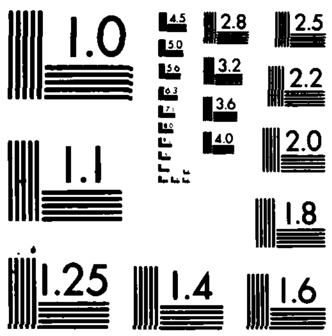

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0

0

INSTITUTE FOR COMPUTATIONAL MATHEMATICS AND APPLICATIONS

Technical Report ICMA-83-66

AN ABSTRACT SYSTOLIC MODEL AND ITS APPLICATION TO THE DESIGN OF FINITE ELEMENT SYSTEMS*)

by

Rami G. Melhem

Department of Mathematics and Statistics

University of Pittsburgh

1

Technical Report ICMA-83-66

AN ABSTRACT SYSTOLIC MODEL AND ITS APPLICATION TO THE DESIGN OF FINITE ELEMENT SYSTEMS*)

by

Rami G. Melhem

Department of Mathematics and Statistics
and
Department of Computer Science
University of Pittsburgh

*)This work was in part supported by the Office of Naval Research under Contract N00014-80-C-0455.

ABSTRACT

An abstract model is suggested to describe precisely systolic networks and to verify their operation. The data items appearing on the communication links of such networks at consecutive time units are represented by data sequences and the operation performed by the network-cells are modeled by a system of equations involving operations on sequences. The input/output relations, which descirbe the global effect of the computations performed by the network, are obtained by solving the corresponding system of sequence equations. This input/output description can then be used to verify the operation of the network.

The model is supplemented with a simple computer language that may be used to express any system of causal equations describing the operation of a systolic network. An interpreter is developed for this language to solve such a system for specific forms of the inputs and to produce the corresponding outputs. The application of this interpreter to the computational assessment of a given systolic network is equivalent to the simulation of its execution.

The abstract model is then applied to the specification and verification of a systolic machine for the computation of the elemental arrays in finite element analysis. Finally, possible organizations for complete finite element systems are suggested based on the idea of pipelining the computations associated with the different elements.

ACKNOWLEDGMENT

I would like to express my appreciation to Dr. Mary Lou Soffa for her comments on the fourth chapter of this thesis, and to the members of my doctorate committee - Dr. Alfs Berztiss and Dr. Errol Lloyd - for their help-full comments and unfailing courtesy. I wish also to express my gratitude to the Co-chairman of the committee - Dr. Henry Chuang - for his excellent advice and for the fruitful discussions that was of great help to me, especially in the early part of this research. My deepest indebtedness, however, is to my advisor - Dr. Werner C. Rheinboldt - who taught me the meaning of research. This thesis could not have been completed without his continuous advice, encouragement and suggestions.

My wife Mona deserves more than thanks. Her love and encouragement during my graduate study were invaluable. This thesis is as much hers as it is mine.

Finally, I have to admit that the joy of watching my son Samer growing gave me the energy and the enthusiasm that I needed throughout my research.

The work described in this thesis was supported in part by the Office of Naval Research under contract number N00014-80-C-0455.

TABLE OF CONTENTS

1. INTRODUCTION.

In the past few years, the concept of systolic architectures [33] became increasingly important and has been intensively studied for the design of computer networks that utilize natural parallelism by moving the data regularly in the network. This type of architectures has two properties desirable in VLSI implementations, namely, regularity and local interconnections.

Although the concept of systolic architectures is very well developed, few techniques appear to be known for a formal specification and verification of such networks. In fact, in most papers on systolic networks, very little formalism is used, and the reader is usually left with a few diagrams and experimental evidence. At best, an ad-hoc proof technique is developed for some examples but is not generally usable.

By treating systolic networks as a collection of communicating, parallel processes, some of the techniques for the verification of distributed systems (see for example [39]) may be applied for the verification of some correctness properties of systolic networks [43, 16]. However, this approach does not make use of the special properties of systolic networks, and hence, gives only rather general results.

In [11], a formal approach for the representation of computational networks was proposed. This approach was elaborated upon in [26, 27, 53] where the so-called wave-front notation was used to map algorithmic descriptions into systolic implementations. Although this notation provides a powerful tool that can be used in the automatic design of systolic arrays [54], it does not appear to have the flexibility needed to describe general systolic networks.

The first part of this dissertation, namely Chapters 2, 3, 4 and 5, concerns a formal treatment of systolic networks. We start in Chapter 2 by introducing an abstract model for systolic computations. The model is based on the representation of the data items appearing on each communication link as an infinite data sequence. Moreover, the operation of each computational cell is modeled by a set of equations involving operators on data sequences. This sequence approach separates the time and the space dimensions of systolic networks and distinguishes between the networks functions and the specific details of the computations, and thus turns out to lead to a clear and precise specification of systolic computations.

