MIXED SYSTOLIC ARRAYS : A RECONFIGURABLE MULTIPROCESSOR ARCHITECTURE

BY

Tung-Liang Chang

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

1982

ABSTRACT

MIXED SYSTOLIC ARRAYS:
A RECONFIGURABLE MULTIPROCESSOR ARCHITECTURE

By

Tung-Liang Chang

Systolic arrays are special-purpose, high-performance data-flow multiprocessor structures which are characterized by having a simple, regular and short communication geometry, which is considered as one of the most desired attributes in VLSI implementation. The principal drawback to these special-purpose arrays is that they are designed for specific algorithms which possess very simple and regular data flow patterns. These restrictions limit the types of algorithms or applications that can effectively be supported by such architectures.

The primary objective of this research is to develop, characterize, and evaluate a basic computing model for a class of architectures, which broadens the scope of algorithms executable on systolic arrays while retaining much of the simplicity and regularity of the original systolic array architecture. Computing elements and control buffers

are mixed in regular geometric patterns to form reconfigurable systolic arrays known as mixed systolic arrays (MSA's). For a particular implementation, the mixing ratio is chosen to match an algorithm's local vs. global data requirements. Classes of algorithms with similar data requirements may be executed on the same array by merely presetting the control buffers at load time. By decomposing an MSA into two basic regions, the dependence on computing power vs. I/O bandwidth as a function of array edge size can be relaxed.

Data-driven computations result in self-directed computational rings within the MSA, and composite cells of data-flow instructions are executed in a pipelined fashion. Vertical grouping of composite cells along critical-path data-flow instructions and horizontal grouping across concurrent data-flow instructions are employed for mapping data-flow directed graphs on computational rings for effective execution. By viewing an MSA as a computing network of interlinked ring pipelines, data-flow programs can be uniformly distributed for efficient resource utilization.

Data flow on the mixed systolic array is demonstrated by solving a second-order recursive equation with non-constant coefficients and by the implementation of a modified hourglass computing model. MSA space-time complexity is compared with that of the systolic array along with other key performance measures.

ACKNOWLEDGEMENTS

The author wishes to express his deepest appreciation to Dr. P.D. Fisher, the author's thesis advisor, for his assistance during the course of this research and for supervising this thesis and showing considerable patience. Thanks are also due to the other members of the graduate committee, Dr. J.B. Kreer, Dr. S.R. Crouch, Dr. D.K. Reinhard, Dr. R.G. Reynolds and Dr. M.A. Shanblatt, for their suggestions and encouragement throughout this research effort. I wish to thank Ronald Kraus for preparing all of the illustrations contained in this thesis.

Very special thanks are due V.E. Leichty for his encouragement during my stay with him in the past four years.

Work reported here was supported in part by NSF under Grant No. MCS 79-09216.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The demand for improved or advanced computer systems continues unabated with special emphasis on increased overall throughput. One approach being considered exploits very-large-scale-integrated (VLSI) circuit technology to scale down IC devices to submicron features, thereby reducing logic gate speed-power products to 0.1 pJ and below [1]. Based on this figure, single chips may contain 10^7 to 10^8 transistors, which is equivalent to about 10^3 unconnected 32-bit microprocessors [2,3]. However, with the interprocessor communication path complexity growing at a much more rapid rate than that of logic circuitry, interconnections may account for most of the chip area. What's more, these interconnection networks may account for the critical time delays [2]. Therefore, the full reward for using VLSI circuits lies with the ability of the computer architect to design practical machines with simple and regular communication paths. Moreover, they must be designed to exploit fully concurrencies that exist in the user's applications.

Array-based architectures appear to be very attractive in this respect. It is argued that such architectures have several important advantages:

* Arrays require shorter communication paths--a shorter communication wire not only improves system operations, but also consumes less chip area [2].
* Arrays have regular communication wiring patterns--a regular layout leads to efficient, high density designs and simplified debugging procedures [4].
* Arrays execute structured algorithms efficiently--through parameterization procedures, algorithms with particular computational demands, control, and I/O structures can be effectively executed on these architectures [5].
* Arrays are built from a small set of predesigned modules --modularity reduces design time because design automation techniques become useful. Modularity also reduces cost since general-purpose modules may be used in multiple applications once they are available to the system designer [4].
* Arrays support very high concurrencies in local computations and communications--locality of computation and communication leads to ease of algorithm decomposition and to an effectiveness of distributed computations [6].
* Arrays are easy to expand--expandability allows the system designer to enhance the computing power or applicability of the architectures with a minimum effort [7].

The purpose of this thesis is to investigate alternative strategies which fully exploit emerging IC technology to yield useful, high-performance, cost-effective array-based computer architectures.

One important class of VLSI architectures which has received a great deal of attention is the systolic array [8,9]. These arrays are characterized by having a simple, regular and short communication geometry which is considered as one of the most desired attributes in VLSI implementation. The principal drawback to these special-purpose arrays is that they are designed for specific algorithms which possess very simple and regular data flow patterns. These restrictions limit the types of algorithms or applications that can efficiently be supported by such architectures. Thus the investigation of a new class of array architectures, which can broaden the scope of algorithms executable on them while retaining much of the simplicity and regularity of the systolic array architectures, can be of practical use.

The primary objective is to develop, characterize, and evaluate a basic computing structure for a class of array architectures--a modified systolic array--which broadens the scope of algorithms executable on systolic arrays while retaining much of the simplicity and regularity of the original systolic array structure. One aspect of the approach taken here is to introduce control buffers (CB's) into the array [5]. In contrast with the systolic array

structure in which data are regularly moved on a limited pattern of communication paths, these control buffers provide a means to increase the flexibility of data movements. More specifically, these CB's are programmable, thereby increasing the array's potential usefulness. As shown in Fig. 1.1, the modified array is partitioned into interconnected control buffers and computing elements. The control buffers (CB's) control the sequence of computations in the array and the computing elements (CE's) implement the primitive mathematical operations. Within the modified array, there are numerous mini-systolic-like subarrays, each of which executes data-flow subgraphs scheduled to it. The control buffers, when initiated by a control signal, tag data in such a manner as to control their direction of movement and the computations performed on the computing elements. As proposed, the array-based architectures constructed from the modified systolic array will support asynchronous as well as synchronous control structures, local as well as global communication structures, distributed functional computation structures, and balanced I/O structures.

In order to help define and establish the significance of this particular class of array architectures, some background information is presented in Chapter II. As the groundwork for the development of this class of array architectures, the design of a triangular programmable systolic array for solving a second-order recursive problem is

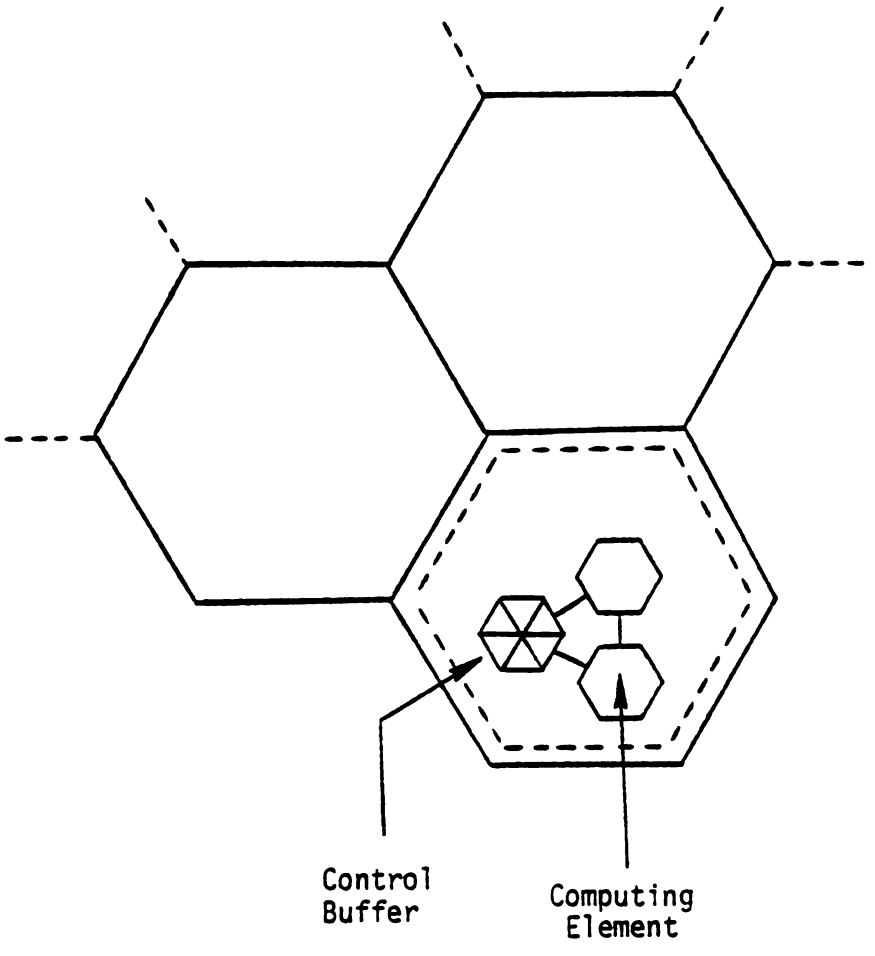


Figure 1.1. A modified systolic array made up of both control buffers and computing elements.

presented in Chapter III. This is followed in Chapter IV by the discussion of mixing in the systolic array and the functional and structural characteristics of the mixed array. Chapter V describes block-driven computations on the mixed systolic array, including the discussion of basic ring computing structures and their subsequent chain operations for the support of effective block-driven computations. Next, in Chapter VI, the design of an hourglass computing machine is presented, along with an analysis based on some performance measures such as throughput, cost-effectiveness, hardware complexity, and VLSI implementation, among other things. And, finally, conclusions and thoughts which might lead to future research possibilities are included in the last chapter, Chapter VII.

CHAPTER II

BACKGROUND

2.1 Limitations of Single-Processor Machines

The single-processor sequential machine performs one elemental activity at a time, much like a single individual would solve an arithmetic problem by hand. But this computational model has serious limitations. To understand these limitations let us consider the structure of such a computer--the von Neumann machine (see Fig. 2.1). Here program instructions and data both reside in the random-access main memory. The program's task of producing output given some input is accomplished entirely by pumping single words across the data channel which connects the processor with its memory. A typical instruction execution cycle proceeds as follows: an instruction is fetched by the controller from main memory. Next, the controller decodes the instruction. Then necessary operands, or addresses of operands, are fetched from main memory. Once all of these operands are available, the instruction is executed. The instruction cycle ends with results being left in CPU registers or deposited in main memory. The communication channel connecting the CPU and main memory is generally "saturated",

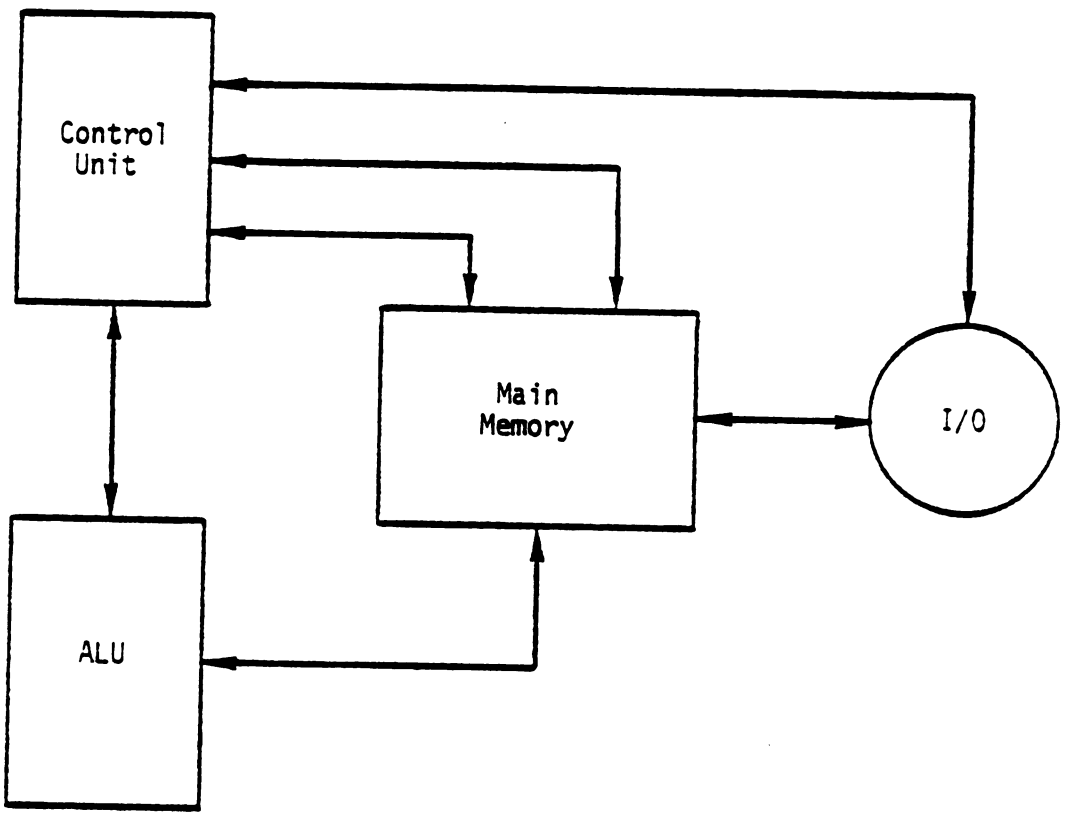


Figure 2.1. The structure of the von Neumann Machine.

leaving the arithmetic logic unit (ALU) idle a significant percentage of the time. Moreover, a large part of the information moving on this channel at any instant is not data at all, rather it is addresses for program statements and data. Backus refers to this channel as the "von Neumann bottleneck" since it limits the machine's throughput [10]. The only way to retain this sequential architecture and increase throughput is by reducing main memory read/write access times.

But the ramifications of this bottleneck do not become completely clear until it is recognized that in a typical scientific application, where one would expect a large percentage of the machine operations to be arithmetic, less than 20% of all machine instructions executed are actually fixed-point or floating-point arithmetic operations [11]. Data access operations and control unit executed branch operations account for approximately one half and one third of the total instructions executed, respectively (see Table 2.1).

Perhaps the greatest shortcoming of the single-processor sequential machine is that the structure of the problem being executed rarely corresponds to the structure of the machine. Specifically, simple operations such as vector addition or matrix multiplication have high degrees of concurrency built into the very structure of the mathematical operation, yet the single-processor sequential

TABLE 2.1

TWO STANDARD VON NEUMANN MACHINE DYNAMIC INSTRUCTION MIXES FOR SCIENTIFIC AND TECHNICAL APPLICATIONS [11]

Instruction class	Gibson mix, %	Flynn mix, %
Load/store	31.2	45.1
Index	18	
Branch	16.6	27.5
Compare	3.8	10.8
Fixed point	6.9	7.6
Floating point	12.2	3.2
Shift/logical	6.0	4.5
Other	5.3	1.3
	100.0	100.0

machine does not take advantage of them to enhance throughput.

2.2 Array Architectures

Array architectures, such as Illiac IV [12] and various distributed arrays [13-16], perform a single operation on a stream of operands available to each of the arrays' elements. As designed, these SIMD (Single Instruction Stream Multiple Data Stream) arrays are very effective for solving algorithms which either have very large data structures, as in the vector and matrix problems, or a high degree of locality, as in the image processing applications [15-18]. Usually these arrays have fixed networks. At each node in the arrays is a von Neumann-style machine known as a processing element. Each processing element is associated with its own memory, and paths are provided for the input of data to the array for processing and the output of results after processing.

Depending on their structure, each N(i,j) in the array can communicate with its neighboring nodes N(i $\pm$ n, j $\pm$ m), where n and m are positive integers. As with the Illiac IV machine, each of the 64 nodes, N(i,j), i,j = 0,1,...,7 is connected to its four nearest nodes N((i $\pm$ 1) mod 8), (j $\pm$ 1) mod 8) (see Fig. 2.2). Data transfers within this array are taken on a uniform shifting basis under a centralized control. Therefore, several continued operations of shifting are required for transferring a data item from one

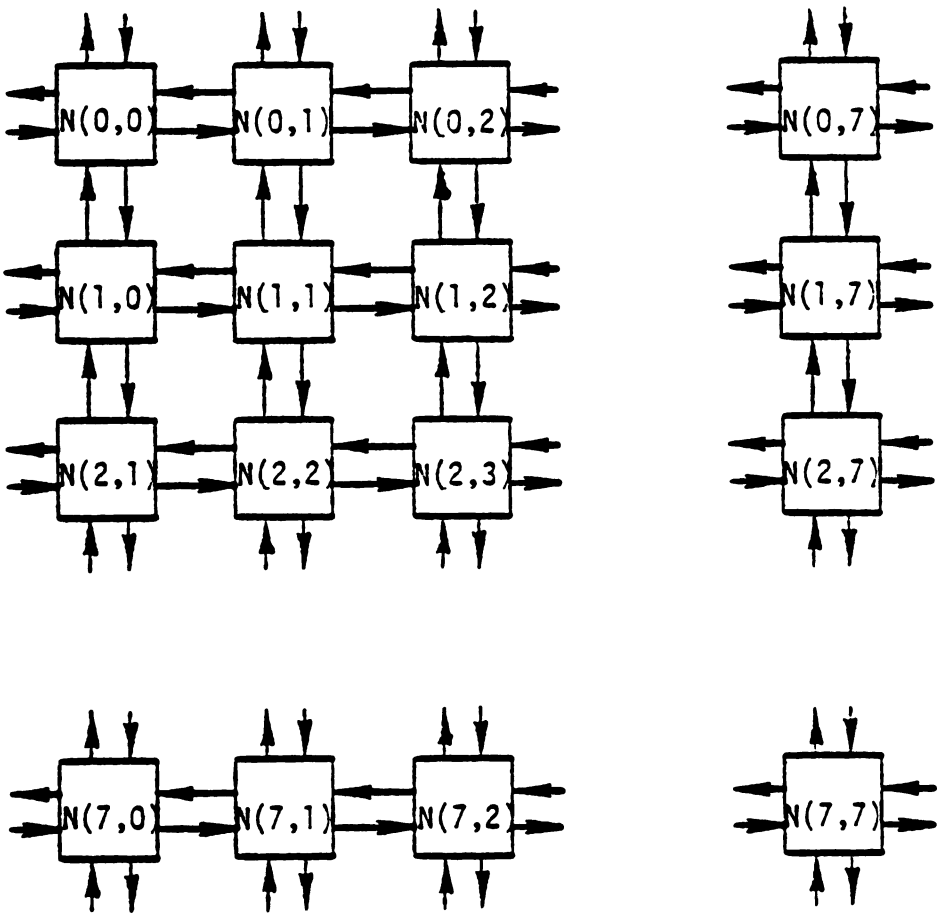


Figure 2.2. The array organization of the Illiac IV [12].

node to a disjointed node.

One important aspect of these array architectures is that the number of interconnections for each processing element is constant and these connecting wires are very regular and simple. This makes them very attractive in VLSI implementation. The drawbacks of these array architectures occur chiefly from the use of a centralized control and from their inherent fixed structures. These drawbacks include:

1. limited applicability--since every processing element must work in the same phase, a lot of effort must be taken in the design of concurrent algorithms before they can be efficiently executed on these machines [19,20].
2. system clock required--the scaling down of IC devices will result in difficulty of moving information from point to point synchronously with a system-wide clocking discipline. This will become even worse as the devices are scaled down to submicron features or as chips get larger [21].
3. I/O constraints--due to the constraint on the number of maximum I/O pinouts in VLSI chips, getting data into or off the parallel arrays can be very costly to the arrays' overall performance [22].

2.3 Systolic Arrays

A systolic array rhythmically computes and passes data through a network of tightly-coupled processors [8-9]. Since data are regularly pumped into and out of each processor, this structure has an advantage in ease of implementation and cost-effectiveness over the traditional parallel processors in which data flows are managed with the help of costly interconnection networks.

The basic component of the systolic array processor is the inner-product step processor (see Fig. 2.3) which consists of three registers, R_a, R_b, and R_c, each register having two ports, one for input and the other for output. During each unit-time interval, the processor loads data from its input lines into registers R_a, R_b, and R_c, respectively, computes R_c <-- R_a x R_b + R_c, and unloads R_a, R_b, and the new value of R_c as output. Using this inner-product processor as the building block, a number of systolic arrays for band matrices multiplication, LU-decomposition and solving triangular linear systems are designed [8-9].

As an example, a hex-connected systolic array for the matrix multiplication of two n x n band matrices, C = A x B with $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$, is shown in Fig. 2.4. In this example, the elements in the bands of A and B and the initialized values of C are pumped through the systolic network in three directions in a synchronous fashion. Entries in C are accumulated as it is shifted

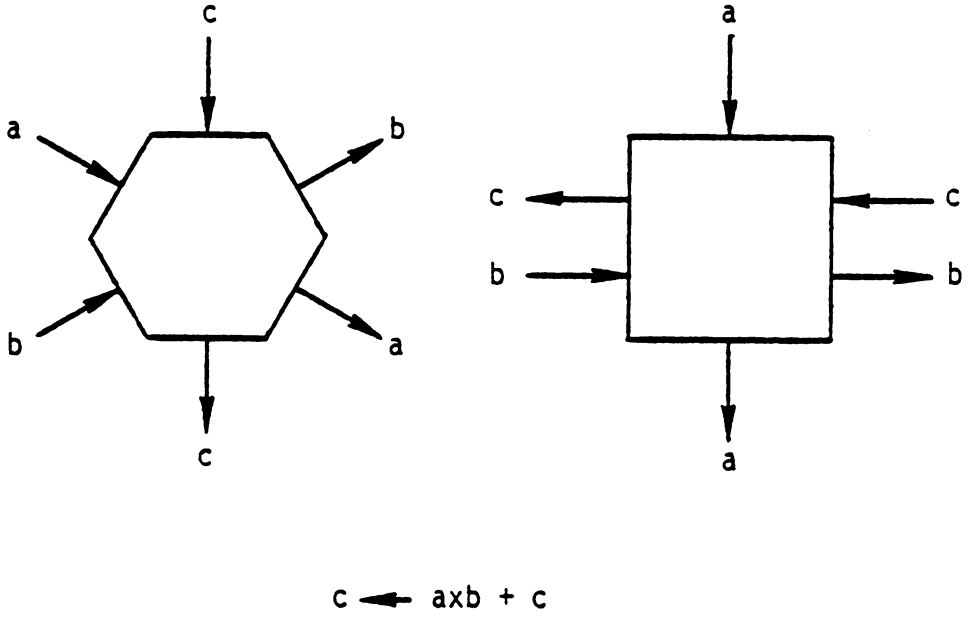


Figure 2.3. Two types of inner-product processors [9].

