STARS

University of Central Florida
STARS

Retrospective Theses and Dissertations

1987

Real Time Signal Processing Using Systolic Arrays

Jack Boulay
University of Central Florida

Part of the Engineering Commons

Find similar works at: https://stars.library.ucf.edu/rtd
University of Central Florida Libraries http://library.ucf.edu

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Boulay, Jack, "Real Time Signal Processing Using Systolic Arrays" (1987). *Retrospective Theses and Dissertations*. 5058.
https://stars.library.ucf.edu/rtd/5058

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

REAL TIME SIGNAL PROCESSING
USING SYSTOLIC ARRAYS

BY

JACK BOULAY
B.S.E, University of Central Florida, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in Engineering
in the Graduate Studies Program of the College of Engineering
University of Central Florida
Orlando, Florida

Fall Term
1987

ABSTRACT

This thesis discusses and presents the design of systolic arrays used in modern real time signal processing. A methodology to map a given algorithm into a systolized VLSI implementation is described. The architectural alternatives for a given signal processing algorithm are discussed and investigated at a function level using a simulation package that has been developed using the "C" programming language.

The similarities and differences between wavefront array processors and systolic array processors are presented.

ACKNOWLEDGEMENTS

This author wishes to express his appreciation to the many people who contributed toward the preparation of this thesis. Dr. Brian Petrasko deserves special thanks for his valuable advice, creative ideas, and, most of all, patience with the author. Also, special thanks are extended to my family that made this thesis a reality by supporting me through my years of school. Thanks Alex. Thanks Debby.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

INTRODUCTION 1

CHAPTER ONE - REAL TIME SIGNAL PROCESSING USING ARRAY OF PROCESSORS 3
Applications in Signal Processing 3
Digital Systems Architectures for Signal Processing 3
Digital signal processing and VLSI technology . . . 5
Types of VLSI Structures 6
Efforts in the design of array processing systems . 12
Task description 13

CHAPTER TWO - ALGORITHM TO ARCHITECTURE 15
Algorithm to Architecture methodology 15
Representation form of The Computational Model . . 16
Examination of a SFG For Systolic Array Implementation 18
Modularity 21
Measure of Performances 22
Systolization procedure 23
Examples 25

CHAPTER THREE - VERIFICATION OF SYSTOLIZATION PROCESS . 37
Language selection for the simulation package . . . 37
Structural definition of the simulation package . . 39
Program layout 39
Program description 40
Data structure 47
Results 48
Conclusion 59

CHAPTER FOUR - CONTROL STRATEGIES 61
Timing consideration 62
Wavefront Array 63
Transformation of a SFG to a WAP 64
Examples of SFG to WAP transformation 65

CONCLUSION 66

APPENDIX 68
A. Tutorial 69
B. Fault Tolerance 82
C. Program listings 85
D. Modularizing an SFG 100

REFERENCES 106

LIST OF TABLES

1. Virtual FIR Timing Requirements 31

2. Virtual FIR Simulation Results 49

3. Forward FIR Simulation Results 51

4. Backward FIR Simulation Results 52

5. Matrix Simulation Snapshots 54

6. Matrix Simulation Results 58

7. Performance Characteristics 60

LIST OF FIGURES

1. Mesh Connected Array Processors 7
2. Function level Architecture of Discussed PE. 10
3. S.F.G. Components 18
4. S.F.G. of parallel Form 19
5. S.F.G. with Spatial Locality 19
6. Virtual FIR 26
7. Identifying cut sets 26
8. Forward Systolized FIR 26
9. Local SFG for Backward FIR 30
10. Cut sets for Backward FIR 30
11. Systolic Backward FIR 30
12. Spatialy local SFG for Matrix Multiplication 34
13. Matrix Systolization Process 35
14. PE for Matrix Multiplication 36
15. Array processor for Matrix Multiplication 36
16. File Linking Structure 41
17. Forward Systolized FIR SFG 71
18. PE for Forward FIR Filter 71
19. Systolic Forward FIR Array interconnections 71
20. S.F.G. for Matrix Multiplication PE. 76
21. PE for Matrix Multiplication 76
22. Array processor for Matrix Multiplication 76
23. Fault Tolerance Scheme for unidirectional Array . . 84

24. S.F.G. Arma Filter Direct Form 1 103
25. S.F.G. with Minimum Number Of Delays 103
26. S.F.G. Direct Form 2 104
27. Linear representation of Arma Filter 104
28. Arma filter Cut Set Selection 105
29. Arma filter time rescaling 105
30. Sytolized Arma Filter 105

INTRODUCTION

As we drift into an age of information it becomes obvious that the early ages of computer architecture are doomed to change. Due to an ever-increasing demand in computational power, it is necessary to investigate new architectures better suited for signal processing in terms of speed volume and cost.

A look at the computer industry shows a rapid transition from single processor to parallel processor machines. This is due to the fact that single processor machines, in order to improve their throughput, need faster and more expensive circuitry. At the end of the scale, circuit speed reaches its limits; as in order to improve the speed distances between components are reduced to a compactness that exceeds the ability of the circuit to dissipate heat.

On the other hand, concurrent array processors (CAP) achieve a greater speed by dividing the program, and processing its parts simultaneously by different processors. Processors range from custom made applications to standard microprocessor units. They can be combined in different ways and linked via a variety of communication schemes.

This new architecture has already resulted in a range of machines that are aimed at markets sectors as diverse as on line transaction processing and fluid flow simulation studies.

CHAPTER 1

REAL TIME SIGNAL PROCESSING USING ARRAYS OF PROCESSORS

1. APPLICATIONS IN SIGNAL PROCESSING

The applications of signal processing are numerous and multiple in the present world of electronics. In the domain of consumer electronics, goods such as telephones, radios and televisions receivers, disks and tape players are tangible examples of how signal processing directly affects us. In the domain of commercial electronics, applications in telecommunications and control systems create a big demand for signal processing. Similarly, in military radars and sonars the demand for high quality components is strong. Finally, but not the last man-machine interfaces in artificial intelligence requires strong signal processing tools. Due to the sharp penetration of signal processing techniques in today's electronics it is important to optimize quality and costs. This is achieved by using digital signal processing (DSP) in conjunction with VLSI implementation.

2. DIGITAL SYSTEMS ARCHITECTURES FOR SIGNAL PROCESSING

In digital signal processing, operations such as averaging, differentiation etc., are performed on a sequence of numbers that represent samples of some analog signal. Many of today's signal processing applications require

immediate interaction between the user or system and the machine. In other words real time processing is required. In the domain of real time signal processing some tasks are difficult to perform due to the inability to match the computational rate to the data input rate. Because time is a constraint, measures such as response time and throughput are becoming increasingly critical. Array processing addresses this need for additional power.

Specific tasks that need to be performed in real time in modern signal processing systems include matrix multiplication, or solution of linear equations. It has been shown [8] that these tasks can be easily solved using a concurrent processor array architecture.

The characteristic of this architecture is the use of a number of arithmetic units each concurrently performing a specific function on a data set. This is a considerable improvement when compared to architectures using a single arithmetic and logic unit a main task can be broken up into subtasks that are processed by different processor elements in a parallel and/or overlap fashion. This concept of concurrently operating arithmetic units is the core concept of array processing. In order to support this new field it is important to formulate new computational models which support parallelism. It must be noted that array structures can be easily implemented in VLSI. Furthermore, the level

of performances in array processing are in part dictated by advances in VLSI technology.

3. DIGITAL SIGNAL PROCESSING AND VLSI TECHNOLOGY

The rapid innovation in VLSI technology in terms of low cost, high density, and speed is having an impact in modern signal processing. The trend is to translate computational models into promising VLSI implementation technologies. Array architectures that were previously hampered by memory cost (i.e., local memories) are now being reexamined and implemented. Those new perspectives are heralding a new era of signal processing using VLSI.

While VLSI is well suited for digital signal processing there are constraints. The level of integration will not always be expandable. Current state of the art chips are fabricated with a minimum feature size of 1 to 2 micrometers. This size could be reduced in the next couple of years to 0.5 micrometer. This reduction in size implies an increase in throughput rate (i.e., clock speed times gate density) from 5 x 10**11 Hz gates/cm**2 to 10**13 Hz gates /cm**2. [9] Beyond this point it seems that higher levels of integration will require more time, effort and money. This is the reason why the throughput rate must be improved through new and more advanced architectures.

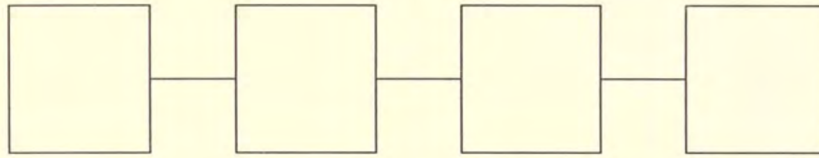
Some of the restrictions associated with the use of integrated circuit technology are directly translated into new problems to be solved in signal processing. For

example, the requirement of local communication between array elements in array processing is due to the fact that interconnections in VLSI must be minimized.

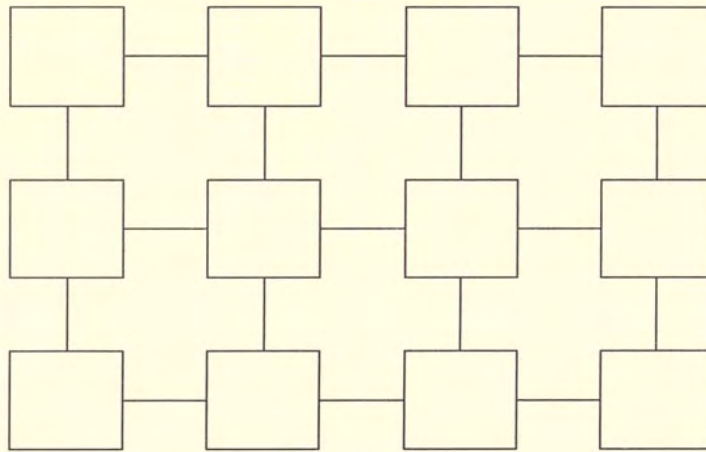
In summary, the increase in performance demanded by real time signal processing has shifted the attention from Von Neuman single instruction stream single data stream (SISD) structures to array structures. VLSI signal processing, using these structures, can be the answer to many of the problems requiring high throughput rate in order to support tremendous computations capabilities in terms of volume and speed.

4. TYPES OF VLSI ARRAY STRUCTURES

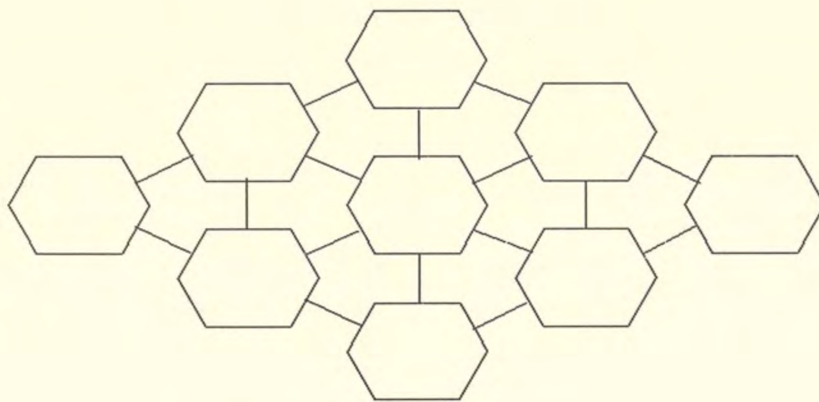
An array structure is a network of interconnected processor elements (PE) which process data in a controlled fashion. The different types of control mechanism provide for an architectural classification of these array structures. For instance, if a synchronous control scheme is used the array is referred as systolic. In the case of an asynchronous control mechanism, a wavefront array is obtained. However, all these architectures capitalize on regular and modular structures with different types of interconnections as shown in Figure 1. The choice of array structure depends on the communication required by the given algorithm and application. Also, the PE's can be dedicated or programmable. A dedicated PE is said to be hardwired; this leads to an inflexible structure and therefore might



a) Linear Connected



b) Orthogonally Connected



c) Hexagonally connected

Figure 1. Mesh Connected Array Processors

limit the range of application of a given machine. A programmable PE offers the advantage of replication as well as flexibility. The disadvantage is the additional control complexity associated with dynamic interconnections or array reconfiguration. The trade-off between a dedicated and a programmable processor element is a very fundamental issue. In order to come up with an optimum choice, the designer must decide how much flexibility will be included in the special purpose computer. Two types of arrays are considered: the systolic array and the wave front array. The major difference is in the control mechanism.

4.1 Systolic Arrays

According to Kung and Leiserson [8], "A systolic array is a network of processors which rhythmically compute and pass data through the system." Systolic arrays use multiprocessing and pipelining to achieve greater throughput. Multiprocessing indicates that various PE's are processing data simultaneously in the array structure. Pipelining takes advantages of dependencies among computations to propagate the result of one PE to the next PE for further data processing. This concept of overlap is important as the data is being used within the pipe thus reducing input output and memory bandwidth requirements. In a systolic pipe the movement of the data is restricted to neighboring PE's and take place in a periodic manner. Within the pipe each processor can perform a given computational

task. For example, a common processor element structure is one which executes the short computation Y <--- Y + A*B (see Figure 2). It is important to realize that the data is passed rhythmically along the pipe.

A systolic array possesses the following characteristics:

a) Spatial locality

The array is a network of PE's with local interconnections. Any PE which is providing the input data for the next PE to process it must be physically close to that next PE. This condition addresses the cost of interconnections in VLSI technology. It is also desirable to have a minimum number of short interconnections between adjacent PE's. Again, the connections or communication between neighboring modules should be minimized.

b) Temporal Locality

The array should present temporal locality. The results of a predecessor module are available to the next module at the next clock time. This condition is a control characteristic which restricts the systolic array to a classical pipeline stucture.

c) Regularity

The array is a network of PE's which are largely of the same type. However there may be some atypical cases at the boundaries of the array or in critical paths. The

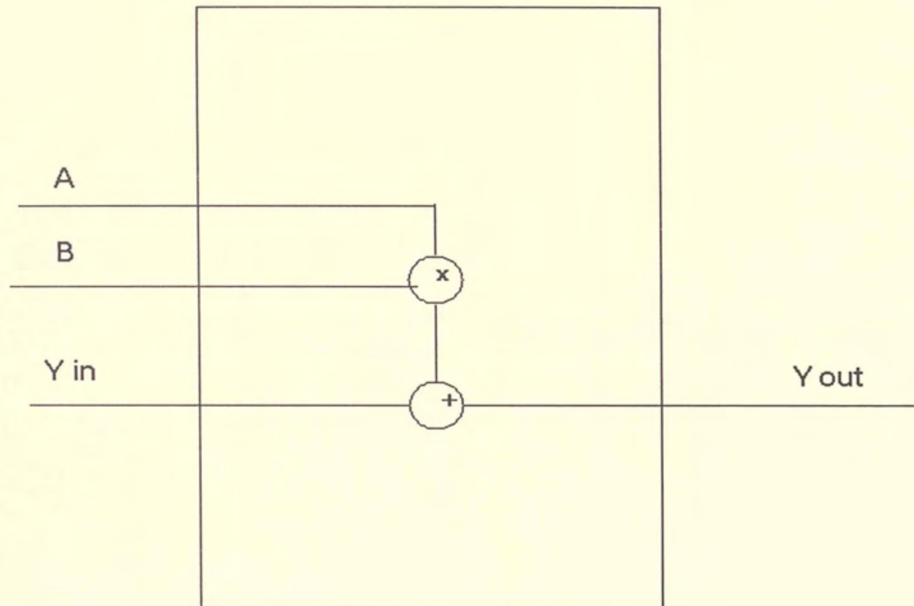


Figure 2. Function Level Architecture of Discussed PE .

condition of modularity is needed to achieve area efficient layouts as required in VLSI

d) Synchrony

A global clock is used to step the data in a rythmic fashion through the system. There is need for a global timing scheme to ensure that the data is available at a specific time.

d) Order(M) Speedup Factor

The systolic array must present an order M speed up factor where M is the number of stages within the pipe. The processing time for the array realization must be less than the processing time for a single processor machine.

Chapter 2 presents further discussion of these characteristics and includes examples.

4.2 Wavefront Array Processor

A wavefront array processor is a data driven machine which has a throughput rate which is potentially higher than that of a systolic array. The spatial locality requirements are essentially the same as those established above for the systolic array. Also regularity and modularity are required. In contrast to a systolic array, the control mechanism is based on the occurrence of a sequence of events rather than on a global synchronous clock. To be more specific, the triggering of instructions depends on the availability of operands and resources required. Therefore the data driven

operation of each PE requires the adoption of handshaking protocols to synchronize data flow.

5. EFFORTS IN THE DESIGN OF ARRAY PROCESSING SYSTEMS

Many real time digital signal processing systems can be implemented using special purpose computer systems or components. There are also a number of applications that require the performances associated with a VLSI implementation. In both cases, design tools and methodology are critical in the design, analysis and development process.