The system of all equations that model the cells in a systolic network represents an implicit relation between the inputs and the outputs of the network. By solving this system of equations, we obtain an explicit formula for the outputs of the network in terms of the inputs. This formula is called the network I/O description. The output of a specific computation may be found by substituting the corresponding particular input into the I/O description. Then a verification of the computation requires only a comparison of the resulting output with the specification of the expected output. This verification technique is described in Chapter 3, where it is also applied to different systolic networks.

The applicability of our verification technique depends largely on our ability to manipulate sequence operators and to solve systems of sequence equations. The solvability of the equations is discussed in Section 3.4 where we show that it is always possible to obtain an analytical expression for the solution of systems of equations resulting from our model of systolic computations. However, the analytical solution may sometimes be very complicated and thus not practical. For this reason, we introduce in Chapter 4 a computer solver that may be used to find numerically the solutions of sequence equations.

Finally, in Chapter 5, we conclude the first part of the dissertation by discussing some topics that demonstrate the power of the abstract model and the flexibility of the sequence notation.

It should be noted here that our abstract model carries some properties of a model called "automaton networks" [19] which in turn, is a modification of the von Neumann cellular arrays [52,8]. It also carries some properties of an abstract model [35] used by Leiserson and Saxe to prove that any synchronous system can be converted into an equivalent systolic system. Moreover, the objective of the model is similar to that of another model developed independently by Chen and Mead [10]. Both models separate the network function from the specific details of a certain computation and allow for a precise specification and a formal verification of systolic networks. However, in our model we follow an algebraic approach, while the model in [10] is oriented toward a procedural approach. More specifically, a procedural language is used for the specification of both the network and its inputs, and the description of the output is obtained by applying fixed point theory [49] for finding the "least solution" of systems of recursive functions.

The second part of the dissertation consists of the Chapters 6, 7, 8 and 9. In this part, we apply the abstract systolic model to the specification and verification of a special purpose system for finite element analysis.

Very briefly, the finite element method (see e.g. [57]) is a technique for solving a partial differential equation on a certain domain Q with given conditions on the boundary of Q. In the case of linear equations, it involves essentially the following four basic steps: 1) The generation of a finite element mesh that divides Q into m finite elements. 2) The generation of elemental stiffness matrices and load vectors for each finite element. 3) The assembly of the global stiffness matrix H and load vector b. 4) The solution of the resulting linear system of equations Hu=b.

Due to its various applications in engineering and mathematics, many finite element software systems have been developed (see e.g. [44]) and widely used for the solution of a variety of boundary value problems. However, the time required to complete any finite element computation on a serial computer may become extremely large for many realistic, practical problems. This usually imposes severe limitations on the size and type of the problems that can be handled, and leads engineers to use lower degrees of approximation and hence less accurate models. This is especially true if a design procedure is based on the results of running a finite element solver repeatedly, with a certain decision to be taken after each run (interactive design), or if the result of the analysis is to be reported within certain time limits, as is often the case in military applications.

For this reason, many researchers have considered the use of some type of parallel processing in the finite element analysis. In fact, in a survey on highly parallel computations [20], Haynes et al. stressed the fact that one of the most important areas in which parallel computations can be explored is the solution of partial differential equations, especially the finite element analysis.

The use of array processors for speeding up the finte element computations were considered by many researchers; for instance, Noor et al. [41,40] studied algorithms for performing a finite element dynamic analysis on the CDC Star-100 computer. Along the same lines, Kamel et al. [48,36] studied the usefulness of array processors, combined with mini and super mini computers, in finite element computations. Both studies showed that only a limited speed up can be obtained via array processors, especially in the generation of the stiffness matrices and load vectors.

Similarly, the use of general purpose multiprocessors in the solution of the linear systems appearing in finite element computations were studied. For example, in [14, 17], the Cm* multiprocessor was used to solve linear systems by iterative techniques. The experiments showed that only a limited number of processors can be used if congestion is to be avoided in inter-process communications. No studies have been reported in the literature on the use of a general multiprocessor system to generate the stiffness matrices or load vectors.

In [47, 55], the problem was partitioned into a large number of separate processes, and each process was assigned to a processor. This system also incorporates the use of a posteriori error estimates and a corresponding refinement of the finite element grid. Although this adaptive approach appears to be very attractive for parallel processing, it was shown in [56] that various parallel configurations, discussed in the literature, are not expected to give a satisfactory gain in the processing speed since the times for communication and data movements between the different processors dominate the running time.

The most significant attempt in this area is the design of a finite element machine at the Institute for Computer Applications in Science and Engineering (ICASE), at the NASA Langley Research Center [29, 28, 45]. In this project, a 16 bit microprocessor with multiply and divide hardware is assigned to each node in the finite element grid. Each processor is directly connected to eight immediate neighbors, and a global bus connects all the processors in the system. The motivation for this connection is that most of the data required to complete the calculations at each node come from the immediate neighbors of that node. Such a machine is planed to have 1024 processors and is still subject to experimental studies.