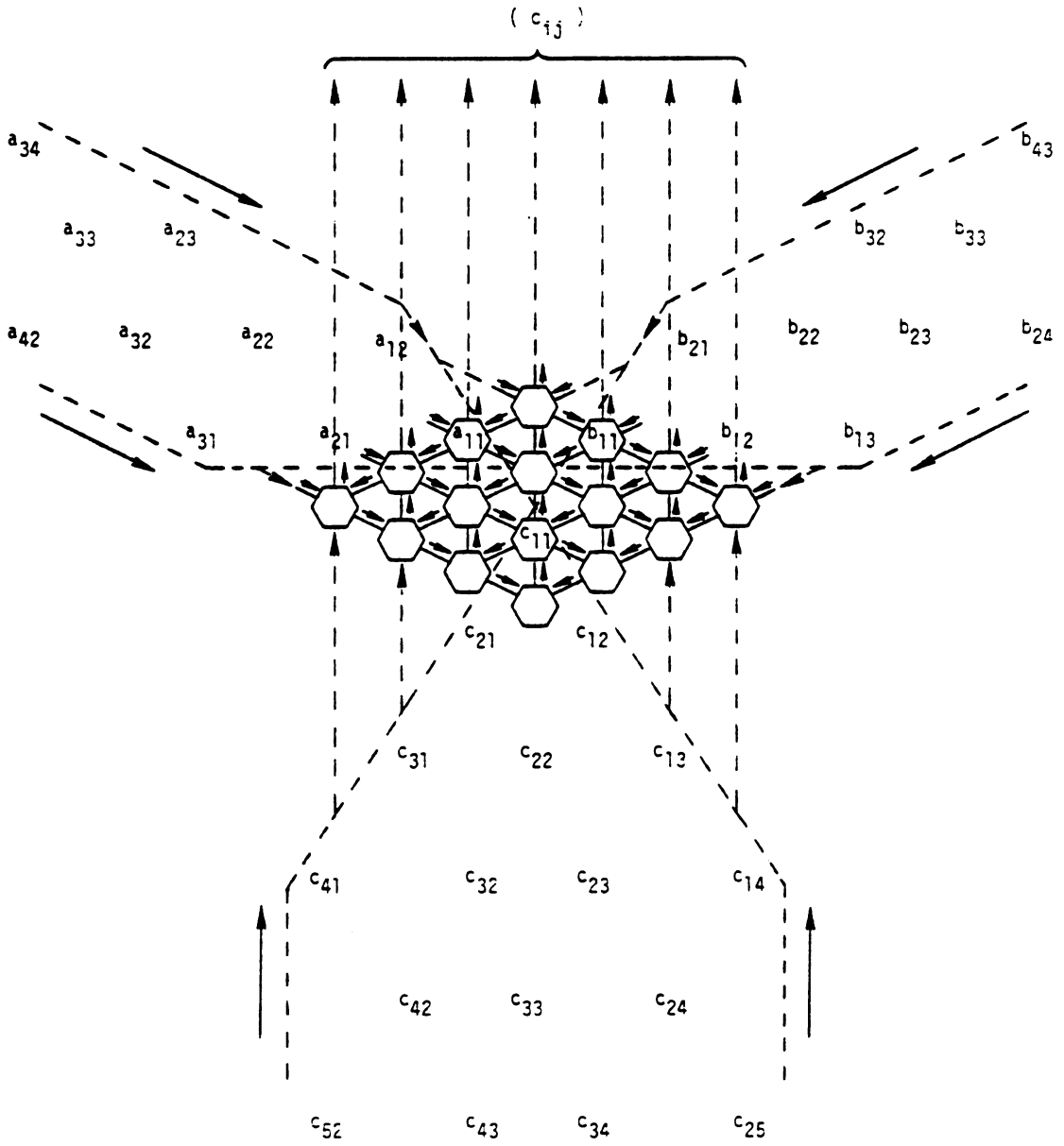


Figure 2.4. The hex-connected systolic array for matrix multiplication problem [9].

upward from the bottom of the array, where the c_{ij} enter with zero value. Each c_{ij} is able to accumulate all of its terms before it leaves through the upper boundaries.

As described, the systolic array architectures provide the capability for realizing a number of important matrix operations. In addition to achieving a high computational rate by means of pipelining and concurrent computation, these arrays are characterized by having a simple, regular and short communication geometry which is considered as one of the most desired attributes in VlSI implementation. The principal drawback to these special-purpose arrays is that they lack the flexibility in implementation since they are designed for specific applications, a redesign and reconstruction for new applications are required. Another drawback is that algorithms to be efficiently executed on these arrays must possess very simple and regular data flow patterns. Besides these drawbacks, due to the I/O constraint of pinouts on a chip, systolic arrays are hardly to be applied for practical implementation.

2.4. Data-Driven Machines

A data-driven machine executes data-flow programs modeled by Karp-Miller computation graphs [23]. These data-flow programs are represented as two-dimensional directed graphs, where nodes represent operations and links represent the data movements from operation to operation. Fig. 2.5 shows a simple Fortran-like statement represented as a

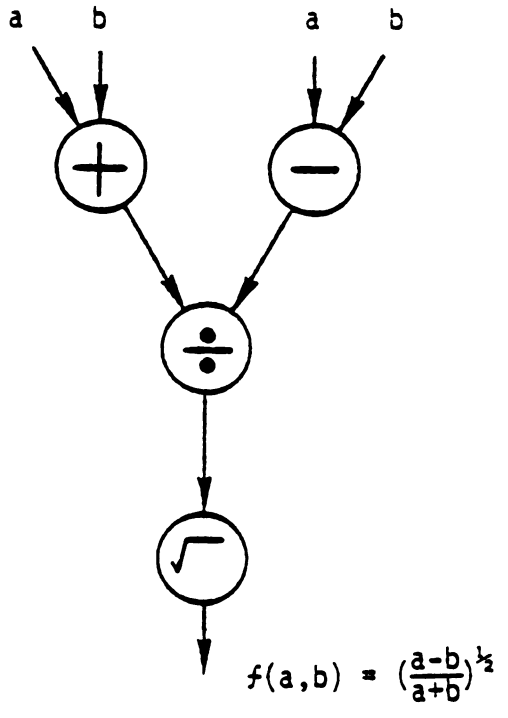


Figure 2.5. A directed graph representation example.

directed graph. The power of the directed graph is that data and control dependencies are explicitly expressed, as are the exact serial-parallel relationship of the set of operations to be executed. Unlike the conventional architectures, where execution of a program is managed by a program counter in a sequential manner, execution of a data-flow program is data-driven; an instruction proceeds on its own when its operands are available. Therefore, the data-driven approach allows a large number of instructions to be executed simultaneously.

Numerous data-driven machines have been reported in the past few years [24-28]. These include a series of machines with upgrading capability described by Dennis and Misunas [24-25]. Fig. 2.6 shows a block diagram of a basic data-driven machine from Dennis's group. In this machine, the instructions of a data-flow program are stored in the instruction memory cell. A data-flow instruction cell consists of a number of operand fields in which other instruction cells should receive the result from this instruction cell. An example of such an instruction cell is shown in Fig. 2.7. When an instruction cell fires, the data in the cell which will be needed in later phases of execution are transmitted in a packet to an arbitration network which routes the packet to one of a number of processors. When a processor completes its execution, a result packet is formed and transmitted to a distribution network. The distribution network transmits the result value computed by

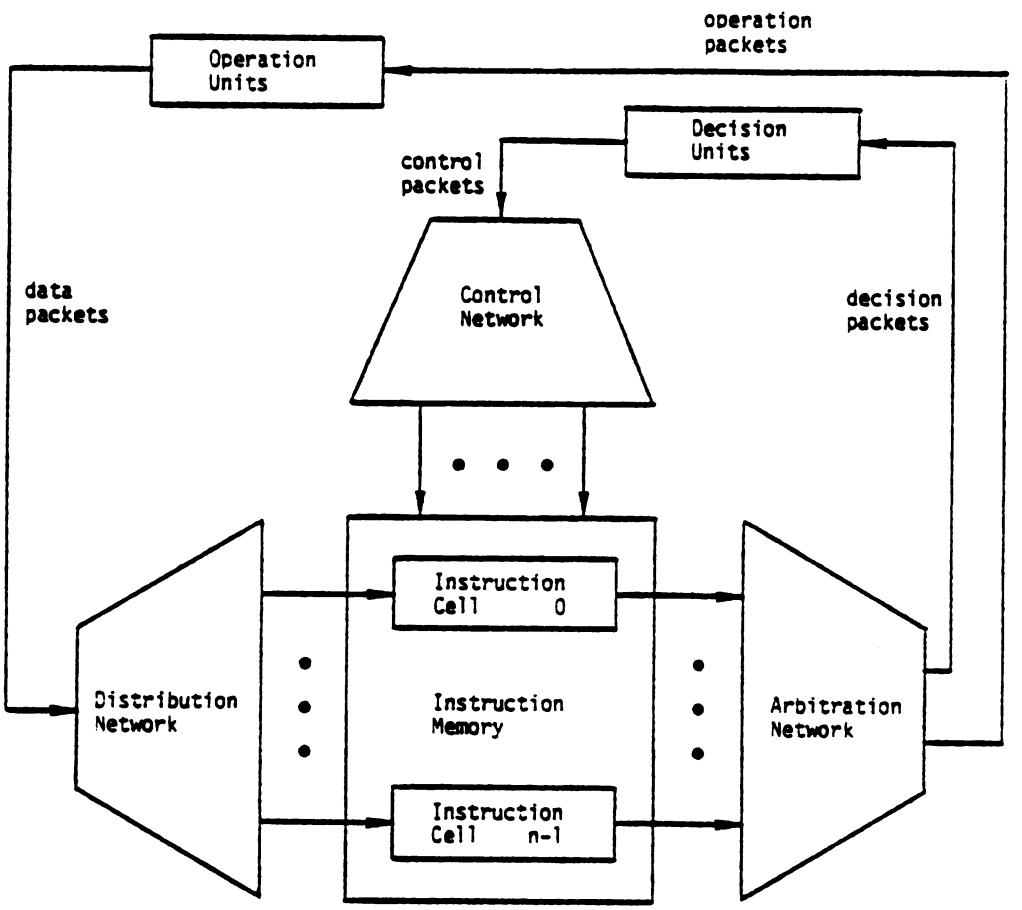


Figure 2.6. The block diagram of a basic data-driven machine [24].

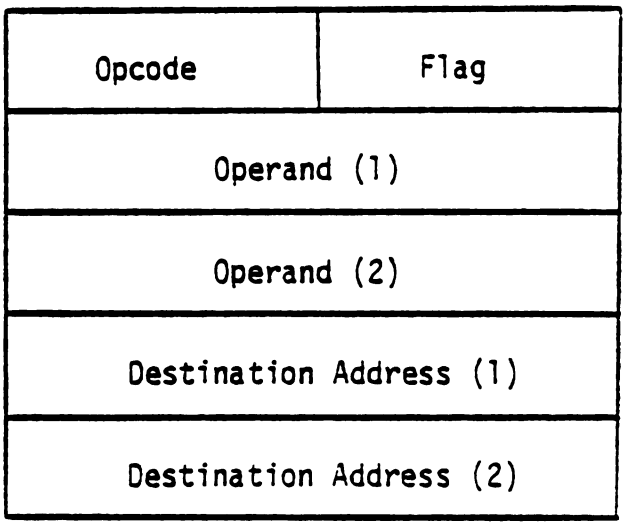


Figure 2.7. A data-flow instruction cell example.

the processor to each destination. Advantages of the data-driven architecture include flexibility in the machine design, flexibility in program control, exploitation of parallelism at all program levels. Disadvantages of the data-driven architecture include difficulties in implementing familiar data structures such as arrays and high overhead for communication between functional units of the machine [29-30].

2.5. Local Computation and Communication

The concept of defining local and global variables developed in structured programming languages suggests a similar approach for researchers who are developing VLSI computing systems. As predicted by Mead, future VLSI computing systems will have hundreds of processing elements clustered together on a single silicon chip [2]. The issue of how to coordinate these processing elements in order to improve the overall processing power has become more critical than before. An interprocessor communication network seems to be practical only if the amount of information flow among these processing elements is smaller than the number of tasks to be performed on them. To avoid a severe jam of information flow resulting from the use of a global communication network, one possible solution is to group a number of processing elements, which interact most frequently, into a block; thereby, data and control messages can move locally on the much shorter communication

paths [6,29].

Computer architectures based on the above approach have been reported [6,29]. One such machine proposed by Kuck et al. [6], is shown in Fig. 2.8. In this machine, local control structures and functional tasks invoked by these structures are fully distributed over blocks of processor clusters under a global controller. Compound functions or local computations are performed in concurrency on each of these processor clusters following the execution of their local control in a dependency-driven fashion. A two-level communication network is provided; the local interconnection network is used for both intracluster and adjacent clusters communications, while a global interconnection network provides facilities for both synchronous and asynchronous transfers of data or codes from the global memory to the clusters or vice versa.

As described in Sections 2.3 and 2.4, the architectures of the data-driven machines and of the systolic arrays represent innovations of the computer advances built to meet specific needs. The data-driven architecture supports asynchronous execution on both of the control and the function structures, while execution of these two structures on the systolic array is most likely to be synchronous rather than asynchronous. By combining these two machines' characteristics, it is possible to design a prototype machine which can be more useful and cost-effective than either of the other two. As proposed, this

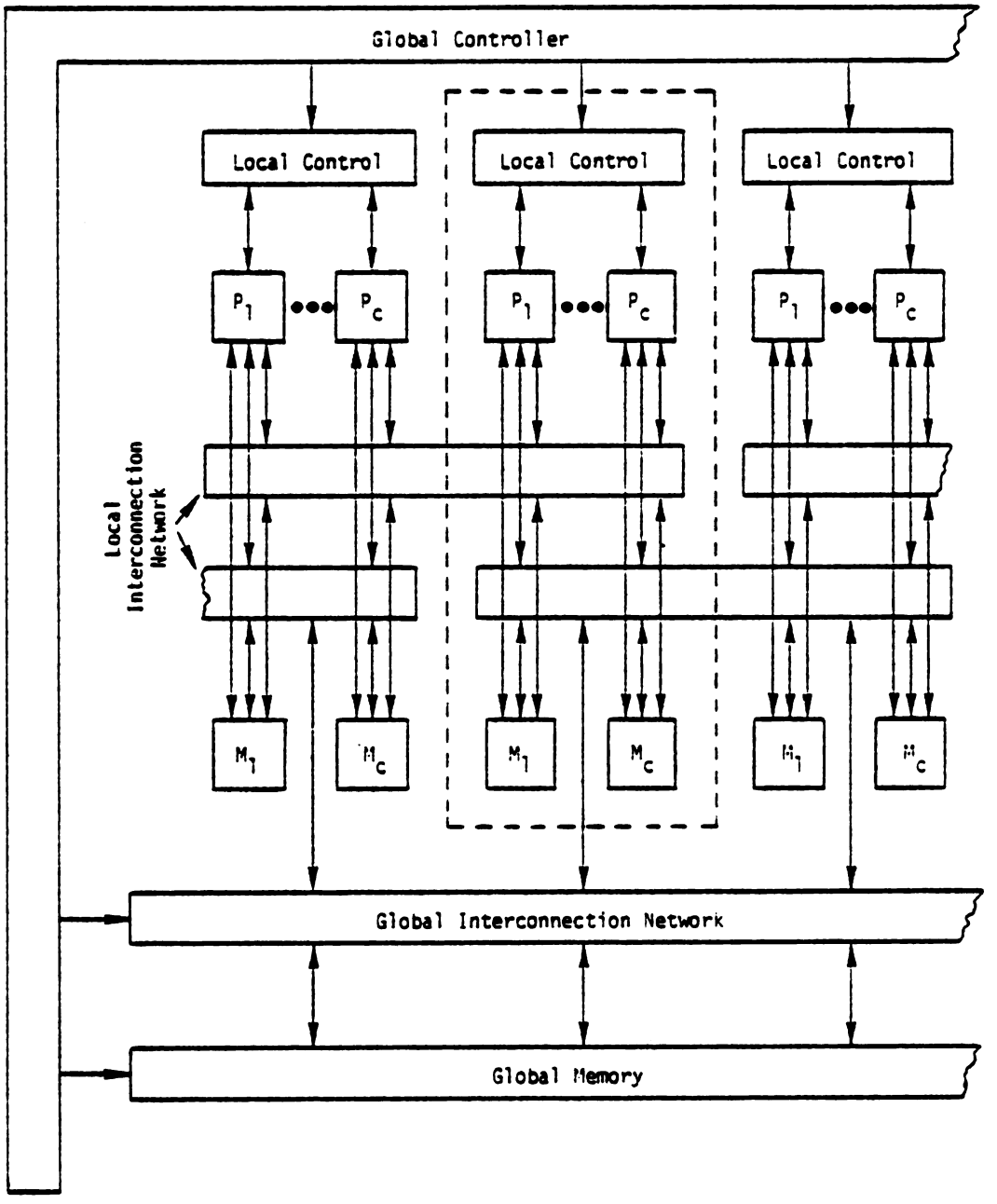


Figure 2.8. A multiprocessor system for the fast executions of numerical programs [6].

new class of machines will support asynchronous execution on the global control structures in a data-driven fashion and execute the functions invoked by local control structures synchronously on a local homogeneous modified systolic array. As described above, information communication in these array machines is also organized into a global-level and local-level basis. Inside the arrays, directed paths or hierarchical paths provide for fast local data transfers. At the global level, a distribution network is used to transfer global data and microcoded control instructions.

fo

Th

sy

we

an

tr

cu

in

ty

ge

3.

st

mo

de

th

co

CHAPTER III

PROGRAMMABLE SYSTOLIC ARRAYS

In this chapter, we provide the requisite groundwork for the development of a new class of array architectures. The array structure to be considered here is a modified systolic array or programmable systolic array [5]. First, we outline the basic programmable systolic array structure and its properties in Section 3.1. Following that, we illustrate the array's ability to implement a second-order recursive equation with nonconstant coefficients. We conclude in Section 3.3 with a discussion of the space-time complexity of this array structure and compare its features and general usefulness with those of the systolic array.

3.1. Functional and Structural Description

In general, a programmable systolic array (PSA) has a structured configuration with a simple and interconnectable module as its building block. A functional and structural description of this PSA follows:

* Processing Module (PM)--The processing module is the basic building block of the PSA. It is a localized computational center and a data router. For our purposes

here, each PM is constructed by connecting two pairs of interlinked computing elements (CE's) and control buffers (CB's) in a ring configuraiton (Fig. 3.1). But in general a PM may have more than two pairs. Also included in the processing module is a global I/O port which provides communication paths for the module.

* Computing Element (CE)--The computing element is a coprocessor on which elementary FLP operations are performed. It can be simply a numeric coprocessor such as the Intel 8087 [31] or a specially designed processor such as an inner-product step processor used in systolic arrays [8,9]. The choice is based on the computational demands of the class of algorithms to be implemented on the machine.

* Control Buffer (CB)--The control buffer is basically a data controller which directs the flow of incoming and outgoing data to and from the processing module according to their specific header and address destination tags. Each CB contains a number of buffer registers and a router. When the transfer of a tagged data item is made, the header is first decoded by the router to determine the destination ports to which this data item is to be sent, and the address is then applied to lead this data item to the proper location once transmitted on the port. Specifically, there are two types of tags being used in this array architecture. The tags, which are attached to a global data item, are not permanent and are removed when the transfer is complete. But those attached to a local data item are

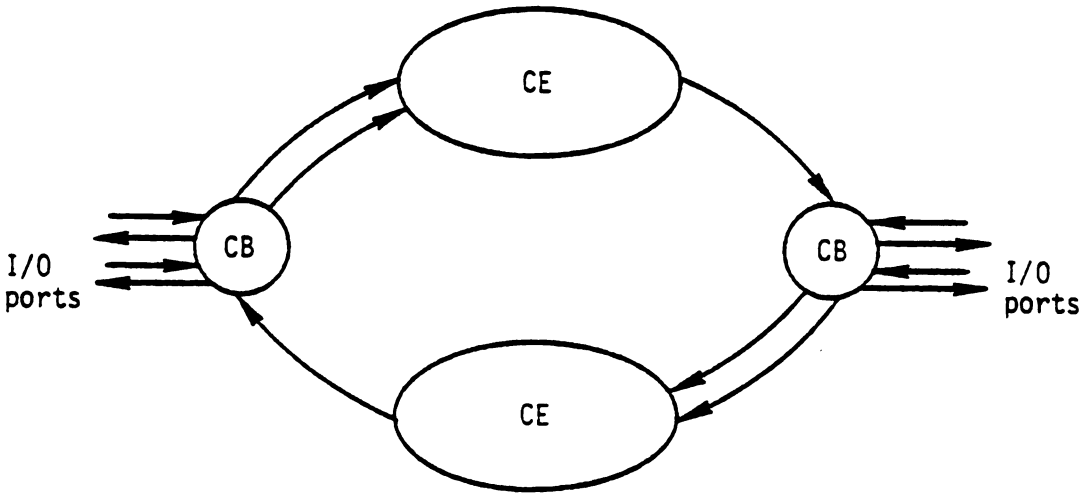


Figure 3.1. A ring-structured processing module.

preset and imbedded in a particular buffer register and are used exclusively to convey the data stored in the register.

To support the global and local hierarchy concept, each control buffer is multiply connected. A global I/O port, shared by the control buffer pair, provides communication for the processing module with the outside world, while local I/O ports are used for intermodule communications.

The structure of a processing module varies depending on the number of local I/O ports in the module. It can be a hexagon, an octagon, a decagon, etc. A PM with N pairs of local I/O ports is shown in Fig. 3.2, where the subscripts ℓ and g represent the local port and global port, respectively. The index k is used to number the module while the index j denotes the j_{th} local port in the module. The number of local I/O ports associated with a processing module can be enlarged, but there is a limit on the increase of this number due to fan-out limitations.