A noteworthy effort to provide a complete design automation system for VLSI signal processing is the Cathedral project [1]. It has produced a silicon compiler for bit serial structures which can include systolic type array processing. However this subset of the design spectrum has not been emphasized. Design techniques specific to systolic and wavefront array processing for VLSI are currently being investigated by a number of research groups, most notably the group at Carnegie-Mellon University (T.C. Kung) and at the University of Southern California (S.Y. Kung). Of major interest is the investigation of methods of systolizing algorithms. Applying these techniques to signal processing applications include the following steps or phases.

1. The first phase will develop the application specification. It is important to clearly define the performances requirements since this determines the

need and degree of array processing.

2. The system function must be developed. The alternative representations of the system function will lead to computational models that can be investigated for systolization. A formulation which provide for spatial locality is crucial.
3. The computing structure must be identified. This is achieved by applying the systolization procedure to a recurrence form of the algorithm or a Signal Flow Graph representation form of the algorithm. This is a critical step as it determines the overall architecture of the array.
4. The array architecture is then examined. This includes a simulation at the function level to examine the correctness, "gross" timing information, and space time trade offs of the solution. Following those results the designer may want to investigate new architectures.
5. The array processor is fabricated and tested

6. TASK DESCRIPTION

The objective of this thesis paper is to provide the basis for the design and development of digital signal array processing design facilities at UCF. To this end, examples of applications of the above methodology steps 3 and 4 are provided. A simulation package for examining the behavioral

or computational model and the structural or systolized model is developed.

CHAPTER 2

ALGORITHM TO ARCHITECTURE

The procedures and requirements necessary to derive an array architecture given a specific algorithm are investigated.

1. ALGORITHM TO ARCHITECTURE METHODOLOGY

As established before, an array processor is a direct hardware implementation of a special computational model used to solve a given problem with speedup. This characteristic is crucial as the objective is to implement an algorithm by a high performance parallel network. Many of the algorithms encountered may not seem to be suited for parallel processing. However, through manipulation and optimization techniques, many real time signal processing algorithms can be made efficient. A necessary step is to reformulate a given algorithm in order to identify any recurrence within its structure. By recurrence we mean any set of operations that are repeated within the formulation. Each operation is given the name of iteration. By assigning an individual processor or processors for each iteration in the recurrence, concurrency is achieved by overlap. This is the objective of pipelining. As is also for the case of two dimensional arrays, parallelism is often achieved.

A requirement for array processing is to have the communication between the iterative step processor be of the local type. The input and output data are labelled with time and space indices (i.e., location of the processor within the array). Spatial locality is achieved if the space index separation within two successive iterations is within a certain limit [8].

For systolic arrays an additional requirement is temporal locality. The simple control mechanism of a single global clock imposes the constraint of concurrent data arrival at the input of each array processing element (PE). This is achieved by manipulation of the delays within the array and/or the addition of delays. This method is also referred to as the systolization procedure.

2. REPRESENTATION FORMS OF THE COMPUTATIONAL MODEL

There are two common representation forms; a recurrence expression and a signal flow graph. The recurrence expression has indices of space and time. For reference the recurrence expression of a square matrix multiplication C = A*B is presented: The superscript in the following relation is the time index.

For k = 1 to N

$$c_{i,j}^{0} = 0$$

$$c_{i,j}^{k} = c_{i,j}^{k-1} + a_{i}^{k} b_{j}^{k}$$

$$a_i^k = a_{ik} \qquad b_j^k = b_{kj}$$

Spatial and temporal locality can be investigated using algorithms which manipulate index sets and derived data dependence vectors [8]. However this mathematical approach is cumbersome and prone to errors.

An easier method is to use a signal flow graph (SFG) which provides a graphical representation of the recurrence formula. This method is more common to DSP and provides a visual statement of the characteristics of the computational model. Spatial and temporal locality are investigated by examining node connections and path delays. This later representation is used in this report.

Signal Flow Graph Representation

The signal flow graph is one of the most useful representation of a signal processing system. A SFG is a collection of nodes and edges. Nodes represent logic or mathematical functions performed with zero delay. Nodes are connected by edges. Three types of operations are used in the following discussion and examples: summation, multiplication and delay. A node with more than one input edge identifies a summation. An edge with a constant identifies a multiplication of the result of the preceding node with the constant. A D over an output edge indicates a time delay of the results coming from the node (see Figure 3).

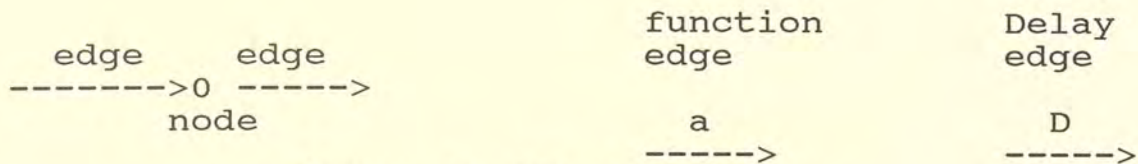


Figure 3: SFG Components

3. EXAMINATION OF A SFG FOR SYSTOLIC ARRAY IMPLEMENTATION

Given a function $y(k) = \sum_{i=1}^{i=M} b_i x(k-i)$, the corresponding SFG is drawn in Figure 4 and investigated for systolic attributes.

3.1 Spatial Locality

As established in Chapter 1, the length and number of connections between PE's should be minimized and restricted to neighboring PE's. For the SFG of Figure 4 an implementation of the computational model requires M (where M is arbitrary) connections to the summing node, of the pipeline. Due to the parallel nature of the network the use of a linear connected structure (see Figure 1) will result in some PE's being closer than others to the summing node. Use of a hexagonally connected structure will allow all the PE's to be separated from the summing node by the same distance. In the first case for an increasing M (M is the number of iteration and is arbitrary) PE's on the boundaries will be separated from the summing node by increased distances and will not be spatially local. In the second case, the type of structure changes with M and the length of the interconnections cannot be minimized. Thus the characteristics of spatial locality are not present.

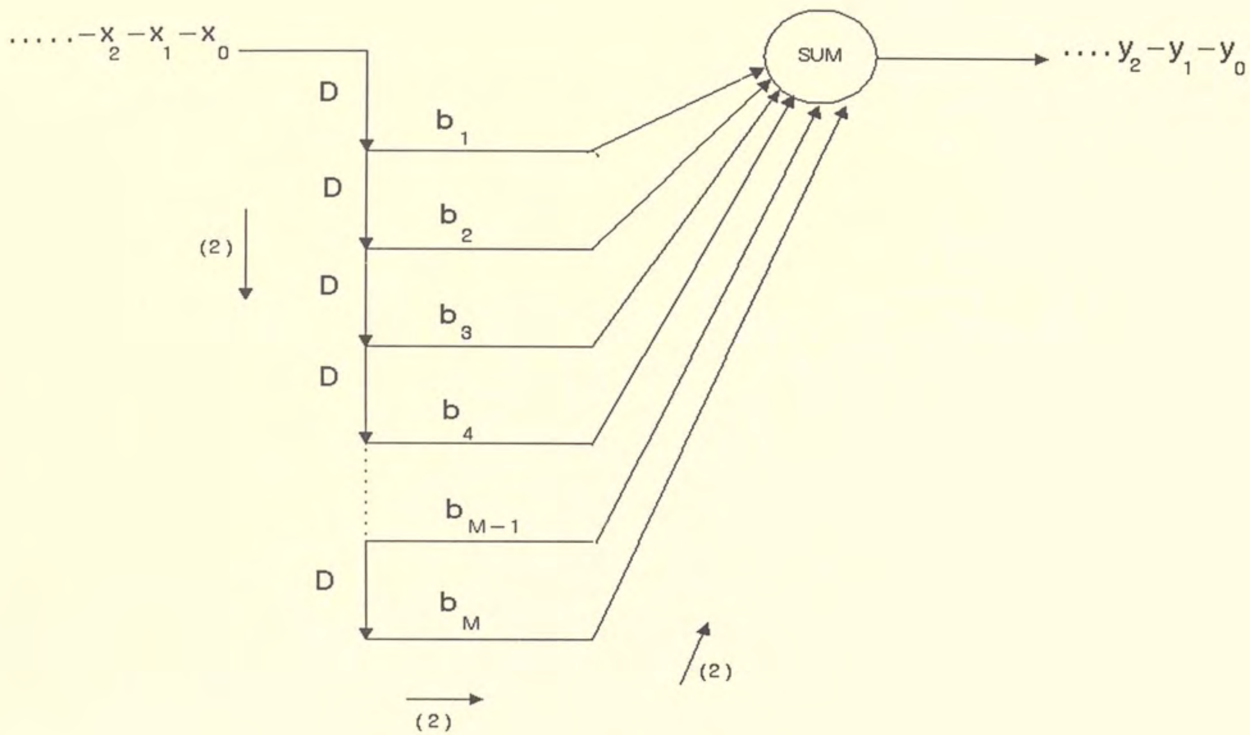


Figure 4. SFG of Parrallel Form

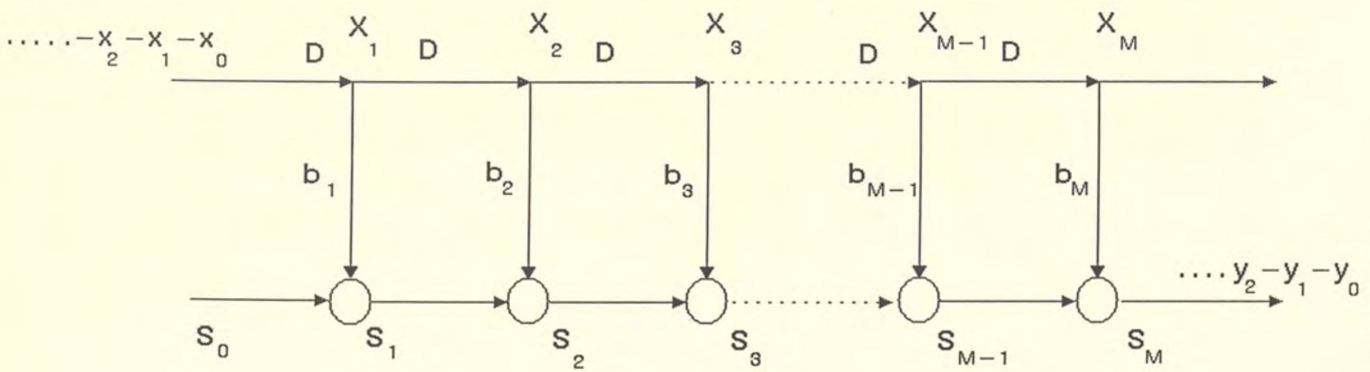


Figure 5. SFG with Spatial Locality

3.2 Temporal locality

Because the results of the parallel multiplications are available to the summing node at the same clock time as they are computed, no temporal locality is achieved. In order to achieve temporal locality, latches must be added to the b labelled edges.

3.3 Regularity

The process of multiplication, $b_i x(k-i)$ is investigated as the target PE process for the SFG of Figure 4. Using this partitioning of processes, regularity can be seen as the partitions are modular since each element performs an identical function on the data set.

3.4 Synchrony

No synchrony is achieved as the data in the b branches is available at the inputs of the summing node without a clocking process. However, synchrony can be achieved with the addition of delays. This results in the data being stepped through the system in a rythmic fashion.

3.5 Order(M) Speedup Factor

The array presents an order M speed up factor where M is the number of stages within the pipe. Speedup is achieved in this array realization by having parallelism in the b branches and having overlap in the D path. Once the pipe is full output samples will be available at each clock cycle. As a consequence the processing time for the array

realization is less than the processing time for a single processor machine.

It will be shown in the next example that a more appropriate starting point is a SFG derived from a recurrence formula of the system equation. Identifying the recurrence formulation of the preceding system as:

For i = 1 to M

$$x_i^k = x_{i-1}^{k-1}$$

$$s_i^k = s_{i-1}^k + b_i x_i^k$$

$$x_0^k = x_{in} \; ; \; s_0^k = 0$$

and drawing the corresponding SFG (see Figure 5); the spatial locality is apparent. However temporal locality is not present as the data on the lower path is fed through a zero delay path. As a consequence some new provisions must be established so that the temporal locality criterion is specified by the SFG.

4. MODULARITY

The major decision in the implementation of the algorithm as an array structure is the type or types of element to be used. Relating this to a SFG, the partitioning used defines the characteristics of the hardware implementation. It can range from a single element for each iteration to a large number of elements within the

iteration. This decision strongly impacts throughput and response time. By having smaller partitions, a finer granularity is achieved (i.e., the number of stages is increased). Using this principle, a higher throughput rate might be achieved [7]. Unfortunately, the response time or latency may also be higher. It is also common to have dissimilar stages. In the above example the array PE processes one iteration; a delay followed by a multiplication, followed by a summation. Using different partitions, or cuts, parallelism can be increased within the PE thus decreasing the stage process time and thus increasing the throughput. However, there is a potential cost of increased communication paths and response time. It should be noted that in this example, the multiplication-summation sequence should not be partitioned if the multiplication is implemented as a successive addition process; as is the case if CSA trees are used [6].

Partitioning requires a careful examination of temporal locality. Kung provides a cut set procedure that advocates lower communication requirements and addresses the temporal issue of systolization [8]. This is presented in section 6.

5 MEASURE OF PERFORMANCES

A pipeline is said to have speedup if N sequential tasks can be performed faster by a pipeline than it could be performed by a single processor. A pipeline provides for a

speedup of order M where M is the number of stages in the pipe. The order is determined by data dependence. Note that in the case of a large system processing a small data set, the latency of the pipe may have an influence on the level of performance (the fill and flush time might be greater than the processing time.)

Another performance measure of the system is the throughput rate. In DSP the throughput is the number of samples processed per clock cycle. In the ideal case the throughput will be equal to the clock frequency. The clock frequency can be increased through two methods:

1) Find a finer granularity of the partitioning.
2) Identifying parallelism within the PE.

In contrast to speedup, the throughput is a function of the stage delay time and not of the number of stages.

6. SYSTOLIZATION PROCEDURE

A systolic array has all of the characteristics of a classical pipeline [6]. Each stage of the pipe is processing a sequential portion of the system task. Each process must be completed for each clock time. As a consequence the data for each PE must be available at each clock time and must be provided by a spatially local process (i.e., the preceding stage). Thus broadcast paths must be eliminated and paths with zero delays must include provisions for extra delays in order to preserve synchronization. A key step in the manipulation of a SFG to

achieve temporal locality is the subtraction of delays from the input of a node and the addition of delays to the output. An SFG can be retimed using this step. Also the timing relationships between inputs and outputs must be respected. These concepts are embodied in the systolization procedure presented below.

An SFG derived from a recurrence formulation gives a good starting point for an array realization. Having achieved spatially local interconnections and modularity Kung [8] has derived a graphical method to systolize a SFG. First he defines a cut set as being "a minimal set of edges which partitions the SFG into two parts." As should be noted the emphasis is on reducing the number of interconnection as required in VLSI. It is important to realize that not all cut sets present the attribute of being a "good cut set." A good cut set should only include:

1) the target edge or selected zero delay edge.
2) non zero delay edges going in either direction
3) zero delay edges going in the same direction as the target edge.

A bad cut set will cut zero delay edges going in the opposite direction of the target edge. Based on this partitioning theory the temporal localization procedure can be identified. The objective of this procedure is to eradicate zero delay edges between modular sections. The

temporal localization procedure is based on two simple rules:

1. TIME SCALING

All delays D may be scaled, i.e., D->aD', by a single positive integer, a. Correspondingly the input and output rates also have to be scaled by a factor a (with respect to the new time unit D'). This process is referred to as retiming by Glasser [5].

2. DELAY TRANSFER

Given any cut set of the SFG, we can group the edges into inbound edges and outbound edges, depending upon the directions assigned to the edges. Rule 2 allows advancing k(D') time units on all the outbound edges and delaying k time units on the inbound edges, and vice versa. All initial timing relationship must be preserved.

Given the information established in the previous section it is now possible to define the systolization procedure as given below.

1) Define the basic processor element.

2) Select good cut sets.

