This is to certify that the

dissertation entitled

ALGORITHM IMPLEMENTATION AND DESIGN OF RECONFIGURABLE MIXED SYSTOLIC ARRAYS

presented by

Anwar Khurshid

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Electrical Engineering

P. David Fisher
Major professor

Date June 4, 1985

ALGORITHM IMPLEMENTATION AND DESIGN OF RECONFIGURABLE MIXED SYSTOLIC ARRAYS

By

Anwar Khurshid

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering
and Systems Science

1985

ABSTRACT

ALGORITHM IMPLEMENTATION AND DESIGN OF RECONFIGURABLE MIXED SYSTOLIC ARRAYS

By

Anwar Khurshid

One drawback of systolic architectures is their fixed-flow structure for data streams, which limits the type of algorithms or applications that can effectively be supported by such architectures. This thesis presents a methodology for algorithm implementation and design of a class of reconfigurable multiprocessor architectures called the mixed systolic array (MSA). In the MSA architecture, switching cells are mixed with computation cells to achieve flexibility in the data-flow patterns. This architecture broadens the scope of systolic arrays by achieving reconfigurability, algorithmic flexibility and fault tolerance. The level sensitive scan design (LSSD) technique is employed to load and implement the distributed control structure required to establish a desired interconnection pattern on the MSA. The control structure for a particular configuration is loaded into the array as a binary vector in a bit serial fashion. This approach enhances testability and incorporates fault tolerance in the MSA structures.

Efficient implementation of algorithms on VLSI structures requires exploitation of parallelism in the algorithm and mapping of the algorithm communication structure into the processor interconnection structure. This thesis presents a general mathematical model for formally representing reconfigurable MSA architectures and a step-by-step procedure for implementing a given algorithm into the MSA structure by generating the control code required to reconfigure the array. The mapping procedure is based on time and space transformations of the data dependence vectors of the algorithm. These transformations provide a description of the data-flow and timing, and dictate the interconnection structure required to implement the algorithm on the array. The procedure presented in this work, will provide a useful tool in the design automation of reconfigurable MSAs. To illustrate the methodology and explain the reconfiguration procedure, two sample algorithms, the finite impulse response (FIR) filtering algorithm and the priority queue algorithm, are mapped into a linear reconfigurable systolic array. A computer-aided design (CAD) facility is also presented, for modeling and simulating mixed systolic arrays. This CAD facility serves as a high-level design tool which supports the design of MSA architectures, and provides the designer of MSAs with a facility to interactively develop an MSA structure for a given set of user-specified attributes and simulate the execution of algorithms on MSA processors at the register-transfer level.

To

MOTHER AND FATHER

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr. P.D. Fisher, for his continuous moral and professional support and guidance throughout my doctoral program. He was a rich source of inspiration, and I gratefully acknowledge the encouragement and motivation provided by him which made possible the successful conclusion of my Ph.D. work. I also would like to thank Dr. Michael Shanblatt for his valuable suggestions and evaluation of this research. I am very thankful to Dr. Lionel Ni for his professional advice and review of this work. The interest and encouragement provided by Dr. S.R. Crouch is gratefully acknowledged. Appreciation is extended to Dr. M.A.P. Jayasumana for his help during this research. I am thankful to Mr. B. MacArthur for his skillful drawings. The support provided by the National Science Foundation (Grant No. MCS 79-09216) for this research is gratefully acknowledged. I shall always be grateful to my parents for their unending support and encouragement through the years. Finally, I am specially thankful to my wife, Nadira, for her patience and understanding in awaiting the completion of this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Rapid advancements in VLSI technology demand innovative computational algorithms and hardware structures which fully exploit the technology to achieve high throughput rates and efficient resource utilization. Efficient implementation of algorithms on VLSI structures requires exploitation of parallelism in the algorithm and mapping of the algorithm communication structure into pipeline, vector or array processor interconnection structures. The main issues in VLSI design are those of modularity, simplicity of communication and control, and extensibility. A modular design with a large number of identical modules organized in a simple regular fashion is an ideal structure for VLSI.

In recent years, systolic arrays have been at the focus of attention of many researchers as pipelined multiprocessor structures suitable for solving a variety of computation-intensive and real-time problems requiring high throughput [2,4,31,32,41]. In the systolic concept, VLSI devices consist of arrays of interconnected primitive processors with distributed control and a high degree of modularity. Each processor operates on a string of data that flows regularly and rhythmically through the array. Systolic arrays feature the important properties of modularity, regularity, locality of interconnection and highly pipelined

multiprocessing. However, one drawback of systolic architectures is their fixed-flow structure for data streams, which limits the type of algorithms or applications that can effectively be supported by such architectures. Therefore, it is desirable to have reconfigurability in the data-flow structure and flexibility in the algorithm implementation to make more general purpose arrays.

The mixed systolic array (MSA) is a class of reconfigurable multiprocessor architectures, introduced by Chang and Fisher [6-8]. In this architecture, control elements are mixed with computing elements according to a certain mixing profile and the data-flow patterns are determined by the distributed control structure stored in the control elements. Classes of algorithms with similar data requirements may be executed on the same array by merely presetting the control elements at load time. MSA architectures broaden the scope of systolic arrays and at the same time preserve VLSI design attributes such as locality of communication, modularity, extensibility and simplicity of control. While Chang and Fisher developed, characterized, and evaluated the basic computing model for MSAs [6], they did not present procedures for loading and implementing the distributed control structure required to establish a desired interconnection pattern on the MSA. Also, no formal methodology was presented for implementing a given algorithm into the MSA structure by systematically generating the control code required for its reconfiguration.

The goal of this research is to investigate structured methodologies for mapping parallel algorithms into reconfigurable MSA architectures,

and implementing the distributed control structure required for the algorithm implementation. Another goal is to exploit the emerging VLSI and Wafer-Scale Integration (WSI) technologies by designing computer architectures, which employ modularity in structure, simplicity and regularity in communication and control paths, and extensibility in design. From a more general standpoint, this research broadens the scope and enhances the applicability of special-purpose VLSI array processors, and at the same time contributes to the understanding of the problems and nature of a parallel processing approach to computation. The specific tasks are outlined as follows:

1. Investigate the procedures for loading and implementing the distributed control structure required to establish a desired interconnection pattern on the MSA structure.
2. Relate the MSA's architectural model with the parallel algorithms implemented on the array and investigate the procedures for implementing algorithms into MSA structures by systematically generating the required control code for reconfiguration.

This research investigates structured methodologies for designing and implementing mixed systolic arrays. The Level Sensitive Scan Design (LSSD) technique [12,25,57] is employed to load and implement the distributed control structure required to establish a desired interconnection pattern on the MSA. The control structure for a particular configuration is loaded into the array as a binary vector in a bit-serial fashion. This approach enhances testability and incorporates

fault tolerance in the MSA structure. This thesis presents a general mathematical model for formally representing reconfigurable MSA architectures and a step-by-step procedure for implementing a given algorithm into the MSA structure by generating the control code required to reconfigure the array. The mapping procedure is based on time and space transformations of data dependence vectors of the algorithm. These transformations provide a description of the data-flow and timing, and dictate the interconnection structure required to implement an algorithm on the array. A computer-aided design facility for modeling and simulating MSAs is designed and partially implemented. This facility helps the designer of MSAs to interactively develop an MSA structure for a given set of user-specified attributes, such as mixing density and array geometry, and to simulate the execution of algorithms on MSA processors at the register-transfer level.

The procedures presented in this thesis, for mapping algorithms onto MSAs and for implementing the control structure necessary for reconfiguration, provide a useful tool in the automated design of reconfigurable MSAs. For instance, in order to design an MSA to implement a set of algorithms, one can start with a high-level language description of algorithms and use the approach presented in this thesis to find a suitable MSA which can implement these algorithms. Usually, many valid time and space transformations are generated in the procedure, providing the designer flexibility to choose the ones which map easily on the array. The procedures are especially useful during the application of an MSA processor. Whenever a new configuration of the MSA is desired

for a new algorithm, the host computer can use the procedure to generate the reconfiguration control vector for the algorithm to be implemented. This control vector can be loaded into the MSA processor to reconfigure the array on-the-fly. The procedure can also be used to determine whether or not an algorithm can be implemented on a given mixed systolic array.

Throughout, we use symbols I and Z to denote the set of all natural numbers and the set of all integers, respectively. A^n denotes the nth cartesian power of a given set A, i.e., the set of all possible n-tuples of elements of A. Chapter 2 reviews some background information and related research work regarding systolic architectures, parallel algorithms, and LSSD techniques. In Chapter 3, design of a one-dimensional MSA architecture, called the Linear Reconfigurable Systolic Array (LRSA), is presented which employs LSSD techniques to achieve reconfigurability and multifunctionality. This approach is extended to two-dimensional arrays in Chapter 4, which describes a Multipurpose Reconfigurable Array Processor (MRAP) architecture for implementing various systolic and semisystolic algorithms. Also, performance of the MRAP is analyzed in Chapter 4, for various matrix-multiplication algorithms based on their Space-Time-Bandwidth complexity. Chapter 5 presents a mathematical formalism for modeling reconfigurable MSA architectures and a methodology for reconfiguring the array by mapping the communication structure of an algorithm into the interconnection structure of the array. Two sample algorithms, the finite impulse response (FIR) filtering algorithm and the priority queue

algorithm, are mapped into a linear reconfigurable systolic array in order to illustrate the reconfiguration procedure. Chapter 6 contains a summary of this research work and some thoughts for future research possibilities. Finally, a computer-aided design facility for modeling and simulating MSAs is described in Appendix A.

CHAPTER 2

BACKGROUND

2.1 SYSTOLIC ARRAYS

Several types of VLSI architectures have been proposed in recent years, such as the systolic arrays [31-35], the Wavefront Array Processor [39], mixed systolic arrays [6-8] and the Configurable Highly Parallel (CHiP) computer [53,54]. Most of these parallel architectures attempt to match the underlying hardware to specific algorithms for fast and efficient execution. Systolic array architectures are particularly attractive for VLSI implementation because of their regular, short and simple communication geometry [44,45]. This section discusses the characteristics of systolic array architectures, systolic algorithms, and semisystolic design.

2.1.1 SYSTOLIC ARRAY ARCHITECTURES AND ALGORITHMS

Systolic array architectures are multiprocessing systems in which data is pipelined among processors by using next-neighbor communication. Systolic algorithms are defined as mathematical algorithms which are implementable with systolic architectures. A number of special purpose systolic arrays, suitable for VLSI and wafer-scale integration (WSI), have been proposed for solving various computation-intensive problems

[4,18,35,38]. Their applications range from numerical problems, such as signal and image processing and matrix arithmetic, to non-numerical tasks, such as searching and sorting, graph algorithms and relational databases.

A sytolic array comprises a network of interconnected cells, where each cell is capable of performing a small set of operations and has some local memory and control logic. Strictly next-neighbor type of connections constitute the interprocessor communication structure and data moves through the architecture in a synchronous pipelined manner. Communication with the outside world takes place only at the array boundary, possibly through some special I/O processors. The following design criteria for systolic arrays have been suggested: First, the design should use only a small number of different types of simple cells. Second, these cells should be interconnected by a network with short, regular connections. Third, multiple use of each input data item should be made in order to achieve high computation rate and throughput without requiring high memory to array bandwidth. And, finally, computational algorithms should be employed which exploit both data pipelining and parallel execution.

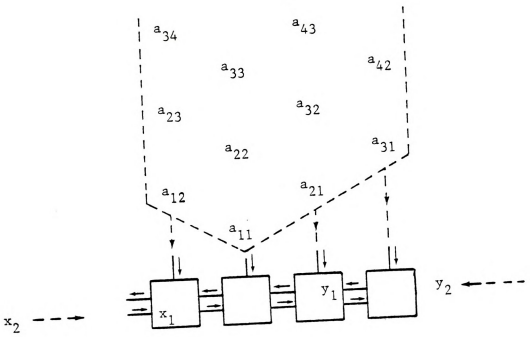
The principles of systolic array design are illustrated here by considering two examples of systolic arrays. A linearly connected systolic array, shown in Figure 2.1 [31], uses W processing elements to multiply an NxN band-matrix, with bandwidth W by a vector of N elements. The basic processing element is called the inner-product step processor (see Figure 2.2), and consists of three internal registers RA, RB and RC. This processing element performs the inner-product operation

C <--- C + A * B; where A, B and C are the contents of the registers RA, RB and RC, respectively. The time to compute the entire multiplication is 2N+W time units. The input and output vector elements march in opposite directions, so that each input vector element meets all the output vector elements before it leaves the array. The same inner-product step processor is used in a hexagonal systolic array to implement band-matrix multiplication. Figure 2.3 illustrates the multiplication algorithm for two NxN band-matrices with bandwidths W_1 and W_2, respectively [31]. This algorithm requires $W_1 x W_2$ inner-product step processors and takes $3N+\min(W_1,W_2)$ units of time for the computation. Several other computation-intensive algorithms can be solved on systolic arrays such as LU decomposition, triangular linear systems, convolution, filter, and discrete Fourier transforms.

Many implementation alternatives exist for systolic array processors providing different interconnection topologies and degrees of flexibility. According to their degree of flexibility, systolic array processors can be classified as follows [34]:

1. Single-purpose systolic arrays [2,4,38]. In this approach, a systolic array is built to implement only one algorithm and a different array needs to be designed for each new algorithm. This approach is reasonable if the performance of the processor is of ultimate importance and the processor is to be used in large quantities despite the fact that it is single-purpose.

2. Multi-purpose systolic arrays [58]. A systolic array processor



$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & \\ & 0 & & & \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \end{bmatrix}$$

A x y

Figure 2.1. Linearly connected systolic array for computing the product of band-matrix A and vector X.

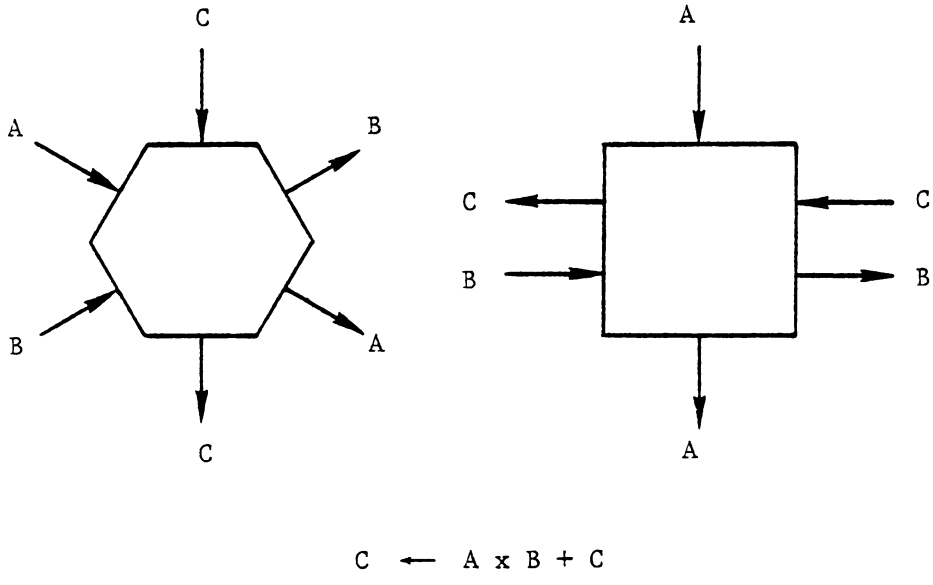


Figure 2.2. Two geometries for the inner-product step processor.

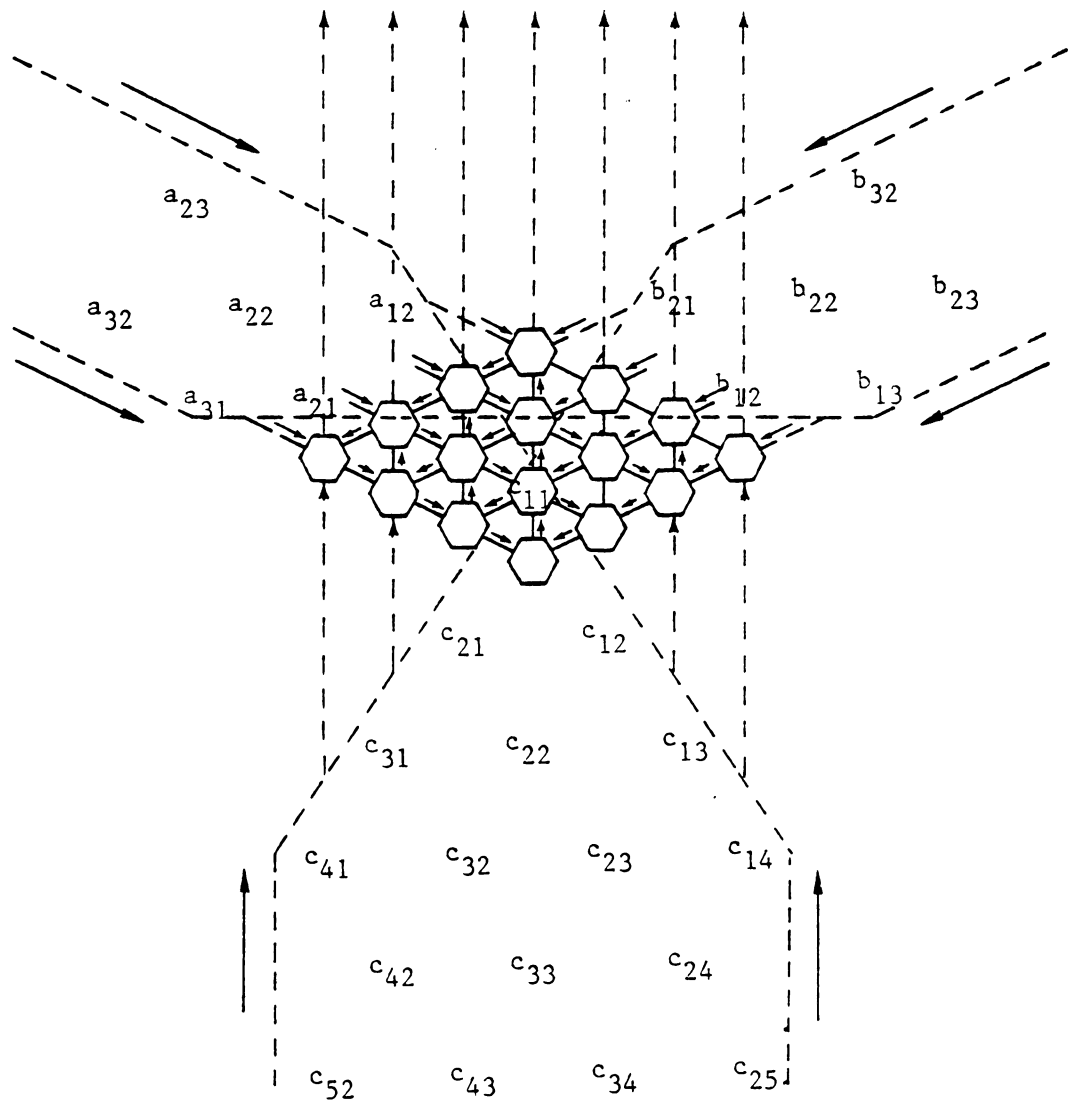


Figure 2.3. Hex-connected systolic array for computing the product matrix C, of matrices A and B [44].

of this type can implement a predefined set of algorithms. The control overhead for providing necessary flexibility should be kept small and the VLSI attributes, such as modularity and locality of communication, should be preserved in the design.

3. Non-programmable building blocks [29]. Building-block processors are constructed which can execute a few predefined commonly used functions. These blocks are connected to form a variety of systolic array processors of different sizes and shapes.

4. Programmable building blocks [14]. The building block is a programmable processor which can be programmed to implement a large family of systolic cells. This approach is not very efficient because of the overhead for supporting the programmability. However some systolic algorithms involving complicated data dependencies, such as greatest common divisor computation [2], can be effectively implemented on this type of arrays.

5. Programmable systolic arrays [3,7]. In this approach, programmable processing elements are mixed with other control units in a certain manner. These arrays are more flexible than the multi-purpose systolic arrays in the sense that the processing elements are programmable and their interconnections can be configured by software control before a computation

starts.

Various interconnection topologies for systolic arrays can be defined according to the number of computations performed for each input/output operation. First, for two-dimensional systolic arrays shown in Figure 2.4(a), O(n) processing elements perform computations in each cycle, whereas $O(\sqrt{n})$ boundary cells perform input/output operations. Thus the computation over input/output ratio is $O(\sqrt{n})$. Systolic arrays for matrix arithmetic algorithms are included in this class [37]. Second interconnection topology can be defined as degenerate two-dimensional systolic arrays shown in Figure 2.4(b), in which O(n) processing elements perform computations and O(n) elements perform input/output operations. So the computation over input/output ratio is O(1) in this case. Systolic arrays for solution of triangular linear systems and orthogonal transformations are examples of degenerated two-dimensional arrays [19,37]. Finally, linear or one-dimensional systolic arrays shown in Figure 2.4(c), perform the input/output via the two processing elements at the ends of the array, and thus the computation over input/output ratio is O(n) for an array of size n. Systolic arrays for filtering or pattern matching come under this category [32,37]. Linear arrays may be preferred over two-dimensional arrays in situations where the input/output bandwidth, between the host system and the systolic array, is a major limiting factor for achieving high performance.

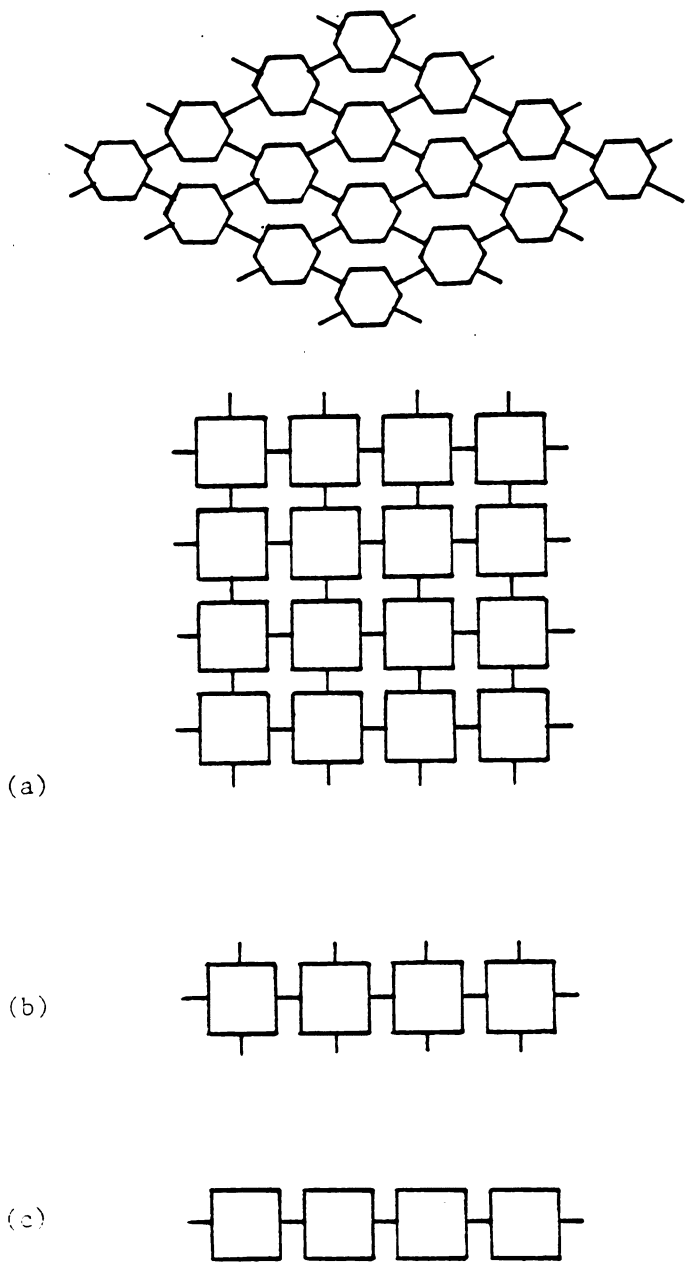


Figure 2.4. Some common interconnection schemes for systolic arrays:
(a) two-dimensional systolic arrays,
(b) degenerate two-dimensional systolic array,
(c) one-dimensional systolic array.

2.1.2 SYSTOLIC AND SEMISYSTOLIC DESIGN

Systolic array architectures do not allow any global data communication and all communication between processing elements is clocked through a register. Semisystolic array architectures, on the other hand, allow global data communication and a data item may be broadcast to many processing elements simultaneously. Mathematically, the structure of a systolic system S(n) is given by a machine graph G = (V,E) of n interconnected Moore machines, where the vertices in V represent the machines and the directed edges in E represent interconnections between the machines [41]. The machines operate synchronously by means of a common clock, and time in the system is measured as the number of clock cycles. A semisystolic system is similar to a systolic system except that some of the machines may be Mealy machines with the condition that the output edges from Mealy machines may not form a cycle in the machine graph [41]. Mealy machines can implement data broadcasting, whereas Moore machines can not. Figure 2.5 shows an example of a semisystolic array which can implement data broadcasting. In this example, the combinational logic for Mealy machines is a simple wire from input to output. The exclusion of Mealy machines in systolic systems makes the clock period independent of the system size.

Semisystolic systems do not meet the design criterion of extensibility as systolic systems do. That is, many semisystolic arrays can not be cascaded together to form an arbitrarily large array, because the clock cycle time, which depends upon the delay due to broadcasting or rippling of logic, may asymptotically become arbitrarily large. In

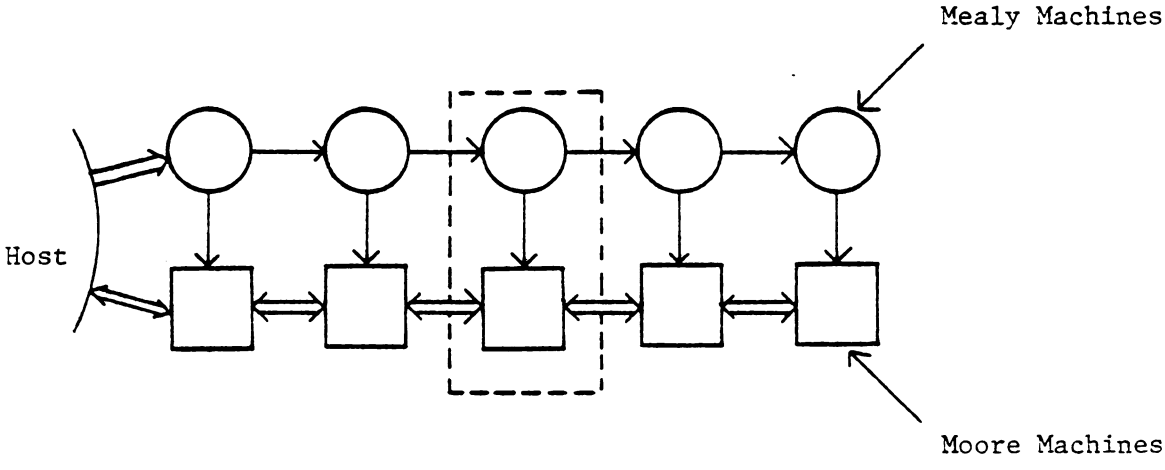


Figure 2.5. A semisystolic system can implement broadcasting by Mealy machines [41].

general, there are two main disadvantages associated with data broadcasting. First, large broadcasts can not be implemented in a single communication cycle, because the broadcast delay can dominate the execution time of an algorithm. Second, large drivers are required for broadcasting in order to drive the combined load of all the cells connected to the broadcasting bus. Two techniques, retiming and slowdown, are described in [41] and [42] for converting a semisystolic design into a systolic one by eliminating broadcasting. For many systolic algorithms, some fraction of the processors are always idle at a given time. For example, in FIR filtering, half of the processors are idle at each clock tick and in band-matrix multiplication, two third of the processors are idle at each clock tick [44]. Leiserson proposed coalescing and interlacing techniques for improving processor utilization [41]. For specific algorithms, introduction of data broadcast concept results in more efficient parallel algorithms and better processor utilization [9,10,21]. Huang and Abraham [21] have compared the efficiencies of systolic and semisystolic arrays for matrix multiplication algorithms. They concluded that, for specific computations, semisystolic arrays perform better than systolic arrays according to the Space-Time-Bandwidth complexity criterion.

2.2 RECONFIGURABLE MIXED SYSTOLIC ARRAYS

The mixed systolic array (MSA) is a class of reconfigurable multiprocessor architectures, first introduced by Chang and Fisher [6-8]. In this architecture, control elements are mixed with computing

elements according to a certain mixing profile and the data-flow patterns are determined by the distributed control structure stored in the control elements. The computing elements are programmable multifunctional arithmetic and logic processors, whereas the control elements are programmable interconnection networks which establish the communication structure of the array. Classes of algorithms with similar data requirements may be executed on the same array by merely presetting the control elements at load time. In this thesis, we refer the computing element and the control element as the computation cell and the switching cell, respectively.

The mixed systolic array provides programmable interconnection structure by way of mixing switching cells with computation cells in a systolic fashion. Its main objective is to broaden the scope of systolic arrays by achieving reconfigurability, algorithmic flexibility and fault tolerance. Another objective is to preserve the design attributes of modularity, uniformity, locality of communication and simplicity of control, in order to exploit the very large scale integration (VLSI) and wafer scale integration (WSI) technologies. The computation cells, in the MSA, are programmable multifunctional arithmetic and logic processors, which process the incoming data according to the control codes stored in their control registers. The switching cells are programmable interconnection networks which establish the interprocessor communication structure in the array to meet the communication requirements of the algorithms implemented. A switching cell directs the data-flow among its neighboring computation cells according to the interconnection configuration defined by its

control code register. An MSA executes a specific algorithm according to the control codes stored in individual cells. The control code registers of computation cells determine the computational structure of the MSA and their contents are determined by the basic computational requirements of the algorithm. The control code registers of switching cells establish the communication structure of the MSA, and their contents are determined by the communication requirements of the algorithm. In order to configure an MSA to implement a specific algorithm, the control code corresponding to that particular algorithm is loaded into the array, and the array performs the execution in a synchronous manner. Whenever a new application of the same MSA is needed, a new control code can be loaded into the array for its reconfiguration.

