. Different types of connections are as shown in Fig.2.3. The input/output data path of a boundary processor may sometimes be designated as an external input/output connection for the array. A boundary processor may receive input from the host memory through an external connection. On the other hand, a boundary processor can send data to the host memory through an external output connection. The processors in a systolic array are synchronous. They are simple and uniform, interprocessor connections are simple and regular and external connections are minimized.

One example of the algorithms that can be implemented using systolic arrays is the Matrix - Vector Multiplication algorithm. The problem is that of multiplying a matrix $A = (a_{ij})$ with a vector x represented by the transpose $(x_1, x_2, \cdots, x_n)^t$. The elements of $y = (y_1, y_2, \ldots, y_n)^t$ are computed by the recurrence given in [7]. There are many ways to implement this algorithm using systolic arrays. In one particular version, the recurrences can be evaluated by pipelining the x_i and y_i through a systolic array consisting of linearly connected inner product step processors, as illustrated in Fig.2.4. Figure 2.5 shows the first seven pulsations of the systolic array.

Another example of an algorithm is the Matrix Multiplication algorithm. This is the problem of multiplying two $n \times n$ matrices. Also, the matrix product $C=(c_{ij})$ of $A=(a_{ij})$ and $B=(b_{ij})$ can be computed by the recurrences in [7].

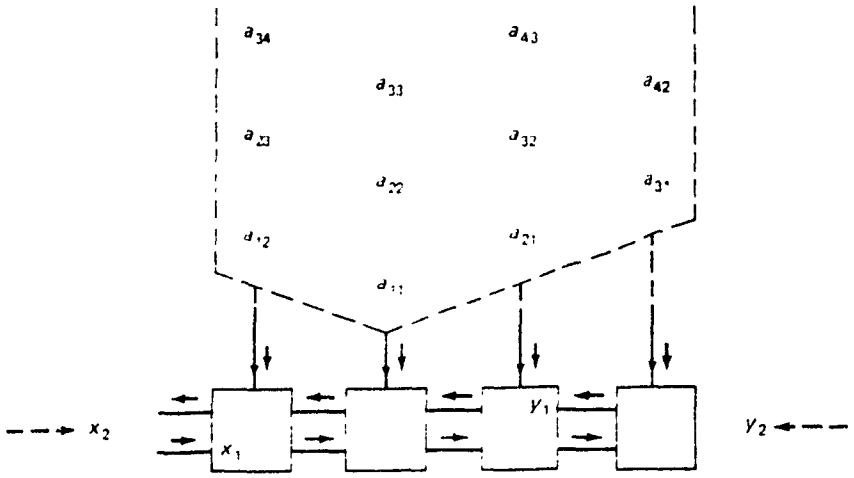


Figure 2.4 Systolic array for multiplication of a vector by a band matrix. [3]

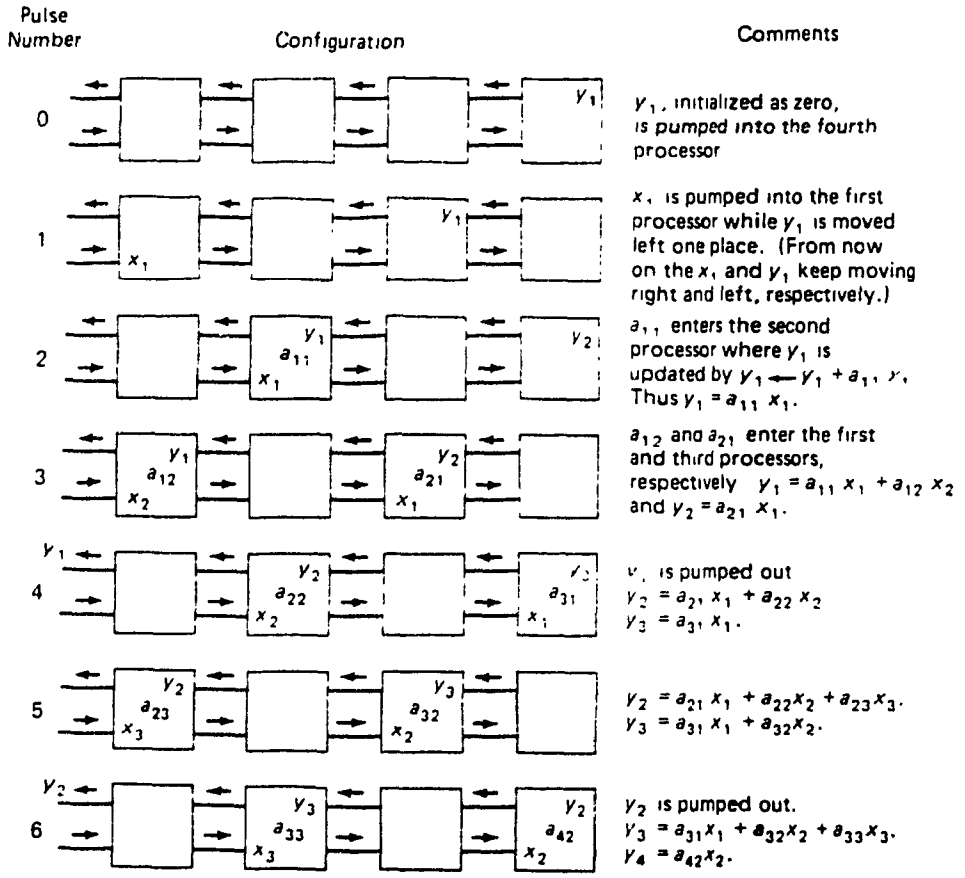


Figure 2.5 The first seven pulsations of the systolic array of Figure 2.4. [3]

One particular way of solving this problem is to evaluate the recurrences by pipelining the a_{ij}, b_{ij} and c_{ij} through a systolic array having hexagonally connected inner product step processors, as shown in Fig.2.6. The elements of matrices A, B and C are pumped through the systolic array in three directions synchronously. Figure 2.7 shows four consecutive pulsations of the hexagonal systolic array.

2.1.4 Applications of Systolic Arrays

In the above examples, the sizes of the systolic arrays required for matrix/vector computations depend only on the bandwidths of the band matrices to be processed and are independent of the lengths of the bands. Therefore, a fixed-size systolic array can pipeline band matrices with arbitrarily long bands. The pipeline aspect of systolic arrays is most effective for matrices with long bands. Band matrices are interesting since many important scientific computations involve them, however, it should be noted that the same techniques apply to dense matrices, since they are regarded as band matrices with maximum possible bandwidth.

Many projects developing systolic arrays for special-purpose applications have been reported in the literature [8, 9, 10]. Some of the applications of these special-purpose systolic architectures include linear algebra computations of the type described in the above examples, various signal processing algorithms, robot control and medical image processing. Other applicational domain of systolic arrays covers computer vision, nuclear physics, structure analysis, analysis of speech, sonar, radar, seismic, weather and astronomical computations and so on.

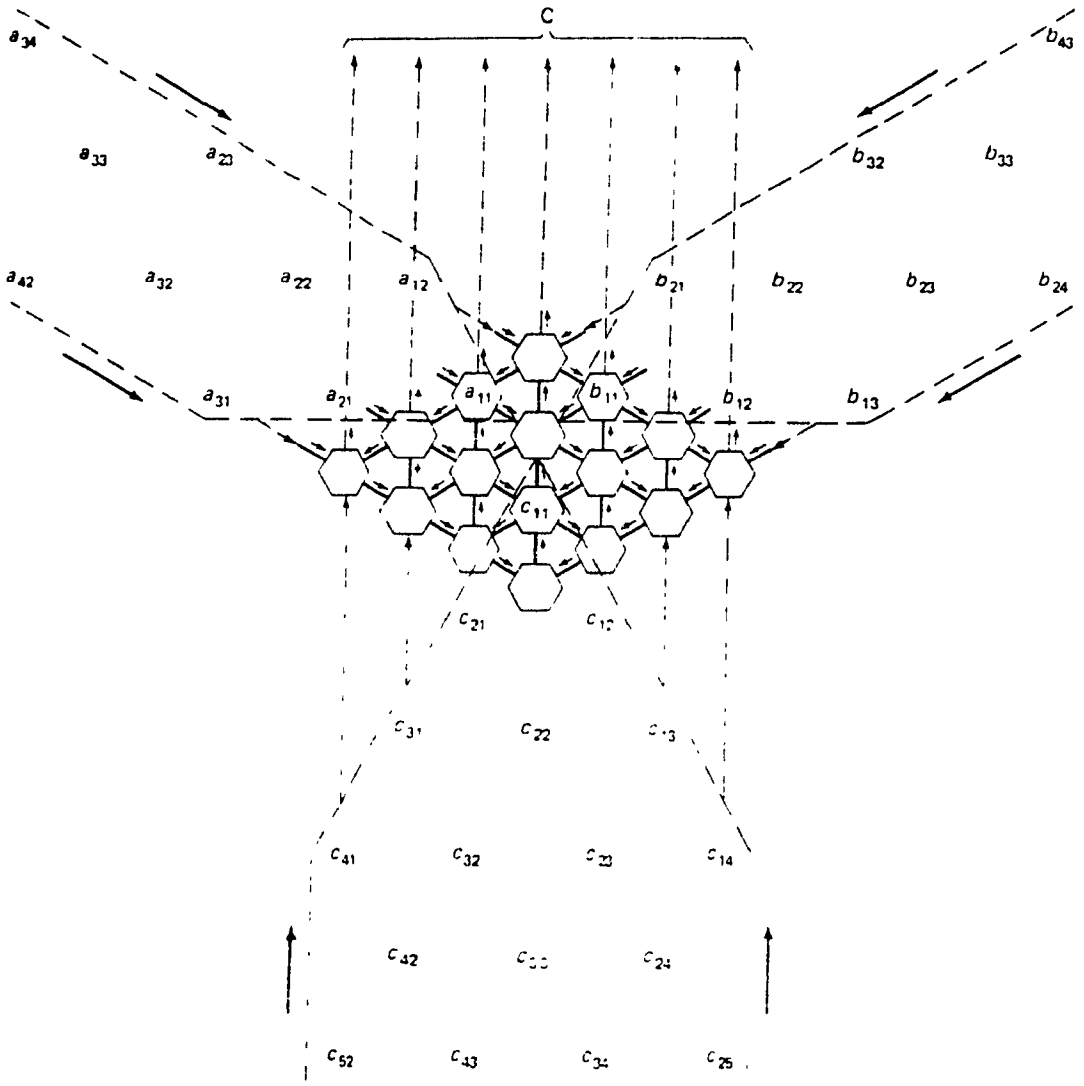


Figure 2.6 Systolic array for band matrix multiplication. [3]

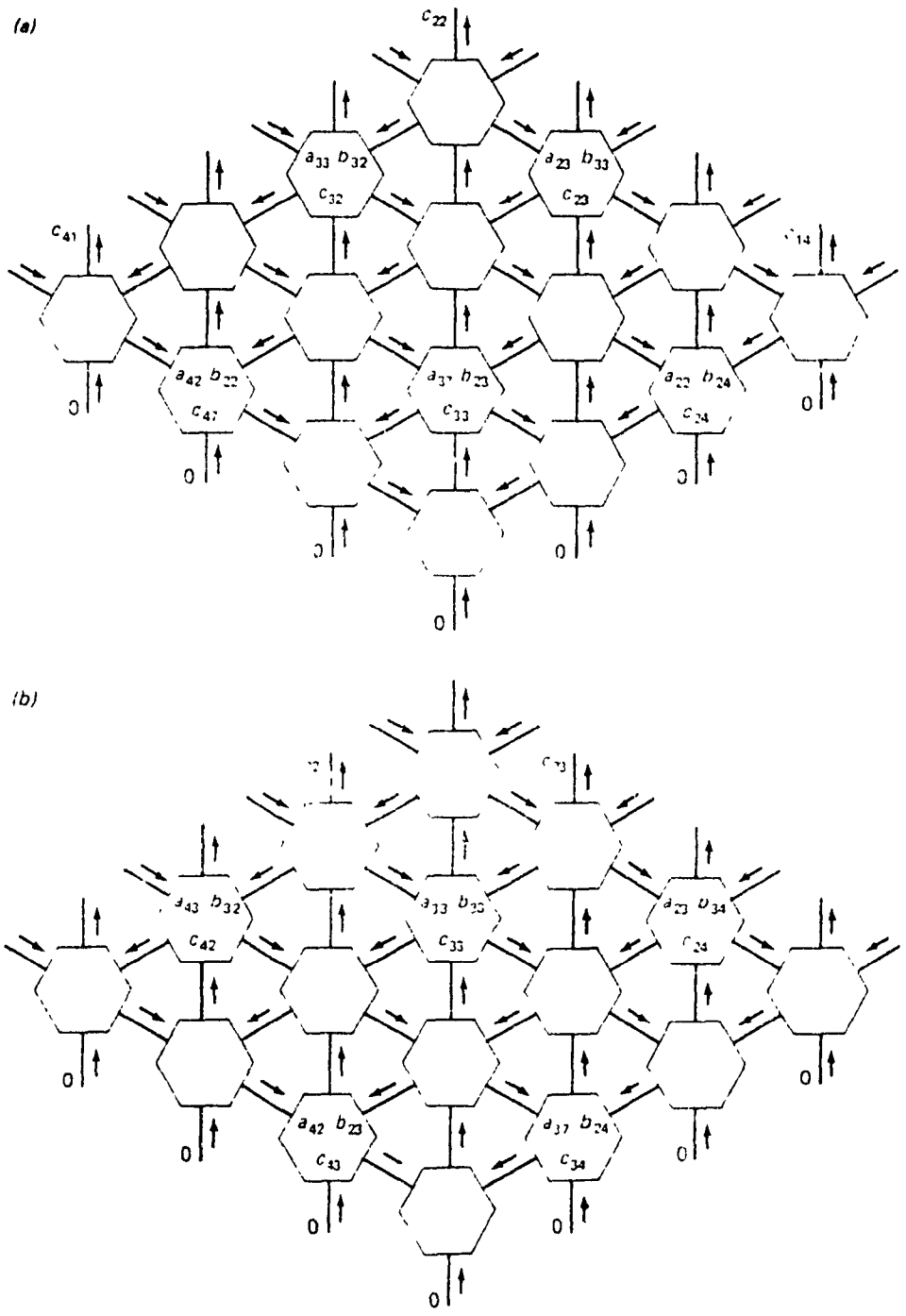


Figure 2.7(a) Two pulsations of the systolic array of Figure 2.6. [3]

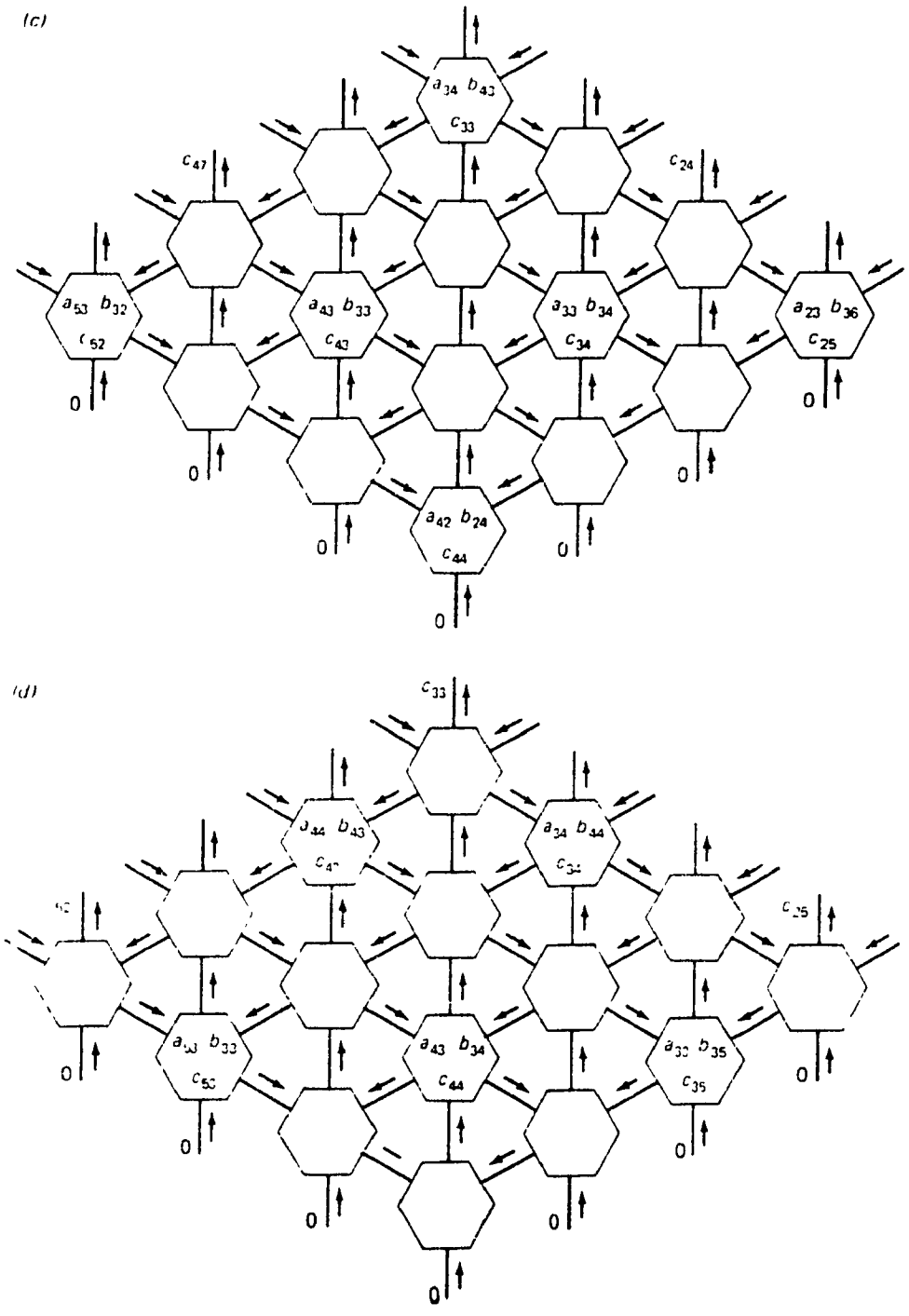


Figure 2.7(b) The next two pulsations of the systolic array of Figure 2.6. [3]

2.2 MAPPING ALGORITHMS INTO SYSTOLIC ARRAY ARCHITECTURES.

Many known designs of systolic array architectures are based on a heuristic approach. However, there has been considerable effort in the development of systematic methods for synthesizing systolic arrays based on algorithm-oriented analyses. As mentioned in the previous section, systolic arrays are preferred architectures for implementing many signal and image processing algorithms. The features of many algorithms used in signal and image processing include localized operations, intensive computation and matrix operations. In order to facilitate the design of special-purpose signal/image array processors, the common features of these algorithms should be exploited. Important issues associated with the design of these special-purpose arrays are how to *express* this special class of algorithms and a systematic method to *transform* an algorithm description to array processor. Algorithm expression is a basic tool for a proper description of an algorithm for parallel and pipeline processing. There are quite a number of research efforts devoted to the formal description of space-time activities in systolic array processors [2,11]. Parallel algorithm expressions may be derived by two approaches: Vectorization of sequential algorithm expressions and Direct parallel algorithm expressions, such as single assignment code, parallel codes, recursive equations, snapshots, dependence graphs and so on. A major factor in selecting an algorithm expression is that it should express algorithms clearly and concisely.

There exist abundant sequential codes for signal/image processing and scientific computing. A conventional approach to concurrent execution is by using a vectorizing compiler. A vectorizing compiler processes a source code written in a sequential language, and, where possible, generates parallel machine instructions. In fact, detecting and analyzing the dependencies between statements within loops is the major task in vectorization [12]. However, since a vectorizing compiler may not be sufficiently effective in extracting the inherent concurrent (parallel and pipeline) processing, it is advantageous

that a user/designer use parallel expressions to describe an algorithm in the first place. This is a key step leading to an *algorithm-oriented array processor design.*

One such parallel expressions is the *Single Assignment Code.* A single assignment code is a form where every variable is assigned one value only during the execution of the algorithm.

A single assignment code is in a sharp contrast to a conventional Fortran code, which is, in general, not written in a single assignment form. For example, consider the following matrix-vector multiplication algorithm.

```
      DO  10  I = 1,4
      C(I) = 0
      DO  10  J = 1,4
      C(I) = C(I) + A(I,J) * B(J)
10    CONTINUE
```

Note that in this program, the value of C(I) is assigned more than once. It is overwritten many times to save storage space. In order to transform the above program to a **single assignment** code, the number of indices of vector C is increased. The FORTRAN program thus obtained is the same as the program above except that the statement { C(I) = C(I) + A(I,J) * B(J) } is replaced by { C(I,J+1) = C(I,J) + A(I,J) * B(J) } and C(I)=0 by C(I,1)=0. Where A and C are 4 x 4 matrices and B is a 4 x 1 vector. Since each element of C will be assigned one value only, thus, this program is indeed a single assignment code. At each index point, three variables A, B and C are defined with no ambiguity.

Another convenient and concise expression for the representation of many algorithms is to use *recursive equations.* A recursive equation with space-time indices uses one index for time and the other indices for space. By so doing, the activities of a parallel algorithm can be adequately expressed. *Snapshot* can also be used to express parallel algorithms. A snapshot is a description of the activities at a particular time instant. Snapshots are perhaps the most natural tool an algorithm designer can adopt to check or

verify a new array algorithm. Sample snapshots for the systolic matrix-vector multiplication algorithm are depicted in Fig.2.8. *Dependence Graph* (DG) is another and most important parallel expression of algorithms. In order to achieve the maximal parallelism in an algorithm, one must carefully study the data dependencies in the computations. In the special case when the operations of a sequential algorithm have no data dependencies between each other, they can be executed at the same time in a parallel computer. However, in general, there is always a certain degree of dependency which dictates the sequence of computation. These data dependencies can be represented in a graphical form called *Dependence Graph* (DG). In essence, a DG is the graphical representation of the data dependencies in the computations. In the previously mentioned single assignment algorithm, C(I,J+1) is said to be directly dependent upon C(I,J), A(I,J), and B(J). By viewing each dependence relation as an arc between the corresponding variables located in the index space, a DG as shown in Fig.2.9(a) will be obtained. Only the dependencies between the nodes are shown in Fig.2.9. The operations inside each node are not shown, since they will be assigned to the same processing element when the DG is used to map an algorithm to an array processor. However, it is straightforward to extend the DG concept to include the operations inside each node. This DG, which is called a *complete* DG, specifies all the dependencies between all variables in the index space. A computation whose DG has loops or cycles will require infinite amount of time to be completed. It cannot be computed within a reasonable time complexity and hence, it is not *computable*. Thus, an algorithm is *computable* if and only if its complete DG contains no loops or cycles[6]. Therefore, the complete DG is very useful for studying issues related to computability. In general, global communication (broadcasting) is involved in array processor design. In many cases, such broadcasting can be avoided and replaced by local communication[6].

At this point, we will define some of the relevant terminologies that have been encountered so far in this section.

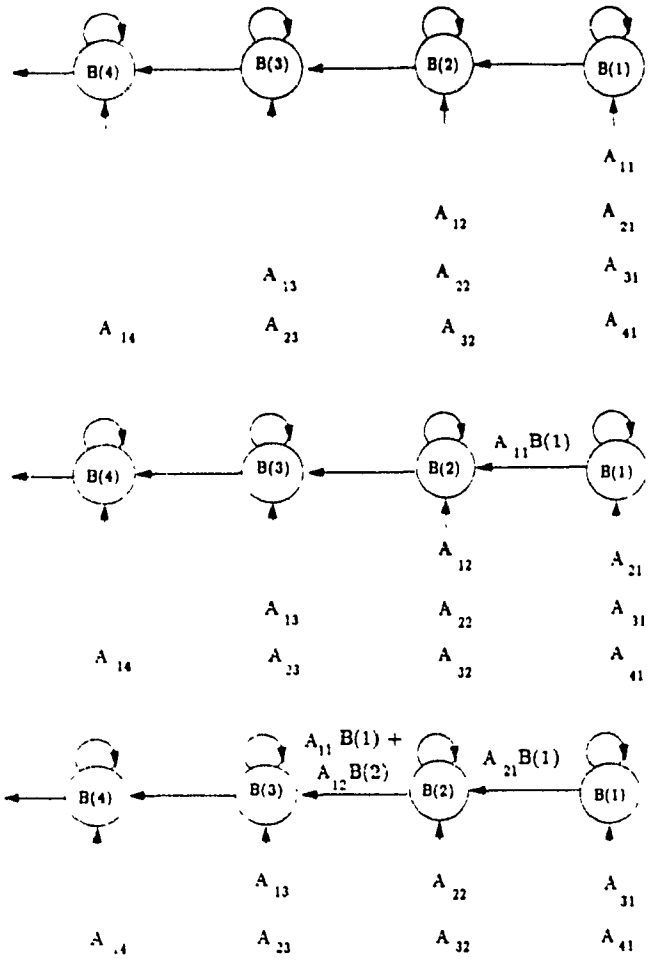


Figure 2.8 Snapshots for a systolic matrix–vector multiplication algorithm. [6]

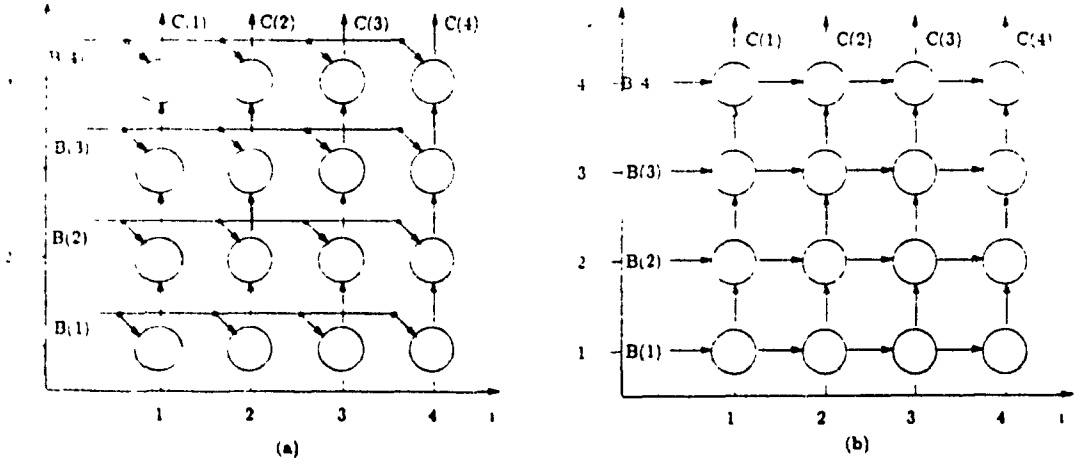


Figure 2.9 DG for matrix–vector multiplication
(a) with global communication
(b) with only local communication. [6]

Definition 2.2 : *Graph Terminologies.* A *graph* $G=|N,A|$ is a set N whose elements are called *nodes* and a set A whose elements are called *arcs* or *edges*. Each arc, $a \in A$, connects a pair of nodes $i,j \in N$ and is written $i \longrightarrow j$. Here i is the *initial endpoint* of a and j is the *terminal endpoint* of a. An arc whose endpoints are the same node is called a *loop*. A *chain* is a sequence of arcs, $L = a_1,....,a_q$, such that arc a_r $(2 \le r \le q-1)$ has one endpoint common with arc a_{r-1} $(a_r \neq a_{r-1})$ and its second endpoint common with arc a_{r+1} $(a_r \neq a_{r+1})$, without regard to the direction of the arcs. The free endpoints of the first and the last arcs of a chain are the endpoints of the chain. A *path* is a chain, all of whose arcs are directed the same way. If the endpoints of a path are the same node, then it is a *cycle*. If the endpoints of a chain are the same node, then it is an *undirected cycle*. An *elementary* chain, path, cycle, undirected cycle is one in which the same node is not encountered twice (with the exception of the endpoints). A graph is *connected* if there exists a chain between every pair of nodes. It is *strongly connected* if there exists a path from each node to all other nodes.

Definition 2.3 : *Dependence Graph.* A dependence graph is a graph that shows the dependence of computations that occur in an algorithm. A DG can be considered as the graphical representation of a single assignment algorithm.

Definition 2.4 : *Localized Dependence Graph.* An algorithm is localized if all variables are (directly) dependent upon the variables of neighboring nodes only. A localized DG, therefore, is one which exhibits this property, i.e., all nodes in the graph show local dependence of computations.

Definition 2.5 : *Locally recursive algorithm.* A locally recursive algorithm is an algorithm whose corresponding DG has only local dependencies, i.e., the length of each dependency arc is independent of the problem size, and most nodes of the DG consist of the same kind of operations [13].

Definition 2.6 : *Signal Flow Graph* (SFG). A signal flow graph is a graphical representation of both the functional and structural description parts of signal processing computations. The SFG expression consists of processing *nodes*, communication *edges* and *delays*. A *node* denoted by a circle represents the arithmetic or logic function performed with *zero delay*. An *edge* denotes either a dependence relation or a delay. The structural part of an SFG can be represented by a finite directed graph, $G = <V, E, D(E)>$. The vertices V model the nodes, the directed edges E model the interconnections between the nodes. Each edge e of E connects an output port of a node to an input port of some node and is weighted with a delay count $D(e)$.

2.2.1 Mapping Methodologies.

Having introduced several algorithm expressions, we shall now briefly consider the issue of transforming such expressions to a systolic array processor design. There are two methodologies for mapping algorithms into systolic arrays [6]. These are the canonical mapping methodology and the generalized mapping methodology [6,13]. The former methodology is for mapping homogeneous (completely regular) DG's onto processor arrays, while the latter methodology is for mapping heterogeneous (semi-regular) DG's onto processor arrays. The canonical mapping is suitable to treat a class of algorithm that have useful properties of being totally regular and localizable. Such algorithms can be expressed by shift-invariant dependence graphs. This mapping methodology consists of three design stages, each utilizing an appropriate canonical form. These design stages are described in more detail in subsection 2.2.1.1 .

The generalized mapping methodology allows the treatment of a broader class of algorithms and the corresponding dependence graphs. There are many other important algorithms that are not completely regular, that is, are not totally shift-invariant, but exhibit a certain degree of regularity. This semi-regularity very often proves to be useful for an efficient mapping methodology. The mapping methodology allows us to deal with an

extended DG classification and to have options by linear or nonlinear assignment/schedule. More flexibilities are also created by using multiple projections, allowing global communication, and treating totally irregular DG structures. The generalized mapping methodology can provide effective designs to many algorithms including Gauss-Jordan elimination, shortest path problems, transitive closure, simulated annealing, partial differential equations (PDE) problems and singular value decomposition (SVD) [6].

In this thesis we concentrate on the canonical mapping methodology and therefore, consider only regular and localizable algorithms.

2.2.1.1 Canonical Mapping Methodology.

The Canonical Mapping Methodology comprises of three design stages. In stage one, called *DG design*, for a given problem, the designer first identifies a suitable algorithm followed by a suitable algorithm expression and then generates its corresponding DG. Since the structure of a DG greatly affects the final array design, further modifications on the DG are often desirable in order to achieve a better design. In the second stage, called *SFG design*, based on different mappings of the DG onto array structure, a number of SFG's can be derived from the DG. In order to determine a valid array structure for a locally recursive algorithm, one design method is to designate one processing element (PE) for each node in a DG. However, this in general, leads to very inefficient utilization of the PE's, since each PE can be active only for a small fraction of the computation time. To improve PE utilization, the nodes of the DG are often mapped onto a fewer number of PE's, in a SFG form. The SFG can be viewed as a simplified graph. It is a more concise representation than the DG. It is more specific i.e., it is closer to hardware level design and also dictates the type of arrays that will be obtained. Also, while there are no loops in any DG, the SFG can have loops, as long as there is at least one delay on each loop.

There are two basic steps for mapping from a DG to an SFG. The first step is the *processor assignment*. This determines the processors to which operations should be assigned to. A criterion, for example, might be to minimize communication and exchange of data between processors. Once the processor assignment is fixed, the second step is the *scheduling*. This determines the ordering by which operations are assigned to a processor. A criterion might be to minimize total computing time. A more detailed discussion of the processor assignment and scheduling is given in [6].

In the third stage, called the *Array Processor Design*, the SFG obtained in the second stage is physically realized in terms of systolic array. Several reasons exist why one should first derive an SFG array and then convert it into a systolic array [6]. These include , (i) SFG offers a concise expression for parallel algorithms, (ii) SFG defines the structure of the array with minimum constraints on timing, and (iii) formal transformations from an SFG to a systolic array can be developed.

In the mapping from DG's onto SFG's, not all SFG schedules satisfy the conditions of the systolic schedule. The major gap is that most SFG's are not given in temporally localized form, even though they are spatially localized. In other words,

systolic array = SFG array + pipeline retiming

The cut-set retiming is the procedure to transform an SFG to an equivalent and temporal localized form so that all the edges between the PE's have at least one delay element. The topic of imposing temporal locality into a computing network has been investigated by several researchers [5,34,35]. A cut-set in an SFG is a minimal set of edges, which partitions the SFG into two parts. The cut-set retiming procedure is based on two rules, namely, *time-scaling* and *delay transfer*. Time scaling allows all delays in the SFG array to be scaled by a factor, thus scaling the input and output rates correspondingly. Delay-transfer allows advancing a number of time units on all the outbound edges of the cut-set and delaying the same number of time units in the inbound edges, or vice versa, without

affecting the overall timing of the system. An SFG is meaningful only when it is computable i.e., there exist no zero-delay loops or cycles in the SFG. All computable SFG's can be made temporally local by following the cut-set retiming rules, consequently, a spatially local and regular SFG array is always systolizable [6].

2.2.1.2 Direct Mapping from Linear Data Dependencies into Systolic Designs.

Many research works on systolic designs are based on direct mapping from DG's onto systolic arrays [14-33]. This case involves combining stage 2 and stage 3 of the canonical mapping methodology. The approach follows the SFG mapping methodology discussed in section 2.2.1.1. The only modification is that the schedule constraint is introduced such that every edge of the resulting SFG will have one or more delay elements, i.e., $D(e) \geq 1$, satisfying the temporal locality condition in the definition of systolic array. In this subsection, we review some of the works that have been done on this subject.

As reviewed in chapter I, Karp, Miller and Winograd [14] proposed the *systems of uniform recurrence equations*. In their systems of uniform recurrence equations, they explored the idea of local and regular Dependence Graph (DG). They used an index space display to show the complete dependency of *locally recursive algorithms*. The idea of uniform recurrence equations was applied later on by Quinton [15,16] to the design of systolic arrays. Gachet *et al* [17] described the methodology underlying the *DIASTOL* system, whose aim is to allow systolic chips to be designed automatically. This methodology, called dependence mapping, is also based on the ability for someone to describe a problem as a system of *uniform recurrence equations*, then mapping the problem on a systolic array. Rao [18] defines a class of algorithms, namely the *regular iterative algorithms* similar to the systems of uniform recurrence equations defined in [14,16]. It is shown that a subclass of the regular iterative algorithms has the characteristics of the systolic algorithms and the corresponding systolic architectures may be systematically

derived. The notion of *locally recursive algorithms* proposed by Kung [19], stresses the locality of spatial and time indices in a recursive algorithm and therefore is expressible in terms of a spatially local DG or Signal Flow Graph (SFG). Many research explorations on this issue are discussed in [20,21,22,23]. A more detailed review can be found in [18,24].

Capello and Steiglitz [25] introduced a geometric interpretation of the linear transformation on index space, which provides an insightful look into how several systolic designs for the same algorithm relate to each other. Along the same line as the approach proposed in [25], several researchers [26-33] address the issue of mapping cyclic (loop) algorithms into systolic arrays. The cyclic algorithms are specified in a high - level language, such as FORTRAN, in form of DO loops. The approach is based on the space-time mapping of different cyclic algorithms into systolic array architectures. The mapping procedure is based on linear transformation of index sets and data dependence vectors. Moldovan [32] extended the mapping to the partitioning problem. He presented a technique for partitioning and mapping algorithms into VLSI systolic arrays. Algorithm partitioning is essential when the size of a computational problem is larger than the size of the VLSI array intended for that problem. His approach to the partitioning problem is to divide the algorithm index set into bands and to map these bands into the processor space. Finally, he presented a six step procedure for the partitioning and mapping technique. The main difference between the mapping approaches proposed in [14-26] and those proposed in [27-33] is that, the latter approaches started from a program using imperative language such as FORTRAN rather than from equations.

In this thesis, we will concentrate on the approach of space-time mapping of different cyclic algorithms into systolic array architectures [18,25-33]. The approach is described and illustrated in detail in section 2.3.

2.3 LINEAR TRANSFORMATIONS OF INDEX SET AND DATA DEPENDENCIES

Before we describe this approach, we will present some definitions which will be encountered in this section. Some of the definitions and notations are similar to those found in [28,29,32].

Definition 2.7: An algorithm A over an algebraic structure S is a 5 tuple $A = (\bar{J}^n, C, D, X, Y)$ where:

$\bar{J}^n$ is a finite index set of A, $\bar{J}^n \subset I^n$; where I denotes the set of nonnegative integers, and Z refer to the set of all integers. The nth cartesian powers of $\bar{J}, I$ and Z are denoted as $\bar{J}^n, I^n$ and Z^n respectively.

C is a set of triples $(\bar{j}, \upsilon, u)$ where $\bar{j} \in \bar{J}^n$, υ is a variable and u is a term built from operations of S and variables ranging over S. υ is called the variable generated at $\bar{j}$, and any variable appearing in u is called a used variable.

D is a set of triples $(\bar{j}, \upsilon, \bar{d})$ where $\bar{j} \in \bar{J}^n$, υ is a generated variable and $\bar{d}$ is an element of Z^n.

X is the set of input variables of A;

Y is the set of output variables of A

There are three types of dependencies in D.

(1) Input dependence; $(\bar{j}, \upsilon, \bar{d})$ is an input dependence if $\upsilon \in X$ and υ is an operand of u in computation $(\bar{j}, \upsilon, u)$; by definition $\bar{d} = 0$.

(2) Self dependence; $(\bar{j}, \upsilon, \bar{d})$ is a self dependence if υ is one of the operands of u in computation $(\bar{j}, \upsilon, u)$; by definition $\bar{d} = 0$.

(3) Internal dependence; $(\bar{j}, \upsilon, \bar{d})$ is an internal dependence if υ is an operand of u in $(\bar{j}, \upsilon, u)$ generated at $(\bar{j}^*, \upsilon, u)$; by definition $\bar{d} = \bar{j} - \bar{j}^*$.

The dependencies D can be represented as a matrix $D = |D^0 D^I|$ where D^0 is a submatrix of D containing all input and self-dependencies, and D^I is the matrix of inter-

nal dependencies. Throughout this thesis, we have considered algorithms with internal dependence, hence the dependency matrix contains only internal dependencies, i.e., $D = [D^I]$. It is important to note that the model of the algorithm given in definition 2.7, includes the algorithm index set, the computations performed at each index point, the data dependencies which ultimately dictate the algorithm communication requirements and the algorithm input and output variables.

Definition 2.8: The execution of an algorithm $A = (\bar{J}_n, C, D, X, Y)$ is described by

(1) the specification of a partial ordering ">" or "<" on $\bar{J}^n$ (called execution ordering) such that for all $(\bar{j}, \upsilon, \bar{d}) \in D^I$ we have $\bar{d} > 0$, i.e., $\bar{d}$ larger than zero in the sense of ">" or $\bar{d} < 0$, i.e., $\bar{d}$ is less than zero in the sense of "<" .

(2) the execution rule: until all computations in C have been performed, execute $(\bar{j}^0, \upsilon, u)$ for all $\bar{j}^0 > \bar{j}$ or $\bar{j} < \bar{j}^0$, for which $(\bar{j}, \upsilon, u)$ have terminated.

The ordering larger than zero (">") or less than zero ("<") is used in the lexicographical sense. Thus, if $\bar{d} = \bar{j} - \bar{j}^* > 0$, or on the other hand, $\bar{j}^* - \bar{j} < 0$, it means that computations indexed by $\bar{j}^*$ must be performed before those indexed by $\bar{j}$.

Definition 2.9: *Data dependencies.* These are the dependencies between the variables generated at different index points, which actually dictate the algorithm communication requirements.

Definition 2.10: *Dependency vectors.* The data dependencies can be described as difference vectors of index points where a variable is used and where that variable was generated. These vectors are called dependency vectors.

Definition 2.11: *Dependency matrix.* A dependency matrix D is a matrix that contains all the dependency vectors.

Definition 2.12: *Transformation matrix.* This is a matrix that consists of the time and space mapping functions used to transform the dependency matrix D into a new

transformed dependency matrix (TDM) or systolic matrix.

Definition 2.13: *Systolic matrix*. A column vector is systolic if its first element is strictly negative while the other elements $\in Z$, where Z is the set of all integer numbers. A matrix is systolic if all its columns are systolic and at least has one non-zero element in the second row or in the third row (for a 2-dimensional array).

Definition 2.14: A two-dimensional systolic array is a two-dimensional array of processing cells each of which communicates with at least one of its eight neighbours. This definition is easily extendible to an n-dimensional systolic array.

Definition 2.15: The *gcd*(a,b) is the largest (positive) integer that divides both a and b, and *gcd*(a,b,c) = *gcd* (*gcd*(a,b),c).

We now introduce the notion of linear transformation of index set and data dependencies.

An interesting mapping approach is the one called Space - Time representation of computation structures [26-33]. In order for a computation structure to be implemented in a VLSI systolic array, the conceptual sites $\{\bar{J}^n = (J^1, J^2, J^3,..., J^n)\}$ must be mapped into $Z^n = \{(\hat{J}^1, \hat{J}^2, \hat{J}^3,..., \hat{J}^n)\}$, where $\bar{J}^n$ denotes the index set of the algorithm and J^n is the nth - dimensional set of an algorithm. Although it is considered that VLSI arrays are $(n-1)$ - dimensional, practical arrays have pure planar layout. Therefore, for a two - dimensional systolic array, the conceptual sites $\{\bar{J}^n = (i,j,k)\}$ must be mapped into $Z^3 = \{(t,x,y)\}$ where t specifies the time when a node is computed and where (x,y) represents the 2-dimensional physical coordinates of the place in the VLSI systolic array where the node is computed. If a computational structure is characterized by a constant dependency matrix which consists of a set of vectors, $D = (\bar{d}_1,\bar{d}_2, \ldots, \bar{d}_k)$. Then the structure may be mapped into a time-space representation in Z^3 with new dependency as a systolic matrix $\Delta = (\bar{\delta}_1,\bar{\delta}_2, \ldots, \bar{\delta}_k)$, where k is the number of columns of the dependence vectors. This is done by means of matrix multiplication, $TD = \Lambda$, where the matrix T is a valid

transformation matrix [28-30,32].

The transformation T must be chosen such that the ordering of the execution of the algorithm is preserved. For instance, if the index set of an algorithm is defined as $\boldsymbol{L}^n[\bar{J}] = \{(J^1, J^2,...,J^n)\}$ and the ordering imposed by the data dependencies on set $\boldsymbol{L}^n$ is denoted with $\boldsymbol{R}$. The elements of $\boldsymbol{L}^n$ and ordering $\boldsymbol{R}$ form together a well - defined algebraic structure $<\boldsymbol{L}^n,\boldsymbol{R}>$ [28]. Thus the transformation T should be sought such that,

$$T:<\boldsymbol{L}^n,\boldsymbol{R}> \rightarrow <\boldsymbol{L}^n,\boldsymbol{R}_T> \tag{2.1}$$

where T has the following properties:

$$\begin{aligned} &(a)\ T \textit{ is a bijection} \text{ and } \textit{monotonic function} \\ &(b)\ \textit{the data dependencies of the new structure } <\boldsymbol{L}^n,\boldsymbol{R}_T> \\ &\quad \textit{can be easily selected.} \end{aligned} \tag{2.2}$$

Since T is a bijection, then the two structures are said to be isomorphic, and since T is monotonic with respect to $\boldsymbol{R}$ and $\boldsymbol{R}_T$, thus,

$$\begin{aligned} &\bar{d}>0 \rightarrow \bar{\delta} = T(\bar{d})>0 \quad or \\ &\bar{d}<0 \rightarrow \bar{\delta} = T(\bar{d})<0 \end{aligned} \tag{2.3}$$

In Eq.(2.3), $\bar{d}>0$ (or $\bar{d}<0$) , means that the data dependencies between the variables generated at different index points in the algorithm, are larger (less) than 0, in a lexicographical sense (see definition 2.8). Therefore, Eq.(2.3) simply means that the transformation T preserves the sense of the data dependencies.

Definition 2.16: Two structures are *isomorphic* as graphs if there is a 1-1 correspondence f between them such that there is an edge p to q if and only if there is an edge from $f(p)$ to $f(q)$. This means that if $p - q$ is a dependence vector, then $f(p) - f(q)$ must also be a dependence vector.

This definition leads to the motivation to consider the case where the dependence vector $p - q$ is mapped onto the dependence vector $f(p) - f(q)$, i.e.,

$$f(p-q) = f(p) - f(q)$$

These are the affine mappings, and disregarding translations, they are given by linear maps.

For (n-1) - dimensional systolic array, the transformation T which transforms D into Δ is defined as,

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & . & . & t_{1n} \\ t_{21} & t_{22} & . & . & t_{2n} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ t_{n1} & t_{n2} & . & . & t_{nn} \end{bmatrix} \qquad (2.4)$$

where Π is the time mapping function and S is the space mapping function.

Since present practical arrays have planar layouts, thus, the transformation matrix T for a 2 - dimensional array is represented by

$$T = \begin{bmatrix} \Pi \\ S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix} \qquad (2.4(a))$$

where the mapping Π is defined as

$$\Pi : L^n \rightarrow L_I^m \quad , \qquad n > m$$

$$\Pi(J^1, J^2, \ldots, J^n) = (\hat{j}^1, \hat{j}^2, \ldots, \hat{j}^m) \quad , \qquad \text{with } \hat{J} \in L_I^n$$

and the mapping S is defined as

$$S : L^n \rightarrow L_I^{n-m}$$

$$S(J^1, J^2, \ldots, J^n) = (\hat{j}^{m+1}, \hat{j}^{m+2}, \ldots, \hat{j}^n) .$$

The dimension of the two functions Π and S is marked by m, which is selected such that Π alone establishes the ordering (R_I) imposed on the elements $\hat{J}$. The first m coordinates of elements $\hat{J} \in L_I^n$ can now be related to time and the last $n-m$ coordinates

can be related to the geometrical properties of the algorithm.

An interesting question is to determine under what conditions the transformation T can exist. For an algorithm with a constant set of data dependencies D, the necessary and sufficient conditions for integer matrix T to be a valid transformation, transforming D to Λ, where Λ is a systolic matrix, are as follows [27-29,31]:

$$
\begin{aligned}
&(i)\ |T| \neq 0 \\
&(ii)\ gcd\ \bar{d}_i = gcd\ \bar{\delta}_i \quad i=1,2,...,k \\
&\quad \text{where gcd is the greatest common divisor} \\
&(iii)\ \text{The first nonzero element of vector } \bar{\delta}_i \ (\Pi\bar{d}_i) \\
&\quad \text{is negative.}
\end{aligned}
\tag{2.5}
$$

This representation and the mapping approach provide a mathematical means to find a valid transformation matrix which maps a constant dependency matrix into a systolic matrix [30]. Moldovan and Fortes presented a six step procedure [32,33] to find a valid transformation matrix of a given algorithm. Their procedure is an exhaustive search with the strategy to find a solution that minimizes the parallel execution time. Li *et al.* [30] proposed an approach to determine the existence of a valid transformation matrix for a given constant dependency matrix, and to construct the valid transformation matrix efficiently. They [30] introduced the restricted normal form of the matrix and showed that the problem of determining the existence of a valid transformation matrix of a given constant dependency matrix D is computable.

In order to determine the valid transformation matrix of D, the following defined restricted elementary row operations [28,30,36] are applied on the matrix D;

(i) Interchange two rows.

(ii) Multiplication of any row by 1 or -1.

(iii) Replacement of row_i by row_i + kk*row_j where kk is an integer ($i \neq j$).

The general description of the algorithm used to obtain a valid transformation matrix T is

given as follows [30]:

Procedure:

Step 1. Let the matrix Δ_F denote the former Δ matrix, and the matrix Δ_P denote the present Δ matrix. Also in the subsequent operations, Δ_P becomes Δ_F. Assume at the beginning that the matrix D is the same as the matrix Δ_F.

Step 2. Perform a restricted row operation on the matrix Δ_F to obtain Δ_P. Then determine the corresponding restricted elementary matrix E_j such that the product of E_j and Δ_F gives the matrix Δ_P.

Step 3. Repeat step 2 until the matrix Δ_P has all negative entries (elements) in the coordinate which represents the time domain. The product of the restricted elementary matrices gives the transformation matrix T.

Therefore, let a set, E^R, consist of all those restricted elementary matrices and a restricted matrix be a product of some restricted elementary matrices, $T_r = \prod_{j=1}^{n} E_j$ where $E_j \in E^R$ $j=1,2,...,n$ for some n.

Step 4. Continue to perform restricted row operations on $TD=\Delta$ to obtain more valid transformation matrices if they exist.

Then, by following the above steps , we can get the valid transformation matrices.

For the purpose of illustrating this procedure, we will consider three - index nested loop algorithms, such as the matrix multiplication algorithm. An example which had been used in [29] is as follows:

Algorithm :

```
      DO  10  I = 1,N
      DO  10  J = 1,N
      DO  10  K = 1,N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
   10 CONTINUE
```

In order to map the above algorithm into a systolic array, we need to obtain the dependency matrix D, and then transform it into Δ. To obtain D, the algorithm has to be in the pipeline format. In the following, we will describe the rules [6,29,30] for converting nests of DO loop algorithms which are not pipelined into pipeline format.

The ability to perform statements of a computer program (like the algorithm shown above) simultaneously depends on aspects of dependence relating these statements [29]. Indeed if for example, statement G references a quantity computed by statement H, then H must be performed before G. Computer programs are written in so many styles that the ability to characterize dependence among program statements systematically requires a processing of the program listing transforming it into a more canonical mode [6,29]. The difficulty of characterizing dependence is due to the problem of *overwriting* (also called *data broadcast* or *propagation*). It is common practice in writing programs to use the same symbol to represent several distinct quantities each of which stands for some preliminary result which is gradually replaced by more complete results. One reason for this practice is to reduce storage requirements [6,29]. Such a practice, inevitably, obscures the natural relationships that exist among the parts of the algorithm.

In the computational techniques where results are developed in an ongoing manner without the return of intermediate values to memory, the economy of storage is of minor importance. It is the structure of the relationships in the algorithm that is more important. Hence, there is a need to distinguish between distinct uses of same symbol. There are two ways that a variable can be overwritten, (i) by reassigning to the same symbol a new value in a new statement or (ii) by repeating the same statement more than once (as in

the matrix multiplication algorithm). To eliminate overwriting, every time a variable is reassigned it is also renamed. For example, consider the matrix multiplication algorithm, the program computes the product of the matrix B and C and stores it is A.

If we consider the first inner loop:

```
      DO  10  K = 1,N
         A(I,J) = A(I,J) + B(I,K) * C(K,J)
   10  CONTINUE
```

The sequence of results of A(I,J) is

B(I,1)*C(1,J), B(I,1)*C(1,J) + B(I,2)*C(2,J),.....

It can be seen that A(I,J) does depend on index K as well, so we can rewrite the first inner loop as follows:

```
      DO  10  K = 1,N
         A(I,J,K) = A(I,J,K) + B(I,K) * C(K,J)
   10  CONTINUE
```

In this form, each time that the statement is performed, the indices (I,J,K) are different, hence each A(I,J,K) has unique values. Thus data broadcasting is eliminated. Each value of the variable A and each step of the computation can be parameterized by the triple (I,J,K). We want each variable to be parameterized by (I,J,K) as well. Variables B and C are not in this form, therefore, they have to be expressed in this form. This can be achieved by adding *four* statements to the DO loop, giving

```
      DO  10  K = 1,N
         B(I,1,K) = B(I,K)
         C(1,J,K) = C(K,J)
         B(I,J,K) = B(I,J-1,K)
         C(I,J,K) = C(I-1,J,K)
         A(I,J,K) = A(I,J,K-1) + B(I,J,K) * C(I,J,K)
   10  CONTINUE
```

This augmentation of index for variables A,B,C is called *pipelining*. By repeating this

process as often as necessary, all data propagation can be eliminated and each variable can be made to depend explicitly on the loop variables of each loop in which it is contained.

Therefore, the entire algorithm becomes,

```
      DO  10  I,J,K = 1,N
         B(I,J,K)  = B(I,J-1,K)
         C(I,J,K)  = C(I-1,J,K)
         A(I,J,K)  = A(I,J,K-1) + B(I,J,K) * C(I,J,K)
   10  CONTINUE
```

In fact, to pipeline an algorithm, all algorithm's variables are supplied with any indices which they are deficient in so that they become pipelined with respect to those indices [29]. Therefore, in the body of the algorithm variables A, B and C are with three indices (i,j,k). This process of pipelining is useful because it eliminates all data broadcasting or propagation which is costly in VLSI implementations. The data of variables B and C propagate step by step-by-step from one cell in the array to the other, without being modified by the computation. This kind of data is called *transmittent data*, and the variables are called *input variables*. The variable A at the left side of the assignment is referred to as a *generated* variable while the one at the right side is referred to as *used* variable [28,29]. The data of variable A is modified by the computation and such data is referred to as *nontransmittent data*.

The dependency vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ corresponds to a used variable where a is the difference of the index i between the used variable and its generated variable. Similarly, b and c are, respectively, the difference of the indices j and k between the used and generated variables. The dependency vectors associated with the used variables, $A(i,j,k-1)$, $B(i,j-1,k)$ and $C(i-1,j,k)$ of the given algorithm, are $\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$, $\begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$.

The dependency matrix of an algorithm is the set of all dependency vectors (the

order is not important). The corresponding constant dependency matrix of the (matrix multiplication) algorithm shown before is

$$D = \begin{matrix} i \\ j \\ k \end{matrix}\begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{2.6}$$

An algorithm is systolic if and only if its dependency matrix is a systolic matrix [30]. The dependency matrix imposes a computational order which must be respected [28,32]. All elements of the dependency matrix are integers. The dependence relations between used and generated variables at one point of the index set are the same at any other points of the index set [28,31,32]. If a given algorithm is not systolic, Moldovan [28] developed a procedure to transfer that algorithm into a systolic structure. The transformation involves a transformation matrix T such that the product of T and D, $TD=\Delta$, is a systolic matrix, provided that the conditions for a valid transformation T [28,29], are met.

In order to build, for example T_1, these steps are followed:

1. $$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

   $\text{row}_2 + \text{row}_1 \rightarrow \text{row}_1$

2. $$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

   $\text{row}_3 + \text{row}_1 \rightarrow \text{row}_1$

3. $$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

That is ,

$$T_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

$$\text{and} \quad T_1D = \begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{2.8}$$

Continuing to perform more restricted row operations on T_1D, we obtain :

$$\text{row}_2 + \text{row}_3 \rightarrow \text{row}_2$$

$$4. \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

$$\text{that is} \quad T_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} T_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

$$\text{and} \quad T_2D = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{2.10}$$

If we continue to perform further restricted row operations on subsequent TD's, we obtain, for example the following transformations:

$$T_3 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix} \quad \text{and} \quad T_3D = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -2 & -1 & 0 \end{bmatrix} \tag{2.11}$$

$$T_4 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 0 & 1 & 2 \end{bmatrix} \quad \text{and} \quad T_4D = \begin{bmatrix} -1 & -1 & -1 \\ -2 & -2 & -1 \\ -2 & -1 & 0 \end{bmatrix} \tag{2.12}$$

$$T_5 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} \quad \text{and} \quad T_5D = \begin{bmatrix} -1 & -1 & -1 \\ -2 & -2 & -1 \\ -3 & -2 & -1 \end{bmatrix} \tag{2.13}$$

$$T_6 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad T_6D = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \tag{2.14}$$

In most practical applications, the majority elements of the dependency matrix are 0, 1 or -1 , and the rest of the elements are small integers which provide relative easy way to use the restricted row operations to build a valid transformation matrix. Once a valid transformation matrix is found, more valid transformation matrices, if they exist, can be obtained by applying more restricted row operations [30].

In most cases, the best transformation matrix T is chosen according to a given cost function [28,32]. In this example, T is chosen such that the parallel execution time is minimized, and the T that meets this requirement is T_2 [29].

$$T(i,j,k)^t = (\hat{i},\hat{j},\hat{k})^t \tag{2.15}$$

Thus ,

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i+j+k \\ j+k \\ k \end{bmatrix} = \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} \tag{2.16}$$

If we assume that the dimensions of our matrices are 3x3 then,

$$[B][C]=[A]$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{2.17}$$

The mapping of the index set (i,j,k) into the new index set $(\hat{i},\hat{j},\hat{k})$ using the transformation T given in (Eq.2.9) is shown in Fig. 2.10

i	j	k	$\hat{i}$ time	$\hat{j}$	$\hat{k}$ processor
1	1	1	3	2	1
1	1	2	4	3	2
1	1	3	5	4	3
1	2	1	4	3	1
1	2	2	5	4	2
1	2	3	6	5	3
1	3	1	5	4	1
1	3	2	6	5	2
1	3	3	7	6	3
2	1	1	4	2	1
2	1	2	5	3	2
2	1	3	6	4	3
2	2	1	5	3	1
2	2	2	6	4	2
2	2	3	7	5	3
2	3	1	6	4	1
2	3	2	7	5	2
2	3	3	8	6	3
3	1	1	5	2	1
3	1	2	6	3	2
3	1	3	7	4	3
3	2	1	6	3	1
3	2	2	7	4	2
3	2	3	8	5	3
3	3	1	7	4	1
3	3	2	8	5	2
3	3	3	9	6	3

Fig. 2.10: Mapping of index set into VLSI array using transformation T_2 .

Because of the way the transformation matrix is selected, the first coordinate $\hat{i}$ indicates the time at which the computation indexed by (i,j,k) is computed and $(\hat{j},\hat{k})$ indicates the processor where the computation is performed. For instance, the computation indexed by (2,2,1) in the algorithm is performed at $T(2,2,1)^t = (5,3,1)^t$, meaning that the computation time is 5 and processor cell is (3,1).

Notice that, from Fig.2.10, the computation starts at cell (2,1) at t=3. Cell (4,1) receives its first data at t=5 (which is two clock cycle later). Each cell computes three partial product terms. Cell (4,1) receives its last data at t=7 (which is four clock cycles later). Therefore, the input data for this line should be piped in such that the first input

data element is received at t=5 and the last at t=7. Cell (6,3) receives its last data at t=9 (6 clock cycles later). In order to drain the pipe or collect the last output from the array, one extra clock cycle is required, hence a total of 7 (seven) clock cycles ($t = \max \hat{t} - \min \hat{t} + 1 = 9 - 3 + 1 = 7$) is required to complete the matrix multiplication computation for N=3 .

The interprocessor communications result from the transformed data dependencies.

$$\hat{D} = TD = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

The first row of the transformed dependencies is ΠD = [-1 -1 -1]. Each element indicates the number of time units allowed for its respective variable to travel from the processor where it is generated to the processor where it will be used. Since each element is one, then no extra time delay is required for each variable to move from one cell to the other. The spatial part of the dependency is specified by the vectors ,

$$\begin{bmatrix} -1 \\ -1 \end{bmatrix} , \begin{bmatrix} -1 \\ 0 \end{bmatrix} , \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad , \quad \text{respectively.}$$

These vectors indicate the required connectivities for variables A, B and C. This means that variable A moves from a cell to the next via a diagonal channel with direction $\begin{bmatrix} -1 \\ -1 \end{bmatrix}$, variable B via a horizontal channel with a direction of $\begin{bmatrix} -1 \\ 0 \end{bmatrix}$, while variable C is stored in the node itself [29]. The VLSI array that implements this matrix multiplication algorithm is shown in Fig.2.11 .

From Fig.2.10 (which shows the mapping of the index set into VLSI array), it is clear that 9 cells are required to perform the matrix multiplication. In the special case when one datum is to be stored in each cell, therefore we need as many cells as the size of the matrix whose data is to be stored in the cells. All cells in the array are identical, and the structure of a cell results from the computations required by the algorithm as well as the timing and data communication dictated by the transformed data dependencies (Λ).

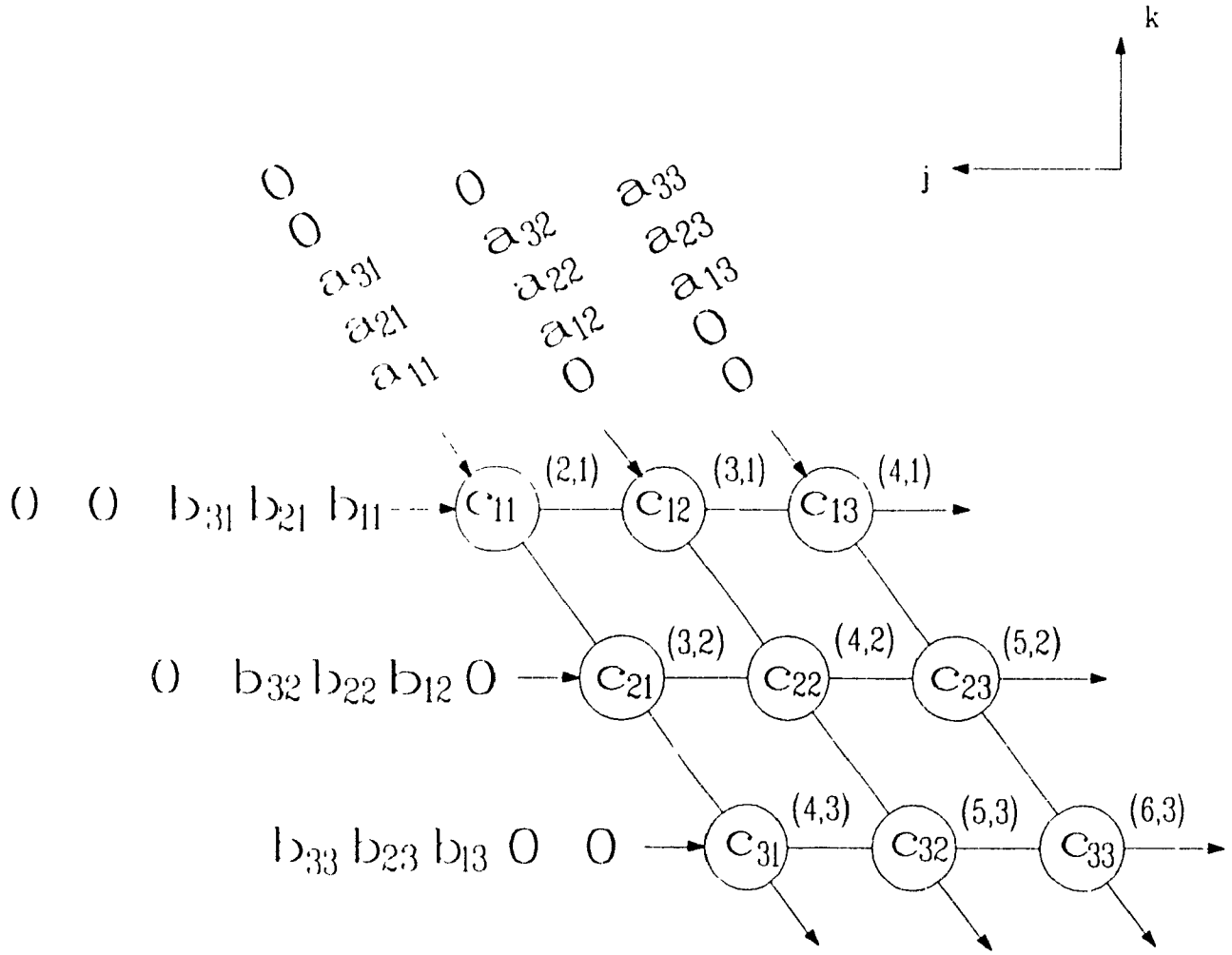


Figure 2.11 VLSI array structure when index i determines the timing (or valid execution ordering) of computations (for N=3).