3) Apply the localization rules

7. EXAMPLES

This section presents specific examples of the algorithm to architecture step. First, the one dimensional FIR algorithm is investigated using three different SFGs,

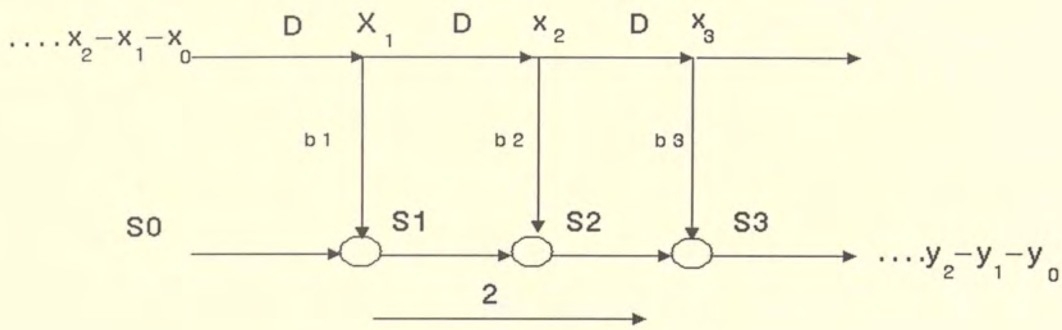


Figure 6. Virtual FIR

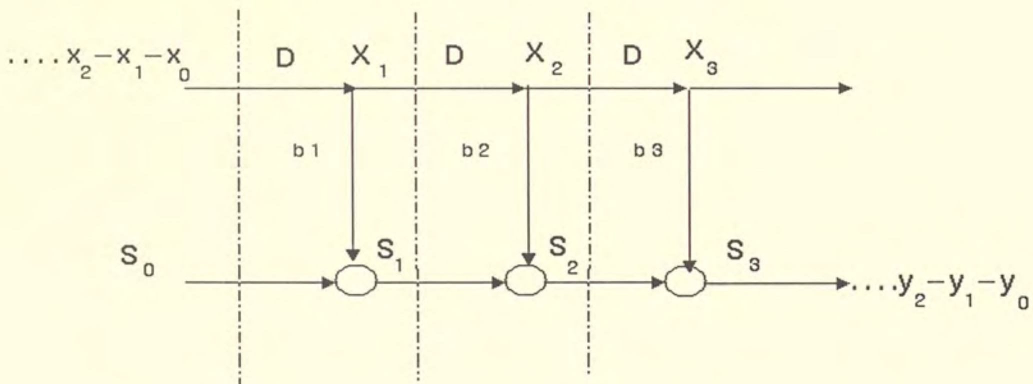


Figure 7. Identifying Cut Sets

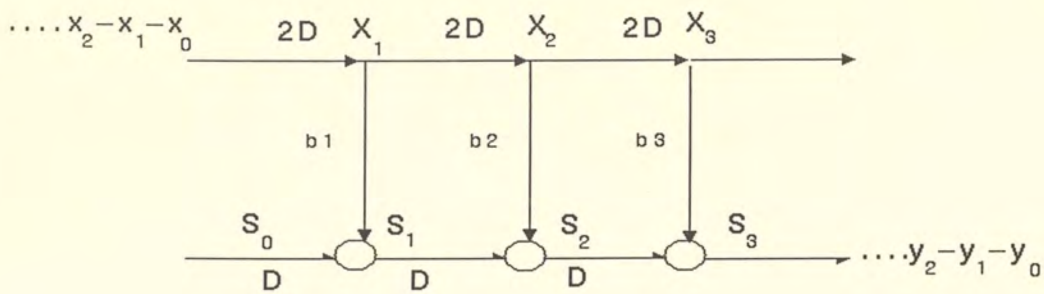


Figure 8. Forward Systolized FIR

then the matrix multiplication algorithm; a two dimensional example is presented.

7.1. FIR Filter

This filter is described by the linear difference equation:

$$y(k) = \sum_{i=1}^{M} b_i x(k-i)$$

We restrict our example to the case of three iterations per sample cycle.

7.1.1 First SFG

A recurrence formulation for this equation is:

For i=1 to 3

$$x_i^k = x_{i-1}^{k-1}$$

$$s_i^k = s_{i-1}^k + b_i x_i^k$$

$$x_0^k = x_{in} \; ; \; s_0^k = 0$$

The recurrence formulation provides for pipelining because the i-1 iteration of the equation can be performed at the same time as the i iteration. Mapping this equation results in the SFG shown in Figure 6. The selected cuts identify the target PE performing a single iteration (see Figure 7).

7.1.1.1 Spatial Locality

It can be seen that the condition of spatial locality is present. The input x_{i-1}^{k-1} is local to x_i^k and input s_{i-1}^k is also spatially local to s_i^k (i.e., space indexes i are separated by a factor of 1).

7.1.1.2 Temporal Locality

However, temporal locality is not achieved as there is a zero delay path indicated by arrow 2 in Figure 6. By introducing a delay in the lower branch the zero delay path is eliminated. Also, in order to preserve the input-input timing relationship (Rule #2 of the systolization procedure), an extra delay is inserted in the upper branch. This concludes the systolization procedure (see Figure 8).

The new recurrence formula for this systolic case is given below for reference only.

For i=1 to 3

$$x_i^k = x_{i-1}^{k-2}$$

$$s_i^k = s_{i-1}^{k-1} + b_i x_i^k$$

$$x_0^k = x_{in} \; ; \; s_0^k = 0$$

7.1.2 Second SFG

In the above SFG, the filter input and output are distant in space. It may be desirable to have them spatially local. Observing the associative property of the

summing process (i.e., A + (B + C) = (A + B) + C) a second SFG can be derived from the above SFG (see Figure 9) which provides for x and y to be spatially local.

7.1.2.1 Spatial locality

Again the SFG is visually spatialy local for the same target PE.

7.1.2.2 Temporal Locality

Because of the zero delay sum path the SFG is missing temporal locality. The systolization procedure is initiated. First, following rule #1; delay D is rescaled as 2D'. Then applying rule #2, the inbound edge is delayed by D' while the outbound edge is advanced by D'. As a result the array is now systolic (see Figure 11). However, the time rescaling required by the systolization process impacts the input rate. For correct operation two beats or clock times are required per input sample. The first beat moves the data through the required (algorithm) delay and the second beat latches the MA process and synchronizes the upper and lower path movement. In order to preserve the input-output relationship lost during time rescaling (original delay difference between input and output paths) zeros must be interleaved in the input data stream. To generalize, it can be said that for non-systolic SFG having inputs and outputs propagating in different directions, time rescaling is required and it will lead to interleaving the input data stream by a number Z of zeros function of the rescaling

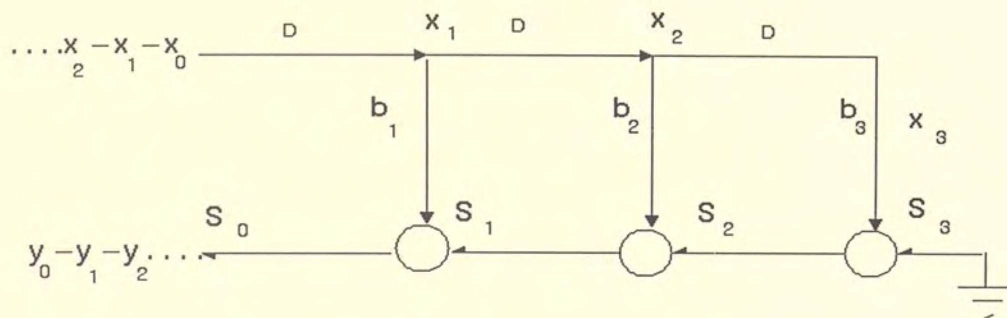


Figure 9. Local SFG for Backward FIR

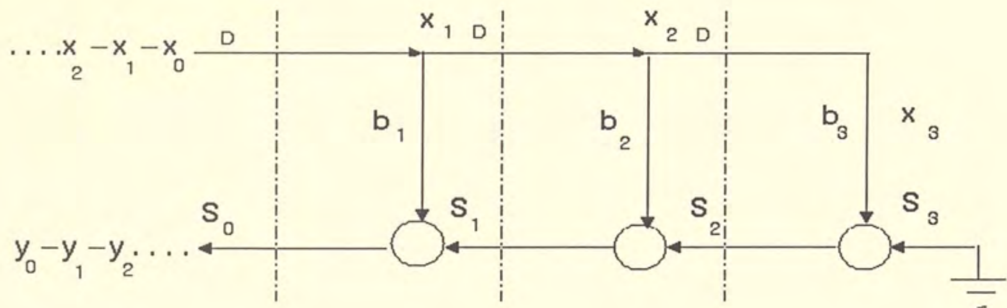


Figure 10. Cut Sets for Backward FIR

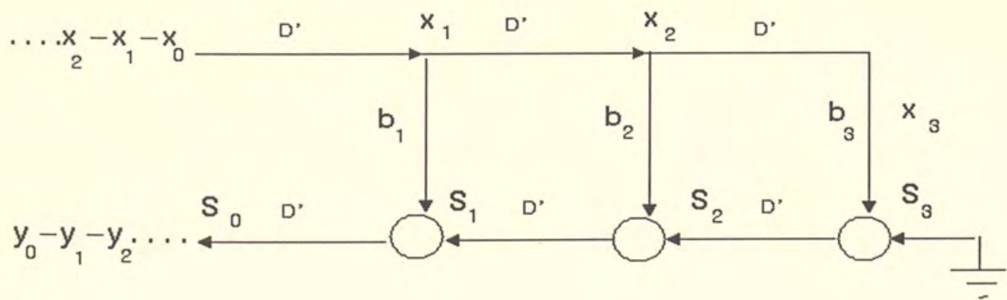


Figure 11. Systolic Backward FIR

factor. This impacts throughput by a factor of 1/Z.

7.1.3 FIR Computational Model

A third example is the case of an SFG that is not temporally localized. Note that the zero path has not been removed thus allowing the data to propagate immediately across the summing path (see Table 1). A non optimum implementation can be derived from this case by noting that each MA has a given latency and before another input can be piped in, the system has to wait for the sum to ripple through all the n MAs. In the case of very large systems the delay associated with this ripple will slow the clock rate and thus the throughput.

TABLE 1 : FIR COMPUTATIONAL MODEL TIMING

Xin	X1	X2	X3	S0	S1	S2	Y=S3
1	0	0	0	0	0	0	0
2	1	0	0	0	b1	b1	b1
3	2	1	0	0	2b1	2b1+b2	2b1+b2
4	2	1	0	3b1	3b1+2b2	3b1+2b2+b3	

Table 1 assumes that the pipe has been previously purged.

7.2. Matrix Multiplication

A systolic architecture can be viewed as the implementation of a set of recurrence relations by a set of identical cells. In this example a systolic cell is derived to compute the matrix product C = A*B. A and B are assumed to be n x n matrices.

As a general formula each element c_{ij} is given by:

$$c_{ij} = \sum_{k=1}^{M} a_{ik} * b_{kj}$$

A recurrence formulation for this equation is:

For k = 1 to M

$$c_{ij}^{k} = c_{ij}^{k-1} + a_{i}^{k} * b_{j}^{k}$$

where

$$a_{i}^{k} = a_{ik} \quad ; \quad b_{i}^{k} = b_{ik}$$

with

$$c_{i,j}^{0} = 0$$

Such summations may be evaluated via recurrences by having the partial sum c_{ij}^{k-1} moving through the structure, or by accumulating partial sums in place. Choosing the latter case a SFG representation of the function is given in Figure 12. Note that the delay associated with the partial sum calculation acts as a small memory element used to store in place, the result of the summation. Observing Figure 12, two zero delay paths can be found for each cell. They correspond to the data paths of input matrixes A and B. A cut set is identified in Figure 13. Observing that the loop is cut by the partition, we can apply rule #2 thus advancing the outbound edges by D'and delaying the inbound edge by D'.

Thus, each cut set adds a delay in the A and B paths. The accumulated delays are added to the inputs of the A and B paths (see Figure 13). These delays at the array input can be omitted by adding leading zeroes to the respective A and B data streams. This timing (leading zeroes) is a start up condition and can be viewed as an initialization of the state variables (D's) associated with the data paths (Figure 15). The in place state variable must be initialized to zero. This leads to the PE architecture shown in Figure 14. The in place state variable is labelled Dc.

- -

Remark that the I/O lines to the PE cin and cout are not used in this particular array realization. However to provide with more flexibility as to the future use of this PE they are included in the realization.