* Computation Series (CS)--A computation series is defined as a finite sequence of identical step computations with the requirement that each succeeding step computation in the series be driven by the result from the previous step. Conceptually, the forming of a computation series is derived by applying the data-driven principle to its execution [24-25]. The complexity of the step computations varies: it can be a simple elementary FLP operation such as

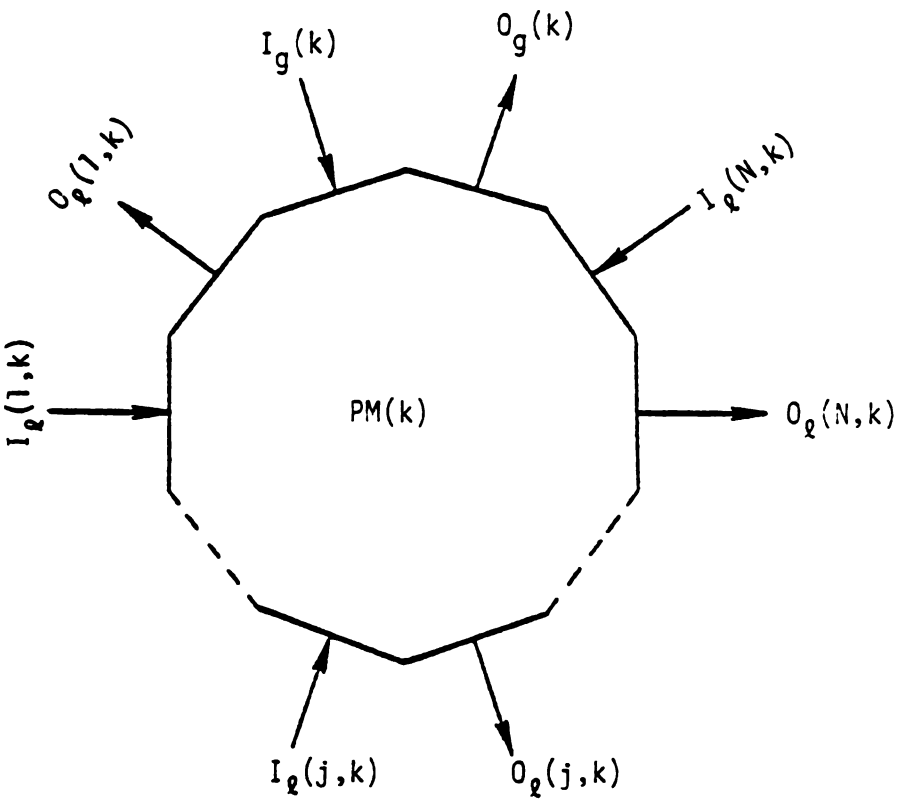


Figure 3.2. A PM with N pairs of local and one pair of global I/O ports.

addition or multiplication, or it can be a term of FLP operations such as (a + b x c), or it can even be as complex as an inner product of $(a_i b_i)$. A computation series can represent either intercomputations or intracomputations, depending on how it is executed inside the array. An intracomputation series refers to computations that are performed within a processing module, while an intercomputation series is taken sequentially on a step-by-step processing module basis. In general, a PSA built witn N processing modules will be able to compute N intracomputation series and one intercomputation series simultaneously. An example of these intra- and intercomputations is illustrated graphically in Fig. 3.3.

* Global Series Control (GSC)--A global series control signal, which is under the control of the host processor, initiates a new intercomputation series.

* Local Series Control (LSC)--A local series control signal, which is under the control of the control buffer, initiates a new intracomputation series.

* Global Data (GD)--The global data are pumped either into or out of the PSA under the control of the host processor.

* Local Data (LD)--The local data circulate within the array under the control of the control buffer. They can be intra types or inter types depending on where they are located.

* Constant Data (CD)--The constant data are

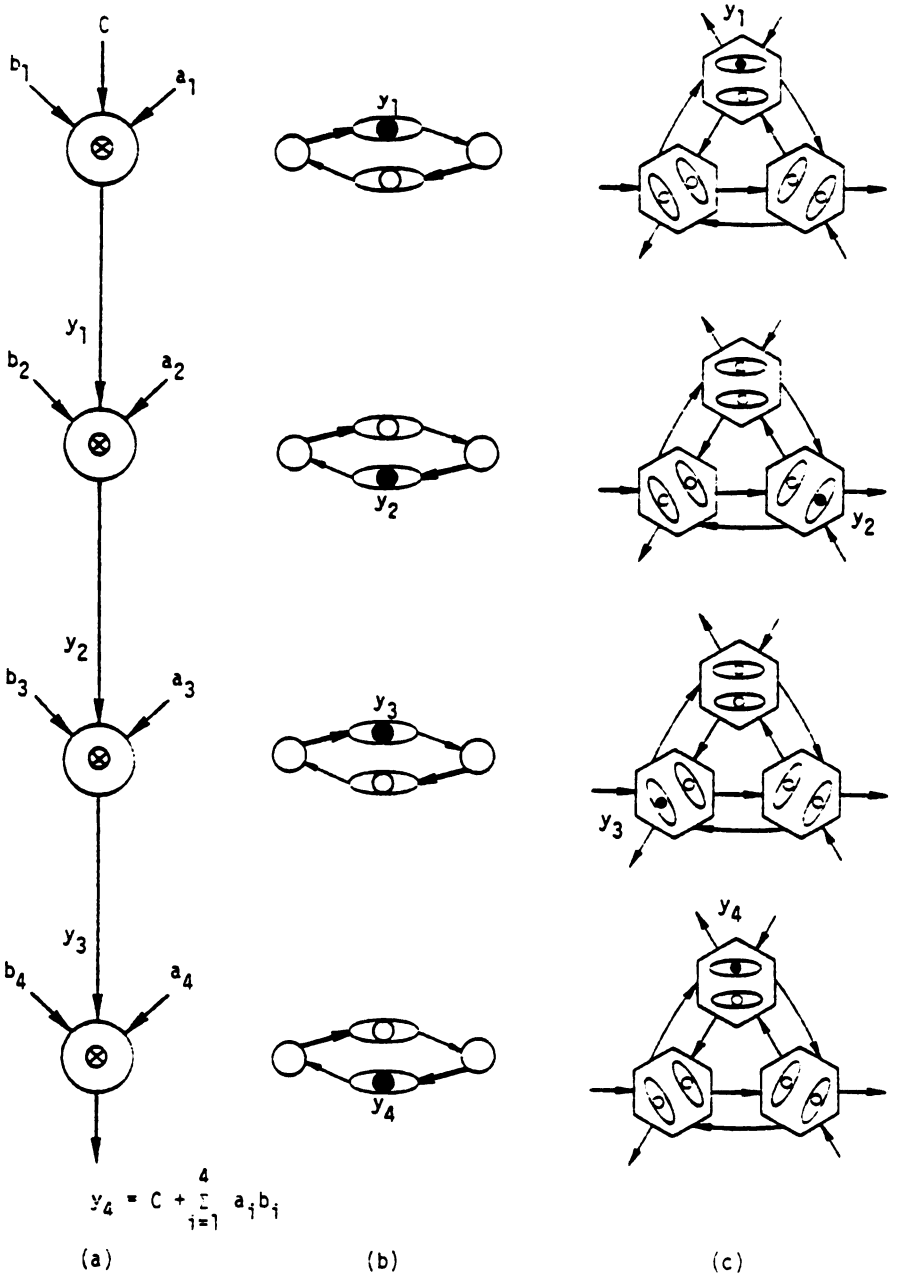


Figure 3.3.

(a) A computation series, $y_1 \rightarrow y_2 \rightarrow y_3 \rightarrow y_4$.

(b) Sequence of intracomputations executed on a PM.

(c) Sequence of intercomputations executed step-by-step on a three-PM array.

stationed in the registers of the control buffer. Usually, these data can be preloaded before computing commences by the host processor.

In a programmable systolic array, the execution of concurrent functions is represented as the parallel execution of inter- and intracomputation series. Each intracomputation series corresponds to a particular function, while each intercomputation series indicates the amount of interactions on these functions. The execution of these series can be visualized as rotating wheels. The principal wheel, which represents the intercomputation series, will rotate against the inner wheels of intracomputation series. The force which is applied to initiate the moving of a wheel is in reality the series control (SC) signal. If all of the rotating wheels are set to move in the clockwise direction, conflict-free computations result. This is accomplished by feeding the address codes and the control headers into the wheels to steer the direction. In Fig. 3.4, we illustrate this rotating wheel computing structure concept.

3.2. Sample Algorithm Implementation

The utility of this programmable systolic array structure can be illustrated through implementation of a second-order recursive equation of the following general form of

$$Y_i = C_i + \sum_{j=i-1}^{i-2} (C_{ij}X_j + d_{ij}Y_j), \quad 2 \leq i \leq n, \; Y_1 = 0$$

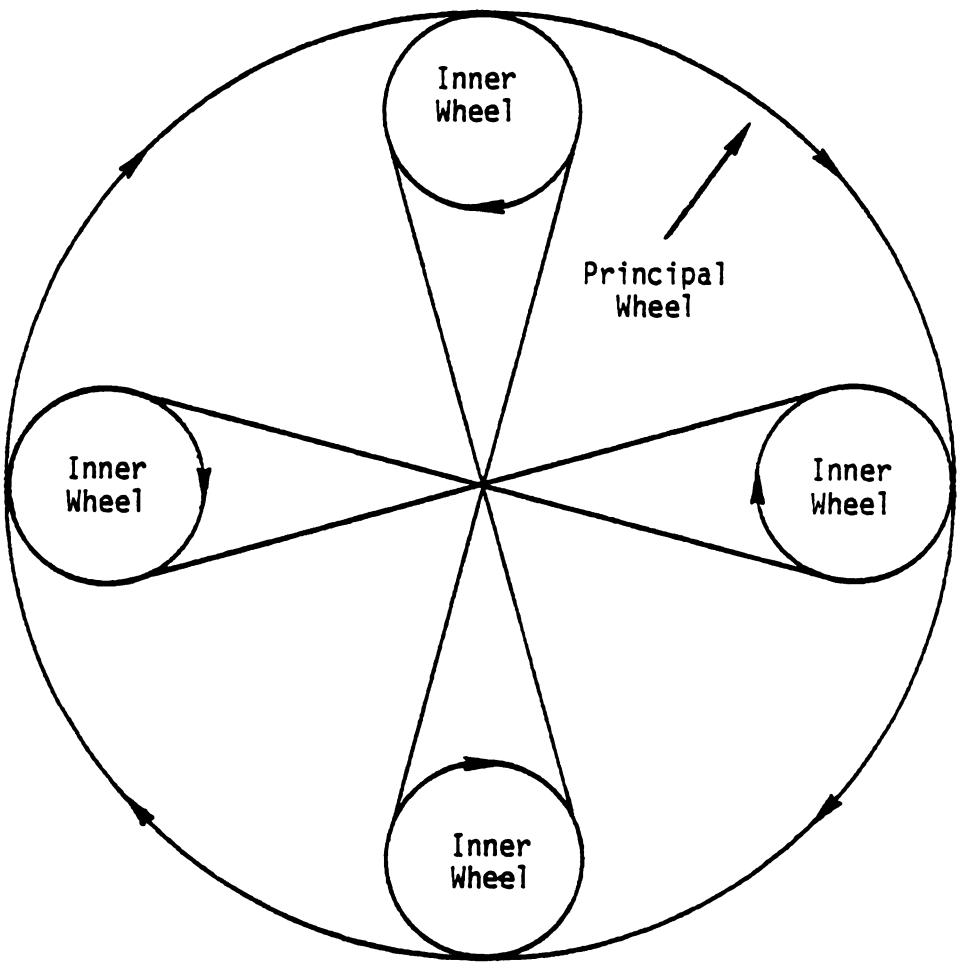


Figure 3.4. A rotating wheel computing structure example.

C_i are constant, C_{ij} and d_{ij} are precomputed weights, X_j are input data, Y_j are output results. Since we are interested in a modular design, the computing of outputs Y_i are distributed over a number of interconnected processing modules. In this particular application, three processing modules PM(1), PM(2), and PM(3) connected in a triangular configuration are used with each PM(k) assigned to compute outputs $Y_{(k + 3\ell + 1)}$, $\ell = 0, 1, 2, \ldots$, etc.

The specific design of a PM is of major concern and consists of the design and interconnection of a CE and a CB. Based on the inner-product type of computation in the computing of outputs Y_i, the grain size of the CE is chosen to be an inner-product step computation of (a + b x c). Since double recurrence occurs in each computation of Y_i, the CB possesses dual local I/O ports. The fully connected PSA with specifications for each of its building components is shown in Fig. 3.5. And to understand how a second-order recursive equation is evaluated on such an array, it is instructive to note the flow of the intercomputation series and those of the intracomputation series within the array.

The intercomputation series is chosen here to be

$$Y_1 \text{-->} Y_2 \text{-->} Y_3 \text{-->} \ldots \text{-->} Y_n.$$

And the intracomputation series are those which advance in computing of Y_i, that is, those series of

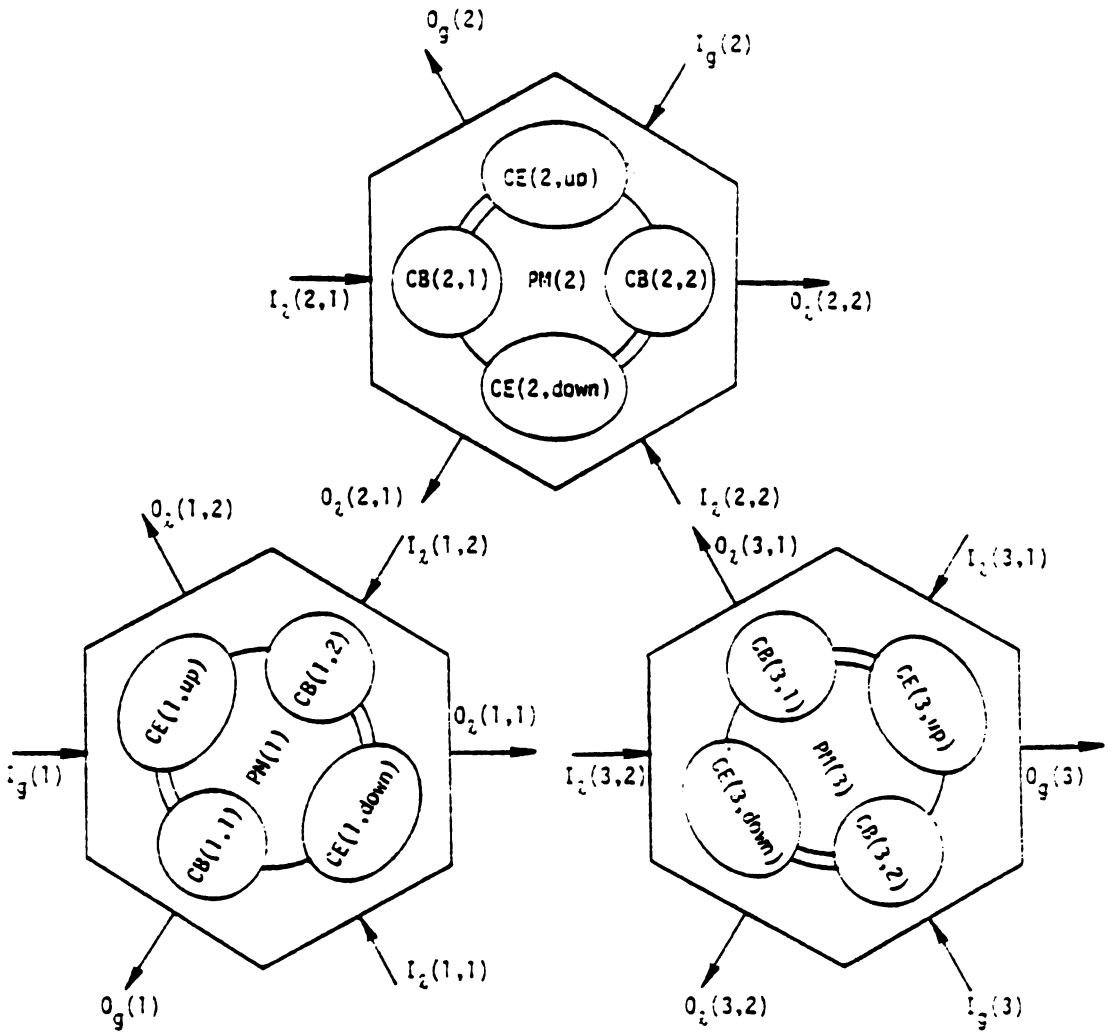


Figure 3.5. The triangular PSA for solving a second-order recursive problem.

$$[C_i + C_{ii-1}X_{i-1}] \rightarrow [(C_i + C_{ii-1}X_{i-1}) + C_{ii-2}X_{i-2}] \rightarrow$$
$$[((C_i + C_{ii-1}X_{i-1}) + C_{ii-2}X_{i-2}) + d_{ii-2}Y_{i-2}], \quad 2 \leq i \leq n.$$

Execution of the intercomputation series is taken along the perimeter of the array with each step computation $Y_{(k+3\ell+1)}$ being computed by the CE(k,up), while three concurrent intracomputation series are executed simultaneously within the ring-structured processing modules.

Before the computation initiates, the constant and the precomputed weights, the address codes, and the control headers are loaded into the registers of the CB's. Following execution of the series controls, the input data $X_{(k+3\ell)}$ are alternately pumped into the $I_g(k)$ ports. Accordingly, intra- and intercomputation series are computed. And, a few cycles later, the computational results $Y_{(k+3\ell+1)}$ are pumped out at the $O_g(k)$ ports; simultaneously, they are broadcast to the addressed modules via the intermodule connected paths to stimulate new computations. We illustrate the first eight computational steps in Fig. 3.6, and a more detailed description of the data flow timing for this example is provided in Table 3.1.

3.3. Discussion

The triangular array executes an n-input second-order recursive computation in O(n) time units after the initial preload step. Basically, the array's performance bears some resemblance to that of a pipeline architecture [32-33]. A

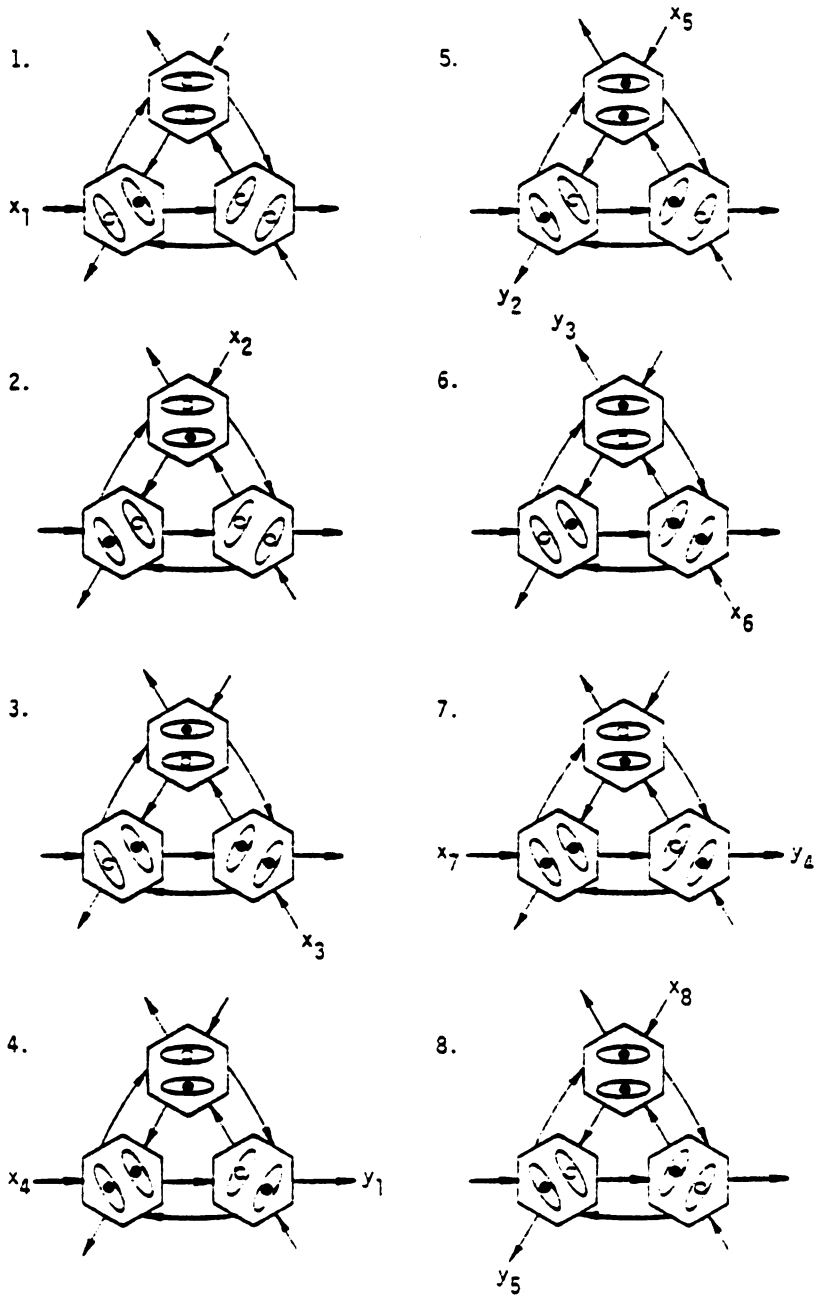


Figure 3.6. Computing steps for a second-order recursive equation on a triangular PSA.

