A new general purpose systolic array for matrix computations

Hai Van Dinh Le
Portland State University

AN ABSTRACT OF THE THESIS OF Hai Van Dinh Le for the Master of Sciences in Electrical Engineering presented May 18, 1988.

Title: A New General Purpose Systolic Array for Matrix Computations.

APPROVED BY MEMBERS OF THE THESIS COMMITTEE:

[redacted]

Marek Perkowski, Chairman

[redacted]

Rajinder Aggarwal

[redacted]

Erasto Kashoro

[redacted]

Mohammad Ghafarzade

It has been conservatively estimated that 75 percent of all scientific applications involve some form of matrix computations. In general, matrix computations are very expensive in term of processing time. For real time operation required by such applications as robotics, signal processing and computer graphics animation, the processing power of serial computers is simply inadequate.

In this thesis, we propose a new systolic architecture which is based on the Faddeev's algorithm. Because Faddeev's algorithm is inherently general purpose, our architecture is able to perform a wide class of matrix computations. And since the architecture is systolic based, it brings massive parallelism to all of its computations. As a result, many matrix operations including addition, multiplication, inversion, LU-decomposition, transpose, and solutions to linear systems of equations can now be performed extremely fast. In addition, our design introduces several concepts which are new to systolic architectures:

- It can be re-configured during run time to perform different functions with the uses of various control signals propagating throughout the arrays.
- It allows for maximum overlaps of processing between consecutive computations, thereby increasing system throughput.

There have been other architectures proposed for this problem. However, a thorough analysis performed in this thesis reveals that they suffer from serious drawbacks, design inefficiencies or even errors. Thus, they are impractical for actual implementation. On the other hand, the new architecture is free from all of these weaknesses while offering many important advantages, some of which are listed as follows:

- It is truly problem size independent, i.e. matrices which are arbitrarily large can be easily decomposed to be processed by a fixed size array.
- It can solve sparse matrix problems efficiently without requiring system re-configuration.
- It provides the same level of performance as the known architectures using a smaller number of cells and arrays.
- It is fully expansible, i.e. linear performance improvement can be achieved by simple addition of identical component arrays.
- Because of its simplicity, it can be implemented inexpensively and with very little effort.

We also describe in this thesis several extensions to Faddeev's algorithm which are ideally suited for problem size independent systolic architectures such as ours. These extensions—classified as horizontal, vertical, and two-dimensional—not only increase a system throughput from two to four fold but also enhance the inherent programmability of Faddeev's algorithm. This allows our architecture to perform very complex matrix calculations. An example of this enhanced programmability for complex matrix calculation is presented as well.

A NEW GENERAL PURPOSE SYSTOLIC ARRAY FOR MATRIX COMPUTATIONS

by

HAI VAN DINH LE

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCES
in
ELECTRICAL ENGINEERING

Portland State University

1988

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Hai Van Dinh Le presented May 18, 1988.

[redacted]

Marek Perkowski, Chairman

[redacted]

Rajinder Aggrawal

[redacted]

Erasto Kashoro

[redacted]

Mohammad Ghafarzade

APPROVED:

[redacted]

Lee Casperson, Chair, Department of Electrical Engineering

[redacted]

Bernard Ross, Vice Provost for Graduate Studies

ACKNOWLEDGMENTS

Faddeev's algorithm became infinitely clearer after Dr. Robert Broussard illustrated a simple example. For this, and also for the several office visits during which he patiently answered my technical questions, I am sincerely grateful.

I also wish to give thanks to Dr. Roy Rathja for providing me with some important articles on the subject of parallel computer architectures. They have been quite valuable and are greatly appreciated.

Special thanks are due to Dr. Marek Perkowski, my thesis advisor, who has suffered the indignation of many long hours going through several "final" versions of this thesis; his extensive detailed comments and suggestions have helped me fix innumerable errors and omissions.

Finally, I would like to express my deepest gratitude to my wife and best friend, Tuyet Uong, for her devotion and unwavering support through all these years. Without her, my entire higher education would have been impossible. This thesis is lovingly dedicated to her.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER I

INTRODUCTION

As a general class of problems, matrix computations are found to be very useful, if not essential, within a broad spectrum of scientific applications. However, they are generally expensive in terms of storage space and processing time. To be sure, numerous algorithms with substantially reduced storage requirement have been devised for specific matrix computations. Yet, it is with the recent abundance of low cost memory that storage demands of matrix computations in general cease to be an important issue. On the other hand, the need for greater throughput rate has become more acute as applications grew in power and complexity. Indeed, for real time operation required in such applications as robotics, signal processing and computer animation, the computing power of serial computers proved to be woefully inadequate. Before long, it was evident that the only way to meet the ever growing computational requirements of many applications is to build faster systems.

WAYS AND OBSTACLES IN SPEEDING UP DIGITAL SYSTEMS

Essentially, there are two ways to build faster systems. One is to use fast components, the other is to use concurrency.[1] Since the technological trend clearly indicates that we are reaching the maximum components speed potential, any major gain in computational speed must come from the concurrent use of many processing elements.[1,2,3,4] As it is, the architecture of conventional computers suffers from two inherent difficulties:

(1) Long communication paths such as buses between CPU and its memory substantially slow down the transmission of information. Also, the system I/O bandwidth provides an absolute upper bound limit on the data rate, which acts as a bottleneck in limiting the system speed.[4,5,6,7]

(2) The single CPU sequentially fetches and executes instructions thereby does not fully exploit its hardware resources at all times, providing little or no concurrency for speeding up processing.[5,6,7]

Indeed, the above problems are widely acknowledged for quite some time. Nevertheless, they remain to be formidable obstacles which must be surmounted before a substantial speed increase is obtained. Several schemes were proposed to address one or both of them while maintaining the same degree of generality offered by the von Neumann architecture. Most widely known among them are:

- pipelining,
- memory-caching,
- replicating CPU's processing units (such as adders, multipliers, ALUs),
- and multiprocessor systems.[4,7]

Pipelining, as a form of parallelism, involves the application of assembly line techniques to improve the performance of an arithmetic or control unit.[7,8,9] Theoretically, maximum utilization of available components can be achieved if the pipeline is kept full at all time. However, in actual operation this ideal condition is impossible to maintain and speed gains occur only in burst between pipeline flushes. Although widely implemented on many high-speed systems today, pipelining does not fully exploit the parallelism inherent in many applications and only constitutes a minor architectural fine tuning of the basic von Neumann structure.[7,10]

Memory caching is used to reduce the cost of memory and alleviate the communication bottleneck at the expense of additional system complexities. A memory cache is a small but high-speed memory system that tries to capitalize on temporal locality, the theory of which basically states: if a particular instruction or piece of data is read from memory, then the probability of it being used again increases. Thus, after the cache is filled with instructions or data brought in from slower memory, the number of subsequent reads by the CPU which can be performed at full speed to the cache increases before access to slower memory is required. The effectiveness of a cache memory is known as the "hit ratio." Given a certain number of instructions (or data) that must be fetched, the hit ratio is the number that can be accessed from the cache versus how many that must be accessed from slower memory. Generally, the cache employs the highest-performance technology—bipolar;[4] however with the performance of CMOS technology steadily bridging the gap and its cost declining, the trade-off seems less attractive. Furthermore, while caches seem to work well with single processor computers, they are difficult to incorporate into multiprocessor systems because of the *cache coherence* problem. Cache coherence relates to the integrity of data between various caches within a system. Suppose a two processor system is tightly coupled through a main memory but each processor has its own data cache. A different routine is running on each processor and the two tasks communicate through the shared memory. If, however, a shared address is present in both caches and the individual processors read and write that address, then each processor would not have the same piece of data in its respective cache. This results in neither processor seeing the changes caused by the other.[7]

Of course there are schemes to remove this problem, but they invariably add further complexity to the system. Thus, while partially improving the performance of the von Neumann architecture, pipelining and caching create other problems of their own.

At the other end, we have systems with *replicated processing units* or *multiple processors* which incorporate a very high degree of parallelism while striving to retain the same level of generality available in von Neumann architecture; however, run-time considerations such as tasks synchronization and memory contention incur rather severe system overhead.[7,11,12] Thus, full utilization of available hardware can never be realized.[8,12]

Simply stated, the price for generality in highly parallel structures is decreased speed, decreased efficiency of hardware utilization, and increased software requirements.[12] During the last decade, there have been many highly parallel general-purpose architectures proposed or implemented. In general, they required many man-years of efforts to design and, because of their complexity, were very costly to build.

Tailored to meet specific application requirements or to off-load computations especially taxing to general-purpose computers,[1] special-purpose systems provide a very high degree of parallelism with minimum system overhead and complexity. They are generally the fastest and most efficient in hardware utilization.[12] However, because of their limited applicability, their cost must be low enough to justify their selection over a general-purpose approach.

THE SYSTOLIC ARCHITECTURE CONCEPT

Because special-purpose systems are seldom produced in large quantities, their design cost is a lot higher comparing to the parts cost.[1,6] This is particularly true when special-purpose designs are implemented with VLSI technology. Even though VLSI offers a number of major benefits—low cost per component, high density, reliability and ease of fabrication,[2,5,6] effective use of the technology to achieve massive parallelism requires careful consideration.

Briefly, a highly parallel VLSI structure should adhere to the following principles:[1,3,10,11,12,13,14]

(1) Simplicity and regularity: the design should consist of only a few simple types of modules which are replicated many times, thus reducing design complexity. A simple and regular structure is therefore highly cost-effective. In addition, such a structure can be easily expanded by increasing the number of basic modules. This, in turn, leads to linear speed improvements.

(2) Concurrency: The degree of concurrency in a system is largely determined by the underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to support a high degree of pipelining and multiprocessing.

(3) Communication: Control and communication become significant in a parallel computing structure, especially with VLSI where routing costs dominate the power, time and area required to implement a computation. The design's underlying algorithm should therefore employ only simple, regular control and communication. In a processor array, communications should occur only between neighboring processors.

(4) I/O consideration: Since a special-purpose device is typically attached to a host, its computational rate should not exceed the host's available I/O bandwidth. Therefore, if multiple computations are performed per I/O access, orders of magnitude improvements on system throughput are possible.

To meet these requirements, Kung and Leiserson in 1977 introduced the concept of *systolic architecture*. Originally proposed for VLSI implementation of some matrix operations,[15] a systolic system consists of an array of processing elements (PE's) called cells, each capable of performing some simple operations. These cells communicate only to their nearest neighbors, and communication with the outside world—i.e. the host—occurs only at the boundary cells.[1,5,6,15] Data flow from the host through the array in a rhythmic fashion, and computations are synchronized by a global clock signal. Each data item once brought out from memory is used effectively at each cell while being moved from cell to cell along the array.

Conceptually, computational tasks can be classified into two categories—compute-bound computations and I/O-bound computations. In a computation, if the total number of operations is larger then the total number of input and output elements, then the computation is compute-bound, otherwise it is I/O-bound. While speeding up an I/O-bound computation must rely on an increase in memory bandwidth, the systolic architecture allows a speed-up of a compute-bound computation without increasing the memory bandwidth requirement.[1]

Since cells in a systolic array are of only a few simple types, cost-effectiveness and ease of VLSI implementation are among the many advantages that systolic architecture offers. Others include simple and regular control and data flow, elimination of global broadcasting and modular expansibility.[1]

SYSTOLIC ARCHITECTURE DESIGN CRITERIA

Today, the systolic approach is increasingly being considered for computational intensive problems and there exist many systolic designs for a wide class of compute-bound applications. In several of his papers,[1,6] Kung suggested a number of systolic design criteria which are briefly outlined below.

(1) The design makes multiple use of each input data item. This property allows systolic systems to achieve high throughputs with modest I/O bandwidths for outside communication.

(2) The design uses extensive concurrency. The underlying algorithm should use as many of the available cells as possible at any given time during a computation. Even higher concurrency is possible if another level of pipelining is introduced to operations within the cells themselves.

(3) There are only a few types of simple cells. A large number of cells are required if a systolic design is to achieve any great performance gains. The cells must therefore be simple and of only a few types to curtail design and implementation costs. However, one should remember that there is always a trade-off between cell simplicity and flexibility. An

exact estimate can only be arrived at on a case by case basis.

(4) Data and control flows are simple and regular. Pure systolic systems totally avoid long-distance or irregular data communication wiring. This is the principal reason why a systolic array is adjustable to various performance goals. The only global communication (besides power and ground) is the system clock.

ORGANIZATION OF THIS THESIS

Even though the systolic architecture offers many advantages, it is not without some drawbacks. One possible problem is that if a systolic array is too large, its global clock signal could be skewed to the point where two cells at its opposite ends could not be synchronized properly.[1,11] Another issue is the degree of utility a systolic device can support. Proposed as a special-purpose architecture, one nonetheless wants a systolic array to be able to perform more than one type of computation. These are issues which cannot be resolved satisfactorily unless both architectural and algorithmic considerations are reviewed carefully.

The rest of this thesis is divided into four chapters. In Chapter II, a brief introduction to Faddeev's algorithm is presented; because the main focus of this thesis is in its architectural mapping, a more thorough treatment of the algorithm is referred to the original book listed in the REFERENCES section. Since matrix triangularization is an essential component of Faddeev's algorithm, descriptions of two systolic arrays for this matrix operation are also included.

Chapter III contains detailed examinations of two systolic implementations of Faddeev's algorithm. Analysis of the designs performance and correctness of operation is presented. Also, their advantages and weaknesses are discussed in this chapter.

In Chapter IV, a new systolic array implementation of Faddeev's algorithm is proposed. Again, a detailed description and a performance analysis of the design are offered. Necessary comparisons to the previous arrays concerning modularity, expansibility, versatility and ease of implementation will show it to be vastly superior.

In Chapter V, three different extensions to Faddeev's algorithm are developed. It will be shown that these techniques are ideally suited to the new systolic array. This leads to a four fold increase in the array throughput when matrix operations are to be solved continuously. Lastly, concluding remarks are offered at the end of this chapter.

Relevant materials that do not fall within the main focus of this thesis but are nonetheless important are included in the three appendices A, B and C. For the reader who wish to verify how Faddeev's algorithm solves various matrix computations, Appendix A contains examples which illustrate different variants of the algorithm. If he wishes to further investigate the operation of all architectures put forth in this thesis, Appendix B contains sequences of snapshots which show these arrays solving the examples of Appendix A. Finally, Appendix C contains the Pascal source listing of SAGS, a Systolic Arrays Graphical Simulator which produces those snapshots, and sample script files.

CHAPTER II

FADDEEV'S ALGORITHM AND MATRIX TRIANGULARIZATION SYSTOLIC ARRAYS

One aspect of systolic arrays that is the focus of several recent research efforts is their lack of generality, i.e. an array designed for one algorithm generally cannot run another. An approach aimed at removing this drawback taken by Kung is the use of a programmable systolic chip.[16,17] While this allows different sequences of operations to be performed within the cells of a systolic array, it is only a partial solution to the problem since the interconnections between neighboring cells are still unalterable. To remove this inflexibility, Snyder proposed a programmable switch lattice structure that gives an array processor re-configurable interconnections between its PEs;[13] however, the added complexity of such a network is beyond the current integration technology for large array sizes.

Another less drastic approach is to find algorithms and their array implementations which are general-purpose within a class of problems. This approach generally results in simpler processor and/or simpler interconnections, thus more array cells can be put into a single chip.

Consequently, the clock skew problem of large array sizes will be effectively reduced since the number of chips required would be smaller.

FADDEEV'S ALGORITHM

One general purpose algorithm, useful for a wide class of matrix operations and especially suited for systolic implementation, is the Faddeev's algorithm[18] illustrated by the simple case of computing the value of **CX** + **D**, given **AX** = **B**, where **A**, **B**, **C**, and **D** are known matrices of order n, and **X** is an unknown matrix.

The problem can be formulated as

$$\left.\begin{array}{cccc|cccc}
a_{11} & a_{12} & \cdots & a_{1n} & b_{11} & b_{12} & \cdots & b_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} & b_{21} & b_{22} & \cdots & b_{2n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{n1} & a_{n2} & \cdots & a_{nn} & b_{n1} & b_{n2} & \cdots & b_{nn} \\
\hline
-c_{11} & -c_{12} & \cdots & -c_{1n} & d_{11} & d_{12} & \cdots & d_{1n} \\
-c_{21} & -c_{22} & \cdots & -c_{2n} & d_{21} & d_{22} & \cdots & d_{2n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
-c_{n1} & -c_{n2} & \cdots & -c_{nn} & d_{n1} & d_{n2} & \cdots & d_{nn}
\end{array}\right. \tag{2.1}$$

or, in abbreviated form

$$\begin{array}{c|c}
\mathbf{A} & \mathbf{B} \\
\hline
-\mathbf{C} & \mathbf{D}
\end{array} \tag{2.2}$$

If by some means a suitable linear combination of the rows of **A** and **B** is found and added to the rows of **-C** and **D** as follow

$$\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{C}+\mathbf{WA} & \mathbf{D}+\mathbf{WB} \end{array}$$

where **W** specifies the appropriate linear combination such that only zeroes appear in the lower left hand quadrant, then the lower right hand quadrant will become matrix $\mathbf{E} = \mathbf{CX} + \mathbf{D}$. This is because annihilating **-C** requires $\mathbf{W} = \mathbf{CA}^{-1}$ so that $\mathbf{D} + \mathbf{WB} = \mathbf{D} + \mathbf{CA}^{-1}\mathbf{B}$, and since $\mathbf{AX} = \mathbf{B}$, $\mathbf{D} + \mathbf{WB} = \mathbf{D} + \mathbf{CX}$. The elegance and simplicity of the algorithm is apparent when one notes that to carry it out, it is only necessary to annul the lower left hand quadrant by applying a suitable *matrix triangularization* procedure to the left side of (2.2) while extending the operation to its right side. We will then have from (2.1)

$$\left[\begin{array}{ccccc|cccc}
a^{(k)}_{11} & a^{(k)}_{12} & \cdots & \cdots & a^{(k)}_{1n} & b^{(k)}_{11} & b^{(k)}_{12} & \cdots & b^{(k)}_{1n} \\
0 & a^{(k)}_{22} & \cdots & \cdots & a^{(k)}_{2n} & b^{(k)}_{21} & b^{(k)}_{22} & \cdots & b^{(k)}_{2n} \\
0 & 0 & a^{(k)}_{33} & \cdots & a^{(k)}_{3n} & b^{(k)}_{31} & b^{(k)}_{32} & \cdots & b^{(k)}_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & \cdots & 0 & a^{(k)}_{nn} & b^{(k)}_{n1} & b^{(k)}_{n2} & \cdots & b^{(k)}_{nn} \\
\hline
0 & 0 & \cdots & \cdots & 0 & e_{11} & e_{12} & \cdots & e_{1n} \\
0 & 0 & \cdots & \cdots & 0 & e_{21} & e_{22} & \cdots & e_{2n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & \cdots & \cdots & 0 & e_{n1} & e_{n2} & \cdots & e_{nn}
\end{array}\right]$$

or, in short

$$\begin{array}{c|c}
\mathbf{A}^{(k)} & \mathbf{B}^{(k)} \\
\hline
0 & \mathbf{E}
\end{array}$$

where $\mathbf{A}^{(k)}$ is an upper triangular matrix and $\mathbf{B}^{(k)}$ is $\mathbf{B}$ modified k times by the procedure. Often used in solving linear systems, *Gaussian elimination* is one of the better known triangularization methods available to perform the Faddeev's algorithm. Since the usual backsubstitution is not needed here, considerable savings in computation and storage are obtained.

With Faddeev's algorithm, a variety of matrix operations can be performed by selective entries in the four quadrants. For example, when $\mathbf{D} = 0$, $\mathbf{C} = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix, and **B** is a column vector, **E** becomes **X**, the solution to the linear system **AX** = **B**. Some other matrix operations possible with Faddeev's algorithm are shown in Figure 1 below. The reader is referred to Appendix A for a detailed treatment of Gaussian elimination and the solutions to a sample linear systems using Faddeev's algorithm.

$$\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{C} & \mathbf{D} \end{array} \Rightarrow \mathbf{CA^{-1}B+D} \qquad \begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{I} & 0 \end{array} \Rightarrow \mathbf{A^{-1}B}$$

$$\begin{array}{c|c} \mathbf{I} & \mathbf{B} \\ \hline -\mathbf{C} & 0 \end{array} \Rightarrow \mathbf{CB} \qquad \begin{array}{c|c} \mathbf{A} & \mathbf{I} \\ \hline -\mathbf{I} & 0 \end{array} \Rightarrow \mathbf{A^{-1}}$$

$$\begin{array}{c|c} \mathbf{I} & \mathbf{B} \\ \hline -\mathbf{C} & \mathbf{D} \end{array} \Rightarrow \mathbf{CB+D}$$

Figure 1. Some matrix operations possible with Faddeev's Algorithm.

SYSTOLIC ARRAYS FOR MATRIX TRIANGULARIZATION

Since the underlying procedure to carry out Faddeev's algorithm is matrix triangularization, any systolic implementation of the algorithm should be based on a structure which can perform triangularization efficiently. Developed by Gentleman and Kung as a common platform for two different triangularization methods, the triangular systolic array of Figure 2 can execute both *Gaussian elimination with neighbor pivoting* or *orthogonal triangularization*.[19,20] The array consists of two types of cells: the *boundary* cells (represented by circles) and the

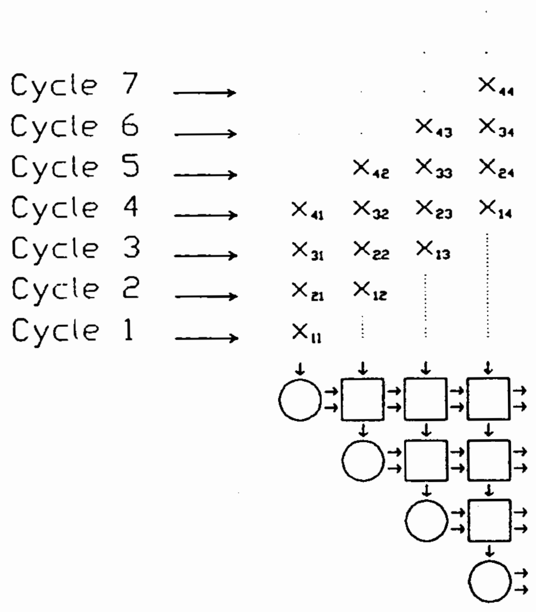


Figure 2. Triangular systolic array for matrix triangularization.

internal cells (represented by squares). These cells are locally interconnected into a triangular mesh. Each cell stores a microprogram, enabling it to interact with its neighbors in such a way that a triangularization procedure can be carried out. Changing the microprograms of the cells will allow the array to execute different procedures.

In the following discussion, the term *data row* refers to a row of entries of matrix **X**, whereas the term *array row* means a row of cells of the array. The triangularization of matrix **X** by the array is as follow. Initially, all cells contain only zeroes. As each data row *i* enters the array via the top boundary, its entries are stored in the cells on the *i*th array row. Before the data row *i* reaches its respective array row however, its entries are modified by cells of previous array rows such that the first $i - 1$ entries are discarded—i.e. became zeroes. The modification of an incoming data row is initiated by a boundary cell. This cell generates *modification factors*, values resulting from computations performed on an incoming entry and the cell's own stored value. The modification factors are then sent rightward to meet other entries of the same data row in the internal cells. There, they are used to modify the entries which are subsequently outputed to the next array row. While cells of any given array row are updating a data row, they may also update their own currently stored values.

Note that because of the critical timing required for the rightward data stream to reach internal cells at proper moments, the input data flow is fed into the array in a skewed order. After completion, modified x values left in cells constitute elements of a triangularized matrix and can then be readily read out, one from each cell.

Gaussian Elimination With Neighbor Pivoting

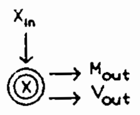
When Gaussian elimination procedure is performed using finite-precision arithmetic, as would be the case for electronic computing devices, a diagonal element that is small compared to the entries below it in the same column can lead to substantial roundoff error. Traditionally, *pivoting* strategies such as *partial* or *total pivoting* have been used to improve its numerical stability.[21] Because of the global communication that may result from pivot selection, they are not quite suitable for systolic implementation. Thus, to maintain the same degree of stability for the triangularization process described above, Gentleman and Kung suggest the use of another pivoting strategy, called *neighbor pivoting*. This technique introduces a zero to a row by subtracting a multiple of an *adjacent* row from it, interchanging the rows when necessary to prevent the multiple from exceeding unity.[19] In Appendix A, examples of Faddeev's algorithm using Gaussian elimination with neighbor pivoting is shown.

The triangular array of Figure 2 can perform Gaussian elimination with neighbor pivoting using the cells shown in Figure 3. As its microcode reveals, the boundary cell generates two modification factors: a multiplier M_{out}, as well as a Boolean variable V_{out}, which signals a row interchange when having value one. This occurs at every array *cycle*, the maximum length of time necessary for a cell to execute its microprogram once.

Orthogonal Triangularization

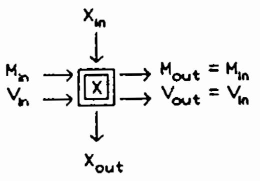
The orthogonal triangularization procedure involves the execution of a series of *plane rotations* (also known as *Givens rotations*) on the target matrix. They are applied initially to the first row and the second row, the first row and the third, the first row and the fourth, and so on to the last row. At this point, all rows except the first will

BOUNDARY CELL :



```
if $|X_{in}| \geq |X|$ then
   begin
     $V_{out} \leftarrow 1$
     $M_{out} \leftarrow$ if $X_{in} \neq 0$ then $-X/X_{in}$
                 else 0
     $X \leftarrow X_{in}$
   end
else
     $V_{out} \leftarrow 0$
     $M_{out} \leftarrow -X_{in}/X$
```

INTERNAL CELL :



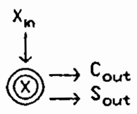
```
if $V_{in}$ then
   begin
     $X_{out} \leftarrow X + M_{in} * X_{in}$
     $X \leftarrow X_{in}$
   end
else
     $X_{out} \leftarrow X_{in} + M_{in} * X$
```

<u>Figure 3</u>. Microcode specifications of boundary cell and internal cell for Gaussian elimination with neighbor pivoting.

have zero entries on their first column. Next, the above process is repeated starting with the second row, then again with the third row, and so on until zeroes are introduced to all columns such that the resultant matrix becomes upper triangular, after which the triangularization procedure is completed. In Appendix A, the reader will find a more detailed description of Givens rotations along with examples of Faddeev's algorithm illustrating their uses.

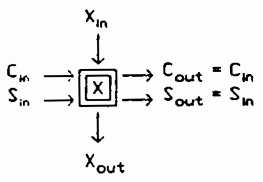
The systolic array of Figure 2 can perform orthogonal triangularization using the cells specified in Figure 4. While this method yields better numerical accuracy than that of the previous section,[22] notice the added complexity necessary for boundary cells because of the square roots. Since all cells in the systolic array must operate at the same throughput rate, the boundary cells could form a bottleneck on the overall system performance.[20]

BOUNDARY CELL :



```
if  $X_{in} = 0$  then
    begin
        $C_{out} \leftarrow 1$
        $S_{out} \leftarrow 0$
    end
else
    begin
        $C_{out} \leftarrow X / \sqrt{X^2 + X_{in}^2}$
        $S_{out} \leftarrow X_{in} / \sqrt{X^2 + X_{in}^2}$
        $X \leftarrow \sqrt{X^2 + X_{in}^2}$
    end
```

INTERNAL CELL :



```
$X_{out} \leftarrow -S_{in} X + C_{in} X_{in}$
$X \leftarrow C_{in} X + S_{in} X_{in}$
```

Figure 4. Microcode specifications of boundary cell and internal cell for orthogonal triangularization.

CHAPTER III

SYSTOLIC IMPLEMENTATIONS OF FADDEEV'S ALGORITHM

In this chapter, we will look at two systolic implementations of Faddeev's algorithm, originated from different authors. Their basic arrays are remarkably similar in most aspects such as interconnection topology, cells layout, I/O requirements and general algorithm mapping. This is not surprising since both are based on the same triangular array we've just examined in the previous chapter. However, they differ in the triangularization methods used to implement Faddeev's algorithm, which lead to dissimilar cells' control codes and numbers of pin-out. We can attribute this to the respective authors' design choices concerning the trade-offs between algorithm's stability and array's throughput rate.

NASH'S IMPLEMENTATION

To improve its numerical stability, Nash et. al.[23,24] suggested a modification to Faddeev's algorithm by replacing the Gaussian elimination procedure used to triangularize the coefficient matrix **A** of (2.2) with orthogonal triangularization.

For clarity, it is useful to divide their algorithm into a two-phase procedure. In the first phase, **A** is triangularized by a series of Givens rotations (simultaneously applied to **B**); in the second phase, the diagonal elements of the resulting triangular matrix are used as pivoting elements in the Gaussian elimination procedure on **C** and **D**, where columns of **C** will be zeroed out and **D** will become the result. Note that for the Gaussian elimination procedure to work properly, it is necessary that these pivoting elements be non-zero, hence the requirement that **A** be *full rank*, i.e. at least one of its square submatrices of order n has a non-zero determinant.

Nash's systolic implementation, shown in Figure 5, consists of a triangular array and its right extension, a square array. The triangular array, based on Kung's design in Figure 2 for orthogonal triangularization, performs Givens rotations on **A** (first phase) and ordinary Gaussian elimination on **C** (second phase). For higher efficiency in performing Givens rotations, cells' microcodes of Figure 4 are slightly modified into those of Figure 6. Furthermore, the added processing of ordinary Gaussian elimination requires the extra codes of Figure 7. The square array simply extends the corresponding processings to **B** and **D** and thus consists only of square cells.

The input data flow involves feeding **A** and **B** through the system from the top with cells executing the

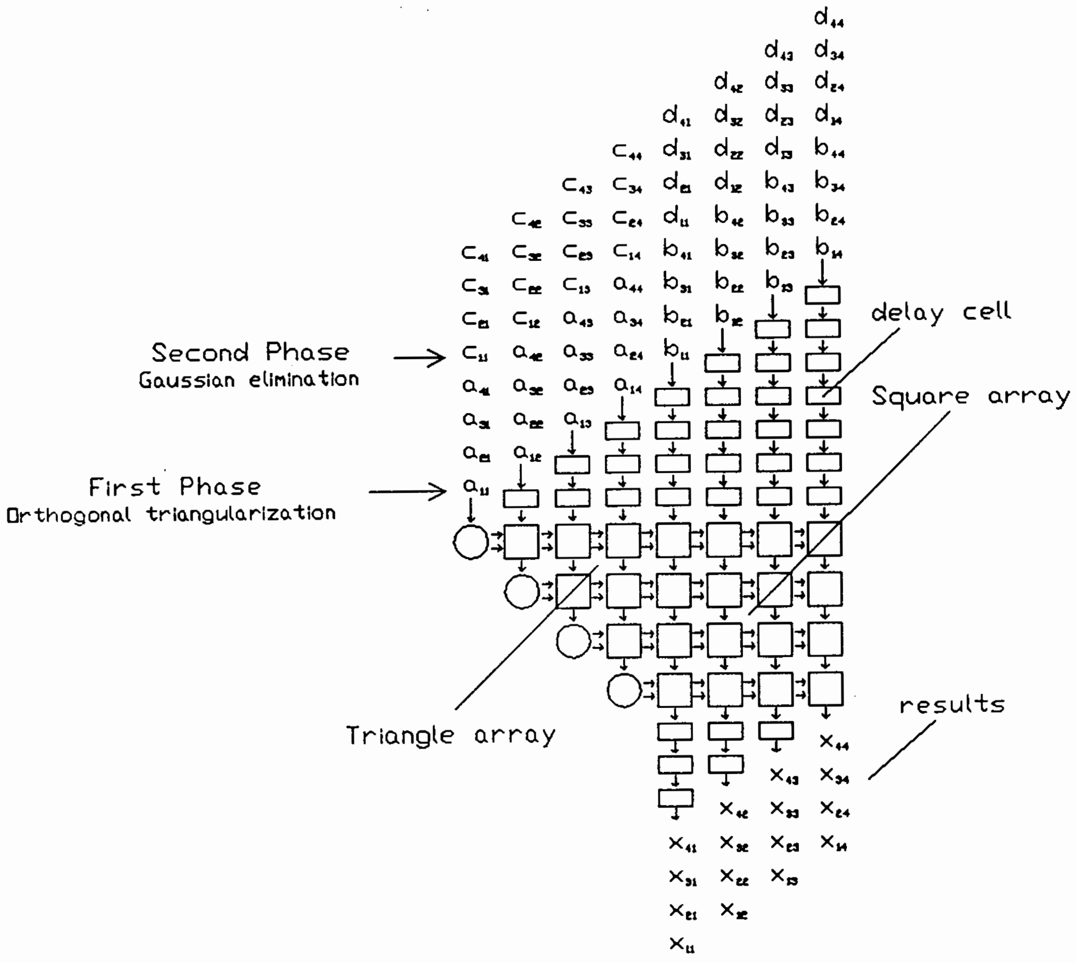
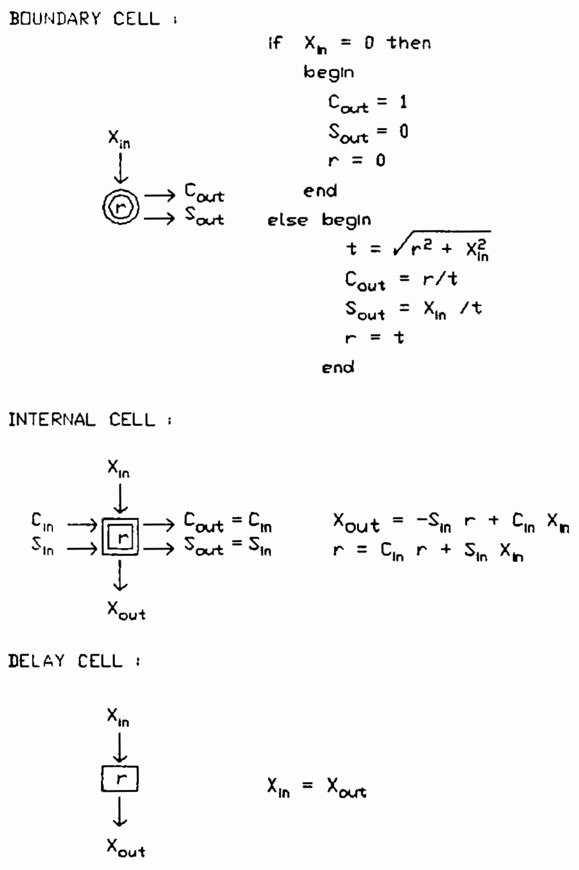


Figure 5. Nash's systolic implementation of modified Faddeev's algorithm. Note the use of delay cells to skew the data flows.

microprograms of Figure 6 on each incoming row. This corresponds to the first phase of the modified algorithm. Notice that the required skewing of the data flow is performed by a triangular array of delay cells (represented by rectangles) above the system. The second phase is accomplished by a similar flow of **C** and **D**, only this time the cells execute the microprograms of Figure 7 on the data elements and the resulting matrix will appear row by row coming out from the bottom of the square array. These output rows are straightened back to normal by another triangular array of delay cells below the square array. With a matching I/O bandwidth, the system will compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ in $5n - 1$ steps and solve a linear system of n equations in $4n$ steps.



```
if  $X_{in} = 0$ then
    begin
      $C_{out} = 1$
      $S_{out} = 0$
      $r = 0$
    end
else begin
        $t = \sqrt{r^2 + X_{in}^2}$
        $C_{out} = r/t$
        $S_{out} = X_{in}/t$
        $r = t$
      end
```

$X_{out} = -S_{in}\ r + C_{in}\ X_{in}$

$r = C_{in}\ r + S_{in}\ X_{in}$

$X_{in} = X_{out}$

Figure 6. Microcode specifications of boundary cell and internal cell used in Nash's array during the first phase, i.e. Givens rotations.

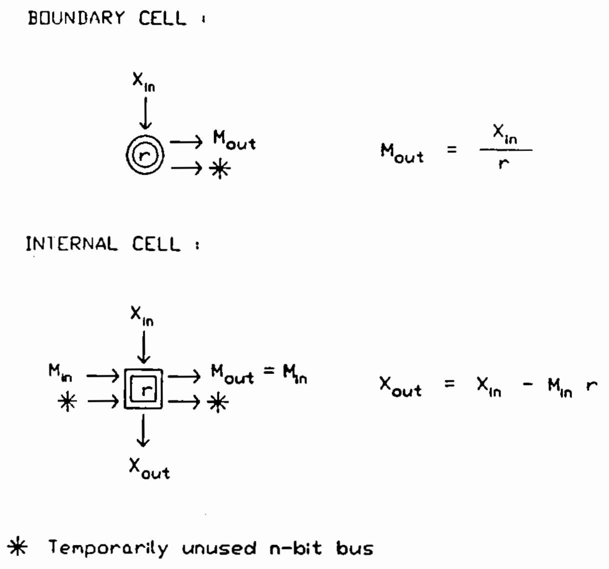


Figure 7. Microcode for boundary cell and internal cell used in Nash's array during the second phase, i.e. Gaussian elimination.

The input data flow can be contiguous, i.e. matrices **A** and **B** and then **C** and **D** can enter the array without any interruption in between. Data flows of separate problems to be solved by the array can also be fed continuously into the array. For this to be possible, additional control capabilities are necessary to switch the cells from one set of codes to another at the proper time. Slight modification of the microprograms will also be required.

Although Nash's modified Faddeev's algorithm is mathematically sound, its systolic implementation, unfortunately, contains some serious deficiencies. For instance, it is possible for the array to produce erroneous results, as illustrated by the following example. Suppose we have a linear system **AX** = **B** of order $n = 3$ where **X** is an unknown matrix, and one or more entries in column 1 of matrix **A** are zeroes, in this case, a_{21}:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 7 \\ 2 & 1 & 3 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 5 \\ 9 \\ 7 \end{bmatrix} \tag{3.1}$$

Since the determinant of **A**, $\Delta(\mathbf{A}) = 9$ is non-zero, **A** is therefore full rank, thus guaranteeing that a solution to the system exists and that it is unique with $x_1 = 1.33$, $x_2 = -0.67$ and $x_3 = 1.67$. However, when **A** is fed into the array of Figure 5, because $a_{21} = 0$, during the second step the boundary cell of row 1 column 1 will clear its r register, previously storing $a_{11} = 1$. This effectively transforms **A** into another matrix, say **E**, whose entries are identical to **A**'s except for e_{11}, which is zero, and all further processings will be on the resulting linear system

$$\mathbf{E} = \begin{bmatrix} 0 & 2 & 3 \\ 0 & 4 & 7 \\ 2 & 1 & 3 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 5 \\ 9 \\ 7 \end{bmatrix} \tag{3.2}$$

In this case, since $\Delta(\mathbf{E}) = 4$ is non-zero, **E** is also full rank and therefore the procedure is completed successfully, but with $x_1 = 3$, $x_2 = 4$ and $x_3 = -1$ which is the solution to (3.2) instead of (3.1).

The cause of the above error can be traced to a bug in the microprogram of the boundary cell. As Figure 6 reveals, this microprogram has the line of code

$$r = 0$$

which always clears the content of register r whenever $x_{in} = 0$. In fact, if at any time during processing the boundary cell of a row i receives a zero-valued x_{in} from an internal cell of row $i - 1$, erroneous result will appear at the end of processing. Thus, to correct the problem, this line should be removed.

For the purpose of verification, the reader is referred to Appendix A where correct solutions to examples (3.1) and (3.2) are arrived at manually using Faddeev's algorithm with Givens rotations. Furthermore, he is encouraged to examine the series of snapshots included in Appendix B which shows the graphics simulation of Nash's array computing (3.1). These pictures illustrate clearly the sequence of events leading up to the erroneous results.

Implementation errors aside, a drawback of Givens transform is the square root needed to compute the values of *sine* and *cosine* for each rotation. Execution time of this operation can easily be ten times that of a multiplication or division. Since timing is critical for proper synchronization of data flow in a systolic array, it is necessary to slow down the entire array correspondingly. Thus the circular cells represent a bottleneck in the system. Of course a hardware implementation of the square root is possible, however, we have to bear in mind the cost of added cell's complexity.

Another drawback of this implementation is the large pin counts for individual cells because of the need to transmit simultaneously the *sine* and *cosine* values to neighboring PEs. Not counting clock and control signals, the boundary cell will require one input and two output data buses and the internal cell will require three input and three output data buses. For n-bit operands, $3n$ and $6n$ I/O pins are needed for the boundary cell and the internal cell, respectively. This translates to a large chip area for each cell. Bus sharing or multiplexing schemes to reduce I/O lines are possible, but they would increase the processing time and consequently, reduce the throughput rate.

CHUANG AND HE'S IMPLEMENTATION

Another systolic implementation of Faddeev's algorithm, proposed by Chuang and He,[25] significantly improves upon the previous array. As shown in Figure 8, many similarities exist between the two arrays' design. To compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ from (2.2), both systems use a triangular array for the triangularization of $\mathbf{A}$ and the annulment of $\mathbf{C}$, and a square array for extending the corresponding processing to $\mathbf{B}$ and $\mathbf{D}$. The input data flow to both systems are similarly organized and skewed, and pipelined through each system in a similar fashion. For the processing of the lower half of the input data flow (i.e. matrices **C** and **D**), both employ ordinary Gaussian elimination.

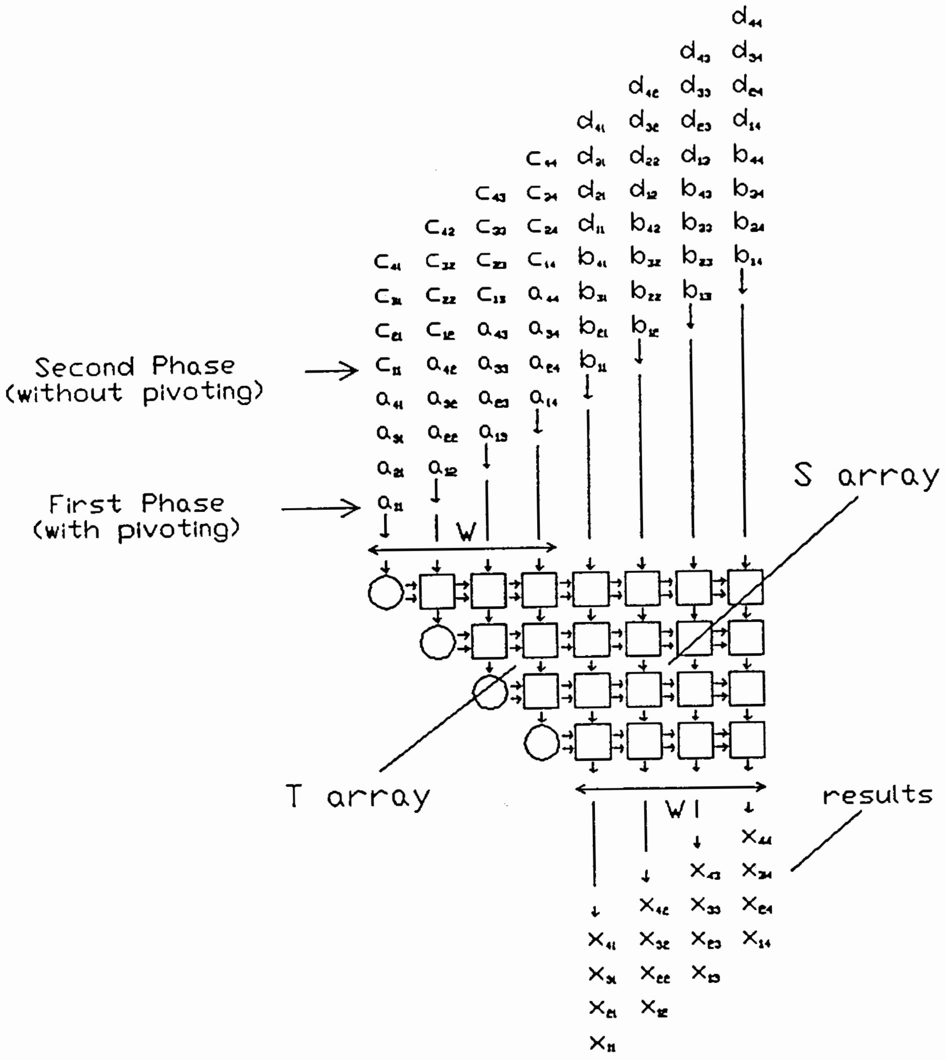


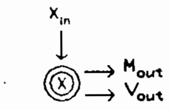
Figure 8. Chuang and He's systolic implementation of Faddeev's algorithm. The triangularization method used here is Gaussian elimination with neighbor pivoting.

However, Chuang and He's system processes the upper half of the input data flow (i.e. matrices **A** and **B**) using Gaussian elimination with neighbor pivoting instead of the Givens transform.[19] Hence, while numerical accuracy is somewhat inferior, this implementation is less expensive in terms of processing time and hardware complexities. Because the square root operation is not used, the array avoids the bottleneck problem created by the boundary cells of the Nash's array. And since the rightward data flow essentially consists of only one operand, M_{out}, the pin counts of boundary cell and internal cell are correspondingly reduced to $3n$ and $4n$, respectively.

Since it is obvious that different phases of processing are required for the upper half and the lower half of the data flow, two separate sets of microprograms for boundary cells and internal cells are needed, as shown in Figure 9 and 10. The first set, the pivoting functions, performs Gaussian elimination with neighbor pivoting on **A** and **B**, while the second set, the non-pivoting functions, performs regular Gaussian elimination on **C** and **D** and is essentially the same as the functions of Nash cells in Figure 7.

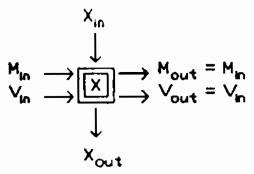
As the data flow is pipelined through the array, each boundary cell stores an input data element and sends a multiplier M_{out} rightwards to modify the input data that enter the internal cells of the same row. Along with each

BOUNDARY CELL :



```
if $|X_{in}| \geq |X|$ then
   begin
     $V_{out} \leftarrow 1$
     if $X_{in} \neq 0$ then
        $M_{out} \leftarrow -X/X_{in}$
     else $M_{out} \leftarrow 0$
     $X \leftarrow X_{in}$
   end
else
   $V_{out} \leftarrow 0$
   $M_{out} \leftarrow -X_{in}/X$
```

INTERNAL CELL :



```
if $V_{in}$ then
   begin
     $X_{out} \leftarrow X + M_{in} * X_{in}$
     $X \leftarrow X_{in}$
   end
else
   $X_{out} \leftarrow X_{in} + M_{in} * X$
```

<u>Figure 9</u>. Microcode specifications of cells used by Chuang and He's array for Gaussian elimination with neighbor pivoting.

M_{out} it generates a one-bit boolean value V_{out} to signal whether pivoting is needed. Each internal cell stores a data value arriving from the top and passes downward all the following data after modification. M_{out} and V_{out} remain unchanged as they travel rightwards through the array. For an input column of length and width $2n$ data elements, the output will be a matrix of order n emerging from the bottom of square array. It can be seen that when the system matches the I/O bandwidth, $5n - 1$ steps are required to obtain $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ and $4n$ steps are needed to solve a linear system of n equations.

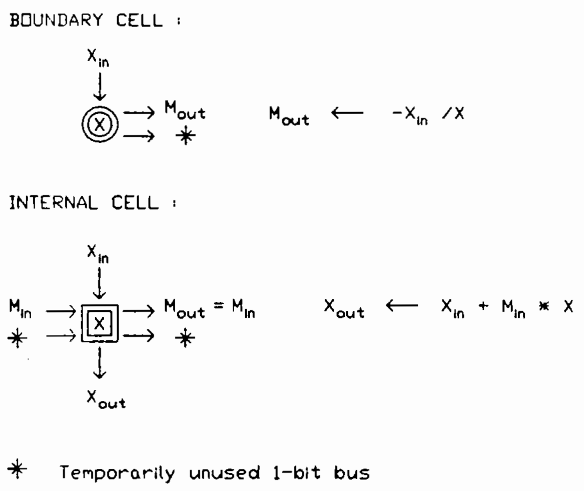

BOUNDARY CELL :
X_{in}
M_{out}
$M_{out} \leftarrow -X_{in} / X$
INTERNAL CELL :
X_{in}
M_{in}
$M_{out} = M_{in}$
$X_{out} \leftarrow X_{in} + M_{in} * X$
X_{out}
✳ Temporarily unused 1-bit bus

Figure 10. Microcode specifications of cells used by the array for ordinary Gaussian elimination.

Like in the Nash's implementation, the input data flow of this array can be continuous if additional control capabilities are used to individually switch each cell from pivoting to non-pivoting mode as required. As published, no technique was mentioned by the authors of both implementations to perform this switching; however, we can think of at least two different techniques to do this. One is to have the host or a dedicated controller generate the controls necessary for each individual cell, thus requiring a complex cell addressing scheme. Another is to tag control bits to input data elements which will then carry the control information with them throughout the array. This method assumes that the host, while generating the input data, will add the necessary control information to it. Its down side is that it will force an enlargement of the I/O bandwidth between the host and the array. In the next chapter, it will be shown that a combination of the above mentioned techniques will be used in our design. Thus, while having the advantages of both, it will avoid some of their inefficiencies.

Input Decomposition

Often, problems in real-world applications are larger in size than the available I/O bandwidth between the host and the array. When this is the case, increasing the array's size or speed does not bring about an increase in throughput since the limiting factor is the I/O bandwidth itself. One solution is to decompose the problems into smaller sub-problems, which can then be stored in the host and later processed in the array one at a time. In general, the tasks of decomposition and post-processing are complex and time consuming: passing intermediate results back and forth between the host and the array reduces the throughput that the I/O bandwidth can support. Furthermore, the array throughput also suffers because of the pipeline flush brought about by the interrupted data flow.

To avoid these problems, Chuang and He propose structuring the array as a feedback array system. The idea is that the system simulates the operation of an arbitrarily large array by using the small arrays over and over, with the output of the small arrays fed back to be processed with other input data at the proper times. To match the input data flow with the I/O bandwidth, it is necessary that the data flow be decomposed. For an I/O width of *w*, it is suggested that the data flow be cut into *strips* of width *w* parallel to the direction of the data flow, or *bands* of width *w* vertical to the data flow. These strips or bands are further cut into blocks of length *w*. A problem of size $2n \times 2n$ where *n* is *m* times *w* will yield $2m \times 2m$ blocks. Depending on the order in which these blocks are fed into the array, we have *parallel*, *vertical* or *hybrid decomposition* as shown in Figure 11.

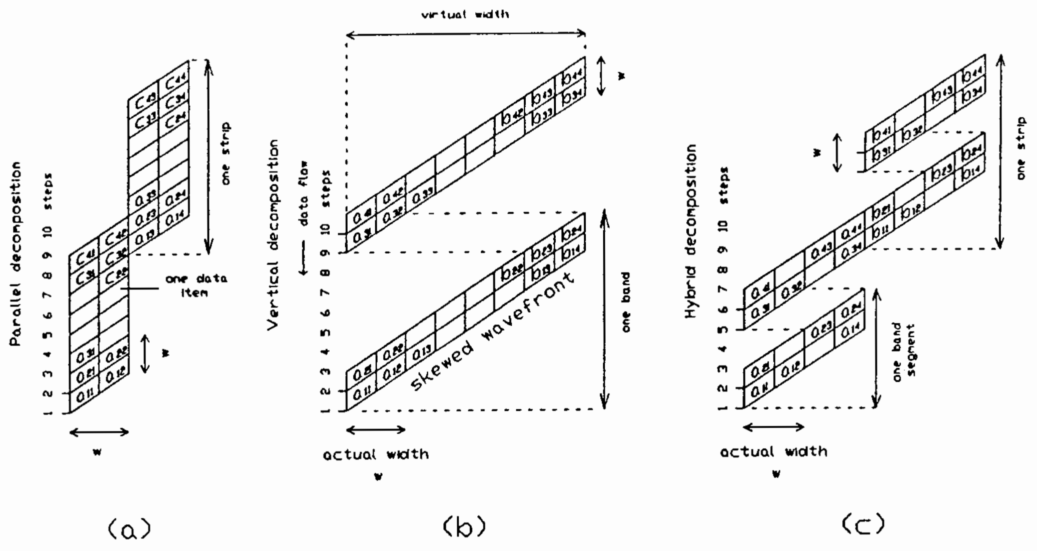


Figure 11. Three ways to decompose the input data flow. (a) Parallel decomposition. (b) Vertical decomposition. (c) Hybrid decomposition.

In this figure, the series of vertical numbers represent the order of the steps in which the strips or bands are fed into an array. Note that in the parallel decomposition (Figure 11a), the end of the first strip overlaps with the beginning of the second strip, i.e. the last data item of the first strip enters the array at the same time (step number 9) as the first data item of the second strip. The bands of the vertical decomposition (Figure 11b) are similarly overlapped, as with the *band segments* and the strips of the hybrid decomposition (Figure 11c). All this overlapping ensures that the input data flow to the array is continuous.

<u>Feedback Systems for Parallel Decomposition</u>

Suppose we want to compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ for matrices of size n using the full size array of Figure 12. Again the available I/O bandwidth is w wide. We can decompose the $2n \times 2n$ input data flow into $2m$ strips, each w wide as in Figure 11a, numbered from V_1 to V_{2m}. For $m = 4$, the full size array of Figure 12 can be thought as consisting of 26 subarrays, with each subarray of type T or S and of size w. Under the given I/O constraint, feeding the strips one after another continuously into this array will not work since the rightward data stream generated by a T subarray from one strip will not meet the following strips at a proper time.

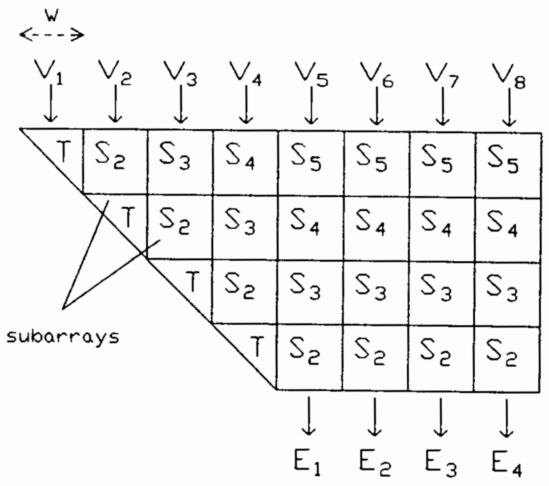


Figure 12. Systolic system with 26 subarrays of types T and S, each of width *w*. The available I/O bandwidth is also *w*.

On the other hand, the feedback array system of Figure 13 will process the same data flow correctly under the same I/O constraint. This feedback array system simulates the large array of Figure 12 by using its component arrays over and over again as follows. Initially, as V_1 is fed into the T array, it generates a horizontal data stream which is then stored into the memory buffer B1. The content of this buffer is recycled into arrays S_2, S_3, S_4 and S_5 for the processing of strips V_2, V_3, V_4 and V_5 respectively as they arrive. When the intermediate result from strip V_2 comes out of S_2, it too goes into the T array to produce another stream of horizontal data which is then stored into buffer B2. Again, the content of B2 is fed back into arrays S_2, S_3 and S_4 to process the intermediate results of V_3, V_4, and V_5 coming out of S_3, S_4 and S_5

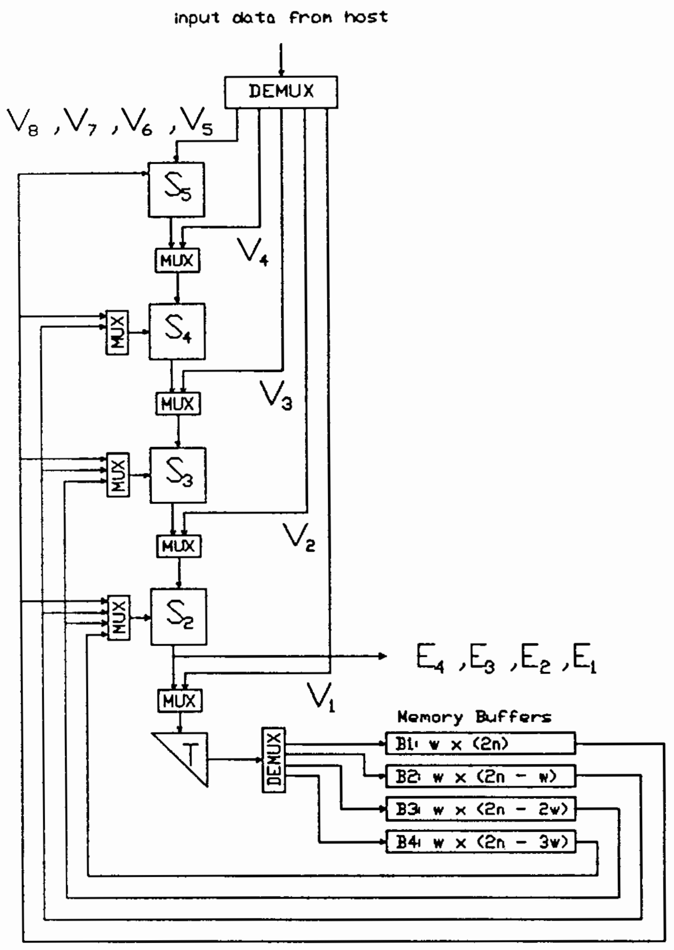


Figure 13. Feedback systolic system with a smaller number of subarrays for parallel decomposition. This system cannot solve problems with $m > 4$.

respectively, and so on. To properly synchronize the horizontal data streams, the buffers B1, B2, B3 and B4 must be of length $2n$, $2n - w$, $2n - 2w$ and $2n - 3w$ respectively. Note that each successive buffer is shorter by w. This is because as a data strip V_i goes through a square array S, it is shortened by a $w \times w$ block of data, which remains inside S. Hence, the T array processing this shortened data strip will generate a correspondingly shortened horizontal stream of modification factors.

This feedback array system achieves maximum throughput using much less component arrays than the larger array in Figure 12. The number of steps for it to compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ is

$$\Big((2m)(2mw) + w - 1\Big) + mw = (4m + 1)n + w - 1 = O(mn) \tag{3.3}$$

where $O(k)$ denotes order of k.

Since this system requires m S arrays and m buffers, it is not quite independent of problem size. Because the S arrays are identical, eliminating all but one reduces the number of component arrays needed and, at the same time, yields a design that is problem size independent. Figure 14 illustrates a one-T one-S feedback array system. The feedback scheme is now two-dimensional, with horizontal and vertical data streams. The input data flow is similarly fed into the system as in the previous system. However, because only one S array is available, each data strip V_r where $r = 2, 3, \ldots, 2m$ will be processed by the same S array $r - 1$ times. While intermediate results of strip V_2 will go directly into the T array, an additional buffer B_s is needed to store the intermediate results generated from strips V_3, $V_4, \ldots,$ and V_{2m}. The feedback of these intermediate results to the S array is inserted in between adjacent strips thus preventing data strips from V_3 onward to be fed continuously into the system.

The throughput of this system is of course lower. The number of steps necessary to complete $\mathbf{CA^{-1}B + D}$ is now

$$2mw + \sum_{k=1}^{m} (2m - k)(2m - k + 1)w + 2w - 1 = \qquad (3.4)$$

$$\frac{7}{3}(m^2n) + \frac{5}{3}(n) + 2w - 1 = O(m^2n)$$

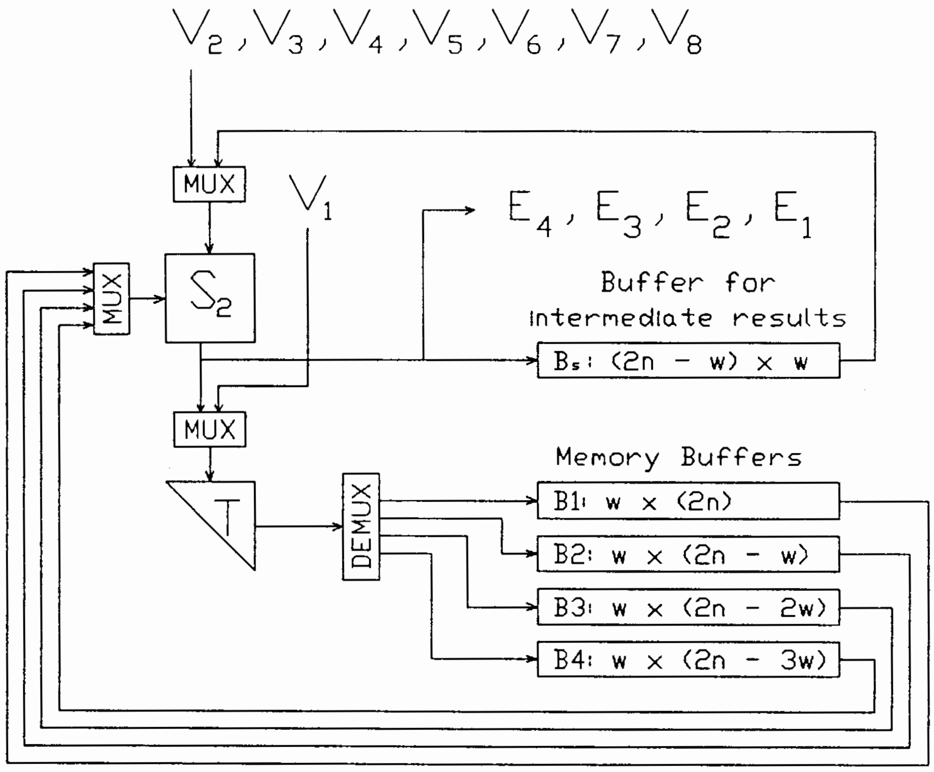


Figure 14. Two-dimensional feedback system with one S and one T subarrays. This system is problem size independent.