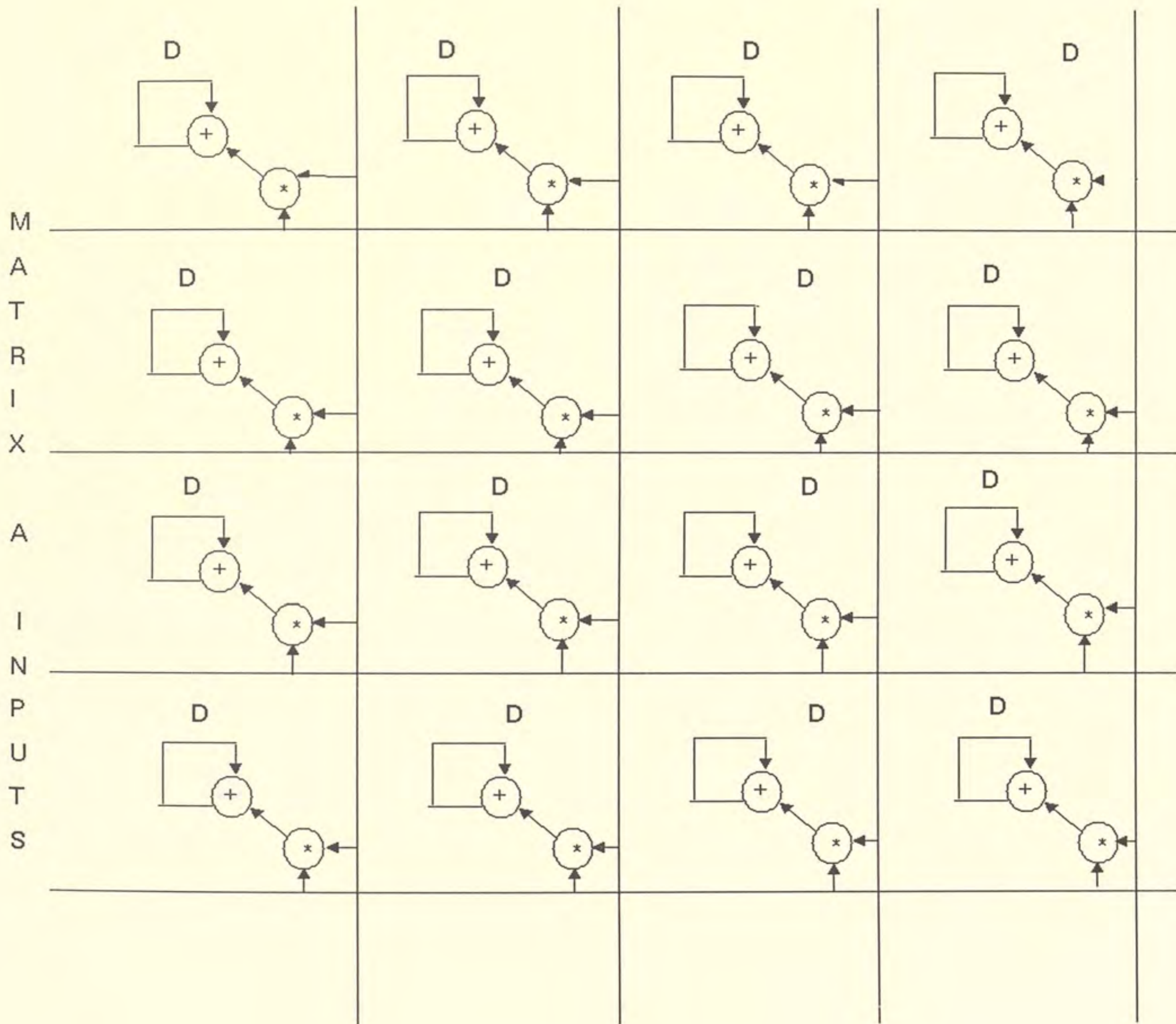


Figure 12. Spatially Local SFG for Matrix Multiplication

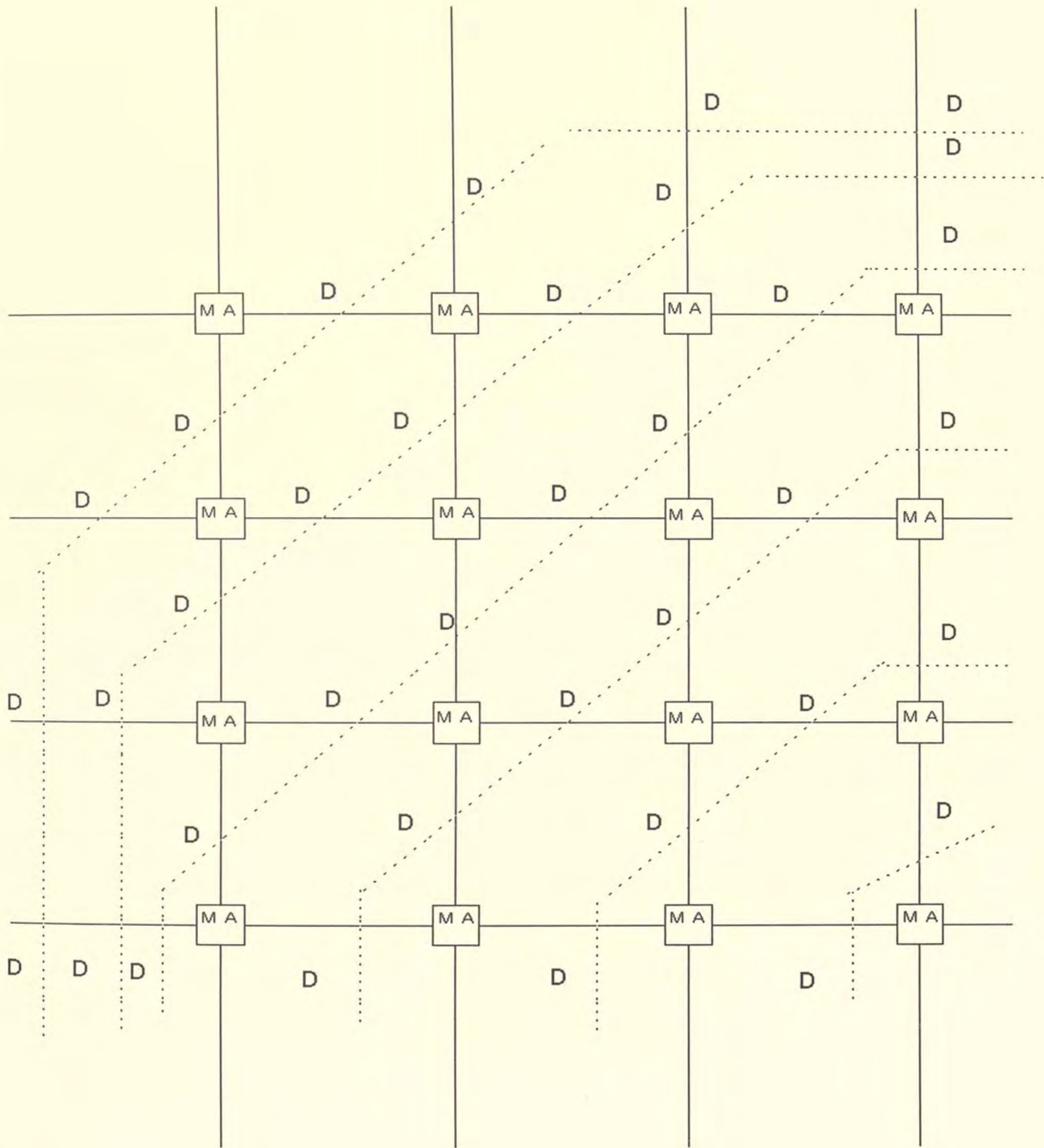


Figure 13. Matrix Systolization Process

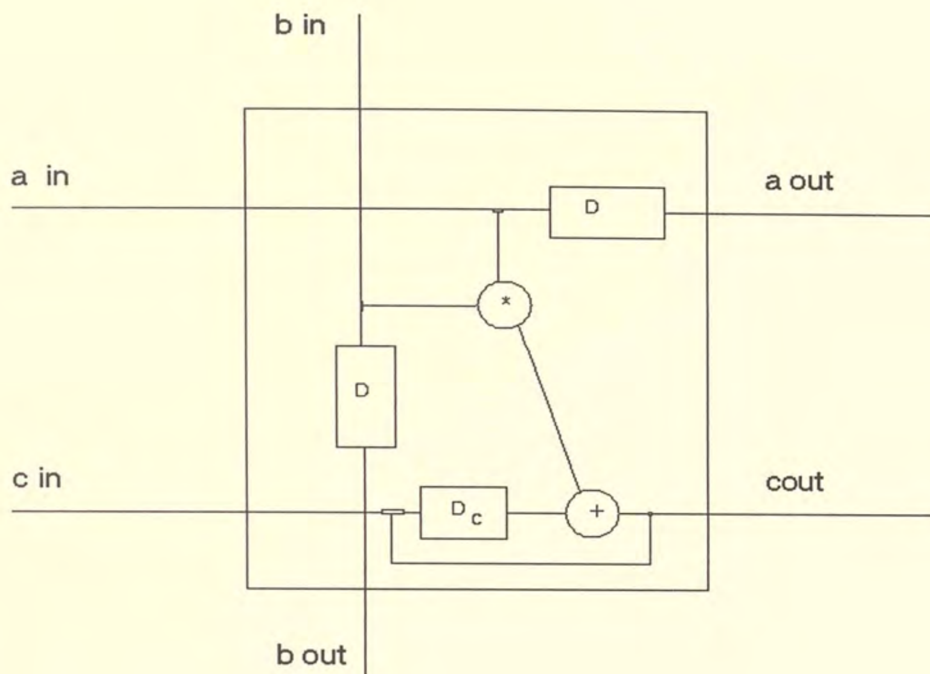


Figure 14. PE. for Matrix Multiplication

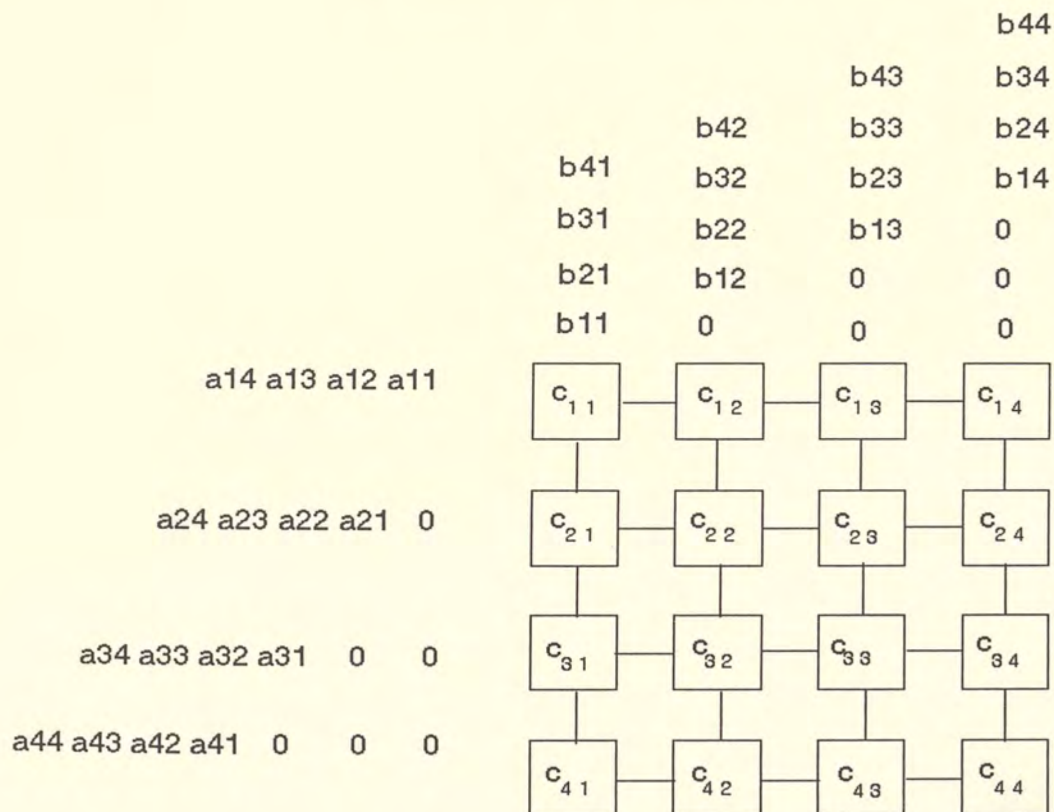


Figure 15. Array Processor for Matrix Multiplication

CHAPTER 3

VERIFICATION OF SYSTOLIZATION PROCESS

Because of the cost associated with the physical implementation of VLSI devices; it is important to be able to simulate the operation of a given architecture, so that the errors in the design phase can be easily corrected. The transformed SFG can be verified by analytical techniques or by simulation. The latter method is the most frequently used by digital architects. Snapshots of the transformed SFG reveal not only correctness at the behavioral level, but also other key measures such as throughput, response time, PE utilization etc. This chapter presents the development of a simulator for verifying the correctness of the systolization process. This newly developed simulation package will reference some work developed in [2] and uses the "C" programming language.

1. LANGUAGE SELECTION FOR THE SIMULATION PACKAGE

The selection of "C" is based on many criteria. First is the fact that the structures of "C" allows for a very flexible implementation of specific functions and thus the creation of a library of modelling parts (functions). Another consideration is the fact that the "C" programming language is a fairly new language with a very fast penetration in the scientific and engineer world. This

should ensure an easier understanding of the coding and availability of cost effective compilers.

"C" is a programming language that was designed by Dennis Ritchie of Bell Laboratories. It was created to be the systems language of the UNIX operating system. Today the Unix world is ever-expanding and is found widely on university campuses around the world. Because of its tight connection to the Unix operating system the "C" programming language is on the way to becoming one of the most important programing languages. However, this interconnection to Unix is not the only reason for the importance of "C". Its portability, size, elegance and power are other reasons of its wide success. "C" is easily portable and inexpensive compilers can be found in the market place (e.g., Borland turboC sells for under $100). The portability comes from omitting system dependent functions from the language. Libraries are created to include new utilities written to conform with the constraints associated with the new system. "C" is a small language which is another advantage when considering portability as lesser constructs need to be translated. The language is very powerful as it allows for any logical combination between structure arrays or enumeration types. Also a very important feature of "C" is the ability to perform address arithmetic using pointers. This allows for very flexible structures as well as great modularity when programming.

2.STRUCTURAL DEFINITION OF THE SIMULATION PACKAGE

The simulation package developed uses the dynamic memory allocation provided through the proper use of the "C" library (malloc) in order to increase the speed. Through a selective use of pointers the efficiency of the coding is tremendously increased. A reason for the use of pointers is that the number of inputs is not restricted as it is the case when using arrays. For instance arrays are bounded by a specific value while pointers can address an infinite number of locations. As a consequence any number of data structures are allowed to be stored in memory. The only limitation is function of the memory available in the computer and the storage capacity of the external devices. Microsoft "C" is used to develop this package.

3.PROGRAM LAYOUT

The simulator is a dedicated one in the sense that once the simulation requirements are set there is a need to recompile and assemble the code in order to run a different environment. However multiple sets of data can be simulated on a specific system without the need of the recompilation phase. The input and output to the simulation is achieved by reading and writing to files structured on a peripheral device. This is to shorten the input output stages of the simulation. Also, the user can by the means of another program create the information (input data) needed to set up the simulation environment. The output file can be used by

many commercially available plotting programs. The user can in this way obtain an accurate representation of the results.

The package is divided into three different and overlapping units: Input -> Processing -> Output. The first unit deals with the input function where the package creates the environment for the simulation from an input data file previously created by the user. This input data file is named Input.dat and is an ASCII file. The input format will be defined later. The second unit is the core of the simulator. In there is found the main program and a small library of parts that will be used to create the pipe. The control mechanism is investigated and the data paths are defined in the main file. The third part is the output of the results to an output data file identified during the run of the simulation. As established before, it is a convenient way to store the data as the user will have a variety of programs to process and analyze the results. The output of the simulation may be plotted using a commercial plotting program such as Omniplot.

4.PROGRAM DESCRIPTION

The programs used in this simulation have been implemented with provisions to increase the size of the pipe. Consequently the user can, within the input file, specify the number of stages within the pipe. The only restriction associated with the size of the pipe is the

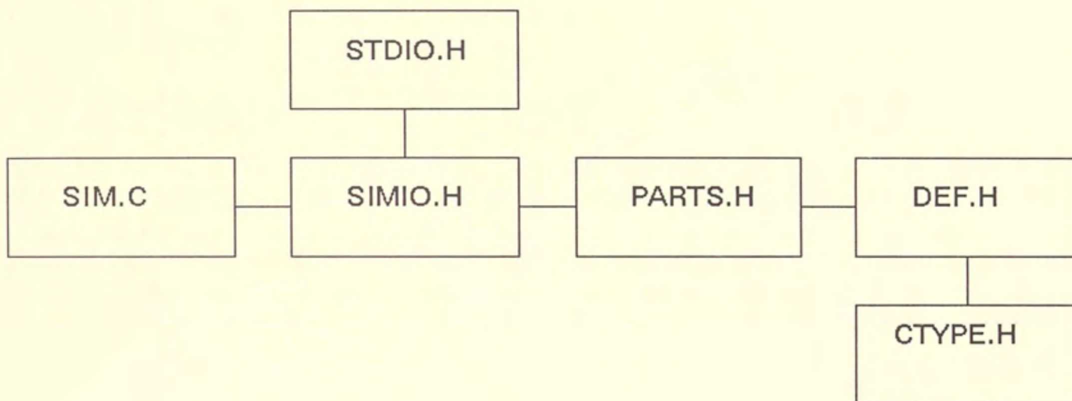


Figure 16. File Linking Structrure

memory size of the machine where the simulation is run. The package consists of four simulation files plus two standard header files. These files are interconnected through a selective use of the include statement as shown in Figure 16. From the point of view of the user, the only concern is to create the right environment within the input data file; and specify the proper data interconnection paths among the PEs in the main program. In other words, care must be taken to type the input.dat file with the correct format (see section e) and to develop a selected pipe architecture using the library of parts or new defined PE's. To run the program the user should type the name of the program that is to be executed. The results will be found in an output data file(s) whose name(s) is/are specified during the run of a program. From a programming point of view a brief description of each file is given in the following sections.

a)def.h

This file defines the data structures, constants and indexes used during the simulation. The indexes are provided for clarity and define a position within the array structure. A structure data_in is used to store all the information pertinent to the pipe for each clock cycle. The first field of this structure indicates the time at which the readings are being taken. In this version of the package this field will only be used for labelling purposes. However, future versions may want to include provisions to

make it part of the clocking mechanism. The second field specifies an array of data elements whose size is restricted by the datasize field found in the input.dat file. Two internal pointers next and prev are then defined and will be used during the link list process. If necessary they also can be used to take care of latency constraints associated with the different PE's in future versions of this package. However, through the course of this simulation, latency constraints are not used. The array identified as "bcte" holds any constant associated with the pipelining process. Note that label is another array whose function is to hold character strings that could be used to label the outputs. It is not used in this version of the simulation package.

b) parts.h

This is the library of parts used to create the different simulations. Four parts are included in the library: a virtual FIR PE (virtfir), a systolic FIR PE (forwfir), another systolic FIR PE (backfir) and a square matrix multiplier PE (matmult). This is the core of the program as the internal architecture of the different PE's are modelled according to the SFG specifications. All of the parts are implemented as functions which are called in the main program. The parameters which are passed reference first the time frame during which the part is used and the spatial links used to interconnect the PE function within the time frame. Also, some specific constants can be passed.

In resume the proper format to call a given function is illustrated below:

function-name(time frame,inconnect(s),cst(s),outconnect(s))

The software implementation of a given PE depends on its specific architecture and data can be moved within the same structure or a different one when temporal condition are encountered.

c) simio.h

This file manages the input output section of the program. It is composed of two different functions getdata and printdata.

The first function, referred to as getdata, reads data from an input file properly formatted and stores the data input in memory to allow for faster computational rates. The data input file is opened through the use of the statement fp= fopen("input.dat,"r"). The pointer fp, points to the file named input.dat and the mode indicated by r states that the file is open for reading. Using the function fscanf the data is transferred from the external file into the data structure data-in. Through the use of malloc, a block of memory of the size defined in data-in is allocated dynamically and the starting address of the memory block is assigned to the pointer, pdata. In order to link those newly created blocks of memory two internal pointers, next and prev of each structure, are used. As the name indicates, the pointer next will have the starting address of the next

block of memory. Also, the pointer prev will have the starting address of the previous block of memory. This process creates a double linkage between successive structures. Two extra pointers (HEAD and TAIL) are provided to indicate the first and last linked memory blocks. The function getdata is passed with a parameter n that defines the dimension of the array. An n=1 indicates that the data input is formatted for a uni-dimensional linear array. A n=2 is used to properly read the input data for a n*n 2 dimensional array. If necessary, new reading functions can be tailored to specific needs. A requirement will be to specify a different passing parameter.

The second function printdata is called with the passing parameter n (n=1,2,3). Its only objective is to write data to an output data file. In order to provide the opportunity to cascade and label a large number of PE's, the titles are written to the file using a control loop statement. The arguments of the function fprintf are: 1) the pointer to the file that indicate which file is to be written upon; 2) the control format that specifies in what fashion the data is to be written to the output.file and 3) the actual data. For each value of the passing parameter corresponds a different output format. A n=1 is used for one-dimensional arrays. While n=2 or n=3 are used for matrix representation. Based on specific needs, these functions can be tailored to a different output format.

d) Main File

This is the executable part of the simulation package and a .C extension is given to these files. Four example files are provided: FORWFIR.C, BACKFIR.C, VFIR.C and MATRIX.C. Each of these files creates a given pipe structure by calling the required PE functions and specifying correct data interconnections. Data interconnections are performed within control loops that allow for any number of cascaded PE's as specified in the input.dat file. In the examples the pipes are composed of identical PE's. However this file can accept any set of different PE's defined in parts.h and interconnect them according to the specification of the pipe. In addition, each file calls the function getdata and printdata for a proper I/O. If a specific architecture needs to be examined the user will have to establish the pipe structure by writing an executable file similar to the ones provided. The only restriction is to use the PE's developed in the parts.h file or to add new PE's.

e) Input Data File

This file is only concerned with creating the right environment for the simulation. It is the responsibility of the user to follow the format given below. Each field within a line is separated by a blank space.