Basically, the structure of an MSA is determined by its mixing profile, which establishes the possible frames of data-flow patterns within the array. An MSA structure is called a regular structure, when there is a basis or a subarray from which the MSA can be constructed. Figure 2.6 shows an MSA constructed from a diamond-like basis. Other possible MSA structures with irregular and partially regular mixing, are discussed in [6]. In this thesis, we are concerned only with the regular MSA structures, mainly because programming and mapping algorithms into irregular arrays are too complex, and most of the existing synchronous parallel algorithms map into regular array structures [4,35].

The mixing profile of an MSA is determined by its mixing density and boundary conditions. The mixing density, ρ, is defined as

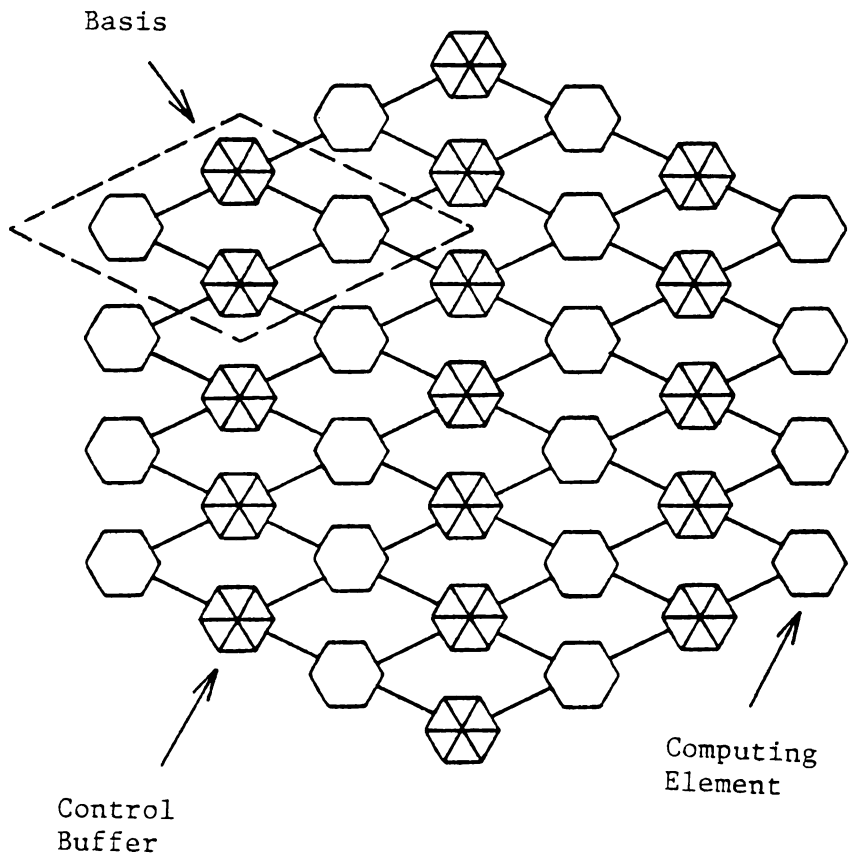


Figure 2.6. A mixed systolic array constructed from a diamond-like basis [6].

$$\rho = \frac{N_{sc}}{N_{sc} + N_{cc}} \tag{2.1}$$

where N_{sc} and N_{cc} are the number of switching cells and the number of computation cells, respectively. An MSA with high mixing density is suitable for implementation of algorithms which require more complex data routing and less computing. On the other hand, an array with low mixing density can be applied to algorithms which have simple data routing but require large amount of computing. When $\rho=0$, the MSA reduces to a fixed structure single-purpose systolic array, and when $\rho=1$, the MSA reduces to a reconfigurable interconnection network with no computing power. The boundary conditions in an MSA are represented by a boundary condition function, Ω, which is defined as

$$\rho = \frac{N_{sc'}}{N_{sc}} \tag{2.2}$$

where $N_{sc'}$ is the number of control buffers placed on the MSA boundary.

Various other general purpose reconfigurable VLSI architectures have been proposed such as Configurable Highly Parallel (CHiP) computer by Snyder [53], and the Programmable Systolic Chip by Fisher and Kung [14]. In the CHiP architecture, a lattice structure of programmable switches is incorporated, into which processing elements are placed at regular intervals. This architecture can implement various special purpose parallel architectures such as mesh structures and tree structures, under the supervision of a master controller. The controller broadcasts commands to all the switches to invoke a

particular configuration setting. This master controller may be undesirable for failure critical applications. In MSA architecture, neither a broadcasting of command signals nor a master controller is required; instead, the control structure is distributed among the switching cells and the computation cells of the array. While Chang and Fisher developed, characterized, and evaluated the basic computing model for MSAs [6], they did not present procedures for loading and implementing the distributed control structure required to establish a desired interconnection pattern on the MSA structure. Also, no formal methodology was presented for implementing a given algorithm into the MSA structure by systematically generating the control code required to reconfigure the array. The main motivation for this research work came from the above mentioned issues regarding algorithm design and implementation of MSAs.

2.3 ALGORITHM TRANSFORMATION AND MAPPING INTO VLSI STRUCTURES

Efficient implementation of algorithms on VLSI structures requires exploitation of parallelism in the algorithm and mapping of the algorithm communication structure into the array processor interconnection structure. An approach to design VLSI algorithms based on recurrences was first suggested by Cohen in [11] and later expanded by Johnnson and Cohen in [22], and Weiser and Davis in [56]. This approach is adopted from z-transforms in signal processing and uses delay operators (Z-operators) for specifying and representing sets of data as wavefront entities in the mathematical expressions.

Mathematical formulas are manipulated to obtain different expressions that correspond to different computational networks. The drawback with this approach is that the notation gets difficult to manage for complex computations. Leiserson and Saxe presented a general theory for optimizing a synchronous circuit by adjusting the number of register delays in the data paths [43]. This theory justifies some transformations used to eliminate broadcasting from semisystolic designs, but does not offer a methodology for systematically designing systolic arrays starting from a high-level language description of an algorithm.

Kuhn introduced the idea of exploiting parallelism in loops with multiple levels of nesting by reindexing the loop computations [30]. He used the concept of program dependence for detection and exploitation of parallelism in programs. He defined dependence as an arc in the dependence graph directed from the source occurrence of a variable to the destination occurrence of the same variable. Both the source and destination nodes are labeled by the values of the loop indices at which the generation and use of a variable occur, and the dependence arc is labeled by the difference vector of the source and destination labels. He mapped several specific algorithms into SIMD computers with single-stage interconnection networks and illustrated how the design of certain VLSI systolic arrays could be done automatically by reindexing algorithms. Moldovan [45-47] and Fortes [15] extended the approach of Kuhn, and formalized the procedure for mapping algorithms into VLSI architectures by transforming dependencies of the original algorithm by a reindexing transformation. Algorithm transformations comprise a time

transformation which dictates the order of execution of computations and a space transformation which determines the data movement, and the array size and geometry. Necessary and sufficient conditions for the existence of a certain type of transformation are given in [45]. This approach is best suited to algorithms described by programs with loops or by recurrence equations.

Some other related work on formalizing the design process of algorithmically specialized devices has been reported by Cappello and Steiglitz [5], Quinton [49], and Lam and Mostow [40]. Cappello's methodology is based on geometric transformations for mapping nested-loop algorithms into systolic arrays. The computation is modeled as a lattice in which nodes represent operations and edges represent data dependencies. Different systolic designs can be derived by applying geometric transformations to the lattice. Quinton's approach finds a uniform recurrent system of equations that is equivalent to the problem to be solved and maps this system of equations into a finite architecture. The methods given in [5] and [49], regarding the formalizing of systolic array design, are suitable for algorithms described by recurrence equations. Lam and Mostow described a design model, in which software transformations are first applied to put the algorithm to be implemented into a regular form conducive to systolic implementation. The algorithm is then mapped into a systolic design described by a structure and a driver. The structure describes the hardware cells and the driver defines data streams in terms of the original variables in the algorithm. This approach can process algorithms with simple FOR-loops and BEGIN-END blocks, but cannot deal

with conditional execution, computed iteration bounds and array indices. A survey of systematic approaches to the design of algorithmically specified systolic arrays can be found in [16].

2.4 FAULT TOLERANCE IN SYSTOLIC ARRAYS

Fault-tolerant systems are capable of performing correctly even in the presence of one or more faulty components. Fault-tolerant systems require some form of redundancy incorporated in their design. This redundancy could be either physical or temporal, or a combination of the two [51]. Physical redundancy is provided by replicating resources and may involve the use of extra gates, memory cells or functional modules. A taxonomy of fault-tolerance techniques and various stages of response to a system-failure in a fault-tolerant system are described in [51].

Fault tolerance in a pipeline architecture is very critical because a single fault in any segment would cause a total failure of the pipeline. Physical redundancy can be incorporated in pipeline architecture either at the pipeline level or at the segment (module) level. Reconfigurable parallel pipelines can be utilized to implement fault tolerance and achieve better performance in terms of average throughput, mean time to failure (MTTF), and mean computation before failure (MCBF)[24]. Initially the system utilizes all the pipelines in parallel, but as soon as one segment in a pipeline fails, the pipeline containing the failed segment ceases operation resulting in a degradation in performance. The other segments in the ceased pipeline become available as spares to mask subsequent faults in adjacent

pipelines. In this way, whenever the system fails, it does it with graceful degradation.

Although systolic array processing is a very efficient method of gaining increased system performance, this architecture is highly susceptible to faults. As a systolic array consists of many parallel pipelines, a single fault in any processing cell will propagate down the pipeline causing the system to fail. In two-dimensional systolic arrays, a single processing cell is shared by more than one pipelines, so the fault can propagate in multiple directions which makes the situation even worse. The above makes clear that the application of fault tolerance to a systolic array-based architecture is very critical. A fault-tolerant design of systolic arrays should enable an array to withstand one or more faults without total failure. This extends the life of the system and increases the mean time to failure (MTTF) and mean computation before failure (MCBF) for the system. Fault tolerance, of course, will require that redundancy be incorporated in the design of systolic arrays [20]. Temporal redundancy is not suitable because it adversely affects the system speed and throughput, which is not desirable if high performance is required. Physical redundancy can be introduced in the systolic array designs at the processing cell level or at the pipeline level. In response to a fault, the faulty cell or pipeline is discarded and replaced by another working one.

One method of designing fault-tolerant systolic arrays uses modular redundancy at the processing cell level. Triple modular redundancy (TMR) and N-modular redundancy will mask out the faulty outputs from a bad processing cell, but these are very costly techniques and their

application is limited to only critical short term uses [50]. Another fault-tolerant scheme, proposed in [36], enhances the yield of wafer-scale integration implementation of systolic arrays by replacing defective cells with clocked delays. This allows data to flow through the array with faulty cells at the original clock speed. Reconfigurable parallel pipelines for fault tolerance can be utilized in systolic arrays in order to achieve graceful degradation property. In case of no fault, all the pipelines are utilized and contribute to increased system performance. In case of faults, however, some pipes will shut down degrading the system performance. This makes many processing cells available to be used to mask out any further faults.

2.5 LEVEL SENSITIVE SCAN DESIGN (LSSD) TECHNIQUES

LSSD is IBM's discipline for structural design for testability [12]. In this concept, the memory elements or latches in an IC can be threaded together to form a serial-in, serial-out shift register. This provides an efficient means for "controlling" and "observing" the internal states of a machine with only three or four additional pinouts.

A key element in this design is the "shift register latch" (SRL) as shown in Figure 2.7 [57]. Since IBM has used the LSSD technique extensively, considerable attention has been given to the efficient implementation of LSSD latches such that the overhead due to complexity of SRLs is substantially reduced [13]. The lines D and C form the normal mode memory function while lines I, A, B and L2 comprise circuitry for the shift register function. The shift registers are

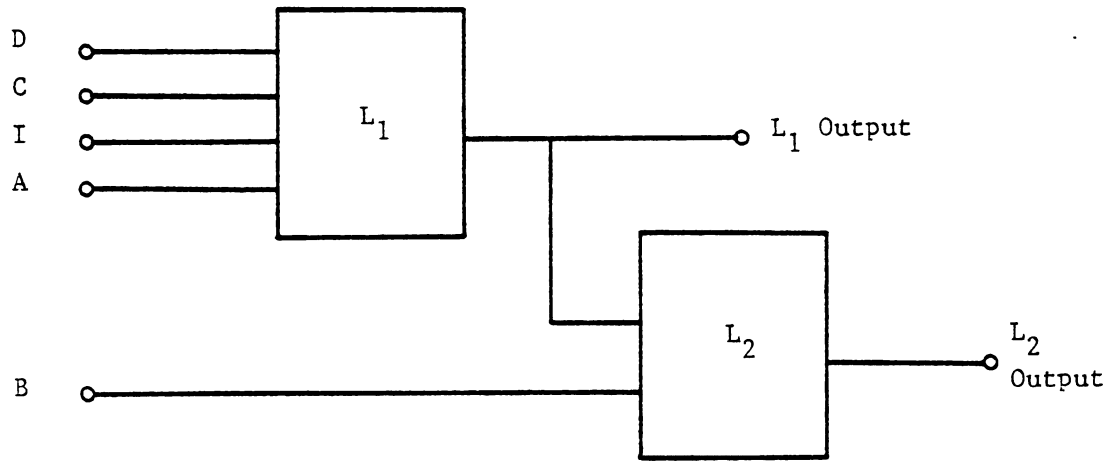


Figure 2.7. Block diagram representation of a shift register latch.

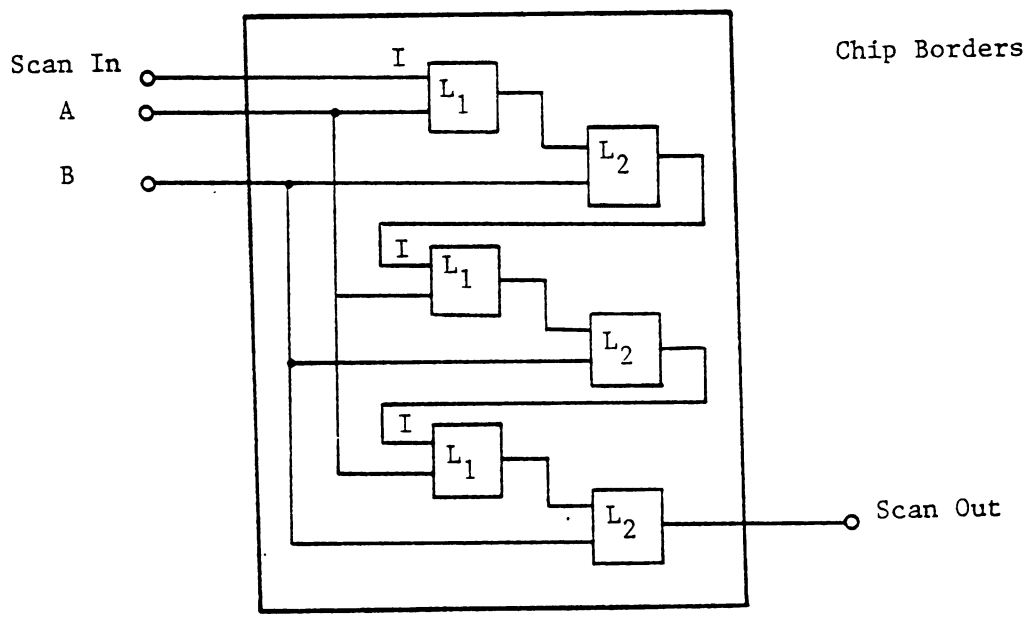


Figure 2.8. Interconnections of SRLs on an integrated circuit [57].

threaded by connecting I to L2 and operated by clock lines A and B in two phase fashion. Figure 2.8 illustrates how four SRLs can be threaded together for shift register action, and Figure 2.9 shows general structure of an LSSD subsystem with two clocks. Specific design rules and constraints concerning gating of clocks, etc., are given in [12].

LSSD techniques are employed in reconfigurable systolic architectures for loading and implementing the control structure required for reconfiguration [25]. The main advantage of using the LSSD technique is that overhead due to additional pinouts does not exceed three or four pins regardless of the size of the array. Shift register latches (SRLs) hold the control information. All SRLs are threaded together in a chain of shift registers in a manner such that a control vector entered in a bit-serial fashion sets up the array configuration for a particular algorithm.

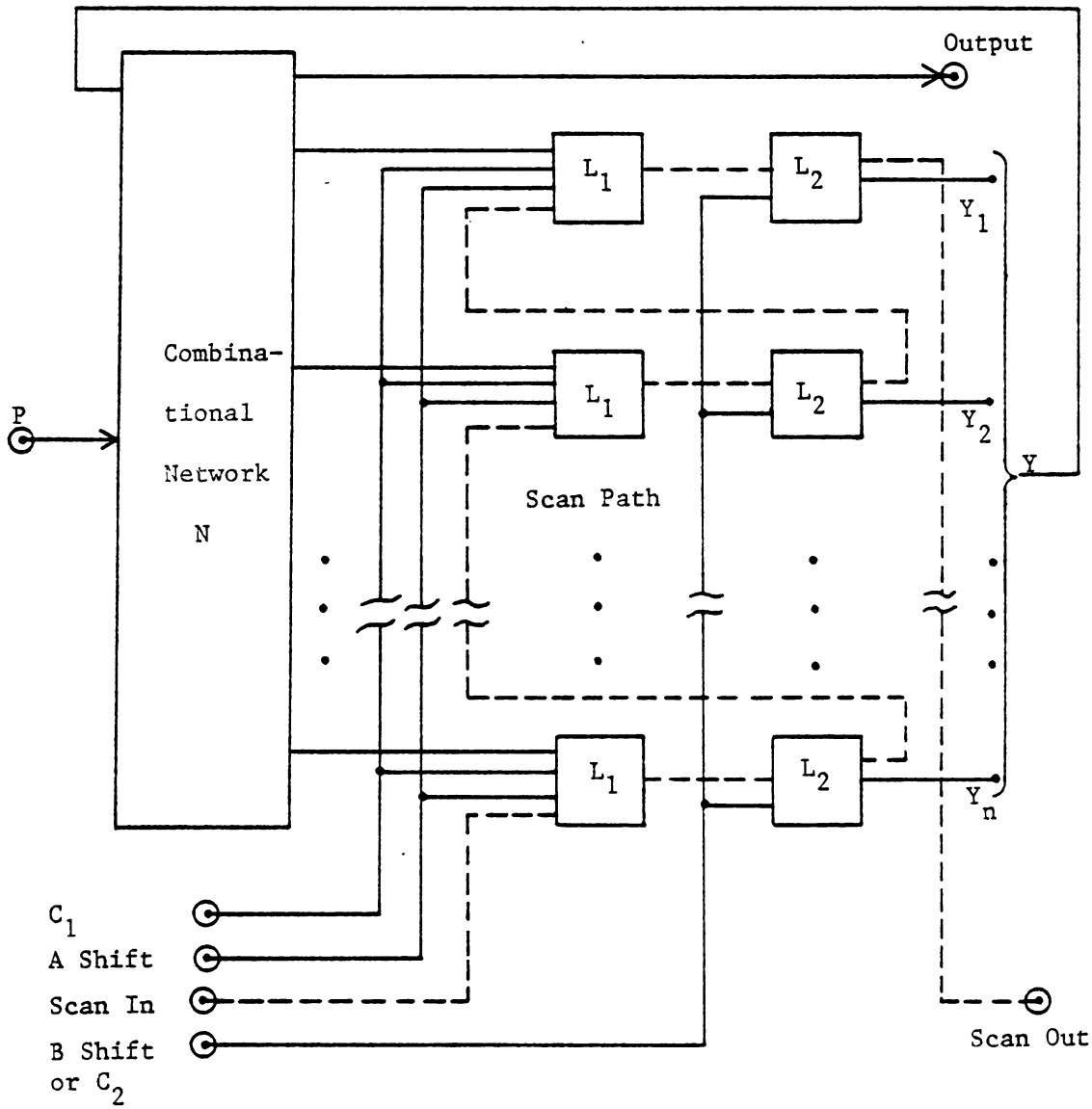


Figure 2.9. General structure of an LSSD subsystem with two system clocks [57].

CHAPTER 3

A LINEAR RECONFIGURABLE SYSTOLIC ARRAY (LRSA) ARCHITECTURE

The Level Sensitive Scan Design (LSSD) technique is employed in mixed systolic arrays to load and implement the distributed control structure required to configure the array for a desired algorithm implementation. To illustrate the concepts, this chapter presents the design of a one-dimensional linear reconfigurable systolic array (LRSA). This array structure is statically reconfigurable to realize any of the following: a filtering array, an FIR filtering array, a pattern matching array, and a Discrete Fourier Transform (DFT) array. What's more, the array structure is partitionable and can be divided into two or more independent subarrays, each capable of executing a preprogrammed algorithm. Shift register latches (SRLs) hold the control information for setting up the interconnections configuration and selecting the functional mapping of computation cells. Both the data flow through the array and the functions executed in the computational cells are established by inputting a control vector in a bit-serial fashion, using a two-phase clock. The next section describes the structure of the LRSA, and Section 3.2 characterizes four systolic algorithms which can be implemented on this structure.

3.1 STRUCTURE OF THE LINEAR RECONFIGURABLE SYSTOLIC ARRAY

The linear reconfigurable systolic array can be configured to implement one or more of the following types of systolic arrays: 1) a general filtering array, 2) a FIR filtering array, which has better resource utilization than the general filtering array, 3) a pattern matching array, and 4) a Discrete Fourier Transform (DFT) array. The systolic algorithms for the above mentioned arrays are described in Section 3.2. The LRSA architecture is configured, by simply loading a binary control vector through the LSSD scan input line, such that the computational and the data-flow structures of the algorithm are implemented on the array architecture. Once the array is configured, the architecture matches the algorithm structure exactly. Structure of the LRSA can be divided into three parts, the computation cells array, the function-select array and the data flow control array. Figure 3.1 illustrates a system block diagram of this structure, and each part is described in the remainder of this section.

3.1.1 THE COMPUTATION CELLS ARRAY

All computation cells are identical so the modularity is preserved in the design. This, of course, is an important factor in the efficient design and implementation of systolic arrays [22]. Each computation cell has three input data ports and three output data ports. These are all local ports except for the cells at the extreme ends of the array, where global I/O ports are used. Due to the local I/O ports the interconnections are regular, simple and short. Three control bits select the functionality of the computation cell. Figure 3.2 shows a

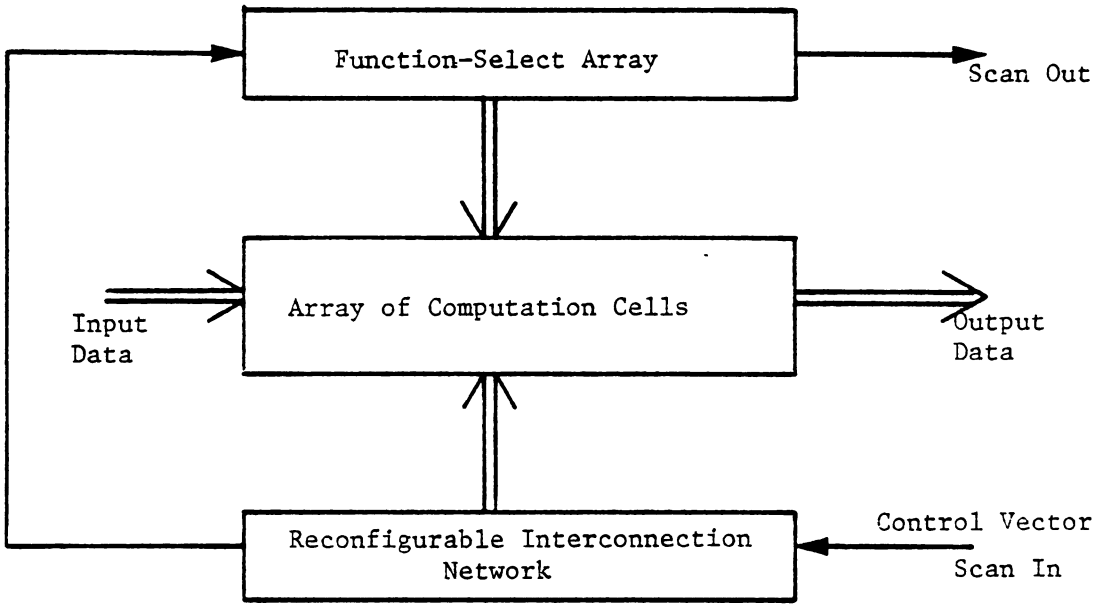


Figure 3.1. General system diagram of the Linear Reconfigurable Systolic Array.

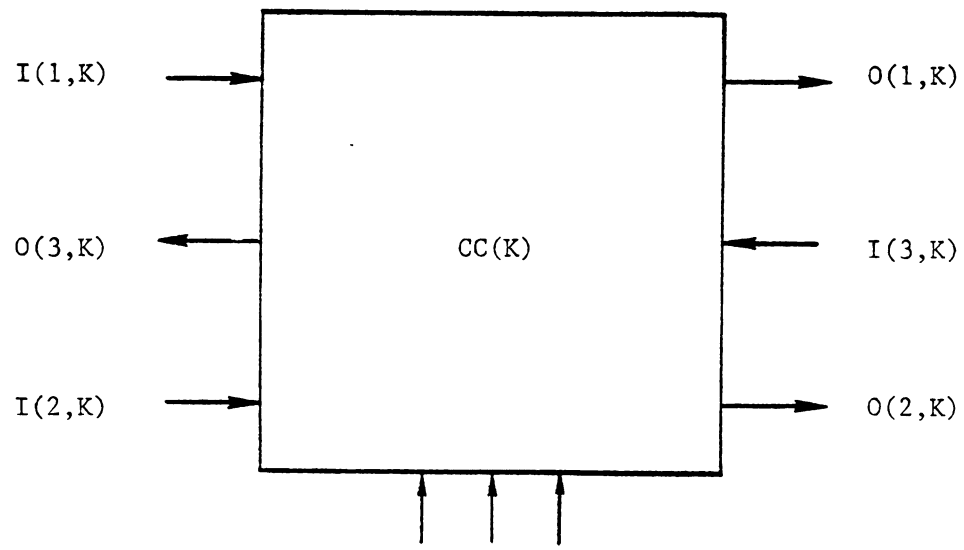


Figure 3.2. Block diagram representation of a Computation Cell.

computation cell at K'th position in the array. The control input lines will be referred to as function-select lines. For a computation cell CC(K), the input ports are referred to as I(1,K), I(2,K) and I(3,K), whereas the output ports are O(1,K), O(2,K) and O(3,K). Each cell contains two operand register buffers A(K) and B(K), which hold the filter coefficients loaded by the host, prior to start of computation.

Let set of function-select lines for cell CC(K) be called S(K). Then, functions of the computation cell can be described as follows:

```
Begin
        if S(K) = 001 then
                    O(1,K) <--- I(1,K)
                    O(2,K) <--- I(2,K)
                    O(3,K) <--- A(K)*I(2,K)+B(K)*I(1,K)+I(3,K)
        elseif S(K) = 010 then
                    O(1,K) <--- B(K)
                    O(2,K) <--- A(K)*I(1,K)+I(2,K)
                    B(K) <--- I(1,K)
        elseif S(K) = 011 then
                    O(1,K) <--- I(3,K)
                    O(3,K) <--- I(3,K)
        elseif S(K) = 100 then
                    O(1,K) <--- I(1,K)
                    O(3,K) <--- I(3,K)
                    R(K) <--- R(K).AND.(I(1,K).EQ.I(3,K))
        elseif S(K) = 101 then
                    O(1,K) <--- I(1,K)
```

```
                O(2,K) <--- I(1,K)*I(2,K)+A(K)
        end if
end
```

R(K) is a flip flop in each computation cell which is used in the pattern matching algorithm. A(K) and B(K) are two registers in each CC(K) which hold the coefficients loaded from the host or main memory.

3.1.2 FUNCTION-SELECT ARRAY

The function-select array contains a three-bit element for each computation cell; the contents of this array select the function performed by the computation cell. Each computation cell can be programmed to perform a selected primitive arithmetic or logic operation by setting its function-select array element. All elements of the function-select array consist of SRLs which are all threaded together to form a shift register. The control information can be entered in a bit-serial fashion using two phase clock via Scan In input line, as shown in Figure 3.3.