Feedback Systems for Vertical Decomposition

In Figure 15 below, Chuang and He illustrated how an array wider than available I/O bandwidth can solve a matching large problem when the input data flow is decomposed vertically like in Figure 11b. Again suppose the I/O bus is w wide and the array is $2n = 2mw$ wide. Essentially the same system as that of Figure 12, this array system has in addition a $2m$-way demultiplexer on the input side and an m-way multiplexer on the output side. The input data flow, consisting of $2m$ bands of $2m$ blocks each, is fed

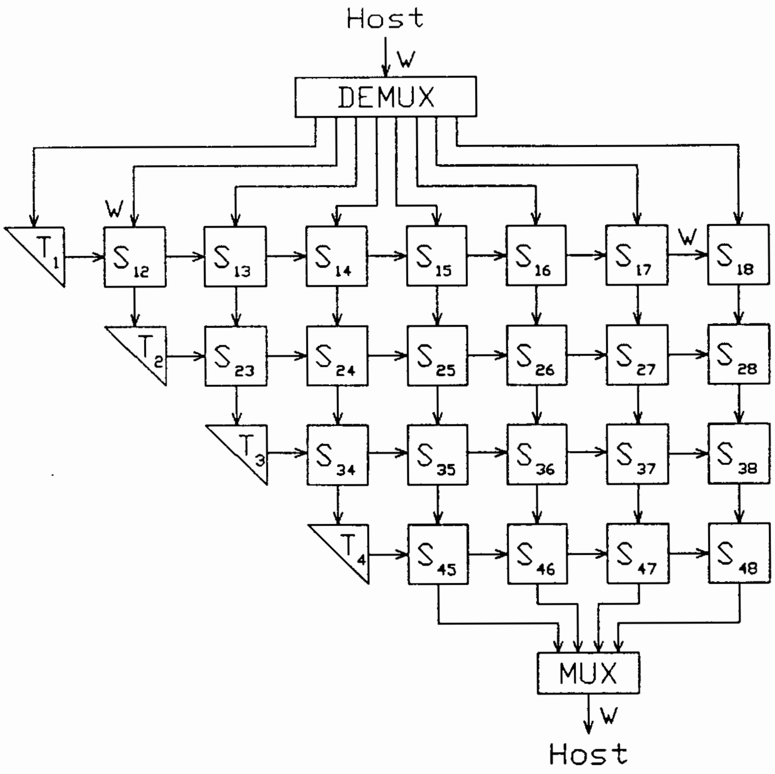


Figure 15. Array system for vertical decomposition of input data flow. With I/O bandwidth w, full utilization of available cells is not possible.

into the array one band after another continuously. The demultiplexer feeds the blocks of each band to the subarrays on the first row of the system one at a time from left to right. Since all the blocks are skewed, each overlapped with its left and right neighbors and the whole band is contiguous as it enters the system.

For this array, the total number of steps to complete the process is

$$\big((2m)(2mw) + w - 1\big) + mw =$$
$$(4m + 1)n + w - 1 = O(mn)$$

which is identical to equation (3.3) of Figure 13. While the array of Figure 15 has many more subarrays, its processing speed is not higher because maximum usage of all cells is not realized due to the I/O bottleneck. Furthermore, this array is not problem size independent.

Although inefficient in terms of usage of available hardware, the array of Figure 15 serves as an example of how a vertically decomposed data flow should be processed. A more flexible system, shown in Figure 16, is problem size independent and delivers the same throughput using a smaller number of subarrays. In this system, the $2m$-way demultiplexer of Figure 15 is reduced into a 2-way demultiplexer which is repeated at the input side of every row of subarrays. As the bands of the input data flow enter the first row of the system continuously, the first block of each band is routed into the T array while the rest are fed

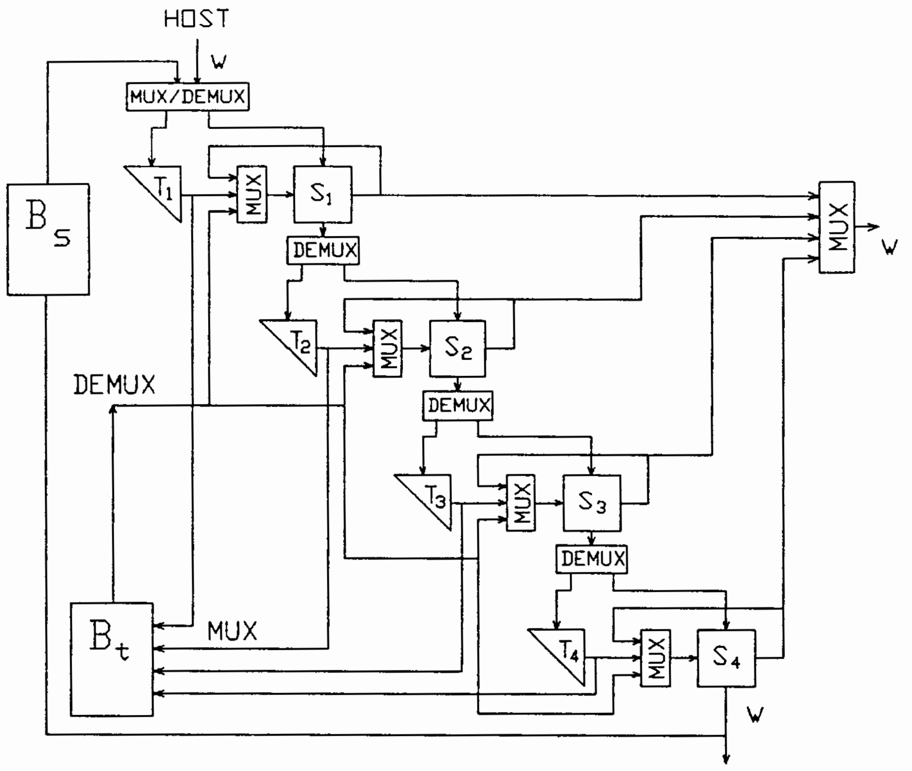


Figure 16. Problem size independent array system for vertical and hybrid decomposition of input data flow. Available I/O bandwidth is *w*.

into the S array on the same row. The rightward data stream generated by the T array is fed into the S array and recycled until all blocks of the same band are processed. Because these blocks form a contiguous data stream, no buffer is needed to store the M_{out} and V_{out} values for recycling. On the other hand, X values stored in the S array cells need to be saved as shifting into the neighboring block begins since they will be used later in the processing of the next band of data. To simplify control and reduce memory access, they will be stored into the recycling shift registers implemented next to each cell as illustrated in Figure 17.

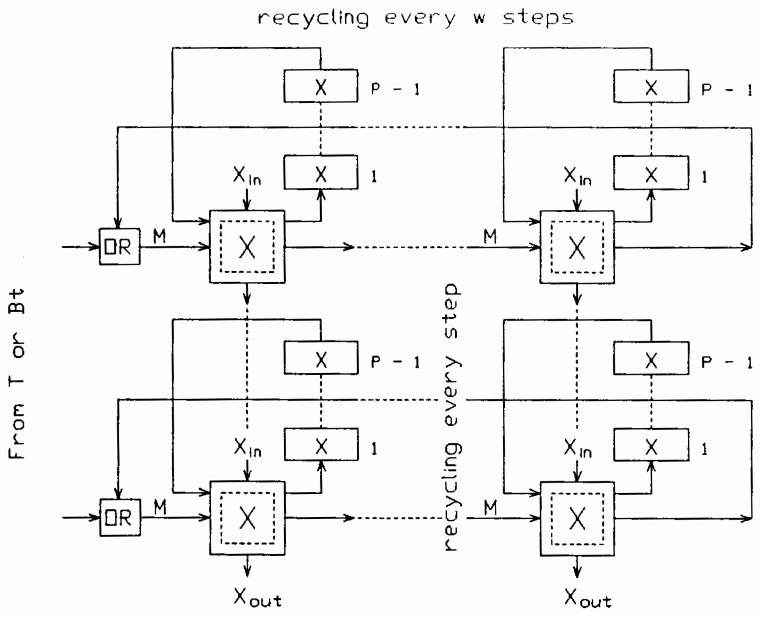


Figure 17. Recycling shift registers for the temporary storage of the X values. Implemented next to each cell, each buffer is *p* in length.

Outputs from the bottom of an S array, the X_{out} values, will be processed in the same way by the T and S arrays on the next row. When the problem is larger than the system, i.e. $2n > 4w$ of Figure 16, the outputs of the last row's S array will be stored in buffer B_s to be recycled back into the system for further processing.

Feedback System for Hybrid Decomposition

Due to the finite capacity *p* of the recycling shift registers of Figure 17, the size of problems that can be solved by the feedback system of Figure 16 is limited. A way to circumvent this limitation is to use the hybrid decomposition of Figure 11c. The input data flow in this case is divided into parallel strips of width pw. These strips are in turn divided into band segments of width w and length pw vertical to the direction of the data flow. Segment by segment, the strips enter the system of Figure 16 one after another continuously as in parallel decomposition. Blocks of each segment are processed as in vertical decomposition and fill the recycling shift registers of the cells with new X values, to be used later with the next segment. The rightward stream of modification factors, generated by segments of the first strip, is saved to be re-used on corresponding segments of the following strips, hence the need for the memory buffer B_t.

Sparsity in Matrices

Another important merit of Chuang and He's feedback array system is that, as they pointed out, it can skip blocks of zeroes in the input data flow, and thus greatly reduce the processing time. As an example, consider the linear system

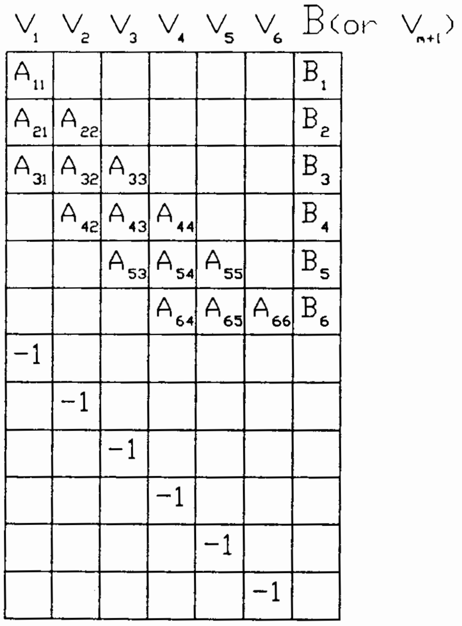
$$\mathbf{AX} = \mathbf{B} \tag{3.5}$$

where **A** is a lower blocked band matrix of order n, i.e.,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & & & & & & \\ \vdots & \mathbf{A}_{22} & & & & & \\ \mathbf{A}_{p1} & \mathbf{A}_{p2} & \vdots & \mathbf{A}_{m-p+1,m-p+1} & & & \\ & \mathbf{A}_{p+1,2} & \vdots & \vdots & \mathbf{A}_{m-p+2,m-p+2} & & \\ & & \vdots & \mathbf{A}_{m,m-p+1} & \vdots & \vdots & \mathbf{A}_{mm} \end{bmatrix}$$

and $\mathbf{B} = \left[\, \mathbf{B}_1, \mathbf{B}_2, \ldots, \mathbf{B}_m \,\right]^T$, $n = mw$, and each $\mathbf{A}_{ij}$ or $\mathbf{B}_i$ with $i = 1, 2, \ldots, m$, $1 \le j \le i$, is a $w \times w$ submatrix, or block.

The data flow is decomposed parallely into w wide strips of $w \times w$ blocks as shown in Figure 18. The blank blocks are the zero submatrices and the $-1_{i,j}$ blocks are the



V_1	V_2	V_3	V_4	V_5	V_6	B(or V_{n+1})
A_{11}						B_1
A_{21}	A_{22}					B_2
A_{31}	A_{32}	A_{33}				B_3
	A_{42}	A_{43}	A_{44}			B_4
		A_{53}	A_{54}	A_{55}		B_5
			A_{64}	A_{65}	A_{66}	B_6
-1						
	-1					
		-1				
			-1			
				-1		
					-1	

Figure 18. Parallel decomposition of a sparse matrix problem with $m = 6$. Note that matrix **B** in this case is the strip V_{m+1}.

diagonal submatrices of the -I matrix. Without loss of generality, **B** is assumed to be an $n \times w$ matrix. In this example, the bandwidth p of **A** is three blocks wide.

To understand how sparse matrices can be exploited to yield better throughput, let us analyze what happens when the system from Figure 12 process the data flow of Figure 18. On its first row, as the $2m$ blocks of V_1 are processed by the T array, $2m$ blocks of M_{out} values are generated horizontally to modify data strips on the right. Since only $p + 1$ blocks of V_1 are non-zero, only $p + 1$ blocks of M_{out} values are non-zero. This is because when incoming $X_{in} = 0$, the boundary cells invariably generate $M_{out} = 0$. Furthermore, because the internal cells always generate $X_{out} = X_{in}$ when $M_{in} = 0$, as the data strip V_{m+1} (containing **B** matrix) goes through array S_{m+1} on the first row, only its corresponding $p + 1$ blocks are modified, with the first zero block below $\mathbf{B}_m$ becoming the result $\mathbf{X}_1$. On the other hand, strips V_2 to V_m emerge from the S arrays of that row unmodified but minus their first blocks. This is because as they pass through these arrays, all zero entries of their leading blocks are retained in the cells' X registers, and thus $X_{out} = X_{in}$.

The above process is repeated on succeeding rows of arrays until all results are computed. Since the S arrays of column i ($i = 2,..., m$) are not needed to process strip V_i, they can be removed from the system and the strip's leading blocks of zeroes can be skipped. Because they do not contribute to the modification of data strips on the right, the zero blocks above and below the diagonal band of **-I** can also be skipped.

The architecture that most efficiently process sparse matrix problems is shown in Figure 19. This system receives the data flow of Figure 18 from the host, where all the zero blocks are eliminated except those of the strip V_{m+1}. As seen from Figure 19, the system uses only one S and one T arrays. The single T array is fed with **A**'s non-zero blocks, one strip after another continuously. Its horizontal data flow, consisting of modification factors M_{out} and V_{out}, is

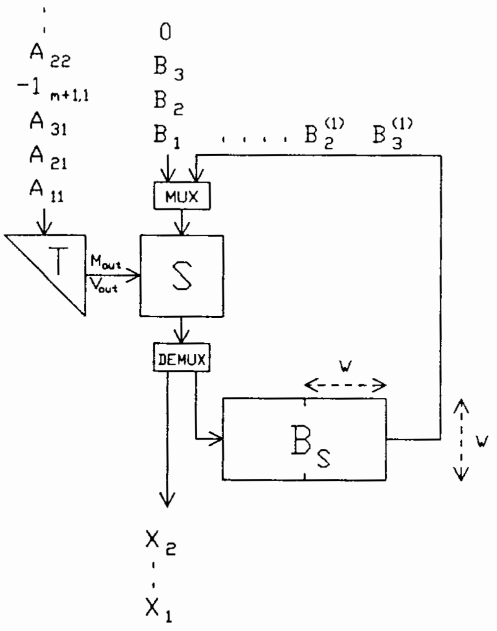


Figure 19. Systolic system for the processing of sparse matrix problems. Note that this design requires an I/O bandwidth of $2w$.

fed directly into the S array to modify V_{m+1}. V_{m+1} iterates through the S array p + 1 blocks at a time, each iteration is concurrent with a strip of **A**. During each iteration, the leading non-zero block remains in the S array where it is used to modify the next p - 1 non-zero blocks, and transform the last block (originally a zero block) into a block of results. The demultiplexer below the array S routes the modified p - 1 non-zero blocks to buffer B_s and outputs the block of results to the host. As they emerge from B_s, the modified p - 1 non-zero blocks are then combined with a new non-zero block and another zero block from V_{m+1} to form input data for the next iteration.

For instance, the first iteration sees the T array process blocks $\mathbf{A}_{11}$, $\mathbf{A}_{21}$, $\mathbf{A}_{31}$ and $-1_{m+1,1}$ of strip V_1 at the same time the S array process blocks $\mathbf{B}_1$, $\mathbf{B}_2$, $\mathbf{B}_3$ and the first zero block of strip V_{m+1}. This produces:

- block $\mathbf{B}_1$ which remains in the S array,
- blocks $\mathbf{B}_2^{(1)}$ and $\mathbf{B}_3^{(1)}$ which are temporarily stored in buffer B_s,
- and the block of results $\mathbf{X}_1$ which is outputed.

During the second iteration, the T array will process blocks $\mathbf{A}_{22}$, $\mathbf{A}_{32}$, $\mathbf{A}_{42}$ and $-1_{m+2,2}$ of strip V_2, while the S array process blocks $\mathbf{B}_2^{(1)}$, $\mathbf{B}_3^{(1)}$, $\mathbf{B}_4$ and the second zero block of V_{m+1}. The entire sequence of processing is illustrated by Figure 20.

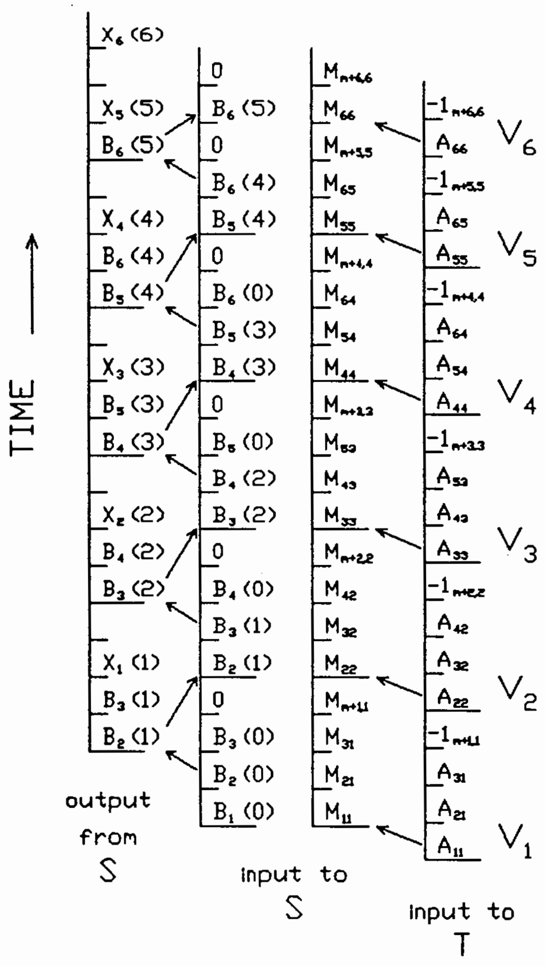


Figure 20. Processing sequence showing the order in which the non-zero blocks of Figure 18 are fed into the system of Figure 19.

Thus, the total number of steps it would take the system to compute **AX** = **B** is

$$\left(m(p+1) - \sum_{k=1}^{p-1} k + 3\right)w - 1 = \qquad (3.7)$$

$$n(p+1) - \frac{1}{2}(p-1)pw + 3w - 1$$

Note that this throughput rate requires that the system process the data strips of **A** concurrently with the data strip of **B**. Consequently, the total I/O bandwidth needed must be $2w$ wide instead of w. Furthermore, if **B** has more than one strip, the system of Figure 19 must be modified. The reader should be aware that the formula (3.7) was derived by the author of this thesis after it was found that the one given in the original paper was erroneous.

ASSESSMENT SUMMARY

As we have examined both systolic implementations of Faddeev's algorithm, several points should be noted. First, the feedback system of Figure 13 as shown can not process problems in which (2.1) is larger than $2n \times 2n$, where $n = mw$; however, by adding another feedback path from the output of its component array S_2 to the input of the top demultiplexer and using external memory for all B_i buffers, the system can be made independent of problem size.

With cells specification of Figures 6 and 7, system configurations of Figures 13, 14 and 16 can perform Faddeev's algorithm using orthogonal triangularization. This means that Nash's implementation of Faddeev's algorithm can be configured to have feedback paths which will allow it to solve problems larger than the available bandwidth.

Since the configurations of Figures 13, 14 and 16 extensively multiplex data flows to and from their component arrays, added control and hardware complexities are unavoidable. Furthermore, because the data flows must be skewed and overlapped, all multiplexers (and demultiplexers) used will need the ability to switch paths sequentially for each column of entries. This will require additional control for each multiplexer (or demultiplexer) which, in turn, adds to the complexity of the systems.

Lastly, although the feedback array systems for the vertical or the hybrid decompositions represent an interesting approach to solve the size independent problems, they require overly complex structures and controls while offering no real benefits or throughput improvement over their counterpart for parallel decomposition. These systems are thus impractical for actual implementation.

CHAPTER IV

A NEW SYSTOLIC ARRAY ARCHITECTURE

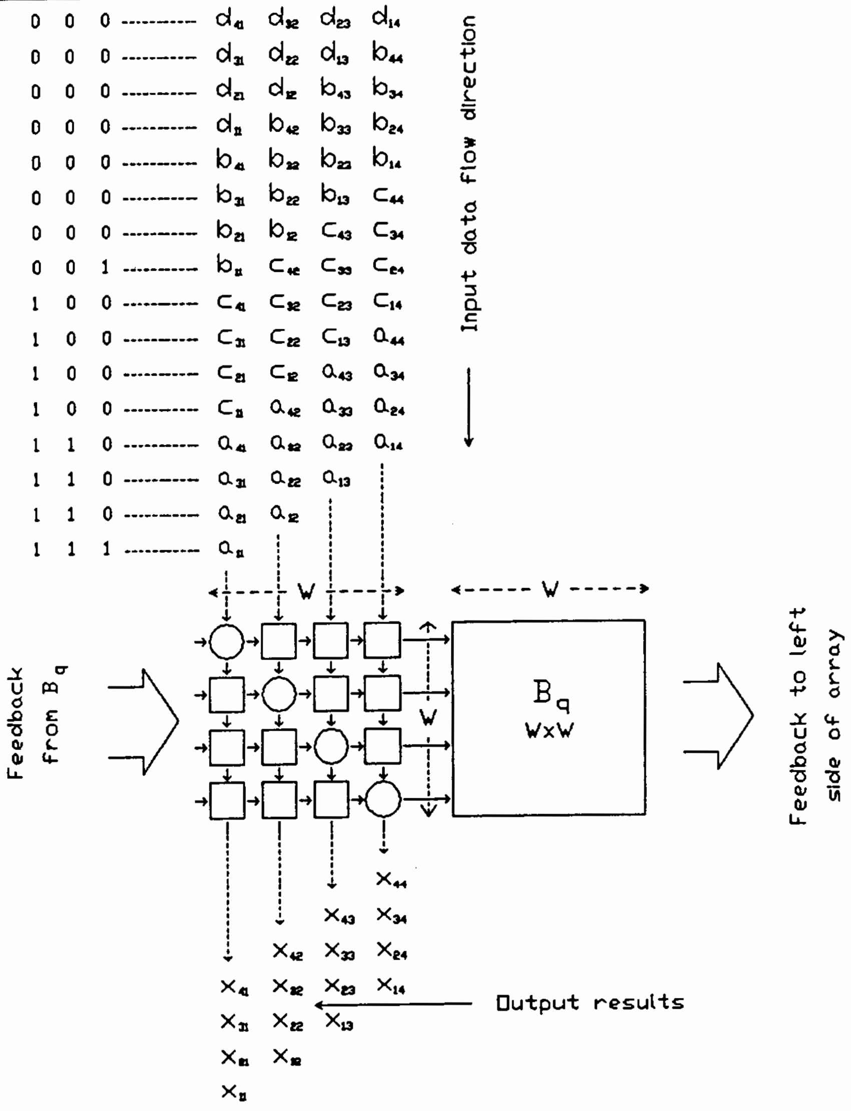
In this chapter, we will introduce a new systolic implementation of Faddeev's algorithm which, in its basic form, reduces the I/O bandwidth requirement by half and the number of cells needed by more than one third. Furthermore, it will eliminate some of the drawbacks that exist in both of the previously described arrays.

ARCHITECTURAL DESCRIPTION

Our design consist of a square array in which the cells are orthogonally connected as illustrated in Figure 21. Data bus interconnections between cells are indicated by arrows. Functionally, there are two types of cells. The first type consists of all the diagonal cells (denoted by circles) of the array, and the second type of all the non-diagonal cells (denoted by squares).

Depending on the actual processing phase, the array functions in one of the two *modes*: the T (triangular) mode or the S (square) mode. Together, these two modes implement Faddeev's algorithm to compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ from (2.2).

Step #	C1	C2	C4				
							d_{44}
						d_{43}	d_{34}
					d_{42}	d_{33}	d_{24}
16	0	0	0	d_{41}	d_{32}	d_{23}	d_{14}
15	0	0	0	d_{31}	d_{22}	d_{13}	b_{44}
14	0	0	0	d_{21}	d_{12}	b_{43}	b_{34}
13	0	0	0	d_{11}	b_{42}	b_{33}	b_{24}
12	0	0	0	b_{41}	b_{32}	b_{23}	b_{14}
11	0	0	0	b_{31}	b_{22}	b_{13}	c_{44}
10	0	0	0	b_{21}	b_{12}	c_{43}	c_{34}
9	0	0	1	b_{11}	c_{42}	c_{33}	c_{24}
8	1	0	0	c_{41}	c_{32}	c_{23}	c_{14}
7	1	0	0	c_{31}	c_{22}	c_{13}	a_{44}
6	1	0	0	c_{21}	c_{12}	a_{43}	a_{34}
5	1	0	0	c_{11}	a_{42}	a_{33}	a_{24}
4	1	1	0	a_{41}	a_{32}	a_{23}	a_{14}
3	1	1	0	a_{31}	a_{22}	a_{13}	
2	1	1	0	a_{21}	a_{12}		
1	1	1	1	a_{11}			



Input data flow direction
W
W
B_q
WxW
Feedback from B_q
Feedback to left side of array
x_{44}
x_{43} x_{34}
x_{42} x_{33} x_{24}
x_{41} x_{32} x_{23} x_{14}
x_{31} x_{22} x_{13}
x_{21} x_{12}
x_{11}
Output results

Figure 21. Dual mode systolic implementation of Faddeev's algorithm. The number of cells needed is smaller and I/O bandwidth requirement is reduced.

When the array is in T mode, cells of rows i where $i = 1, 2, .., w$ and columns j where $j \geq i$, form a triangular sub-array which, based on Gentlemen and Kung's array of Figure 2, performs Gaussian elimination with neighbor pivoting on **A**, and ordinary Gaussian elimination on **C**. During this mode of operation, the circular and square cells essentially carry out the same functions specified by Figure 3 boundary and internal cells, respectively.

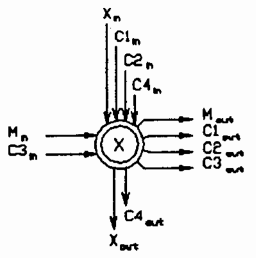
When in S mode, the entire array is used to process **B** and **D**. In this mode, every cell of the array acts similarly to the internal cell of Figure 3, i.e. circular cells functionally become square cells. In order to switch the array from one mode to another, it is only necessary to change the program of the diagonal cells. This is accomplished with cells microprograms listed in Figure 22.

By alternating between the two operational modes T and S, our array essentially simulates the system of Chuang and He (the one-T and one-S system in Figure 8) to solve (2.2) with a smaller number of cells and half the bandwidth requirement. Naturally, the input data flow will have to be slightly modified because of the differences in array's topology.

PEs' Description

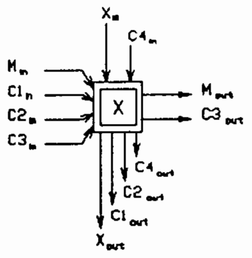
The circular and square cells, as shown in Figure 22, have identical I/O and control bandwidth: two n-bits data input ports, two n-bits data output ports, four one-bit

CIRCULAR CELL :



```
if $C4_{in} = 1$ then
  $X \leftarrow 0.0$ ;
if $C1_{in} = 1$ then
  begin
  if $|X_{in}| \geq |X|$ and $C2_{in} = 1$ then
      begin
      $C3_{out} \leftarrow 1$ ;
      if $X_{in} \neq 0.0$ then
        $M_{out} \leftarrow - X / X_{in}$
      else
        $M_{out} \leftarrow 0.0$ ;
      $X \leftarrow X_{in}$ ;
      end
  else begin
        $C3_{out} \leftarrow 0$ ;
        $M_{out} \leftarrow - X_{in} / X$ ;
        end
  end
else begin        $M_{out}$ [illegible]
     if $C3_{in} = 1$ then
        begin
        $X_{out} \leftarrow X + M_{in} * X_{in}$ ;
        $X \leftarrow X_{in}$ ;
        end
     else $X_{out} \leftarrow X_{in} + M_{in} * X$ ;
     $C3_{out} \leftarrow C3_{in}$ ;
     end ;
$C1_{out} \leftarrow C1_{in}$ ;
$C2_{out} \leftarrow C2_{in}$ ;
$C4_{out} \leftarrow C4_{in}$ ;
```

SQUARE CELL :



```
if $C4_{in} = 1$ then
  $X \leftarrow 0.0$ ;
if $C3_{in} = 1$ then
  begin
  $X_{out} \leftarrow X + M_{in} * X_{in}$ ;
  $X \leftarrow X_{in}$ ;
  end
else $X_{out} \leftarrow X_{in} + M_{in} * X$ ;
$C1_{out} \leftarrow C1_{in}$ ;
$C2_{out} \leftarrow C2_{in}$ ;
$C3_{out} \leftarrow C3_{in}$ ;
$C4_{out} \leftarrow C4_{in}$ ;
$M_{out} \leftarrow M_{in}$ ; ??
```

Figure 22. Microprogram specifications of the circular and square cells for the array's dual mode operation.

control input ports and four one-bit control output ports, for a total bandwidth of $4n+8$. In fact, this number is comparable to the actual pin count that Chuang and He's internal cell (in Figure 9) would need, since their cell does require extra control capabilities to work properly.

Although the choice of processors for our cells will be implementation dependent, the following observations nevertheless can be of support.

Physically, one type of processor can be used to implement both circular and square cells because of the same I/O and control bandwidth requirement and similar general functionalities.

Such a processor would have to be on a single chip for the array's chip count to be kept at a minimum. Another advantage is that functional blocks of the processor can work together without the time and pin-out penalty of off-chip communication.

Internally, the architecture of the processor should allow for a significant amount of parallelism, i.e CPU functions should be partitioned into units that can operate concurrently. To supply data efficiently to these units, multiple internal data buses are essential. Additionally, a horizontal microinstruction set is mandatory to support such a structure; this in turn will dramatically shorten microprograms and will enhance performance.

A large internal storage for microprograms and a microsequencer with good branching facility must be provided by the processor for adequate cell programmability. Also, provision must be made for the transmission of pipelined systolic control signals, which are crucial for run time operation of the array.

And finally, the processor should have fast, on-chip arithmetic and logical capabilities, with a rich set of register files for flexibility of operation.

Because of these atypical requirements, conventional microprocessors which are available commercially are not quite suitable as PEs in a systolic array. For now, dedicated systolic chips are scarce and the few that are being offered on the market lack some of the above features. However, this situation is expected to change soon as the use of systolic architecture will become more widespread.

Control Signals Interconnections

As shown in Figure 21, the circular cell relies on three external control signals C1, C2, and C4 for internal computation and itself generates signal C3, all of which it broadcasts locally to its neighbors for correct operation of the entire array. The square cell uses only C3 and C4, and passes all control signals it receives to neighboring cells unchanged. C1, C2, C3, and C4 are all one-bit boolean values whose functions and interconnection patterns are described below.

C1 controls the behavior of diagonal cells and consequently selects the operation mode of the array. When C1 is *true*, the diagonal cells execute the portion of their code that enables them to function like Kung's boundary cells, thus changing the array into T mode. Otherwise, with

C1 *false*, diagonal cells function like square cells, and the array is in S mode.

Because of the strict timing required, mode switching should occur as entries of the first row of **B** reach each cell, i.e. the switching sweeps across the array in skewed waves as the transition between **C** and **B** flows through the cells. This can be accomplished without the need to address separate control signals to each individual diagonal cell. In fact, C1 needs to be fed only to the top left diagonal cell of the array and, with cell interconnections of Figure 23, will be pipelined through the array to reach every diagonal cell.

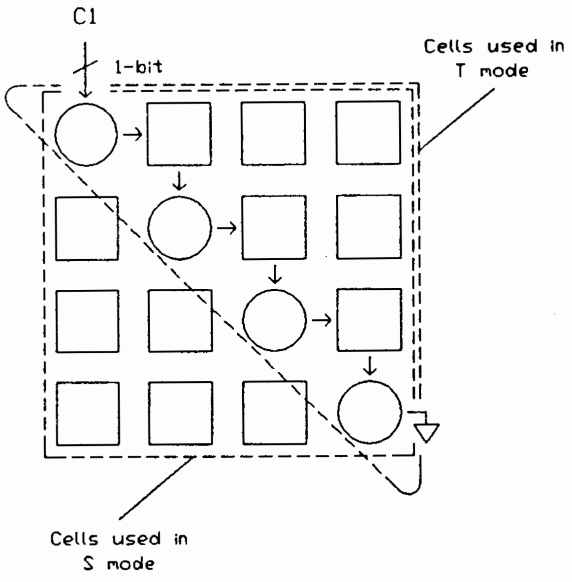


Figure 23. Dual mode array shown only with the interconnection pattern for control signal C1.

As the data flow changes from matrix **A** to matrix **C**, T mode processing in the array gradually switches from Gaussian elimination with pivoting to non-pivoting. This event is started with C2, whose value is *true* for *pivoting allowed* and *false* for *pivoting not allowed*. Again, C2 is fed only to the top left diagonal cell and propagated through the array via the connection patterns shown in Figure 24.

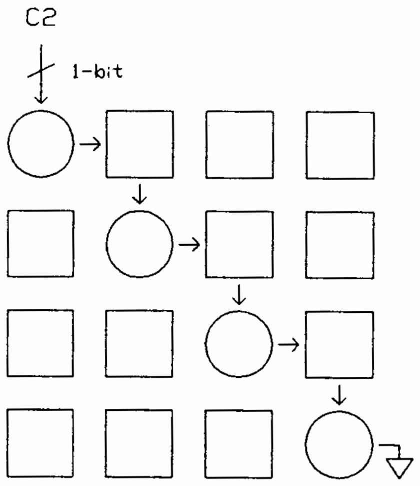


Figure 24. Dual mode array shown only with the interconnection pattern for control signal C2.

Generated internally by diagonal cells when they are in T mode, C3 is the functional equivalent of M_{out} of the boundary cell from Figure 3. It is thus used to direct square cells on the same row to pivot incoming data when *true*, or not to pivot when *false*. Figure 25 shows C3 connections in the array.

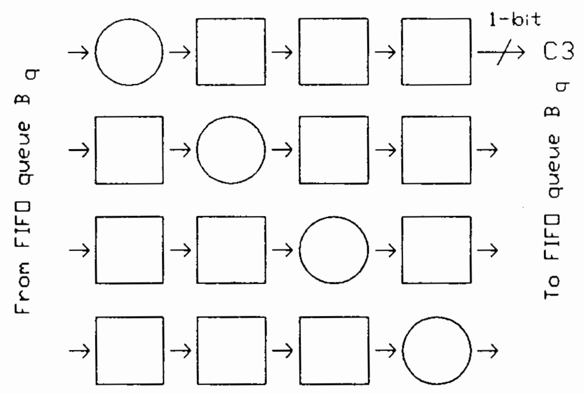


Figure 25. Dual mode array shown only with the interconnection pattern for control signal C3.

When switching between the T and S modes of operation, it is essential that the X registers in each and every cell of the array are cleared to zero *before* the new data elements arrive. If C4 is *true*, a cell will clear its X register prior to receiving X_{in} from its northern neighbor. The X register remains unchanged if C4 is *false*. C4 is distributed throughout the array by the interconnections illustrated in Figure 26.

Control Interface With Host

We have shown how external control signals are distributed throughout the array with only simple and regular interconnections. The need for complex individual cell addressing scheme is thus effectively eliminated while accurate timing at cell level is maintained.

Typically, systolic arrays are attached to a general purpose host running UNIX, an operating system favored by

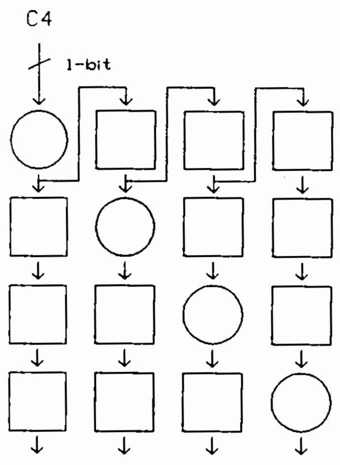


Figure 26. Array showing only the interconnection pattern of control signal C4.

the scientific and engineering community. This is because UNIX provides a programing support environment that is crucial to the development of systolic application software. However, the real time response of such host is inadequate for the critical control timing of systolic arrays. This is due to the software overhead associated with various peripherals supported by the operating system. Thus, the computational power of a systolic array cannot be fully exploited unless effective interface with the host exists.

In our case, a cost effective approach would be to generate and buffer all necessary control signals along with data prior to the initialization of a process; if buffer storage is sufficiently large, multiple problems can be solved by the array in burst before refill is necessary. For a small number of arrays, this approach is efficient and rather simple to implement. However, it becomes less desirable as the number of arrays increases.

A more efficient solution requires the use of a dedicated controller for array management. Advances in VLSI technology today have made the cost of fast and powerful conventional microprocessors very affordable. Acting as an intelligent interface between a slow host and fast arrays, such a device requires minimum supervision from the host while is able to control a large number of attached arrays.

In any case, the sequence of control signals needed by the new array to solve (2.2) is simple and straightforward. The task of programing the host or the controller to generate it is trivial. In the next section, such a sequence will be specified with the corresponding input data flow.

DATA FLOW DESCRIPTION

Again suppose that **A**, **B**, **C** and **D** of (2.2) are $n \times n$ matrices and the available bandwidth is $w = n$. The input data flow, of width n and length $4n$, will be continuous and consists of matrices **A**, **C**, **B** and **D**, in that order, skewed as shown in Figure 21. Note that the control signals necessary for each step are displayed alongside the data flow.

Processing will be as follow. Initially, **A** enters the array followed by **C**; because C4 is *true* (for the duration of one cycle), all cells will clear their X register of values left from any previous problem. With C1 and C2 both *true*, cells of the upper triangle begin performing Gaussian elimination (with neighbor pivoting) to triangularize **A** as its data elements are upon them. As C1 reaches each diagonal cell, the array gradually switches to T mode.

When entry c_{11} of matrix **C** arrives at the top left cell, C2 becomes *false* which disables neighbor pivoting in the diagonal cells. Thus, only the ordinary Gaussian elimination is performed to annul **C**. Throughout this period, C1 remains true, hence the array remains in T mode.

Next, as **B** reaches the array, C4 goes *true* again for the duration of one cycle (step), long enough for the top left cell to store this value; the signal is then propagated to all cells and clears their X registers. At the same time, C1 becomes *false* and remains so until the last row of **D** is in the array. As C1 reaches each diagonal cell, it turns it into a square cell and thus gradually changes the array to S mode as the data elements of **B** are pipelined through the array. The results, shown in Figure 21, fully emerge from the bottom of the array after $6n - 1$ steps for $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ and $5n$ steps for the solution to a linear system.

Storage and Feedback of Modification Factors

During the processing of matrices **A** and **C**, modification factors M_{out} and pivoting control bits C3 are generated by diagonal cells based on incoming values X_{in}. They are then sent rightwards to the square cells on the same row to modify adjacent X_{in} values. As it reaches the edge of the array, this rightward data stream is stored in B_q, a FIFO queue of size $w \times w$ shown in Figure 21. This queue acts as a delay mechanism that will recirculate its contents to the left side of the array for the processing of **B** and **D** as they arrive at the array.

To reduce demands on available bandwidth between the host and the array, B_q should not be implemented using host conventional memory. Instead, the queue should be a dedicated buffer made up entirely of shift registers and run at the same clock rate as the array. This represents the most efficient way to implement the horizontal feedback path.

SOLVING SIZE INDEPENDENT PROBLEMS

Another virtue of the array in Figure 21 is that it can readily handle problems of arbitrary size without requiring any architectural modification. Furthermore, the throughput can be improved proportionally by adding any number of arrays to an existing system. This gives the array a degree of flexibility that makes it truly useful in real life implementation: performance is adjustable according to cost constraint while versatility is preserved regardless of expansion of any size.

For problems larger than array size, the input data flow shown in Figure 21 will be decomposed into smaller strips which are processed continuously by the array, one after another. The intermediate results from each strip will then be fed back to the array for further processing. This vertical feedback and the horizontal feedback of the modification factors constitute two dimensional feedback paths for the array.

Input Decomposition and Vertical Feedback Path

With matrices of size n where n is m times the available bandwidth w, (2.2) can be parallely decomposed into $2m$ strips, each w in width and $2n$ in length as in Figure 11a. Each strip in turn consists of $2m$ $w \times w$ blocks which are of the same size as the array.

For $w = 2$, $n = 4$ and $m = 2$, Figure 27 shows an array with its input data flow decomposed parallely into four strips numbered from V_1 to V_4. These strips are processed by the array one after another continuously. The procedure begins with the array set to T mode as V_1 arrives. While V_1 is being processed, a horizontal data stream consisting of values M_{out} and signals C3 is generated and moved rightwards into B_q. Subsequently, the array is switched to S mode for the computation of the remaining strips, V_2 to V_4. In this mode, the contents of B_q is recirculated back to the array as vertical data of each strip arrive, thus ensuring proper processing.

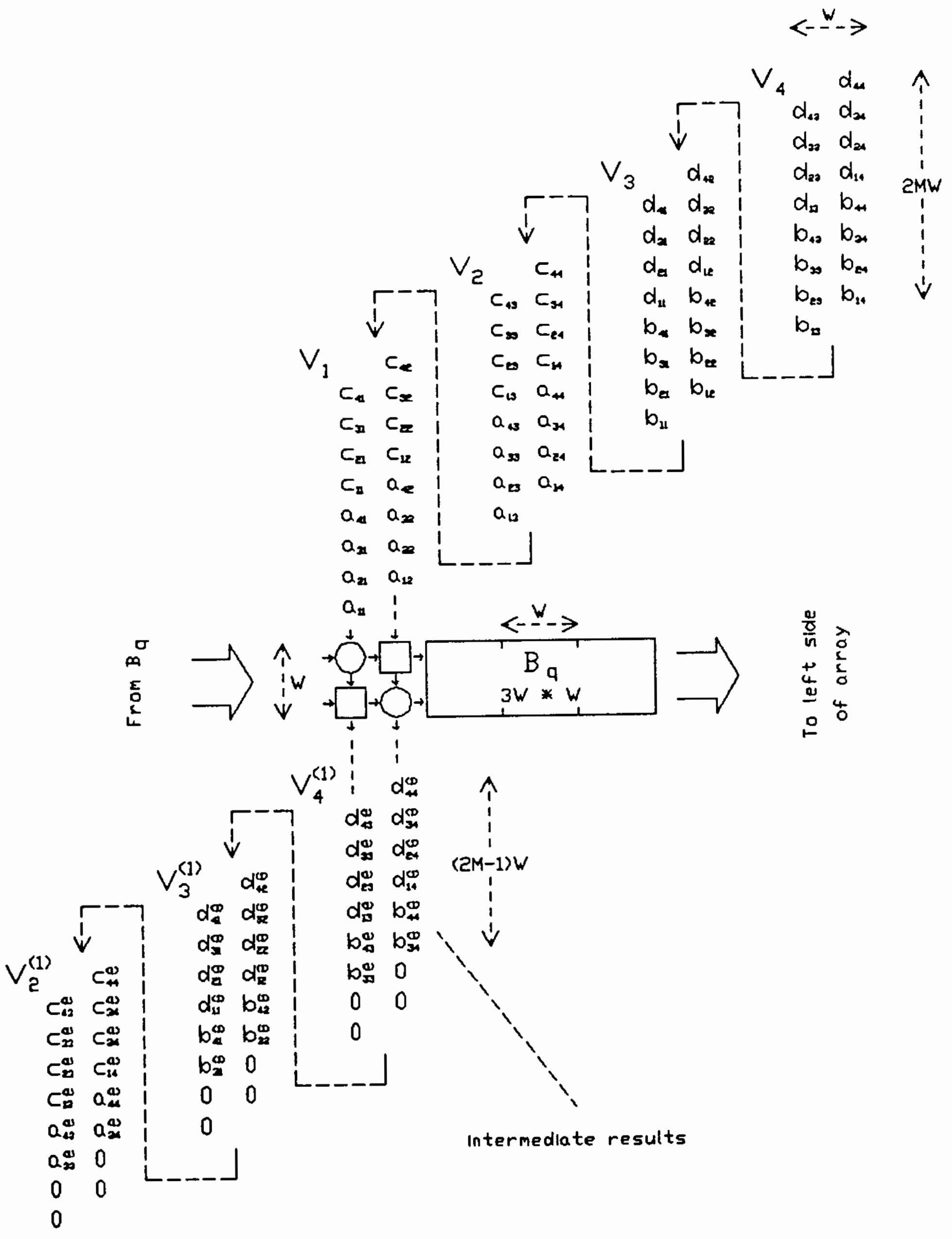


Figure 27. First iteration in the processing of a problem larger than the array size. Note that the strips of intermediate results all have leading blocks of zeroes.

As shown in Figure 27, each input strip V_2, V_3, V_4 generates an output strip $V_2^{(1)}$, $V_3^{(1)}$, $V_4^{(1)}$ of length $(2m - 1)w = 6$ that is preceded by a block of zeroes as it emerges from the array. In Figure 28, these intermediate results are stripped of their zero blocks and then fed back to the array where the above procedure is repeated. The final results, strips E_1 and E_2, come out from the bottom of the array, each $(2m - 2)w = 4$ in length and likewise, is preceded by a zero block.

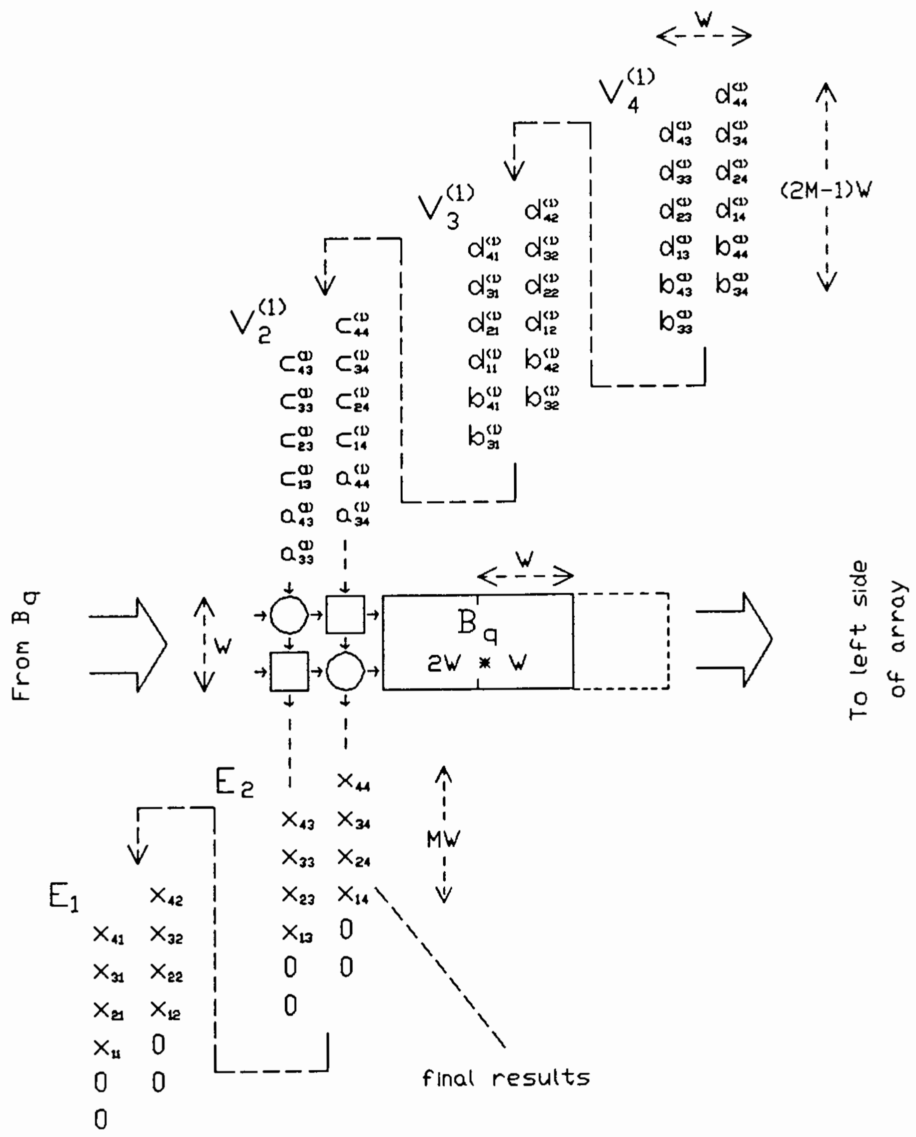


Figure 28. Second iteration of the problem. Intermediate results are stripped of their leading blocks of zeroes before re-entering the array.

Figure 29 shows a mapping of input and output data flow of each iteration to array execution steps. Notice that input data flow of the second iteration is optimized, i.e. zero blocks that exist between output strips of the first iteration are eliminated.

In general, a $w \times w$ array will solve a problem which is decomposed into $2m$ strips of length $2mw$ and width w, in m iterations. During the i^{th} iteration, where $i = 1, 2, \ldots, m$, the array eliminates the strip V_i (in T mode) and reduces the length of each of the remaining strips by w (in S mode). This is because each remaining strip leaves behind one $w \times w$ block of data in the X registers as it is being processed by the array, and subsequently emerges with a $w \times w$ block of zeroes preceding it. These zero blocks can be skipped in the next iteration to shorten processing time without incurring any error. Final results after the m^{th} iteration consists of m strips, each mw in length and w in width.

The number of steps needed for the array of Figure 27 to compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ is:

$$(2w - 1) + \sum_{k=1}^{m} (2m - k + 1)^2 w =$$

$$\frac{7}{3}(m^2 n) + \frac{3}{2}(mn) + \frac{1}{6}(n) + 2w - 1 = O(m^2 n)$$

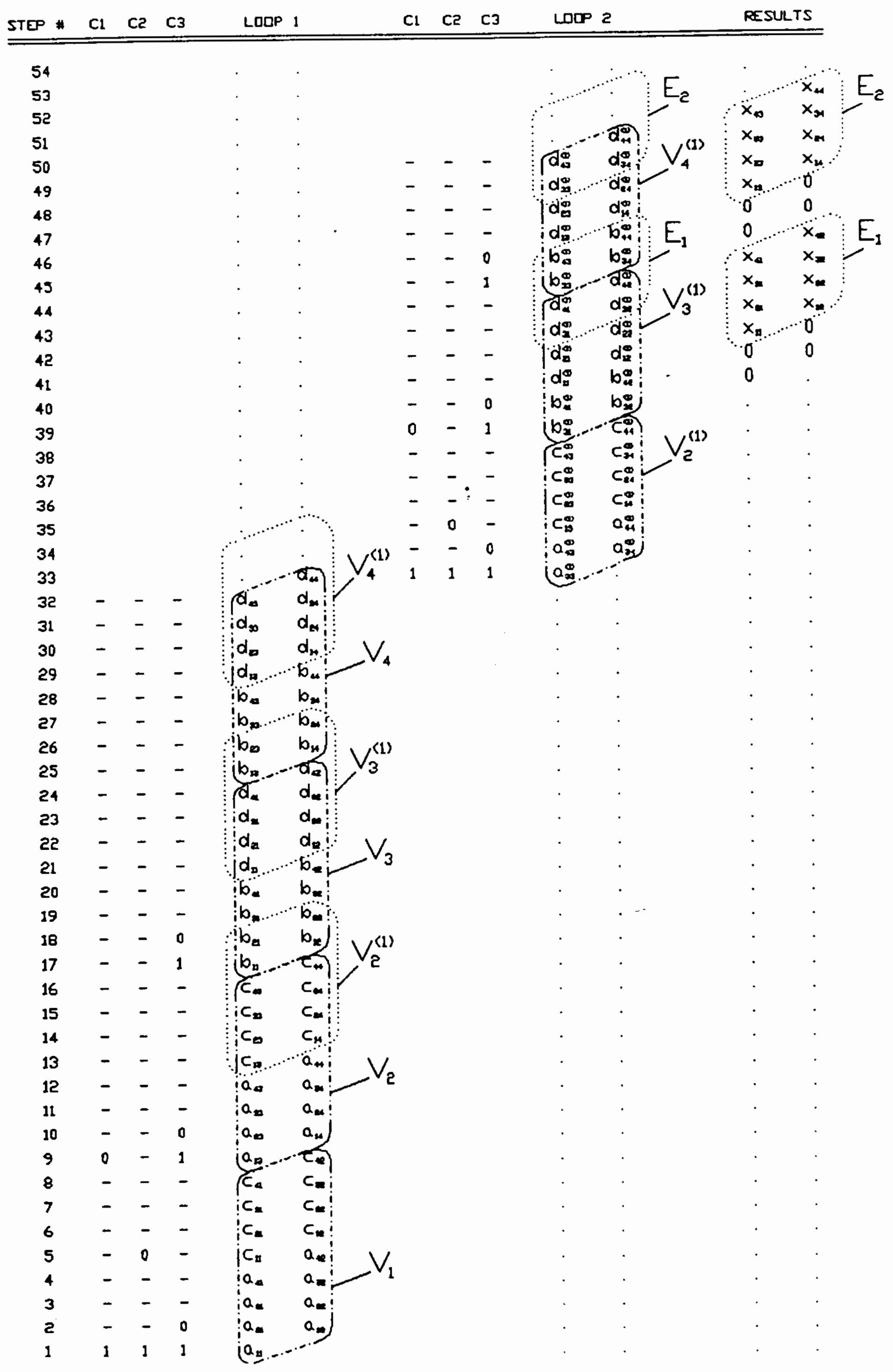


Figure 29. Control/timing sequences of input and output data flow for each iteration. The dash/dotted lines represent input strips, while the dotted lines represent the output strips.

Controls and Horizontal Feedback Path

In Figure 29, values of C1, C2, and C4 necessary for the above example are illustrated at each step. C3 is not shown since it is dependent on input data and generated on the fly by the diagonal cells. For each control signal, a 1 represents the boolean value *true* and 0 represents *false*; when a signal remains unchanged from its previous value, a dash (-) entry is entered. The pattern is as follow: for each iteration, C1 is *true* during the first strip and *false* throughout the remaining strips. C2 is *true* only where pivoting is allowed, i.e. the portion of the first strip which contains data elements of matrix **A**, and *false* anywhere else. C4 clears the X registers of the array each time a new strip arrives, therefore it is *true* at the first step of each strip and *false* elsewhere.

In general, an input strip with N blocks of vertical data will generate a corresponding N blocks of horizontal modification factors pairs (M_{out} and C3); thus, the storage of the horizontal data stream should be N blocks long so that timings for horizontal feedback are accurate. Because the array itself acts as a $w \times w$ block of storage, for each ith iteration, the FIFO queue B_q should be $(2m - i)w$ long. With $m = 2$ and $w = 2$, Figures 27 and 28 show the corresponding length of B_q for each iteration.

The buffer B_q should have the addressing capability such that its length can vary in units of blocks. This permits the array to solve problems of arbitrary size, as long as B_q maximum length is adequate for the largest of them. The control for the addressing can be generated by the host or the dedicated controller.

Multiple Arrays Configurations

Even though both have throughput time $O(m^2 n)$, the system of Figure 13 is slightly faster when compared to the array from Figure 27. Given a problem, the former will solve it with

$$\frac{7}{3}(m^2 n) + \frac{3}{2}(mn) + \frac{1}{6}(n) + 2w - 1$$
$$- \frac{7}{3}(m^2 n) - \frac{5}{3}(n) + 2w + 1 = \frac{3}{2}(m - 1)n$$

steps less than the latter. This stems from its use of two subarrays, where some overlaps in processing are possible when the S array is working on a strip while the T array processes intermediate results from the previous strip.

Likewise, by using multiple arrays, the system of Figure 30 gives better throughput than the single array of Figure 27 under the same I/O constraint. This is because each subarray effectively replaces one iteration, with partial results from one subarray immediately processed by the next, thereby maximizing concurrency while eliminating the corresponding iteration. Such a system will be called L-tuple arrays system (L = 2 in Figure 30), or L-subarrays

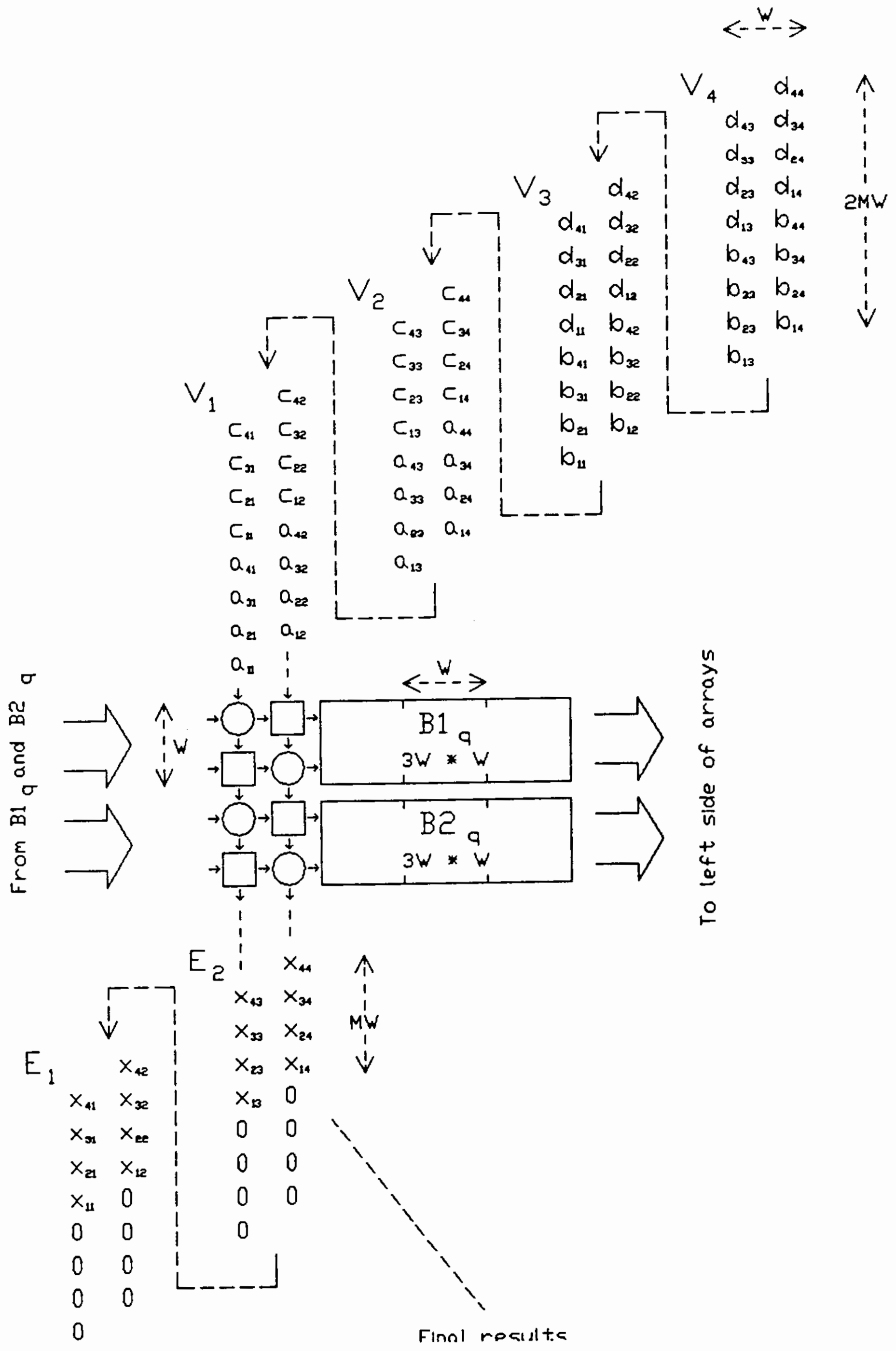


Figure 30. *L*-tuple arrays system processing a problem larger than the I/O bandwidth *w*. Again *w* = 2, *n* = 4 and *m* = 2. With *L* = 2 arrays, the problem is solved in one iteration.

system. In Figure 31, control and timing sequences of Figure 30 subarrays are illustrated. Because the input strips $V_i^{(1)}$ of the second array are interspersed by blocks of zeroes which cannot be removed, buffer $B2_q$ is required to have the same length as $B1_q$, instead of being one block shorter.

In general, a problem requiring m iterations on a single array will need only $k = m / L$ iterations on a system of L-tuple arrays, assuming that m is an exact multiple of L. After each ith iteration, the length of partial results will be $(2m - iL)^2 w$. Hence, the system will compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ of such a problem in

$$(L + 1)w - 1 + \sum_{k=1}^{m/L} \left(2m - (k - 1)L\right)^2 w = \tag{4.1}$$

$$\frac{7}{3}(kmn) + \frac{3}{2}(mn) + \frac{1}{6}(nL) + (L + 1)w - 1 = O(kmn)$$

steps. The first part of (4.1) represents the number of steps taken for input data of the last iteration to traverse the system, and the summation term gives the number of steps to feed input data of all iterations into the system. Final results in this case always emerge from the bottom of the last array of the system.