Table 3.1

Data Flow Timing Example

Computation steps	Module Specifications		
	PM(1)	PM(2)	PM(3)
1	$:X_1 \rightarrow I_g(1) \rightarrow CB(1,2) \rightarrow CB(2,1)$ CE(1,up) : idle CE(1,down) : $C_2 + C_{21}\ X_1$	$:X_1 \rightarrow CB(2,1)$ CE(2,up) : idle CE(2,down) : idle	CE(3,up) : idle CE(3,down) : idle
2	CE(1,up) : $(C_2 + C_{21} X_1) \rightarrow 0\ X\ 0$ CE(1,down) : idle	$:X_2 \rightarrow I_g(2) \rightarrow CB(2,2) \rightarrow CB(3,1)$ CE(2,up) : idle CE(2,down) : $C_3 + C_{32}\ X_2$	$:X_2 \rightarrow CB(3,1)$ CE(3,up) : idle CE(3,down) : idle
3	$:X_3 \rightarrow CB(1,1)$ CE(1,up) : idle CE(1,down) : $(C_2 + C_{21}\ X_1) \rightarrow 0\ X\ 0$	CE(2,up) : $(C_3 + C_{32}\ X_2) + C_{31}\ X_1$ CE(2,down) : idle	$:X_3 \rightarrow I_g(3) \rightarrow CB(3,2) \rightarrow CB(1,1)$ CE(3,up) : Y_1 CE(3,down) : $C_4 + C_{43}\ X_3$
4	$:X_4 \rightarrow I_g(1) \rightarrow CB(1,2) \rightarrow CB(2,1)$ CE(1,up) : Y_2 CE(1,down) : $C_5 + C_{54}\ X_4$	$:X_4 \rightarrow CB(2,1)$ CE(2,up) : idle CE(2,down) : $((C_3 + C_{32}\ X_2) + C_{31}\ X_1) + d_{31}\ Y_1$	CB(3,2) : $y_1 \rightarrow \begin{cases} O_g(3) \\ CB(1,1) \end{cases}$ CE(3,up) : $(C_4 + C_{43}\ X_3) + C_{42}\ X_2$ CE(3,down) : idle
5	CB(1,2) : $Y_2 \rightarrow \begin{cases} O_g(1) \\ CB(2,1) \\ CB(3,2) \end{cases}$ CE(1,up) : $(C_5 + C_{54}\ X_4) + C_{53}\ X_3$ CE(1,down) : idle	$:X_5 \rightarrow I_g(2) \rightarrow CB(2,2) \rightarrow CB(3,1)$ CE(2,up) : Y_3 CE(2,down) : $C_6 + C_{65}\ X_5$	$:X_5 \rightarrow CB(3,1)$ CE(3,up) : idle CE(3,down) : $((C_4 + C_{43}\ X_3) + C_{42}\ X_2) + d_{42}\ Y_2$
6	$:X_6 \rightarrow CB(1,1)$ CE(1,up) : idle CE(1,down) : $((C_5 + C_{54}\ X_4) + C_{53}\ X_3) + d_{53}\ y_3$	CB(2,2) : $Y_3 \rightarrow \begin{cases} O_g(2) \\ CB(3,1) \\ CB(1,2) \end{cases}$ CE(2,up) : $(C_6 + C_{65}\ X_5) + C_{64}\ X_4$ CE(2,down) : idle	$:X_5 \rightarrow I_g(3) \rightarrow CB(3,2) \rightarrow CB(1,1)$ CE(3,up) : Y_4 CE(3,down) : $C_7 + C_{76}\ X_6$
7	$:X_7 \rightarrow I_g(1) \rightarrow CB(1,2) \rightarrow CB(2,1)$ CE(1,up) : Y_5 CE(1,down) : $C_8 + C_{87}\ X_7$	$:X_7 \rightarrow CB(2,1)$ CE(2,up) : idle CE(2,down) : $((C_6 + C_{65}\ X_5) + C_{64}\ X_4) + d_{64}\ Y_4$	CB(3,2) : $Y_4 \rightarrow \begin{cases} O_g(3) \\ CB(1,1) \\ CB(2,2) \end{cases}$ CE(3,up) : $(C_7 + C_{76}\ X_6) + C_{75}\ X_5$ CE(3,down) : idle
8	CB(1,2) : $Y_5 \rightarrow \begin{cases} O_g(1) \\ CB(2,1) \\ CB(3,2) \end{cases}$ CE(1,up) : $(C_8 + C_{87}\ X_7) + C_{86}\ X_6$ CE(1,down) : idle	$:X_8 \rightarrow I_g(2) \rightarrow CB(2,2) \rightarrow CB(3,1)$ CE(2,up) : Y_6 CE(2,down) : $C_9 + C_{98}\ X_8$	$:X_8 \rightarrow CB(3,1)$ CE(3,up) : idle CE(3,down) : $((C_7 + C_{76}\ X_6) + C_{75}\ X_5) + d_{75}\ Y_5$

start-up time, t_x, is required to set the array for later pipelining operations. It includes the time, t_h, to preload the tags of header and address and the initial computing time, t_i. As in a 3-stage pipeline, the second-order recursive computation requires an initial computing time of 3 x t_y based on the unit time latency t_y. Since a multiply and an add operation are performed during each unit time, after adding the data transfer time, t_c, we can express the latency of the pipeline operation as

$$t_y = t_p + t_d + t_c$$

where t_p is the time of a multiply operation and t_d is the time of an add operation. Thus, for an n-input second-order recursive problem the execution time, T_r, on a triangular PSA can be expressed as

$$\begin{aligned} T_r &= t_h + 3(t_p + t_d + t_c) + n(t_p + t_d + t_c), \\ &= t_h + (3 + n)(t_p + t_d + t_c). \end{aligned}$$

As with a von Neumann machine, it takes 6 multiply operations and 6 add operations for each second-order recurrence. Thus, by assuming the same data transfer time required for each operation, the execution time, T_v, on such a machine is

$$\begin{aligned} T_v &= n(6t_p + 6t_d + 12t_c), \text{ or} \\ &= 6n(t_p + t_d + 2t_c). \end{aligned}$$

Therefore, the PSA represents a speed-up of

$$\frac{T_v}{T_r} = \frac{6n(t_p + t_d + 2\,t_c)}{t_h + (3 + n)(t_p + t_d + t_c)}$$

The significance of this speed-up has stemmed in part from the concurrent operations of three processing modules and in part from the 2-stage pipeline operation in each module and also from applying data-driven computations which, as observed, have reduced the overhead of the so-called von Neumann bottleneck [10] by a factor of 3. Like the pipeline operation, the PSA will have a maximum speed-up of 6 provided the number of operations n is much greater than the order of recurrence.

The programmable systolic array described here excels over the systolic array in two aspects. First, the PSA possesses a dynamic probelm-solving capability. Through microprogramming different control headers into the control buffers, a PSA can be applied to solve a variety of probelms as compared to a static systolic array, which is specially designed to solve a particular problem. For example, the described triangular PSA can be used in solving a second-order recursive problem, a 3 x 3 matrix multiplication, an inner-product computation, or any computational tasks which can be decomposed into up to three concurrent operations; this is not possible with a systolic array. Second, the PSA contains a set of internal buffer registers, which provide a means to preload the constant

data, thereby reducing the traffic density at the global I/O ports. For example, the precomputed weights and the constant data in the second-order recursive computation described in the previous section or the elements in one of the two matrices in the matrix multiplication can be pre-loaded. But there is an added cost in using the PSA. The host processor must interact with a more complex control structure. Moreover, the control buffer and the amount of intermodule connections cause the PSA to reduce the maximum number of computing elements per unit chip area.

CHAPTER IV

MIXED SYSTOLIC ARRAYS

Attempts to broaden the systolic array's potential usefulness have been made either through designing new structures of systolic systems, such as systolic priority queues [34], or through modifying the systolic array. Examples of the latter approach include programmable systolic arrays [5] and cellular systolic arrays for the dynamic programming computation and the transitive closure problem [35]. Designing new systolic structures for specific applications has its inherent limitation because many specialized VLSI devices would be required, i.e., one for each algorithm. However, the modifying approach may be more fruitful. Methodologically, this approach implements control structures or switching circuits into the systolic array, or simply includes control information in the data packets [5,35,36]. In this chapter, we present strategies for mixing control buffers and computing elements in the array structure. The word mixing is employed here to indicate the presence of both control buffers and computing elements within the arrays and to specify, as a result, how these arrays' characteristics change from their original

ones. First, the basic structure of a mixed systolic array (MSA) is described. Next, in Section 4.2, we discuss the impact that different mixing ratios have on a given uniform network and how this relates to global vs. local data requirements for a particular algorithm. This is followed by the discussion of partial uniform mixing and various MSA structures.

4.1. Structure of Mixed Systolic Arrays

The first step toward the modification of a systolic array is through mixing, a strategy of incorporating both control buffers and computing elements into an array. A control buffer is a data router; it directs the flow of incoming data and outgoing data to and from the computing element according to their specific data type headers and destination address tags, but it is also a data and control code buffer, providing a local storage queue for data-flow instruction cells. For our purposes, a computing element is a coprocessor on which elementary arithmetic operations (+,-,x,/) are performed.

A mixed systolic array (MSA) differs from Kung's systolic arrays in a number of ways [37]. Structurally, a mixed systolic array is formed by using two basic elements, as compared to the original array, which contains only one basic computing element. If the basic computing elements and the control buffers are not uniformly placed in the array, an MSA array displays some degree of irregularity in

its structure. Functionally, an MSA executes its computational steps according to the sequence of programmed control codes stored in the control buffers. Therefore, a given implementation may be applied to solve a broader spectrum of user design algorithms than a conventional systolic array. Geometrically, an MSA communicates with the outside world through its control buffers, which are considered as the masters. This master-slave MSA architecture is more easily interfaced with other chips than a slave-based systolic array chip.

Basically, the structure of an MSA is determined by the mixing profile. By the term uniform mixing, we mean there is a subarray or basis from which the array can be constructed. We consider a uniformly mixed array a regular array. Fig. 4.1 shows a mixed systolic array constructed from a diamond-like basis. When mixing is placed on some particular regions in the array with emphasis, it yields an irregular array structure. And an irregular array is partially regular if it can be segmented into two or more regions which are each uniformly mixed (see Fig. 4.2).

The mixing profile determines the possible frames of data-flow patterns within the MSA. Unlike the original systolic array, on which execution of an algorithm is based on two-dimensional or two-way pipelined data flows moving along in the preformed paths from one side of the array to

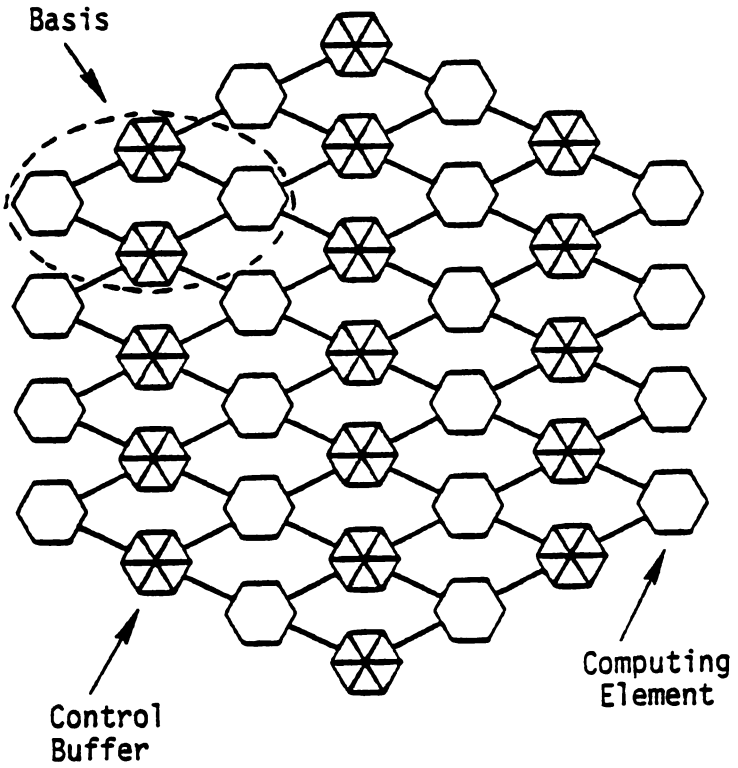


Figure 4.1. A mixed systolic array constructed from a diamond-like basis.

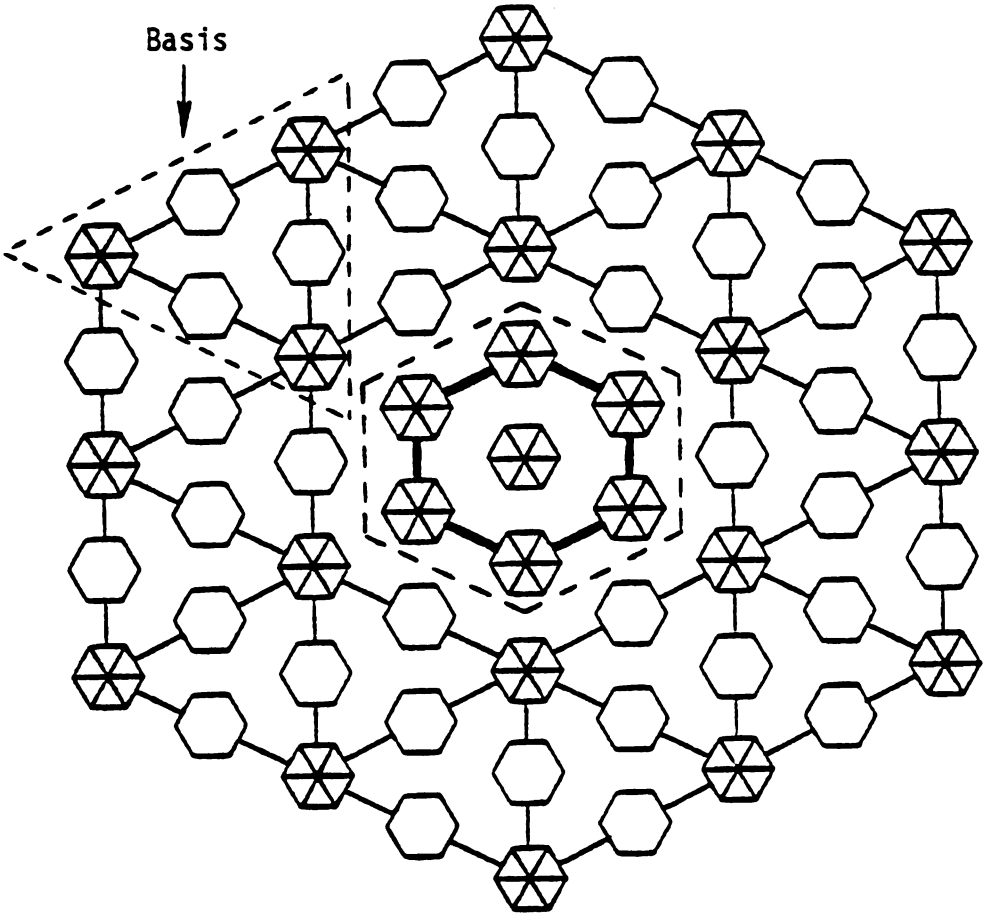


Figure 4.2. A partially regular hexagonal mixed systolic array composed of a central region and an outer shell. The latter is constructed from a triangular basis.

the other, an MSA, in contrast, has reconfigurable pipelined paths. Therefore, algorithms with similar global vs. local data requirements can all be effectively executed on it rather than on numerous systolic arrays.

The mixing profile contains information about the mixing density and the boundary conditions. The mixing density function, ρ , is defined as

$$\rho = \frac{N_{CB}}{N_{CB} + N_{CE}} \tag{4.1}$$

where N_{CB} And N_{CE} are the number of control buffers and the number of computing elements contained in the array, respectively. When the mixing density is high, the array most likely will be used for algorithms which require more complex data routing and less complex computing. However, the array with a low mixing density will be applied primarily to algorithms which have simple data routing but require a large amount of computing. There are two limiting cases: when ρ=0, the MSA reduces to a systolic array. And when ρ=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:

$$\Omega = \frac{N_{CB'}}{N_{CB}} , \tag{4.2}$$

where $N_{CB'}$ is the number of control buffers placed on the

boundary of the array. Since the control buffers on the boundary of the MSA are the communication centers, connections from these centers to the outside world provide the basic I/O operations. Therefore, the evaluation of Ω is a direct measure of the complexity of the I/O connections. Thus, the proper choice of ρ and Ω for a particular implementation relates directly to the ratio of the global and local data requirements of the algorithms for which the MSA is intended.

4.2. Uniform Mixing

Although the number of mixing profiles which yield a regular array structure is large, only a few of them appear to be practical for real algorithm applications. What's more, a balance must be achieved between the computing power and the data routing capability in order to maximize resource utilization and throughput. For these two reasons, we will focus here only on those profiles which yield a geometric balance between the control buffers and the computing elements in the structure of their corresponding bases. Also, for ease in implementation we assume that each control buffer has the same physical size as the computing element.

The hexagonal array is a very important array structure because of its utilization of the planar communication and its high packing density. In a hexagonal array the total number of elements, N, contained in the array can be

evaluated from the array's edge length, n, as

$$N = 2[n + (n + 1) + (n + 2) +...+ (2n -2) + (2n-1)]$$
$$= 3n^2 - 3n + 1 \qquad (4.3)$$

The smallest hexagonal array is the one which has size N=7 when n=2. Since most of the array structures of interest are subsets of a hexagonal topology [30] and the hexagonal array of any size can be grown from such a structure with sharing or overlapping, this elemental array structure conveniently serves as the mixing basis. When this smallest hexagonal array is mixed with the control buffers, the basis-mixing density, ρ_b, in the basis can be one of the following:

$$\rho_b = \frac{i}{7}, \quad i=1,2,....,7.$$

But mixing profile constraints state that each control buffer to be placed into a basis (1) will have the same number of computing elements at its nearest neighbors except the one which is to be placed in the center of the basis and (2) will seek to have the maximum number of computing elements at its nearest neighbors. The first constraint assures uniform mixing in the mixed array when it is grown from such a basis, while the second maximizes the computing power of the mixed array. Based on these two constraints, the set of computing elements neighboring each control buffer with respect to the above basis-mixing

density set is

(6, 3, 3, 2, 1, 1, 0).

The bases corresponding to the first three mixing profiles are shown in Fig. 4.3. MSA's grown from these particular bases are intended for computation use.

With an increased fraction of control buffers being mixed into the arrays, we are interested in knowing to what extent the computing power has been sacrificed, as well as what constraints must be added due to the presence of the control buffers. The hexagonal MSA's grown from the bases in Fig. 4.3 are presented here for comparison. The hexagonal arrays, as shown in Fig. 4.4, are formed (or grown) by adding a new basis adjacent either to each of the growing bases' edges or corners. The mixing density function, ρ, and the boundary condition function, Ω, in these MSA's are given as follows:

Case 1: (MSA's constructed from a $\rho_b = 1/7$ basis)

The number of control buffers in this type of MSA's is given by

$$N_{CB} = 6\,[p + (p-1) + \ldots + 0] + 1$$

where $p = \left\lceil \frac{n-1}{2} \right\rceil$.

If n is odd,

$$N_{CB} = \frac{3n^2 + 1}{4}$$

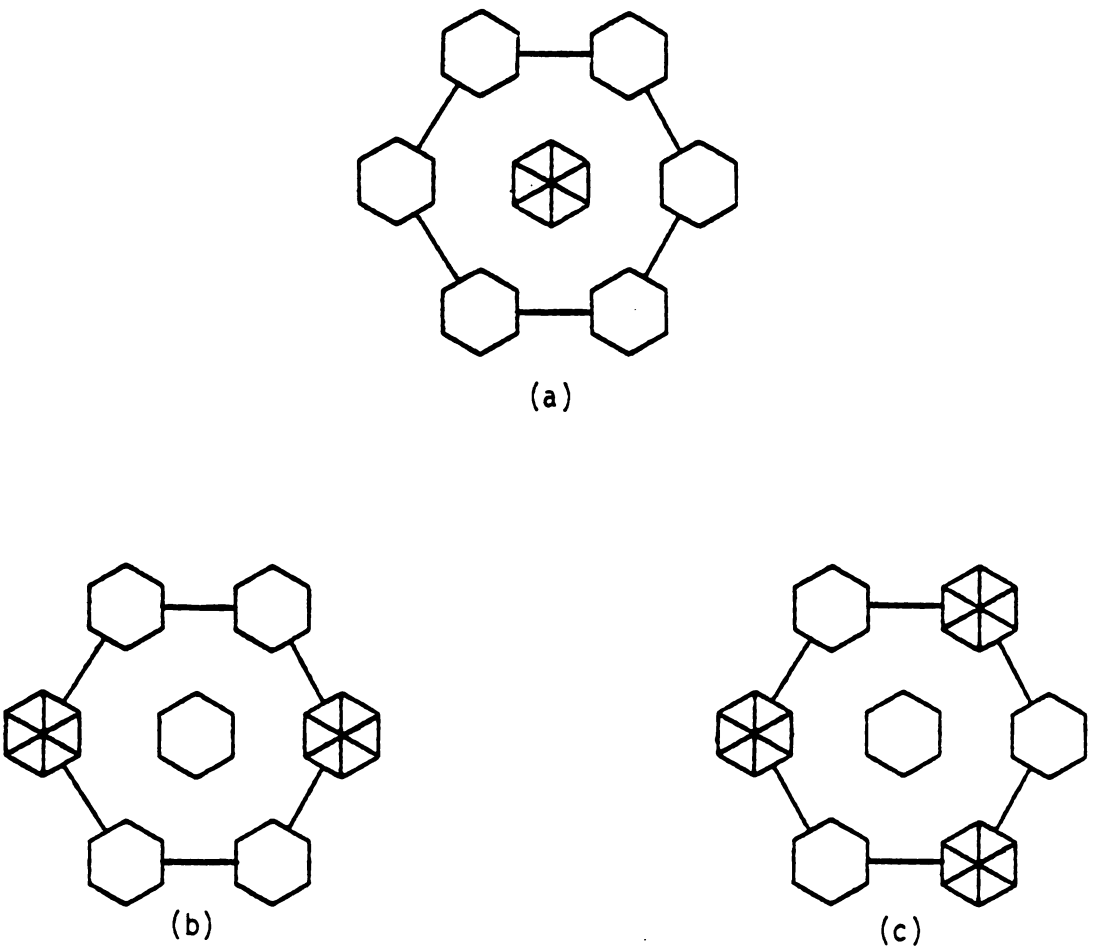


Figure 4.3. Hexagonal array bases with (a) ρ_b = 1/7, (b) ρ_b = 2/7, and (c) ρ_b = 3/7.