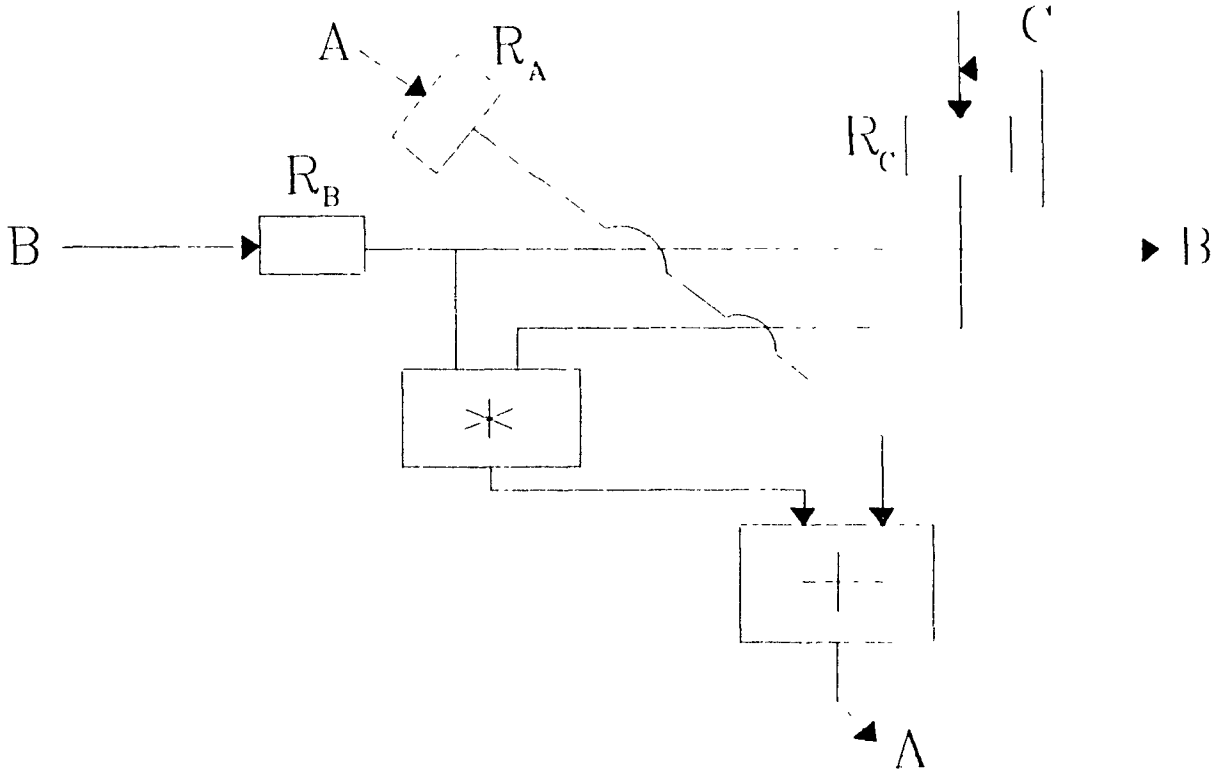


Figure 2.12 The structure of the cell in Figure 2.11.

Figure 2.12 depicts the structure of the cell. It consists of an adder, multiplier, registers for storing data of each variable and no delay elements.

Most often, many transformation matrices T can be found and each transformation leads to a different array. This flexibility apparently complicates matters, but in fact, it gives the systolic array designer the possibility to choose between a large number of arrays with different characteristics [32]. Also tradeoffs between time and space characteristics are possible.

In the next chapter, we will discuss a new method of generating the transformed dependency matrices. Also, a systematic approach of mapping algorithms into optimal systolic array architectures will be proposed.

2.4 REFERENCES

[1] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)," *In Sparse Matrix Symposium*, pp. 256-282, SIAM, 1978.

[2] H.T. Kung, "Why Systolic Architectures?," *IEEE Computer*, Vol. C-31, pp. 37-46, Jan, 1982.

[3] A.L. DeCegama, The Technology of Parallel Processing, Parallel Processing Architectures and VLSI Hardware, Vol. I, *Prentice Hall*, Englewood Cliffs, New Jersey, 1989.

[4] J.D. Ullman, Computational Aspects of VLSI, *Computer Science Press*, 1984.

[5] S.Y. Kung, "On Computing with Systolic/Wavefront Array Processors," Invited Paper, *Proceedings of the IEEE*, Vol. 72(7), July, 1984.

[6] S.Y. Kung, VLSI Array Processors, *Prentice Hall*, 1988.

[7] C. Mead and L. Conway, Introduction to VLSI Systems, *Addison-Wesley*, 1980.

[8] M. Annavatone et al., "Architecture of Warp," *1987 IEEE Conference Proceedings on Computer Architecture*, pp. 264-267, 1987.

[9] B. Bruegge *et al.*, "Programming Warp," *1987 IEEE Conference Proceedings on Computer Architecture*, pp. 268-271, 1987.

[10] M. Annavatone *et al.*, "Applications and Algorithm Partition on Warp," *1987 IEEE Conference Proceedings on Computer Architecture*, pp 272-275, 1987.

[11] M.C. Chen, "Space-Time Algorithm: Semantics and Methodology," Ph.D thesis, Computer Science Department, California Institute of Technology, 1983.

[12] R.W. Hockney and C.R. Jesshope, Parallel Computers, Adam Hilger Ltd., Bristol, U.K., 1983.

[13] S.Y. Kung, P.S. Lewis and S.N. Jean, "Canonic and Generalized Mapping from Algorithms to Arrays - A Graph Based Methodology," *In Proc. of the Hawaii Inter. Conf. on System Sciences*, Vol. 1, pp 124-133, January 1984.

[14] R.M. Karp, R.E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of ACM*, 14(3), pp. 563-590, July, 1967.

[15] P. Quinton, "The Systematic Design of Systolic Arrays," *IRISA Research Report*, No. 193, March 1983.

[16] P. Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *In Proceedings of 11th Annual Symposium on Computer Architecture*, pp 208-214, 1984.

[17] P. Gachet, B. Joinnault and P. Quinton, "Synthesizing Systolic Arrays using DIASTOL," Systolic Arrays (edited by W. Moore, A. McCabe and R. Urquhart), pp.25-36, *Adam Hilger Ltd.*, Bristol, UK, 1987.

[18] S.K. Rao, "Regular Iterative Algorithms and their Implementation on Processor Arrays," Ph.D thesis, Stanford University, Stanford, California, 1985.

[19] S.Y. Kung, "From Transversal Filter to VLSI Wavefront Array", *In Proc. Int'l Conf. on VLSI 1983, IFIP*, Trondheim, Norway, 1983.

[20] J-M. Delosme and I.C.F. Ispen, "Efficient Systolic Arrays for the Solution of Toeplitz Systems: An illustration of a Methodology for the construction of Systolic Architectures in VLSI," *Int'l Workshop on Systolic Arrays*, University of Oxford, pp. F2, July, 1986.

[21] M.C. Chen, "A Synthesis Method for Systolic Designs," *Technical Report 334*, Yale University, March, 1985.

[22] M.C. Chen, "Synthesizing Systolic Designs," *Technical Report 374*, Yale University, March, 1985.

[23] M.C. Chen, "Synthesizing VLSI Architectures: Dynamic Programming Solver," *In Int. Conf. in Parallel Processing*, pp. 776-784, Chicago, IL, August, 1986.

[24] J.A.B. Fortes, K.S. Fu and B.W. Wah, "Systematic Approaches to the Design of Algorithmic Specified Systolic Arrays," *In Proc. IEEE ICASSP '85*, pp. 300-303, Tampa, Florida, March, 1985.

[25] P.R. Capello and K. Steiglitz, "Unifying VLSI array Designs with Geometric Transformations," *In Int'l Conf. on Parallel Processing*, 1983.

[26] U. Weiser and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," in *VLSI Systems and Computations*, Rockville, Maryland: Computer Science Press, 1981.

[27] D. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Trans. on Computers*, vol. C-31, No. 11, November 1982.

[28] D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proc.IEEE*, Vol. 71, No. 1, January 1983.

[29] W. L. Miranker and A. Winkler, "Spacetime Representations of Computational Structures," *Journal of Computing*, Vol. 32, 1984.

[30] H. F. Li et al., "A Systematic Approach for Mapping Algorithms into Systolic Arrays," Technical Report, Dept. of Comp. Sc., Concordia University, Montreal, Quebec, Canada.

[31] D. I. Moldovan, "Tradeoffs Between Time and Space Characteristics in the Design of Systolic Arrays," *Proc. of ISCAS*, 1985.

[32] D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays'" *IEEE Trans. Comput.*, Vol. C-35, No. 1, pp. 1-12, January 1986.

[33] D. I. Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays," *IEEE Trans. Comput.-Aided Design*, Vol. CAD-6 January 1987.

[34] C.E. Leiserson, "Area-Efficient VLSI Computation," PH.D thesis, Carnegie-Melon University, Pittsburgh, Penn., October, 1981.

[35] C. Caraiscos and B. Liu, "From Digital Filter Flow-Graphs to Systolic Arrays," *IEEE Trans. on ASSP*, 1984.

[36] I. N. Herstein and D. J. Winter, "Matrix Theory and Linear Algebra," Macmillan Publishing Company, New York, N. Y., 1988.

CHAPTER III

PERFORMANCE ANALYSIS AND DESIGN OF OPTIMAL SYSTOLIC ARRAYS

3.1 INTRODUCTION

As we mentioned in Chapter I and also as illustrated in section 2.3 (Space-Time approach of mapping algorithms into systolic array architectures), several transformation matrices (T) are generated for a given constant dependency matrix. It is also known that each transformation leads to a different array. The best transformation matrix among the many transformations is selected based on the cost function. Although the resulting systolic array obtained with the selected transformation matrix represents a feasible design, in most cases, it does not satisfy some important VLSI requirements like improving the fault-tolerant capability of the architecture, minimizing the silicon area and delay, minimizing the VLSI routing complexity, improving the speedup of computations, maximizing throughput and obtaining the fastest propagation of output data variables.

For instance, in section 2.3, the transformation matrix T_2 (Eq.(2.9)) is chosen because it meets the requirement of minimizing the parallel execution time. In the resultant systolic array obtained using this selected transformation matrix (T_2), the data for variable C are stored in the individual processors of the array (Fig. 2.11). If this systolic array is replicated such that error detection or fault - tolerance can be achieved in the replicated array, then, to compare variable C for discrepancies, extra time and space will be required to do so. More time will be required to flush out the data stored in the cells after computation. Also, extra control or synchronization circuitry might be necessary to synchronize the corresponding stored data of the replicated array. This procedure could be costly, thus the resultant systolic array obtained using T_2 is not suitable for error detec-

tion or fault-tolerance. Although variable C is an input variable, the draw back is even more pronounced for a generated variable.

One of the objectives of this thesis is to provide a cost effective systematic approach for mapping algorithms into systolic arrays. Rather than deriving several transformation matrices and selecting the best one which might not even give a transformed dependency matrix (TDM) that does satisfy our VLSI requirements, we propose a methodology to obtain the desired TDM directly from the original dependency matrix (DM). By doing so, we can eliminate many TDM's and avoid deriving their corresponding transformations. In our approach, we generate only a subset of the TDM's of the given algorithm, that satisfies given VLSI requirements. Then, the corresponding valid transformation matrices of these TDM's are derived, if they exist. The advantages of this approach are two fold. First, it ensures that the generated TDM satisfies our VLSI requirements, thus eliminating the need for extra time and hardware to achieve, for example, fault-tolerance, when the TDM is mapped or translated into systolic array architecture. Second, since only a reduced set of the TDM's is generated, this leads to a reduction in the computation time required to generate all the TDM's (which include the ones that satisfy our requirements and those that do not), as demonstrated by the example in section 2.3.

As mentioned before, the best TDM that satisfies the VLSI requirements, can be chosen based on the cost function or the optimality criteria. Therefore, in this thesis, we are also concerned with the performance and the optimality of systolic arrays. The most suitable optimality criteria are hard to pinpoint and optimizing one factor may sacrifice other factors. There are many factors in determining the optimization criteria for the design of systolic arrays. The choice of optimality criteria is, in general, application dependent.

As discussed in chapter I, there have been several works on how to design optimal systolic array architectures [1-14], with each work concentrating on certain optimization criteria. Moldovan [1], Miranker [2], Wong [3], O'keefe [4], Delosme [5] and Fortes [7]

worked on how to minimize the *Computation time* of a systolic array. Rao [6] worked on minimizing the *Pipelining period* (α) and the *Block pipelining period* (β) of a systolic array. In addition to minimizing the execution or computation time, Fortes [7] proposed a heuristic approach for optimizing the hardware cost. The *array size*, which is defined as the number of processors in the array, obviously determines the basic hardware cost. Therefore, a systolic array which has the minimum number of processors gives the optimal solution with respect to this cost function. Gachet *et al.* [8] described a methodology, called the dependence mapping, which is based on the ability for someone to describe a problem as a system of uniform recurrence equations, then mapping the problem on a systolic array. In their design approach, they were interested in those transformations (projections) that will give fewer number of cells in the systolic design. Hence, their cost function is to minimize the number of processors in the array. Leiserson [9] mentioned that it is desirable to minimize hardware cost by using minimal delays in a systolic array, while preserving optimal α. He proposed a procedure to optimize the total number of delays in a systolic array, which is based on delay transfer through nodes. This is well known systolization scheme or the retiming of a synchronous circuit.

A mapping technique to design systolic structures having limited I/O requirement has been proposed in [10,11,12,13]. They proposed a design methodology which is based on the linearization of arrays The systolic design uses linear connected arrays with data and control signals pumped at either end [11,12,13]. Their approach requires $O(\sqrt{N})$ I/O bandwidth for N x N systolic matrix multiplication algorithm. Li *et al.* [14] proposed a *parameter* method of designing optimal systolic arrays using the parameters: **velocities of data flows, spatial distributions of data and the periods of computation.** By relating these parameters in constraint equations that govern the correctness of the design, the design is formulated as an optimization problem. The performance of a systolic design is expressed in terms of these parameters. The number of PE's required, denoted by m_{PE}, depends on the directions in which the inputs are moving. The completion time (T_c) can

be expressed as a function of the PE configuration and velocity. The design problem is then formulated to minimize m_{PE} x T_c^2, or m_{PE} x T_c, or T_c. Hence, the optimal solution is the systolic design that is minimized with respect to any of these criteria.

In Ko *et al.* [15], algorithms are specified in terms of data dependency, and implementations are specified in terms of data propagation and sequence behavior. By establishing a relation between data propagation and sequence, an optimal mapping strategy is formulated as a problem of finding an integer solution of a set of linear equations. In the above approaches [1-15], the criterion used for optimum architecture selection is a space - time cost function formed by the total number of processing elements and the required clock cycles for a task. These are purely architectural arguments, which rely on the assumption that the maximum clock frequency is independent of processing element count. In a monolithically integrated circuit, this is not necessarily true [17,18]. There is an interplay between heat dissipation, signal propagation and chip area which must be satisfied if a system has to function in a synchronous fashion without overheating. This interplay sets physical limits on clock frequency [16]. Lee *et al.* [16] compared systolic architectures for matrix multiplication, in terms of the maximum speedup which can be achieved with increased processor count in a monolithically integrated circuit. The comparison process integrates the architectural characteristics and the technological parameters. The optimum systolic architecture is found for different physical limiting factors including **switch delay, power dissipation, I/O bandwidth and clock skew**.

The review of the previous approaches of designing optimal systolic array [1-16], reveals that the optimality of systolic arrays in these approaches is determined by optimizing only one optimization factor. This factor could be either the computation time, or the total number of the processing elements, or a cost function formed by combining the number of processors and the required number of clock cycles or the maximum speedup of the systolic array. It is important to mention that, although the number of processors and the number of clock cycles represent two optimality criteria, however, a combination

of both of them is regarded as one optimization factor. Since optimizing one factor may sacrifice other factors, it becomes imperative to identify a **unifying performance index** to measure the *overall array performance* (speed, cell's complexity, number of interprocessor connections, practical design considerations, etc.).

An important problem associated with the issue of identification of a unifying performance index is the formulation of the more realistic and the most suitable optimality criteria. For instance, the area of a systolic array does not consist of only the number of processing elements in the array, as employed in the previous approaches. The area is a complex function of the number of PE's, the number of buffers, the interconnection pattern and the available technology. By using such realistic optimality criteria, which take into account the practical design considerations, more realistic optimal systolic array architectures can be designed.

Therefore, in this thesis, in addition to identifying a unifying performance index to measure the overall systolic array performance, we are also interested in formulating the most suitable and more realistic optimality criteria for the design of systolic arrays. Having formulated the suitable optimality criteria, a unifying performance index to measure the overall array performance can be identified.

Thus, an algorithm that maps any given algorithm with constant data - dependencies, into an optimal systolic architecture is proposed in this chapter. We formulate a compound objective function (COF) to measure the cost of each TDM obtained directly from the original DM. Therefore, the design is formulated into an optimization problem of finding the TDM with minimum cost function. The idea behind obtaining the TDM directly from the DM is to avoid deriving the transformation for each TDM. We first find one valid TDM and then evaluate the objective function. If a new TDM does not provide a better cost value than the one obtained so far, it is rejected. If one TDM gives a better cost value, then it is retained. The TDM with the minimum cost function is always retained.

The formulated COF is composed of several cost functions. It is a unified performance index to mea ure the overall performance of the systolic array architecture [19]. The cost functions include, the fault-tolerant capability, the silicon area, the throughput, the product of the silicon area and total computation time and the speedup of computation for the systolic array. In this chapter, we compare various systolic architectures in terms of the different cost functions [20]. The optimal or near optimal systolic array architecture given the respective performance measures is determined for each cost function [20]. Finally, by employing the unified performance index [19], we obtain the systolic array with the best overall array performance. The comparison process takes into consideration the architectural features and the technological parameters of the arrays. This approach provides an efficient method for selecting an optimal or near optimal systolic matrix. We illustrate our method using some examples of the iterative algorithms.

The outline of this chapter is as follows: In section 3.2, a method for obtaining a new TDM is presented. Section 3.3 describes the optimization algorithm for designing optimal systolic arrays. Also, the basic formula of the suitable optimality criteria are given in this section. Some examples to illustrate our method are presented in section 3.4. Finally, section 3.5 contains the summary and concluding remarks.

3.2 DETERMINATION OF THE TRANSFORMED DEPENDENCY MATRIX (TDM)

This section describes an approach to determine the transformed dependency matrix (TDM) of an algorithm.

If a computational structure is characterized by a constant dependency matrix which consists of a set of vectors, $D = (\bar{d}_1, \bar{d}_2, \ldots, \bar{d}_k)$, then the structure may be mapped into a time - space representation in Z^n in which, the new transformed dependency matrix is represented as a systolic matrix $\Delta = (\bar{\delta}_1, \bar{\delta}_2, \ldots, \bar{\delta}_k)$. This is done by means of matrix

multiplication $TD = \Delta$, where the matrix T is a valid transformation matrix that satisfies the three conditions given in section 2.3.

In the previous approaches, a valid transformation matrix T is first obtained and then it is used to evaluate Δ by performing the matrix multiplication $TD = \Delta$. This procedure has to be repeated many times until an optimal or near optimal solution is obtained. In this case therefore, the structure of the TDM (Δ) is not known until all the operations to generate T are performed and Δ is selected. Although, the corresponding systolic array of the selected Δ may represent a feasible design, it may not meet some of the desired VLSI requirements. In view of this problem, our objective is then, to find an approach that will allow us to select Δ first, and examine it in order to determine if it meets our desired VLSI requirements. In essence, the goal is to be able to first, select the systolic structure (structure of Δ) that we want and then derive its corresponding transformation matrix T, if there exists a valid one. This will avoid deriving T for each TDM, if we know that its TDM may not satisfy our requirements. In this way, those TDM's that are not of interest to us can be eliminated without the need to obtain their corresponding transformations.

In order to achieve the above objective, we propose an approach whereby, rather than producing several transformation matrices and then selecting the one that gives optimal Δ, we obtain the new transformed dependency matrix Δ directly from the dependency matrix by adding another matrix Ω to the original dependency matrix. The validity of this approach lies on showing that in general, it is possible to add any valid matrix Ω to D and (a) obtain a Δ that does not violate the dependency constraints (b) find non-singular integer T and (c) show that it will be easier to generate Ω than T. Therefore, in this approach, the issues are first, to prove that by adding another matrix (Ω) to the original DM to obtain the new TDM, the dependency constraints are not violated and also, the order of computations of the algorithm is preserved. Second, to show that the proposed approach of generating Δ is more cost effective than the existing approaches.

Before we proceed to state the theorem, we give the following definition.

Definition 3.1 : *Preserving Execution Order Matrix (PEO).* A matrix Ω is an PEO matrix if it has the following two properties:

(i) the row(s) or the submatrix of Ω that determine(s) the valid execution ordering of the computations ($\Omega_{|_{1D}}$) has only zero (0) or negative integer coefficients.

(ii) the submatrix of Ω which defines the space (Ω_{SD}) has integer coefficients.

Theorem : A transformed dependency matrix (TDM) can be obtained by adding to the original dependency matrix (DM) a PEO matrix Ω. This TDM preserves the order of computations of the given algorithm if the elements of $\Omega_{\Pi D}$ are selected such that $D\left[a_{i\mathrm{k}}\right] + \Omega\left[b_{i\mathrm{k}}\right] = \Delta\left[c_{i\mathrm{k}}\right]$ and $c_{i\mathrm{k}} < 0$ for all k. Where i is the time index and k is the number of columns of the dependency matrix.

Proof :

Given that

$$D + \Omega = \Delta \tag{3.1}$$

Since the resultant matrix is the TDM therefore, if $c_{i\mathrm{k}} < 0$, then the order of execution is preserved. Also $c_{j\mathrm{k}}$ and $c_{l\mathrm{k}}$ for all j, l and k (i.e. Ω_{SD}) are integers since D is an integer matrix.

Adding negative coefficients to the Π row of D, affects Δ in such a way that it introduces further delay between the time the affected variables are generated and the time they are used. If the results of a variable can be computed after ($n\tau$) clock cycles to satisfy the dependency constraints, then computing the results after ($n\tau$ + mc) clock cycles will not violate the constraints. Where mc is any integer. Thus, if $\Omega_{\Pi D}$ consists of zero or negative elements, the Δ obtained by adding Ω to D, will preserve the order of execution of the algorithm. Also, the dependency constraints will not be violated.

Also, the addition of negative, zero or positive integers (Ω_{SD}) to the rows of D that define the physical locations where the computations are performed (i.e. the space), merely establishes the desired geometrical properties of the structure of the architecture. This does not affect the ordering of the computations of the new TDM (Λ), consequently, the ordering of the execution of the algorithm is preserved. Thus, a TDM can be obtained by adding a PEO matrix to the original dependency matrix. The resultant TDM will preserve the order of computations of the algorithm and also, satisfy the dependency constraints.

Q.E.D

Now since $TD = \Delta = D + \Omega$, thus, Ω has to be selected such that the delay units and the interprocessor communication requirements of the resulting Λ are minimized. Concentrating on the algorithms with two or three abstract indices, therefore, if $TD = \Lambda$, then starting with a given D and a desired Δ, we may write

$$T = \Delta D^{-1} \tag{3.2}$$

Hence, if D is a non-singular invertible matrix then T can be evaluated easily. D is an n x k matrix, it should have a full rank, r, to be invertible, where r is the number of abstract indices of the algorithm. For the planar geometries that we are considering, i.e., for 1-D, $r = 2$ and for 2-D, $r = 3$.

The procedure to calculate T is simple if D is a square matrix. However, when D is an n x k matrix and for $r = n \leq$ k, then D^{-1} is given by

$$D^{-1} = D^{T} (DD^{T})^{-1} \tag{3.2a}$$

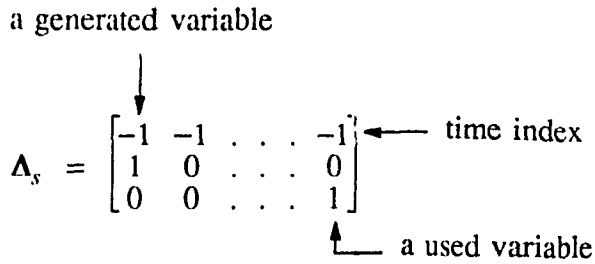
$$\text{subject to } |DD^{T}| \neq 0$$

The method requires the evaluation of D^{-1} only once to generate various transformation matrices. A transformation matrix can be generated by a single matrix multiplication and

its validity checked according to the criteria given by Moldovan [21]. If D is not invertible, then the methods proposed by Moldovan [21] and Li [24] can be used to generate the transformation matrices (T's) and their corresponding transformed dependency matrices (Λ's). However, with these methods an exhaustive search is required and the structure of the array will not be known until its T and Δ are generated.

3.2.1 Procedure for the Generation and Selection of Δ

For the problem of transforming iterative algorithms with three indices, we start with an arbitrary structure of $\Delta = \Delta_s$ given below, where Δ is a 3 x k matrix.



$$\Delta_s = \begin{bmatrix} -1 & -1 & \cdots & -1 \\ 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

For practical systolic arrays, both the number of the delay units and the length of the interconnection lines between the processors are to be kept to a minimum. Therefore, elements of the Δ matrix with large magnitudes are not desirable.

A search for a new TDM is made by varying c_{pq} of Δ. We will limit our search space by the following constraints

$$\sum_{p=1}^{n} \sum_{q=1}^{k} |c_{pq}| \leq E_1$$

where

$$E_1 = 5 * k * e_m$$

and

$$E_2 = |c_{pq}| \leq 2|e_m|$$

$$\text{where} \quad e_m = \max |d_i| \tag{3.2b}$$

Each time a Δ is selected, its transformation matrix T is calculated. The validity of the T is checked and if it is valid, then the corresponding Δ is a viable solution which can be included among the list of the candidates for optimization. Having exhausted the constraint search, the optimization technique described in [19] is applied to select the transformed dependency matrix (Δ) that gives the best architecture with respect to the overall optimization function. It is important to note that, with this methodology we start by looking directly into viable solutions.

Example

In order to illustrate our approach of mapping algorithms into optimal systolic structures we use the example that has been used previously by Moldovan and Fortes [21] which is given as follows:

Algorithm :

```
for i = 1 to N do
for j = 1 to N do
for k = 1 to N do

begin

  A(i,j,k) = A(i-1, j+1, k) * B(i-1, j, k+1) ;
  B(i,j,k) = B(i-1, j-1, k+2) + B(i, j-3, k+2) ;

end;
```

The corresponding constant dependency matrix is

$$D = \begin{bmatrix} -1 & -1 & -1 & 0 \\ 1 & 0 & -1 & -3 \\ 0 & 1 & 2 & 2 \end{bmatrix} \tag{3.2c}$$

A B B B

given

$$D^{-1} = \frac{1}{6}\begin{bmatrix} -11 & -6 & -9 \\ -2 & 0 & 0 \\ 7 & 6 & 9 \\ -6 & -6 & 6 \end{bmatrix} \tag{3.2d}$$

We may apply our search technique and thus generate various transformed dependency matrices using Eq.(3.2), as follows:

$$T_1 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \text{for} \quad \Delta_1 = \begin{bmatrix} -1 & -2 & -3 & -2 \\ 0 & 0 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \tag{3.3}$$

$$T_2 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \quad \text{for} \quad \Delta_2 = \begin{bmatrix} -1 & -2 & -3 & -2 \\ 0 & 0 & 0 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix} \tag{3.4}$$

$$T_3 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & 0 \\ 2 & 1 & 1 \end{bmatrix} \quad \text{for} \quad \Delta_3 = \begin{bmatrix} -1 & -2 & -3 & -2 \\ -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & -1 \end{bmatrix} \tag{3.5}$$

$$T_4 = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 1 & 2 \end{bmatrix} \quad \text{for} \quad \Delta_4 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 1 \end{bmatrix} \tag{3.6}$$

$$\Delta_5 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ 0 & -1 & 0 & -1 \\ -1 & 0 & 1 & 1 \end{bmatrix} \tag{3.7}$$

$$T_6 = \begin{bmatrix} 2 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \quad \text{for} \quad \Delta_6 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ -1 & 0 & 1 & 1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \tag{3.8}$$

It is important to realize that not all transformed dependency matrices will give a valid T. For example for $\Delta = \Delta_5$, a valid transformation matrix T does not exist.

Similarly, for the matrix multiplication algorithm example which has been used in [2], some examples of Δ obtained using this method are given as follows:

$$\Delta_1 = \begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad T_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8a}$$

$$\Delta_2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad T_2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{3.8b}$$

$$\Delta_3 = \begin{bmatrix} -1 & -1 & -1 \\ -2 & -2 & -1 \\ -2 & -1 & 0 \end{bmatrix} \quad \text{and} \quad T_3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 0 & 1 & 2 \end{bmatrix} \tag{3.8c}$$

$$\Delta_4 = \begin{bmatrix} -1 & -1 & -1 \\ -2 & -2 & -1 \\ -3 & -2 & -1 \end{bmatrix} \quad \text{and} \quad T_4 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} \tag{3.8d}$$

$$\Delta_5 = \begin{bmatrix} -1 & -2 & -3 \\ -1 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad T_5 = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{3.8e}$$

$$\Delta_6 = \begin{bmatrix} -1 & -1 & -2 \\ 0 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad T_6 = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{3.8f}$$

In the above examples of Δ

(i) $|T_l| \neq 0$

(ii) gcd $\bar{d}_i$ = gcd $\bar{\delta}_i$ $\quad$ i = 1, 2,...,k

for all Δ_l

therefore, all the transformation matrices (T_l) are valid transformations, transforming D to Δ, and the order of computations are also preserved.

The procedure to generate a valid transformed dependency matrix (TDM) using this approach can be summarized as follows:

Procedure

1. **Initialization** : Given a dependency matrix D, check if it is invertible and start with initial TDM (Δ_s).

2. Systematically generate a new Δ by modifying the elements of previously generated Δ. Check the values of E_1 and E_2.

3. If the upper limits on E_1 and E_2 are not reached, generate the corresponding T, deter-

mine its validity and GO TO 2.

4. When all permutations are exhausted GO TO 5. Otherwise GO TO 2.

5. **END** .

The elements of Δ are changed monotonically. First, we make an element of the time function more negative by 1 (i.e. add -1 to it). Tnen the elements of the space function are varied one at a time to give a different systolic array structure. In this case, the timing structure of the arrays will be the same while their interprocessor communication requirements will be different. This process is continued until the constraint search space is exhausted.

It is important to mention that this method of evaluating the TDM (Δ) and hence T, is applicable if the matrix D has an inverse. In the case where it is not invertible, then the existing procedure for generating a valid T and its corresponding Δ [1,2,21,22,24] can be employed to determine several transformation and transformed dependency matrices.

3.2.2 Comparison of Our Approach and the Existing Approaches of Determining the Valid TDM

In section 2.3, we presented the general description of the existing procedure used to obtain a valid transformation matrix T and its corresponding TDM. The procedure involves performing restricted row operation on the matrix Δ_F to obtain Δ_P. Where Δ_F is the former Δ matrix, and Δ_P is the present Δ matrix. At the beginning of the process of generating valid transformations the D matrix is assumed to be the same as the matrix Δ_F. After performing the restricted row operation, then the corresponding restricted elementary matrix E_j is determined such that, the product of E_j and Δ_F gives the matrix Δ_P. This step is repeated until the matrix Δ_P becomes a systolic matrix, that is, it has all negative elements in the row that represents the time domain. Finally, the product of the

restricted elementary matrices, gives the transformation matrix.

In this approach, the generation of a transformation matrix requires at least two matrix multiplication computation and at least one matrix addition. The matrix multiplication operations come from performing $E_j \Delta_F = \Delta_P$ and $T_r = \prod_{j=1}^{n} E_j$. The addition operations come from performing at least one restricted row operation.

However, in the proposed approach described in section 3.2.1, the generation of a TDM requires at most one matrix multiplication ($T = \Delta D^{-1}$), and at most one matrix addition (to modify the previous Δ) Table 3.1 shows the comparison of the computation complexity of the existing approach and our approach for generating a TDM.

Computation Complexity	Existing Approaches (1,2,21,22,24)	Proposed Approach
Number of Multiplications	≥ 2	1
Number of Additions	≥ 1	1

Table 3.1 Comparison of the computation complexity for generating a TDM of an algorithm.

Since the matrix multiplication operation requires more computation than the addition operation, hence, the proposed approach performs less computations to generate valid TDM's, than the existing approaches. Also, since only a subset of those Λ that satisfy our requirements is generated, this leads in reduction of the total computation time for generating the TDM's. Therefore, from the comparison it can be seen that in general, our approach is more cost effective than the existing approaches for the generation of valid transformation matrix T, and the transformed dependency matrix Λ.

In the next section, we will formulate the optimality criteria and then describe the optimization algorithm for the design of optimal systolic array architectures.

3.3 DESIGNING OPTIMAL SYSTOLIC ARRAY ARCHITECTURES

This section describes an optimization algorithm for designing optimal systolic arrays. Using the results derived in section 3.2, we can in principle find a new transformed dependency matrix (TDM). In order to determine the validity of the new TDM, it is tested to see if it has a valid transformation. If it does, then it is possible to map it into a systolic architecture. In the case where a valid transformation T does not exist for the new TDM, then it cannot be successfully mapped into an architecture. Several TDM's can be generated as described above and the validity of each TDM can also be determined.

However, our motivation for obtaining the new TDM directly from the DM is to be able to select the TDM that meets the desired VLSI requirements. We want to avoid deriving the transformation matrix T for each TDM, a costly procedure, if we know that the derived TDM does not meet our requirements. Thus, by so doing, we can eliminate many TDM's without the need to obtain the corresponding transformations.

The new TDM may or may not be the optimum transformed dependency matrix. Since several TDM's that meet our requirements can be obtained, then it is imperative that we select the TDM that will give the optimal or near optimal systolic array, based on certain measure of cost. Therefore, we need to develop methods to measure the cost of the new TDM and then investigate how the cost parameters are affected for the different TDM's. The TDM that has the minimum cost value constitutes the optimal or near optimal transformed dependency matrix.

Before proceeding to investigate the methods to measure the cost of the new dependency matrix, we first identify the features we are interested in optimizing.

3.3.1 Optimization Criteria

In this thesis, our objective is to select a TDM so that when it is mapped into a systolic array architecture, it will :

(i) improve the fault - tolerant capability of the array

(ii) minimize the propagation delay

(iii) maximize the throughput

(iv) minimize the silicon area

(v) minimize the VLSI routing complexity

(vi) improve the speedup of computation for the systolic array

(vii) all or a combination of the above.

We also acknowledge that some of the above features are related. Having identified the features we are interested in optimizing, the next step is to derive the expressions that represent the cost measure of the identified features.

(i) Fault - Tolerance Measure

In order to improve the fault - tolerant capability of the array, it is desired that all the data variables or at least the generated variables, in a given algorithm be propagated from one cell to the other, when the algorithm is mapped into a VLSI architecture. In other words, all the data variables should be accessible and should not be stored in the cells.

For a 2-dimensional systolic array, the transformation matrix T is represented by (Eq.(2.4a))

$$T = \begin{bmatrix} \Pi \\ S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix}$$

where Π is the time mapping function and S is the space mapping function.

Therefore, the fault - tolerant feature requires that the elements $S_1\bar{d}_i$ and $S_2\bar{d}_i$ are not both zero, i.e., $t_{2j}\bar{d}_i$ and $t_{3j}\bar{d}_i$ (for j = 1,2,3) do not simultaneously have zero elements.

The need for the data variables to be accessible is such that the redundant output variables can be easily compared at the output of the array for discrepancies in the computed results. Hence, the results of the replicated output variables can be voted on to mask the effects of any tolerable faults. However, if the results of any data variable is stored in the cell, extra cycles may be required to flush out the output results from the cells. Also, in the case where data variables are replicated, complex synchronization circuitry may be required to synchronize the corresponding output results of this variable, and this will definitely require more storage space. Hence all the data variables in a given algorithm should be propagated from cell to cell in order to minimize these effects. Therefore, to ensure the flow of data such that all generated outputs may be compared simultaneously, we have to select S accordingly in such a way that no variable is stored in the processors. Thus

if $S_1\bar{d}_i = S_2\bar{d}_i = 0$

then FT = W otherwise FT = 0 , W > 0 (3.10)

where W is a constant (a coefficient in the objective function) which will be assigned any value corresponding to the cost of storing a data variable in the processors.

(ii) **Interconnection Delay Measure**

This consists of the delay in the interconnection lines of the systolic array. To minimize the delay required for the data to propagate through the VLSI systolic architecture, it is desired that the routing between the cells be to the nearest neighbors. In other words, the interconnection of the processors is to the nearest neighbors. This means that the elements in $S_1\bar{d}_i$ and $S_2\bar{d}_i$ in the TDM, should be as small as possible, e.g. 0, -1, 1 (i.e elements $\in$ {0, -1, 1}) if possible. This will ensure that no additional propagation

delay is introduced in the interconnection lines.

The interconnection delay parameter can be measured as follows:

Let $D_{int.}$ represent the delay of an interconnection line with unit length, where D_{int} is a constant determined by the technology. A unit length interconnection is defined as the amount of interconnection wire required to either horizontally or vertically connect two neighboring processors, from the center of one to the center of the other (i.e. the pitch) and the width of this wire depends on the chosen technology. The worst case delay of the systolic array architecture will be given by the interconnection line with the longest path (assuming other parameters are constant), and this is given by the interconnection line which has the highest number of data routing steps (R_{int}). Thus the interconnection line with the longest path, assuming the Manhattan geometry, is ,

$$R_{int.} = \max\left[| S_1\bar{d}_i | + | S_2\bar{d}_i | \right]$$

$$= \max\left[\sum_{j=1}^{2} | S_j\bar{d}_i | \right]$$

for $i = 1, 2, ...,k$

Therefore the delay of the interconnection line is given by

$$\tau_L = D_{int.} * R_{int} \qquad (3.11)$$

It is important to mention that $D_{int.} \propto (length)^2$, and we are segmenting the length here for simplicity.

(iii) **Throughput Measure**

The throughput of the systolic array is defined as the number of results that can be completed by the array per unit time. This can be measured by the total number of clock

cycles required to complete the computation of any given algorithm for any given size of the problem. Therefore, we wish to select the TDM that requires the smallest number of clock cycles to complete the execution of the algorithm. The total number of clock cycles (C) required for the computation of the algorithm can be determined in the following way :

Once a new TDM is generated and a transformation, a new parallel algorithm results immediately. In mapping the algorithm, if index i determines the valid execution ordering of the computation, then the total number of cycles (C) required for computation is given by [1,21,22]

$$C = t_{1l}(i^2,j^2,k^2) - t_{1l}(i^1,j^1,k^1) + 1$$
$$\text{for } l = 1, 2, 3$$

where $t_{1l}(i,j,k)$ is given by

$$\begin{bmatrix} t_{11} & t_{12} & t_{13} \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = t_{11}i + t_{12}j + t_{13}k$$

That is ,

$$C = \max\hat{i} - \min\hat{i} + 1$$

In general, C is given by the ratio

$$C = \left\lceil \frac{\max t_{1l}(i^2,j^2,k^2 - i^1,j^1,k^1) + 1}{\min |t_{1l}\bar{d}_i|} \right\rceil \qquad (3.12)$$

for any (i^2,j^2,k^2) , $(i^1,j^1,k^1) \in \bar{J}^n$, where $\bar{J}^n$ is the original index set

(iv) **Silicon Area Measure**

The area of the VLSI architecture consists of the area of the processing elements, the area of the delay units in the processors and the routing area for the interconnection lines. The area depends on the architecture of the processing elements, implementation and the available technology and has to be assessed for each configuration separately. Therefore, by choosing the TDM whose systolic array architecture consists of the optimum number of processors for computations, minimum number of buffers (delay units) and utilizes minimum area for routing the interconnection lines, then, the overall area of the architecture can be minimized. Although the degree of complexity of the interconnection pattern affects the area of the systolic architecture, it also has as effect on the difficulty of routing of the interconnection lines. That is, the more complex an interconnection pattern is, the greater the degree of difficulty to route such a pattern and vice versa.

For systolic operation, $\Pi\bar{d}_i < 0$, and for example when $\Pi\bar{d}_i \leq -1$, at least one time step is required for data item to propagate from one cell to the other. If more than one time unit is needed for data to propagate from one cell to another, then delay units are added in the processors. Therefore, the number of delay units (d_u) in a processor is given by

$$d_u = \sum_{i=1}^{k} |\Pi\bar{d}_i + 1| \tag{3.13a}$$

and the total number of delay units in the array architecture is

$$U_d = m_{PE} * d_u \tag{3.13b}$$

Therefore, the total area of the delay units in the architecture is given by

$$AD = U_d * A_d \tag{3.13c}$$

where A_d is the area of a delay unit. m_{PE} is the number of processing elements required to implement any algorithm in a systolic array design and it can be determined as described later in this section.

The total area of the processing elements is given by

$$AP = m_{PE} * A_{PE} \tag{3.14}$$

where A_{PE} is the area of one processor.

It is also important to consider the contribution of the routing area to the overall area of the systolic array architecture. In this analysis, we assume that the technology and the fabrication process permit the interconnection lines to be routed in a horizontal and vertical fashion. Also, assuming that the technology can support multi - layer metalization process, thus all the horizontal interconnection lines can be routed in one layer of metalization and the vertical lines in another layer. Then the interconnection lines in one metalization layer can be connected to the others in another metalization layer through vias, with contact cuts made at each point of connection. Therefore, in addition to the silicon area occupied by the interconnection lines, there is also the area of the contact cuts. Since the placement of the interconnection lines must satisfy certain design rules, the more the number of connections that have to be made between different layers of metalization, the more the number of contact cuts and hence, the more the interconnection lines have to be placed further apart to obey the design rules. This invariably increases the area required to route the interconnection lines.

Here, we measure the complexity of the interconnection pattern in terms of the total number of data routing steps in both the horizontal and vertical directions and the number of contact cuts.

We define R_u and R_v as the number of data routing steps in the horizontal and vertical directions respectively, for each interconnection line. They are given as follows:

$$R_{u_i} = | S_1 \bar{d}_i |$$

$$R_{v_i} = | S_2 \bar{d}_i |$$

In order to take into account the area of the contact cuts and the area resulting from the placement of the interconnection lines, we redefine v in the following way,

$$R_{v_i} = 2 | S_2 \bar{d}_i |$$

The factor of 2 here, indicates that we are using extra data routing step in the vertical direction, to accomodate the area of the contact cut at the point where the horizontal and vertical interconnection lines are connected.

The complexity of the interconnection pattern is then given as,

$$K = \sum_{i=1}^{k} (R_{u_i} + R_{v_i})$$

where k is the number of the dependency vectors which corresponds to the number of columns in the dependency matrix D.

Therefore the area required for routing the interconnection lines is given by

$$A_{int.} = K \, m_{PE} \, A_L \tag{3.15}$$

where A_L is the area of a unit length of the interconnection line and it depends on the technology, and the unit length interconnection line is as defined before. K is a constant which indicates the degree of complexity of the interconnection pattern of the architecture, the higher the value of K, the more complex the interconnection pattern is, and hence the higher the routing area.

Then, the silicon area (A_{si}) required to implement any given algorithm is obtained by combining Eqs. (3.13) - (3.15) and this is given by

$$A_{si} = m_{PE} \, A_{PE} + m_{PE} \, A_d \sum_{i=1}^{k} | \Pi \bar{d}_i + 1 | + K \, m_{PE} \, A_L \tag{3.16}$$

The number of processing elements (m_{PE}) required to implement any algorithm in a systolic array design can be obtained using the following procedure :

for each (i,j,k)
begin
Determine the pair $(\hat{j},\hat{k})$ as follows :

$$(\hat{j},\hat{k}) = (t_{2l}(i,j,k),\ t_{3l}(i,j,k));$$

where $l = 1,2,3$
end;
begin

$$m_{PE} = \sum_{\hat{k}=\min\hat{k}}^{\hat{k}=\max\hat{k}} (\max\hat{j} - \min\hat{j} + 1);$$

end; (3.17)

(v) **Speedup Factor Measure**

The speedup of a systolic architecture is defined as the ratio of the total time required for a single PE chip to complete the computation of a given problem size and the total time required for a multi PE chip.
That is,

$$S_m = \frac{\text{Total Time Required for a Single PE Chip}}{\text{Total Time Required for a Multi PE Chip}} \quad (3.18)$$

The speedup per processing element is defined as [16]

$$SP = \frac{S_m}{m_{PE}} \quad (3.19)$$

where m_{PE} is the required number of processing elements on the multi PE chip.

In [16], Lee *et al.* compared the speedup of some classes of systolic architectures for matrix multiplication. The comparison was done for different physical limiting cases such as heat dissipation, propagation delay in interconnection lines, clock skew and the I/O

bandwidth, which are of importance in VLSI technology. They showed that the maximum clock frequency not only depends on the number of processing elements, but depends on both architecture and technology. At the most abstract level, one encounters purely problem-dependent limits associated with the inherent complexity of a given computational task. At a somewhat less abstract level, there are architectural features associated with the control of data flow in space and time. However, the technology can not be avoided. Delays and power dissipation can be associated with virtually any operation at the circuit level. These delays are those which change when comparisons between various technologies such as CMOS versus bipolar, or perhaps Si versus GaAs, are made. All of these various levels are united, however, by the integration process.

The results they obtained are general for systolic architectures and are adopted here for part of our analysis of the cost function of the TDM. Consequently, not only do we take into consideration the architectural features of the array in deriving the cost function of the TDM, we also consider the circuit level parameters of the architecture and this gives a more complete and realistic analysis. Therefore our objective will be to select the TDM that has the best speedup given the different physical limiting cases.

The speedup per PE of the new TDM's is compared for the following physical limiting cases [16] : (a) switching delay, (b) power dissipation, (c) clock skew and (d) I/O bandwidth. These cases are described below.

A) *Switching Delay Limit*:

For a chip containing m_{PE} PE's, when the switching delay of a gate is much larger than the interconnection line delay and the power dissipation is less than the heat removal capacity of the chip, the operating clock frequency of the chip is limited by the switching delay. The speedup per PE of the systolic array architecture given this physical limiting case is

$$SP_1 = N_c \; / \; C \; m_{PE} \tag{3.20a}$$

where N_c is the total number of computations and C is the total number of cycles required for a single PE chip to compute the given problem, e.g. for an N xN matrix multiplication, $N_c = N^3$, where N is the size of a matrix.

B) *Power Dissipation Limit*:

The clock frequency of a chip is limited by the heat dissipation and the heat removal capacity of the chip. The speedup per PE for this limiting case is given by,

$$SP_2 = N_c \; / \; (\eta \; C \; m_{PE}) \tag{3.20b}$$

where η is the efficiency of the systolic architecture. This is measured by the percentage of (busy time - space spans) over the (total time - space spans, which equals the sum of all busy and idle time - space spans). In other words, this is the percentage of the number of processors that are active in any given period of time. Equation (3.20b) shows that the more the number of processing elements that are active at any given time, the more power the chip dissipates and this reduces the speedup of the systolic array architecture.

C) *Clock Skew*:

When the propagation delay in the interconnection lines is significant, a large systolic array operating in a synchronous fashion may have a problem in synchronizing data arrivals. If an H - tree distribution network driven by a single buffer is used for the clock, RC line modeling of interconnections yields the clock skew asymptotically proportional to N^3 for an N xN processor array. The clock frequency of the large array of PE's is limited by the clock skew. Thus, from [16], the speedup per PE for this case is,

$$SP_3 = N_c \; / \; (F_s \; D_{int.} \; C \; m_{PE}^{5/2}) \tag{3.20c}$$

where F_s is the clock frequency of a single PE chip, $D_{int.}$ is the delay of an

interconnection line and it is a constant determined by the technology and the dimension of a PE.

It is realized that with H - tree clock distribution, the clock skew is almost minimized, however, this model is used for generality.

D) *I/O Bandwidth*:

When the I/O requirement of a PE is greater than the I/O capacity, off - chip interconnection requirements set a limit on the clock frequency. The I/O requirement of a PE is the sum of the word lengths of the coefficients or variables in the algorithm multiplied by the clock frequency. From [16], the speedup per PE given the I/O bandwidth limited case is,

$$SP_4 = N_c \ / \ (\eta \ C \ m_{io} \ m_{PE}^{1/2}) \tag{3.20d}$$

where m_{io} is the number of I/O's (which includes the external and the internal interconnection lines of the array) required for any systolic array architecture and this is given by

$$m_{io} = 2 \, m_{PE} * \left[\mathrm{k} - \sum_{i=1}^{\mathrm{k}} (S_1 \bar{d}_i = 0 \ \Lambda \ S_2 \bar{d}_i = 0) \right] \tag{3.21}$$

where k is the number of columns of the dependence vectors (which also represents the number of operands or variables in the algorithm), and Λ represents the AND operator.

Therefore, the speedup factor per PE of the architecture is given by

$$SP = \min\langle k_1 SP_1, k_2 SP_2, k_3 SP_3, k_4 SP_4 \rangle \tag{3.22}$$

where k_1, k_2, k_3 and k_4 are constants. (At this stage, for those physical limiting cases we are interested in, we select $k_i = 1$ (for $i = 1, 2, 3, 4$), otherwise, $k_i = 0$).

3.3.2 The Compound Objective Function (COF)

The compound objective function (COF) that associates certain values with each corresponding optimization parameter can be expressed in the following form:

$$\begin{aligned} COF = {} & \boldsymbol{a1} * (\text{Fault - tolerant parameter}) \\ & + \boldsymbol{a2} * (\text{interconnection delay parameter}) \\ & + \boldsymbol{a3} * (\text{area x square of total execution time } (AT_c^2)) \\ & + \boldsymbol{a4} * (\text{speedup per PE parameters}) \end{aligned}$$

Here, we define T_c as the total execution time and it can be derived as follows. If we let t_u represent the computation time in each processor and t_r represent the propagation time delay of the interconnection line with the longest path, then the clock cycle time $t_c = t_u + t_r$. This is equal to $(\frac{1}{F_s} + \tau_L)$. Therefore, the total execution time is

$$T_c = C * (1/F_s + \tau_L) \tag{3.23}$$

Thus,

$$COF = \boldsymbol{a1}\, FT + \boldsymbol{a2}\, \tau_L + \boldsymbol{a3}\, (A_{si} T_c^2) + \boldsymbol{a4}\, SP \tag{3.24}$$

where $\boldsymbol{a1}$, $\boldsymbol{a2}$, $\boldsymbol{a3}$ and $\boldsymbol{a4}$ are weighted constants. For example, $\boldsymbol{a4} * SP$ is a factor indicating the optimum choice of a given architecture as far as the circuit limitations (switching delay, power dissipation, etc.) are concerned. The selection of the constants $\boldsymbol{a1}$, $\boldsymbol{a2}$, $\boldsymbol{a3}$ and $\boldsymbol{a4}$ are very important. For instance, if we let $\boldsymbol{a2} = 1$ and the rest of the coefficients = 0, then we are interested only in the interconnection line delay constraint. In addition to the binary values, these constants can also assume any value to indicate the weight (importance) of the corresponding cost function. For example if $\boldsymbol{a1} >> \boldsymbol{a2}, \boldsymbol{a3}$ or

$\alpha 4$, then more consideration is given to the fault - tolerance capability of the architecture and so on.

By changing the coefficients or elements of the new TDM, the values of the cost parameters are affected. Therefore, the optimization procedure has to determine how these changes affect the cost parameters and which change optimizes the compound objective function. Consequently, the coefficient values of the new TDM can be changed to reflect the importance of some of these parameters for a given algorithm.

3.3.3 An Approach to Choose the Values of the Weighted Constants

Equation (3.24) gives a closed form representation of the compound objective function (COF), which is the performance index to measure the overall performance of the systolic array architectures. In order to evaluate this function, the values of the weighted constants $\alpha 1$, $\alpha 2$, $\alpha 3$ and $\alpha 4$ must be appropriately chosen so that the dimensions of the different parameters in the function are unified. One possible approach to unify the dimensions of the parameters would be to choose the constants such that the COF is dimensionless. However, in this analysis, we would like to unify the dimension of the compound objective function to the dimension of Area x square of time (AT^2). This is because the dimension of AT^2 gives a meaningful measure of the merits of VLSI designs.

In Eq. (3.24), the fault - tolerant parameter (FT) is dimensionless, therefore, $\alpha 1$ has to be chosen so as to have a dimension of AT^2. The interconnection delay parameter, τ_L, has the dimension of time, hence $\alpha 2$ should be selected so that it has a dimension of AT. The (area x square of time) parameter already has the dimension of AT^2, and as such the constant $\alpha 3$ is dimensionless and can be chosen to be a constant. Finally, the speedup per PE parameter (SP) is dimensionless, therefore $\alpha 4$ should have the dimension of AT^2. If we represent the first parameter of the COF function of Eq.(3.24) in the form $A_1T_1^2$, the second parameter in the form $A_2T_2^2$ and so on, then Eq.(3.24) can be rewritten as follows:

$$COF = c_1 A_1 T_1^2 \, FT + c_2 A_2 T_2 \, \tau_L + c_3 A_3 T_3^2 + c_4 A_4 T_4^2 \, SP \qquad (3.25)$$

where $A_i T_i^2$ corresponds to the area and time components of the respective optimization parameters, and c_i are the associated weighting constants and they are dimensionless.

Comparing Eq.(3.24) and Eq.(3.25), thus, the weighted constants can be selected as follows:

$\boldsymbol{a1} = c_1 A_1 T_1^2$, $\boldsymbol{a2} = c_2 A_2 T_2$, $\boldsymbol{a3} = c_3$, and $\boldsymbol{a4} = c_4 A_4 T_4^2$.

The next step is to determine the values of $A_i T_i$. Some interesting and meaningful values to be assigned to $A_i T_i$ would be ,

$A_1 = A_2 = A_3 = A_{si}$ while $A_4 = m_{PE} \, A_{si}$

and $T_1 = T_3 = T_4 = T_c$ while $T_2 = \dfrac{T_c^2}{D_{int.}}$

Thus Eq.(3.25) can be expressed as follows:

$$A_{si} T_c^2 \left[c_1 W + c_2 \frac{\tau_L}{D_{int.}} + c_3 + c_4 m_{PE} SP \right] \qquad (3.25a)$$

For each Δ, $A_{si} T_c^2$ will be different, this is then multiplied by a factor.

The reason for assigning A_{si} (silicon area) to A_i is to associate the values of A_i with the actual value of the silicon area of the corresponding systolic array. Also, by assigning T_c (total computation time) to T_i, we established a tie between the values of T_i and the total computation time of the array. Thus. when the compound objective function of the individual systolic array is calculated, the values will properly reflect the actual architectural and technological parameters of the corresponding systolic array.

The criterion for selecting A_4 as $m_{PE} \, A_{si}$ is that, since SP is the speedup per PE, we want to represent the speedup factor by the total speedup of the systolic architecture rather than by the average speedup. Hence, by this assignment, we are in effect multiply-

ing SP by m_{PE}. T_2 is assigned $T_c^2/D_{int.}$ because, first, all the other T_i (T_1,T_3,T_4) contain T_c. Therefore, to be consistent, we want T_2 to contain T_c also. Second, since $\tau_L=D_{int.} * R_{int.}$, has the unit of time, by this assignment, we ensure that the dimension of the interconnection line delay factor will be area x time squared. From the manner by which A_i and T_i are selected, we have succeeded in unifying the dimensions of all the optimization criteria, while at the same time, maintaining the performance contribution made by the individual optimization factor, towards the design of the systolic array architecture.

It is important to note that unifying the dimensions of the optimization factors, does not eliminate the possibility of assigning different degrees of design importance to any of these factors. In the case where any of these systolic array design criteria is to be given more importance than the others, then a higher weight should be assigned to the corresponding modulating constant c_i of this factor. Also, different weights can be assigned to each of these factors to reflect the respective design importance. For the case when the same weight is assigned to all the modulating constants c_i (i=1–4), then , the same design importance is given to all the optimization factors.

Therefore, by substituting the corresponding A_i and T_i in the equations for the weighted constants, the values of $\boldsymbol{a1}$, $\boldsymbol{a2}$, $\boldsymbol{a3}$ and $\boldsymbol{a4}$ can be appropriately chosen so as to unify the dimension of the performance index (COF) used to measure the overall performance of the systolic array and c_i (i = 1-4) are the modulating constants.

It is important to note that in this analysis, we have modulated the performance index (COF) to have the dimensions of AT^2. However, this does not mean that only the (area x square of time) parameter in the COF function is used to measure the performance of the array. Rather, all the parameters in COF (fault - tolerant, interconnection delay, area x time squared and speedup factor) are taken into consideration in measuring the overall performance of the systolic array architecture.

3.3.4 Optimization Algorithm to Select a Transformed Dependency Matrix (TDM) that Minimizes the Objective Function .

Since several new TDM's which satisfy our VLSI requirements can be generated, it becomes imperative to select one of them such that the cost function is minimized. At first sight, it seems that the TDM with the following features Π = [-1 -1 -1] and $S_1\bar{d}_i$, $S_2\bar{d}_i$ = { 0, 1, -1 } will give a local optimum solution. However, we explore the possibility of optimizing the TDM with respect to the compound objective function, so as to select the TDM whose systolic array gives the best overall performance. The optimization algorithm to select the desired TDM is given as follows :

1. **Initialization:** initialize the modulating constants c_i's of COF, find a new valid transformed dependency matrix (TDM) based on the procedure outlined in section 3.2 and evaluate its cost function COF.

2. Determine a new TDM and evaluate its COF value and compare this value with the COF value of the previous TDM obtained.

3. If the new TDM is better than what is found so far, make it the current best TDM.

4. When the change in the improvement is significantly small, then **GO TO** step 5, otherwise **GO TO** step 2.

5. The current TDM is the near (or local) optimal solution.

6. **End.**

Having selected the TDM with the lowest cost, this TDM is then mapped into a systolic array architecture.

In the next section, we will illustrate our design approach using two iterative algorithms. Systolic array architectures will be designed which are optimal with respect to the

different cost functions and a combination of all of them.

3.4 ILLUSTRATIVE EXAMPLES

In this section, we will illustrate the approach to obtain a local optimal systolic array. We will compare various systolic architectures in terms of the different cost functions which include, (i) the silicon area (ii) throughput (iii) the product of the silicon area and the speed (time) of the architecture (iv) the speedup of computation for the systolic array and (v) a combination of the above respective cost functions. Then the optimum architecture for the different cases is determined. Also, the determination of the optimal systolic array architecture with the best overall performance will be investigated.

In order to illustrate our approach of mapping algorithms into optimal systolic architectures, we will use two examples of iterative algorithms that perform matrix multiplication, which have been used in [20,21] and [2] respectively. For the first example, we will consider the algorithm used in [20,21] which is also given in section 3.2.1.
The corresponding constant dependency matrix of the algorithm is

$$D = \begin{bmatrix} -1 & -1 & -1 & 0 \\ 1 & 0 & -1 & -3 \\ 0 & 1 & 2 & 2 \end{bmatrix} \qquad (3.26)$$

A B B B

If we generate a new transformed dependency matrix as follows:

$$\Delta_1 = \begin{bmatrix} -1 & -2 & -3 & -2 \\ 0 & 0 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \qquad (3.27)$$

$$T_1 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} i-k \\ i+j+k \\ i \end{bmatrix} \qquad (3.28)$$

Let N=3, therefore,

$$C = \max \hat{i} - \min \hat{i} + 1 = t_{1l}(3,3,1) - t_{1l}(1,3,3) + 1$$

$$= 2 - (-2) + 1 = 5 \text{ Cycles} = 2N - 1$$

The number of processors is determined as follows:

$(i,j,k) = (1,1,3), (1,2,3), (1,3,3)$

$(\hat{j},\hat{k}) = (5,1), (6,1), (7,1)$

$\hat{i} = -2$,

$(i,j,k) = (1,1,2), (1,2,2), (1,3,2), (2,1,3), (2,2,3), (2,3,3)$

$(\hat{j},\hat{k}) = (4,1), (5,1), (6,1), (6,2), (7,2), (8,2)$

$\hat{i} = -1$,

$(i,j,k) = (1,1,1), (1,2,1), (1,3,1), (2,1,2), (2,2,2), (2,3,2), (3,1,3), (3,2,3), (3,3,3)$

$(\hat{j},\hat{k}) = (3,1), (4,1), (5,1), (5,2), (6,2), (7,2), (7,3)\ (8,3), (9,3)$

$\hat{i} = 0$,

$(i,j,k) = (2,1,1), (2,2,1), (2,3,1), (3,1,2), (3,2,2), (3,3,2)$

$(\hat{j},\hat{k}) = (4,2), (5,2), (6,2), (6,3), (7,3), (8,3)$

$\hat{i} = 1$,

$(i,j,k) = (3,1,1), (3,2,1), (3,3,1)$

$(\hat{j},\hat{k}) = (5,3), (6,3), (7,3)$

$\hat{i} = 2$,

$$m_{PE} = \sum_{\hat{k}=1}^{\hat{k}=3} (\max\hat{j} - \min\hat{j} + 1) = 5 + 5 + 5$$

Thus, the number of processors (m_{PE}) is :

m_{PE} = 15 processors = N(2N - 1) .

The number of I/O's is m_{io} = 8N(2N-1)

Since the array has fault - tolerant capability, therefore FT = 0 .

The delay of the longest interconnection line is $\tau_L = D_{int.}$.

The number of the delay units in each processor is $d_u = 4$,

and the number of the delay units in the overall array is $U_d = 4\, m_{PE}$.

The data routing complexity is K = 7 .

Therefore, the total silicon area is given by

$$A_{si} = N(2N-1)A_{PE} + 4N(2N-1)A_d + 7\,N(2N-1)\,A_L$$

Now the speedup per PE parameters are evaluated as :

$$SP_1 = N_c \;/\; (2N-1)\,N(2N-1) = N_c \;/\; N(2N-1)^2$$

$$SP_2 = N_c \;/\; 1\,.\,N(2N-1)^2$$

$$SP_3 = N_c \;/\; (\,F_s\,.\,D_{int.}\,.\,(2N-1)\,.\,[N(2N-1)]^{5/2}\,)$$

$$SP_4 = N_c \;/\; (\,1\,.\,(2N-1)\,.\,8N(2N-1)\,.\,[N(2N-1)]^{1/2}\,)$$

$$COF = \boldsymbol{a2}\, D_{int.} + \boldsymbol{a3}\,(2N-1)^2\,(\,1/F_s + D_{int.}\,)^2 * [N(2N-1)A_{PE} + 4N(2N-1)A_d + 7\,N(2N-1)A_L] + \boldsymbol{a4}\, N_c \min\left[\frac{1}{N(2N-1)^2}\,,\ \frac{1}{N(2N-1)^2}\,,\ \frac{1}{F_s\, D_{int.}\,(2N-1)\,[N(2N-1)]^{5/2}}\,,\ \frac{1}{8\,(2N-1)\,[N(2N-1)]^{3/2}}\right]$$

$$COF_1 = \boldsymbol{a2}\, D_{int.} + \boldsymbol{a3}\,25\,(\,1/F_s + D_{int.}\,)^2 * [15A_{PE} + 60A_d + 105\,A_L] + \boldsymbol{a4}\, N_c \min\left[\frac{1}{75}\,,\ \frac{1}{75}\,,\ \frac{1}{4357\,F_s\,D_{int}}\,,\ \frac{1}{2323.8}\right] \qquad (3.29)$$

Now repeating the procedure for another TDM, we obtain

$$\Lambda_2 = \begin{bmatrix} -1 & -2 & -3 & -2 \\ 0 & 0 & 0 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix} \tag{3.30}$$

$$T_2 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix}, \quad \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} i-k \\ i+j+k \\ 2i+j+k \end{bmatrix} \tag{3.31}$$

Again the transformation exists and hence the new transformed dependency is valid.

$$COF_2 = \boldsymbol{a2}\ 2\ D_{int} + \boldsymbol{a3}\ 25\ (\ 1/F_s + 2\ D_{int.}\ )^2 * [\ 15A_{PE} + 60A_d + 135A_L\ ] + \boldsymbol{a4}\ N_c \min\left[\frac{1}{75}, \frac{1}{75}, \frac{1}{4357\ F_s\ D_{int.}}, \frac{1}{2323.8}\right] \tag{3.32}$$

Evaluating the third COF for a new TDM,

$$\Delta_3 = \begin{bmatrix} -1 & -2 & -3 & -2 \\ -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & -1 \end{bmatrix} \tag{3.33}$$

$$T_3 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & 0 \\ 2 & 1 & 1 \end{bmatrix}, \quad \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} i-k \\ i \\ 2i+j+k \end{bmatrix} \tag{3.34}$$

$$COF_3 = \boldsymbol{a2}\ 2\ D_{int.} + \boldsymbol{a3}\ 25\ (\ 1/F_s + 2\ D_{int}\ )^2 * [15A_{PE} + 60A_d + 165A_L] + \boldsymbol{a4}\ N_c \min\left[\frac{1}{75}, \frac{1}{75}, \frac{1}{4357\ F_s\ D_{int.}}, \frac{1}{2323.8}\right] \tag{3.35}$$

Further still,

$$\Delta_4 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 1 \end{bmatrix} \tag{3.36}$$

$$T_4 = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 1 & 2 \end{bmatrix}, \quad \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} 2i+j+k \\ i+j+k \\ 2i+j+2k \end{bmatrix} \tag{3.37}$$

$$COF_4 = \boldsymbol{a1}\, W + \boldsymbol{a2}\, 2\, D_{int.} + \boldsymbol{a3}\, 81\, (\, 1/F_s + 2\, D_{int}\,)^2 * |\, 15A_{PE} + 105A_L\,| + \boldsymbol{a4}\, N_c \min\left[\frac{1}{135}\,,\ \frac{1}{135}\,,\ \frac{1}{7843\, F_s\, D_{int}}\,,\ \frac{1}{3137}\right] \tag{3.38}$$

$$\Delta_5 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ 0 & -1 & 0 & -1 \\ -1 & 0 & 1 & 1 \end{bmatrix} \tag{3.39}$$

T does not exist for Δ_5 and hence it does not have a valid transformation.

$$\Delta_6 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ -1 & 0 & 1 & 1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \tag{3.40}$$

$$T_6 = \begin{bmatrix} 2 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix}\,, \quad \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} 2i+j+k \\ 2i+j+2k \\ i \end{bmatrix} \tag{3.41}$$

$$COF_6 = \boldsymbol{a2}\, 2\, D_{int} + \boldsymbol{a3}\, 81\, (\, 1/F_s + 2\, D_{int.}\,)^2 * |\, 21A_{PE} + 189A_L\,| + \boldsymbol{a4}\, N_c \min\left[\frac{1}{189}\,,\ \frac{1}{189}\,,\ \frac{1}{1.8\text{x}10^4\, F_s\, D_{int}}\,,\ \frac{1}{6929}\right] \tag{3.42}$$

It is important to note that several other transformed dependency matrices (TDM's) can be generated of which some of them may have valid transformations. However, we generated only the above set of TDM's because, from extensive studies, it is observed that the magnitude of the coefficients of further valid TDM's are higher than those in Λ_1, and as such would not produce any further minimum cost.