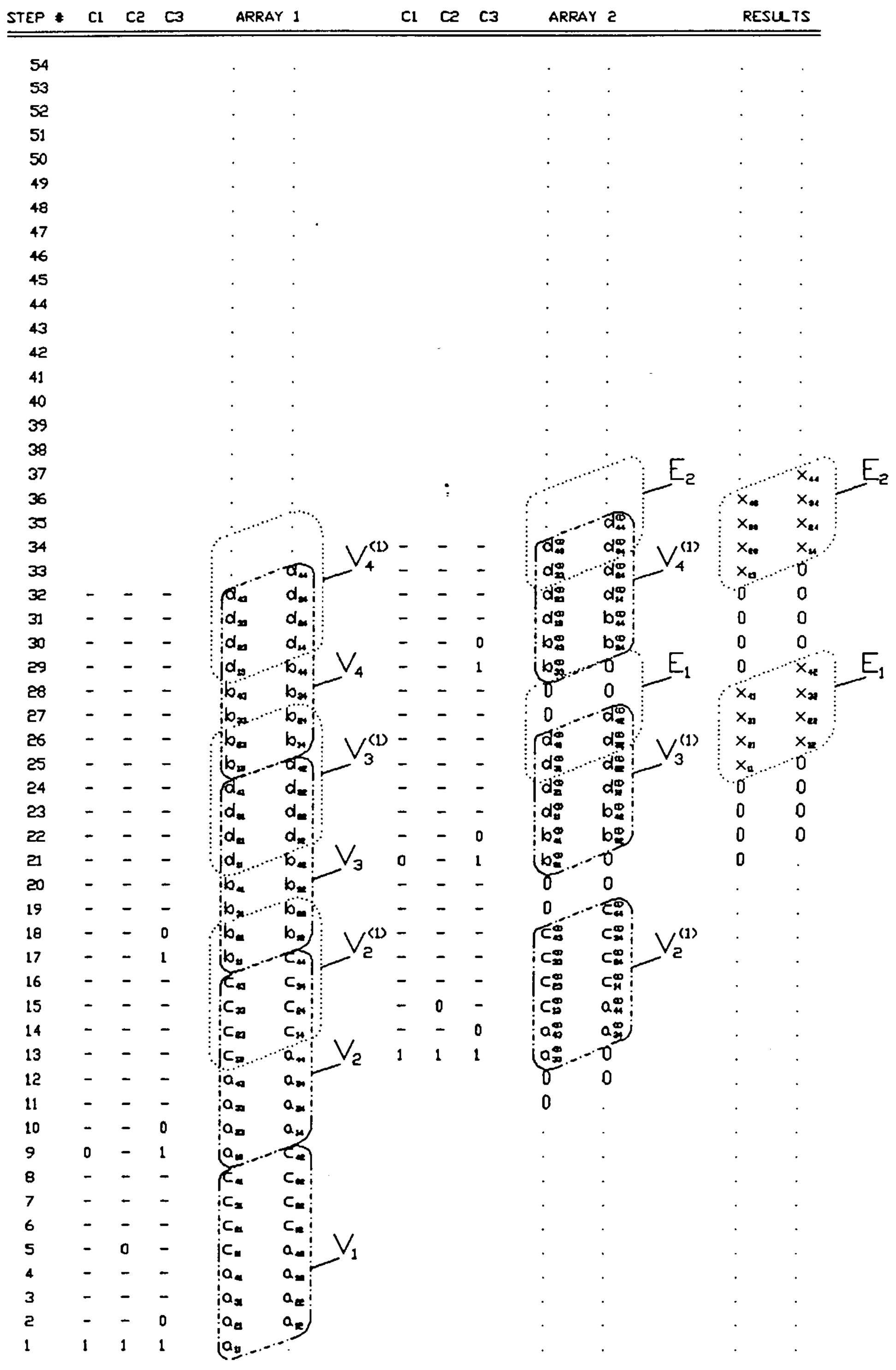


Figure 31. Control/timing sequences for each array. Note that both arrays 1 and 2 process their respective input strips concurrently.

Thus, when $m = L$ (as with the example used in Figure 30), $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ is computed in a single pass with total processing time equal to

$$(4m + 1)n + w - 1 = O(mn)$$

which is identical to the performances of the systems from Figure 13 and 16. However, note that the system of Figure 30 is totally independent of problem's size and the number of cells used is smaller since the T arrays are eliminated.

When m is not an exact multiple of L, that is when $m_{\bmod L} \neq 0$, the number of iterations required to complete the problem is $k = \lceil m/L \rceil$, with the k^{th} iteration employing only the first $m_{\bmod L}$ subarrays of the system. The total processing time will be

$$(m_{\bmod L} + 1)w - 1 + \sum_{k=1}^{\lceil m/L \rceil} \left(2m - (k - 1)L\right)^2 w$$

Again, the summation term represents the time necessary to feed input data of k iterations into the system. However, since only the first $m_{\bmod L}$ subarrays of the system are used during the k^{th} iteration, final results will emerge from the bottom of the $m_{\bmod L}{}^{th}$ subarray, instead of the last subarray. Therefore, the first term of the throughput equation reflects the shorter path through which data has to traverse during the k^{th} iteration. Figure 32

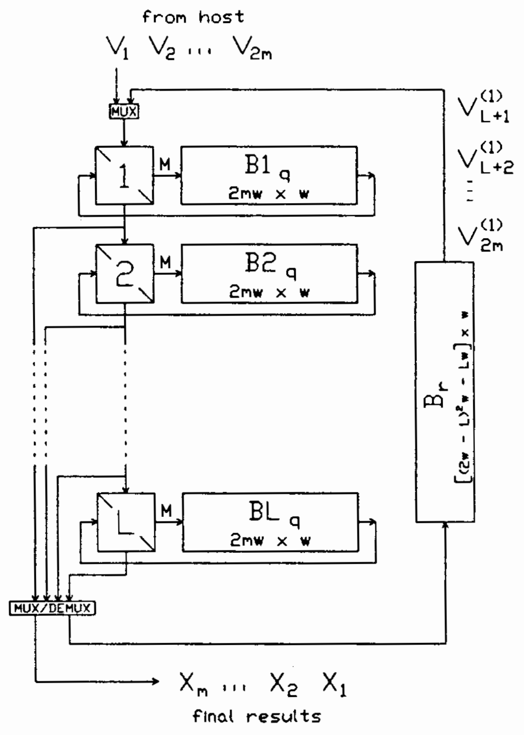


Figure 32. An *L*-tuple arrays system with a common data bus from each array to host. The vertical feedback path has a FIFO queue B_r for temporary storage of intermediate results.

shows a multiple arrays system which provides a common data bus that delivers final results from any one of its subarrays to the host.

Intermediate Results Storage

Until now it was assumed that the intermediate results, generated in between iterations by all of the systems discussed in this chapter, are handled by the host and that the blocks of zeroes can be stripped in the host. However, the resulting back and forth of data between host and system places heavy demands on valuable I/O resources.

A more efficient approach, used in the system of Figure 32, is to route this vertical feedback into the FIFO queue B_r. Similar in concept to the use of B_q for the horizontal feedback, this queue acts as a buffer storage in which intermediate results emerging from the bottom of the system are delayed from being fed back to its top until inputs of the previous iteration are fully processed. An added benefit is that, during processing, the queue can be used to eliminate zero blocks generated by temporarily halting the pipeline for some corresponding durations.

B_r should be $(2m - L)^2w - Lw$ long, i.e. long enough to accommodate partial results of the first iteration of the largest problem likely to be solved by the system, minus the combined length of all subarrays. And since each iteration produces ever shorter output streams, like B_q, B_r should also be given the addressing capability which allows its length to be altered by an external control. This ensures that data enters the array continuously for maximum throughput.

PROCESSING OF SPARSE MATRICES

Another feature which further enhances the versatility of our array is that it can compute problems involving sparse matrices efficiently by skipping blocks of zeroes, similar to the system from Figure 19. Furthermore, because the design functions in both triangle and square mode, only one array is needed for problems of such type. While a multi-array system like that in Figure 32 is fully capable of processing sparse matrices efficiently, the procedure involves only the first array; thus, in Figure 33, it was reduced to a single array system for the sake of clarity. In the following discussion, the example (3.1) will be used, with $p = 3$ and the input data flow decomposed parallely like in Figure 18. Because only one array is needed, the continuous stream of input data alternates between non-zero blocks of strips V_1, V_2, ..., V_m which are processed by the array in T mode, and the corresponding blocks of strip V_{m+1}, processed in S mode.

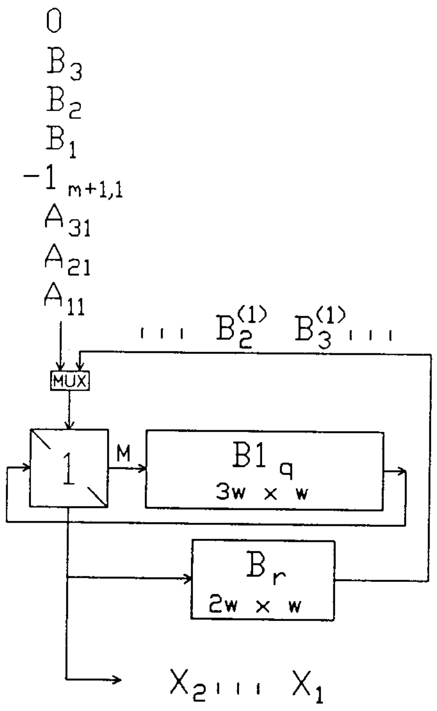


Figure 33. Reduced system for sparse matrix processing.

Initially, non-zero blocks $\mathbf{A}_{11}$, $\mathbf{A}_{21}, \ldots, \mathbf{A}_{p1}$ and block $-1_{m+1,1}$ of strip V_1 are fed into the array. They in turn generate corresponding blocks of M_{out} and C3 which move rightward into buffer $B1_q$. Of length pw, $B1_q$ is long enough to provide the required delay so that its contents can be used by the array (in S mode) to modify subsequent blocks $\mathbf{B}_1$, $\mathbf{B}_2, \ldots, \mathbf{B}_p$ and the first zero block below $\mathbf{B}_m$. Thereafter, $\mathbf{B}_1$ is left stored in the array, whereas $\mathbf{B}_2, \ldots, \mathbf{B}_p$ emerge from the array as $\mathbf{B}_2^{(1)}, \ldots, \mathbf{B}_p^{(1)}$, to be stored in queue B_r. Thus, the capacity of B_r should be $(p - 1)w$ to hold these modified B blocks. The zero block, after modification, becomes the first block of result $\mathbf{X}_1$ and is sent to the host.

From V_2 to V_m, the computation proceeds similarly with blocks $\mathbf{A}_{i,\ldots,p+i-1,i}$ and $-1_{m+i,i}$ of strip V_i generating their own M_{out} and C3 values to modify $\mathbf{B}^{(i-1)}_{i,\ldots,p+i-2}$, $\mathbf{B}_{p+i-1}$ and zero block $\mathbf{B}_{m+i}$. The modified block $\mathbf{B}_i^{(i-1)}$ is then left in the array; blocks $\mathbf{B}^{(i-1)}_{i+1,\ldots,p+i-2}$, $\mathbf{B}_{p+i-1}$ become blocks $\mathbf{B}^{(i)}_{i+1,\ldots,p+i-1}$ which are then stored in B_r for the succeeding strip V_{i+1}, and the modified $\mathbf{B}_{m+i}$ emerges from the array to become the result $\mathbf{X}_i$.

The throughput time of this system is

$$\left(m(p + 1) - \sum_{k=1}^{p-1} k\right)2w + w - 1 =$$

$$2n(p + 1) - pw(p - 1) + w - 1$$

which nearly doubles the throughput time of the system from Figure 19. This is to be expected since the single array from Figure 33 system is doing the work of two. However, such a comparison would be misleading because it does not take into account the fact that, for the two subarrays T and S of Figure 19 to work concurrently, the total I/O bandwidth of that system would have to be $2w$. Or to put it in another way, with a total I/O bandwidth of w, these two subarrays will each have only a bandwidth of $w/2$. Consequently, a problem will have to be decomposed into twice as many input data strips with width that are only half as wide. This effectively doubles the throughput time of the system such that it is actually comparable to that of Figure 33.

OVERLAPS IN PROCESSING BETWEEN PROBLEMS

In the simplest term, a systolic architecture can be thought of as a pipeline architecture in which each row of cells of subarrays in the system represents a stage in the pipeline. A pipeline reaches its peak performance when it outputs a usable piece of data for each of its cycles. This peak performance is attained only after the pipeline is completely filled with data, a process termed *pipeline fill*. To maintain its peak performance, the pipeline must be fed continuously.

Similarly, a systolic system can reach its *maximum throughput rate* only after it is completely filled with data. This maximum throughput rate is defined as the rate in which the solution sets to problems emerge from the system, with minimum times elapse between any two consecutive sets. Note that these elapsed times between solution sets may be of different lengths since the sizes of the problems themselves can vary. To maintain this maximum throughput rate, the input data flow must be continuous, i.e. problems to be solved must be fed into the system without any empty gap in between them. An empty gap in the data flow will result in a corresponding length of time during which cells are idle, and solutions to problems will be that much farther apart. A gap which exceeds the total length of the system will cause the system to completely empty itself of data, resulting in what is commonly termed a *pipeline flush*. A pipeline flush is expensive because it takes a finite amount of time to refill a system.

To put in another way, the maximum throughput rate of a systolic system is achievable and, more important, sustainable only if processing overlaps between problems are fully exploited. Say that two matrix problems, P_P and P_N, are to be solved in that order by a system of L subarrays. For an I/O bandwidth w, P_P is decomposed into m_P data strips. A processing overlap between P_P and P_N occurs when data of the last iteration of P_P and data of the first iteration of P_N are processed by the system at the same time. Maximizing this processing overlap can shave off substantial amount of computing time from P_N. It can be seen that the time saved, in number of steps, is calculated as the number of subarrays through which data of the last iteration of P_P must travel, times the size w of these subarrays, plus the skew factor $w - 1$ of the data flow. Thus, when m_P is an exact multiple of L, the total number of cycles necessary to solve P_N is reduced by

$$(L + 1)w - 1$$

When m_P is not an exact multiple of L, the last iteration of P_P involved only $m_{P \bmod L}$ subarrays of the system. Therefore, P_N is solved with

$$(m_{P \bmod L} + 1)w - 1$$

less cycles. Lastly, if P_P is a sparse matrix as described in the previous section, the number of cycles reduced from the computation of P_N will always be

$$2w - 1$$

This is because sparse matrices are processed only by the first array of the system.

CHAPTER V

EXTENSIONS TO FADDEEV'S ALGORITHM AND CONCLUSION

In the previous chapter, the reader has seen the ease with which the new systolic array uses massive parallelism to solve many types of matrix problems via Faddeev's algorithm. The actual size of the array, and therefore its throughput, is shown to be restricted *only* by the available bandwidth between the host and the array. Even this restriction is effectively circumvented when a number of such arrays are combined into a system to give a desired level of performance. Such a multiple arrays system reach its maximum throughput rate when its pipeline is completely filled with data. By ensuring that the input data flow is continous, this maximum throughput rate is maintained at all times. It would seem then, algorithmically speaking, that nothing further can be done to induce more parallelism into matrix computations.

However, that last observation is simply not true. We have found that, by extending Faddeev's algorithm, the maximum throughput rate of a system can be nearly quadrupled. Furthermore, such a tremendous improvement in system throughput requires absolutely no architectural modification to the system.

HORIZONTAL EXTENSION TO FADDEEV'S ALGORITHM

Before illustrating how we extend Faddeev's algorithm, let us introduce the concept of *compatibility* between matrix problems. Suppose we have matrices **A**, **B** and **D** of order n, upon which we wish to perform the operations $\mathbf{A}^{-1}$, $\mathbf{A}^{-1}\mathbf{B}$ and $\mathbf{A}^{-1} + \mathbf{D}$. From Figure 2, we can solve these matrix problems with Faddeev's algorithm by formulating them as

$$\underset{(1)}{\begin{array}{c|c} \mathbf{A} & \mathbf{I} \\ \hline -\mathbf{I} & 0 \end{array}} = \mathbf{A}^{-1} \qquad \underset{(2)}{\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{I} & 0 \end{array}} = \mathbf{A}^{-1}\mathbf{B} \qquad \underset{(3)}{\begin{array}{c|c} \mathbf{A} & \mathbf{I} \\ \hline -\mathbf{I} & \mathbf{D} \end{array}} = \mathbf{A}^{-1}+\mathbf{D} \tag{5.1}$$

where **I** is the identity matrix. These constructs reveals that they all have identical left halves, i.e. they consist of the same matrix **A** in their top left quadrant and the same matrix **-I** in their bottom left quadrant. When this is the case, we say that the problems are *horizontally compatible*.

Obviously, solving x horizontally compatible problems involves repeating the calculations for the same left side x number of times. In the case of (5.1) where $x = 3$, solving (1), (2) and (3) requires repeating the process of triangularizing **A** and annulling **-I** three times. If by some means the redundant iterations of this process are eliminated, nearly half of the calculations necessary to solve (2) and (3) of (5.1) can be skipped. This would yield a large savings in computing time.

To accomplish this, we extend Faddeev's algorithm horizontally to the right so that (5.1) is reformulated as

$$\begin{array}{c|c|c|c} \mathbf{A} & \mathbf{I} & \mathbf{B} & \mathbf{I} \\ \hline -\mathbf{I} & 0 & 0 & \mathbf{D} \\ & (1) & (2) & (3) \end{array} \tag{5.2}$$

Grouping (1), (2) and (3) together as in (5.2) allows us to triangularize **A** and annul **-I** only once, and reuse the multipliers generated from that several times on the right. The results will appear as

$$\begin{array}{c|c|c|c} \mathbf{A}^{(k)} & \mathbf{I}^{(k)} & \mathbf{B}^{(k)} & \mathbf{I}^{(k)} \\ \hline 0 & \mathbf{A}^{-1} & \mathbf{A}^{-1}\mathbf{B} & \mathbf{A}^{-1}+\mathbf{D} \\ & (1) & (2) & (3) \end{array}$$

It is easy to see that the horizontal extension to Faddeev's algorithm maps particularly well to a system using our systolic array design: it requires absolutely no architectural nor algorithmic modification, either at the system level, subarray level or cell level. When the available I/O bandwidth is w, (5.2) is parallely decomposed into $(x + 1)m$ input strips, each $2mw$ in length, as shown in Figure 34.

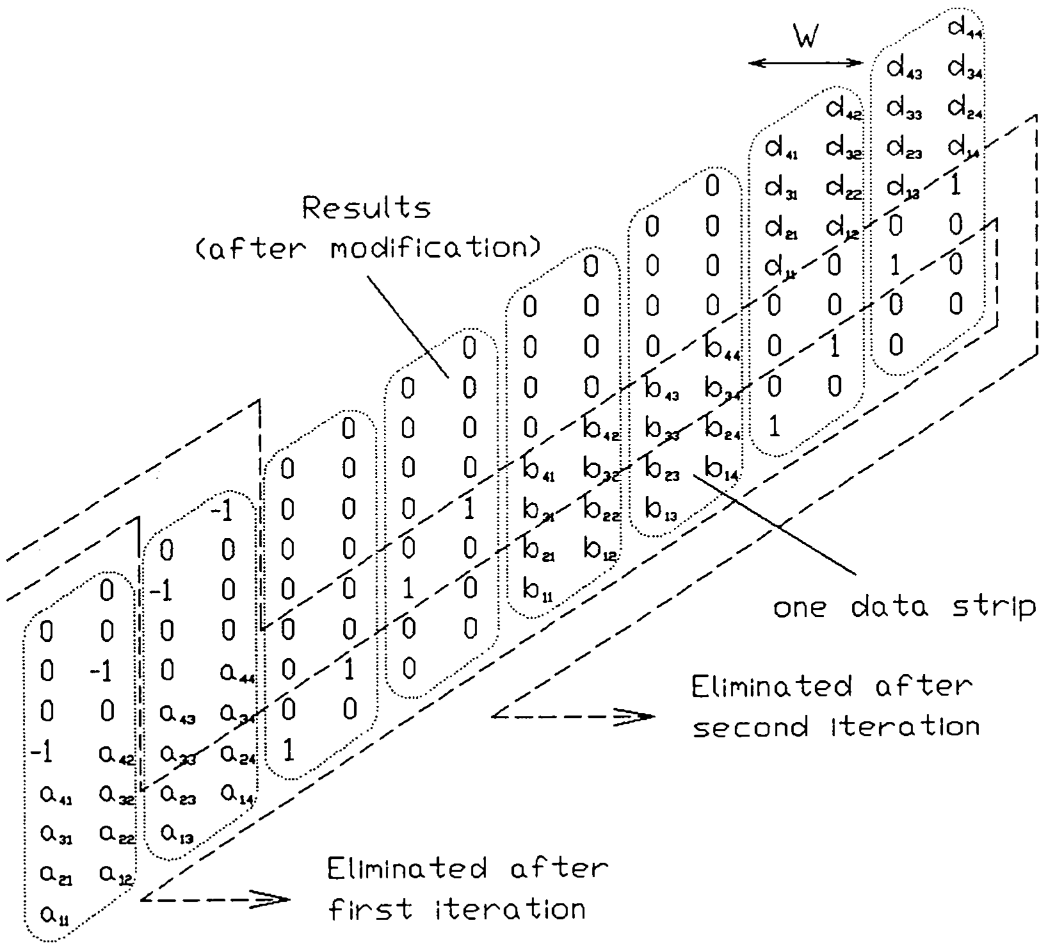

W
Results (after modification)
one data strip
Eliminated after second iteration
Eliminated after first iteration

Figure 34. Parallel decomposition of $x = 3$ horizontally compatible problems. For this example, $n = 4$, $w = 2$ and $m = 2$.

As before, the L-subarrays system of Figure 32 will process this input data flow in k iterations, where the value of k depends on m and L. When m is an exact multiple of L, we have $k = m/L$ and the system will compute x horizontally compatible problems in

$$(L + 1)w - 1 + \sum_{k=1}^{m/L} \left[(x + 1)m - (k - 1)L\right]\left[2m - (k - 1)L\right]w \tag{5.3}$$

cycles. In the above equation, the first product term of the summation represents the number of input strips for each iteration, while the second term indicates the strips length. The solution to the first problem will come out after

$$(L + 1)w - 1 + \sum_{k=1}^{(m/L)-1} \left[(x + 1)m - (k - 1)L\right]\left[2m - (k - 1)L\right]w$$
$$+ (m + L)^2 w$$

cycles, with the second line of the equation indicating that only part of the k^{th} iteration is needed. Afterward, solutions to subsequent $(x - 1)$ problems are outputed one for every $(m + L)n$ cycles. In the special case when $m = L$, we have $k = 1$ and the system will solve the first problem in

$$(4m + 1)n + w - 1$$

cycles. As to subsequent problems, the system will complete one every $2mn$ cycles. The difference between the two throughput equations of the first problem is due to the fact that the input data flow for x horizontally compatible problems consist of $(x - 1)m$ more strips than that of a single problem. This means that during each iteration, the system has that many more strips to process. Thus when $k > 1$, the previous iterations will delay the output of results whereas with $k = 1$, those delays are non-existent.

When m is not an exact multiple of L, the number of iterations required for the system to process (5.2) is $k = \lceil m/L \rceil$, with the k^{th} iteration involving only the first $m_{\mathrm{mod}\, L}$ subarrays of the system. The total throughput will be

$$(m_{\mathrm{mod}\, L} + 1)w - 1 + \sum_{k=1}^{\lceil m/L \rceil} \left[(x + 1)m - (k - 1)L\right]\left[2m - (k - 1)L\right]w \tag{5.4}$$

with solution to the first problem coming out after

$$(m_{\mathrm{mod}\, L} + 1)w - 1 + \sum_{k=1}^{\lfloor m/L \rfloor} \left[(x + 1)m - (k - 1)L\right]\left[2m - (k - 1)L\right]w$$
$$+ (m + m_{\mathrm{mod}\, L})^2 w$$

cycles. Again, the second line of the above equation indicates that only part of the last iteration is needed by the system to compute the first problem. Afterward, solutions to subsequent $x - 1$ problems will emerge one for every $(m + m_{\mathrm{mod}\, L})n$ cycles.

Since the input data flow of x horizontally compatible problems consists of only $(x + 1)m$ strips, versus the $2xm$ strips required if they are not compatible, large saving in storage space can be gained on the host side. On the other hand, the length of the FIFO buffer B_r should be $((x + 1)m - L)(2m - L)w - Lw$ since the intermediate results after the first iteration have many more strips. Because the length of each strip is still $2mw$, the capacity of the buffers B_q should remain unchanged.

To get an idea of how much the system throughput can be improved when horizontal extension is applied, suppose that we have a system of $L = 4$ subarrays, with each array of size $w = 32$. On this system, we wish to perform $x = 50$ operations with matrices of order $n = 128$. If these operations are not compatible, solving them one at a time without processing overlaps will take a total of 110,350 steps. With processing overlaps, this number is reduced to 102,559. However, if the operations are horizontally compatible, they can be processed by the system in 52,383 steps. The improvement in throughput is

$$\frac{102{,}559}{52{,}383} = 1.96,$$

nearly by a factor of two. Of course, this number can vary depending on x. As x gets larger, the improvement factor gets closer to two.

VERTICAL EXTENSION TO FADDEEV'S ALGORITHM

Even when a group of matrix problems are not horizontally compatible, they may exhibit another type of compatibility which can also be exploited to give an equivalent speedup in system throughput. To expand on this, let's suppose that we have $y = 3$ matrix operations to perform, namely **CB**, **B** + **D** and **EB** + **D** where **B**, **C**, **D** and **E** are of order n. Like before, we can express these problems as

$$\underset{(1)}{\begin{array}{c|c} \mathbf{I} & \mathbf{B} \\ \hline -\mathbf{C} & 0 \end{array}} = \mathbf{CB} \qquad \underset{(2)}{\begin{array}{c|c} \mathbf{I} & \mathbf{B} \\ \hline -\mathbf{I} & \mathbf{D} \end{array}} = \mathbf{B+D} \qquad \underset{(3)}{\begin{array}{c|c} \mathbf{I} & \mathbf{B} \\ \hline -\mathbf{E} & \mathbf{D} \end{array}} = \mathbf{EB+D} \qquad (5.5)$$

Because the left side of problems (1), (2) and (3) of (5.5) are not the same, they are not horizontally compatible. However, it can be observed that they all have the identity matrix **I** in their top left quadrant and matrix **B** in their top right quadrant. To put it differently, these problems all have identical top half. When this is the case, we say that the problems are *vertically compatible*.

To avoid repeating the same calculations on the identical top sides of vertically compatible problems, we extend Faddeev's vertically such that (5.5) becomes

$$\begin{array}{c|cl} \mathbf{I} & \mathbf{B} & \\ \hline -\mathbf{C} & 0 & (1) \\ \hline -\mathbf{I} & \mathbf{D} & (2) \\ \hline -\mathbf{E} & \mathbf{D} & (3) \end{array} \qquad (5.6)$$

When y vertically compatible problems are grouped together as in (5.6), the common top side needs to be processed only once. This means that after the top left quadrant is triangularized and the top right quadrant is modified with the generated multipliers, they can be used repeatedly to annul the left side of succeeding stages and transform their right side into solutions.

In the case of (5.6), solving it involves only the annulment **-C**, **-I** and **-E**. This is because the identity matrix **I** in the top left quadrant is, by its nature, already triangularized; as a consequence, matrix **B** in the top row will remain unmodified. Annulling **-C**, **-I** and **-E** while extending the operations to the right will give

$$\begin{array}{c|cl} \mathbf{I} & \mathbf{B} & \\ \hline 0 & \mathbf{CB} & (1) \\ \hline 0 & \mathbf{B+D} & (2) \\ \hline 0 & \mathbf{EB+D} & (3) \end{array}$$

which shows the solutions to (1), (2) and (3) in the right quadrants.

As with horizontal extension, systems using our array design can handle vertical extension to Faddeev's algorithm without any modification. Shown in Figure 35, the input

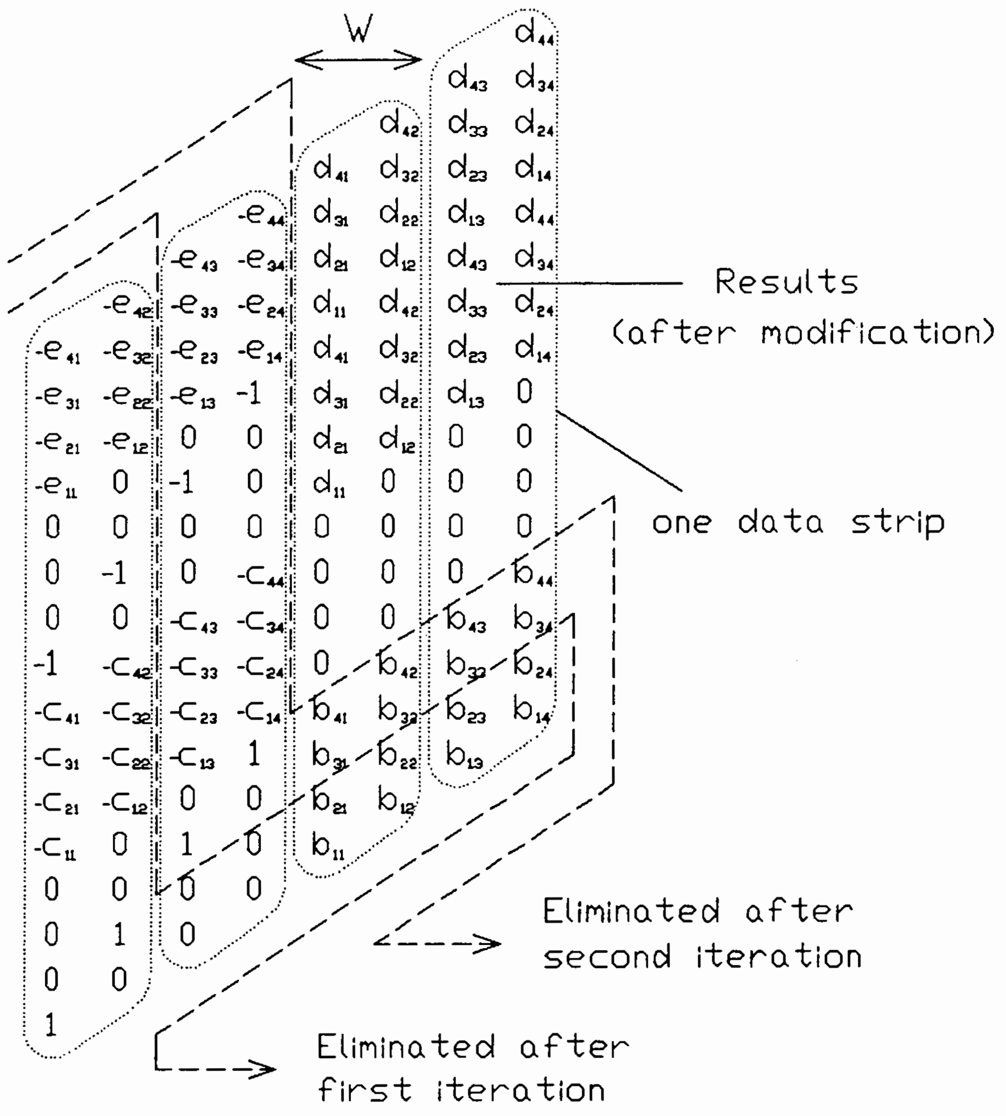


Figure 35. Parallel decomposition of $y = 3$ vertically compatible problems. Again $n = 4$, $w = 2$ and $m = 2$.

data flow of y vertically compatible problems consists of $2m$ strips, where each strip is $(y + 1)m$ blocks long. The L-subarrays system of Figure 32 will process this data flow in k iterations. When m is an exact multiple of L, $k = m/L$ and the process will be completed in

$$(L + 1)w - 1 + \sum_{k=1}^{m/L} \left[2m - (k - 1)L\right]\left[(y + 1)m - (k - 1)L\right]w \tag{5.7}$$

cycles. When m is not an exact multiple of L, $k = \lceil m/L \rceil$ and the throughput is computed as

$$(m_{\bmod L} + 1)w - 1 + \sum_{k=1}^{\lceil m/L \rceil} \left[2m - (k - 1)L\right]\left[(y + 1)m - (k - 1)L\right]w \tag{5.8}$$

In throughput equations (5.7) and (5.8), the first product term within the summation represents the number of input strips for each iteration. The length of each strip, on the other hand, is indicated by the second product term. Even so, note that (5.7) and (5.8) are identical to (5.3) and (5.4), respectively, save for the variables x and y.

After the k^{th} iteration, the set of y solutions emerges in m output strips. As shown in Figure 35, an output strip consists of y segments, each of width w and length mw. Each segment $i = 1, 2, \ldots, y$ is part of the solution to the i^{th} problem. Because a solution is divided into m segments with each segment part of an output strip, the solutions will not be completely out until the last strip has emerged. Thus, the number of steps needed for the first solution to come out is computed by subtracting $(y - 1)mw$ from (5.7) or (5.8). Each following solutions takes another mw steps.

Again, storage space needed on the host side is greatly reduced since the input data flow of y vertically compatible problems is only $2(y + 1)m^2w$ long, as opposed to $4ym^2w$ were they not compatible. However, the length of the FIFO buffer B_q should be $((y + 1)m - 1)w$ to accommodate longer strips of modification factors. In addition, the length of B_r should be $(2m - L)((x + 1)m - L)w - Lw$ to adequately hold intermediate results with longer strips.

TWO-DIMENSIONAL EXTENSION TO FADDEEV'S ALGORITHM

While using either one of the previously described extensions yields substantial reduction in computing time, still greater improvement in throughput is possible when both techniques are combined into a two-dimensional extension to Faddeev's algorithm. To illustrate, consider the matrix operations **AB**, **AE** + **F**, **B** + **D** and **E** + **G**. As before, **A**, **B**, **D**, **E**, **F** and **G** are all matrices of order n. Formulating the operations as follow:

$$\begin{array}{cc} \underset{(1)}{\begin{array}{c|c} I & B \\ \hline -A & 0 \end{array}} = AB & \underset{(2)}{\begin{array}{c|c} I & E \\ \hline -A & F \end{array}} = AE+F \\ \\ \underset{(3)}{\begin{array}{c|c} I & B \\ \hline -I & D \end{array}} = B+D & \underset{(4)}{\begin{array}{c|c} I & E \\ \hline -I & G \end{array}} = E+G \end{array} \tag{5.9}$$

reveals that (1) and (2) are horizontally compatible, as with (3) and (4). Furthermore, (5.9) also shows that (1) and (3) are vertically compatible, as with (2) and (4). Thus, using horizontal extension, (5.9) becomes

$$\begin{array}{c} \begin{array}{c|c|c} I & B & E \\ \hline -A & 0 & F \\ \multicolumn{1}{c}{} & \multicolumn{1}{c}{(1)} & \multicolumn{1}{c}{(2)} \end{array} \\ \\ \begin{array}{c|c|c} I & B & E \\ \hline -I & D & G \\ \multicolumn{1}{c}{} & \multicolumn{1}{c}{(3)} & \multicolumn{1}{c}{(4)} \end{array} \end{array} \tag{5.10}$$

Since both constructs of (5.10) have identical top halves, vertical extension can also be used to further obtain

$$\begin{array}{c|c|cl} I & B & E & \\ \hline -A & 0 & F & (1) \text{ and } (2) \\ \hline -I & D & G & (3) \text{ and } (4) \end{array} \tag{5.11}$$

This results in a two-dimensional extension to Faddeev's algorithm. Annulling **-A** and **-I** in (5.11) and extending the operations to its right prompt the solutions to (1), (2), (3) and (4) to appear as

I	B	E	
0	AB	AE+F	(1) and (2)
0	B+D	E+G	(3) and (4)

As (5.11) reveals, the two-dimensional extension to Faddeev's algorithm allows a compatible matrix problem to share three of its quadrants with others, instead of two. This translates into the elimination of a larger number of calculations per problem.

The input data flow of (5.11) for the *L*-subarrays system is shown in Figure 36. When the number of problems is *x* across by *y* long, the input data flow is decomposed into $(x + 1)m$ parallel strips, each $(y + 1)mw$ in length. If *m* is an exact multiple of *L*, the total number of steps for the *L*-subarrays system of Figure 32 to process this data flow is

$$(L + 1)w - 1 + \sum_{k=1}^{m/L} \Big[(x + 1)m - (k - 1)L\Big] \Big[(y + 1)m - (k - 1)L\Big]w \tag{5.12}$$

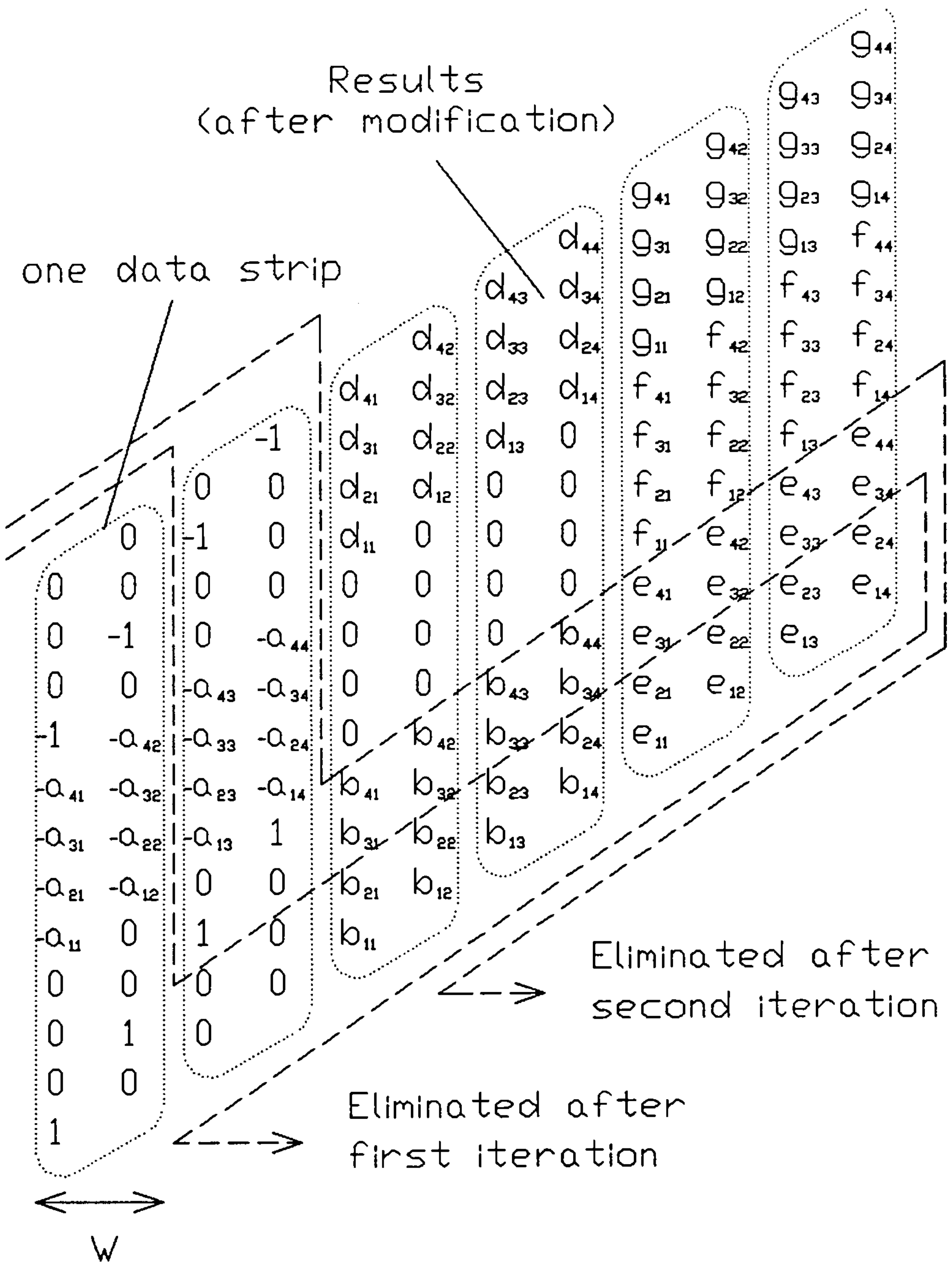

Results
(after modification)
one data strip
Eliminated after second iteration
Eliminated after first iteration
W

Figure 36. Parallel decomposition of x by y compatible problems. $x = 2$ is the number of horizontally compatible problems, and $y = 2$ is the number of vertically compatible problems. As before, $n = 4$, $w = 2$ and $m = 2$.

If m is not an exact multiple of L, then the number of steps needed is computed as

$$(m_{\bmod L} + 1)w - 1 + \sum_{k=1}^{\lceil m/L \rceil} \left[(x + 1)m - (k - 1)L\right] \left[(y + 1)m - (k - 1)L\right]w \qquad (5.13)$$

Subtracting $[(x - 1)(ym + L) + (y - 1)]mw$ from (5.12) or $[(x - 1)(ym + m_{\bmod L}) + (y - 1)]mw$ from (5.13) will, in both cases, give the number of steps elapsed before the solution to the first problem is completely out. The interval between solutions to problems on the same column is mw steps. Between problems on the same row, this interval is computed as $(ym + L)mw$ when $m_{\bmod L} = 0$, or $(ym + m_{\bmod L})mw$ when $m_{\bmod L} \neq 0$.

Because of the increases in number of strips and in their length, the capacity of buffers B_q and B_r should be expanded as previously indicated.

To see how much of an improvement over single dimension extensions this technique is capable of, let us again assume that we have a system of $L = 4$ subarrays, with each array of size $w = 32$. With this system, 10000 operations are to be performed on a number of matrices of order $n = 128$. Solving the problems one at a time without processing overlaps will take a total of 22,070,000 steps. Maximizing processing overlaps will reduce this number to 20,480,159. If single dimension extensions can be used, the problems can be solved in 10,241,183 steps. The improvement in throughput is

$$\frac{20,480,159}{10,241,183} = 2.0$$

However, if compatibilities between these problems are exploited such that the two-dimensional extension can be used with $x = 100$ and $y = 100$, the total throughput will be 5,223,071 steps. The improvement factor is thus

$$\frac{20,480,159}{5,223,071} = 3.92,$$

almost doubling the speedup figure achieved with single dimension extension. As was noted before, the improvement factor grows closer to four as x and y get larger.

Another advantage of the two-dimensional extension is that it further enhances the inherent programmability of Faddeev's algorithm. For example, should it be necessary to compute **U**, where

$$\mathbf{U} = (\mathbf{AE} + \mathbf{F})(\mathbf{E} + \mathbf{G})^{-1}(\mathbf{B} + \mathbf{D}) + \mathbf{AB}, \tag{5.12}$$

(5.11) can be rearrange to become

$$\begin{array}{c|c|c} \mathbf{I} & \mathbf{E} & \mathbf{B} \\ \hline -\mathbf{I} & \mathbf{G} & \mathbf{D} \\ \hline -\mathbf{A} & \mathbf{F} & 0 \end{array} \tag{5.13}$$

Solving (5.13), that is annulling **-I** and **-A** while extending the operations to the right will give

$$\begin{array}{c|c|c} \mathbf{I} & \mathbf{E} & \mathbf{B} \\ \hline 0 & \mathbf{E}+\mathbf{G} & \mathbf{B}+\mathbf{D} \\ \hline 0 & \mathbf{AE}+\mathbf{F} & \mathbf{AB} \end{array} \tag{5.14}$$

Observe that within the box of (5.14), the necessary components of (5.12) are already correctly positioned such that repeating the Faddeev's procedure on them will produce the final result

$$\begin{array}{c|c|c} \mathbf{I} & \mathbf{E} & \mathbf{B} \\ \hline 0 & (\mathbf{E}+\mathbf{G})^{(k)} & (\mathbf{B}+\mathbf{D})^{(k)} \\ \hline 0 & 0 & \mathbf{U} \end{array} \tag{5.15}$$

In short, to compute **U** from (5.13), one only needs to triangularize the augmented matrix formed from **I**, **E**, **-I** and **G**, then annul the augmented matrix formed from **-A** and **F** while extending both operations to the rightmost column of (5.13). Using the *L*-subarrays system, **U** is computed from the input data flow of (5.13) in 2*k* iterations. The first *k* iterations are needed to compute the matrices in the box of (5.14). This intermediate results is immediately fed back into the system for another *k* iterations, after which **U** is outputed.

CONCLUDING REMARKS

By now, it is clearly obvious that the symbiosis of Faddeev's algorithm and the new systolic array system described in Chapter IV has given rise to a very powerful and versatile tool. The algorithm itself provides a considerable generality of operation which should allow the system to have a large range of application in the scientific and industrial fields. In return, the system has brought massive parallelism to the multitude of matrix operations capable by the algorithm. Furthermore, the system's enormous potential for parallelism can now be fully exploited to yield very high throughput with the Faddeev's algorithm extensions described in Chapter V.

As compared to other designs from Chapter III, this system does not suffer any of their drawbacks while providing many practical advantages, some of which can be summarized as follow:

- Either in single or multiple arrays form, the system is totally independent of problem size and will solve sparse matrix problems efficiently without any reconfiguration.
- The system provides identical performance using a smaller number of cells or arrays. Indeed, given an equal number of arrays, its performance will be superior. When taken into account the fact that its design is

ideally suited for the extensions made to Faddeev's algorithm, its throughput potential far outdistances any other system previously considered.

- From a user point of view, operating the system is exceedingly simple: the input data flow is fed only to the top array and system controls consist of a few signals to each array top left cell.
- The design of the system is truly modular, with simple and regular interconnections between cells and between modules. Hence it is very amenable to expansion: adding extra blocks of shift registers will allow it to handle correspondingly larger problems, while increasing the number of arrays will yield higher throughput.
- Since all modules are square blocks $w \times w$ in size, it is topologically more economical and efficient in terms of PC board area.

In conclusion, the system's most important advantage is that while its design is simple enough for implementation to be an easy task, it is abundantly powerful and versatile to make that task worthwhile. Therefore, it is this author's opinion that the system should be built as soon as possible.

REFERENCES

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

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

3. Kai Hwang and Fayé A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, New York, 1984, pp. 768-774.

4. Charles L. Seitz and Juri Matisoo, "Engineering Limits on Computer Performance," Physics Today, Vol. 37, No. 5, May 1984, pp. 38-45.

5. C. A. Mead and L. A. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980, pp.263-292.

6. H. T. Kung, "Notes on VLSI Computation," in Parallel Processing Systems, ed. by David J. Evans, Cambridge University Press, Cambridge, MA, 1982, pp.339-356.

7. Ronald Collett, "CPU Architecture, Part I: Problems And Limitations of Von Neumann Computers," Digital Design, Vol. 14, No. 11, November 1984, pp. 90-95.

8. Wolfgang Händler, "Innovative Computer Architecture—How to Increase Parallelism but Not Complexity," in Parallel Processing Systems, ed. by David J. Evans, Cambridge University Press, Cambridge, MA, 1982, pp.23-32.

9. R. W. Hockney and C. R. Jesshope, Parallel Computers, Adam Hilger, Ltd., Bristol, 1981, pp. 1-51.

10. P. M. Dew, "VLSI Architectures for Problems in Numerical Computation," in Supercomputers and Parallel Computation, ed. by D. J. Paddon, Oxford University Press, New York, 1984, pp. 2-21.

11. S. Y. Kung, "VLSI Array Processors," IEEE ASSP Magazine, Vol. 2, No. 3, July 1985, pp. 4-22.

12. Leonard S. Haynes et al., "A Survey of Highly Parallel Computing," IEEE Computer Magazine, Vol. 15, No. 1, January 1982, pp. 9-24.

13. Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computer," IEEE Computer Magazine, Vol. 15, No. 1, January 1982, pp. 47-56.

14. Douglas G. Fairbairn, "VLSI: A New Frontier for Systems Designers," IEEE Computer Magazine, Vol. 15, No. 1, January 1982, pp. 87-96.

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

16. A. L. Fisher et al., "Design of the PSC: A Programmable Systolic Chip," in Proc. of the Third Caltech Conference on Very Large Scale Integration, ed. by R. Bryant, Computer Science Press, Rockville, MD, March 1983, pp. 287-302.

17. A. L. Fisher et al., "The Architecture of a Programmable Systolic Chip," Journal of VLSI and Computer Systems, Vol. 1, No. 2, Computer Science Press, Rockville, MD, 1984, pp. 153-169.

18. D. K. Faddeev and V. N. Faddeeva, Computational Methods of Linear Algebra, W. H. Freeman and Company, 1963, pp. 150-158.

19. W. W. Gentleman and H. T. Kung, "Matrix Triangularization by Systolic Arrays," Proc. SPIE—The International Society of Optical Engineering, vol. 298, 1981, pp. 19-26.

20. H. T. Kung, "Systolic Array for Orthogonal Triangularization," Proc. SPIE, San Diego, CA, 1981, pp. 19-26.

21. Richard L. Burden et al, Numerical Analysis, PWS Publishers, Boston, MA, 1981, pp. 289-294.

22. W. M. Gentleman, "Error Analysis of QR Decompositions by Givens Transformations," Linear Algebra and Its Application, American Elsevier Publishing Company, New York, 1975, pp. 189-197.

23. J. Greg Nash, "A Systolic/Cellular Computer Architecture for Linear Algebraic Operations," Proc. 1985 IEEE International Conference on Robotics and Automation, March 1985, pp. 779-784.

24. J. G. Nash and S. Hansen, "Modified Faddeev Algorithm for Matrix Manipulation," Proc. SPIE, Vol. 495, August 1984, pp. 39-46.

25. Henry Y. H. Chuang and Guo He, "A Versatile Systolic Array For Matrix Computations," The International Symposium on Computer Architecture, 1985, pp. 315-322.

APPENDIX A

EXAMPLES OF FADDEEV'S ALGORITHM

In the following, we will solve sample matrix problems using Faddeev's algorithm and its variants. The unmodified Faddeev's procedure, involving only ordinary Gaussian elimination, is illustrated with the first example. Its variant form using Gaussian elimination with neighbor pivoting is illustrated in the next two examples. Taken from chapter III, examples (3.1) and (3.2) are solved using the Faddeev's procedure combined with Givens rotations.

All calculations in the examples are carried out using nine decimal places precision; however, because this thesis' line formatting allows only a finite number of characters, results are shown rounded off to two decimal places.

Using Ordinary Gaussian Elimination

Suppose we want to compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$, where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are matrices of order $n = 3$ and

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 3 \\ -1 & 0 & 2 \\ 4 & -4 & 5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 1 & -3 \\ 1 & 7 & 9 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} -1 & 2 & 3 \\ 0 & 7 & -4 \\ 1 & -5 & 0 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 0 & 4 & -6 \\ -2 & 1 & 0 \\ 7 & 3 & 2 \end{bmatrix}$$

With Faddeev's algorithm, this problem can be expressed as

$$\left[\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{C} & \mathbf{D} \end{array}\right] = \left[\begin{array}{rrr|rrr} 2 & -1 & 3 & 1 & 2 & 4 \\ -1 & 0 & 2 & 3 & 1 & -3 \\ 4 & -4 & 5 & 1 & 7 & 9 \\ \hline 1 & -2 & -3 & 0 & 4 & -6 \\ 0 & -7 & 4 & -2 & 1 & 0 \\ -1 & 5 & 0 & 7 & 3 & 2 \end{array}\right] \qquad (A.1)$$

where, by means of matrix triangularization, all entries below the diagonal elements of **A** are zeroed out such that **A** is triangularized and **C** is annulled. After completion, the results should appear in the place of **D**.

Matrix triangularization procedures are often used, among other things, to solve linear systems. In solving a linear system, three operations are permitted on its rows:

1) Entries of row R_i can be multiplied by any non-zero constant λ and the resulting row used in place of R_i. This operation will be denoted $(\lambda R_i) \rightarrow (R_i)$
2) Entries of row R_j can be multiplied by any constant λ, added to row R_i, and the resulting row used in place of R_i. This operation will be denoted $(R_i + \lambda R_j) \rightarrow (R_i)$.
3) Rows R_i and R_j can be transposed in order. This operation will be denoted $(R_i) \leftrightarrow (R_j)$.

When used within Faddeev's algorithm, the third operation has a restriction which states that i and j cannot be larger than the order n of the matrices, i.e. transposing the order of the two said rows is not allowed if either one or both rows belong to the lower half of (A.1). Furthermore, although the entries in the affected rows are expected to change after any of these three operations, for ease of notation we will again denote the entry in the i^{th} row and the j^{th} column of matrix **X** (**X** here represents **A**, **B**, **C** or **D** of (A.1)) by x_{ij}. With this in mind, we can apply Gaussian elimination procedure to (A.1) by sequentially, for $i = 1, 2, \ldots, n-1$, perform the operation $(R_j - (a_{ji}/a_{ii})R_i) \rightarrow (R_j)$ on the upper half of (A.1) with $j = i+1, i+2, \ldots, n$, and the operation $(R_k - (-c_{k-n,i}/a_{ii})R_i) \rightarrow (R_k)$ on the lower half of (A.1) with $k = n+1, n+2, \ldots, 2n$, provided that $a_{ii} \neq 0$. When $a_{ii} = 0$, a search is made for the *first* non-zero element a_{ji} where $j = i+1, i+2, \ldots, n$ and the operation $(R_i) \leftrightarrow (R_j)$ is performed so that the procedure can continue.

Thus, by performing the operations $(R_2 + .5R_1) \rightarrow (R_2)$, $(R_3 - 2R_1) \rightarrow (R_3)$, $(R_4 - .5R_1) \rightarrow (R_4)$, and $(R_6 + .5R_1) \rightarrow (R_6)$ on (A.1), row R_1 is effectively used to zero out all entries below a_{11} to give:

$$\begin{array}{rrr|rrr}
2 & -1 & 3 & 1 & 2 & 4 \\
0 & -.5 & 3.5 & 3.5 & 2 & -1 \\
0 & -2 & -1 & -1 & 3 & 1 \\
\hline
0 & -1.5 & -4.5 & -.5 & 3 & -8 \\
0 & -7 & 4 & -2 & 1 & 0 \\
0 & 4.5 & 1.5 & 7.5 & 4 & 4
\end{array}$$

In this system, R_2 is used to eliminate entries below a_{22} by the operations $(R_3 - 4R_2) \rightarrow (R_3)$, $(R_4 - 3R_2) \rightarrow (R_4)$, $(R_5 - 14R_2) \rightarrow (R_5)$ and $(R_6 + 9R_2) \rightarrow (R_6)$. The resulting system is then

$$\begin{array}{rrr|rrr}
2 & -1 & 3 & 1 & 2 & 4 \\
0 & -.5 & 3.5 & 3.5 & 2 & -1 \\
0 & 0 & -15 & -15 & -5 & 5 \\
\hline
0 & 0 & -15 & -11 & -3 & -5 \\
0 & 0 & -45 & -51 & -27 & 14 \\
0 & 0 & 33 & 39 & 22 & -5
\end{array}$$

Finally, with the operations $(R_4 - R_3) \to (R_4)$, $(R_5 - 3R_3) \to (R_5)$ and $(R_6 - 2.2R_3) \to (R_6)$, we obtain the system

$$\begin{array}{ccc|ccc}
2 & -1 & 3 & 1 & 2 & 4 \\
0 & -.5 & 3.5 & 3.5 & 2 & -1 \\
0 & 0 & -15 & -15 & -5 & 5 \\
\hline
0 & 0 & 0 & 4 & 2 & -10 \\
0 & 0 & 0 & -6 & -12 & -1 \\
0 & 0 & 0 & 6 & 11 & 6
\end{array}$$

which shows the result $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ in its lower right hand quadrant.

Using Gaussian Elimination With Neighbor Pivoting

We have indicated earlier that obtaining a zero for a diagonal element a_{ii} during the Gaussian elimination procedure necessitated a row interchange of the form $(R_i) \leftrightarrow (R_j)$ where $i < j < n$ was the smallest integer with $a_{ji} \neq 0$. Actually, it is often desirable to perform row interchanges (or *pivoting*) involving the diagonal elements even when they are not zero. This is because when the calculations are performed using finite-digit arithmetic, as would be the case for calculators or computer-generated solutions, a diagonal element that is small compared to the entries below it in the same column can lead to substantial roundoff error.