LINE 1: Type in the datasize (integer). This specifies the number of external inputs to the array.

LINE 2: Type in the pipesize (integer). This indicates the

number of PE's in the pipe.

LINE 3: Type in the bctesize (integer). This indicates the number of constants used in the pipe.

Type in the constants (integer) associated with the simulation.

LINE 4: Type in the clock cycle (integer). This is used for labelling only. Future versions of this package will use it to assure a correct clocking mechanism.

Type in the data input (integers). The number of data inputs is restricted by datasize.

LINE 5: Repeat line 3 until all the data is passed.

LINE 6: File terminator. hit <CR>.

In the case of the matrix array lines 2 and 3 are omitted.

5.DATA STRUCTURE

The data input is stored in an array within the structure data-in. This stucture is addressed through the use of the pointer, pdata. The array is initialized with zeroes and intermediate results are stored at proper array locations. The time frame pointer pdata is always incremented by 1. This ensures a clocked process. It is important to realize that for each clock frame there is an associated structure containing all intermediate results. This allows for an accurate representation of the process for each clock cycle. For clarification, pt->data[#] indicates which array location within the structure must be accessed and the data at this location is used in

different computations.

When pt->next->data[#] is used it indicates that pt is pointing to the next structure at the address specified by data[#]. In short, next points to the data structure for the next clock cycle.

6.RESULTS

6.1 Finite Impulse Response filter (FIR).

The FIR filter was implemented in three distinct architectures: a virtual FIR, a forward FIR and a backward FIR. The terms forward and backward refer to the direction of the sum propagation.

6.1.1 Virtual Case

The first architecture referred to as the virtual VIRTFIR includes zero delay paths and is a direct implementation of the SFG of the FIR. It is a very fast machine as the data is immediately available at the output. This can be seen in the output data file by watching the broadcast of s1 across the pipe for clock 1. In this instance we are assuming that the multiplication-addition function is performed without delay. This idealized model is no longer true if the pipe is allowed to grow bigger as all the latencies of the individual function will be added and cannot be considered as negligible. The condition of spatial locality can be observed by watching the data flowing to the neighboring nodes. The rate of utilization

TABLE 2. VIRTUAL FIR SIMULATION RESULTS

clk	x0	x1	x2	x3	b1x1	b2x2	b3x3	s0	s1	s2	s3
1	1	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	1	0	0	0	1	1	1
3	3	2	1	0	2	1	0	0	2	3	3
4	4	3	2	1	3	2	1	0	3	5	6
5	5	4	3	2	4	3	2	0	4	7	9
6	6	5	4	3	5	4	3	0	5	9	12
7	7	6	5	4	6	5	4	0	6	11	15
8	8	7	6	5	7	6	5	0	7	13	18
9	9	8	7	6	8	7	6	0	8	15	21
10	0	9	8	7	9	8	7	0	9	17	24
11	1	0	9	8	0	9	8	0	0	9	17
12	2	1	0	9	1	0	9	0	1	1	10

of the pipe is maximal as all the PE's are used for a given clock cycle. The throughput is one clock cycle (ideal case) as it takes only one clock cycle to obtain a correct result at the output of the pipe. When compared to a sequential machine the speed up is 3. For a better vizualization, refer to Table 8 . This case was only presented to reference what would be the ideal case; however, due to delays associated with the MA's processes it is not wise to implement this architecture.

<u>6.1.2 Forward FIR</u>

In the systolic model the data is piped cyclically for both the x's and the sums. As a consequence, no major timing consideration needs to be addressed except for the fact that a clock skew may induce some errors if the pipe is very long. This simulation and the following one illustrate the fact that different SFG mappings lead to distincts architectures with different performances. In this case the two delay path in the x's can be observed in Table 3 by noting that the first available digit, a 1 in this case is passed to x1 only at clock cycle #3. Therefore there is a delay of 2 clock cycles. Also, the one delay in the sum path can be seen by observing the propagation of the 1 in the sums from clock cycle 3 to 5. In terms of performances the utilization rate of the PE's is maximal as all PE's are operating at each clock cycle. The throughput of the system once the pipe is filled is one clock cycle. The only

TABLE 3. FORWARD FIR SIMULATION RESULTS

clk	x0	x1	x2	x3	b1x1	b2x2	b3x3	s0	s1	s2	s3
1	1	0	0	0	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0	0	0	0
3	3	1	0	0	1	0	0	0	1	0	0
4	4	2	0	0	2	0	0	0	2	1	0
5	5	3	1	0	3	1	0	0	3	3	1
6	6	4	2	0	4	2	0	0	4	5	3
7	7	5	3	1	5	3	1	0	5	7	6
8	8	6	4	2	6	4	2	0	6	9	9
9	9	7	5	3	7	5	3	0	7	11	12
10	0	8	6	4	8	6	4	0	8	13	15
11	1	9	7	5	9	7	5	0	9	15	18
12	2	0	8	6	0	8	6	0	0	17	21

TABLE 4. BACKWARD FIR SIMULATION RESULTS

clk	x0	x1	x2	x3	b1x1	b2x2	b3x3	s0	s1	s2	s3
1	1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	1	0	0	0	0
3	2	0	1	0	0	1	0	0	0	0	1
4	0	1	0	2	1	0	2	0	0	1	0
5	3	0	2	0	0	2	0	0	1	0	3
6	0	2	0	3	2	0	3	0	0	3	0
7	4	0	3	0	0	3	0	0	2	0	6
8	0	3	0	4	3	0	4	0	0	5	0
9	5	0	4	0	0	4	0	0	3	0	9
10	0	4	0	5	4	0	5	0	0	7	0
11	6	0	5	0	0	5	0	0	4	0	12
12	0	5	0	6	5	0	6	0	0	9	0

difference when compared to the ideal case is the latency of the pipe which is now 4 clock cycles as compared to only one clock cycle.

6.1.3 Backward FIR

The need to interleave the input data stream with zeroes reduces the performance of the machine as a correct output is available only every two clock cycles. Analyzing Table 4 it can be seen that the utilization rate is reduced to 50%. A major improvement to this scheme will be to include provisions for multiplexing so that a full utilization of the pipe can be achieved. In other words, the PE's that are not currently used at a given clock cycle create a virtual path that can be used by another input data stream. It is easy to see that the performance of the backward case is lower than the forward case. The result of this architecture is a lower throughput, a lower utilization rate, and a speed up of 1.5.

6.2 Square Matrix Multiplier

In this simulation the multiplication of two square matrix A and B is investigated.

$$C = A * B$$

$$A = B = \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix}$$

The result of this simulation can be found in two independent data files. The first one emphasizes the concurrent activities in the pipe for a given clock cycle.

TABLE 5. MATRIX SIMULATION SNAPSHOTS

```
                              clk time  1
propagation of matrix A
    1     0     0     0
    0     0     0     0
    0     0     0     0
    0     0     0     0
propagation of matrix B
    1     0     0     0
    0     0     0     0
    0     0     0     0
    0     0     0     0
       matrix C
    0     0     0     0
    0     0     0     0
    0     0     0     0
    0     0     0     0


                              clk time  2
propagation of matrix A
    2     1     0     0
    5     0     0     0
    0     0     0     0
    0     0     0     0
propagation of matrix B
    5     2     0     0
    1     0     0     0
    0     0     0     0
    0     0     0     0
       matrix C
    1     0     0     0
    0     0     0     0
    0     0     0     0
    0     0     0     0
```

```
                              clk time  3
propagation of matrix A
    3    2    1    0
    6    5    0    0
    9    0    0    0
    0    0    0    0
propagation of matrix B
    9    6    3    0
    5    2    0    0
    1    0    0    0
    0    0    0    0
       matrix C
   11    2    0    0
    5    0    0    0
    0    0    0    0
    0    0    0    0


                              clk time  4
propagation of matrix A
    4    3    2    1
    7    6    5    0
   10    9    0    0
   13    0    0    0
propagation of matrix B
   13   10    7    4
    9    6    3    0
    5    2    0    0
    1    0    0    0
       matrix C
   38   14    3    0
   35   10    0    0
    9    0    0    0
    0    0    0    0


                              clk time  5
propagation of matrix A
    0    4    3    2
    8    7    6    5
   11   10    9    0
   14   13    0    0
propagation of matrix B
    0   14   11    8
   13   10    7    4
    9    6    3    0
    5    2    0    0
       matrix C
   90   44   17    4
   98   46   15    0
   59   18    0    0
   13    0    0    0
```

```
                              clk time  6
propagation of matrix A
     0     0     4     3
     0     8     7     6
    12    11    10     9
    15    14    13     0
propagation of matrix B
     0     0    15    12
     0    14    11     8
    13    10     7     4
     9     6     3     0
        matrix C
    90   100    50    20
   202   116    57    20
   158    78    27     0
    83    26     0     0


                              clk time  7
propagation of matrix A
     0     0     0     4
     0     0     8     7
     0    12    11    10
    16    15    14    13
propagation of matrix B
     0     0     0    16
     0     0    15    12
     0    14    11     8
    13    10     7     4
        matrix C
    90   100   110    56
   202   228   134    68
   314   188    97    36
   218   110    39     0


                              clk time  8
propagation of matrix A
     0     0     0     0
     0     0     0     8
     0     0    12    11
     0    16    15    14
propagation of matrix B
     0     0     0     0
     0     0     0    16
     0     0    15    12
     0    14    11     8
        matrix C
    90   100   110   120
   202   228   254   152
   314   356   218   116
   426   260   137    52
```

```
                          clk time  9
propagation of matrix A
    0    0    0    0
    0    0    0    0
    0    0    0   12
    0    0   16   15
propagation of matrix B
    0    0    0    0
    0    0    0    0
    0    0    0   16
    0    0   15   12
       matrix C
   90  100  110  120
  202  228  254  280
  314  356  398  248
  426  484  302  164


                          clk time 10
propagation of matrix A
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0   16
propagation of matrix B
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0   16
       matrix C
   90  100  110  120
  202  228  254  280
  314  356  398  440
  426  484  542  344


                          clk time 11
propagation of matrix A
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
propagation of matrix B
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
       matrix C
   90  100  110  120
  202  228  254  280
  314  356  398  440
  426  484  542  600
```

TABLE 6. MATRIX SIMULATION RESULTS

clk	a 1	a 2	a 3	a 4	a 5	a 6	a 7	a 8	a 9	a10	a11	a12	a13	a14	a15	a16
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	5	0	0	0	0	0	0	0	0	0	0	0
3	3	2	1	0	6	5	0	0	9	0	0	0	0	0	0	0
4	4	3	2	1	7	6	5	0	10	9	0	0	13	0	0	0
5	0	4	3	2	8	7	6	5	11	10	9	0	14	13	0	0
6	0	0	4	3	0	8	7	6	12	11	10	9	15	14	13	0
7	0	0	0	4	0	0	8	7	0	12	11	10	16	15	14	13
8	0	0	0	0	0	0	0	8	0	0	12	11	0	16	15	14
9	0	0	0	0	0	0	0	0	0	0	0	12	0	0	16	15
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

clk	b 1	b 2	b 3	b 4	b 5	b 6	b 7	b 8	b 9	b10	b11	b12	b13	b14	b15	b16
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	5	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	9	6	3	0	5	2	0	0	1	0	0	0	0	0	0	0
4	13	10	7	4	9	6	3	0	5	2	0	0	1	0	0	0
5	0	14	11	8	13	10	7	4	9	6	3	0	5	2	0	0
6	0	0	15	12	0	14	11	8	13	10	7	4	9	6	3	0
7	0	0	0	16	0	0	15	12	0	14	11	8	13	10	7	4
8	0	0	0	0	0	0	0	16	0	0	15	12	0	14	11	8
9	0	0	0	0	0	0	0	0	0	0	0	16	0	0	15	12
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

clk	c 1	c 2	c 3	c 4	c 5	c 6	c 7	c 8	c 9	c10	c11	c12	c13	c14	c15	c16
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	11	2	0	0	5	0	0	0	0	0	0	0	0	0	0	0
4	38	14	3	0	35	10	0	0	9	0	0	0	0	0	0	0
5	90	44	17	4	98	46	15	0	59	18	0	0	13	0	0	0
6	90	100	50	20	202	116	57	20	158	78	27	0	83	26	0	0
7	90	100	110	56	202	228	134	68	314	188	97	36	218	110	39	0
8	90	100	110	120	202	228	254	152	314	356	218	116	426	260	137	52
9	90	100	110	120	202	228	254	280	314	356	398	248	426	484	302	164
10	90	100	110	120	202	228	254	280	314	356	398	440	426	484	542	344
11	90	100	110	120	202	228	254	280	314	356	398	440	426	484	542	600

Table 6 gives snapshots of the data propagation within the matrix. It is a very convenient way to observe the correct propagation of the data along the paths as well as the computational process. It should be noted that this architecture emcompasses both parallelism and overlap.

This can be seen by observing the cut sets derived in Figure 13 as the data propagates as a wave in the array structure and multiple PE's are operating at each edge of the wave. The throughput of the multiplier is 11 clock cycles. This is the time it takes for all the data to be computed.

7.CONCLUSION

The simulator is a useful tool to observe the correctness of a given systolic array. It is a very convenient way to simulate the SFG to architecture step. However this simulator can be improved by adding new parts to the library and by creating specific functions that could give an on screen information of the performances of the architecture simulated. It is interesting to note that this program has some common characteristics with a commercial language OCCAM used to program "off the shelf" array parts into a specific array architectures.

Table 8 : Performance characteristics

ARCHITECTURE MODEL	CLOCK RATE	TROUGHPUT	LATENCY	SPEEDUP	INTERLEAF #
Virtual FIR	10MHz	1micros	1micros	3	0
Forward FIR	10MHz	1micros	4micros	3	0
Backward FIR	10Mhz	2micros	2micros	1.5	1
Matrix Mult	10MHz	11micros	5micros		0

Future work on this package include writing an additional program to create the input data file. The main objective of that program is to prompt the user with the right sequence of questions, so that the keyed in inputs can be written directly to the input data file. The output process should be made more flexible by including provisions to print in the output data file labels which will have been previously typed in the input data file sequences. Note that the label array is already included in the package to perform this objective. Finally the program should be made more user friendly by making it menu driven.

CHAPTER 4

CONTROL STRATEGIES

Systolic arrays are based on parallelism and pipelineability but still emphasize the classical approach of sequencing the processes by a global clock. Wavefront array processors in contrast are data driven machines which means that processes are executed as soon as all operands are available. This new technique goes one step beyond as no global clock is required to sequence the operations. As a result the level of performances is increased.

1. TIMING CONSIDERATIONS

Highly parallel structures consist of many interconnected PEs operating simultaneously. To have a proper propagation of the data, some restrictions must be associated with the control schemes. Systolic arrays use a global timing scheme while wavefront array processor (WAP) use asynchronous control schemes based on handshaking techniques. In the following paragraphs we will address some of the advantages and limitation of both control schemes.

1.1 Synchronization in a Globally Timed System

When using a global clock data is rhythmically passed along the pipe at a constant frequency. For small systems (a few cascaded PE's) this scheme is preferred as the

control mechanism is simple, cost effective and external to the PE.

However, as the system grows physically bigger, a global clock is difficult to implement because of problems associated with clock skew. Clock skew refers to the problem of a global clock signal arriving at individual PE's at different times. The time differences are due to delays in the clock path. For instance, the time required for the clock signals to propagate on the wires is not instantaneous. Due to the characteristics of the wire (i.e., resistivity, capacitance) a diffusion delay occurs; for large systems this cannot be ignored. For reference, the diffusion equation is given below as : RC(dV/dt) = d(dV/dt)/dt. Its solution is complex but it can be stated that the time for a transient to propagate a distance x is proportional to x**2. Because of fan out problems a clock signal is generally distributed in a tree stucture with each branch consisting of a chain of inverters. Again a problem can be seen as the propagation delay through each inverter will cause a clock skew. For large systems those added delays will cause a collapse of the synchronization. Therefore synchronous systems present timing problems in the case of a large array implementation.

1.2 Synchronization in a Self-Timed System

In the self-timed or asynchronous case, each PE starts to compute as soon as all the incoming data from the

previous cells are present at its input. The control logic associated with this scheme is more complex and is achieved by handshaking between processors.

A simple two line handshaking scheme includes the familiar signals, ready (R) and acknowledge (A). The transfer of data between two processors is accomplished by the source processor raising its ready line when data is available and the target processor raising the acknowledge line when the data has been accepted. In a WAP, each processing element "fires" or raises its ready line when all the operands are ready and the results are available, thus data flows at a maximum rate. A negligible time T is contributed by the control mechanism. Thus, the major delays are due to actual processing times of the processor elements.