The comparison of the architectural features of the above set of generated TDM's is shown in table 3.1(a).

TDM	No. of PE's (m_{PE})	No. of cycles (C)	No. of I/O's (m_{io})
Δ_1	N(2N-1)	2N-1	8N(2N-1)
Δ_2	N(2N-1)	2N-1	8N(2N-1)
Δ_3	N(2N-1)	2N-1	8N(2N-1)
Δ_4	N(2N-1)	4(N-1)+1	6N(2N-1)
Δ_5	-	-	-
Δ_6	N(2N+1)	4(N-1)+1	8N(2N+1)

Table 3.1(a) Comparison of the architectural features of the generated TDM's of example 1 .

Similar analysis can be done for the algorithm in example two [2] whose dependency matrix is given in Eq.(2.6). Some sample examples of the generated TDM's for this case are given in Eqs.((3.8a) - (3.8f)). The comparison of the architectural features of the set of generated TDM's in this case, is shown in table 3.1(b).

TDM	No. of PE's (m_{PE})	No. of cycles (C)	No. of I/O's (m_{io})
Δ_1	N^2	3N-2	$4N^2$
Δ_2	6N+1	3N-2	6(6N+1)
Δ_3	8N+1	3N-2	6(8N+1)
Δ_4	7N	3N-2	42N
Δ_5	6N+1	6N-5	6(6N+1)
Δ_6	4N+3	4N-3	6(4N+3)

Table 3.1(b) Comparison of the architectural features of the generated TDM's of example 2 .

In order to select, in both examples, the TDM with the minimum cost, either in terms of the different cost functions or in terms of the compound objective function, we need to evaluate the corresponding cost functions of the respective TDM's. In this analysis, we illustrate our approach using a design example of a systolic array processor

[23]. This processor, which is a word level bit parallel architecture, has been designed using the Northern Telecom CMOS 3-μm double metal layer technology. It comprises of a parallel array of multiply - add/ cells and latches. It performs the multiplication of two 4 - bit integers.

The following values for the unknown parameters in the COF function have been obtained from the VLSI design of the systolic processor [23] :

Area of the processor (A_{PE}) = 2.5 x 10^6 μm^2

Area of one delay unit (A_d) = 5. x 10^4 μm^2

Area of a unit length interconnection line (A_L) = 4.8 x 10^3 μm^2

Unit length interconnection delay ($D_{int.}$) = 1.7 NS

Frequency of operation of the processor (F_s) = 10 MHz

For the given matrix multiplication algorithm, $N_c = N^3$

For the fault - tolerant capability, we choose W = 2

3.4.1 Optimization with respect to the Silicon Area

Equation (3.24) (COF function), consists of several VLSI design parameters that can be optimized when designing optimal systolic architectures. In this subsection, we are assuming that according to the specific purpose and the fabrication process or facilities, the silicon area is given more importance than the other cost functions. In other words, in this particular case, we are concerned more with the silicon area than any of the other optimizing parameters. Using the above design values of the systolic array processor example, the quantitative values corresponding to the cost associated to the silicon area of the respective TDM's can be determined. The comparison of the silicon area of the generated TDM's in example 1 is shown in table 3.2(a) (for N=3), while that for example 2 is shown in table 3.2(b).

TDM	Silicon Area (A_{si})
Δ_1	4.1004 x 10^7 μm^2
Δ_2	4.1148 x 10^7 μm^2
Δ_3	4.1292 x 10^7 μm^2
Δ_4	3.8004 x 10^7 μm^2
Δ_5	-
Δ_6	5.3407 x 10^7 μm^2

Table 3.2(a) Comparison of the silicon area of the generated TDM's of example 1 .

TDM	Silicon Area (A_{si})
Δ_1	2.2630 x 10^7 μm^2
Δ_2	4.8047 x 10^7 μm^2
Δ_3	6.3820 x 10^7 μm^2
Δ_4	5.4214 x 10^7 μm^2
Δ_5	5.0897 x 10^7 μm^2
Δ_6	3.8610 x 10^7 μm^2

Table 3.2(b) Comparison of the silicon area of the generated TDM's of example 2 .

As seen from table 3.2(a), Δ_4 corresponds to the TDM with the minimum cost function, which in this case is the silicon area. The value of the silicon area corresponding to the systolic array of Δ_4 is 3.8004 x 10^7 μm^2, and this represents the best array performance. Since our objective is to select a TDM such that the silicon area is minimized, thus, we select ,

$$\Delta_4 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 1 \end{bmatrix}$$

$$A \quad B \quad B \quad B$$

The VLSI array corresponding to the transformed dependency matrix Δ_4 is shown in Fig.3.1 . The structure of the cell is shown in Fig.3.2. All the cells in the array of Fig.3.1 are identical. It consists of an adder, a multiplier and no delay elements.

Similarly, as seen in table 3.2(b), Δ_1 is the TDM with the minimum silicon area, in the case of example 2. The value of the silicon area is 2.2630×10^7 μm^2. Thus, in order to minimize only the silicon area in this example, we select ,

$$\Delta_1 = \begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{matrix} \\ \\ \\ \end{matrix}$$
$$\begin{matrix} A & B & C \end{matrix}$$

The corresponding VLSI array structure is shown in Fig.3.3 . The cell structure is as shown in Fig.3.3(a). It is similar to that shown in Fig.2.12, but the only difference is the interprocessor communication requirements. In Fig.3.3(a), the data of variable A travels via a vertical channel, variable B via a horizontal channel and variable C is stored in the cell itself.

From the above analysis, Δ_4 in example 1 and Δ_1 in example 2, respectively, give the best array performance in terms of the silicon area. However, they are not suitable for fault-tolerance, since the data of one of the variables in the respective TDM's are stored in the processors. This does not satisfy the fault-tolerance VLSI requirements.

3.4.2 Optimization with respect to the Throughput

As mentioned in section 3.3, the throughput of the systolic array can be measured by the total number of the clock cycles required to complete the computation of any given algorithm, for any given problem size. Therefore, the comparison of the throughput of the generated TDM's in examples 1 and 2 are respectively shown in table 3.3(a) and table 3.3(b).

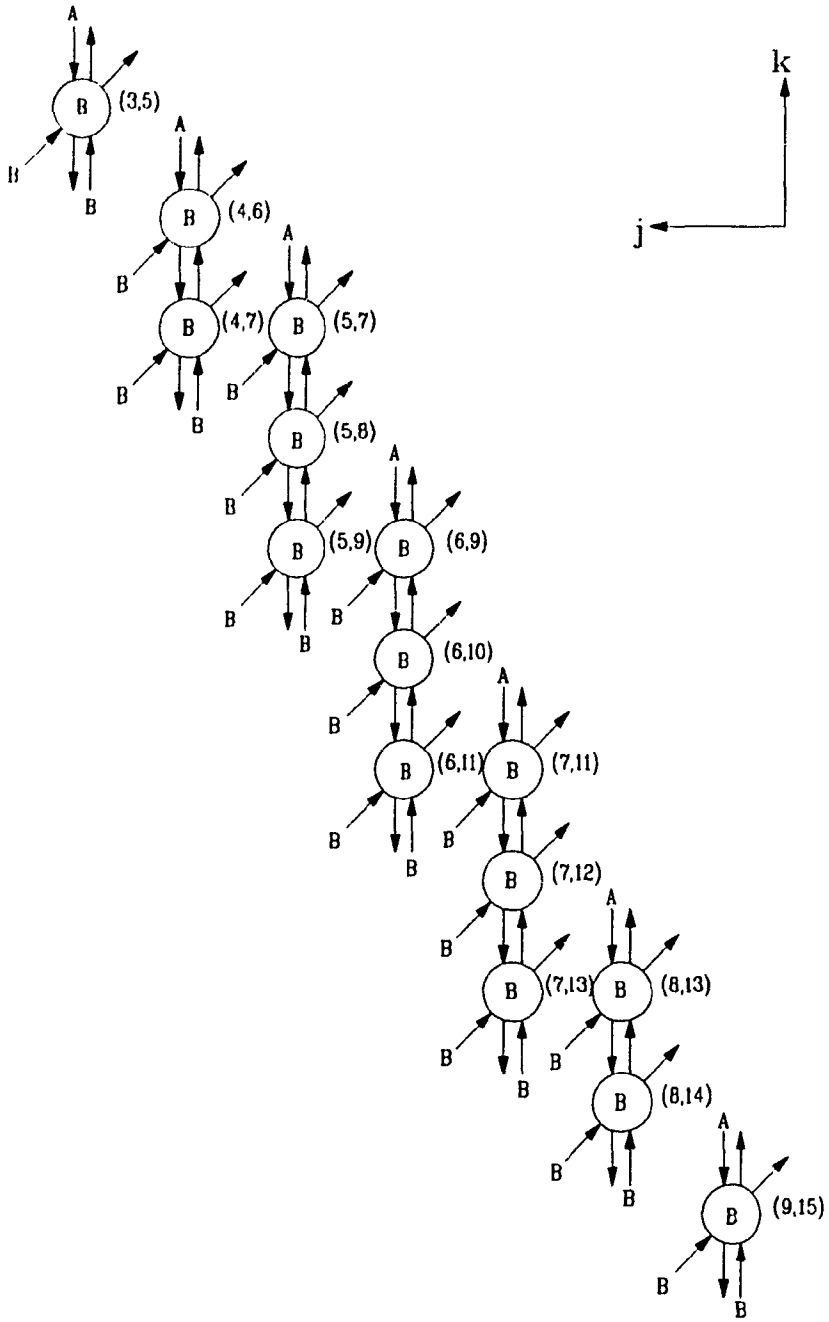


Figure 3.1 VLSI array structure using the transformation T_4 in example 1.

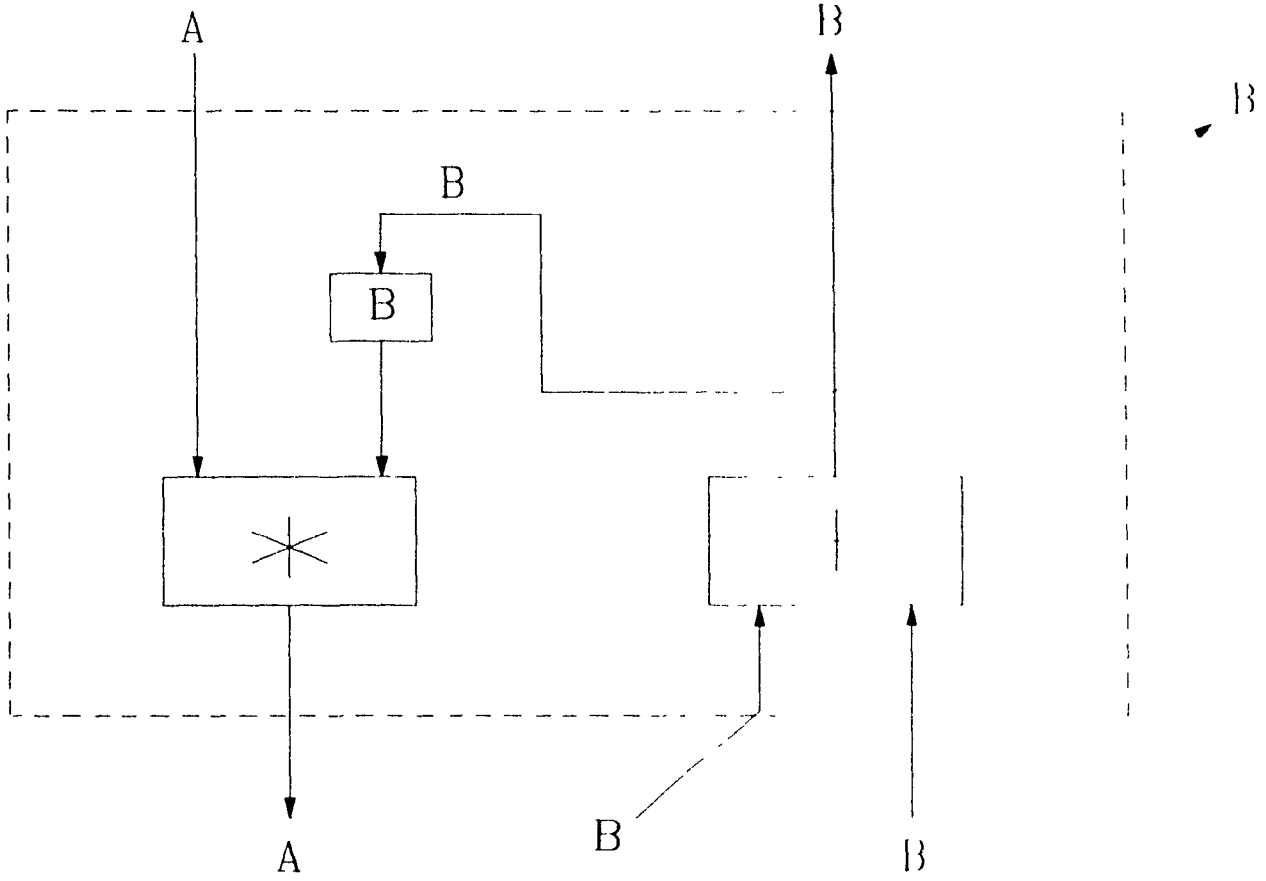


Figure 3.2 The structure of the cell in Figure 3.1

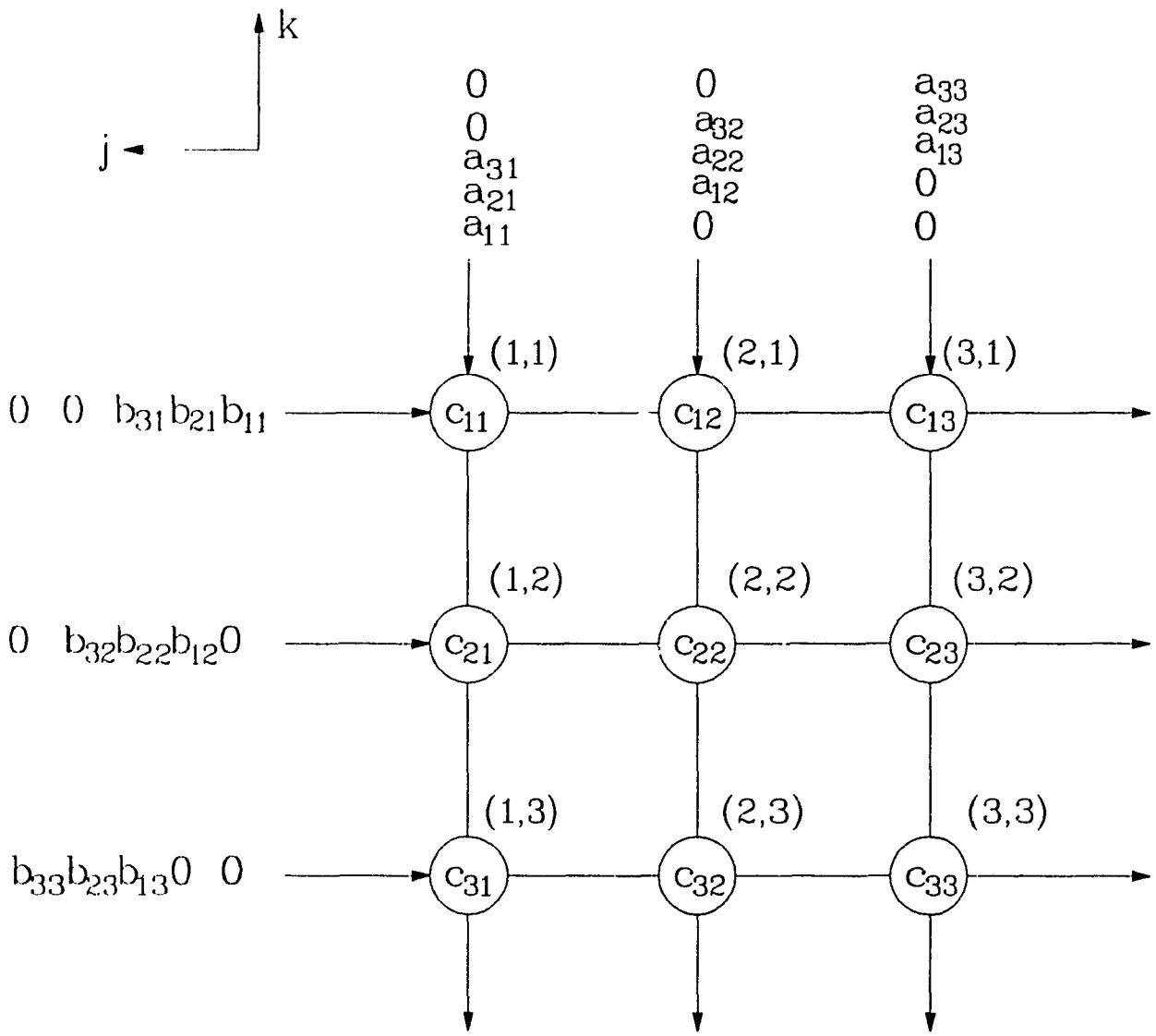


Figure 3.3 VLSI array structure using the transformation T_1 in example 2.

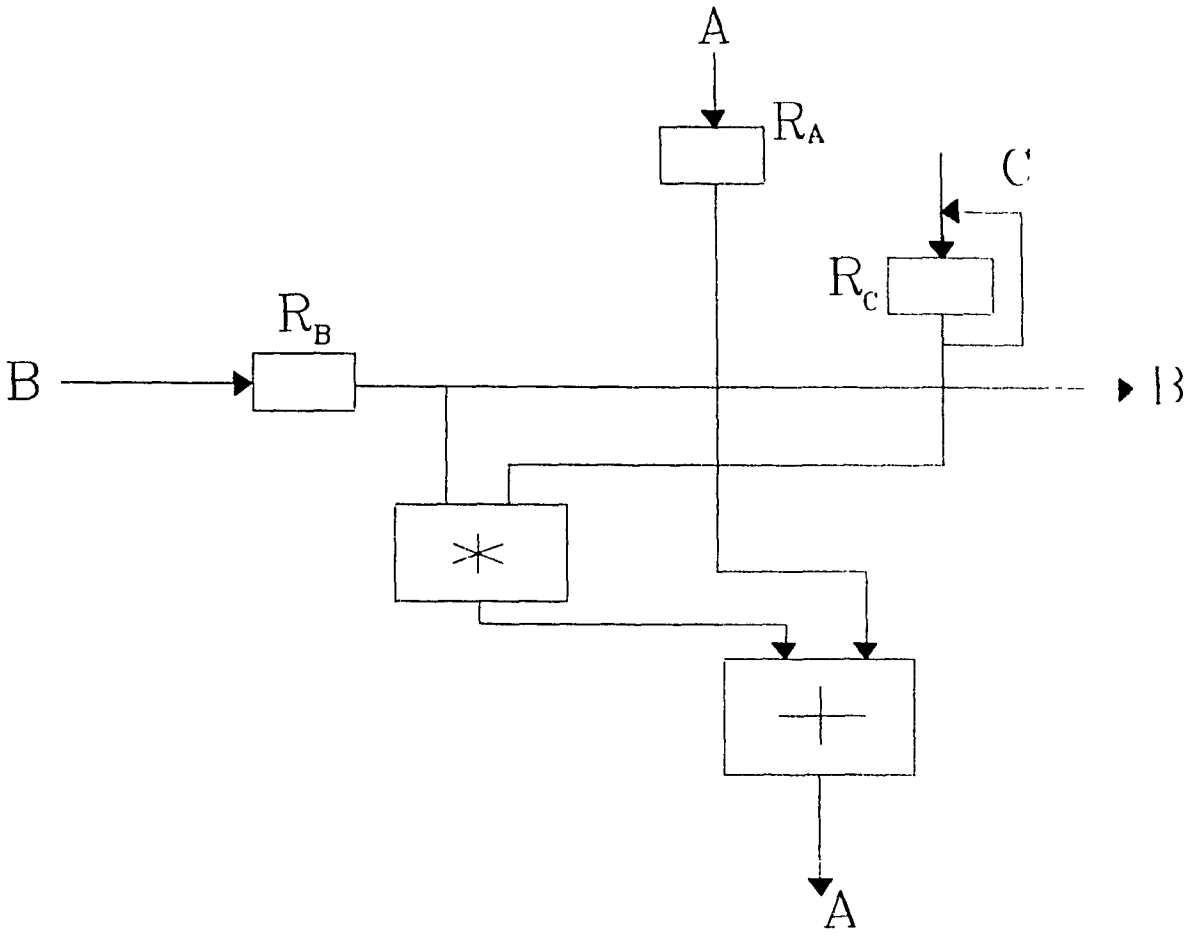


Figure 3.3(a) The structure of the cell in Figure 3.3.

TDM	Throughput (No. of cycles C)
Δ_1	5
Δ_2	5
Δ_3	5
Δ_4	9
Δ_5	-
Δ_6	9

Table 3.3(a) Comparison of the Throughput of the generated TDM's of example 1 .

TDM	Throughput (No. of cycles C)
Δ_1	7
Δ_2	7
Δ_3	7
Δ_4	7
Δ_5	13
Δ_6	9

Table 3.3(b) Comparison of the Throughput of the generated TDM's of example 2 .

From table 3.3(a), we find that Δ_1,Δ_2 and Δ_3 have the best throughput of 5 cycles respectively. This is followed by Δ_4 and Δ_6, with throughput of 9 cycles respectively. Therefore, to maximize the throughput, any of the TDM's Δ_1,Δ_2 or Δ_3 can be selected. If we select Δ_1, the corresponding VLSI array is shown in Fig.3.4 (for N=3) [20,21]. As shown in Fig.3.4, for example, the outputs of the generated variables A and B, corresponding to A(1,1,1), B(1,1,1), A(3,3,3) and B(3,3,3) are produced a time t=3, that is, after three clock cycles. It takes 5 clock cycles to complete the computation of the algorithm. The structure of the cell is depicted in Fig.3.5.

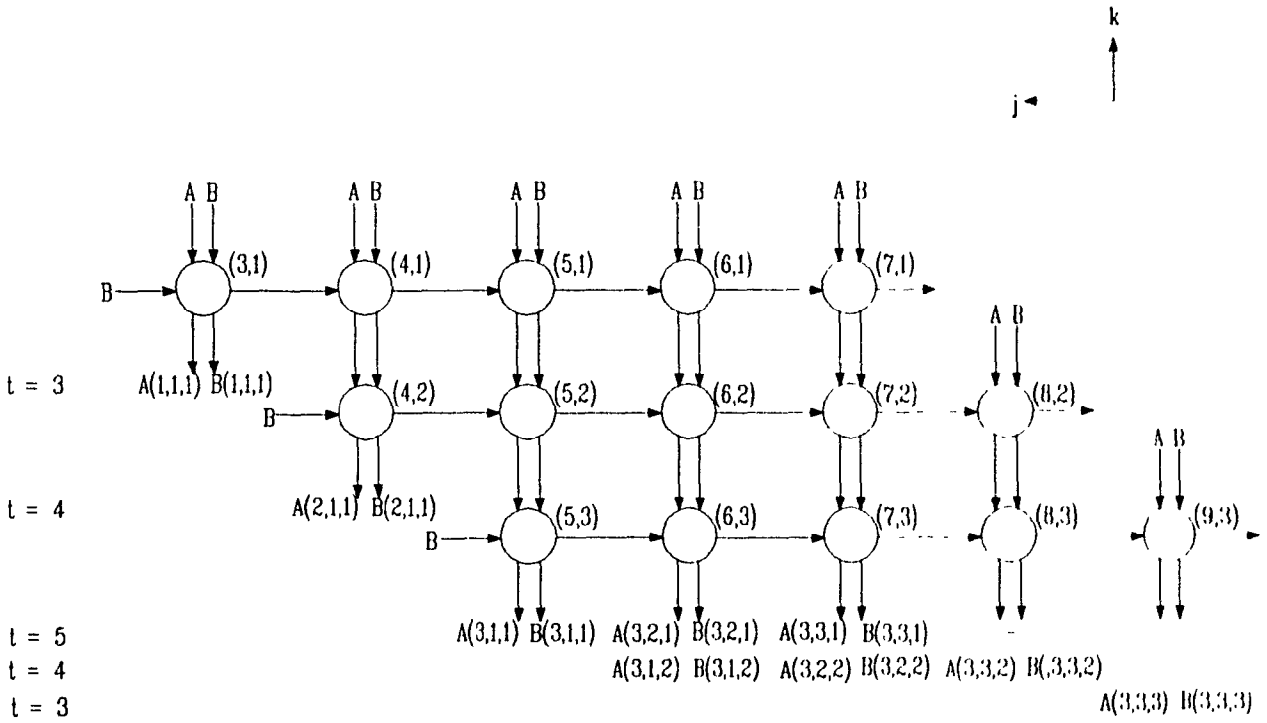


Figure 3.4 VLSI array structure using the transformation T_1 in example 1 (for N=3).

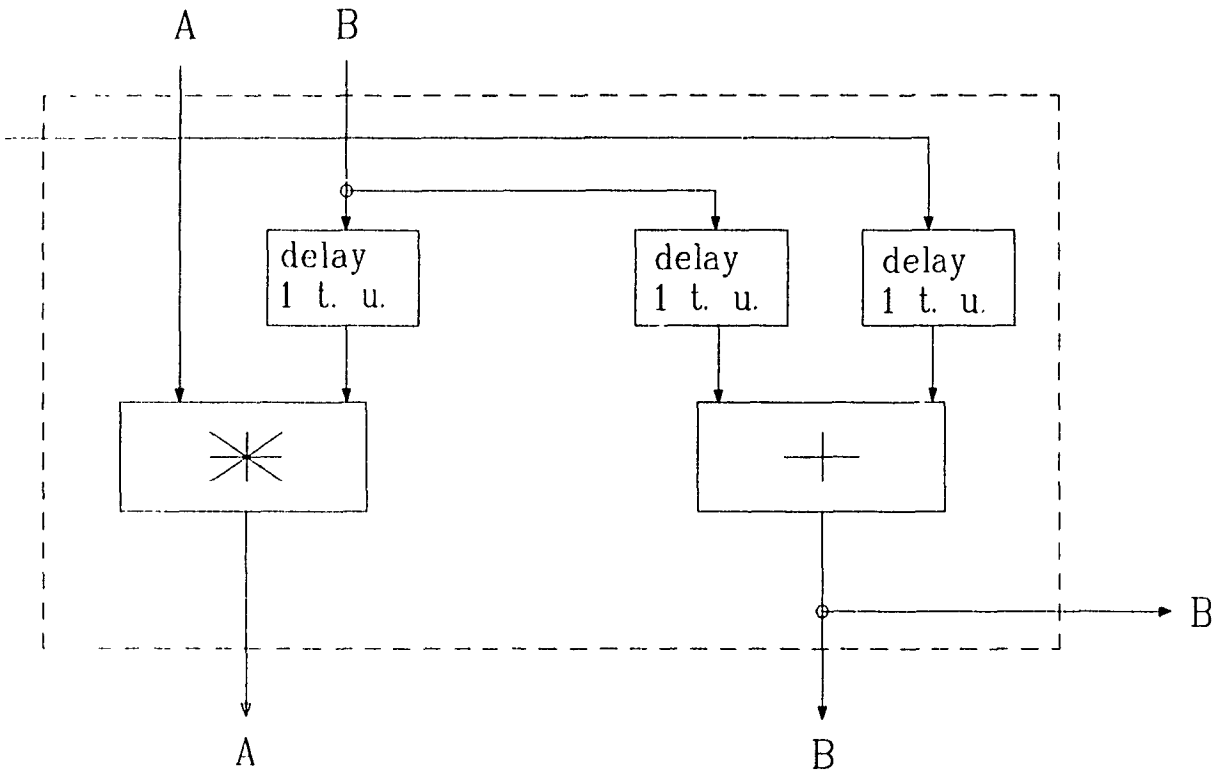


Figure 3.5 The structure of the cell in Figure 3.4.

Similarly, as seen from table 3.3(b) (example 2), Δ_1,Δ_2,Δ_3 and Δ_4 have the best throughput of 7 cycles, followed by Δ_6 and then Δ_5. In this case, either Λ_1,Λ_2,Λ_3 or Λ_4 can be selected as the TDM with the best array performance in terms of the throughput. If we select Δ_1, the corresponding VLSI array structure is the same as that shown in Fig.3.3. Also, the cell structure is the same as that of Fig.3.3(a) .

3.4.3 Optimization in terms of the Propagation Delay

The propagation delay consists of the delay in the interconnection lines. It is desired to minimize the delay required for the data to propagate through the VLSI systolic architecture. The comparison of the propagation delay of the generated TDM's in examples 1 and 2 are respectively shown in tables 3.4(a) and table 3.4(b).

TDM	Propagation Delay (τ_L)
Δ_1	1.7 NS
Δ_2	3.4 NS
Δ_3	3.4 NS
Δ_4	3.4 NS
Δ_5	-
Δ_6	3.4 NS

Table 3.4(a) Comparison of the Propagation Delay of the generated TDM's of example 1 .

TDM	Propagation Delay (τ_L)
Δ_1	1.7 NS
Δ_2	3.4 NS
Δ_3	6.8 NS
Δ_4	8.5 NS
Δ_5	3.4 NS
Δ_6	1.7 NS

Table 3.4(b) Comparison of the Propagation Delay of the generated TDM's of example 2 .

In table 3.4(a), Δ_1 has the best performance in terms of the propagation delay, since it has the lowest propagation delay among all the TDM's in example 1. Thus, to minimize the propagation delay of the systolic array design, we will select Δ_1. The corresponding VLSI structure is the same as shown in Fig.3.4 and the cell structure is the same as that in Fig.3.5. On the other hand, in table 3.4(b), Δ_1 and Δ_6 have the lowest propagation delay value. Either of them can be chosen in order to minimize the propagation delay of the systolic array. If we choose Δ_1, then we obtain the systolic array shown in Fig.3.3 . However, if we choose Δ_6, the corresponding VLSI array structure is shown in Fig.3.6. The cell structure is as shown in Fig.3.7.

3.4.4 Optimization with respect to AT

In this subsection, we will optimize the systolic array in terms of the product of the silicon area and the total computation time of the algorithm (AT). The comparison of the generated TDM's in terms of this optimization parameter is shown in table 3.5(a) for example 1 and table 3.5(b) for example 2.

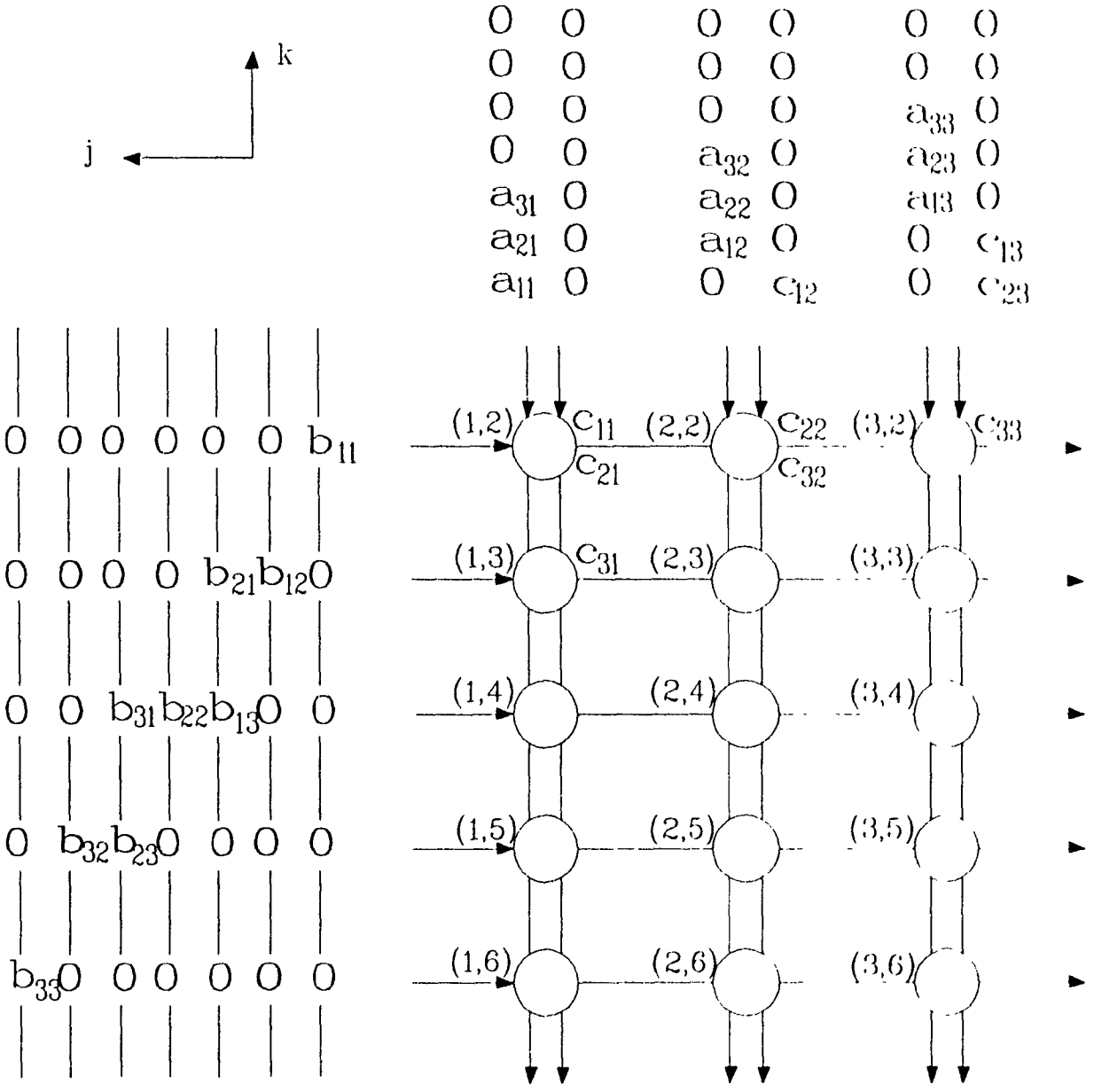


Figure 3.6 Optimal VLSI array structure for matrix multiplication (for N=3).

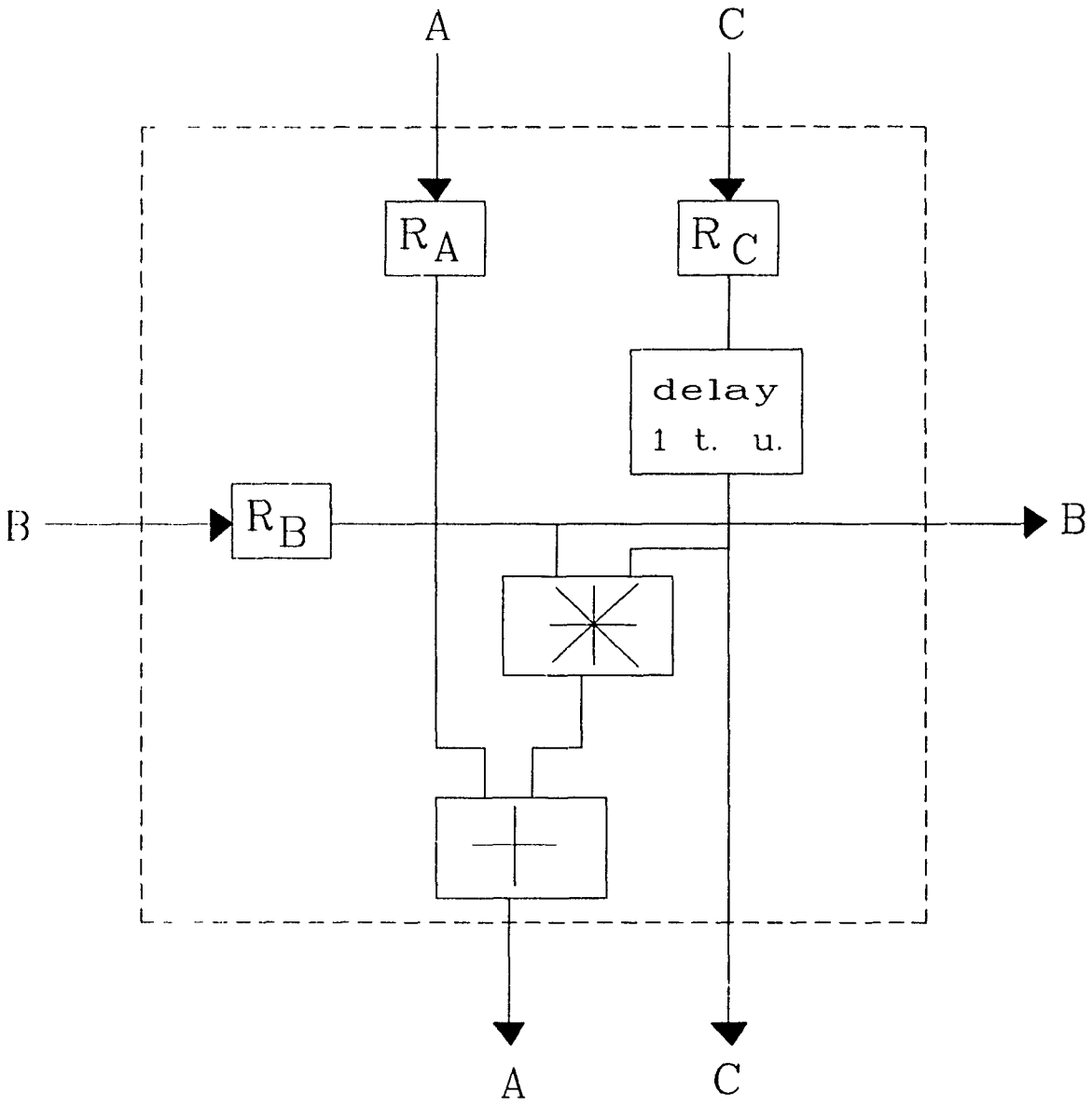


Figure 3.7 The structure of the cell in Figure 3.6.

TDM	AT Values
Δ_1	2.085×10^1 μm^2sec
Δ_2	2.127×10^1 μm^2sec
Δ_3	2.135×10^1 μm^2sec
Δ_4	3.537×10^1 μm^2sec
Δ_5	-
Δ_6	4.970×10^1 μm^2sec

Table 3.5(a) Comparison of the AT values of the generated TDM's of example 1 .

TDM	AT Values
Δ_1	1.611×10^1 μm^2sec
Δ_2	3.478×10^1 μm^2sec
Δ_3	4.772×10^1 μm^2sec
Δ_4	4.118×10^1 μm^2sec
Δ_5	6.842×10^1 μm^2sec
Δ_6	3.534×10^1 μm^2sec

Table 3.5(b) Comparison of the AT values of the generated TDM's of example 2 .

As seen in table 3.5(a), Δ_1 has the lowest AT value of 2.085×10^1 μm^2sec. Since our objective function is to select the TDM with the minimum cost, therefore, we select Δ_1 whose array structure is shown in Fig.3.4. Similarly, in table 3.5(b), Λ_1 (example 2) has the best AT performance (minimum value of AT). The AT value is 1.611×10^1 μm^2sec. This is followed by Δ_2,Δ_6, etc. Therefore, in this case, since Λ_1 will give the best performance in terms of the AT cost function, it is then selected. The VLSI array is already shown in Fig.3.3 .

3.4.5 Optimization with respect to AT^2

In Eq.(3.24), to optimize the systolic array in terms of the AT^2 value, the values of the coefficients $a1$,$a2$ and $a4$ can be set to 0 while $a3 = 1$. Table 3.6(a) consists of the comparison of the generated TDM's in example 1 for the AT^2 parameter. Table 3.6(b) shows the corresponding results but for the generated TDM's in example 2.

TDM	AT^2 Values
Δ_1	1.0602497×10^{-5} $\mu m^2 sec^2$
Δ_2	1.0998408×10^{-5} $\mu m^2 sec^2$
Δ_3	1.1036897×10^{-5} $\mu m^2 sec^2$
Δ_4	3.2912086×10^{-5} $\mu m^2 sec^2$
Δ_5	-
Δ_6	4.6251509×10^{-5} $\mu m^2 sec^2$

Table 3.6(a) Comparison of the AT^2 values of the generated TDM's of example 1 .

TDM	AT^2 Values
Δ_1	1.1468718×10^{-5} $\mu m^2 sec^2$
Δ_2	2.5171277×10^{-5} $\mu m^2 sec^2$
Δ_3	3.5669366×10^{-5} $\mu m^2 sec^2$
Δ_4	3.1272587×10^{-5} $\mu m^2 sec^2$
Δ_5	9.1964809×10^{-5} $\mu m^2 sec^2$
Δ_6	3.2346458×10^{-5} $\mu m^2 sec^2$

Table 3.6(b) Comparison of the AT^2 values of the generated TDM's of example 2 .

In table 3.6(a), Δ_1 has the minimum AT^2 value of 1.0602497×10^{-5} $\mu^2 sec^2$. This is followed by Δ_2,Δ_3, etc. With regard to this cost function, therefore, Δ_1 will be selected since it gives the best performance measure. The VLSI array is shown in Fig.3.4.

Similarly, n table 3.6(b) (example 2), Δ_1 has the lowest AT^2 value of 1.1468718 x 10^{-5} $\mu m^2 sec^2$. Hence it is selected. The VLSI array structure is the same as in Fig.3.3.

3.4.6 Optimization with respect to Speedup per PE

In order to optimize the systolic array in terms of the speedup per PE, the coefficients of $a1$,$a2$ and $a3$ in Eq.(3.24) can be set to 0 while $a4$ is set to 1. The comparison of the generated TDM's in example 1, for this cost function, is depicted in table 3.7(a). Table 3.7(b) shows the corresponding results for the generated TDM's in example 2.

TDM	SP
Δ_1	1.161891×10^{-2}
Δ_2	1.161891×10^{-2}
Δ_3	1.161891×10^{-2}
Δ_4	8.606949×10^{-3}
Δ_5	-
Δ_6	3.896666×10^{-3}

Table 3.7(a) Comparison of the Speedup per PE values of the generated TDM's of example 1 .

TDM	SP Values
Δ_1	3.571429×10^{-2}
Δ_2	7.714286×10^{-3}
Δ_3	5.142857×10^{-3}
Δ_4	6.683168×10^{-3}
Δ_5	4.153846×10^{-3}
Δ_6	8.437500×10^{-3}

Table 3.7(b) Comparison of the AT values of the generated TDM's of example 2 .

As seen in table 3.7(a), Δ_1,Δ_2 and Δ_3 have the same speedup per PE value, which is the maximum among all the SP values. Since our objective here is to select a TDM which improves the speedup per PE of the systolic array, therefore, either Δ_1,Δ_2 or Δ_3 can be selected. Assuming that Δ_1 is selected as the TDM which offers the best improvement, then the corresponding VLSI array is as shown in Fig.3.4. Similarly, in table 3.7(b), Λ_1 has the best SP value of 3.571429×10^{-2}. This is followed by Δ_6,Δ_2,Δ_4,Δ_3 and finally Λ_5. Thus, for this example, given the cost function of SP, Δ_1 is selected. The VLSI array corresponds to that shown in Fig.3.3.

3.4.7 Optimization in terms of the *COF*

Using the above design values for the unknown parameters in the cost function, the values of the respective weighted constants can be selected. By substituting these values in Eq.(3.25), the cost of the respective TDM's can be determined. We show a sample calculation of how to determine the cost of a TDM for the case of Δ_1 in example 1.

The total silicon area $(A_{si}) = 4.1004 \times 10^7 \ \mu m^2$.

The total execution time $T_c = 508.5$ NS

Assuming $c_1=c_2=c_3=c_4=1$

therefore,

$a2 = 6.2367627 \times 10^3 \ \mu m^2 sec$, $a3 = 1$ and $a4 = 1.5903745 \times 10^{-4} \ \mu m^2 sec^2$

Thus,

$$COF_1 = 2.3052836 \times 10^{-5} \ \mu m^2 sec^2$$

The other cost functions can be evaluated in a similar fashion. The performance measure of the respective TDM's in example 1, given the compound objective function, is shown in table 3.8.

COF_1	=	2.3052836×10^{-5}	$\mu m^2 sec^2$
COF_2	=	3.4912067×10^{-5}	$\mu m^2 sec^2$
COF_3	=	3.5034243×10^{-5}	$\mu m^2 sec^2$
COF_4	=	1.6880952×10^{-4}	$\mu m^2 sec^2$
COF_5	=	—	
COF_6	=	1.4247500×10^{-4}	$\mu m^2 sec^2$

Table 3.8 The performance measure of the respective TDM's in example 1.

As seen from table 3.8, Δ_1 corresponds to the TDM with the minimum cost. It has a unified AT^2 value of 2.3052836×10^{-5} $\mu m^2 sec^2$, which represents the best overall array performance. Since our objective is to select a TDM such that the compound objective function is minimized, therefore we select ,

$$\Delta_1 = \begin{bmatrix} -1 & -2 & -3 & -2 \\ 0 & 0 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

Analysis of the above calculations clearly shows that, since Δ_1, Δ_2 and Δ_3 have similar architectural features, either Δ_2 or Δ_3 would have been selected. However, due to the fact that the effect of routing is more dominant in Δ_2 and Δ_3, thus, Δ_1 is selected instead. Δ_2 has one diagonal routing path $\begin{bmatrix} -1 \\ -1 \end{bmatrix}$ and Δ_3 has three diagonal routing paths, but Δ_1 does not have any. This means that the routing area and the interconnection line delay are more in Δ_2 and Δ_3 than in Δ_1, and as result Δ_1 gives a better performance measure than Δ_2 or Δ_3.

It is also interesting to note that either Δ_4 or Δ_6 is not selected. Both Δ_4 and Δ_6 do not have extra delay units in their processors as does Δ_1, therefore, their silicon area comprises of only the area of the processors and the interconnection line routing area. Δ_1 produces an array which requires 15 PE's and 5 clock cycles to perform 27 (N^3, for N=3) computations of matrix multiplication. On the other hand, Δ_4 uses 15 PE's and 9 clock cycles to do the same task, while Δ_6 utilizes 21 PE's and 9 clock cycles to perform the

same number of computations. Therefore, Δ_1 has more available computational power than Δ_4 or Δ_6. Also, the power consumption is less in Δ_1 followed by Δ_4 and then Δ_6. Δ_4 has one diagonal routing path while Δ_6 has two diagonal routing paths, consequently, the interconnection line delay is more in Δ_4 and Δ_6 than in Δ_1. Although Δ_4 and Δ_6 do not have extra delay units in their processors, however, their respective AT^2 value is dominated by the number of clock cycles in Δ_4, the number of PE's (and hence the total area of the PE's) and the number of clock cycles in Δ_6. Therefore, Δ_4 and Δ_6 do not respectively produce the minimum overall array performance measure, as indicated in table 3.8, and hence are not selected.

Also it is interesting to see the difference between Δ_4 and Δ_6 since both have the same timing structure and similar interconnection pattern. Due to the lack of fault - tolerant capability of Δ_4, Δ_6 gives a lower COF value than Δ_4. Therefore, although one TDM may be better than another TDM given the cost of one objective function (e.g. interconnection line delay), however, the optimum TDM is the one which gives the best overall array performance given the compound objective function.

The VLSI array corresponding to the transformed dependency matrix Δ_1 is shown in Fig. 3.4 (for N=3) [20,21]. The structure of the cell is depicted in Fig. 3.5. All the cells in the array shown in Fig. 3.4 are identical. It consists of an adder, multiplier and delay elements. Variable A with dependence $\bar{d}_1$ moves from a cell to the next via a vertical channel with direction $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$, and it has no time delay associated with it, in each cell. Variable B is used for three operands, the operand which has dependence $\bar{d}_2$ moves via a vertical channel with direction $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$ and has one - time delay unit added in the cell in front of the multiplier. The second operand of variable B with dependence $\bar{d}_3$, corresponds to a vertical channel $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$ and has two - time delay units inserted in front of the adder in the cell . The last operand of variable B with dependence $\bar{d}_4$ corresponds to a horizontal channel of $\begin{bmatrix} -1 \\ 0 \end{bmatrix}$ and has one - time delay unit inserted in front of the adder

in the cell.

Table 3.9(a) consists of the comparison of the performance measure of the generated TDM's in example 2, for W = 2 . Similar results are given for the case when W = 5 and this is as shown in table 3.9(b).

COF_1	=	4.9561247×10^{-5}	$\mu m^2 sec^2$
COF_2	=	7.9203220×10^{-5}	$\mu m^2 sec^2$
COF_3	=	1.8293289×10^{-4}	$\mu m^2 sec^2$
COF_4	=	1.9202452×10^{-4}	$\mu m^2 sec^2$
COF_5	=	2.8315257×10^{-4}	$\mu m^2 sec^2$
COF_6	=	6.8786765×10^{-5}	$\mu m^2 sec^2$

Table 3.9(a) The performance measure of the respective TDM's in example 2, for W = 2 .

COF_1	=	8.3967345×10^{-5}	$\mu m^2 sec^2$
COF_2	=	7.9203220×10^{-5}	$\mu m^2 sec^2$
COF_3	=	1.8293289×10^{-4}	$\mu m^2 sec^2$
COF_4	=	1.9202452×10^{-4}	$\mu m^2 sec^2$
COF_5	=	2.8315257×10^{-4}	$\mu m^2 sec^2$
COF_6	=	6.8786765×10^{-5}	$\mu m^2 sec^2$

Table 3.9(b) The performance measure of the respective TDM's in example 2, for W = 5

In this example, as seen from table 3.9(a) (for W = 2), Δ_1 is the TDM with the minimum cost. It has a unified AT^2 value of 4.9561247×10^{-5} $\mu m^2 sec^2$, which is the best overall array performance. Since our goal is to select the TDM such that the *COF* function is minimized, then Δ_1 should be selected. However, it is important to note that, though Δ_1 gives the minimum cost in terms of the COF function, it is not suitable for fault-tolerance. The data of one of the variables (variable C) is stored in the cells. This

will be undesirable in situations where multiple output results need to be compared at the same time in order to mask dynamically errors in a fault-tolerant systolic array.

Table 3.9(b) shows the performance measure of the respective TDM's for W = 5 . In this case, Δ_6 which has a unified AT^2 value of 6.8786765 x 10^{-5} $\mu m^2 sec^2$, represents the best overall array performance. Therefore, we select ,

$$\Delta_6 = \begin{bmatrix} -1 & -1 & -2 \\ 0 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix}$$

It is worth while noting that Δ_6 is suitable for fault-tolerance since all the data variables propagate from one cell to the other. This example highlights the importance of choosing the value of W. For a very low value of W, a TDM which is not suitable for fault-tolerance might still give the best overall array performance, as seen in table 3.9(a). Therefore, the value of W should be chosen such that a TDM that lacks the fault-tolerant capability is not selected.

In Eq.(3.25), the importance of each factor or cost function is not the same. Each can be chosen according to the specific purpose of the systolic array architecture and the fabrication facilities. Similar analysis performed to support the selection of Δ_1 in example 1, can also be done for this example. Δ_6 is chosen as the optimum TDM since it gives the best overall array performance given the compound objective function. The other TDM's are not selected because they do not give the minimum overall performance measure. Some of them may give better performance than Δ_6 given one cost function, however, Δ_6 gives the best performance given the overall cost functions.

The VLSI array corresponding to the transformed dependency matrix Δ_6 is shown in Fig.3.6 (for N=3). The structure of the cell is shown in Fig.3.7. All the cells in the array of Fig.3.6 are identical. Each cell consists of an adder, a multiplier and a delay element.

3.4.8 Summary of the Comparison Results

Tables 3.10(a) and 3.10(b) give the comprehensive summary of the selected TDM's given the respective cost function. As seen in table 3.10(a), Δ_1 gives the best performance given most of the cost functions. Also, it gives the best overall array performance given the compound objective function. On the other hand, in table 3.10(b), Δ_1 gives the best performance for most of the cost functions. However, it does not give the overall array performance given the COF function, due to its lack of fault-tolerant capability. Instead Δ_6 is selected as the optimal TDM because it satisfies the desired VLSI requirements.

Cost Function	TDM
Silicon Area	Δ_4
Throughput	Δ_1 , Δ_2 or Δ_3
Propagation Delay	Δ_1
AT values	Δ_1
AT^2	Δ_1
Speedup per PE	Δ_1 , Δ_2 or Δ_3
COF	Δ_1

Table 3.10(a) The Selected TDM's in example 1 for the respective cost functions.

Cost Function	TDM
Silicon Area	Δ_1
Throughput	Δ_1 , Δ_2 , Δ_3 or Δ_4
Propagation Delay	Δ_1
AT values	Δ_1
AT^2	Δ_1
Speedup per PE	Δ_1
COF	Δ_6

Table 3.10(b) The Selected TDM's in example 2 for the respective cost functions.

3.5 CONCLUDING REMARKS

In this chapter, we have described a cost effective systematic approach for mapping algorithms into optimal systolic array architectures. We proposed a methodology for obtaining the new transformed dependency matrix (TDM) directly from the original dependency matrix (D) so as to select the TDM that meets our desired VLSI requirements. Thus, we avoid the derivation of several transformation matrices (T) and the selection of the best one which might not even give a TDM that does satisfy our requirements.

Also, we proposed a unifying performance index to measure the overall performance of the systolic array architecture taking into consideration the architectural features and the technological parameters of the array. The proposed procedure is formulated as an optimization problem to obtain the TDM with the minimum cost function. Various systolic array architectures have been compared in terms of the different cost functions. The optimal systolic array architecture given the respective performance measures has been determined for each case. It is observed that, although one TDM may be better than another TDM given the cost of one objective function, however, the optimum TDM is the one that gives the best overall array performance given the compound objective function. Therefore, not only that the proposed optimization mapping algorithm selects the TDM that meets the desired VLSI requirements, without deriving the transformation for every TDM, it also determines the optimal TDM given the respective cost function. This approach provides an efficient and a cost effective method for mapping algorithms into optimal systolic array architectures.

In the next chapter, we will present a novel approach for designing Fault-tolerant Systolic Array Architectures.

3.6 REFERENCES

[1] D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proc.IEEE,* Vol. 71, No. 1, January 1983.

[2] W. L. Miranker and A. Winkler, "Spacetime Representations of Computational Structures," *Journal of Computing*, Vol. 32, 1984.

[3] Y. Wong and J-M Delosme, "Optimal Systolic Implementations of N - dimensional Recurrences," *ICCD*, pp. 618-621, 1985.

[4] M. T. O'Keefe and J. A. B. Fortes, "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," *In International Conference on Parallel Processing*, pp. 672-675, Chicago, IL, August, 1986.

[5] J - M Delosme and I. C. F. Ispen, "Efficient systolic arrays for the solution of Toeplitz systems : An illustration of a methodology for the construction of systolic architectures in VLSI," *International Workshop on Systolic Arrays*, University of Oxford, pp. F2, July, 1986. Also in Systolic Arrays: edited by W. Moore, A. McCabe and R. Urquhart, pp. 27-46, 1987.

[6] S. K. Rao, "Regular Iterative Algorithms and Their Implementations on Processor Arrays," PhD Thesis, Stanford University, Stanford, California, 1985.

[7] J. A. B. Fortes, "Algorithm Transformations for Parallel Processing and VLSI Architecture Design," Ph.D. dissertation, Univ. Southern California, Los Angeles, CA., Dec., 1983.

[8] P. Gachet, B. Joinnault and P. Quinton, "Synthesizing Systolic Arrays using DIASTOL," *International Workshop on Systolic Arrays*, University of Oxford, pp. F2, July, 1986.

[9] C. E. Leiserson, F. M. Rose and J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *in Proceedings, Caltech VLSI Conference, Pasadena, CA, 1983.*

[10] C. S. Raghavendra, V. K. Prasanna Kumar and A. Varma, "On systolic processing with bounded I/O bandwidth," *in Proc. ICCD*, 1985.

[11] I. V. Ramakrishnan and P. J. Varman, "Synthesis of an optimal family of matrix multiplication algorithms on linear arrays," *Tech. Rep.*, Univ. of Maryland, Dept Comput. Sci., *Proc. ICPP*, 1985.

[12] P. J. Varman and I. V. Ramakrishnan, "Optimal matrix multiplication on fault-tolerant VLSI array," *Proc. ICALP*, 1985.

[13] V. K. P. Kumar and Y-C Tsai, "On Mapping Algorithms to Linear and Fault-tolerant Systolic Arrays," *IEEE Trans. on Comput.*, Vol. 38, No. 3, pp. 470-478, March, 1989.

[14] G - J Li and B. W. Wah, "The Design of Optimal Systolic Arrays," *IEEE Trans. Comput.*, Vol. C-34, No. 1, pp 66-77, January 1985.

[15] C. K. Ko and O. Wing, "Mapping Strategy for Automatic Design of Systolic Arrays," *in Proc. 1988 International Conf. on Systolic Arrays*, pp. 285-294, 1988.

[16] H. B. Lee and R. O. Grondin, "A Comparison of Systolic Architectures for Matrix Multiplication," *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 1, pp. 285-289, February 1988.

[17] R. W. Keyes, "Physical Limits in Digital Electronics," *Proc. IEEE*, Vol. 63, No. 5, pp. 740-767, May, 1975.

[18] R. O. Grondin, W. Porod and D. K. Ferry, "Delay Time and Signal Propagation in Large - Scale Integrated Circuits," *IEEE J. Solid - State Circuits, Vol. SC-19, No. 2, pp. 263-263, April, 1984.*

[19] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "Design of Optimal Systolic Arrays : A Systematic Approach," *IEEF Symposium on Parallel and Distributed Processing*, Dallas, Texas, pp. 166-173, Dec. 9-11, 1990.

[20] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "Systolic Arrays: How to choose them," accepted for publication *in Part E of the IEE Proceedings, Computers and Digital Techniques, 1991.*

[21] D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays'" *IEEE Trans. Comput.*, Vol. C-35, No. 1, pp. 1-12, January 1986.

[22] D. I. Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays," *IEEE Trans. Comput.-Aided Design,* Vol. CAD-6 January 1987.

[23] V. Poorniah and O. Ahmad, "Design of Systolic Cells using Domino Logic," Tech. Rep., Dept. Elec/Comput. Engr. Concordia University, Montreal, Que., 1990.

[24] H. F. Li, C. N. Zhang and R. Jayakumar, "Latency of Computational Data Flow and Concurrent Error Detection in Systolic Arrays," *CCVLSI '89*, pp. 251-258, 1989.

CHAPTER IV

SYSTEMATIC APPROACH FOR DESIGNING FAULT - TOLERANT SYSTOLIC ARRAY ARCHITECTURES

4.1 INTRODUCTION

As mentioned in Chapters I and III, we are interested in developing methodologies for designing systolic array architectures that satisfy some important VLSI requirements. One of those requirements is that the systolic array should have the *Fault - tolerant* (FT) capability. In view of this, this chapter will focus on a systematic approach for designing fault - tolerant systolic arrays. However, before we proceed to propose our approach of designing fault - tolerant systolic architectures, we will first review some work that has been done in this area.

The organization of this chapter is as follows: Section 4.2 briefly discusses the importance and basic concepts of fault - tolerance. It also gives the various kinds of fault - tolerant techniques for systolic arrays. In section 4.3, we will describe in detail our fault tolerant mapping technique and apply it to an algorithm for matrix multiplication in order to demonstrate the generality and novelty of our approach to design fault-tolerant systolic arrays. Section 4.4 discusses the fault tolerance capability of the proposed design scheme and compares it with the other relevant design schemes. Finally, section 4.5 contains the summary and the concluding remarks.

4.2 THE IMPORTANCE AND BASIC CONCEPTS OF FAULT - TOLERANCE

Recent advances in Very Large Scale Integration (VLSI) and Wafer Scale Integration (WSI) technologies have made it possible to manufacture special purpose VLSI chips

with multiple copies of low-cost processors to provide a large amount of computational capability for a small cost [1,2]. With the increase in the complexity of such VLSI chips, the probability of physical failures occurring on a chip increases. Since each processor in such a system contributes to the computation, any single temporary or permanent failure in a processor can break down an entire computing system, therefore, *reliability* and *fault tolerance* have increasingly become inevitably important design issues [3].

Reliability is defined to be the probability that a given system will perform its required function under specified conditions for a specified period of time [3]. There are two fundamentally different approaches that can be taken to increase the reliability of computing systems [4,5]. The first approach is called *fault prevention* (also known as *fault intolerance*). This approach employs the method of worst case design, using high - quality components and imposing strict quality control procedures during the assembly phase. However, such measures can increase the cost of the system significantly, and it is not related to VLSI directly. Since this is almost impossible to achieve in practice, the goal of fault prevention is to reduce the probability of system failure to an acceptably low value. Hence, an alternative approach to achieve high reliability is through the use of fault - tolerant systems.

In fault-tolerant approaches, faults are expected to occur during computation, but their effects are automatically counteracted by incorporating *redundancy*, i.e., additional facilities, into a systolic system, so that valid computation can continue even in the presence of faults. The redundancy introduced can be of several forms: hardware, software, time, or a combination of all of these. They are redundant in the sense that they could be omitted from a fault-free system without affecting its operations [4]. This approach does not require the use of high quality components, instead, standard components can be used in a redundant and reconfigurable architecture. As a result of the decreasing cost of hardware components, due to the advances in VLSI technology, it is certainly less expensive to use the fault-tolerance approach to design reliable systems than the fault-

prevention approach.

The main purpose of a fault-tolerant system [6], whether centralized or distributed, is to produce correct results even in the presence of faulty units. The secondary objectives are to provide fault - tolerance at maximum performance and to minimize the hardware overhead. Hence, the most widely accepted definition of a fault-tolerant computing system is that *it is a system which has the built-in capability to preserve the continued correct execution of its programs and input/output functions in the presence of a certain set of operational faults* [4]. An *operational fault* is an unspecified deviation of the correct value of a logic variable in the system hardware or a design fault in the software. *Correct execution* means that the programs, the data and the results do not contain errors and that the execution time does not exceed a specified limit. There are two types of faults encountered during system operation and these are the *anticipated* and the *unanticipated* faults [4]. *Anticipated faults* are those whose occurrence in an operational system can be foreseen. An example of this type of fault is the inevitable deterioration of hardware components in a system, giving rise to faults. On the other hand, *unanticipated faults* cannot be foreseen but affect the operation of the system. For example, a VLSI chip can fail in so many modes that it is almost impossible to anticipate the consequences. It is important therefore, that fault-tolerance of both anticipated and unanticipated faults be taken into consideration in high reliability applications.

Fault-tolerance involves the following steps [7]: (i) fault detection ; (ii) fault diagnosis (identification of the faulty module); (iii) fault recovery (correction of the error) and (iv) system reconfiguration. The detection and correction can be achieved in many different ways. For example, checks on the computation can detect an error, and the computation can be rolled back to a previous state in order to correct the error. On the other hand, if there is sufficient redundancy in the computation, an error due to a module failure can be masked by the correct values from other modules. This technique is used in highly redundant systems for space applications. Another way is by reconfiguration of the

system to isolate the faulty module. This requires of course, that the faulty module be identified first.

In the following subsection, we will address the redundancy techniques to achieve fault-tolerance, i.e. to detect and correct errors and to identify faulty modules.

4.2.1 Redundancy Techniques

Fault-tolerance can be achieved through redundancy and various techniques have been proposed over the years [7]. These have usually been general techniques that can be applied at the module level in a system. The generality makes them applicable to most problems. There are three redundancy approaches to fault-tolerance and these include: *space, time* and *algorithmic* redundancies. *Space* redundancy can be static, hybrid or dynamic. In the static redundancy approach, also known as *masking redundancy*, N copies (where N is odd) of a module and a majority voter are used to mask the errors from failed modules. The system will tolerate upto (N-1)/2 faulty modules. This scheme is known as N - Modular Redundancy (NMR). A popular version with N=3 is called the Triple Modular Redundancy (TMR) and several TMR strategies have been proposed for systolic arrays [4-5,8-11]. The static technique can be combined with a set of spares through the use of a disagreement detector and a switching unit to produce what is known as a *hybrid redundant* system [4]. In the dynamic case, the faulty modules are identified and the system is reconfigured by replacing faulty modules by spares. Various array reconfiguration strategies have been proposed by many authors [7,12-25].