Referred to as *neighbor pivoting*, the two adjacent rows R_i and R_j where $i < j < n$ are interchanged whenever

$|a_{ii}| < |a_{ji}|$, immediately before an operation of the form $(R_j - (a_{ji}/a_{ii})R_i) \rightarrow (R_j)$ can be performed on them. To illustrate this, let us consider the problem of computing $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ with matrices of order $n = 3$

$$\mathbf{A} = \begin{bmatrix} -1 & 5 & -3 \\ 3 & 4 & 1 \\ 6 & 7 & -2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -2 & -7 & 6 \\ 1 & 3 & 1 \\ 5 & 9 & 4 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 1 & -2 & 4 \\ 3 & 4 & -1 \\ -5 & 3 & 2 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 2 & 1 & -5 \\ 2 & 4 & 6 \\ -3 & 2 & 9 \end{bmatrix}$$

Like before, the problem is expressed as

$$\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{C} & \mathbf{D} \end{array} = \begin{array}{rrr|rrr} -1 & 5 & -3 & -2 & -7 & 6 \\ 3 & 4 & 1 & 1 & 3 & 1 \\ 6 & 7 & -2 & 5 & 9 & 4 \\ \hline -1 & 2 & -4 & 2 & 1 & -5 \\ -3 & -4 & 1 & 2 & 4 & 6 \\ 5 & -3 & -2 & -3 & 2 & 9 \end{array} \tag{A.2}$$

Since (A.2) shows that $|a_{11}| < |a_{21}|$, pivoting is therefore required between rows R_1 and R_2. Thus, performing the operation $(R_1) \leftrightarrow (R_2)$ gives us

$$\begin{array}{rrr|rrr} 3 & 4 & 1 & 1 & 3 & 1 \\ -1 & 5 & -3 & -2 & -7 & 6 \\ 6 & 7 & -2 & 5 & 9 & 4 \\ \hline -1 & 2 & -4 & 2 & 1 & -5 \\ -3 & -4 & 1 & 2 & 4 & 6 \\ 5 & -3 & -2 & -3 & 2 & 9 \end{array}$$

where, after the operation $(R_2 + .33R_1) \rightarrow (R_2)$, we have

$$\begin{array}{rrr|rrr}
3 & 4 & 1 & 1 & 3 & 1 \\
0 & 6.33 & -2.67 & -1.67 & -6 & 6.33 \\
6 & 7 & -2 & 5 & 9 & 4 \\
\hline
-1 & 2 & -4 & 2 & 1 & -5 \\
-3 & -4 & 1 & 2 & 4 & 6 \\
5 & -3 & -2 & -3 & 2 & 9
\end{array}$$

Note that how neighbor pivoting has just been carried out by the two previous steps. Once again, the above system shows that pivoting is required between R_1 and R_3 since $|a_{11}| < |a_{31}|$. Therefore the operation $(R_1) \leftrightarrow (R_3)$ will subsequently give

$$\begin{array}{rrr|rrr}
6 & 7 & -2 & 5 & 9 & 4 \\
0 & 6.33 & -2.67 & -1.67 & -6 & 6.33 \\
3 & 4 & 1 & 1 & 3 & 1 \\
\hline
-1 & 2 & -4 & 2 & 1 & -5 \\
-3 & -4 & 1 & 2 & 4 & 6 \\
5 & -3 & -2 & -3 & 2 & 9
\end{array}$$

which, after the operations $(R_3 - .5R_1) \rightarrow (R_3)$, $(R_4 + .17R_1) \rightarrow (R_4)$, $(R_5 + .5R_1) \rightarrow (R_5)$ and $(R_6 - .83R_1) \rightarrow (R_6)$, becomes

$$\begin{array}{rrr|rrr}
6 & 7 & -2 & 5 & 9 & 4 \\
0 & 6.33 & -2.67 & -1.67 & -6 & 6.33 \\
0 & .5 & 2 & -1.5 & -1.5 & -1 \\
\hline
0 & 3.17 & -4.33 & 2.83 & 2.5 & -4.33 \\
0 & -.5 & 0 & 4.5 & 8.5 & 8 \\
0 & -8.83 & -.33 & -7.17 & -5.5 & 5.67
\end{array}$$

The procedure is carried out further with the elimination of entries below a_{22} by applying the operations $(R_3 - .08R_2) \rightarrow (R_3)$, $(R_4 - .5R_2) \rightarrow (R_4)$, $(R_5 + .08R_2) \rightarrow (R_5)$ and $(R_6 + 1.39R_2) \rightarrow (R_6)$. We thus have

$$\begin{array}{rrr|rrr}
6 & 7 & -2 & 5 & 9 & 4 \\
0 & 6.33 & -2.67 & -1.67 & -6 & 6.33 \\
0 & 0 & 2.21 & -1.37 & -1.03 & -1.5 \\
\hline
0 & 0 & -3 & 3.67 & 5.5 & -7.5 \\
0 & 0 & -.21 & 4.37 & 8.03 & 8.5 \\
0 & 0 & -4.05 & -9.49 & -13.87 & 14.5
\end{array}$$

After the elimination of the entries below a_{33} with the operations $(R_4 + 1.36R_3) \rightarrow (R_4)$, $(R_5 + .09R_3) \rightarrow (R_5)$ and $(R_6 + 1.83R_3) \rightarrow (R_6)$, the solution to $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$ appears in the lower right hand quadrant of

$$\left[\begin{array}{ccc|ccc} 6 & 7 & -2 & 5 & 9 & 4 \\ 0 & 6.33 & -2.67 & -1.67 & -6 & 6.33 \\ 0 & 0 & 2.21 & -1.37 & -1.03 & -1.5 \\ \hline 0 & 0 & 0 & 1.81 & 4.11 & -9.54 \\ 0 & 0 & 0 & 4.24 & 7.93 & 8.36 \\ 0 & 0 & 0 & -12 & -15.75 & 11.75 \end{array}\right]$$

The following is another example of Faddeev's algorithm with neighbor pivoting. Given matrices **A**, **B**, **C** and **D** of order $n = 4$, with

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 3 & 0 \\ 4 & -2 & 7 & 0 \\ -3 & -4 & 1 & 5 \\ 6 & -6 & 8 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -8 & 3 & 0 & 3 \\ -20 & 5 & 1 & 6 \\ -2 & -9 & 7 & 8 \\ 4 & 7 & 4 & 2 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 1 & -1 & 2 & -1 \\ 2 & -2 & 3 & -3 \\ 1 & 1 & 1 & 0 \\ 1 & -1 & 4 & 3 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 1 & 3 & -5 & 7 \\ 0 & -4 & 1 & 7 \\ 2 & 1 & 3 & 0 \\ 1 & -3 & -1 & 9 \end{bmatrix},$$

we want to compute $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$. Formulating the problem as follow

$$\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{C} & \mathbf{D} \end{array} = \left.\begin{array}{rrrr|rrrr} 2 & -1 & 3 & 0 & -8 & 3 & 0 & 3 \\ 4 & -2 & 7 & 0 & -20 & 5 & 1 & 6 \\ -3 & -4 & 1 & 5 & -2 & -9 & 7 & 8 \\ 6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\ \hline -1 & 1 & -2 & 1 & 1 & 3 & -5 & 7 \\ -2 & 2 & -3 & 3 & 0 & -4 & 1 & 7 \\ -1 & -1 & -1 & 0 & 2 & 1 & 3 & 0 \\ -1 & 1 & -4 & -3 & 1 & -3 & -1 & 9 \end{array}\right. \qquad \text{(A.3)}$$

reveals that, because $|a_{11}| < |a_{21}|$, pivoting of rows R_1 and R_2 is necessary. Thus, the operation $(R_1) \leftrightarrow (R_2)$ produces the system

$$\begin{array}{rrrr|rrrr} 4 & -2 & 7 & 0 & -20 & 5 & 1 & 6 \\ 2 & -1 & 3 & 0 & -8 & 3 & 0 & 3 \\ -3 & -4 & 1 & 5 & -2 & -9 & 7 & 8 \\ 6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\ \hline -1 & 1 & -2 & 1 & 1 & 3 & -5 & 7 \\ -2 & 2 & -3 & 3 & 0 & -4 & 1 & 7 \\ -1 & -1 & -1 & 0 & 2 & 1 & 3 & 0 \\ -1 & 1 & -4 & -3 & 1 & -3 & -1 & 9 \end{array}$$

which, after we perform the operations $(R_2 - .5R_1) \rightarrow (R_2)$ and $(R_3 + .75R_1) \rightarrow (R_3)$, becomes

$$\begin{array}{rrrr|rrrr}
4 & -2 & 7 & 0 & -20 & 5 & 1 & 6 \\
0 & 0 & -.5 & 0 & 2 & .5 & -.5 & 0 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
\hline
-1 & 1 & -2 & 1 & 1 & 3 & -5 & 7 \\
-2 & 2 & -3 & 3 & 0 & -4 & 1 & 7 \\
-1 & -1 & -1 & 0 & 2 & 1 & 3 & 0 \\
-1 & 1 & -4 & -3 & 1 & -3 & -1 & 9
\end{array}$$

Before we can proceed any further in eliminating entries in the first column, because $|a_{11}| < |a_{41}|$, we have to perform the operation $(R_1) \leftrightarrow (R_4)$:

$$\begin{array}{rrrr|rrrr}
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
0 & 0 & -.5 & 0 & 2 & .5 & -.5 & 0 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
4 & -2 & 7 & 0 & -20 & 5 & 1 & 6 \\
\hline
-1 & 1 & -2 & 1 & 1 & 3 & -5 & 7 \\
-2 & 2 & -3 & 3 & 0 & -4 & 1 & 7 \\
-1 & -1 & -1 & 0 & 2 & 1 & 3 & 0 \\
-1 & 1 & -4 & -3 & 1 & -3 & -1 & 9
\end{array}$$

Now, all remaining entries in the first column can be eliminated with $(R_4 - .67R_1) \rightarrow (R_4)$, $(R_5 + .17R_1) \rightarrow (R_5)$, $(R_6 + .33R_1) \rightarrow (R_6)$, $(R_7 + .17R_1) \rightarrow (R_7)$ and $(R_8 + .17R_1) \rightarrow (R_8)$, to give

$$\begin{array}{rrrr|rrrr}
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
0 & 0 & -.5 & 0 & 2 & .5 & -.5 & 0 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
0 & 2 & 1.67 & 0 & -22.67 & .33 & -1.67 & 4.67 \\
\hline
0 & 0 & -.67 & 1 & 1.67 & 4.17 & -4.33 & 7.33 \\
0 & 0 & -.33 & 3 & 1.33 & -1.67 & 2.33 & 7.67 \\
0 & -2 & .33 & 0 & 2.67 & 2.17 & 3.67 & .33 \\
0 & 0 & -2.67 & -3 & 1.67 & -1.83 & -.33 & 9.33
\end{array}$$

Prior to zero out entries in the second column, because $a_{22} = 0$, the operation $(R_2) \leftrightarrow (R_3)$ is used to obtain

$$\begin{array}{rrrr|rrrr}
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
0 & 0 & -.5 & 0 & 2 & .5 & -.5 & 0 \\
0 & 2 & 1.67 & 0 & -22.67 & .33 & -1.67 & 4.67 \\
\hline
0 & 0 & -.67 & 1 & 1.67 & 4.17 & -4.33 & 7.33 \\
0 & 0 & -.33 & 3 & 1.33 & -1.67 & 2.33 & 7.67 \\
0 & -2 & .33 & 0 & 2.67 & 2.17 & 3.67 & .33 \\
0 & 0 & -2.67 & -3 & 1.67 & -1.83 & -.33 & 9.33
\end{array}$$

Applying $(R_4 + .36R_2) \rightarrow (R_4)$ and $(R_7 - .36R_2) \rightarrow (R_7)$ to the above system, we are left with

$$\begin{array}{rrrr|rrrr}
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
0 & 0 & -.5 & 0 & 2 & .5 & -.5 & 0 \\
0 & 0 & 3.94 & 1.82 & -28.85 & -1.58 & 1.15 & 9.21 \\
\hline
0 & 0 & -.67 & 1 & 1.67 & 4.17 & -4.33 & 7.33 \\
0 & 0 & -.33 & 3 & 1.33 & -1.67 & 2.33 & 7.67 \\
0 & 0 & -1.94 & 1.82 & 8.85 & 4.08 & .85 & -4.21 \\
0 & 0 & -2.67 & -3 & 1.67 & -1.83 & -.33 & 9.33
\end{array}$$

which requires pivoting of rows R_3 and R_4. Therefore, after $(R_3) \leftrightarrow (R_4)$, we have

$$\begin{array}{rrrr|rrrr}
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
0 & 0 & 3.94 & 1.82 & -28.85 & -1.58 & 1.15 & 9.21 \\
0 & 0 & -.5 & 0 & 2 & .5 & -.5 & 0 \\
\hline
0 & 0 & -.67 & 1 & 1.67 & 4.17 & -4.33 & 7.33 \\
0 & 0 & -.33 & 3 & 1.33 & -1.67 & 2.33 & 7.67 \\
0 & 0 & -1.94 & 1.82 & 8.85 & 4.08 & .85 & -4.21 \\
0 & 0 & -2.67 & -3 & 1.67 & -1.83 & -.33 & 9.33
\end{array}$$

where we can proceed to eliminate all entries below a_{33} with the operations $(R_4 + .13R_3) \to (R_4)$, $(R_5 + .17R_3) \to (R_5)$, $(R_6 + .08R_3) \to (R_6)$, $(R_7 + .49R_3) \to (R_7)$ and $(R_8 + .68R_3) \to (R_8)$. The resulting system will be

$$\begin{array}{rrrr|rrrr}
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
0 & 0 & 3.94 & 1.82 & -28.85 & -1.58 & 1.15 & 9.21 \\
0 & 0 & 0 & .23 & -1.66 & .3 & -.35 & 1.17 \\
\hline
0 & 0 & 0 & 1.31 & -3.22 & 3.9 & -4.14 & 8.89 \\
0 & 0 & 0 & 3.15 & -1.11 & -1.8 & 2.45 & 8.45 \\
0 & 0 & 0 & -.92 & -5.35 & 3.3 & 1.42 & .32 \\
0 & 0 & 0 & -1.77 & -17.86 & -2.9 & .45 & 15.57
\end{array}$$

Finally, annulling the lower left hand quadrant completely with the operations $(R_5 - 5.67R_4) \to (R_5)$, $(R_6 - 13.67R_4) \to (R_6)$, $(R_7 + 4R_4) \to (R_7)$ and $(R_8 + 7.67R_4) \to (R_8)$ will give us the solution in the lower right hand quadrant of

$$\begin{array}{rrrr|rrrr}
6 & -6 & 8 & 0 & 4 & 7 & 4 & 2 \\
0 & -5.5 & 6.25 & 5 & -17 & -5.25 & 7.75 & 12.5 \\
0 & 0 & 3.94 & 1.82 & -28.85 & -1.58 & 1.15 & 9.21 \\
0 & 0 & 0 & .23 & -1.66 & .3 & -.35 & 1.17 \\
\hline
0 & 0 & 0 & 0 & 6.2 & 2.2 & -2.13 & 2.27 \\
0 & 0 & 0 & 0 & 21.6 & -5.9 & 7.27 & -7.53 \\
0 & 0 & 0 & 0 & -12 & 4.5 & 0 & 5 \\
0 & 0 & 0 & 0 & -30.6 & -.6 & -2.27 & 24.53
\end{array}$$

Using Givens Rotations

A Givens transformation rotating the two row vectors R_i and R_j

$$\begin{array}{ccccccccccc} 0 & \cdots & 0 & a_{ii} & a_{i,i+1} & \cdots & a_{ik} & \cdots & a_{in} \\ 0 & \cdots & 0 & a_{ji} & a_{j,i+1} & \cdots & a_{jk} & \cdots & a_{jn} \end{array}$$

of a given matrix **A** of order n replaces them with two new vectors

$$\begin{array}{ccccccccccc} 0 & \cdots & 0 & a'_{ii} & a'_{i,i+1} & \cdots & a'_{ik} & \cdots & a'_{in} \\ 0 & \cdots & 0 & 0 & a'_{j,i+1} & \cdots & a'_{jk} & \cdots & a'_{jn} \end{array}$$

such that, with $k = i+1, i+2, \ldots, n$, their entries are

$$\begin{aligned} a'_{ii} &= \alpha_{ij} \\ a'_{ik} &= \cos\alpha_{ij} a_{ik} + \sin\alpha_{ij} a_{jk} \\ a'_{jk} &= -\sin\alpha_{ij} a_{ik} + \cos\alpha_{ij} a_{jk} \end{aligned} \qquad \text{(A.E.1)}$$

where

$$\begin{aligned} \alpha_{ij} &= \sqrt{a_{ii}^2 + a_{ji}^2} \\ \cos\alpha_{ij} &= \frac{a_{ii}}{\alpha_{ij}} \\ \sin\alpha_{ij} &= \frac{a_{ji}}{\alpha_{ij}} \\ \cos^2\alpha_{ij} + \sin^2\alpha_{ij} &= 1. \end{aligned}$$

The transformation obviously leaves unchanged zeroes appearing in corresponding entries of both vectors. Thus a matrix of order n can be triangularized by applying a succession of Givens rotations to its rows R_i and R_{i+1}, R_i

and $R_{i+2}, \ldots, R_i$ and R_{i+n} for $i = 1, 2, \ldots, n-1$ such that zeroes are introduced into every columns below the diagonal elements.

When combined with Faddeev's algorithm, Givens rotations are used on the rows above the horizontal line to triangularize **A** and ordinary Gaussian elimination is used on rows below the horizontal line to annul **C**. The procedure involved can be illustrated much easier with an example.

Let us find the solutions of the linear system (3.1) of chapter III. This system has three unknowns, x_1, x_2 and x_3, and its equations are represented here in matrix form as

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 7 \\ 2 & 1 & 3 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 5 \\ 9 \\ 7 \end{bmatrix}$$

The solutions' column vector **X** can then be expressed as $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$ or, by expanding it to become $\mathbf{X} = \mathbf{I}\mathbf{A}^{-1}\mathbf{B} + \vec{0}$ where **I** is the identity matrix and $\vec{0}$ is a zero vector

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \vec{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

allows us to formulate the problem as

$$\begin{array}{c|c} A & B \\ \hline -I & 0 \end{array} = \begin{array}{ccc|c} 1 & 2 & 3 & 5 \\ 0 & 4 & 7 & 9 \\ 2 & 1 & 3 & 7 \\ \hline -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array} \tag{A.4}$$

Since $a_{21} = 0$ in (A.4), we can skip row R_2 and, by directly rotating rows R_1 and R_3 using the equations of (A.E.1) with

$$\alpha_{1,3} = \sqrt{a^2_{1,1} + a^2_{3,1}} = \sqrt{1 + 4} = 2.24$$

$$\cos\alpha_{1,3} = \frac{a_{1,1}}{\alpha_{1,3}} = \frac{1}{2.24} = .45$$

$$\sin\alpha_{1,3} = \frac{a_{3,1}}{\alpha_{1,3}} = \frac{2}{2.24} = .89,$$

subsequently get the following system

$$\begin{array}{ccc|c} 2.24 & 1.79 & 4.02 & 8.5 \\ 0 & 4 & 7 & 9 \\ 0 & -1.34 & -1.34 & -1.34 \\ \hline -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array}$$

Gaussian elimination is now used to continue the procedure below the horizontal line. Performing the operation $(R_4 + .45R_1) \to (R_4)$, we have

$$\left[\begin{array}{ccc|c} 2.24 & 1.79 & 4.02 & 8.5 \\ 0 & 4 & 7 & 9 \\ 0 & -1.34 & -1.34 & -1.34 \\ \hline 0 & .8 & 1.8 & 3.8 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array}\right.$$

Once again, we rotate rows R_2 and R_3 with

$$\alpha_{2,3} = \sqrt{a_{2,2}^2 + a_{3,2}^2} = \sqrt{16 + 1.8} = 4.22$$

$$\cos\alpha_{2,3} = \frac{a_{2,2}}{\alpha_{2,3}} = \frac{4}{4.22} = .95$$

$$\sin\alpha_{2,3} = \frac{a_{3,2}}{\alpha_{2,3}} = \frac{-1.34}{4.22} = -.32$$

to obtain

$$\left[\begin{array}{ccc|c} 2.24 & 1.79 & 4.02 & 8.5 \\ 0 & 4.22 & 7.06 & 8.96 \\ 0 & 0 & 0.95 & 1.59 \\ \hline 0 & .8 & 1.8 & 3.8 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array}\right.$$

which we can further modify by applying the operations $(R_4 - .19R_2) \rightarrow (R_4)$ and $(R_5 + .24R_2) \rightarrow (R_5)$, giving us

$$\left[\begin{array}{ccc|c} 2.24 & 1.79 & 4.02 & 8.5 \\ 0 & 4.22 & 7.06 & 8.96 \\ 0 & 0 & 0.95 & 1.59 \\ \hline 0 & 0 & .46 & 2.1 \\ 0 & 0 & 1.67 & 2.12 \\ 0 & 0 & -1 & 0 \end{array}\right.$$

Since **A** is now fully triangularized, performing the operations $(R_4 - .48R_3) \rightarrow (R_4)$, $(R_5 - 1.75R_3) \rightarrow (R_5)$ and $(R_6 + 1.05R_3) \rightarrow (R_6)$ to completely annul the lower left hand quadrant of the above system yields $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$ in the lower right hand quadrant of

$$\left[\begin{array}{ccc|c} 2.24 & 1.79 & 4.02 & 8.5 \\ 0 & 4.22 & 7.06 & 8.96 \\ 0 & 0 & 0.95 & 1.59 \\ \hline 0 & 0 & 0 & 1.33 \\ 0 & 0 & 0 & -0.67 \\ 0 & 0 & 0 & 1.67 \end{array}\right.$$

For the purpose of comparison, we will also present here the solutions to example (3.2) of chapter III. Later on in appendix B, this example will be used for the graphics simulation of Nash's array to show that it produces erroneous results as mentioned in chapter III.

Example (3.2) gives us a linear system which is expressed in matrix form as

$$\mathbf{A} = \begin{bmatrix} 0 & 2 & 3 \\ 0 & 4 & 7 \\ 2 & 1 & 3 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 5 \\ 9 \\ 7 \end{bmatrix}.$$

Solving this linear system with Faddeev's algorithm requires us to formulate it as

$$\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{I} & 0 \end{array} = \begin{array}{ccc|c} 0 & 2 & 3 & 5 \\ 0 & 4 & 7 & 9 \\ 2 & 1 & 3 & 7 \\ \hline -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array} \tag{A.5}$$

Because $a_{1,1} = 0$ and $a_{2,1} = 0$, we can make things a lot easier by interchanging rows R_1 and R_3 of (A.5) with the operation $(R_1) \leftrightarrow (R_3)$, to give

$$\begin{array}{ccc|c} 2 & 1 & 3 & 7 \\ 0 & 4 & 7 & 9 \\ 0 & 2 & 3 & 5 \\ \hline -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array}$$

Performing the operation $(R_4 + .5R_1) \rightarrow (R_4)$ reduces all entries below $a_{1,1}$ to zeroes, and the above system becomes

$$\left[\begin{array}{ccc|c} 2 & 1 & 3 & 7 \\ 0 & 4 & 7 & 9 \\ 0 & 2 & 3 & 5 \\ \hline 0 & .5 & 1.5 & 3.5 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array}\right.$$

Rotating rows R_2 and R_3 with

$$\alpha_{2,3} = \sqrt{a_{2,2}^2 + a_{3,2}^2} = \sqrt{16 + 4} = 4.47$$

$$cos\alpha_{2,3} = \frac{a_{2,2}}{\alpha_{2,3}} = \frac{4}{4.47} = .89$$

$$sin\alpha_{2,3} = \frac{a_{3,2}}{\alpha_{2,3}} = \frac{2}{4.47} = .45$$

will completely triangularize **A** to give

$$\left[\begin{array}{ccc|c} 2 & 1 & 3 & 7 \\ 0 & 4.47 & 7.6 & 10.29 \\ 0 & 0 & -.45 & .45 \\ \hline 0 & .5 & 1.5 & 3.5 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{array}\right.$$

in which all entries in the second column of the lower left hand quadrant can be eliminated with the operations

$(R_4 - .11R_2) \rightarrow (R_4)$ and $(R_5 + .22R_2) \rightarrow (R_5)$. This produces the system

$$\begin{array}{rrr|r}
2 & 1 & 3 & 7 \\
0 & 4.47 & 7.6 & 10.29 \\
0 & 0 & -.45 & .45 \\
\hline
0 & 0 & .65 & 2.35 \\
0 & 0 & 1.7 & 2.3 \\
0 & 0 & -1 & 0
\end{array}$$

Finally, the procedure is completed with the operations $(R_4 + 1.45R_3) \rightarrow (R_4)$, $(R_5 + 3.8R_3) \rightarrow (R_5)$ and $(R_6 - 2.24R_3) \rightarrow (R_6)$, to yield

$$\begin{array}{rrr|r}
2 & 1 & 3 & 7 \\
0 & 4.47 & 7.6 & 10.29 \\
0 & 0 & -.45 & .45 \\
\hline
0 & 0 & 0 & 3 \\
0 & 0 & 0 & 4 \\
0 & 0 & 0 & -1
\end{array}$$

which shows the solutions to the linear system in its lower right hand quadrant.

APPENDIX B

REAL TIME GRAPHICAL SIMULATION OF SYSTOLIC ARRAYS

Simulation techniques play an important role in the verification of a design's correctness of operation and debugging. Because serial computers are by nature sequential machines, their software simulators are often little more than conventional language interpreters.

For systolic arrays, this is simply inadequate. To verify whether a given algorithm is correctly mapped into a corresponding array architecture, a system designer must be able to observe, *at all times*, the movement of *every* piece of data as they traverse through the array, as well as the results from operations performed on each of them by *any* of the cells. Furthermore, for debugging purposes, he must be able to look into the registers of every cell *at any one time*, and see the values of all control signals present in that cell. In short, he must have the most detailed view of the entire system, which may consists of many arrays and many cells, at all times.

To meet the above requirements, a new breed of simulator—a systolic arrays simulator—was developed and built to aid a hardware or software designer in the task of designing and debugging systolic systems. For reasons which will become clear later, it was deemed essential that this simulator should be graphics based, hence its name Systolic Arrays Graphical Simulator, or SAGS in short.

From the very beginning, SAGS was designed to simulate systolic systems of any configurations. These configurations are specified to SAGS by way of script files. A script file contains vital informations about a system such as its number of arrays, their types and sizes, the way they are linked together and the microprograms each cell will use. A script file also specifies when and where input data and control signals should be fed into—and output data taken from—a system. SAGS allows for systems with multiple input, control and output data streams. Each input or control stream is stored into ASCII files prior to being accessed by SAGS. Similarly, outputs of SAGS are written into ASCII files.

During run time simulation, SAGS executes all steps of a problem one after another without pause, showing results of each step on the screen. This is called *multi-step* mode of execution; it can be stopped and restarted at any time. Alternatively, SAGS can *single-step* through the problem, allowing a more detailed inspection of the results. Switching between these two modes can be accomplished easily at any time.

Visually, SAGS allows all arrays of a system to be seen on a monitor screen, as long as each array has a reasonable number of cells. Because the real estate of a monitor screen is limited, arrays can be overlapped such that one in the background can be brought into the foreground for observation at any time. In addition, individual arrays can be interactively positioned anywhere on the screen to closely match the system schematic. SAGS allows an array to have two different views: a *real* view, with the array and its cells appearing smaller and therefore containing less information, and a *full* view, where the cells show all their registers content. The view of an array can be specified in the script file, or changed during run time. All visual changes made to a system configuration during run time can be recorded back to the script file for reuse. A status bar on top of the screen displays additional informations such as the current step number, the total execution time and the array being selected.

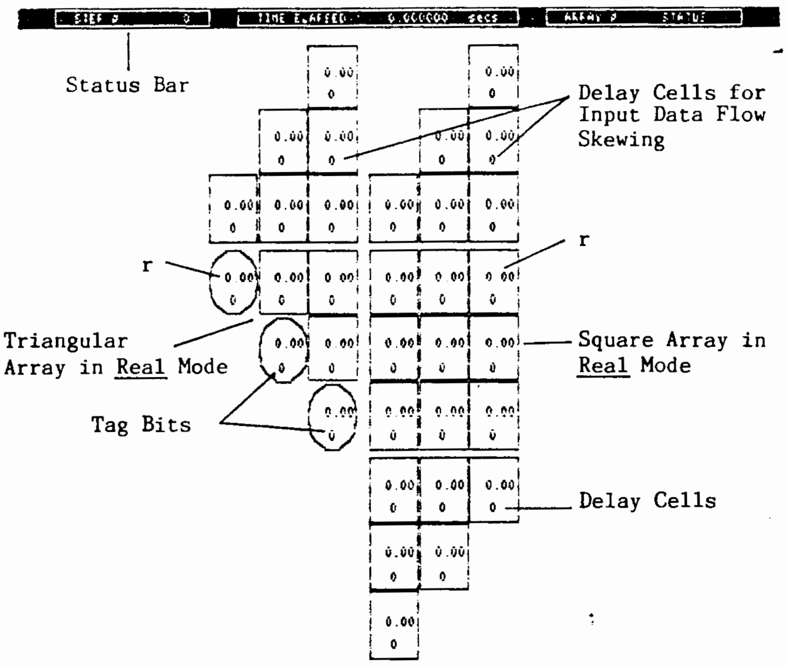
In this author's experience, SAGS has been quite useful in verifying and debugging the designs presented in this thesis. Indeed, it is while using SAGS to simulate Nash's implementation of Faddeev's algorithm that the bug in its boundary cell microprogram was discovered and identified. For the reader's convenience, SAGS source code is listed in Appendix C.

In the following, three series of snapshots illustrate the simulations of three different systolic designs. Each snapshot is a screen output of SAGS for one execution step. All problems used in these simulations are examples taken from Appendix A.

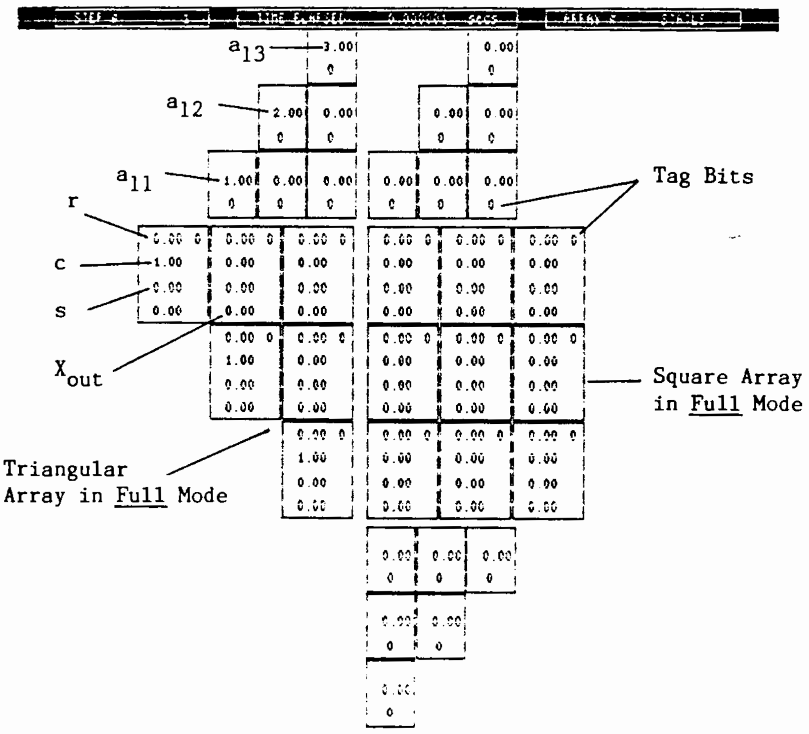
The first series of snapshots B.1 shows the simulation of Nash's system (from Figure 5) as it solves example (A.4). It can be seen that this implementation of Faddeev's algorithm produces erroneous results.

The second series of snapshots B.2 shows the simulation of Chuang and He's system (from Figure 8) using example (A.2).

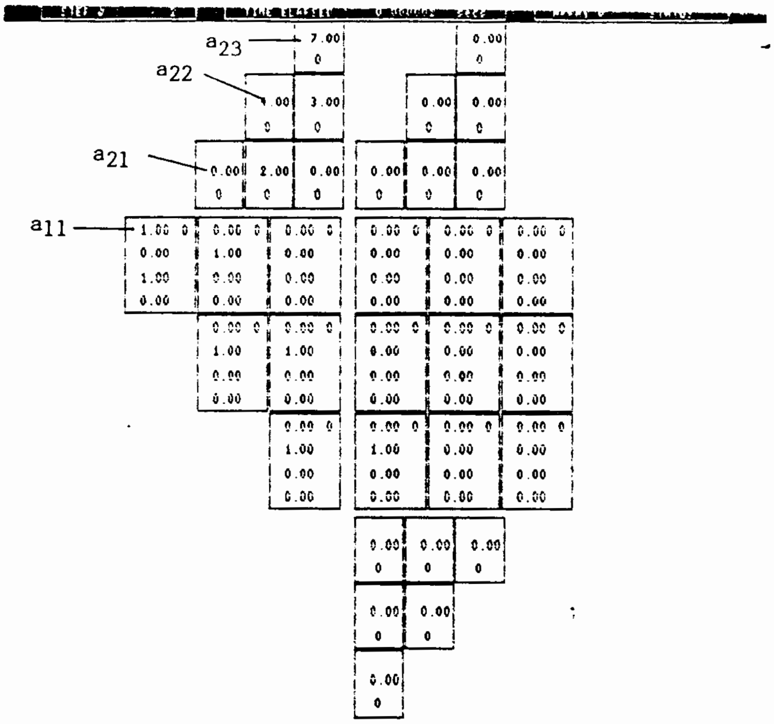
In the last series B.3, the *L*-tuple arrays system of Figure 30 is simulated, with $L = 2$. This system is shown here solving example (A.3).


Status Bar
Delay Cells for Input Data Flow Skewing
r
r
Triangular Array in Real Mode
Square Array in Real Mode
Tag Bits
Delay Cells

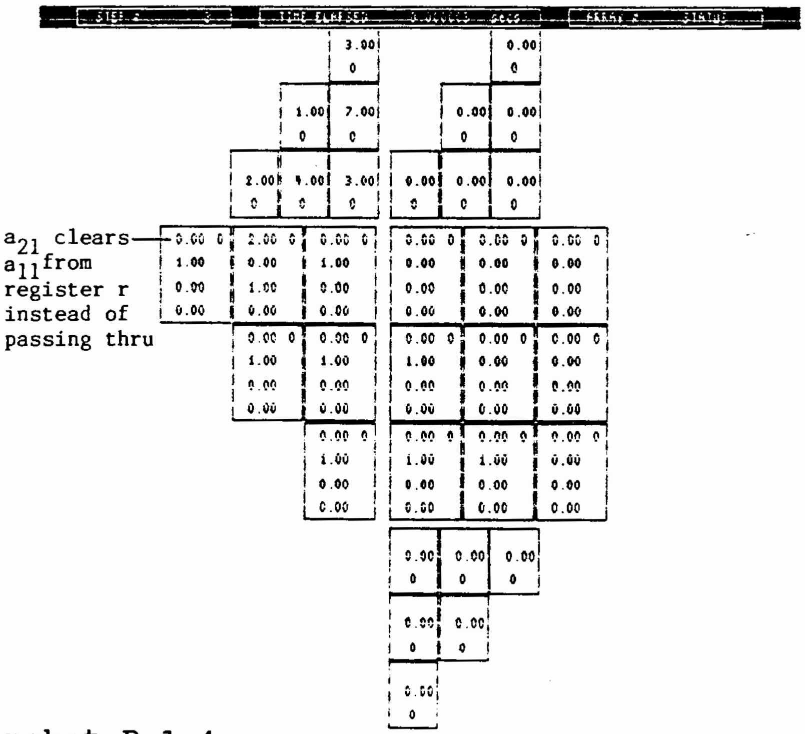
Snapshot B.1.1. Simulation of Nash's systolic array solving example (A.4).



Snapshot B.1.2

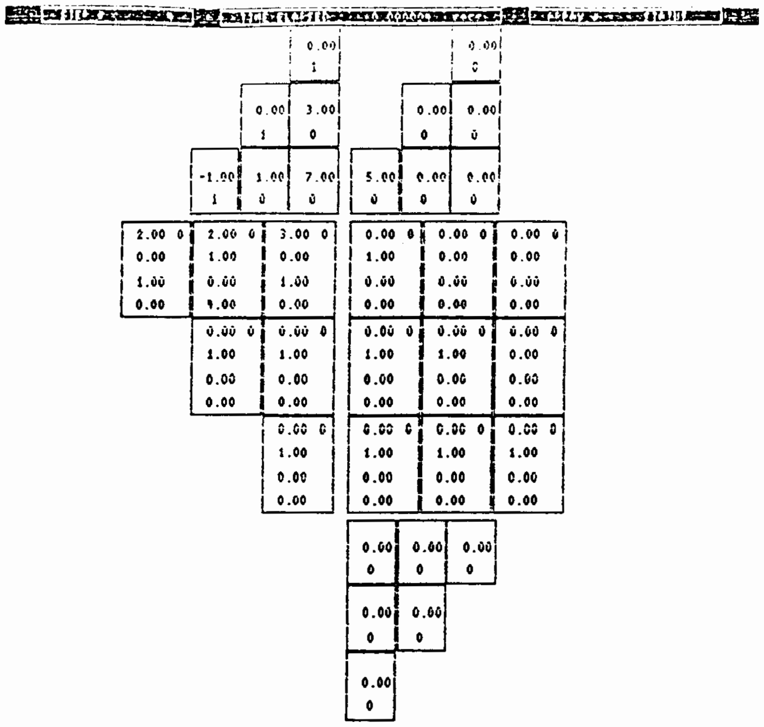


Snapshot B.1.3

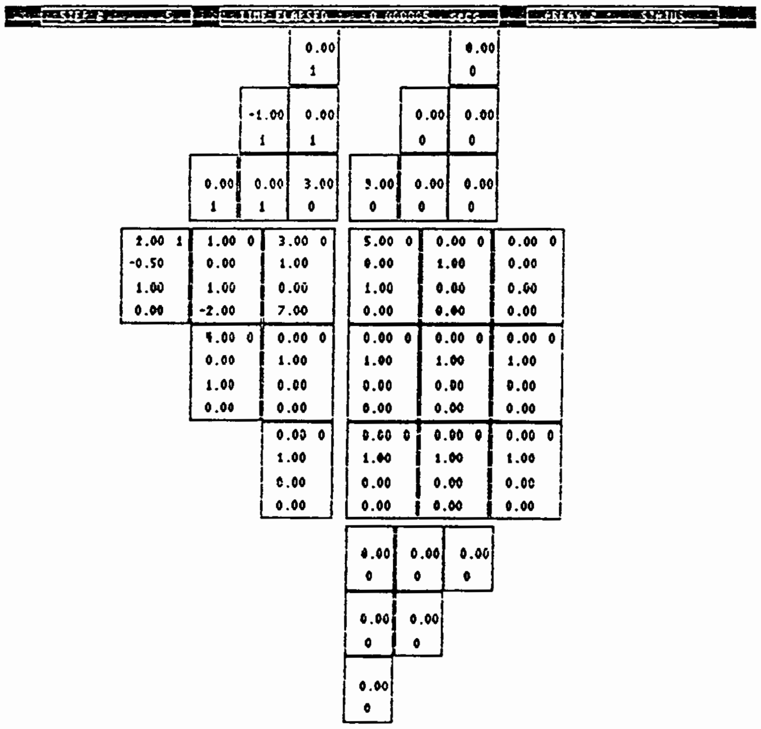


a_{21} clears a_{11} from register r instead of passing thru

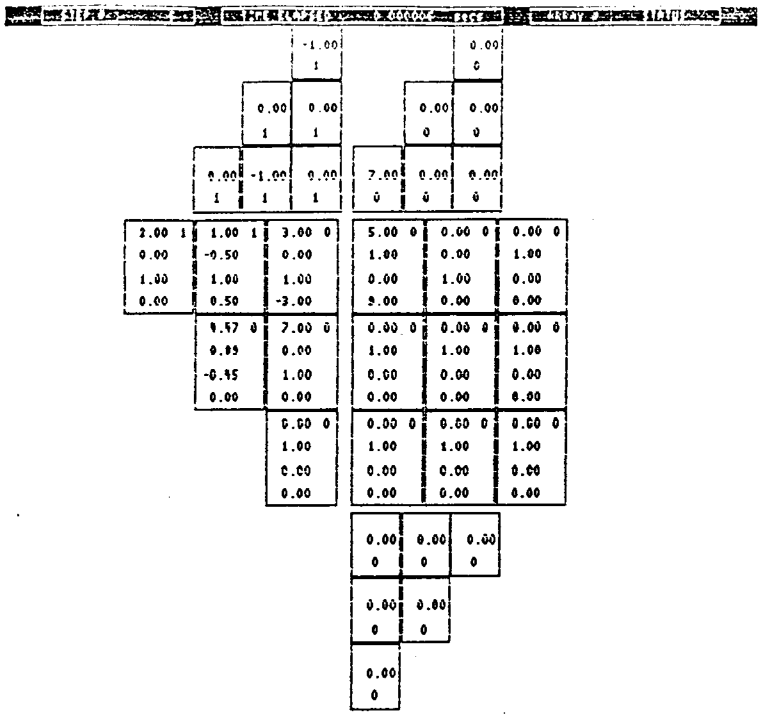
Snapshot B.1.4



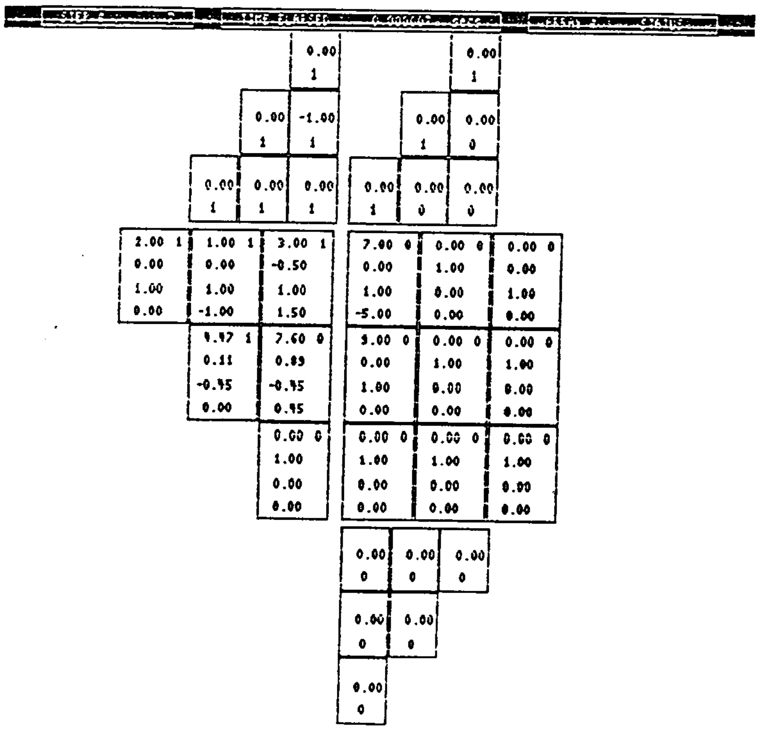
<u>Snapshot B.1.5</u>



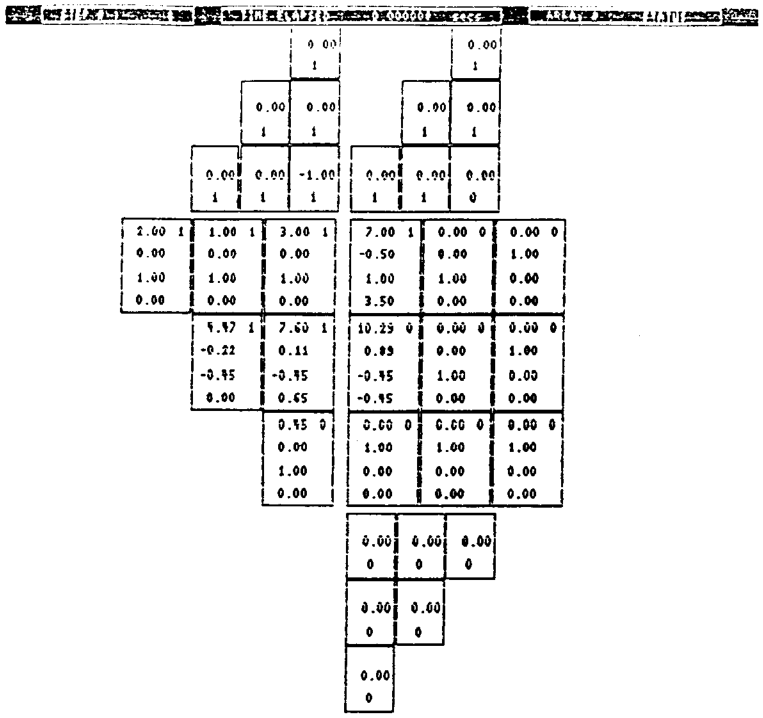
<u>Snapshot B.1.6</u>



Snapshot B.1.7

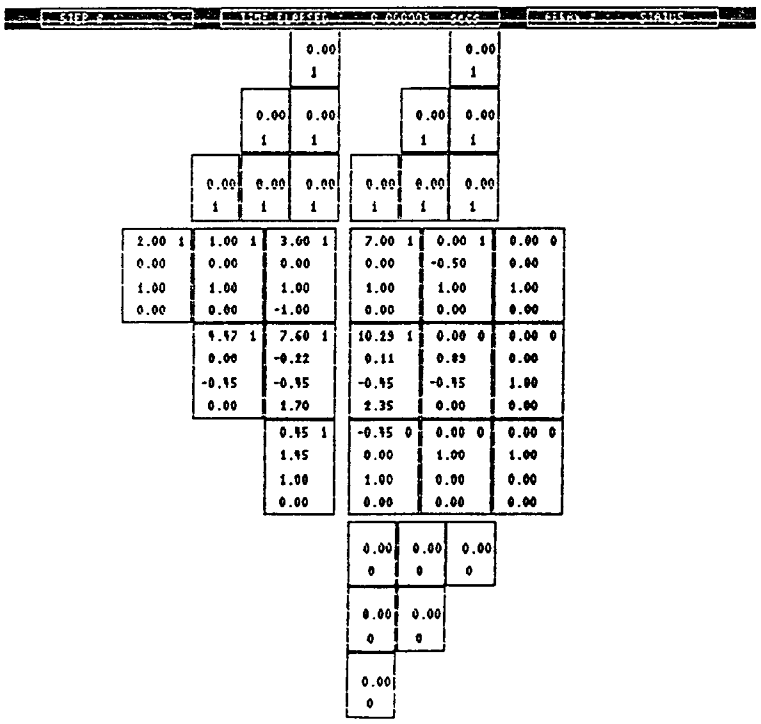


Snapshot B.1.8



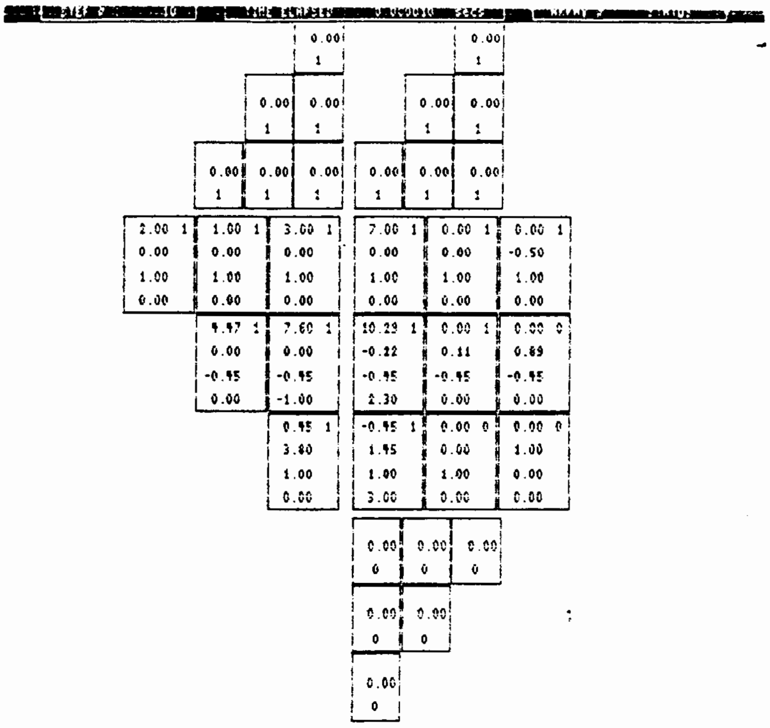
```
                                  0.00                0.00
                                  1                   1

                          0.00    0.00        0.00    0.00
                          1       1           1       1

                  0.00    0.00   -1.00   0.00 0.00    0.00
                  1       1       1      1    1       0

          2.00 1  1.00 1  3.00 1    7.00 1   0.00 0   0.00 0
          0.00    0.00    0.00     -0.50     0.00     1.00
          1.00    1.00    1.00      1.00     1.00     0.00
          0.00    0.00    0.00      3.50     0.00     0.00

                  4.47 1  7.60 1   10.25 0   0.00 0   0.00 0
                 -0.22    0.11      0.89     0.00     1.00
                 -0.45   -0.45     -0.45     1.00     0.00
                  0.00    0.65     -0.45     0.00     0.00

                          0.45 0    0.00 0   0.00 0   0.00 0
                          0.00      1.00     1.00     1.00
                          1.00      0.00     0.00     0.00
                          0.00      0.00     0.00     0.00

                                    0.00  0.00  0.00
                                    0     0     0

                                    0.00  0.00
                                    0     0

                                    0.00
                                    0
```

Snapshot B.1.9



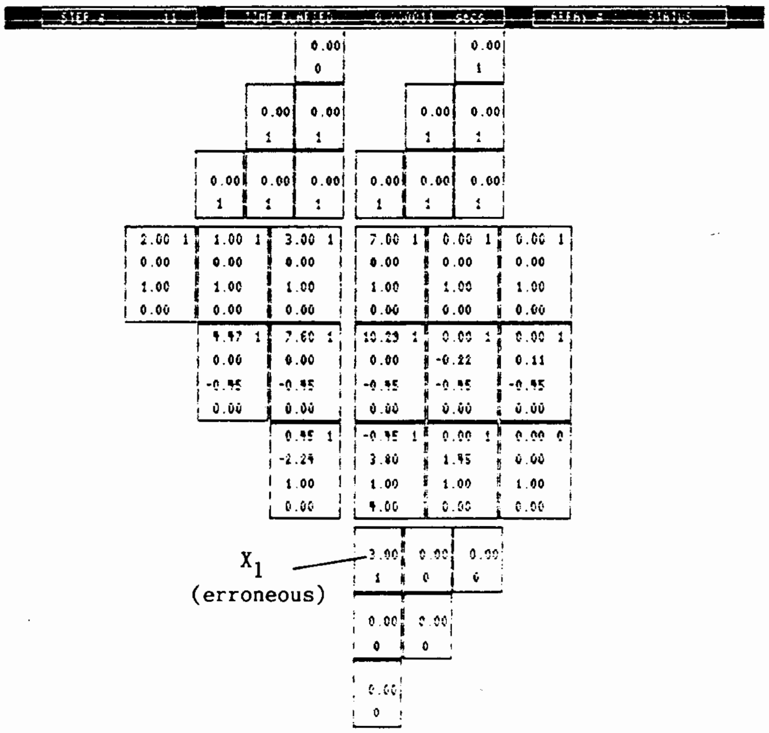
```
                                  0.00                0.00
                                  1                   1

                          0.00    0.00        0.00    0.00
                          1       1           1       1

                  0.00    0.00    0.00   0.00 0.00    0.00
                  1       1       1      1    1       1

          2.00 1  1.00 1  3.60 1    7.00 1   0.00 1   0.00 0
          0.00    0.00    0.00      0.00    -0.50     0.00
          1.00    1.00    1.00      1.00     1.00     1.00
          0.00    0.00   -1.00      0.00     0.00     0.00

                  4.47 1  7.60 1   10.25 1   0.00 0   0.00 0
                  0.00   -0.22      0.11     0.89     0.00
                 -0.45   -0.45     -0.45    -0.45     1.00
                  0.00    1.70      2.35     0.00     0.00

                          0.45 1   -0.45 0   0.00 0   0.00 0
                          1.45      0.00     1.00     1.00
                          1.00      1.00     0.00     0.00
                          0.00      0.00     0.00     0.00

                                    0.00  0.00  0.00
                                    0     0     0

                                    0.00  0.00
                                    0     0

                                    0.00
                                    0
```

Snapshot B.1.10



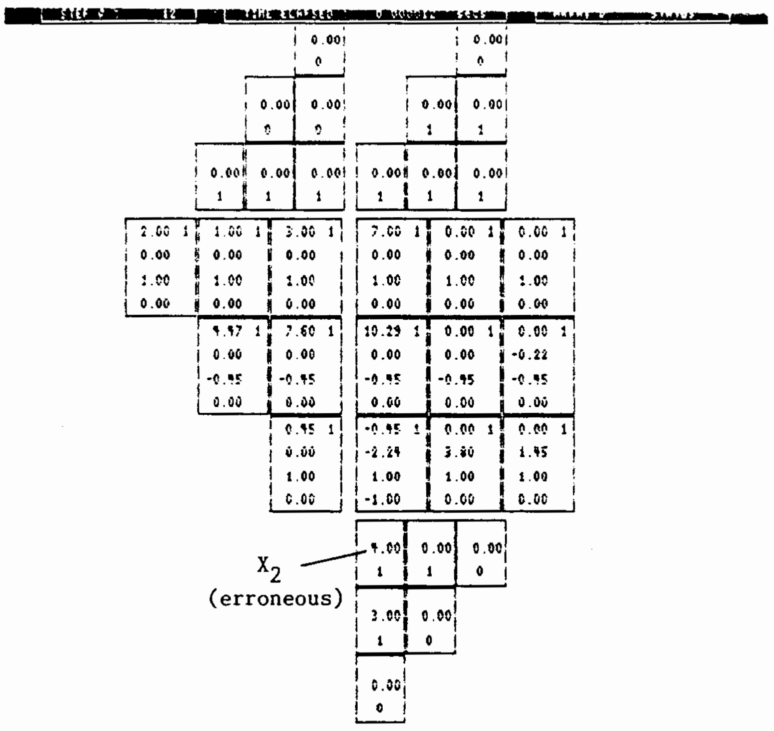
```
                            0.00                0.00
                            1                   1

                    0.00    0.00        0.00    0.00
                    1       1           1       1

            0.00    0.00    0.00    0.00    0.00    0.00
            1       1       1       1       1       1

  2.00 1    1.00 1    3.00 1    7.00 1    0.00 1    0.00 1
  0.00      0.00      0.00      0.00      0.00     -0.50
  1.00      1.00      1.00      1.00      1.00      1.00
  0.00      0.00      0.00      0.00      0.00      0.00

            4.47 1    7.60 1   10.29 1    0.00 1    0.00 0
            0.00      0.00     -0.22      0.11      0.89
           -0.95     -0.95     -0.95     -0.95     -0.95
            0.00     -1.00      2.30      0.00      0.00

                      0.95 1   -0.95 1    0.00 0    0.00 0
                      3.80      1.95      0.00      1.00
                      1.00      1.00      1.00      0.00
                      0.00      3.00      0.00      0.00

                                0.00    0.00    0.00
                                0       0       0

                                0.00    0.00
                                0       0

                                0.00
                                0
```

Snapshot B.1.11



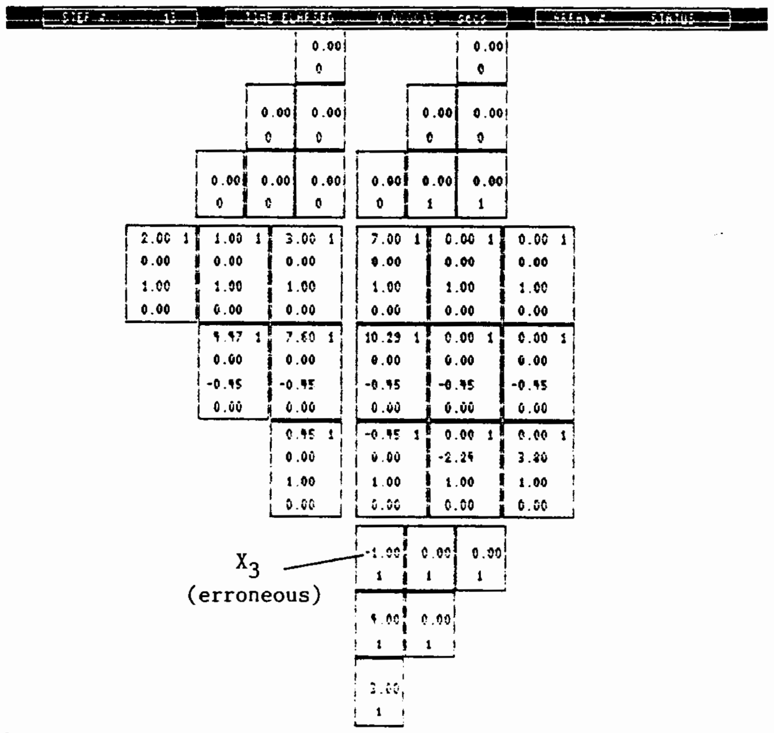
```
                            0.00                0.00
                            0                   1

                    0.00    0.00        0.00    0.00
                    1       1           1       1

            0.00    0.00    0.00    0.00    0.00    0.00
            1       1       1       1       1       1

  2.00 1    1.00 1    3.00 1    7.00 1    0.00 1    0.00 1
  0.00      0.00      0.00      0.00      0.00      0.00
  1.00      1.00      1.00      1.00      1.00      1.00
  0.00      0.00      0.00      0.00      0.00      0.00

            4.47 1    7.60 1   10.29 1    0.00 1    0.00 1
            0.00      0.00      0.00     -0.22      0.11
           -0.95     -0.95     -0.95     -0.95     -0.95
            0.00      0.00      0.00      0.00      0.00

                      0.95 1   -0.95 1    0.00 1    0.00 0
                     -2.24      3.80      1.95      0.00
                      1.00      1.00      1.00      1.00
                      0.00      4.00      0.00      0.00

                                3.00    0.00    0.00
                                1       0       0

                                0.00    0.00
                                0       0

                                0.00
                                0
```

Snapshot B.1.12



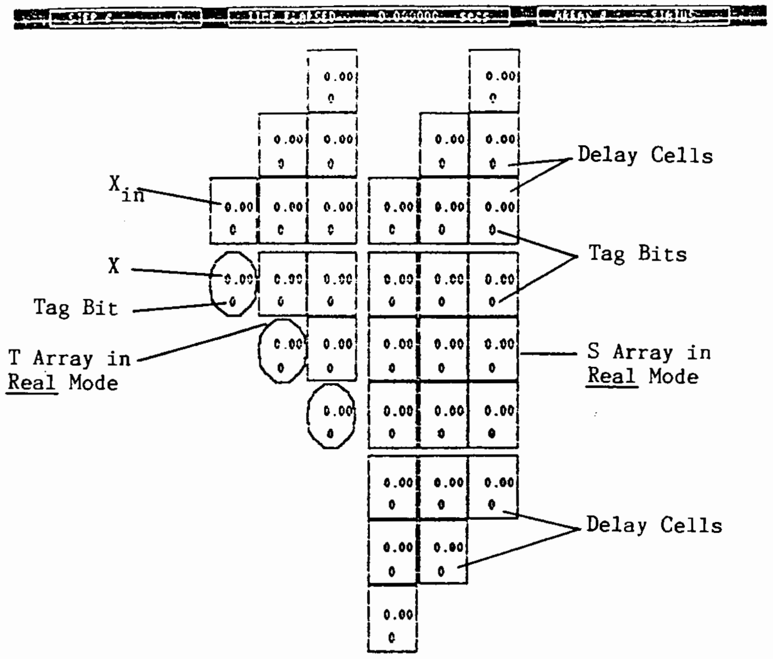
```
 STEP # 12      TIME ELAPSED  [illegible] SECS      [illegible]

                               | 0.00|                        | 0.00|
                               | 0   |                        | 0   |
                         | 0.00| 0.00|                  | 0.00| 0.00|
                         | 0   | 0   |                  | 1   | 1   |
                   | 0.00| 0.00| 0.00|            | 0.00| 0.00| 0.00|
                   | 1   | 1   | 1   |            | 1   | 1   | 1   |
          | 2.00  1| 1.00  1| 3.00  1|      | 7.00  1| 0.00  1| 0.00  1|
          | 0.00   | 0.00   | 0.00   |      | 0.00   | 0.00   | 0.00   |
          | 1.00   | 1.00   | 1.00   |      | 1.00   | 1.00   | 1.00   |
          | 0.00   | 0.00   | 0.00   |      | 0.00   | 0.00   | 0.00   |
                   | 9.97  1| 7.60  1|      |10.29  1| 0.00  1| 0.00  1|
                   | 0.00   | 0.00   |      | 0.00   | 0.00   |-0.22   |
                   |-0.95   |-0.95   |      |-0.95   |-0.95   |-0.95   |
                   | 0.00   | 0.00   |      | 0.00   | 0.00   | 0.00   |
                            | 0.95  1|      |-0.95  1| 0.00  1| 0.00  1|
                            | 0.00   |      |-2.29   | 3.80   | 1.95   |
                            | 1.00   |      | 1.00   | 1.00   | 1.00   |
                            | 0.00   |      |-1.00   | 0.00   | 0.00   |
                                            | 9.00| 0.00| 0.00|
                                            | 1   | 1   | 0   |
                                            | 3.00| 0.00|
                                            | 1   | 0   |
                                            | 0.00|
                                            | 0   |
```

Snapshot B.1.13

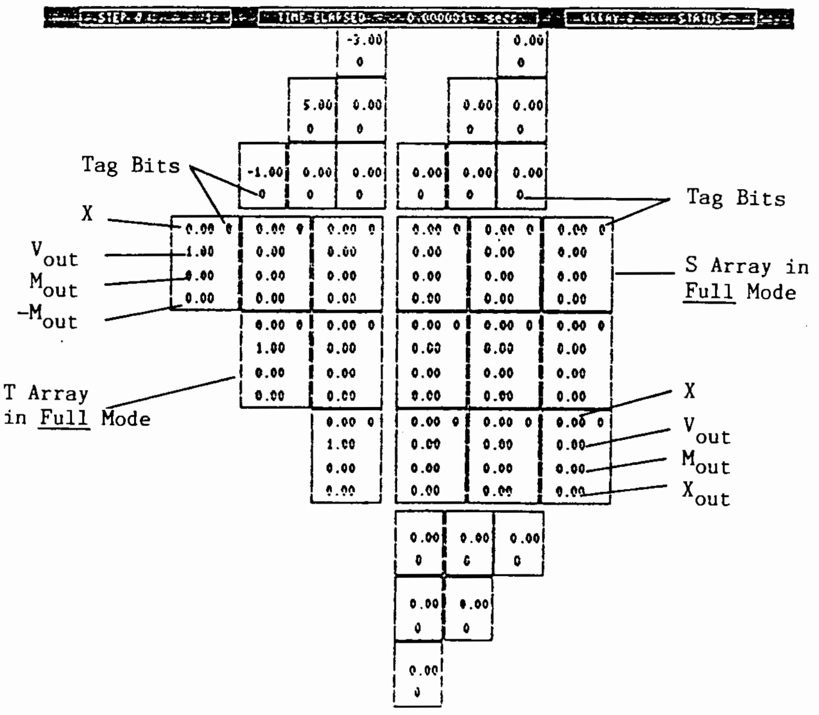


```
 STEP #    13      TIME ELAPSED  [illegible] SECS      ARRAY #     STATUS

                               | 0.00|                        | 0.00|
                               | 0   |                        | 0   |
                         | 0.00| 0.00|                  | 0.00| 0.00|
                         | 0   | 0   |                  | 0   | 0   |
                   | 0.00| 0.00| 0.00|            | 0.00| 0.00| 0.00|
                   | 0   | 0   | 0   |            | 0   | 1   | 1   |
          | 2.00  1| 1.00  1| 3.00  1|      | 7.00  1| 0.00  1| 0.00  1|
          | 0.00   | 0.00   | 0.00   |      | 0.00   | 0.00   | 0.00   |
          | 1.00   | 1.00   | 1.00   |      | 1.00   | 1.00   | 1.00   |
          | 0.00   | 0.00   | 0.00   |      | 0.00   | 0.00   | 0.00   |
                   | 9.97  1| 7.60  1|      |10.29  1| 0.00  1| 0.00  1|
                   | 0.00   | 0.00   |      | 0.00   | 0.00   | 0.00   |
                   |-0.95   |-0.95   |      |-0.95   |-0.95   |-0.95   |
                   | 0.00   | 0.00   |      | 0.00   | 0.00   | 0.00   |
                            | 0.95  1|      |-0.95  1| 0.00  1| 0.00  1|
                            | 0.00   |      | 0.00   |-2.29   | 3.80   |
                            | 1.00   |      | 1.00   | 1.00   | 1.00   |
                            | 0.00   |      | 0.00   | 0.00   | 0.00   |
                                            |-1.00| 0.00| 0.00|
                                            | 1   | 1   | 1   |
                                            | 9.00| 0.00|
                                            | 1   | 1   |
                                            | 3.00|
                                            | 1   |
```

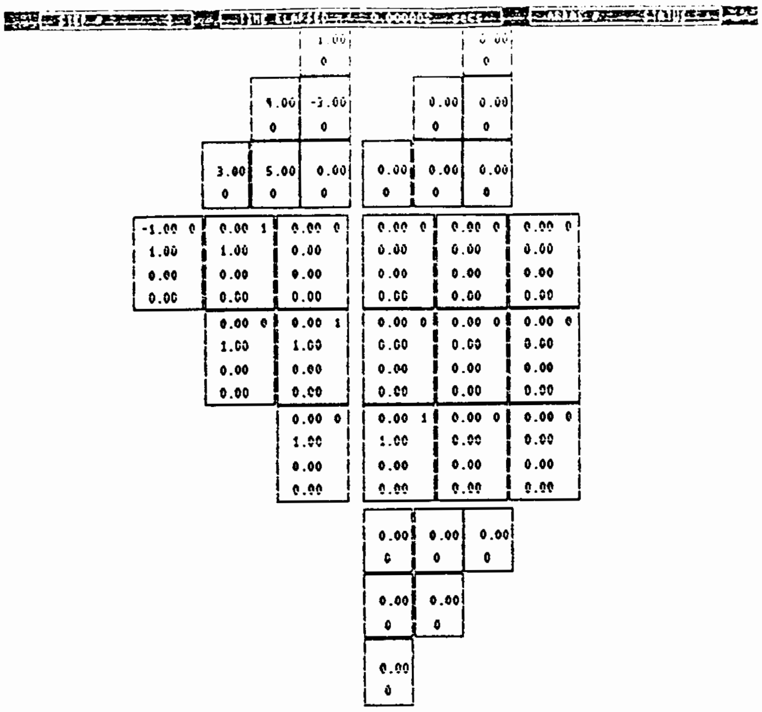
Snapshot B.1.14



Snapshot B.2.1. Simulation of Chuang and He's systolic array solving example (A.2).