3.1.3 DATA FLOW CONTROL ARRAY

The data flow control array contains the control information for the interconnection network configuration. Data flow control is achieved by employing interchange boxes which are devices with two inputs and two outputs. Figure 3.4 shows the four legitimate states of an interchange box. Two-bit control is required for each box and the control bits for each box are stored in an SRL. All SRLs are threaded together as in case of the function-select array. Figure 3.5 shows a hardware implementation

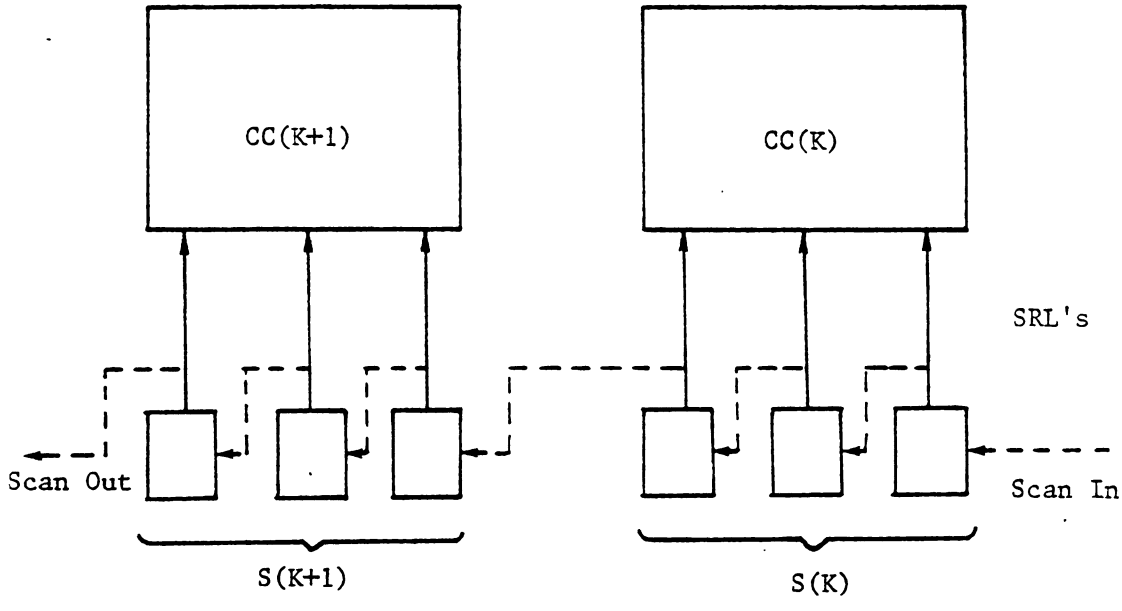


Figure 3.3. SRLs are used in the function-select array.

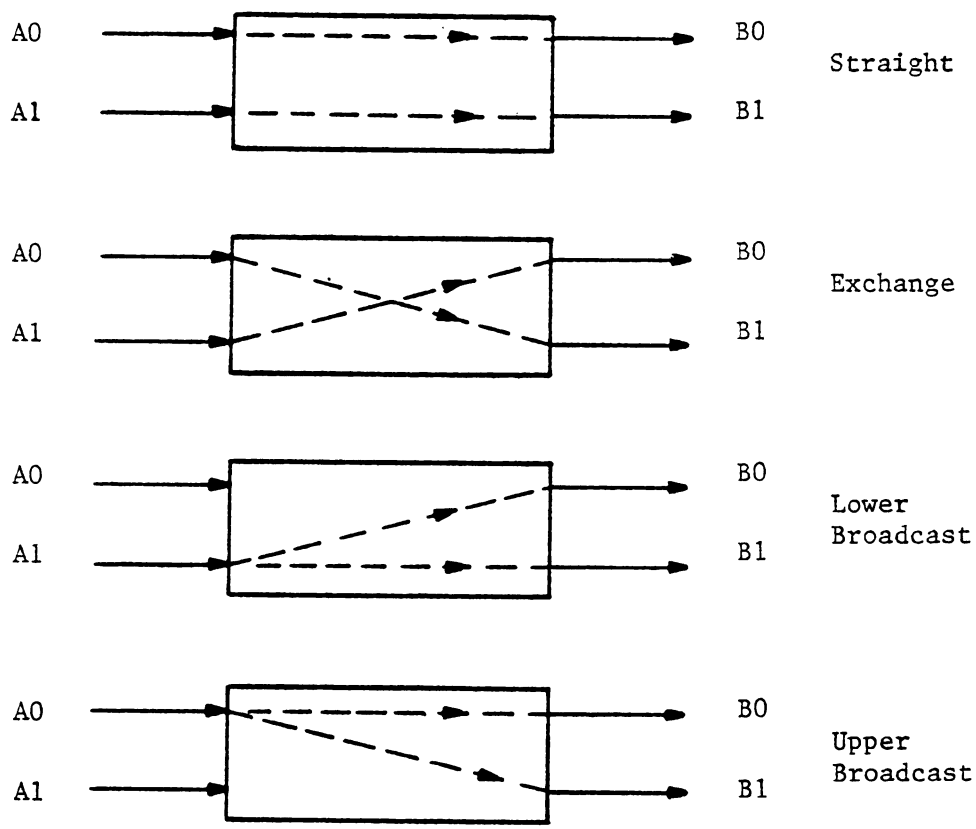


Figure 3.4. Four interconnection states of an interchange box.

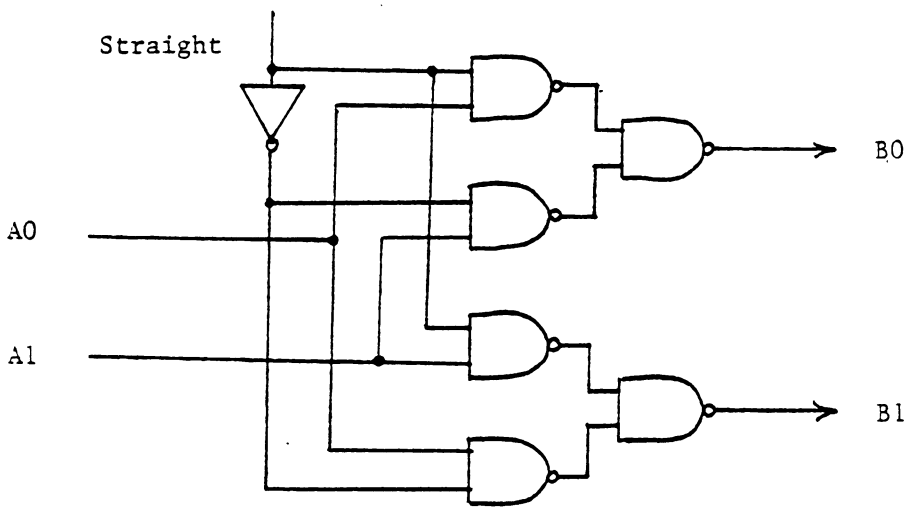


Figure 3.5. Logic diagram of a two-function interchange box.

of an interchange box.

Figure 3.6 illustrates how two adjacent cells in the LRSA are connected together with interchange boxes and function-select elements. Although the computation cells are multifunctional, yet various functions are computationally not much different from each other. By employing some data selection logic in an inner product step processor, computation cells for LRSA could be obtained. So, the overhead because of introducing complexity in the computation cell by making it multifunctional should be small. All the control information is entered through one pin in a bit-serial fashion, so overhead because of additional pinouts is minimal.

3.2 SYSTOLIC ALGORITHMS IMPLEMENTED ON THE LRSA

This section describes four systolic algorithms implemented on the LRSA. These algorithms include general filtering, Finite Impulse Response (FIR) filtering, pattern matching and Discrete Fourier Transform (DFT).

3.2.1 SYSTOLIC FILTERING ARRAY

The general filtering problem is defined as follows [49]:

Given the weighting coefficients $\{w_0, w_1, \ldots, w_h\}$, $\{r_1, r_2, \ldots, r_k\}$, the initial values $\{y_{-k}, y_{-k+1}, \ldots, y_{-1}\}$, and the input sequence $\{x_{-h}, x_{-h+1}, \ldots, x_0, x_1, \ldots, x_n\}$,

compute the output sequence $\{y_0, y_1, \ldots, y_n\}$ defined by

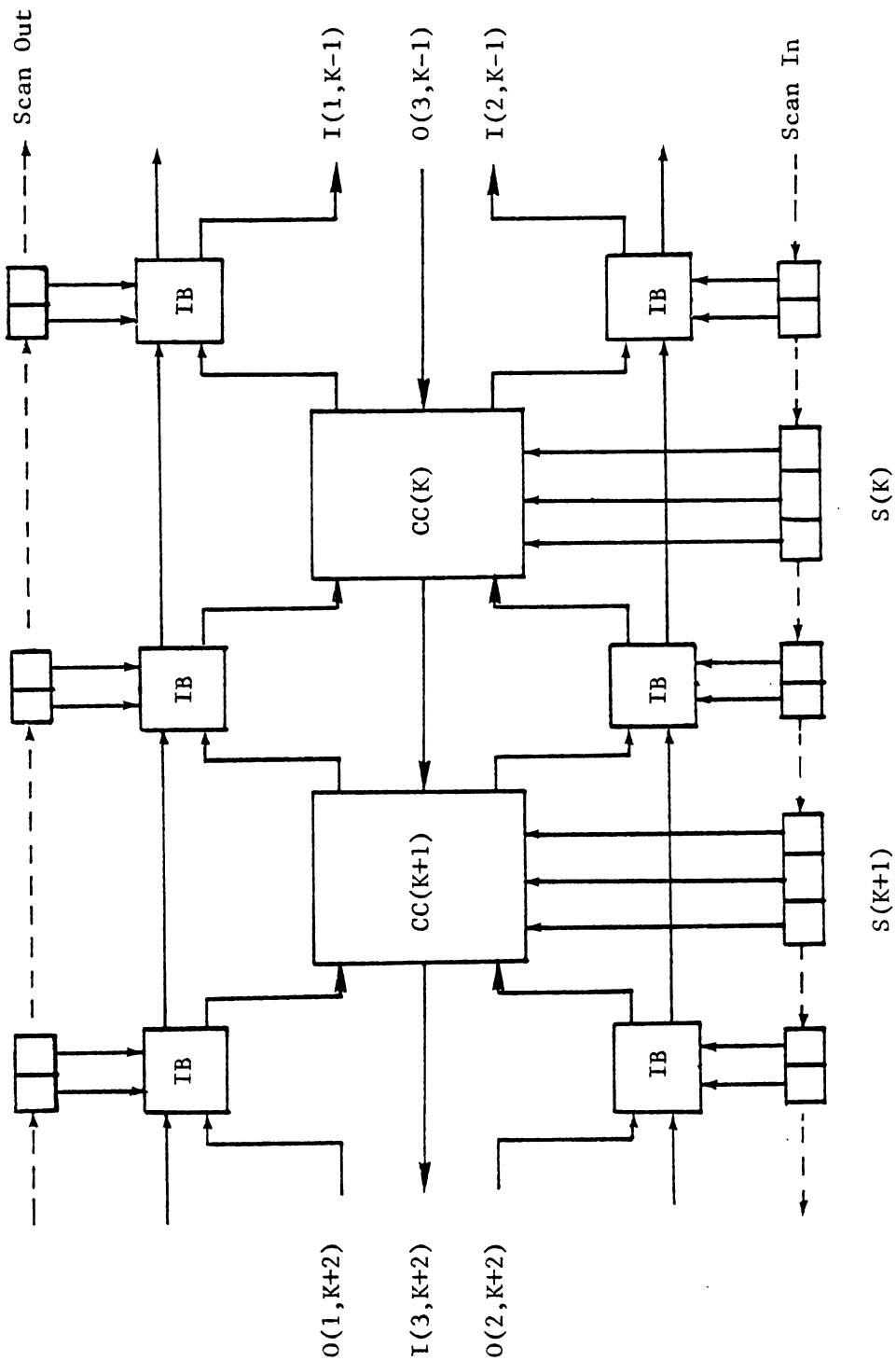


Figure 3.6. Structure of the Linear Reconfigurable Systolic Array.

$$y_i = \sum_{j=0}^{h} w_j x_{i-j} + \sum_{j=1}^{k} r_j y_{i-j}. \qquad (3.1)$$

A systolic array to implement the above filtering problem is shown in Figure 3.7 [49]. This array computes a new output y_i every two cycles, where a cycle is the time to perform two multiplications and two additions. The weighting coefficients w_i's and r_i's are preloaded into the array. The filtering computation starts by loading the x_i's from the host to the systolic array. When the array has received all the x_i's for $-h \leq i < 0$, it starts outputting the computed y_i's at the rate of one every two cycles.

The two types of basic cells used are shown in Figure 3.7(a). The systolic array for filtering is a linear array and consists of m type-1 cells and one type-2 cell, where m = max(h+1,k). Each y_i is initialized as zero as entering the array from the right-most cell. It accumulates terms as it travels along the array towards left and eventually achieves its final value y_i when reaching the left-most cell. The output y_i is fed back into the array for use in other computations.

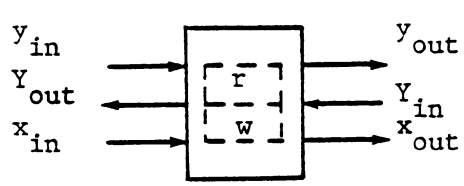
In order to implement the above filtering algorithm on an LRSA consisting of (m+1) computation cells, the function-select elements should be as follows:

S(K) = 001, $1 \leq K \leq m$;

S(K) = 011, K = m+1

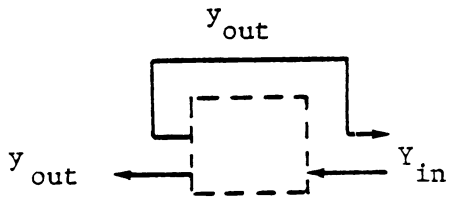
The switching cells in the array should be used in straight configuration. It can be easily seen from Figure 3.6, that exchange and broadcast configurations of the switching cell can be used to partition

TYPE-1 CELL:



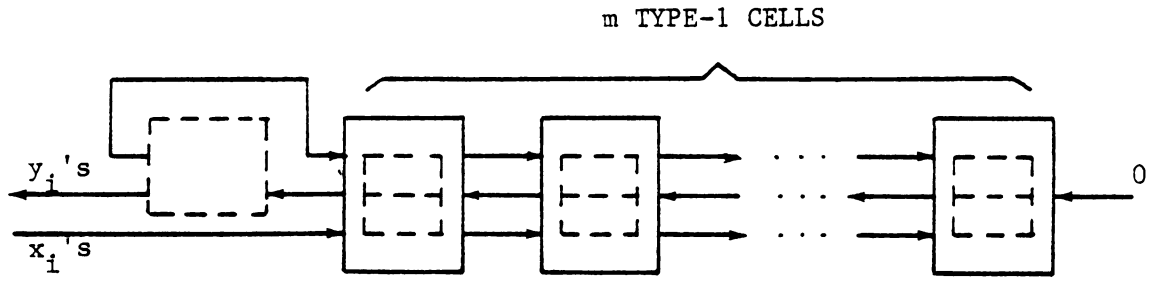
$$y_{out} \leftarrow y_{in}$$
$$Y_{out} \leftarrow w \cdot x_{in} + r \cdot y_{in} + Y_{in}$$
$$x_{out} \leftarrow x_{in}$$

TYPE-2 CELL:



$$y_{out} \leftarrow Y_{in}$$

(a)



(b)

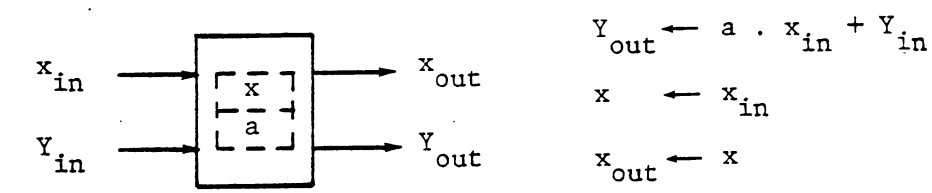
Figure 3.7. One-dimensional systolic array for filtering:
(a) basic cells used in the array,
(b) the systolic filtering array.

the LRSA into subarrays, or to bypass faulty computation cells.

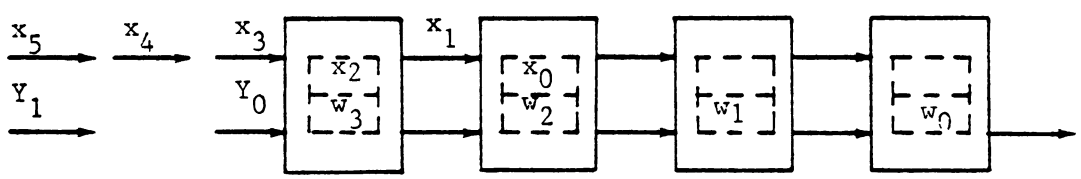
3.2.2 SYSTOLIC FIR FILTERING ARRAY

The general filtering problem described in the previous subsection realizes systems having Infinite-duration Impulse Response (IIR). Such systems involve a recursive computational algorithm. In the case of Finite-duration Impulse Response (FIR) systems [49], realization generally takes the form of a nonrecursive computational algorithm. An FIR filtering problem is a special case of the general filtering problem given in Subsection 3.2.1 where $r_i=0$ for all $1 \leq i \leq k$. The dedicated throughput of the systolic array for filtering described in the previous subsection is one half, i.e. only one half the cells in the array are active at any given time. Figure 3.8 shows a systolic array for FIR filtering algorithm [24]. Data streams move in the same direction at two different speeds in the systolic array and all the cells are used all the time. The basic cell used in the design is shown in Figure 3.8(a). The w_i's are preloaded in the array during the initialization phase. Both the x_i's and y_i's travel towards right but y_i's travel twice as fast as x_i's. Each y_i accumulates terms as travelling towards right and achieves its final value as it leaves the right-most cell. The FIR filtering problem is mathematically identical to the convolution problem, so the systolic array described here also applies to convolution computations.

In order to implement the FIR filtering algorithm on the LRSA, the function-select elements for all the computation cells in the array, should contain 010, i.e.,



(a)



(b)

Figure 3.8. One-dimensional systolic array for FIR filtering:
(a) basic cells used in the array,
(b) the systolic FIR filtering array.

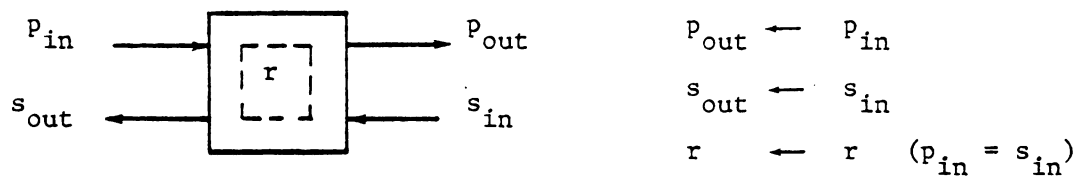
$S(K) = 010,\ 1 \leq K \leq h.$

3.2.3 PATTERN MATCHING SYSTOLIC ARRAY

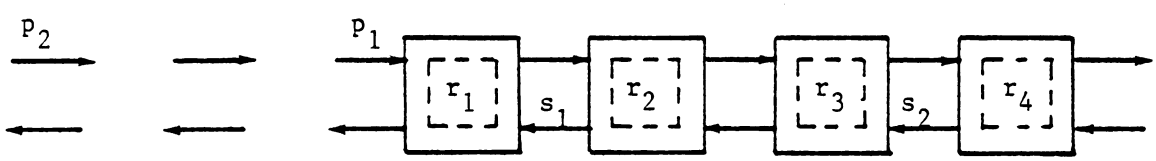
A systolic array design for pattern matching problem is shown in Figure 3.9 [24]. The pattern matching problem can be defined as follows:

> Given a text string $(s_1, s_2, \ldots, s_n)$ and a pattern string $(p_1, p_2, \ldots, p_m)$, with n much larger than m,
>
> compute positions of all occurrences of the pattern within the text string.

For example, if the text string is DBABBFBABABB and the pattern is BAB, then the result is 010000101000, where each '1' indicates the beginning position of an occurrence of the pattern inside the text string. Let the resulting Boolean string be $\{r_1, r_2, \ldots, r_{n-m+1}\}$, such that r_i=1 if and only if $(s_i, s_{i+1}, \ldots, s_{i+n-1}) = (p_1, p_2, \ldots, p_m)$. Then the systolic array shown in Figure 3.9, using the basic cell as shown, can compute the r_i's by comparing the characters of the pattern with the characters of the text string. Each r_i is initialized to be a '1'. A pattern character p_k and a text string character s_j are compared at the cell where they meet and the cell updates the value of r_i such that $r_i \leftarrow r_i$ AND $(p_k=s_j)$. In case of a mismatch, the value of r_i is reset to '0'. The pattern string travels towards right and the text string travels towards left, such that each pattern character meets each text character. The value of r_i at a cell finalizes when the last character in the pattern string passes through that cell. This systolic array with m cells can solve the pattern matching problem in time n, whereas a sequential software solution takes time proportional to m*n.



(a)



(b)

Figure 3.9. One-dimensional systolic array for pattern matching:
(a) basic cells used in the array,
(b) the systolic pattern matching array.

In order to implement the pattern matching algorithm on an LRSA consisting of m computation cells, the function-select elements for all the computation cells in the array should contain 100, i.e.,

$$S(K) = 100,\ 1 \leq K \leq m.$$

3.2.4 DISCRETE FOURIER TRANSFORM (DFT) SYSTOLIC ARRAY

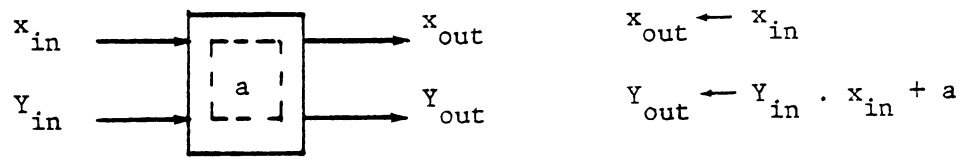
An n-point Discrete Fourier Transform (DFT) is defined as follows:

Given $\{a_0, a_1, \ldots, a_{n-1}\}$ be n samples of a time function, compute $\{y_0, y_1, \ldots, y_{n-1}\}$ defined by

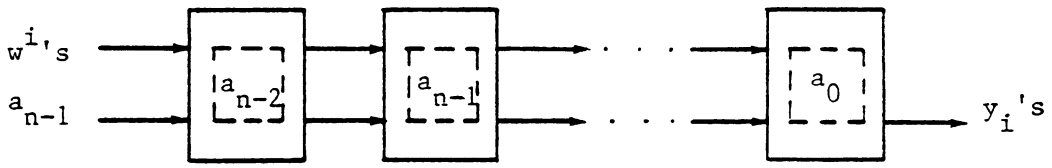
$$y_i = \sum_{j=0}^{n-1} a_j * w_{ij}, \qquad (3.2)$$

where $w = e^{2\pi k/n}$ and $k = -1$.

The straightforward method for computation of n-point DFT requires $O(n^2)$ operations and the Fast Fourier Transform (FFT) algorithm requires O(nlogn) operations for the same computation. The linear systolic array shown in Figure 3.10 [48] with (n-1) basic cells can compute an n point DFT in O(n) time. However, the communication scheme for the systolic array is much simpler compared to the complicated data communication requirements for the FFT algorithm. The basic cell used in the DFT systolic array is essentially a multiplier-accumulator cell and is shown in Figure 3.10(a). The array consists of (n-1) basic cells and the input samples a_{n-2} to a_0 are preloaded in the array cells. The inputs Y_{in} and x_{in} to the left-most cell are a_{n-1} and some power of w, respectively. The output x_{out} from the right-most cell is always ignored. Each y_i,



(a)



(b)

Figure 3.10. Linear systolic array for DFT algorithm:
(a) basic cells used in the array,
(b) the systolic DFT array.

initialized as a_{n-1}, accumulates its terms as travelling towards right, and reaches its final value as it leaves the right-most cell.

In order to implement an n-point discrete Fourier transform algorithm on the LRSA, the function-select elements for all the computation cells in the array, should contain 101, i.e.,

$$S(K) = 101,\ 1 \leq K \leq n-1.$$

3.3 SUMMARY AND DISCUSSION

This chapter presented the architecture and design of a one-dimensional Linear Reconfigurable Systolic Array. This design approach illustrates how the level sensitive scan design technique can be employed in mixed systolic arrays, to load and implement the distributed control structure required to configure the array for a desired algorithm implementation. The control structure for a particular configuration is loaded into the array through the LSSD scan path as a binary vector in a bit-serial fashion. This approach has the advantage that overhead in terms of additional pinouts is limited to only three or four pins. Also, serial loading of the control structure reduces the overhead in terms of additional interconnections on the chip due to the control hardware.

The major incentive for employing the LSSD technique in the design of the LRSA is to load the configuration control code in the array. This approach, in addition, enhances testability, incorporates fault tolerance, and provides capability for initializing the internal data registers of individual cells. In order to achieve testability and data

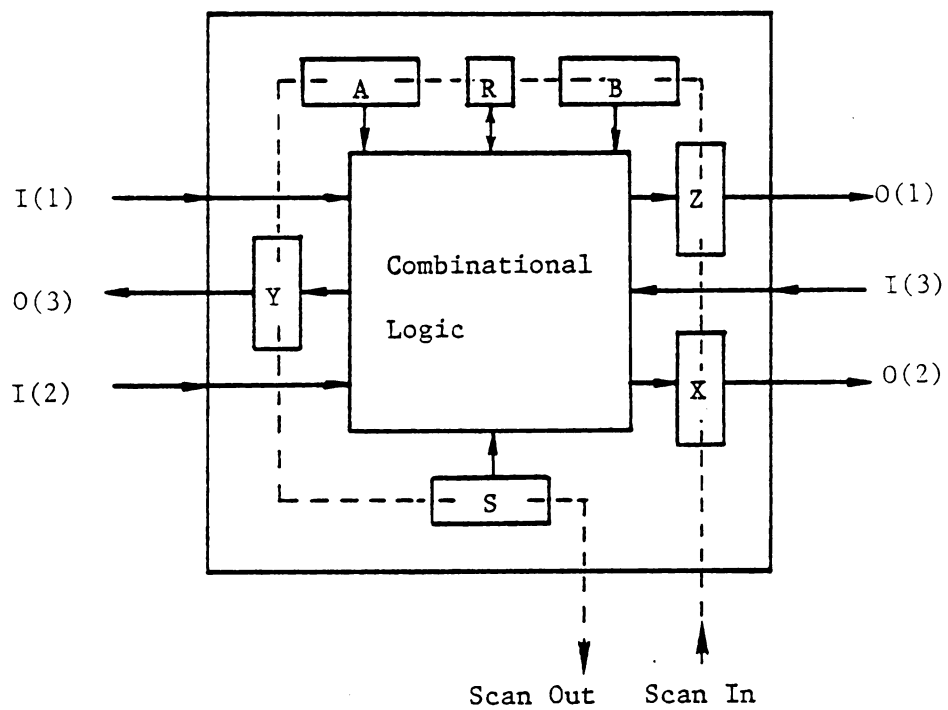


Figure 3.11. LSSD scan path within a computation cell incorporating testing and register initialization.

register initialization capability, the LSSD scan path must pass through the data registers and the input/output ports of each cell. Figure 3.11 shows one possible arrangement for a scan path within a computation cell. Registers X, Y and Z are the local output ports; S is the control register which selects the function performed by the computation cell; A and B are the data registers for holding the preloaded coefficients and R is a flip-flop used to save the result of a Boolean operation in pattern matching algorithm. This arrangement provides capability for testing the functionality of the individual cells, initializing the registers and preloading the coefficients. During test mode, a desired bit-vector is loaded into the array through LSSD scan path. This vector contains the control information required for each cell as well as the test data for data registers. After the vector has been loaded, one system clock is applied so that each cell can perform its required function. The results in the local output ports are scanned out using LSSD shift clock. This vector contains results of the operations performed by all the cells. A comparison of this vector with the expected results helps identify the faulty cells which may be bypassed by reconfiguring the chip. Fault tolerance in mixed systolic arrays employing LSSD techniques is discussed in more detail in Chapter 4, where this approach is extended to two-dimensional mixed systolic arrays.

CHAPTER 4

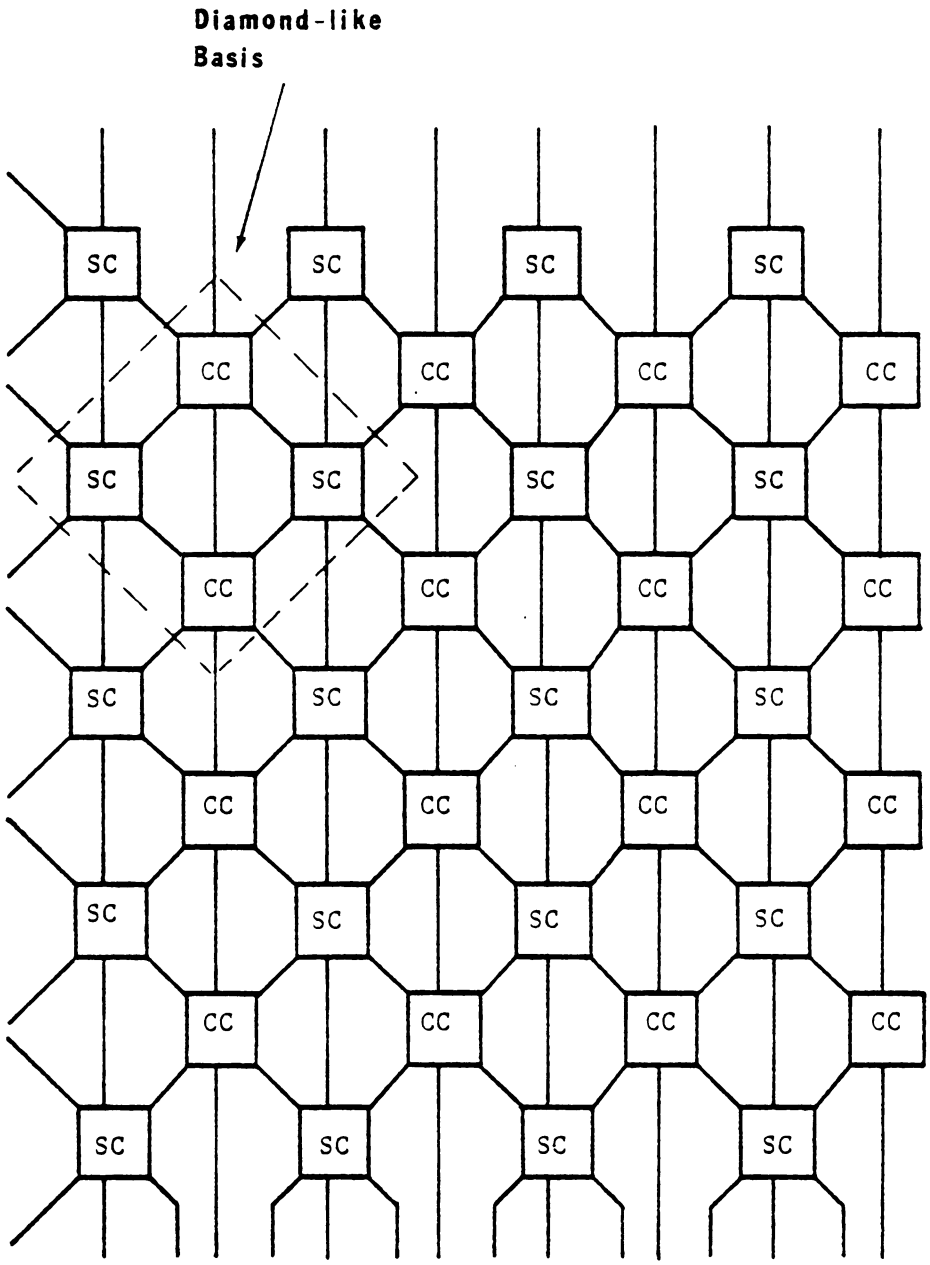
DESIGN OF A MULTIPURPOSE RECONFIGURABLE ARRAY PROCESSOR (MRAP)

Chapter 3 presented a design approach for mixed systolic arrays, which employs the level sensitive scan design technique for configuring the array architecture to match the structure of the algorithm to be implemented. This chapter extends the same approach to two-dimensional mixed systolic array architectures, and discusses the issues of fault tolerance, performance analysis and algorithm implementation. We present the design of an MSA-based processor, called the multipurpose reconfigurable array processor (MRAP). This architecture implements both systolic and semisystolic algorithms, and incorporates fault tolerance in its design. The performance of MRAP, taking into account the total computation time and the data-transfer bandwidth, is analyzed for specific algorithms. Also, it is demonstrated, by way of examples, how the MRAP can efficiently implement different systolic and semisystolic algorithms. Structure of the MRAP is a two-dimensional array type, which can be configured to implement a number of systolic and semisystolic algorithms involving two-dimensional linear recurrences such as matrix manipulations. This two-dimensional array could also be partitioned into many independent linear arrays to implement independent algorithms involving one-dimensional linear recurrences such as finite impulse response (FIR) filtering.