Conventional time redundancy [9,26-28] involves recomputing the same computation twice in the same module or in adjacent modules at two different but close enough time periods and then comparing the results. If they match, there is no fault. This has the advantages of eliminating wrong results due to the intermittent faults. However, permanent faults may not be detected. Although time redundancy is a powerful technique that, in general, requires only a small amount of hardware overhead, it has a fundamental

limitation when applied to high - performance systolic arrays. Many of these systems are designed for high throughput signal-processing applications, and the 100 percent degradation in throughput required by the time redundancy techniques may adversely affect the system performance [7].

Algorithm-based fault-tolerance has been proposed [6,7,29-35], based on a data - encoding approach. The input data to the algorithm are encoded at the system level in the form of error-correcting or error-detecting codes. The original algorithm must then be redesigned to operate on the encoded data and to produce encoded output data. This redundancy would enable the correct data to be recovered, or at least, to recognize that the data are erroneous. The checksum scheme [7,29-32] has been employed as an appropriate low - cost code to achieve fault-tolerance in systolic arrays.

Having outlined the various techniques to achieve fault-tolerance in highly parallel computing structures, and in systolic arrays in particular, in the next section, we will present a new approach for designing fault-tolerant systolic array architectures.

4.3 MAPPING ALGORITHMS INTO NON-OPTIMAL AND OPTIMAL FAULT-TOLERANT SYSTOLIC ARCHITECTURES

The conventional approach of designing fault tolerant systolic architectures is based on the mapping of an algorithm onto a specific VLSI systolic architecture, and then modifying the design to make it fault-tolerant [12-25]. In most cases, the techniques employed to make a particular systolic architecture fault-tolerant is specific only to that systolic architecture and thus may not be applicable to another systolic array with different topology and data flow characteristics. In our approach, fault - tolerant algorithms are designed by introducing redundant computations at the algorithmic level, so that when these algorithms are mapped into specific VLSI systolic architectures, using the Space - Time (S-T) mapping techniques (section 2.3), the architectures will be inherently fault-tolerant. The

VLSI array need not be modified in order to make them fault-tolerant. The mapping procedure is based on the mathematical transformations of index sets and data dependence vectors.

We introduce redundant computations in the original algorithm by obtaining different versions of the algorithm. This is done by first, obtaining the Dependency matrix (D) of the algorithm, and then modifying the dependency matrix by letting different indices in the index set, one at a time, determine the execution ordering of the computations. In each case, the motivation is that if only one coordinate of the index set preserves the correctness of the computation by maintaining an execution ordering, then the rest of the index coordinates can be selected to meet some VLSI communication requirements. Then, the different dependency matrices corresponding to the different versions of the algorithm are mapped into systolic array architectures. Our approach consists of three steps: in the first step, the dependency matrix (D) of an algorithm is modified to reflect a given fault-tolerance requirement; in the second step, mapping techniques are applied to the fault-tolerant dependency matrix to obtain a fault - tolerant systolic array implementation of that algorithm. In most cases, this array is not the optimal systolic array design for the given algorithm. Therefore, alternatively, we apply our proposed unifying performance index which is used to measure the overall array performance (section 3.3), to select the transformed dependency matrix which gives the optimal systolic array design. Finally, in the third step, the fault-tolerant transformed dependency matrix is translated into a fault-tolerant systolic array implementation of the given algorithm.

Three fault-tolerant mapping methods are investigated; in the first method, one version of the algorithm is mapped into a VLSI systolic array and then this array is replicated [10,11]. In the second method, three versions of the algorithm are obtained and each is mapped into a separate VLSI systolic array architecture. A fault-tolerant systolic array is constructed by merging the respective systolic arrays of the three versions of the algorithm. In the third method, the different dependency matrices corresponding to the

different versions of the algorithm are merged together to obtain a fault-tolerant dependency matrix. The merged dependency matrix is mapped into an optimal fault-tolerant systolic array architecture using the approach proposed in (section 3.3). Finally, an example is given where, the three transformed dependency matrices that give the near optimal systolic array designs are derived, and then respectively, mapped into systolic architectures. In the first method, the data flow characteristics for the replicated copies of the VLSI design are the same and hence, they exhibit the same dynamic properties. Any temporary hard fault in one copy of the result will also affect the other copies in a similar manner. Therefore, such faults, if they occur will go undetected and hence cannot be tolerated. In the second and third methods, the different versions of the algorithm exhibit different dynamic properties since their data flow characteristics are not the same. The effects of any temporary hardware fault on the results produced by one version of the algorithm will not be the same as for the other versions. Thus, such faults, if present in the VLSI array, can be tolerated. In our approach, we adopt the spatial redundancy technique to achieve fault-tolerance and mask dynamically, errors using fault-tolerant voting schemes.

We will briefly describe below the fault model that will be used for the fault-tolerant design.

4.3.1 Fault Model

Faults may be inherent in the specification or design of a system. Also, they may have physical causes, being introduced during the manufacture of the system or due to wear-out in the field. Tolerance of design and specification faults has been an important area of study [36]. Problems in preventing faults in hardware and tolerating faults in software have been studied [37]. In this thesis, we will consider only the problem of tolerating physical failures.

Any fault-tolerance technique is designed to tolerate a given class of faults within a system. A fault can be treated at any level of integration within the system, from a very low level, such as the transistor level, to a higher level such as the logic, register transfer, or module level. The description of the effects of physical failures at these different levels of integration is referred to as the *fault model* [38]. Most fault-tolerance techniques have been designed to tolerate faults in some module within a system. Such a *module - level fault model* [7] is ideal for VLSI. This is because as the geometric features of VLSI integrated circuits are scaled down, any failure in a small area of the circuit will affect a greater amount of the circuitry and will cause a large block of logic to become faulty. As a result, the traditional gate level single stuck-at fault assumption is not sufficient for VLSI technology. Therefore we assume a generalized fault model for the processing elements (PE's) or cells in the systolic array architecture.

We will assume that faults can occur in the refined level of the processing element such as the computation unit, input/output latch registers, communication links and switches. We will also assume that at most one of these modules or parts of the PE is faulty at any given time. Furthermore, it is assumed that both temporary and permanent faults can occur in the array. Finally, it is assumed that the outputs of the faulty cells may assume any logical values independent of the inputs.

4.3.2 Fault-Tolerant Mapping Techniques

Designing a fault-tolerant computing system involves fault-detection, fault containment, fault-diagnosis, and fault recovery [9]. Static redundancy, in which multiple modules perform the same operations and faults can be detected concurrently by comparing the outputs of these redundant modules, can be used to achieve all these steps concurrently. The faulty modules are isolated and then prevented from disrupting the rest of the system as long as the majority vote can be attained. Triple Modular Redundancy (TMR) is a special case of the static redundancy in which all system components are

triplicated.

In this subsection, we will describe four methods for mapping algorithms into fault-tolerant systolic arrays using TMR configuration.

4.3.2.1 Method (1) : Triplicating One Version Of The Algorithm .

In order to illustrate our fault-tolerant (FT) design approach, we will use the matrix multiplication algorithm described in section 2.3, and whose dependency matrix is shown in Eq.(2.6). Here, the one version of the algorithm that we will use corresponds to the case when index i is assumed to determine the valid execution ordering of the computations (section 2.3). The VLSI array is as shown in Fig.4.1 while the cell structure is as shown in Fig.4.2. The VLSI structure of Fig.4.1 is triplicated and the results are voted on using redundant voters as shown in Fig.4.3. Apart from the mapping technique, this TMR method is a standard technique.

4.3.2.2 Method (2) : Creating Three Different Versions Of The Algorithm.

This subsection describes the ideas of designing fault- tolerant algorithms, and the mapping of the fault-tolerant algorithms into VLSI systolic array architectures. Different versions of an algorithm are obtained by letting the different indices (i,j,k) in the algorithm respectively, determine the execution order of the algorithm. In the above example, index i is assumed to determine the valid execution order of the computations, however, the same procedure is applicable to other indices in the algorithm.

If we assume index j (for example) to determine the valid execution ordering of the computations, therefore, to satisfy the constraints of finding a valid transformation matrix, we need to select a mapping such that the transformed data dependency matrix $\Delta = \hat{D} = TD$, has negative entries in the second row. Using the steps of finding a valid transformation matrix described in section 2.3, and applying them to the example of the

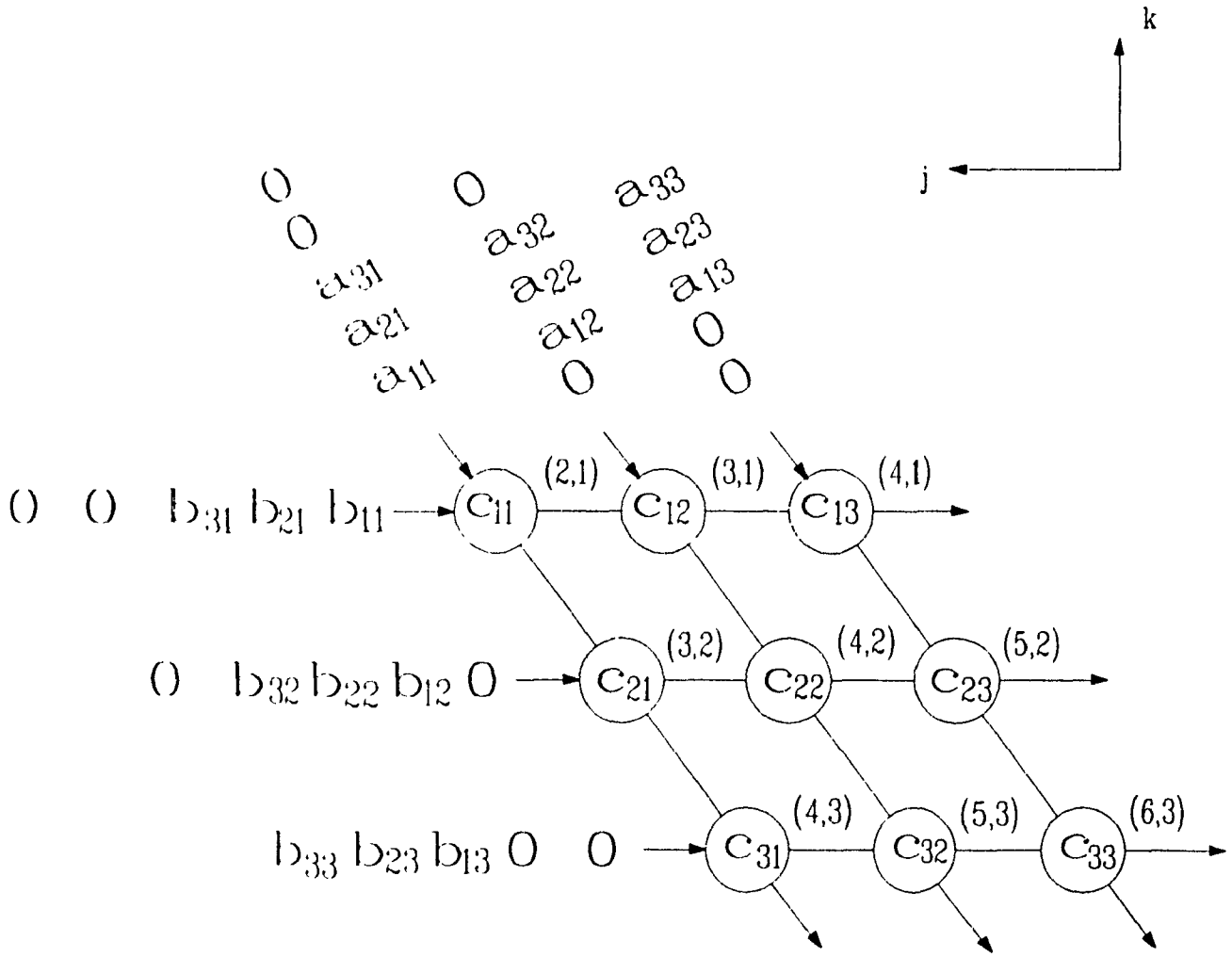


Figure 4.1 VLSI array structure when index i determines the timing (or valid execution ordering) of computations (for N=3).

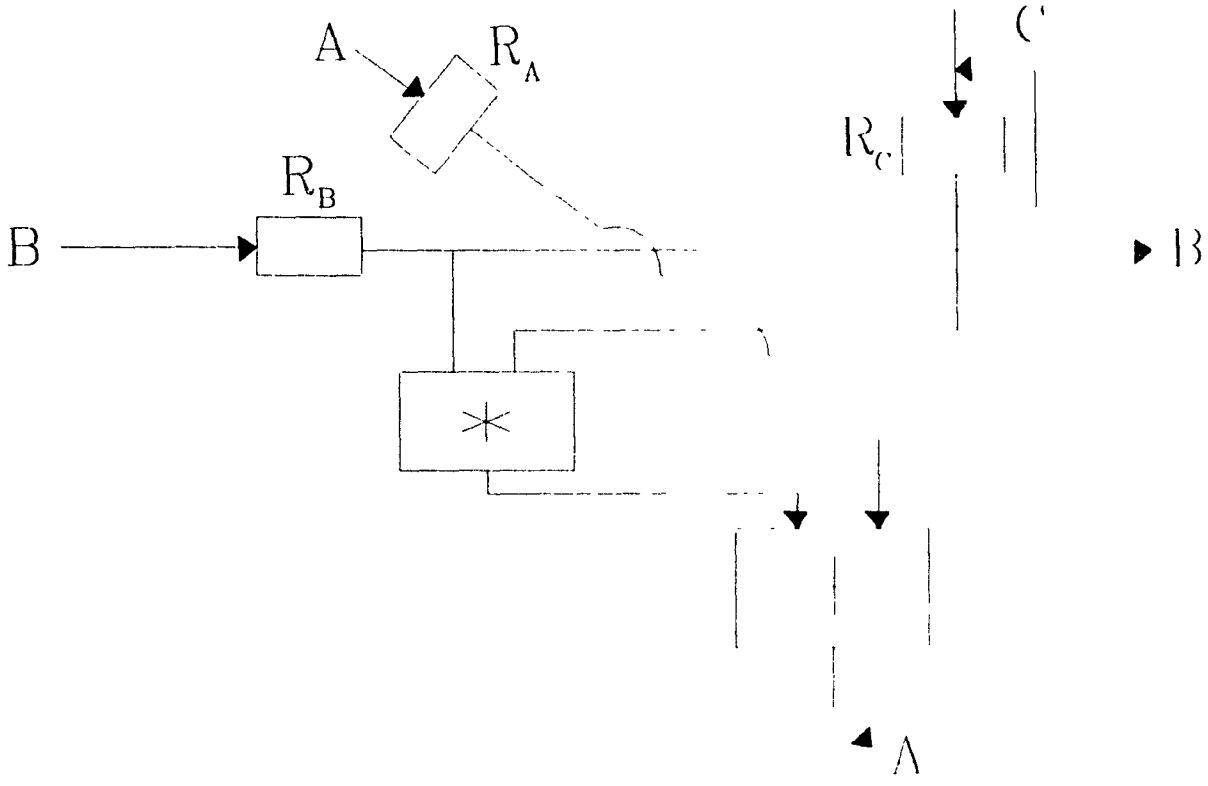


Figure 4.2 The structure of the cell in Figure 4.1.

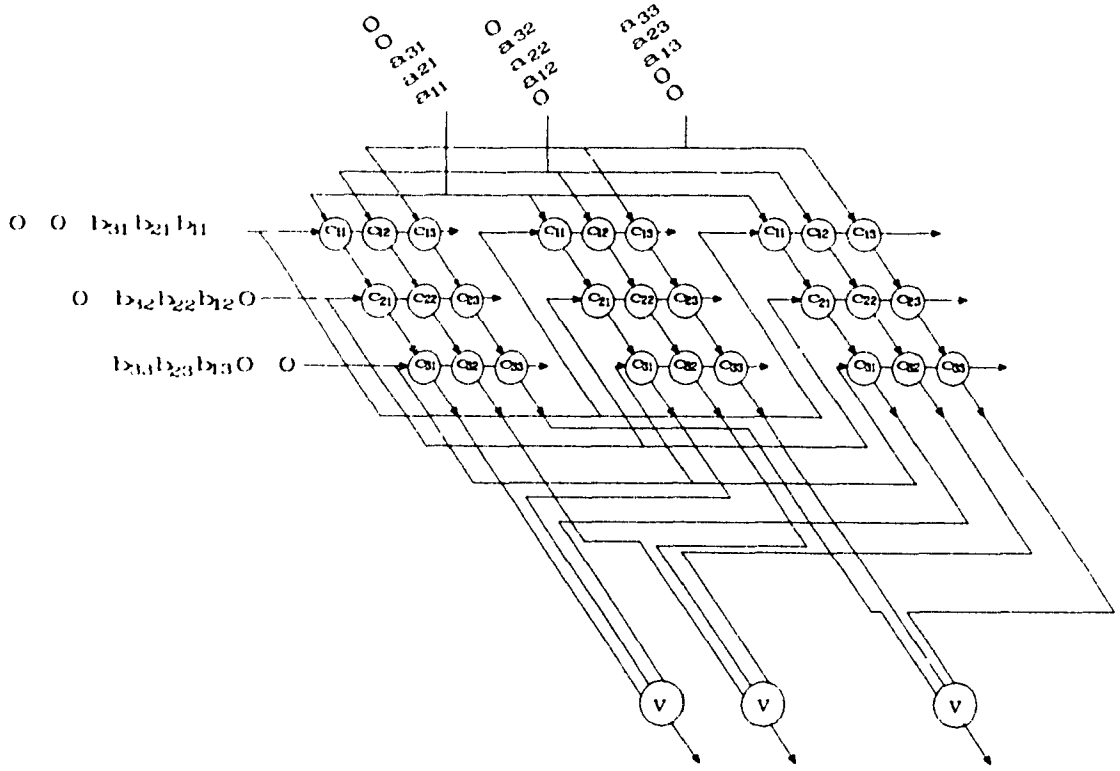


Figure 4.3 Fault-tolerant systolic array for matrix multiplication with one version of the algorithm triplicated.

matrix multiplication algorithm, we can choose the transformation matrix as follows :

$$T = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

That is

$$\Delta = \hat{D} = TD = \begin{matrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{matrix} \begin{bmatrix} -1 & 0 & -1 \\ -1 & -1 & -1 \\ -1 & 0 & 0 \end{bmatrix} \tag{4.2}$$

The mapping of the index set is given as follows:

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i + k \\ i + j + k \\ k \end{bmatrix} = \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} \tag{4.3}$$

In this case, the second coordinate $\hat{j}$ indicates the time at which computation indexed by corresponding (i,j,k) in the algorithm is computed and $(\hat{k},\hat{i})$ indicates the processor where the computation is performed. Figure 4.4 depicts the VLSI array structure. The cell structure is similar to that shown in Fig. 4.2 except that the data for variable B is stored in the node of the cell. Variable A moves from one cell to the next diagonally with direction $\begin{bmatrix} -1 \\ -1 \end{bmatrix}$ and variable C moves via a horizontal channel with a direction $\begin{bmatrix} -1 \\ 0 \end{bmatrix}$.

The third version of the algorithm can be obtained by letting index k determine the valid execution ordering of the computations. However, in this case, the resulting transformed dependency matrix, for the chosen transformation matrix, would require the results of variable A to be stored in the cells. With this arrangement, the array structure would require extra clock cycles to flush out the results of variable A from the cells. Consequently, in order to access the three output results at the same time and vote on them, extra circuitry is required to synchronize the data propagation of the generated output results. Therefore, to avoid this costly procedure, the third version of the algorithm is

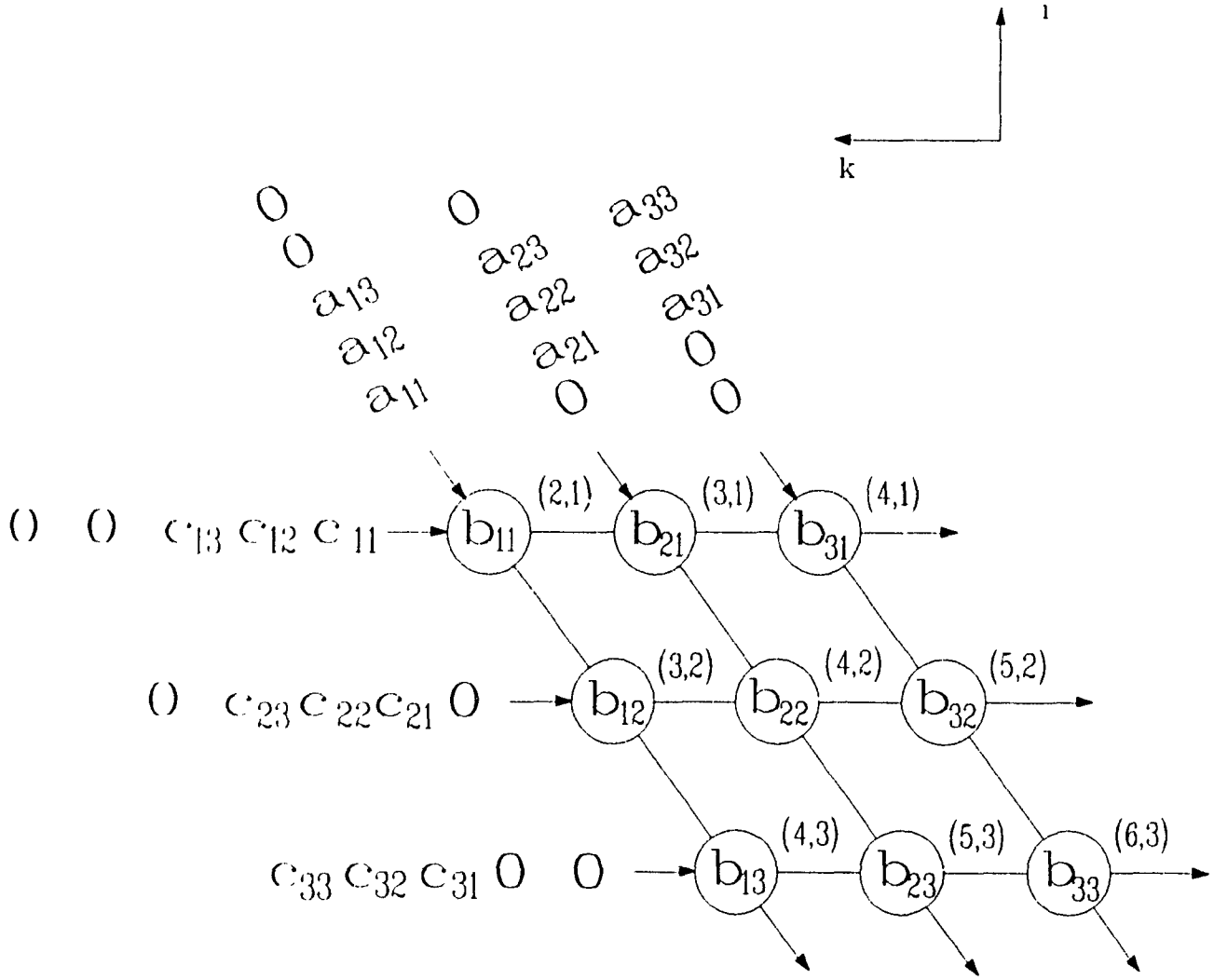


Figure 4.4 VLSI array for index j determining the timing of the computations.

obtained by exchanging the second row with the third row in the matrices of Eqs. (2.9) and (2.10). Thus the transformation matrix T becomes,

$$T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{4.4}$$

and, $$\Delta = \hat{D} = TD = \begin{matrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{matrix} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 0 & 0 \\ -1 & -1 & 0 \end{bmatrix} \tag{4.5}$$

The mapping of the index set can be achieved as follows :

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i + j + k \\ j \\ j + k \end{bmatrix} = \begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} \tag{4.6}$$

The VLSI array structure corresponding to this case is shown in Fig.4.5. The cell structure is also similar to that shown in Fig.4.2, the only difference is the interprocessor communication requirements. The data for variable A travels via a diagonal channel with direction $\begin{bmatrix} -1 \\ -1 \end{bmatrix}$, variable B moves from one cell to the next via a vertical channel with direction $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$ and variable C is stored in the cell itself.

The structures of the systolic arrays shown in Figures 4.1, 4.4 and 4.5 are combined into one architecture and the resulting VLSI array is shown in Fig.4.6 . The results obtained from the three versions are voted on using redundant voters as shown in Fig.4.6. It is important to note that data skewing is required (which is performed by the alignment circuit) in order for the voting procedure to be performed on the corresponding output results produced by the three versions.

4.3.2.3 Method (3) : Combining The Dependency Matrices Of All The Versions Of The Algorithm.

The technique employed in this method is similar (to certain extent) to that described

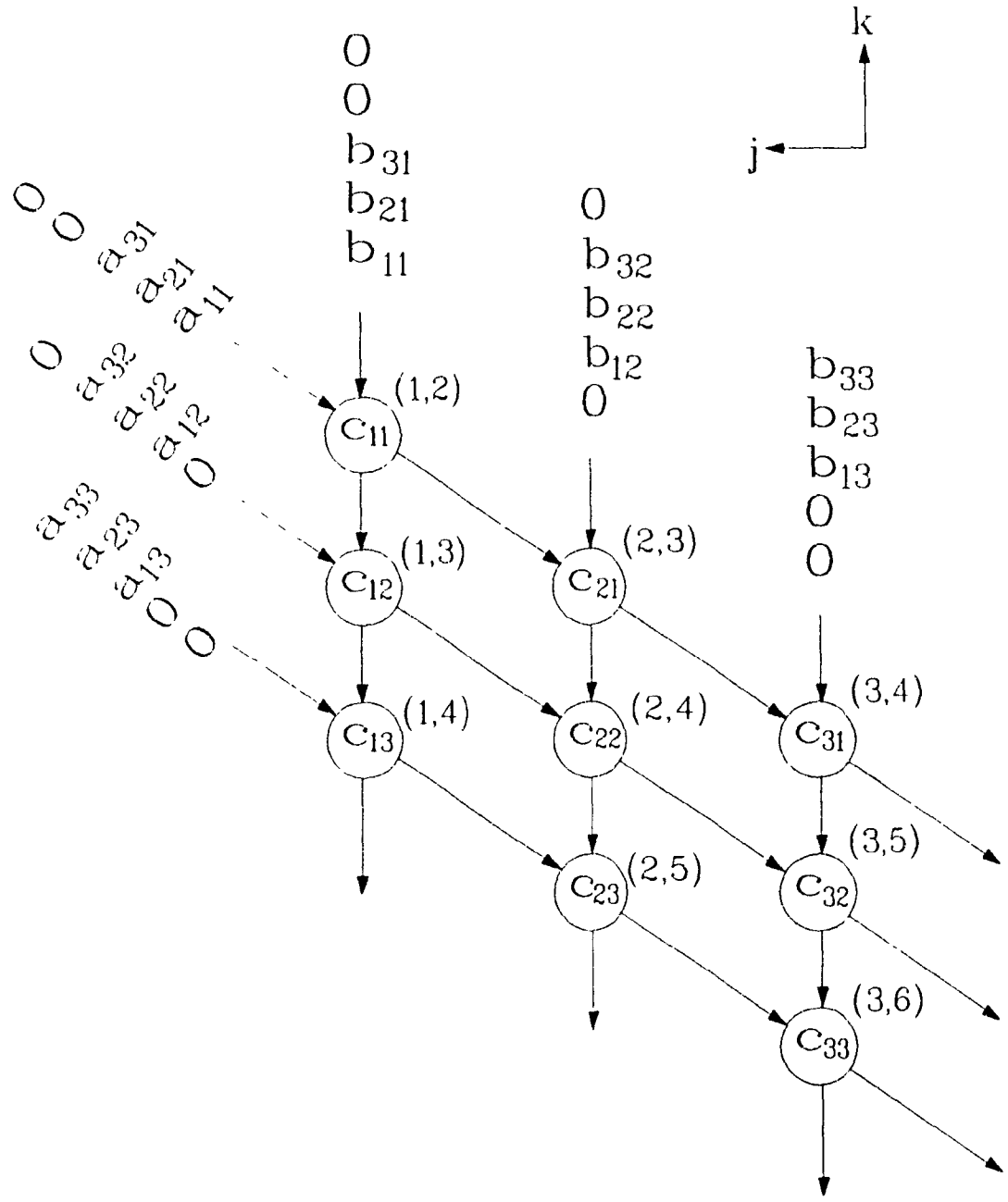


Figure 4.5 Another VLSI structure when index i determines the valid execution ordering of the computations (for N=3).

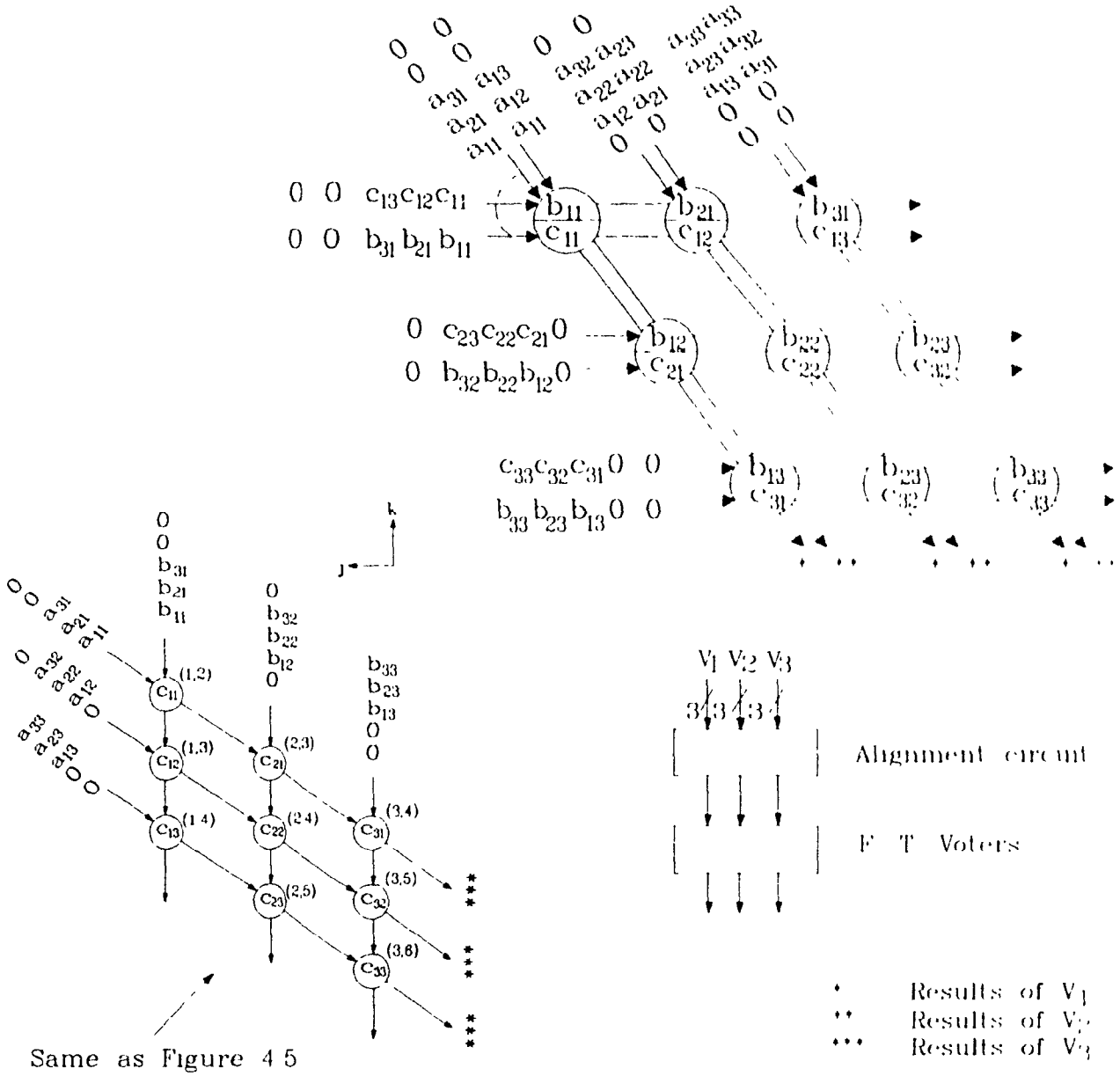


Figure 4.6 VLSI array structure which represents a combination of the VLSI array structures for the different versions of the algorithm.

in section 4.3.2.2. The difference being that, rather than mapping the different versions of the algorithm into separate VLSI architectures (as in method 2), in this method, the dependency matrices of the three versions of the algorithm are combined to give one dependency matrix. This resultant dependency matrix is then mapped into an optimal fault-tolerant systolic array architecture. Similarly, different versions of an algorithm are obtained by letting the different indices (i,j,k) in the algorithm respectively, determine the valid execution ordering of the computation of the algorithm. If we assume that the first coordinate of the dependency matrix always indicates the time at which the computation indexed by (i,j,k) is computed while the other two coordinates indicate the space where the computation is performed. Then the time and spatial components of the dependency matrix when indices i,j and k ,respectively, determine the valid execution ordering of the computations are given as follows:

$$D_i = \begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \quad \begin{matrix} A & B & C \end{matrix} \tag{4.7}$$

$$D_j = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad \begin{matrix} A & B & C \end{matrix} \tag{4.8}$$

$$D_k = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{matrix} A & B & C \end{matrix} \tag{4.9}$$

In Eqs.((4.7)-(4.9)), and as mentioned in section 2.3, the first columns of D_i, D_j and D_k, respectively, consist of the dependency vectors of variable A of the matrix multiplication algorithm, second columns for variable B and the third columns for variable C.

By superimposing the time and spatial components of the dependency matrices in Eqs.((4.7) - (4.9)), we obtain :

$$D_R = \begin{bmatrix} 0 & 0 & -1 & | & 0 & -1 & 0 & | & -1 & 0 & 0 \\ 0 & -1 & 0 & | & -1 & 0 & 0 & | & 0 & 0 & -1 \\ -1 & 0 & 0 & | & 0 & 0 & -1 & | & 0 & -1 & 0 \\ A & B & C & & A & B & C & & A & B & C \end{bmatrix} \tag{4.10}$$

As observed from Eq.(4.10), some of the dependency vectors in matrix D_R are the same. For example, the dependency vector in column one (1) is the same as those in columns six (6) and eight (8). In this case, only one of the dependency vectors needs to be considered. Therefore, by selecting the distinct dependency vectors in matrix D_R, thus Eq.(4.10) reduces to ,

$$D = \begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \\ A & B & C \\ B & C & A \\ C & A & B \end{bmatrix} \tag{4.11}$$

It is important to note that though the modified dependency matrix reduces to or happens to be the same as its original form, in general, this is not the case.

Several valid transformation matrices (TDM's) can be generated for the dependency matrix of Eq.(4.11). The optimization mapping algorithm proposed in section 3.3 is used to select the TDM which has the best overall array performance. From the analysis introduced in chapter III, the transformed dependency matrix with the best overall array performance given the performance index (Δ_6 in table 3.9(b)) is

$$\Lambda = \begin{bmatrix} -1 & -1 & -2 \\ 0 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{4.12}$$

$$\begin{matrix} A & B & C \\ B & C & A \\ C & A & B \end{matrix}$$

The fault-tolerant VLSI systolic array structure corresponding to the transformed dependency matrix Λ of Eq.(4.12) is shown in Fig.4.7(a) . The respective output results of the generated variable (variable A) of the three versions of the algorithm, are computed in the array, according to the data flow requirements of each version. Figures 4.7(b,c and d) show the VLSI array structures that indicate the different data flow computations of the three versions of the algorithm. The structure of the cell in Figs.4.7(a,b,c,d) is as shown in Fig.3.7 (section 3.4). All the cells in the arrays of Figs.4.7(a,b,c,d) are identical. If we assume that each version of the algorithm has separate data paths, then the structure of the cells can be represented as shown in Fig.4.8. Each of the arrays in Figs.4.7(a,b,c,d) contains 15 processors and requires a total of 9 clock cycles to complete the matrix multiplication computation for N=3. In general, to multiply any two N x N matrices, (4N+3) processing elements and (4N-3) clock cycles are required to complete the algorithm computations. The results obtained from the three versions are voted on using redundant voters as shown in Fig.4.7(a). It is important to note that data skewing is required (which is performed by the alignment circuit) in order for the voting procedure to be performed on the corresponding output results produced by the three versions.

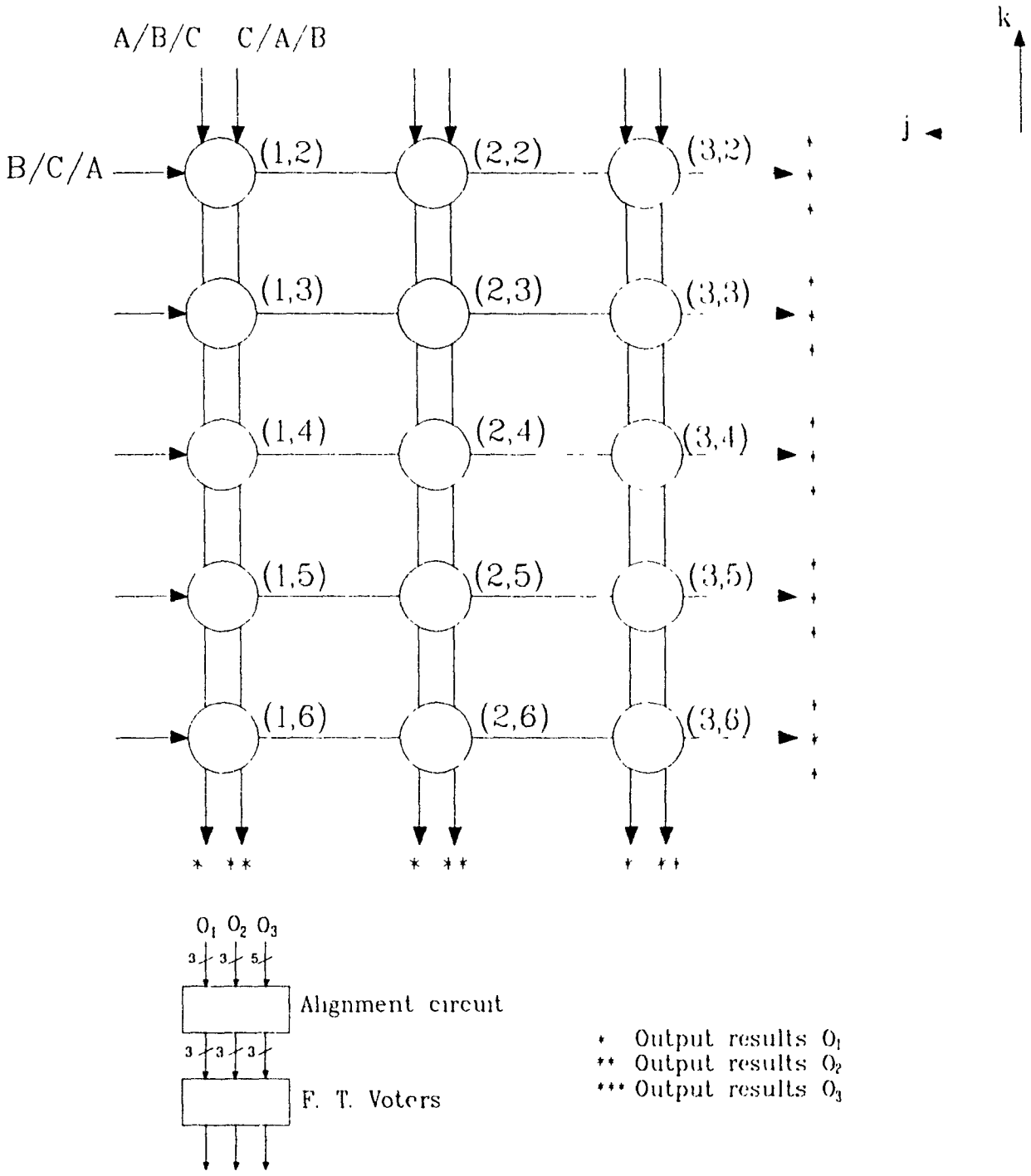


Figure 4.7(a) VLSI array structure when the dependency matrices of all versions of the matrix algorithm are combined (for N 3)

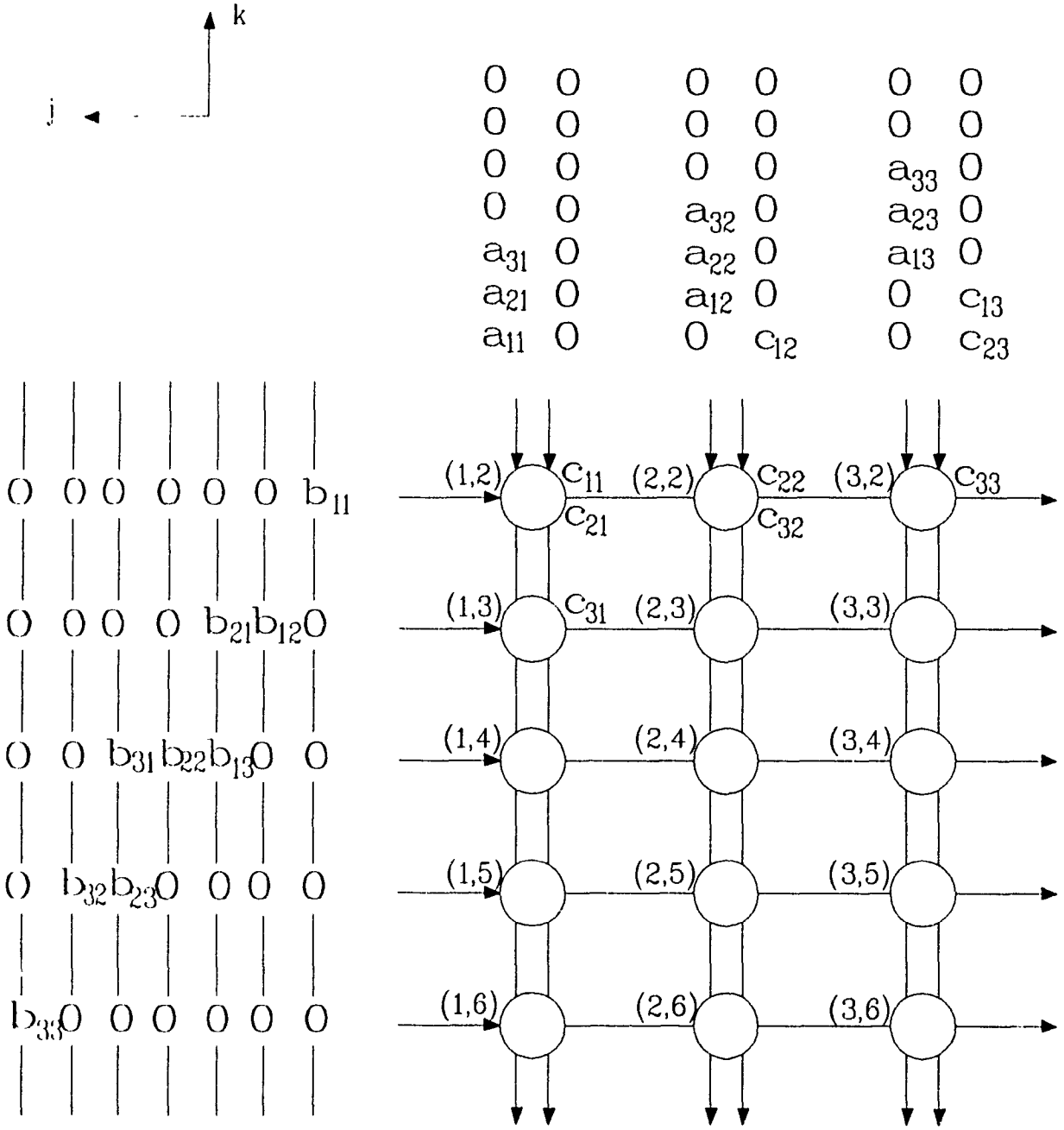


Figure 4.7(b) VLSI array structure indicating the data flow of one version of the algorithm in Figure 4.7(a) (for N=3).

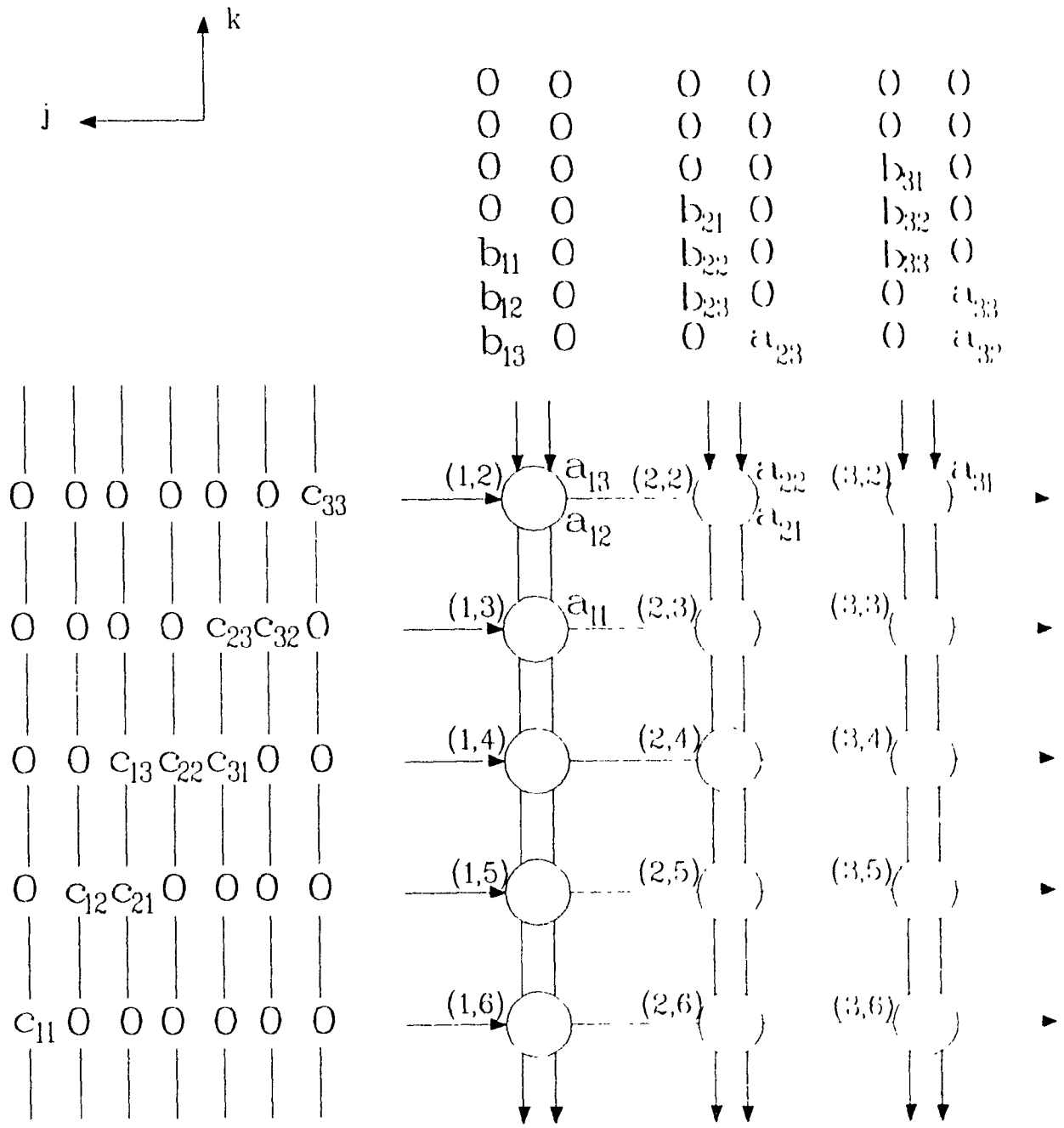


Figure 4.7(c) VLSI array structure showing the data flow of one version of the algorithm in Figure 4.7(a) (for N=3).

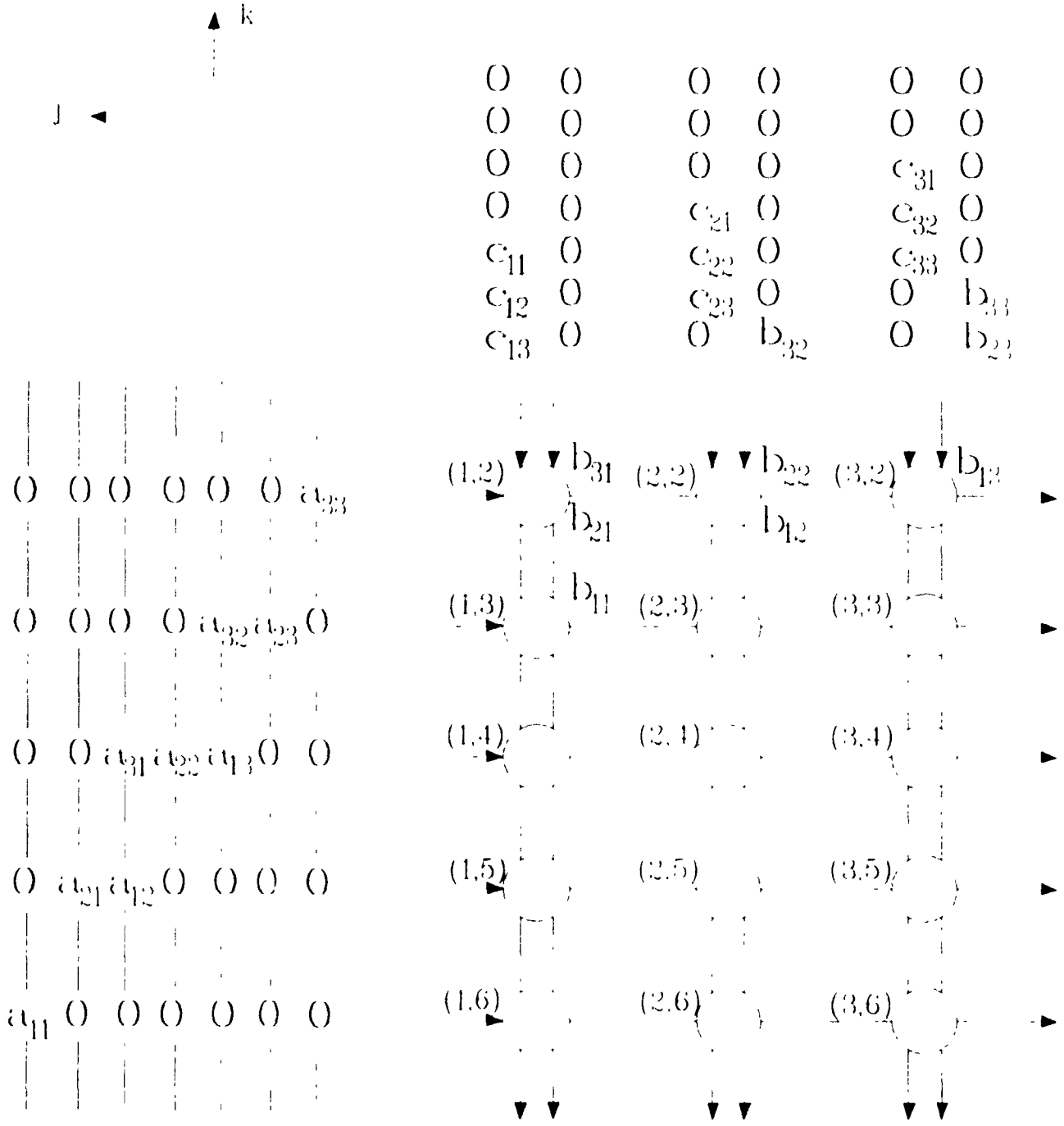


Figure 4.7(d) Data flow for the third version of the algorithm in Figure 4.7(a) (for N=3).

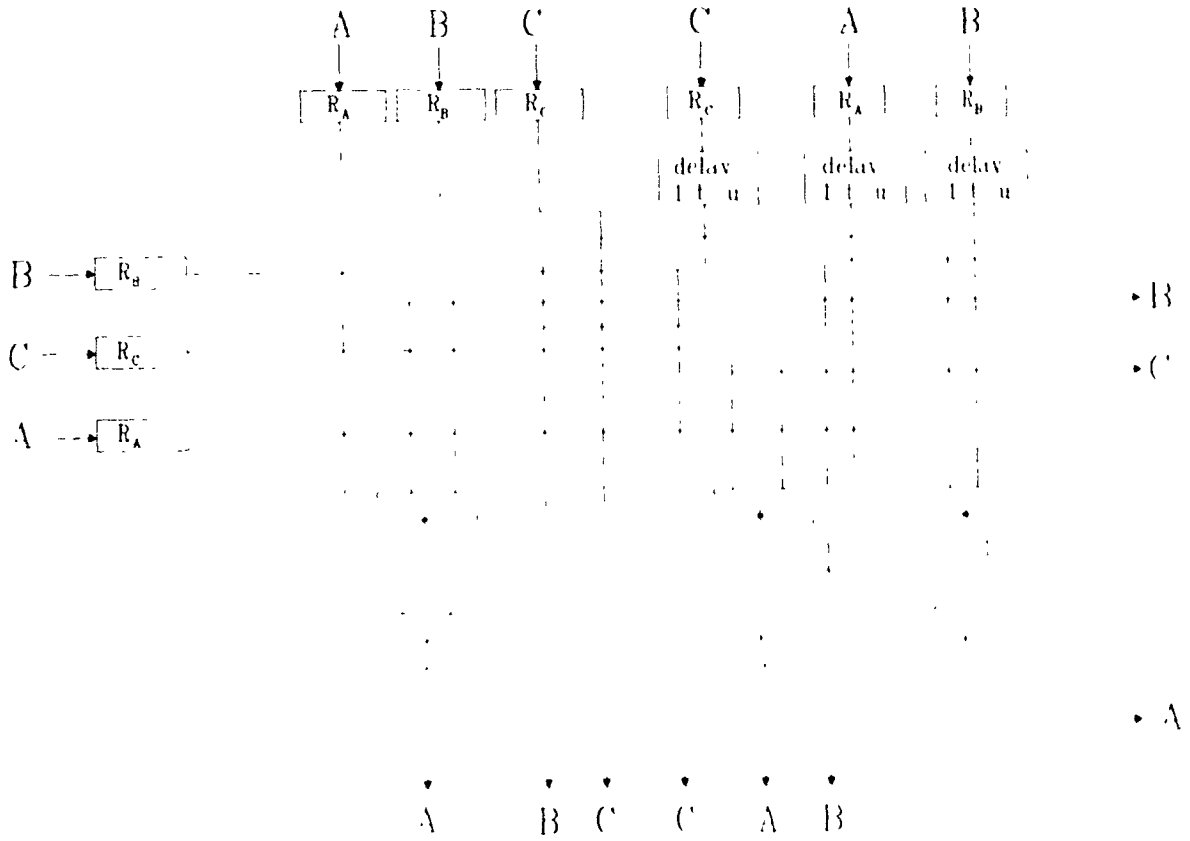


Figure 4.8 The cell structure of the fault tolerant systolic array of Figure 4.7

4.3.2.4 Deriving Three TDM's That Result in Near Local Optimal Systolic Array Architectures.

In the preceding subsection, we have used the TDM (Eq.(4.12)) that gives a local optimal solution to design fault-tolerant systolic array for the matrix multiplication algorithm. In this subsection, we will derive three TDM's that give near optimal solutions, using the same procedure for obtaining optimal systolic arrays discussed in section 3.3. From the analysis in section 3.3, the following three TDM's have been derived as the ones with the next best overall array performance, after the TDM of Eq.(4.12) :

$$\Lambda_u = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad T_u = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{4.13}$$

$$\Lambda_v = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ 0 & -1 & -1 \end{bmatrix} \quad \text{and} \quad T_v = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{4.14}$$

$$\Lambda_w = \begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & -1 \\ -1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad T_w = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{4.15}$$

The three TDM's have the same timing structure but their interprocessor communication requirements are different. Figure 4.9 depicts the VLSI array structure for Δ_u (Eq.(4.13)). The structure of the cell in Fig.4.9 is similar to that shown in Fig.4.2, the only difference is that variable C is stored in the cell of Fig.4.2, while in the cell of Fig.4.9, variable C travels via a vertical channel with direction $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$. All the cells in Fig.4.9 are also identical. They do not contain any delay elements like the cells in Fig.4.8. The array consists of 19 processors and utilizes a total of 7 clock cycles to complete the computation of the algorithm (for N=3). In general, in order to multiply two NxN matrices using this systolic structure, (6N+1) processors and (3N-2) clock cycles are required to complete the computation of the matrix multiplication algorithm.

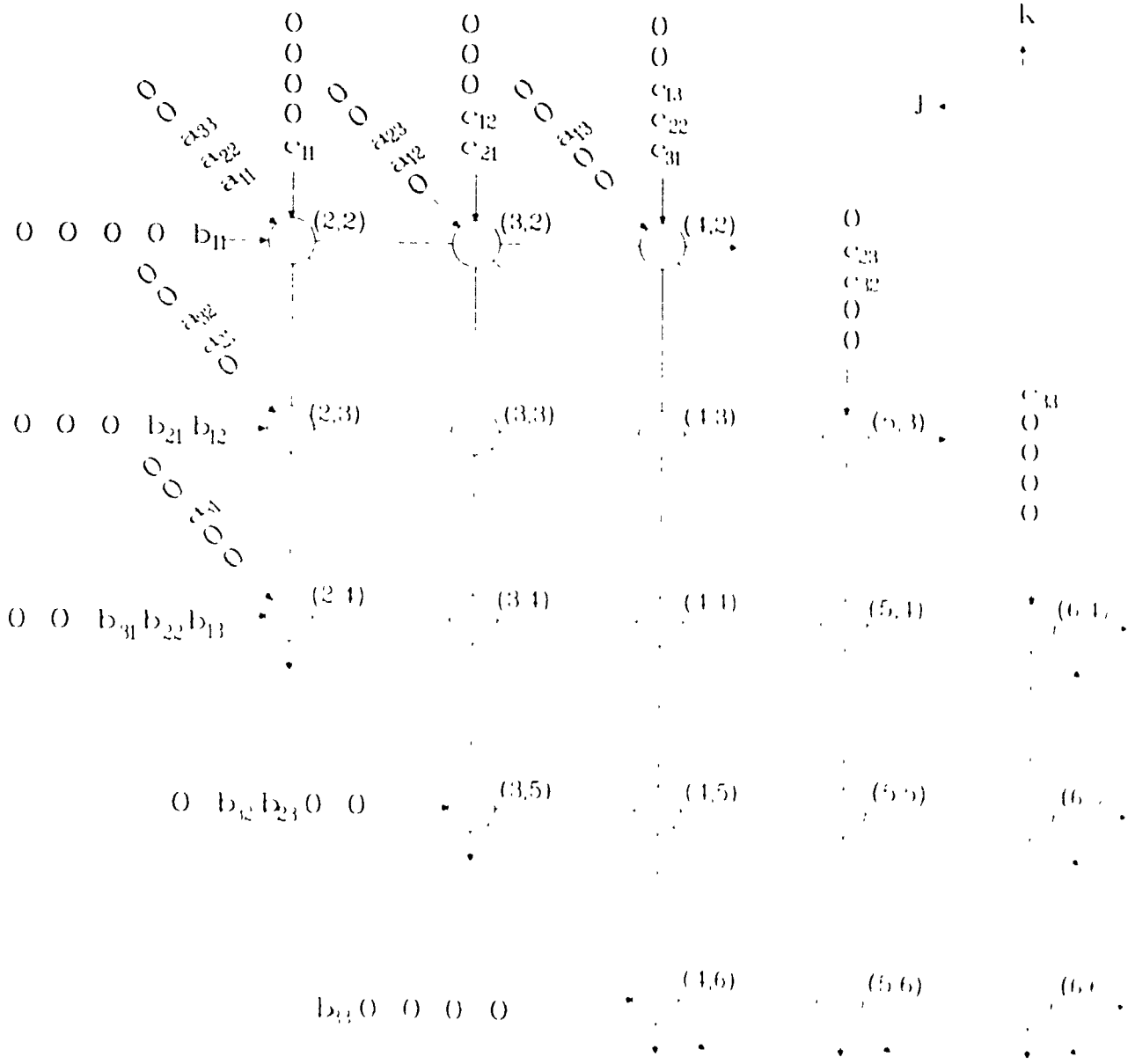


Figure 4.9 VLSI array structure for Δ_{11} (for N=3)

Since the three TDM's have same timing and similar spatial structures, hence, the VLSI systolic array structures for Δ_v and Δ_w will be similar to that shown in Fig.4.9, with the difference only in the direction of data flow for the different variables in the algorithm. If we combine the VLSI systolic arrays of Δ_u, Δ_v and Δ_w, the resultant VLSI array shown in Fig.4.10, has a similar structure as the array shown in Fig.4.9. Also, in this case, the output results of the different versions of the algorithm can be calculated using separate data paths and computation units. The results obtained from the three versions would be voted on using redundant voters in a similar fashion as shown in Fig.4.7(a). It should be noted also, that data skewing is required in order for the voting procedure to be performed on the corresponding output results produced by the three versions.

The fault-tolerant design method described in section 4.3.2.3 is equivalent to that described in section 4.3.2.4. By combining the dependency matrices of all the versions into one dependency matrix and taking the transformation, results to one array as in section 4.3.2.3. On the other hand, in section 4.3.2.4, by composing the transformation that yields one version of the algorithm from the original algorithm, with T in Eq.(4.13), results to the transformation that yields one array of the original algorithm. Doing this for the three versions results in three transformations and hence three systolic arrays. These arrays can be combined into one array as in section 4.3.2.4.

It is worthwhile to mention the distinction between the fault-tolerant systolic arrays designed using methods 2 and 3. As seen from Fig.4.6 (using the FT design method 2), one version of the three versions of the algorithm is mapped onto a different VLSI space while the other two versions are mapped onto the same space. On the other hand, as shown in Figs. 4.7(a) and 4.10, the three versions of the algorithm are mapped into the same VLSI space in each case, since the FT design methods are equivalent.

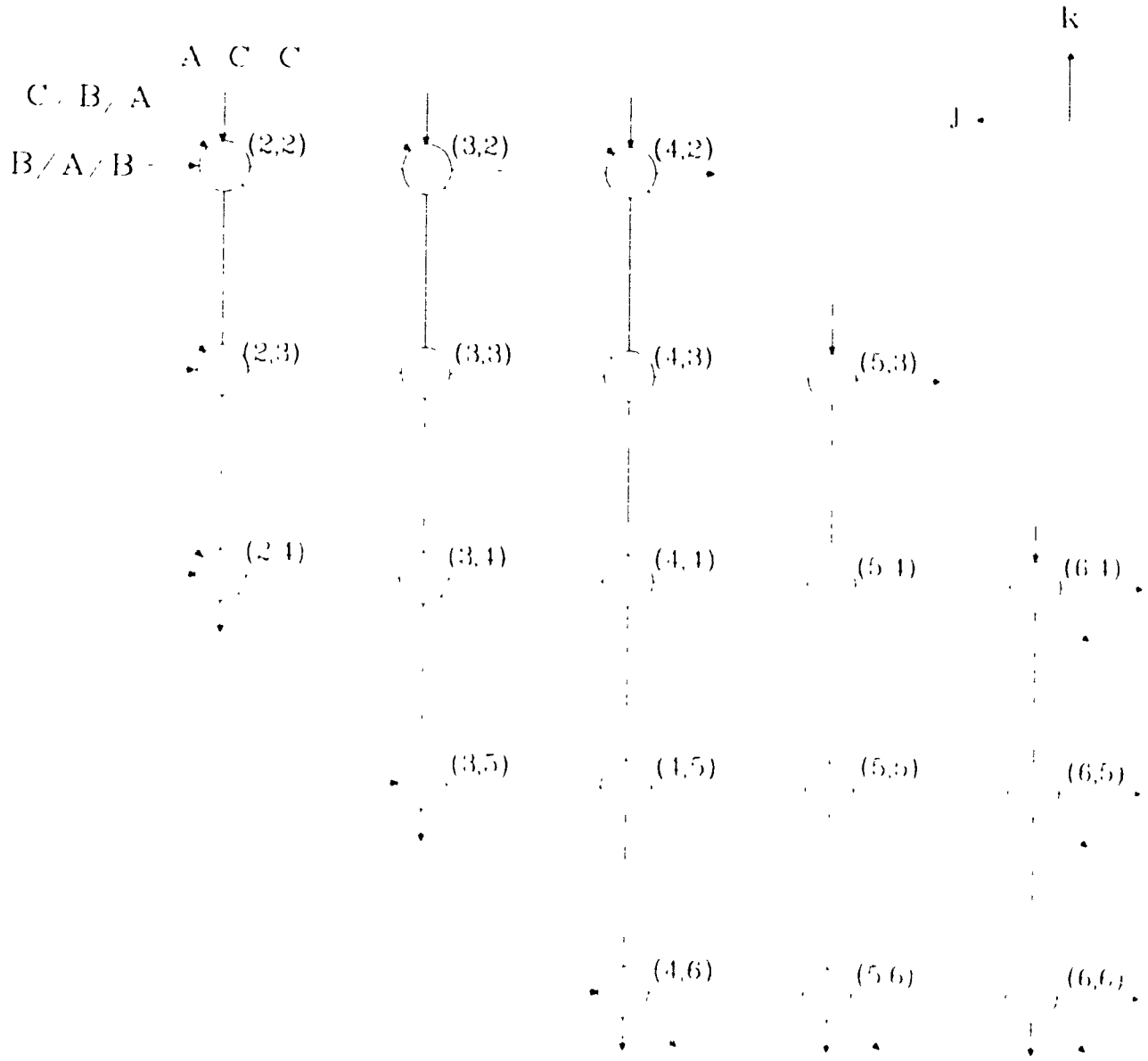


Figure 4.10 VLSI array structure which represent a combination of the VLSI array structures for Δ_U, Δ_V, and Δ_W

The systolic arrays in Figs.4.7(a) and 4.10 are more regular and granular than the array in Fig.4.6. Less silicon area will be required for routing data into the fault-tolerant VLSI systolic architectures of Figs.4.7(a) and 4.10 than into the array of Fig.4.6. Consequently, the scheme proposed in method 3 is preferred to that in method 2.