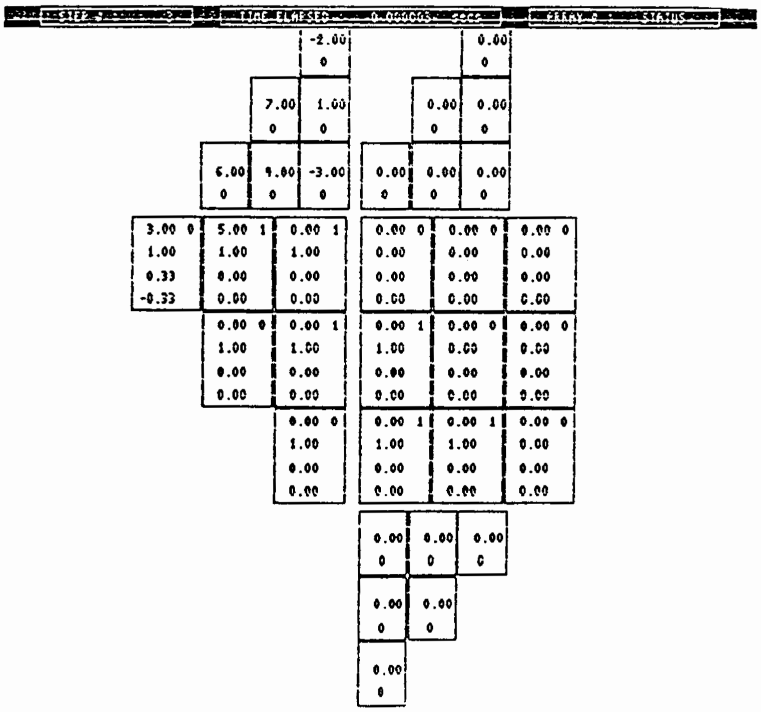


Snapshot B.2.2



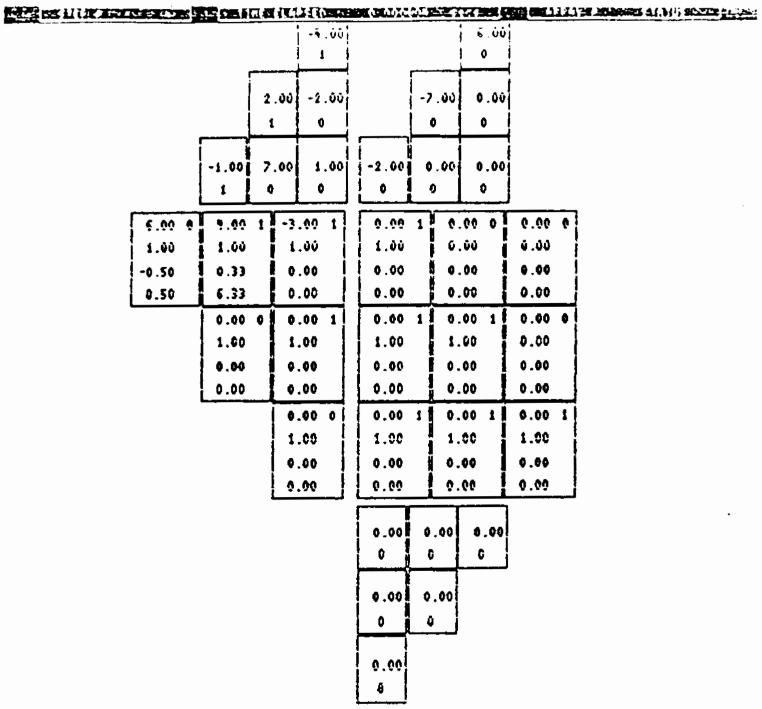
```
                         1.00          0.00
                         0             0

                 4.00   -3.00    0.00  0.00
                 0       0       0     0

          3.00   5.00    0.00    0.00  0.00  0.00
          0      0       0       0     0     0

 -1.00 0  0.00 1  0.00 0   0.00 0  0.00 0  0.00 0
  1.00    1.00    0.00     0.00    0.00    0.00
  0.00    0.00    0.00     0.00    0.00    0.00
  0.00    0.00    0.00     0.00    0.00    0.00

          0.00 0  0.00 1   0.00 0  0.00 0  0.00 0
          1.00    1.00     0.00    0.00    0.00
          0.00    0.00     0.00    0.00    0.00
          0.00    0.00     0.00    0.00    0.00

                  0.00 0   0.00 1  0.00 0  0.00 0
                  1.00     1.00    0.00    0.00
                  0.00     0.00    0.00    0.00
                  0.00     0.00    0.00    0.00

                           0.00  0.00  0.00
                           0     0     0

                           0.00  0.00
                           0     0

                           0.00
                           0
```

Snapshot B.2.3



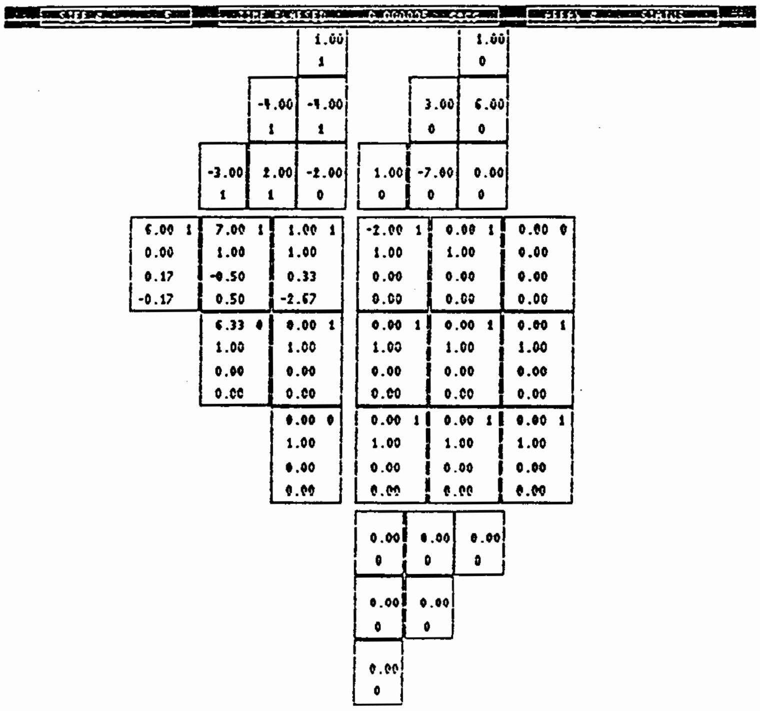
```
 STEP                TIME ELAPSED    0.000005  SECS     ARRAY      STATUS

                        -2.00          0.00
                         0             0

                 7.00    1.00    0.00  0.00
                 0       0       0     0

          6.00   4.00   -3.00    0.00  0.00  0.00
          0      0       0       0     0     0

  3.00 0  5.00 1  0.00 1   0.00 0  0.00 0  0.00 0
  1.00    1.00    1.00     0.00    0.00    0.00
  0.33    0.00    0.00     0.00    0.00    0.00
 -0.33    0.00    0.00     0.00    0.00    0.00

          0.00 0  0.00 1   0.00 1  0.00 0  0.00 0
          1.00    1.00     1.00    0.00    0.00
          0.00    0.00     0.00    0.00    0.00
          0.00    0.00     0.00    0.00    0.00

                  0.00 0   0.00 1  0.00 1  0.00 0
                  1.00     1.00    1.00    0.00
                  0.00     0.00    0.00    0.00
                  0.00     0.00    0.00    0.00

                           0.00  0.00  0.00
                           0     0     0

                           0.00  0.00
                           0     0

                           0.00
                           0
```

Snapshot B.2.4



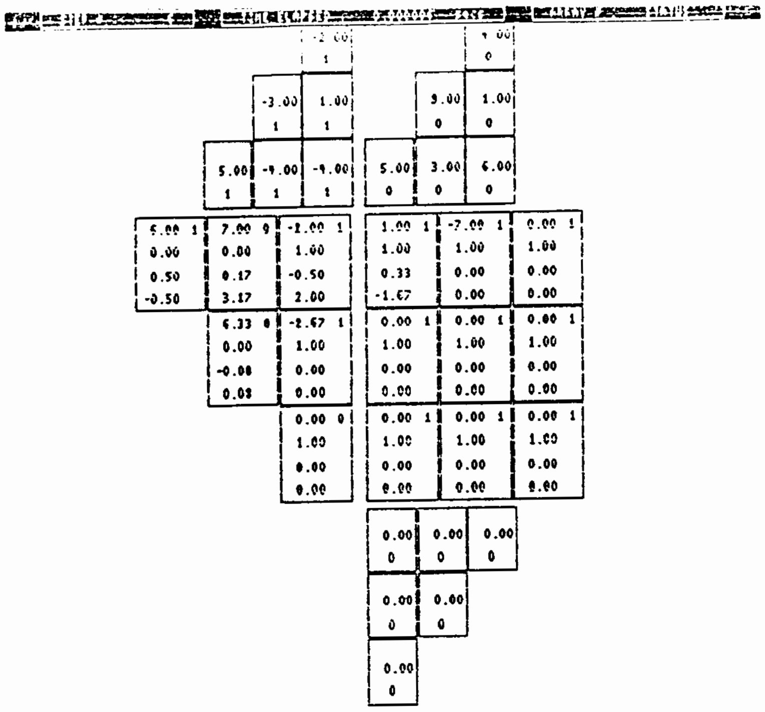
```
                                  |-4.00|              | 6.00|
                                  |  1  |              |  0  |
                            | 2.00|-2.00|        |-7.00| 0.00|
                            |  1  |  0  |        |  0  |  0  |
                      |-1.00| 7.00| 1.00|  |-2.00| 0.00| 0.00|
                      |  1  |  0  |  0  |  |  0  |  0  |  0  |
 | 6.00  0 | 4.00  1 |-3.00  1 |  | 0.00  1 | 0.00  0 | 0.00  0 |
 | 1.00    | 1.00    | 1.00    |  | 1.00    | 0.00    | 0.00    |
 |-0.50    | 0.33    | 0.00    |  | 0.00    | 0.00    | 0.00    |
 | 0.50    | 6.33    | 0.00    |  | 0.00    | 0.00    | 0.00    |
           | 0.00  0 | 0.00  1 |  | 0.00  1 | 0.00  1 | 0.00  0 |
           | 1.00    | 1.00    |  | 1.00    | 1.00    | 0.00    |
           | 0.00    | 0.00    |  | 0.00    | 0.00    | 0.00    |
           | 0.00    | 0.00    |  | 0.00    | 0.00    | 0.00    |
                     | 0.00  0 |  | 0.00  1 | 0.00  1 | 0.00  1 |
                     | 1.00    |  | 1.00    | 1.00    | 1.00    |
                     | 0.00    |  | 0.00    | 0.00    | 0.00    |
                     | 0.00    |  | 0.00    | 0.00    | 0.00    |
                                  | 0.00| 0.00| 0.00|
                                  |  0  |  0  |  0  |
                                  | 0.00| 0.00|
                                  |  0  |  0  |
                                  | 0.00|
                                  |  0  |
```

Snapshot B.2.5

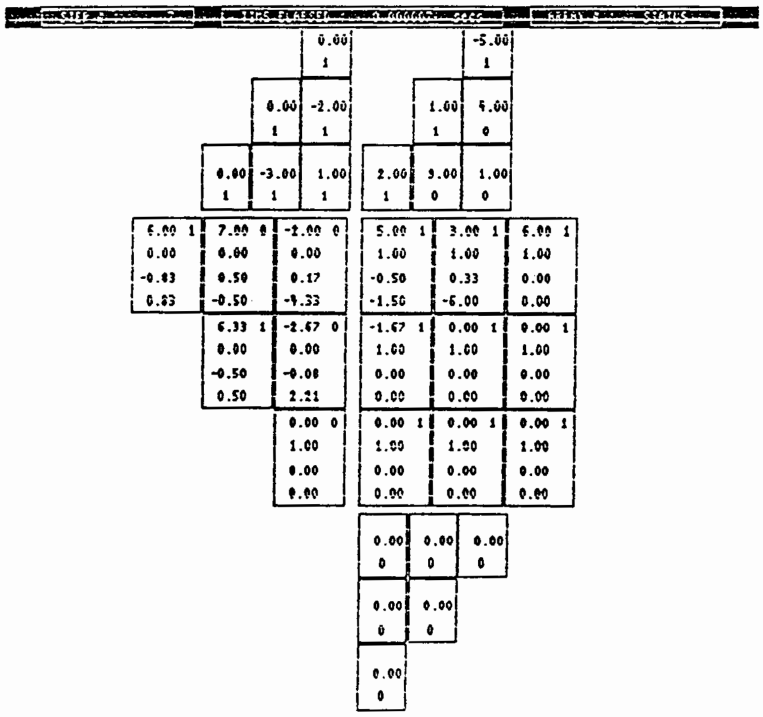


```
                                  | 1.00|              | 1.00|
                                  |  1  |              |  0  |
                            |-4.00|-4.00|        | 3.00| 6.00|
                            |  1  |  1  |        |  0  |  0  |
                      |-3.00| 2.00|-2.00|  | 1.00|-7.00| 0.00|
                      |  1  |  1  |  0  |  |  0  |  0  |  0  |
 | 6.00  1 | 7.00  1 | 1.00  1 |  |-2.00  1 | 0.00  1 | 0.00  0 |
 | 0.00    | 1.00    | 1.00    |  | 1.00    | 1.00    | 0.00    |
 | 0.17    |-0.50    | 0.33    |  | 0.00    | 0.00    | 0.00    |
 |-0.17    | 0.50    |-2.67    |  | 0.00    | 0.00    | 0.00    |
           | 6.33  0 | 0.00  1 |  | 0.00  1 | 0.00  1 | 0.00  1 |
           | 1.00    | 1.00    |  | 1.00    | 1.00    | 1.00    |
           | 0.00    | 0.00    |  | 0.00    | 0.00    | 0.00    |
           | 0.00    | 0.00    |  | 0.00    | 0.00    | 0.00    |
                     | 0.00  0 |  | 0.00  1 | 0.00  1 | 0.00  1 |
                     | 1.00    |  | 1.00    | 1.00    | 1.00    |
                     | 0.00    |  | 0.00    | 0.00    | 0.00    |
                     | 0.00    |  | 0.00    | 0.00    | 0.00    |
                                  | 0.00| 0.00| 0.00|
                                  |  0  |  0  |  0  |
                                  | 0.00| 0.00|
                                  |  0  |  0  |
                                  | 0.00|
                                  |  0  |
```

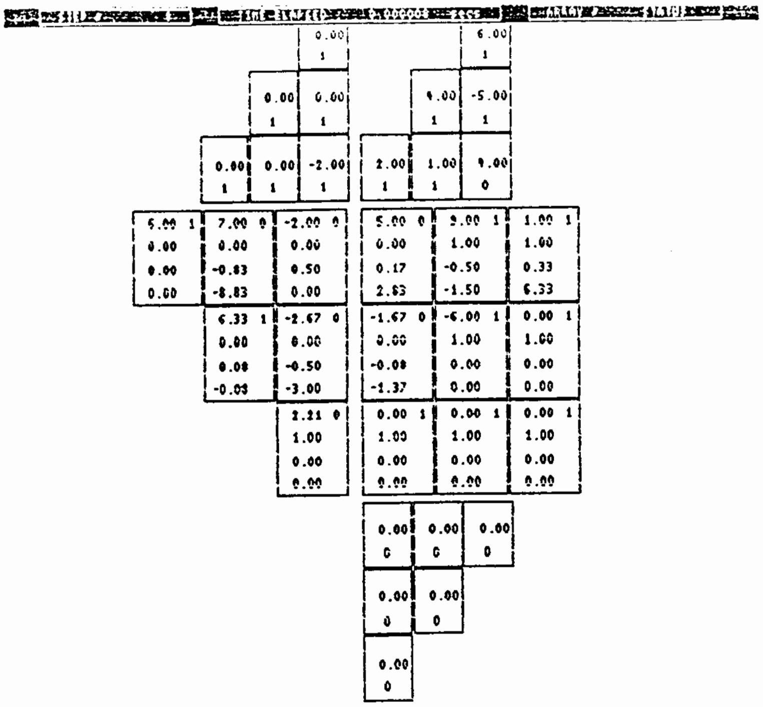
Snapshot B.2.6



Snapshot B.2.7

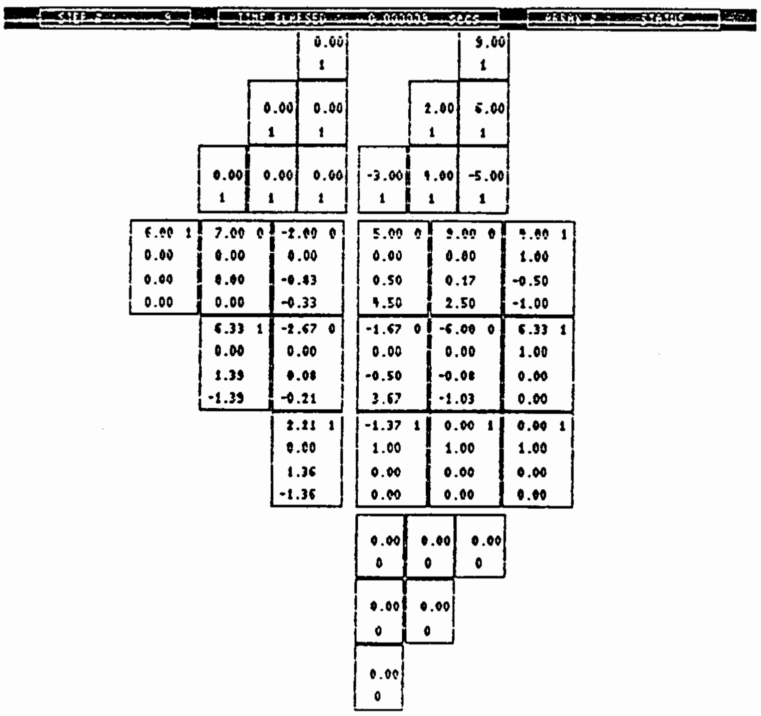


Snapshot B.2.8



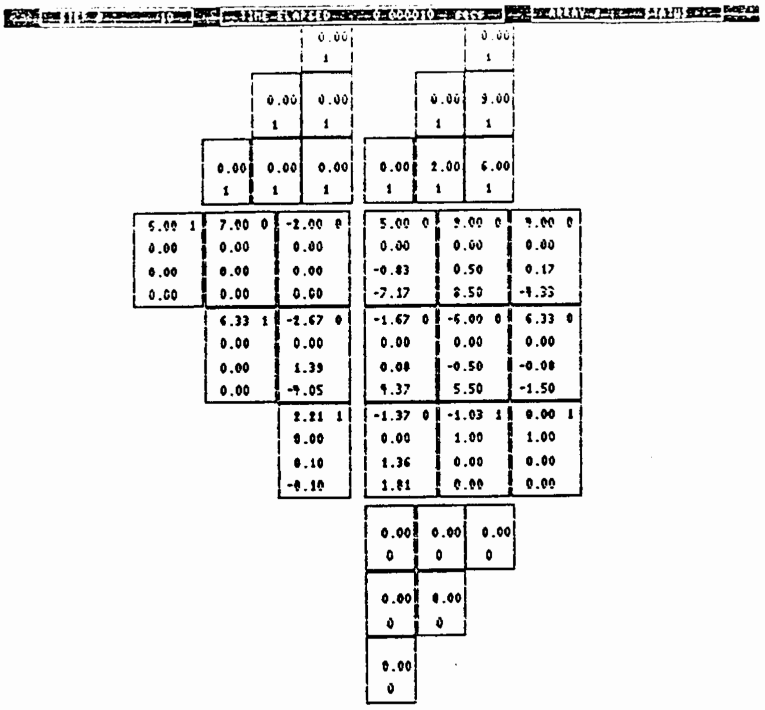
```
[illegible]
                              0.00                      6.00
                                 1                         1

                     0.00     0.00             4.00    -5.00
                        1        1                1        1

            0.00     0.00    -2.00     2.00    1.00     4.00
               1        1        1        1       1        0

 5.00 1    7.00 0   -2.00 0    5.00 0   9.00 1    1.00 1
 0.00      0.00      0.00      0.00     1.00      1.00
 0.00     -0.83      0.50      0.17    -0.50      0.33
 0.00     -8.83      0.00      2.83    -1.50      6.33

           6.33 1   -2.67 0   -1.67 0  -6.00 1    0.00 1
           0.00      0.00      0.00     1.00      1.00
           0.08     -0.50     -0.08     0.00      0.00
          -0.08     -3.00     -1.37     0.00      0.00

                     2.21 0    0.00 1   0.00 1    0.00 1
                     1.00      1.00     1.00      1.00
                     0.00      0.00     0.00      0.00
                     0.00      0.00     0.00      0.00

                               0.00     0.00     0.00
                               0        0        0

                               0.00     0.00
                               0        0

                               0.00
                               0
```

Snapshot B.2.9

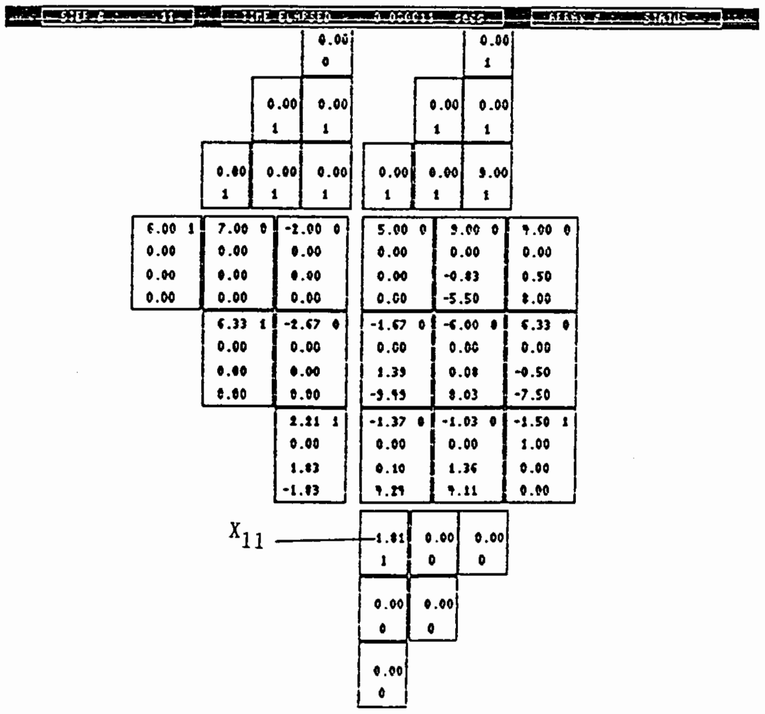


```
STEP # [illegible] 9    TIME ELAPSED [illegible] 0.000000 secs    ARRAY # [illegible] STATUS
                              0.00                      9.00
                                 1                         1

                     0.00     0.00             2.00     6.00
                        1        1                1        1

            0.00     0.00     0.00    -3.00    4.00    -5.00
               1        1        1        1       1        1

 6.00 1    7.00 0   -2.00 0    5.00 0   9.00 0    4.00 1
 0.00      0.00      0.00      0.00     0.80      1.00
 0.00      0.00     -0.83      0.50     0.17     -0.50
 0.00      0.00     -0.33      4.50     2.50     -1.00

           6.33 1   -2.67 0   -1.67 0  -6.00 0    6.33 1
           0.00      0.00      0.00     0.00      1.00
           1.39      0.08     -0.50    -0.08      0.00
          -1.39     -0.21      3.67    -1.03      0.00

                     2.21 1   -1.37 1   0.00 1    0.00 1
                     0.00      1.00     1.00      1.00
                     1.36      0.00     0.00      0.00
                    -1.36      0.00     0.00      0.00

                               0.00     0.00     0.00
                               0        0        0

                               0.00     0.00
                               0        0

                               0.00
                               0
```

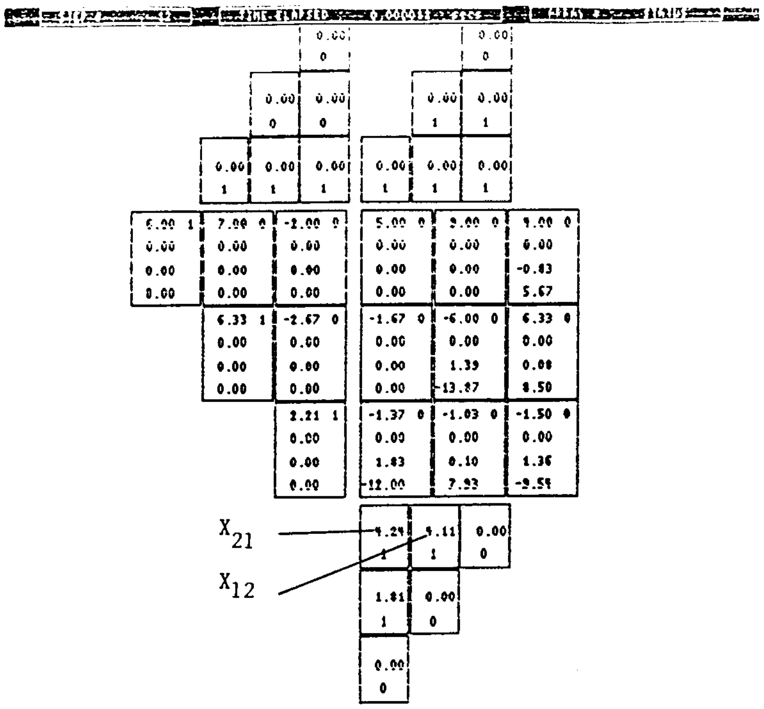
Snapshot B.2.10



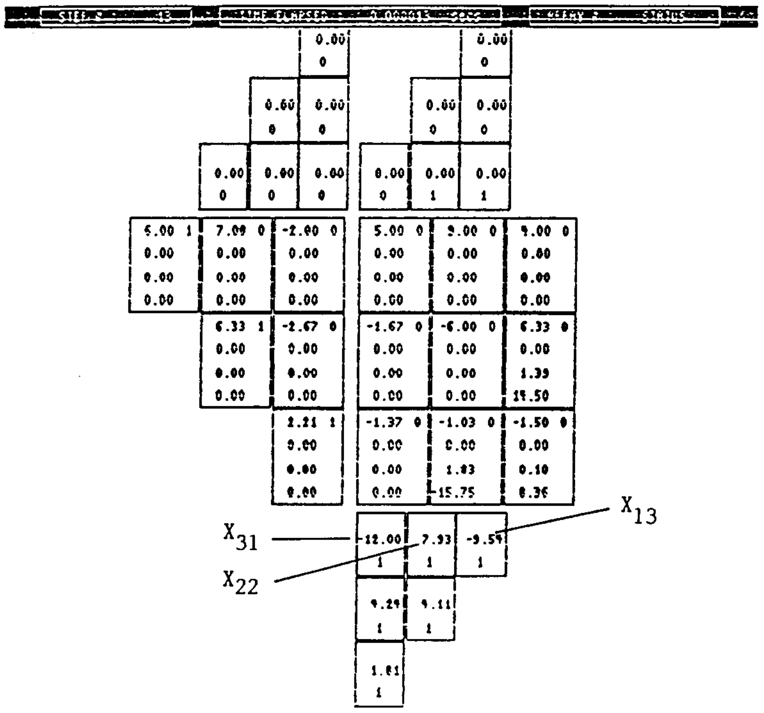
Snapshot B.2.11



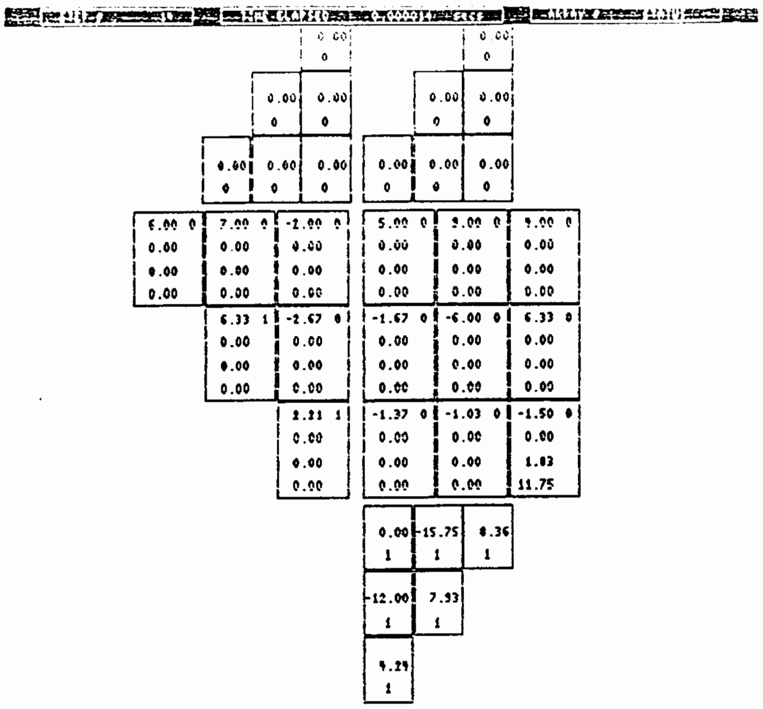
Snapshot B.2.12



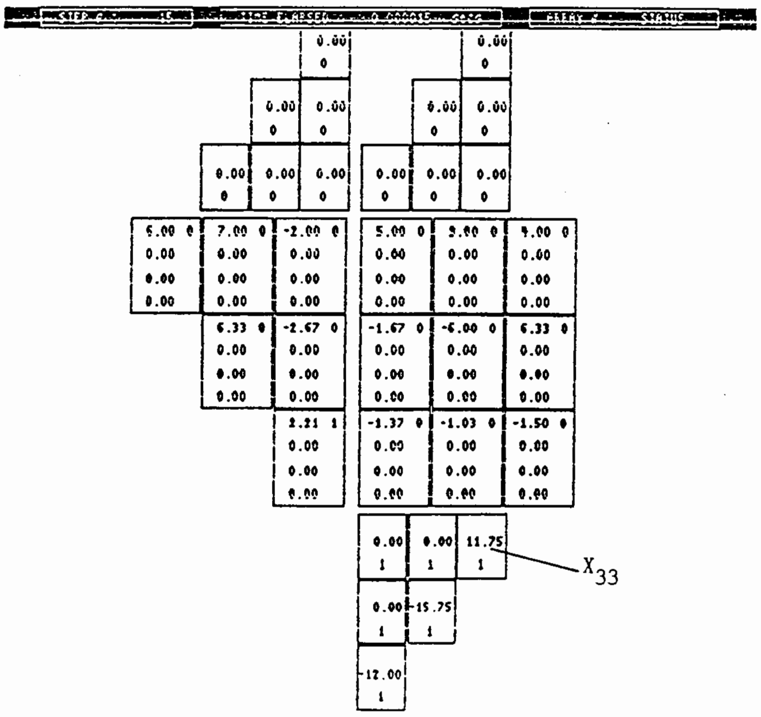
Snapshot B.2.13



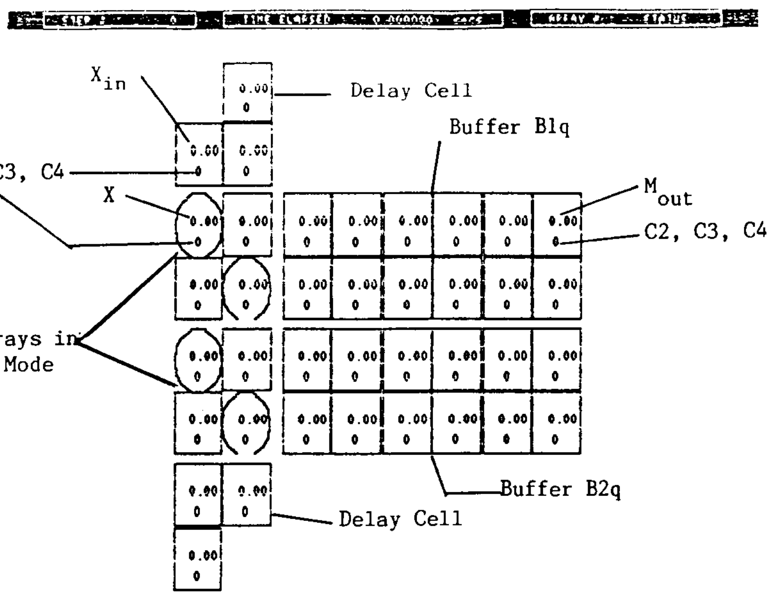
Snapshot B.2.14



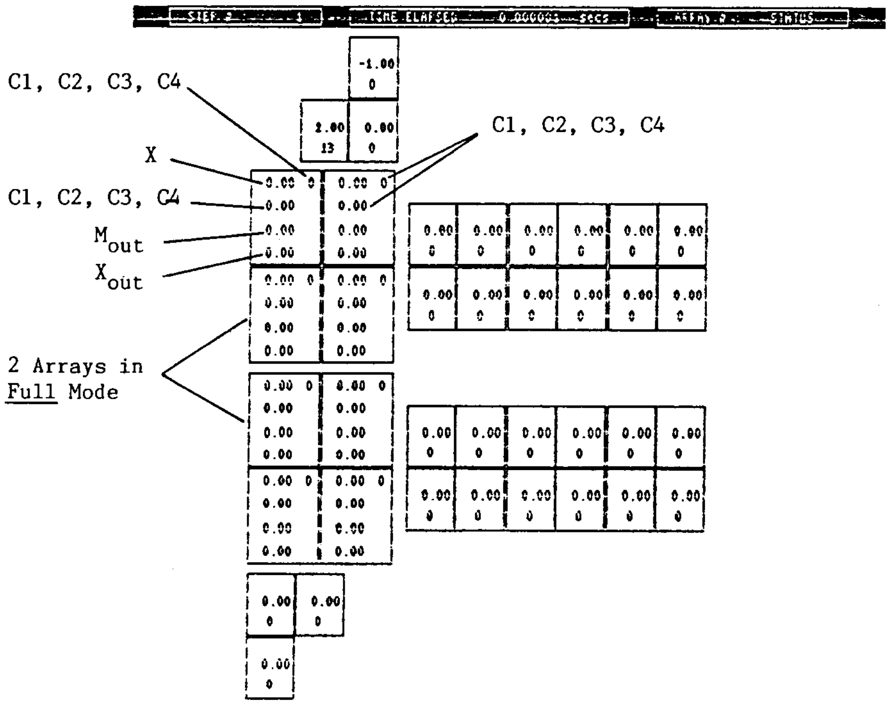
Snapshot B.2.15



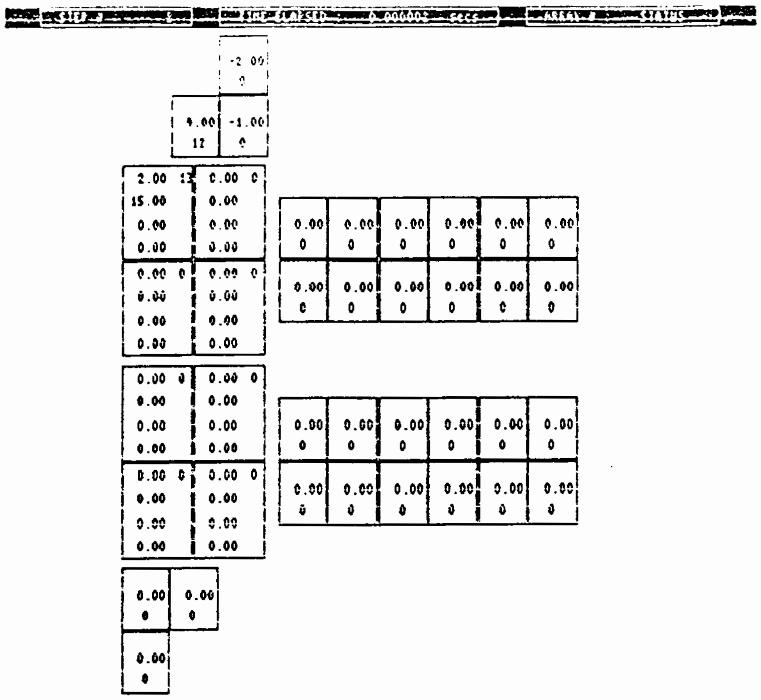
Snapshot B.2.16



Snapshot B.3.1. Simulation of an L-tuple arrays system solving example (A.3), with $L = 2$. Note that $n = 4$, $w = 2$ and therefore $m = 2$.

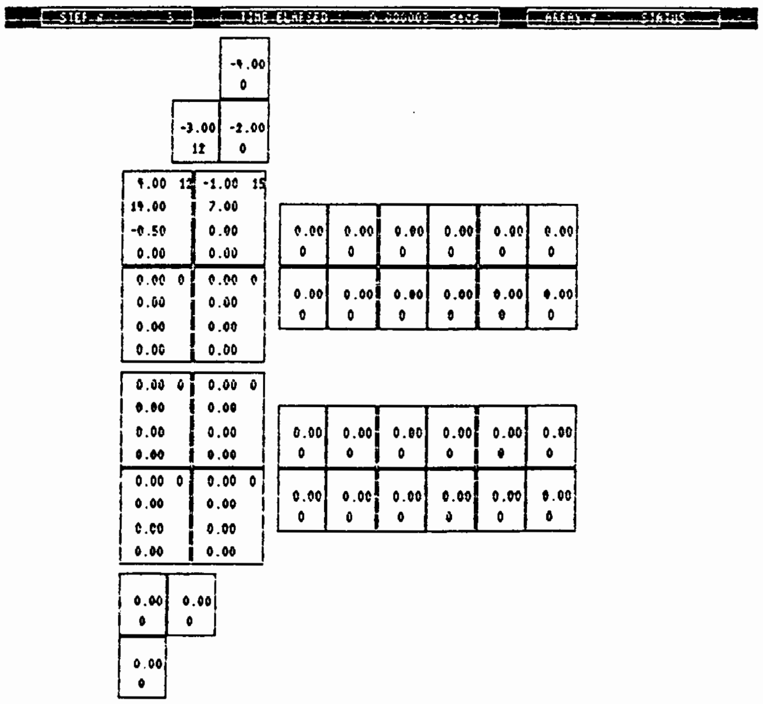


Snapshot B.3.2



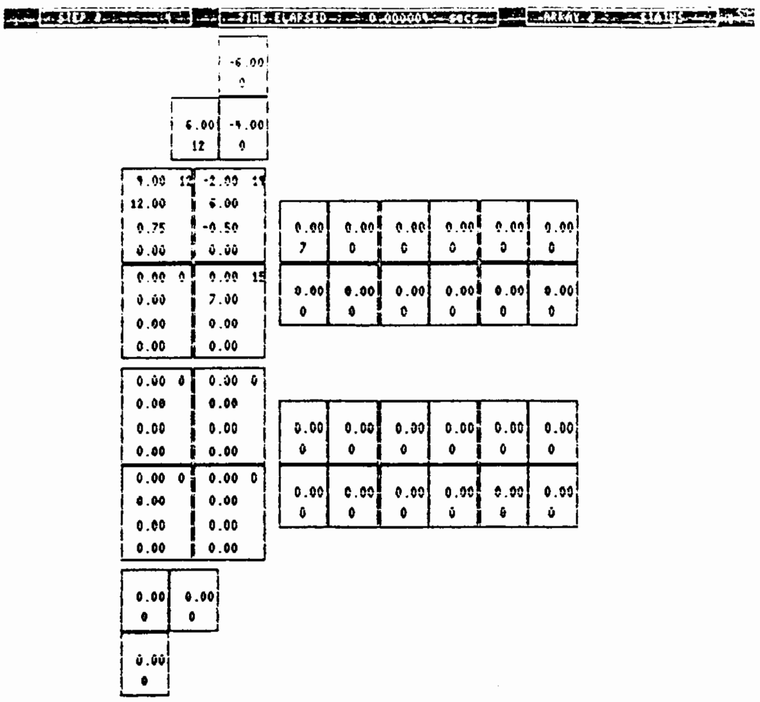
```
STEP # : 2        TIME ELAPSED : 0.000002 secs        ARRAY # : STATUS

                 -2.00
                   0
          4.00  -1.00
           12      0
   2.00  12   0.00  0
  15.00       0.00       0.00  0.00  0.00  0.00  0.00  0.00
   0.00       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00  0    0.00  0    0.00  0.00  0.00  0.00  0.00  0.00
   0.00       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00       0.00

   0.00  0    0.00  0
   0.00       0.00       0.00  0.00  0.00  0.00  0.00  0.00
   0.00       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00  0    0.00  0    0.00  0.00  0.00  0.00  0.00  0.00
   0.00       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00       0.00

   0.00  0.00
   0     0
   0.00
   0
```

Snapshot B.3.3



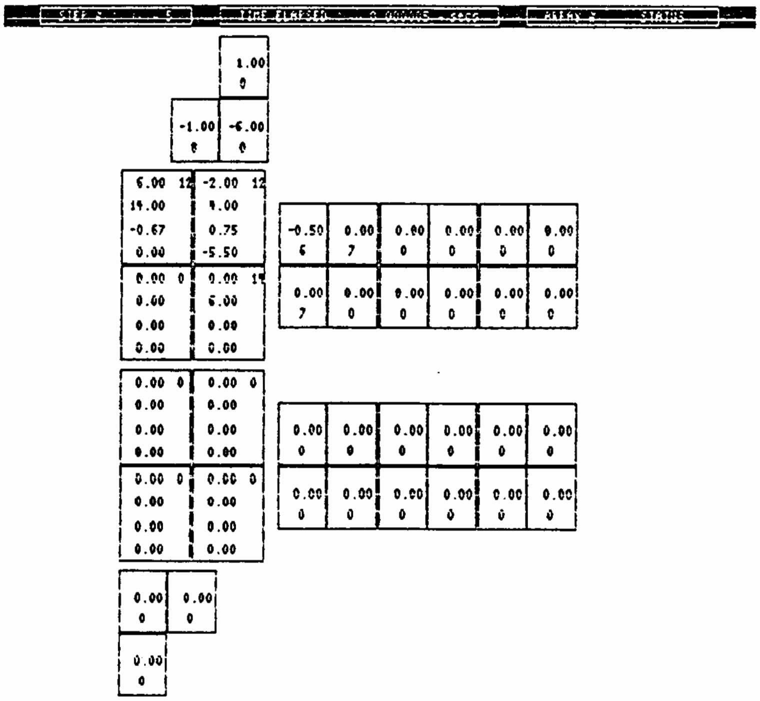
```
STEP # : 3        TIME ELAPSED : 0.000003 secs        ARRAY # : STATUS

                 -4.00
                   0
         -3.00  -2.00
           12      0
   4.00  12  -1.00  15
  14.00       7.00       0.00  0.00  0.00  0.00  0.00  0.00
  -0.50       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00  0    0.00  0    0.00  0.00  0.00  0.00  0.00  0.00
   0.00       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00       0.00

   0.00  0    0.00  0
   0.00       0.00       0.00  0.00  0.00  0.00  0.00  0.00
   0.00       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00  0    0.00  0    0.00  0.00  0.00  0.00  0.00  0.00
   0.00       0.00       0     0     0     0     0     0
   0.00       0.00
   0.00       0.00

   0.00  0.00
   0     0
   0.00
   0
```

Snapshot B.3.4



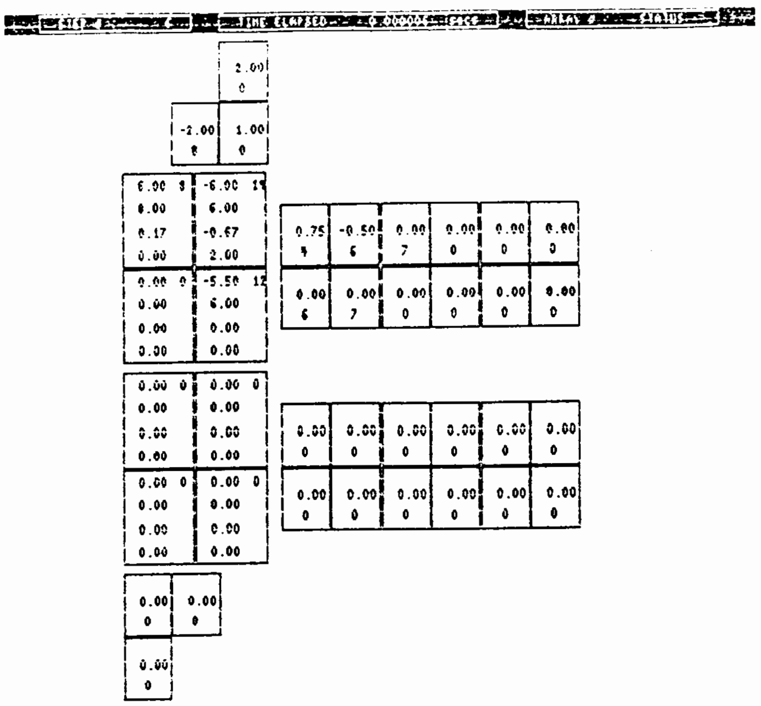
```
STEP # 4      TIME ELAPSED   0.000004 secs      ARRAY #    STATUS

                      -6.00
                        0
               6.00   -4.00
                12      0
          4.00 12  -2.00 15
         12.00      6.00
          0.75     -0.50        0.00  0.00  0.00  0.00  0.00  0.00
          0.00      0.00        7     0     0     0     0     0
          0.00 0    0.00 15
          0.00      7.00        0.00  0.00  0.00  0.00  0.00  0.00
          0.00      0.00        0     0     0     0     0     0
          0.00      0.00

          0.00 0    0.00 0
          0.00      0.00
          0.00      0.00        0.00  0.00  0.00  0.00  0.00  0.00
          0.00      0.00        0     0     0     0     0     0
          0.00 0    0.00 0
          0.00      0.00        0.00  0.00  0.00  0.00  0.00  0.00
          0.00      0.00        0     0     0     0     0     0
          0.00      0.00

          0.00  0.00
          0     0
          0.00
          0
```

Snapshot B.3.5



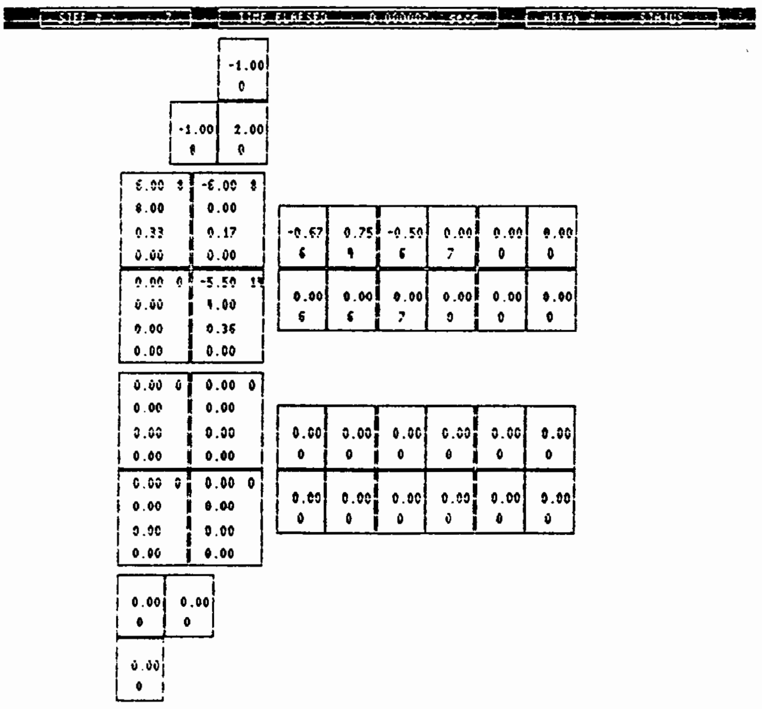
```
STEP #       5      TIME ELAPSED   0.000005 secs      ARRAY #    STATUS

                       1.00
                        0
              -1.00   -6.00
                8       0
          6.00 12  -2.00 12
         14.00      4.00
         -0.67      0.75       -0.50  0.00  0.00  0.00  0.00  0.00
          0.00     -5.50        6     7     0     0     0     0
          0.00 0    0.00 14
          0.00      6.00        0.00  0.00  0.00  0.00  0.00  0.00
          0.00      0.00        7     0     0     0     0     0
          0.00      0.00

          0.00 0    0.00 0
          0.00      0.00
          0.00      0.00        0.00  0.00  0.00  0.00  0.00  0.00
          0.00      0.00        0     0     0     0     0     0
          0.00 0    0.00 0
          0.00      0.00        0.00  0.00  0.00  0.00  0.00  0.00
          0.00      0.00        0     0     0     0     0     0
          0.00      0.00

          0.00  0.00
          0     0
          0.00
          0
```

Snapshot B.3.6



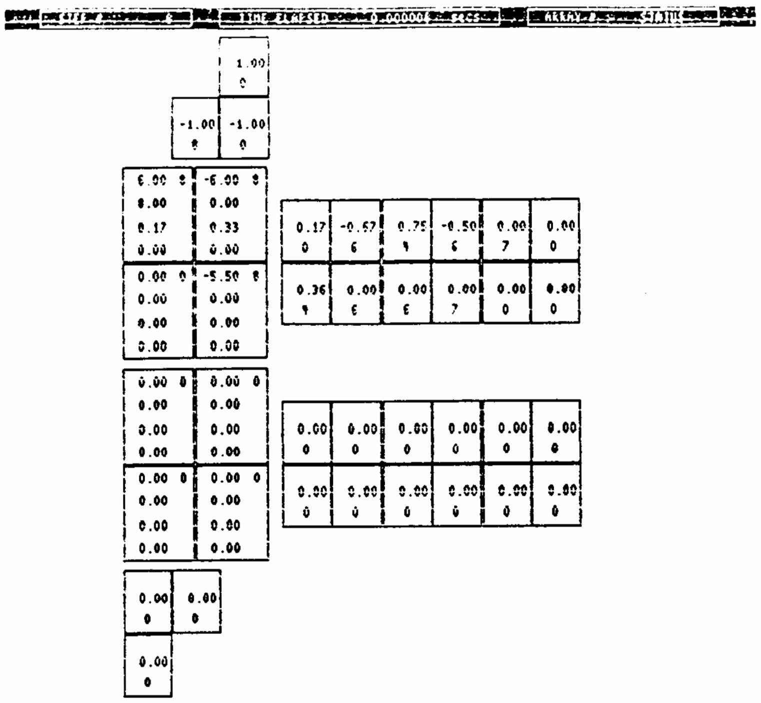
```
STEP # [illegible] 6    TIME ELAPSED  0.00000[illegible] secs    ARRAY #    STATUS

                   2.00
                   0
            -2.00  1.00
            8      0
 6.00  8   -6.00  19
 8.00       6.00
 0.17      -0.67          0.75  -0.50   0.00   0.00   0.00   0.00
 0.00       2.00          9     6       7      0      0      0
 0.00  0   -5.50  12      0.00   0.00   0.00   0.00   0.00   0.00
 0.00       6.00          6     7       0      0      0      0
 0.00       0.00
 0.00       0.00

 0.00  0    0.00  0
 0.00       0.00
 0.00       0.00          0.00   0.00   0.00   0.00   0.00   0.00
 0.00       0.00          0     0       0      0      0      0
 0.00  0    0.00  0       0.00   0.00   0.00   0.00   0.00   0.00
 0.00       0.00          0     0       0      0      0      0
 0.00       0.00
 0.00       0.00

 0.00  0.00
 0     0
 0.00
 0
```

Snapshot B.3.7



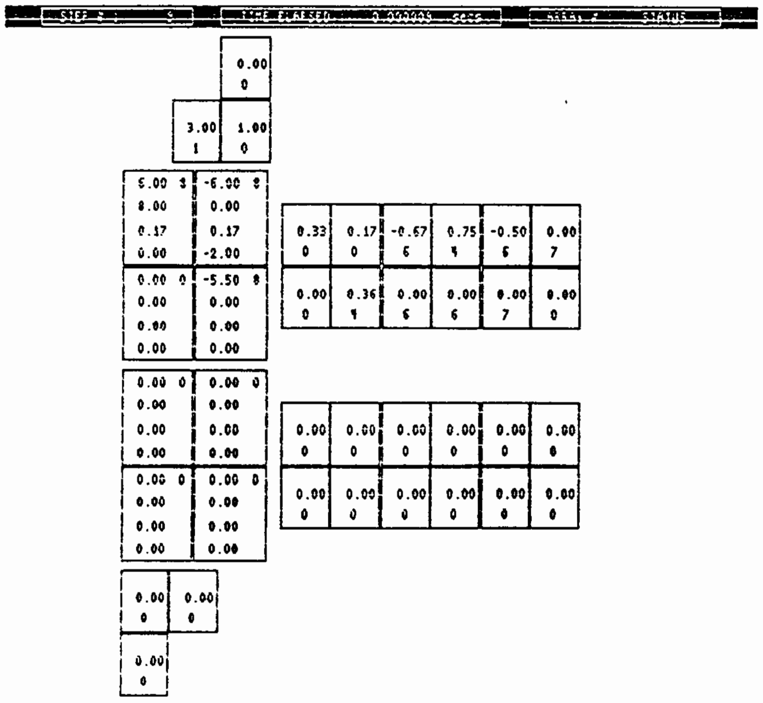
```
STEP #    7    TIME ELAPSED  0.00000[illegible] secs    ARRAY #    STATUS

                   -1.00
                   0
            -1.00  2.00
            8      0
 6.00  9   -6.00  8
 8.00       0.00
 0.33       0.17     -0.67   0.75  -0.50   0.00   0.00   0.00
 0.00       0.00      6      9     6       7      0      0
 0.00  0   -5.50  19  0.00   0.00   0.00   0.00   0.00   0.00
 0.00       4.00      6      6     7       0      0      0
 0.00       0.36
 0.00       0.00

 0.00  0    0.00  0
 0.00       0.00
 0.00       0.00      0.00   0.00   0.00   0.00   0.00   0.00
 0.00       0.00      0      0     0       0      0      0
 0.00  0    0.00  0   0.00   0.00   0.00   0.00   0.00   0.00
 0.00       0.00      0      0     0       0      0      0
 0.00       0.00
 0.00       0.00

 0.00  0.00
 0     0
 0.00
 0
```

Snapshot B.3.8



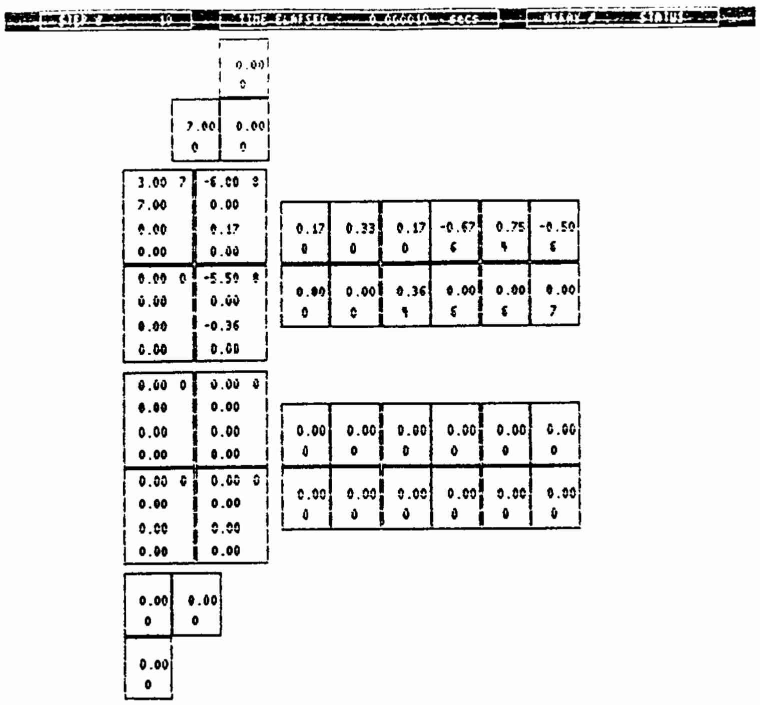
```
                 1.00
                 0
          -1.00 -1.00
          8      0
6.00  8  -6.00  8     0.17 -0.67  0.75 -0.50  0.00  0.00
8.00      0.00        0     6      9     6     7     0
0.17      0.33
0.00      0.00
0.00  0  -5.50  8     0.36  0.00  0.00  0.00  0.00  0.00
0.00      0.00        9     6      6     7     0     0
0.00      0.00
0.00      0.00

0.00  0   0.00  0     0.00  0.00  0.00  0.00  0.00  0.00
0.00      0.00        0     0      0     0     0     0
0.00      0.00
0.00      0.00
0.00  0   0.00  0     0.00  0.00  0.00  0.00  0.00  0.00
0.00      0.00        0     0      0     0     0     0
0.00      0.00
0.00      0.00

0.00  0.00
0     0
0.00
0
```

Snapshot B.3.9



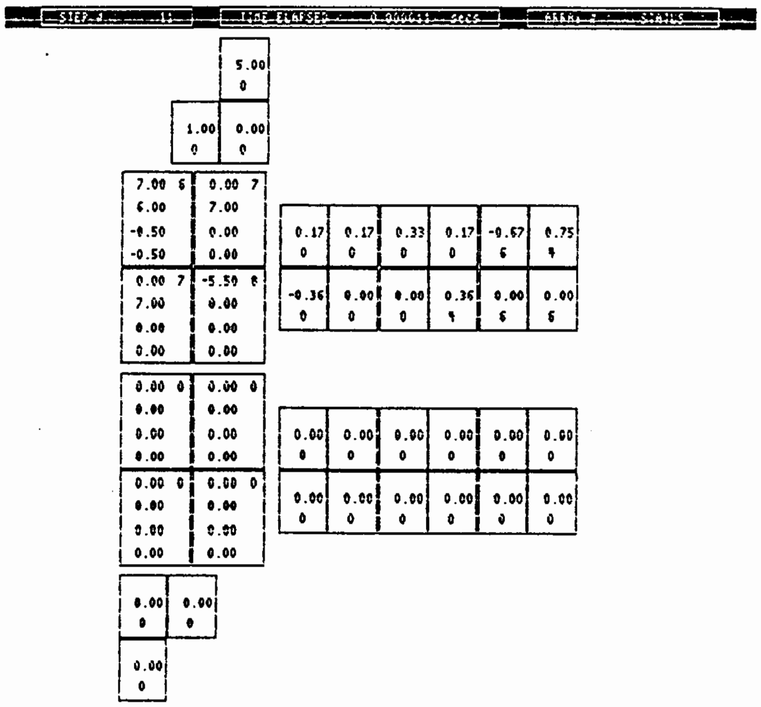
```
STEP # :    9    TIME ELAPSED   0.00000  secs    ARRAY #    STATUS

                 0.00
                 0
           3.00  1.00
           1     0
5.00  3  -6.00  8     0.33  0.17 -0.67  0.75 -0.50  0.00
8.00      0.00        0     0      6     9     6     7
0.17      0.17
0.00     -2.00
0.00  0  -5.50  8     0.00  0.36  0.00  0.00  0.00  0.00
0.00      0.00        0     9      6     6     7     0
0.00      0.00
0.00      0.00

0.00  0   0.00  0     0.00  0.00  0.00  0.00  0.00  0.00
0.00      0.00        0     0      0     0     0     0
0.00      0.00
0.00      0.00
0.00  0   0.00  0     0.00  0.00  0.00  0.00  0.00  0.00
0.00      0.00        0     0      0     0     0     0
0.00      0.00
0.00      0.00

0.00  0.00
0     0
0.00
0
```

Snapshot B.3.10



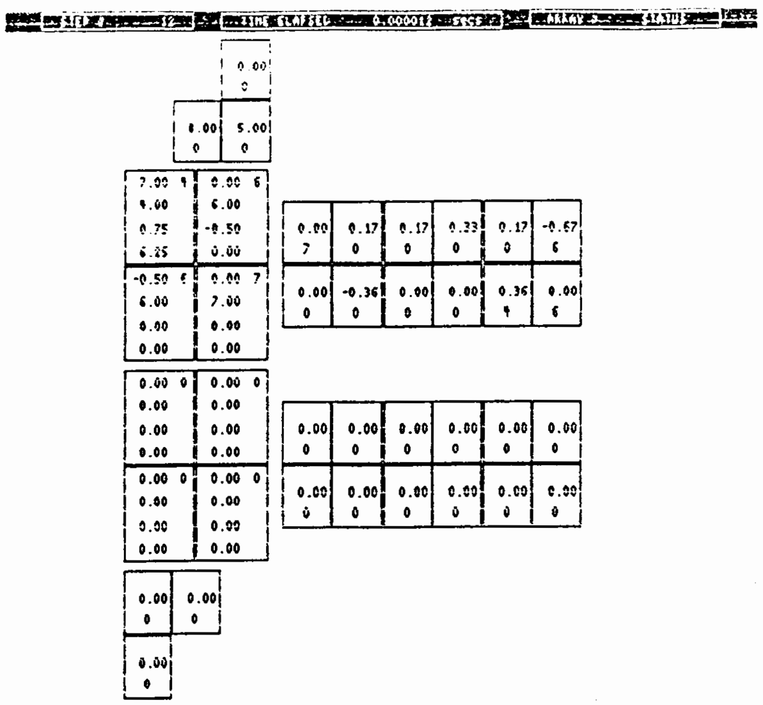
```
STEP #   10    TIME ELAPSED : 0.000010 secs    ARRAY #   STATUS

                     0.00
                     0
               7.00  0.00
               0
                     0
        3.00 7   -6.00 8
        7.00      0.00        0.17  0.33  0.17  -0.67  0.75  -0.50
        0.00      0.17        0     0     0     6      4     6
        0.00      0.00
        0.00 0   -5.50 8
        0.00      0.00        0.00  0.00  0.36   0.00  0.00   0.00
        0.00     -0.36        0     0     4      6     6      7
        0.00      0.00

        0.00 0    0.00 0
        0.00      0.00        0.00  0.00  0.00   0.00  0.00   0.00
        0.00      0.00        0     0     0      0     0      0
        0.00      0.00
        0.00 0    0.00 0
        0.00      0.00        0.00  0.00  0.00   0.00  0.00   0.00
        0.00      0.00        0     0     0      0     0      0
        0.00      0.00

        0.00  0.00
        0     0
        0.00
        0
```