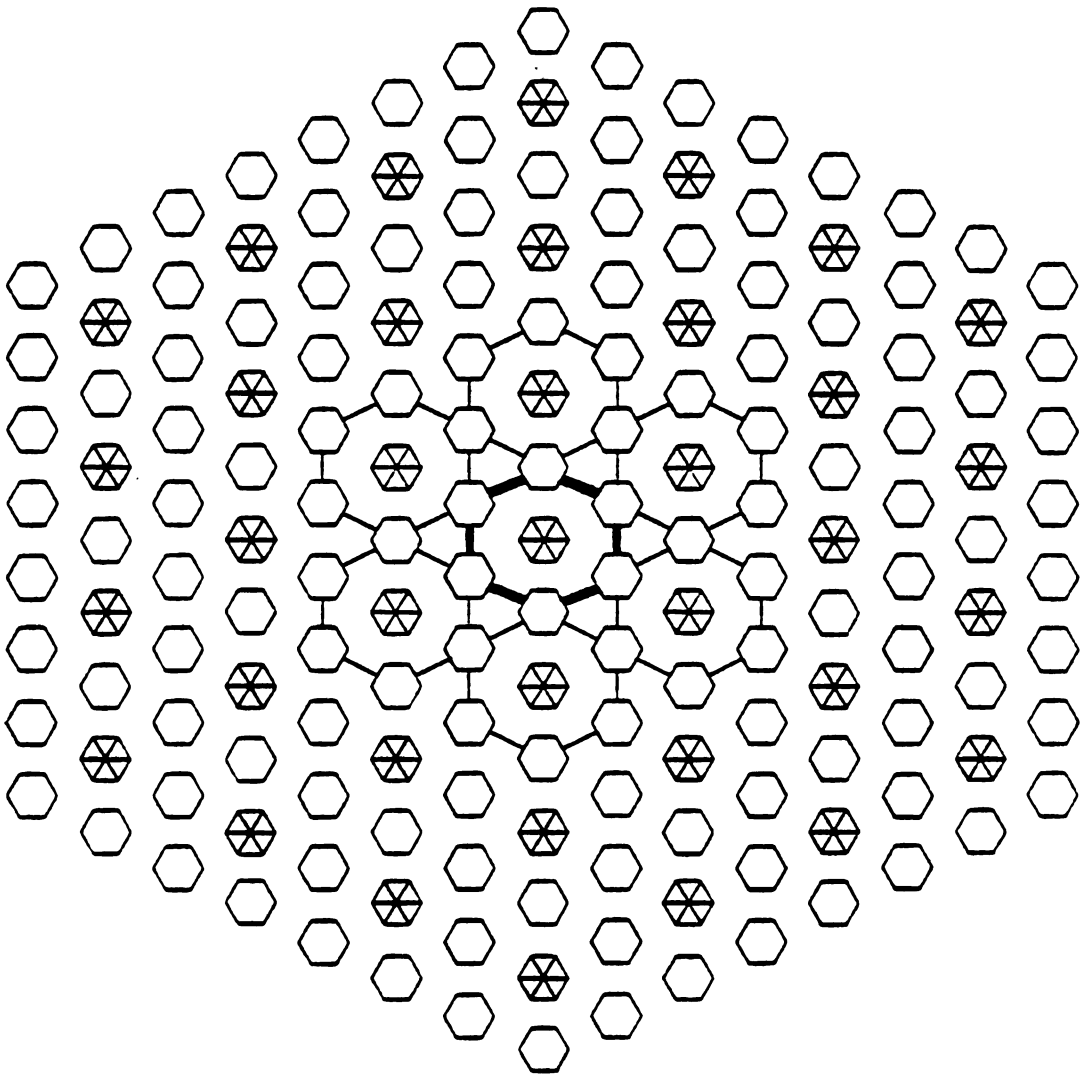


Figure 4.4.(a) A regular hexagonal mixed systolic array constructed from a $\rho_b = 1/7$ basis.

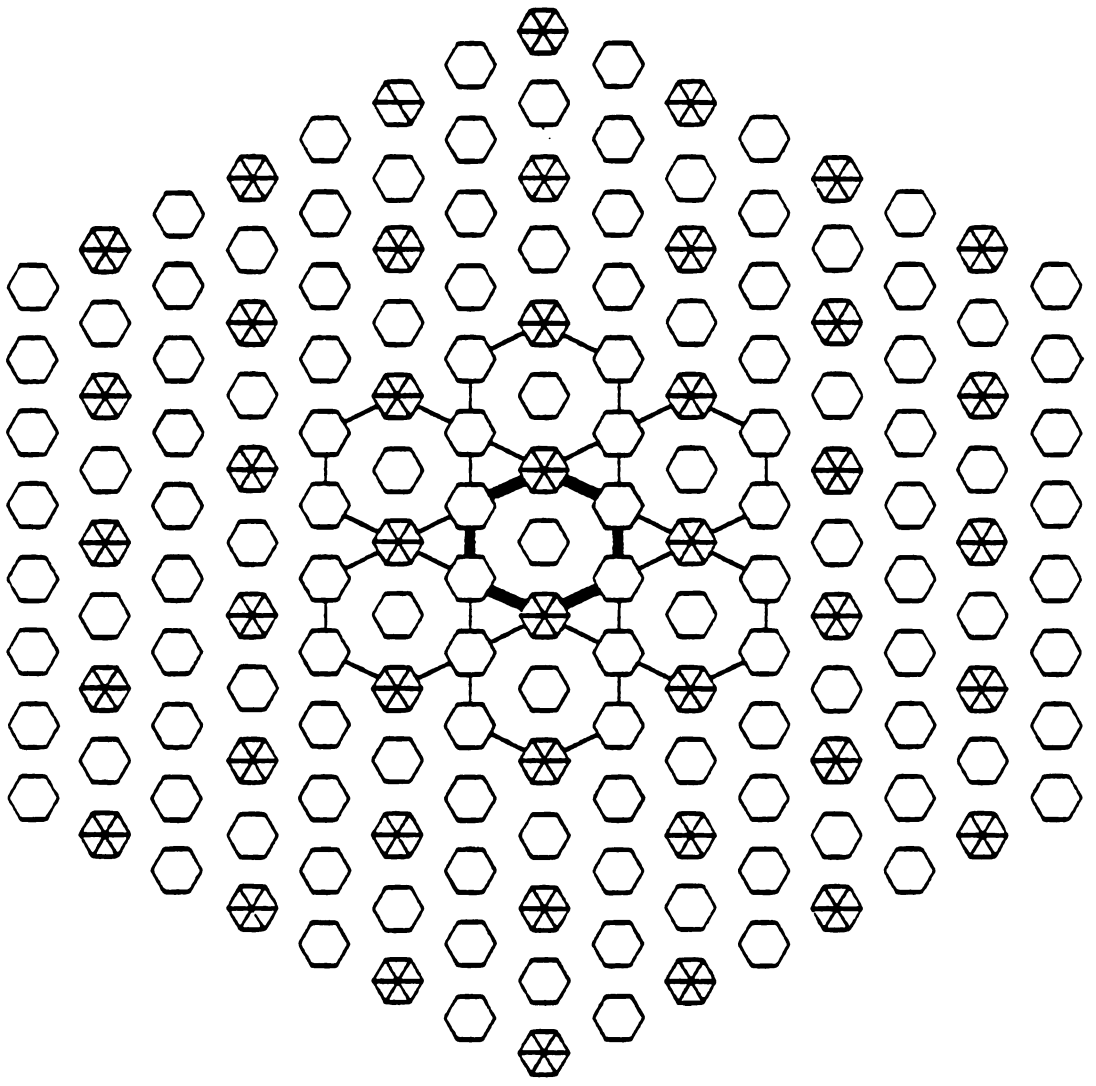


Figure 4.4.(b) A regular hexagonal mixed systolic array constructed from a $\rho_b = 2/7$ basis.

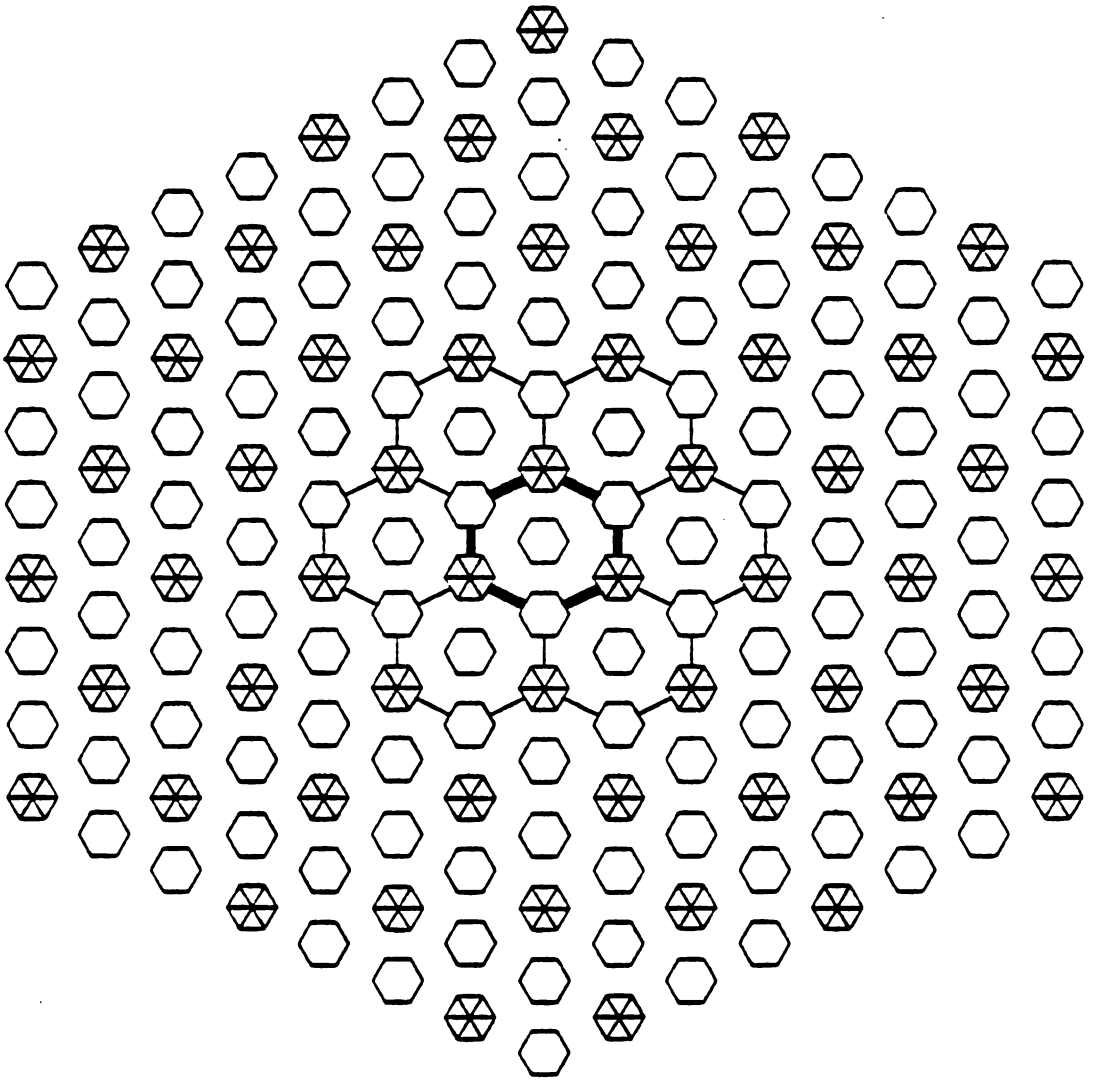


Figure 4.4.(c) A regular hexagonal mixed systolic array constructed from a $\rho_b = 3/7$ basis.

If n is even,

$$N_{CB} = \frac{3(n-1)^2 + 1}{4} \quad .$$

Thus,

$$\rho = \begin{cases} \dfrac{3n^2 + 1}{4(3n^2 - 3n + 1)} & , \text{ n is odd} \\ \dfrac{3(n-1)^2 + 1}{4(3n^2 - 3n + 1)} & , \text{ n is even.} \end{cases}$$

The number of control buffers on the boundary of this type of MSA's is given by

$$N_{CB'} = 0 \qquad , \text{ n is even}$$
$$N_{CB'} = 3(n-1) \qquad , \text{ n is odd.}$$

Thus,

$$\Omega = \begin{cases} \dfrac{12(n-1)}{3n^2 + 1} & , \text{ n is odd} \\ 0 & , \text{ n is even.} \end{cases}$$

Case 2: (MSA's constructed from a $\rho_b = 2/7$ basis)

The number of control buffers in this type of MSA's is given by

$$N_{CB} = 2\,[p + (p+1) + \ldots + (n-1-n(\text{mod } 2))] + (n-1 + (n-1)(\text{mod } 2)),$$

where

$$p = \left\lceil \frac{n + (-1)^{n(\text{mod } 2)}}{2} \right\rceil \quad .$$

If n is an odd number,

$$N_{CB} = 2\left[\frac{n-1}{2} + \left(\frac{n-1}{2} + 1\right) + \ldots + (n-2)\right] + (n-1) = \frac{3n^2 - 4n + 1}{4}.$$

If n is an even number,

$$N_{CB} = 2\left[\frac{n+2}{2} + \frac{n+4}{2} + \ldots + (n-1)\right] + n = \frac{3n^2 - 2n}{4}.$$

Thus,

$$\rho = \begin{cases} \dfrac{3n^2 - 4n + 1}{4(3n^2 - 3n + 1)}, & n \text{ is odd} \\ \dfrac{3n^2 - 2n}{4(3n^2 - 3n + 1)}, & n \text{ is even.} \end{cases}$$

The number of control buffers on the boundary of this type of MSA's is given by

$$N_{CB'} = \begin{cases} (n-1), & \text{if } n \text{ is odd} \\ 2(n-1), & \text{if } n \text{ is even.} \end{cases}$$

Thus,

$$\Omega = \begin{cases} \dfrac{4(n-1)}{3n^2 - 4n + 1}, & n \text{ is odd} \\ \dfrac{8(n-1)}{3n^2 - 2n}, & n \text{ is even.} \end{cases}$$

Case 3: (MSA's constructed for a $\rho_b = 3/7$ basis)

The number of control buffers in this type of MSA's is

given by

$$N_{CB} = \begin{cases} n \times (n-1) + 1, & \text{if } n = 3i-1,\ i = 1,2,\ldots \\ n \times (n-1), & \text{otherwise.} \end{cases}$$

Thus,

$$\rho = \begin{cases} \dfrac{n^2 - n + 1}{3n^2 - 3n + 1}, & \text{if } n=3i-1,\ i=1,2,\ldots \\ \dfrac{n^2 - n}{3n^2 - 3n + 1}, & \text{otherwise.} \end{cases}$$

The number of control buffers on the boundary of this type of MSA's is given by

$$N_{CB'} = \begin{cases} 2n-1, & \text{if } n = 3i-1, \\ 2n-2, & \text{if } n = 3i+1,\ i = 1,2,\ldots \\ 2n-3, & \text{if } n = 3i. \end{cases}$$

Thus,

$$\Omega = \begin{cases} \dfrac{2n-1}{n^2-n+1}, & \text{if } n = 3i-1, \\ \dfrac{2n-2}{n^2-n}, & \text{if } n = 3i+1,\ i = 1,2,\ldots \\ \dfrac{2n-3}{n^2-n}, & \text{if } n = 3i. \end{cases}$$

The mixing density function measures reveal that the hexagonal MSA's (Fig. 4.4) contain roughly from two thirds to three quarters of their constituents as computing elements. Since in hex-connected systolic arrays only a

portion (one third) of the computing elements is active at a time, the presence of control buffers in these MSA's leads to no reduction in the real computing power. The boundary condition function measures show a similar linear decrease in n in these MSA's. So, I/O communication decrease as n^{-1} relative to on-chip communication. Thus, for a proper balance between global data and local data in algorithms executed on such uniformly mixed arrays, global data requirements must also decrease as the inverse of the local data. But this constraint can be overcome by relaxing the uniform mixing requirement.

4.3. Partial Uniform Mixing

The design parameters of the uniformly mixed systolic arrays described in the previous section emphasize balancing the computing power and the local data routing capability inside a given array. However, since the number of I/O pinouts is a major constraint in VLSI implementation [22] and since balancing on-chip processing with I/O is a must on a VLSI chip [35], an MSA should be designed to include the built-in I/O structure as well. Unlike the systolic arrays, in which I/O can take place only at the boundary computing element, in an MSA, I/O is granted to the control buffer. By carefully mixing control buffers into MSA's, various I/O structures can be created. An I/O structure, in general, can be constructed as being either centralized or

decentralized depending on the global data transfer bandwidth. A centralized I/O structure has only one single port, while a decentralized I/O structure has multiple ports.

One important option is mixing a given MSA into two uniform regions, one region for computing and local data routing purposes and the other for I/O or interconnection purposes. Thus a partially regular array is formed. Topologically, a partially regular array has a centralized input or output structure, or both, depending on the specific applications for which it is designed. The region which is mixed for exclusively the I/O structure or interconnection purposes contains control buffers only; that is, the mixing density has a value of one. Since within a hexagonal array structure, the most likely spot to be mixed for the centralized I/O structure is in the central region, we consider only the partially regular array which has a structure shown in Fig. 4.2. Based on the built-in I/O structure (see Fig. 4.5), an MSA can be classified as follows:

(1) CICO (centralized input centralized output). A CICO type of MSA is characterized by having a relatively low I/O bandwidth for its global data. Such an MSA has a fast response time and is very suitable for on-line pipelined database systems [38].

(2) CIDO (centralized input decentralized output). A

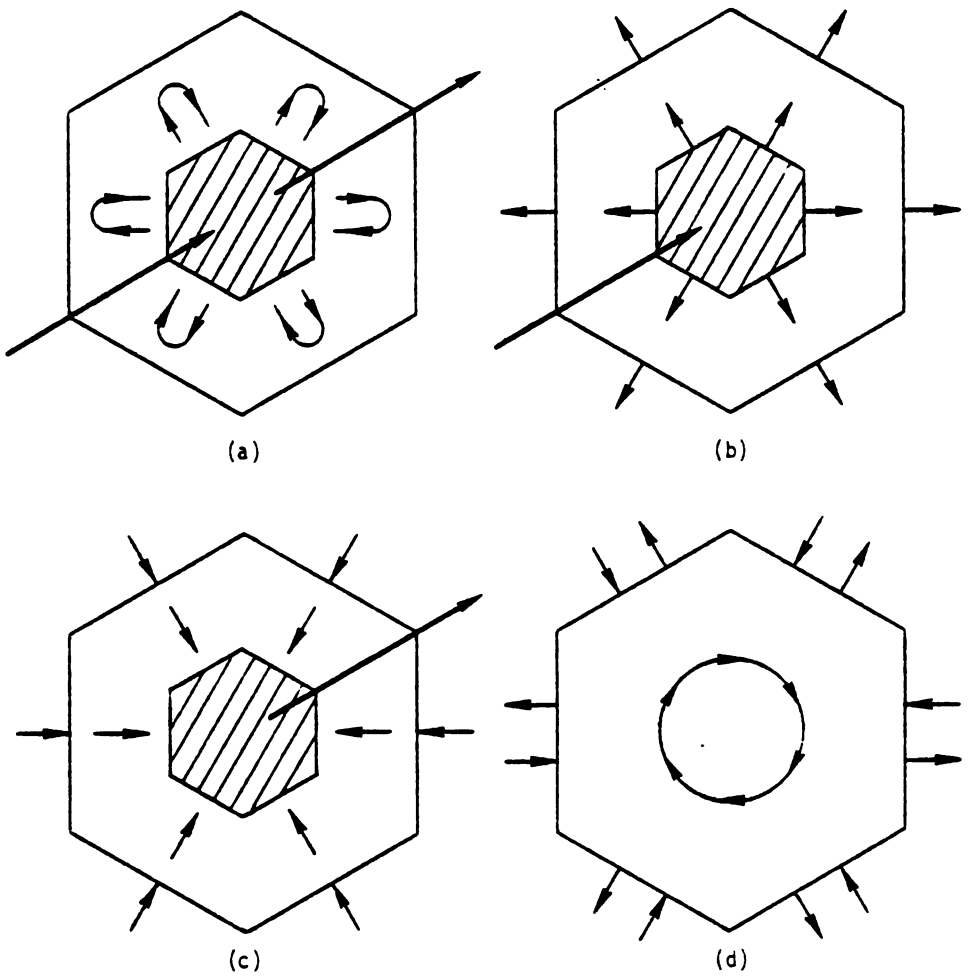


Figure 4.5. Four classes of mixed systolic arrays:

(a) CICO, (b) CIDO, (c) DICO, and (d) DIDO. The arrows indicate the direction of data flow.

CIDO type of MSA is characterized by having a relatively low input bandwidth but a relatively high output bandwidth for its global data. Such an MSA is very useful for centralized massive system control applications or for algorithms which can be decomposed into hierarchical computations [27].

(3) DICO (decentralized input centralized output). A DICO type of MSA is characterized by having a relatively high input bandwidth but a relatively low output bandwidth for its global data. Such an MSA is best used for distributed stream-oriented computing systems [39].

(4) DIDO (decentralized input decentralized output). A DIDO type of MSA is characterized by having a relatively high I/O bandwidth for its global data. Such an MSA will likely be applied to simple two-dimensional pipelined operations [8-9]. This DIDO structure is the classic systolic array architecture.

By decomposing an MSA into two basic regions, the constraint on uniformly mixed systolic arrays can be relaxed since there is now less of a dependence on computing power vs. I/O bandwidth as a function of array edge size. This is so because the boundary condition function, Ω, in a partially regular array can be reduced by a factor of 2 over the uniform array to meet pinout and/or global I/O data requirements.

CHPATER V

DATA-FLOW COMPUTATIONS

Computations are carried out in a data-driven fashion on an MSA. The control buffer is the data-driven master, and the computing element is the slave. The driving mechanism takes place in the control buffer; when the required data are available a data-flow instruction cell is executed on a neighboring computing element [24]. One important consequence of applying data-driven principles is that computations can be uniformly distributed over self-directed computational rings within the MSA. Thus, a computational ring is the basic data-flow computing structure in the MSA.

In this chapter, we focus on data-flow computations on an MSA. First, the store-and-forward control buffer is presented. Next, in Section 5.2, composite cell computations on computational rings are illustrated. This is followed in Section 5.3 by the description of forming composite cells on a data-flow graph. And finally, in Section 5.4, we discuss data flow within the MSA and we also compare the array's performance to the systolic array.

5.1. Store-and-Forward Control Buffers

Fundamentally, the control buffer is designed for store-and-forward purposes. It stores data-flow instruction cells at load time, arbitrates the incoming data and forwards outgoing data at execution time. The basic structure of the control buffer is illustrated in Fig. 5.1. The components consist of an input FIFO, an output buffer, a queueing FIFO, a driven memory, a linking buffer, and a router. The I/O buffers are connected to the I/O ports for I/O operations. The queueing FIFO is connected to a neighboring computing element for discipling executable instruction cells. The linking buffer is used for internetwork purposes. The driven memory is applied exclusively for the purpose of storing data-flow instruction cells. And the router is used to arbitrate the input data to one of the queueing buffer, the output buffer, the linking buffer or the driven memory.