In the following section, we will analyze the proposed fault-tolerant systolic arrays.

4.4 ANALYSIS OF THE PROPOSED FAULT-TOLERANT MAPPING SCHEMES

Two redundancy approaches can be used to tolerate single faults in the arrays of Figs.4.6, 4.7(a) and 4.10. These are the time and hardware redundancy techniques. By employing the time redundancy approach, in Fig.4.6, the results of the two different versions of the algorithm that are mapped onto the same VLSI space, are computed at different times using the same building blocks in the processing elements. Then, the other version of the algorithm can be computed at the same time as any of the other two versions. Similarly, in Figs. 4.7(a) and 4.10, the results of each version of the algorithm can be computed at different times using the same building blocks. The key difference between our technique and the traditional fault-tolerant techniques that use time redundancy [8] is that, in the latter case, the data set for one version of the algorithm is replicated and used to perform the matrix multiplication computation three times. In other words, the same data set, having the same data flow characteristics, are applied to the systolic array to perform the computations. However, for the scheme proposed here, the data set for each version is of course the same, but they possess different data flow characteristics, as shown in Figs.4.7(b,c,d).

Our approach provides an alternative to the traditional fault-tolerance techniques using time redundancy. The motivation for using three versions of the algorithm, rather than merely replicating one version of the algorithm, is based on the possibility that the

latter case might mask some temporary faults if the same fault appears in each of the replicated data set. Since the data flow characteristics for the replicated version of the algorithm is the same, hence, they exhibit the same dynamic properties. The effects of any temporary faults on one copy of the results produced with one of the replicated data set, will also be the same on the other copies. Therefore, such faults can neither be detected nor tolerated using the traditional approach.

In the proposed scheme, the three versions of the algorithm have different data flow characteristics, and hence, their respective dynamic properties are different. The three versions will not be affected the same way by temporary faults [40,41]. Thus, such faults, if they occur will be detected and masked by the proposed scheme.

For instance, let's consider the array structure of Fig.4.7(b) for the computation of the matrix multiplication algorithm. In the traditional case, the data for this version is triplicated. In our approach, three versions of the algorithm are used, and we assume that their data flow characteristics are shown in Figs.4.7(b,c,d). The array structures in Figs.4.7(b,c,d) are the same, however, the flow of data into the FT array is different. All single temporary faults in the fault model, that affect only one version of the algorithm, will be masked by both approaches. For example, if a temporary fault occurs in cell (1,4), then the output results of elements a_{11}, a_{21} and a_{31} will be unreliable, if the traditional approach is used. In our case, either the output results of (a_{11}, a_{21}, a_{31}) or (a_{11}, a_{12}, a_{13}) or (a_{13}, a_{22}, a_{31}) will be unreliable. Since three copies of the output results are produced using both schemes, the majority of the results will be reliable

Furthermore, let's assume for example, that a single transient fault (such as a voltage spike) occurs in the path that is traversed by the elements of a_{12}, a_{22} and a_{33} in Fig.4.7(b). Also, if we assume that this fault affects only the elements of a_{ij} and not those of b_{ij} and c_{ij}. Since in the traditional approach, the same data set is replicated, this fault will affect all three copies of a_{ij}. The effects of this fault will not be masked and will go undetected by the output voter. Such faults cannot be tolerated. This example

clearly illustrates the strength of our approach of designing FT systolic arrays. Since the data flow characteristics of all the versions are different, only one copy of the output results will be affected by the fault. The majority of the results will be reliable. Thus, such faults can be detected and tolerated by our design scheme.

In addition, both approaches can tolerate some multiple fault patterns that consist of single temporary faults in every version. For instance, if a single temporary fault occurs in cells (1,4), (2,3) and (3,2) for versions one, two and three, respectively. For the traditional approach, only one copy of the results of the elements a_{11}, a_{21}, a_{31}, a_{12}, a_{22}, a_{32}, a_{13}, a_{23} and a_{33} , will be erroneous. Hence, this multiple fault pattern will be tolerated. On the other hand, employing the proposed scheme, the output results of the following elements will be affected : a_{11}, a_{21}, a_{31}, a_{21}, a_{22}, a_{23} and a_{33}. Since two copies of the results of a_{21} are affected by this multiple version fault pattern, the effects of the faults cannot be masked by the proposed scheme.

However, consider the multiple version fault pattern which occurs in cell (1,2) for two versions. This fault pattern cannot be tolerated by the traditional approach, since two copies of the results of a_{11},a_{21} and a_{31} will be erroneous. If we consider the two versions in Figs.4.7(b,d), only one copy of the results of the elements a_{11},a_{21},a_{31} and a_{33} will be erroneous. Thus, this multiple version fault pattern can be tolerated by our design scheme.

Both the proposed and traditional time redundancy schemes, utilize the same area and time to compute the matrix multiplication algorithm. There is no area overhead since no hardware is introduced in the FT array. It is important to mention that in our analysis throughout this thesis, the area of the voting circuits is not taken into account for area overhead measurement. This is because, other FT design schemes where voters are employed do not take such circuits into consideration in determining the additional hardware redundancy required for fault-tolerance. Only the number of processors are cosidered. In Fig.4.7(a), the multiplication of two NxN matrices requires a time

redundancy of 2N. The time redundancy ratio is $O(2N / (4N-3))$. The *redundancy ratio* is defined as the ratio of the hardware or time overhead required by the fault-tolerant technique to the hardware or time complexity of the original system without fault-tolerance [29-32]. Thus, when N is large, the hardware redundancy ratio of the two design schemes is 0% while their time redundancy ratio is less than 50%.

One of the problems of time redundancy techniques is that only transient faults can be tolerated. Permanent faults will not be tolerated if they occur. Also, there is a degradation of time and hence the technique might not be suitable for real time application. Furthermore, if the output results of the three versions of the algorithm have to be produced at different times, then the corresponding results from the different versions need to be synchronized before a majority decision can be made on them. This type of synchronization of the data can only be accomplished using complex control circuitry or synchronization mechanism.

Considering the amount of time that will be required to compute the results of three versions of the algorithm, the complex control circuitry for data synchronization and the fact that only temporary faults will be covered using the time redundancy technique, spatial redundancy technique can be adopted to achieve fault-tolerance in the systolic array architectures of Figs. 4.6, 4.7(a) and 4.10. In Fig.4.6, the registers, the computational units and the links of the VLSI array for computing the results of the two versions of the algorithm are duplicated in each cell so as to satisfy the data flow requirements of each version of the algorithm. In Figs. 4.7(a) and 4.10, the registers, the computational units and the links in each cell of the VLSI array for computing the results of the three versions of the algorithm are triplicated such that each version of the algorithm is computed on a different set of modules. The structure of the cell in Fig.4.7(a) is shown in Fig.4.8. It depicts how the different data flow computations are performed in the cells. Three copies of the output results are respectively produced by the VLSI structures of Figs.4.6, 4.7(a) and 4.10, and then a majority decision is made on the three output results in each case.

The scheme used here is the same as the TMR scheme since the number of the building blocks in Figs.4.6, 4.7(a) and 4.10 is three times the number of the building blocks in the irredundant array structure (Fig.4.1). The principle distinctions between the traditional TMR approach and our approach are, in the former case, the three copies of identical results are completed and they appear at the output of the array at the same time. In the proposed scheme, the three copies of the identical results do not appear at the output at the same time. For instance, the results of a_{11} in version one (Fig.4.7(b)) appears at the output of the array after 5 clock cycles. That of the second version (Fig.4.7(c)) appears after 7 clock cycles, while the third copy (Fig.4.7(d)) appears after 9 clock cycles. Hence, the time to produce the identical sequences of the output results is not constant. However, both schemes use the same amount of time to complete the computation of the algorithm.

Since identical output results are produced at the same time, in the traditional TMR approach, therefore, there is the option of voting on the partial results produced by the individual cells or voting on the final identical results produced at the output of the FT array. Thus, both local and global voting on the identical results are possible. Our scheme requires that voting be performed on identical output results of the different versions of the algorithm. Therefore, only global voting on the final results produced by the different versions, is possible. The advantage of local voting is that, it increases the possibility of tolerating multiple cell failures in the systolic array. However, this will be achieved at the expense of increasing the complexity of the processing elements, since each cell will be incorporated with a voting circuit.

Furthermore, in the traditional TMR, the same data paths can be shared between cells at the boundary of the FT systolic array. On the other hand, in the proposed scheme, since the data flow of each version of the algorithm is different, this limits the sharing of data paths between the cells at the boundary of the array. However, if we assume that an efficient data sequencer (data feeding mechanism) can be used to put the input data in

order, then the different versions may share the same data path at the boundary of the array. The input data sequencing could be sequential or parallel. In this case therefore, although area is saved when two or three cells are mapped onto the same VLSI space (saving on the routing of data bus), however, this might also have the disadvantage that a physical fault may affect the versions that share the data bus.

The proposed TMR scheme can tolerate all single transient and permanent faults in the fault-tolerant systolic array. Three sets of the computed results are produced using different data paths in the arrays of Figs. 4.6, 4.7(a) and 4.10. A majority of the computed results is obtained using a fault-tolerant voter [39]. The voter masks the effects of all single faults in the fault-tolerant systolic arrays. In order to ensure the reliability of the voted results, it is important, however, that such a majority circuit be protected against faulty components within itself. Also, since the data flow characteristics and hence the dynamic properties of the different versions are not identical, their corresponding output results will not be affected the same way by temporary faults [40,41]. Therefore, in addition to providing tolerance against hardware faults, the proposed scheme has the ability to tolerate certain categories of temporary faults. In the traditional TMR, since replicated copies of one version of the design are used for the FT array, it does not have the ability to tolerate all the single transient fault in the FT systolic array.

Although, we have considered the results of single faults, however, a number of multiple fault patterns, in the fault-tolerant systolic array, can also be tolerated by our design scheme. For instance, consider the FT array of Fig.4.7(a). A multiple fault pattern that includes a failure of one R_A,R_B,R_C, delay unit, adder and multiplier, in any cell in the array, will be tolerated. This multiple fault pattern will cause the output results of only one version of the algorithm to be erroneous. Hence, the majority of the output results will be fault-free. Also, a combination of single module failure in each cell of the array of Fig.4.7(a), will have the same effect as the above mentioned multiple fault pattern. As a result, this fault pattern can also be tolerated. It is important to mention that,

those multiple fault patterns that affect two or more versions of the algorithm can not be tolerated by our FT design scheme.

The scheme requires no time overhead since the three copies of the output results are computed at the same time. However, it requires additional hardware redundancy of, at least, a factor of 2, to tolerate single module failures.

In the following subsection, we discuss how our design schemes compare with other fault-tolerant design techniques.

4.4.1 Comparison Of The Proposed Design Schemes With The Other Fault-Tolerant Schemes.

In this section, we compare our design scheme with other fault-tolerant design schemes described in the literature. Table 4.1 contains the comparison of the complexity and performance of the proposed design scheme and those presented in [8,9,12,29-32,34,35]. The irredundant systolic array structure for matrix multiplication (Fig.4.7(a)) consists of (4N + 3) processors. The total execution time required by the array is (4N - 3). For example, if N=3, the array consists of 15 PE's and will require a total of 9 clock cycles to complete the computation. In our FT design approach, we have adopted the spatial and temporal redundancy techniques to achieve fault-tolerance in the systolic array. In the first technique, the building blocks or modules in each cell are triplicated. Therefore, the hardware overhead is a factor of 2 (200%). There is no time overhead (0%), since all the output results of the three versions of the algorithm are computed at the same time. The data flow characteristics of the three versions are not identical. Hence, the corresponding partial results of the elements of the variables in the algorithm, are computed using the same space, but in different processors and at different times.

The proposed TMR design scheme can tolerate all single transient and permanent faults in the FT systolic array. Also, certain multiple fault patterns can be tolerated. For the proposed temporal redundancy technique, the additional time required is O(2N).

Category	Von-Neuman [8]	Huang & Abraham [29-31]	Jou and Abraham [32]	Kim & Reddy [9]
FT Technique	Triple Modular Redundancy (TMR)	Checksum Encoding Scheme	Weighted Checksum Code (WCC)	TMR Approach
Complexity without FT techniques	Processor: X Time: T	- Dense matrix multiplication Processor: N*N Time: N - Band matrix multiplication Processor: W_1*W_2 Time: $N+\min(W_1,W_2)$	Processor: N^2 Time: N	Processor: W_1*W_2 Time: N
Redundancy required with the technique	Processor/interconnections: 2X Time: 0 Voter: $O(N)$ Where X is the number of processing elements in the irredundant array. T is the total number of clock cycles.	Dense matrix multiplication Processor/interconn.: 2N+1 Time: $(2*r*\log_2(N)) + T_{cor}$ - Band matrix multiplication Processor: W_1+W_2 Time: $(2*r*\log_2(N)) + T_{cor}$ Adder: $3(W_1+W_2)-4$ Buffer: $s\log_2(W_1) + \log_2(W_2) + 2\min(W_1,W_2) + 2$ TSC comparator: 2	Processor: $N((2N/l)+2)$ Time: $(N/l)+T_{cor}$ Adders: $O(2N)$	Processor: $N(3m)-(W_1*W_2)$ if $W_1=2m-1$ $N(3m+1)-(W_1*W_2)$ if $W_1=2m$ Time: N MUX: 3 two-port MUX per PE Voter: 1 per PE Buffer: 2 per PE Extra interconnections between PE's
Redundancy Ratio	Processor/interconnections: 2 Time: 0	- Dense matrix multiplication Processor: 2/N Time: $((2*r*\log_2(N))+T_{cor})/N$ - Band matrix multiplication Processor: $O(2/W_1)$ Time: $((2*r*\log_2(N))+T_{cor})/N$ r is the ratio of execution time of an addition to that of a multiplication performed in a processor array. T_{cor} is the time required to correct the errors. Assume that $O(W_1)=O(W_2)$, W_1 and W_2 are the width of the band matrices.	Processor: $O(((2N/l)+2)/N)$ Time: $O((\lceil N/l \rceil+T_{cor})/N)$ - l is the word length of the weighted summation elements. - T_{cor} is the time required to correct the errors.	Processor: $[N(3m)-(W_1*W_2)]/(W_1*W_2)$ if $W_1=2m-1$ $[N(3m+1)-(W_1*W_2)]/(W_1*W_2)$ if $W_1=2m$ Time: $O(1)$ m is any integer.
Diagnosis Performance	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - detection: No - location: No	Faults - types: Transient - # allowed: Single fault & certain multiple fault patterns - detection: Yes - location: Yes	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - detection: Necessary - location: No	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - detection: No - location: No
Utilization	On-line local or global error masking.	Off-line error correction.	Off-line error correction.	On-line local error correction.

Table 4.1 The comparison of the complexity and diagnosis performance of various existing fault-tolerant techniques with our proposed design scheme.

.../3

Category	Cosentino [12]	Cosentino [34]	Ari & Friedlander [35]	Varman & Ramakrishnan [21a]
FT Technique	Dual Redundancy (Duplication)	Residue Number System (RNS) Approach	Algorithm-based Error Detection Approach	Reconfiguration Approach
Complexity without FT techniques	Processor: N^2 Time: 3N-1	Processor: $3N^2$ Time: $2N_2$ Delay Units: $2N^2$-N Other: Binary-to-Residue converter Residue-to-Binary converter	Processor: $(2N+1)^2$ Time: 4N+2	Processor: N√N Time: 4N√N - N - 3√N Total Buses: 4√N Delay Units: 2(4√N - 2) per PE MUX: 2 per PE
Redundancy required with the technique	Processor: N (host processor) Time: 2N Accumulator register: N^2+N-1 Monitoring circuit: N Correction circuit: N Memory: Yes Host processor: Yes	Processor: $2N^2$ Time: T_{r-b-c} Tsc: 2N Other: N error calculator (correction circuit) Where T_{r-b-c} is the time for the residue-to-binary conversion.	Processor: P Time: T_p P and T_p are respectively the additional hardware and time to implement the diagnostic expression.	Processor: K Time: 2K Delay Units: 4 per PE Where K is the number of faulty processors in the wafer or array.
Redundancy Ratio	Processor: $O(1/N)$ Time: $O(2N/(3N-1))$	Processor: $2N^2/3N^2$ Time: $(T_{r-b-c})/2N$	Processor: $P/(2N+1)^2$ Time: $T_p/(4N+2)$	Processor: K/(N√N) Time: 2K/(4N√N - N - 3√N)
Diagnosis Performance	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - detection: Yes - location: Possible	Faults - types: Transient & Permanent - # allowed: Single fault & certain multiple fault patterns - detection: Necessary - location: No	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - detection: Required - location: Necessary	Faults - types: Permanent - # allowed: Single & multiple PE failures - detection: Required - location: Necessary
Utilization	On-line global error correction.	On-line global error correction.	Off-line error correction.	Off-line fault-tolerance.

Table 4.1 continues.

...2/3

Category	P. Kumar & Y-Chen Tsai [22]	Proposed Approach (Method 1)	Proposed Approach (Method 2)
FT Technique	Reconfiguration Approach	Time Redundancy Approach	TMR Approach
Complexity without FT techniques	Processor: $N\sqrt{N}$ Time: $3N\sqrt{N} - 2$ Total # Buses: $2\sqrt{N} + 2$ Delay Units: $N\sqrt{N}$ per PE MUX: 2 per PE	Processor: $4N+3$ Time: $4N-3$	Processor: $4N+3$ Time: $4N-3$
Redundancy required with the technique	Processor: K Time: 2K Delay Units: 4 per PE	Processor: 0 Time: $O(2N)$ Voter: N	Processor: $2(4N+3)$ Time: 0 Voter: N
Redundancy Ratio	Processor: $K/N\sqrt{N}$ Time: $2K/(3N\sqrt{N} - 2)$ Where K is the number of faulty processors in the wafer or array.	Processor: 0 Time: $O(2N/(4N-3))$	Processor: $O(2(4N+3)/(4N+3))$ Time: 0
Diagnosis Performance	Faults - types: Permanent - # allowed: Single & multiple PE failures. - detection: Required - location: Necessary	Faults - types: Transient - # allowed: Single fault & some multiple faults in the computation unit and data paths. - detection: Not required - location: No	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - detection: Not required - location: No
Utilization	Off-line fault-tolerance.	On-line global error masking.	On-line global error masking.

Table 4.1 continues.

...3/3

If N=3, for example, the time redundancy ratio is 66%, and for a large value of N, this ratio approaches 50%. There is no area overhead (0%) since the same array is used to compute the results of the three versions of the algorithm. The different data flow computations are performed on the same irredundant systolic array, but at different times. All single transient faults in the systolic array can be tolerated. In many other FT design schemes, using time redundancy approach [8,9], one copy of the design implementation of a given algorithm can be used to perform the matrix multiplication computation several times. Although, some transient faults will be tolerated, not all the categories of transient faults can be captured, since recomputation is done using the same operands with the same data flow characteristics. Therefore, our design schemes have better fault coverage than many other brute force TMR design schemes.

The technique described by Von-Neuman [8], also uses the TMR approach to achieve fault-tolerance. The hardware redundancy ratio is 200% and there is no time overhead. Due to the fact that the data flow characteristics of the three copies of the design are the same, there is the possibility that the effects of any temporary fault in one copy of the results, will be manifested in the other copies as well. Therefore, such faults if they occur, will not be detected by the voter and hence, can not be tolerated. Thus, although this technique can tolerate all permanent faults that affect only one PE in the array, it cannot tolerate some categories of transient faults.

Huang and Abraham [29-31] proposed a matrix encoding scheme (checksum technique) to detect and correct errors in matrix operations performed by processor arrays. Their strategy is to encode the input matrices and to check the correctness of the encoded result matrices. Their technique requires a hardware redundancy ratio of O(2/N) and a time redundancy ratio of $O(\log_2(N) / N)$ to complete the computation of the output results. Here, N is the dimension of a square matrix or the width of a band matrix. Applying the technique to a processor array system with N=100, for example, the hardware redundancy ratio is 2% and the time redundancy ratio is less than 4%. The

approach can tolerate transient and permanent faults in the computation unit of the processors. It has the capability of fault location.

The drawbacks of the technique include, it cannot tolerate faults in the communication lines and registers. More time is required to correct the errors after they are located. The error correction is done off-line. However, because of the predominance of the transient faults in VLSI, the off-line testing and reconfiguration method becomes less useful. Techniques are needed to tolerate faults concurrently with normal operation [9]. Transient errors are becoming more frequent due to low supply voltages and decreased signal-to-noise ratios on VLSI chips [42]. The scheme can only correct errors in matrix multiplication. It can detect, but cannot correct, errors in many other signal processing problems, such as matrix-vector multiplication, matrix inversion and so on. Furthermore, the scheme requires only two-dimensional arrays for fault tolerance, and the types of systolic architectures required to perform the matrix operation are fixed. The hexagonally connected processor arrays are used for band matrix multiplication, and the mesh - connected processor arrays are used for the multiplication of two dense matrices. The structure resulting from the approach is no longer highly regular and granular. The nonregularity even depends on the array size. As a result of this, more silicon area is required to route data into the FT systolic array. This could cause the hardware redundancy to rise to more than 100% instead of 2%. Also, the diagnosis and correction latency is very long and the scheme is vulnerable to false alarms brought on by roundoff errors.

Our method does not involve any fault location, errors are masked concurrently with normal operation, hence, on-line error correction. It can be applied to any two-dimensional array. Redundant computations are systematically introduced at the algorithmic level to satisfy a given fault-tolerant requirement. Since many transformation matrices can be generated for any given data dependency matrix, hence, the fault-tolerant dependency matrix of the algorithm can be systematically mapped into several fault-tolerant systolic array architectures with different characteristics.

In order to solve the problem of the inability of the scheme in [29-31] to correct errors in many signal processing problems, Jou and Abraham [32] proposed a new encoding technique, called the *Weighted Checksum Code* (WCC). The checksum code is a subset of this new code. The code can be used wi linear arrays. Like the scheme proposed in [29-31], this scheme involves the encoding of the data used by the algorithm, redesigning of the algorithm to operate the encoded output data.

Some of the problems of this technique are: It is not easy to implement. When the weights are chosen, the complexity of the implementation should be considered. The user needs to know the weights that are easy and less complex to implement. Furthermore, choosing the checksum matrix is a problem. It must fit within a given processor array. There is also the problem of the word length (**l**) of the weighted summation elements. In order for the hardware, which is used to compute the weighted checksums, to be minimal, the size of the linear array should not be large compared to the word length. Therefore, the scheme is limited by the size of the linear array. The hardware redundancy ratio for the matrix multiplication algorithm is O((2N/**l** + 2)/N). For N much larger than **l**, the hardware overhead increases. As seen from table 4.1, the time redundancy ratio approaches 100% if N=**l**, and more than 100% if N is too large compared to **l**.

The scheme requires less hardware redundancy but more time redundancy than our design scheme. However, it can only tolerate faults in the computing units of the processor in the linear array. It cannot tolerate faults in the registers and communication lines for data transfer. The error correction of the scheme is not done with the normal operation of the system (off-line error correction). Our technique is simple and easy to implement. Faults are tolerated concurrently with the normal operation.

Kim and Reddy [9] proposed a TMR approach whereby three adjacent cells in a linear bidirectional systolic array, are used to produce three copies of the computed results. The scheme requires increasing the complexity of the cells such that, in addition to the computation unit, each cell consists of three two-port multiplexers, a voter and two

latches. Applying their approach to solve a problem of multiplying two matrices, requires a hardware redundancy ratio of over 117% and a time redundancy ratio of 100%. Their scheme can be applied to only those systolic architectures where the data for the variables are moving from one cell to the other. It is not applicable to the case when the data are stored in the cells. The scheme can only work if there is a reduction in throughput of at most 50%. It is not systematic and will incur some difficulty to extend it to any two-dimensional systolic array. Three copies of the results are computed using the same data set of an algorithm. With the increased complexity of the processing elements and the interconnection between them, the probability of failure of any of the cells is greatly increased.

Cosentino [12] proposed a dual redundancy approach whereby identical sequences of inputs are entered into two adjacent processing elements. This characteristics provides a degree of controllability and observability that allows localization of faults to within an adjacent pair of cells. The detection and correction of errors arising from faults in the array can be performed either by software in the host processor or by hardware following the system output. This software or hardware compares adjacent output terms for equality. The scheme requires a hardware redundancy ratio of O(1/N) (i.e. over 1% for large N), to perform the multiplication of two matrices. The time redundancy ratio is O(2N/(3N-1)), which is about 66%. It can detect and correct some permanent and temporary faults in the computation unit of the cells. A number of multiple fault patterns can also be detected and corrected. It is possible to locate faulty cells, because from the time each output term appears at the output, it can determine which cell has calculated that output result. Error detection and correction are done during normal operation of the system.

The limitations of the technique are as follows: It can only be applied to those systolic arrays in which the cells retain partial results rather than pass them on. It requires halving the maximum effective output rate, that is, the throughput is reduced by 50%. It cannot detect nor correct faults in the input/output registers and the interconnection lines.

Also, it cannot tolerate faults in the monitoring circuit, correction circuit or host processor. Finally, since the data flow of the sequences of inputs entered into two adjacent processing elements are identical, some categories of transient faults will not be tolerated by the scheme.

Cosentino [34] proposed a concurrent error correction technique that combines systolic array circuit architectures with residue numbers system (RNS) computations. Independent computations are performed in modulo-controlled processing channels. Calculations done in each of the RNS processing channels are identical but for the moduli they employ. Each cell is an independent processing element. A column of five cells constitutes a processing block. Two channels in the processing model can be made redundant, to automatically form what is in effect a triply redundant array. The input interface converts a binary input sequence into five residue sequences. The output converter receives five unmodified residues simultaneously and converts them to their binary equivalent. The technique does not remove the faulty processing elements, like all fault-tolerance techniques applied to signal processing arrays, that depend on reconfiguring the processing elements, but removes the errors produced by that faulty element.

The scheme requires a hardware redundancy ratio of $O(2N^2 / 3N^2)$, which is about 67%. This percentage value should be higher because, it requires five cells to perform the computation which would have been done by only one cell. As a result, the hardware redundancy ratio could be up to 400%. Based on the same argument, the time redundancy ratio is over 100%. The technique can tolerate single temporary and permanent faults occurring in the cell provided that one output in one of the channels is affected. With the capability to correct any erroneous residue, the system becomes tolerant to any pattern of faulty cells that has no more than one faulty cell in a processing block. A correctable pattern may include cells from any combination of channels. It has on-line error correction capability, since faults are corrected during the normal operation of the systolic array.

The limitations of the approach are, it requires large area and time overheads. It can

be applied to only the variations of systolic architectures in which the partial results remain in place while the input data sequences are shifted through the cells. It can correct faults in the cells, however, it cannot correct faults in the data paths, binary-to-residue and residue-to-binary conversion circuits, and the error calculator. Also the conversion and the correction circuits are very complex hardware and require large silicon area for implementation. It is prone to inaccuracies arising from the conversion.

Ari and Friedlander [35], proposed a top-down approach for implementing algorithm-based fault-tolerance in parallel processing arrays. The approach integrates error detection with the execution of the algorithm itself. It is based on the notion of *diagnostics invariance* : during error-free operation of a system, certain characteristics of the input data, which is called diagnostics, match the characteristics of the output data. Error detection is done during normal operation. Once an error is detected, normal operation is suspended and full processing power of the array can be assigned to locate and correct, or replace, the faulty component. The approach consists of five steps: (i) obtain a projective regular iterative algorithm (P-RIA). (ii) identify potential diagnostic expressions. (iii) select the desired diagnostic expressions and merge them with the P-RIA representation of the algorithm itself. (iv) convert the augmented P-RIA representation into an RIA format. (v) design a parallel processing implementation for the resulting RIA format.

The problems with this approach are as follows: It is not systematic. Only steps (iv) and (v) can be carried out in a systematic fashion. The least systematic step is the identification (selection) of an efficient diagnostic expression. A great amount of effort is required to perform steps (i)-(iii). With this approach, the type of error detection scheme depends on the diagnostic expression. This means that, the error-detection scheme could change from one problem to the other. The properties of efficient diagnostic expressions are not discussed, and moreover, the technique to determine and select the efficient diagnostic expressions are not proposed.

The scheme can detect faults in the cells but not in the interconnection between the

cells of the array. Also, how to tolerate faults in the error detection circuit is not addressed. The fault location issue which is the basis of achieving fault-tolerance with this approach is not discussed. The effects of faults can only be corrected off-line after the faults have been detected or located. However, the predominance of transient faults in VLSI arrays requires faults to be corrected with normal operations [9,21,29]. The complexity of the additional hardware and time required for error detection depends on the diagnostic expressions. The more efficient the expressions, the less complex the hardware for error detection. Our design scheme overcomes the short-comings of the technique proposed in [35].

Choi and Malek [33] proposed an algorithm-based fault-tolerance similar to the technique proposed in [35]. It involves testing the invariants of the systolic array during normal operation, which provides sufficient information for fault diagnosis. Transient and permanent computation errors may be detected by using error checking code and redundant cells. A block with single faulty cell can be located. Off-line fault testing, location and reconfiguration are used to achieve fault-tolerance. This approach inherits most of the short-falls of the approach proposed in [35].

Varman and Ramakrishnan [21a], Kumar and Tsai [22] proposed reconfiguration approaches to achieve fault-tolerance in systolic arrays. Both approaches involve mapping algorithms to linear and fault-tolerant systolic arrays having limited I/O requirements. The approach in [22] is the same as that in [21a], only that the former is an improved version. Our design scheme is not based on reconfiguration approach. The main reason for comparing it with these reconfiguration approaches is that the latter approaches considered design of systolic arrays with optimal I/O bandwidth of $O(\sqrt{N})$. In the linearization of the systolic array, PE's are laid out in a straight line with a system of buses running parallel to them. The partial results do not move, each element of the generated variable is computed by one PE. The PE's are configured using the buses through switch settings, after scanning the wafer for faulty PE's. Buses and switches are assumed to be reli-

able.

The advantages of the schemes are, they require less number of processors ($O(N\sqrt{N})$). Also, they require I/O bandwidth of $O(\sqrt{N})$ which is less than $\Omega(N)$, that many of the designs in the literature require for problems of size N^2 [22]. The limitations of the approaches are: the issue of fault location which is necessary for fault-tolerance is not discussed. Only permanent faults can be tolerated, temporary faults are not tolerated. Furthermore, the faults that can be tolerated are those that occur in the PE's. Fault in the buses and switches are not covered. The schemes cannot be applied in real time application because of high computation delay. Finally, error correction cannot be done with normal operation of the system.

The TMR scheme proposed in this chapter overcomes most of problems of the existing schemes, the only drawback is that it requires high hardware overhead to do so. It is important to mention that transient faults are the most common and frequent faults in VLSI systolic arrays. Any good fault-tolerant approach should address this type of faults. Our approaches have made a significant progress in handling such faults.

4.5 CONCLUDING REMARKS

In this chapter, we have presented a systematic approach for designing fault-tolerant systolic array architectures using space-time mapping techniques. Our approach involves the introduction of redundancy at the algorithmic level, so that when these algorithms are mapped into specific VLSI systolic array architectures, the architectures will be inherently fault-tolerant. The procedure followed in the proposed approach include the derivation of different dependency matrices corresponding to the different versions of a given algorithm and the application of space-time mapping techniques to obtain a fault-tolerant systolic array implementation of that algorithm.

Three fault-tolerant mapping methods have been proposed and investigated. These can be divided into two groups. In the first group, the dependency matrices corresponding to the different versions of the algorithm are obtained and then mapped into respective systolic arrays. Fault-tolerant systolic array is constructed by merging the respective systolic arrays. In the second group, the dependency matrices are combined to give a resultant dependency matrix that reflects a given fault-tolerance requirement. This resultant dependency matrix is then mapped into a fault-tolerant systolic array. It is observed that, for those design methods where the three versions of the algorithm are mapped into the same VLSI space, less silicon area is required for routing data into the fault-tolerant systolic arrays. Conversely, more silicon area for data routing is required for those methods where the versions are mapped onto different VLSI space.

In the approaches described in this chapter, both temporal and spatial redundancy techniques are employed to tolerate the faults in the systolic architecture. The temporal redundancy technique require a time redundancy of O(2N) and no area overhead. While the spatial redundancy technique requires a hardware overhead of a factor of 2. The TMR technique can tolerate all single permanent faults and a majority of the multiple fault patterns. In addition to providing tolerance against hardware faults like some other design schemes, our approach has the ability to tolerate certain categories of transient faults that will go undetected in some of the existing schemes. Although, the proposed alternative TMR approach utilizes higher hardware overhead to achieve fault-tolerance, however, it overcomes most of the limitations of the other approaches proposed in the literature. This fault-tolerant mapping technique could be extended such that fault diagnosis and rescheduling of the data flow are implemented to achieve better utilization of the inherent redundancy in the VLSI systolic array architecture.

In our approach and other approaches that use TMR technique to design fault-tolerant systems, at least, three modules are necessary in the voting system. The hardware overhead for fault-tolerance is, thus, at least 200 percent, without counting the cost of the

voter, which could be quite complex. Therefore, the cost of the fault-tolerant systolic arrays designed in this chapter is very high, although, the arrays have a better fault coverage than those designed using the conventional TMR technique. Since area and yield are strongly related, it is essential that a methodology be developed by which the area requirements of the FT technique is minimized, thus enhancing yield as well as the overall cost.

A lower cost fault-tolerance technique is to design the FT systolic array to produce an indication of errors in the computation during normal operation. This will then be followed by further steps that will identify the faulty module and also provide the correction of the errors. In order to deal with these issues, we will present in chapter V, an approach to design area efficient architectures for concurrent error detection (CED) in systolic arrays. Our CED technique will also be compared with many other techniques that have been proposed in the literature. The concurrent error detection technique proposed in chapter V will be extended to design lower cost (in terms of area) fault-tolerant systolic arrays and this issue will be addressed in chapter VI.

4.6 REFERENCES

[1] H. T. Kung, "Why Systolic Architectures?," *IEEE Computer*, Vol. C-31, pp. 37-46, Jan. 1982.

[2] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer and D. V. B. Rao, "Wavefront Array Processors: Language, Architecture and Applications," *IEEE Trans. Comput.*, Vol. C-31, pp. 1054-1066, Nov. 1982.

[3] H. T. Kung and M. S. Lam, "Fault-Tolerance and Two Level Pipelining in VLSI Systolic Arrays," *MIT Conference on ADV Research in VLSI*, pp. 74-83, Jan. 1984.

[4] P. K. Lala, *Fault Tolerant and Fault Testable Hardware Design*, Prentice Hall International, 1985.

[5] T. Anderson and P. A. Lee, *FAULT TOLERANCE - Principles and Practice*, Prentice Hall, 1981.

[6] P. Agrawal, "RAFT : A Recursive Algorithm For Fault Tolerance," *International Conf. on Parallel Processing*, pp. 814-821, August, 1985.

[7] J. A. Abraham, P. Banerjee, C-Y Chen, W. K. Fuchs, S-Y Kuo and N Reddy, "Fault - Tolerance techniques for systolic arrays," *IEEE Computer*, pp. 65-74, July 1987.

[8] J. V. Neuman, "Probabilistic Logics and Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, No. 34, pp. 43-99, Princeton, NJ : Princeton University Press.

[9] J-H Kim and S.M. Reddy, "A Fault-Tolerant Systolic Array Design using TMR Method," *1985 ICCD*, pp. 769-773.

[10] M. O. Esonu, S. Hariri and A. J. Al-Khalili, "A Systematic Approach for Designing Fault - Tolerant Systolic Architectures," in *Proc. 1989 Joint Tech. Conf. on Circuits/Systems, Comput. and Communications*, Sapporo, Japan, June 25-27, 1989.

[11] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "Variation on the Theme for Designing Fault-Tolerant Systolic Array Architectures," *Pacific RIM Conference on Communications*, Victoria, B.C., May, 1991.

[12] R. J. Cosentino, "Concurrent Error Correction in Systolic Architectures," *Proc. IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 1, pp. 117-125, January 1988.

[13] S-W Chan and C-L Wey, "The Design of Concurrent Error Diagnosable Systolic Arrays for Band Matrix Multiplication," *Proc. IEEE Trans. on Computer-Aided Design*, Vol.7, No.1, pp. 21-37, January 1988.

[14] J-H Kim and S. M. Reddy, "On the Design of Fault-Tolerant Two-Dimensional Systolic Arrays for Yield Enhancement," *IEEE Trans. Comput.*, Vol. 38, No. 4, April

1989.

[15] F. T. Leighton and C. E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Trans. Computers,* Vol. C-34, pp. 448-461, May 1985.

[16] D. S. Fussell and P. J. Varman, "Designing Systolic Algorithms For Fault-Tolerance," *Proc. of the IEEE Int'l Conf. on Comp. Design: VLSI in Comp.*, pp. 616-622, 1984.

[17] Y-H Choi, S. H . Han and M. Malek, "Fault Diagnosis of Reconfigurable systolic arrays," *ICCD '84*, pp. 451-455, 1984.

[18] M. S. Lee and G Frieder, "Massively Fault-tolerant Cellular Array," *IEEE Int'l Conf. Parallel Processing*, pp. 343-350, 1986.

[19] J. H. Hwang and C. S. Raghavendra, "VLSI Implementation of Fault-Tolerant Systolic Arrays," *ICCD '86*, pp. 110-113, 1986.

[20] J. H. Kim and S. M. Reddy, "On Easily Testable and Reconfigurable Two-Dimensional Systolic Arrays," *ICPP '87*, pp. 101-109, 1987.

[21] L. A. Shombert and D. P. Siewiorek, "Using Redundancy for Concurrent Testing and Repairing of Systolic Arrays," *FTCS*, pp. 244-249, July 1987.

[21a]P. J. Varman and I. V. Ramakrishnan, "Optimal matrix multiplication on fault-tolerant VLSI array," in *Proc. ICALP*, 1985.

[22] V. K. Prasanna Kumar and Y-C Tsai, "On mapping Algorithm to Linear and Fault-Tolerant Systolic Arrays," *IEEE Trans. Comput.*, Vol. 38, No. 3, pp. 470-480, March 1989.

[23] H. F. Li, R. Jayakumar and C. Lam, "Restructuring for Fault-Tolerant Systolic Arrays," *IEEE Trans. on Comp.*, vol. 38, No. 2, pp. 307-311, Feb., 1989.

[24] F. Lombardi, M. G. Sami and R. Stefanelli, "Reconfiguration of VLSI Arrays by Covering," *IEEE trans on Computer-Aided Design*, vol. 8, No. 9, pp. 952-965, September, 1989.

[25] I. Koren and M. A. Breuer, "On Area and Yield Considerations for Fault-Tolerant VLSI Processor Arrays," *IEEE Trans. on Comput.*, Vol. C-33, No. 1, pp. 21-27, Jan., 1984.

[26] J. H. Patel and L-Y Fung, "Concurrent Error Detection in ALUs by Recomputing With Shifted Operands," *IEEE Trans. Computers*, pp. 589-595, July, 1982.

[27] M. G. Sami and R. Stefanelli, "Fault-Tolerance of VLSI Processing Arrays : The Time Redundancy Approach," *IEEE Comp. Soc. Press; Proceedings of the Real-Time Systems Symp*, pp. 200-207, 1984.

[28] R. K. Gulati and S. M. Reddy, "Concurrent Error Detection in VLSI Array Structures," *IEEE Int'l Conf. Computer Design: VLSI in Computers*, pp. 488-491, Oct., 1986.

[29] K-H Huang and J. A. Abraham, "Low Cost Schemes for Fault Tolerance in Matrix Operations with Processor Arrays," *Proc. 9th Symp. on Computer Architecture*, pp. 330-337, May, 1982.

[30] K-H Huang and J. A. Abraham, "Fault-Tolerant Algorithms and their Application to Solving Laplace Equations," *IEEE Int'l Conf. Parallel Processing*, pp. 117-122, August, 1984.

[31] K-H Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Comput.*, vol. C-33, No. 6, pp. 518-528, June, 1984.

[32] J-Y Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE*, vol.74, No. 5, pp. 732-741, May, 1986.

[33] Y-H Choi and M Malek, "A Fault-Tolerant Systolic Sorter," *IEEE Trans. on Comput.*, vol. 37, No. 5, pp. 621-624, May, 1988.

[34] R. J. Cosentino, "Fault Tolerance in a Systolic Residue Arithmetic Processor Array," *IEEE Trans. on Comput.*, vol. 37, No. 7, pp. 886-890, July, 1988.

[35] H. Lev-Ari and B. Friedlander, "On the Systematic Design of Fault-Tolerant Processor Arrays with Application to Digital Filtering," *1988 VLSI SIGNAL PROCESSING III*, pp. 483-494, 1988.

[36] B. Randell, "Design Fault-Tolerance," *in Proc. IFIP symp on the Evolution of Fault Tolerant Computing*, Baden Austria, pp. 110-121, June 30, 1986.

[37] J-C Laprie, J. Arlat, C. Beounes, K. Kanoun and C. Hourtolle, "Hardware - and Software - Fault Tolerance: Definition and Analysis of Architectural Solutions," *FTCS*, pp. 116-121, 1987.

[38] D. K. Pradhan, *Fault - Tolerant Computing, Theory and Techniques*, Vol. I, Prentice Hall, 1986.

[39] V. P. Nelson and B. D. Carroll, Tutorial: Fault-Tolerant Computing, IEEE Comput. Society Press, pp. 205-211, 1987.

[40] L. Chen and A. Avizienis, "N-version Programming: A Fault-tolerant Approach to Reliability of Software Operation," *Proc. Int. Symp. Fault-tolerant Computing*, pp. 3-9 1978.

[41] H. Hecht, "Fault-tolerant software," *IEEE Trans. Reliability*, pp. 227-232, August 1979 .

[42] D. F. Barbe, "VHSIC Systems and Technology," *Computer*, pp. 13-22, February, 1981.

CHAPTER V

AREA EFFICIENT COMPUTING STRUCTURES FOR CONCURRENT ERROR DETECTION IN SYSTOLIC ARCHITECTURES

5.1 INTRODUCTION

As indicated in chapter IV, fault-tolerance can be achieved at a lower cost by employing Concurrent Error Detection (CED) techniques. Consequently, error detection has become an important aspect of fault-tolerance techniques.

Concurrent error detection is the process of detecting and reporting errors while, at the same time performing the normal operations of the systems [1]. In general, error detection requires some form of redundancy in either *hardware, information, time,* or *software.* The earliest form of the error - checking technique was the use of information redundancy such as error - detecting and correcting codes. It is quite a difficult problem to incorporate error detection and correc ' ı capabilities into general functional modules [2]. This is because simple parity-based encodings are not preserved under computations such as arithmetic operations. The methodology of using self - checking circuits for on-line error detection in computers was first proposed by Carter and Schneider [3,4], and has led a large amount of research on the topic.

Another common form of redundancy is the hardware redundancy. For example, a typical hardware redundancy method is Duplication With Comparison (DWC) [3,4]. As the name implies, the module used is duplicated and the results are compared using a comparator. DWC involves a simple and straight forward design that is effective in detecting all single faults that can result in an error. The fundamental problem with hardware redundancy is its impact on physical weight, size, power consumption and cost.

The DWC method, for example, uses at least 100 percent redundancy in hardware of the module.

In order to overcome some of the difficulties with hardware redundancy, time redundancy has recently received much attention [5-8]. Time redundancy attempts to reduce the amount of extra hardware at the expense of using extra time. In many applications, additional time may be much more affordable than extra hardware [4]. For example, the speed at which operations are performed internal to an integrated circuit (IC) may be five to ten time the speed at which information can be transferred across the IC boundaries [9]. Therefore, it may be completely reasonable to perform operations internal to an IC, multiple times. The basic concept of time redundancy is the repetition of computations in ways that allow errors to be detected. In order to allow time redundancy to be used to detect permanent faults, the repeated computations are performed differently [4]. During the first computation at time, say t_0, the input operands to the module, are used directly, and the results are stored for later comparison. During the second computation at time $t_0+\delta t$, the operands are modified, prior to use, in such a way that errors resulting from permanent faults can be detected when the results are compared. Another basic concept of time redundancy is that the same hardware is used many times in differing ways such that comparison of the results obtained at the two times will allow error detection.

Many of the above classical CED techniques may be readily applied to systolic array architectures. However, the overhead in hardware or in performance may be too high for many of these techniques to be used in practice. Further along in the chapter, we will show that the combination of space and time redundancy can lead to a very attractive form of fault-tolerance.

Several CED schemes for systolic arrays have been proposed in the literature [2,6,12-24]. Gulati and Reddy [12] proposed a CED scheme called Comparison with Concurrent Redundant Computation (CCRC), in which each computation and its redundant counter part are performed in two adjacent cells simultaneously. This approach can be

applied to a class of 1-D or 2-D systolic arrays in which data as well as the (sub) results keep moving from cell to cell during computation. Wu [13] proposed a similar approach as in [12] which is applicable to unidirectional data flow linear systolic arrays. Kung [14] and Manolakos [15] proposed an approach which is similar to the approaches in [12,13]. The error detection and correction approach is called time - redundancy with interleaving for fault-tolerance (TRIFT). Instead of using PE duplication for the purpose of error detection and correction, the idea is to perform the same computation twice (time redundancy) in adjacent PE's at two different but close enough time periods, and then compare the results. If they match, there is no fault. Otherwise a roll-back is necessary to correct the fault. The most interesting feature of time redundancy versus PE duplication, is the limited increase in chip area which in turn leads to lower cost and higher production yield. An obvious drawback is the time overhead due to the increasing total problem latency and the necessary roll-back due to possible faults [14]. Like the approaches in [12,13], this scheme is restricted to systolic structures where the data as well as (sub) results keep moving.

Abraham *et al.* [2] proposed a concurrent error detection approach using linear property encoding. In their approach, CED can be easily incorporated into systolic arrays that satisfy these two conditions: (i) Each processing element in the repetitive part of the systolic array is itself a linear system. (ii) The coefficient stream passes through the array without being modified. In order to perform CED, new variables are introduced to the algorithm so that it will be evaluated in a linear fashion. The key point in designing two-dimensional systolic arrays with the CED is to suitably partition the array by rows or by columns (so that each row or column will be a linear system), and apply the scheme to each of the partitioned rows or columns. An extra PE is added at the end of each row/column to perform the operation of the algorithm before it is linearized. The results computed by the extra PE are compared with the results exiting at the output of the linear array of each row/column. Any inconsistency will reveal that there exists a faulty PE.

This scheme is applicable to the linear systolic arrays where the generated variables must propagate from one cell to another until the final results appear at the output of the linear array.

Huang and Abraham [16] also proposed a technique to detect and correct errors in matrix operations performed by processor arrays. The strategy used in the technique is to encode the input matrices and check the encoded result matrices to judge whether or not the outputs are erroneous. Cosentino [17] proposed a concurrent error correction scheme which is also based on concurrent redundant computation (CRC) approach. The scheme is restricted to a class of systolic arrays in which the partial results must stay in the cells. Furthermore, Cosentino [18] proposed a concurrent error detection and correction technique that combines systolic array circuit architectures with residue number system (RNS) computations. Calculations done in each of the RNS processing channels are identical but for the moduli they employ. Two residue calculations are performed, if the results match, there is no fault in the cells of a channel. If there is a discrepancy, it means that one of the residues is in error.

Lev-Ari and Friedlander [19] proposed an approach for implementing algorithm-based fault-tolerance in parallel processing arrays. Their approach integrates error detection with the execution of the algorithm itself. It is based on the notion of diagnostic invariance. During error-free operation of a system, certain characteristics of the input data, which is called diagnostics, match characteristics of the output data. This equality or match can be violated only by the occurrence of errors in the computations. Thus, if there is a mis-match between the characteristics of the input data and those of the output data, the fault in the system has manifested itself as an error and is detected.

Chan and Wey [20] developed a RESO - based (REcomputation with Shifted Operands) time redundancy CED scheme for the study of fault-tolerant systolic array of one dimension. The CED scheme was later implemented in a Band Matrix Multiplication Systolic Array (BMMSA). For their approach, each PE must be RESO-lized. That is,

after a PE is active for one cycle, it can be drafted to recompute the same operands (shifted) again, in one of the following idle cycles. The latches of each PE will receive new data only once every two cycles. Following a regular active cycle, a PE will be activated again to carry out the recomputation step, instead of being left in idleness. Error - contingency algorithms will be executed upon the issuance of an error signal by the comparator inside the PE. The state of error detection is consequential of conflicting results from the original and recomputed steps. They extended their scheme to implement RESO - based CED in both the Kung-Leiserson BMMSA [10] and Huang-Abraham BMMSA [11]. The scheme can be applied to hex-connected systolic arrays in which the data as well as the (sum) results keep moving from cell to cell during the computation.

Patel and Fung [6,21] and, Cheng and Patel [22] presented CED schemes which also use the approach of RESO to test a systolic array by repeating every computation with shifted operands. Their approach uses the principle of time redundancy, where the coding function c is the left shift operation and the decoding function c^{-1} is the right shift operation. Thus, in the first computation step, $f(x)$ is computed and stored in the register. During the recomputation step, x is shifted left by k bits (RESO-k) and then input to the unit f. the output is shifted right k bits and compared to the results of the first step. A mismatch indicates an error in either computation step. The function c is assumed to be such that $c^{-1}(f(c(x))) = f(x)$.

Gupta and Bayoumi [23] proposed a scheme termed as LOED (Logarithm based On-line Error Detection) which is based on the use of logarithmic coding (the addition theorem of logarithms) to test systolic cells. The addition theorem is given by $\log ab = \log a + \log b$, where a and b are two positive numbers. Considering base 2, this can be rewritten as $ab = 2^{(\log_2 a + \log_2 b)}$. Thus, the computation of the product 'ab' and $2^{(\log_2 a + \log_2 b)}$, can take place in parallel within the cell structure itself. The two separate operations can be compared, and a mismatch indicates a faulty multiplication or faulty addition process. Finally, the relationship between concurrent error detection via CRC

and space-time transformation T of a given systolic array design has been established in [24].

Some of the disadvantages suffered by most of the existing schemes [2,6,12-24] include: Reduced throughput of a unidirectional array by 50% [12,13,14,15,17,20]. They cannot detect all transient faults without incurring major overheads in hardware [2,12-15,19]. They allow the detection of only single faults. Multiple fault detection requires high hardware overhead [6,17,19,21]. Some of techniques are limited to only those systolic implementations in which the data as well as (sub) results keep moving [12-15]. Others are applied to systolic arrays where data are stored in the cells [17]. The resultant systolic structure of the scheme in [16] is not regular and granular. The techniques proposed in [2,16,18,19,23] are vulnerable to false alarm due to roundoff errors. Also, the accuracy of the results obtained using the approach in [23] depends on the number of bits required to calculate the antilogarithm. The hardware overhead is proportional to the accuracy of the results.

In this chapter of the thesis, we propose an interesting method for designing testable systolic architectures [25]. Our scheme is based on CRC approach and space-time mapping of algorithms into systolic arrays. The basic concept involves rescheduling the input data, rearranging data flow and increasing the utilization of the array cells. In the proposed approach, redundant computations are introduced in the VLSI algorithms, such that when these algorithms are mapped into specific VLSI systolic architectures, certain degree of observability and controllability are inherent in these architectures that allow concurrent error detection in the systolic architectures. We obtain two transformed dependency matrices (TDM's) that represent the two different versions of the given algorithm. The first TDM is obtained by selecting a valid transformation matrix which transforms the dependency matrix of the algorithm into the new transformed dependency matrix. On the other hand, the second one is obtained by rotating the systolic array corresponding to the first TDM by 180 degrees about any of the indices that represent the spatial com-

ponent of the TDM. These TDM's are mapped into respective systolic arrays. Concurrent error detection (CED) systolic array is constructed by merging the corresponding systolic array of the two versions of the algorithm.

A careful observation of the data flow characteristics of the CED systolic array reveals that, two identical sets of output results (one for each version) may need to be computed at the same computational site and at the same time. Our objective is to identify such computations so that by adding extra hardware only at those computational sites at which they will be computed and by re-scheduling the input data (such that the original input data scheduling is satisfied), the interaction between the two wavefronts can be isolated. Two corresponding sets of computed output results are produced at the same time by the CED systolic array. Concurrent error detection is achieved by comparing these output results using totally self-checking circuits.

This approach offers the following advantages: It is area efficient, there is no need to replicate all the hardware in the CED systolic array. Redundant hardware is introduced only at those computational sites where it is needed. All the single transient and permanent faults in the CED array can be detected. In addition to detecting all single faults, this method has the capability of detecting multiple fault patterns with high probability of coverage. By rotating the systolic array of the first version of the algorithm by 180 degrees about one of the indices, it becomes possible to schedule the two independent computations so as to start at the same time with both using the same computational space but at different times. Consequently, there is no high-delay cost incurred. It can also be applied to any systolic array implementation, whether the case in which the data as well as (sub) results keep moving or the case in which the partial results stay in the cells. There is no reduction in throughput. This method provides an effective means of designing CED systolic array architectures and it has been shown that our scheme overcomes the major disadvantages of the other CED schemes.

The design method is applied to the matrix multiplication algorithm described in

chapter II, in order to demonstrate the generality and novelty of our approach to design testable VLSI systolic architectures. The cost of this error detection capability is minimal and this includes, the time required to flush out the computations of the two versions of the algorithm, additional hardware (in worst case, the area is increased by less than 1% only) in some of the processing elements in the CED systolic array and the totally self-checking circuits to compare the two output results of the generated output variables for equality.

The outline of this chapter is as follows: Section 5.2 describes in detail our approach for designing concurrent error detection (CED) systolic arrays. Section 5.3 discusses the analysis of the fault coverage of the proposed scheme. The area and time overhead of the design scheme is presented in section 5.4. The comparison of the CED scheme proposed in this chapter, with other CED schemes is described in section 5.5. Finally, section 5.6 contains the summary and concluding remarks.

5.2 CONCURRENT ERROR DETECTION

In this section, we present our design scheme for achieving concurrent error detection in systolic array architectures. The basic concept of the approach is described in detail and then illustrated using the matrix multiplication algorithm. Also, a five-step procedure for designing Area Efficient CED systolic architectures is presented. Before proceeding to describe the approach, we will briefly mention the class of faults that the design scheme will detect.

5.2.1 Fault Model

The fault model considered for the error detection analysis discussed in this chapter is the same as that described in chapter IV (section 4.3.1). That is, we are assuming that faults can occur in the refined level of the processing element such as the computational

unit, input/output latch registers, communication links and switches. We also assume that at most one of these modules or parts of the PE is faulty at any given time. Furthermore, it is assumed that both temporary and permanent faults can occur in the CED systolic array. Finally, we assume that the outputs of the faulty cells may assume any logical values independent of the inputs.

5.2.2 The Proposed Scheme

The developed testing scheme achieves concurrent error detection (CED) through concurrent redundant computation (CRC). It is based on the observation that at any given time the data flow computational activity is located in only some cells in the systolic array. That is, at any given time instant, some of the cells in the systolic array are performing meaningful computations of the results of the data variables while some are not. Hence, there is inherent spatial redundancy in the array which could be exploited to perform concurrent redundant computations. Two independent computations can be launched into the CED systolic array in such a way that they are performed on different but partly overlapping regions of the array. If two corresponding computations of the independent wavefronts have to share the same computational resource at the same time, then spatial redundancy can be added only at this resource to ensure that the two computations are simultaneously but separately computed on the different computational resources. Thus at the time instant when the computational wavefront of the required computation reaches the faulty cell, its redundant counter part would have been confined to a fault-free region of the array. Consequently, a comparison of the corresponding results would lead to the detection of the fault and this is true because of our single fault assumption.

In our approach, redundant computations are introduced in the VLSI algorithms. We obtain two transformed dependency matrices (TDM's) that represent the two different versions of the given algorithm. The first TDM is obtained by selecting a valid transformation matrix which transforms the dependency matrix of the algorithm into a

transformed dependency matrix. However, the second one is obtained by rotating the systolic array corresponding to the first TDM by 180 degrees about any of the indices that represent the spatial component of the TDM. These TDM's are mapped into respective systolic arrays. A concurrent error detection (CED) systolic array is constructed by merging the corresponding systolic array of the two versions of the algcrithm.

Since the systolic array corresponding to the second version of the algorithm is derived from that of the first version, therefore, both arrays have the same number of processing elements and similar interconnection patterns. However, the direction of propagation of the two data flows may be different. Hence, in this method, a merge of the two systolic arrays is equivalent to superimposing the corresponding computational sites in the two arrays. For example, let S represent the first systolic array with the following computational sites: $\{(\hat{j}_1,\hat{k}_1), (\hat{j}_2,\hat{k}_2),..., (\hat{j}_n,\hat{k}_n)\}$ and let S' represent the second systolic array with the computational sites: $\{(\hat{j}_1',\hat{k}_1'), (\hat{j}_2',\hat{k}_2'),..., (\hat{j}_n',\hat{k}_n')\}$. Therefore, a merge of the two arrays represented by $C=S \cup S'$ results in a systolic array which has the foliowing computational sites: $\{((\hat{j}_1,\hat{k}_1),(\hat{j}_1',\hat{k}_1')) , ((\hat{j}_2,\hat{k}_2),(\hat{j}_2',\hat{k}_2')) ,..., ((\hat{j}_n,\hat{k}_n),(\hat{j}_n',\hat{k}_n'))\}$. If the computational sites $(\hat{j}_i,\hat{k}_i)$ and $(\hat{j}_i',\hat{k}_i')$, (for $i = 1,2,...,n$), are the same then the two cells are the same and hence can be represented by one cell. However, it is important to note that this resultant cell must satisfy the communication requirements of the two original cells.

Although the data flow characteristics of the two versions are different, two identical sets of output results (one for each version) may need to be computed at the same computational site and at the same time. Therefore our goal is to identify such computations so that by adding extra hardware only at those computational sites at which they will be computed and by re-scheduling the input data (such that the original input data scheduling is satisfied), the interaction between the two wavefronts can be isolated. In other words, by introducing redundant hardware in only some of the CED systolic array cells, the two independent computations can be separately computed. There is no need to duplicate all

the hardware in the cells of the CED array. Also due to the input data rescheduling, all the interconnection links need not be replicated. Two corresponding sets of computed output results are produced at the same time by the CED systolic array. Concurrent error detection is achieved by comparing these output results using totally self-checking circuits.

As mentioned above, we derive the second version of the algorithm by rotating the systolic array of the first version by 180^{o} about any of the principal axis (horizontal or vertical axis). This has the advantages in that it makes it easier to merge the two systolic arrays just by superimposing their respective computational sites. Thus, minimizing the number of cells in the CED systolic array and producing a regular structure. Also, rotation by 180^{o} makes it possible to launch the two independent computations into the CED systolic array at the same time, with both wavefronts propagating towards the center of the array. Then redundant hardware is added only where these two wavefronts meet at the same time. Therefore, because of these advantages, rotation by 180^{o} will lead to better implementation.

The derivation of the second version of the algorithm is achieved as follows:
Since present practical arrays have planar layouts, thus, the transformation matrix T for a 2 - dimensional array is represented by

$$T = \begin{bmatrix} \Pi \\ S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix} \tag{5.1}$$

where Π is the time mapping function and S is the space mapping function.
If we let T be the transformation matrix of the first version of the algorithm and T' for the second version, then,

$$T' = \begin{bmatrix} t_{11}' & t_{12}' & t_{13}' \\ t_{21}' & t_{22}' & t_{23}' \\ t_{31}' & t_{32}' & t_{33}' \end{bmatrix} \tag{5.2}$$

For a rotation by 180° about the vertical axis, the elements of T' are given by,

$$\begin{aligned} t_{1l}' &= t_{1l} \\ t_{2l}' &= \begin{cases} -t_{2l} & \text{for } t_{3l} = 0 \\ t_{2l} & \text{otherwise} \end{cases} \\ t_{3l}' &= t_{3l} \end{aligned} \tag{5.3}$$

Hence,

$$\Delta' = T'D \tag{5.4}$$

On the other hand, for a rotation about the horizontal axis, the elements of T' are given by,

$$\begin{aligned} t_{1l}' &= t_{1l} \\ t_{2l}' &= t_{2l} \\ t_{3l}' &= \begin{cases} -t_{3l} & \text{for } t_{2l} = 0 \\ t_{3l} & \text{otherwise} \end{cases} \end{aligned} \qquad l=1,2,3 \tag{5.5}$$

Thus,

$$\Delta' = T'D \tag{5.6}$$

The mapping of the index set (i,j,k) into the new index set $(\hat{i}',\hat{j}',\hat{k}')$, using the transformation T' is given by,

$$(\hat{i}',\hat{j}',\hat{k}')^t = T'(i,j,k)^t + \boldsymbol{M} \tag{5.7}$$

where $\boldsymbol{M}$ is a 1x3 matrix defined as follows:

For a rotation of Δ by 180° about the vertical axis, the matrix $\boldsymbol{M}$ has the given representation,

$$\boldsymbol{M} = \begin{bmatrix} 0 \\ f(t_{2l}',t_{3l}') \\ 0 \end{bmatrix} \tag{5.8}$$

Therefore,

$$\begin{bmatrix} \hat{i}' \\ \hat{j}' \\ \hat{k}' \end{bmatrix} = \begin{bmatrix} T' \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ f(t_{2l}', t_{3l}') \\ 0 \end{bmatrix} \tag{5.9}$$

On the other hand, for a rotation about the horizontal axis, $\boldsymbol{M}$ is given by

$$\boldsymbol{M} = \begin{bmatrix} 0 \\ 0 \\ f(t_{2l}', t_{3l}') \end{bmatrix} \tag{5.10}$$

Hence,

$$\begin{bmatrix} \hat{i}' \\ \hat{j}' \\ \hat{k}' \end{bmatrix} = \begin{bmatrix} T' \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f(t_{2l}', t_{3l}') \end{bmatrix} \tag{5.11}$$

The function $f(t_{2l}', t_{3l}')$ is defined as follows:

Let I_l be the indicator of the values of the elements (t_{2l}', t_{3l}') such that, for a rotation by 180° about the vertical axis,

$$\begin{aligned} I_l &= 1 \qquad \text{for } t_{2l}' \neq 0 \text{ and } t_{3l} = 0 \\ &= 0 \qquad \text{otherwise} \end{aligned} \tag{5.12}$$

On the other hand, for a rotation about the horizontal axis,

$$\begin{aligned} I_l &= 1 \qquad \text{for } t_{2l}' = 0 \text{ and } t_{3l} \neq 0 \\ &= 0 \qquad \text{otherwise} \end{aligned} \tag{5.13}$$

Therefore,

$$f(t_{2l}', t_{3l}') = C * \sum_{l=1}^{3} I_l \tag{5.14}$$

where C is a constant given by

$$C = \min\left\{ (\max\hat{j} + \min\hat{j}) , (\max\hat{k} + \min\hat{k}) \right\} \tag{5.15}$$

5.2.3 Application of the Proposed Scheme

A Example 1

In order to illustrate our design scheme, we will consider the matrix multiplication algorithm whose dependency matrix is given in Eq.(2.7) (section 2.3). Again, as an example of the transformed dependency matrix (TDM) of one version of the algorithm, we will choose the TDM given in Eq.(2.7), which is repeated below for the sake of easy reference.

$$T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{then,} \quad \Delta = TD = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{2.7}$$

The corresponding VLSI implementation of this TDM is as shown in Fig. 5.1. The cell structure is the same as depicted in Fig.2.10. In this thesis, our approach for determining the TDM corresponding to the second version of the algorithm is to *rotate*, the systolic array corresponding to the TDM of the first version, by 180 degrees about the horizontal-axis or the vertical-axis (for a 2-D array). Thus, if we rotate the corresponding VLSI implementation of the TDM in Eq.(2.7) by 180^{o} about the vertical-axis and using the results derived in section 5.2.2, we obtain the following transformation matrix T' and hence Δ',

$$T' = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and,} \quad \Delta' = \Delta_{rot-vert} = T'D = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{5.16}$$

Therefore, we have to prove that this TDM generated by rotation preserves the order of computation of the given algorithm and also, all other dependency constraints are not violated.