Snapshot B.3.11



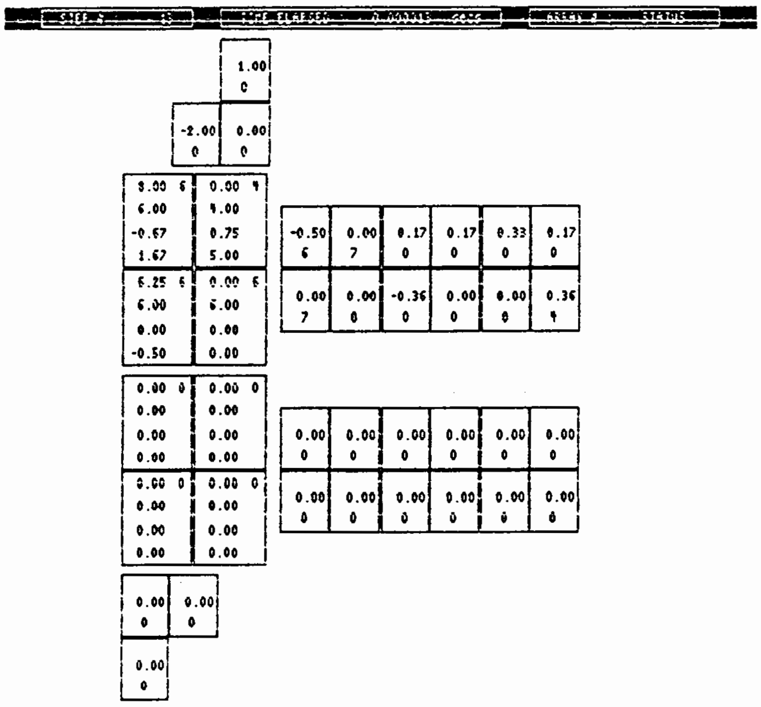
```
STEP #   11    TIME ELAPSED : 0.000011 secs    ARRAY #   STATUS

                     5.00
                     0
               1.00  0.00
               0     0
        7.00 6    0.00 7
        6.00      7.00        0.17  0.17  0.33  0.17  -0.67  0.75
       -0.50      0.00        0     0     0     0     6      4
       -0.50      0.00
        0.00 7   -5.50 8
        7.00      0.00       -0.36  0.00  0.00  0.36   0.00  0.00
        0.00      0.00        0     0     0     4      6     6
        0.00      0.00

        0.00 0    0.00 0
        0.00      0.00        0.00  0.00  0.00  0.00   0.00  0.00
        0.00      0.00        0     0     0     0      0     0
        0.00      0.00
        0.00 0    0.00 0
        0.00      0.00        0.00  0.00  0.00  0.00   0.00  0.00
        0.00      0.00        0     0     0     0      0     0
        0.00      0.00

        0.00  0.00
        0     0
        0.00
        0
```

Snapshot B.3.12



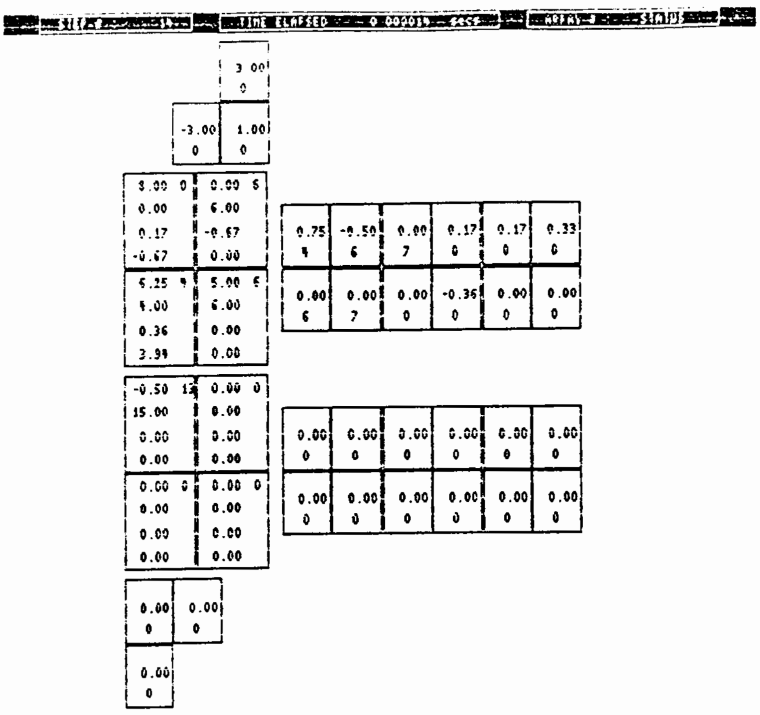
```
STEP #  12    TIME ELAPSED  0.000012  SECS    ARRAY 3   STATUS

                     0.00
                     0
               8.00  5.00
               0     0
        7.00 9  0.00 6
        9.00    6.00       0.00  0.17  0.17  0.33  0.17  -0.67
        0.75   -0.50       7     0     0     0     0     6
        6.25    0.00
       -0.50 6  0.00 7     0.00  -0.36  0.00  0.00  0.36  0.00
        6.00    7.00       0     0      0     0     9     6
        0.00    0.00
        0.00    0.00

        0.00 0  0.00 0
        0.00    0.00       0.00  0.00  0.00  0.00  0.00  0.00
        0.00    0.00       0     0     0     0     0     0
        0.00    0.00
        0.00 0  0.00 0     0.00  0.00  0.00  0.00  0.00  0.00
        0.00    0.00       0     0     0     0     0     0
        0.00    0.00
        0.00    0.00

        0.00  0.00
        0     0
        0.00
        0
```

Snapshot B.3.13

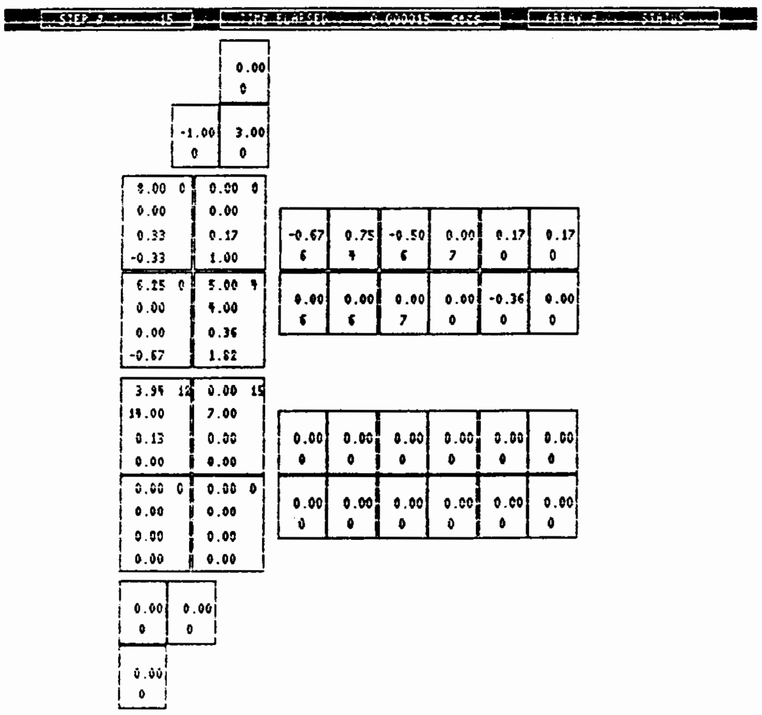


```
STEP #  [illegible]    TIME ELAPSED  [illegible]  SECS    ARRAY 3   STATUS

                     1.00
                     0
              -2.00  0.00
               0     0
        8.00 6  0.00 9
        6.00    9.00       -0.50  0.00  0.17  0.17  0.33  0.17
       -0.67    0.75        6     7     0     0     0     0
        1.67    5.00
        6.25 6  0.00 6      0.00  0.00  -0.36  0.00  0.00  0.36
        6.00    6.00        7     0     0      0     0     9
        0.00    0.00
       -0.50    0.00

        0.00 0  0.00 0
        0.00    0.00        0.00  0.00  0.00  0.00  0.00  0.00
        0.00    0.00        0     0     0     0     0     0
        0.00    0.00
        0.00 0  0.00 0      0.00  0.00  0.00  0.00  0.00  0.00
        0.00    0.00        0     0     0     0     0     0
        0.00    0.00
        0.00    0.00

        0.00  0.00
        0     0
        0.00
        0
```

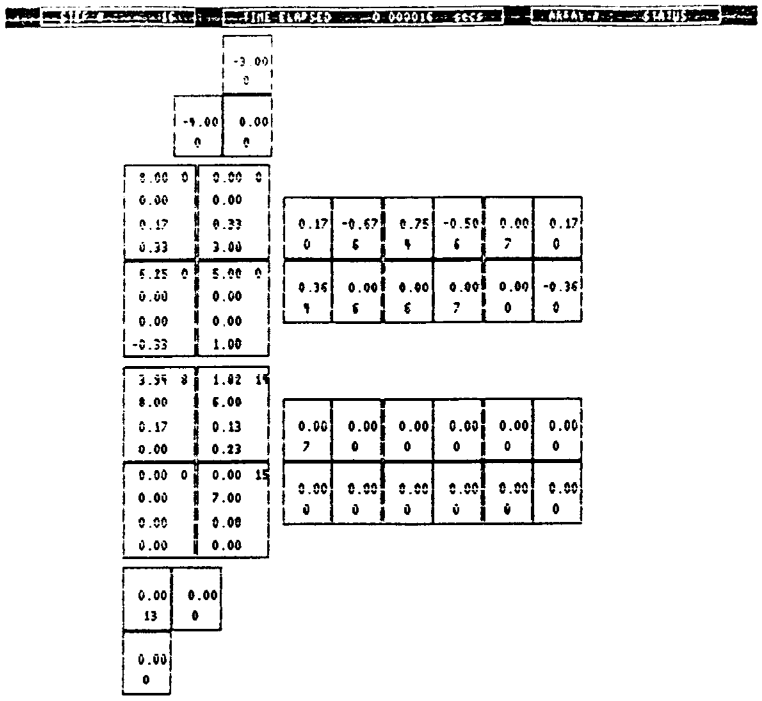
Snapshot B.3.14



Snapshot B.3.15

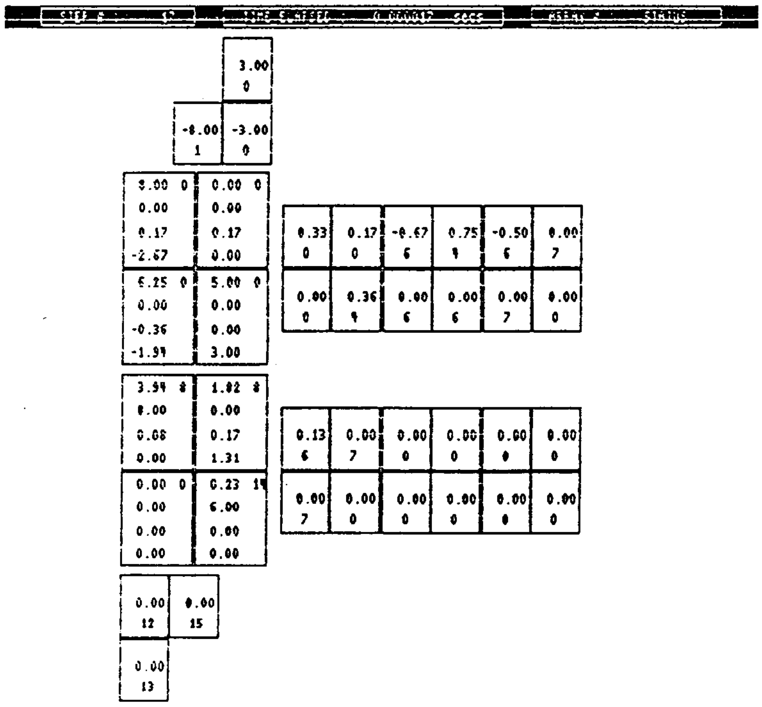


Snapshot B.3.16



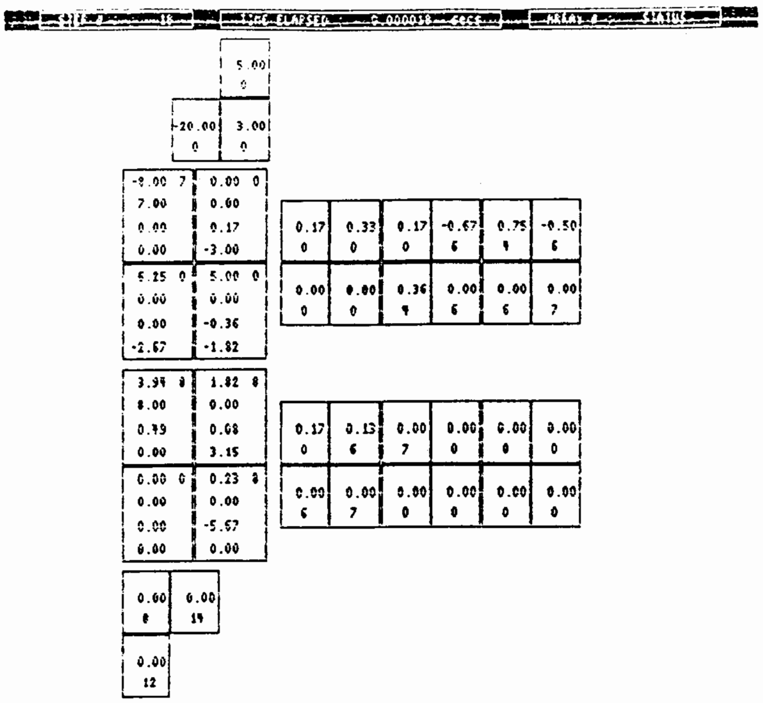
```
SIZE # 16    TIME ELAPSED 0.000016 SECS    ARRAY # STATUS

                 -3.00
                   0
         -4.00   0.00
           0       0
 8.00 0   0.00 0
 0.00     0.00
 0.17     0.33          0.17  -0.67   0.75  -0.50   0.00   0.17
 0.33     3.00          0      6      4      6      7      0
 6.25 0   5.00 0
 0.00     0.00          0.36   0.00   0.00   0.00   0.00  -0.36
 0.00     0.00          4      6      6      7      0      0
-0.33     1.00

 3.94 8   1.02 14
 8.00     6.00
 0.17     0.13          0.00   0.00   0.00   0.00   0.00   0.00
 0.00     0.23          7      0      0      0      0      0
 0.00 0   0.00 15
 0.00     7.00          0.00   0.00   0.00   0.00   0.00   0.00
 0.00     0.00          0      0      0      0      0      0
 0.00     0.00

 0.00   0.00
  13     0
 0.00
  0
```

Snapshot B.3.17



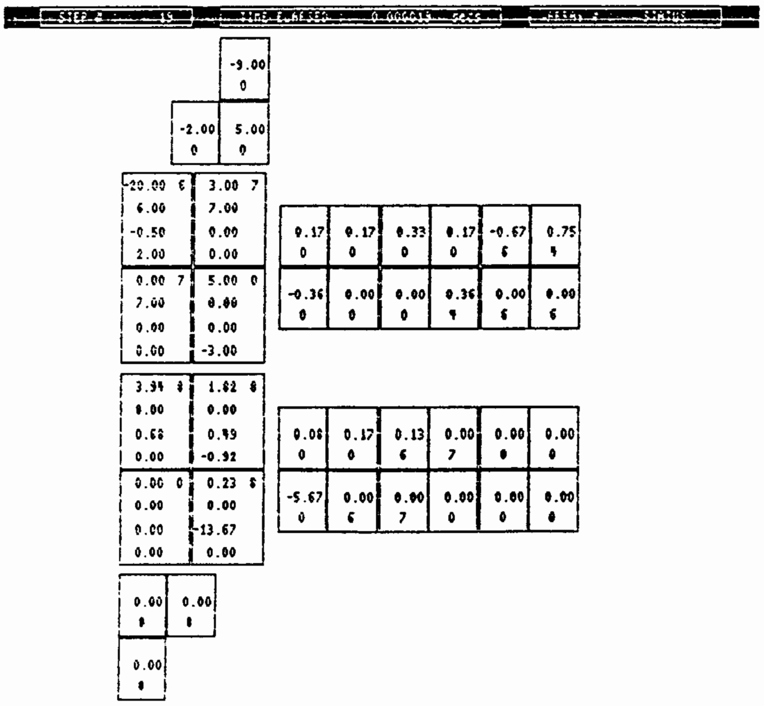
```
SIZE # [illegible]    TIME ELAPSED [illegible] SECS    ARRAY # STATUS

                  3.00
                   0
         -8.00   -3.00
           1       0
 8.00 0   0.00 0
 0.00     0.00
 0.17     0.17          0.33   0.17  -0.67   0.75  -0.50   0.00
-2.67     0.00          0      0      6      4      6      7
 6.25 0   5.00 0
 0.00     0.00          0.00   0.36   0.00   0.00   0.00   0.00
-0.36     0.00          0      4      6      6      7      0
-1.94     3.00

 3.94 8   1.82 8
 8.00     0.00
 0.08     0.17          0.13   0.00   0.00   0.00   0.00   0.00
 0.00     1.31          6      7      0      0      0      0
 0.00 0   0.23 14
 0.00     6.00          0.00   0.00   0.00   0.00   0.00   0.00
 0.00     0.00          7      0      0      0      0      0
 0.00     0.00

 0.00   0.00
  12     15
 0.00
  13
```

Snapshot B.3.18



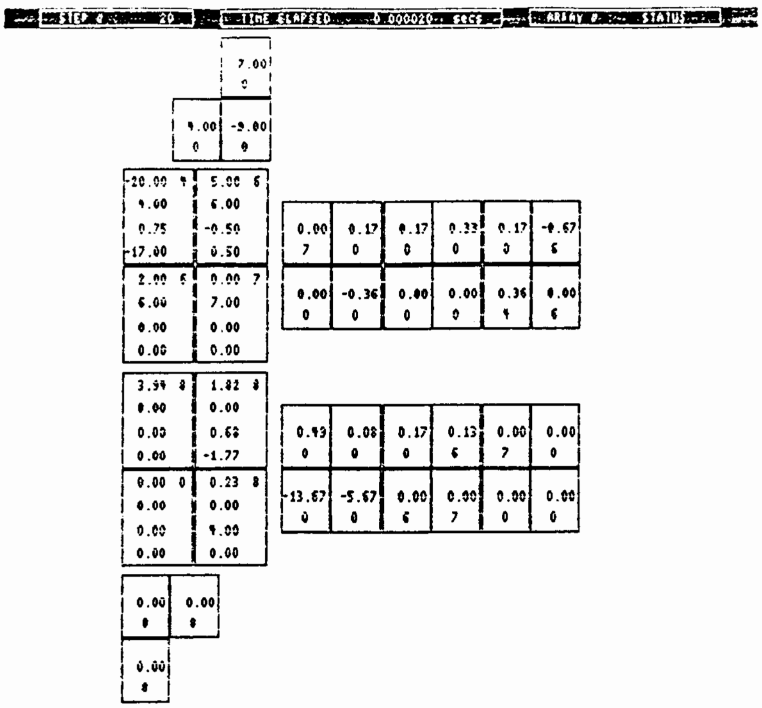
```
STEP #  18   TIME ELAPSED  0.000038  secs   ARRAY #  STATUS

                 5.00
                 0
         -20.00  3.00
         0       0
 -9.00 7   0.00 0
  7.00     0.60      0.17   0.33   0.17  -0.67   0.75  -0.50
  0.00     0.17      0      0      0      6      4      6
  0.00    -3.00
  5.25 0   5.00 0    0.00   0.00   0.36   0.00   0.00   0.00
  0.00     0.00      0      0      4      6      6      7
  0.00    -0.36
 -2.67    -1.82

  3.94 8   1.82 8
  8.00     0.00      0.17   0.13   0.00   0.00   0.00   0.00
  0.49     0.08      0      6      7      0      0      0
  0.00     3.15
  0.00 0   0.23 8    0.00   0.00   0.00   0.00   0.00   0.00
  0.00     0.00      6      7      0      0      0      0
  0.00    -5.67
  0.00     0.00

  0.00   0.00
  8      14
  0.00
  12
```

Snapshot B.3.19



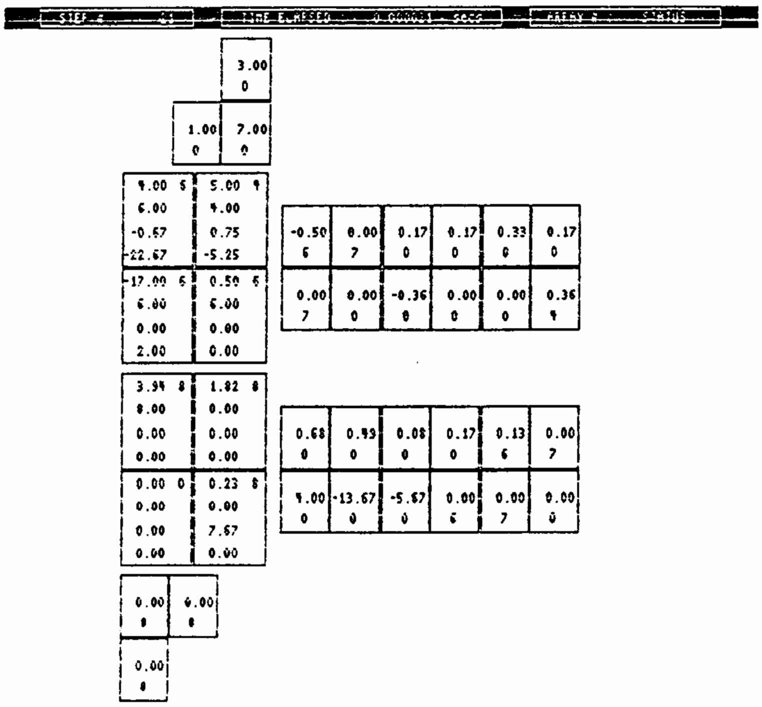
```
STEP #  19   TIME ELAPSED  [illegible]  secs   ARRAY #  STATUS

                 -9.00
                 0
         -2.00   5.00
         0       0
-20.00 6   3.00 7
  6.00     7.00      0.17   0.17   0.33   0.17  -0.67   0.75
 -0.50     0.00      0      0      0      0      6      4
  2.00     0.00
  0.00 7   5.00 0   -0.36   0.00   0.00   0.36   0.00   0.00
  7.00     0.00      0      0      0      4      6      6
  0.00     0.00
  0.00    -3.00

  3.94 8   1.82 8
  8.00     0.00      0.08   0.17   0.13   0.00   0.00   0.00
  0.68     0.49      0      0      6      7      0      0
  0.00    -0.92
  0.00 0   0.23 8   -5.67   0.00   0.00   0.00   0.00   0.00
  0.00     0.00      0      6      7      0      0      0
  0.00   -13.67
  0.00     0.00

  0.00   0.00
  8      8
  0.00
  8
```

Snapshot B.3.20



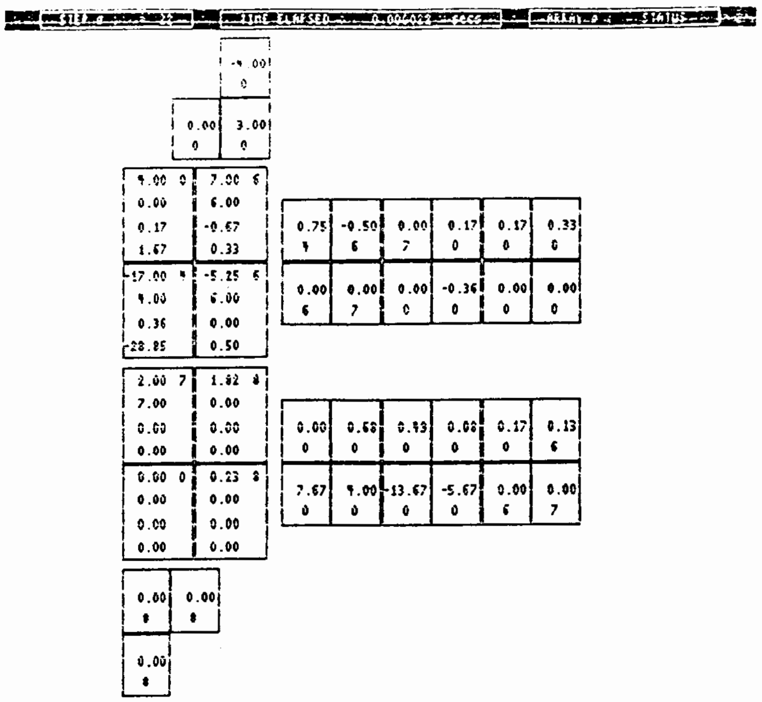
```
STEP # .... 20   TIME ELAPSED ..... 0.000020.. secs   ARRAY # .... STATUS

                      7.00
                      0
               4.00  -3.00
               0      0
       -20.00 4   5.00 6
         4.00     6.00        0.00   0.17   0.17   0.33   0.17  -0.67
         0.75    -0.50        7      0      0      0      0      6
       -17.00     0.50
         2.00 6   0.00 7      0.00  -0.36   0.00   0.00   0.36   0.00
         6.00     7.00        0      0      0      0      4      6
         0.00     0.00
         0.00     0.00

         3.94 8   1.82 8
         8.00     0.00        0.93   0.08   0.17   0.13   0.00   0.00
         0.00     0.68        0      0      0      6      7      0
         0.00    -1.77
         0.00 0   0.23 8     -13.67  -5.67  0.00   0.00   0.00   0.00
         0.00     0.00        0      0      6      7      0      0
         0.00     4.00
         0.00     0.00

         0.00  0.00
         8     8
         0.00
         8
```

Snapshot B.3.21



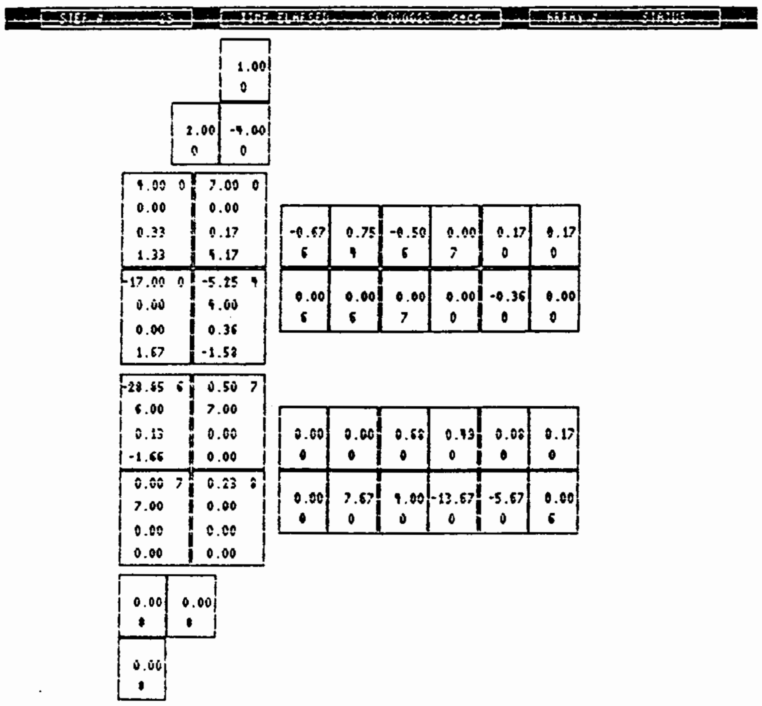
```
STEP # ..... [illegible]   TIME ELAPSED ..... [illegible] .. secs   ARRAY # .... STATUS

                      3.00
                      0
               1.00   7.00
               0      0
         4.00 6   5.00 4
         6.00     4.00       -0.50   0.00   0.17   0.17   0.33   0.17
        -0.67     0.75        6      7      0      0      0      0
       -22.67    -5.25
       -17.00 6   0.50 6      0.00   0.00  -0.36   0.00   0.00   0.36
         6.00     6.00        7      0      8      0      0      4
         0.00     0.00
         2.00     0.00

         3.94 8   1.82 8
         8.00     0.00        0.68   0.93   0.08   0.17   0.13   0.00
         0.00     0.00        0      0      0      0      6      7
         0.00     0.00
         0.00 0   0.23 8      4.00 -13.67  -5.67   0.00   0.00   0.00
         0.00     0.00        0      0      0      6      7      0
         0.00     7.67
         0.00     0.00

         0.00  6.00
         8     8
         0.00
         8
```

Snapshot B.3.22



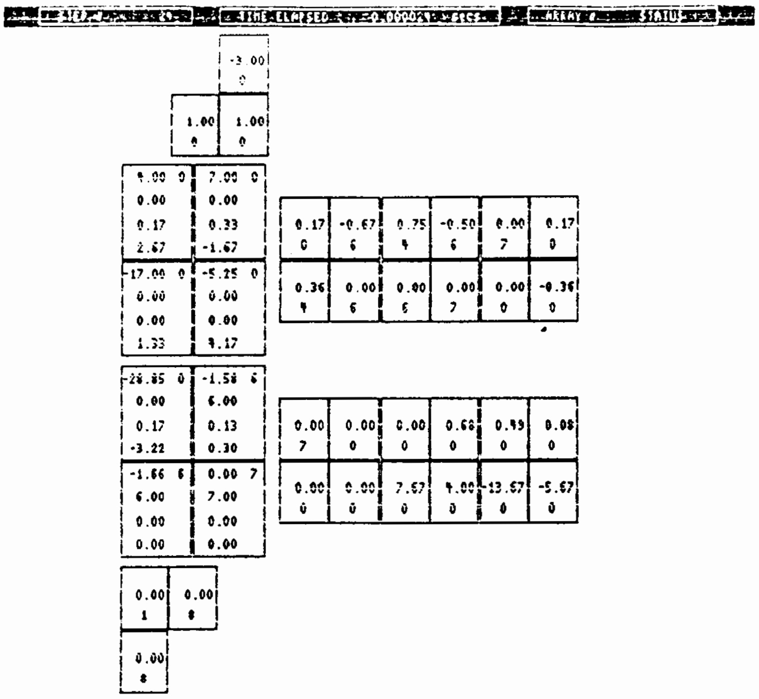
```
STEP # : 22     TIME ELAPSED : 0.006022 secs     ARRAY # : STATUS

                 -4.00
                  0
          0.00   3.00
           0      0
  4.00 0   7.00 6
  0.00     6.00        0.75  -0.50  0.00  0.17  0.17  0.33
  0.17    -0.67         4     6      7     0     0     0
  1.67     0.33
-17.00 4  -5.25 6      0.00  0.00  0.00 -0.36  0.00  0.00
  4.00     6.00         6     7     0     0     0     0
  0.36     0.00
-28.85     0.50

  2.00 7   1.82 8
  7.00     0.00        0.00  0.68  0.43  0.08  0.17  0.13
  0.00     0.00         0     0     0     0     0     6
  0.00     0.00
  0.00 0   0.23 8      7.67  4.00 -13.67 -5.67  0.00  0.00
  0.00     0.00         0     0     0      0     6     7
  0.00     0.00
  0.00     0.00

  0.00   0.00
   8      8
  0.00
   8
```

Snapshot B.3.23



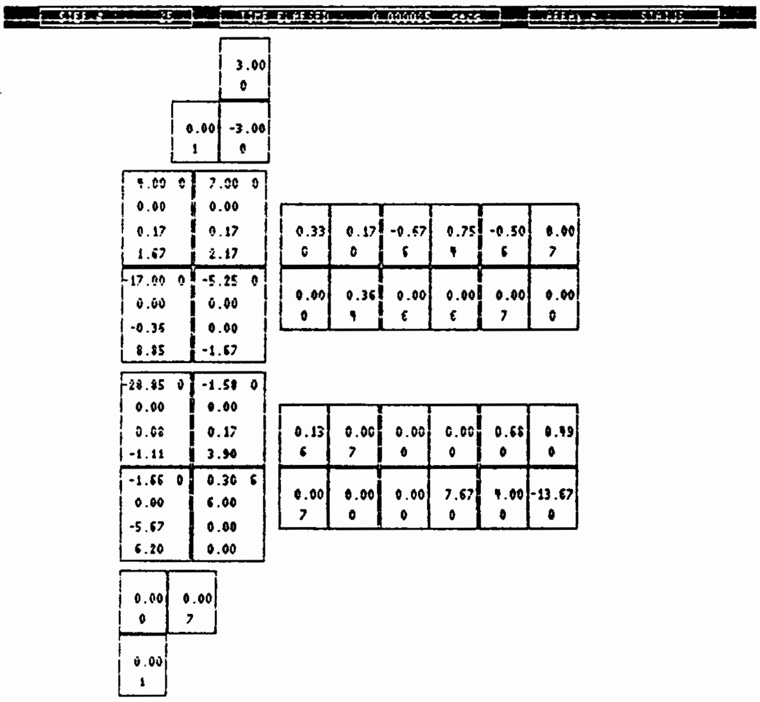
```
STEP # : 23     TIME ELAPSED : 0.006663 secs     ARRAY # : STATUS

                  1.00
                   0
          2.00  -4.00
           0      0
  4.00 0   7.00 0
  0.00     0.00       -0.67  0.75 -0.50  0.00  0.17  0.17
  0.33     0.17         6     4     6     7     0     0
  1.33     4.17
-17.00 0  -5.25 4      0.00  0.00  0.00  0.00 -0.36  0.00
  0.00     4.00         6     6     7     0     0     0
  0.00     0.36
  1.67    -1.58

-28.65 6   0.50 7
  6.00     7.00        0.00  0.00  0.68  0.43  0.08  0.17
  0.13     0.00         0     0     0     0     0     0
 -1.66     0.00
  0.00 7   0.23 8      0.00  7.67  4.00 -13.67 -5.67  0.00
  7.00     0.00         0     0     0      0     0     6
  0.00     0.00
  0.00     0.00

  0.00   0.00
   8      8
  0.00
   8
```

Snapshot B.3.24



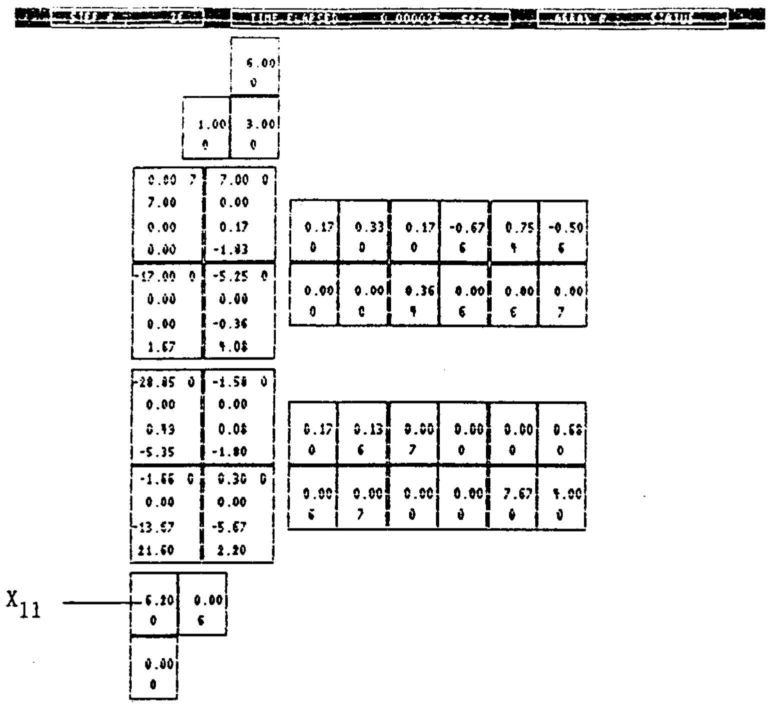
```
STEP #   25     TIME ELAPSED = 0.00024 secs     ARRAY #     STATUS

                  -3.00
                    0
           1.00   1.00
             0      0
    4.00 0  7.00 0
    0.00    0.00        0.17  -0.67  0.75  -0.50  0.00  0.17
    0.17    0.33          0     6      4     6      7     0
    2.67   -1.67
  -17.00 0 -5.25 0
    0.00    0.00        0.36   0.00  0.00   0.00  0.00  -0.36
    0.00    0.00          4     6      6     7      0     0
    1.33    4.17

  -28.85 0 -1.58 6
    0.00    6.00
    0.17    0.13        0.00   0.00  0.00   0.68  0.93  0.08
   -3.22    0.30          7     0      0     0      0     0
   -1.66 6  0.00 7
    6.00    7.00        0.00   0.00  7.67   4.00 -13.67 -5.67
    0.00    0.00          0     0      0     0      0     0
    0.00    0.00

    0.00   0.00
      1      8
    0.00
      8
```

Snapshot B.3.25

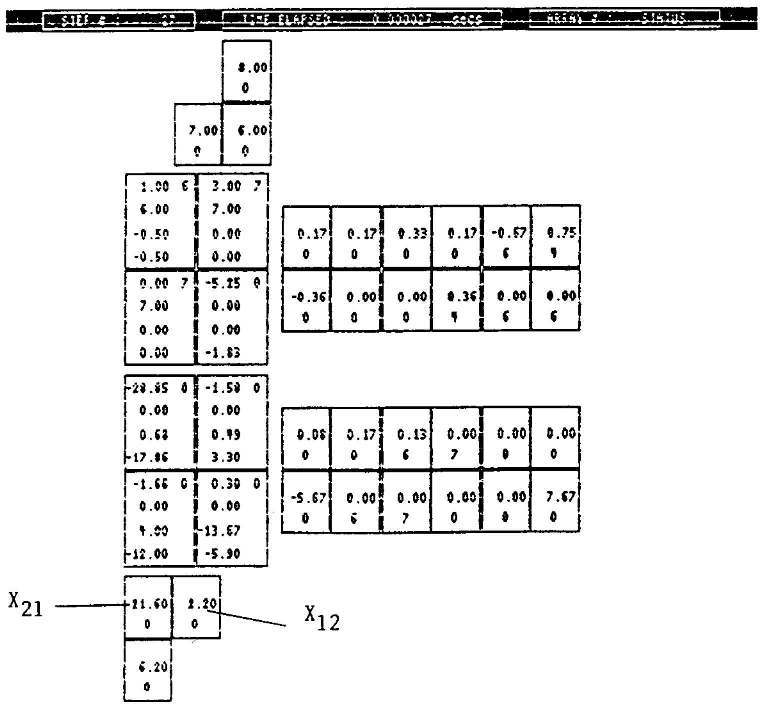


```
STEP #   26     TIME ELAPSED = 0.00025 secs     ARRAY #     STATUS

                  3.00
                    0
           0.00  -3.00
             1      0
    4.00 0  7.00 0
    0.00    0.00        0.33   0.17  -0.67  0.75  -0.50  0.00
    0.17    0.17          0     0      6     4      6     7
    1.67    2.17
  -17.00 0 -5.25 0
    0.00    0.00        0.00   0.36   0.00  0.00   0.00  0.00
   -0.36    0.00          0     4      6     6      7     0
    8.85   -1.67

  -28.85 0 -1.58 0
    0.00    0.00
    0.08    0.17        0.13   0.00   0.00  0.00   0.68  0.93
   -1.11    3.90          6     7      0     0      0     0
   -1.66 0  0.30 6
    0.00    6.00        0.00   0.00   0.00  7.67   4.00 -13.67
   -5.67    0.00          7     0      0     0      0     0
    6.20    0.00

    0.00   0.00
      0      7
    0.00
      1
```

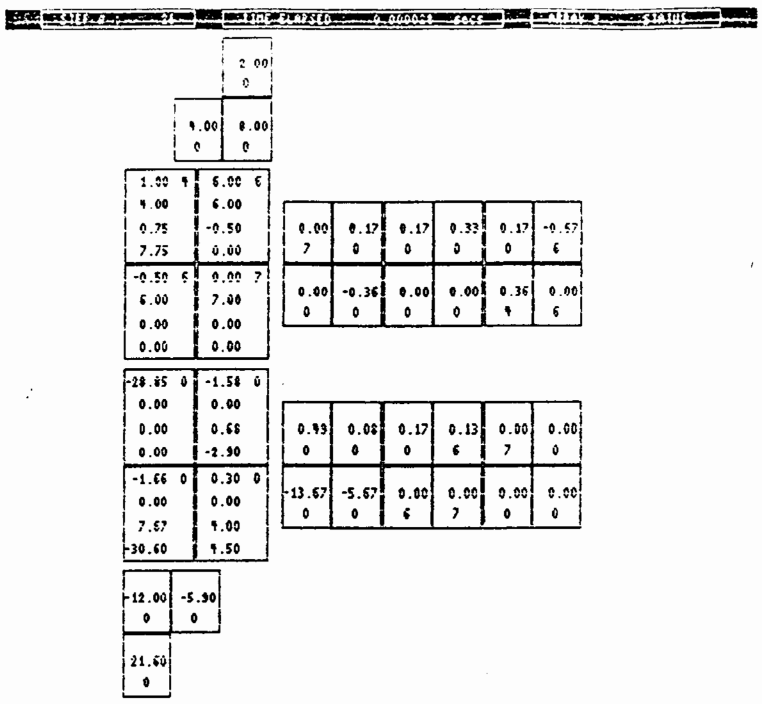
Snapshot B.3.26



Snapshot B.3.27

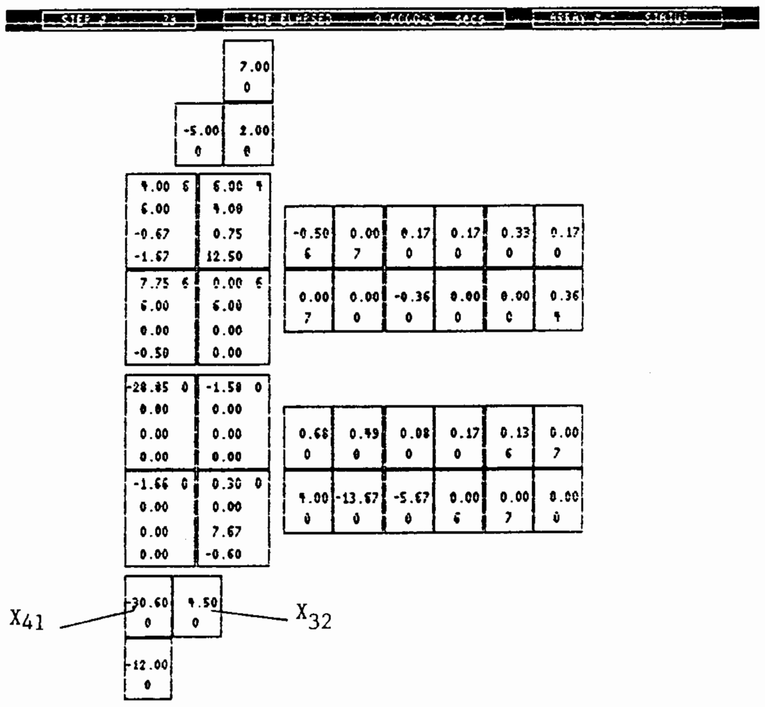


Snapshot B.3.28



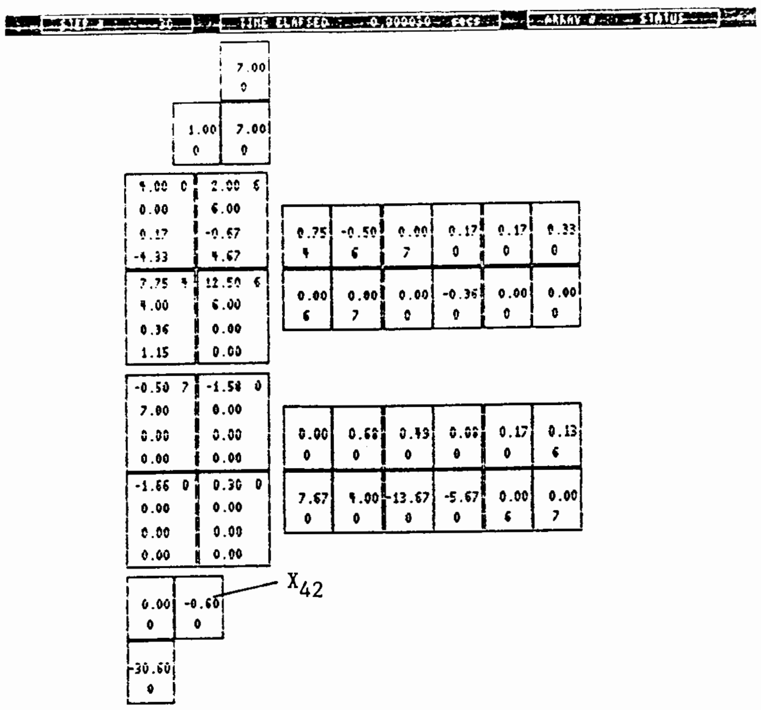
```
STEP #   23     TIME ELAPSED   0.00003  secs     ARRAY #   STATUS

                     2.00
                     0
              4.00   8.00
              0      0
   1.00  4   6.00  6
   4.00      6.00       0.00   0.17   0.17   0.33   0.17  -0.57
   0.75     -0.50       7      0      0      0      0      6
   7.75      0.00
  -0.50  6   0.00  7
   6.00      7.00       0.00  -0.36   0.00   0.00   0.36   0.00
   0.00      0.00       0      0      0      0      4      6
   0.00      0.00

 -28.85  0  -1.58  0
   0.00      0.00
   0.00      0.68       0.43   0.08   0.17   0.13   0.00   0.00
   0.00     -2.90       0      0      0      6      7      0
  -1.66  0   0.30  0
   0.00      0.00     -13.67  -5.67   0.00   0.00   0.00   0.00
   7.67      4.00       0      0      6      7      0      0
 -30.60      4.50

 -12.00  -5.90
   0       0
  21.60
   0
```

Snapshot B.3.29



```
STEP #   23     TIME ELAPSED   0.00003  secs     ARRAY #   STATUS

                     7.00
                     0
             -5.00   2.00
              0      0
   4.00  6   6.00  4
   6.00      4.00      -0.50   0.00   0.17   0.17   0.33   0.17
  -0.67      0.75       6      7      0      0      0      0
  -1.67     12.50
   7.75  6   0.00  6
   6.00      6.00       0.00   0.00  -0.36   0.00   0.00   0.36
   0.00      0.00       7      0      0      0      0      4
  -0.50      0.00

 -28.85  0  -1.58  0
   0.00      0.00
   0.00      0.00       0.68   0.49   0.08   0.17   0.13   0.00
   0.00      0.00       0      0      0      0      6      7
  -1.66  0   0.30  0
   0.00      0.00       4.00 -13.67  -5.67   0.00   0.00   0.00
   0.00      7.67       0      0      0      6      7      0
   0.00     -0.60

 -30.60   4.50
   0       0
 -12.00
   0
```

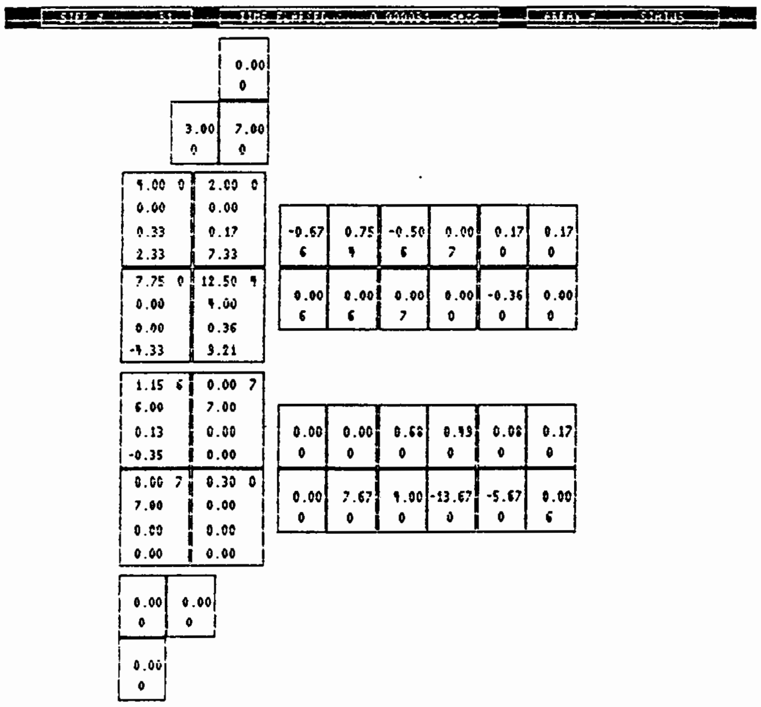
Snapshot B.3.30



```
STEP # 30   TIME ELAPSED 0.000030 secs   ARRAY #   STATUS

                 7.00
                 0
           1.00  7.00
           0     0

 4.00 0 | 2.00 6      0.75  -0.50  0.00  0.17  0.17  0.33
 0.00   | 6.00        4     6      7     0     0     0
 0.17   |-0.67
-4.33   | 4.67        0.00  0.00   0.00  -0.36 0.00  0.00
 7.75 4 |12.50 6      6     7      0     0     0     0
 4.00   | 6.00
 0.36   | 0.00
 1.15   | 0.00

-0.50 7 |-1.58 0      0.00  0.68   0.49  0.00  0.17  0.13
 7.00   | 0.00        0     0      0     0     0     6
 0.00   | 0.00
 0.00   | 0.00        7.67  4.00 -13.67 -5.67  0.00  0.00
-1.66 0 | 0.30 0      0     0      0     0     6     7
 0.00   | 0.00
 0.00   | 0.00
 0.00   | 0.00

 0.00  -0.60
 0     0
-30.60
 0
```

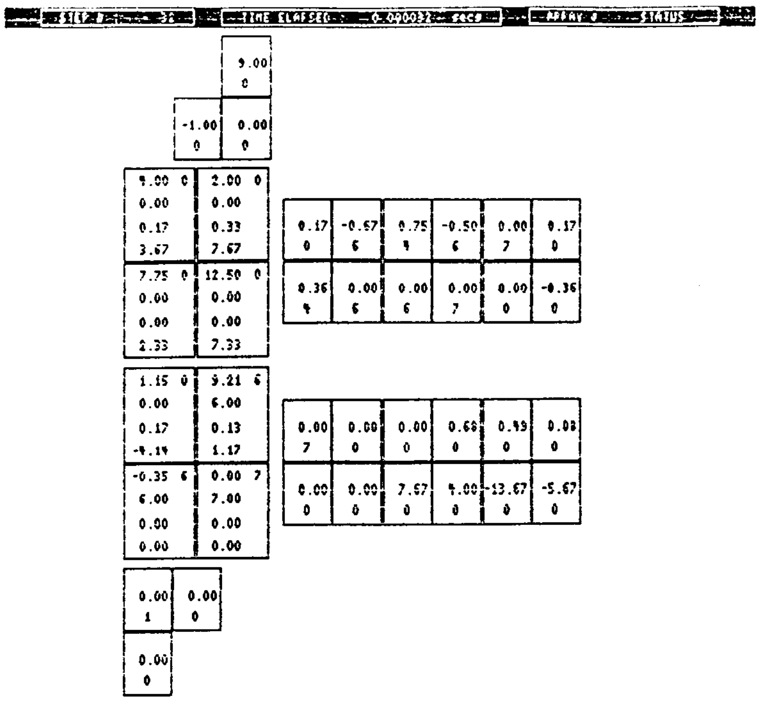
X_{42}

Snapshot B.3.31

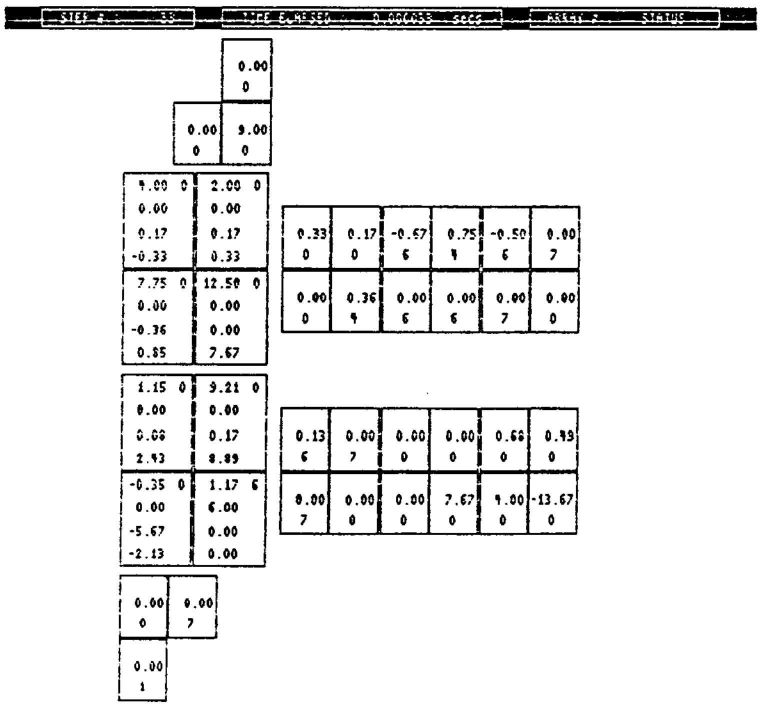


```
STEP # 33   TIME ELAPSED 0.000034 secs   ARRAY #   STATUS

                 0.00
                 0
           3.00  7.00
           0     0

 4.00 0 | 2.00 0     -0.67  0.75  -0.50  0.00  0.17  0.17
 0.00   | 0.00        6     4      6     7     0     0
 0.33   | 0.17
 2.33   | 7.33        0.00  0.00   0.00  0.00 -0.36  0.00
 7.75 0 |12.50 4      6     6      7     0     0     0
 0.00   | 4.00
 0.00   | 0.36
-4.33   | 3.21

 1.15 6 | 0.00 7      0.00  0.00   0.68  0.49  0.08  0.17
 6.00   | 7.00        0     0      0     0     0     0
 0.13   | 0.00
-0.35   | 0.00        0.00  7.67   4.00 -13.67 -5.67 0.00
 0.00 7 | 0.30 0      0     0      0     0     0     6
 7.00   | 0.00
 0.00   | 0.00
 0.00   | 0.00

 0.00  0.00
 0     0
 0.00
 0
```

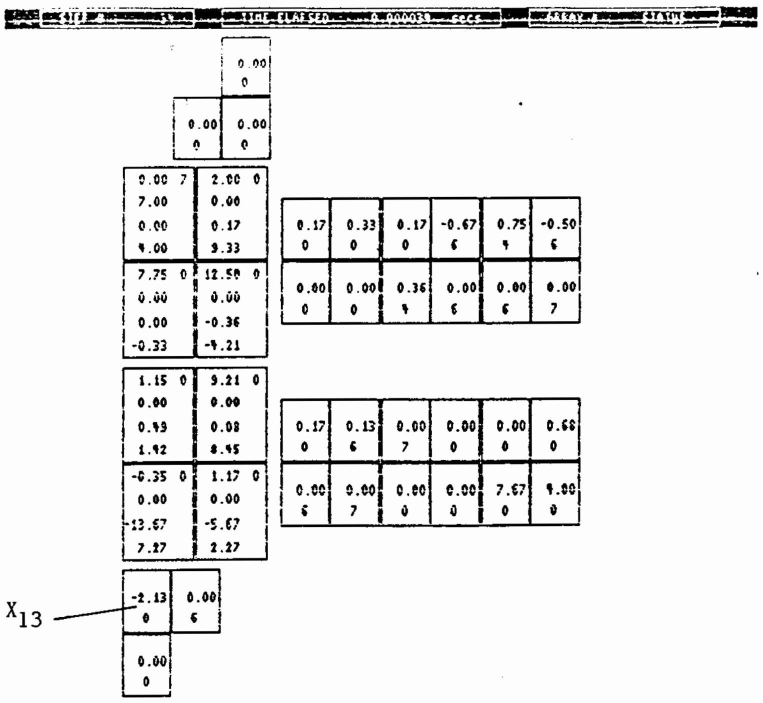
Snapshot B.3.32



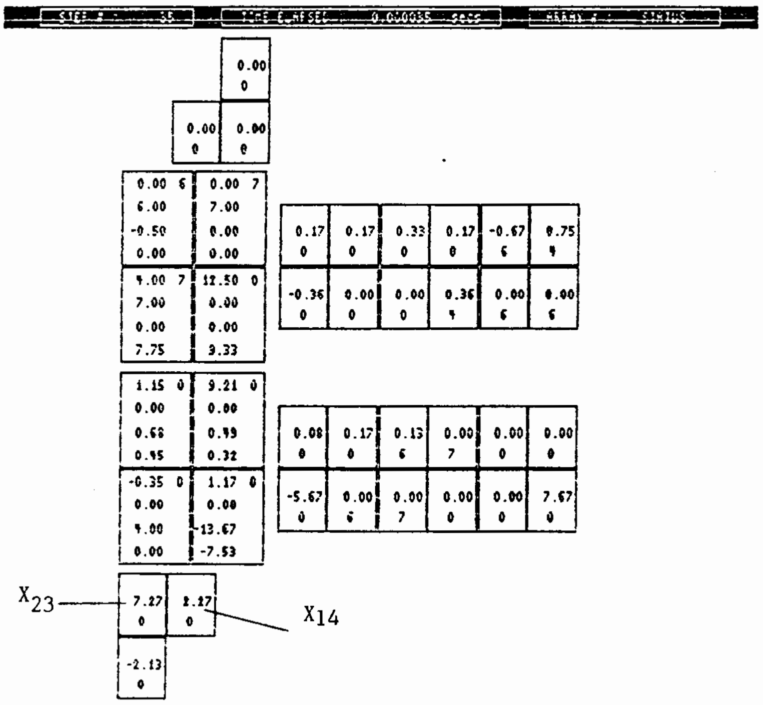
Snapshot B.3.33



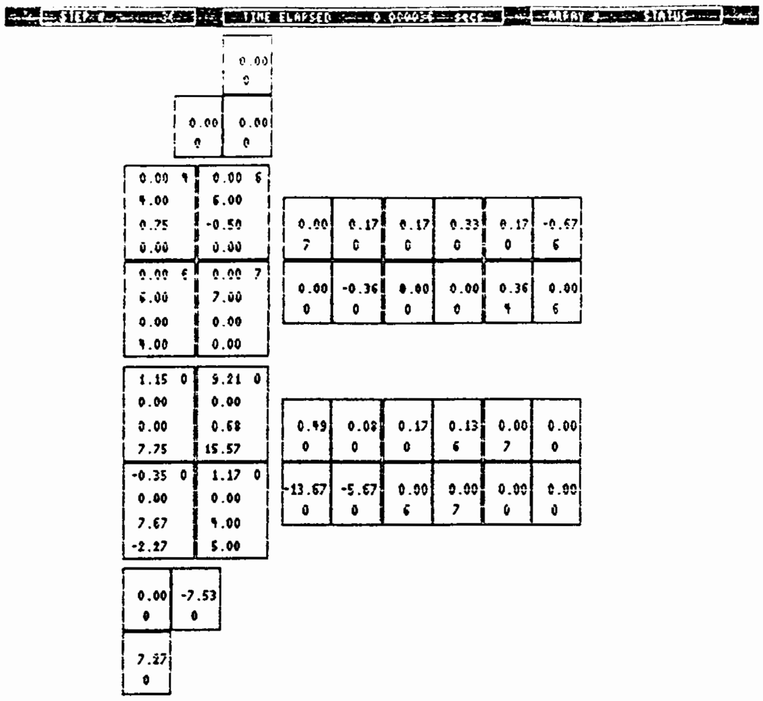
Snapshot B.3.34



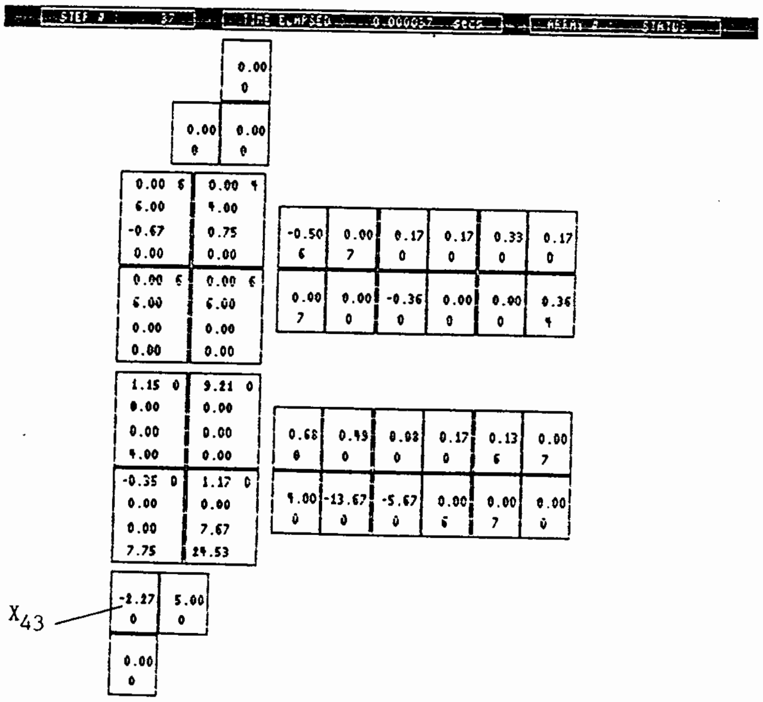
Snapshot B.3.35



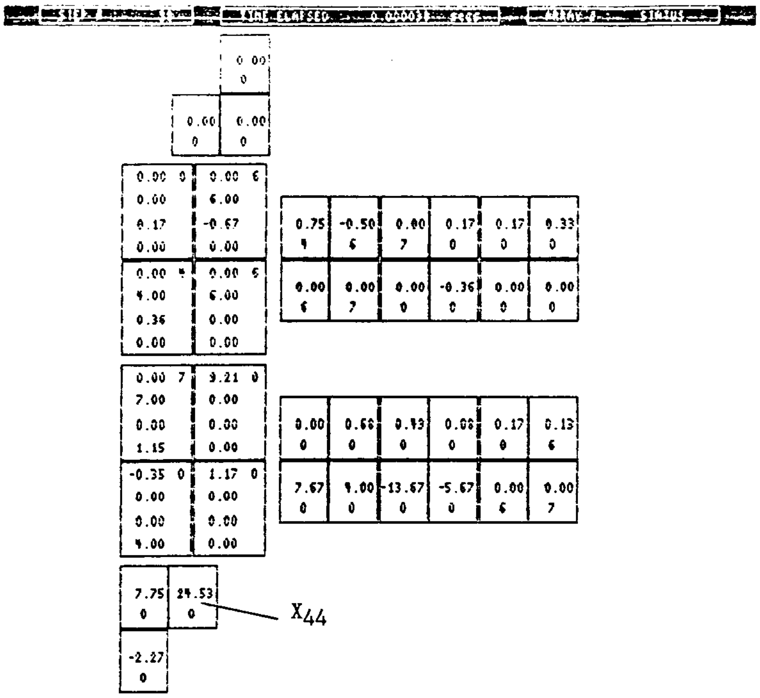
Snapshot B.3.36



Snapshot B.3.37



Snapshot B.3.38



Snapshot B.3.39

APPENDIX C

SAGS PROGRAM LISTING

SAGS was developed on an IBM Personal Computer, running the MS/DOS operating system. It was written in Turbo Pascal, a dialect of the Pascal programing language as described by Wirth and Jensen in Pascal User Manual and Report. The source code of SAGS is listed in this appendix along with a sample script file. This script file represents the simulation that produces the third series of snapshots in Appendix B. Input and control files are also included. The source code of SAGS and many sample script files are also available in ASCII format on floppy disks.