WAP are better suited for large array implementation as there is no problem associated with clock distribution. It has the added advantage that the time to propagate from one cell to the other is independent of the pipe length but instead depends on the latency of each individual PE; thus allowing for a higher throughput rate.

2. WAVEFRONT ARRAY

A wavefront array is a computing network with almost the same characteristics as expressed for systolic arrays.

2.1 Modularity and Local Interconnections

The array consists of a regular set of PEs with local interconnections. The spatial locality is dictated by constraints associated with VLSI implementation. The size of the array can be extended indefinitely as the timing is not a constraint.

2.2 Control

The sequencing of the data through the system is achieved in a self-timed environment based on handshaking techniques. This allows for faster throughput rate as different PE's may have different latencies. In contrast the maximum clock rate for systolic arrays is function of the slowest stage in the pipe.

2.3 Speed up

The speed up is linear as for the case of systolic arrays.

It is important to realize that the condition of temporal locality is no longer necessary as no specific timing reference exists in a data driven environment. The name wavefront array comes from the fact that each PE acts as a secondary source and is responsible for the propagation of the wavefront.

3. TRANSFORMATION OF A SFG TO A WAP

An SFG is transformed into a WAP by partitioning the SFG into interconnected PE's. Delay operators associated with incoming operands are realized by storage devices within the

PE. The PE control mechanism synchronizes the use of inputs and the availability of outputs.

4. EXAMPLES OF SFG TO WAP TRANSFORMATION

Because a WAP is a direct implementation of an SFG, the only concern lays with including the correct control mechanism within the given PE. In other words the handshaking protocol should be clearly defined. Below is given the pseudocode of such a controller for an FIR PE.

```
1.  if READY1 = 0  go to 5
    else
        Latch data
        raise ACK1
        READY1 = 0
        ACK1 = 0


5. if READY2 = 0 go to 1
   else
       Latch data
       raise ACK2
       READY2 = 0
       ACK2 = 0
```

READY1 identifies available data on the x line
READY2 identifies available data on the sum line

CHAPTER 5

CONCLUSION

Systolic arrays are network of processor elements interconnected in a regular and local manner and synchronized with a global clock. The importance of systolic arrays lays in the improved throughput achieved when computational intensive algorithms are mapped into these architectures. Systolic arrays are particularly suited for real time signal processing which provides very fast data input rates due to to a high sampling frequency. However, not all real time algorithms are suited for a systolic implementation. For instance, these algorithms must be recursive and perform simple and, if possible, identical operations so that modularity can be achieved. One must understand that most systolic arrays are hardware implementations of a given algorithm. Special purpose systolic arrays are currently being developed but the complexity associated with the reconfiguration of the pipe have reduced their cost effectiveness.

Because the structure of the array is strongly dependent on the recursive algorithm, there is an emphasis on mapping complex algorithms into SFG's. The advantage of the SFG is a better understanding of the inherent properties of the algorithm. Also, this graphical representation lends itself

to a systolization procedure based on the principle of cut sets. The systolization methodology was presented in Chapter 2 and resulted in spatial and temporal locality. As noted in the examples, the condition of temporal locality involved time rescaling of the inputs and outputs and for some cases the necessity to interleave the inputs with a number of zeros.

As it is important to verify the correctness of the data path through the array structure a simulator using the "C" programing language was developed. In addition to providing snapshots of the activity of the pipe, it provides observation of performances measures such as throughput, utilization rate, latency and speedup. This is presented in Chapter 3.

In large systems synchronization problems may arise due to clock skew. In order to avoid catastrophic synchronization failures, an asynchronous scheme can be adopted at a higher hardware cost. In this scheme, also referred to as self-timed, a given processor element performs its computational requirement on the express condition that all incoming data be present at its inputs. This control mechanism requires the definition of a handshaking protocol between PE's. The trade-off between both systems involve cost and efficiency. In a systolic version, the control mechanism is very simple and thus less expensive. However the throughput is function of the

slowest stage in the pipeline. On the other side, wavefront array processor, have dedicated control elements within each PE to perform the handshaking mechanism. The advantage of these arrays is an improved throughput as it is not dependent on the delay of one given stage. This advantage shrinks if the array is implemented with an identical PE. This could lead to a slower throughput because of the need to wait for the control mechanism to be finished.

Improvement of systolic arrays performances are strongly dictated by advances in VLSI/WSI. Their regularity and modularity provide for very efficient area layouts.

APPENDIX A
TUTORIAL

TUTORIAL

The construction and running of a simulation which uses a new processing element and a new interconnection scheme is presented.

1.1 New Processing Element

If a new processing element is needed the user should:
1) Edit the file parts.h and include a software model of the processor element following the specifications of the signal flow graph (see Figures 17, 20). The model should respect the package format (see examples in section 2).
2) Create a main program where the pipe architecture is specified. This file should be labelled with a .c extension. The object of this file is to interconnect the different processor elements, according to a selected pipe architecture (see example in section 2).

1.2 New Pipe Structure

If a new pipe is desired the new interconnection path should be specified by creating a new main program.

1.3 Running The Program

After editing the proper files the program should be compiled and linked via a C compiler/linker software package. Once the executable file is created the user only needs only to type the file name in order to have a simulation run. Note that in Simio.h the printdata function can be customized to have a specific output format.

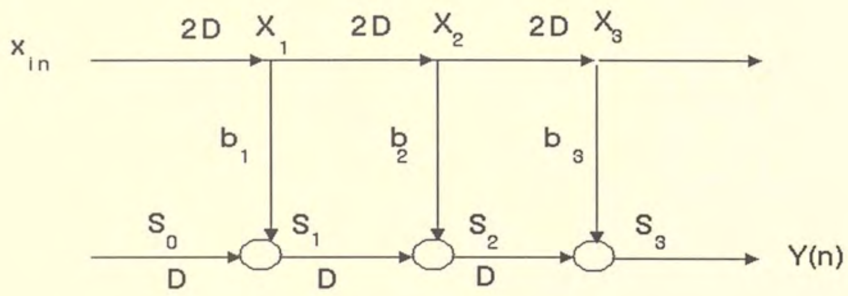


Figure 17. Forward Systolized FIR

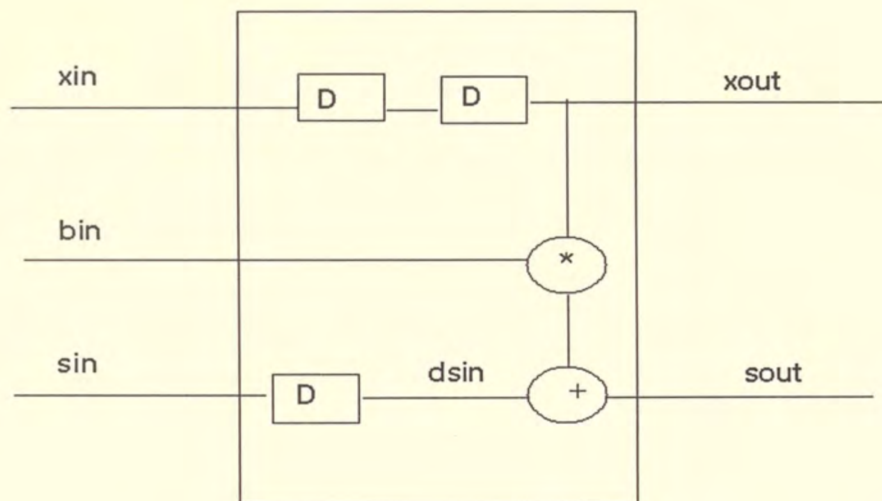


Figure 18. PE for Forward FIR Filter

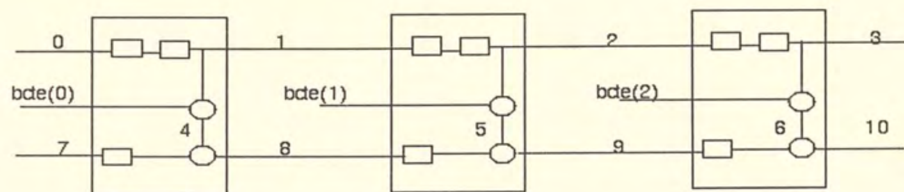


Figure 19. Systolic Forward FIR Filter

2 Examples

Two examples are presented: -Forward FIR

-Matrix Multiplication

2.1 Forward FIR

The PE architecture is defined in parts.h and the pipe structure is created in FORWFIR.C

2.1.1 PE Architecture

The SFG model is shown in Figure 17 and the PE architecture (see Figure 18) is identified by creating the function forwfir in parts.h. A proper listing of the code is shown in Table 8.

TABLE 8

```
1 forwfir(pt,xin,sin,bin,xout,sout,bxout)
2  int xin,sin,bin,xout,sout,bxout;
3  struct data_in *pt;
      {
4     pt->data[bxout] = bcte[bin] * pt->data[xout];
5     pt->data[sout]=pt->data[dsin]+pt->data[bxout];
6     pt->next->next->data[xout]=pt->data[xin];
7     pt->next->data[dsin]=pt->data[sin];
      }
```

xin,sin,xout,sout,bxout represent data locations within the array data. Remark that next is a time frame pointer. As a consequence delays identified in the PE architecture are implemented using this pointer (see lines 6 and 7). Because sout is not reused between consecutive time frame the intermediate delayed value of sin (dsin) can be replaced by sout. Although it is not a correct representation of the physical process the software model allows for this slight transgression.

The following comments apply:

Line 1 -The function parameters are identified, they consist of:

--pt : time frame pointer

--xin : x input to forwfir

--sin : sum input to forwfir

--bin : an internal constant used in the PE

--xout : x output of forwfir

--sout : sum output of forwfir

--bxout : the result of an internal computation within forwfir

Line 2 -This is a local declaration of the parameter list

Line 3 -This is a local declaration of pt being a pointer to struct data_in

Line 4 -The architecture of the PE is now defined

-bxout = constant * xout

Line 5 -sout = sout + bxout

Line 6 -xout = xin^{2D}

Line 7 -sout = sin^{1D}

Remark that lines 4 to 6 give a mathematical representation of the computational model. Line 6 identifies a delay of two clock cycles in the x path while line 7 represents a delay of 1 clock cycle in the sum path.

2.1.2 Pipe Structure (main program)

Due to the implementation of the simulator each time frame contains the total information of the pipe. This pipe consists of three identical PE's found in parts.h (see Figure 19). The proper code format is identified below :

```
/* this is the FIR program for the forward case
  for bigger pipes define INPUTSIZE as: ((3*pipesize) + 2)
*/
#define INPUTSIZE 11
#define CTESIZE 3
#include "simio.h"
main()
{
getdata(1);

pt=head;

while (pt->next != NULL)
   {
    forfir (pt, 0, 7, 0, 1, 8, 4);
    forfir (pt, 1, 8, 1, 2, 9, 5);
    forfir (pt, 2, 9, 2, 3, 10, 6);
      }
   pt=pt->next;
   }
printdata(1);
}
```

The following comments apply :

line 1 define statements

-INPUTSIZE total number of variables in the pipe

-CTESIZE total number of internal constants in the pipe

line 2 include statements for proper linking

line 3 call getdata to input data

line 4 set pointer to first time frame.

line 5 time frame terminator criterion

line 6 interconnection as follows from the SFG :

```
forfir (pt, 0, 7, 0, 1, 8, 4)
forfir (pt, 1, 8, 1, 2, 9, 5)
forfir (pt, 2, 9, 2, 3, 10, 6)
```

The interconnections can be seen by observing that xout for forwfir i is equal to xin for forwfir i+1.

line 7 printdata

To make the pipe structure more general control loops are included to increase the pipe size as shown in the following listing.

```
/* this is the FIR program for the forward case
  for bigger pipes define INPUTSIZE as: ((3*pipesize) + 2)
*/
#define INPUTSIZE 11
#define CTESIZE 3
#include "simio.h"
main()
{
getdata(1);

pt=head;

while (pt->next != NULL)
   {
   i=0;
   k= i + pipesize +1;
   l= k + pipesize;
   m=0;
   for(j=0;j<pipesize;j++)
      {
      ni = 1+i;
      nl = 1+l;
      forfir(pt,i,l,m,ni,nl,k);
      m++;
      k++;
      i++;
      l++;
      }
   pt=pt->next;
   }
printdata(1);
}
```

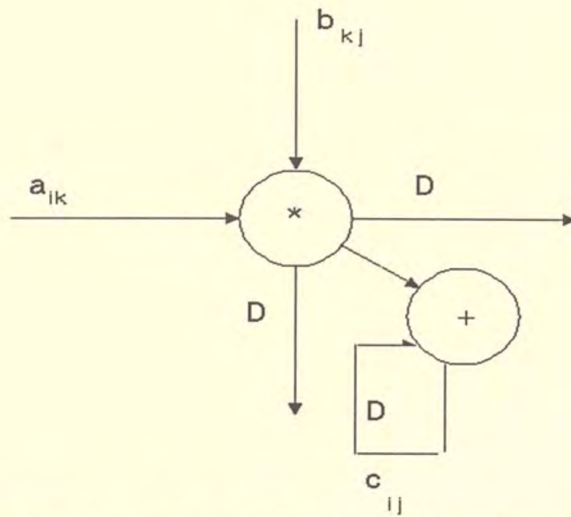


Figure 20. S F G For Matrix Multiplication PE

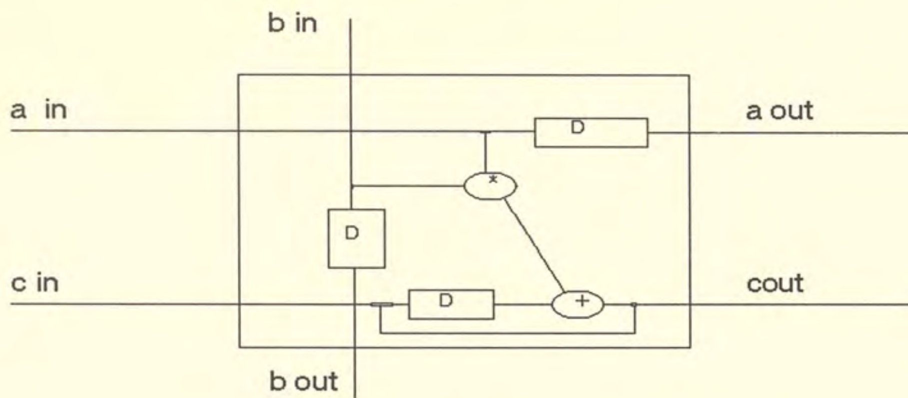


Figure 21. PE. for matrix multiplication

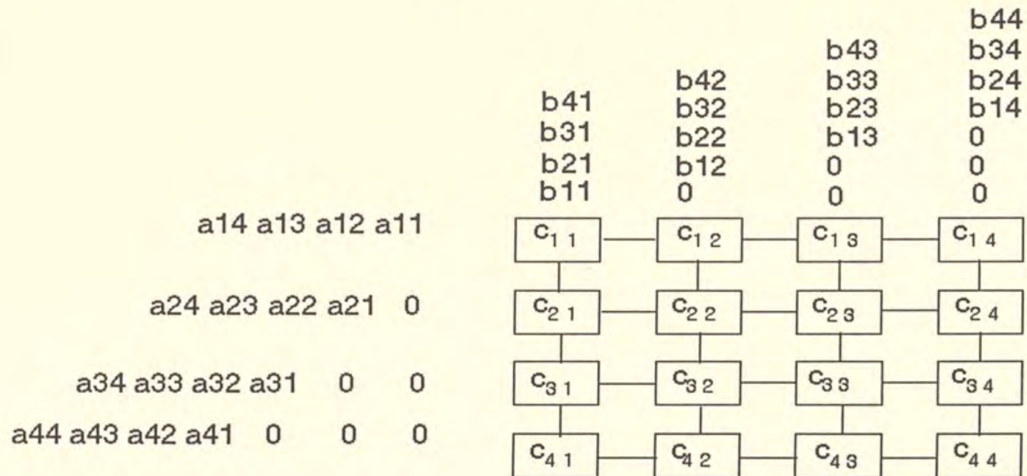


Figure 22. Array processor for Matrix Multiplication

2.1 Matrix Multiplier

The PE architecture is defined in parts.h and the pipe structure is created in Matrix.c

2.1.1 PE Architecture

The SFG model of an individual PE is shown in Figure 20 the corresponding PE architecture (see Figure 21) is identified by creating the function matmult in parts.h. A proper listing of the code is shown in Table 9.

TABLE 9

```
1 matmul(pt,ain,bin,cin,aout,bout,cout)
2 int ain,bin,cin,aout,bout,cout;
3 struct data_in *pt;
    {
4   pt->data[cout]=pt->data[cin]
                   + (pt->data[aout]) * (pt->data[bout]);
5   pt->next->data[aout]=pt->data[ain];
6   pt->next->data[bout]=pt->data[bin];
7   pt->next->data[cin]=pt->data[cout];
    }
```

ain,bin,cin,aout,bout,cout represent data locations within the array data. The following comments apply :

Line 1 -The function parameters are identified, they consist of:

--pt : time frame pointer

--ain : horizontal input to matmult

--bin : vertical input to matmult

--cin : an internal intermediate term used in the PE

--aout : horizontal output of matmult

--bout : vertical output of matmult

--cout : contains the result of the internal

computation within matmult.