The next section describes the structure of the MRAP, the design of basic functional cells and the data flow in the array at register-transfer level. Section 4.2 illustrates, by way of examples, how the MRAP can be configured to implement different parallel algorithms. In Section 4.3, performance of the MRAP is analyzed and compared with other existing systolic structures. Section 4.4 investigates fault tolerance capabilities inherent in the MRAP design. Finally, a summary and discussion is presented in Section 4.5.

4.1 STRUCTURE OF THE MULTIPURPOSE RECONFIGURABLE ARRAY PROCESSOR

The multipurpose reconfigurable array processor (MRAP) employs two types of basic elements, computation cells and switching cells, as shown in Figure 4.1. In general, the computation cells and the switching cells are mixed according to a certain mixing profile which determines the possible frames of data-flow patterns within the array. Mixing profile is specified by the mixing density and the basis of the array [6]. The MRAP designed here, is constructed from a diamond-like basis with a mixing density of 1/2, i.e., switching cells and computation cells are equally represented in the basis. The control structure for communication and computation is embedded in both types of cells in form of control registers. We employ the LSSD technique to realize this control structure. All the control registers are threaded together through a Scan-In line as shown in Figure 4.2. For a particular configuration the control structure is realized by loading a binary vector through the Scan-In line in a bit-serial fashion. In the next



CC : Computation Cell

SC : Switching Cell

Figure 4.1. The Structure of the Multipurpose Reconfigurable Array Processor (MRAP).

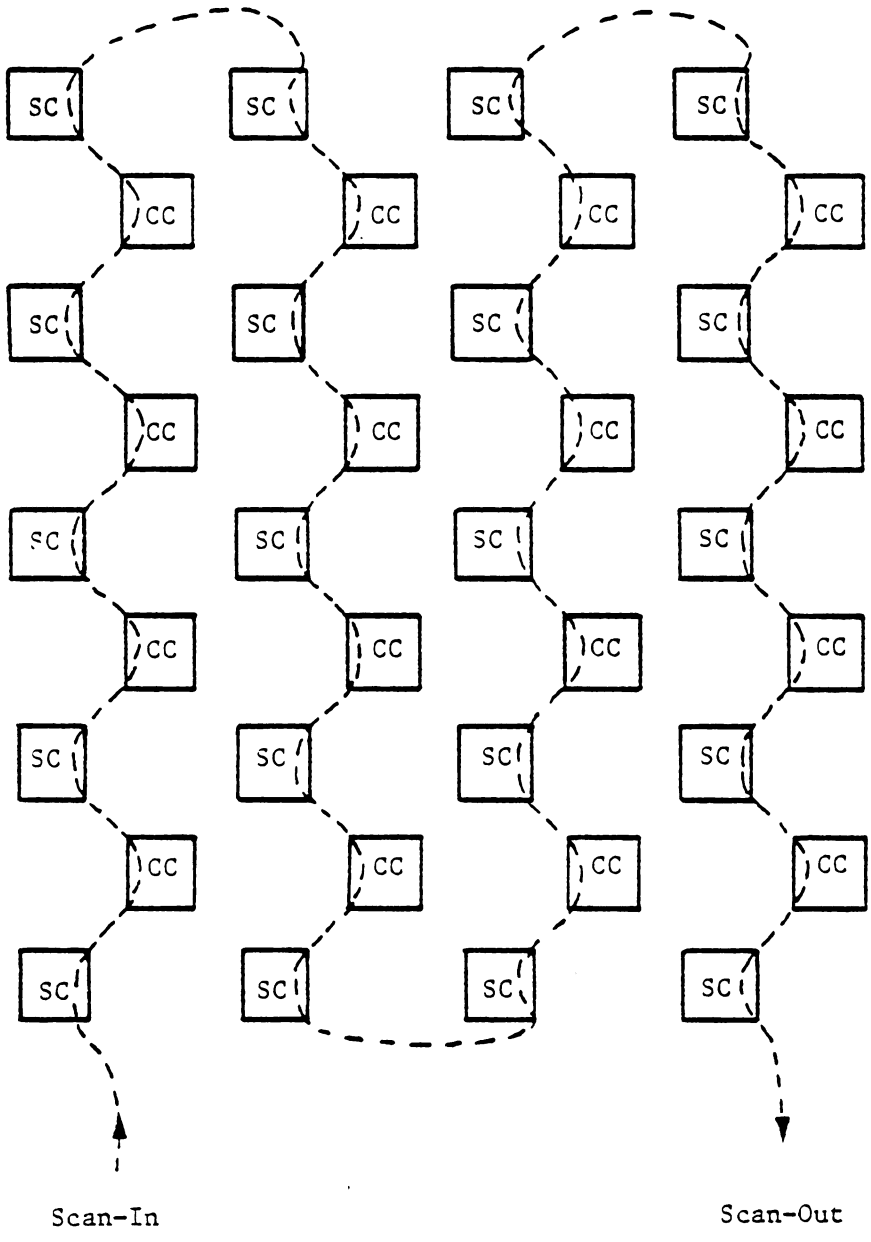


Figure 4.2. Scan Path for Loading the Control Vectors.

section, we provide some examples of how the control vectors can be generated for implementing different algorithms. Figure 4.3 shows the system block diagram of the array and each part of the structure is described later in this section. The array can be configured in various topologies depending upon the application, such that a variety of algorithms can be efficiently implemented. Various configurations obtained from this structure are discussed in Section 4.2. In the remainder of this section, we describe the design of the computation cell and the switching cell. Also, a register-transfer level data-flow description and a timing diagram for the computation cell are given.

4.1.1 THE COMPUTATION CELL

The computation cell is basically an inner product step processor with some extra control logic added into it. All computation cells are identical so the modularity is preserved in the design. This, of course, is an important factor in the efficient design and implementation of structures for VLSI computation. Each computation cell has six data ports, which are all local ports except for the cells at the boundary of the array, where global I/O ports are used [6,8]. Due to local I/O ports the interconnections are simple, regular and short. Three control bits are used to select the functionality of the computation cell. There are three dedicated data registers DRA, DRB and DRR which can be used to hold the preloaded coefficients or the results of a local computation. There is a three-bit control register, comprising of shift-register latches and threaded into the Scan-In line. We shall refer to this control register as RC and the three latches in it as RC0, RC1 and RC2. The input-output

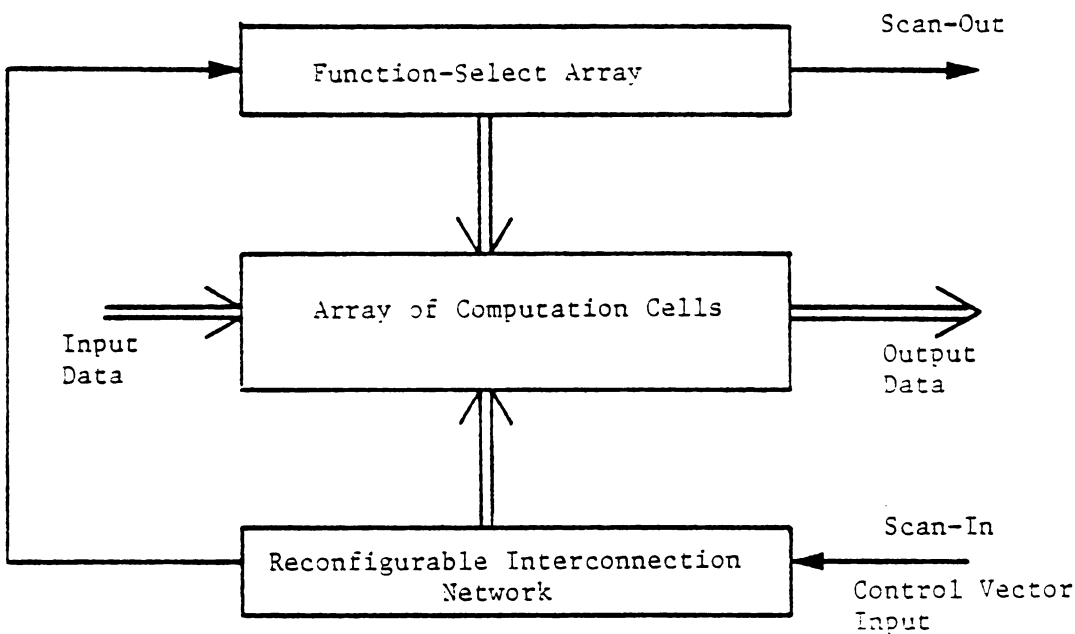


Figure 4.3. General System Diagram of the MRAP.

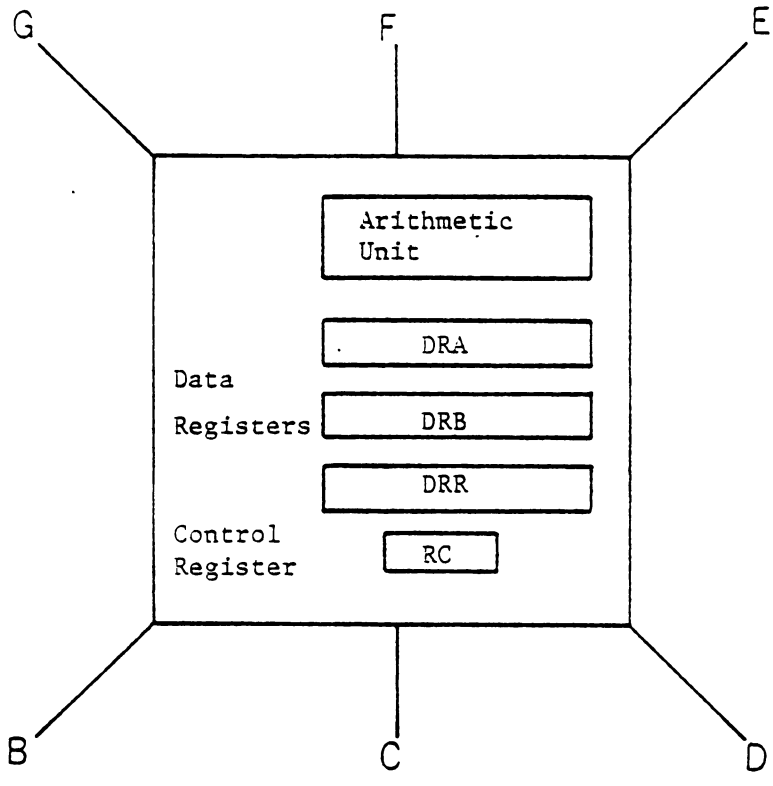
functional mapping of a cell is determined by the contents of RC. Figure 4.4 shows a computation cell and describes some of its functions for different control vectors. A register-transfer level description of various operations is given later in this section.

4.1.2 THE SWITCHING CELL

The switching cell used in this array is an extension of the two-by-two interchange box [52] and is shown in Figure 4.5. When the R/W control line is low, this switching cell acts as a single-stage, nonblocking SIMD interconnection network which requires three-bit control to realize all possible connections from input lines A1, A2 to the output lines B0, B1 and B2, without resulting in a conflict. When R/W line is high, A2 acts as an output line which is connected to the input A0. R/W is a local I/O control provided by the neighboring computation cell at line A2. The control bits are held in the three-bit control register RS associated with each switching cell. Figure 4.5 shows a logic implementation of the switching cell. A function table for the switching cell is given in Table 4.1, where RS0, RS1 and RS2 are the bits in the control register RS.

4.1.3 DATA-FLOW AND TIMING

This section describes a register-transfer level description of data-flow in the array. Figure 4.6 shows a timing diagram for the instruction cycle of the computation cell, when a two-phase system clock is used. There are three phases in the instruction cycle, as shown in Figure 4.6. Phases t_0 and t_1 are used to latch input data from the




```
Begin
      DRA◄ — — G
   if RC=001 then
      E  ◄ — — DRA
      DRR◄ — — DRR + DRA * B
   elseif RC=010 then
      E  ◄ — — DRA
      C  ◄ — — F + DRA * B
   elseif RC=011 then
      E ◄ — — DRA
      G ◄ — — D + DRA * F
      C ◄ — — F
   elseif RC=100 then
      G ◄ — — B
      C ◄ — — F + DRA * B
   end if
end;
```

Figure 4.4. The Computation Cell.

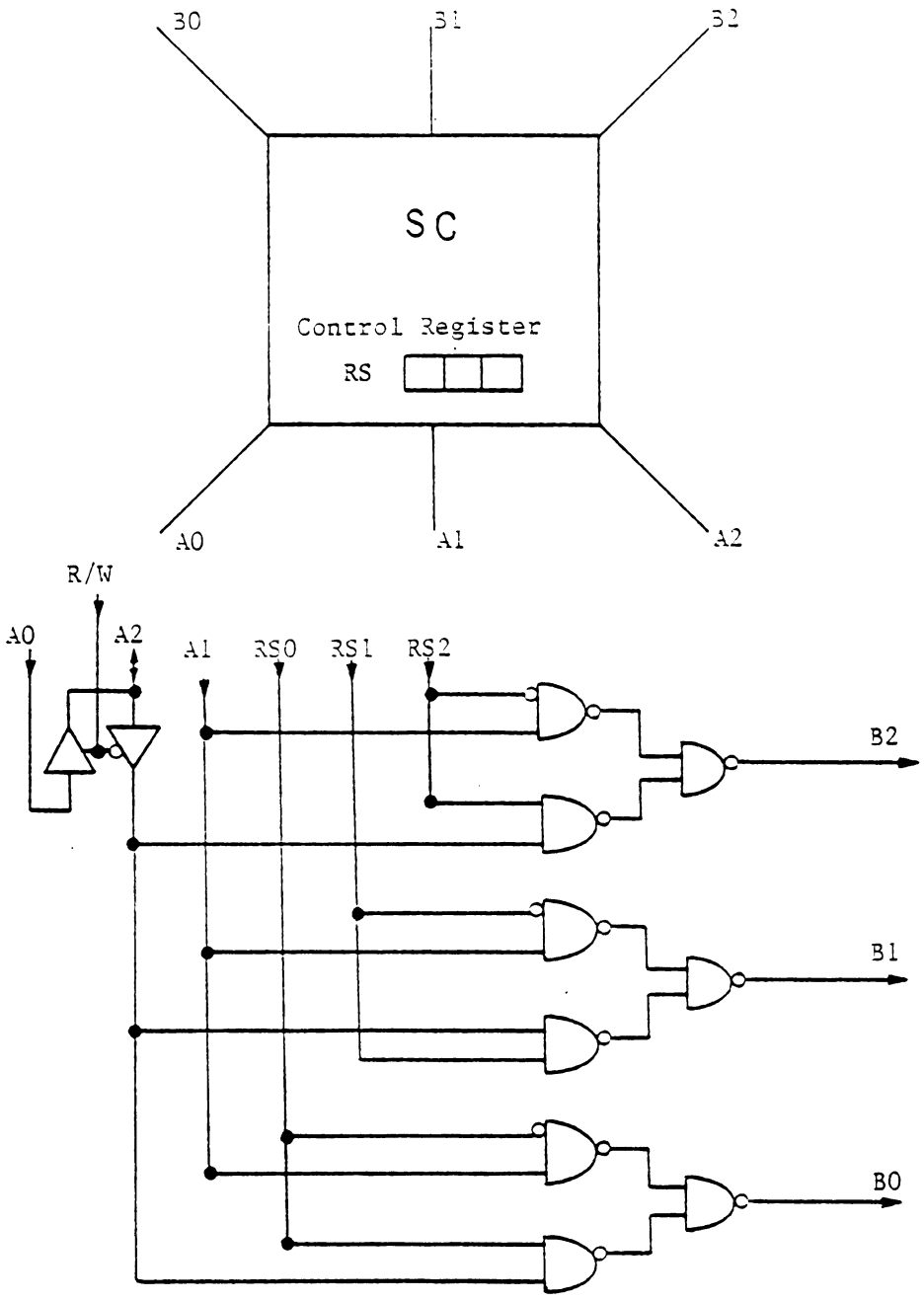


Figure 4.5. A Gate-level Logic Implementation of the Switching Cell.

RS2	RS1	RS0	B2	B1	B0
0	0	0	A1	A1	A1
0	0	1	A1	A1	A2
0	1	0	A1	A2	A1
0	1	1	A1	A2	A2
1	0	0	A2	A1	A1
1	0	1	A2	A1	A2
1	1	0	A2	A2	A1
1	1	1	A2	A2	A2

Table 4.1. Truth Table for the Switching Cell (R/W = Low)

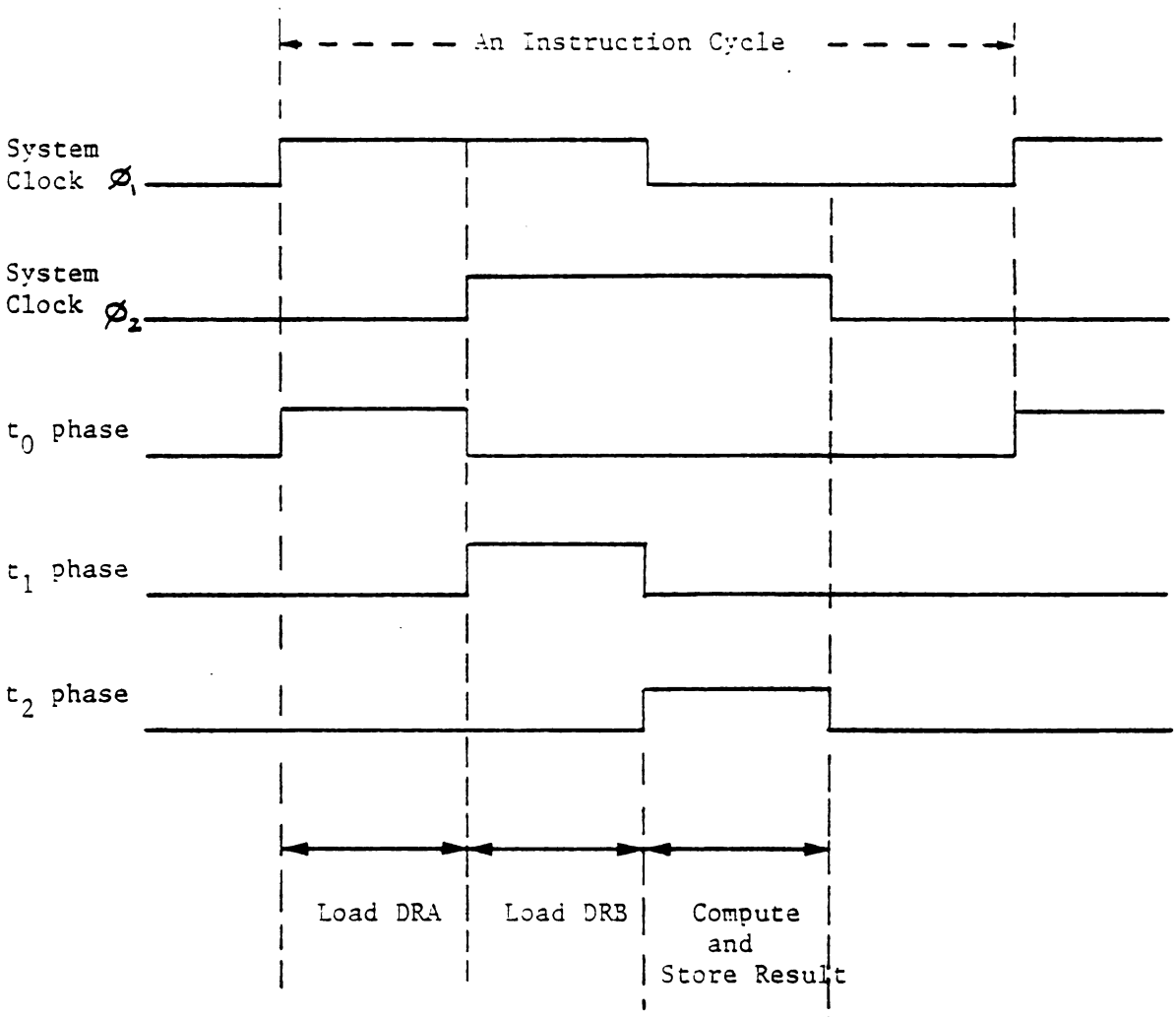


Figure 4.6. A Timing Diagram for Inter-processor Communication.

neighboring cells. We need two phases for this purpose due to the bidirectional nature of the interconnections. The third phase, t_2, is used for computing and storing the results in the output registers. Figure 4.7 shows a computation cell $CC_{i,j}$ along with its neighboring cells. A register-transfer level description of data-flow for a computation cell $CC_{i,j}$, in terms of its neighboring cells, is given as follows:

RC	Phase	Register Transfer Operations
001	t_0	$DRA_{i,j} \leftarrow A2_{i,j+1} = DRA_{i-1,j}$
	t_1	$DRB_{i,j} \leftarrow B2_{i,j}$
	t_2	$DRR_{i,j} \leftarrow DRR_{i,j} + DRA_{i,j} * DRB_{i,j}$
010	t_0	$DRA_{i,j} \leftarrow A2_{i,j+1} = DRA_{i-1,j}$
	t_1	$DRB_{i,j} \leftarrow B2_{i,j}$
	t_2	$DRR_{i,j} \leftarrow DRR_{i,j+1} + DRA_{i,j} * DRB_{i,j}$
011	t_0	$DRA_{i,j} \leftarrow A2_{i,j+1} = DRA_{i-1,j}$
		$DRB_{i,j} \leftarrow DRB_{i,j+1}$
	t_1	$DRR_{i,j} \leftarrow B0_{i+1,j} = DRR_{i+1,j-1}$
	t_2	$DRR_{i,j} \leftarrow DRR_{i,j} + DRA_{i,j} * DRB_{i,j}$
100	t_0	$DRB_{i,j} \leftarrow B2_{i,j}$
	t_1	idle; DRA contains a preloaded coefficient
	t_2	$DRR_{i,j} \leftarrow DRR_{i,j+1} + DRA_{i,j} * DRB_{i,j}$

4.1.4 PROGRAMMING DATA-FLOW CONTROL

Each switching cell is individually programmable to achieve a particular connection scheme; in this way the computation cells can be

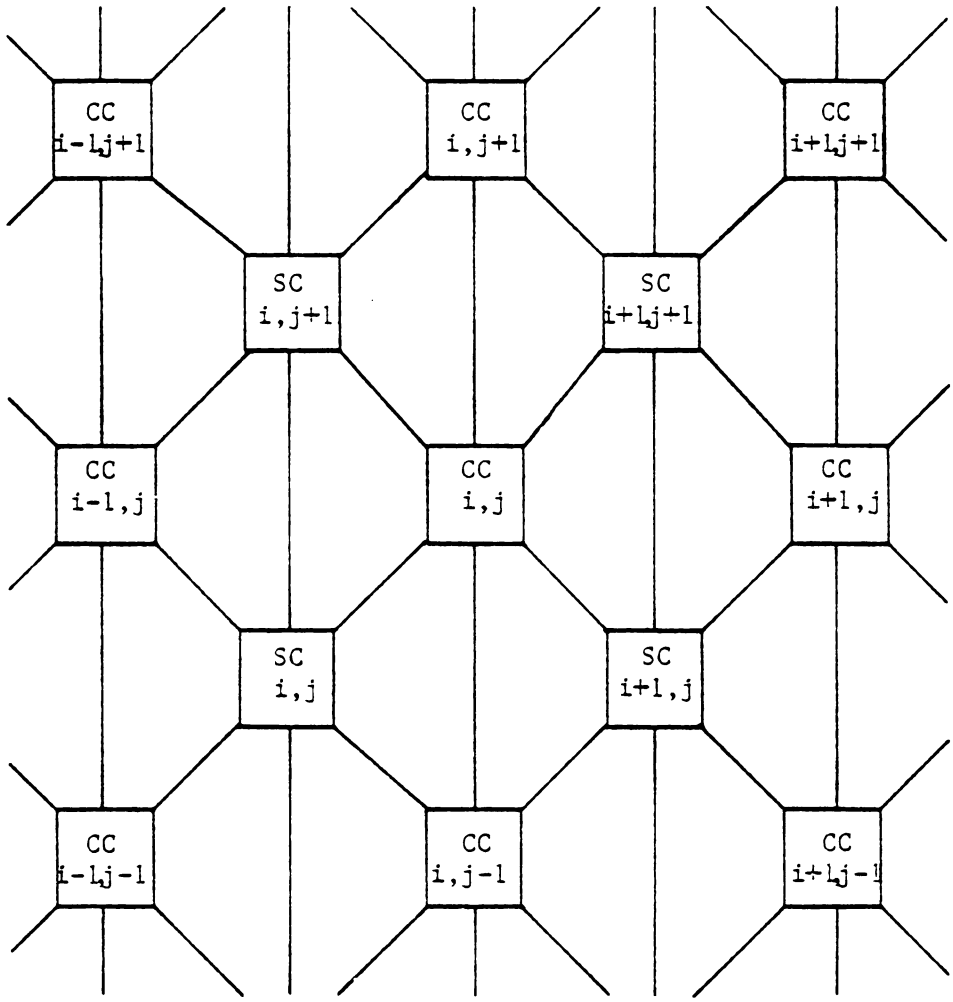


Figure 4.7. A Computation Cell $CC_{i,j}$ and its Neighbors.

interconnected in a number of possible configurations. Each switching cell contains a control registers, and contents of the control register determine the switching cell configuration. The flip-flops used in the control registers are shift-register latches. The control register flip-flops of all the switching cells are threaded together to form a serial-in serial-out shift register. The control information pertaining to a particular array configuration is entered through the Scan-In input line in form of a binary control vector in a bit-serial fashion. The array can be used for that implementation for as long as desired and when a new application of the chip is needed, it can be reconfigured by programming the switching cells by entering another control vector through Scan-In input line. In this fashion static reconfigurability is achieved in this architecture for implementing a variety of interconnection patterns.

Multifunctionality in the array is achieved by introducing function-select control registers (RC registers) in the computation cells (CCs). Input/output functional mapping of a CC is determined by the contents of its function-select control register which comprises of shift-register latches. All the RC registers are threaded together along with the control registers of switching cells in a long chain of shift registers. The control vector entered through the Scan-In input line consists of the desired bit pattern which determines the interconnection scheme as well as the functions performed by each computing element.

4.2 IMPLEMENTATION OF ALGORITHMS ON THE MRAP

Multiple data streams flow in statically reconfigurable multifunctional pipelines in the multipurpose reconfigurable array processor (MRAP) described here. Data-flow patterns are controlled by the control vector which is fed into the control registers through the Scan-In line during the preprocessing phase. For each array configuration and algorithm implementation, a binary control vector is generated containing control bits for the switching cells as well as the computation cells. The control bits for switching cells correspond to the communication structure of the implemented algorithm, and determine the inter-processor interconnection scheme. The control bits for computation cells correspond to the computational structure of the algorithm, and determine the functions performed by the computation cells on incoming data.

The length of the control vector is fixed for one chip regardless of the configuration and this length is equal to the total number of shift register latches threaded into the scan line in the system. In general the size of the vector can be written as:

$$S = p * NC + q * NS \ , \tag{4.1}$$

where

S = size of the binary control vector;

NC = number of computation cells in the array;

NS = number of switching cells in the array;

p = number of bits in the computation cell control register;

q = number of bits in the switching cell control register.

Inclusion of data registers in the scan path to provide testability for the functional cells will, however, make the input scan vector longer than that given in the above equation. In this case the control vector is mixed with initialization data for data registers of functional cells. In the following discussion, only the control vector generation is considered and the initialization bits for data registers are not included.