To produce an executable copy of SAGS, two software packages are needed: a copy of the DOS-based Turbo Pascal™ compiler (version 3.0) and a copy of the Turbo Graphix Toolbox™ (version 1.07), both available commercially from Borland International, Inc. Also, since SAGS is graphics based, a video card with bit-mapped graphics capabilities is needed to run the program. The included source code is written for the EGA standard; however, simple changes can be made to the program so that it will run on other PC graphics standards. Entry points for these modifications are fully documented in the source code to ease that task.

Because computer graphics and simulations are floating-point intensive applications, the use of a numeric coprocessor is highly recommended. For SAGS to take advantage of the numeric coprocessor, it must be compiled using a version of the compiler that support 8087 floating point math.

```
(***************************************************************************
 *                                                                         *
 *  SAGS is a Systolic Array Graphical Simulator program for a recon-     *
 *  figurable set of arrays of processors. The Max number of arrays is 15. *
 *  This is because of the limitation of Turbo Graphix, not of SAGS.      *
 *  This module is the main module.                                       *
 *                                                                         *
 ***************************************************************************)

program SAGS;

{$I c:\bin\tbl\tbgraphx\typedef.sys}   {include the graphics system code}
{$I c:\bin\tbl\tbgraphx\graphix.sys}
{$I c:\bin\tbl\tbgraphx\kernel.sys}
{$I c:\bin\tbl\tbgraphx\windows.sys}
{$I typedef.pas}                         {include others of SAGS modules}
{$I initglbl.pas}
{$I initcell.pas}
{$I initsqre.pas}
{$I initrngl.pas}
{$I initrng2.pas}
{$I initrng3.pas}
{$I initrng4.pas}
{$I writext.pas}
{$I drwsqre.pas}
{$I drwtrngl.pas}
{$I drwtrng2.pas}
{$I drwtrng3.pas}
{$I drwtrng4.pas}
{$I drwstat.pas}
{$I drwsystm.pas}
{$I xcolor.pas}
{$I dpmode.pas}
{$I swchwind.pas}
{$I writscrp.pas}
{$I promptus.pas}
{$I seeknxtw.pas}
{$I statemnt.pas}
{$I getsystm.pas}
{$I sidetrav.pas}
{$I lnkioflw.pas}
{$I getioflw.pas}
{$I lnkdtflw.pas}
{$I getdtflw.pas}
{$I readscrp.pas}
{$I pecodes.pas}
{$I updatear.pas}
{$I snglstep.pas}
{$I multstep.pas}
```

```
begin

  PromptUser;                           {gets script file name}
  if not ReadScript then                {reads in the script file and build}
     begin                              {system's internal structures}
       close(ScriptFile);
       writeln('!! SAGS aborted !!');
       exit;
     end
  else close(ScriptFile);

  InitGraphic;                          {init. the graphix system and screen}
  SetAspect(1);                         {sets aspect ratio for true circle}
  DefineWorld(FirstWorld,               {defines the shared world}
              0,0,WrldCoordXY,
              WrldCoordXY);
  DefineWorld(StatusWorld,              {defines the shared world}
              0,0,StatWorldX,
              StatWorldY);
  Foreground:=DefltColor;               {establishes system default}
  SetForegroundColor(Foreground);       {drawing color}
  SetBreakOff;                          {don't error when window edge hit}
  SetMessageOff;
  DrawSystem(CurrntPtr);

  repeat
    read(Kbd,Ch);                       {read the keystroke}
    if (Ch=#27) and                     {one more char ?}
       keypressed then
       read(Kbd,Ch);
    with CurrntPtr^ do
      case Ch of
         #13 : MultiStepsExec;          {RETURN ? multi steps execution}
                                        {until a key (any key) is pressed}
         #27 : ;                        {ESC ? waits for end of current loop}
         #32 : SingleStepExec           {SPACE ? single step execution}
                     (IOPtr);
         #59 : ChangeColor(-1);         {F1 ? changes to last drawing color..}
         #60 : ChangeColor(1);          {F2 ? changes to next drawing color..}
         #61 : HardCopy(False,1);       {F3 ? prints the screen image}
         #62 : WriteScriptFile;         {F4 ? writes updated script file}
         #72 : begin                    {up arrow ?}
               MoveVer(-2,TRUE);        {then moves current window and..}
               StoreWindow(Number);     {stores it with new position}
               HiY:=Y1RefGlb;
               end;
         #75 : begin                    {left arrow ?}
               MoveHor(-1,TRUE);
               StoreWindow(Number);
               HiX:=X1RefGlb;
               end;
```

```
          #77 : begin                     {right arrow ?}
                MoveHor(1,TRUE);
                StoreWindow(Number);
                HiX:=X1RefGlb;
                end;
          #80 : begin                     {down arrow ?}
                MoveVer(2,TRUE);
                StoreWindow(Number);
                HiY:=Y1RefGlb;
                end;
          #73 : SwitchWindow              {PgUp ?}
                  (CurrntPtr,0);
          #81 : SwitchWindow              {PgDn ?}
                  (CurrntPtr,1);
          #82 : ChangeDisplayMode         {Ins ?}
                  (CurrntPtr);
          else begin                      {for any other keys..}
                 sound(500);              {screams at 1000 Hertz}
                 delay(300);              {for 3 tenths of a second}
                 nosound;                 {then shuts up}
               end;
       end;
   until Ch=#27;                          {ESC char exits program}
   SetForegroundColor(0);                 {sets foreground color to black}
                                          {before exits}
   LeaveGraphic;                          {gracefully shuts down graphix system}
   while IOPtr<>NIL do                    {and the IO system by..}
     with IOPtr^ do
       begin
       case IO of
         INPUT: if Active then            {closing any active input file,}
                Close(FileVar);
         OUTPUT: begin
                 Flush(FileVar);          {and flush internal disk buffers..}
                 Close(FileVar);          {of any output files and closes them}
                 end;
         end;
       IOPtr:=NextIO;
       end;

 end.
```

```
(***************************************************************************
 *                                                                         *
 *  This is the header file of SAGS. It contains all global definitions and *
 *  declarations of constants, types and variables. All of SAGS data struc-  *
 *  tures are explained here. Be sure to include this file at compile time.  *
 *                                                                         *
 ***************************************************************************)

const

     TimeUnit = 0.000001;              {execution time for each step}
   DefltColor = 13;                    {default drawing color value}
     TextSize = 8;                     {max no. of char displayed in PE}
   FirstWorld = 1;                     {world shared by all windows}
  StatusWorld = 2;                     {world used by status box}
 MaxArraySize = 5;                     {max no. of PEs/array side allowable}
  MaxSequence = 5;                     {max no. of procedures in a script}
  MaxFileName = 64;                    {max no. of filenames}
       MaxStr = 10;                    {max no. types of script statements}
      MaxWord = 12;                    {max length of statement}
      MaxLine = 45;                    {max length of error message}
     MaxError = 16;                    {max no. error message}
       MaxBox = 4;                     {max no. boxes in status window}
      MaxRegs = 4;                     {max no. registers of one type in PE}
     MaxCodes = 12;                    {max no. of PE executable codes}
  MaxTxtCoord = 5;                     {max no. displayable text lines in PE}
       MaxBus = 2;                     {max no. of pair I/O bus on each side}
    CharSizeX = 4;                     {character size in pixel}
    CharSizeY = 6;                     {character size in pixel}
       Digits = 6;                     {no. of digits of value displayed}
       Deciml = 2;                     {no. of decimal places}
          Gap = 1;                     {size of gap between PEs in PIXELs}
  WrldCoordXY = 1000.0;                {default world coord.}
   StatWorldX = 79.0;                  {world coords. for status window}
   StatWorldY = 12.0;
  MaxRadRatio = 0.023;                 {1.15*2/100 ratio of twice the radius}
                                       {of the largest circle that will fit}
                                       {inside a 100x100 PIXELs window}

   StringList : array                  {list of valid script statements}
                [1..MaxStr]
                of string[MaxWord]
              = ('ARRAYSIZE',
                 'SYSTEMSPECS',
                 'INFILES',
                 'OUTFILES',
                 'SETUP',
                 'Pecodes',
                 'NorthInput',
                 'EastInput',
                 'SouthInput',
                 'WestInput');
     ErrorList : array                  {list of all possible error messages}
                 [1..MaxError]
                 of string[MaxLine]
               = ('!! Bad statement !!',
                  '!! Array size too large !!',
                  '!! Delimiter "." not found !!',
                  '!! Bad delimiter or delimiter not found !!',
                  '!! IO file name too long !!',
                  '!! Non-existing array !!',
                  '!! Bad statement in context !!',
                  '!! Statement out of sequence !!',
                  '!! Arrays are allowed to have only 4 sides !!',
                  '!! Bad type of array !!',
                  '!! Array number should be within 1 to 16 !!',
                  '!! Triangular arrays only have 3 sides !!',
                  '!! Input file not found !!',
                  '!! Invalid bus specification !!',
                  '!! Unknown display mode !!',
                  '!! Unknown PE code !!');

type

   RealPtrtype = ^real;                 {pointer to reals}
   LinkPtrType = ^LinkType;             {pointer to link info between arrays}
     IOPtrtype = ^IOtype;               {pointer to IO buffers}
  ArrayPtrtype = ^SysArraytype;         {pointer to array processors}
      FileName = string                 {file name storage}
                 [MaxFileName];
       Textype = string                 {storage for text for screen output}
                 [TextSize];
    SrcDstType = array                  {pair indicating sides of src. & dest.}
                 [1..2]                 {arrays for dataflow info}
                 of integer;
      Pointype = record                 {pair of coord. for a point}
                   X,Y                  : real;
                 end;
    Pointstype = record                 {All of PE's texts coords.}
                   X,Y                  : array
                                          [1..MaxTxtCoord]
                                          of real;
                 end;
   DisplayMode = (Full,Arrays,Buffer);  {mode of display of an array}
     PEtextype = record                 {stores default coord. for each PE's}
                                        {text in an array, essentially acts}
                                        {as a template for a particular}
                                        {display mode}
                   Mode                 : DisplayMode;
                   Lines,
                   PEsize,
                   WDSizeX,
                   WDSizeY              : integer;
```

```
              PEsizeXY,
              GapXY,
              TrueRad,
              Radius                  : real;
              TextCoord               {storage for all PEs' text coord.}
                                      {of an array}
                                      : array
                                        [1..MaxArraySize,
                                        1..MaxArraySize]
                                        of Pointstype;
            end;
   PEtype = record                    {internal PE representation, with}
                                      {all necessary registers}
              X_Reg                   : real;
              Out_Regs,
              Last_Out                : array
                                        [1..MaxRegs,
                                        1..MaxBus]
                                        of real;
              In_Regs                 {pointers to X_out registers in}
                                      {neighboring PE cells}
                                      : array
                                        [1..MaxRegs,
                                        1..MaxBus]
                                        of RealPtrtype;
              Regs_Txt                {Regs_Txt[1] is for TAG}
                                      {Regs_Txt[2] is for X,}
                                      {........[3]........Vout}
                                      {........[4]........Mout}
                                      {........[5]........Xout}
                                      : array
                                        [1..MaxTxtCoord]
                                        of Textype;
              C124,                   {control codes registers}
              C3                      : integer;
              TAG,                    {for display purpose only}
              Code                    {holds PE's execution code number}
                                      : byte;
            end;
StatusBox = record
              Xhi,Yhi,                {coord. of box and texts within box}
              Xlo,Ylo,
              Xdgt,Xtxt,
              Ytxt                    : real;
              Txt,Dgt                 : string[15];
            end;
   LinkType = record              {storage for dataflow info to}
                                  {other arrays}
             Sides,               {from which side of src. array to}
                                  {which side of dest. array}
             ArNums               : SrcDstType;
             ArPtrs               : array
                                    [1..2]
                                    of ArrayPtrType;
             LnkStart,
             LnkStop              : integer;
             NxtLink              : LinkPtrType;
            end;
    IOflag = (INPUT,OUTPUT);
    IOtype = record               {IO link to and from host, that is}
                                  {to and from external data files}
             Name                 : FileName;
             ArNum,               {links with which array}
             Side,                {to which of its side}
             Bus,                 {and which bus}
             IOStart              {step to start feeding data}
                                  : integer;
             Filevar              : text;
             ArPtr                : ArrayPtrtype;
             NextIO               : IOPtrtype;
             Active               {is it still feeding data or not}
                                  : boolean;
             case
             IO                   : IOflag
             of
             INPUT                : (InRegs  : array
                                               [1..MaxArraySize]
                                               of real
                                    );
             OUTPUT               : (OutRegs : array
                                               [1..MaxArraySize]
                                               of RealPtrtype
                                    );
            end;
TypeOfArray = (Square,            {square array of PE's}
            Triangle1,            {upper triangular array of PEs with}
                                  {diagonal line from top left corner}
                                  {to lower right corner}
            Triangle2,            {lower triangular array of PEs with}
                                  {diagonal line from top left corner}
                                  {to lower right corner}
            Triangle3,            {upper triangular array of PEs with}
                                  {diagonal line from top right corner}
                                  {to lower left corner}
            Triangle4,            {lower triangular array of PEs with}
                                  {diagonal line from top right corner}
                                  {to lower left corner}
            Status);              {storage for each box in status band}
```

```
SysArrayType = record                 {storage for systolic array's data}
                 Number,              {including all PEs within it}
                 HiX,HiY              : integer;
                 Last,Next            : ArrayPtrType;
                 StatTxt              : Textype;
                 case ArrayType       : TypeOfArray
                 of
                   Status             : (    LoX ,
                                             LoY : integer;
                                           Boxes : array
                                                   [1..MaxBox]
                                                   of StatusBox;
                                           Steps : integer;
                                           Times : real
                                        );
                   Triangle1,
                   Triangle2,
                   Triangle3,
                   Triangle4,
                   Square             : ( DPmode : DisplayMode;
                                              PE : array
                                                   [1..MaxArraySize,
                                                   1..MaxArraySize]
                                                   of PEtype
                                        );
               end;
   ErrorType = set of 1..MaxError;

var

  Foreground ,                        {current drawing color}
   ArraySize : integer;               {size of bandwidth of dataflow}
        Zero : real;                  {value for PE's grounded input}
     ZeroPtr : RealPtrtype;
  PEtxtArray : array                  {stores all display mode's templates}
               [DisplayMode]
               of PEtextype;
  ScriptName : FileName;
  ScriptFile : text;
       IOPtr : IOPtrtype;             {always points to top of linked list}
     LinkPtr : LinkPtrType;           {always points to top of linked list}
    FixedPtr ,                        {always points to top of linked list}
   CurrntPtr ,                        {always points to the current array}
     StatPtr : ArrayPtrtype;          {always points to status window}
    ErrorSet : ErrorType;             {stores error type values}
          Ch : char;                  {keyboard input storage}
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure initializes all global variables needed for drawing      *
 *  an array. Depending on the specified array size, it will find a         *
 *  suitable window size and world coordinates for the array. It also       *
 *  computes an array of coordinates for PEs' text.                         *
 *  This procedure is very machine-dependent, i.e. graphics card specific,  *
 *  and is used only once after the  script file is read in.                *
 *                                                                          *
 ****************************************************************************)

procedure InitGlbStorage;

var          I ,
          Xcnt ,
          Ycnt : integer;
         Templ ,
         Temp2 ,
         Temp3 ,
         Temp4 : real;

begin

  with PEtxtArray[Full] do
    begin
    Mode := Full;
    Lines := MaxTxtCoord;
    PEsize := CharSizeY*(Lines-1)+22;  {value 22 is for EGA; 2 is for CGA}
    WDSizeY:=(PEsize+Gap)              {computes window dimensions for}
             *ArraySize                {a particular array size}
             +Gap;
    WDSizeX:=round(WDSizeY
             /(8*AspectFactor));
    GapXY:=Gap*WrldCoordXY             {compute value of gap in w.c.}
           /(ArraySize*PEsize
           +(ArraySize+1)*Gap);
    PEsizeXY:=(WrldCoordXY             {compute value of PE size in w.c.}
              -GapXY*(ArraySize+1))
              /ArraySize;
    TrueRad:=PEsizeXY/2;               {compute round PE's radius in w.c.}
    Radius:=MaxRadRatio*PEsize/2;
    Templ:=PEsizeXY+GapXY;
    Temp2:=PEsizeXY-GapXY;
    Temp3:=Temp2/4;
    Temp4:=Temp3/2;
    for Xcnt:=1 to ArraySize do
    for Ycnt:=1 to ArraySize do
    with TextCoord[Xcnt,Ycnt] do       {compute text coord. for array of PEs}
      begin
        for I:=2 to Lines
            do begin
            X[I]:=Templ*Ycnt-Temp2;
          Y[I]:=Templ*Xcnt
               -PEsizeXY
               -Temp4
               +Temp3*(I-1);
          end;
      X[1]:=Templ*Ycnt                    {text coord. for TAG bit}
               -1.5*Temp4;
      Y[1]:=Y[2];
    end;
  end;

with PEtxtArray[Arrays] do
  begin
  Mode := Arrays;
  Lines := 2;
  PEsize := CharSizeX*Digits+10;
  WDSizeX:=trunc(((PEsize+Gap)          {computes window dimensions for}
           *ArraySize+1)/8+1);          {arrays displays}
  WDSizeY:=trunc(WDSizeX*8
                *AspectFactor
                +1);
  GapXY:=1.3*WrldCoordXY                {compute value of gap in w.c.}
         /(ArraySize*PEsize
         +(ArraySize+1)*Gap);
  PEsizeXY:=(WrldCoordXY                {compute value of PE size in w.c.}
            -GapXY*ArraySize)
            /ArraySize;
  TrueRad:=PEsizeXY/2;                  {compute round PE's radius in w.c.}
  Radius:=MaxRadRatio*PEsize/4.1;
  Templ:=PEsizeXY+GapXY;
  Temp2:=PEsizeXY-GapXY;
  Temp3:=PEsizeXY/5;
  for Xcnt:=1 to ArraySize do
  for Ycnt:=1 to ArraySize do
  with TextCoord[Xcnt,Ycnt] do          {compute text coord. for array of PEs}
    begin
      X[2]:=Templ*Ycnt-Temp2;
      Y[2]:=Templ*Xcnt
           -PEsizeXY
           +Temp3*2;
      X[1]:=X[2]+Temp3*2;               {text coord. for TAG bit}
      Y[1]:=Y[2]+Temp3*1.8;
    end;
  end;

with PEtxtArray[Buffer] do
  begin
  Mode := Buffer;
  Lines := 2;
  PEsize := CharSizeX*Digits+10;
  WDSizeX:=trunc(((PEsize+Gap)          {computes window dimensions for}
           *ArraySize+1)/8+1);          {arrays displays}
  WDSizeY:=trunc(WDSizeX*8
                 *AspectFactor
                 +1);
  GapXY:=1.3*WrldCoordXY               {compute value of gap in w.c.}
         /(ArraySize*PEsize
         +(ArraySize+1)*Gap);
  PEsizeXY:=(WrldCoordXY               {compute value of PE size in w.c.}
            -GapXY*ArraySize)
            /ArraySize;
  TrueRad:=PEsizeXY/2;                 {compute round PE's radius in w.c.}
  Radius:=MaxRadRatio*PEsize/3.87;
  Temp1:=PEsizeXY+GapXY;
  Temp2:=PEsizeXY-GapXY;
  Temp3:=PEsizeXY/5;
  for Xcnt:=1 to ArraySize do
  for Ycnt:=1 to ArraySize do
  with TextCoord[Xcnt,Ycnt] do         {compute text coord. for array of PEs}
    begin
      X[2]:=Temp1*Ycnt-Temp2;
      Y[2]:=Temp1*Xcnt
            -PEsizeXY
            +Temp3*2;
      X[1]:=X[2]+Temp3*2;              {text coord. for TAG bit}
      Y[1]:=Y[2]+Temp3*1.8;
    end;
  end;

New(StatPtr);
with StatPtr^ do
  begin
  ArrayType:=Status;
  Number:=MaxWindowsGlb;               {STATUS panel window number is 16}
  HiX:=2;HiY:=0;                       {for IBM CGA : HiX=0, HiY=0}
  LoX:=77;LoY:=12;                     {for IBM CGA : LoX=79, LoY=12}
  StatTxt:='STATUS';
  Steps:=0;
  Times:=0.0;
  for Xcnt:=1 to 4 do
      with Boxes[Xcnt] do
           begin
           Yhi:=2.0;
           Ylo:=10.0;
           Ytxt:=5.0;
           end;
  Boxes[1].Xhi:=4.0;
  Boxes[1].Xlo:=20.0;
  Boxes[1].Xtxt:=6.0;
  Boxes[1].Xdgt:=15.0;
  Boxes[1].Txt:='STEP # :';
  Boxes[2].Xhi:=23.0;
  Boxes[2].Xlo:=52.0;
  Boxes[2].Xtxt:=25.0;
```

```
    Boxes[2].Xdgt:=38.0;
    Boxes[2].Txt:='TIME ELAPSED :';
    Boxes[4].Xtxt:=47;
    Boxes[4].Txt:='secs';
    Boxes[3].Xhi:=55.0;
    Boxes[3].Xlo:=75.0;
    Boxes[3].Xtxt:=57.0;
    Boxes[3].Xdgt:=67.0;
    Boxes[3].Txt:='ARRAY # :';
    end;
  Zero := 0.0;
  ZeroPtr:=Addr(Zero);                   {get address of ground value}

end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This procedure initializes all registers and texts storages of a cell to *
 *  zero.                                                                    *
 *                                                                           *
 *****************************************************************************)

procedure InitializeCell
    (var Cell : PEtype);

var       I,J : integer;

begin

  with Cell do                           {with this cell, Thou shall}
    begin                                {initialize all of its registers}
    X_Reg:=0.0;                          {on all buses to zero..}
    for I:=1 to MaxRegs do
    for J:=1 to MaxBus do
      begin
      Out_Regs[I,J]:=0.0;
      Last_Out[I,J]:=0.0;
      end;
    C124:=0;
    C3:=0;
    TAG:=0;
    Regs_Txt[1]:='0';                    {..and all of its texts storage to}
    for I:=2 to MaxTxtCoord do           {strings '  0.00' or '0'}
        Regs_Txt[I]:='  0.00';
    end;

end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This procedure initializes a newly allocated square array specified in   *
 *  the script. It is called by the procedure :                              *
 *     - GetSystemSpecs.                                                     *
 *                                                                           *
 *****************************************************************************)

procedure InitializeSquare
         (Ptr : ArrayPtrType);

var     I,X,Y : integer;

begin

  with Ptr^ do
    for X:=1 to ArraySize do            {for each PEs in square array}
    for Y:=1 to ArraySize do
        begin
        InitializeCell(PE[X,Y]);        {..init. all of its registers on all}
                                        {buses and all of its texts storages}
        with PE[X,Y] do
          for I:=1 to MaxBus do         {with all buses, link PEs together}
            begin                       {as follow..}
            if X=1 then                 {if PE is on north border of array..}
              In_Regs[1,I]:=ZeroPtr     {its north input is grounded for now}
            else In_Regs[1,I]:=         {else its north input is from its}
                 Addr(PE[X-1,Y].        {north neighbor}
                 Last_Out[3,I]);
            if Y=ArraySize then         {and so on for the east border..}
              In_Regs[2,I]:=ZeroPtr
            else In_Regs[2,I]:=
                 Addr(PE[X,Y+1].
                 Last_Out[4,I]);
            if X=ArraySize then         {except this time we have the south}
              In_Regs[3,I]:=ZeroPtr     {border and..}
            else In_Regs[3,I]:=
                Addr(PE[X+1,Y].
                Last_Out[1,I]);
            if Y=1 then                 {the west border to take care of.}
              In_Regs[4,I]:=ZeroPtr
            else In_Regs[4,I]:=
                 Addr(PE[X,Y-1].
                 Last_Out[2,I]);
            end;
        end;

end;
```

```
(***************************************************************************
 *                                                                         *
 *  This procedure initializes a newly allocated type 1 triangle array     *
 *  specified by the script. It is called by the procedure :               *
 *    - GetSystemSpecs.                                                    *
 *                                                                         *
 ***************************************************************************)

procedure InitializeTriangle1
        (Ptr : ArrayPtrType);

var     I,X,Y : integer;

begin

  with Ptr^ do
    for X:=1 to ArraySize do             {for each PE in triangle array..}
    for Y:=X to ArraySize do
      begin
      InitializeCell(PE[X,Y]);           {..init. all of its registers on all}
                                         {buses and all of its texts storages}
      with PE[X,Y] do
        for I:=1 to MaxBus do            {then for all existing buses..}
          begin
          if X=1 then                    {if PE is on north border of array..}
            In_Regs[1,I]:=ZeroPtr        {its north input is grounded for now}
          else In_Regs[1,I]:=            {else its north input is from its}
               Addr(PE[X-1,Y].           {north neighbor}
               Last_Out[3,I]);
          if Y=ArraySize then            {east border is grounded if PE's on}
            In_Regs[2,I]:=ZeroPtr        {east boundary,..}
          else In_Regs[2,I]:=            {else it's connected to the east}
               Addr(PE[X,Y+1].           {neighbor}
               Last_Out[4,I]);
          if X=Y then                    {south and west inputs are grounded}
             begin                       {if PE's on the diagonal boundary..}
             In_Regs[3,I]:=ZeroPtr;
             In_Regs[4,I]:=ZeroPtr;
             end
          else begin                     {else they are connected to the south}
               In_Regs[3,I]:=            {and west neighbors}
                 Addr(PE[X+1,Y].
                 Last_Out[1,I]);
               In_Regs[4,I]:=
                 Addr(PE[X,Y-1].
                 Last_Out[2,I]);
               end;
          end;
      end;

end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure initializes a newly allocated type 2 triangle array      *
 *  specified by the script. It is called by the procedure :                *
 *     - GetSystemSpecs.                                                    *
 *                                                                          *
 ****************************************************************************)

procedure InitializeTriangle2
         (Ptr : ArrayPtrType);

var     I,X,Y : integer;

begin

  with Ptr^ do
    for X:=1 to ArraySize do              {for each PE in triangle array..}
    for Y:=1 to X do
      begin
      InitializeCell(PE[X,Y]);            {..init. all of its registers on all}
                                          {buses and all of its texts storages}
      with PE[X,Y] do
        for I:=1 to MaxBus do             {then, for all existing buses..}
          begin
          if X=Y then                     {if PE's on the diagonal boundary}
             begin                        {then its north and east inputs}
             In_Regs[1,I]:=ZeroPtr;       {are grounded for now..}
             In_Regs[2,I]:=ZeroPtr;
             end
          else begin                      {else they are connected to PE's}
               In_Regs[1,I]:=             {north and east neighbors}
                 Addr(PE[X-1,Y].
                 Last_Out[3,I]);
               In_Regs[2,I]:=
                 Addr(PE[X,Y+1].
                 Last_Out[4,I]);
               end;
          if X=ArraySize then             {south input is grounded if PE's}
            In_Regs[3,I]:=ZeroPtr         {at the bottom of array..}
          else In_Regs[3,I]:=             {else it's connected to PE's south}
               Addr(PE[X+1,Y].            {neighbor}
               Last_Out[1,I]);
          if Y=1 then                     {west input is grounded if PE's}
            In_Regs[4,I]:=ZeroPtr         {at the west boundary..}
          else In_Regs[4,I]:=             {else it's connected to the west cell}
               Addr(PE[X,Y-1].
               Last_Out[2,I]);
          end;
      end;

end;
```

```
(*************************************************************************
 *                                                                      *
 *  This procedure initializes a newly allocated type 3 triangle array  *
 *  specified by the script. It is called by the procedure :            *
 *     - GetSystemSpecs.                                                *
 *                                                                      *
 *************************************************************************)

procedure InitializeTriangle3
         (Ptr : ArrayPtrType);

var   I,J,X,Y : integer;

begin

  with Ptr^ do
    for X:=1 to ArraySize do              {for each PE in this triangular array}
      begin
      I:=ArraySize+1-X;
      for Y:=1 to I do
        begin
        InitializeCell(PE[X,Y]);          {..init. all of its registers on all}
        with PE[X,Y] do                   {buses and all of its texts storages}
        for J:=1 to MaxBus do             {then, for all existing buses..}
          begin
          if X=1 then                     {if PE's on the north border}
             In_Regs[1,J]:=ZeroPtr        {then ground its north input..}
          else In_Regs[1,J]:=             {else connect the north input}
               Addr(PE[X-1,Y].            {to the northern neighbor}
               Last_Out[3,J]);
          if Y=I then                     {if PE's on the diagonal boundary}
            begin
            In_Regs[2,J]:=ZeroPtr;        {then its east input and..}
            In_Regs[3,J]:=ZeroPtr;        {its south input is grounded for now}
            end
          else begin                      {else..}
               In_Regs[2,J]:=             {its east input is from its}
               Addr(PE[X,Y+1].            {east neighbor and..}
               Last_Out[4,J]);
               In_Regs[3,J]:=             {its south input is from its south}
               Addr(PE[X+1,Y].            {neighbor}
               Last_Out[1,J]);
               end;
          if Y=1 then                     {west input is grounded if PE's on}
             In_Regs[4,J]:=ZeroPtr        {the west boundary..}
          else In_Regs[4,J]:=             {else connect it to the west}
               Addr(PE[X,Y-1].            {neighboring PE}
               Last_Out[2,J]);
          end;
        end;
      end;
end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This procedure initializes a newly allocated type 4 triangle array       *
 *  specified by the script. It is called by the procedure :                 *
 *     - GetSystemSpecs.                                                     *
 *                                                                           *
 *****************************************************************************)

procedure InitializeTriangle4
         (Ptr : ArrayPtrType);

var   I,J,X,Y : integer;

begin

  with Ptr^ do
    for X:=1 to ArraySize do              {for each PE in this triangular array}
      begin
      I:=ArraySize+1-X;
      for Y:=I to ArraySize do
        begin
        InitializeCell(PE[X,Y]);          {..init. all of its registers on all}
        with PE[X,Y] do                   {buses and all of its texts storages}
        for J:=1 to MaxBus do             {then, for all existing buses..}
          begin
          if Y=I then                     {if PE's on the diagonal boundary}
            begin
            In_Regs[1,J]:=ZeroPtr;        {then its north input and}
            In_Regs[4,J]:=ZeroPtr;        {its west input is grounded for now}
            end
          else begin                      {else..}
               In_Regs[1,J]:=             {its north input is from its..}
               Addr(PE[X-1,Y].            {north neighbor and..}
               Last_Out[3,J]);
               In_Regs[4,J]:=             {its west input is from its west..}
               Addr(PE[X,Y-1].            {neighbor}
               Last_Out[2,J]);
               end;
          if Y=ArraySize then             {if PE's on the east border then}
             In_Regs[2,J]:=ZeroPtr        {ground its east input..}
          else In_Regs[2,J]:=             {else connect it to the eastern}
               Addr(PE[X,Y+1].            {neighboring PE.}
               Last_Out[4,J]);
          if X=ArraySize then             {if PE's on the south border then}
             In_Regs[3,J]:=ZeroPtr        {ground its south input..}
          else In_Regs[3,J]:=             {else connect it to the southern}
               Addr(PE[X+1,Y].            {neighboring PE}
               Last_Out[1,J]);
          end;
        end;
      end;
end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure writes text inside each PE of an array according to      *
 *  values of the PE's registers. It's smart enough to know the display mode *
 *  of the array and write texts accordingly.                               *
 *                                                                          *
 ****************************************************************************)

procedure WritePEtxt
         (X,Y : integer;
          Ptr : ArrayPtrtype);

var          I : integer;

begin

  with Ptr^ do
    with PEtxtArray[DPmode] do             {depending on array's display mode..}
      for I:=1 to Lines do                 {writes all displayable registers}
        DrawTextW                          {values}
        (TextCoord[X,Y].X[I],
         TextCoord[X,Y].Y[I],
         1,PE[X,Y].Regs_Txt[I]);

end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure define a window, give it a world coordinate system,      *
 *  and then depending on array's display mode, will draw a square systolic *
 *  array inside the window. This window will overlap on top of all         *
 *  previously defined windows.                                             *
 *                                                                          *
 ****************************************************************************)

procedure DrwSquare
    (WorldNum : integer;
          Ptr : ArrayPtrtype);

var       X,Y : integer;
       TempXY : real;

begin

  with Ptr^ do
    with PEtxtArray[DPmode] do
      begin
      DefineWindow(Number,                    {define window where drawing}
                   HiX,HiY,                   {will take place}
                   HiX+WDSizeX,
                   HiY+WDSizeY);
      SelectWorld(WorldNum);                  {select world for array window}
      SelectWindow(Number);                   {select the window}
      SetBackground(0);                       {give it a (black) background..}
                                              {else it won't overlap others}
      TempXY:=PEsizeXY+GapXY;
      for X:=1 to ArraySize do
      for Y:=1 to ArraySize do
          begin
          if (Y=X) and                        {if PE's boundary type then draw}
             (DPmode=Arrays)
             then DrawCircle                  {it as a circle. Else..}
                  (TempXY*Y-TrueRad,
                   TempXY*X-TrueRad,
                   Radius)
          else
            DrawSquare                        {..draw PE as a square}
            (TempXY*Y-PEsizeXY,
             TempXY*X-PEsizeXY,
             TempXY*Y,TempXY*X,
             false);
          WritePEtxt(X,Y,Ptr);
          end;
      end;

end;
```

```
(*************************************************************************
 *                                                                       *
 *  This procedure define a window, give it a world coordinate system,   *
 *  and then depending on array's display mode, will draw a type 1 triangu-  *
 *  lar systolic array inside the window. This window will overlap on top of *
 *  all previously defined windows.                                      *
 *                                                                       *
 *************************************************************************)

procedure DrwTriangle1
    (WorldNum : integer;
          Ptr : ArrayPtrtype);

var        X,Y : integer;
       TempXY : real;

begin

  with Ptr^ do
    with PEtxtArray[DPmode] do
      begin
      DefineWindow(Number,                   {define window where drawing}
                   HiX,HiY,                  {will take place}
                   HiX+WDSizeX,
                   HiY+WDSizeY);
      SelectWorld(WorldNum);                 {select world for array window}
      SelectWindow(Number);                  {select the window}
      SetBackground(0);                      {give it a (black) background..}
                                             {else it won't overlap others}
      TempXY:=PEsizeXY+GapXY;
      for X:=1 to ArraySize do
      for Y:=X to ArraySize do
          begin
          if (Y=X) and                       {if PE's boundary type and display}
             (DPmode=Arrays)
             then DrawCircle                 {mode is Arrays then draw it as a}
                  (TempXY*Y-TrueRad,         {circle.}
                   TempXY*X-TrueRad,
                   Radius)
          else
            DrawSquare                       {.. Else draw PE as a square}
            (TempXY*Y-PEsizeXY,
             TempXY*X-PEsizeXY,
             TempXY*Y,TempXY*X,
             false);
          WritePEtxt(X,Y,Ptr);
          end;
      end;

end;
```

```
(**************************************************************************
 *                                                                        *
 *  This procedure define a window, give it a world coordinate system,    *
 *  and then depending on array's display mode, will draw a type 2 triangu-  *
 *  lar systolic array inside the window. This window will overlap on top of *
 *  all previously defined windows.                                       *
 *                                                                        *
 **************************************************************************)

procedure DrwTriangle2
    (WorldNum : integer;
          Ptr : ArrayPtrtype);

var       X,Y : integer;
       TempXY : real;

begin

  with Ptr^ do
    with PEtxtArray[DPmode] do
      begin
      DefineWindow(Number,                   {define window where drawing}
                   HiX,HiY,                  {will take place}
                   HiX+WDSizeX,
                   HiY+WDSizeY);
      SelectWorld(WorldNum);                 {select world for array window}
      SelectWindow(Number);                  {select the window}
      SetBackground(0);                      {give it a (black) background..}
                                             {else it won't overlap others}
      TempXY:=PEsizeXY+GapXY;
      for X:=1 to ArraySize do
      for Y:=1 to X do
          begin
          if (Y=X) and                       {if PE's boundary type and display}
             (DPmode=Arrays)
             then DrawCircle                 {mode is Arrays then draw it as a}
                  (TempXY*Y-TrueRad,         {circle.}
                   TempXY*X-TrueRad,
                   Radius)
          else
            DrawSquare                       {.. Else draw PE as a square}
            (TempXY*Y-PEsizeXY,
             TempXY*X-PEsizeXY,
             TempXY*Y,TempXY*X,
             false);
          WritePEtxt(X,Y,Ptr);
          end;
      end;

end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This procedure define a window, give it a world coordinate system,       *
 *  and then depending on array's display mode, will draw a type 3 triangu-  *
 *  lar systolic array inside the window. This window will overlap on top of *
 *  all previously defined windows.                                          *
 *                                                                           *
 *****************************************************************************)

procedure DrwTriangle3
    (WorldNum : integer;
          Ptr : ArrayPtrtype);

var     I,X,Y : integer;
       TempXY : real;

begin

  with Ptr^ do
    with PEtxtArray[DPmode] do
      begin
      DefineWindow(Number,                  {define window where drawing}
                   HiX,HiY,                 {will take place}
                   HiX+WDSizeX,
                   HiY+WDSizeY);
      SelectWorld(WorldNum);                {select world for array window}
      SelectWindow(Number);                 {select the window}
      SetBackground(0);                     {give it a (black) background..}
                                            {else it won't overlap others}
      TempXY:=PEsizeXY+GapXY;
      for X:=1 to ArraySize do
        begin
        I:=ArraySize-X+1;
        for Y:=1 to I do
          begin
          if (Y=I) and                      {if PE's boundary type and display}
             (DPmode=Arrays)
             then DrawCircle                {mode is Arrays then draw it as}
                  (TempXY*Y-TrueRad,        {a circle.}
                   TempXY*X-TrueRad,
                   Radius)
          else
            DrawSquare                      {.. Else draw PE as a square}
            (TempXY*Y-PEsizeXY,
             TempXY*X-PEsizeXY,
             TempXY*Y,TempXY*X,
             false);
          WritePEtxt(X,Y,Ptr);
          end;
        end;
      end;
end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure define a window, give it a world coordinate system,      *
 *  and then depending on array's display mode, will draw a type 4 triangu- *
 *  lar systolic array inside the window. This window will overlap on top of *
 *  all previously defined windows.                                         *
 *                                                                          *
 ****************************************************************************)

procedure DrwTriangle4
    (WorldNum : integer;
          Ptr : ArrayPtrtype);

var     I,X,Y : integer;
       TempXY : real;

begin

  with Ptr^ do
    with PEtxtArray[DPmode] do
      begin
      DefineWindow(Number,                  {define window where drawing}
                   HiX,HiY,                 {will take place}
                   HiX+WDSizeX,
                   HiY+WDSizeY);
      SelectWorld(WorldNum);                {select world for array window}
      SelectWindow(Number);                 {select the window}
      SetBackground(0);                     {give it a (black) background..}
                                            {else it won't overlap others}
      TempXY:=PEsizeXY+GapXY;
      for X:=1 to ArraySize do
        begin
        I:=ArraySize-X+1;
        for Y:=I to ArraySize do
          begin
          if (Y=I) and                      {if PE's boundary type and display}
             (DPmode=Arrays)
             then DrawCircle                {mode is Arrays then draw it as}
                  (TempXY*Y-TrueRad,        {a circle.}
                   TempXY*X-TrueRad,
                   Radius)
          else
            DrawSquare                      {.. Else draw PE as a square}
            (TempXY*Y-PEsizeXY,
             TempXY*X-PEsizeXY,
             TempXY*Y,TempXY*X,
             false);
          WritePEtxt(X,Y,Ptr);
          end;
        end;
      end;
end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure will draw the status window at the default location and  *
 *  writes initial text within its boxes. It is used only once.             *
 *                                                                          *
 ****************************************************************************)

procedure DrwStatusWindow
    (WorldNum : integer;
          Ptr : ArrayPtrtype);

var         I : integer;

begin

  with Ptr^ do
    begin
    DefineWindow(Number,                  {define window where drawing}
                 HiX,HiY,                 {will take place}
                 LoX,LoY);
    SelectWorld(WorldNum);                {select world for array window}
    SelectWindow(Number);                 {select the window}
    SetBackground(0);                     {clears window of all possible}
                                          {background garbage}
    DrawBorder;
    for I:=1 to 3 do
        with Boxes[I] do
             begin
             DrawSquare(Xhi,Yhi,
                        Xlo,Ylo,
                        false);
             DrawTextW(Xtxt,Ytxt,
                       1,Txt);
             end;
    DrawTextW(Boxes[4].Xtxt,
              Boxes[4].Ytxt,
              1,Boxes[4].Txt);
    Str(Steps:4,Boxes[1].Dgt);
    Str(Times:9:6,Boxes[2].Dgt);
    Boxes[3].Dgt:=CurrntPtr^.StatTxt;
    for I:=1 to 3 do
        with Boxes[I] do
          DrawTextW(Xdgt,
                    Ytxt,1,
                    Dgt);
    end;

end;
```

```
(***************************************************************************
 *                                                                         *
 *  This procedure draws up the configuration of arrays read in from script *
 *  file. It also stores all configured windows in the window stack for    *
 *  later updating. Depending on the type of the array, it will call these *
 *  procedures :                                                           *
 *      - DrwSquare (),                                                    *
 *      - DrwTriangle1 (),                                                 *
 *      - DrwTriangle2 (),                                                 *
 *      - DrwTriangle3 (),                                                 *
 *      - DrwTriangle4 ()                                                  *
 *  to properly draw the array itself.                                     *
 *                                                                         *
 ***************************************************************************)

procedure DrawSystem
         (Ptr : ArrayPtrType);          {points to current array}

var   TempPtr : ArrayPtrType;          {moving array pointer}

begin

  SelectScreen(2);                     {at RAM screen..}
  case Ptr^.ArrayType of               {draw the current array, depending}
    Square:                            {on which type it is}
       DrwSquare(FirstWorld,Ptr);
    Triangle1:
       DrwTriangle1(FirstWorld,Ptr);
    Triangle2:
       DrwTriangle2(FirstWorld,Ptr);
    Triangle3:
       DrwTriangle3(FirstWorld,Ptr);
    Triangle4:
       DrwTriangle4(FirstWorld,Ptr);
  end;
  StoreWindow(Ptr^.Number);            {and stores it in window stack}
  ClearScreen;                         {then clears RAM.}
  TempPtr:=Ptr^.Next;                  {starts with a non-current array}
  while TempPtr<>Ptr do                {and as with all non-current array..}
    begin
    if TempPtr<>StatPtr then           {except the status window}
      with TempPtr^ do
        begin
          case ArrayType of
          Square:
          DrwSquare(FirstWorld,        {draws them and..}
                    TempPtr);
          Triangle1:
          DrwTriangle1(FirstWorld,
                       TempPtr);
        Triangle2:
        DrwTriangle2(FirstWorld,
                     TempPtr);
        Triangle3:
        DrwTriangle3(FirstWorld,
                     TempPtr);
        Triangle4:
        DrwTriangle4(FirstWorld,
                     TempPtr);
        end;
        StoreWindow(Number);          {stores their image into window stack}
      end;
   TempPtr:=TempPtr^.Next;
   end;
  DrwStatusWindow(StatusWorld,        {then draw status window}
                  StatPtr);
  StoreWindow(StatPtr^.Number);       {don't forget to stores it}
  CopyScreen;                         {and copy'm all to displayed screen.}
  SelectScreen(1);                    {now selects displayed screen..}
  RestoreWindow(Ptr^.Number,0,0);     {restores current window to its}
                                      {current position,}
  SelectWindow(Ptr^.Number);          {selects it}
  InvertWindow;                       {then shows that it's current.}

end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure changes the displays' color, backward or forward.        *
 *                                                                          *
 ****************************************************************************)

procedure ChangeColor
   (Direction : integer);           {negative for previous color,}
                                    {positive for next.}

begin

  Foreground:=                      {computes next or previous color}
      (Foreground+Direction)
        mod 16;
   if Foreground=0 then             {remember to skips color black}
      if Direction<0 then
         Foreground:=15
      else if Direction>0 then
              Foreground:=1;
   SetForegroundColor               {set it}
         (Foreground);

end;
```

```
(***************************************************************************
 *                                                                         *
 *  This procedure redraws the current array in the next display mode. This *
 *  will allow a user to look at all registers of PEs in the array at the   *
 *  same time for easy debuging. Depending on the type of array it will call *
 *  these procedures :                                                     *
 *     - DrwSquare (),                                                     *
 *     - DrwTriangle1 (),                                                  *
 *     - DrwTriangle2 (),                                                  *
 *     - DrwTriangle3 (),                                                  *
 *     - DrwTriangle4 ()                                                   *
 *  to properly draw the array itself.                                     *
 *                                                                         *
 ***************************************************************************)

procedure ChangeDisplayMode
        (Ptr : ArrayPtrType);           {points to current array}

var   TempPtr : ArrayPtrType;           {moving array pointer}

begin

  if Ptr=StatPtr then                   {if this is the status panel then..}
     begin
     sound(500);                        {screams at 1000 Hertz}
     delay(300);                        {for 3 tenths of a second}
     nosound;                           {then shuts up}
     end
  else with Ptr^ do
    begin
    case DPmode of
      Full : DPmode:=Arrays;
      Arrays : DPmode:=Buffer;
      Buffer : DPmode:=Full;
      end;
    ClearWindowStack(Number);           {erase old window from window stack}
    SelectScreen(2);                    {select RAM screen..}
    ClearScreen;                        {wipes it clean and..}
    case ArrayType of                   {draw the current array, depending}
      Square:                           {on which type it is}
       DrwSquare(FirstWorld,Ptr);
      Triangle1:
       DrwTriangle1(FirstWorld,Ptr);
      Triangle2:
       DrwTriangle2(FirstWorld,Ptr);
      Triangle3:
       DrwTriangle3(FirstWorld,Ptr);
      Triangle4:
       DrwTriangle4(FirstWorld,Ptr);
      end;
    StoreWindow(Number);                {and stores it in window stack}
    ClearScreen;                        {then clears RAM.}
    TempPtr:=Next;                      {starts with a non-current array}
    while TempPtr<>Ptr do               {and as with all non-current array..}
      begin
      if TempPtr<>StatPtr then          {except the status window}
         RestoreWindow                  {restores all windows to their}
         (TempPtr^.Number,0,0);         {current position,}
      TempPtr:=TempPtr^.Next;
      end;
    RestoreWindow                       {then restore status window to its}
    (StatPtr^.Number,0,0);              {current position}
    CopyScreen;                         {and copy RAM to displayed screen.}
    SelectScreen(1);                    {now selects displayed screen..}
    RestoreWindow(Number,0,0);          {restores current window to its}
                                        {current position,}
    InvertWindow;                       {then shows that it's current.}
    end;

end;
```

```
(**************************************************************************
 *                                                                        *
 *  This procedure makes either the previous or the next window current for  *
 *  any operation, for example moving a window. The current window can     *
 *  overlaps other windows without destroying them.                       *
 *                                                                        *
 **************************************************************************)

procedure SwitchWindow
     (var Ptr : ArrayPtrType;              {points to current array}
            I : integer);                  {0 for previous, 1 for next}

var   TempPtr : ArrayPtrType;
      TempStr : Textype;
       TempNo : integer;

begin

  TempStr:=Ptr^.StatTxt;                   {remember text and number of}
  TempNo:=Ptr^.Number;                     {current window}
  if I=0 then Ptr:=Ptr^.Last               {if backward, makes previous window}
  else Ptr:=Ptr^.Next;                     {current, else next window.}
  TempPtr:=Ptr^.Next;
  InvertWindow;                            {Shows window isn't current anymore}
  StoreWindow(TempNo);                     {and stores it in window stack}
  SelectScreen(2);                         {now, selects the RAM screen..}
  ClearScreen;                             {clears it, then..}
  with StatPtr^,Boxes[3] do                {updates the ARRAY # box of the}
    begin                                  {status window by..}
    RestoreWindow(Number,0,0);
    SelectWorld(StatusWorld);
    SelectWindow(Number);
    SetColorBlack;                         {erasing the old status text}
    DrawTextW(Xdgt,Ytxt,1,TempStr);
    SetColorWhite;                         {and write in status text of}
    DrawTextW(Xdgt,Ytxt,1,                 {current window}
                Ptr^.StatTxt);
    StoreWindow(Number);
    end;
  ClearScreen;
  while TempPtr<>Ptr do                    {and as with all non-current windows..}
    with TempPtr^ do
      begin
      if NumBer<>MaxWindowsGlb then        {except the status window}
         RestoreWindow(Number,0,0);        {brings them back to RAM screen at}
      TempPtr:=Next;                       {their current position.}
      end;
  if Ptr<>StatPtr then                     {now draw status window if it's}
     RestoreWindow                         {not the current one.}
     (MaxWindowsGlb,0,0);
  CopyScreen;                              {copy to displayed screen..}
  SelectScreen(1);                         {then selects displayed screen,}
  RestoreWindow(Ptr^.Number,0,0);      {restores current window to}
                                       {its current position,}
  SelectWindow(Ptr^.Number);           {selects it}
  InvertWindow;                        {and shows that it's current}

end;
```

```
(*************************************************************************
 *                                                                       *
 *  This procedure writes back out the (possibly updated) script file to *
 *  disk. Since its logic is fairly straightforward, no comments within its *
 *  body will be needed. So, there will be none.                          *
 *                                                                       *
 *************************************************************************)

procedure WriteScriptFile;

var    SysPtr : ArrayPtrtype;
       IOPntr : IOPtrtype;
      LnkPtr1 ,
      LnkPtr2 : LinkPtrType;
    Int1,Int2 ,
          X,Y : integer;

begin

  assign(ScriptFile,ScriptName);
  rewrite(ScriptFile);
  writeln(ScriptFile,'ARRAYSIZE :');
  writeln(ScriptFile,ArraySize,' .');
  writeln(ScriptFile,'SYSTEMSPECS :');
  SysPtr:=FixedPtr;
  while SysPtr<>StatPtr do
  with SysPtr^ do
    begin
    Int1:=integer(ArrayType);
    Int2:=integer(DPmode);
    writeln(ScriptFile,Number,' ',
            Int1,' ',Int2,' ',
            HiX,' ',HiY,' ,');
    write(ScriptFile,'Pecodes :');
    for X:=1 to ArraySize do
      begin
      for Y:=1 to ArraySize do
          write(ScriptFile,
                ' ',PE[X,Y].Code:2);
      if X<ArraySize then
         begin
         writeln(ScriptFile);
         write(ScriptFile,
               '          ');
         end;
      end;
    if Next<>StatPtr then
       writeln(ScriptFile,' ;')
    else writeln(ScriptFile,' .');
    SysPtr:=Next;
    end;
  IOPntr:=IOPtr;
```

```
writeln(ScriptFile,'INFILES :');
while IOPntr^.IO=INPUT do
with IOPntr^ do
  begin
  write(ScriptFile,Name,' ',
        ArNum,' ',Side,' ',
        Bus,' ',IOStart);
  if NextIO^.IO=INPUT then
     writeln(ScriptFile,' ,')
  else writeln(ScriptFile,' .');
  IOPntr:=NextIO;
  end;
writeln(ScriptFile,'OUTFILES :');
while IOPntr<>NIL do
with IOPntr^ do
  begin
  write(ScriptFile,Name,' ',
        ArNum,' ',Side,' ',
        Bus,' ',IOStart);
  if NextIO<>NIL then
     writeln(ScriptFile,' ,')
  else writeln(ScriptFile,' .');
  IOPntr:=NextIO;
  end;
LnkPtr1:=LinkPtr;
LnkPtr2:=LinkPtr;
writeln(ScriptFile,'SETUP :');
while LnkPtr1<>NIL do
with LnkPtr1^ do
  begin
  writeln(ScriptFile,ArNums[1]);
  while (LnkPtr2<>NIL) and
        (LnkPtr2^.ArNums[1]=
         ArNums[1]) do
    begin
    case LnkPtr2^.Sides[1] of
      1: write(ScriptFile,
               'NorthInput : ');
      2: write(ScriptFile,
               'EastInput : ');
      3: write(ScriptFile,
               'SouthInput : ');
      4: write(ScriptFile,
               'WestInput : ');
      end;
    write(ScriptFile,
          LnkPtr2^.ArNums[2],' ',
          LnkPtr2^.Sides[2],' ',
          LnkPtr2^.LnkStart,' ',
          LnkPtr2^.LnkStop,' ');
    if LnkPtr2^.NxtLink=NIL then
       writeln(ScriptFile,'.')
      else if LnkPtr2^.NxtLink^.
              ArNums[1]<>ArNums[1]
              then
              writeln(ScriptFile,';')
      else writeln(ScriptFile,',');
      LnkPtr2:=LnkPtr2^.NxtLink;
      end;
    LnkPtr1:=LnkPtr2;
    end;
    close(ScriptFile);

end;
```

```
(***************************************************************************
 *                                                                         *
 *  This procedure gets script file name specified by user on the command  *
 *  line or failing that it will prompt user for it.                      *
 *                                                                         *
 ***************************************************************************)

procedure PromptUser;

var        OK : boolean;

begin

  ClrScr;                              {clears out display}
  if ParamCount=0 then                 {if no parameter on command line}
     begin
     writeln('** Script filename ?');  {prompts user and reads in script}
     write(' > ');                     {file name}
     readln(ScriptName);
     end
  else ScriptName:=ParamStr(1);
  repeat                               {IO loop check here}
    assign(ScriptFile,ScriptName);
    {$I-}
    reset(ScriptFile);
    {$I+}
    OK:=(IOresult=0);                  {check IO result for error}
    if not OK then
      begin
      writeln('!! File not found !!'); {let user knows if error}
      writeln('** Script filename ?'); {prompts user again}
      write(' > ');
      readln(ScriptName);
      end;
  until OK;                            {until no more IO error}

end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This function returns the first (non-space) char of the next word on    *
 *  the current line. If EOLn is encounter, it will return a '@' character. *
 *                                                                           *
 *****************************************************************************)

function SeekNxtWord
 (var FileVar : text )
              : char ;

var   TempChr : char;

begin

  if SeekEoLn(FileVar) then              {skips all spaces and tabs to first}
     SeekNxtWord:='@'                    {non-blank char. Return @ if char}
                                         {is EOLn..}
  else begin                             {else return it}
         read(FileVar,TempChr);
         SeekNxtWord:=TempChr;
       end;

end;
```

```
(****************************************************************************
 *                                                                          *
 *  This function reads statements out of the script file and returns      *
 *  their type to the calling block. It will set an appropriate error value *
 *  and returns a zero if there are any syntax error in the statements.     *
 *                                                                          *
 ****************************************************************************)

function StatementType
              : integer ;

var         I : integer;
      TempStr : string[MaxWord];

begin
  I:=0;
  while SeekEoLn(ScriptFile) do          {skips all spaces, tabs}
        readln(ScriptFile);              {and blank lines}
  repeat                                 {then reads in the statement}
    I:=I+1;
    read(ScriptFile,TempStr[I]);
  until Eoln(ScriptFile) or
        (I=MaxWord)      or
        (TempStr[I]=' ') or
        (TempStr[I]=':');
  if (TempStr[I]=' ') and                {then finds the delimiter.}
     not SeekEoln(ScriptFile) then
     read(ScriptFile,TempStr[I]);
  if TempStr[I]=':' then                 {if found, see what type of}
    begin                                {statement it is}
    TempStr[0]:=Chr(I-1);
    I:=1;
    while (I<=MaxStr) and
          (TempStr<>StringList[I])
          do I:=I+1;
    if I<=MaxStr then                    {and return its type}
      begin
      StatementType:=I;
      if SeekEoln(ScriptFile) then
         readln(ScriptFile);
      end
    else                                 {else, sets error-type value}
      begin StatementType:=0;
      ErrorSet:=[1]; end;
    end
  else                                   {screams here too}
    begin
    StatementType:=0;
    ErrorSet:=[1];
    end;

end;
```

```
(***************************************************************************
 *                                                                         *
 *  This procedure build and initializes an array specified in the script  *
 *  file. It does minimal syntax error checking on the script.             *
 *    It is called by :                                                    *
 *  - procedure ReadScript().                                              *
 *                                                                         *
 ***************************************************************************)

procedure GetSystemSpecs
     (var Ptr : ArrayPtrType);

var    ArType ,
         Mode ,
          X,Y : integer;
      TempChr : char;

begin

  with Ptr^ do
    begin
    read(ScriptFile,Number,              {retrieve values for the array}
         ArType,Mode,HiX,HiY);
    if not (Number in                    {error if array number is >= 16}
       [1..MaxWindowsGlb-1]) then
       begin
       ErrorSet:=[10];
       exit
       end;
    ArrayType:=TypeOfArray(ArType);      {stores the type of array or..}
    if not (ArrayType in                 {error when type is unknown}
       [Square,Triangle1,Triangle2,
        Triangle3,Triangle4]) then
       begin
       ErrorSet:=[11];
       exit;
       end;
    DPmode:=DisplayMode(Mode);           {stores the array's display mode..}
    if not (DPmode in                    {or error when type is unknown}
       [Full,Arrays,Buffer]) then
       begin
       ErrorSet:=[15];
       exit;
       end;
    TempChr:=SeekNxtWord                 {look for delimiter}
             (ScriptFile);
    If not (TempChr=',') then            {and if not found, gives error}
       begin
       ErrorSet:=[4];
       exit;
       end
    else readln(ScriptFile);
    if StatementType=6 then                {Next, look for array's PEs codes}
     for X:=1 to ArraySize do              {layout and read it in}
       for Y:=1 to ArraySize do
         begin
         read(ScriptFile,
              PE[X,Y].Code);
         if not (PE[X,Y].Code in           {error if something is wrong}
                [0..MaxCodes]) then
            begin
            ErrorSet:=[16];
            exit;
            end
         end
    else begin                             {error if something is wrong}
         ErrorSet:=[8];
         exit;
         end;
    case ArrayType of                      {depending on type of array, call}
      Square:     InitializeSquare         {propper procedure to initialize}
                  (Ptr);                   {its PEs}
      Triangle1: InitializeTriangle1
                  (Ptr);
      Triangle2: InitializeTriangle2
                  (Ptr);
      Triangle3: InitializeTriangle3
                  (Ptr);
      Triangle4: InitializeTriangle4
                  (Ptr);
      end;
    Str(Number:4,StatTxt);                 {convert the array number to its}
                                           {string equivalent for status panel}
    new(Next);                             {then get storage space for next}
    Next^.Last:=Ptr;                       {array}
    Ptr:=Next;
    end;

end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure sets the start and stop index values to traverse the side *
 *  of an array depending on the array type and which side of the array.    *
 *                                                                          *
 ****************************************************************************)

procedure SideTraversal
  (      Side : integer;
       ArType : TypeOfArray;
   var  X1,X2 ,
        Y1,Y2 : integer);

begin

  case Side of                             {depending on which side and type}
                                           {of array is involved, prepares}
                                           {the start and stop index values}
                                           {to traverse the side of array}
    1: case ArType of                      {North side}
       Square,
       Triangle1,
       Triangle3: begin
                  X1:=1;X2:=1;
                  Y1:=1;
                  Y2:=ArraySize+1;
                  end;
       Triangle2: begin
                  X1:=1;
                  X2:=ArraySize+1;
                  Y1:=1;
                  Y2:=ArraySize+1;
                  end;
       Triangle4: begin
                  X1:=ArraySize;
                  X2:=0;
                  Y1:=1;
                  Y2:=ArraySize+1;
                  end;
       end;
    2: case ArType of                      {East side}
       Square,
       Triangle1,
       Triangle4: begin
                  X1:=1;
                  X2:=ArraySize+1;
                  Y1:=ArraySize;
                  Y2:=ArraySize;