Line 2 -This is a local declaration of the parameter list

Line 3 -This is a local declaration of pt being a pointer to struct data_in

Line 4 -The architecture of the PE is now defined

-cout = cin + (aout * bout)

Line 5 -aout = ain^{1D}

Line 6 -bout = xin^{1D}

Line 7 -cout = cin^{1D}

Remark that lines 4 to 7 give a mathematical representation of the computational model. Lines 5, 6 and 7 identify a time delays of 1 clock cycle for each out parameter.

2.1.2 Pipe Structure (main program)

Due to the implementation of the simulator each time frame contains the total information of the pipe. This pipe consists of sixteen identical PE's (see Figure 22) found in parts.h. The proper code format of this main file is shown below.

```
/* this program multiplies two square matrix of size n
  data input is in integer if the matrix size is going
to be incremented INPUTSIZE is (3*n + 2*n) */

1 #define INPUTSIZE 58
1 #define CTESIZE 3
2 #include "simio.h"

  main()
  {
3 getdata(2);
```

```
4   pt=head;

5 while (pt->next != NULL)
      {

6      matmult (pt, 1, 1,37, 2, 5,37)
6      matmult (pt, 2, 2,38, 3, 6,38)
6      matmult (pt, 3, 3,39, 4, 7,39)
6      matmult (pt, 4, 4,40, 53,8,40)
6      matmult (pt, 5, 5,41, 6, 9,41)
6      matmult (pt, 6, 6,42, 7, 10,42)
6      matmult (pt, 7, 7,43, 8, 11,43)
6      matmult (pt, 8, 8,44,54, 12,44)
6      matmult (pt, 9, 9,45, 10, 13,45)
6      matmult (pt, 10,10,46,11, 14,46)
6      matmult (pt, 11,11,47,12,15,47)
6      matmult (pt, 12,12,48,55, 16,48)
6      matmult (pt, 13,13,49,14,17,49)
6      matmult (pt, 14,14,50,15,18,50)
6      matmult (pt, 15,15,51,16,19,51)
6      matmult (pt, 16,16,52,56, 20,52)

      }

7 printdata(2);
7 printdata(3);

   }
```

The following comments apply.

line 1 define statements

-INPUTSIZE total number of variables in the pipe

-CTESIZE reserves space for bcte array

line 2 include statements for proper linking

line 3 call getdata to input data

line 4 set pointer to first time frame.

line 5 time frame terminator criterion

line 6 interconnection as follows from the SFG :

```
matmult (pt, 1, 1,37, 2, 5,37)
matmult (pt, 2, 2,38, 3, 6,38)
matmult (pt, 3, 3,39, 4, 7,39)
matmult (pt, 4, 4,40, 53,8,40)
matmult (pt, 5, 5,41, 6, 9,41)
```

```
matmult (pt, 6, 6,42, 7, 10,42)
matmult (pt, 7, 7,43, 8, 11,43)
matmult (pt, 8, 8,44,54, 12,44)
matmult (pt, 9, 9,45, 10, 13,45)
matmult (pt, 10,10,46,11, 14,46)
matmult (pt, 11,11,47,12,15,47)
matmult (pt, 12,12,48,55, 16,48)
matmult (pt, 13,13,49,14,17,49)
matmult (pt, 14,14,50,15,18,50)
matmult (pt, 15,15,51,16,19,51)
matmult (pt, 16,16,52,56, 20,52)
```

The interconnections can be seen by observing that aout for matmult i is equal to ain for matmult i+1 and that bout for matmult i is equal to bin for matmult i+1.

line 7 printdata

To make the pipe structure more general control loops are included to increase the pipe size as shown in the code below.

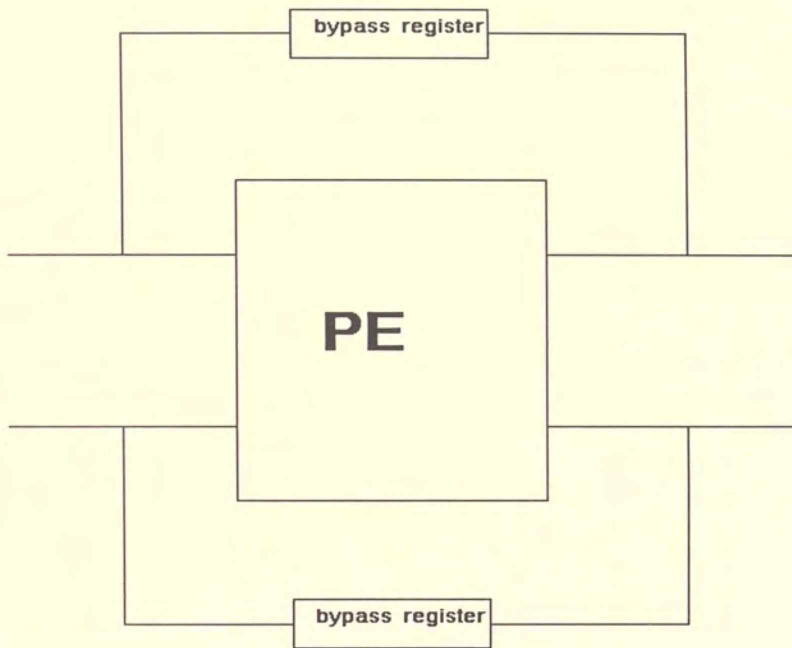
```
/* this program multiplies two square matrix of size n
  data input is in integer if the matrix size is going
to be incremented INPUTSIZE is (3*n + 2*n) */

#define INPUTSIZE 58
#define CTESIZE 10
#include "simio.h"

main()
{
getdata(2);
    pt=head;
while (pt->next != NULL)
   {
   outa= 3*matlength + matsize + 1;
   i=1;
   j=matlength+1;
   k=(2*matlength)+ matsize +1;
   for(m=0;m<matsize;m++)
      {
      for(l=1;l<=matsize;l++)
         {
         if (l<matsize)
           {
           ni = i+1;
           nj = j+4;
```

```
          matmul(pt,i,j,k,ni,nj,k);
          i++;
          j++;
          k++;
          }
        if (l==matsize)
          {
          ni = i+1;
          nj = j+4;
          matmul(pt,i,j,k,outa,nj,k);
          i++;
          j++;
          k++;
          }
        }
      outa++;
      }
   pt=pt->next;
   }
printdata(2);
printdata(3);
}
```

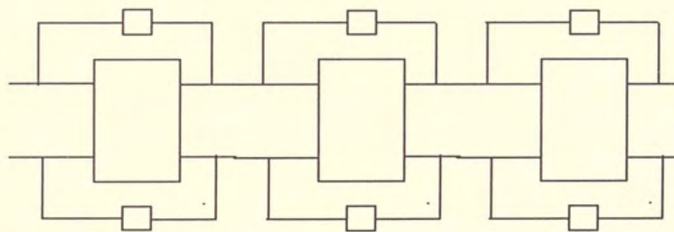
APPENDIX B
FAULT TOLERANCE

FAULT TOLERANCE

Because of the intrinsic nature of an array architecture the overall performance of the pipe must not be affected by the occurrence of faulty processors. If a fault tolerant scheme is not devised by the VLSI engineer such occurrences can be catastrophic for the entire pipeline and will result in higher costs. The fault tolerant scheme must be resolved by the VLSI engineer in such a fashion that the overall performance of the array will not be degraded. A solution to that problem is to bypass all faulty processors. An example is given in Figure 23 for the case of a linear uni direction array. When a faulty processor is encountered the data is fed through the bypass register delaying it one unit cycle time. Thus to the overall system it appears as if the iteration i was not performed. Another scheme will be to reroute faulty cells to operating ones during the testing phase. This solution implies higher costs because of the rerouting process but will not degrade the performance of the array. Others schemes exists for different architecture and it must be the responsibility of the engineer to include fault tolerance in its final design.



a) PE Bypass Register Scheme



b) Array with Fault Tolerance Provision

Figure 23. Fault Tolerance Scheme for Unidirectional Array

APPENDIX C

INPUT FILES & PROGRAM LISTINGS

INPUT DATA FILE FOR FIR SIMULATION
(VIRTUAL AND FORWARD CASE)

```
1
3
3 1 1 1
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 0
11 1
12 2
9 9
```

INPUT DATA FILE FOR FIR SIMULATION
(BACKWARD CASE)

```
1
3
3 1 1 1
1 1
2 0
3 2
4 0
5 3
6 0
7 4
8 0
9 5
10 0
11 6
12 0
9 9
```

MATRIX DATA INPUT

```
8
1    1    0    0    0    1    0    0    0
2    2    5    0    0    5    2    0    0
3    3    6    9    0    9    6    3    0
4    4    7   10   13   13   10    7    4
5    0    8   11   14    0   14   11    8
6    0    0   12   15    0    0   15   12
7    0    0    0   16    0    0    0   16
8    0    0    0    0    0    0    0    0
9    0    0    0    0    0    0    0    0
10   0    0    0    0    0    0    0    0
11   0    0    0    0    0    0    0    0
12   8    8    8    8    8    8    8    8
```

```
/* this is the FIR program for the virtual case
  for bigger pipes define INPUTSIZE as: ((3*pipesize) + 2)
*/
#define INPUTSIZE 11
#define CTESIZE 10
#include "simio.h"
main()
{
getdata(1);

pt=head;

while (pt->next != NULL)
   {
   i=0;
   k= i + pipesize +1;
   l= k + pipesize;
   m=0;
   for(j=0;j<pipesize;j++)
      {
      ni = 1+i;
      nl = 1+l;
      virtualfir(pt,i,l,m,ni,nl,k);
      m++;
      k++;
      i++;
      l++;
      }
   pt=pt->next;
   }
printdata(1);
}
```

```
/* this is the FIR program for the forward case
  for bigger pipes define INPUTSIZE as: ((3*pipesize) + 2)
*/
#define INPUTSIZE 11
#define CTESIZE 10
#include "simio.h"
main()
{
getdata(1);

pt=head;

while (pt->next != NULL)
   {
   i=0;
   k= i + pipesize +1;
   l= k + pipesize;
   m=0;
   for(j=0;j<pipesize;j++)
      {
      ni = 1+i;
      nl = 1+l;
      forfir(pt,i,l,m,ni,nl,k);
      m++;
      k++;
      i++;
      l++;
      }
   pt=pt->next;
   }
 printdata(1);
 }
```

```
/* this is the FIR program for the backward case
  for bigger pipes define INPUTSIZE as: ((3*pipesize) + 2)
*/
#define INPUTSIZE 11
#define CTESIZE 10
#include "simio.h"
main()
{
getdata(1);

pt=head;

while (pt->next != NULL)
   {
   i=0;
   ni = pipesize;
   k= 2*pipesize;
   l= 3*pipesize + 1;
   m= 0;
   for(j=0;j<pipesize;j++)
      {
      nl = l-1;
      backfir(pt,i,l,m,ni,nl,k);
      m++;
      i = ni;
      ni--;
      k--;
      l--;
      }
   pt=pt->next;
   }
printdata(1);
}
```

```
/* this program multiplies two square matrix of size n
  data input is in integer if the matrix size is going
to be incremented INPUTSIZE is (3*n + 2*n) */

#define INPUTSIZE 58
#define CTESIZE 10
#include "simio.h"

main()
{
getdata(2);

    pt=head;

while (pt->next != NULL)
   {
   outa= 3*matlength + matsize + 1;
   i=1;
   j=matlength+1;
   k=(2*matlength)+ matsize +1;
   for(m=0;m<matsize;m++)
      {
      for(l=1;l<=matsize;l++)
         {
         if (l<matsize)
           {
           ni = i+1;
           nj = j+4;
           matmul(pt,i,j,k,ni,nj,k);
           i++;
           j++;
           k++;
           }
         if (l==matsize)
           {
           ni = i+1;
           nj = j+4;
           matmul(pt,i,j,k,outa,nj,k);
           i++;
           j++;
           k++;
           }
         }
      outa++;
      }
   pt=pt->next;
   }

 printdata(2);
 printdata(3);

 }
```

```
/* FILENAME: SIMIO.H

   This  is  the input output file that reads and writes  to
   the data files

*/

#include <stdio.h>
#include "parts.h"
  FILE *fp,*fpo, *fopen(), *fclose();

linkdata()
        {
        if (head == NULL)
         {
         head=pdata;
         tail=pdata;
         pdata->prev=NULL;
         pdata->next=NULL;
         }
        else
         {
         pdata->prev=tail;
         tail->next=pdata;
         tail=pdata;
         pdata->next=NULL;
         }
       }

getdata(n)
int n;

  {
  fp = fopen("input.dat","r");

  printf("%s\n","simulation running please wait");

  fscanf(fp,"%d", &datasize);

/*
  fscanf(fp,"%d", &labelsize);
  for(j=0;j<labelsize;j++) fscanf(fp," %s",label[j]);
*/

  if(n==1)
     {

 printf("%s\n","the results are to be found in output.dat");

      fscanf(fp,"%d", &pipesize);

      fscanf(fp,"%d", &bctesize);
```

```
      for(j=0;j<bctesize;j++) fscanf(fp,"%d",&(bcte[j]));

      head=NULL;
      tail=NULL;

      while(fscanf(fp,"%d",&c) != EOF)
        {
  pdata = (struct data_in *) malloc(sizeof(struct data_in));
        pdata->time = c;
        for(j=0;j<datasize;j++)
         fscanf(fp," %d",&(pdata->data[j]));
        for(j=datasize;j<INPUTSIZE;j++)
         (pdata->data[j]=0);

        linkdata();
        }
      fclose(fp);
      }

   if(n==2)
     {
     printf("%s\n","simulation running please wait");
 printf("%s\n","the results are to be found in output2.dat");
     printf("%s\n","and output3.dat for snapshots");

     head=NULL;
     tail=NULL;
     matsize = datasize / 2;
     matlength = matsize*matsize;
     while(fscanf(fp,"%3d",&c) != EOF)
       {
  pdata = (struct data_in *) malloc(sizeof(struct data_in));
       pdata->time = c;

       for(j=0;j<INPUTSIZE;j++) (pdata->data[j]=0);
       i =1;
       for(j=0;j<matsize;j++)
         {
         fscanf(fp," %d",&(pdata->data[i]));
         i = i+4;
         }
       k = (matsize*matsize) + 1;
       for(j=0;j<matsize;j++)
         {
         fscanf(fp," %d",&(pdata->data[k]));
         k++;
         }
       linkdata();
       }
     fclose(fp);
     }
   }
```

```
  /* ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ */
  /* ^                    OUTPUT SECTION                       ^ */
  /* ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ */

printdata(n)
int n;
  {
  if(n==1)
    {
    fpo = fopen("output.dat","w");

    fprintf(fpo,"%5s", "clk");
    for(j=0;j<=pipesize;j++) fprintf(fpo,"%4s%d", "x",j);
    for(j=1;j<=pipesize;j++) fprintf(fpo,"%2s%d%s%d","b",j,"x",j);
    for(j=0;j<=pipesize;j++) fprintf(fpo,"%4s%d", "s",j);
    fprintf(fpo,"\n");

    pdata=head;

    while (pdata->next != NULL)
     {
     fprintf(fpo,"%5d",pdata->time);
for(j=0;j<INPUTSIZE;j++) fprintf(fpo,"%5d", pdata->data[j]);
     fprintf(fpo,"\n");
     pdata=pdata->next;
     }
    printf("bye bye\n");
    }

  if(n==2)
    {
       fpo = fopen("output3.dat","w");
       pdata=head;
    while (pdata->nex!= NULL)
       {
       fprintf(fpo,"%30s%3d\n", "clk time",pdata->time);
       fprintf(fpo,"%20s\n", "propagation of matrix A");
       k=1;
          for(l=0;l<matsize;l++)
          {
            for(j=0;j<matsize;j++)
              {
              fprintf(fpo,"%5d", pdata->data[k]);
              k++;
              }
          fprintf(fpo,"\n");
          }

       fprintf(fpo,"%20s\n", "propagation of matrix B");
         k=matlength +1;
```

```
      for(l=0;l<matsize;l++)
        {
          for(j=0;j<matsize;j++)
            {
            fprintf(fpo,"%5d", pdata->data[k]);
            k++;
            }
        fprintf(fpo,"\n");
        }

     fprintf(fpo,"%15s\n", "matrix C");
        k=(matlength*2)+5;
        for(l=0;l<matsize;l++)
        {
          for(j=0;j<matsize;j++)
            {
            fprintf(fpo,"%5d", pdata->data[k]);
            k++;
            }
        fprintf(fpo,"\n");
        }
    fprintf(fpo,"\n");
    fprintf(fpo,"\n");
    pdata=pdata->next;
    }
  }

if(n==3)
  {
    fpo = fopen("output2.dat","w");
    pdata=head;
    fprintf(fpo,"%5s", "clk");
   for(j=1;j<=matlength;j++) fprintf(fpo,"%3s%2d", "a",j);
    fprintf(fpo,"\n");

  while (pdata->next != NULL)
    {
       k=1;
       fprintf(fpo,"%5d",pdata->time);
       for(j=0;j<matlength;j++)
         {
         fprintf(fpo,"%5d", pdata->data[k]);
         k++;
         }
       fprintf(fpo,"\n");

       pdata=pdata->next;
    }
  fprintf(fpo,"\n");
  fprintf(fpo,"\n");

  fprintf(fpo,"%5s", "clk");
```

```
 for(j=1;j<=matlength;j++) fprintf(fpo,"%3s%2d", "b",j);
 fprintf(fpo,"\n");


 pdata = head;

 while (pdata->next != NULL)
    {
       k=matlength +1;
       fprintf(fpo,"%5d",pdata->time);
       for(j=0;j<matlength;j++)
          {
          fprintf(fpo,"%5d", pdata->data[k]);
          k++;
          }
       fprintf(fpo,"\n");

       pdata=pdata->next;
    }
 fprintf(fpo,"\n");
 fprintf(fpo,"\n");

 fprintf(fpo,"%5s", "clk");
 for(j=1;j<=matlength;j++) fprintf(fpo,"%3s%2d", "c",j);
 fprintf(fpo,"\n");

 pdata=head;

 while (pdata->next != NULL)
    {
       k=(matlength*2)+5;
       fprintf(fpo,"%5d",pdata->time);
       for(j=0;j<matlength;j++)
          {
          fprintf(fpo,"%5d", pdata->data[k]);
          k++;
          }
       fprintf(fpo,"\n");

       pdata=pdata->next;
    }
    printf("bye bye\n");
  }
}
```

```
/* FILENAME: PARTS.H */

#include "def.h"

/* This is the library of parts used for parallel processing
   simulation. Each part is implemented in a function. Each
   processor element is assumed to have a latency of 1 word

        FIR IMPLEMENTATION AND MATRIX MULTIPLICATION

These  functions implements a Finite Impulse Response  (FIR)
filter   (virtualfir,   forwfir,   backfir)  and  a   matrix
multiplier (matmult).
*/

virtualfir(pt,xin,sin,bin,xout,sout,bxout)
  int xin,sin,bin,xout,sout,bxout;
  struct data_in *pt;
    {
    pt->data[bxout] = bcte[bin] * pt->data[xout];
    pt->data[sout]=pt->data[sin]+pt->data[bxout];
    pt->next->data[xout]=pt->data[xin];
    }

forwfir(pt,xin,sin,bin,xout,sout,bxout)
  int xin,sin,bin,xout,sout,bxout;
  struct data_in *pt;
    {
    pt->data[bxout] = bcte[bin] * pt->data[xout];
    pt->data[sout]=pt->data[sout]+pt->data[bxout];
    pt->next->next->data[xout]=pt->data[xin];
    pt->next->data[sout]=pt->data[sin];
    }

backfir(pt,xin,sin,bin,xout,sout,bxout)
  int xin,sin,bin,xout,sout,bxout;
  struct data_in *pt;
    {
    pt->data[bxout] = bcte[bin] * pt->data[xout];
    pt->next->data[sin]=pt->data[sout]+pt->data[bxout];
    pt->next->data[xout]=pt->data[xin];
    }

matmul(pt,ain,bin,cin,aout,bout,cout)
int ain,bin,cin,aout,bout,cout;
struct data_in *pt;
   {
pt->data[cout]=pt->data[cin]+(pt->data[aout])*(pt->data[bout]);
   pt->next->data[aout]=pt->data[ain];
   pt->next->data[bout]=pt->data[bin];
   pt->next->data[cin]=pt->data[cout];
   }
```

```
#include <ctype.h>
/* FILENAME: DEF.H
  This file defines the data structures and constants used
  for the simulation. It is the declaration part of the main
  program.
*/

#define namesize 20
#define OUTSIZE 10

struct data_in
   {
   int time;
   int data[INPUTSIZE];
   struct data_in *next;
   struct data_in *prev;
   } data_array;

struct data_in *pdata,*head,*tail,*pt;

char a[10],b[10],label[10][12];
int c,i,datasize,labelsize,bctesize,pipesize,ni,nl;
int matsize,matlength,nj,l,m,time,j=0,k,l,m,outa;
int bcte[CTESIZE];
```