For the array shown in Figure 4.1, the size of the control vector can be computed from the above equation. Here, p=3, q=3, NS=5*4=20 and NC=16. So the size of the control vector is 108 bits. This binary vector is loaded into all of the RS and RC control registers through the Scan-In line in a bit-serial fashion at the time of configuring the array for a certain implementation. The time required for configuring depends on the length of the control vector and the system clock frequency. So, for example, about 4.3 micro-seconds are required to configure the above array with a 25 MHz system clock. Scan-Out line could be used to cascade more than one chip together or it could also be used to verify the control structure by shifting out the complete control vector and matching with the desired vector. This feature introduces, also, some degree of testability in the structure.

In the remainder of this section, various algorithms are implemented on the MRAP architecture, and in each case the array is configured to realize an efficient computational structure for algorithm

implementation. We consider an example to illustrate this point. Kung's systolic array [31] for band-matrix multiplication is quite efficient when multiplying matrices with narrow bandwidths, but it can not perform efficiently in the case of dense-matrix multiplication. A broadcast two-dimensional array as proposed by Huang and Abraham in [21], can perform dense-matrix multiplication very efficiently, but suffers from the problem of being considerably inefficient for narrow band-matrix multiplication. The MRAP can be configured to realize Kung's array structure for efficient narrow band-matrix multiplication, and Huang's 2-dimensional array for efficient dense-matrix multiplication.

4.2.1 DENSE-MATRIX/DENSE-MATRIX MULTIPLICATION

Dense-matrix multiplication is performed on the MRAP by configuring it as a broadcast two-dimensional array, as proposed by Huang and Abraham in [21]. Figure 4.8(a) shows the data streams for processing a dense-matrix multiplication on a broadcast two-dimensional array, which multiplies matrices A and B to get the product matrix C. As matrix B is broadcast from the bottom edge of the array and A is fed into the array from the left side, each computation cell in the array accumulates the partial product terms for an element of the matrix C. After the computation is done, the elements of the resulting matrix C which reside in the computation cells can be shifted out. This is an efficient algorithm for dense-matrix multiplication.

Control vector for realizing the above mentioned array is determined from the states of the computation cells and the switching cells needed to implement the above algorithm. The switching cells must be used in a

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

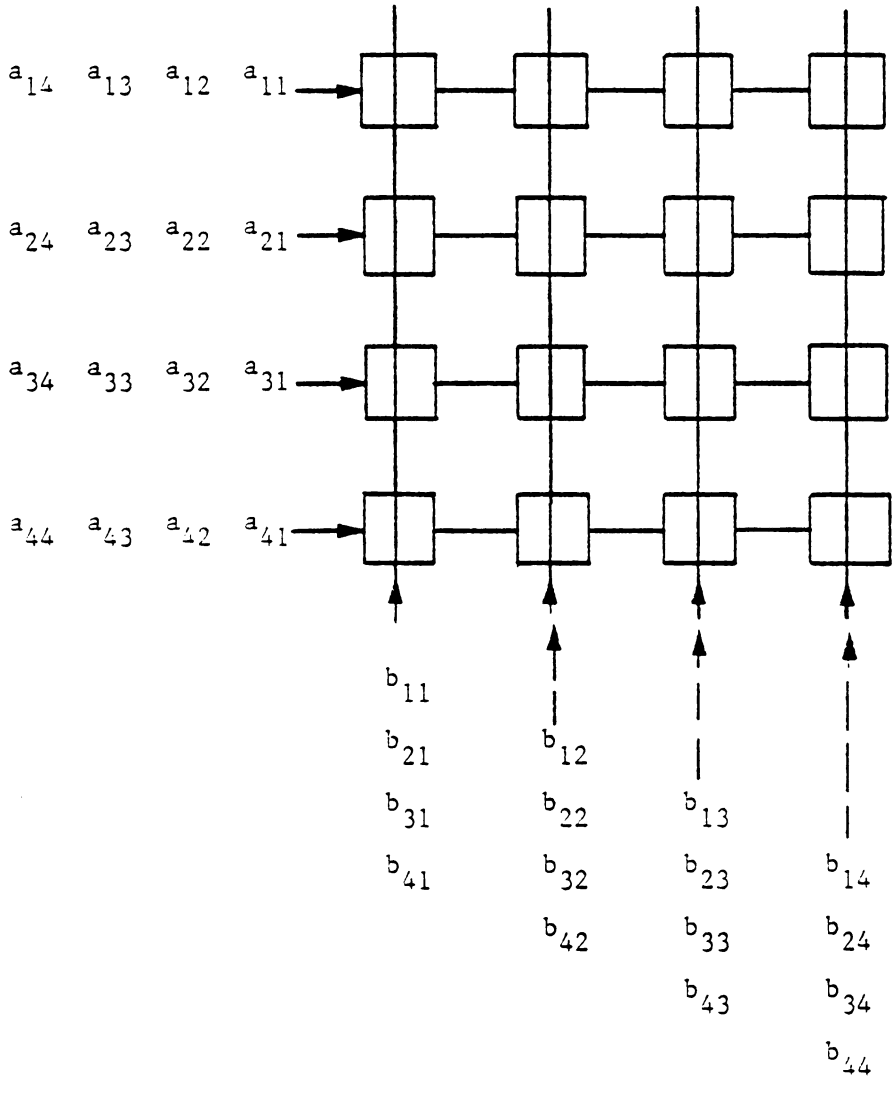


Figure 4.8.(a) Dense Matrix/ Dense Matrix Multiplication.

broadcast type of configuration and should have the following interconnections:

$$B1 = A1 \; ; \; B2 = A1. \qquad (4.2)$$

This implies, from Table 4.1, that the control bits for switching cells can be 000 or 001. We arbitrarily choose 001 here. The computation cell should perform the following functions:

DRA <-- G ; E <-- DRA ; DRR <-- DRR + DRA*B .

The control bits for the above functions are 001 from Figure 4.4. So, the control vector for the MRAP, in order to implement a broadcast two-dimensional array for efficient dense-matrix multiplication, is given as follows:

Control Vector (in HEX digits):

249249249249249249249249

The above vector is loaded through the Scan-In line using a two phase clock with the right most digit entering first. After all the 108 bits are shifted into the array in a bit-serial manner, each control register in the array contains appropriate control bits.

4.2.2 BAND-MATRIX/DENSE-MATRIX MULTIPLICATION

Let A be an NxN band-matrix with a bandwidth W (W<N), and B be an NxW dense-matrix. The matrix C = A*B can be computed by using an array configuration as shown in Figure 4.8(b). This algorithm is similar to that described in [9], except that broadcasting is used only in one

$$\begin{bmatrix} a_{11} & a_{12} & & . & 0 \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & . & . \\ & a_{42} & a_{43} & . & . \\ & & a_{53} & . & . \\ 0 & & & . & . \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & . & . \\ b_{31} & . & . & . \\ & & & \\ & & & \\ b_{N1} & . & . & b_{N4} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & . & . \\ c_{31} & . & . & . \\ & & & \\ & & & \\ c_{N1} & . & . & c_{NN} \end{bmatrix}$$

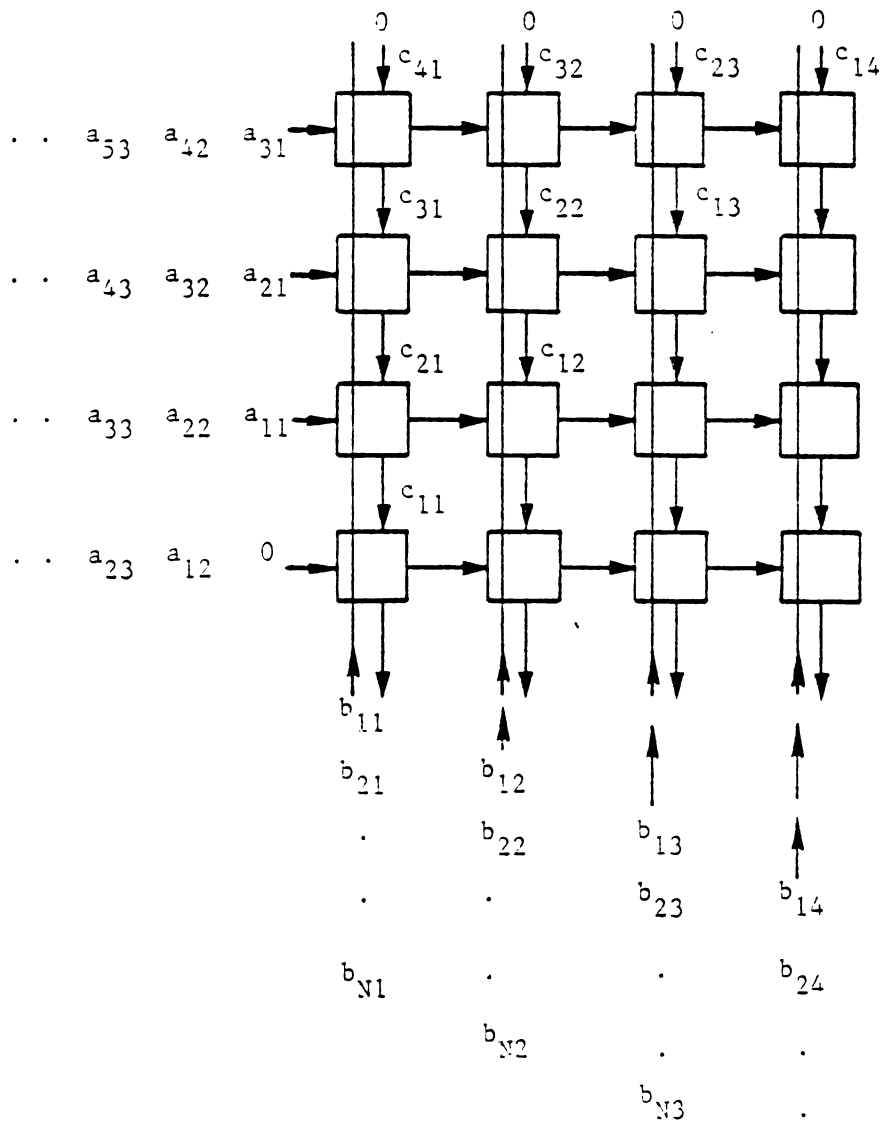


Figure 4.8.(b) Band Matrix/ Dense Matrix Multiplication.

direction in our case while in [9] broadcasting is used in both directions. The array configuration described in Subsection 4.2.1, for dense-matrix/dense-matrix multiplication, requires NxN processors to solve the above problem, whereas the array described here needs only WxW processors to solve the same problem.

The control vector for realizing the array, shown in Figure 4.8(b), is determined as follows. The switching cells are still used in broadcast configuration, connecting A1 to both B1 and B2. From Table 4.1, control bits 001 are chosen. The computation cells perform the following functions:

DRA <-- G ; E <-- DRA ; C <-- F + DRA*B .

The control bits for realizing the above functions are 010, as seen from Figure 4.4. So, in order to realize the configuration for efficient band-matrix/dense-matrix multiplication, the control vector for the MRAP is as follows:

Control Vector (in HEX digits):

28A28A25145144A28A289451451

4.2.3 BAND-MATRIX/BAND-MATRIX MULTIPLICATION

Let A and B be two band matrices with bandwidths W1 and W2, and the product matrix C = A*B is to be computed. The array configuration for this computation is shown in Figure 4.8(c), for W1=W2=4. This algorithm is the same as proposed by Kung in [31] for efficient band-matrix multiplication. It can be noted that the data-flow is in three dimensions in this case. This array can be realized by setting switches

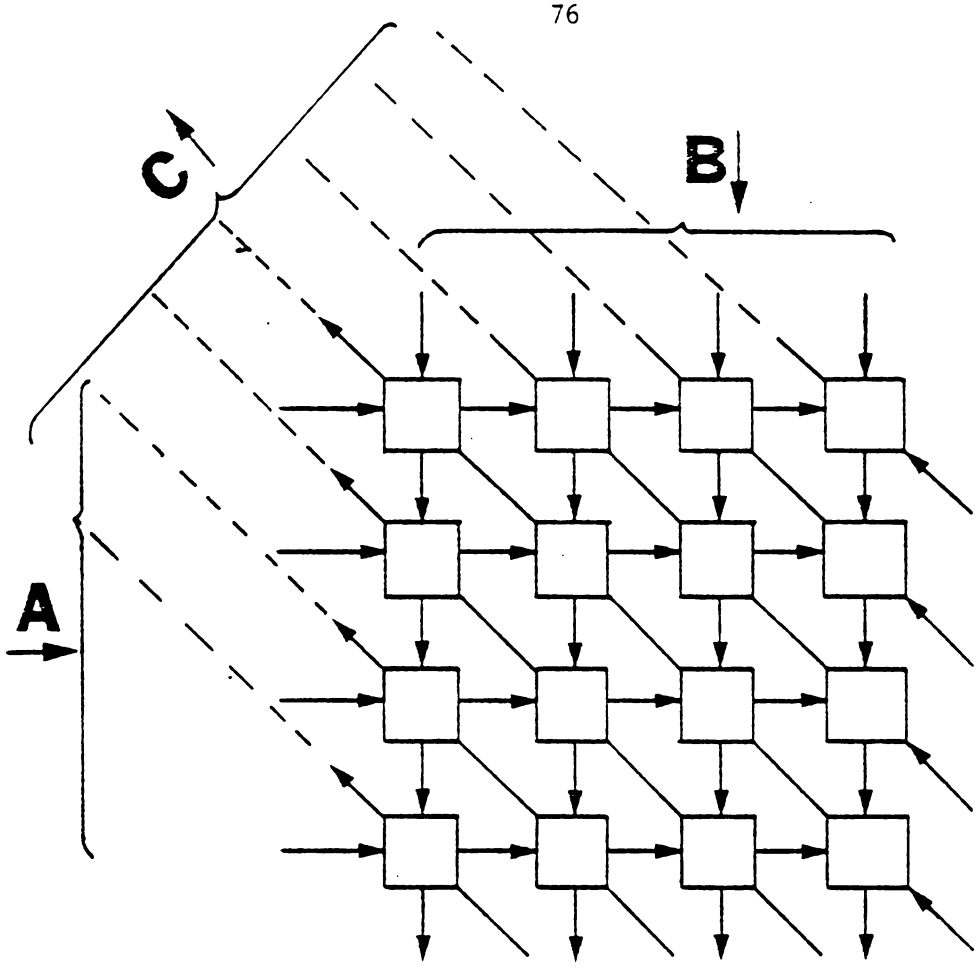


Figure 4.8.(c) Band Matrix/ Band Matrix Multiplication.

to support diagonal data-flow. Control bits for the switching cells are chosen to be 011 (from Table 4.1), and the control vector for computation cells is 011 (from Figure 4.4). The computation cells realize the following functions.

DRA <-- G ; E <-- DRA ; G <-- D + DRA*F ; C <-- F .

The following control vector configures the MRAP to realize Kung's hexagonal systolic array for narrow band-matrix multiplication:

Control Vector (in HEX digits):

6DB6DB6DB6DB6DB6DB6DB6DB

4.2.4 RECURSIVE FILTERING

An MRAP containing nxn computation cells can be partitioned into n independent, one-dimensional, linear systolic arrays. A one-dimensional linear array consisting of n processors can be used to solve an n'th order recurrence problem. Recursive digital filtering in signal processing is an example where a recurrence equation is used. An n'th order recurrence problem is defined as follows:

Given $x_0, x_{-1}, \ldots, x_{-n+1}$, compute $x_1, x_2, \ldots$, defined by

$$x_i = F_i(x_{i-1}, \ldots, x_{i-n}), \text{ for } i > 0; \qquad (4.3)$$

where F_i is a given recurrence function.

For a large class of recurrence functions, an n'th order recurrence problem can be solved in real time on n linearly connected processors, such that a new x_i can be obtained at the output at regular time

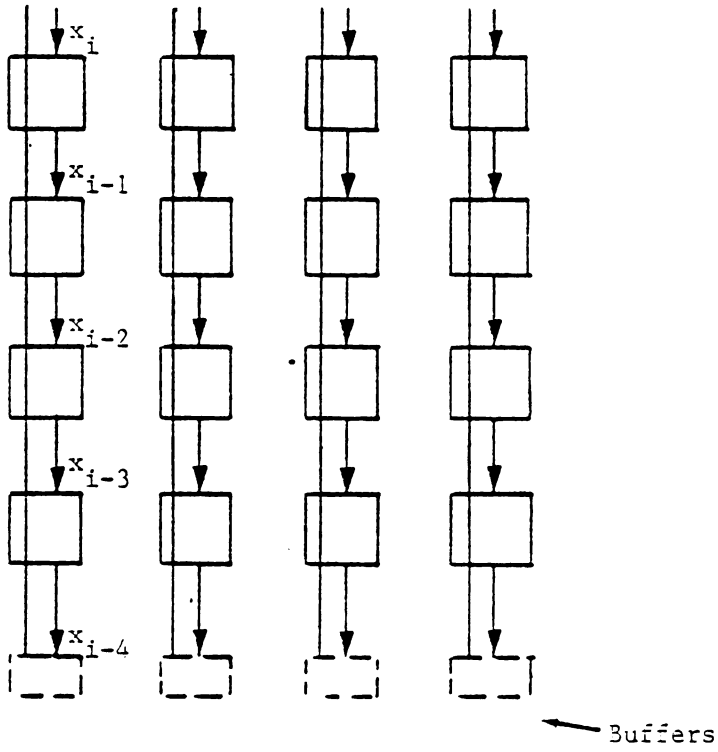


Figure 4.8.(d) One Dimensional Arrays for Filtering.

intervals [31]. A linearly connected broadcast array with n processors and one buffer can solve the n'th order recurrence problem [21]. Figure 4.8(d) shows the array configuration for implementing recursive filtering algorithm. This algorithm is given in detail in [21], where it is shown that this algorithm is optimal in the limit. The coefficients of the recursive equation are preloaded into the computation cells. The control vector for this configuration is as follows:

Control Vector (in HEX digits):

30C30C26186184C30C309861861

4.3 PERFORMANCE ANALYSIS AND COMPARISON

Performance of an array processor can be measured by the efficiency of parallel algorithms executed on it. Huang and Abraham have developed a criteria for measuring the efficiency for parallel algorithms, based on the Space-Time-Bandwidth complexity [21]. Bandwidth, here, refers to the data transfer bandwidth and is defined as the maximum number of words which have to be transferred through the I/O ports of the boundary cells in a time unit (a time unit is the period of time a processing element performs an operation). Let

P = number of required processing elements;

T = turnaround time of the computation;

B = data transfer bandwidth;

C = number of operations in the computation task;

I = number of input and output operands transferred.

It can be seen from the above, that for any computation task,

$$T \geq C/P, \tag{4.4}$$

and

$$T \geq I/B. \tag{4.5}$$

Thus,

$$PBT^2 \geq CI. \tag{4.6}$$

The product of P, B, and T^2 is the Space-Time-Bandwidth complexity of an algorithm executed in a processor array, and the product CI is the lower bound of this complexity. A measure of the efficiency of an algorithm is then defined as

$$R = PBT^2/CI. \tag{4.7}$$

A lower value of R means a better performance and R=1 means an optimally implemented algorithm.

In the performance analysis of any semisystolic system, we must consider the time delays associated with the rippling of logic when deriving performance parameters such as throughput and response time. This is important because semisystolic systems do not meet the design criterion of extensibility as systolic systems do. We can not cascade many semisystolic arrays together to form an arbitrarily large array because the clock cycle time, which depends upon the delay due to rippling, may asymptotically become arbitrarily large. In the MRAP structure, data ripples through the switching cells (SCs) whenever global

computations are required, as in case of semisystolic algorithms. We take into account the time delay due to this rippling in our analysis by defining the clock cycle time of the MRAP as follows:

$$T = T_c + m*T_r, \tag{4.8}$$

where,

T = clock cycle time for the MRAP ,

T_c = computing time for a computation cell (CC) ,

T_r = propagation delay of a switching cell (SC) ,

m = maximum number of SCs in a ripple path .

In pure systolic systems, throughput and response time are derived in terms of the computing time of a computation cell (T_c). For most of the matrix arithmetic algorithms, the computation cell is a multiplier accumulator so T_c approximately equals the time required for one multiplication and one addition. Data transfer bandwidth of the array is defined as the maximum number of words transferred through the I/O ports of the array boundary in one time unit T_c. We can write the clock cycle time for the MRAP in terms of T_c time units as follows:

$$T = 1 + m*k, \qquad \text{where } k = T_r/T_c. \tag{4.9}$$

In the remainder of this section, we analyze the performance of the MRAP for implementing various matrix multiplication algorithms and compare this performance with other existing systolic computing structures. This analysis also gives us the upper bounds on the size of

specific semisystolic arrays, beyond which they do not remain efficient because of extremely long clock periods. Space-Time-Bandwidth complexity of an algorithm is used as a criterion for evaluating the performance of an array processor.

4.3.1 DENSE-MATRIX/DENSE-MATRIX MULTIPLICATION

For dense-matrix multiplication, as described in Subsection 4.2.1, the total number of operations in the computation task is N^3 and the number of operands is $3N^2$. So the lower bound on the complexity of this algorithm is

$$CI = 3N^3. \tag{4.10}$$

The cmplexity of the algorithm when implemented on the MRAP, as shown in Figure 4.8(a), is derived as follows:

$$P = N^2 ,$$

$$B = 2N ,$$

$$T = 3N*(1 + kN) ,$$

and

$$PBT^2 = 18N^3*(1 + kN)^2. \tag{4.11}$$

Therefore,

$$R = 6*(1 + kN)^2. \tag{4.12}$$

The same computation, when executed on the systolic array in [31], gives the following complexity and performance:

$$P = 3N^2 ,$$

$$B = 2N ,$$

$$T = 5N ,$$

and

$$PBT^2 = 150*N^5 . \qquad (4.13)$$

Therefore,

$$R = 50 . \qquad (4.14)$$

Comparing Equations (4.12) and (4.14), we get

$$6*(1 + kN)^2 < 50 ,$$

or

$$kN < 1.9 . \qquad (4.15)$$

For a value of k = 0.05, the MRAP algorithm performs better for N < 38.

4.3.2 BAND-MATRIX/DENSE-MATRIX MULTIPLICATION

For band-matrix/dense-matrix multiplication, as described in Subsection 4.2.2, the number of computations, C, approaches N*W*W and the number of the input and output operands, I, approaches 3*W*N. So the lower bound on this algorithm is

$$CI = 3N^2W^3. \tag{4.16}$$

The algorithm, as implemented on the MRAP, is shown in Figure 4.8(b). The Space-Time-Bandwidth (PBT^2) complexity of this algorithm can be derived as follows:

$$P = W^2 ,$$

$$B = 3W ,$$

$$T = (N + 2W)*(1 + kW),$$

and

$$PBT^2 = 3W^3*(N + 2W)^2*(1 + kW)^2. \tag{4.17}$$

Therefore, from Equations (4.7), (4.16) and (4.17),

$$R = (N + 2W)^2*(1 + kW)^2/N^2,$$

and,

$$R \simeq (1 + kW)^2, \text{ when } N >> W. \tag{4.18}$$

The performance of the systolic algorithm, given in [31] for solving the same problem, can be derived as follows:

$$P = (N + W)*W \simeq N*W \qquad \text{when } N >> W ,$$

$$B = (2/3)*(N + 2*W) \simeq 2N/3 ,$$

$$T \simeq 4N ,$$

and

$$PBT^2 = 32N^4W/3. \tag{4.19}$$

Therefore,

$$R = 32N^2/9W^2. \tag{4.20}$$

Performance of the MRAP can be compared with Kung's systolic array by looking at Equations (4.18) and (4.20). It can be observed that the performance of the MRAP is not affected by the size N of the matrix, whereas the performance of systolic array degrades with the size N of the input matrix. The MRAP performs better as long as the result from Equation (4.18) is less than that from Equation (4.20). Comparing Equations (4.18) and (4.20), we get the following condition when the MRAP performs better:

$$(1 + kW)^2 < (32/9)*N^2/W^2,$$

and,

$$kW < 1.9*N/W, \text{ when } N >> W. \tag{4.21}$$

Since typically k << 1 and W << N, the above condition is easily met. For an array involving 16-bit integer multiplier-accumulator (MAC) cells, if a 16 x 16 multiplier array and an accumulator are used in the MAC cell, the multiply-accumulate time is approximately equal to 60 gate-delays (assuming that a full-adder has a propagation time equal to three gate-delays) [55]. The propagation delay in the switching cell is equal to three gate-delays (see Figure 4.5). So, the value of k is approximately 0.05, which means that the condition when the MRAP performs

better is W < 38*N.

4.3.3 BAND-MATRIX/BAND-MATRIX MULTIPLICATION

For band-matrix multiplication, as described in Subsection 4.2.3, the MRAP is configured as a pure systolic array and the algorithm given in [31] is implemented on it. The performance of the MRAP, for this computation, is analyzed as follows:

$$P = W_1 * W_2,$$

$$B = 2*(W_1+W_2)/3,$$

$$T \simeq 3N*(1 + k),$$

So,

$$PBT^2 = 6*(W_1+W_2)*W_1*W_2*N^2*(1 + k)^2, \tag{4.22}$$

and

$$CI = 2*(W_1+W_2)*W_1*W_2*N^2. \tag{4.23}$$

Therefore,

$$R = 3*(1 + k)^2. \tag{4.24}$$

For the systolic array in [31], the value of R is 3. So, the performance of the MRAP nearly equals that of Kung's systolic array, which is very efficient for narrow band-matrix multiplication. The MRAP performance is degraded by a factor of $(1 + k)^2$, due to the propagation delay in the switching cells. Since the value of $k \ll 1$, this degradation in performance is insignificant.

4.4 FAULT TOLERANCE

The MRAP architecture incorporates fault tolerance in its design due to its properties of reconfigurability and programmability at individual cell level. In MRAP, switch settings are used for data routing among computation cells. An MRAP architecture can be viewed as many reconfigurable pipelines working together. If a cell in the array fails, the array might be able to be reconfigured to bypass the faulty cell with data now flowing through the properly functioning cells. The performance of the array may be degraded, but the entire system does not fail. It is assumed here that the fault detection is done by the host and faulty cells can be identified. This is possible in MRAP structure due to the presence of LSSD latches and Scan-In and Scan-Out lines which provide both observability and controllability in the system. Inclusion of data registers in the scan path provides testability for the functional cells and also facilitates the initialization of data registers within the cells at the time of preprocessing the chip.

Some examples shown below demonstrate fault tolerance capability of the MRAP. Figure 4.9 shows computation cells of the MRAP, configured to implement linear arrays to solve recursive filtering problems of fourth order. 'F' indicates a faulty cell, 'S' shows a spare cell and 'U' denotes a used cell. If a fault occurs in a computation cell, the column containing that computation cell shuts down and other working cells in this column become spares, as shown in Figure 4.9(a). Figure 4.9(b) shows how spare cells can be used to mask out two other faulty cells A and B such that the array has the same performance as with only one

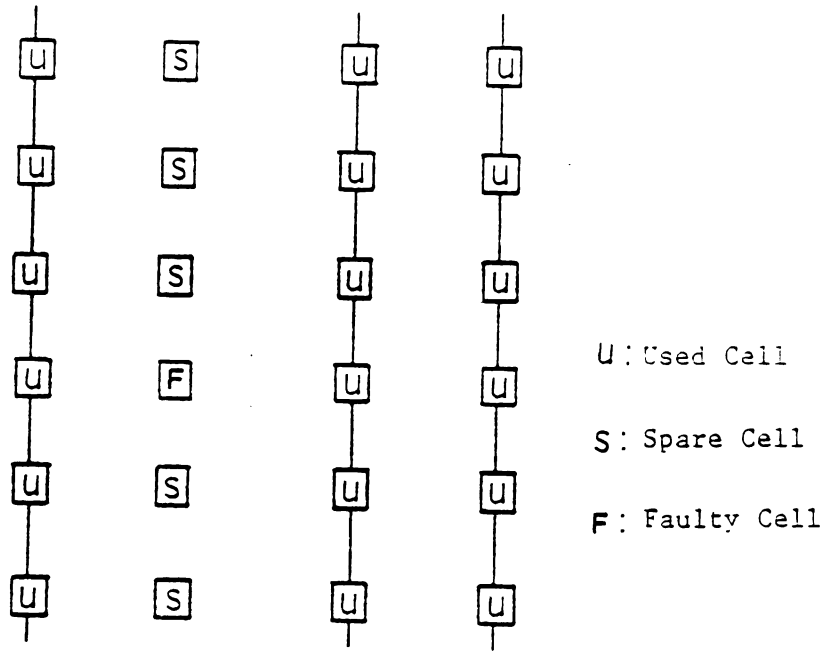


Figure 4.9.(a) The MRAP with One Faulty Computation Cell.

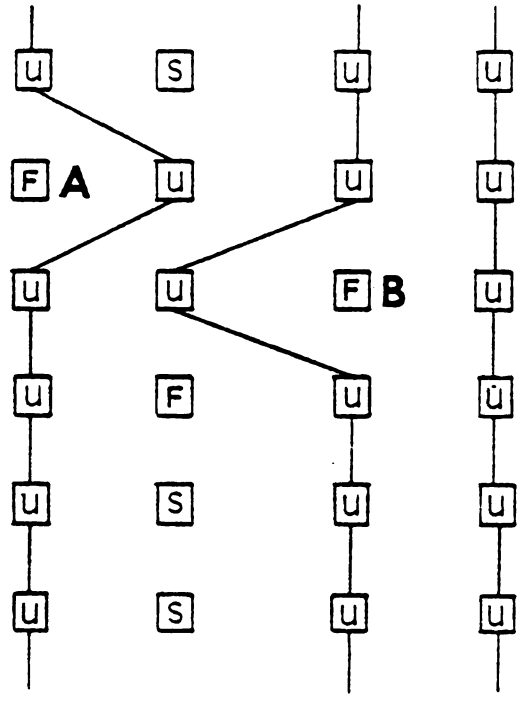


Figure 4.9.(b) Spare Cells are used for Fault Masking.

fault. More faults may cause more columns to be shut down, but that will provide even more spare cells to mask out further faults. So the system can fail eventually but with graceful degradation.