Each data item entering a control buffer is identified by an attached data type header, which specifies a certain operation such as a memory read/write, an output operation, or a by-pass operation, and is guided by an attached address tag which leads the data item to its destinations. When a write operation occurs, the data item is written into the addressed instruction cell within the driven memory of the control buffer; simultaneously, a flag is raised to indicate a condition that the content at that particular cell is ready to be driven. However, when a read

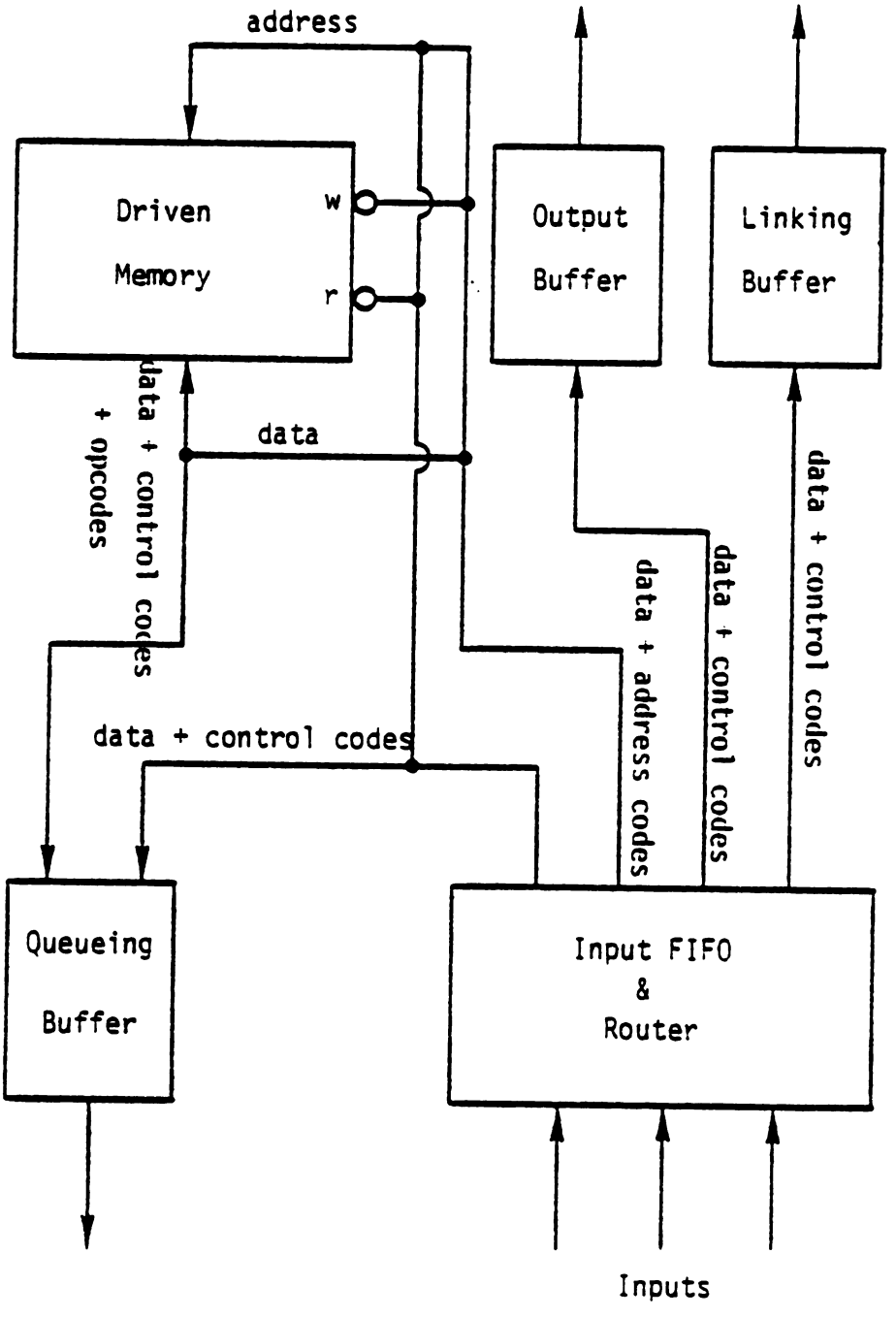


Figure 5.1. The structure of the control buffer.

operation occurs, it is to drive out the content of the accessed instruction cell into the queueing buffer ready for execution on a neighboring computing element. A read operation resulting from driving a not-ready instruction cell is called "aborted." An aborted read operation will be forced to reverse its operation; that is, to become a write operation. An output operation is the result of a terminated computation. The by-pass operation is requested when the data item is to seek no more than a pass through the control buffer and its neighboring computing element.

5.2. Computational Rings

A computational ring is the basic data-flow computing structure in the MSA. It is formed by connecting control buffers and computing elements alternately together into a ring configuration. A computational ring formed with p pairs of control buffers and computing elements is called a p-stage computational ring. Fig. 5.2 shows a typical two-stage computational ring within a hexagonal MSA.

In a computational ring, computations are carried on in a pipelined fashion with the constraint that each succeeding computation requires a data item from its previous computation. In other words, the result of a computation becomes the driving force to the execution of its succeeding computation. For this reason, the "driving operation" is used to describe this computation mechanism. Mathematically, such computations may be represented by a composite

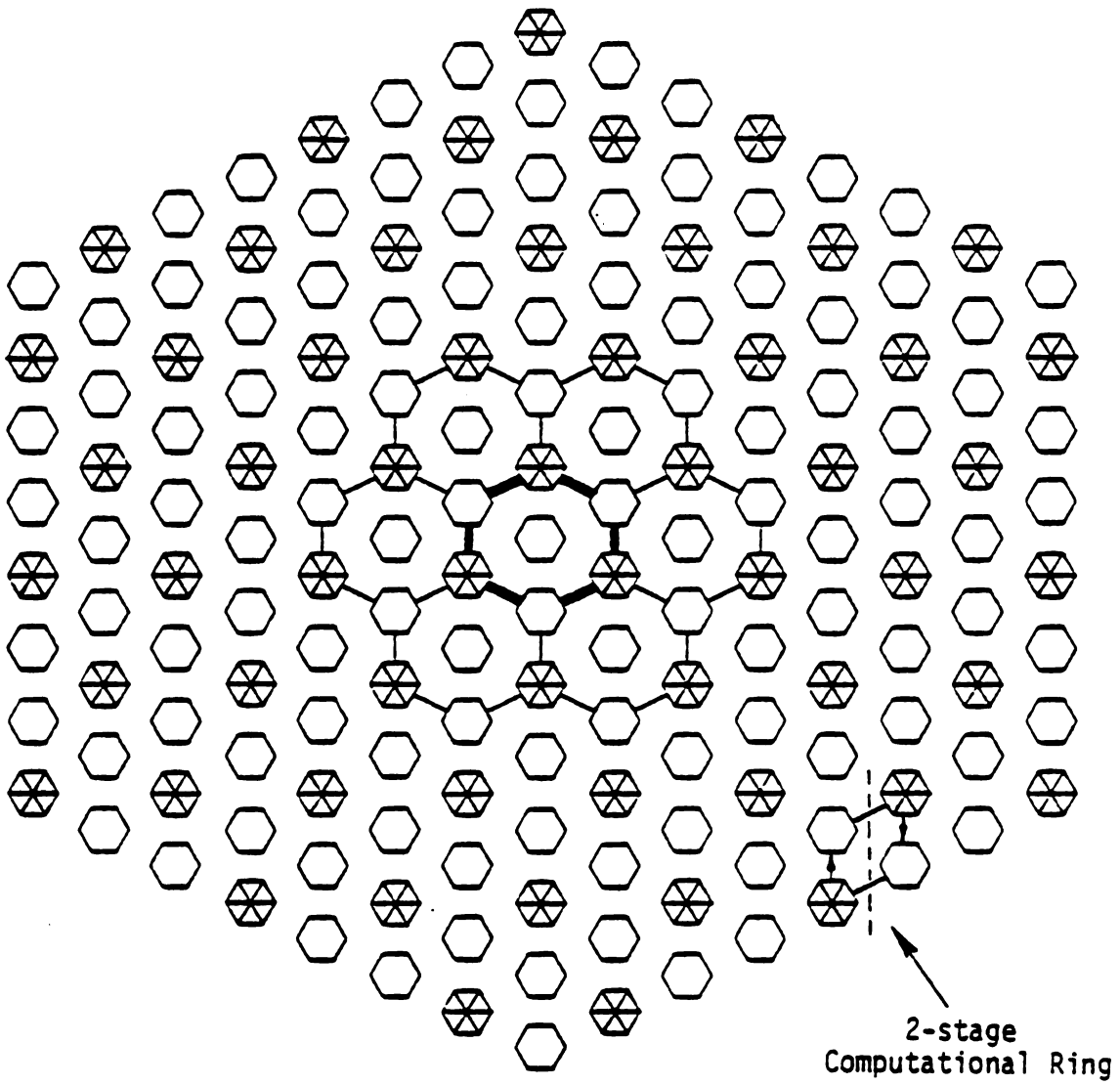


Figure 5.2. A typical two-stage computational ring within a hexagonal MSA.

expression of a form

$$f_\ell(a_\ell,\ldots, f_i(a_i,\ldots, f_2(a_2, f_1(a_1, a_0))\ldots), \; 0 \leq i \leq \ell$$

in which the f_i's denote either the elementary arithmetic function (+,-,x,/) or the results and the a_i's are the data. In data-flow formation, these composite computations can be compiled into a composite cell of data-flow instructions as

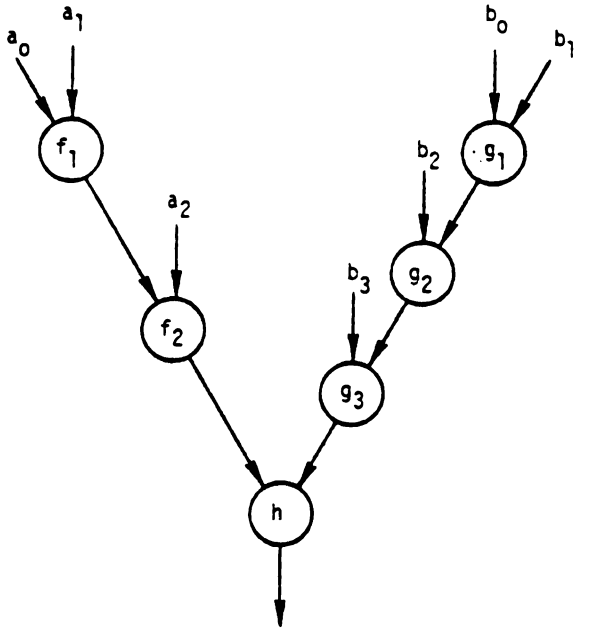
$$((f_1|a_1,a_0), (f_2|a_2), \ldots (f_\ell|a_\ell))$$

When executed, this composite cell proceeds as

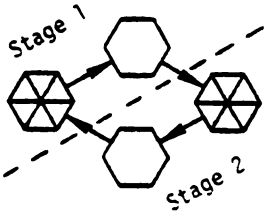
```
Begin (Composite Cell Computation)
  f1 := f(a1, a0);
  For i := 2 to ℓ Do
       Begin
       If Flag (Ai-1) = 1   Then
       ai <-- M (Ai-1);
       fi := f (fi-1, ai)  Else
       M(Ai-1) <-- fi-1;
       Flag (Ai-1) := 1;
       fi-1 <-- M (Ai-1);
       fi := f (fi-1, ai)
       End
End (Composite Cell Computation)
```

where M denotes a driven memory operation and A_i's represent the address tags associated to the a_i's or the results of f_i's. Obviously, the continuity of composite computations depends on the availability of the data a_i's. When the data are not available, the computing element may stand idle.

In performance, a p-stage computational ring resembles a p-stage asynchronous circular pipeline. The delay in the control buffer is referred to as that of latching operations while the time taken in the computing element is considered as a stage delay in the pipeline. With execution of a composite cell on a stage-by-stage basis, p composite cells can be put into execution simultaneously on a p-stage computational ring. Fig. 5.3 illustrates this concurrent pipelining operation through carrying out a data-flow directed graph on a two-stage computational ring. Initially, two sets of data b_0, b_1, b_3, a_2 and a_0, a_1, b_2 are loaded into CB(1) and CB(2). Immediately upon the firing, instruction cells $g_1(b_0,b_1)$ and $f_1(a_0,a_1)$ are executed on CE(1) and CE(2), respectively. The results of f_1 and g_1 are then to drive out a_2 and b_2 for a new phase of data-flow computations with $f_2(a_2,f_1)$ on CE(1) and $g_2(b_2,g_1)$ on CE(2). Following this, f_2 is written into CB(2) while g_2 is to drive out b_3 for a third round of computation on CE(1). And one step later, g_3 is to drive out f_2 from CB(2), and $h(f_2,g_3)$ is executed on CE(2).



(a)



(b)

Function	CB	CE	CB	CE	CB	CE	CB	CE
Stage 1 CB(1) + CE(1)	b_0, b_1 b_3, a_2	$g_1(b_0, b_1)$	b_3 a_2	$f_2(a_2, f_1)$	b_3	$g_3(b_3, g_2)$		Idle
Stage 2 CB(2) + CE(2)	a_0, a_1 b_2	$f_1(a_0, a_1)$	b_2	$g_2(b_2, g_1)$	f_2	Idle	f_2	$h(f_2, g_3)$

(c)

Figure 5.3.(a) A data-flow directed graph example.

(b) A two-stage computational ring.

(c) The Gantt chart illustrating the execution of (a) on (b).

As can be seen, four computational steps are taken and the only transfer times taken are mainly from the memory driving operations, each of which consumes a memory cycle time. The significance of composite cell execution on a computational ring is that each instruction is locally executed. As a result, communication overhead is reduced to a minimum.

Apparently, the purpose of forming computational rings within an MSA is to support locality of computation and data transfer. But this is not all. On a p-stage computational ring, control buffers are linked like a p-stage pipeline. This allows data and control codes to be pipelined into these buffers when the MSA is required for loading. A savings of a factor of p in the number of I/O pinouts with the MSA is expected.

5.3. Loading and Scheduling of Composite Cells

As in the pipelining operation, to keep the computational ring full is essential to its performance. Returning to our previous example (Fig. 5.3), notice that each CE(1) and CE(2) is idle once during four computational steps. It would be more practical if a new composite cell could be initiated immediately for execution when a discontinuity of a driving operation has occurred. This could be achieved by reasonably increasing the queueing buffers size. However, two issues arise: the first relates to how a data-flow graph can be transformed into a scheduled set of

composite cells; the second is concerned with how these scheduled cells should be loaded on a particular computational ring. To deal with these two issues, transformation of a data-flow graph to a simplified time/space grid representation is applied. In this grid representation, a data-flow operation is represented by a darkened circle at a cross point and a data-flow link is represented by either a vertical arc or a sloped arc. In such a representation, the vertical axes are marked with the computational steps and the horizontal axes are measured by the degree of concurrency. Each computational step represents a stage delay on the computational ring and is assumed to be a constant. Concurrent data-flow operations are drawn on different vertical axes but on a single horizontal axis. Fig. 5.4 shows a time/space grid representation for a data-flow graph example.

Basically, a composite cell can be formed by grouping the circles along a particular vertical line or by grouping the circles on several different lines. Fig. 5.5 illustrates how particular composite cells can be formed by grouping the circles in the grid representation of Fig. 5.4. A composite cell which is formed by grouping those circles on a critical path in the grid representation is called the critical-path composite cell.

Def: Let $C_i = ((f_1|a_1), (f_2|a_2),...)$ be a critical-path composite cell in a grid representation and let $L(C_i)$ be the length or the number of computational

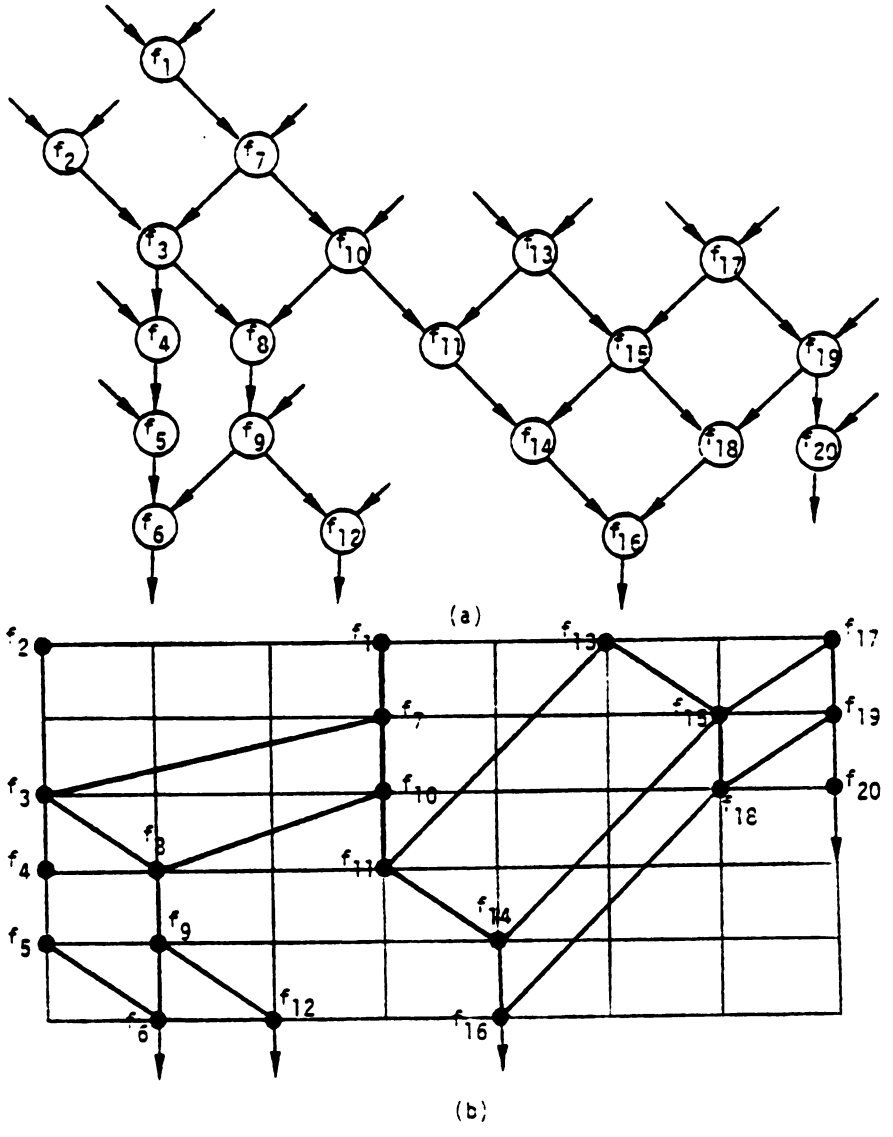


Figure 5.4. (a) A data-flow directed graph example.
(b) The simplified time/space grid representation for (a)

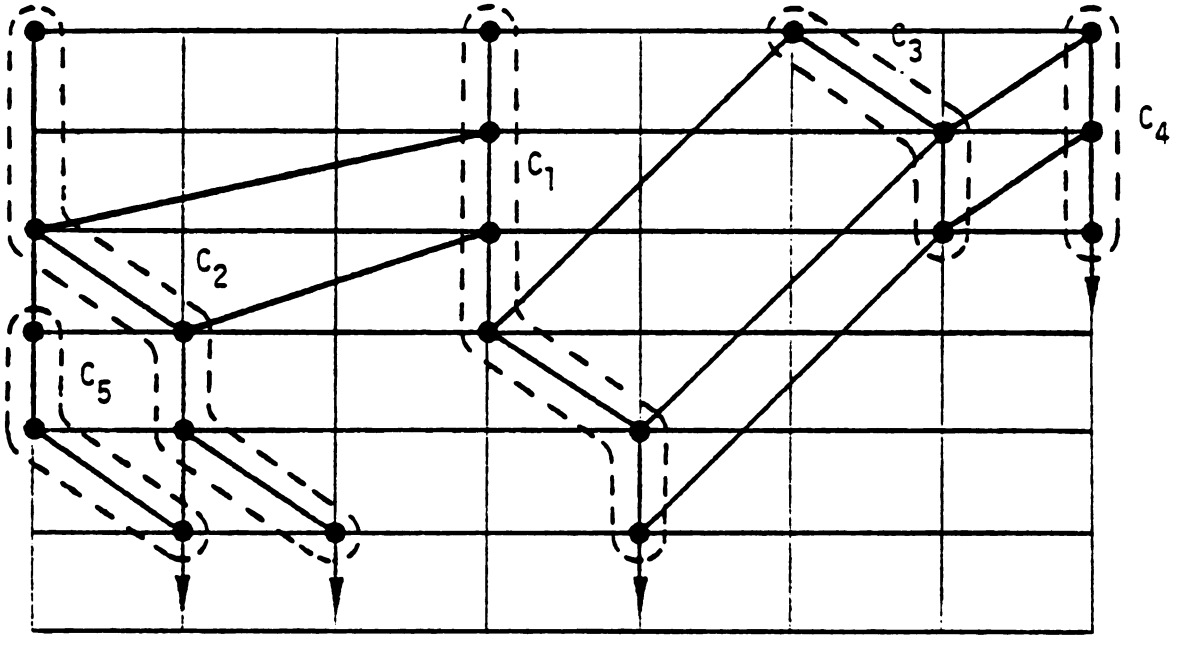


Figure 5.5. An example of vertical grouping of composite cells on a grid representation graph.

steps in C_i, then $L(C_i) \geq L(C_j)$, for every composite cell C_j starting at $(f_1|a_1)$.

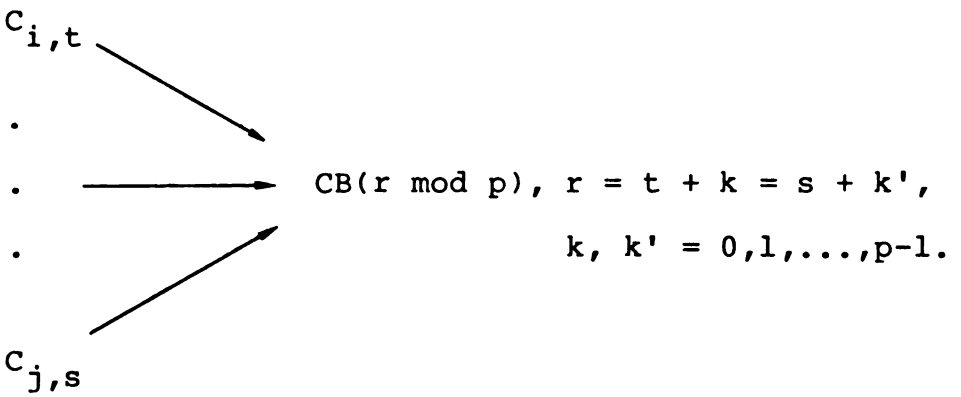
A grid representation may consist of a number of critical-path composite cells. C_1 and C_2 in Fig. 5.5 are two such cells, each of which contains six computational steps. If we remove critical-path composite cells from a grid representation, the remaining grid is called a subgrid. Critical-path composite cells formed in a subgrid contain fewer computational steps than those in the original grid. Again if we remove those critical-path composite cells from a subgrid, another subgrid remains. Through recursively grouping critical-path composite cells and then removing them from a grid or a subgrid, one can obtain an ordered set of composite cells with decreased computational steps. It is easy to see that this conquer-and-divide decomposition scheme results in a minimum number of composite cells in a given grid representation.