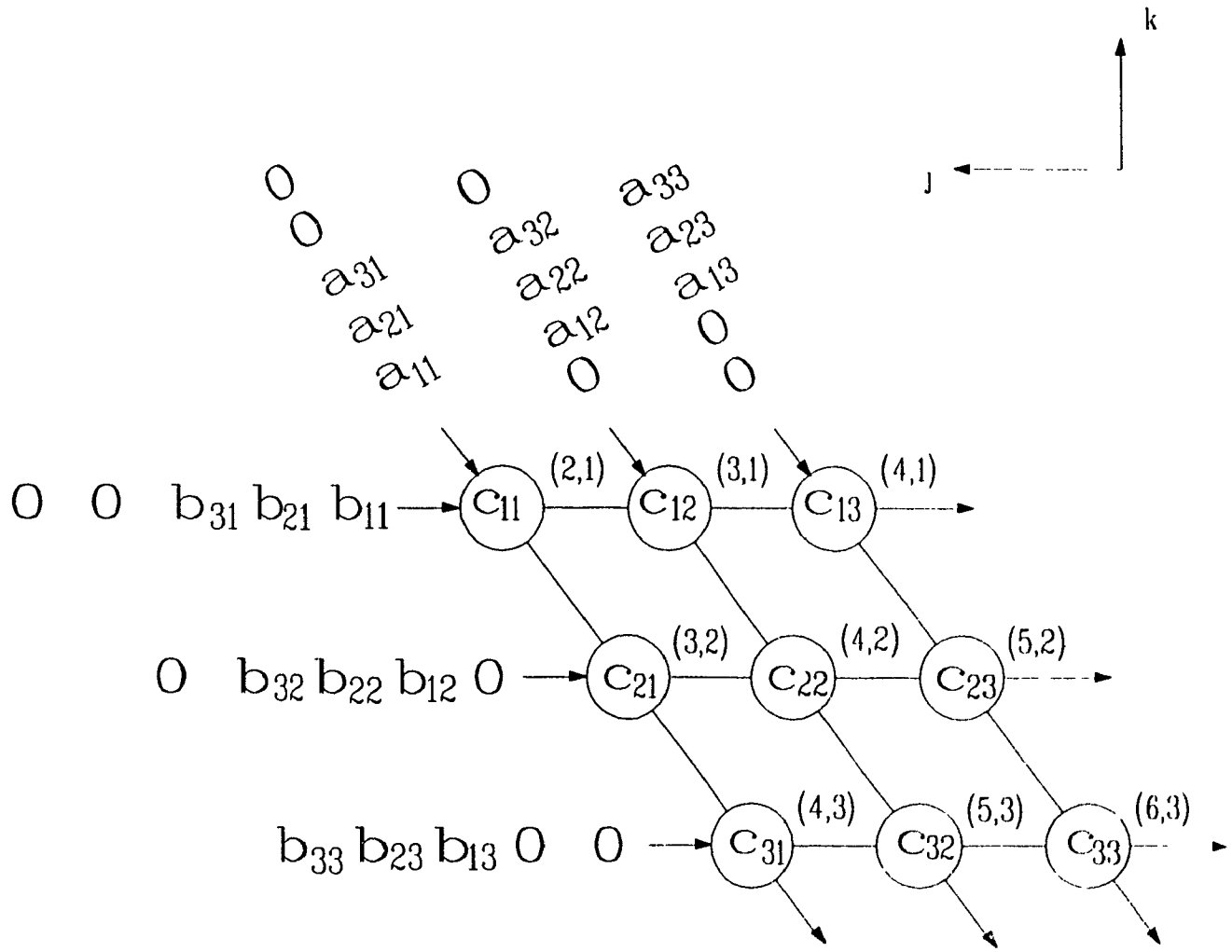


Figure 5.1 VLSI array structure that implements the matrix multiplication algorithm (for N=3).

Theorem 5.1: Rotating the systolic array corresponding to any Δ by 180 degrees about any of the principle axis, merely changes the direction of propagation of the input data of the variables affected. The order of computation is preserved and all other dependency constraints are not violated.

Proof: If the resultant systolic array of a given TDM is rotated about any of the principle axis and by any angle, this operation does not affect either the time component of the TDM or the number of interconnections of each cell in the systolic array. In section II, it is stated that, for a two - dimensional systolic array, the conceptual sites $\{\bar{J} = (i,j,k)\}$ must be mapped into $Z^3 = \{(t,y\ y)\}$ where t specifies the time when a node is computed and where (x,y) represents the 2-dimensional physical coordinates of the place in the VLSI systolic array where the node is computed. In other words, $(\hat{i},\hat{j},\hat{k})^t = T(i,j,k)^t$, where $\hat{i}$ is the time component and $(\hat{j},\hat{k})$ is the spatial component of the transformed indices. Since $(\hat{j},\hat{k})$ represent the physical coordinate of a node in the VLSI array, and since this physical coordinate of a node is not a function of $\hat{i}$, thus, any operation on the array itself affects only the nodes and not the timing structure. The time at which the nodes will compute the output results still remains the same. Hence the rotation about any of the principle axis and by any angle does not affect the time component of the TDM. Furthermore, since the number of the interconnection links is unchanged and since the number of the cells remains the same, consequently, the only geometrical property of the array structure that is affected by the rotation by 180^o, is the direction of propagation of the data variables. Therefore, since the timing structure and the interconnection pattern of the TDM is not altered by the rotation operation, thus, the order of computation of the algorithm must be preserved and also all the other dependency constraints must be satisfied.

Q.E.D

The mapping of the index set (i,j,k) into new index set $(\hat{i}',\hat{j}',\hat{k}')$ using the transformation T' is given by ,

$$\begin{bmatrix} \hat{i}' \\ \hat{j}' \\ \hat{k}' \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ C \\ 0 \end{bmatrix} = \begin{bmatrix} i + j + k \\ -j + k + C \\ k \end{bmatrix} \tag{5.17}$$

In section 2.3, the mapping of the index set into VLSI array using the transformation T (Eq.(2.7)) is shown in Fig.2.10. By employing the results of Eq.(5.15), and from Fig.2.10, the value of the constant C is 4. Figure 5.2 consists of the mapping of the index set into VLSI arrays using the transformation matrices T and T' (for N=3). In this case, $\hat{i}' = \hat{i}$.

The VLSI systolic array structure for the TDM in Eq.(5.16) is shown in Fig.5.3 (for N=3). The array structure is the same as that in Fig.5.1, with the difference being that the data variable B in Fig.5.3 propagates in the opposite direction to that in Fig.5.1. Also the cell structure of Fig.5.3 is similar to that depicted in Fig.2.12. The CED systolic array, shown in Fig.5.4, is constructed by merging the systolic arrays of Figs. 5.1 and 5.3, using the merging techniques described in section 5.2.2. One method of achieving concurrent error detection in the array of Fig.5.4 would be to duplicate all the functional blocks in each cell so that the results of the two independent computations are computed at the same time using each set of the functional blocks in each cell [3,4]. Then the two corresponding independent results are compared at the output of the array for equality using the totally self-checking equality checker. Since the data flow characteristics of the two versions of the algorithm are different, hence the two versions exhibit different dynamic properties. The effects of any temporary faults on one version will not be the same as in the other version as opposed to the case if only one version of the algorithm is to be duplicated [26]. Therefore, such temporary faults, if they occur will be detected.

i	j	k	$\hat{i}$ time	$\hat{j}$ processor	$\hat{k}$	$\hat{j}'$ processor	$\hat{k}'$
1	1	1	3	2	1	4	1
1	1	2	4	3	2	5	2
1	1	3	5	4	3	6	3
1	2	1	4	3	1	3	1
1	2	2	5	4	2	4	2
1	2	3	6	5	3	5	3
1	3	1	5	4	1	2	1
1	3	2	6	5	2	3	2
1	3	3	7	6	3	4	3
2	1	1	4	2	1	4	1
2	1	2	5	3	2	5	2
2	1	3	6	4	3	6	3
2	2	1	5	3	1	3	1
2	2	2	6	4	2	4	2
2	2	3	7	5	3	5	3
2	3	1	6	4	1	2	1
2	3	2	7	5	2	3	2
2	3	3	8	6	3	4	3
3	1	1	5	2	1	4	1
3	1	2	6	3	2	5	2
3	1	3	7	4	3	6	3
3	2	1	6	3	1	3	1
3	2	2	7	4	2	4	2
3	2	3	8	5	3	5	3
3	3	1	7	4	1	2	1
3	3	2	8	5	2	3	2
3	3	3	9	6	3	4	3

Fig.5.2 : Mapping of index set into VLSI arrays using transformation matrices T and T' (for N=3).

In our proposed approach of achieving CED in Fig.5.4, rather than duplicating all the functional blocks in each cell, additional hardware is added only to those cells where the corresponding results of the two independent computations have to be computed at the same time. As seen from Figs.5.2 and 5.4, the results of the coefficients a_{i2} (i=1,2,3) for the two versions need to be computed in cells (3,1), (4,2) and (5,3) (at t=4,5,6 , t=5,6,7 and t=6,7,8 respectively). If the same functional blocks in these cells are to be used to compute the two independent results and if any of these blocks fails, this fault might have the same effect on the two output results and hence will not be detected. Consequently, the functional blocks in cells (3,1), (4,2) and (5,3) are duplicated in order to compute the corresponding results separately.

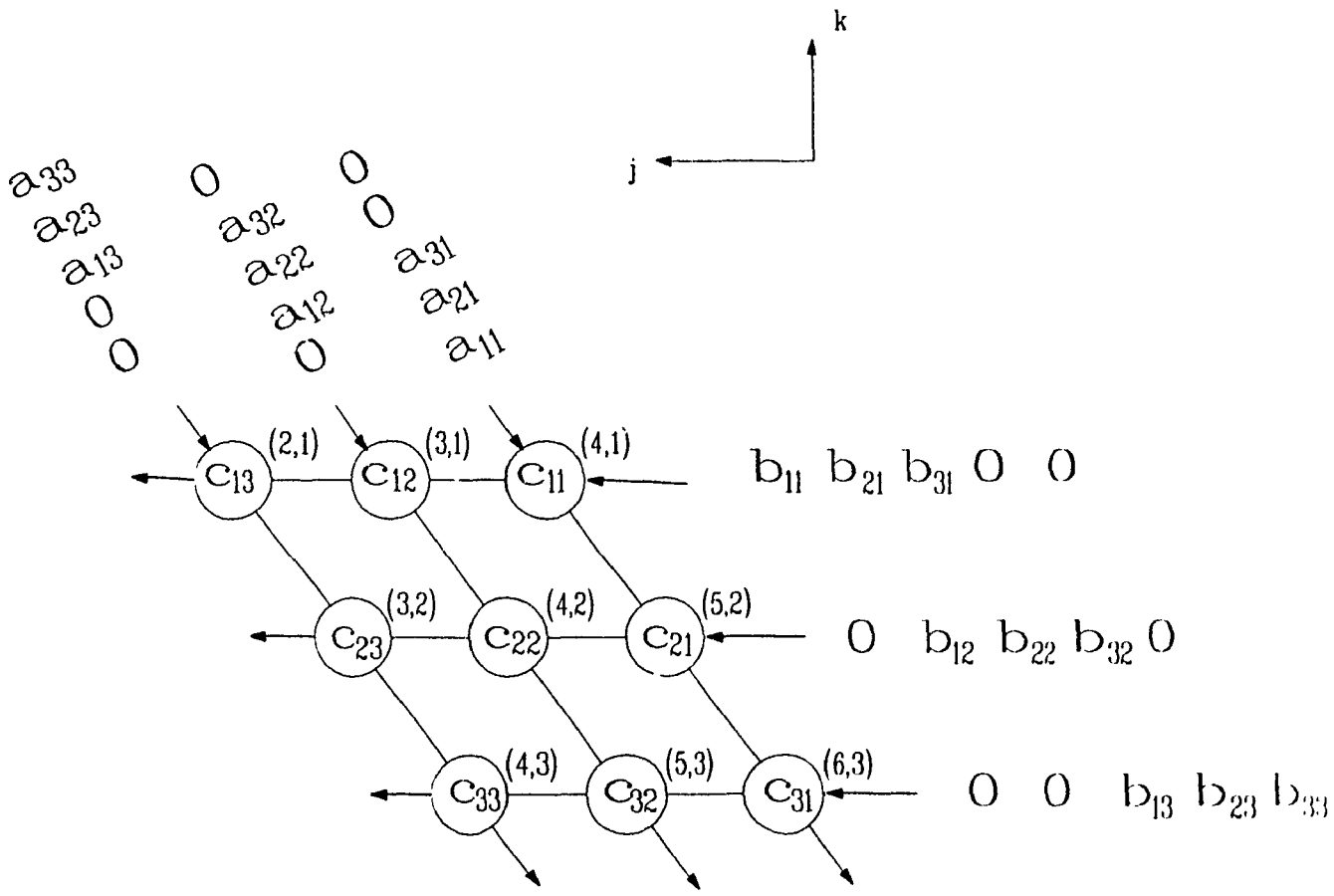


Figure 5.3 The VLSI array structure resulting from the rotation of Figure 5.1 by 180° about the vertical axis.

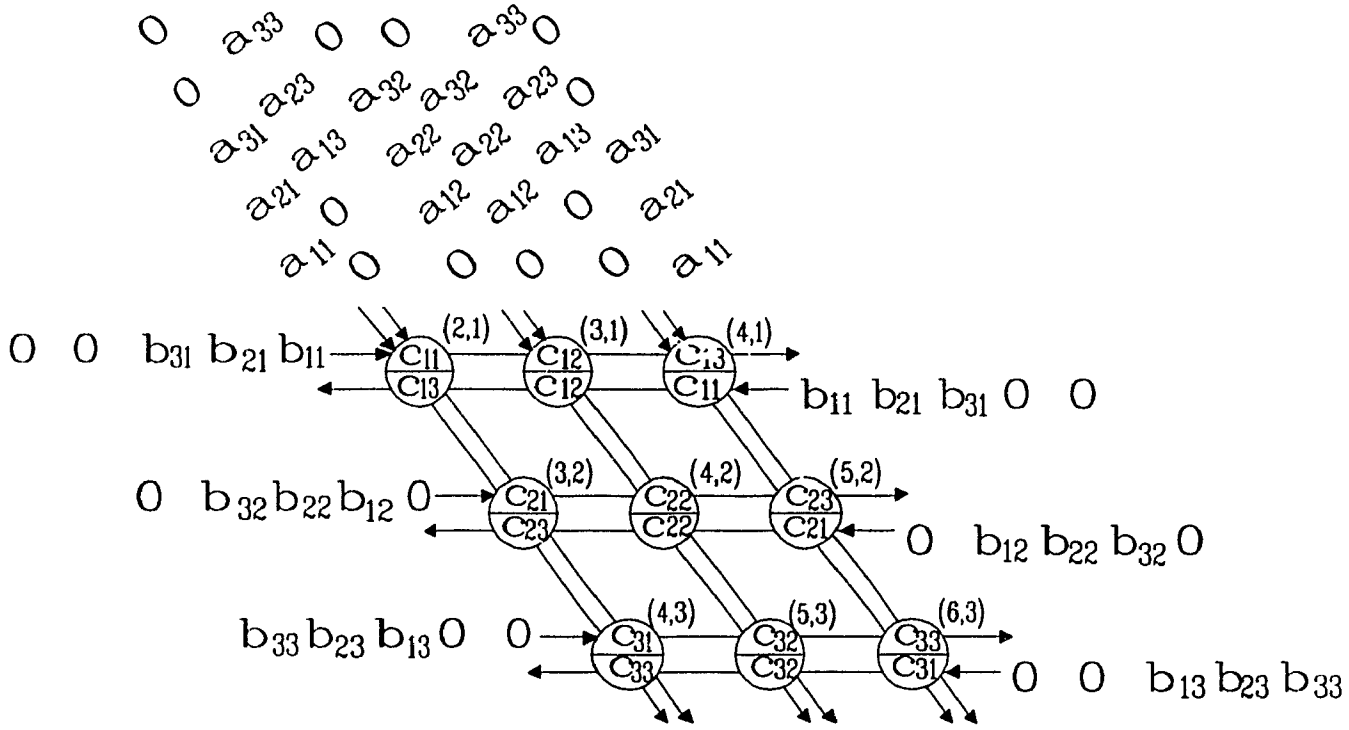


Figure 5.4 The VLSI array structure resulting from merging the systolic arrays of Figures 5.1 and 5.3 (for N=3).

The input data for variable A are re-scheduled so as to satisfy the original data scheduling. In order to synchronize the data propagation of each of the computations, additional delay elements are introduce in the interconnection lines for variable B. The input data for variable C are stored in the cells.

Figure 5.5 depicts the CED systolic array (for N=3) designed using our technique. The array consists of two types of cell structures. In the first type, the functional blocks are not duplicated, while the functional blocks are duplicated in the second type. The respective structure of the two types of cells are shown in Fig.5.6(a) and Fig.5.6(b). Figure 5.5 also contains 6 latches which are used as delay elements to synchronize the data propagation of the two independent computations. Also, as observed in this figure, the corresponding results of the two versions arrive at the output of the array at the same time. Therefore additional circuitry is not required to synchronize the arrival of the output results. Hence our scheme can be applied in real-time applications. The output results of the two independent computations are compared for discrepancies (as a result of a fault in the array) using a totally self-checking (TSC) comparator as shown in Fig.5.5.

Figures 5.5a (i), (ii), (iii), (iv) and (v) show the first five pulsations of CED systolic array. The outputs labeled UU produce the results which are computed using the data for variable C stored in the upper part of the cells as shown in Fig.5.5. Similarly, the outputs labeled LL compute the partial results of the matrix multiplication using the data for variable stored in the lower part of the cells. In addition, the paths labeled UU and LL indicate the data flow of the two independent computations. It can be seen that the two data flow computations traverse different functional blocks and interconnection lines in the CED systolic array. Hence, any single fault in one of the computations will be detected at the output of the CED systolic array.

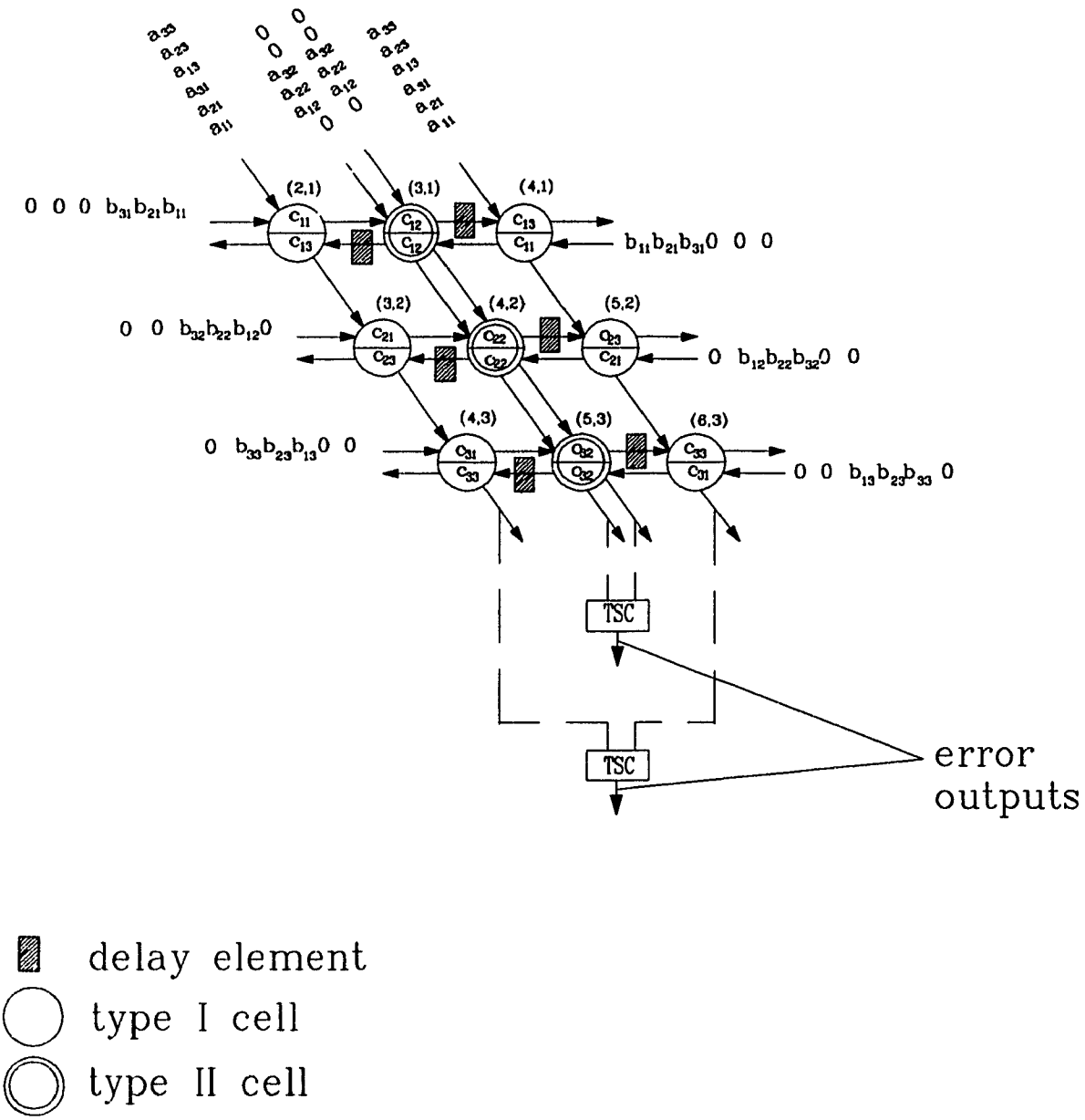


Figure 5.5 CED systolic array for matrix multiplication (for N=3).

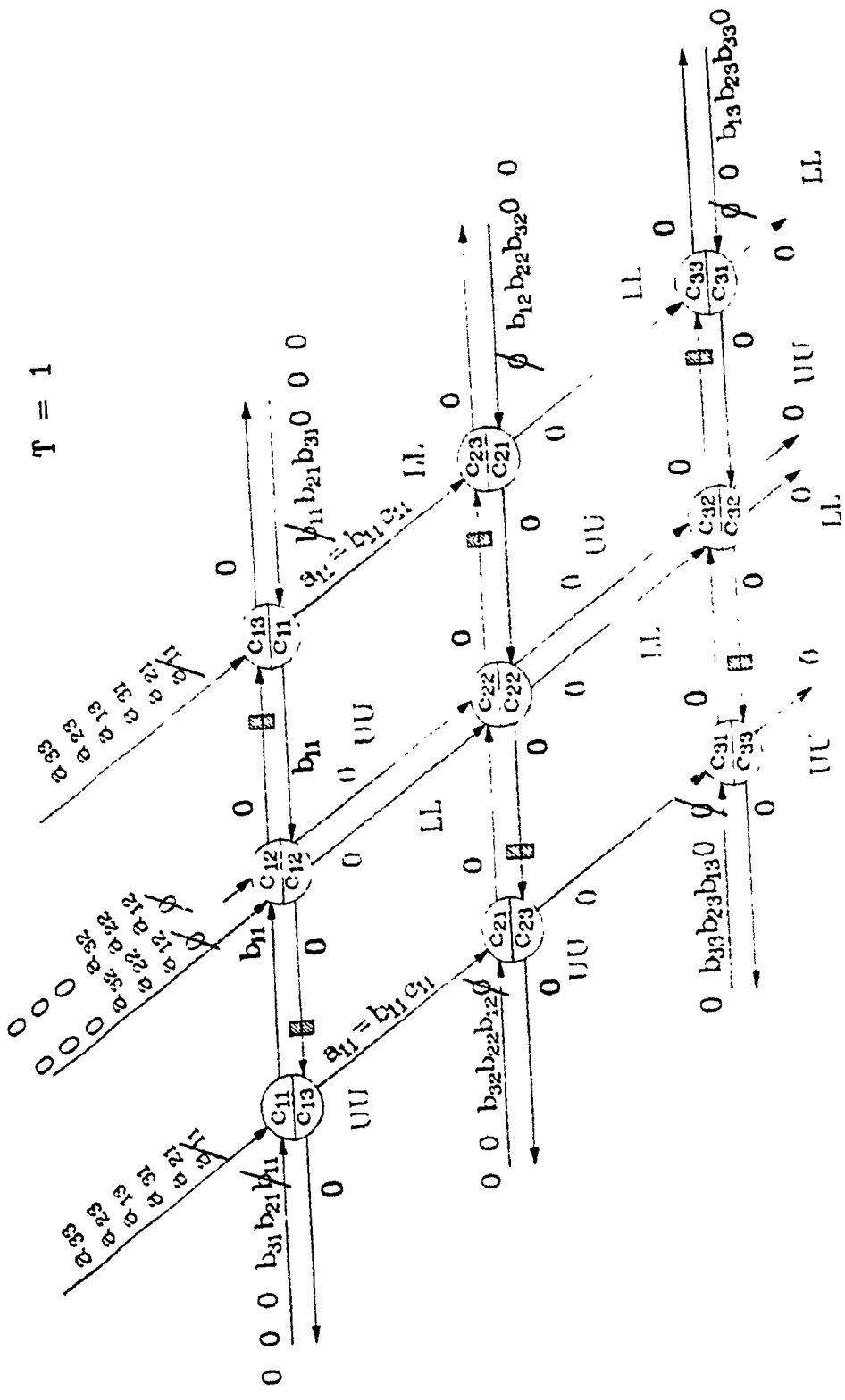


Figure 5.5(a) (i) The first pulsation of the CED systolic array for matrix multiplication (for N=3)

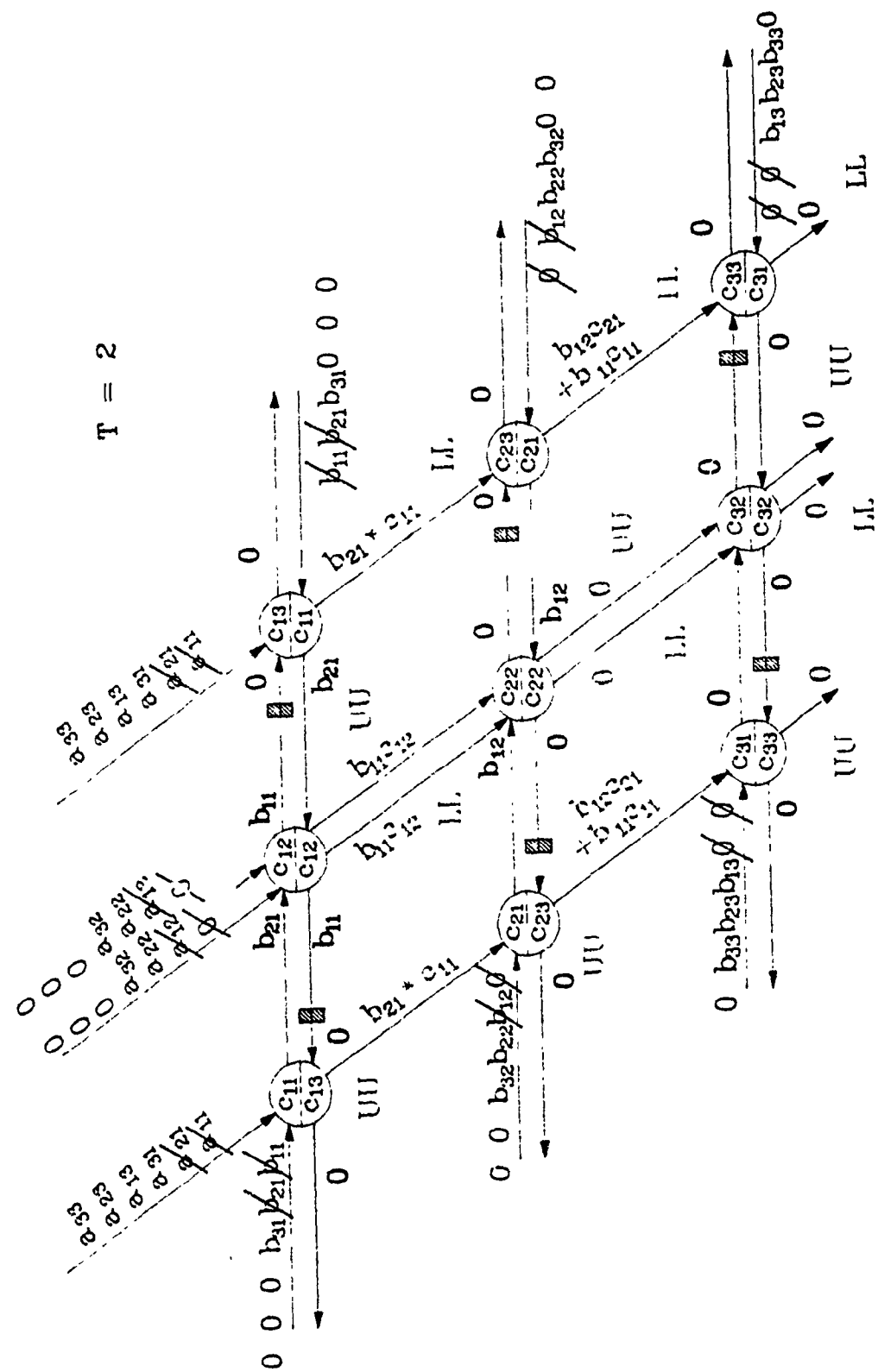


Figure 5.5(a) (ii) The second pulsation of the CED systolic array for matrix multiplication (for N=3).

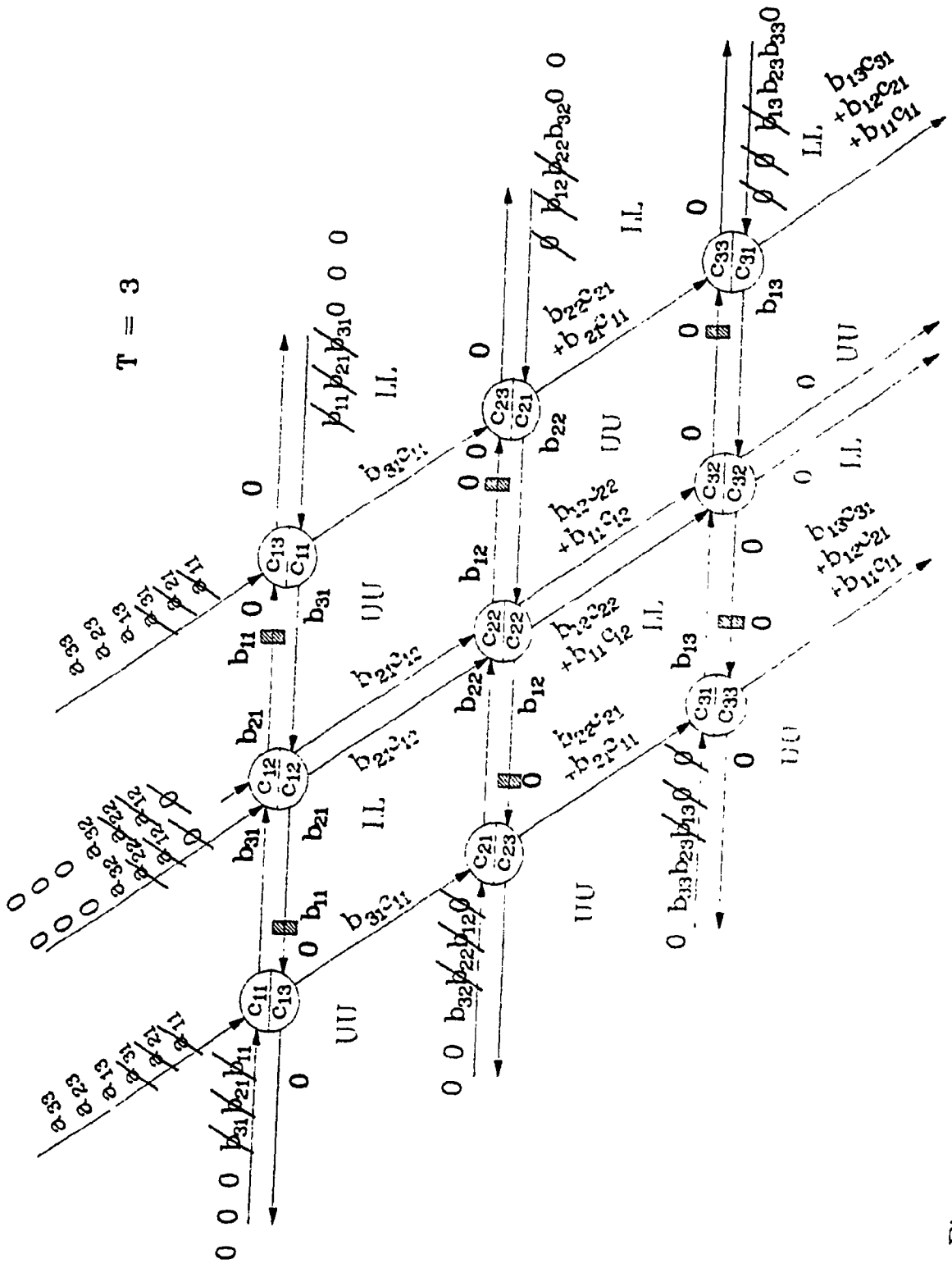


Figure 5.5(a) (iii) The third pulsation of the CED systolic array for matrix multiplication (for N=3)

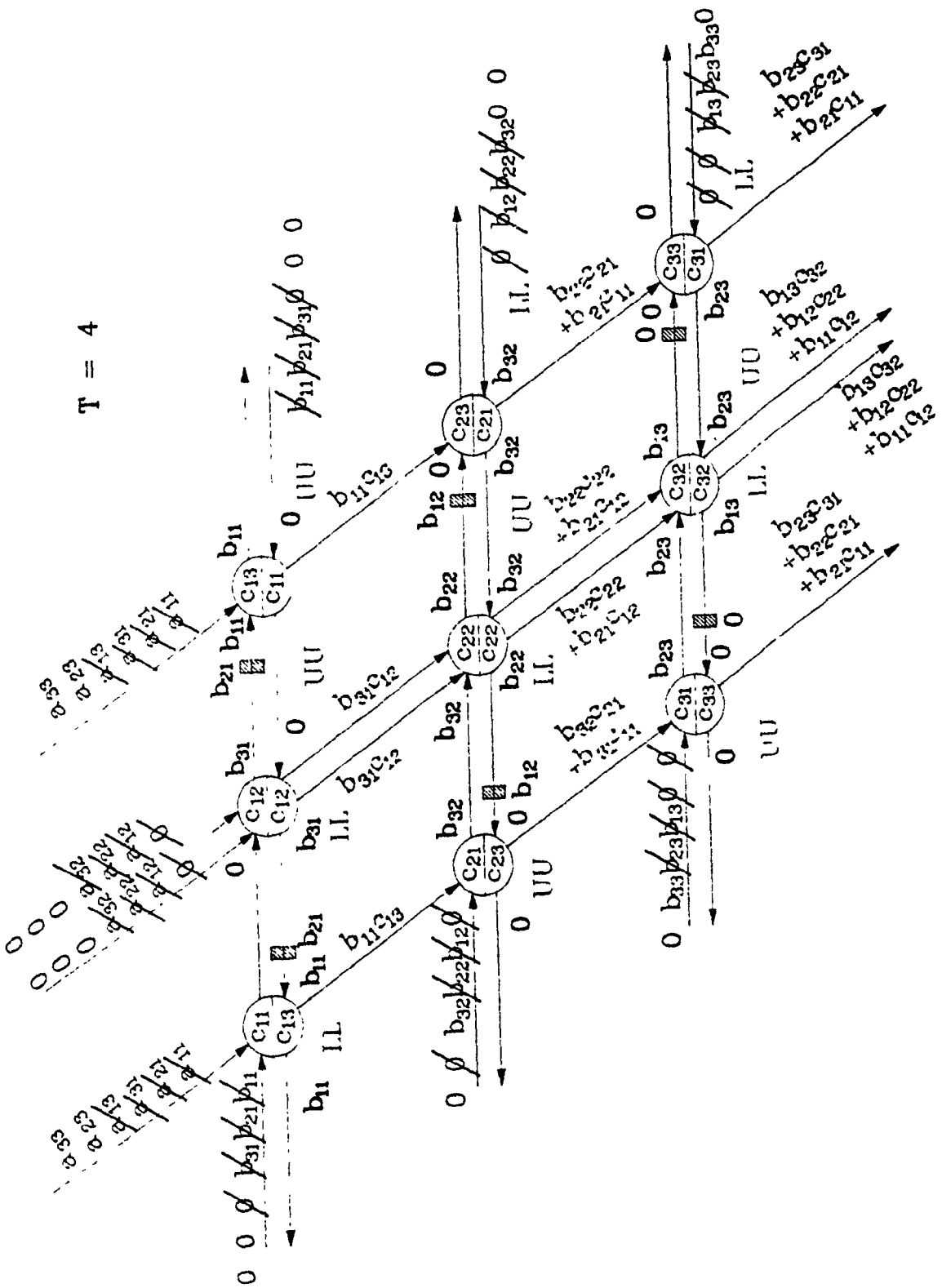


Figure 5.5(a) (iv) The fourth pulsation of the CED systolic array for matrix multiplication (for N=3).

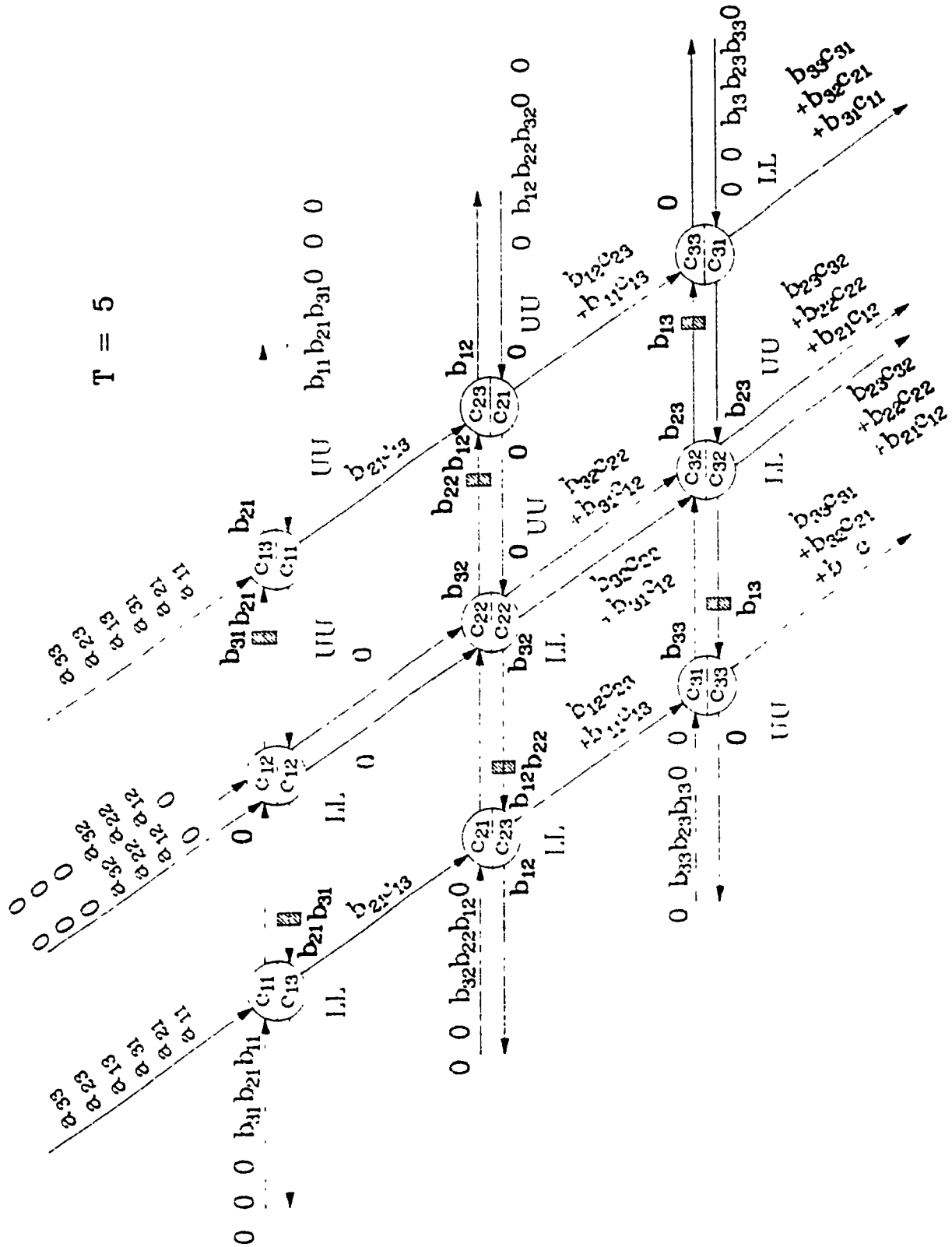


Figure 5.5(a) (v) The fifth pulsation of the CED systolic array for matrix multiplication (for N=3).

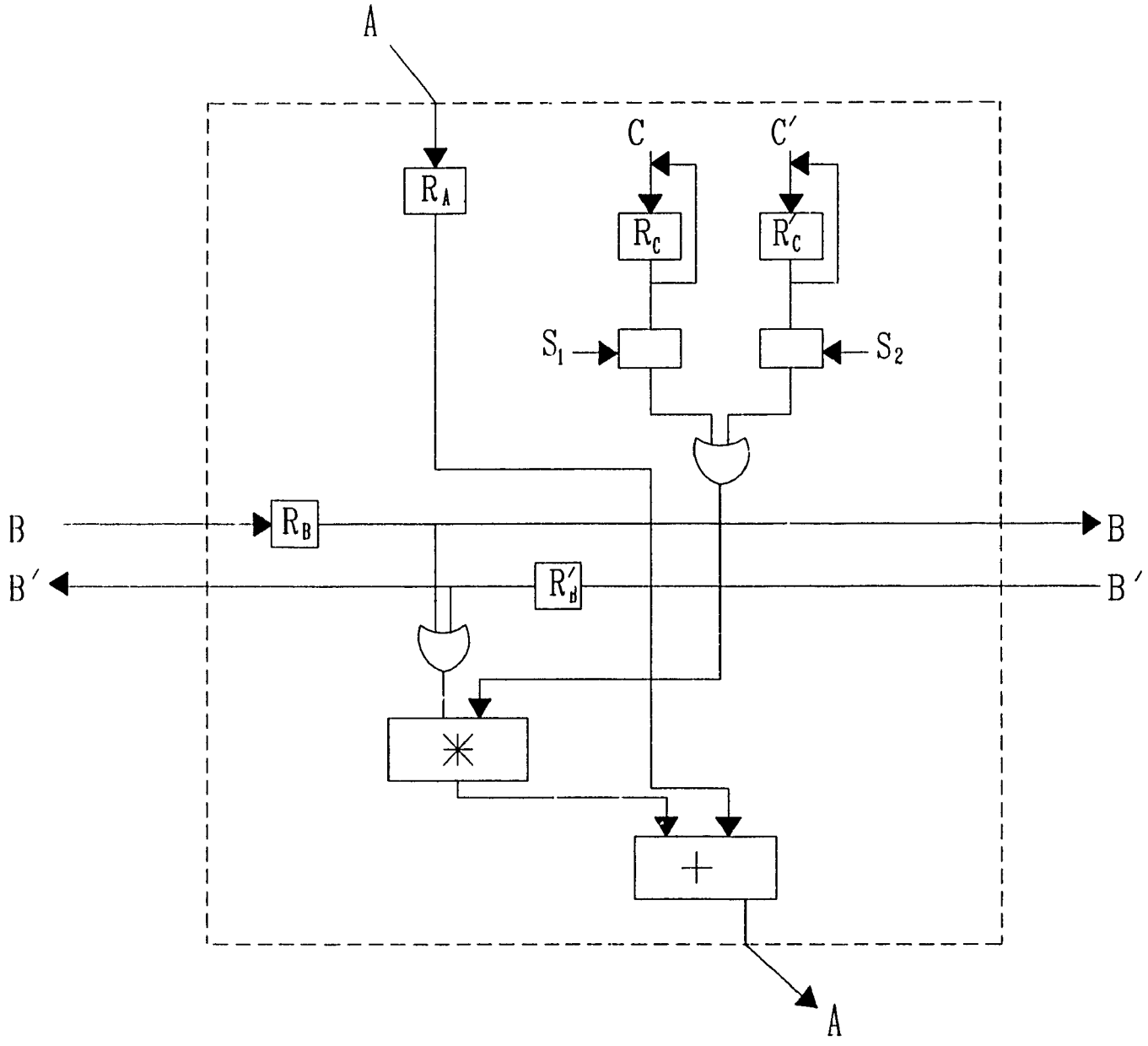


Figure 5.6(a) The cell structure of type I cell in Figure 5.5.

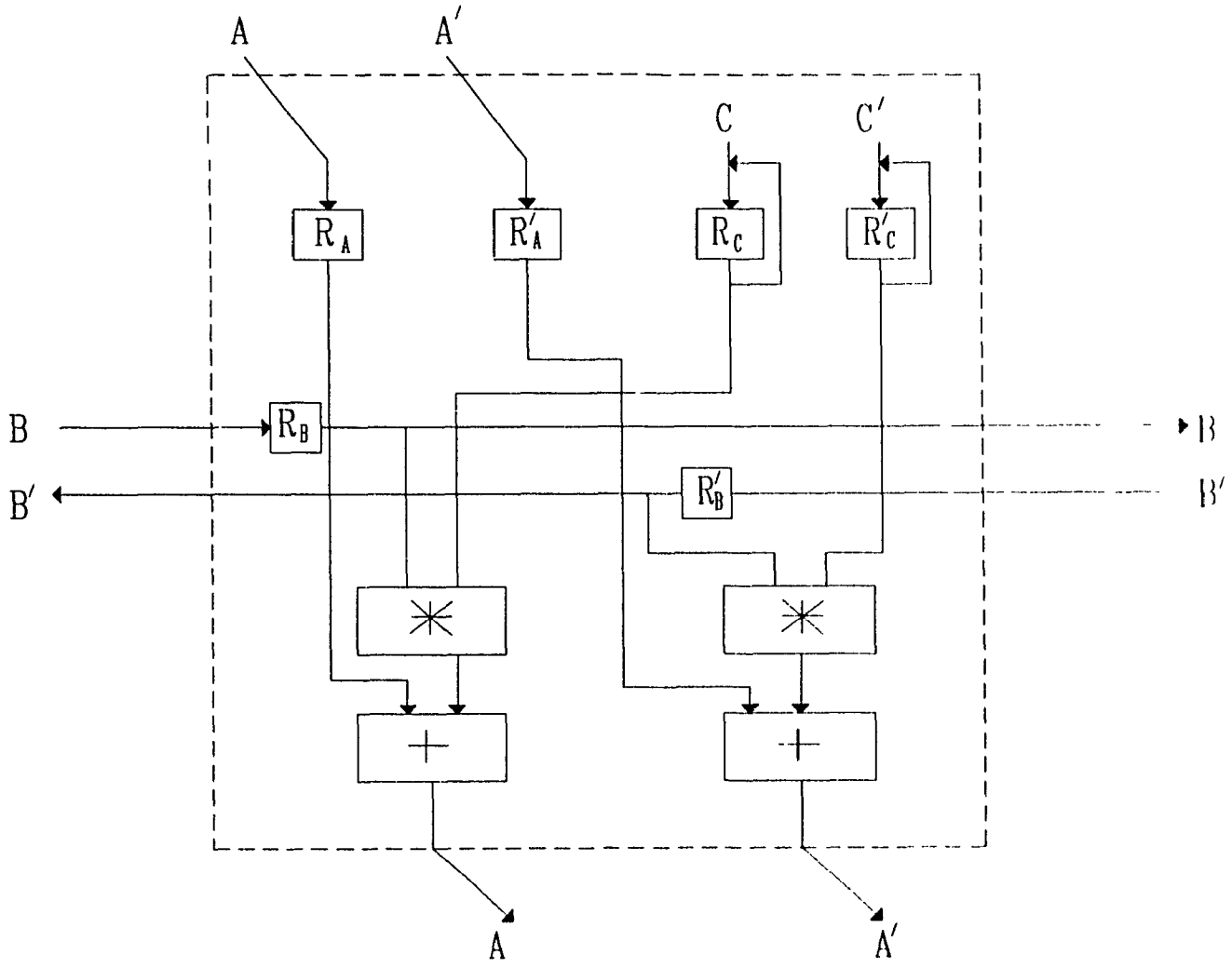


Figure 5.6(b) The cell structure of type II cell in Figure 5.5.

B Example 2

The purpose of example 2 is to show that CED can be obtained almost at no area or time overhead in some cases (usually when N is even). Figure 5.7 consists of the mapping of the index set into VLSI arrays using the transformations T and T' for N=4 .

Figure 5.8 shows the CED systolic array implementation of the matrix multiplication algorithm for N=4. All the cells in the array are identical. Since no two identical computations need to be computed using the same set of cells, hence there is no duplication of any functional blocks in the cells. The only added hardware are the latches (three in each interconnection line for variable B) which are used to synchronize the propagation of the corresponding data elements for each computation, into the systolic array. As in Fig.5.5, the input data for variable A are re-scheduled to satisfy the original input data scheduling for this variable. The corresponding output results appear at the output of the array at the same time and these are compared for discrepancies using the TSC error detecting circuits. The purpose of comparing Figs.5.5 and 5.8 is to point out the differences between the two CED systolic arrays. For instance, in Fig.5.5, the functional blocks of some of the cells are duplicated because they are required to compute the identical output results of the two independent computations. Consequently, Fig.5.5 consists of two types of cell structure. However in Fig.5.8, the functional blocks of the cells are not duplicated and hence all the cells are identical and thus, CED is achieved almost at no area overhead, but with extra delay (time overhead).

i	j	k	$\hat{i}$ time	$\hat{j}$ processor	$\hat{k}$	$\hat{j}'$ processor	$\hat{k}'$
1	1	1	3	2	1	5	1
1	1	2	4	3	2	6	2
1	1	3	5	4	3	7	3
1	1	4	6	5	4	8	4
1	2	1	4	3	1	4	1
1	2	2	5	4	2	5	2
1	2	3	6	5	3	6	3
1	2	4	7	6	4	7	4
1	3	1	5	4	1	3	1
1	3	2	6	5	2	4	2
1	3	3	7	6	3	5	3
1	3	4	8	7	4	6	4
1	4	1	6	5	1	2	1
1	4	2	7	6	2	3	2
1	4	3	8	7	3	4	3
1	4	4	9	8	4	5	4
2	1	1	4	2	1	5	1
2	1	2	5	3	2	6	2
2	1	3	6	4	3	7	3
2	1	4	7	5	4	8	4
2	2	1	5	3	1	4	1
2	2	2	6	4	2	5	2
2	2	3	7	5	3	6	3
2	2	4	8	6	4	7	4
2	3	1	6	4	1	3	1
2	3	2	7	5	2	4	2
2	3	3	8	6	3	5	3
2	3	4	9	7	4	6	4
2	4	1	7	5	1	2	1
2	4	2	8	6	2	3	2
2	4	3	9	7	3	4	3
2	4	4	10	8	4	5	4
3	1	1	5	2	1	5	1
3	1	2	6	3	2	6	2
3	1	3	7	4	3	7	3
3	1	4	8	5	4	8	4
3	2	1	6	3	1	4	1
3	2	2	7	4	2	5	2
3	2	3	8	5	3	6	3
3	2	4	9	6	4	7	4
3	3	1	7	4	1	3	1
3	3	2	8	5	2	4	2
3	3	3	9	6	3	5	3
3	3	4	10	7	4	6	4
3	4	1	8	5	1	2	1
3	4	2	9	6	2	3	2
3	4	3	10	7	3	4	3
3	4	4	11	8	4	5	4
4	1	1	6	2	1	5	1
4	1	2	7	3	2	6	2
4	1	3	8	4	3	7	3

4	1	4	9	5	4	8	4
4	2	1	7	3	1	4	1
4	2	2	8	4	2	5	2
4	2	3	9	5	3	6	3
4	2	4	10	6	4	7	4
4	3	1	8	4	1	3	1
4	3	2	9	5	2	4	2
4	3	3	10	6	3	5	3
4	3	4	11	7	4	6	4
4	4	1	9	5	1	2	1
4	4	2	10	6	2	3	2
4	4	3	11	7	3	4	3
4	4	4	12	8	4	5	4

Fig.5.7 : Mapping of index set into VLSI arrays using transformation matrices T and T' (for N=4).

C Example 3

The purpose of this example is to show the generality of the proposed method. It is also applicable to arrays where all data variables keep moving from one cell to another, which is a different case to that given in example 1. In example 1, the data for one of the variables are stored in the cells. Also, in the above examples, the TDM's of the CED systolic array are arbitrarily selected. It is not certain if these TDM's are the optimum ones. We will take a different example where all data variables are flowing from one cell to another. Furthermore, we will use an optimum systolic array for this particular computation. An approach for designing optimal systolic architectures has been proposed in chapter III [27]. Using the results in chapter III, the TDM with the best overall array performance given the performance index, has been obtained for the matrix multiplication algorithm, whose dependency matrix is given in Eq.(2.6). The TDM is given as follows [27]:

$$\Delta = \begin{bmatrix} -1 & -1 & -2 \\ 0 & -1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{5.18}$$

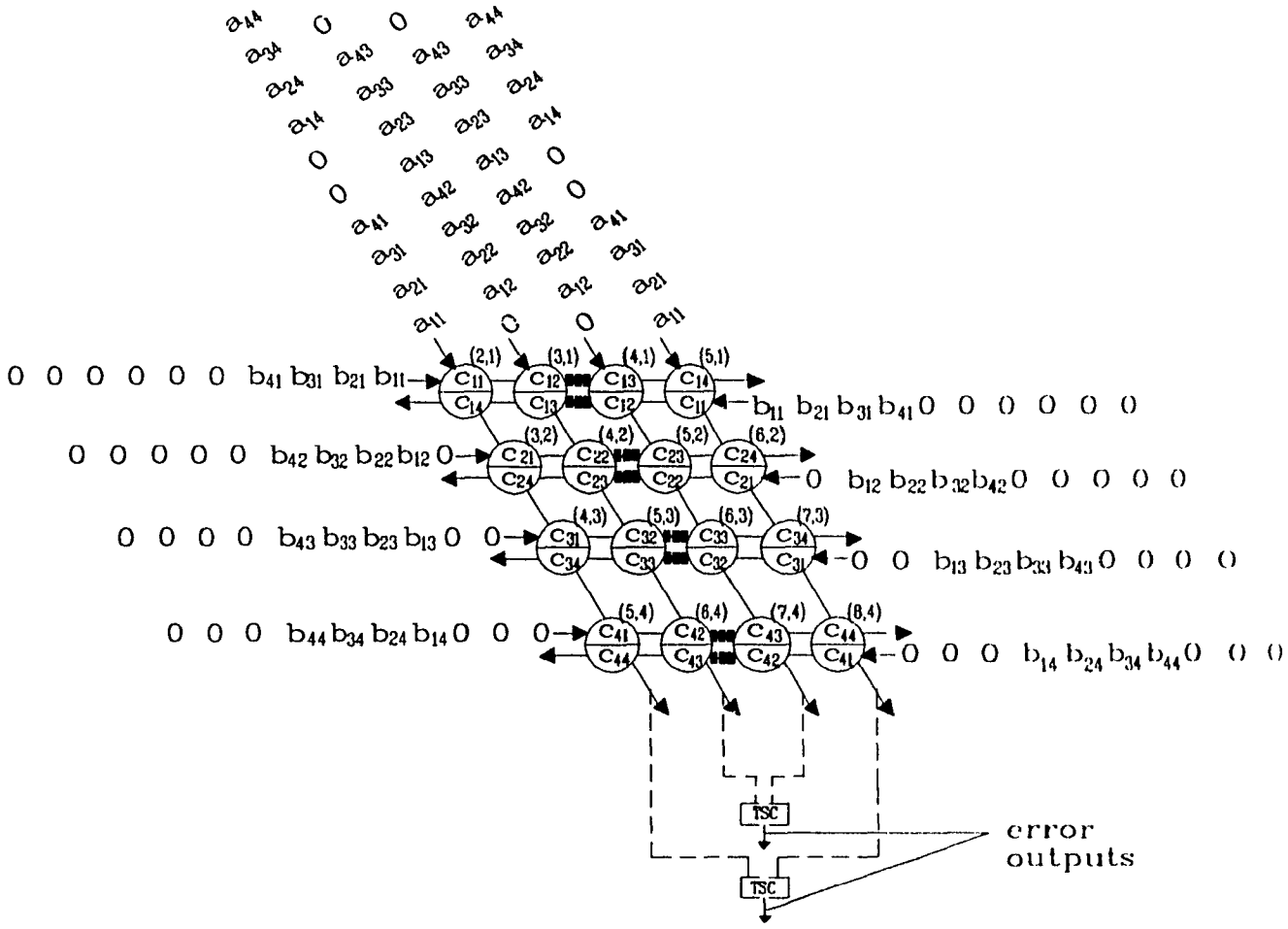


Figure 5.8 CED systolic array structure for matrix multiplication (for N=4).

The mapping of the index set (i,j,k) into the new index set $(\hat{i},\hat{j},\hat{k})$ using the transformation T (Eq.(5.18)) is given by

$$\begin{bmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} 2i + j + k \\ j \\ i + k \end{bmatrix} \tag{5.19}$$

The VLSI array structure corresponding to the transformed dependency matrix Δ of Eq.(5.18) is shown in Fig.5.9 (for N=3). The data variables A and C propagate from a cell to the next via vertical channels while the data variable B moves from cell to cell via a horizontal channel. The structure of the cell is shown in Fig.5.10. It consists of an adder, a multiplier, registers and one delay element in the path of variable C. In order to obtain the second version of the algorithm, the TDM of Eq.(5.18) is rotated by 180 degrees about the vertical-axis. The transformation matrix (T') and the TDM (Δ') for the second version, in this case, are given as follows:

$$T' = \begin{bmatrix} 2 & 1 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \quad \text{and,} \quad \Delta' = \Delta_{rot-vert} = T'D = \begin{bmatrix} -1 & -1 & -2 \\ 0 & 1 & 0 \\ -1 & 0 & -1 \end{bmatrix} \tag{5.20}$$

The new index set $(\hat{i}',\hat{j}',\hat{k}')$ is given by

$$\begin{bmatrix} \hat{i}' \\ \hat{j}' \\ \hat{k}' \end{bmatrix} = \begin{bmatrix} 2i + j + k \\ -j + C \\ i + k \end{bmatrix} \tag{5.21}$$

Figure 5.11 shows the mapping of the index set into VLSI arrays using the transformations T (Eq.(5.18)) and T' (Eq.(5.20)) (for N=3). Here also, the value of C is 4 and $\hat{i}' = \hat{i}$.

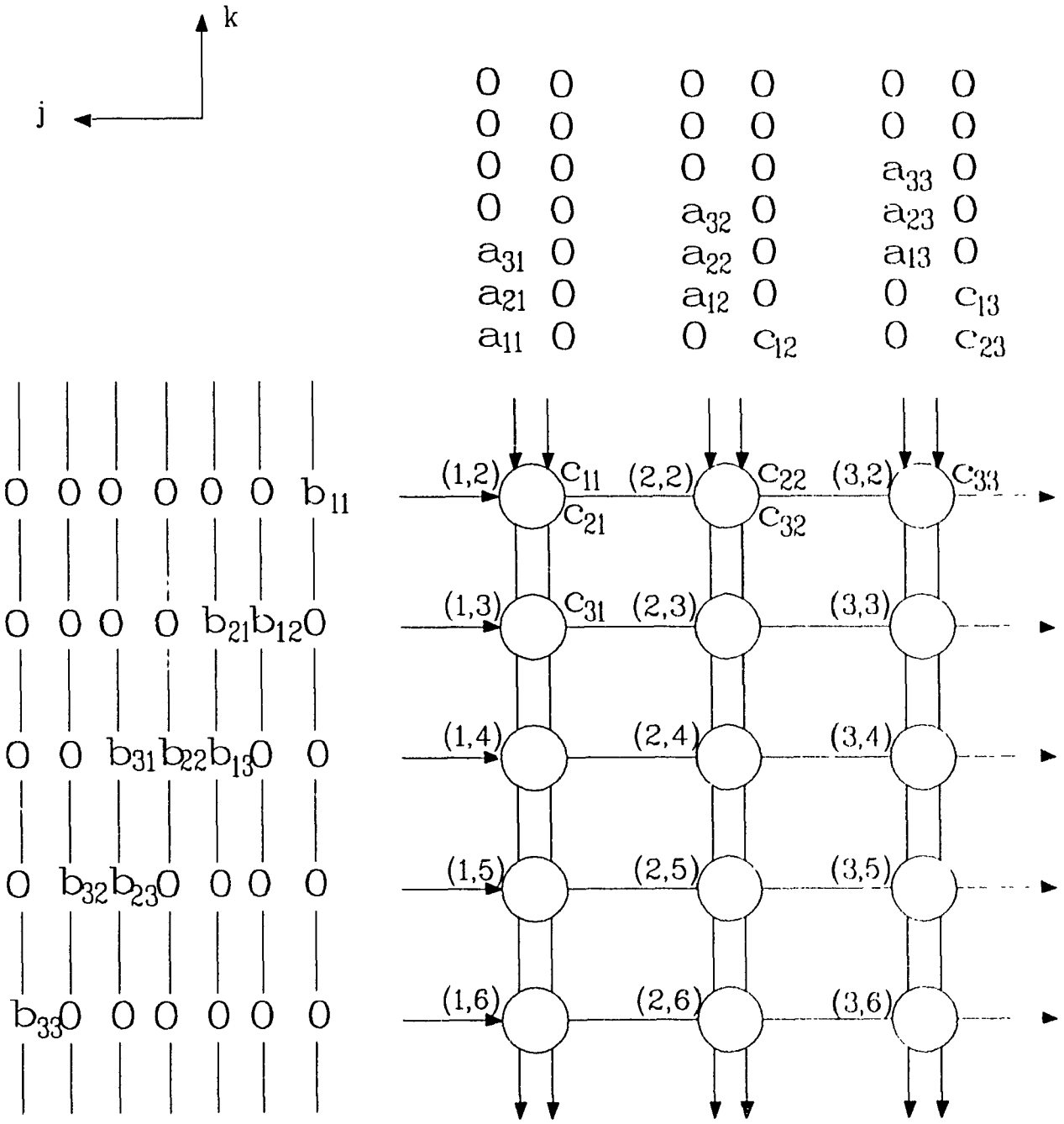


Figure 5.9 Optimal VLSI array structure for matrix multiplication (for N=3).