                  end;
       Triangle2: begin
                  X1:=1;
                  X2:=ArraySize+1;
```

```
              Y1:=1;
              Y2:=ArraySize+1;
              end;
   Triangle3: begin
              X1:=ArraySize;
              X2:=0;
              Y1:=1;
              Y2:=ArraySize+1;
              end;
   end;
3: case ArType of                  {South side}
   Square,
   Triangle2,
   Triangle4: begin
              X1:=ArraySize;
              X2:=ArraySize;
              Y1:=1;
              Y2:=ArraySize+1;
              end;
   Triangle1: begin
              X1:=1;
              X2:=ArraySize+1;
              Y1:=1;
              Y2:=ArraySize+1;
              end;
   Triangle3: begin
              X1:=ArraySize;
              X2:=0;
              Y1:=1;
              Y2:=ArraySize+1;
              end;
   end;
4: case ArType of                  {West side}
   Square,
   Triangle2,
   Triangle3: begin
              X1:=1;
              X2:=ArraySize+1;
              Y1:=1;Y2:=1;
              end;
   Triangle1: begin
              X1:=1;
              X2:=ArraySize+1;
              Y1:=1;
              Y2:=ArraySize+1;
              end;
   Triangle4: begin
              X1:=ArraySize;
              X2:=0;
              Y1:=1;
              Y2:=ArraySize+1;
              end;
```

```
          end;
        end;

    end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure, given the info contains in a IOType record, will link   *
 *  a buffer of an IO file to a side of an array at the proper time. If the *
 *  IO file is of type INPUT, all In_Regs of PEs' on the proper side of the *
 *  array will contain the addresses of the buffer's individual registers.  *
 *  If the IO file is of type OUTPUT, the reverse is true.                  *
 *    This procedure is called by :                                         *
 *  - procedure MultiStepsExec().                                           *
 *  - procedure SingleStepExec().                                           *
 *                                                                          *
 ****************************************************************************)

procedure LinkIOFlow
        ( Ptr : IOPtrType;
         Step : integer);

var          I ,
     X1,X2,X3 ,
     Y1,Y2,Y3 : integer;

begin
  while Ptr<>NIL do
    with Ptr^ do
      begin
      if IOStart=Step then                  {if it's time to link IO to array..}
         begin
         Active:=TRUE;                      {marks that IO channel is now active.}
         SideTraversal(Side,                {depending on which side and type}
                    ArPtr^.ArrayType,       {of array is involved, prepares the}
                    X1,X2,Y1,Y2);           {start and stop index values}
         X3:=X1; Y3:=Y1; I:=1;
         repeat                             {actual linking is done here while}
           with ArPtr^ do case IO of        {traversing the side of array}
             INPUT:
               PE[X3,Y3].                   {PE's input registers gets addresses}
               In_Regs[Side,Bus]:=          {of IO channel's input buffers}
               Addr(InRegs[I]);
             OUTPUT:                        {IO channel output buffer gets}
               OutRegs[I]:=                 {addresses of PE's output registers}
               Addr(PE[X3,Y3].
               Last_Out[Side,Bus]);
             end;
           if X1<X2 then X3:=X3+1
           else if X1>X2 then X3:=X3-1;
           if Y1<Y2 then Y3:=Y3+1;
           I:=I+1;
         until (X3=X2) and (Y3=Y2);
         end;
      Ptr:=NextIO;
      end;
end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This procedure build and initializes the IO system of the configuration  *
 *  from the script file. It does lots of error checking on the script.      *
 *    It is called by :                                                      *
 *  - procedure ReadScript().                                                *
 *    Also, it called :                                                      *
 *  - procedure LinkIOFlow().                                                *
 *                                                                           *
 *****************************************************************************)

procedure GetIOSpecs
     (var Ptr : IOPtrtype;
         Flag : IOflag);

var       Can : boolean;                     {garbage can for expediency}
            I : integer;
      TempChr : char;

begin

  repeat with Ptr^ do                        {does this 'til "." or error is met..}
    begin
    IO:=Flag;                                {set IO type}
    Active:=FALSE;                           {IO channel is not active yet}
    ArPtr:=CurrntPtr;                        {gets the address of systems arrays}
    Can:=SeekEoLn(ScriptFile);               {get all blanks in between data}
    I:=0;
    repeat                                   {retrieves IO filename to storage}
      I:=I+1;
      read(ScriptFile,Name[I]);
    until (Name[I]=' ') or
         (I=MaxFileName);
    If Name[I]<>' ' then                     {if bad name, sets error alarm}
       ErrorSet:=[5];
    Name[0]:=char(I-1);                      {sets the length of the name string}
    assign(FileVar,Name);                    {IO file preprocessing starts here}
    case Flag of
      INPUT: begin                           {if input file, open for reading..}
             {$I-}reset(FileVar);
             {$I+}
             if not (IOresult=0)             {then checks IO result for error}
                then ErrorSet:=              {and set error if there are any.}
                     ErrorSet+[13];
             end;
      OUTPUT: rewrite(FileVar);              {if output file, open for writing.}
      end;
    Can:=SeekEoLn(ScriptFile);               {get all blanks in between data}
    read(ScriptFile,ArNum,Side,              {then get all remaining data.}
         Bus,IOStart);
    if not (ArNum in                         {valid array number ?}
           [1..MaxWindowsGlb-1])
      then ErrorSet:=[11]
    else
      while (ArPtr^.Number<>ArNum)      {search for the specified array..}
        and (ArPtr^.Next<>CurrntPtr)
        do ArPtr:=ArPtr^.Next;
    if ArPtr^.Number<>ArNum then        {does array exist ?}
       ErrorSet:=ErrorSet+[6];
    if not (Side in [1..MaxRegs])       {valid side ?}
       then ErrorSet:=ErrorSet+[9];
    if not (Bus in [1..MaxBus])         {valid bus ?}
       then ErrorSet:=ErrorSet+[14];
    case Flag of                        {init. all IO buffer's registers}
      INPUT:
      for I:=1 to MaxArraySize do
          InRegs[I]:=0.0;
      OUTPUT:
      for I:=1 to MaxArraySize do
          OutRegs[I]:=ZeroPtr;
      end;
    TempChr:=SeekNxtWord(ScriptFile);   {where is delimiter ?}
    If not (TempChr in [',','.'])
       then ErrorSet:=ErrorSet+[4]
    else readln(ScriptFile);
    if (TempChr=',') or (Flag=INPUT)
       then begin
            new(NextIO);                {gets storage space for next}
            Ptr:=NextIO;                {IO unit and points to it}
            end
    else NextIO:=NIL;
    end;
  until (ErrorSet<>[]) or
        (TempChr='.');

end;
```

```
(****************************************************************************
 *                                                                          *
 *  This procedure, given the info contains in a list of LinkType records,  *
 *  will link a side of a source array to a side of an destination array,   *
 *  or it will cut off the link by pointing input registers to value zero   *
 *  The link is achieved by having all In_Regs of PEs' on the proper side of *
 *  the destination array store addresses of Out_Regs of all PE's on the    *
 *  proper side of the source array.                                        *
 *    This procedure is called by :                                         *
 *  - procedure MultiStepsExec().                                           *
 *  - procedure SingleStepExec().                                           *
 *                                                                          *
 ****************************************************************************)

procedure LinkDataFlow
        (Link : LinkPtrType;
         Step : integer);

var  X1,X2,X3 ,
     Y1,Y2,Y3 : SrcDstType;
            I : integer;

begin

  while Link<>NIL do                   {start at begining of Link list}
    with Link^ do                      {and until the end of list..}
      begin                            {do all things below.}
      if LnkStart=Step then            {if the moment of truth arrives}
        begin                          {then..}
        for I:=1 to 2 do               {..for both source and destination,}
          begin                        {depending on which side and type}
          SideTraversal(Sides[I],      {of array is involved, prepares}
              ArPtrs[I]^.ArrayType,    {the start and stop index values}
              X1[I],X2[I],             {to traverse the side of array}
              Y1[I],Y2[I]);
          X3[I]:=X1[I];
          Y3[I]:=Y1[I];
          end;
        repeat                         {repeats doing the following..}
          for I:=1 to MaxBus do        {for all buses, points Input registers}
            ArPtrs[1]^.                {of destination array to the Output}
            PE[X3[1],Y3[1]].           {registers of the source array}
            In_Regs[Sides[1],I]:=
            Addr(ArPtrs[2]^.
                PE[X3[2],Y3[2]].
                Last_Out[Sides[2],I]);
          for I:=1 to 2 do             {increments side traversal index}
            begin                      {values for both source and}
            if X1[I]<X2[I] then        {destination array}
               X3[I]:=X3[I]+1
            else if X1[I]>X2[I] then
               X3[I]:=X3[I]-1;
        if Y1[I]<Y2[I] then
           Y3[I]:=Y3[I]+1;
        end;
    until (X3[1]=X2[1]) and       {until array's side is fully}
          (Y3[1]=Y2[1]) ;         {traversed}
    end
  else if (LnkStart<Step) and     {when it's time to cut the link}
          (LnkStop=Step) then
    begin
    SideTraversal(Sides[1],       {for the destination array, prepares}
            ArPtrs[1]^.ArrayType, {the start and stop index values}
            X1[1],X2[1],          {to traverse its side}
            Y1[1],Y2[1]);
    X3[1]:=X1[1];
    Y3[1]:=Y1[1];
    repeat                        {repeats doing the following..}
      for I:=1 to MaxBus do       {for all buses, points Input registers}
        ArPtrs[1]^.               {of destination array to the}
        PE[X3[1],Y3[1]].          {value zero}
        In_Regs[Sides[1],I]:=
        ZeroPtr;
      if X1[1]<X2[1] then         {increments side traversal index}
         X3[1]:=X3[1]+1           {values for destination array}
      else if X1[1]>X2[1] then
         X3[1]:=X3[1]-1;
      if Y1[1]<Y2[1] then
         Y3[1]:=Y3[1]+1;
    until (X3[1]=X2[1]) and       {until array's side is fully}
          (Y3[1]=Y2[1]);          {traversed}
    end;
  Link:=NxtLink;
  end;

end;
```

```
(***************************************************************************
 *                                                                         *
 *  This procedure gets info of the data flow from array to array, including *
 *  feedback paths, according to a script file. It does some error checking  *
 *  on the script file and on the way user specified connective path between *
 *  arrays.                                                                *
 *    The procedure is called by :                                         *
 *  - procedure ReadScript().                                              *
 *    It calls :                                                           *
 *  - procedure LinkDataFlow().                                            *
 *                                                                         *
 ***************************************************************************)

procedure GetDataFlow
 (var    Link : LinkPtrType;
          Ptr : ArrayPtrType);

var
      TempChr : char;
      NewLink ,
      TmpLink : LinkPtrType;

begin

  new(Link);
  TmpLink:=Link;
  NewLink:=Link;
  repeat
    with TmpLink^ do                        {does all this until '.' encountered}
      begin
      ArPtrs[1]:=Ptr;                       {initializes pointers}
      ArPtrs[2]:=Ptr;
      read(ScriptFile,ArNums[1]);           {gets the destination array..}
      while (ArPtrs[1]^.Number<>            {is it valid ?}
        ArNums[1]) and
        (ArPtrs[1]^.Next<>Ptr)
        do ArPtrs[1]:=ArPtrs[1]^.Next;
      if (ArPtrs[1]^.Number<>
         ArNums[1]) or                      {if not, sets error and says goodbye}
         (ArNums[1]=MaxWindowsGlb)
        then begin
        ErrorSet:=ErrorSet+[6];
        exit;
        end;
      end;
    repeat
      with NewLink^ do                      {then does all this until ';' is met}
        begin
        Sides[1]:=StatementType-6;          {gets input side of destination array}
        if not (Sides[1] in                 {array. If it's not valid, sets error}
           [1..MaxRegs]) then
           ErrorSet:=ErrorSet+[7];
      read(ScriptFile,ArNums[2],        {Now, gets source array, its output}
           Sides[2],LnkStart,           {side and start and stop values}
           LnkStop);
      while (ArPtrs[2]^.Number<>        {validates source array here..}
        ArNums[2]) and (ArPtrs[2]^
        .Next<>Ptr) do
        ArPtrs[2]:=ArPtrs[2]^.Next;
      if (ArPtrs[2]^.Number<>
         ArNums[2]) or
         (ArNums[2]=MaxWindowsGlb)
        then begin
        ErrorSet:=ErrorSet+[6];
        exit;
        end;
      if not (Sides[2]                  {and the output side here}
         in [1..MaxRegs]) then
         ErrorSet:=ErrorSet+[9];
      if ErrorSet<>[] then exit;        {leaves if any errors}
      end;
    TempChr:=SeekNxtWord                {then seeks out delimiter. If none}
             (ScriptFile);              {found, sets error}
    If not (TempChr in
       [',',';','.']) then
      begin
      ErrorSet:=[4];
      exit;
      end
    else if (TempChr=',') then
      begin
      new(NewLink^.NxtLink);            {create new link storages and..}
      NewLink:=NewLink^.NxtLink;        {and points to it}
      NewLink^.ArPtrs[1]:=
       TmpLink^.ArPtrs[1];
      NewLink^.ArNums[1]:=
       TmpLink^.ArNums[1];
      NewLink^.ArPtrs[2]:=Ptr;
      end
    else if (TempChr=';') then
      begin
      new(NewLink^.NxtLink);
      TmpLink:=NewLink^.NxtLink;
      NewLink:=TmpLink;
      end
    else NewLink^.NxtLink:=NIL;
    readln(ScriptFile);
  until (TempChr in [';','.']);         {stops getting input direction for}
                                        {destination array}
 until (TempChr='.') or                 {stops reading dataflow set up infos}
       Eof(ScriptFile);                 {entirely}

end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This function sets up the SAGS internals according to a script file     *
 *  specified by the user. It will generate error messages and returns      *
 *  a FALSE boolean value if any error or inconsistency is encountered in    *
 *  the file. Graphics errors such as drawing a window out of screen range   *
 *  will not be handle by this function.                                     *
 *                                                                           *
 *****************************************************************************)

function ReadScript
              : boolean ;

var
   ActionType ,
    Sequencer : integer;
      TempChr : char;
      TempPtr : ArrayPtrType;
    TempIOPtr : IOPtrtype;

begin

  ErrorSet:=[];                     {clears error register and init.}
  Sequencer:=1;                     {sequence counter}
  while (ErrorSet=[])               {continues the system setup sequence}
        and (Sequencer<=MaxSequence)  {until error occurs}
    do begin
    ActionType:=StatementType;      {gets the step number and if it's}
    if ActionType=Sequencer then    {in sequence then proceeds}
      begin
        case ActionType of
        1: begin                    {reads in array size of system}
           read(ScriptFile,ArraySize);
           if ArraySize>MaxArraySize  {if array size too large, sets error}
              then ErrorSet:=
              ErrorSet+[2];
           if SeekNxtWord             {look for the delimiter and}
              (ScriptFile)<>'.'
              then ErrorSet:=         { if not found, error}
                   ErrorSet+[3]
           else readln(ScriptFile);
           end;
        2: begin                    {creates arrays system here}
           InitGlbStorage;          {init. all global graphics values}
           new(FixedPtr);
           FixedPtr^.Last:=StatPtr;
           StatPtr^.Next:=FixedPtr;
           CurrntPtr:=FixedPtr;
           TempPtr:=FixedPtr;
           repeat
             GetSystemSpecs(TempPtr);
```

```
          TempChr:=SeekNxtWord       {look for delimiter}
                   (ScriptFile);
          If not                     {and if not found, gives error}
            (TempChr in [';','.'])
            then ErrorSet:=
                 ErrorSet+[4]
          else readln(ScriptFile);
        until (ErrorSet<>[]) or
              (TempChr='.');
        if ErrorSet=[] then          {if no error,}
          begin                      {then create a circular}
            TempPtr:=TempPtr^.Last;  {doubly linked list which included}
            Dispose(TempPtr^.Next);  {the status window.}
            TempPtr^.Next:=StatPtr;
            StatPtr^.Last:=TempPtr;
          end;
        end;
      3: begin                       {creates IO system here}
        new(IOPtr);                  {starts IO linked list}
        TempIOPtr:=IOPtr;
        GetIOSpecs(TempIOPtr,        {then reads in IO specs..}
                   INPUT);
        end;
      4: GetIOSpecs(TempIOPtr,       {and reads in some more then..}
                    OUTPUT);
      5: GetDataFlow(LinkPtr,        {get description of the flow of data}
                     CurrntPtr);
      end;
      Sequencer:=Sequencer+1;
    end
  else ErrorSet:=ErrorSet+[8];
  end;                               {setup sequence ends here.}
if ErrorSet<>[] then                 {looks through error list acumulated}
  begin                              {thus far and displays appropriate}
  for Sequencer:=1 to MaxError do    {error message}
    begin
      if Sequencer in ErrorSet
         then writeln
         (ErrorList[Sequencer]);
      ErrorSet:=
      ErrorSet-[Sequencer];
    end;
  ReadScript:=FALSE;
  end
else ReadScript:=TRUE;               {if no error, signal calling block}
                                     {to continue}
end;
```

```
(****************************************************************************
 *                                                                          *
 *   This procedure represents the execution code of a shift down register  *
 *   array. This array moves data in the North to South direction.          *
 *   _  R        is     X_Reg                                               *
 *   _  Xin      is     In_Regs[1,1]^                                       *
 *   _  TAGin    is     In_Regs[1,2]^                                       *
 *   _  Xout     is     Out_Regs[3,1]                                       *
 *   _  TAGout   is     Out_Regs[3,2]                                       *
 *                                                                          *
 ****************************************************************************)

procedure N2Scode
      (var PE : PEtype);

begin

  with PE do
    begin
    Out_Regs[3,1]:=In_Regs[1,1]^;
    Out_Regs[3,2]:=In_Regs[1,2]^;
    X_Reg:=Out_Regs[3,1];                {put value in here for display}
    TAG:=Trunc(Out_Regs[3,2]);
    end;

end;
```

```
(****************************************************************************
 *                                                                          *
 *   This procedure represents the execution code of a shift left register  *
 *   array. This array moves data in the East to West direction.            *
 *   _ R        is    X_Reg                                                 *
 *   _ Xin      is    In_Regs[2,1]^                                         *
 *   _ TAGin    is    In_Regs[2,2]^                                         *
 *   _ Xout     is    Out_Regs[4,1]                                         *
 *   _ TAGout   is    Out_Regs[4,2]                                         *
 *                                                                          *
 ****************************************************************************)

procedure E2Wcode
      (var PE : PEtype);

begin

  with PE do
    begin
    Out_Regs[4,1]:=In_Regs[2,1]^;
    Out_Regs[4,2]:=In_Regs[2,2]^;
    X_Reg:=Out_Regs[4,2];              {put values in these registers for}
    TAG:=Trunc(Out_Regs[4,1]);         {display}
    end;

end;
```

```
(****************************************************************************
 *                                                                          *
 *   This procedure represents the execution code of a shift up register   *
 *   array. This array moves data in the South to North direction.          *
 *   _  R         is     X_Reg                                              *
 *   _  Xin       is     In_Regs[3,1]^                                      *
 *   _  TAGin     is     In_Regs[3,2]^                                      *
 *   _  Xout      is     Out_Regs[1,1]                                      *
 *   _  TAGout    is     Out_Regs[1,2]                                      *
 *                                                                          *
 ****************************************************************************)

procedure S2Ncode
      (var PE : PEtype);

begin

  with PE do
    begin
    Out_Regs[1,1]:=In_Regs[3,1]^;
    Out_Regs[1,2]:=In_Regs[3,2]^;
    X_Reg:=Out_Regs[1,1];                  {put values in these registers for}
    TAG:=Trunc(Out_Regs[1,2]);             {display}
    end;

end;
```

```
(***************************************************************************
 *                                                                         *
 *   This procedure represents the execution code of a shift up register  *
 *   array. This array moves data in the West to East direction.          *
 *   _  R        is     X_Reg                                             *
 *   _  Xin      is     In_Regs[4,1]^                                     *
 *   _  TAGin    is     In_Regs[4,2]^                                     *
 *   _  Xout     is     Out_Regs[2,1]                                     *
 *   _  TAGout   is     Out_Regs[2,2]                                     *
 *                                                                         *
 ***************************************************************************)

procedure W2Ecode
      (var PE : PEtype);

begin

  with PE do
    begin
    Out_Regs[2,1]:=In_Regs[4,1]^;
    Out_Regs[2,2]:=In_Regs[4,2]^;
    X_Reg:=Out_Regs[2,2];               {put values in these registers for}
    TAG:=Trunc(Out_Regs[2,1]);          {display}
    end;

end;
```

```
(*****************************************************************************
 *                                                                           *
 *   This procedure represents the execution code of HE's systolic array    *
 *   for boundary cell.                                                      *
 *   _ X        is    X_Reg                                                  *
 *   _ Xin      is    In_Regs[1,1]^                                          *
 *   _ TAGin    is    In_Regs[1,2]^                                          *
 *   _ Vout     is    Out_Regs[2,1]                                          *
 *   _ Mout     is    Out_Regs[2,2]                                          *
 *   _ -Mout    is    Out_Regs[3,1]                                          *
 *                                                                           *
 *****************************************************************************)

procedure HEcode1
      (var PE : PEtype);

begin

  with PE do
    begin
    Out_Regs[4,2]:=In_Regs[1,1]^;          {get Xin and..}
    TAG:=Trunc(In_Regs[1,2]^);             {pivoting TAG bit values}
    if (TAG=0) and                         {if pivoting is allowed and Xin is}
       (abs(Out_Regs[4,2])>=               {greater in magnitude than X, then..}
        abs(X_Reg)) then
       begin
       Out_Regs[2,1]:=1.0;                 {tell the East neighboring cell to}
       if Out_Regs[4,2]<>0.0 then          {pivot and send it a modifying value}
          Out_Regs[2,2]:=
          -X_Reg/Out_Regs[4,2]
       else Out_Regs[2,2]:=0.0;
       X_Reg:=Out_Regs[4,2];
       end
    else begin
         Out_Regs[2,1]:=0.0;               {else, no pivoting..}
         Out_Regs[2,2]:=                   {with modifying value}
             -In_Regs[1,1]^/X_Reg;
         end;
    Out_Regs[3,1]:=-Out_Regs[2,2];         {moves Mout}
    end;

end;
```

```
(****************************************************************************
 *                                                                          *
 *   This procedure represents the execution code of HE's systolic array    *
 *   for internal cell.                                                     *
 *                                                                          *
 *   _  X is X_Reg                                                          *
 *   _  Xin     is    In_Regs[1,1]^                                         *
 *   _  Vin     is    In_Regs[4,1]^                                         *
 *   _  Min     is    In_Regs[4,2]^                                         *
 *   _  Xout    is    Out_Regs[3,1]                                         *
 *   _  Vout    is    Out_Regs[2,1]                                         *
 *   _  Mout    is    Out_Regs[2,2]                                         *
 *                                                                          *
 ****************************************************************************)

procedure HEcode2
      (var PE : PEtype);

begin

  with PE do
    begin
    Out_Regs[2,1]:=In_Regs[4,1]^;
    Out_Regs[2,2]:=In_Regs[4,2]^;
    TAG:=Trunc(Out_Regs[2,1]);           {get TAG bit for display}
    if TAG=1 then
      begin
      Out_Regs[3,1]:=X_Reg
                    +Out_Regs[2,2]
                    *In_Regs[1,1]^;
      X_Reg:=In_Regs[1,1]^;
      end
    else Out_Regs[3,1]:=In_Regs[1,1]^
                       +Out_Regs[2,2]
                       *X_Reg;
    Out_Regs[3,2]:=In_Regs[1,2]^;        {pass on pivoting allowed bit}
    end;

end;
```

```
(*****************************************************************************
 *                                                                           *
 *   This procedure represents the execution code of NASH's systolic array  *
 *   for boundary cell.                                                      *
 *                                                                           *
 *  _ R             is     X_Reg                                             *
 *  _ TAG           is     In_Regs[1,2]^                                     *
 *  _ Xin           is     In_Regs[1,1]^                                     *
 *  _ Cout or Xout  is     Out_Regs[2,1]                                     *
 *  _ Sout          is     Out_Regs[2,2]                                     *
 *                                                                           *
 *****************************************************************************)

procedure NASHcode1
      (var PE : PEtype);

var         T : real;

begin

  with PE do
    begin
    TAG:=Trunc(In_Regs[1,2]^);
    if TAG=0 then
       if In_Regs[1,1]^ = 0.0 then
          begin
          Out_Regs[2,1]:=1.0;
          Out_Regs[2,2]:=0.0;
          X_Reg:=0.0;                      {This line will give us incorrect}
                                           {result. Delete it will cure all of}
                                           {Nash's ailments.}

          end
       else begin
            T:= sqrt(sqr(X_Reg)
                +sqr(In_Regs[1,1]^));
            Out_Regs[2,1]:=X_Reg/T;
            Out_Regs[2,2]:=
                   In_Regs[1,1]^/T;
            X_Reg:=T;
            end
    else Out_Regs[2,1]:=
                In_Regs[1,1]^/X_Reg;
    end;

end;
```

```
(**************************************************************************
 *                                                                         *
 *   This procedure represents the execution code of NASH's systolic array  *
 *   for internal cell.                                                    *
 *                                                                         *
 *   _ R          is    X_Reg                                              *
 *   _ TAG        is    In_Regs[1,2]^                                      *
 *   _ Xin        is    In_Regs[1,1]^                                      *
 *   _ Cin or Y is      In_Regs[4,1]^                                      *
 *   _ Sin        is    In_Regs[4,2]^                                      *
 *   _ TAGout     is    Out_Regs[3,2]                                      *
 *   _ Xout       is    Out_Regs[3,1]                                      *
 *   _ Cout       is    Out_Regs[2,1]                                      *
 *   _ Sout       is    Out_Regs[2,2]                                      *
 *                                                                         *
 **************************************************************************)

procedure NASHcode2
      (var PE : PEtype);

begin

  with PE do
    begin
    Out_Regs[3,2]:=In_Regs[1,2]^;
    TAG:=Trunc(Out_Regs[3,2]);
    if TAG=0 then
      begin
      Out_Regs[3,1]:=
         -(In_Regs[4,2]^ * X_Reg)
         +(In_Regs[4,1]^
           * In_Regs[1,1]^);
      X_Reg:=In_Regs[4,1]^
           * X_Reg + In_Regs[4,2]^
           * In_Regs[1,1]^;
      end
    else Out_Regs[3,1]:=In_Regs[1,1]^
                      -In_Regs[4,1]^
                      *X_Reg;
    Out_Regs[2,1]:=In_Regs[4,1]^;
    Out_Regs[2,2]:=In_Regs[4,2]^;
    end;

end;
```

```
(*************************************************************************
 *                                                                       *
 *   This procedure represents the execution code of my systolic array   *
 *   design for diagonal cell.                                           *
 *                                                                       *
 *   _ X         is    X_Reg                                             *
 *   _ Xin       is    In_Regs[1,1]^                                     *
 *   _ C124in    is    In_Regs[1,2]^                                     *
 *   _ C3in      is    In_Regs[4,1]^                                     *
 *   _ Min       is    In_Regs[4,2]^                                     *
 *   _ Xout      is    Out_Regs[3,1]                                     *
 *   _ C124out   is    Out_Regs[3,2]                                     *
 *   _ C3out     is    Out_Regs[2,1]                                     *
 *   _ Mout      is    Out_Regs[2,2]                                     *
 *                                                                       *
 *************************************************************************)

procedure LEcode1
      (var PE : PEtype);

begin

  with PE do
    begin
    C124 := Trunc(In_Regs[1,2]^);        {stores C1, C2, C3, and C4}
    C3 := Trunc(In_Regs[4,1]^);
    if Odd(C124) then                    {if C4 is 1 then clear X}
       X_Reg := 0.0 ;
    if C124>7 then                       {if C1 is 1 then Triangle mode}
       begin
       if (abs(In_Regs[1,1]^)>=          {if 3Xin3 r 3X3 and C2 is 1 then}
           abs(X_Reg)) and               {pivoting is needed and allowed}
          (C124>11) then
          begin
          if C124 in [12,13] then        {set C3 to 1..}
             Out_Regs[2,1]:=C124+2
          else Out_Regs[2,1]:=C124;
          if (In_Regs[1,1]^<>0.0) then
             Out_Regs[2,2]:=
             -X_Reg/In_Regs[1,1]^
          else Out_Regs[2,2]:=0.0;
          X_Reg:=In_Regs[1,1]^;
          end
       else begin                        {else pivoting is not allowed}
            if C124 in [10,11,14,15]     {set C3 to 0..}
               then Out_Regs[2,1]:=
                      C124-2
            else Out_Regs[2,1]:=C124;
            Out_Regs[2,2]:=
                       -In_Regs[1,1]^
                       /X_Reg;
            end;
      TAG:=C124;                        {display that cell is triangle mode}
      end
   else begin                           {else C1 is in Square mode.}
        If C3 in [2,3,6,7] then         {if C3 is 1 then..}
           begin
           Out_Regs[3,1]:=X_Reg
                  +In_Regs[4,2]^
                  *In_Regs[1,1]^;
           X_Reg:=In_Regs[1,1]^;
           end
        else Out_Regs[3,1]:=            {else if C3 is 0 then..}
                In_Regs[1,1]^
               +In_Regs[4,2]^
               *X_Reg;
        Out_Regs[2,1]:=C3;              {pass on C3.}
        Out_Regs[2,2]:=In_Regs[4,2]^;   {Pass on Min.}
        TAG:=C3;                        {display that cell in square mode}
        end;
   Out_Regs[3,2]:=C124;                 {In any case, pass on C1, C2, C4.}
   end;

end;

{************************************}

procedure LEcode2
     (var PE : PEtype);

begin

  with PE do
   begin
   C124 := Trunc(In_Regs[1,2]^);        {stores C1, C2, C3, and C4}
   C3 := Trunc(In_Regs[4,1]^);
   if Odd(C124) then                    {if C4 is 1 then clear X}
      X_Reg := 0.0 ;
   if C124>7 then                       {if C1 is 1 then Triangle mode}
      begin
      if (abs(In_Regs[1,1]^)>=          {if 3Xin3 r 3X3 and C2 is 1 then}
          abs(X_Reg)) and               {pivoting is needed and allowed}
         (C124>11) then
         begin
         if C124 in [12,13] then        {set C3 to 1..}
            Out_Regs[2,1]:=C124-6
         else Out_Regs[2,1]:=C124-8;
         if (In_Regs[1,1]^<>0.0) then
            Out_Regs[2,2]:=
            -X_Reg/In_Regs[1,1]^
         else Out_Regs[2,2]:=0.0;
         X_Reg:=In_Regs[1,1]^;
         end
      else begin                        {else pivoting is not allowed}
```

```
          if C124 in [10,11,14,15]    {set C3 to 0..}
             then Out_Regs[2,1]:=
                  C124-10
          else Out_Regs[2,1]:=C124-8;
          Out_Regs[2,2]:=
                     -In_Regs[1,1]^
                     /X_Reg;
          end;
       TAG:=C124;                     {display that cell is triangle mode}
       end
    else begin                        {else C1 is in Square mode.}
         If C3 in [2,3,6,7] then      {if C3 is 1 then..}
            begin
            Out_Regs[3,1]:=X_Reg
                   +In_Regs[4,2]^
                   *In_Regs[1,1]^;
            X_Reg:=In_Regs[1,1]^;
            end
         else Out_Regs[3,1]:=         {else if C3 is 0 then..}
                  In_Regs[1,1]^
                 +In_Regs[4,2]^
                 *X_Reg;
         Out_Regs[2,1]:=C3;           {pass on C3.}
         Out_Regs[2,2]:=In_Regs[4,2]^; {Pass on Min.}
         TAG:=C3;                     {display that cell in square mode}
         end;
    Out_Regs[3,2]:=C124;              {In any case, pass on C1, C2, C4.}
    end;

end;
```

```
(*************************************************************************
 *                                                                       *
 *   This procedure represents the execution code of my systolic array   *
 *   design for square cells.                                            *
 *                                                                       *
 *   _ X        is     X_Reg                                             *
 *   _ Xin      is     In_Regs[1,1]^                                     *
 *   _ C3in     is     In_Regs[4,1]^                                     *
 *   _ Min      is     In_Regs[4,2]^                                     *
 *   _ Xout     is     Out_Regs[3,1]                                     *
 *   _ C3Eout   is     Out_Regs[3,2]                                     *
 *   _ C3Sout   is     Out_Regs[2,1]                                     *
 *   _ Mout     is     Out_Regs[2,2]                                     *
 *                                                                       *
 *************************************************************************)

procedure LEcode3
      (var PE : PEtype);

begin

  with PE do
    begin
    C3 := Trunc(In_Regs[4,1]^);          {stores C1, C2, C3, C4.}
    TAG:=C3;                             {display control code}
    if Odd(C3) then                      {if C4 is 1 then clear X}
       X_Reg := 0.0 ;
    If C3 in [2,3,6,7,                   {if C3 is 1 then..}
       10,11,14,15] then
       begin
       Out_Regs[3,1]:=X_Reg
              +In_Regs[4,2]^
              *In_Regs[1,1]^;
       X_Reg:=In_Regs[1,1]^;
       end
    else Out_Regs[3,1]:=                 {else if C3 is 0 then..}
            In_Regs[1,1]^
           +In_Regs[4,2]^
           *X_Reg;
    Out_Regs[3,2]:=C3;                   {pass on C1, C2, C3, C4.}
    if C3>7 then
       Out_Regs[2,1]:=C3-8
    else Out_Regs[2,1]:=C3;
    Out_Regs[2,2]:=In_Regs[4,2]^;        {Pass on Min.}
    end;

end;
```

```
(***************************************************************************
 *                                                                         *
 *   This procedure represents the execution code of my systolic array     *
 *   design for square cells.                                              *
 *                                                                         *
 *   _ X         is    X_Reg                                               *
 *   _ Xin       is    In_Regs[1,1]^                                       *
 *   _ C124in    is    In_Regs[1,2]^                                       *
 *   _ C3in      is    In_Regs[4,1]^                                       *
 *   _ Min       is    In_Regs[4,2]^                                       *
 *   _ Xout      is    Out_Regs[3,1]                                       *
 *   _ C124out   is    Out_Regs[3,2]                                       *
 *   _ C3out     is    Out_Regs[2,1]                                       *
 *   _ Mout      is    Out_Regs[2,2]                                       *
 *                                                                         *
 ***************************************************************************)

procedure LEcode4
      (var PE : PEtype);

begin

  with PE do
    begin
    C124 := Trunc(In_Regs[1,2]^);        {stores C1, C2, C3, and C4}
    C3 := Trunc(In_Regs[4,1]^);
    TAG:=C3;                             {display control code}
    if Odd(C124) then                    {if C4 is 1 then clear X}
       X_Reg := 0.0 ;
    If C3 in [2,3,6,7] then              {if C3 is 1 then..}
       begin
       Out_Regs[3,1]:=X_Reg
              +In_Regs[4,2]^
              *In_Regs[1,1]^;
       X_Reg:=In_Regs[1,1]^;
       end
    else Out_Regs[3,1]:=                 {else if C3 is 0 then..}
            In_Regs[1,1]^
           +In_Regs[4,2]^
           *X_Reg;
    Out_Regs[2,1]:=C3;
    Out_Regs[2,2]:=In_Regs[4,2]^;        {Pass on Min.}
    Out_Regs[3,2]:=C124;                 {In any case, pass on C1, C2, C4.}
    end;

end;
```

```
(*****************************************************************************
 *                                                                           *
 *  This procedure updates the image of an array in its window to reflect   *
 *  the state of the computation at a particular step. Depending on the     *
 *  particular type of array, it will only updates allowable PEs.           *
 *                                                                           *
 *****************************************************************************)

procedure UpdateArray
        (Ptr : ArrayPtrType);

var     X,Y,I : integer;

begin

  with Ptr^ do
    begin
    RestoreWindow(Number,0,0);         {brings out the proper window,}
    SelectWindow(Number);              {selects it, and..}
    SetColorBlack;                     {erase the old texts..}
    with PEtxtArray[DPmode] do         {depending on array's display mode.}
      for X:=1 to ArraySize do         {within every PE of the array..}
      for Y:=1 to ArraySize do
        with PE[X,Y] do
        if Code<>0 then                {if the PE has a valid code then..}
          for I:=1 to Lines do         {erases all displayable registers}
              DrawTextW                {values}
              (TextCoord[X,Y].X[I],
               TextCoord[X,Y].Y[I],
               1,Regs_Txt[I]);
    for X:=1 to ArraySize do            {Then, with every PE of the array..}
    for Y:=1 to ArraySize do
      with PE[X,Y] do if Code<>0 then   {if it has a valid code..}
        begin
        Str(X_Reg:6:2,                  {updates its text storages of X,}
            Regs_Txt[2]);
        Str(Out_Regs[2,1]:6:2,          {of Vout,}
            Regs_Txt[3]);
        Str(Out_Regs[2,2]:6:2,          {of Mout,}
            Regs_Txt[4]);
        Str(Out_Regs[3,1]:6:2,          {of Xout,}
            Regs_Txt[5]);
        Str(TAG:1,                      {of TAG}
            Regs_Txt[1]);
        end;
    SetColorWhite;                      {At last, write in the new texts..}
    with PEtxtArray[DPmode] do          {depending on array's display mode.}
      for X:=1 to ArraySize do          {within every PE of the array..}
      for Y:=1 to ArraySize do
        with PE[X,Y] do
        if Code<>0 then                 {if the PE has a valid code then..}
          for I:=1 to Lines do          {rewrites all new registers}
              DrawTextW                 {values}
              (TextCoord[X,Y].X[I],
               TextCoord[X,Y].Y[I],
               1,Regs_Txt[I]);
    StoreWindow(Number);                {Now, stores the updated window.}
    end;

end;
```

```
(***************************************************************************
 *                                                                          *
 *  This procedure simulates a single step of execution of the systolic     *
 *  system of arrays. It first links all necessary IO channels for the      *
 *  current step to the system of arrays, then it feeds data into input     *
 *  buffers, gets data from arrays into output files, then for each PE, it  *
 *  executes its microcodes until the entire system of arrays is traversed. *
 *  At last it will move the result of each PE's micro-execution into its   *
 *  suitable output register and updates the graphics image of each array   *
 *  and the status panel. It really can do that much works in so short a    *
 *  time span.                                                              *
 *    This procedure is called by :                                         *
 *  - main block.                                                           *
 *    This procedure calls :                                                *
 *  - procedure LinkIOFlow().                                               *
 *  - procedure LinkDataFlow().                                             *
 *  - procedure UpdateArray().                                              *
 *  - procedures HEcode1(), HEcode2().                                      *
 *  - procedures NASHcode1(), NASHcode2().                                  *
 *  - procedures LEcode1(), LEcode2()                                       *
 *  - procedures N2Scode(), S2Ncode(), E2Wcode(), W2Ecode().                *
 *                                                                          *
 ***************************************************************************)

procedure SingleStepExec
       (IOPntr : IOPtrtype);

var    SysPtr : ArrayPtrtype;
      I,J,X,Y : integer;

begin

  with StatPtr^ do                    {update status panel's registers}
    begin
    Times:=Times+TimeUnit;            {increments time..}
    Steps:=Steps+1;                   {and step counters}
    LinkDataFlow(LinkPtr,Steps);      {establishes all necessary links and}
    LinkIOFlow(IOPntr,Steps);         {IO channels for this step}
    while IOPntr<>NIL do              {starts at begining of IO linked list}
      with IOPntr^ do                 {for each I/O channel..}
        begin
        if Active then                {if channel is still active, then}
          case IO of                  {depending on the type of IO channel}
          INPUT:                      {for input channel..}
          if EOF(FileVar) then        {if all data in file are read}
             begin
             Close(FileVar);          {then closes input file,}
             Active:=FALSE;           {marks input channel as inactive}
             for I:=1 to ArraySize    {and grounds input buffers.}
                 do InRegs[I]:=0.0;
             end
        else begin
             for I:=1 to ArraySize     {else reads in data on line..}
                 do read(FileVar,
                    InRegs[I]);
             readln(FileVar);          {and go to next line}
             end;
        OUTPUT: begin                  {for output channel..}
                for I:=1 to            {write data to file}
                    ArraySize do
                    write(FileVar,
                      OutRegs[I]^
                      :12:2);
                writeln(FileVar);
                end;
        end;
      IOPntr:=NextIO;                  {then goes to next IO channel}
      end;
  end;
SysPtr:=FixedPtr;                      {start with the 1st array in system..}
while SysPtr<>StatPtr do               {as with all arrays except STATUS..}
  with SysPtr^ do
    begin
    for X:=1 to ArraySize do           {with every single PE of array..}
    for Y:=1 to ArraySize do
        case PE[X,Y].Code of           {depending on its individual code..}
          0: ;                         {do nothing, or..}
          1: N2Scode(PE[X,Y]);         {executes the proper PE's microcode}
          2: E2Wcode(PE[X,Y]);
          3: S2Ncode(PE[X,Y]);
          4: W2Ecode(PE[X,Y]);
          5: HEcode1(PE[X,Y]);
          6: HEcode2(PE[X,Y]);
          7: NASHcode1(PE[X,Y]);
          8: NASHcode2(PE[X,Y]);
          9: LEcode1(PE[X,Y]);
         10: LEcode2(PE[X,Y]);
         11: LEcode3(PE[X,Y]);
   MaxCodes: LEcode4(PE[X,Y]);
        end;
    SysPtr:=Next;                      {then go to the next array}
    end;
SysPtr:=FixedPtr;                      {THEN moves the flow of data}
while SysPtr<>StatPtr do               {of each array except the STATUS}
  with SysPtr^ do                      {by updating each PE's Last_Out}
    begin                              {buffers on all sides and bus..}
    for X:=1 to ArraySize do
    for Y:=1 to ArraySize do
    with PE[X,Y] do
      if Code<>0 then                  {if its code is not 0}
        for I:=1 to MaxRegs do
        for J:=1 to MaxBus do
```

```
            Last_Out[I,J]:=
                 Out_Regs[I,J];
    SysPtr:=Next;                       {then of course, go to next array}
    end;
  InvertWindow;                         {updates graphics image of system
                                        {starts here. First, invert current}
                                        {window to normal..}
  StoreWindow(CurrntPtr^.Number);       {..then stores it.}
  SelectScreen(2);                      {Now, on the RAM screen,}
  ClearScreen;                          {clears it..}
  if CurrntPtr<>StatPtr then            {then, if current window is not..}
    begin                               {the status window, updates it and}
    SelectWorld(FirstWorld);
    UpdateArray(CurrntPtr);
    ClearScreen;                        {clears RAM screen again}
    end;
  with StatPtr^ do                      {then with the status panel,}
    begin
    RestoreWindow(Number,0,0);          {restores it to the RAM screen..}
    SelectWorld(StatusWorld);           {and starts updating the panel..}
    SelectWindow(Number);
    SetColorBlack;                      {by erasing the old status text}
    for X:=1 to 2 do with Boxes[X]
      do DrawTextW(Xdgt,Ytxt,1,Dgt);
    Str(Steps:4,Boxes[1].Dgt);
    Str(Times:9:6,Boxes[2].Dgt);
    SetColorWhite;                      {and write in new status text}
    for X:=1 to 2 do with Boxes[X]
      do DrawTextW(Xdgt,Ytxt,1,Dgt);
    StoreWindow(Number);                {and stores the new panel.}
    end;
  ClearScreen;
  SysPtr:=CurrntPtr^.Next;              {then updates all other windows}
  SelectWorld(FirstWorld);              {to the RAM screen..}
  while SysPtr<>CurrntPtr do
    with SysPtr^ do begin
    if NumBer<>MaxWindowsGlb            {except the status panel..}
    then UpdateArray(SysPtr);
    SysPtr:=Next;
    end;
  if StatPtr<>CurrntPtr then            {which, if it's not the current..}
     RestoreWindow                      {window, restores it last to the..}
     (MaxWindowsGlb,0,0);               {RAM screen}
  CopyScreen;                           {Now, dump contents of RAM screen to}
  SelectScreen(1);                      {the MAIN screen, and select it..}
  RestoreWindow                         {and restore the current window to it}
    (CurrntPtr^.Number,0,0);
  SelectWindow(CurrntPtr^.Number);      {then select current window and..}
  InvertWindow;                         {hilite it.}

end;
```

```
(***************************************************************************
 *                                                                         *
 *  This procedure simulates a single step of execution of the systolic   *
 *  system of arrays. It first links all necessary IO channels for the    *
 *  current step to the system of arrays, then it feeds data into input   *
 *  buffers, gets data from arrays into output files, then for each PE, it *
 *  executes its microcodes until the entire system of arrays is traversed. *
 *  At last it will move the result of each PE's micro-execution into its *
 *  suitable output register and updates the graphics image of each array *
 *  and the status panel. It will keeps executing until a key is hit on the *
 *  keyboard.                                                              *
 *    This procedure is called by :                                        *
 *  - main block.                                                          *
 *    This procedure calls :                                               *
 *  - procedure LinkIOFlow().                                              *
 *  - procedure LinkDataFlow().                                            *
 *  - procedure UpdateArray().                                             *
 *  - procedures HEcode1(), HEcode2().                                     *
 *  - procedures NASHcode1(), NASHcode2().                                 *
 *  - procedures LEcode1(), LEcode2()                                      *
 *  - procedures N2Scode(), S2Ncode(), E2Wcode(), W2Ecode().               *
 *                                                                         *
 ***************************************************************************)

procedure MultiStepsExec;

var    SysPtr : ArrayPtrtype;
       IOPntr : IOPtrtype;
      I,J,X,Y : integer;
          Chr : char;

begin

  if CurrntPtr<>StatPtr then          {first, if current window is not}
     begin                            {the status panel then stores it}
     InvertWindow;                    {as a non-current window and}
     StoreWindow(CurrntPtr^.Number);  {then make the status panel current}
     with StatPtr^ do                 {by inverting it.}
       begin
       SelectWindow(Number);
       InvertWindow;
       StoreWindow(Number);
       end;
     end
  else StoreWindow(CurrntPtr^.        {else stores the status panel as}
                   Number);           {current}
  SelectScreen(2);                    {Then select RAM screen}
  repeat                              {REPEAT all following until a key is}
                                      {pressed..}
    with StatPtr^ do                  {update status panel's registers}
      begin
      Times:=Times+TimeUnit;          {increments time..}
  Steps:=Steps+1;                 {and step counters}
  LinkDataFlow(LinkPtr,Steps);    {establishes all necessary links and}
  LinkIOFlow(IOPtr,Steps);        {IO channels for this step}
  IOPntr:=IOPtr;
  while IOPntr<>NIL do            {starts at begining of IO linked list}
    with IOPntr^ do               {for each I/O channel..}
      begin
      if Active then              {if channel is still active, then}
        case IO of                {depending on the type of IO channel}
        INPUT:                    {for input channel..}
        if EOF(FileVar) then      {if all data in file are read}
           begin
           Close(FileVar);        {then closes input file,}
           Active:=FALSE;         {and marks input channel as inactive}
           for I:=1 to ArraySize  {and grounds input buffers.}
                do InRegs[I]:=0.0;
           end
        else begin
             for I:=1 to ArraySize {else reads in data on line..}
               do read(FileVar,
                       InRegs[I]);
             readln(FileVar);     {and go to next line}
             end;
        OUTPUT: begin             {for output channel..}
                for I:=1 to       {write data to file..}
                    ArraySize do
                    write(FileVar,
                      OutRegs[I]^
                      :12:2);
                writeln(FileVar); {and go to next line}
                end;
        end;
      IOPntr:=NextIO;             {then go to next IO channel}
      end;
  end;
SysPtr:=FixedPtr;                 {start with the 1st array in system..}
while SysPtr<>StatPtr do          {as with all arrays except STATUS..}
  with SysPtr^ do
    begin
    for X:=1 to ArraySize do      {with every single PE of array..}
    for Y:=1 to ArraySize do
        case PE[X,Y].Code of      {depending on its individual code..}
          0: ;                    {do nothing, or..}
          1: N2Scode(PE[X,Y]);    {executes the proper PE's microcode}
          2: E2Wcode(PE[X,Y]);
          3: S2Ncode(PE[X,Y]);
          4: W2Ecode(PE[X,Y]);
          5: HEcode1(PE[X,Y]);
          6: HEcode2(PE[X,Y]);
          7: NASHcode1(PE[X,Y]);
          8: NASHcode2(PE[X,Y]);
          9: LEcode1(PE[X,Y]);
          10: LEcode2(PE[X,Y]);
          11: LEcode3(PE[X,Y]);
    MaxCodes: LEcode4(PE[X,Y]);
         end;
     SysPtr:=Next;                     {then go to the next array}
     end;
  SysPtr:=FixedPtr;                    {THEN moves the flow of data}
  while SysPtr<>StatPtr do             {of each array except the STATUS}
    with SysPtr^ do                    {by updating each PE's Last_Out}
     begin                             {buffers on all sides and bus..}
     for X:=1 to ArraySize do
     for Y:=1 to ArraySize do
     with PE[X,Y] do
       if Code<>0 then                 {if its code is not 0}
         for I:=1 to MaxRegs do
         for J:=1 to MaxBus do
             Last_Out[I,J]:=
                  Out_Regs[I,J];
     SysPtr:=Next;                     {then of course, go to next array}
     end;
  SysPtr:=FixedPtr^.Next;              {start with the first array..}
  ClearScreen;                         {clears the RAM screen..}
  SelectWorld(FirstWorld);             {select the array's world..}
  while SysPtr<>StatPtr do             {for all windows that are not status}
    begin                              {panel or current, updates them to}
    UpdateArray(SysPtr);               {reflect the new values in each}
    SysPtr:=SysPtr^.Next;              {PE's registers.}
    end;
  UpdateArray(FixedPtr);
  with StatPtr^ do                     {Then updates the status panel..}
    begin
    RestoreWindow(Number,0,0);
    SelectWorld(StatusWorld);
    SelectWindow(Number);
    for X:=1 to 2 do with Boxes[X]
      do DrawTextW(Xdgt,Ytxt,1,Dgt);
    Str(Steps:4,Boxes[1].Dgt);
    Str(Times:9:6,Boxes[2].Dgt);
    SetColorBlack;
    for X:=1 to 2 do with Boxes[X]
      do DrawTextW(Xdgt,Ytxt,1,Dgt);
    StoreWindow(Number);
    SetColorWhite;
    end;
  CopyScreen;                          {and copy RAM to displayed screen.}
until keypressed;                      {end of REPEAT}
read(Kbd,Chr);                         {clears stdin of recent key pressed}
ClearScreen;                           {Now that multiple step execution..}
SysPtr:=CurrntPtr^.Next;               {is stop, clears the RAM screen to..}
while SysPtr<>CurrntPtr do             {start re-displaying all system in}
  begin                                {the same order before execution..}
  if SysPtr<>StatPtr then              {starting with restoring all non-}
```

```
      RestoreWindow(SysPtr^.                    {current, non-status windows first..}
                   Number,0,0);
    SysPtr:=SysPtr^.Next;
    end;
  if StatPtr<>CurrntPtr then                    {then restore status panel..}
     with StatPtr^ do
       begin
       RestoreWindow(Number,0,0);
       SelectWindow(Number);
       InvertWindow;
       StoreWindow(Number);
       end;
  CopyScreen;                                   {then updates displayed screen..}
  SelectScreen(1);                              {and at last, select displayed screen}
  with CurrntPtr^ do                            {to restore current window}
    begin
    RestoreWindow(Number,0,0);
    SelectWindow(Number);
    end;
  if CurrntPtr<>StatPtr then
     InvertWindow;                              {and invert it if it's not already}
                                                {invert, meaning if the current}
                                                {window is not the status panel}

end;
```

```
******************************************************************************
*                                                                            *
*  This is the script file for the simulation of Nash's array solving        *
*  example (A.4).  It allows SAGS to produces the sequence of snapshots B.1  *
*  with the data and control files below.                                    *
*  Remove all comments before using them with SAGS.                          *
*                                                                            *
******************************************************************************

ARRAYSIZE :
3 .
SYSTEMSPECS :
1 1 1 21 129 ,
Pecodes :  7  8  8
           0  7  8
           0  0  7 ;
2 0 2 37 129 ,
Pecodes :  8  8  8
           8  8  8
           8  8  8 ;
3 4 2 21 29 ,
Pecodes :  0  0  1
           0  1  1
           1  1  1 ;
4 4 2 37 29 ,
Pecodes :  0  0  1
           0  1  1
           1  1  1 ;
5 3 2 37 229 ,
Pecodes :  1  1  1
           1  1  0
           1  0  0 .
INFILES :
triang34 3 1 1 1 ,
trtag3 3 1 2 1 ,
square34 4 1 1 4 ,
sqtag3 4 1 2 4 .
OUTFILES :
result 5 3 1 14 .
SETUP :
1
NorthInput : 3 3 1 1 ;
2
NorthInput : 4 3 1 1 ,
WestInput : 1 2 1 1 ;
5
NorthInput : 2 3 1 1 .
```

```
****************************************************************************
*                                                                          *
*  Infile triang34.                                                        *
*  Contains the input data flow to be fed into the triangular array of the *
*  system.                                                                 *
*                                                                          *
****************************************************************************

  1.00        2.00        3.00
  0.00        4.00        7.00
  2.00        1.00        3.00
 -1.00        0.00        0.00
  0.00       -1.00        0.00
  0.00        0.00       -1.00

****************************************************************************
*                                                                          *
*  Infile square34.                                                        *
*  Contains the input data flow to be fed into the square array of the     *
*  system.                                                                 *
*                                                                          *
****************************************************************************

  5.00        0.00        0.00
  9.00        0.00        0.00
  7.00        0.00        0.00
  0.00        0.00        0.00
  0.00        0.00        0.00
  0.00        0.00        0.00

****************************************************************************
*                                                                          *
*  Infile trtag3.                                                          *
*  Contains the control signals necessary for the triangular array of the  *
*  system.                                                                 *
*                                                                          *
****************************************************************************

  0.00        0.00        0.00
  0.00        0.00        0.00
  0.00        0.00        0.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        0.00
  1.00        0.00        0.00
```

```
*****************************************************************************
*                                                                           *
*  Infile sqtag3.                                                           *
*  Contains the control signals necessary for the square array of the      *
*  system.                                                                  *
*                                                                           *
*****************************************************************************

  0.00        0.00        0.00
  0.00        0.00        0.00
  0.00        0.00        0.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        0.00
  0.00        0.00        0.00
  0.00        0.00        0.00
  0.00        0.00        0.00
```

```
*****************************************************************************
*                                                                           *
*  This is the script file for the simulation of Chuang and He's array      *
*  solving example (A.2).  It allows SAGS to produces the sequence of       *
*  snapshots B.2 with the data and control files below.                     *
*  Remove all comments before using them with SAGS.                         *
*                                                                           *
*****************************************************************************

ARRAYSIZE :
3 .
SYSTEMSPECS :
1 1 1 22 121 ,
Pecodes :  5  6  6
           0  5  6
           0  0  5 ;
2 0 2 38 121 ,
Pecodes :  6  6  6
           6  6  6
           6  6  6 ;
3 4 2 22 21 ,
Pecodes :  0  0  1
           0  1  1
           1  1  1 ;
4 4 2 38 21 ,
Pecodes :  0  0  1
           0  1  1
           1  1  1 ;
5 3 2 38 221 ,
Pecodes :  1  1  1
           1  1  0
           1  0  0 .
INFILES :
triang32 3 1 1 1 ,
trtag3 3 1 2 1 ,
square32 4 1 1 4 ,
sqtag3 4 1 2 4 .
OUTFILES :
result 5 3 1 14 .
SETUP :
1
NorthInput : 3 3 1 1 ;
2
NorthInput : 4 3 1 1 ,
WestInput : 1 2 1 1 ;
5
NorthInput : 2 3 1 1 .
```

```
*****************************************************************************
*                                                                           *
*  Infile triang32.                                                         *
*  Contains the input data flow to be fed into the T array of the system.  *
*                                                                           *
*****************************************************************************

 -1.00        5.00       -3.00
  3.00        4.00        1.00
  6.00        7.00       -2.00
 -1.00        2.00       -4.00
 -3.00       -4.00        1.00
  5.00       -3.00       -2.00

*****************************************************************************
*                                                                           *
*  Infile square32.                                                         *
*  Contains the input data flow to be fed into the S array of the system.  *
*                                                                           *
*****************************************************************************

 -2.00       -7.00        6.00
  1.00        3.00        1.00
  5.00        9.00        4.00
  2.00        1.00       -5.00
  2.00        4.00        6.00
 -3.00        2.00        9.00

*****************************************************************************
*                                                                           *
*  Infile trtag3.                                                           *
*  Contains the control signals necessary for the T array of the system.   *
*                                                                           *
*****************************************************************************

  0.00        0.00        0.00
  0.00        0.00        0.00
  0.00        0.00        0.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        1.00
  1.00        1.00        0.00
  1.00        0.00        0.00
```

```
******************************************************************************
*                                                                            *
*  Infile sqtag3.                                                            *
*  Contains the control signals necessary for the S array of the system.     *
*                                                                            *
******************************************************************************

  0.00          0.00          0.00
  0.00          0.00          0.00
  0.00          0.00          0.00
  1.00          1.00          1.00
  1.00          1.00          1.00
  1.00          1.00          1.00
  1.00          1.00          1.00
  1.00          1.00          1.00
  1.00          1.00          0.00
  0.00          0.00          0.00
  0.00          0.00          0.00
  0.00          0.00          0.00
```

```
******************************************************************************
*                                                                            *
*  This is the script file for the simulation of a double arrays system of  *
*  our own design.  This system is shown in the sequence of snapshots B.3   *
*  solving example (A.3).  It uses the data and control files below.        *
*  Remove all comments before using them with SAGS.                         *
*                                                                            *
******************************************************************************

ARRAYSIZE :
2 .
SYSTEMSPECS :
1 0 1 19 106 ,
Pecodes :  9 11
          12 10 ;
2 0 2 30 106 ,
Pecodes :  4  4
           4  4 ;
3 0 2 40 106 ,
Pecodes :  4  4
           4  4 ;
4 0 2 50 106 ,
Pecodes :  4  4
           4  4 ;
5 0 1 19 172 ,
Pecodes :  9 11
          12 10 ;
6 0 2 30 172 ,
Pecodes :  4  4
           4  4 ;
7 0 2 40 172 ,
Pecodes :  4  4
           4  4 ;
8 0 2 50 172 ,
Pecodes :  4  4
           4  4 ;
9 4 2 19 37 ,
Pecodes :  0  1
           1  1 ;
10 3 2 19 242 ,
Pecodes :  1  1
           1  0 .
INFILES :
data241 9 1 1 1 ,
control1.24 9 1 2 1 ,
control2.24 5 1 2 14 .
OUTFILES :
result 10 3 1 28 .
SETUP :
1
WestInput : 4 2 1 1 ,
NorthInput : 9 3 1 1 ;
```

```
2
WestInput : 1 2 1 1 ;
3
WestInput : 2 2 1 1 ;
4
WestInput : 3 2 1 1 ;
5
WestInput : 8 2 1 1 ,
NorthInput : 1 3 14 14 ;
6
WestInput : 5 2 1 1 ;
7
WestInput : 6 2 1 1 ;
8
WestInput : 7 2 1 1 ;
10
NorthInput : 5 3 1 1 .
```

```
*************************************************************************
*                                                                       *
*  Infile data241.                                                      *
*  Contains the input data flow to be fed into the first array of the   *
*  system.                                                              *
*                                                                       *
*************************************************************************

  2.00    -1.00
  4.00    -2.00
 -3.00    -4.00
  6.00    -6.00
 -1.00     1.00
 -2.00     2.00
 -1.00    -1.00
 -1.00     1.00
  3.00     0.00
  7.00     0.00
  1.00     5.00
  8.00     0.00
 -2.00     1.00
 -3.00     3.00
 -1.00     0.00
 -4.00    -3.00
 -8.00     3.00
-20.00     5.00
 -2.00    -9.00
  4.00     7.00
  1.00     3.00
  0.00    -4.00
  2.00     1.00
  1.00    -3.00
  0.00     3.00
  1.00     6.00
  7.00     8.00
  4.00     2.00
 -5.00     7.00
  1.00     7.00
  3.00     0.00
 -1.00     9.00
```

```
*************************************************************************
*                                                                       *
*  Infile controll.24.                                                  *
*  Contains the control signals necessary for the first array of the system. *
*                                                                       *
*************************************************************************

13 0
12 0
12 0
12 0
 8 0
 8 0
 8 0
 8 0
 1 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 1 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 1 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
```

```
************************************************************************
*                                                                      *
*  Infile control2.24.                                                 *
*  Contains the control signals necessary for the second array of the  *
*  system.                                                             *
*                                                                      *
************************************************************************

13 0
12 0
 8 0
 8 0
 8 0
 8 0
 8 0
 8 0
 1 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 1 0
 0 0
 0 0
 0 0
 0 0
 0 0
```