When two-dimensional algorithms are implemented on the MRAP, the fault tolerance becomes more difficult and degradation occurs at a higher rate. This is so because data flows in multiple directions and the output from a faulty cell will be propagated in multiple pipelines which share that computation cell. In this case, a single faulty cell will cause the shut down of all the pipelines which share that cell. All the working cells in the shut down pipelines become available as spares. Figure 4.10 illustrates how data-flow can be reconfigured to mask out faults due to failed cells A and B.

4.5 SUMMARY AND DISCUSSION

In this chapter, we presented the design and analyzed the performance of a VLSI compatible two-dimensional mixed systolic array architecture called multipurpose reconfigurable array processor (MRAP). This array processor can implement efficiently various systolic and semisystolic algorithms. Level-sensitive scan design is employed to achieve reconfigurability and multifunctionality. This design approach provides fault tolerance capabilities in the mixed systolic array architecture.

In general, the MRAP performs better than systolic arrays for specific semisystolic algorithms. We developed expressions for performance of the MRAP when executing various algorithms. These

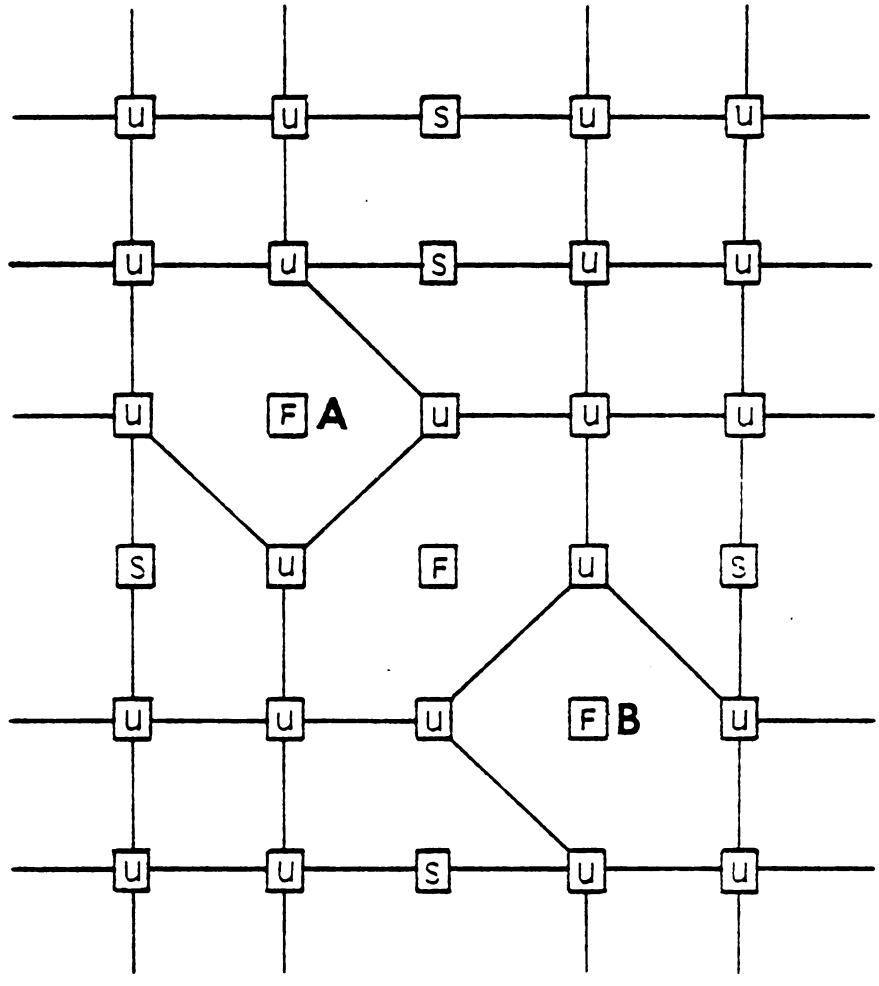


Figure 4.10. Fault Tolerance in the MRAP for 2-Dimensional Algorithms.

expressions determine the conditions under which a certain algorithm can be efficiently implemented on the MRAP according to a specific performance criterion. The performance of MRAP degrades with the array size for semisystolic algorithms because the propagation delay due to the switching cells becomes more significant in large arrays and tend to dominate the execution time. However, when systolic algorithms are implemented on the MRAP, the array performance is not affected by the size. It can be seen from the expression developed for the performance of band-matrix multiplication algorithm in Section 4.3, that the degradation occurs by a factor of $(1+k)^2$. This degradation is quite insignificant because the degradation factor is a constant, and $k<<1$. This is a relatively small penalty on the performance in view of the reconfigurability and algorithmic flexibility that we achieve with this architecture.

Since amplitude levels are regenerated in passing through the switching cells, no broadcasting bus or large bus drivers are required in the implementation of the MRAP. In general, there are two main disadvantages associated with data broadcasting. First, large broadcasts can not be implemented in a single communication cycle, because the broadcast delay can dominate the execution time of an algorithm. Second, large drivers are required for broadcasting in order to drive the combined load of all the cells connected to the broadcasting bus. In MRAP, broadcasting is implemented through switching cells and each switching cell output has to drive a load of only one gate input. As switching cells are used for reconfigurability purposes, the elimination of broadcasting bus or drivers come as an additional benefit in the MRAP

design.

In the next chapter, we present a mathematical model for reconfigurable mixed systolic array processors and a step-by-step procedure to automatically generate control vectors required for configuring the array to implement a specific algorithm. The procedure is illustrated by mapping the finite impulse response filtering algorithm and the priority queue algorithm into the linear reconfigurable systolic array.

CHAPTER 5

ALGORITHM IMPLEMENTATION ON MIXED SYSTOLIC ARRAYS

In the previous chapters, we presented a methodology for loading and implementing the configuration control structure in mixed systolic array processors, to establish a desired interconnection pattern required for algorithm implementation. This methodology employs LSSD techniques, and the control structure for a particular configuration is loaded into the array as a binary control vector in a bit-serial fashion. Efficient implementation of algorithms on MSA processors requires exploitation of parallelism in the algorithm and mapping of the algorithm communication structure into the MSA processor interconnection structure. This chapter deals with the issues of relating MSA's architectural model with parallel algorithms, and investigating procedures for generation of control code required for implementing algorithms on MSA architectures. In this chapter, we present a mathematical formalism for modeling reconfigurable mixed systolic array architectures and a methodology for reconfiguring the array, by mapping the communication structure of an algorithm into the interconnection structure of the array. A step-by-step procedure is presented to map a given algorithm into the mixed systolic array architecture and then generate the control code required to implement the corresponding interconnection structure. The mapping procedure is based on the time and space transformations of data dependence vectors of the algorithm. These transformations describe the data-flow and timing of

the algorithm and dictate the interconnection structure required for its implementation.

Sections 5.1 and 5.2 present mathematical models for reconfigurable MSA processors and parallel algorithms, respectively. A procedure for mapping algorithms with MSA processors is given in Section 5.3. And, this procedure is used in Section 5.4 to map two sample algorithms, the finite impulse response (FIR) filtering algorithm and the priority queue algorithm, into the linear reconfigurable systolic array.

5.1 A MODEL FOR RECONFIGURABLE MIXED SYSTOLIC ARRAYS

A reconfigurable mixed systolic array processor is a 5-tuple (Q^n, D, F_c, F_s, R), where

$Q^n \subset Z^n$ is the index set of the array processor;

D is a transformation on Q^n to $B = \{0, 1\}$, i.e., $D: Q^n \rightarrow B$ describes if an index corresponds to a switching cell or a computation cell;

F_c is a set of the functions a computation cell can perform;

F_s is a set of interconnection configurations of a switching cell; and

$R \in Z^{nxr}$, $r \in I$, is a matrix of cell-to-cell interconnection primitives.

This mathematical model is general enough to represent conventional systolic arrays with fixed structures as well as multipurpose reconfigurable arrays containing programmable interconnection cells and multifunctional arithmetic and logic units.

The index set Q^n refers to an n-dimensional array structure. Most of the practical arrays are either linear (n = 1) or two-dimensional (n = 2). Each cell in the array is represented by its index $\bar{q} \in Z^n$, such that for all $\bar{q} \in Q^n$,

$$\bar{q} = [q_0, q_1, \ldots, q_n]; \; q_i \in Z, \; 0 \leq i \leq n.$$

D is a mapping on the index set Q^n to a set of binary digits {0,1} such that D specifies every cell in the array to be either a computation cell (CC) or a switching cell (SC). So, for all $\bar{q} \in Q^n$

$D(\bar{q}) = 0$ if $\bar{q}$ corresponds to a SC,

$= 1$ if $\bar{q}$ corresponds to a CC.

F_c is a set of computations (arithmetic or logical) performed by the computation cells.

F_s is a set of all interconnection configurations of a switching cell.

For VLSI structures, we can assume that all CCs and SCs are identical in order to preserve modularity and extensibility. In a case where cells are not identical, a surjective mapping f_c can be defined from Q^n to F_c such that $f_c: Q^n \dashrightarrow F_c$, where f_c associates a subset $f_c(\bar{q}) \subseteq F_c$ for index $\bar{q} \in Q^n$. R is a matrix of cell-to-cell interconnection primitives such that $R = [\bar{r}_1 \; \bar{r}_2 \; \bar{r}_3 \ldots \bar{r}_p]$, where $\bar{r}_j$, $1 \leq j \leq p$, is a column vector indicating directed communication link. If $\bar{r}_j \in R$, then for any $\bar{q} \in Q^n$, $\bar{q}$ is connected to $\bar{q}' = q + \bar{r}_j$ if $\bar{q}' \in Q^n$ and is connected to an I/O port if $\bar{q}' \notin Q^n$.

VLSI arrays with fixed structures, such as in [4,32,35], can be represented by the above model with $D(\bar{q}) = 1$ for all $\bar{q} \in Q^n$ and F_s being an empty set. In the remainder of this section, we present two examples

to illustrate how the above model represents reconfigurable array structures.

EXAMPLE 5.1.1

The Linear Reconfigurable Systolic Array (LRSA), described in Chapter 3 and shown in Figure 5.1, is represented as (Q^2, D, F_c, F_s, R) where

$$Q^2 = \{\bar{q} = (q_1, q_2)^T : 0 \leq q_1 < N, 0 \leq q_2 < 2\}; \tag{5.1}$$

$$D(\bar{q}) = q_2 \bmod 2 ; \tag{5.2}$$

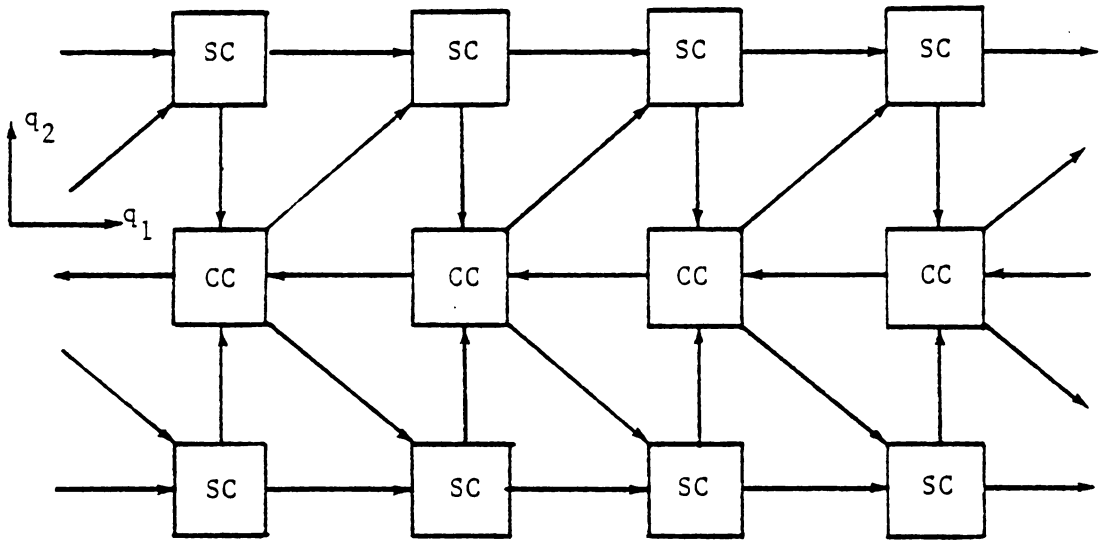
$$\begin{aligned} F_c = \{&(O_1 \leftarrow\!- I_1, O_2 \leftarrow\!- I_2, O_3 \leftarrow\!- A * I_2 + B * I_1 + I_3), \\ &(O_1 \leftarrow\!- B, O_2 \leftarrow\!- A * I_1 + I_2, B \leftarrow\!- I_1), \\ &(O_1 \leftarrow\!- I_3, O_3 \leftarrow\!- I_3), \\ &(O_1 \leftarrow\!- I_1, O_3 \leftarrow\!- I_3, R \leftarrow\!- R \text{ AND } (I_1 \text{ EQ } I_3), \\ &(O_1 \leftarrow\!- I_1, O_2 \leftarrow\!- I_1 * I_2 + A)\}; \end{aligned} \tag{5.3}$$

$$F_s = \{(S_3 = S_1, S_4 = S_2), (S_3 = S_2, S_4 = S_1)\}; \tag{5.4}$$

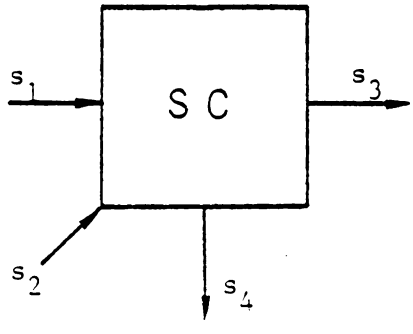
$$R = \begin{bmatrix} 1 & -1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 & 1 & -1 \end{bmatrix}. \tag{5.5}$$

For an LRSA with N=4, the Q^2 and D are as follows:

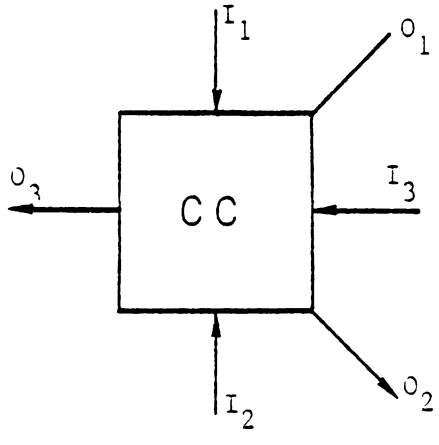
Q^2 = {00, 10, 20, 30, 01, 11, 21, 31, 02, 12, 22, 32};



(a) Structure of the LRSA.



(b) A Switching Cell (SC).



(c) A Computation Cell (CC).

Figure 5.1. The Linear Reconfigurable Systolic Array (LRSA) Architecture.

$D(q_1, q_2)$ =

q_2 \ q_1	0	1	2	3
0	0	0	0	0
1	1	1	1	1
2	0	0	0	0

EXAMPLE 5.1.2

The Multipurpose Reconfigurtable Array Processor (MRAP), described in Chapter 4 and shown in Figure 5.2, is represented as (Q^2, D, F_c, F_s, R) where

$$Q^2 = \{\{\overline{q}=(2i,2j)\} \cup \{\overline{q}=(2i+1,2j+1)\},\ 0 \le i \le N-1,\ 0 \le j \le M-1\}; \tag{5.6}$$

$$D(\overline{q}) = (q_1 + 1) \bmod 2; \tag{5.7}$$

$$R = \begin{bmatrix} 1 & -1 & 1 & -1 & 0 & 2 \\ 1 & -1 & -1 & 1 & 2 & 0 \end{bmatrix}. \tag{5.8}$$

F_c and F_s can be written, in a fashion similar to Example 5.1.1, as the set of computations performed in the computation cell and the set of interconnection configurations in the switching cell, respectively.

5.2 A MODEL FOR ALGORITHMS

It is important to describe a mathematical model for algorithms in order to map them into the mixed systolic arrays. We consider the algorithm model defined in [47], which contains information about the algorithm index set, the computations performed at each index point, the

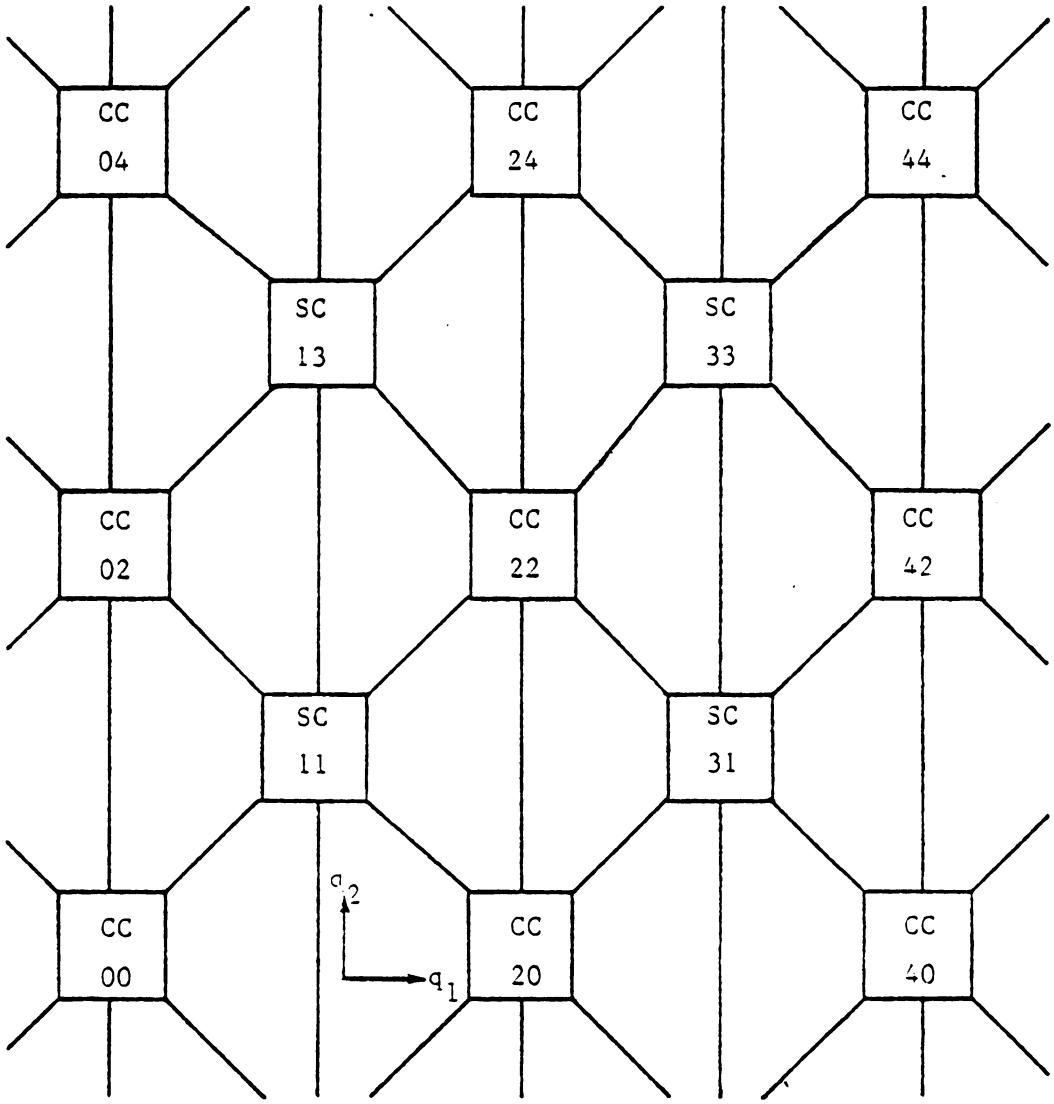


Figure 5.2. Structure of the Multipurpose Reconfigurable Array Processor.

data dependencies which dictate the algorithm communication requirements and the algorithm input and output variables.

An algorithm A is a 5-tuple

$$A = (J^n, C, G, X, Y) \tag{5.9}$$

where,

$J^n \subset Z^n$ is an index set;

C is the set of computations of A;

G is the set of dependencies, i.e. a relation from Z^n to the set of all pairs $(v, \bar{j})$ where $\bar{j} \in J^n$ and v is a variable such that $v \leftarrow C(\bar{j}')$, $\bar{j}' \in J^n$.

X is the set of input variables of A;

Y is the set of output variables of A such that if a variable $v \in Y$ then $v \in X$ or $v \leftarrow C(\bar{j})$ for some $\bar{j} \in J^n$.

The set of output variables is a subset of the union of the sets of input variables and the generated variables as a result of some computation $C(\bar{j})$. G is a Z^{nxq} matrix whose columns are dependence vectors. A dependence vector $\bar{d}$ is defined as $\bar{d} = \bar{j} - \bar{j}'$ if a variable v is used by $C(\bar{j})$ and generated by $C(\bar{j}')$, i.e. $\bar{j}$ is the index point at which v is used and $\bar{j}'$ is the index point at which v is generated. The dependency matrix G is then written as

$$G = [\bar{d}_1 \ \bar{d}_2 \ . \ . \ . \ \bar{d}_q]. \tag{5.10}$$

The conditions for the validity of the above model are given in [15]. This model can conveniently represent the numerical algorithms normally written in the form of nested loops in conventional high-level language. A distinct class of algorithms are those for which data dependencies are constant, i.e. the computations repeat at different index points over the entire index set. Algorithms belonging to this class are easier to map into VLSI arrays [30]. We present two examples to illustrate how the dependence vectors are generated for the algorithm model. Sometimes, a reindexing of variables is required to get constant dependence vectors. This will be demonstrated in Example 5.2.2, for the finite impulse response filtering algorithm.

EXAMPLE 5.2.1

Consider the algorithm described as follows:

```
FOR i = 0 TO N
  FOR k = 0 TO N
    a(i,k) = a(i-1,k-1) * b(i-1,k)
    b(i,k) = a(i-1,k) + b(i,k+1)
  END k;
END i ;
```

The model for the above algorithm is as follows:

$J^2 = \{(j_1,j_2): \ 0 \leq j_1,j_2 \leq N\};$

$C = \{C(j_1,j_2) : \ a(j_1,j_2) = a(j_1-1,j_2-1) * b(j_1-1.j_2),$

$b(j_1,j_2) = a(j_1-1,j_2) + b(j_1,j_2+1)\};$

In this example, at every point in the index space an addition and a multiplication are performed. The dependencies which dictate the algorithm communication requirements can be described as difference vectors of index points where a variable is used and where it is generated. The dependence vectors are as follows:

$\bar{d}_1 = (1 \quad 1)^T$, between variables a(i,k) and a(i-1,k-1);

$\bar{d}_2 = (1 \quad 0)^T$, between variables a(i,k) and b(i-1,k);

$\bar{d}_3 = (1 \quad 0)^T$, between variables b(i,k) and a(i-1,k);

$\bar{d}_4 = (0 \quad -1)^T$, between variables b(i,k) and b(i,k+1).

The dependency matrix G is given below. A label below each column points out the generated variable pertaining to the dependence vector comprising that column.

$$G = [\bar{d}_1 \ \bar{d}_2 \ \bar{d}_3 \ \bar{d}_4] = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix}$$
$$\begin{matrix} a & b & a & b \end{matrix}$$

$X = \{(a(-1,j_2-1),\ b(-1,j_2)):\ 0 \leq j_2 \leq N+1\}$

$Y = \{a(j_1,j_2), b(j_1,j_2):\ 0 \leq j_1, j_2 \leq N\}$.

EXAMPLE 5.2.2

In this example, we illustrate how a Finite Impulse Response (FIR) filtering algorithm can be represented by the above model and in Section 5.4, we shall use this model to map the algorithm into a linear reconfigurable systolic array.

A Finite Impulse Response (FIR) Filter can be defined as

$$y_i = \sum_{j=0}^{M} a_j * x_{i-j} \ , \ 0 \leq i, \qquad (5.11)$$

where $X = \{x_{-M}, \ . \ . \ , x_0, x_1, \ . \ .\}$, is the input to the filter and $\{a_j: j=0, 1, \ . \ . \ , M\}$, are filter coefficients. $Y = \{y_i\}$ is the output of the filter. We can write the following algorithm for the above problem.

```
FOR i = 0 TO N
  FOR j = 0 TO M
    y(i,j) = y(i,j-1) + a(j) * x(i-j)
  END j;
END i;
```

In order to get dependencies, we first complete all the missing indices in all the variables. It can be noticed in the above algorithm that x(i-j) may be taken from the calculation of y(i-1,j-1) as

y(i-1,j-1) = y(i-1,j-2) + a(j-1) * x(i-1-j+1).

Similarly for (i-1,j) calculation, a(j) is used. So, we can write the algorithm as follows:

```
FOR i = 0 TO N
  FOR j = 0 TO M
    y(i,j) = y(i,j-1)+a(i-1,j)*x(i-1,j-1)
    a(i,j) = a(i-1,j)
    x(i,j) = x(i-1,j-1)
  END j;
```

END i;

For the above algorithm, data dependencies can be found to be $(0,1)^T$ for y, $(1,0)^T$ for a and $(1\ 1)^T$ for x. The dependency matrix can be written as

$$G = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad \begin{matrix} \\ \\ y\ \ a\ \ x \end{matrix} \tag{5.12}$$

So, the model for FIR filtering algorithm is (J^2, C, G, X, Y), where

$$J^2 = \{(j_1,j_2):\ 0 \leq j_1 \leq N,\ 0 \leq j_2 \leq M\}; \tag{5.13}$$

$$C = \{C(j_1,j_2): y(j_1,j_2)=y(j_1,j_2-1)+a(j_1-1,j_2)*x(j_1-1,j_2-1)\}; \tag{5.14}$$

$$G = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{5.15}$$

$$X = \{x(j_1-1,j_2-1):\ 0 \leq j_1 \leq N,\ 0 \leq j_2 \leq M\}; \tag{5.16}$$

$$Y = \{y(j_1,M):\ 0 \leq j_1 \leq N\}. \tag{5.17}$$

5.3 MAPPING ALGORITHMS INTO MIXED SYSTOLIC ARRAYS

Mapping an algorithm into a reconfigurable array processor, such as a mixed systolic array, requires the following three tasks. The first

task is to find all the dependence vectors in the algorithm and a time transformation on the index set of the algorithm such as to determine a valid execution ordering on the algorithm for VLSI implementation. The second task is to select a space transformation S whose interconnection structure can be implemented on the reconfigurable MSA by programming the switching cells. Finally, the control code is assigned to all the cells in the array and the control vectors are generated to be loaded into the array through LSSD scan-in line. This procedure provides the capability of automatically generating the configuration control vectors for implementing an algorithm on a mixed systolic array processor. Various candidate valid space and time transformations are considered and the one that is suitable for the given array processor, is selected.

Definition 5.3.1:

M is a set of control codes for the switching cells in the array. Assuming that a switching cell can have m different interconnection configurations, then we write

$$M = \{m_1, m_2, \ldots, m_m\} \text{ where } m_i \in I,\ 1 \leq i \leq m.$$

m_i is the control code for the ith interconnection configuration of the switching cell, in the set F_s. Elements of sets M and F_s have one-to-one correspondence. A possible set M for the switching cell shown in Figure 1(b) can be M= {0,1}, where a 0 implies a straight configuration ($S_3 = S_1$, $S_4 = S_2$) and a 1 implies an exchange configuration ($S_3 = S_2$, $S_4 = S_1$).

Definition 5.3.2:

N is a set of control codes for the computation cells in the array. Assuming that a computation cell can perform n different functions, then

$$N = \{n_1, n_2, \ldots, n_n\} \text{ where } n_i \in I, 1 \leq i \leq n.$$

Elements of sets N and F_c have a one-to-one correspondence.

Definition 5.3.3:

E is a set of unit vectors in the n-dimensional vector space of the index set such that $E = \{e_i: 0<i<p\}$, where p is the number of communication links associated with a CC and e_i is the direction of the communication link. A basis $\{u_1, u_2, \ldots, u_n\}$ for the vector space can be selected such that every communication link can be represented by the linear combination

$$e_i = a_{1_i} * u_1 + a_{2_i} * u_2 + \ldots + a_{n_i} * u_n.$$

For a 2-dimensional array in a mesh connected network, one possible basis is $u_1 = (1,0)$ and $u_2 = (0,1)$. For example, for Figure 5.3, we can define

$$E = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ -1 & 1 \\ -1 & -1 \\ 0 & -1 \\ 1 & -1 \end{bmatrix}^T \begin{matrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \end{matrix}$$

Definition 5.3.4:

H is a matrix of direction vectors for all communication links of a computation cell to its neighboring computation cells.

$$H = [h_1 \; h_2 \; \ldots \; h_r],$$

where h_i, $1 \leq i \leq r$, is a direction vector of a communication link and r

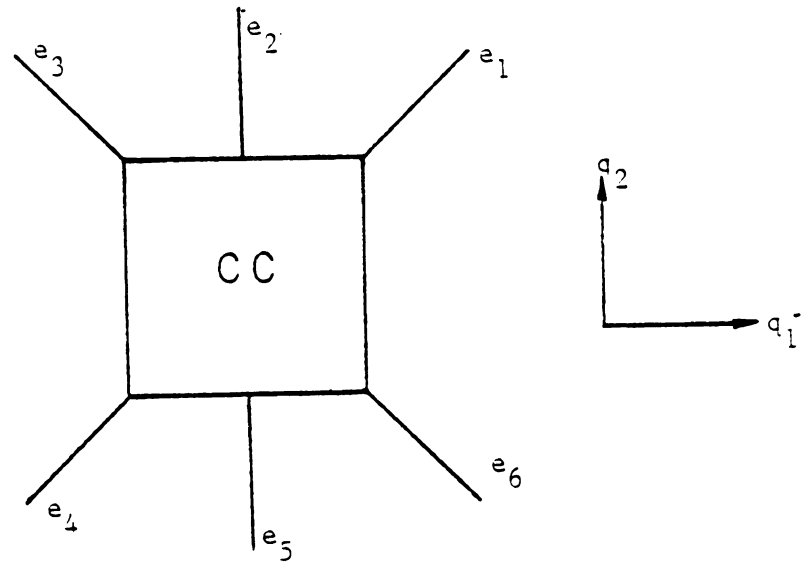


Figure 5.3. A Computation Cell with Communication Links.

is the total number of communication links connecting a computation cell to its neighboring computation cells.