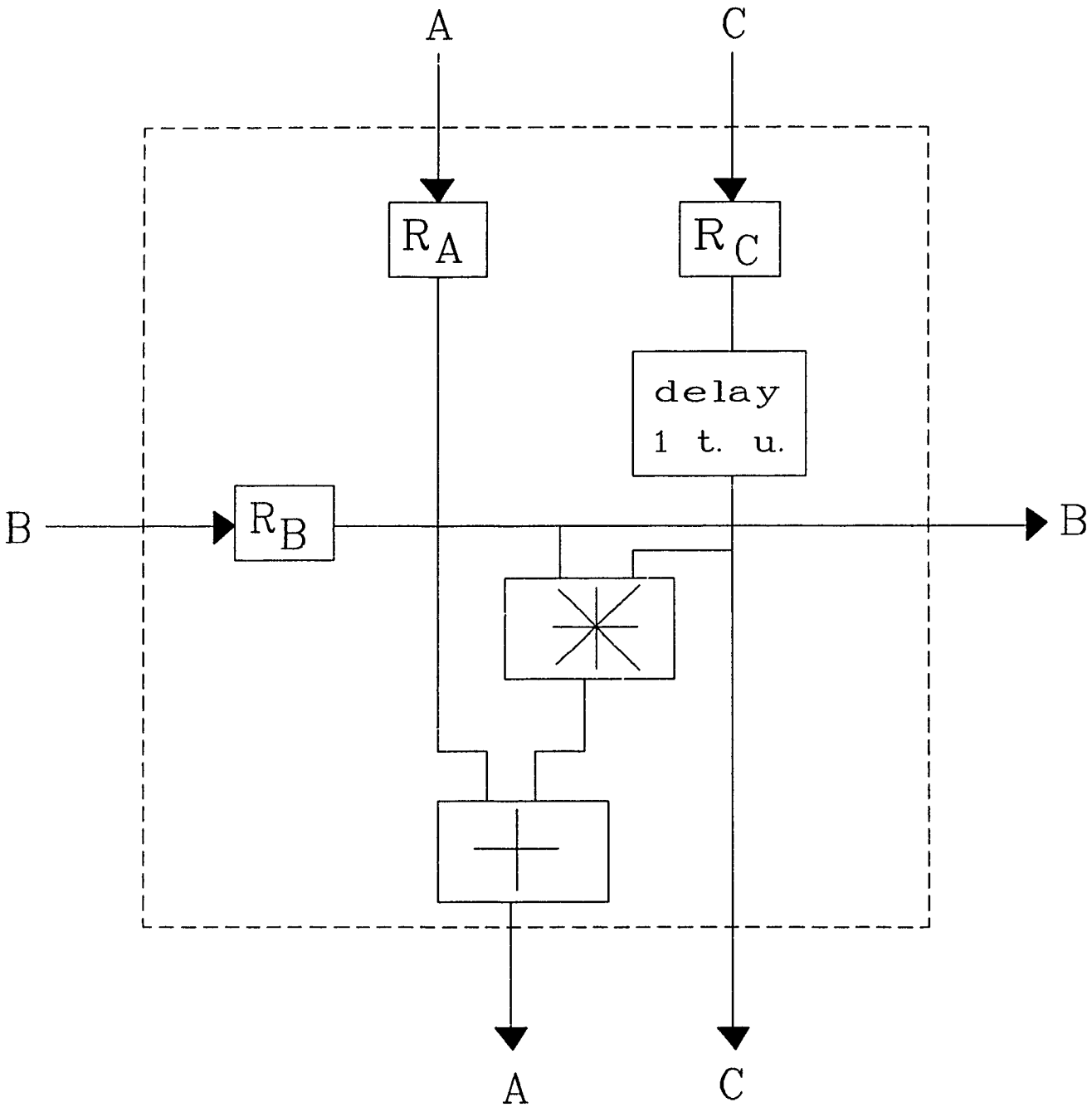


Figure 5.10 The structure of the cell in Figure 5.9.

i	j	k	$\hat{i}$ time	$\hat{j}$ processor	$\hat{k}$	$\hat{j}'$ processor	$\hat{k}'$
1	1	1	4	1	2	3	2
1	1	2	5	1	3	3	3
1	1	3	6	1	4	3	4
1	2	1	5	2	2	2	2
1	2	2	6	2	3	2	3
1	2	3	7	2	4	2	4
1	3	1	6	3	2	1	2
1	3	2	7	3	3	1	3
1	3	3	8	3	4	1	4
2	1	1	6	1	3	3	3
2	1	2	7	1	4	3	4
2	1	3	8	1	5	3	5
2	2	1	7	2	3	2	3
2	2	2	8	2	4	2	4
2	2	3	9	2	5	2	5
2	3	1	8	3	3	1	3
2	3	2	9	3	4	1	4
2	3	3	10	3	5	1	5
3	1	1	8	1	4	3	4
3	1	2	9	1	5	3	5
3	1	3	10	1	6	3	6
3	2	1	9	2	4	2	4
3	2	2	10	2	5	2	5
3	2	3	11	2	6	2	6
3	3	1	10	3	4	1	4
3	3	2	11	3	5	1	5
3	3	3	12	3	6	1	6

Fig.5.11 : Mapping of index set into VLSI arrays using transformation matrices T (Eq.(5.18)) and T' (Eq.(5.20)) (for N=3).

By merging the resultant systolic array of the second version of the algorithm with that in Fig.5.9, the CED systolic array shown in Fig.5.12 is constructed. The systolic array of Fig.5.12 has been modified to reflect the given fault detection requirements. The array consists of two types of cell structures as in Fig.5.5. In the first type which constitutes of all the cells in the array except cells (2,2), (2,3), (2,4), (2,5) and (2,6), redundant hardware are not added in the cells, while in the second type redundant functional blocks are introduced by duplicating these blocks. The input data for variables A and C are rescheduled so as to satisfy the original data scheduling of these variables. The interconnection lines for the data variable B are also duplicated and a delay element is added in each line in order to synchronize the data propagation of the independent computations.

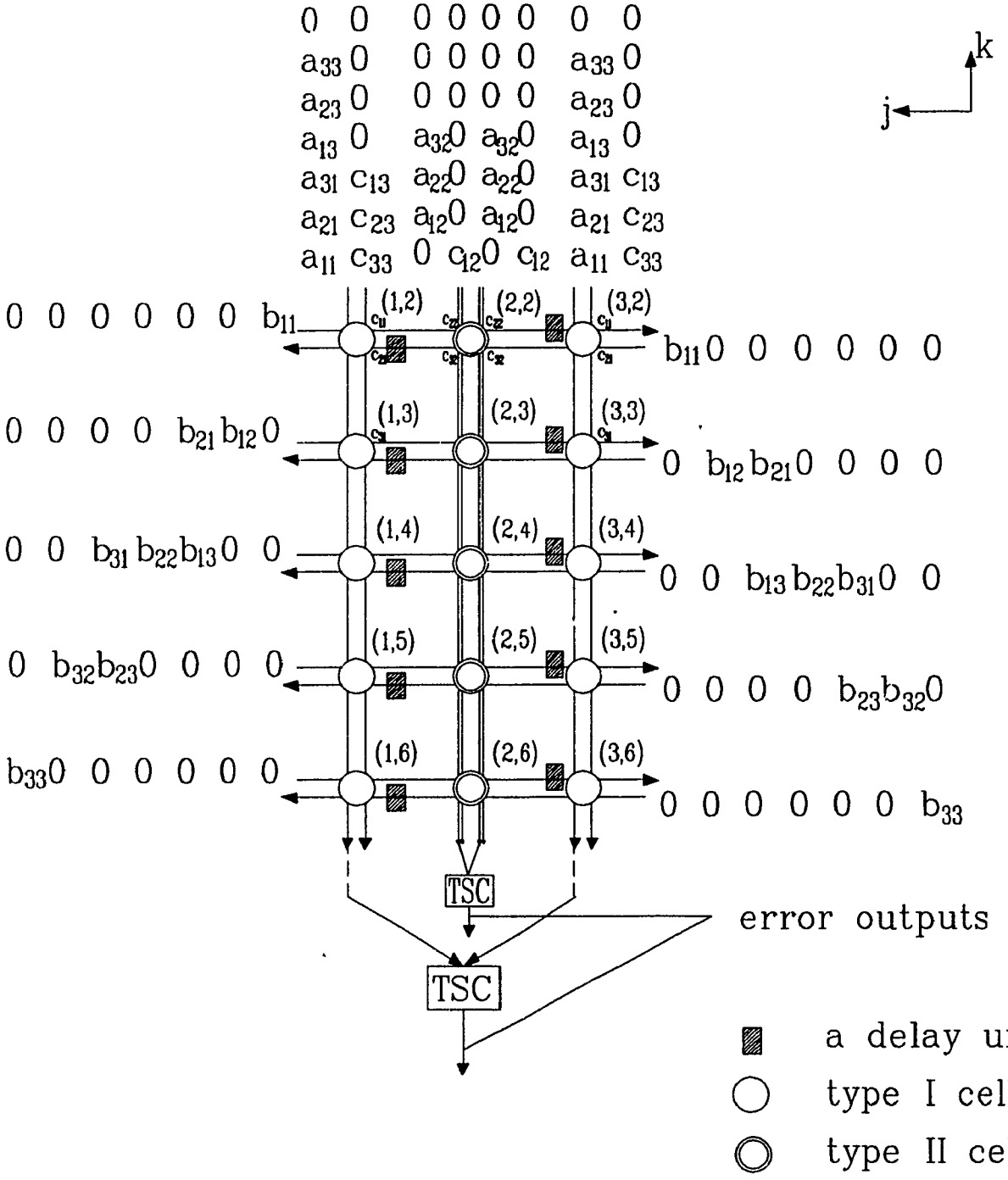


Figure 5.12 Optimal CED systolic array structure for matrix mutiplication (for N=3).

As seen from Fig.5.12, the corresponding output results of the two versions of the algorithm arrive at the output of the array at the same time and these are compared for discrepancies using TSC circuits.

5.2.4 Procedure for Designing Area Efficient CED Systolic Architectures Using The Scheme Proposed in This Chapter

In this subsection, we present the following five-step procedure to design a CED systolic array architecture using our scheme :

1. The TDM of a given algorithm is determined and then mapped into a systolic architecture.
2. A second version of the algorithm is determined by rotating the systolic array in step 1 by 180 degrees about any of the principal axis (horizontal, vertical or diagonal axis).
3. The two systolic arrays are merged to construct the CED systolic array.
4. Add extra hardware only in those cells of the CED array that would be computing the corresponding output results for the two independent computations.
5. The input data of some of the variables in the algorithm are re-scheduled to satisfy their original input data scheduling. Extra time is required for the completion of the computations and this corresponds to the maximum number of extra delay elements introduced in the interconnection path of any of the data variables.

5.3 ANALYSIS OF THE FAULT COVERAGE OF THE PROPOSED SCHEME

In this section, we analyze the fault coverage of the proposed scheme. Fault coverage can be defined as the number of detected faulty outputs given the number of faulty outputs.

For the proposed scheme, a functional fault model has been assumed. Therefore, our interest lies basically in detecting a corrupt output, to ensure the validity of the final result. A fault that does not produce an error, or in other words, does not affect the functionality of the module under test, will not be detected. The cells used to compute the various outputs a_{ij} for the two independent computations of the matrix multiplication algorithm are shown in Figs.5.5, 5.8 and 5.12. For instance, in Fig.5.5, the output result of one of the a_{11} is calculated in cells (2,1), (3,2) and (4,3) while the alternate result is calculated in cells (4,1), (5,2) and (6,3). A discrepancy in the two results indicates a fault or malfunction (temporary or permanent) either in any of the functional blocks of any of these cells or in any of the interconnection lines in the systolic array or in any of the added delay elements. Similarly, the results of, for instance, a_{12} are calculated with the duplicated functional blocks in cells (3,1), (4,2) and (5,3). In this case also, a fault either in any of the functional blocks in these cells or in any interconnection line in the array will produce a discrepancy in the two answers. Since the error detecting circuits are totally self-checking, hence any single fault in the CED systolic array or a faulty component in the error detecting logic itself would be detected. Therefore all single faults in the systolic array of Fig.5.5 are detectable.

It can also be shown that all single faults in the CED systolic array of Fig.5.8 can be detected. Consider a single functional module failure of, for instance, the multiplier in cell (3,2). This fault will cause the partial results of a_{i1} and a_{i4} (i=1,2,..,4) computed in the cell to be erroneous. Since all the partial results of these outputs are not computed by only one cell, therefore, the error propagates to other cells until it reaches the output. The erroneous results computed in this cell will corrupt the other partial results computed in cells (4,3) and (5,4). Consequently, the final results of the outputs a_{i1} and a_{i4} will be erroneous. The redundant results of these outputs are computed in cells (5,1), (6,2) (7,3) and (8,4), which are fault-free, since single fault assumption is made. The redundant output results will be fault-free, and a comparison of the original output results with their

redundant counterparts will produce a discrepancy. Hence, this fault is detectable.

Furthermore, let us consider a single failure of register B in cell (4,3) of Fig.5.8. Since the data for variable B is a transmittent data, it percolates from cell to cell. This fault will corrupt the output results of only one version of a_{ij}, because, the output results of the other version of the computation (redundant), will be computed using the data for register B'. Consequently, one set of computations will be fault-free while the other will be erroneous. Hence, by comparing the corresponding output results, the fault in register B will be detected.

In Figs. 5.5 and 5.8, a single fault in either register A, register C, register C', TSC circuit, an adder or a multiplier, in any of the cells, will manifest itself as an error in some output results of one version of the algorithm. In other words, all the outputs a_{ij} of one independent computation will not be affected by the occurrence of any of these faults. The consequence is that, these faults can be detected by any of the TSC circuits, but not both. That is, the error indication output of only one TSC circuit will be activated in the presence of any of these faults. On the other hand, a fault in either register B, register B' or any interconnection line in the array, will cause all the output results of one independent computation to be erroneous. The occurrence of any of these faults will activate the outputs of both TSC circuits, and hence, will be detected.

The significance of this is that, from the state of the outputs of the TSC circuits, we can determine the nature of the fault present in the array. We can know if the fault is from any of the two categories discussed above. Since the effects of the faults are detected at the output of the array (global error detection), we have no means of locating the faulty cells. Therefore, in the proposed design scheme, the error indicator will alert us when a fault is present. This will give us an indication of the category of the fault, but, the location of the exact module that has failed, is not possible.

Similar analysis is also applicable to the CED systolic array of Fig. 5.12.

So far, we have considered the results of single faults. A number of multiple fault patterns can also be detected by our design scheme, provided that the corresponding output results of the two independent computations are not affected by the multiple fault pattern. For example, a multiple fault pattern which comprises of single faults in cells (3,1), (4,2), (5,3), (4,1), (5,2) and (6,3) in Fig.5.5, produces a detectable error pattern when the faulty result stream that exits from cell (6,3) is compared with the fault-free stream exiting from cell (4,3). Similarly, in Fig.5.8, an example of a detectable multiple fault pattern consists of single faults in cells (2,1), (3,2), (4,3) (5,4), (4,1), (5,2), (6,3) and (7,4).

Furthermore, let's consider the effects of a multiple fault pattern which is a combination of all single failures of register B in every cell in the array. This multiple fault pattern will cause all the output results of one version of a_{ij} to be erroneous. Consequently, the proposed design scheme will detect this particular multiple fault pattern. Several other multiple fault patterns can be detected in a similar fashion as described above, provided that the output results of two corresponding independent computations are not affected by the fault pattern. It is important to note that a multiple fault pattern that affects two corresponding output results may cause both results to have the same erroneous values. In this case, the error detecting circuit would not indicate a discrepancy in the two answers and hence, such multiple fault patterns cannot be detected. However, the probability of occurrence of such multiple fault patterns is very low.

Also, in addition to permanent faults, intermittent errors or temporary faults can be detected in the CED systolic arrays of Figs. 5.5, 5.8 and 5.12, by comparing the corresponding output results of the two independent computations. They are distinguishable from permanent errors since they are temporary. It is known [17] that intermittent faults missed during manufacturing testing may be caught by concurrent testing during operation. For this reason, concurrent testing is increasingly being regarded as complementary to manufacturing tests, especially in systems that require high reliability.

In the following section, we will determine the cost of our proposed design scheme,

in terms of the area and time overhead required to achieve concurrent error detection.

5.4 AREA AND TIME OVERHEAD OF THIS SCHEME

In VLSI system design, there is always a trade-off involved between the silicon area and the desired efficiency in terms of throughput and speed. Our motivation is to propose a scheme which is applicable to different systolic array implementations without causing any loss in throughput. This goal is achieved by our scheme but not without some overhead in silicon area and time.

The silicon overhead includes the additional functional blocks introduced only in those cells that are required to compute the redundant output results. It also includes the delay elements used to synchronize the flow of input data into the CED systolic array and the error detection circuitry required at the output of the array. As regards to the time, since the two independent computations are launched into the CED systolic array at the same time, there comes a time when a computational resource would be requested by the two different computations at the same time. In order to resolve this conflict, extra time is introduced by delaying one computation (using the delay elements) until the other has completely utilized the computational resource. Therefore in this respect, we can say that our scheme involves overhead in time. For instance, in Fig.5.5, one extra clock cycle is required to complete the two independent computations. The number of the extra clock cycles required corresponds to the maximum number of the delay elements introduced in the path of the interconnection line of any of the variables in the algorithm. This is as exemplified in Fig.5.5 where an extra delay element is introduced in the path of variable B to synchronize the propagation of data into the CED array.

In Fig.5.5, by observing the scheduling of the input data for variable A, it appears that one computation is launched after the other. Consequently, at most twice the number of clock cycles would be required to complete the computation of the two independent

results. The systolic array of Fig.5.5 requires 7 clock cycles to complete the computation of one version of the algorithm. From the nature of the data scheduling in Fig.5.5, a total of 10 clock cycles would be required to complete the computation of the two independent output results. However, the CED systolic array of Fig.5.5 requires only 8 clock cycles to complete the two computations. Hence, given the nature of the data flow into the CED systolic array after re-scheduling the input data, we can conclude that our scheme does not involve any overhead in time. There is , of course, no loss in throughput. One of the advantages of our scheme is that though the input data is re-scheduled, the corresponding output results of the two computations arrive at the output of the array at the same time. This facilitates the comparison of the two answers without the introduction of any additional control or synchronization circuitry.

Similar analysis and argument are applicable to the CED systolic array of Fig. 5.12.

In Fig.5.8, the silicon overhead includes 24 delay elements, two error detection circuitry (TSC) required at the output of the array and four interconnection lines. Unlike in Fig.5.5, no additional functional blocks are introduced in the cells of Fig.5.8. This is because, all the elements of the generated variable (matrix A), for the two independent computations, are computed at different times, using different cells in the array. Considering that the area of a delay element and that of a TSC circuit are much smaller than the area of a processing element, therefore, when N is even, concurrent error detection is achieved almost at no hardware cost. Three extra delay elements are introduced in the original and redundant data paths of variable B. They are used to synchronize the propagation of data into the CED array. This means that, three extra clock cycles (time overhead) are required to complete the computation of the two independent results. In order to compute the results of one version of the algorithm, the systolic array of Fig.5.8 requires 10 clock cycles to do so. With the incorporation of the CED scheme, a total of 13 clock cycles is required.

It is important to note that the distribution and assignment of the delays depend on

the array size and the algorithm under execution. For instance, in the case of the matrix multiplication algorithm, if N is odd, extra functional blocks and delay elements are introduced in the array. For the CED systolic structure shown in Fig.5.5, the cell location in each row, where additional functional blocks are introduced, is given by ((N-1)/2 + 1). Hence, in Fig.5.5, for N=3, extra functional blocks should be added in cells (3,1), (4,2) and (5,3). If for example, N=7, the extra functional blocks should be introduced in cells (5,1), (6,2), (7,3), (8,4), (9,5), (10,6) and (11,7). Thus, in general, the allocation of the extra functional blocks in the array can be accomplished in this manner, and this allocation criterion exists only if N is odd. For the delay assignment, the delay elements are located in both directions, at the interconnection lines after the cell ((N-1)/2 + 1). Hence, in Fig.5.5, if N=3, the delay elements are located in one direction after cells (3,1), (4,2) and (5,3), in the opposite direction after the same set of cells. If N=7, the delay elements will be located in both directions after cells (5,1), (6,2), (7,3), (8,4), (9,5) (10,6) and (11,7).

The number of delay elements assigned to these locations is (N-2). That is, if N=3, only one extra delay element is introduced in each direction as shown in Fig.5.5. For N=7, a total of five delay elements are introduced at these locations. Therefore, if N is odd, the time overhead required by the proposed CED scheme is (N-2) [{N + (N-1)/2 - [(N-1)/2 + 1] - 1} = (N-2)] .

In the case when N is even, no extra functional blocks need to be introduced in the systolic array. Only delay elements are assigned to some locations in the array. The delay elements are located in both directions, at the interconnection lines after the cell (N/2). The number of the delay elements assigned to these locations is given as [{N + N/2 - [N/2 + 1]} = N-1]. Thus, if N is even, the time overhead required to complete the two independent computations is (N-1) clock cycles.

In addition to the time overhead for a single instance of the matrix multiplication problem, there is the possibility of an extra time redundancy for many successive matrix

multiplications. For instance, in Fig.5.8, (without the CED scheme) if a problem instance is initiated after the other, then the second problem instance could start at time t=5 (after token b_{41} is loaded into the array). The third problem instance could then start at time t=9. Since two problem instances are required for error detection, this means that, the second error diagnosis problem instance, has to wait until t=9 to be initiated. Similarly, the third one has to wait until t=17 to start. However,if the proposed CED scheme is incorporated, as shown in Fig.5.8, a new problem instance has to wait until t=11 cycles to be initiated (after a_{44} is loaded into the array). The next one will be initiated at t=21. The block pipelining period for many problem instances is increased in this case.

In the following section, we will compare the merits of the scheme proposed in this chapter against other CED schemes proposed in the literature.

5.5 COMPARISON OF OUR CED SCHEME WITH OTHER CED SCHEMES

Several CED schemes have been proposed which achieve fault detection by duplication of operation, either of individual cells or the whole array. In this section, we compare the proposed CED scheme with other schemes proposed in the literature [6,12-24]. Table 5.1 shows the comparison of the complexity and performance of the existing schemes against our design methodology.

The CED method (CCRC) proposed by Gulati and Reddy [12] requires a silicon overhead of one additional cell in every row of a systolic array, multiplexer, links for error detection and propagation, and the error detection logic in each cell. The scheme is limited to only those systolic implementations in which the data as well as (sub) results keep moving. It is only applicable to structures like the bidirectional and hex connected systolic arrays where inputs are required to be applied in alternate cycles by the

Category	Gulati & Reddy [12]	C-C. Wu & T-S. Wu	S.Y. Kung & Manolakos [14,15]	J.A. Abraham et al. [2]
CED Technique	Comparison with Concurrent Redundant Computation (CCRC)	CCRC in unidirectional data flow systolic arrays.	Time Redundancy with interleaving for fault-tolerance (TRIFT).	CED using Linear Property Encoding
Complexity without CED techniques	Processor: $(2N-1)^2$ Time: $3N-1$	Processor: N^2 Time: $3N-2$	Processor: N^2 Time: $2N-1$	Processor: N^2 Time: N
Redundancy required with the technique	Processor: N Time: N_2 Register: N^2 TSC: $2N^2$ MUX: N Other: links for error detection and propagation.	Processor: N Time: N_2 Register: N^2 MUX: N TSC: $2N^2$ Other: links for error detection and propagation.	Processor: N Time: N_2^1 Register: N^2 TSC: $2N^2$	Processor: N Time: 1 Other: Adders: $2(N+1)^2$
Redundancy Ratio	Processor: $O(N/(2N-1)^2)$ Time: $O(N/(3N-1))$	Processor: $O(N/N^2)$ Time: $O(N/(3N-2))$	Processor: $O(N/N^2)$ Time: $O((N+1)/(2N-1))$	Processor: $O(1/N)$ Time: $O(1/N)$
Diagnosis Performance	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - location: Yes	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - location: Yes	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - location: Yes	Faults - types: Transient & Permanent - # allowed: Single fault & certain multiple fault patterns - location: No
Utilization	On-line error detection.	On-line local error detection.	On-line local error detection.	On-line global error detection

Table 5.1 Comparison of the complexity and diagnosis performance of the various existing CED techniques with our proposed design scheme.

...1/4

Category	Huang & Abraham [12]	Cosentino [17]	Cosentino [18]	Ari & Friedlander
CED Technique	Checksum Encoding Scheme	Dual Redundancy (Duplication)	Residue Number System (RNS) Approach	Algorithm-based Error Detection Approach
Complexity without CED techniques	Processor: $W_1 * W_2$ Time: $N+\min(W_1, W_2)$	Processor: N^2 Time: $3N-1$	Processor: $3N^2$ Time: $2N_2$ Delay Units: $2N^2-N$ Other: Binary-to-Residue converter Residue-to-Binary converter	Processor: $N(2N+1)$ Time: $4N+2$
Redundancy required with the technique	Processor: W_1+W_2 Time: $2*r*\log_2(N)/N$ Adder: $3(W_1+W_2) - 4$ Buffer: $s\log_2(W_1) + \log_2(W_2) + 2\min(W_1, W_2) + 2$ TSC comparator: 2 -W_1 and W_2 are the band-widths of N*N band matrices. -r is the ratio of execution time of an addition to that of a multiplication performed in a processor array. -Assume that $O(W_1) = O(W_2)$	Processor: N (host processor) Time: 2N Accumulator register: N^2+N-1 Monitoring circuit: N Memory: Yes	Processor: N^2 Time: T_{r-b-c} Other: Tsc: 2N Where T_{r-b-c} is the time for the residue-to-binary conversion.	Processor: P Time: T_p P and T_p are respectively the additional hardware and time redundancies required to implement the diagnostic expression.
Redundancy Ratio	Processor: $O(1/W_2)$ Time: $2*r*\log_2(N)/N$	Processor: $O(1/N)$ Time: $O(2N/(3N-1))$	Processor: $N^2/3N^2$ Time: $(T_{r-b-c})/2N$	Processor: $P/N(2N+1)$ Time: $T_p/(4N+2)$
Diagnosis Performance	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - location: Yes	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - location: Possible	Faults - types: Permanent & Transient - # allowed: Single fault & certain multiple fault patterns - location: No	Faults - types: Permanent & Temporary - # allowed: Single & certain multiple PE failures - location: Yes
Utilization	On-line global error detection.	On-line global error detection.	On-line global error detection.	On-line global error detection.

Table 5.1 continues.

...2/4

Category	Chan & Wei [20]	CED-invested Kung-Leiserson BMMSA [10,20]	CED-invested Huang-Abraham BMMSA [11,20]	J.H. Patel et al. [6,21,22]
CED Technique	RESO-based Time Redundancy CED Investment.	RESO-based Time Redundancy CED Investment.	RESO-based Time Redundancy CED Investment.	RESO-based CED Approach
Complexity without CED techniques	Processor: $W_1 * W_2$ Time: $N+\min(W_1,W_2)$	Processor: $W_1 * W_2$ Time: $N + \min(W_1,W_2)$	Processor: $W_1 * W_2$ Time: $N + \min(W_1,W_2)$	Processor: N^2 Time: $3N-2$
Redundancy required with the technique	Processor: 0 Time: N+1 Registers: $2(W_1 * W_2)$ TSC: $W_1 * W_2$ MUX: $3(W_1 * W_2)$	Processor: 0 Time: 2N+1 Registers: $2(W_1 * W_2)$ TSC: $W_1 * W_2$ MUX: $3(W_1 * W_2)$	Processor: 0 Time: N+1 Registers: $2(W_1 * W_2)$ TSC: $W_1 * W_2$ MUX: $3(W_1 * W_2)$	Processor: $N^2[((n+1)^2-n^2+4n+1)/n^2]$ Time: $2(3N-1)$ Shifters: $3N^2$ Registers: N^2 TSC: N^2
Redundancy Ratio	Processor: 0 Time: $O((N+1)/(N+\min(W_1,W_2)))$	Processor: 0 Time: $O((2N+1)/(N+\min(W_1,W_2)))$	Processor: 0 Time: $O((N+1)/(N+\min(W_1,W_2)))$	Processor: $O((6n+2)/n^2)$ Time: $O(2(3N-1)/(3N-1))$ Where n is the number of bits of the operands to be multiplied.
Diagnosis Performance	Faults - types: Transient - # allowed: Single & multiple PE failures - location: Possible	Faults - types: Transient - # allowed: Single & multiple PE failures - location: Possible	Faults - types: Transient - # allowed: Single & multiple PE failures - location: Possible	Faults - types: Transient - # allowed: Single faults - location: Possible
Utilization	On-line local error detection.	On-line local error detection.	On-line local error detection.	On-line local error detection

Table 5.1 continues.

...3/4

Category	Gupta & Bayoumi [23]	Our Approach
CED Technique	Logarithm-based Coding Approach	Concurrent Redundant Computation (CRC) and Space-Time Approach
Complexity without CED techniques	Processor: N^2 Time: 3N-2	Processor: N^2 Time: 3N-2
Redundancy required with the technique	Processor: - Time: - Counter: N^2 Shift Register: N^2 TSC: N^2 Other: ROM: N^2 Address Register: N^2 Memory Buffer Register: N^2	- N is an even number Processor: 0 Time: N-1 Buffer: N-1 - N is an odd number Processor: N Time: N-2 Buffer: N-2 Other: TSC: $\lceil N/2 \rceil$
Redundancy Ratio	Processor: - Time: -	- N is an even number Processor: 0 Time: $O((N-1)/(3N-2))$ - N is an odd number Processor: $O(N/N^2)$ Time: $O((N-2)/(3N-2))$
Diagnosis Performance	Faults - types: Permanent & Transient - # allowed: Single & multiple cell failures - location: Yes	Faults - types: Permanent & Transient - # allowed: Single & certain multiple fault pattern - location: No
Utilization	On-line local error detection.	On-line global error detection.

Table 5.1 continues

...4/4

underlying algorithm itself. In this case, there is no loss in throughput. However, when the scheme is applied to a systolic structure where inputs are applied in every cycle, there is a loss of throughput of 50%. This is highly undesirable in situations where a high throughput is a crucial requirement. The scheme cannot detect all transient faults. Since the method can only be applied to structures like the bidirectional systolic array, where to perform an NxN matrix multiplication, $(2N-1)^2$ cells are required instead of N^2, there is an inherent hardware redundancy when this method is used. If this hardware redundancy is coupled with the additional hardware required for CED, the hardware redundancy ratio is greater than 200%, when the value of N is large. The time redundancy ratio is of O(N / 3N-1), which is about 33%. In addition to this time redundancy, there is an extra time redundancy factor that has to be taken into account. Since bidirectional data movement is employed, a new problem instance has to wait until the data for the former problem instance have drained the pipeline of a row of processors. Consequently, the block pipeline period is increased and this reduces the average throughput drastically.

The approach proposed by Wu [13] is similar to that proposed by Gulati and Reddy [12]. The difference being that while the latter method is applicable to bidirectional systolic array structures, the former method is applicable to unidirectional data flow linear systolic arrays. Both approaches [12,13] have similar attributes. However, Wu's approach [13] requires a hardware redundancy ratio of $O(N / N^2)$. For very large values of N (e.g N=100), this is about 1%. The time redundancy ratio is of O(N / 3N-2), which is about 33%. There is a loss of throughput by 50%, since the CED technique requires inputs to be applied to the array in alternate cycles. Although, the techniques proposed in [12] and [13] are similar, however, the method in [13] has better design features (like the hardware and time redundancy ratios) than the method in [12].

The TRIFT approach proposed in [14,15] requires a hardware redundancy ratio of about 1%. the time redundancy ratio is of O(N / 2N), and this is about 50%. Like in [12,13], the scheme in [14,15] is limited to only those systolic implementation in which

the data as well as (sub) results move from one cell to the other. There is a reduction in throughput by 50%. Also, the scheme can detect some transient faults, and with roll-back of the computations, the system can recover form such transient faults. However, since the data flow of the two redundant computations are the same, any transient fault that affects the results of one data flow will also affect the results of the other data flow in a similar manner. Thus, a comparison of these results will not show any discrepancy. Hence, such categories of transient faults cannot be detected by this scheme. The scheme can handle multiple simultaneous faults. Since fault location is possible, the scheme can also be easily integrated with architectural FT schemes for permanent fault recovery.

For the CED approach proposed by Abraham *et al* [2], the hardware redundancy ratio required to detect errors in a systolic array and to perform an NxN matrix multiplication, is more than 1%. The time redundancy ratio is also 1%. This will be true if the respective inputs of each of the extra PE's, are applied to the respective PE's at the same time. Otherwise, extra time redundancy would be required if the input data to the extra PE's are to be pipelined from one extra PE to the other. This scheme is not without its drawbacks. It is applicable to those systolic structures in which the data as well as (sub) results move from cell to cell. The CED method can only be employed if the given problem can be linearized. There is also the difficulty of linearizing the problem, there is no systematic or unique approach of converting a given algorithm into a linear model to be implemented using linear arrays. With the introduction of the new variables required to perform the CED, the complexity of the PE's is increased. There is also the possibility of increase in computation of each PE, as a result of the introduction of the new variables. This will in turn reduce the frequency of operation of each linear system (row or column). Furthermore, the scheme can only diagnose errors from a single PE in each row or column. Multiple faulty cells in a row may be handled. The diagnosis is vulnerable to false alarms brought on by roundoff errors. Finally, with the introduction of the extra PE and the application of input data to the respective PE's at the same time, the

structure of the systolic array is no longer highly regular and granular.

When the technique proposed by Huang and Abraham [16] is applied to a processor array system with N=100, the hardware redundancy ratio (in terms of the number of processors) is 2%. The time redundancy ratio is about 4%. One of the drawbacks of this technique is that the structure resulting from the approach is no longer highly regular and granular. The non-regularity even depends on the array size. As the array size increases, more silicon area is required to route data into the array. Thus, for large values of N, the hardware redundancy ratio could rise to more than 100% instead of 2%. There is also the possibility that as the array grows bigger, long wires and non-uniform delays are introduced. This could increase the time redundancy ratio. As these problems become severe, they could limit the scalability of the design. Other drawbacks of this approach include. the diagnosis and correction latency is very long and it is vulnerable to false alarms brought on by roundoff errors.

For the approach proposed by Cosentino [17], the scheme requires a hardware redundancy ratio of 1%. The time redundancy ratio is 66%. Some of the drawbacks of the scheme include: It is applicable to only those classes of systolic arrays in which the partial results must stay in the cells. There is a reduction in throughput by 50%. The scheme cannot detect all transient and permanent faults. It allows only single faults in each row of the array, multiple fault coverage requires high hardware overhead.

The technique proposed in [18] requires a hardware redundancy of 33%. In the RNS computations, three channels of cells constitute the non-redundant channels which are used to perform independent residue calculations. For error detection, one redundant channel is required to perform another independent residue computation, which can be compared against the residue produced by the non-redundant channels. If we assume that one channel of the cells could perform the computation otherwise done by the three non-redundant channels, then, there exists extra hardware redundancy. In this case, the hardware redundancy ratio becomes 300%. The time redundancy ratio depends on the

time required for the conversion of the residues produced by the non- redundant channel, to the binary output. The longer this time is and also the more complex the residue-to-binary circuit is, the higher the time redundancy ratio. As can be seen from table 5.1, the RNS technique uses 2N clock cycles to perform a computation that could have been done in N clock cycles. This increase in latency can also be taken as time redundancy. In view of this, the time redundancy ratio can be considered to be 100%. Due to its high hardware and time redundancy ratios, this scheme may not be employed in large problem size systolic design. The scheme can only detect one erroneous residue, and thus, it can detect any pattern of faulty cells that has no more than one faulty cell in any processing block. The scheme cannot detect all the faults in the array. Only those faults that affect the computation units of the cells are detectable, whereas faults in the data paths cannot be detected.

In order to detect errors in systolic structures, the approach proposed by Lev-Ari and Friedlander requires a hardware redundancy to implement the error detection subsystem. The cost of the subsystem depends on the ability to select efficient diagnostic expressions. The less efficient the diagnostics, the more the hardware redundancy required to implement the error detection scheme. The time overhead of the scheme also depends on the selected diagnostics. The drawbacks of this approach are for two reasons: (i) there is no systematic method of selecting an efficient diagnostic expression; (ii) the error detection technique is not unique for any given problem. The properties of efficient diagnostic expressions are not specified.

The central problem in implementing algorithm-based error detection using this approach is to select efficient diagnostic expression. Since the approach to select the expressions is not straight forward, this increases the degree of difficulty in using this CED technique. The scheme can detect faults in the cells but not in the interconnection between the cells of the array. N(2N+1) processors are required to perform matrix operations which could have been done using N^2 processors. This constitutes a hardware

redundancy ratio of 100%. This hardware overhead coupled with that of the error-detection subsystem increases the total hardware redundancy of the scheme. It is also mentioned in [19] that one of the efficient diagnostic expressions that could be used for matrix multiplication operations, is the checksum method [16]. In that case, the CED scheme in [19] will inherit all the short falls of the approach in [16]. One of such pitfalls is that, the resulting CED structure will no longer be highly regular and this will limit the scalability of the systolic design.

The RESO-based CED implementation in the BMMSA proposed by Chan and Wey [20] requires a time redundancy ratio of 100%. Since the scheme uses RESO-based time redundancy approach, the hardware overhead is minimal. As pointed out before, Chan and Wey [20] extended their approach to the RESO-based CED implementation in the Kung-Leiserson BMMSA [10], and in the Huang and Abraham BMMSA [11]. The CED-invested Kung-Leiserson BMMSA requires a time redundancy ratio of 200% and also, minimal hardware overhead. On the other hand, the CED-invested Huang-Abraham BMMSA [11] requires a time redundancy ratio of 100%. Like the other two CED BMMSA, it requires minimal hardware overhead.

Although, the CED approach proposed by Chan and Wey [20] can detect errors down to a single processor, the scheme is not without its drawbacks. It can only detect transient faults since the same cell is used to perform the two redundant computations. Even at that, it can only detect some of the transient faults. This is because, if the duration of the intermittent or transient fault is greater than one clock cycle, the two redundant computations would be affected the same way by this single transient fault. Thus, a comparison of the two results would be regarded as error-free results. Hence, this fault cannot be detected. Furthermore, since the scheme can detect only some transient faults, hence, permanent faults cannot be detected. There is a reduction in throughput by at most 50%. The scheme can only work for a 50% per - cycle PE idlenes. It is restricted to a class of systolic arrays in which data as well as (sub) results keep moving from cell to

cell during computations. Their proposed CED approach is not systematic. There is no methodology of partitioning the PE's and assigning them to the different clock groups. The clock for each group determines when the processors in that group will be active. Also, there is an increased complexity of the PE's, each PE consists of extra three MUX, two registers and a TSC circuit. In addition , the scheme requires more control signals - two control clocks CL_1 and CL_2 (where CL_1 is to lead CL_2 by one cycle), MUX control signals and the control signals for the registers to latch the results at some specific times. Finally, their BMMSA design is concentrated on the hex-connected systolic arrays, the extention of the design strategy to other types of systolic structures is not straight forward.

The CED technique proposed by Patel and Fung [6,21], Cheng and Patel [22] requires both time and hardware redundancies for error detection. Using this scheme to detect errors in a systolic array that performs matrix multiplication operation, the hardware redundancy ratio is of $O((6n+2) / n^2)$, where n is the number of bits in the operands to be multiplied. For n less than 6 bits, the hardware redundancy ratio is greater than 100%. However, this ratio decreases as the number of bits in the operands (n) increases. The time redundancy required by the technique is of $O(2(3N-1) / (3N-1))$, which is over 200%. The scheme can only detect single faults. In order to achieve higher error coverage, extra hardware is needed. It is more effective for transient error detection. The diagnosis latency is very longer. Since error detection is performed in the individual cells, it is possible to locate the faults that occur in the systolic array down to the individual cells.

For the LOED CED technique proposed by Gupta and Bayoumi [23], the error detection logic is incorporated in each cell to make every cell self-testing. The harware required to compute the anti-logarithm is very complex. It consists of a counter, a shift register, a comparator, memory element (ROM), memory address register and a memory buffer. Also, the accuracy of the anti-logarithm operation depends on the size of the

ROM. Therefore, the larger the size of the ROM, the higher will be the fault coverage. With the increase in the complexity of the redundant hardware, the hardware redundancy ratio could be as high as 100%. As regards to the time redundancy, if the logarithmic operations take longer time than the multiplication operations, this could slow down the system and hence, the frequency of operation could be reduced. The scheme can provide fault location. Since the cells are self-testing, LOED technique has the ability to detect some of the multiple faults.

The shortfalls of the scheme include: It cannot detect all the faults in the array. The adder in the cell is used for both the multiplication and the logarithm operations. If a fault occurs in this unit such that both operations become erroneous in a similar manner, the fault cannot be detected. The scheme is also vulnerable to false alarms due to rounding errors. The logarithmic coding is an approximation and it is prone to introduce roundoff errors. The exact match between the product and anti-logarithm of sum of logs cannot be expected with finite precision. Furthermore, the accuracy of the approach and hence, the fault coverage, lies on the size of the ROM which is used as a look-up table during the computation of the anti-logarithm. The more accurate the anti-logarithm computation, the more complex is the size of the ROM. This could limit the scalability of the design. Therefore, in order to detect most of the faults in the systolic array, high hardware overhead with complex circuitry is required to do so.

In our CED approach, the hardware and time redundancy required for CED depend on whether the size of the matrix N, is even or odd number. If N is even, there is almost no hardware redundancy required. The hardware redundancy ratio (in terms of the number of processors) is 0%. The time redundancy ratio is of O(N-1 / 3N-2). For very large values of N, the time redundancy ratio is 33%. On the other hand, if N is odd number, the hardware redundancy is of $O(N/N^2)$, that is about 1%. The time redundancy ratio is of O(N-2 / 3N-2). This represents a time overhead of 33%. Our scheme can detect all single permanent and temporary faults and most multiple fault patterns. There is no

reduction in throughput. The scheme can be applied to any systolic implementation whether the partial results must stay in the cells, or in which the data as well as the (sub) results keep moving from cell to cell. Although, it requires extra delay elements, however, it does not require the error detecting logic in every cell in the array. It is not vulnerable to false alarms caused by roundoff errors. The introduction of the CED technique does not affect the regularity and granularity of the systolic array. Also, since the hardware redundancy required is very small, the scalabilty of the design scheme is not limited to small or medium size systolic arrays. Since the error detection is done only at the output of the CED systolic array, our scheme cannot locate faults down to the individual cells.

Table 5.2 consists of the comparison of the percentage hardware and time redundancy ratios required by various CED schemes to perform matrix multiplication in systolic arrays. From tables 5.1 and 5.2, it is observed that our scheme requires the least hardware overhead to achieve CED in systolic arrays. If N is an even number, CED is achieved at almost no hardware cost. Since the proposed scheme provides the best hardware redundancy ratio, hence it is said to be more area efficient than the schemes proposed in the literature. Althogh, the time redundancy ratio of 33% required by our scheme is not the best time redundancy ratio, however, it is comparable or even better than time redundancy ratios of some of the existing schemes.

The CED strtegy proposed in this thesis rely on global mechanism to detect faults in the systolic array. Therefore, if it is employed to achieve fault-tolerance, the FT technique will basically rely on global error masking. On the other hand, since the scheme cannot detect errors in the individual cells, it cannot be easily integrated with some reconfiguration algorithms to locate and correct errors within the cells of the systolic array. However, with our scheme, errors produced as a result of faults in the array can be detected and possibly masked concurrently during the normal operation of the systolic array.

CED Techniques	Percentage Hardware Redundancy (N=100) (%)	Percentage Time Redundancy (%)
Gulati & Reddy [12]	200	33
Wu & Wu [13]	1	33
Kung & Manolakos [14,15]	1	50
Abraham et al. [2]	1	50
Huang & Abraham [16]	100	4
Cosentino [17]	1	66
Cosentino [18]	300	100
Ari & Friedlander [19]	100	33
Chan & Wei [20]	1	100
Patel et al. [6,21,22]	100	200
Gupta & Bayoumi [23]	100	-
Proposed Scheme	0	33

Table 5.2 Summary of the comparison of the percentage hardware and time redundancy ratios required by various CED schemes to perform matrix multiplication in systolic arrays.

The design scheme presented in this chapter will be extended in chapter VI, to design area efficient fault-tolerant systolic array architectures.

5.6 CONCLUDING REMARKS

In this chapter, we have presented an area efficient CED scheme for VLSI systolic array architectures. This scheme offers solutions to several problems of the other known CED schemes. It is applicable to a wide class of VLSI implementation of algorithms. The technique is not limited to only those systolic implementations where the data as well as (sub) results keep moving. It can also be applied to those implementations where either the data or the (sub) results are stored. It has a feature of exploiting the advantages of the interleaved computations without any reduction in the throughput of the array structure. The scheme can detect all single permanent and temporary faults and a majority of the multiple fault patterns. It is area efficient, there is no need to replicate all the hardware in the CED systolic array. Redundancy is introduced only at those places where it is needed. The silicon area is limited to one row of extra functional blocks in those cells that compute the corresponding results of the two independent computations, delay elements and the error detection logic. By rotating the systolic array of the first version of the algorithm by 180 degrees about one of the indices, the computation of the output results of the two versions of the algorithm can be scheduled to start at the same time with both using the same computational space but at different times. Although the input data is re-scheduled, the corresponding output results of the two computations arrive at the output of the array at the same time. This facilitates the comparison of the two answers without the introduction of any additional control or synchronization circuitry. Consequently, there is no high delay cost incurred and thus, the technique can be employed in real-time applications. This scheme can be applied to any systolic array structure (e.g. 2D, tree - connected etc.) and also to other types of VLSI arrays.

5.7 REFERENCES

[1] D. Siewiorek and R. Swarz, "The Theory and Practice of Reliable System Design," *Bedford, MA: Digital Press*, 1982.

[2] J. Abraham, P. Banerjee, C-Y. Chen, W. Fuchs, S-Y Kuo and A. Reddy, "Fault Tolerance Techniques for Systolic Arrays," *IEEE Computer*, pp. 65-74, 1987.

[3] J. F. Wakerly, "Error Detecting Codes, Self-Checking Circuits and Applications," *Elsevier - North Holland, New York*, 1978.

[4] B. W. Johnson, J. H. Aylor and H. H. Hana, "Efficient Use of Time and Hardware Redundancy for Concurrent Error Detection in 32-bit VLSI Adder," *IEEE Journal of Solid-State Circuits*, vol. 23, No. 1, February, 1988.

[5] D. Reynolds and G. Metze, "Fault Detection Capabilities of Alternating Logic," *IEEE Trans. Comput.*, vol. C-27, pp. 157-162, Dec., 1978.

[6] J. H. Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. Comput.*, Vol C-31, pp. 589-595, 1982.

[7] H. H. Hana and B. W. Johnson, "Concurrent Error Detection in VLSI Circuits Using Time Redundancy," *in Proc. IEEE Southeast con'86 Regional Conf.*, pp.208-212, Mar. 23-25, 1986.

[8] J. Shetlesky, "Error Correction by Alternate Data Retry," *IEEE Trans. Comput.*, vol C-27, pp. 106-112, Feb., 1978.

[9] D. Patterson and C. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE trans. Comput.*, vol. C-29, pp. 108-116, Feb., 1980.

[10] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays," *in Introduction to VLSI Systems*, by C. A. Mead and L. A. Conway. Reading, MA: Addison - Wesley, 1980.

[11] K. H. Huang and J. A. Abraham, "Efficient Parallel Algorithms for Processor Arrays," *in Proc. IEEE ICPP*, 1982, pp. 271-279.

[12] R. K. Gulati and S. M. Reddy, "Concurrent Error Detection in VLSI Array Structures," *Proc. IEEE Intl. Conf. on Computer Design*, pp. 488-491, 1986.

[13] C-C Wu and T-S Wu, "Concurrent Error Correction in Unidirectional Linear Arithmetic Arrays," *Proc. 17-th Intl. Symp. on Fault-Tolerant Computing*, pp. 136-141, 1987.

[14] S. Y. Kung, VLSI Array Processors, *Prentice Hall*, 1988.

[15] E. S. Manolakos, "Transient Fault Recovery Techniques for the VLSI Processor Arrays," Ph.D thesis, University of Southern California, May, 1989.

[16] K. H. Huang and J. A. Abraham, "Algorithm-based fault-tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, pp. 518-528, June 1984.

[17] R. J. Cosentino, "Concurrent Error Correction in Systolic Architectures," *Proc. IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 1, pp. 117-125, January 1988.

[18] R. J. Cosentino, "Fault Tolerance in a Systolic Residue Arithmetic Processor Array," *IEEE Trans. on Comput.*, vol. 37, No. 7, pp. 886-890, July, 1988.

[19] H. Lev-Ari and B. Friedlander, "On the Systematic Design of Fault-Tolerant Processor Arrays with Application to Digital Filtering," *VLSI Signal Processing III*, pp. 483-493, 1988.

[20] S-W Chan and C-L Wey, "The Design of Concurrent Error Diagnosable Systolic Arrays for Band Matrix Multiplication," *Proc. IEEE Trans. on Computer-Aided Design*, Vol.7, No.1, pp. 21-37, January 1988.

[21] J. H. Patel and L. Y. Fung, "Concurrent Error Detection in Multiply and Divide Arrays," *IEEE Trans. Comput.*, vol. C-32, No. 4, pp. 417-422, April 1983.

[22] W-T Cheng and J. H. Patel, "Concurrent Error Detection in Iterative Logic Arrays," *FTCS*, pp. 10-15, June 1984.

[23] S. R. Gupta and M. A. Bayoumi, "Concurrent Error Detection In Systolic Arrays For Real-Time DSP Applications," *VLSI Signal Processing III*, edited by Robert W. Brodersen and Howard S. Moscovitz, IEEE Press, 1988.

[24] H. F. Li, C. N. Zhang and R. Jayakumar, "Latency of Computational Data Flow and Concurrent Error Detection in Systolic Arrays," *CCVLSI '89*, pp. 251-258, 1989.

[25] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "Area Efficient Computing Structures for Concurrent Error Detection in Systolic Architectures," *accepted for presentation in the IEEE Int'l Conf. on Parallel Processing (ICPP '91)*, Philadelphia, 1991.

[26] M. O. Esonu, S. Hariri and A. J. Al-Khalili, "A Systematic Approach for Designing Fault - Tolerant Systolic Architectures," in *Proc. 1989 Joint Tech. Conf. on Circuits/Systems, Comput. and Communications*, Sapporo, Japan, June 25-27, 1989.

[27] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "Design of Optimal Systolic Arrays: A Systematic Approach," *IEEE Symp. on Parallel and Distributed Processing*, pp. 166-173, Dallas Texas, Dec. 9-13, 1990.

[28] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "On the Design of Optimal Fault-Tolerant Systolic Array Architectures," *IEEE 5th Int'l Parallel Processing Symp.*, pp. 352-357, Anaheim, CA., May, 1991.

CHAPTER VI

AREA EFFICIENT FAULT-TOLERANT COMPUTING STRUCTURES FOR SYSTOLIC ARRAYS

6.1 INTRODUCTION

In chapter IV, we described an approach for designing fault-tolerant (FT) systolic array architectures using the TMR technique. In this approach, each processing element in the FT systolic array is provided with three sets of functional blocks, with each set used to compute the output results of one version of the three versions of the algorithm. Although, this technique is very general and can be applied to any level in a highly parallel system, however, the cost of the fault-tolerant system is high. The hardware overhead for fault-tolerance is at least 200 percent, without counting the cost of the voting circuit. It has been mentioned that fault-tolerance can be achieved in systolic arrays, at a lower cost by employing concurrent error detection (CED) techniques. Therefore, error detection has become one of the corner stones of many fault-tolerance techniques. In view of this, we proposed and presented in chapter V, an area efficient technique to design concurrent error detection systolic array architectures.

In this chapter, the results of the CED technique will be extended to develop a technique for designing area efficient fault-tolerant computing structures for systolic arrays. We propose two fault-tolerant design methods: in the first method, a fault-tolerant systolic array is constructed using two CED systolic arrays. While in the second method, the respective systolic arrays of three versions of the algorithm are merged, to produce a fault-tolerant systolic array. The merits of these methods include: They are area efficient, there is no need to replicate all the hardware in the FT systolic array. Redundant hardware is introduced only at those places where it is needed. All the single transient and permanent faults in the FT array can be detected and corrected. In addition to

detecting and correcting all single faults, these methods have the capability of tolerating multiple fault patterns with high probability of coverage. Since it is possible to schedule the computation of the output results of the three versions of the algorithm, to start at the same time with both using the same computational space but at different times, consequently, there is no high-delay cost incurred. They have a feature of exploiting the advantages of the interleaved computations without any reduction in the throughput of the FT systolic array structure. These methods provide effective means of designing fault-tolerant systolic array architecture.

The outline of this chapter is as follows: In section 6.2, we will describe in detail the methods for mapping algorithms into fault-tolerant systolic arrays. The design methods will be applied to an algorithm for matrix multiplication in order to demonstrate the generality and novelty of our approach to design fault-tolerant VLSI systolic architectures. Procedures for designing area efficient FT systolic architectures are also presented. The fault-tolerant analysis of the proposed design schemes will be discussed in section 6.3. Section 6.4 gives the area and time overhead of the proposed schemes. In section 6.5, we will compare the two methods of designing fault-tolerant systolic arrays proposed in this chapter. Also, our FT scheme will be compared with other FT schemes proposed in the literature. Finally, section 6.6 contains the summary and concluding remarks.

6.2 CONCURRENT ERROR DETECTION AND CORRECTION

In this section we present the design scheme for fault-tolerant systolic arrays. A methodology to design and implement area efficient testable systolic arrays has been proposed in [1] and also in chapter V. Here, we extend the techniques in [1] to design fault-tolerant systolic architectures. The class of faults within the systolic array that the scheme described here will tolerate is the same as that given in chapter V. In other words, the same fault model used in chapter V will be applicable to the design scheme discussed in

this chapter.

6.2.1 The Proposed Scheme

The developed fault-tolerant scheme achieves concurrent error detection and correction of the faults through concurrent redundant computation (CRC). It is based on the observation that at any given time the data flow computational activity is located in only some cells in the FT systolic array. That is, at any given time instant, some of the cells in the FT systolic array are performing meaningful computations of the output results of the data variables while some are not. Hence, there is inherent spatial redundancy in the array which could be exploited to perform concurrent redundant computations. Two or more independent computations can be launched into the FT systolic array in such a way that they are performed on different but partly overlapping regions of the array. If two or more corresponding computations of the independent wavefronts have to share the same computational resource at the same time, then spatial redundancy can be added only at this resource to ensure that the independent computations are simultaneously but separately computed on the different computational resources. Thus at the time instant when the computational wavefront of the required computation reaches the faulty cell, its redundant counter parts would have been confined to a fault-free region of the array. Consequently, a comparison of the corresponding output results would lead to the detection and correction (masking) of the fault and this will be true for the case of the single fault assumption.

In this subsection, we will describe two methods for mapping algorithms into fault-tolerant systolic arrays. In our approach, fault-tolerant algorithms are designed by introducing **redundant computations** at the algorithmic level. Thus, when these algorithms are mapped into specific VLSI systolic architectures, using the space-time mapping techniques (described in section 2.3 and 3.2), the resultant arrays are fault-tolerant at minimal additional cost in terms of area and frequency of operation. In the first method, an area

efficient concurrent error detection (CED) systolic array [1] is constructed as follows: We obtain two transformed dependency matrices (TDM's) that represent the two different versions of the given algorithm. The first TDM is obtained by selecting a valid transformation matrix which transforms the dependency matrix of the algorithm into the new transformed dependency matrix. However, the second one is obtained by rotating the systolic array corresponding to the first TDM by 180 degrees about any of the indices that represent the spatial component of the TDM. These TDM's are mapped into respective systolic arrays. Concurrent error detection (CED) systolic array is constructed by merging the corresponding systolic array of the two versions of the algorithm. Having constructed the CED systolic array, it is then duplicated such that four sets of the output results are produced by the two CED systolic architectures. Concurrent error correction or fault-tolerance is achieved by voting on these results, using redundant voters. It is worthwhile to mention that, the total area of the fault-tolerant systolic array designed using this approach, is much less than designs that use the TMR approach, yet we obtain four versions of the output results through different dynamic paths.

In the second method, three transformed dependency matrices that represent the three versions of the algorithm are obtained and then mapped into respective systolic arrays. Concurrent error correction (CEC) systolic array is constructed by merging the corresponding systolic array of the three versions of the algorithm. In this case also, as in the first method, we identify the identical computations that may need to be computed at the same computational site and at the same time. By adding extra hardware only at those computational sites at which they will be computed and by re-scheduling the input data (such that the original input data scheduling is satisfied), the interactions between the three wavefronts are isolated. There is no need to replicate all the hardware in the cells of the CEC systolic array and also, due to the input data scheduling, all the interconnection links need not be replicated. Three sets of the computed output results are produced by the different versions of the algorithm. Fault-tolerance is achieved by first, comparing two

of the three sets of output results for discrepancy. If these two sets of the output results differ, then the third copy is selected as the fault - free output, otherwise, any of the compared two sets of the output results is selected. Hence in this method, error detection is followed by error correction in the presence of a fault.

The three transformed dependency matrices for the three versions of the algorithm are derived in the following way. The first TDM is obtained in a similar fashion as the first TDM in the first method. That is, by selecting a valid transformation matrix which transforms the dependency matrix of the algorithm into the new transformed dependency matrix. The TDM of the second version is the same as the TDM of the first version. Therefore, their corresponding systolic arrays perform the same computations. However, the two systolic arrays are related in such a way that the identical computations performed by one systolic array at some computational sites, are performed by the other array at a space which is a small incremental distance from the previous computational sites [2]. In which case, the two arrays can be said to be in a Concurrent Redundant Computation (CRC) mode.

For instance, let D and D' be two identical computations (D' is a redundant version of D) such that $S = \Delta = TD$ and $S' = \Delta' = TD'$ and $C_D = (i,j,k)$, $C_{D'} = (i',j',k')$, $C_S = \{(t,x,y)\}$, $C_{S'} = (t',x',y')$. In particular, C_D is related to $C_{D'}$ and C_S is related to $C_{S'}$ by $i'=i+d_i$, $j'=j+d_j$, $k'=k+d_k$, and $t' = t + d_t$, $x' = x + d_x$, $y' = y + d_y$, where d_i, d_j, d_k, d_t, d_x and d_y are small constants and $\begin{bmatrix} d_t \\ d_x \\ d_y \end{bmatrix} = T \begin{bmatrix} d_i \\ d_j \\ d_k \end{bmatrix}$. It is observed that $D=D'$ and $\Delta=\Delta'$.

It is defined [2] that,

S' is a concurrent redundant computation (CRC) of S in a systolic array implementation if and only if

(i) $C_{S'} \subset Z^3$,

(ii) $C_{S'} \cap C_S = \phi$, and

(iii) d_t, d_x, d_y are small constants.

It has also been proven [2] that there always exists a S' forming a CRC of a given S if and only if $\Delta t_{min} > 1$ or $\Delta x_{min} > 1$ or $\Delta y_{min} > 1$ for a 2-D systolic array, and $\Delta t_{min} > 1$ or $\Delta x_{min} > 1$ if S is a 1-D systolic array. Δt is the time delay before a computational site (x,y) will be used again to perform another operation after it has just completed one. Δx (Δy) is the difference of any two values chosen from x (y).

The TDM corresponding to the third version of the algorithm is obtained by rotating the systolic array corresponding to the second TDM by 180^0 about any of the indices that represent the spatial component of the TDM. This TDM can be derived from the TDM of the second version, using the results presented in chapter V (Eqs.(5.1-5.15)) and also the approach presented in [1]. In the following subsection, we will demonstrate the design scheme using an example of an iterative algorithm.

6.2.2 Application of the Proposed Scheme

In order to illustrate our design scheme, we will consider the matrix multiplication algorithm whose dependency matrix is given in Eq.(2.6). The same example of the algorithm will be used to demonstrate the two design methods.

6.2.2.1 Method One

In the first method, our technique for designing fault-tolerant systolic arrays is to duplicate the CED systolic array such that four sets of the output results are produced by the two CED systolic architectures. As an example, the CED systolic array shown in Fig. 5.5 will be considered for the design of the FT systolic array. The resultant fault-tolerant systolic array architecture designed using this CED array, is shown in Fig.6.1. Fault-tolerance is achieved by masking dynamically, errors using fault-tolerant voting schemes.

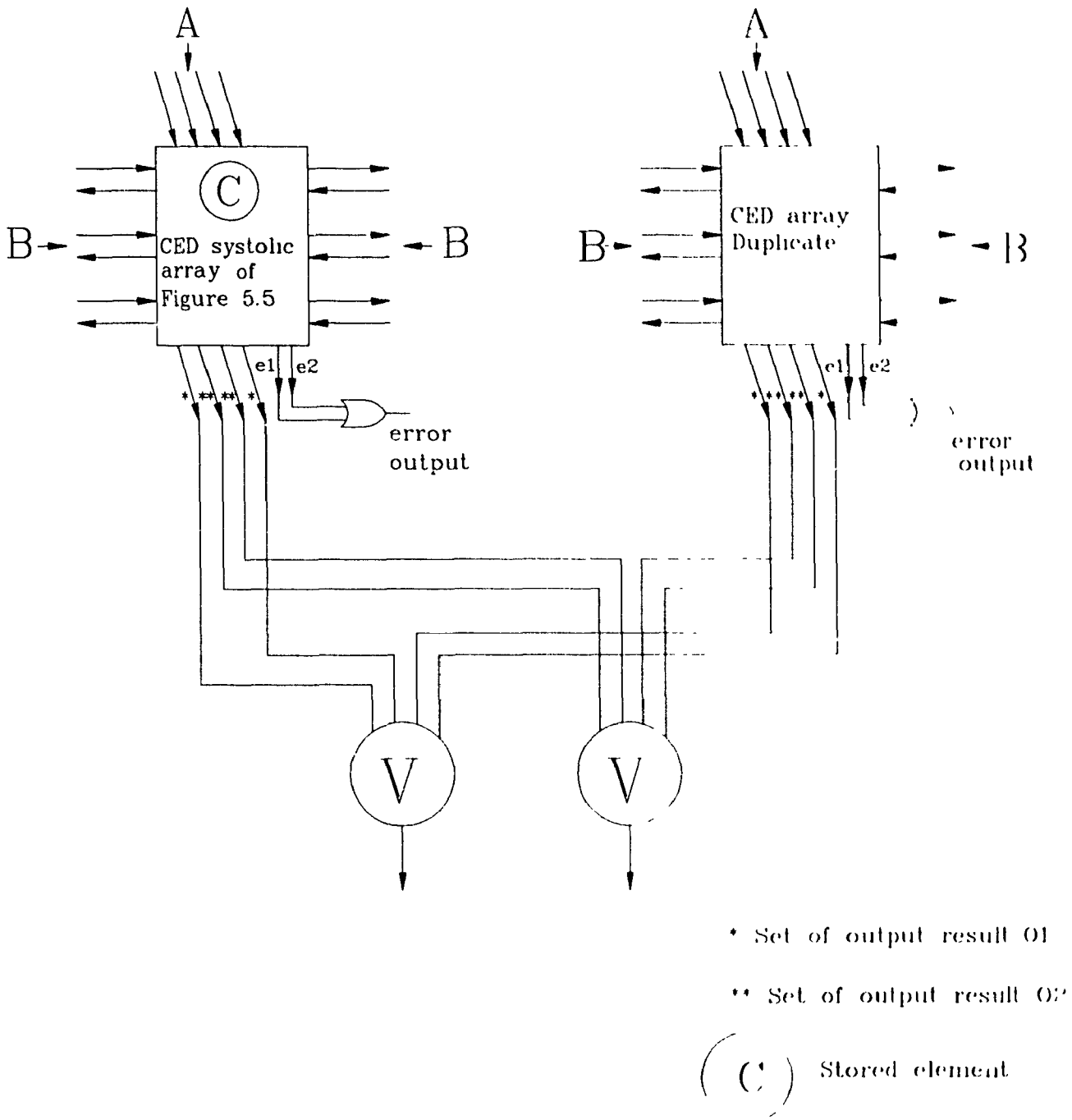


Figure 6.1 Fault-tolerant systolic array for matrix multiplication (for N=3).

6.2.2.2 Method Two

In the second method, the three versions of the algorithm, and hence their respective systolic arrays, are derived as demonstrated in the following example.

Example

For the purpose of illustration, here, we will denote S as the TDM of Eq.(2.6), and also choose it as the TDM of one version of the algorithm. The corresponding VLSI implementation of S is shown in Fig.6.2. The TDM of the second version of the algorithm, S', is the redundant version of S. It can be obtained using the results presented in section 6.2.1 [2]. If we choose a CRC S' of S with $d_t = 0$, $d_x = 1$ and $d_y = 0$, then we have,

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_i \\ d_j \\ d_k \end{bmatrix} \tag{6.1}$$

From Eq.(6.1), $d_i = -1$, $d_j = 1$, $d_k = 0$. Thus, $i' = i + d_i = i - 1$, $j' = j + 1$ and $k' = k$. The mapping of the index set of S and S' into VLSI arrays using the transformation T (Eq.(2.6)) is shown in Fig.6.3. The corresponding systolic array of S' is shown in Fig.6.4(a). The cell structure is the same as that of S.

The TDM of the third version of the algorithm, S'', is determined by rotating the systolic array of S', by 180^{o} about the vertical-axis. This can be derived using the results presented in chapter V (Eqs.(5.1 - 5.15)). The corresponding systolic array of S'' is shown in Fig.6.4(b). The fault-tolerant systolic array is constructed by merging the systolic arrays of S, S' and S'' as shown in Fig.6.4(c). As seen from Fig.6.4(c), the coefficients of the generated variable (variable A) in two of the versions of the algorithm need to be computed in cells (2,1), (3,2), (4,3), (5,1), (6,2) and (7,3). Also, the results of the three independent wavefronts need to be computed at the same time in cells (3,1), (4,2), (5,3), (4,1), (5,2) and (6,3).