APPENDIX D

MODULARIZING AN SFG

Digital signal processing involves the sampling of a continuous process. This example shows that through the manipulation of a SFG modularity can be achieved. For the purpose of the presentation an ARMA filter (IIR) is presented. The filter transfer function is given by :

$$H(z) = \frac{\sum_{k=1}^{N} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}}$$

This equation can be rewritten as

$$y(n) = \sum_{k=1}^{N} x(n-k) b_k + \sum_{k=1}^{N} y(n-k) a_k$$

It becomes apparent by analyzing this equation that we need to remember the history of the previous samples in terms of both output and inputs. As a consequence we need to include provisions for storage or delay of those values. By inspection of the algorithm we can derive the direct form 1 flow graph shown in Figure 24. It is important to realize that the value is broadcast along the nodes. Since the coefficients b_k correspond to the numerator polynomial and the coefficients a_k correspond to the denominator. We can redraw the S.F.G. as a cascade of the denominator and numerator circuits. (see Figure 25). To eliminate the redundancy in the use of delays we combine the delays as shown in Figure 26 . By inspection we can redraw this last

figure to come to the representation shown in Figure 27. At this stage of the design we need to apply the systolization procedure in order to have a localizable S.F.G.

First we redraw the circuit as shown in Figure 28. This is a straight representation of Figure 27. Given the fact that the array is regular; cut sets can be selected (dashed lines in Figure 29. Applying the localization rules we need to rescale the delays associated with the representation assume D' = 2D. Now by subtracting one delay to the left bound edges and adding one delay to the right bound edges we yield to Figure 30. This complete the systolization procedure as we have locality in both time and space.

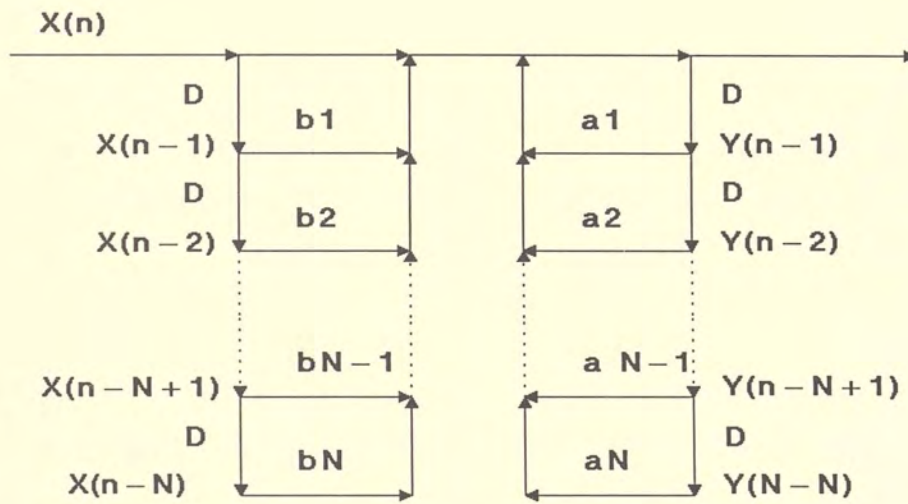


Figure 24. S.F.G. Arma Filter Direct Form 1

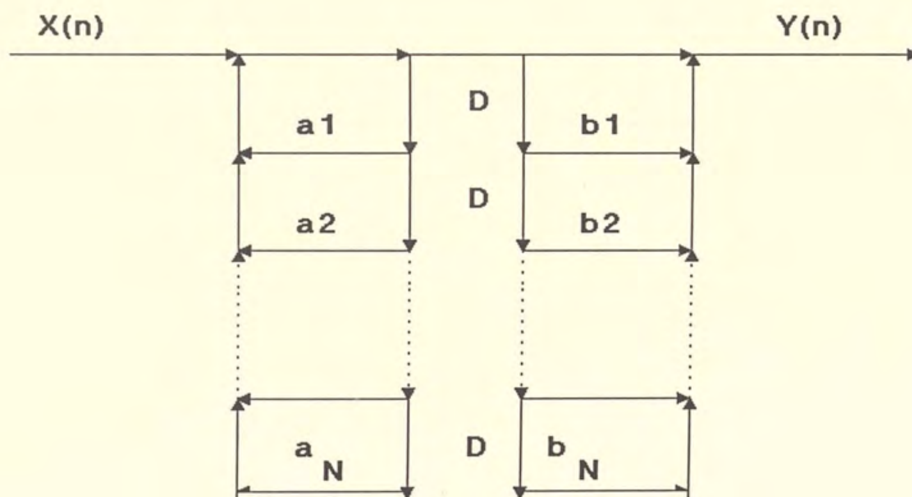


Figure 25. First Manipulation of the SFG

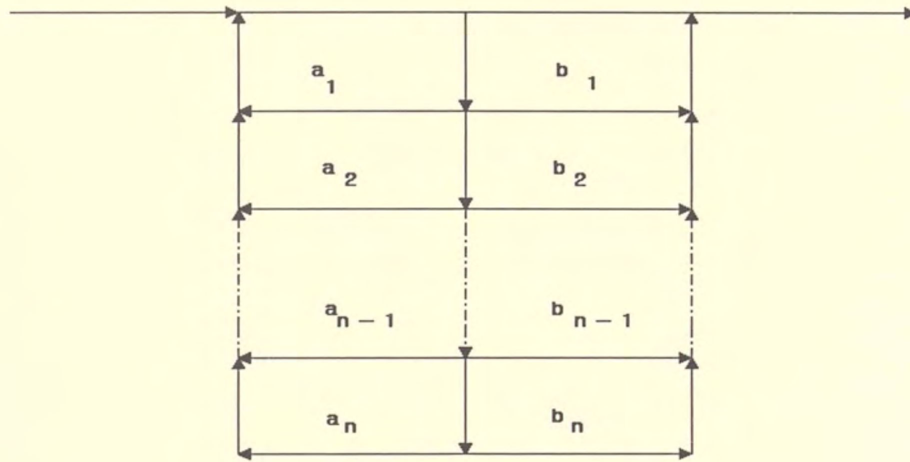


Figure 26. SFG Direct Form 2.

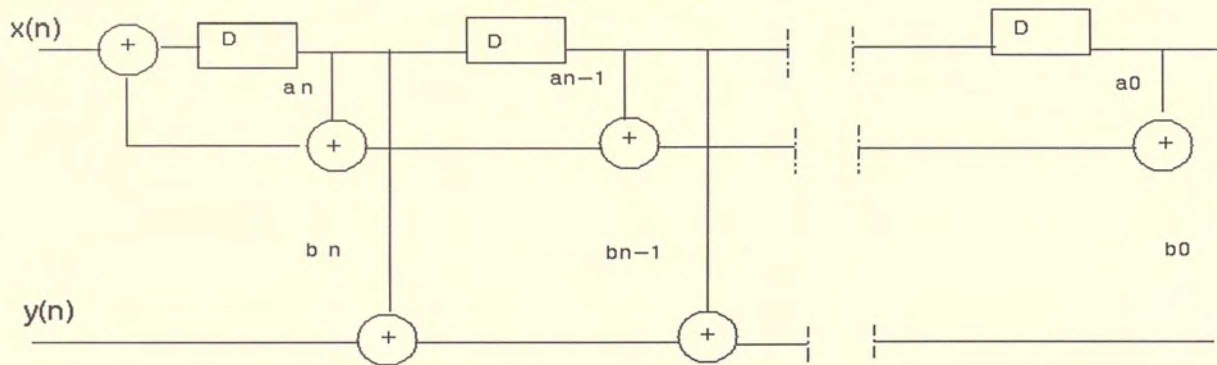


Figure 27. Linear Representation of Figure 26

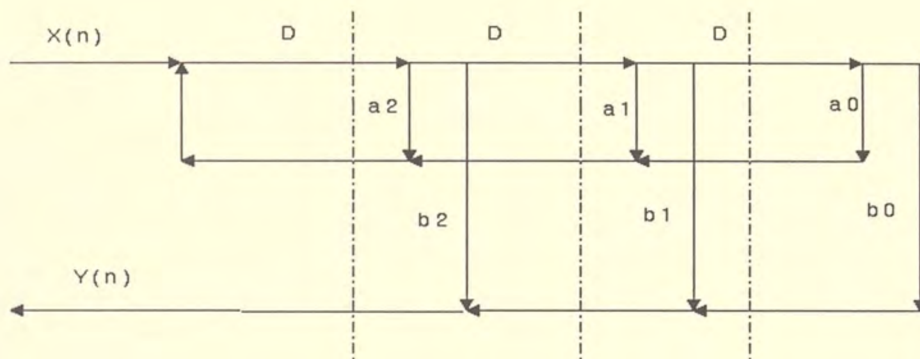


Figure 28. Arma Filter Cut Set

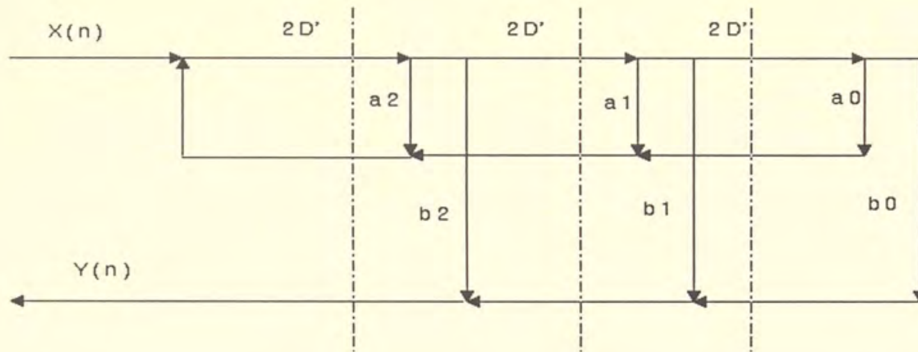


Figure 29. ARMA Cut Set & Time Rescaling

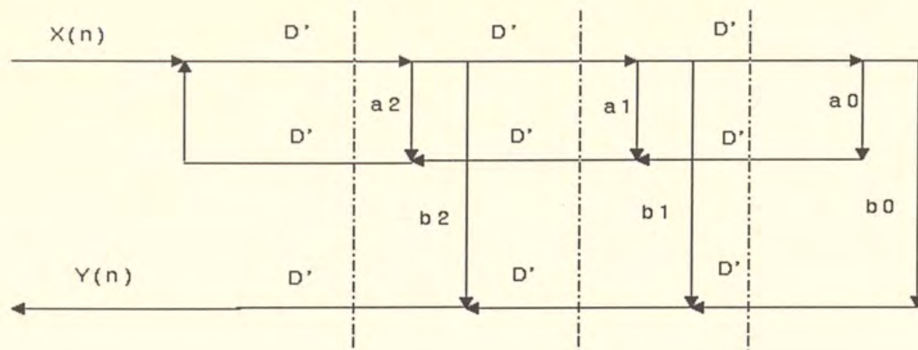


Figure 30. Systolic ARMA Filter Array

REFERENCES

[1] De Man,H.; Rabaey,J; Six,P.; and Claesen,L. "Cathedral-2 A Silicon Compiler for Digital Signal Processing", EEE ASSP, December 1986.

[2] Denyer,P. and Renshaw,D. VLSI Signal Processing: A Bit Serial Approach, Addison-Wesley Publishing Company, 1985.

[3] Fisher,A. and Kung,H.T. "Synchronizing large VLSI processor arrays", presented at the 10th. Annual International Symposium on computer architecture, 1983, Stockolm, Sweden.

[4] Franklin,M. and Wann,D. "Asynchronous and clocked control structures for VLSI based interconnection networks", presented at the 9th Annual Symposium on Computer Architecture, April 1982, Austin, TX.

[5] Glasser,L.A. and Dobberpuhl,D.W. The design and analysis of VLSI circuits, Addison-Wesley Publishing Company, 1985.

[6] Hwang,T. and Briggs,F. Computer Architectures and Parallel Processing, New York: Mc Graw-Hill, 1984.

[7] Kung,S.Y. "VLSI Array Processors", IEEE ASSP, pp. 4-22, July 1985.

[8] Kung,S.Y. "On Supercomputing with Systolic/Wavefront Array Processors", Proceedings IEEE, Volume 72, Number 7, pp. 531-548, July 1984.

[9] Mead,C. and Conway,L. "Introduction to VLSI systems", Addison-Wesley Publishing Company, 1980.

[10] Savitzky,T.Real time signal microprocessor systems, Van Nostrand Reinhold Company, New York, 1985.