Definition 5.3.5:

An LSSD scan path is defined as a sequence W from set A into the index set Q^n of the array. A is a finite set of positive integers, A = {1,2, . . ,N}, where N is the total number of cells in the array processor. So,

$$W = \{w_1, w_2, w_3, \ldots, w_N\},\ w_i \in Q^n \text{ and } i \in A.$$

W is determined by the physical implementation of the LSSD scan path on the chip layout. The control code for w_N is entered first and that for w_1 is entered the last.

5.3.1 PROCEDURE FOR MAPPING ALGORITHM INTO MIXED SYSTOLIC ARRAYS

We present a step-by-step procedure for generating the reconfiguration control vector for a given MSA structure, starting from a high-level language representation of an algorithm. An explanation of the procedural steps follows the procedure.

1. Find the set of data dependence vectors for all variables after pipelining the variables in the algorithm. Form the dependency matrix G, where each column in G corresponds to a data dependence vector.
2. Find all time transformations T which can map the index set of the algorithm J^n into the unidimensional time space. An optimal transformation is that which minimizes the execution time.

Select the optimal transformation for the first iteration and select a different transformation in each iteration. Conditions required to find valid transformations and the optimal transformation are given in the explanation following the procedure.

3. Find a space transformation, S, by solving the set of diophantine equations

$$S \cdot G = H \cdot K. \tag{5.18}$$

K is a matrix which indicates the utilization of the interprocessor communication links. $K = [k_{ji}]$, such that

$$k_{ji} \geq 0 \tag{5.19}$$

and

$$\sum_j k_{ji} \leq T \cdot \bar{d}_i. \tag{5.20}$$

If an S can be found, proceed to step 4;

otherwise go to step 2 to select another time transformation.

4. Form the transformation matrix $\Delta = \begin{bmatrix} T \\ S \end{bmatrix}$.

If Δ is a nonsingular matrix, proceed to step 5;

otherwise, go to step 3.

The product $\Delta \cdot G$ contains the information about the data-flow and timing of the algorithm implemented on the array processor.

5. Map every computation cell in the array to F_c and every switching cell to F_s.
6. Find the control vector, from the LSSD sequence W and the control codes defined for each cell. This binary vector is serially loaded into the array through the LSSD scan-in line to reconfigure the array for implementing a specific algorithm.

EXPLANATION OF THE PROCEDURE

Step 1 finds the data dependence vectors of the algorithm and forms the dependency matrix G. This can be done in an automatic manner for a large class of algorithms as shown in [30,45].

Step 2 determines an execution ordering by mapping the index set of an algorithm by a linear transformation T such that $T:J^n \dashrightarrow J'^1$, where J'^1 is a one-dimensional array consisting of positive integers. So, a valid execution ordering should satisfy the condition

$$T \cdot \bar{d}_i > 0 \text{ for all } \bar{d}_i \varepsilon G. \tag{5.21}$$

T is a (1xn) vector which maps the index set of the algorithm into the unidimensional time space. Total execution time of the algorithm is given by

$$t = \frac{\max[T \cdot (\bar{j}^1 - \bar{j}^2) + 1]}{\min\ [T \cdot \bar{d}_i]} \text{ for } \bar{j}^1, \bar{j}^2 \varepsilon J^n \text{ and } d_i \varepsilon G. \tag{5.22}$$

An optimal transformation is the one which minimizes t in the above equation. Optimal transformation is tried in the first pass, but if it

cannot be mapped then other suboptimal transformations are tried.

Step 3 is probably the most computationally intensive since a system of diophantine equations has to be solved. Existence of a valid transformation S indicates that the algorithm can be mapped on the mixed systolic array model (Q^n, D, F_c, F_s, R). S contains the information about the communication structure of the algorithm. Dimensions of the utilization matrix K are rxq, where r is the number of communication links from one CC to its neighboring CCs in different directions and q is the number of dependence vectors in the matrix G. Directions of data-flow correspond to those unit vectors in H, for which the rows of K has at least one nonzero element. Rows of K with all zero elements correspond to the directions in which the communication will not take place. Matrix K can be generated by combining the patterns satisfying Constraints (5.19) and (5.20), as columns of K.

Step 4 gives the information about the timing and data-flow of the algorithm. Product T.G gives the speed at which different variables travel and the product S.G describes their directions of travel.

Step 5 determines the configuration of each switching cell and the computation performed by each computation cell. This gives the control code to be stored in each cell. Sets M and N are used in conjunction with F_s and F_c to determine the appropriate control codes.

Step 6 generates the final reconfiguration control vector by concatenating the control codes for the cells in the sequence such that the binary vector loaded through the LSSD scan path will put the appropriate codes in all the cells.

5.4 EXAMPLES

In this section, we illustrate how the above procedure is used for generating configuration control vectors for implementing two algorithms, the finite impulse response filtering and the priority queue, on the linear reconfigurable systolic array (LRSA) described in Chapter 3. The structure of the LRSA is shown in Figure 5.1, and the mathematical model of this array is given in Example 5.1.1. We assume that the computation cell of the LRSA can perform a compare and exchange operation, in addition to the functions described in Equation (5.3). Compare and exchange is the basic operation needed in a priority queue algorithm. This operation, f_{c1}, is described as follows, with reference to Figure 5.1(b):

$$\begin{aligned} f_{c1} = (&O_1 \leftarrow\!- \text{Maximum}(I_1, I_2, I_3), \\ &O_2 \leftarrow\!- \text{Medium}(I_1, I_2, I_3), \\ &O_3 \leftarrow\!- \text{Minimum}(I_1, I_2, I_3)). \end{aligned} \tag{5.23}$$

The set of control codes for the computation cells, N, consists of six elements;

$N = \{1,2,3,4,5,6\}$,

where codes 1 through 5 imply the same computations as described in Chapter 3, and code '6' implies the compare and exchange operation described above. The set of control codes, M, for the switching cell consists of only two elements, since we assume that only two configurations of a switching cell are implementable; straight and

exchange. Assume $M = \{0,1\}$, where a 0 implies a straight configuration ($S_3=S_1$, $S_4=S_2$) and a 1 implies an exchange configuration ($S_3=S_2$, $S_4=S_1$) (see Figure 5.1(b)). The control codes for all the cells are eventually loaded in their binary form into the corresponding registers. Each switching cell has a one-bit control code register and each computation cell has a 3-bit control code register. For the LSSD scan path, as shown in Figure 5.4, the sequence W can be written as

$$W = \{30,31,32,22,21,20,10,11,12,02,01,00\}. \tag{5.24}$$

5.4.1 FINITE IMPULSE RESPONSE (FIR) FILTERING ALGORITHM

We demonstrate how an FIR filtering algorithm is mapped on the Linear Reconfigurable Systolic Array shown in Figure 1. The high-level language description of the FIR filtering algorithm and derivation of its dependency matrix G are given in the Example 5.2.2.

Now we follow the above procedure step by step, for generating the configuration control vector required to implement the FIR filtering algorithm on the LRSA.

Step 1:

From Section 5.2, the dependency matrix for FIR filtering algorithm is

$$G = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{5.12}$$
$$\quad\; y \;\; a \;\; x$$

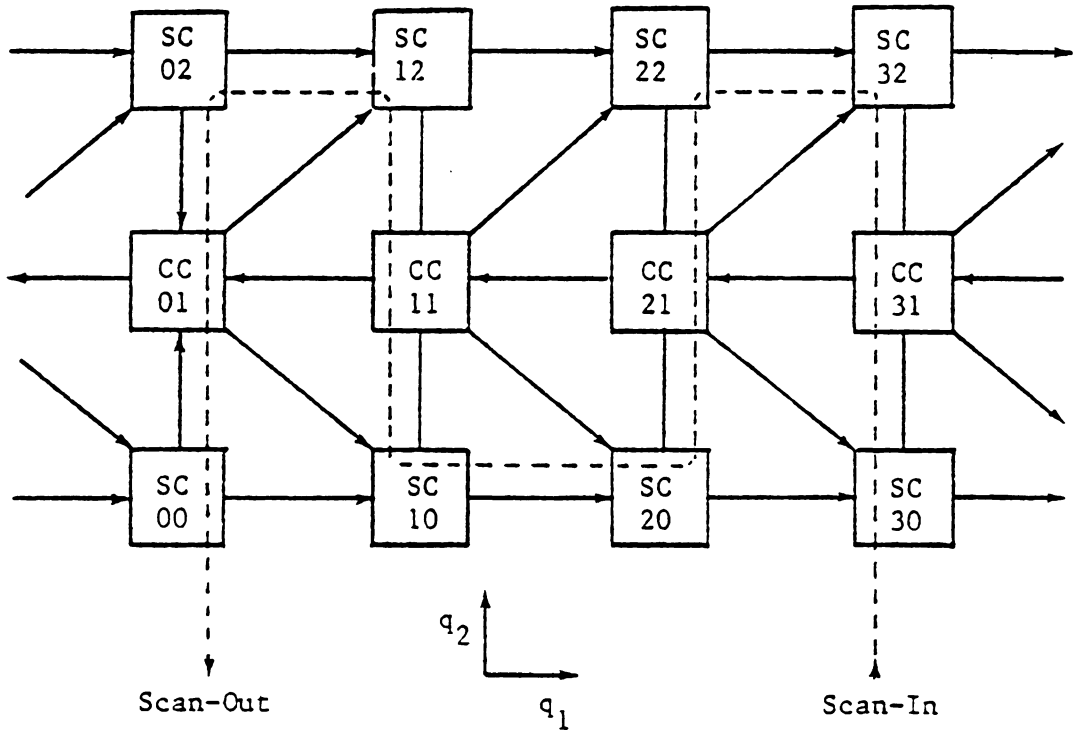


Figure 5.4. LSSD Scan Path Sequence in a LRSA.

Step 2:

Many time transformations T exist which give a valid execution ordering. We choose T = [1 1], which satisfies the condition $T\bar{d}_i > 0$ for any $\bar{d}_i \in G$ and minimizes the execution time of the algorithm.

Step 3:

Solve the set of diophantine equations

$$S \cdot G = H \cdot K . \tag{5.18}$$

H is a set of direction vectors, which indicates the CC-to-CC communication links. For the case of a linear array, like the LRSA,

$$H = [1\ 0\ -1]. \tag{5.25}$$

A 1 in H implies that the data flows towards right, a -1 implies the opposite direction for data-flow and a 0 implies that data stays at its location for later use. The utilization matrix chosen to solve the above equations is

$$K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} . \tag{5.26}$$

K is selected such that

$$k_{ji} \geq 0 \tag{5.19}$$

and

$$\sum_j k_{ji} < T\bar{d}_i. \tag{5.20}$$

The only solution for S, with matrix K as in Equation (5.26), is [0 1]. A row consisting of all zero elements indicates that communication link in the corresponding direction is not required. In the matrix K described above, the last row is a zero row. This row corresponds to the column of H containing the direction vector -1. So no data-flow occurs in this direction.

Step 4:

The transformation matrix is

$$\Delta = \begin{bmatrix} T \\ S \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}. \tag{5.27}$$

As Δ is a nonsingular matrix, we have a valid transformation. Now the data-flow and timing of the algorithm can be determined by looking at the product $\Delta \cdot G$. First row of $\Delta \cdot G$ corresponds to the timing and the other rows correspond to the directions in which different variables will travel.

$$\Delta \cdot G = \begin{bmatrix} T \cdot G \\ S \cdot G \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 1 \end{bmatrix}. \tag{5.28}$$

$$\begin{matrix} & & & & & & \\ & & & y & a & x & \end{matrix} \qquad \begin{matrix} & & & & & & \\ & & & y & a & x & \end{matrix}$$

The first row indicates that variable x travels twice as slow as variable y and both x and y travel in the same direction as seen from the second row of the above matrix. The element of the second row corresponding to variable a is 0, which indicates that variable a does not travel, but stays at the same location. Since x travels half as fast as y, a delay equal to one time step is introduced in the path of x at each computation cell. The algorithm implementation for this transformation is shown in Figure 5.5. This is the same algorithm as proposed by Kung in [32].

Steps 5 and 6:

The function performed by a computation cell is given as (see Figure 5.1(c))

$$(O_1 \leftarrow B,\ O_2 \leftarrow A * I_1 + I_2,\ B \leftarrow I_1),$$

and the switching cells are in straight configuration. The control code for each computation cell is $2_{10} = (010)_2$, and for each switching cell, the control code is (0). The LSSD sequence W is given as

$$W = \{30,31,32,22,21,20,10,11,12,02,01,00\}. \tag{5.24}$$

So the configuration control vector, V, is

$$\begin{aligned} V &= \{0,\ 010,0,0,010,0,0,010,0,0,010,0\}; \\ &= (001000010000100100)_2; \\ &= (21084)_{16}. \end{aligned} \tag{5.29}$$

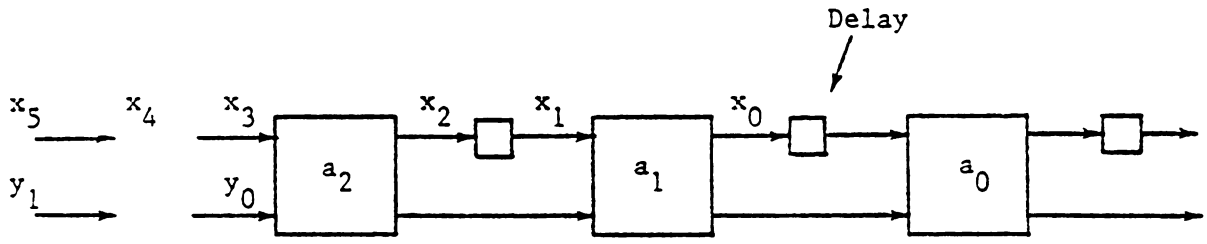


Figure 5.5. A Linear Systolic Array for FIR Filtering.

V is loaded serially with the right most bit first, through the LSSD scan-in line shown in Figure 5.4.

5.4.2 PRIORITY QUEUE:

A priority queue is an abstract data type based on the set model with the operations INSERT and DELETEMIN, as well as the operation for initialization of the data structures [1]. The elements of the set A have a priority function defined on them; for each element a, the priority of a, p(a), is a real number from some linearly ordered set. Operation INSERT(a,A) replaces the set A with the set A U {a} and the operation DELETEMIN returns some element of the smallest priority and deletes it from the set.

Let A be the sorted list of elements in the queue, A = $\{a_1, a_2, \ldots, a_n\}$, such that a_1 contains the smallest priority element and a_n contains the highest priority elements. Another array B = $\{b_1, b_2, \ldots b_n\}$ contains the elements which are being sorted. We define two constants 'max' and 'min', such that 'max' is higher than the highest possible priority and 'min' is lower than the least priority that can exist. Variables a_0 and b_0 are input variables.

The sequential algorithm can be described by the following procedure:

```
Procedure INSERT(x)
    begin
        b0 := x
        a0 := min
```

```
for j = 0 to n

   for i = 0 to n

      a_{i+1}^{j+1} = med(b_i^j, a_i^j, a_{i+1}^j)
      a_i^{j+1}     = min(b_i^j, a_i^j, a_{i+1}^j)
      b_{i+1}^{j+1} = max(b_i^j, a_i^j, a_{i+1}^j)

   end i

end j

end (INSERT);
```

Now we show how this algorithm is implemented by following the procedure given in Section 5.3.

Step 1:

There are nine data dependence vectors between the generated and the used variables. The data dependence matrix is then formed by combining these vectors as follows:

$$G = \begin{array}{c} \overbrace{\begin{bmatrix} a_i & a_{i+1} & b_i \end{bmatrix}}^{a_{i+1}} \quad \overbrace{\begin{bmatrix} a_i & a_{i+1} & b_i \end{bmatrix}}^{b_{i+1}} \quad \overbrace{\begin{bmatrix} a_i & a_{i+1} & b_i \end{bmatrix}}^{a_i} \\ \left[\begin{array}{ccc|ccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 & 0 & 1 & 0 & -1 & 0 \end{array}\right] \begin{array}{c} j \\ - \\ i \end{array} \end{array} \qquad (5.30)$$

Step 2:

Find a time transformation T satisfying the Constraint (5.21). One T which satisfies this constraint is T = (1 0). The execution time for this value of T is found from Equation (5.22), to be N+1 time steps for inserting a number in the queue of size N. This indicates that the inserted element moves at a rate of one position per time step. This

transformation also minimizes the execution time, hence is an optimal transformation. This can be verified by solving for the minima of Equation (5.22) with respect to variables t_1 and t_2, where $T = (t_1\ t_2)$.

Step 3:

Solve the diophantine equations $S \cdot G = H \cdot K$ to find a suitable S. Matrix H is the same as in the previous example, i.e., $H = [1\ 0\ -1]$. Size of the utilization matrix K will be 3x9 because there are 9 data dependence vectors in the matrix G and there are three possible directions of data flow. $S = [0\ 1]$ satisfies the above equation, for the following utilization matrix K

$$K = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}. \qquad (5.31)$$

So, the space transformation is $S = [0\ 1]$.

Step 4:

The transformation matrix is

$$\Delta = \begin{bmatrix} T \\ S \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Δ is a nonsingular matrix, so it is a valid transformation.

$$\Delta \cdot G = \begin{bmatrix} T \cdot G \\ S \cdot G \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & -1 & 0 \end{bmatrix}. \qquad (5.32)$$

The row corresponding to S.G implies that variable a_i should travel in the direction of the unit vector (-1), i.e., towards left, and the variables a_{i+1} and b_{i+1} should travel towards right. The systolic array formed by this transformation is shown in Figure 5.6. This array is similar to the systolic priority queue given in [41].

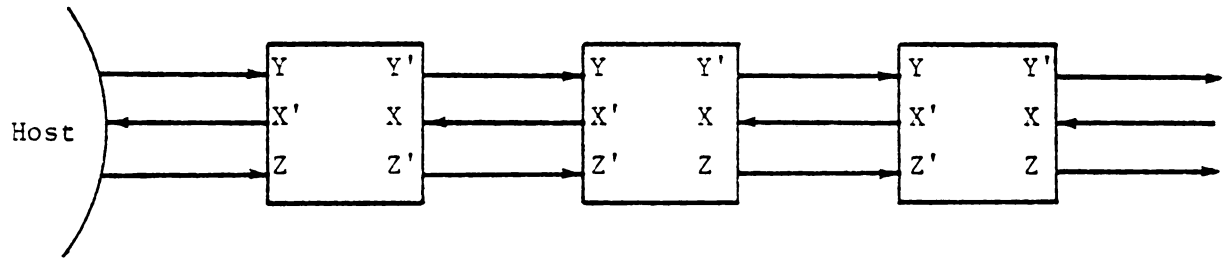
Steps 5 and 6:

The control code for each computation cell, to perform the compare and exchange operation, is $(6)_{10} = (110)_2$. The control code for each switching cell is 0. From the LSSD sequence W, we can write the configuration control vector, V, as

$$\begin{aligned} V &= \{0,110,0,0,110,0,0,110,0,0,110,0\}; \\ &= (01100011000110001100)_2 \\ &= (6318C)_{16}. \end{aligned} \tag{5.33}$$

5.5 SUMMARY AND DISCUSSION

We presented a mathematical formalism for modeling reconfigurable mixed systolic arrays and a step-by-step procedure for generating the control code required for implementing algorithms into mixed systolic array structures. The procedure presented in this chapter provides a facility for automating the reconfiguration process and the design of mixed systolic array architectures. Starting from a high-level language description of an algorithm, this procedure generates the final configuration control vectors required to be loaded into the array through the LSSD scan path in a bit-serial manner. One might argue that



X' = min (X,Y,Z) ; Y' = med (X,Y,Z) ; Z' = max (X,Y,Z) .

Figure 5.6. A Systolic Priority Queue.

data dependency matrix cannot be easily derived; but research in optimizing compilers has shown that it can be done in an automatic manner for a large class of algorithms [30].

The basic idea of this chapter is to relate the mixed systolic array's architectural model with the parallel algorithms implemented on the array to facilitate their implementation. Data dependencies in an algorithm determine the communication structure required for its implementation. Linear time and space transformations of the algorithm are selected from the data dependency matrix of the algorithm and the interconnection structure of the mixed systolic array. For a given mixed systolic array structure, the optimal time transformation is first chosen in order to minimize the total execution time. If a valid space transformation does not exist to implement the optimal time schedule, a suboptimal time transformation is chosen for which a valid space transformation exists. Algorithms with constant data dependence vectors are easier to map on array structures and result in simple data-flow patterns. For some algorithms, such as the fast Fourier transform (FFT) and the bitonic sort, a remapping transformation is required in order to increase the efficiency of the algorithms [30]. Transformations for algorithms with tree communication geometry, such as searching algorithm, are not known.

The concepts presented in this chapter are not restricted only to mixed systolic arrays; they can also be used for formalizing the reconfiguration processes for some other polymorphic multiprocessor architectures. Several research issues emanate from this work and need further investigations. Procedures should be developed which allow the

user to obtain sets of valid transformations for other classes of algorithms with arbitrary index sets and nonconstant data dependencies. Second, this approach should be extended to silicon compilation of reconfigurable arrays to generate masks for an optimal or near-optimal array which can implement algorithms in a given set. Finally, further research is needed to define a unifying performance index for reconfigurable arrays to measure the overall array performance taking into consideration the speed, the reconfiguration process complexity, the cell's complexity and applicability of the array.

CHAPTER 6

CONCLUSIONS

6.1 SUMMARY

This research work investigated structured methodologies for designing and mapping parallel algorithms into a class of reconfigurable multiprocessor architectures called the mixed systolic array (MSA). The primary objective of this research was to investigate procedures for loading and implementing the configuration control structure in MSAs, and relate the architectural model of the MSA with parallel algorithms implemented on these architectures. The following specific tasks were addressed in this work:

1. Investigate the procedures for loading and implementing the distributed control structure required to establish a desired interconnection pattern on the MSA structure.
2. Relate the MSA's architectural model with the parallel algorithms implemented on the array and investigate the procedures for implementing algorithms into MSA structures by systematically generating the required control code for reconfiguration.

The work completed under each of the above tasks is summarized below.

In Chapter 3, we presented a serial loading procedure for implementing the distributed control structure required for reconfiguration in MSAs. This procedure employs LSSD techniques, and loads the control structure for a particular configuration into the array through the LSSD scan path, as a binary vector, in a bit-serial fashion. The main advantage of this aproach is that the overhead due to additional pinouts is limited to only three or four pins. Also, serial loading of the control structure reduces the overhead in terms of additional interconnections on the chip due to the control hardware. Chapter 3 presented the design of a linear reconfigurable systolic array architecture, which is statically reconfigurable to realize any of the following: a filtering array, an FIR filtering array, a pattern matching array, and a Discrete Fourier Transform (DFT) array. Shift register latches (SRLs) are used to hold the control information for setting up the interconnections configuration and selecting the functional mapping of computation cells. Both the data flow through the array and the functions executed in the computational cells are established by serially loaded configuration control vector.

In Chapter 4, we extended the approach presented in Chapter 3, for loading the MSA configuration control structure, to two-dimensional MSA structures. Design of an MSA based processor, called the multipurpose reconfigurable array processor was presented. This arhitecture implements both systolic and semisystolic algorithms, and incorporates fault tolerance in its design. The performance of MRAP, taking into account the total computation time and the data-transfer bandwidth, is analyzed for specific algorithms, and it is demonstrated, by way of

examples, how MRAP can efficiently implement different systolic and semisystolic algorithms. This two-dimensional array could also be configured as many independent linear arrays to implement independent algorithms involving one-dimensional linear recurrences such as finite impulse response (FIR) filtering.

The properties of reconfigurability and programmability at individual cell level, lead to fault tolerance capabilities in the MSA architectures. This was discussed in Section 4.4. A two-dimensional MSA architecture can be viewed as many reconfigurable pipelines working together. If a cell in the array fails, the array might be able to be reconfigured to bypass the faulty cell with data now flowing through the properly functioning cells. Performance of the array may be degraded, but the entire system does not fail, and the system achieves a graceful degradation property.

Efficient implementation of algorithms on VLSI structures requires exploitation of parallelism in the algorithm and mapping of the algorithm communication structure into the processor interconnection structure. Chapter 5 presented a step-by-step procedure for implementing a given algorithm into the MSA structure by generating the control code required to reconfigure the array. The mapping procedure is based on time and space transformations of the data dependence vectors of the algorithm. These transformations provide a description of the data-flow and timing, and dictate the interconnection structure required to implement the algorithm on the array. Starting from a high-level language description of an algorithm, this procedure generates the final configuration control vectors required to be loaded into the array through the LSSD scan path

in a bit-serial manner. The basic idea of Chapter 5 is to relate the mixed systolic array's architectural model with the parallel algorithms implemented on the array to facilitate their implementation. Data dependencies in an algorithm determine the communication structure required for its implementation. Linear time and space transformations of the algorithm are selected from the data dependency matrix of the algorithm and the interconnection structure of the mixed systolic array. For a given mixed systolic array structure, the optimal time transformation is first chosen in order to minimize the total execution time. If a valid space transformation does not exist to implement the optimal time schedule, a suboptimal time transformation is chosen for which a valid space transformation exists. Algorithms with constant data dependence vectors are easier to map on array structures and result in simple data-flow patterns. To illustrate the methodology and explain the reconfiguration procedure, two sample algorithms, the finite impulse response (FIR) filtering algorithm and the priority queue algorithm, are mapped into a linear reconfigurable systolic array.

The design methodologies investigated in this research broaden the scope of systolic architectures by achieving reconfigurability, algorithmic flexibility, partitionability and fault tolerance in MSAs. In addition, these methodologies preserve VLSI design attributes, such as locality of communication in order to avoid long and irregular interconnections, modularity in order to reduce the design time and cost, extensibility in order to enhance the computing power and simplicity of control. The LSSD technique is employed for loading and implementing the control structure required for reconfiguration of MSAs. This scheme

offers low overhead in terms of additional pinouts and extra hardware needed for implementing the MSA control structure. The procedure presented in this research, for mapping algorithms onto MSA architectures, presents a formalism for algorithm implementation and design of reconfigurable MSAs. This procedure can be used for designing MSA processors to implement a pre-defined set of algorithms, and it can also be used for implementing a new algorithm on a given MSA processor by generating the control vector required for its reconfiguration.

6.2 FUTURE RESEARCH

The concepts presented in this thesis are not restricted only to mixed systolic arrays; they can also be used for formalizing the reconfiguration processes for some other polymorphic multiprocessor architectures. Several research issues emanate from this work and need further investigations. First, procedures should be developed which allow the user to obtain sets of valid transformations for other classes of algorithms with arbitrary index sets and nonconstant data dependencies. Second, the approach, presented in Chapter 5, should be extended to silicon compilation of reconfigurable arrays to generate masks for an optimal or near-optimal array which can implement algorithms in a given set. Third, further research is needed to define a unifying performance index for reconfigurable arrays to measure the overall array performance taking into consideration the speed, the reconfiguration process complexity, the cell's complexity and applicability of the array. Finally, research should be done to develop CAD tools for MSAs at the

architectural description level, and integrate these tools with other CAD systems supporting circuit diagram and physical layout levels of design, to create an integrated system design environment where masks can be generated for optimal or near-optimal MSA processors.

APPENDIX

APPENDIX A

A COMPUTER-AIDED DESIGN FACILITY FOR MIXED SYSTOLIC ARRAYS

The rapidly growing complexity of digital systems and the ever-increasing scale of integration on silicon chips provide a challenge to developers of computer-aided design (CAD) systems, to create an integrated system design environment with high-level CAD tools. More complex systems require a greater need for CAD tools to do the design task in a cost effective way. In a top-down hierarchical design approach, it is common practice to structure the design process of a computer system into the following steps [17]:

1. Design and description of architecture,
2. General circuit diagram,
3. Design of boards and integrated circuits,
4. Technology process and production.

Although commercially available computer-aided engineering workstations provide a sound environment with integrated tools for the logical and physical design levels, they are not suited to support the high-level design of hardware and functional system architectures. High-level CAD tools are, therefore, needed to support the system designer at higher levels in the design hierarchy, such as design of functional system architecture from system behavior specifications.

The main motivation behind this work is to provide the designer of MSAs with a facility to interactively develop an MSA structure for a given set of user-specified attributes, such as mixing density and array geometry, and to simulate the execution of algorithms on MSA processors at the register-transfer level. This appendix describes a computer-aided design facility for modeling and simulating mixed systolic arrays. The CAD facility serves as a high-level design tool which supports the design of MSA architectures. This facility develops the structural model of an MSA by interactively interrogating the user about various attributes of the MSA structure, such as mixing density and array geometry. In accordance with the user-specified attributes, the system generates a model for the MSA structure and displays it on the user's graphics terminal showing the relative positions of all functional cells in the array. Defining of the desired communication structure is then done interactively. The system also performs register-transfer level MSA simulation of the algorithms, observes the data-flow patterns in the array and helps evaluate the throughput of the system.