Scheduling composite cells for execution on a p-stage computational ring can be considered as a mapping problem. For a single composite cell, $C_i = (C_{i,1}, C_{i,2},...)$ with $C_{i,t} = (f_t|a_t)$, the scheduling is simply a one-dimensional one-to-one mapping as

$$C_{it} \text{ --> } CB((t+k) \bmod p),\ k=0,1,...,p-1$$

where the offset index k refers to the k_{th} stage in the computational ring on which the composite cell's first

computation $C_{i,l}$ is started. However, when more than one composite cell is to be scheduled, we may refer to a multiple-to-one mapping



Computations which are scheduled on a multiple-to-one mapping are mainly of interaction types such as a fork or a join. A fork activates concurrent driving operations while a join implements data-flow firing upon two interacted composite cells. By grouping circles which represent data-flow operations either from forks or to joins, the scheduling problem can be extended to a two-dimensional mapping. In Fig. 5.6, we illustrate the horizontal mapping on the time/space grid representation example of Fig. 5.4.

5.4. Data-Driven Computations

By considering an MSA as a computing network of m tightly-linked p-stage computational rings (an example, see Fig. 4.2, which has m=6, p=3), an MSA is capable of executing p x m data-flow instructions in one computational step. The "block-driven principle" is applied to describe this

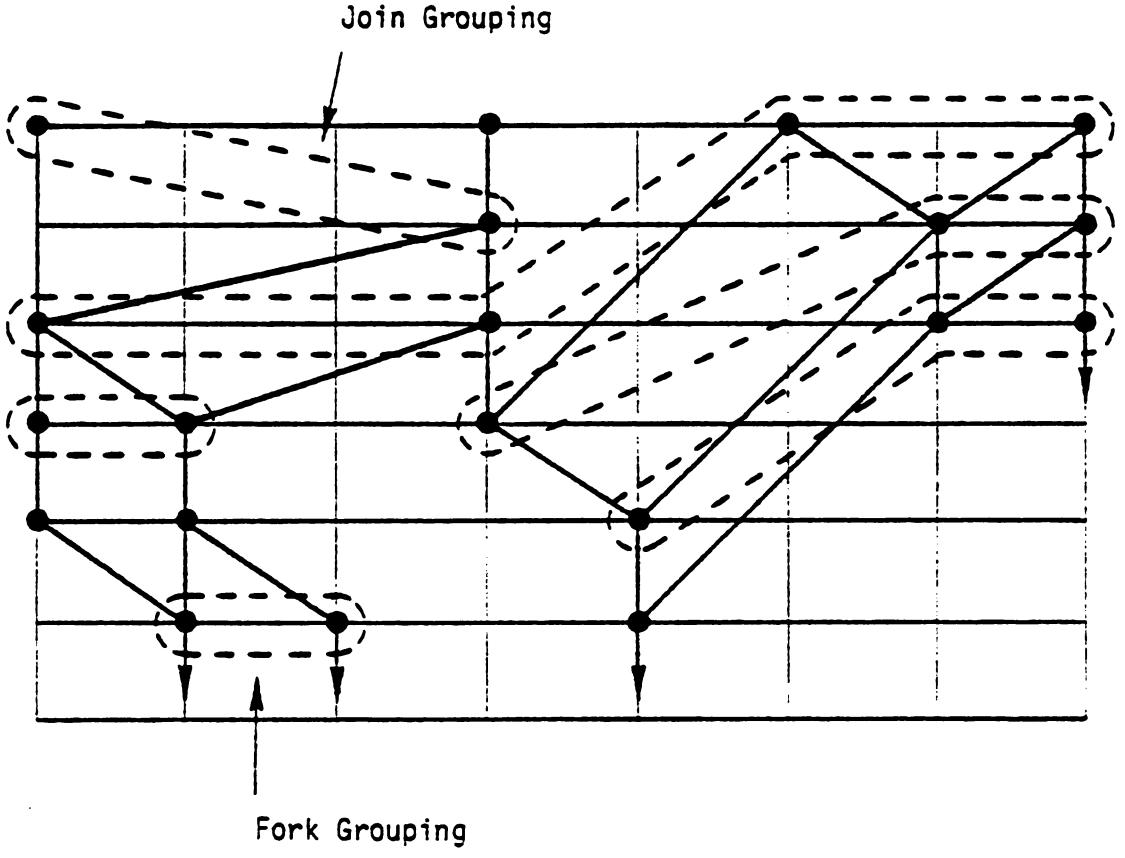


Figure 5.6. An example of horizontal grouping .

phenomenon, which is the firing of a computational block consisting of an arbitrarily large number of composite cells [29].

In data-flow notation, a computational block, B, is represented by an acyclic directed graph of size m by d, where m is the number of computational rings within the mixed systolic array, and d is the computation depth or the critical path in B. When the computation depth d is constant, the computational block has fixed size; when it is nonconstant, the computational block has a varied size. Through data-flow decomposition methodology, any applicative data-flow program can be structured as a sequence of computational blocks (B_1, B_2,...,B_k). Each block transforms an input data stream into an output data stream [40-41], with $B_i = m \times d_i$, where d_i is the computation depth in B_i, $1 \leq i \leq k$. Any acyclic directed graph of data-flow computations can be represented by a data record and a collection of composite cells, which are specified by a frame of microprogramming control codes. Therefore, a computational block B_i can be functionally specified by

$$\{F_i | D_i\},$$

where F_i denotes the particular frame associated with B_i and D_i denotes the size of the data record.

The performance based on block-driven computations is

measured by the average computational rate, $\bar{R}$,

$$\bar{R} = K^{-1} \sum_{i=1}^{k} R_i, \tag{5.1}$$

where R_i is the frame computational rate for B_i. Let

N_i --the number of computations in B_i,

α_i --the frame set-up time for $\{F_i | D_i\}$,

d_i --the critical path in B_i,

β_i --the communication overhead,

T --the execution time for a single computation or the delay time in a control buffer,

h --the word-length to the pinout number ratio, and

$\bar{f}(m)$--the average transfer overhead per computation in T .

Then

$$R_i = \frac{pN_i}{\alpha_i + \beta_i + d_i T} \tag{5.2}$$

with

$$m^{-1} N_i \leq d_i \leq N_i \tag{5.3a}$$

$$d_i T \leq \beta_i \leq d_i T + \bar{f}(m) d_i T \tag{5.3b}$$

$$\frac{hD_i T}{m} \leq \alpha_i \leq hD_i T \tag{5.3c}$$

Therefore, for a uniform computational block ($d_i = N_i m^{-1}$) being executed on an m-input DICO type of MSA, R_i becomes

$$R_i = \frac{pN_i}{\frac{N_i T}{m} + [1 + \bar{f}(m)]\frac{N_i T}{m} + \frac{hD_i T}{m}}$$

$$= \frac{pm}{[2 + \bar{f}(m) + \frac{hD_i}{N_i}]T} \qquad (5.4)$$

With block-driven computations, the size of the data record to be loaded into the MSA (i.e., the input global data) is usually much less than the total number of computations (i.e., amount of local data); therefore, we assume

$$\frac{hD_i}{N_i} << 1.$$

Equation (5.4) thus reduces to

$$R_i = \frac{pm}{[2 + \bar{f}(m)]T} \quad , \qquad (5.5)$$

which is quite similar to the performance of the benchmark hexagonal systolic array since it has a constant computational rate of $(2/3)pmT^{-1}$ [8]. For the CIDO or CICO type of MSA's, the frame set-up factor is $mhD_iN_i^{-1}$, which may become the bottleneck to the system. As can be seen, the average transfer overhead coefficient, $\bar{f}(m)$, plays a significant role in the overall performance measure. The decreased computational rate with the MSA, when compared with the systolic array, represents the primary cost for achieving

the flexibility of a reconfigurable multiprocessor structure that is capable of implementing more than a single algorithm.

CHAPTER VI

THE HOURGLASS MACHINE

A data-flow machine, based on a modified hourglass computing model, is presented in this chapter. The hourglass machine is loaded with a pair of MSA's, with a DICO type of MSA at one end and a CIDO at the other. Functionally, the mixed systolic array with a DICO structure is intended for block-driven computations, while the CIDO type is used for storing data-flow instruction cells and data-flow firing. Objectively, the structure of this machine is not designed for a specific application; rather it is intended to illustrate the architectural potential of mixed systolic arrays and the hierarchy of data transfers resulting from block-driven computations.

This chapter is organized as follows: First, the hourglass computing model is introduced in Section 6.1. Next, Section 6.2 describes the functional structures of this data-flow hourglass machine. In Section 6.3, we present a two-phase data transfer mechanism. This is followed in Section 6.4 by the analysis of performances measured on this particular machine. And, finally, Section 6.5 dis-

cusses some premises of this machine regarding VLSI implementation.

6.1. Modified Hourglass Computing Model

The modified hourglass computing model takes its name from the concept of two-dimensional data moving in an hourglass. The hourglass model has a data-moving end and a data-receiving end with a narrow passage in between. The data-moving end is loaded with a block of data-flow instructions. And the execution of these data-flow instructions can be considered as the moving of data, while the data-receiving end collects the computational results. Within the hourglass, a data item moves from point to point with the guidance of an appended control code. This control code contains a data type header and a destination address field. The data type guides the data item either into transmission paths or into reflection paths (see Fig. 6.1). The length of the transmission paths is fixed; therefore, there is no preference for all global data transfers. The reflection paths vary, ranging from the shortest paths localized on a computational ring to the logarithmic paths. The destination address specifies the particular location the data item is headed for. The hourglass computing model, as characterized by its structure, provides highly concurrent activities at both ends which gradually decrease toward the narrow passage.

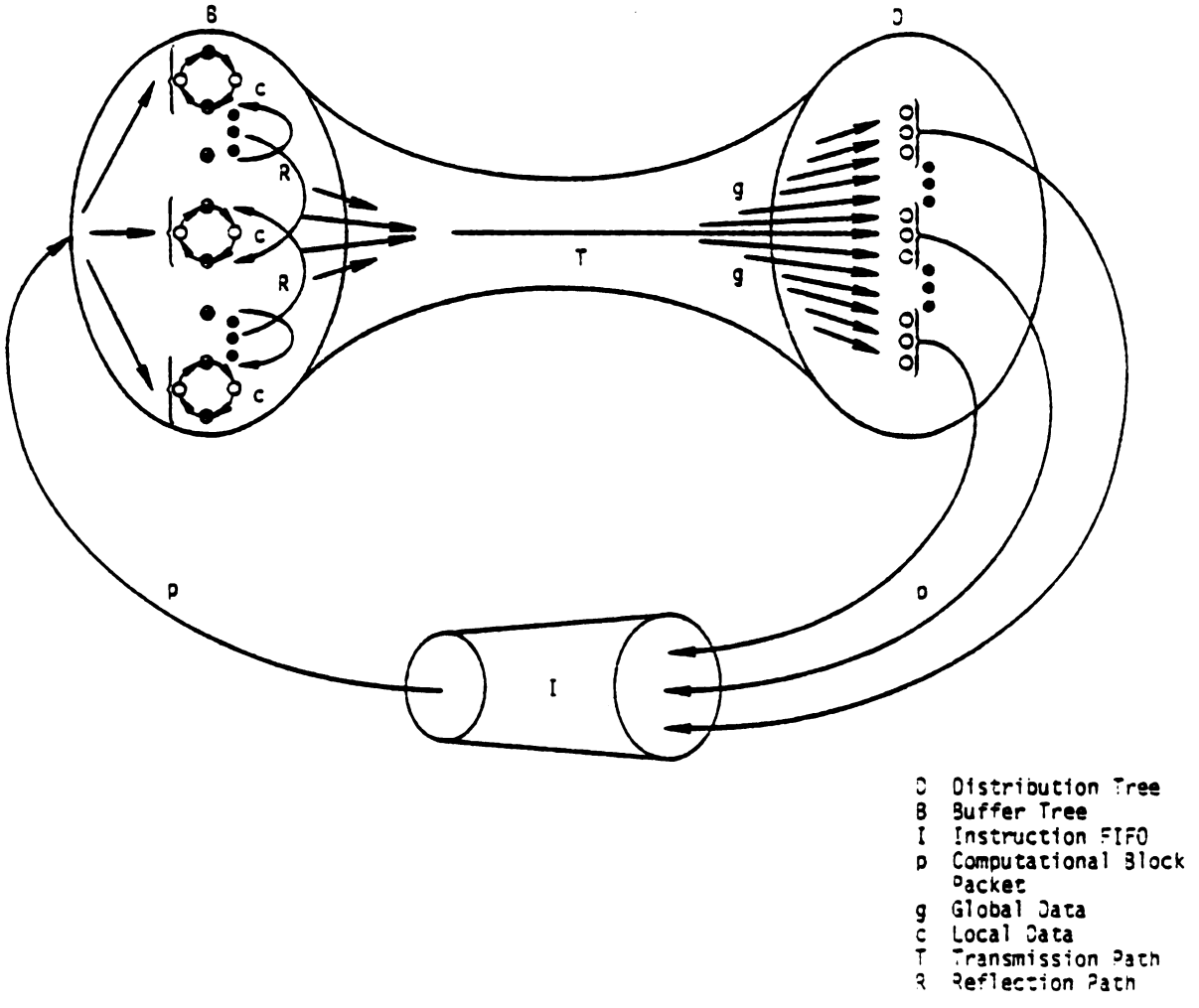


Figure 6.1. The modified hourglass computing model.

6.2. Hourglass Tree Machine

As it was proposed, the novel hourglass computing model aimed at exploring as much as possible the computation locality as well as communication locality in block-driven computations. Based on this model, we synthesize a data-flow tree machine, or an hourglass tree machine. As shown in Fig. 6.2, this particular tree machine contains primarily a pair of "mirrored trees", one is called the buffer tree and the other is the distribution tree. In this section, we describe the structure of these two trees and the functional units associated with the tree machine.

Buffer Tree--The buffer tree is a DICO type of MSA, containing m tightly-linked p-stage computational rings along with a tree linking structure (an example, see Fig. 6.3, a five-level linking structure). It is capable of computing (m x p) FLP operations concurrently and transferring the results in logarithmic unit time. In the buffer tree, there are (m-2) nodes and one root node. Each one of these nodes is a control buffer, at which arriving data will either be buffered down or be forwarded out. Associated with each leaf node is the p-stage computational ring on which composite data-flow computations are performed. Besides functioning as the computing power, the buffer tree plays two other major roles: first, it is an interconnection network for the m computational rings; second, it is a buffering channel between the processing power and the instruction memory cell.

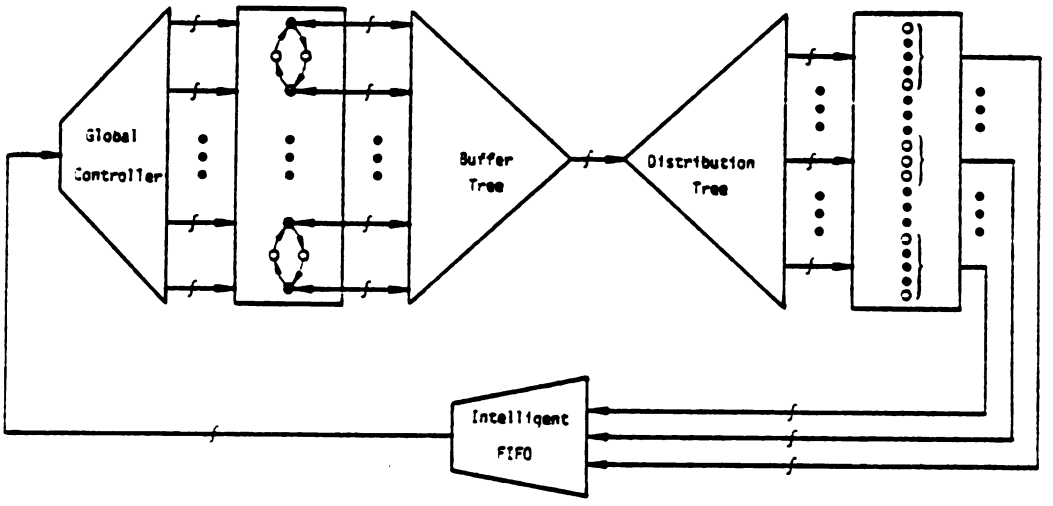


Figure 6.2. The structure of the data-flow hourglass computing machine.

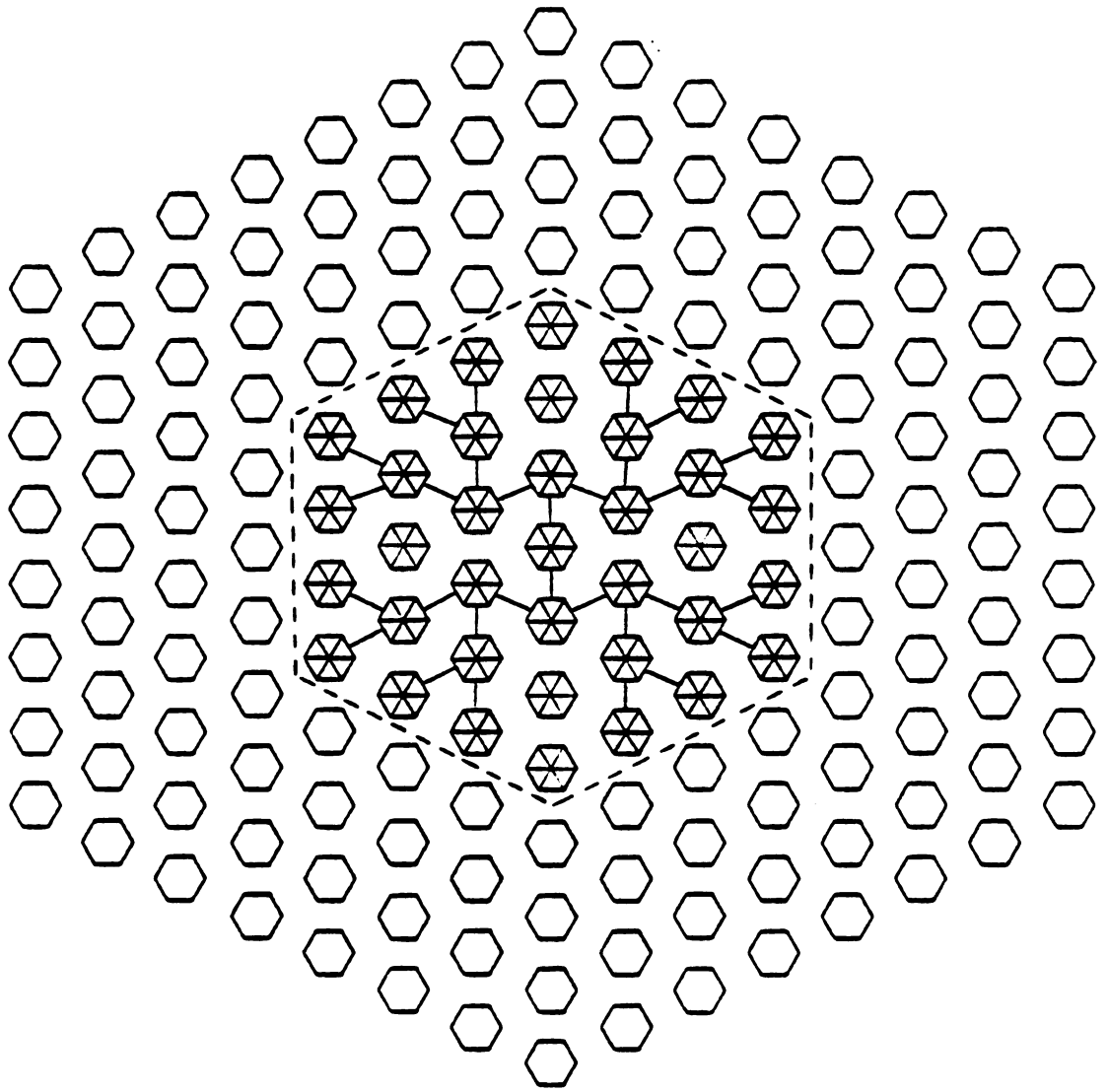


Figure 6.3. A five-level linking structure in the buffer tree.

Distribution Tree--The distribution tree is an n-level pipelined binary switching network of CIDO type. It provides the basic mechanism for routing or multicasting streams of global data to a set of data-flow computational blocks; and at the same time, supervises the firing of computational blocks. There are 2^{n-1} computational blocks of data-flow instruction cells tied to the lowest-level leaf nodes of the tree. When a data item enters the root node, it is pipelined through the n-level distribution tree based on the appended control code. The address field in the control code contains two parts: a major address and a minor address. The major address specifies what computational blocks the data item is headed for while the minor address selects the location of the destination instruction cell in the specified computational block. For providing proper firing, in each of the 2^{n-1} computational blocks there is a decremented count. The decremented count is initially set to a number indicating the amount of dependent data required for block firing. When a data item is written into an instruction cell in that block the decremented count is reduced by one. And when the count reaches zero, the computational block is fired and execution commences.

Global Controller--The global controller plays two major roles: first, it distributes composite cells of data-flow instructions over m computational rings at load time; second, it monitors the status of each of the m

computational rings at execution time. Within the global controller, there is both a ring count and a decoder array. The ring count is initially set to m. Each time the controller receives an acknowledge signal from a released computational ring, the ring count is reduced by one. As the ring count reaches zero, the global controller starts fetching a new computational block of data-flow composite cells from the Data-Flow Intelligent FIFO if they are available. The computational block is then decoded on the decoder array and the resulting composite cells are distributed over m computational rings for execution.

Intelligent FIFO--The intelligent first-in-first-out unit is used for queueing the computational blocks which are fired and ready for execution. Due to the irregular number of data-flow instructions contained in various computational blocks, each FIFO is designed to manage a fixed number of data-flow instructions.

6.3. Two-Phase Data Transfer Mechanism

Because there are two types of data to be transferred in the same buffer tree network, each node of the buffer tree, a control buffer, is designed to work on a two-phase basis. In the push mode, data are pushed forward from the lower-level nodes to the higher-level nodes; whereas, in the pull mode, data are pulled backward in an opposite way. Each data item is tagged with a destination address field and a one-bit data type header. The width of the address is

determined by the tree height--the higher the tree height, the wider the field. Specifically, a locally dependent data item has a relative displacement address and a one-bit direction header, while a globally independent data item has an absolute address. The relative displacement address is determined by the distance in which the two communicating leaf nodes are apart and by the relative position in which the two nodes are located (see Fig. 6.4). Data to be pushed forward or pulled backward depend on a one-bit mode control by ORing the one-bit data type header and selected bits in the address field. With a tree height of m, this mode control at the i_{th} level, M_i, is given by

$$M_i = a_0 U \left(\bigcup_{k>i}^{m-1} a_k \right), \quad 0 \leq i \leq m-1 \tag{6.1}$$

where the notation U stands for the logic OR operation and a_i are the binary value in the address. If the mode control is "1", data will be pushed forward; otherwise, they will be buffered at the node at which they last visited and be ready to be pulled backward.