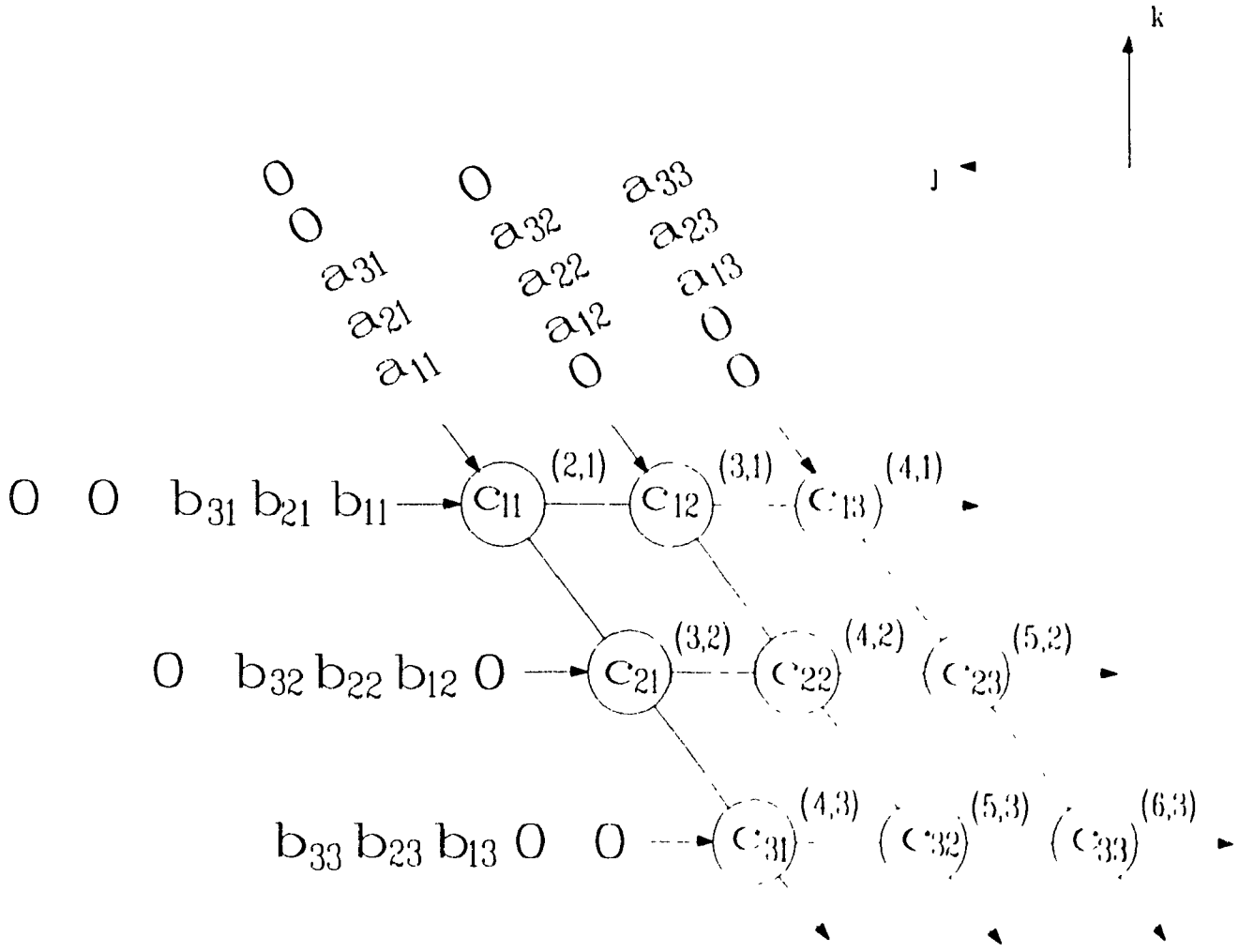


Figure 6.2 VLSI array structure of S that implements the matrix multiplication algorithm (for N 3).

i	j	k	$\hat{i}$ time	$\hat{j}$	$\hat{k}$ processor	i'	j'	k'	$\hat{i}'$ time	$\hat{j}'$	$\hat{k}'$ processor
1	1	1	3	2	1	0	2	1	3	3	1
1	1	2	4	3	2	0	2	2	4	4	2
1	1	3	5	4	3	0	2	3	5	5	3
1	2	1	4	3	1	0	3	1	4	4	1
1	2	2	5	4	2	0	3	2	5	5	2
1	2	3	6	5	3	0	3	3	6	6	3
1	3	1	5	4	1	0	4	1	5	5	1
1	3	2	6	5	2	0	4	2	6	6	2
1	3	3	7	6	3	0	4	3	7	7	3
2	1	1	4	2	1	1	2	1	4	3	1
2	1	2	5	3	2	1	2	2	5	4	2
2	1	3	6	4	3	1	2	3	6	5	3
2	2	1	5	3	1	1	3	1	5	4	1
2	2	2	6	4	2	1	3	2	6	5	2
2	2	3	7	5	3	1	3	3	7	6	3
2	3	1	6	4	1	1	4	1	6	5	1
2	3	2	7	5	2	1	4	2	7	6	2
2	3	3	8	6	3	1	4	3	8	7	3
3	1	1	5	2	1	2	2	1	5	3	1
3	1	2	6	3	2	2	2	2	6	4	2
3	1	3	7	4	3	2	2	3	7	5	3
3	2	1	6	3	1	2	3	1	6	4	1
3	2	2	7	4	2	2	3	2	7	5	2
3	2	3	8	5	3	2	3	3	8	6	3
3	3	1	7	4	1	2	4	1	7	5	1
3	3	2	8	5	2	2	4	2	8	6	2
3	3	3	9	6	3	2	4	3	9	7	3

Fig.6.3 : Mapping of index sets of S and S' into VLSI arrays using transformation matrix T (for N=3).

In this method, our approach for designing fault-tolerant systolic array is to add redundant computational unit in only those cells in the FT systolic array that need to compute the results of the three wavefronts at the same time. Then, the input data is re-scheduled such that the results of two independent wavefronts are simultaneously computed using different functional blocks. The results of the third wavefront are computed using any of the two computational units, provided that two identical computations are not performed using the same computational unit.

Figure 6.5(a) depicts the FT systolic array (for N=3) designed using this technique. The array consists of two types of cell structures. In the first type (type I cell), the computational units are not duplicated (cells (2,1), (3,2), (4,3), (5,1), (6,2) and (7,3)), while in

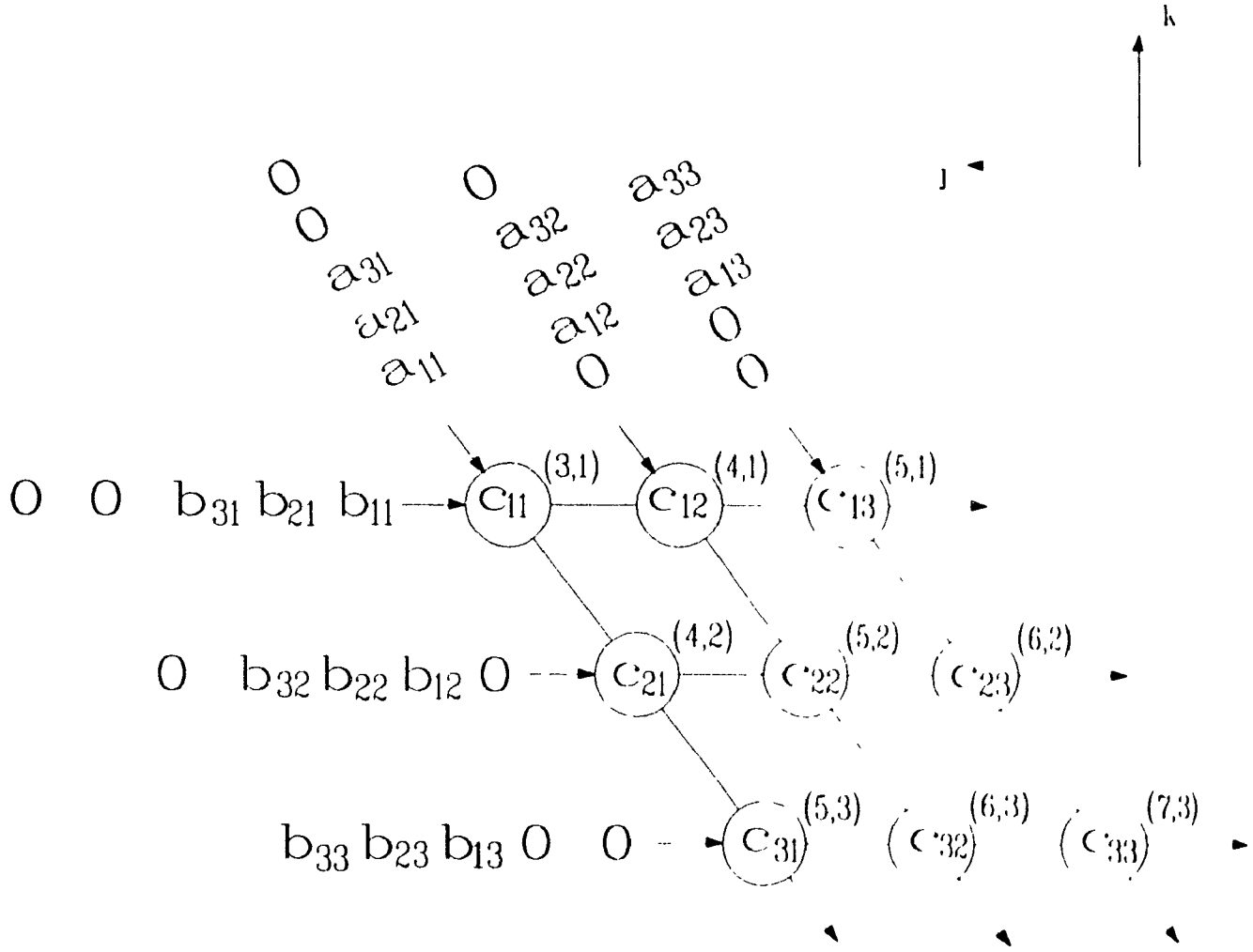


Figure 6.4(a) VLSI array structure of S' that implements the matrix multiplication algorithm (for N = 3)

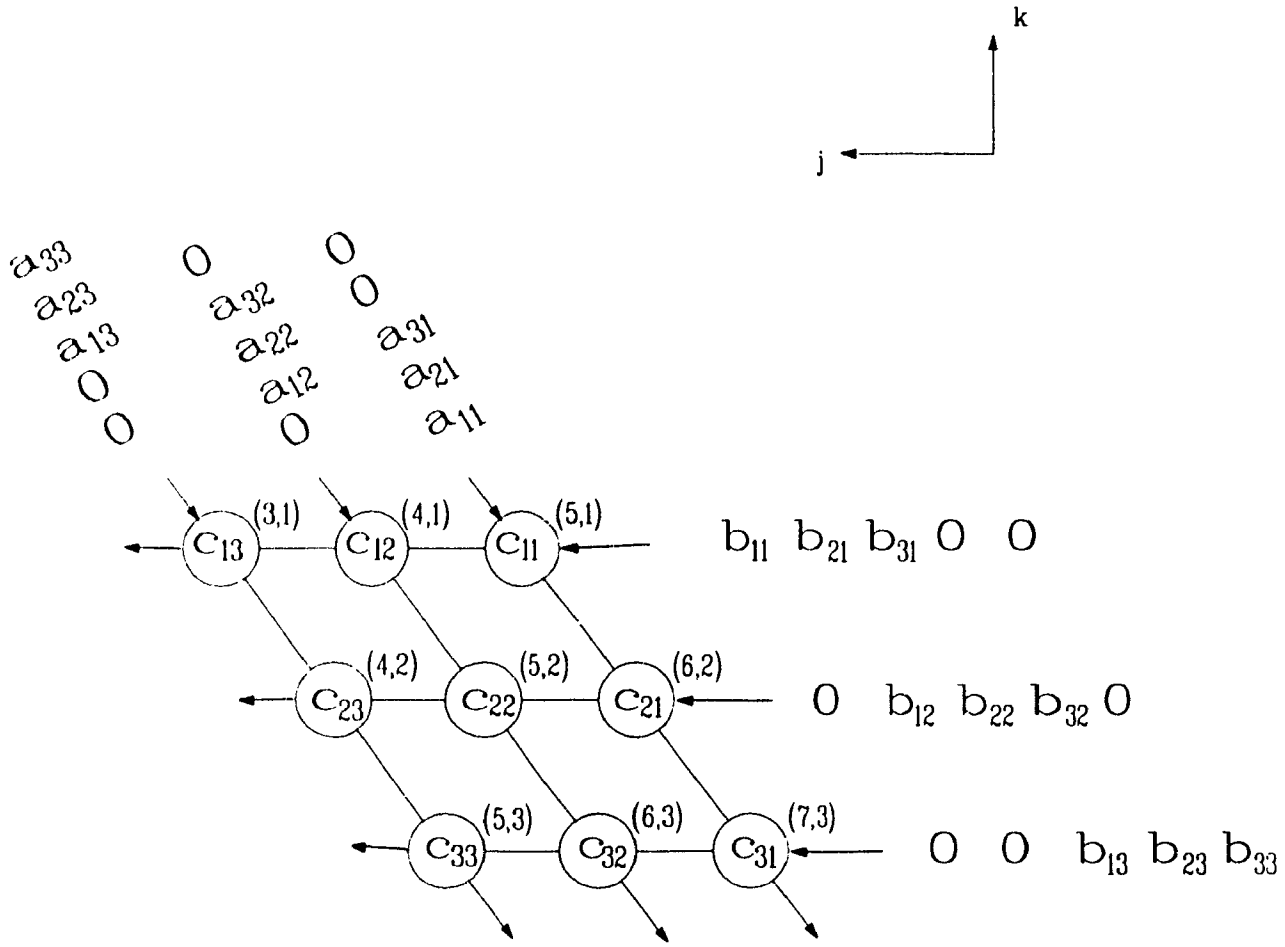


Figure 6.4(b) The VLSI array structure of S' resulting from rotation of Figure 6.4(a) by 180° about the vertical axis.

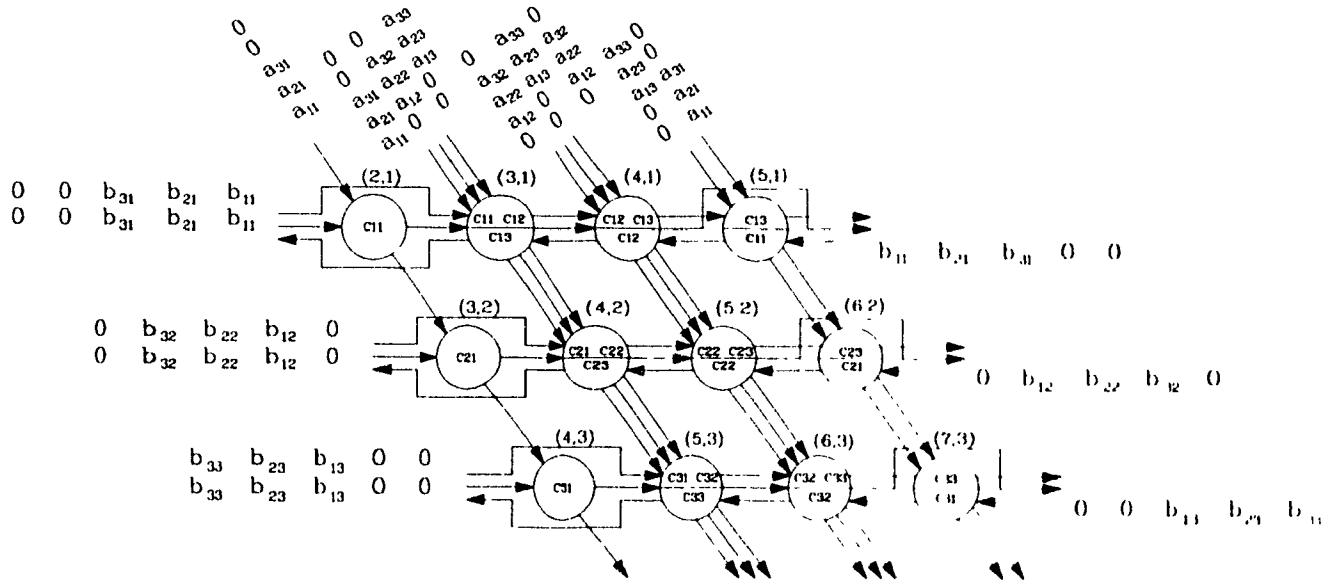
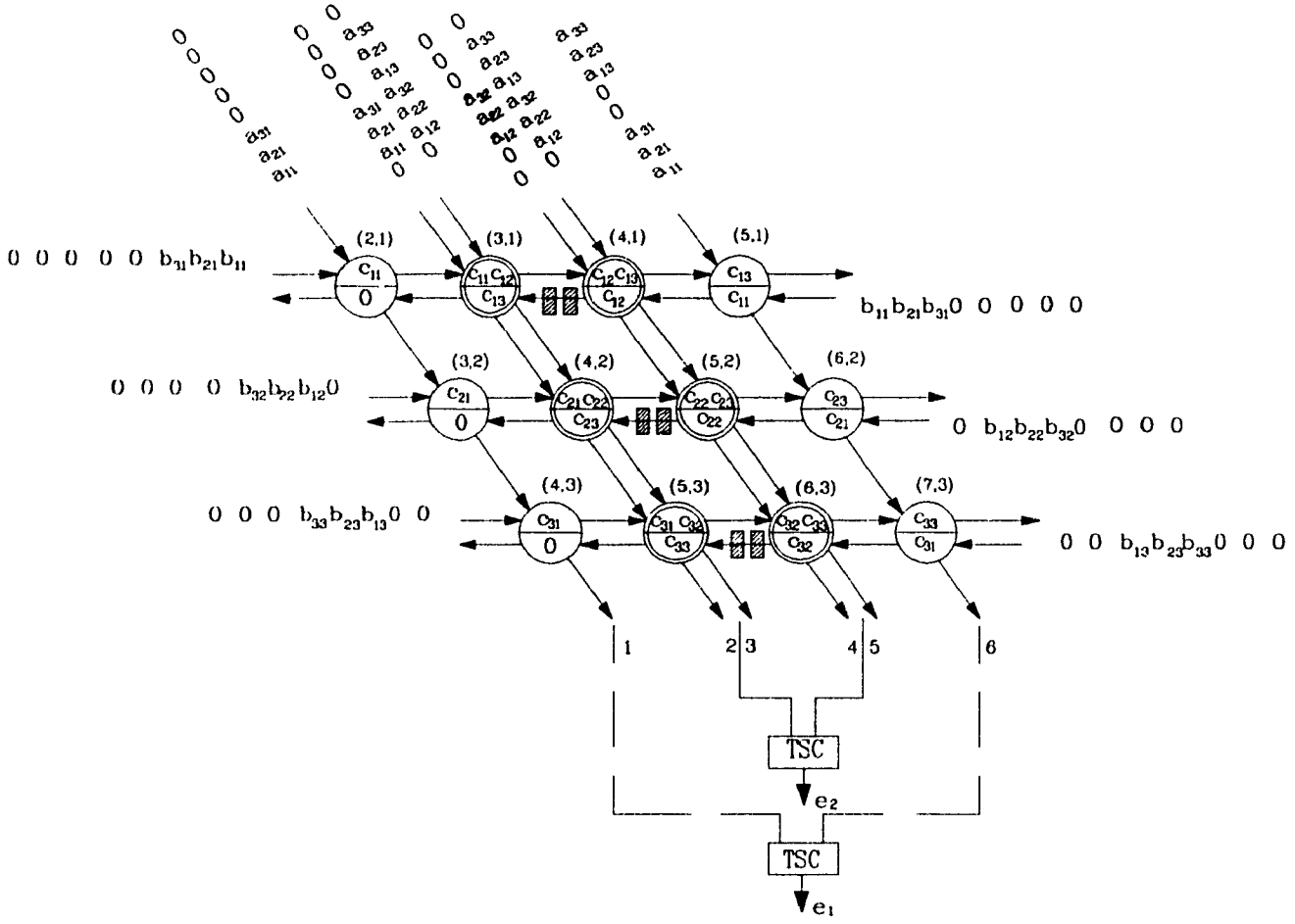
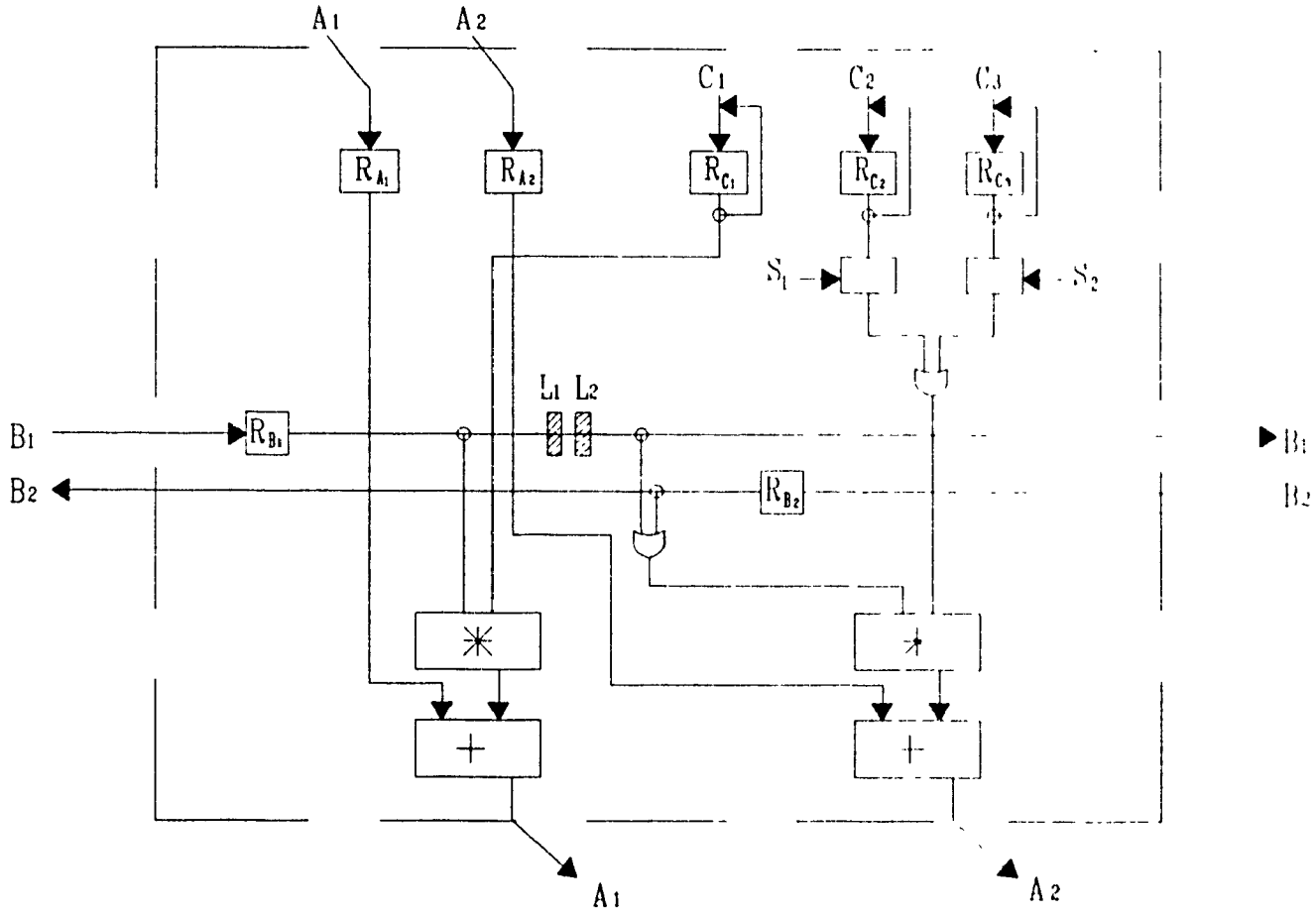


Figure 6.4(c) The VLSI array structure resulting from merging the systolic arrays of S, S' and S" (for N=3).



e_1 error output 1
e_2 error output 2
[] delay element
( ) type I cell
(( )) type II cell

Figure 6.5(a) FT systolic array for matrix multiplication (for N=3).



L_1, L_2 – delay elements

Figure 6.5(b) The internal structure of a set of type II cells showing the two delay elements incorporated inside the cells.

the second type (type II cell), the functional blocks are duplicated (cells (3,1), (4,2), (5,3), (4,1), (5,2), and (6,3)). The input data for variable A are re-scheduled so as to satisfy the original data scheduling. In order to synchronize the data propagation of each of the computations, two additional delay elements are introduced in each interconnection line of variable B in the FT systolic array. In one case, the two delay elements are introduced in the interconnection lines outside the cells, while in another case, they are introduced inside some of the type II cells (cells (4,1), (5,2) and (6,3)). The internal structure of the type II cells is shown in Fig.6.5(b).

In order to achieve concurrent error correction in the FT systolic array of Fig.6.5(a), the output results of outputs 1 and 6 are compared for discrepancies using a TSC circuit. If they are the same, it signifies that the output results are fault-free, then the results of either output can be selected as the error-free output results. However, if the two output results disagree, it indicates the presence of faults in the FT systolic array. The effect of the faults can be masked by selecting the output results of output 2 in the next clock cycle. Similarly, outputs 3 and 5 can be compared and depending on the status of the second error output, the fault-free output results in this case, are selected. The concurrent error correction procedure can be summarized as follows:

Let e_1 be the error output 1 and e_2 be the error output 2 (Fig.6.5(a)). Also, O_i represents the output result of output i.

(1) Compare O_1 and O_6

If $e_1 = 0$, select O_1 or O_6

Otherwise, if $e_1 = 1$, select O_2 in the next cycle.

(2) Compare O_3 and O_5

If $e_2 = 0$, select O_3 or O_5

Otherwise, if $e_2 = 1$, select O_4 or O_6 in the next cycle.

The concurrent error correction is accomplished by using the error correction circuit. This can be placed on-chip or off-chip and embody the logic for correcting errors. Issues such as how to tolerate faults in the error correction circuit (that is how to test the tester), are avoided to focus on the basic ideas of the technique. However, since the results are produced in triplets, errors in the FT systolic array of Fig.6.5(a) can also be masked dynamically using fault-tolerant voters.

6.2.3 Procedure for Designing Area Efficient FT Systolic Architectures

In this subsection, we present the following two strategies to design a FT systolic array architecture using our schemes :

A Method One

1. In the first method, the CED systolic array is constructed.
2. It is then duplicated to produce four sets of output results.
3. Finally, concurrent error correction is determined.

B Method Two

1. In the second method, the TDM of a given algorithm is determined and then mapped into a systolic architecture.
2. The second version of the algorithm is the redundant version of the first one but in CRC mode.
3. A third version of the algorithm is determined by rotating the systolic array in step 2 by 180 degrees about any of the principal axis (horizontal, vertical or diagonal axis).
4. The three systolic arrays are merged to construct the FT systolic array.
5. Add extra hardware only in those cells of the FT array that would compute the output results for the independent computations or wavefronts.
6. The input data of some of the variables in the algorithm are re-scheduled to satisfy their original input data scheduling. Extra time is required for the completion of the com-

putations and this corresponds to the maximum number of extra delay elements introduced in the interconnection path of any of the data variables.

6.3 FAULT-TOLERANT ANALYSIS OF THE PROPOSED DESIGN SCHEMES

In this section, we will illustrate how faults that occur in the FT systolic arrays can be tolerated by using the two fault-tolerance design methods described in this chapter. From the analysis of how and which faults are tolerated, then, one can have an idea of the percentage of the fault coverage of the design scheme.

For the proposed scheme, a functional fault model has been assumed. Therefore, our interest lies basically in detecting and correcting a corrupt output, to ensure the validity of the final result. A fault that does not produce an error, or in other words, that does not affect the functionality of the module under test, will not be detected. It is important to note also that the output results will be fault-free since no error is produced by the fault.

The fault-tolerant systolic array designed by employing the scheme described in the first method is shown in Fig.6.1 . As mentioned before, this FT array is constructed using two CED systolic arrays of Fig.5.5. In Fig.5.5, for instance, the output result of one of the coefficient a_{11} is calculated in cells (2,1), (3,2) and (4,3), while the alternate result is calculated in cells (4,1), (5,2) and (6,3). A discrepancy in the two results indicates a fault or malfunction (temporary or permanent) either in any of the functional blocks of any of these cells or in any of the interconnection lines in the systolic array or in any of the added delay elements. Similarly, the results of, for instance, a_{12} are calculated with the duplicated functional blocks in cells (3,1), (4,2) and (5,3). In this case also, a fault either in any of the functional blocks in these cells or in any interconnection line in the array will produce a discrepancy in the two answers. Since the error detecting circuits are totally self-checking, hence any single fault in the CED systolic array or a faulty component in the error detecting logic itself would be detected. Therefore all single faults in

the systolic array of Fig.5.5 are detectable. Also, all single faults in the copy of the CED systolic array would be detected in a similar fashion as in the original one.

Four sets of output results are produced by the FT systolic array of Fig.6.1 and each set of the output results is computed using separate functional blocks. These four output results are voted on using fault-tolerant majority voter circuits. Since the voter circuit is fault-tolerant, the combination of the four copies of the output results produced by the FT array and the output voter can tolerate any single fault in the FT systolic array or a faulty component in the voter itself. Therefore, all single faults in the FT systolic array of Fig.6.1 can be tolerated by dynamically masking the effects of the faults.

So far, we have considered the results of single faults. A number of multiple fault patterns can also be detected and masked by this design scheme, provided that the corresponding output results of two or more independent computations are not affected by the multiple fault pattern. For example, a multiple fault pattern which comprises single faults in cells (3,1), (4,2), (5,3), (4,1), (5,2) and (6,3) in Fig.5.5, produce a detectable and maskable error pattern, since no corresponding output results of a_{ij} would be affected by the fault pattern. However, it is important to note that a multiple fault pattern that affects two corresponding output results may cause both results to have the same erroneous values. In this case, the error detecting circuit would not indicate a discrepancy in the two answers and hence, such multiple fault patterns cannot be detected. Also, since the erroneous values of these two output results would be different from the values of the other two fault-free outputs, thus, a majority decision would not be reached at the output and hence, such multiple fault pattern cannot be tolerated. We have illustrated the cases where a multiple fault pattern can either be detected and corrected or not detected and not tolerated. It is possible to have cases where the multiple fault pattern can either be detected but not tolerated or vice versa. To illustrate the case where the fault pattern can be detected but not tolerated, consider a single fault in cell (2,1) (Fig.5.5) of one copy of the CED array. If the same fault occurs in cell (2,1) of the other copy of the CED array

in Fig.6.1, this multiple fault pattern is detectable. However, it cannot be tolerated since this will not produce a majority vote at the output of the FT systolic array. The case where the fault pattern cannot be detected but can be tolerated, corresponds to when a multiple fault pattern does not produce an error or does not affect the functionality of the circuit under investigation. It is important to note however, that the probability of occurrence of these multiple fault patterns is very low.

The CED and the error masking strategies employed in this method, focus on a global mechanism for fault-tolerance. The proposed method does not focus on local error detection or local error masking. As a result, it does not provide error location capabilities. We are not interested in locating faulty PE's, rather we want to mask the effects of faults that occur in these PE's. Due to the fact that fault location is not provided by this scheme, it may not be integrated with some reconfiguration algorithms. However, since errors are dynamically masked at the output of the fault-tolerant systolic array, reconfiguration algorithm is not necessary.

Furthermore, the fault-tolerant systolic array designed using the scheme of the second method is shown in Fig. 6.5(a) Figure 6.5(a) consists of two types of cell structures. In the first type (type I cell), the computational units in the cells are not duplicated, while they are duplicated in the second type (type II cell). The type I cells correspond to those that compute part of the output results of only two independent computations (e.g. cells (2,1), (5,1)). On the other hand, type II cells compute part of the output results of the three independent computations (e.g. cells (4,1), (5,2), (6,3)). As seen from Fig.6.5(a), the output results of all the coefficients a_{ij}, except those of a_{i2}, (for the three versions of the algorithm) are computed on separate functional blocks. For instance, the result of coefficient a_{11} for the three versions, are respectively computed in cells ((2,1), (3,2), (4,3)), ((3,1), (4,2), (5,3)) and ((5,1), (6,2), (7,3)). The output results of a_{i2} for two independent computations are computed in cells (4,1), (5,2) and (6,3). Since the computational unit in these cells are duplicated, the two results are computed using separate

functional blocks. Therefore, the output results of the three versions of the algorithm are computed in the FT systolic array of Fig.6.5(a), using separate functional blocks of the cells but at different times.

Also, a set of the input data for variable B is used to compute the output results of the two versions of the algorithm, as shown in Fig.6.5(a). Any fault that occurs in the B - data path will corrupt the results of two independent computations. Hence such faults cannot be tolerated or corrected by the correction circuitry. However, in designs where the data paths are conservatively designed, faults in these elements may be much more unlikely and can essentially be treated as hardcore [5]. In order to truly tolerate faults in this B - data path, the input data of the B variable for this two independent computations should be duplicated. Another possibility to achieve fault-tolerance in the B - data path is to employ error correcting codes [27,28]. If we assume that the error correction circuit is fault-free, and from the above analysis, then all single faults in the FT systolic array of Fig.6.5(a) can be tolerated. Also in addition to permanent faults, intermittent errors or temporary faults are detected and corrected by these design schemes.

Like the design scheme proposed in method one, this scheme can also tolerate a number of multiple fault patterns, provided that the corresponding output results of two independent computations are error-free. For instance, consider a multiple fault pattern consisting of single faults in cells (2,1), (3,2), (4,2), (4,1), (5,2) and (6,3). This multiple fault pattern will cause only one set of the output results of (a_{11},a_{21},a_{31}) or (a_{12},a_{22},a_{33}) or (a_{13},a_{23},a_{33}) to be erroneous. Since the majority of the output results of these elements are fault-free, hence the multiple fault pattern can be tolerated. This example demonstrates the case when the multiple fault pattern can be detected and corrected.

In order to illustrate the case when the multiple fault pattern can be detected but not corrected, consider a single fault in cells (3,2) and (4,2) respectively. It is assumed that the fault in cell (4,2) affects the computation unit used to compute the output results of elements a_{11},a_{21} and a_{31}. The three copies of the results of these elements are respec-

tively produced at outputs O_1, O_2 and O_6 of the FT systolic array of Fig.6.5(a). The results from outputs O_1 and O_2 will be faulty due to the occurrence of the multiple fault pattern, while those from output O_6 will be fault-free. The comparison of the corresponding results of outputs O_1 and O_6 will detect the discrepancy in the results. Since the results of output O_2 are to be selected when those from O_1 and O_6 disagree, thus, a faulty output result is selected as the fault-free one. Hence, this multiple fault pattern can be detected but not tolerated.

Also, it is possible to have the case where the multiple fault pattern can neither be detected nor corrected. This case can be illustrated by considering a combination of single faults in cells (4,3) and (7,3). If the corresponding output results of the elements a_{11}, a_{21} and a_{31}, from outputs O_1 and O_6 have the same erroneous values, the comparison of the results from these two outputs will not indicate any discrepancy. Either of the faulty outputs will be selected and hence, this multiple fault pattern can neither be detected nor tolerated. Although, some of the multiple fault patterns cannot be detected nor tolerated, however, a great majority of them will be tolerated as long as they affect only one copy of the three copies of the output results produced by the fault-tolerant systolic array of Fig.6.5(a).

6.4 AREA AND TIME OVERHEAD OF THE PROPOSED SCHEMES

It is a well known practice in VLSI system design, there is always a trade-off involved between the silicon area and the desired efficiency in terms of throughput and speed. Our motivation is to propose a scheme which is applicable to different systolic array implementations without causing any loss in throughput. This goal is achieved by our scheme but not without some overhead in silicon area and time.

In the first method, the silicon overhead includes the additional functional blocks introduced only in those cells that are required to compute the corresponding output

results of the two independent computations. It also includes the delay elements used to synchronize the flow of input data into the CED systolic array, the duplicate of the CED array and the fault-tolerant voters. The error detection circuitry is not necessary since the errors at the output of the array are masked by the redundant voters. As regards to the time, since the two independent computations are launched into the CED systolic array at the same time, there comes a time when a computational resource would be requested by the two different computations at the same time. In order to resolve this conflict, extra time is introduced by delaying one computation (using the delay elements) until the other has completely utilized the computational resource. Therefore in this respect, we can say that this scheme involves overhead in time. For instance, in Fig.5.5, one extra clock cycle is required to complete the computation of the two independent results. The number of the extra clock cycles required corresponds to the maximum number of the delay elements introduced in the path of the interconnection line of any of the variables in the algorithm. This is exemplified in Fig.5.5 where an extra delay element is introduced in the path of variable B to synchronize the propagation of data into the CED array. Since the duplicated CED arrays produce their output results at the same time, hence only one (rather than two) clock cycle is required to complete the computation of the output results.

In the second method, the silicon overhead includes, the additional functional blocks introduced only in those cells that are required to compute part of the output results of the three versions of the algorithm. It also includes the delay elements used to synchronize the flow of the input data into the FT systolic array, and the fault-tolerant voters or the error detection and correction circuits. Two delay elements are introduced in the interconnection lines of variable B. Here, the number of the extra clock cycle required is given by the (number of delay units + 1), hence, three extra clock cycles are required to complete the computation of the three independent computations.

In Fig.6.5(a), from the way by which the input data for variable A are re-scheduled,

it appears that one computation is launched after the other. Consequently, at most twice the number of clock cycles (required by the irredundant array) would be required to complete the computation of the three independent results. The systolic array of Fig.6.2 requires 7 clock cycles to complete the computation of one version of the algorithm. From the nature of the data scheduling in Fig.6.5(a), a total of 13 clock cycles would have been required to complete the computation of the three independent output results. However, the CED systolic array of Fig.5.5 requires only 8 clock cycles to complete the two computations. While the FT systolic array of Fig.6.5(a) requires only 10 clock cycles to complete the computation of the three independent computations. Hence, given the nature of the data flow into the CED (Fig.5.5) and FT (Fig.6.5(a)) systolic arrays after re-scheduling the input data, we can conclude that our scheme does not involve any overhead in time. There is , of course, no loss in throughput. One of the advantages of these schemes is that though the input data is re-scheduled, the corresponding output results arrive at the output of the array almost at the same time. This facilitates the comparison and voting on the output results without the introduction of any additional control or synchronization circuitry.

In the following section, we will compare the two proposed methods of designing fault-tolerant systolic arrays. Also, we will compare our FT schemes with other FT design schemes.

6.5 COMPARISON OF THE PROPOSED FT SCHEMES OF THE TWO METHODS

The FT systolic array of Fig.6.1 designed using the technique described in the first method (example 1, section 5.2.3), consists of 18 processing elements (PE's) and 12 delay elements. Figure 6.5(a) depicts the FT systolic array designed using the second method. It consists of 12 PE's and 12 delay elements. By taking into account the extra interconnection lines and extra area required to route data into the duplicate CED array in

Fig.6.1, it can be observed that the FT scheme of the second method has less area overhead than that of the first method. However, the FT scheme of the first method has less time overhead than that of the second one. It utilizes only one extra clock cycle to complete the computation of the output results, while the second method requires three extra clock cycles. Although the above comparison is based on an example, it reflects a general result.

Also, four copies of the output results are produced by the FT design scheme of the first method, while three copies of the output results are produced by the scheme of the second method. Hence, more work is done and hence more energy is consumed by the FT systolic array designed using the first method than that designed employing the scheme of the second method. Therefore, there is a trade-off between the FT schemes of the two methods. However, in general, the FT scheme of the second method offers better advantages than that of the first method.

6.5.1 Comparison of Our FT Schemes With Other FT Schemes

Several fault-tolerant schemes for systolic arrays have been proposed in the literature [3-26]. The approaches proposed in [3-14] use reconfiguration techniques to achieve fault-tolerance in systolic arrays. In [3], the proposed systolic fault-tolerant scheme maintains the original data flow pattern by bypassing defective cells with few registers. Other approaches to reconfigure VLSI systolic arrays have also been presented in [4-14]. The difference between the approaches lies in the reconfiguration strategy employed by each approach. However, in general, the problem of how to tolerate the defects once they are located is addressed in each approach. Therefore concurrent error correction cannot be achieved using these techniques. Also, the fault detection problem, requiring a totally different set of techniques such as voting and self testing is not discussed. In most cases, such fault detection techniques are specific to a given architecture and may not be applicable to another architecture. Additionally, the techniques may be expensive in terms of

area and time overheads and as such cannot be applied in real-time systems. Also, the approach proposed in [3] can only tolerate faults in the computation units once the faults are located. It cannot tolerate faults in the registers and the interconnection links.

The fault-tolerant methods proposed in this chapter do not use reconfiguration schemes, hence we will compare our design schemes against other relevant schemes. In table 6.1, we compared the complexity and performance of the schemes proposed in this chapter with those shown in table 4.1. Two fault-tolerant methods have been presented in this chapter. Table 6.1 consists of the comparison of the complexity of these fault-tolerant methods. For ease of comparison, the hardware and time redundancy ratios of the existing and proposed schemes are presented in table 6.2. The proposed technique in method one of this chapter, requires a hardware redundancy ratio of a factor of 1 (i.e. 100%), and a time redundancy ratio of O(N-1 / 3N-2). For large values of N, the time redundancy ratio is about 33%. The technique employed in method two requires a hardware redundancy ratio of about 2% and a time redundancy ratio of less than 33%. It can be seen that the technique of method one has the same time redundancy ratio as that of method two. However, the hardware redundancy ratio of the scheme in method two is much less than that proposed in method one.

The time redundancy approach of achieving fault-tolerance, proposed in chapter IV (method one), requires hardware and time redundancy ratios of 0% and 50% respectively. While the spatial redundancy approach (TMR, method two), requires hardware and time redundancy ratios of 200% and 0%. The FT approach (TMR) proposed by Von Neumann [15] requires a hardware redundancy ratio of 200%. There is no time overhead required for computations since the replicated arrays perform the computations at the same time. The technique, like all TMR techniques, is very general and can be applied to any level in a parallel system.

For the FT technique proposed in [16-18], the structure of the resulting FT systolic array is not highly regular and granular. More silicon area is required to route data into

Category	Proposed Approach (Method 1)	Proposed Approach (Method 2)
FT Technique	Dual Redundancy (Duplication) Approach	Error Detection/Masking Approach
Complexity without FT techniques	Processor: N^2 Time: 3N-2	Processor: N^2 Time: 3N-2
Redundancy required with the technique	Processor: N^2 Time: N-1 Voter: $\lceil N/2 \rceil$ Buffer: 4N TSC: - Other: -	Processor: 2N Time: N Voter: - Buffer: 4N TSC: $\lceil N/2 \rceil$ Other: Error correction circuit
Redundancy Ratio	Processor: $O(N^2/N^2)$ Time: $O((N-1)/(3N-2))$	Processor: $O(2N/N^2)$ Time: $O(N/(3N-2))$
Diagnosis Performance	Faults - types: Permanent & Temporary - # allowed: Single fault & certain multiple fault patterns - detection: Yes - location: No	Faults - types: Permanent & Temporary - # allowed: Single fault & certain multiple fault patterns - detection: Yes - location: Possible
Utilization	On-line global error masking.	On-line global error correction, or error masking.

Table 6.1 Comparison of the complexity and diagnosis performance of the fault-tolerant techniques proposed in Methods 1 and 2 of Chapter VI.

FT Techniques	Hardware Redundancy Ratio	Time Redundancy Ratio
Von-Neuman [15]	$O(2)$	0
Huang & Abraham [16-18]	- dense matrix multiplication $O(2/N)$ - band matrix multiplication $O(2/W_1)$	- dense matrix multiplication $O(((2*r*\log_2(N))+T_{cor})/N)$ - band matrix multiplication $O(((2*r*\log_2(N))+T_{cor})/N)$
Jou & Abraham [19]	$O(((2N/1)+2)/N)$	$O((\lceil N/1 \rceil+T_{cor})/N)$
Kim & Reddy [20]	$O([N(3m)-(W_1*W_2)]/(W_1*W_2)$ if $W_1=2m-1$ $O([N(3m+1)-(W_1*W_2)]/(W_1*W_2)$ if $W_1=2m$	$O(1)$
Cosentino [21]	$O(1/N)$	$O(2N/(3N-1))$
Cosentino [22]	$O(2N^2/3N^2)$	$O(T_{r-b-c}/2N)$
Ari & Friedlander [23]	$O(P/(2N+1)^2)$	$O(T_p/(4N+2))$
Kung & Manolakos [26,27]	$O(1/N)$	$O((N+1)/(2N-1))$
Varman & Ramakrishnan [24]	$O(K/N\sqrt{N})$	$O(2K/(4N\sqrt{N}-N-3\sqrt{N}))$
Kumar & Tsai [25]	$O(K/N\sqrt{N})$	$O(2K/(3N\sqrt{N}-2))$
Proposed Approach (Method 1, Chap.IV)	0	$O(2N/(4N-3))$
Proposed Approach (Method 2, Chap.IV)	$O(2)$	0
Proposed Approach (Method 1, Chap.VI)	$O(1)$	$O((N-1)/(3N-2))$
Proposed Approach (Method 1, Chap.VI)	$O(2/N)$	$O(N/(3N-2))$

Table 6.2 Comparison of the hardware and time redundancy ratios of the existing and proposed schemes.

the array. As a result, the hardware redundancy may approach 100%. The error correction is done off-line. Extra time is required to correct the errors after they are located. For error detection, the time redundancy ratio is about 4%. However, for error correction, this ratio increases and may approach 50%. In the case of the technique proposed in [19], if the array size is much larger than the word length of the weighted summation elements, the hardware redundancy ratio is about 100%. When the extra time required for error correction is added to the time redundancy for computations of the matrix elements, the time redundancy ratio will be greater than 100%. The hardware redundancy ratio of the approach proposed in [20] is over 100%. Since the throughput of the array is reduced by half, hence, the time redundancy ratio may be about 100%.

The approach proposed in [21] requires a hardware redundancy ratio of 1% (for large values of N). The time redundancy ratio is over 66%. In [22], the hardware redundancy ratio required by the scheme is 300% while the time redundancy ratio is 100%. For the approach proposed in [26,27], the hardware and time redundancy ratios are respectively 1% and 50%. The hardware and time overhead required by the scheme in [23] depend on the diagnostic expression chosen for error detection. If the checksum error detection scheme is selected as the diagnostics expression, then, the hardware and time redundancy ratio will be similar to those in [16-18], that is, 100% and 50% respectively. The hardware and time overhead required by the reconfiguration schemes in [24,25], depend on the error detection techniques and the number of faulty processing elements in the systolic array.

Table 6.3 consists of the summary of the percentage hardware and time redundancy ratio required by the various fault-tolerant schemes to perform matrix multiplication in systolic arrays. As observed in table 6.3, the FT approaches proposed in [21] and [26,27], require less hardware redundancy than the scheme proposed in method two of this chapter. Although, their hardware redundancy ratio is 1% while that of the proposed scheme is 2%, nonetheless, the values are comparable. However, in [21], each row of the

FT Techniques	Percentage Hardware Redundancy Ratio (N=100) (%)	Percentage Time Redundancy Ratio (%)
Von-Neuman [15]	200	0
Huang & Abraham [16-18]	100	50
Jou & Abraham [19]	100	100
Kim & Reddy [20]	100	100
Cosentino [21]	1	66
Cosentino [22]	300	100
Ari & Friedlander [23]	100	50
Kung & Manolakos [26,27]	1	50
Varman & Ramakrishnan [24]	-	-
Kumar & Tsai [25]	-	-
Proposed Approach (Method 1, Chap. IV)	0	50
Proposed Approach (Method 2, Chap. IV)	200	0
Proposed Approach (Method 1, Chap. VI)	100	33
Proposed Approach (Method 2, Chap. VI)	2	33

Table 6.3 Summary of the comparison of the percentage hardware and time redundancy ratios required by various FT schemes to perform matrix multiplication in systolic arrays.

FT systolic array requires an error correction circuitry, hence, the number of such circuits increases with the size of the array. In our case, only a single error correction circuit is required. If the number of the error correction circuits is taken into account, this could increase the hardware overhead in [21] considerably. The scheme in [26,27] can only correct transient faults, correction of permanent faults require additional hardware. Even though, the schemes proposed in [21] and [26,27] have slightly better hardware redundancy ratio than our scheme, the proposed scheme has the best hardware and time redundancy ratio combination, 2% and 33% respectively. Therefore, our approach provides a low cost technique to detect and mask errors in processor arrays.

In addition to possessing better figure of merits in terms of the combination of the hardware and time redundancy ratios, our design scheme overcomes most of the problems of the other schemes. For instance, the proposed scheme can be applied to any systolic array implementation. It is not restricted to a class of systolic arrays in which the partial results must stay in the cells as in [21]. Also, it is neither restricted to implementations where the data as well as the (sub) results move from one cell to another as in [20]. There is no halving of the maximum effective output rate, that is, there is no reduction in throughput by 50% as in the schemes proposed in [20,21,26,27]. The proposed scheme is effective for transient and permanent faults, while the schemes in [16-18, 26,27] are effective for transient faults. Error detection and error masking are performed with the normal operation of the system (i.e. on-line error masking), while in [16-19], the error correction is done off-line.

The proposed scheme can detect and mask all single (both permanent and transient) faults and most multiple fault patterns in the FT systolic array. In [20,21,23], the schemes cannot detect or correct faults in the input/output registers and the interconnection lines for data transfer. The scheme in [22] cannot correct faults in the data paths, binary-to-residue and residue-to-binary conversion circuits, and the error calculator. The diagnosis and correction latency of the proposed scheme is not very long as in the case of the

schemes in [16-19,22,23]. Also, the schemes in [16-19,22,23] are vulnerable to false alarms brought about by roundoff errors. In our scheme, there is no roundoff errors introduced in the computations of the matrix multiplication algorithm. The redundant computations produce exact output results if there is no fault in the array. A discrepancy exists only in the presence of a fault.

In the proposed approach, redundant hardware is added only at those computational resources that need to compute three (two) independent computations at the same time. This, in most cases, results to area efficient systolic designs. Furthermore, the FT design approach is systematic as opposed to the approaches in [20,23] which are not systematic. Finally, the structure of the FT systolic array resulting from our design scheme is regular and granular. This is not the case with the schemes proposed in [16-19,23], where the non-regularity of the structures even depends on the array size.

The proposed design scheme detects and masks the effects of faults at the output of the FT systolic array, it cannot locate faults in the array. Also, is not possible to reconfigure the array. However, since our fault-tolerant design scheme is based on error masking approach, thus, fault location and array reconfiguration are not required.

6.6 CONCLUDING REMARKS

In this chapter, we have presented fault-tolerant (FT) schemes for VLSI systolic array architectures. From the comparison of the two proposed methods, it is observed that the second method has better advantages than the first method and hence, it is the preferred technique for designing FT systolic array architectures. This scheme offers solutions to several problems of the other known FT schemes. It is applicable to a wide class of VLSI implementation of algorithms. The technique is not limited to only those systolic implementations where the data as well as (sub) results keep moving. It can also be applied to those implementations where either the data or the (sub) results are stored. It

has a feature of exploiting the advantages of the interleaved computations without any reduction in the throughput of the array structure. The scheme can tolerate all single permanent and temporary faults and a majority of the multiple fault patterns. It is area efficient, there is no need to replicate all the hardware in the FT systolic array. Redundancy is introduced only at those places where it is needed. The silicon area is limited to one row of extra functional blocks in those cells that compute the output results of the two or more independent computations, delay elements and the error correction logic. The computation of the output results of the two or more versions of the algorithm can be scheduled to start at the same time with both using the same computational space but at different times. Although the input data is re-scheduled, the corresponding output results of the independent computations arrive at the output of the array almost at the same time. This facilitates the comparison of the results without the introduction of any additional control or synchronization circuitry. Consequently, there is no high delay cost incurred and thus, the technique can be employed in real-time applications. This scheme can be applied to any systolic array structure (e.g. 2D, tree - connected etc.) and also to other types of VLSI arrays.

6.7 REFERENCES

[1] M. O. Esonu, A. J. Al-Khalili and S. Hariri, "Area Efficient Architectures for Concurrent Error Detection in Systolic Arrays," *IEEE Int'l Conf. on Parallel Processing (ICPP '91)*, Chicago, August, 1991.

[2] H. F. Li, C. N. Zhang and R. Jayakumar, "Latency of Computational Data Flow and Concurrent Error Detection in Systolic Arrays," *CCVLSI '89*, pp. 251-258, 1989.

[3] H. T. Kung and M. S. Lam, "Fault-Tolerance and Two Level Pipelining in VLSI Systolic Arrays," *MIT Conference on ADV Research in VLSI*, pp. 74-83, Jan. 1984.

[4] F. T. Leighton and C. E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Trans. Computers,* Vol. C-34, pp. 448-461, May 1985.

[5] I. Koren and M. A. Breuer, "On Area and Yield Considerations for Fault-Tolerant VLSI Processor Arrays," *iEEE Trans. on Comput.*, Vol. C-33, No. 1, pp. 21-27, Jan., 1984.

[6] P. J. Varman and I. V. Ramakrishnan, "A Fault-Tolerant VLSI Matrix Multiplier," *ICPP*, pp. 351-357, August, 1986.

[7] T. Ishikawa, S. Momoi, S. Shimada, Y. Ogawa, "Hierarchical Array Processor (HAP) Featuring High Reliability and High System Performance," *ICPP*, pp. 293-300, August, 1986.

[8] F. Lombardi, R. Negrini, M. G. Sami, and R. Stefanelli, "Reconfiguration of VLSI Arrays: A Covering Approach," *FTCS*, pp. 251-256, July, 1987.

[9] H. F. Li, D. Pao and R. Jayakumar, "Dynamic Reconfiguration for Fault-Tolerant Systolic Arrays," *ICPP*, pp. 110-113, August, 1987.

[10] D. L. Landis, W. A. Check and D. C. Muha, "Influence of Built-In Self-Test on the Performance of Fault-Tolerant VLSI Multiprocessors," *ICPP*, pp. 114-116, August, 1987.

[11] M Hou and H. T. Mouftah, "Fault-tolerant System Using 3-Value Logic Circuits," *IEEE Trans. on Reliability*, Vol. R-36, No.2, pp. 227-231, June, 1987.

[12] J. M. Char *et al.*, "Distributed and Fault-Tolerant Computation for Retrieval Tasks Using Distributed Associative Memories," *IEEE Trans. on Comput.*, Vol. 37, No. 4, pp. 484-490, April, 1988.

[13] J. A. Abraham, P. Banerjee, C-Y Chen, W. K. Fuchs, S-Y Kuo and N. Reddy, "Fault - Tolerance techniques for systolic arrays," *IEEE Computer*, pp. 65-74, July 1987.

[14] J-H Kim and S. M. Reddy, "On the Design of Fault-Tolerant Two-Dimensional Systolic Arrays for Yield Enhancement," *IEEE Trans. Comput.*, Vol. 38, No. 4, April 1989.

[15] J. V. Neuman, "Probabilistic Logics and Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, No. 34, pp. 43-99, Princeton, NJ : Princeton University Press.

[16] K-H Huang and J. A. Abraham, "Low Cost Schemes for Fault Tolerance in Matrix Operations with Processor Arrays," *Proc. 9th Symp. on Computer Architecture*, pp. 330-337, May, 1982.

[17] K-H Huang and J. A. Abraham, "Fault-Tolerant Algorithms and their Application to Solving Laplace Equations," *IEEE Int'l Conf. Parallel Processing*, pp. 117-122, August, 1984.

[18] K-H Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Comput.*, vol. C-33, No. 6, pp. 518-528, June, 1984.

[19] J-Y Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE*, vol.74, No. 5, pp. 732-741, May, 1986.

[20] J-H Kim and S.M. Reddy, "A Fault-Tolerant Systolic Array Design using TMR Method," *1985 ICCD*, pp. 769-773.

[21] R. J. Cosentino, "Concurrent Error Correction in Systolic Architectures," *Proc. IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 1, pp. 117-125, January 1988.

[22] R. J. Cosentino, "Fault Tolerance in a Systolic Residue Arithmetic Processor Array," *IEEE Trans. on Comput.*, vol. 37, No. 7, pp. 886-890, July, 1988.

[23] H. Lev-Ari and B. Friedlander, "On the Systematic Design of Fault-Tolerant Processor Arrays with Application to Digital Filtering," *1988 VLSI SIGNAL PROCESSING III*, pp. 483-494, 1988.

[24] P. J. Varman and I. V. Ramakrishnan, "Optimal matrix multiplication on fault-tolerant VLSI array," in *Proc. ICALP*, 1985.

[25] V. K. Prasanna Kumar and Y-C Tsai, "On mapping Algorithm to Linear and Fault-Tolerant Systolic Arrays," *IEEE Trans. Comput.*, Vol. 38, No. 3, pp. 470-480, March 1989.

[26] S. Y. Kung, VLSI Array Processors, *Prentice Hall*, 1988.

[27] E. S Manolakos, "Transient Fault Recovery Techniques for the VLSI Processor Arrays," Ph.D thesis, University of Southern California, May, 1989.

[28] F. J. Aichelmann, Jr., "Fault-Tolerant Design Techniques for Semiconductor Memory Applications," *IBM J. RES. DEVELOP.*, Vol. 28, No. 2, pp. 177-183, March, 1984.

[29] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Tech. J.*, Vol-29, No. 1, pp. 147-160, Jan., 1950.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

In this thesis, one of our goals has been to provide a cost-effective systematic approach for mapping algorithms into optimal systolic arrays. In particular, we have considered and provided solutions for the following problems:

(i) the direct method to produce a transformed dependency matrix (TDM) of an algorithm.

(ii) the problem of identifying a unifying performance index to measure the overall systolic array performance (speed, cell's complexity, number of interprocessor connections, practical design considerations, etc.)

(iii) the formulation of suitable and more realistic optimality criteria for the design of systolic arrays.

(iv) the formulation of the Compound Objective Function (COF) to measure the cost of each TDM obtained directly from the original dependency matrix (DM).

(v) the problem of developing an optimization algorithm for designing optimal systolic arrays.

An important problem associated with the design of systolic arrays is the mapping of algorithms into systolic architectures. In this thesis, using the space-time mapping technique, we have proposed a new methodology to obtain the desired TDM directly from the original dependency matrix, so as to avoid deriving the TDM's that do not satisfy our VLSI requirements. By so doing, we can eliminate many TDM's and also avoid deriving their corresponding transformation matrices. This technique has the advantage of reducing the computation complexity.

As demonstrated before, in the thesis, several TDM's that meet the important VLSI requirements can be generated using the proposed methodology. As a result, it becomes necessary to select the TDM that will yield the optimal systolic array based on the cost function. The choice of the optimality criteria is very important. By using those optimality criteria that take into consideration the practical design issues like the architectural features and technological parameters of the array, realistic optimal systolic arrays can be designed. We have identified the more realistic optimality criteria for designing systolic arrays and also, have derived the expressions that represent their cost measures. We established the relationship between the cost measures and the coefficients or elements of the TDM. Thus by changing the coefficients or the elements of a new generated TDM, the values of the cost parameters are affected. This enables us to investigate the practical design issues at the design stage of the systolic array rather than after the implementation of the array.

A Compound Objective Function (COF) is formulated based on these factors that determine the optimality criteria. Since optimizing one factor may sacrifice other factors, it becomes worthwhile to measure the cost of a TDM with respect to all the factors rather than one factor. In this dissertation, we proposed a *unifying performance index* to allow us to measure the overall performance of the systolic array. The COF function which is the performance index, is modulated to have the dimensions of AT^2. This does not indicate that the only cost factor in the COF function that is used to measure the performance of the array is the area x square of time. Rather , all the parameters or factors in the COF function (which is a complex function of fault-tolerance, interconnection delay, area x time squared and speedup) are taken into consideration in the measure of the overall systolic array performance. This approach eliminates the problem of sacrificing many other factors of the optimality criteria when only one factor is optimized.

Furthermore, we formulated the design of systolic arrays into an optimization problem. An optimization algorithm for designing optimal systolic arrays is developed. Since

changes in the coefficients of the new TDM are reflected in the cost parameters, therefore, the optimization procedure has to determine how these changes affect the cost parameters and which change optimizes the Compound Objective Function. Consequently, the coefficient values of the new TDM can be changed to reflect the importance of some of the cost factors for a given algorithm. The advantages of the optimization algorithm proposed in this thesis can be summarized as follows:

(i) It is computationally efficient, only the TDM's that satisfy some of the important VLSI requirements are generated.

(ii) The procedure allows the designer to investigate some practical design issues such as fault-tolerance, speedup, etc. at the design stage rather than the implementation stage. Thus, by exporting the practical design problems to the design stage, efforts will not be wasted in designing feasible but sub-optimal systolic arrays.

(iii) Since the approach is systematic, it can easily be automated. The designer can specify the algorithm for a specific application as an input to the optimization routine and then an optimal VLSI systolic array implementation of the algorithm can be generated. This in other words, will be like a silicon compiler for systolic arrays.

Because of these advantages, the approach proposed in this thesis provides an efficient method for mapping algorithms into optimal systolic arrays than the presently available approaches.

During the course of this research, not only have we made contributions in the design of optimal systolic array architectures (which is one of our goals), but also in the design of reliable systolic arrays. Reliability of systolic arrays can be increased through fault-tolerance. In this thesis, we have developed a methodology for designing fault-tolerant systolic array architectures using space-time mapping techniques. The basic concept of our approach of designing fault-tolerant systolic arrays, involves the introduction of redundancy at the algorithmic level, such that when these algorithms are mapped into systolic VLSI array architectures, certain degree of hardware availability are inherent in these architectures that allow concurrent error detection and correction in the systolic

architectures. The procedure followed in the proposed approach includes the derivation of different dependency matrices corresponding to the different versions of algorithm and the application of space-time techniques to obtain a fault-tolerant systolic array implementation of that algorithm.

The approach proposed in this thesis is different from the conventional approach of designing fault-tolerant systolic architectures, which is based on the mapping of an algorithm onto a specific VLSI systolic architecture , and then modifying the design to make it fault-tolerant. In most cases, the techniques employed to make the particular systolic architecture fault-tolerant is specific only to that systolic architecture and thus may not be applicable to any other systolic array with different topology and data flow characteristics.

We have contributed towards the development of a methodology to design Concurrent Error Detection and Fault-Tolerant systolic arrays. Several fault-tolerance techniques have been presented based on our approach. The basic idea of each technique is that independent redundant computations can be launched into a concurrent error detection or fault-tolerant systolic array at the same time. Two or more identical sets of output results (one for each version of the algorithm) may need to be computed at the same computational site and at the same time. By identifying such computations, extra hardware can be added only at those computational sites at which they will be computed. Then by re-scheduling the input data such that the original input Data Scheduling is satisfied, the interaction between the multiple wavefronts can be isolated. Since the independent computations are performed on different computational sites, thus concurrent error detection or correction can be achieved by comparing corresponding sets of computed output results for discrepancies. We have demonstrated in this thesis some methods of achieving CED or FT in systolic arrays. Also, the advantages of our techniques over the existing ones have been pinpointed. One of the major advantages of our approach is the area efficiency. There is no need to replicate all the hardware in the CED or FT systolic array, redundant hardware is introduced only at those computational sites where it is

needed. Therefore, based on this methodology, we have proposed a new systematic approach for designing reliable systolic array architectures.

7.2 FUTURE WORK

It is in the nature of research that the solution of one problem often gives rise to many new issues or questions or open problems. In most cases, these questions lead to the improvement of the present research. However, in some cases, the open problems culminate into a new ground of research. In the case of the research which is presented in this thesis, the following issues need to be investigated:

Systolic Array Mapper :

We have proposed a unifying performance index to measure the overall performance of the systolic array architecture taking into consideration the architectural features and the technological parameters of the array. The proposed procedure is formulated as an optimization problem to obtain the TDM with the minimum cost function. The generation of the desired TDM's and the selection of the optimal one (given the cost function) is presently performed manually. There is a need to develop a software package in order to facilitate the performance evaluation procedure and hence, the mapping of any given algorithm with constant dependence vectors into an optimal systolic array architecture.

Rearrangement of data flow :

In order to isolate the interaction between components of the different versions of the algorithm in a CED or FT systolic array, the input data is re-scheduled such that the original input data scheduling is satisfied. The re-scheduling is presently done in an adhoc manner. It will be useful to develop a mathematical analysis or representation of the propagation of the input data of the variables of the algorithm, after re-scheduling.

Multi-dimensional Systolic Arrays :

Present practical arrays have pure planar layouts. That is, the conceptual sites of a computation structure $\{(\bar{J} = J^1, J^2, J^3)\}$ must be mapped into $Z^3 = \{(t,x,y)\}$. Recently, vertical processes are becoming available, thus increasing the available implementation space. It will be interesting to investigate the design of multi-dimensional systolic array architectures (i.e. for $n > 3$, e.g. $n = 4$) with indices $\{(\bar{J} = J^1, J^2, J^3, J^4)\}$. This approach can be used to design layered systolic architectures which can be employed in the design of super systems.

Minimization of TDM's search space :

Currently, we have proposed a methodology for deriving the transformed dependency matrix (TDM) directly from the original dependency matrix. Although the methodology is clear, but, since several TDM's that satisfy the VLSI requirements can be generated, a limit has been imposed on their search space, in the optimization algorithm. The search for a new valid TDM is stopped when the change in the improvement of the performance of further generated TDM's, is significantly small. It will be ideal if a systematic approach can be found to limit the search space of the valid TDM's.