The CAD facility, described here, is a useful tool for designing and programming MSA architectures. It provides a design environment that facilitates mapping of parallel algorithms into the reconfigurable MSA architecture and automatic generation of reconfiguration control code required for algorithm implementation. The register-transfer level simulation is used to validate the MSA architecture and the reconfiguration control structure during the design phase. This work is a step forward in the direction of automated design of reconfigurable multiprocessor architectures involving multiple pipelines. The next

section presents a system description of the CAD facility for modeling and simulating MSA architectures.

A.1 DESCRIPTION OF THE SYSTEM

In this section, we describe a CAD facility that supports the design of MSA architectures through interactive modeling and simulation. This facility serves as a high-level CAD tool, which develops a structural model of the MSA from user-defined architectural attributes of the array, such as mixing density and array geometry. In addition, this facility performs register-transfer level MSA simulation of algorithms, observes the data-flow patterns and helps evaluate the throughput of the system.

This system provides the following facilities to the designer of the MSA architecture.

1. Generation and graphic display of the structural model of a regular MSA when the array attributes, such as mixing density and array geometry, are user-specified.

2. Storage of interconnection specifications, in form of a list of records, which are interactively specified by the user on the graphics terminal.

3. Options to interactively modify or delete selected array cells.

4. Reconfiguration of the MSA by entering reconfiguration control

information.

5. Register-transfer level simulation of algorithms.

Figure A.1 shows a diagrammatic representation of the basic structure of this facility. As can be seen, the system consists of four sets of data files and seven programs. A brief description of each system block is given below.

STRUCTURE GENERATOR.

This program generates the structural model of the MSA for a specified mixing density and array size, and maps the index points of the array's index set into the set of available functional cells. The output of the structure generator program is used by the plotting program and the simulator.

STRUCTURE EDITOR.

This program is used to edit the MSA structure displayed at the user's graphics terminal. Two routines, namely PRUNE and MODIFY, are used for this purpose. PRUNE eliminates any extra boundary cells not required in the MSA implementation. User can move the cursor and select the cell to be eliminated. MODIFY selects a cell position within the array and changes the cell type at that location. This program is robust in the sense that it can sense an improper selection made by the user, i.e., a location where no cell is present, and prompt the user about the mistake. This program provides a facility for designing irregular architectures, as well as for including special I/O processors at the

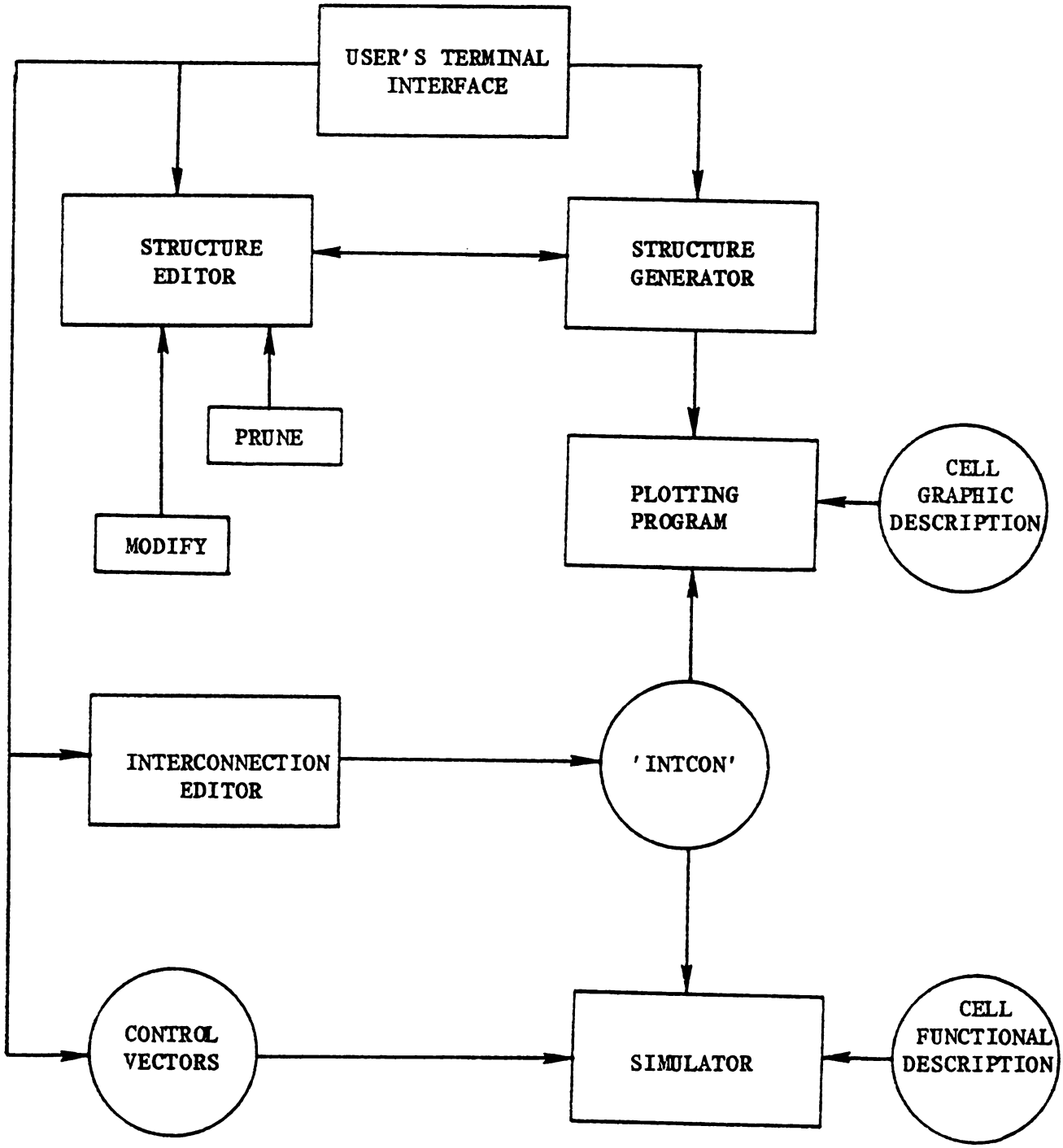


Figure A.1. The system diagram

array boundary.

PLOTTING PROGRAM.

This program plots the representation of the MSA structure as defined by the user-specified attributes and developed by the structure generator program. The plot is produced on a graphics terminal, where the user can interactively edit the MSA structure or specify the interconnections. A hard copy of the plot can be obtained at any stage in the design.

INTCON.

This set of files contains list of records of interconnections specified for specific MSA structures. The simulator uses these specifications to determine the data-flow for the array. One record specifies one interconnection link and contains information about the input and output cell coordinates and their respective local ports. A detailed discussion of the storage schemes for interconnection links is given later in this section.

INTERCONNECTION EDITOR.

This program modifies and specifies the interconnections among the array cells via a graphics terminal. This program decodes the screen coordinates of the graphics terminal into the array's index set points and generates a record for each specified interconnection to be stored in the INTCON. This program can be extended to check the legality of an interconnection link specified by the user.

CELLS' GRAPHIC DESCRIPTION.

This set of files contains the graphic representation of all the cells of which a specific MSA is composed. Each file contains sequence of graphics commands which define the shape and size of a cell. Different symbols are used to represent different types of cells.

CELLS' FUNCTIONAL DESCRIPTION.

This set of files contains the functional description of computation cells and switching cells. A file pertaining to a specific type of computation cell contains the data-transfer statements for processing incoming data at the local ports of the cells. In case of a switching cell, a file contains information regarding the one-to-one correspondence of the set of control codes for switching cells with various interconnection configurations. The simulator program uses the functional description while performing the execution of an algorithm.

RECONFIGURATION CONTROL VECTORS.

This set of files contains various control vectors, each capable of reconfiguring an MSA structure to implement a specific algorithm. The configurations of a specific MSA, and the algorithms implemented on it, are determined by its reconfiguration control vectors. User can add new control vectors in this set of files, which when loaded into the array, establish the computation and communication structure of the array.

SIMULATOR.

This program evaluates the execution behavior of the MSA architecture by determining the state of the machine after each time step when simulating execution of algorithms. Once the MSA architecture and the communication structure are defined by using the structure generator and the interconnection editor, a particular MSA configuration can be achieved by loading the corresponding control vector. The simulator merely calls two routines to simulate the processor operations and interprocessor communication at each time step. One routine is called CMPCEL, which simulates the input to output functional mapping for all ports of each computation cell as defined by the control code bits in each cell. The second routine is called SWTCEL, which transfers the data available at the output ports of the computation cells to the input ports of appropriate cells as determined by the control code stored in the switching cells. So, at each time step the two routines, CMPCEL and SWTCEL, are called and executed to simulate the MSA behavior.

In the remainder of this section, we describe how the MSA Structure Generator generates and displays the MSA structures with regular hexagonal geometry when the basis mixing density is specified. Although there exists a large number of mixing profiles which yield a regular array structure, only a few of them are practical for real algorithm applications. The hexagonal array geometry, first used for band-matrix multiplication algorithm [31], is an important structure because of its utilization of planar communication and its high packing density. The smallest hexagonal array comprises seven cells and serves as a basis for hexagonal MSAs, as larger array structures can be grown from this basis.

The mixing density of the basis, ρ_b, is called the basis mixing density and can be one of the following:

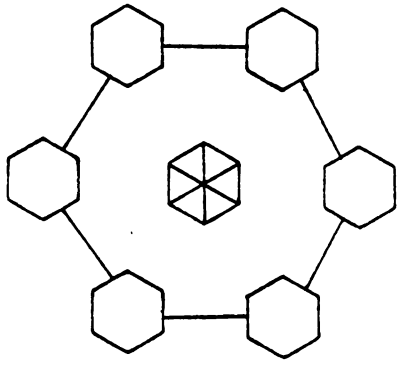
$$\rho_b = i/7 \ , \ i = 1, 2, 3, \ . \ . \ , 7.$$

Figure A.2 shows the bases corresponding to three different mixing profiles. MSAs grown from the bases are intended for computation use. The density of a regular hexagonal MSA is a function of the basis mixing density, and these relations are given in [6]. The structure generator program determines the index set Q^n of the MSA and the transformation matrix D, which is a mapping on the index set to the set {0, 1}, where a 0 indicates a SC and a 1 indicates a CC.

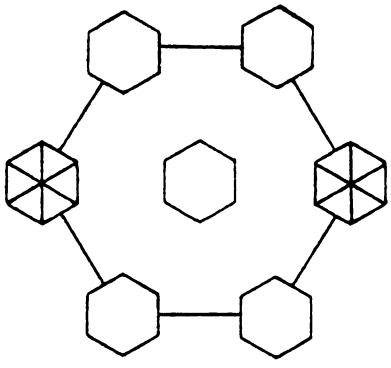
In order to get a hexagonal geometric pattern on the display a screen coordinate transformation is performed which, of course, is transparent to the user. Each cell (CC or SC) in the array is labelled by an ordered pair (x,y) where x and y are the values of the X and Y coordinates of that cell position. In order to convert the rectangular coordinate system, where X-axis and Y-axis are at an angle of 90 degrees, into a hexagonal coordinate system, where both axis are at an angle of 120 degrees, the following transformation matrix is calculated:

$$\begin{bmatrix} 1.732 & -1.732 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix}$$

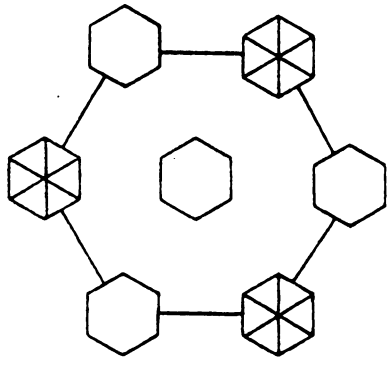
X_{new} and Y_{new} are the new screen coordinates for hexagonally patterned display.



(a)



(b)



(c)

Figure A.2. Hexagonal array bases with (a) $\rho_b = 1/7$, (b) $\rho_b = 3/7$, and (c) $\rho_b = 3/7$.

Interprocessor communication structure is defined by specifying all interconnections among desired MSA cells. The problem of specifying these interconnections is synonymous to the problem of defining and representing a directed graph (or a digraph). One approach which could be considered here, is to represent the digraph by an adjacency matrix. For a graph with p points there is defined an adjacency matrix $A=[a_{i,j}]$ in which $a_{i,j}=1$ if there is an arc from vertex v_i to vertex v_j and $a_{i,j}=0$ otherwise. This approach, although quite comprehensive, is not chosen in our system mainly because of the following reasons. In certain implementations, all of the available cells may not be used, and hence the indegrees and outdegrees of some vertices in the digraph will be zero. This results in rows and columns containing all 0's, which will slow down the simulation because the whole adjacency matrix needs to be exhaustively scanned after each clock cycle to simulate the communication structure. Another reason is that MSA structures employ short and regular communication geometry, so a vertex in its digraph is only connected to a few of its neighbors, whereas adjacency matrix takes into account all possible arcs from any vertex to every other vertex in the vertex set. Hence the adjacency matrix approach does not match well with the class of digraphs we are considering here. Other approaches of representing digraphs such as Reachability matrix and Distance matrix are also not suitable for MSA communication graph representation and simulator implementation.

The scheme we have employed here to represent the digraph and hence the communication structure of the array, is to store in a data file a list of all the arcs (or edges) in the graph. Each arc is specified in

the list as a record which contains the originating cell's coordinates and its output port as well as the terminating cell's coordinates and its input port. For example in case of a hexagonal array, the arc shown in Figure A.3 will be stored as the following record

(3 m n 6 k 1),

where first and fourth entries refer to the input and output ports respectively such that AIN=1, BIN=2, CIN=3, AOT=4, BOT=5 and COT=6. The remaining four entries in the record refer to the coordinates of the incident vertices of the arc.

Defining of the desired communication structure is done interactively using a CAD graphics terminal. Two possible modes are available to the user at this stage. In the 'Cursor' mode, a cell can be selected by moving the cursor anywhere inside the cell and pressing any key. In this fashion user can enter the desired interconnections on the terminals, which are converted into the record format and saved in the data file containing the list of arcs. In the 'Text' mode, user types in the coordinates of the incident cells and specifies the input and output ports of the cells. It can be possible in future, to make provision in the program for checking the legality of the specified connections and prompt the user in case of an illegal specification.

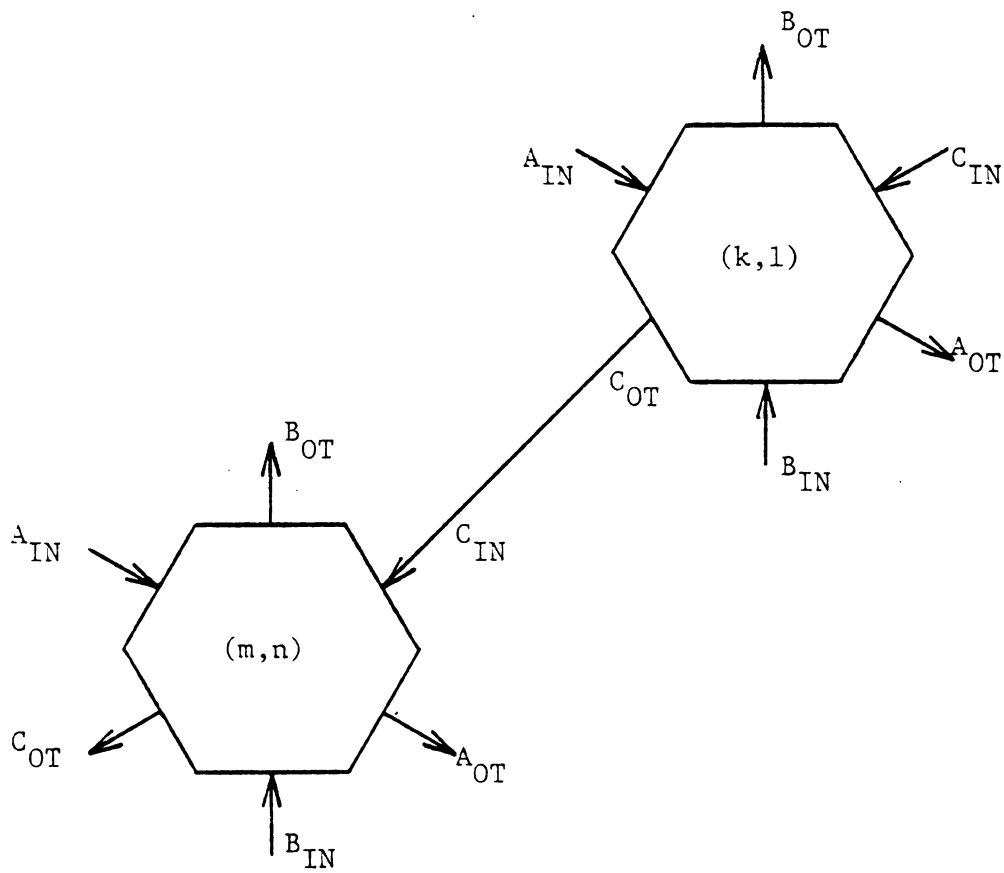


Figure A.3. An arc indicating a connection between two neighboring cells.

A.2 DISCUSSION

We described a CAD tool, which supports the design of MSA processors at the architectural design and description level. The register-transfer level simulator is used to validate the MSA architecture and the reconfiguration control structure during the design phase. The system was implemented on a PRIME-750 computer, for hexagonal array geometry, and the simulation was performed, for the band-matrix multiplication algorithm, as given in [31], on a hexagonal MSA with a basis mixing density of 1/7. A more general implementation of the system can include other array geometries, such as rectangular or triangular, in the array structure generator program. In order to implement the band-matrix multiplication algorithm, the switching cells and selected computation cells should be in straight and bypass configurations, respectively. That is, for these cells,

$$A_{OT} = A_{IN};$$

$$B_{OT} = B_{IN};$$

$$C_{OT} = C_{IN}.$$

The active computation cells in the MSA, perform the inner product step function, given by

$$A_{OT} = A_{IN};$$

$$B_{OT} = B_{IN};$$

$$C_{OT} = A_{IN} * B_{IN} + C_{IN}.$$

The data communication with the outside world takes place at the MSA boundary only. The input and output is controlled by the simulator program. Since the timing for data input and output depends heavily on

the algorithm to be simulated, a separate I/O routine is required for each algorithm. In case of the band-matrix multiplication algorithm, considered here, each cell performs active computation at every third time step. The simulator output gives the timing for the data input and output at the MSA boundary cells. The simulation results provide the status of each cell after each time step. This feature helps observe the data-flow and find throughput of the MSA.

The work, presented here, is a step forward in the direction of automated design of reconfigurable multiprocessor architectures involving multiple pipelines. For further research, this high-level CAD tool can be integrated with other CAD systems supporting circuit diagram and physical layout levels of design, to create an integrated system design environment for generating masks for MSA processors.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Aho A.V., Hopcroft J.E. and Ullman J.D., Data Structures and Algorithms, Addison-Wesley, Reading, Massachusetts, 1983.

2. Brent R.P. and Kung H.T., "Systolic VLSI Arrays for Linear-Time GCD Computation," VLSI 83, F. Anceau and E.J. Aas, eds., North Holland, New York, August 1983, pp. 145-154.

3. Bromley K., Symanski J.J., Speiser J.M. and Whitehouse H.J., "Systolic Array Processor Developments," VLSI Systems and Computations, H.T. Kung et al. eds., Computer Science Press, October 1981, pp. 273-284.

4. Cappello P.R. and Steiglitz K., "Digital Signal Processing Applications of Systolic Algorithms," in VLSI Systems and Computations, H. T. Kung et al. eds, Computer Science Press, October 1981, pp. 245-254.

5. Cappello P.R. and Steiglitz K., "Unifying VLSI Array Designs by Geometric Transformations," Proc. 1983 Int'l Conf. on Parallel Processing, August 1983, pp. 448-457.

6. Chang T.L. and Fisher P.D., Mixed Sytolic Arrays: A Reconfigurable Multiprocessor Architecture, Division of Engineering Research Technical Report No. MSU ENGR-82-002, Department of Electrical Engineering and Systems Science, Michigan State University, East Lansing, Michigan 48824, 1982.

7. Chang T.L. and Fisher P.D., "Programmable Systolic Arrays," Digest of Papers, IEEE COMPCON, Spring 1982, pp. 48-53.

8. Chang T.L. and Fisher P.D., "Master-Slave Mixed Arrays for Data-Flow Computations," Digest of Papers, IEEE COMPCON, Spring 1983, pp 468-472.

9. Chern M.Y. and Murata T., "Efficient Matrix Multiplications on a Concurrent Data-Loading Array Processor," Proc. 1983 Int'l. Conf. on Parallel Processing, August 1983, pp 90-94.

10. Chern M.Y. and Murata T., "A Fast Algorithm for Concurrent LU Decomposition and Matrix Inversion," Proc. 1983 Int'l. Conf. on Parallel Processing, August 1983, pp 79-86.

11. Cohen D., "Mathematical Approach to Iterative Computational Networks," Proc. Fourth Symp. Computer Arithmetic, October 1978, pp. 226-238.

12. Eichelberger E.B. and Williams T.W., "A Logic Design Structure for LSI Testability," Proc. 14th Design Automation Conf., New Orleans, June 1977, pp. 462-468.

13. Eichelberger E.B., "Latch Design Using Level Sensitive Scan Design," Digest of Papers, COMPCON 1983, February 28 - March 3, pp. 380-383.

14. Fisher A.L., Kung H.T., Monier L.M. and Dohi Y., "Architecture of the PSC: A Programmable Systolic Chip," Proc. 10th Annual Symposium on Computer Architecture, Vol. 11, No. 3, June 1983, pp. 48-53.

15. Fortes J.A.B., Algorithm Transformation for Parallel Processing and VLSI Architecture Design, Ph.D. Thesis, Dept. of Electrical Engineering Systems, University of Southern California, Los Angeles, December 1983.

16. Fortes J.A.B., Fu K.S. and Wah B.W., "Systematic Approach to the Design of Algorithmically Specified Systolic Arrays," Int'l. Conf. on Acoustics, Speech and Signal Processing, March 1985, pp. 300-303.

17. Gonauser M. and Sauer A.M., "Needs for High-Level Design Tools," Proc. 1983 Int'l. Conf. on Computer Design: VLSI in Computers, October 1983, pp. 415-418.

18. Guibas L.J., Kung H.T. and Thompson C.D., "Direct VLSI Implementation of Combinatorial Algorithms," Proc. Conf. on VLSI Architecture, Design, and Fabrication, Calif. Inst. of Tech., January 1979, pp. 509-525.

19. Heller D.E. and Ipsen I.C.F., "Systolic Networks for Orthogonal Equivalence Transformations and Their Applications," Proc. Conf. on Advanced Research in VLSI, M.I.T., January 1982, pp. 113-122.

20. Hopkins A.L., Jr. "Fault-Tolerant System Design: Broad Brush and Fine Print," Computer, March 1980, pp. 39-45.

21. Huang K.H. and Abraham J.A., "Efficient Parallel Algorithms for Processor Arrays," Proc. 1982 Int'l. Conf. on Parallel Processing, pp. 271-279.

22. Johnson L. and Cohen D., "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks," VLSI Systems and Computations, H.T. Kung et al. eds., Computer Science Press, October 1981, pp. 213-225.

23. Johnson L., Weiser V. Cohen D. and Davis A.L., "Towards a Formal Treatment of VLSI Arrays," Proc. Caltech Conf. on VLSI, January

1981.

24. Kessler A.J. and Patel J.H., "Reconfigurable Parallel Pipelines for Fault Tolerance," IEEE Int'l. Conf. on Circuits and Computers, September 1982, pp. 118-121.

25. Khurshid A. and Fisher P.D., "Design of a Reconfigurable Systolic Array Using LSSD Techniques," Proc. 1984 Int'l. Conf. on Computer Design: VLSI in Computers, October 1984, pp. 171-175.

26. Khurshid A. and Fisher P.D., "Algorithm Implementation on Reconfigurable Mixed Systolic Arrays," Proc. 1985 Int'l. Conf. on Parallel Processing, August 1985.

27. Khurshid A. and Fisher P.D., "A Computer-Aided Design Facility for Mixed Systolic Arrays," Sixteenth Annual Pittsburgh Modeling and Simulation Conference, April 25-26, 1985.

28. Khurshid A. and Fisher P.D., "A Multipurpose Reconfigurable Array Processor for Systolic and Semisystolic Algorithms," Submitted for publication.

29. Kuekes P. and Shen J., "One-Gigaflop VLSI Systolic Processor," Proc. SPIE Symp., Vol. 431, Real-Time Signal Processing VI, August 1981, pp. 10-18.

30. Kuhn R.H., Optimization and Interconnection Complexity for: Parallel Processors, Single Stage Networks, and Decision Trees, Ph.D. Thesis, Dept. of Computer Science, Rpt. 80-1009, University of Illinois, Urbana-Champaign, Illinois, 1980.

31. Kung H.T., "Let's Design Algorithms for VLSI Systems," Proc. Caltech Conf. on Very Large Scale Integration, January 1979, pp. 55-90.

32. Kung H.T., "Why Systolic Architectures," Computer, Vol. 15, No. 1, January 1982, pp. 37-46.

33. Kung H.T., "Notes on VLSI Computation," CREST Parallel Processing Systems Course, Loughborough, England, 1980.

34. Kung H.T., "On the Implementation and Use of Systolic Array Processors," Proc. 1983 Int'l. Conf. on Computer Design: VLSI in Computers, pp. 370-373.

35. Kung H.T., "Special-purpose Devices for Signal and Image Processing: An Opportunity in Very Large Scale Integration (VLSI)," Proc. SPIE Vol. 241 Real-Time Signal Processing III, July 1980, pp. 76-84.

36. Kung H.T. and Lam M.S., "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays," Journal of Parallel and Distributed Computing 1, 1984, pp. 32-63.

37. Kung H.T. and Leiserson C.E., "Systolic Arrays (for VLSI)," Sparse Matrix Proceedings 1978, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

38. Kung H.T., Ruance L.M. and Yen D.W.L., "A Two-Level Pipelined Systolic Array for Convolutions," in VLSI Systems and Computations, H. T. Kung et al. eds., Computer Science Press, October 1981, pp. 255-264.

39. Kung S.Y., Arun K.S., Gal-Ezer R.J. and Rao D.V.B., "Wavefront Array Processor: Language, Architecture and Applications," IEEE Trans. on Computers, Vol. C-31, No. 11, November 1982, pp. 1054-1066.

40. Lam M.S., and Mostow J., "A Transformational Model for VLSI Systolic Design," Computer, Vol. 18, No. 2, February 1985, pp. 42-52.

41. Leiserson C.E., Area Efficient VLSI Computation, Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, October 1981. Published in book form as part of the ACM Doctoral Dissertation Award Series by the MIT Press, Cambridge, Massachusetts, 1983.

42. Leiserson C.E., "Systolic and Semisystolic Design," Proc. 1983 Intl. Conf. on Computer Design: VLSI in Computers, pp. 627-632.

43. Leiserson C.E. and Saxe J.B., "Optimizing Synchronous Systems," Journal of VLSI and Computer Systems, Vol. 1, No. 1, 1983, pp. 41-68.

44. Mead C. and Conway L., Introduction to VLSI Systems, Addison Wesley, 1980.

45. Moldovan D.I., "On the Design of Algorithms for VLSI Systolic Arrays," Proceedings of the IEEE, Vol. 71, No. 1, January 1983, pp. 113-120.

46. Moldovan D.I., "ADVIS: A Software Package for the Design of Systolic Arrays," 1984 Int'l. Conf. on Computer Design: VLSI in Computers, October 1984, pp. 158-164.

47. Moldovan D.I., Wu C.I. and Fortes J.A.B., "Mapping an Arbitrarily Large QR Algorithm into a Fixed Size VLSI Array," Proc. 1984 Int'l. Conf. on Parallel Processing, pp. 365-373.

48. Oppenheim A.V. and Schafer R.W., Digital Signal Processing, Prentice-Hall, 1975.

49. Quinton P., "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," Proc. 11th Int'l. Symposium on Computer Architectures, 1984, pp. 208-214.

50. Serlin O., "Fault-Tolerant Systems in Commercial Applications," IEEE Computer, Vol. 17, No. 8, August 1984, pp. 19-30.

51. Siewiorek D.P., "Architecture of Fault-Tolerant Computers," IEEE Computer, Vol. 17, No. 8, August 1984, pp. 9-18.

52. Smith S.D. and Siegel H.J., "Recirculating, Pipelined, and Multistage SIMD Interconnection Networks," 1978 Int'l. Conf. on Parallel Processing, August 1978, pp. 206-214.

53. Snyder L., "Introduction to the Configurable, Highly Parallel Computer," Computer, Vol. 15, No. 1, January 1982, pp. 47-56.

54. Snyder L., "Supercomputers and VLSI: The Effect of Large-Scale Integration on Computer Architecture," Advances in Computers, Vol. 23, Academic Press, 1984, pp. 1-33.

55. TRW LSI Products, "LSI Multiplier-Accumulators," TRW Inc., La Jolla, California, 1983.

56. Weiser U. and Davis A., "A Wavefront Notation Tool for VLSI Array Design," VLSI Systems and Computations, H.T. Kung et al. eds., Computer Science Press, October 1981, pp. 226-234.

57. Williams T.W. and Parker K.P., "Design for Testability - A Survey," Proc. IEEE, Vol. 71, No. 1, January 1983, pp. 98-112.

58. Yen D.W.L. and Kulkarni A.V., "Systolic Processing and an Implementation for Signal and Image Processing," IEEE Transactions on Computers, Vol. C-31, No. 10, October 1982, pp. 1000-1009.