Global data, which carry a "1" in the data type header, a_0, will allow themselves to be pushed forward through the buffer network. However, local data which carry a "0" in the data type header can never be pushed forward beyond the root level, because the mode control at the root level is always "0" for these data. Data which have already been buffered down to a node at some level will be pulled

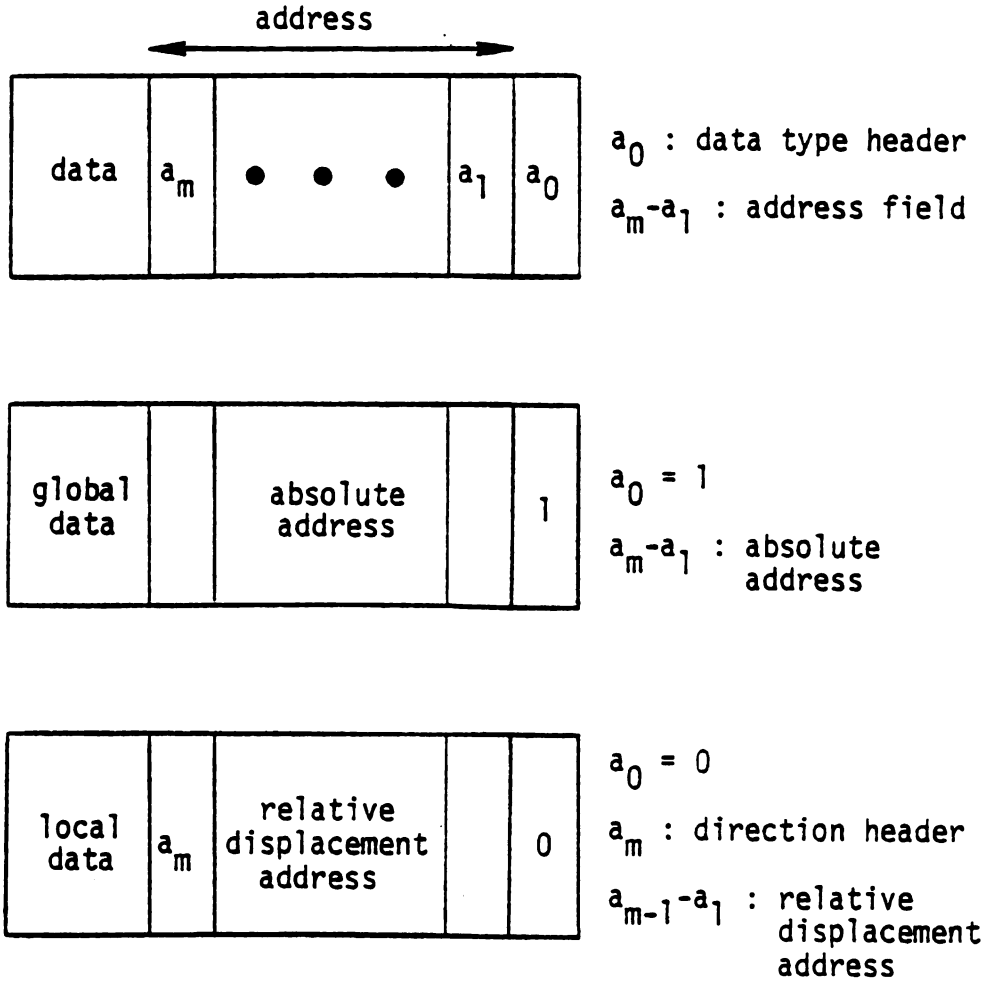


Figure 6.4. The data address syntax.

backward by one of the two son nodes. The decision is determined by a left-right control, LR_i, at that level, with

$$LR_i = a_m a_i + \bar{a}_m \bar{a}_i, \quad 1 \leq i \leq m-1 \tag{6.2}$$

where a_n is the direction header which determines whether the data to be directed to their right or to their left. If the left-right control is "1", the data will be pulled backward by the right son node, and if it is "0", they will be pulled backward by the left son node.

6.4. Hourglass Machine Performance Measures

The modified hourglass computing machine is characterized by its distributed computing structure and its hierarchical communication geometry. The distributed computing structure, as described in Section 5.4, provides a potential computing power of p x m. And utilization of this computing power provides a direct measure on the hourglass machine's computational rate. The hierarchical communication geometry is integrated with a linear intra-ring data path structure and a logarithmic inter-ring data path structure with a combined data transfer bandwidth of (p+1)m-1 of which p x m is the intra-ring bandwidth and m-1 is the inter-ring bandwidth. Since there exists a considerable difference in the bandwidth as well as the data transfer times between these two types of data transfers, the

ratio of the data transfer rate (the number of data transfers in a given computational block) between these two types will have a great impact on the hourglass machine's overall performance.

Let u_i be the data transfer rate corresponding to a data path of i unit times and let

$$V = \sum_{i=1}^{\log_2 m} u_i$$

then the average transfer overhead $\bar{f}(m)$ in Equation 5.4 can be evaluated as

$$\bar{f}(m) = \frac{\sum_{i=2}^{\log_2 m} i \times u_i}{V} . \qquad (6.3)$$

Here we assume each intra-ring transfer is merely a memory driving operation and is taken in a unit time. This can be accomplished with the construction of fast pipelined data paths within the computational rings. In the case that each computational ring has an equal number of inter-ring data transfers with every other ring, u_i can be expressed as

$$u_i = \frac{V - u_1}{(\log_2 m - 1)}, \quad i = 2, 3, \ldots, \log_2 m \qquad (6.4)$$

Thus, $\bar{f}(m)$ becomes

$$\bar{f}(m) = \frac{(V - u_1)V^{-1}}{(\log_2 m - 1)} \sum_{i=2}^{\log_2 m} i$$

$$= \frac{V - u_1}{V \log_2(2^{-1}m)} (2 + 3 + \ldots + \log_2 m)$$

$$= \frac{V - u_1}{2V} (\log_2 m + 2)$$

$$= \frac{\xi}{2} (\log_2 m + 2) \qquad (6.5)$$

where ξ is the ratio of the inter-ring data transfer rate to the intra-ring transfer rate. Algorithms which support uniform inter-ring data transfers show a low degree of locality. However, the data flow of these algorithms usually has a very regular and simple pattern.

In another case, each computational ring interacts in a weighted manner, more frequently with rings on shorter data paths. For example, the data transfer rate, u_i, has an exponentially decreased funciton of

$$u_i = \frac{V - u_1}{2^i} .$$

For such an example, the average transfer overhead, $\bar{f}(m)$, becomes

$$\bar{f}(m) = V^{-1} \sum_{i=2}^{\log_2 m} \frac{(V - u_1)i}{2^i}$$

$$= \frac{V - u_1}{V} \sum_{i=2}^{\log_2 m} \frac{i}{2^i}$$

$$= \xi \left(\frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots + \frac{\log_2 m}{m} \right)$$

$$= \frac{3}{2} \xi \quad , \text{ for } m >> 1. \qquad (6.6)$$

As evaluated, algorithms which support an exponential decay inter-ring data transfer rate show a high degree of locality and have a constant average transfer overhead.

6.5 Implementation Considerations

The feasibility of constructing a large hourglass machine relies on the assumption that a great deal of computing elements and control buffers can be integrated onto a single chip. Two issues which concern us most are the I/O connections issue [8-9] and the on-chip interconnections issue [22]. In its structure, the hourglass machine shows some promising aspects on these two issues. It is observed that (1) the I/O activities take place only at the two ends of the hourglass; the number of I/O pinouts grows only proportionally with the number of computational rings and (2) the regular, simple and short communication geometry inside the hourglass helps reduce substantially a large amount of on-chip interconnections. Yet, the following premises make the hourglass computing machine attractive to us.

(1) The hourglass machine supports asynchronous computations based on data-driven principles. This allows the use of fast asynchronous logic or self-timing logic circuitry which, as being studied, represents a considerable speed-up over synchronous logic [42].

(2) The use of the boundary control buffer as the I/O communication center relaxes some constraints imposed on the systolic array architecture. The most significant one is that the I/O pipelined rate must be kept at the same as that of an inner-product computation. In the hourglass the I/O data can be fed into or pumped out of the hourglass at a rate independent of the computing element's computational speed.

(3) The DICO and CIDO combination inside the hourglass allows different IC technologies to be used. For example, the centralized region is grown from a basis-mixing density of one and this region can be constructed from a fast IC technology such as ECL logic as separated from the other decentralized region. By this approach, the computational results from the DICO tree can be swept across the narrow passage at a much higher rate than the I/O data rate taking place at the two ends of the hourglass.

(4) The high degree of locality in the structure of the hourglass machine promises the use of a local clocking discipline for data transfer. This avoids using a more complex system-wide clock discipline on a VLSI chip.

CHAPTER VII

CONCLUSIONS

7.1. Summary

VLSI defines a technology which promises to provide for the future development of a chip with a circuit density of three orders of magnitude greater than today's state-of-the-art 32-bit microprocessor [2-3]. This promise, coupled with the increased ability to design multiprocessor systems, has spawned an extensive research in computational paradigms that differ from the conventional von Neumann paradigm--the paradigm, which, as pointed out by Meyer [43], has dominated the computer industry in the past two decades. In fact, except for a few commercial machines (e.g., some made by Burroughs Corporation), there have been no significant advances in computer architecture since the 1950's. Some so-called advances that might come to mind (e.g., Cache memories, instruction pipelining, the microprogramming concept) are not really computer architecture advances; they are improvements in the implementation of a particular computer architecture.

It has become increasingly apparent that, even with this upgrading, the von Neumann paradigm does not provide

the computational structure necessary to efficiently implement solutions to many problems [9,10,41]. Kung's systolic array paradigm [8-9], and Dennis's data-flow paradigm [24-28] are both examples of computational alternatives to the von Neumann model--alternatives that promise to be more cost-effective, when implemented, through using VLSI techniques than the von Neumann paradigm.

Data-flow machines are built to take advantage of parallelism inherent in data-flow programs. Systolic arrays are special-purpose, high-performance data-flow structures which are characterized by having a simple, regular and short communication geometry. But a principal drawback to these special-purpose arrays is that they lack flexibility in implementation; a given systolic array only implements one specific algorithm (e.g., matrix multiplication, where the size of the matrices are fixed for a particular array). Another drawback is that an algorithm to be executed efficiently on a systolic array must possess a very simple and regular data-flow pattern. What's more, due to the I/O constraint of pinouts on the chip, it is questionable that the systolic array can be applied broadly as a coprocessor for practical algorithm implementations.

The purpose of this research was to investigate a class of modified systolic array architectures known as mixed systolic arrays which broaden the scope of applications executable on systolic arrays while retaining much of the simplicity and regularity of the systolic array

architecture. Specifically, we investigated a class of array architectures which can be reprogrammed, can be implemented with a reasonable amount of I/O pinouts, can self-execute after the arrays have been loaded, and can implement both asynchronous and synchronous computations. To carry out this goal, our research efforts were based upon four tasks: (1) development of the mixed systolic array architectures, (2) modeling of the mixed systolic array architectures, (3) development of a classification scheme and evaluation model for these array architectures, and (4) development of a sample implementation for these array architectures.

In the development phase, we first focused on the design of a small-scale mixed systolic array or a programmable systolic array in Chapter III. The structure, the components, and the functional behavior of this programmable systolic array were described, and the execution of a second-order recursive equation with nonconstant coefficients on this array was illustrated. It was observed that this programmable systolic array accomplished a dynamic problem-solving capability without degrading its performance as compared to Kung's nonprogrammable systolic array. In Chapter IV, our research next centered upon the development of the more generalized mixed systolic arrays. We applied the mixing profile concept to construct mixed systolic arrays from a building block as a basis. A case study was made by developing mixed systolic arrays from a

number of hexagon bases with different basis-mixing densities. We also identified the mixing density function and the boundary condition function--the two parameters which characterize a mixed systolic array. The mixing density function (Eq. 4.1) suggested that a particular MSA be used for algorithms supporting a complexity ratio between data routing and computing. The boundary condition function (Eq. 4.2) related a particular implementation to the ratio of the global and local data bandwidth requirements of the algorithms for which the MSA was intended. We further developed four different types of mixed systolic arrays, CICO, CIDO, DICO, DIDO, based on a two-region mixing strategy, and specified their potential uses.

In the modeling phase described in Chapter V, our research advanced to studying how mixed systolic arrays functioned. We began by forming a local computational ring within a mixed systolic array. A computational ring was formed to support both local computation and local communication, as well as to execute data-flow composite cells based on data-driven principles. We then developed methods to map a two-dimensional directed graph on a particular computational ring for effective execution. In vertical mapping, data-flow composite cells were formed along critical paths; while in horizontal mapping, the fork type of data-flow instructions and the join type of data-flow instructions were grouped together. By overlapping these two mappings, we were able to load or schedule data-flow

instructions from a directed graph on computational rings for local executions.

In the evaluation phase, we developed a general evaluation model for measuring the performance of the mixed systolic array based on an average load time and an average transfer overhead measure in Equation 5.4. The average load time reflects the I/O structure of a particular MSA implementation, and the average transfer overhead measures the locality in algorithms executed on the mixed systolic array. A case study was made with a DICO type of MSA based on block-driven computations. This study showed that, when most of the computations were uniformly distributed over the computational rings, a DICO type of MSA had a performance similar to a hex-connected systolic array. The necessary locality represents the primary cost for achieving the flexibility of a reconfigurable multiprocessor structure that is capable of implementing more than a single algorithm.

In the sample implementation phase discussed in Chapter VI, we developed a modified hourglass computing model by using a DICO type of MSA and a CIDO type of MSA. The hourglass computing model showed some promising aspects in both VLSI computations and VLSI implementation. Based on block-driven computations, the hourglass model provides a programmable communication path hierarchy for fast data transfer while at the same time achieving very high levels

of concurrency. Performance measures showed that the hourglass had a low constant average transfer overhead for algorithms which support a high degree of locality in data transfer.

7.2. Future Research

The mixed systolic array architecture under investigation here represents a new approach which applies the data-flow concepts to the execution of parallel algorithms on a modified systolic array. It fits nicely into the mainstream of architecture studies reported elsewhere [24-28,34-36]. Moreover, the investigation of this architecture points toward several important areas for additional study.

An investigation could be made into the system design of matching parallel algorithms for a certain type of mixed systolic array, or a combination of different types. With a hierarchical system design facility, one is able to characterize algorithms based on a fixed set of parameters (e.g., computational demands, control structures, and global data vs. local data bandwidth requirements). And through a one-to-one mapping, the matching processes can be carried out in an iterative manner until a prespecified precision level is reached. As an example [44], a class of algorithms can be evaluated in terms of the size of its component modules, the ways in which they can communicate or interact, and how these interactions are controlled. Candidate MSA's are then each evaluated in terms of their relative abilities to

support these algorithms' demands. A second investigation that could be pursued is the development of an on-chip communication model for various MSA's structures. With an on-chip communication model, the communication cost can be evaluated in terms of computing power and queueing size and I/O bandwidth when the algorithms are executed on a supporting MSA architecture. As an example, consider Kung's systolic algorithms on VLSI computations. The argument, "computation is cheap while communication costly", can be evaluated when such a model is available. And, finally, further investigation that could be taken is the on-chip scheduling problem. Since communications are likely to be the dominant cost factor on future VLSI chips, on-chip scheduling methodologies must be developed to minimize these costs. As an example, the scheduling problem discussed in Section 5.3 can be extended to the algorithm or system level so as to minimize the total communication cost.

REFERENCES

1. Keyes, R.W., "The Evolution of Digital Electronics towards VLSI," IEEE Journal of Solid-State Circuits, Vol. Sc-14, No. 2 (April 1979), pp. 193-201.

2. Mead, C.A., and Rem, M., "Cost and Performance of VLSI Computing Structures," Technical Report 1584, Calif. Institute of Technology Computer Science Dept. (1978).

3. Johnson, R.C., "32-Bit Microprocessors Inherit Mainframe Features," Electronics, Vol. 54, No. 4 (Feb. 1981), pp. 138-140.

4. Patterson, D.A., and Sequin, C.H., "Design Considerations for Single-Chip Computers of the Future," IEEE Trans. on Computers, Vol. C-29, No. 2 (Feb. 1980), pp. 108-115.

5. Chang, T.L., and Fisher, P.D., "Programmable Systolic Arrays," to be presented at the IEEE Compcon, Spring 1982.

6. Gajski, D.D., Kuck, D.J., and Padua, D.A., "Dependence Driven Computation," Digest of Papers, IEEE Compcon Srping 81 (Feb. 1981), pp. 168-172.

7. Kuck, D.J., The Structure of Computers and Computations, Vol. I, John Wiley & Sons, New York, 1978.

8. Kung, H.T., and Leiserson, C.E., "Algorithms for VLSI Processor Arrays," Introduction to VLSI Systems, by C.A. Mead and L.A. Conway, Addison-Wesley, 1980, pp. 271-292.

9. Kung, H.T., "Let's Design Algorithms for VLSI Systems," Proc. Caltech Conf. on VLSI (Jan. 1979), pp. 65-90.

10. Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Program," CACM, Vol. 21, No. 8 (Aug. 1978), pp. 613-641.

11. Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978, pp. 245-255.

12. Barnes, G., et al., "The Illiac IV Computer," IEEE Trans. Comp., Vol. C-17, No. 8 (Aug. 1968), pp. 746-777.

13. Robinson, A.L., "Array Processors: Maxi Number Crunching for a Mini Price," Science, Vol. 203, No. 12 (Jan. 1979), pp. 156-160.

14. Gostick, R.W., "Software and Algorithms for the Distributed-Array Processors," ICL Technical Journal (May 1979), pp. 116-135.

15. Kruse, B., "A Parallel Picture-Processing Machine," IEEE Trans. Computers, Vol. C-14 (April 1975), pp. 424-433.

16. Narendra, P., "VLSI Architectures for Real-Time Image Processing," Digest of Papers, IEEE Compcon Spring 1981, (Feb. 1981), pp. 303-306.

17. Flanders, P.M., et al., "Efficient High Speed Computing with the Distributed Array Processor," in High Speed Computer and Algorithm Organization, edited by D.J. Kuck et al., Academic Press, 1977, pp. 113-128.

18. McCormick, B.H., "The Illinois Pattern Recognition Computer," IEEE Trans. on Computers (Dec. 1963), pp. 791-813.

19. Stone, H.S., "An Effective Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," J. ACM (Jan. 1973), pp. 27-38.

20. Thurber, K.J., and Wald, L.D., "Associative and Parallel Processors," Computing Surveys, Vol. 7, No. 4 (Dec. 1975), pp. 215-255.

21. Seitz, C.L., "Self-timed VLSI Systems," Proc. Caltech Conf. on VLSI (Jan. 1979), pp. 345-355.

22. Franklin, M.A., and Wann, D.F., "Pin Limitations and VLSI Interconnection Networks," Proc. 1981 Intl. Conf. on Parallel Processing (Aug. 1981), pp. 253-258.

23. Karp, R.M., and Miller, R.E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," SIAM J. Appl. Math., Vol. 14 (Nov. 1966), pp. 1390-1411.

24. Dennis, J.B., "Data Flow Supercomputers," Computer Magazine, Vol. 13, No. 11 (Nov. 1980), pp. 48-56.

25. Dennis, J.B., and Misunas, D.P., "A Computer Architecture for Highly Parallel Signal Processing," Proc. of the ACM 1974 National Conf. (Nov. 1974), pp. 402-409.

26. Watson, I., and Gurd, J., "A Prototype Data-Flow Computer with Token Labeling," AFIPS Conf. Proc., National Computer Conf. (June 1979), pp. 623-638.

27. Davis, A.L., "A Data-Driven Machine Architecture Suitable for VLSI Implementation," Proc. Caltech Conf. on VLSI (Jan. 1979), pp. 479-494.

28. Rumbaugh, J.E., "A Parallel Asynchronous Computer Architecture for Data Flow Programs," Project MAC TR-150, M.I.T., Cambridge, Mass. (May 1975).

29. Chang, T.L., and Fisher, P.D., "A Block-Driven Data-Flow Processor," Proc. 1981 Intl. Conf. on Parallel Processing (Aug. 1981), pp. 151-155.

30. Davis, A.L., Denny, W.M., and Sutherland, I., "A Characterization of Parallel Systems," Technical Report 108, Computer Science Dept., University of Utah (Aug. 1980).

31. Palmer, J., "The Intel 8087 Numeric Data Processor," Proc. Seventh Annual Symp. on Computer Architecture (May 1980), pp. 174-181.

32. Chen. T.C., "Parallelism, Pipelining, and Computer Efficiency," Computer Design, Vol. 10 (Jan. 1971), pp. 69-74.

33. Davidson, E.S., and Larson, A.G., "Pipelining and Parallelism in Cost-Effective Processor Design," Res. Report, Digital System Lab, Stanford University, Stanford, Calif. (1973).

34. Leiserson, C.E., "Systolic Priority Queues," Proc. Caltech Conf. on VLSI (Jan. 1979), pp. 199-214.

35. Guibas, L.J., Kung, H.T., and Thompson, C.D., "Direct VLSI Implementation of Combinatorial Algorithms," Proc. Caltech Conf. on VLSI (Jan. 1979), pp. 509-525.

36. Gannon, D., and Snyder, L., "Linear Recurrence Systems for VLSI: The Configurable, Highly Parallel Approach," Proc. 1981 Intl. Conf. on Parallel Processing (Aug. 1981), pp. 259-260.

37. Chang, T.L., and Fisher, P.D., "Mixed Systolic Arrays," submitted for presentation at Ninth Annual Symp. on Computer Architecture, Texas, 1982.

38. Song, S.W., "A Highly Concurrent Tree Machine for Database Applications," Proc. 1980 Intl. Conf. on Parallel Processing (Aug. 1980), pp. 259-268.

39. Weng, K., "Stream Oriented Computation in Recursive Data-Flow Schemes," Project MAC TM-68, M.I.T., Cambridge, Mass. (Oct. 1975).

40. Bergland, G.D., "A Guided Tour of Program Design Methodologies," IEEE Computer, Vol. 14 (Oct. 1981), pp. 13-37.

41. Arvind, "Decomposing of a Program for Multiple Processor Systems," Proc. 1980 Intl. Conf. on Parallel Processing (Aug. 1980), pp. 7-14.

42. Bridge, C.L., Fisher, P.D. and Reynolds, R.G., "A Synchronous Arithmetic Algorithms for Data-Driven Machines, "Proceedings of the 5th Symposium on Computer Arithmetic, (May 1981), pp. 56-62.

43. Meyer, G., Advances in Computer Architecture, Wiley, New York, 1978.

44. Reynolds, R. G. and Chang, T.L., "The PAL System-A Parallel Algorithm Design System for VLSI-Based Array Architectures," to be presented at The 10th IMACS World Congress on SYSTEMS SIMULATION AND SCIENTIFIC COMPUTATION, (Aug. 1982), Montreal, Canada.