Although its idea is attractive, the finite element machine has many serious drawbacks: First of all, the direct relation between the number of processors and the number of nodes in the finite element grid imposes severe limitations on the size of the grid. Secondly, a suitable mapping has to be found which associates the nodes with the processors. In [4], S. H. Bokhari showed a possible method for accomplishing this task but stated that the problem becomes very complex and time consuming for regular meshes of size larger than 30x30, and is even more complex for irregular meshes. Finally, in [30] the authors concluded that some additional tree like hardware, besides the global bus, is needed to implement global functions such as the sum and maximum over quantities distributed over the nodes [5]. According to Jordan [28], the finite element machine is most suitable if the interconnections between its processors follow the same pattern as the finite element mesh which is a very rigid restriction.

By carefully analyzing the different steps in the linear finite element analysis, we may note that the involved computations are highly regular, and that they can be divided into separate phases, where each phase depends only on the preceding phase. Hence the data can be transferred from phase to phase in a pipe-lined fashion. The computation within each phase is also well structured and mostly compute bound, which makes it a very suitable application for systolic architectures.

After briefly introducing the finite element analysis in Chapter 6, we apply in Chapter 7, the abstract model to the specification and verification of a systolic system for the generation of the elemental arrays. This generation is often a major time consuming task in workaday finite element computations.

In Chapters 8 and 9, we describe the architecture of a complete finite element system. The suggested system is based on the idea of pipelining the computations associated with the elements rather than processing them in parallel on an array of processor, as is the case in the machine described in [29]. This latter machine uses an iterative scheme for the solution of the linear system of equations resulting from the finite element formulation. It reduces the processing time considerably by employing a number of processors proportional to the size of of the problem. On the other hand, the pipeline/systolic approach may be applied with either direct or iterative solution schemes, and it results in an architecture that is not dependent on the number of elements in the mesh that covers the domain of the problem. Basically, it uses a fixed number of processors to complete the analysis in a time proportional to the size of the problem. Each approach has its own merits and may be suitable for certain applications.

It should be mentioned, however, that the architecture of the system that applies a direct solution scheme has the disadvantage of being dependent on the bandwidth of the global stiffness matrix. Systems that apply iterative solution schemes do not share this limitation but are relatively slower. In fact, it turns out that the time for completing each iteration step is proportional to the number of elements in the mesh.

2. AN ABSTRACT SYSTOLIC MODEL.

Systolic architectures, pioneered by H. T. Kung, are becoming increasingly attractive due to continuous advances in VLSI technology. This type of network architectures has two properties very desirable in VLSI implementations; namely, regularity and local interconnections.

A systolic network can be viewed as a network composed of a few types of computational cells, regularly interconnected via local data links and organized such that streams of data flow smoothly within the network. For an introduction to systolic architectures, we refer to [33,31] where further references to specific examples are given.

As an introductory example, we briefly review a simple systolic network for the computation of one dimensional convolution expressions [31]. More specifically, given a sequence of numbers $\{x_1, x_2, \dots x_n\}$, and a sequence of weights $\{w_1, w_2, \dots w_k\}$, we want to compute the sequence $\{y_1, y_2, \dots y_{n+1-k}\}$ where each y_i is defined by:

$$y_i = \sum_{j=1}^{k} w_j \; x_{i+j-1} \tag{2.1}$$

Figure 2.1 shows the building cell of the 1-D convolution network under discussion. It is a multiply/add cell with a one word memory to store a real number w.

At each clock pulse, the cell receives two input data items: x_{in} and y_{in}, performs its computation and delivers at the next clock pulse the outputs $x_0 = x_{in}$ and $y_o = y_{in} + w\, x_{in}$. Figure 2.2 shows three such cells connected into a network that performs the convolution calculation for the case k=3. The elements $x_1, x_2, \dots x_n$ are pumped in at the left end of

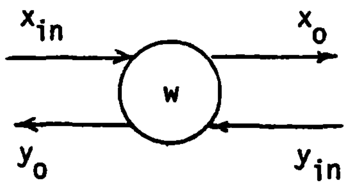


Figure 2.1
A multiply/add cell.

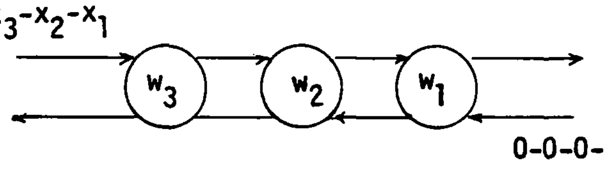


Figure 2.2
A 1-D convolution network.

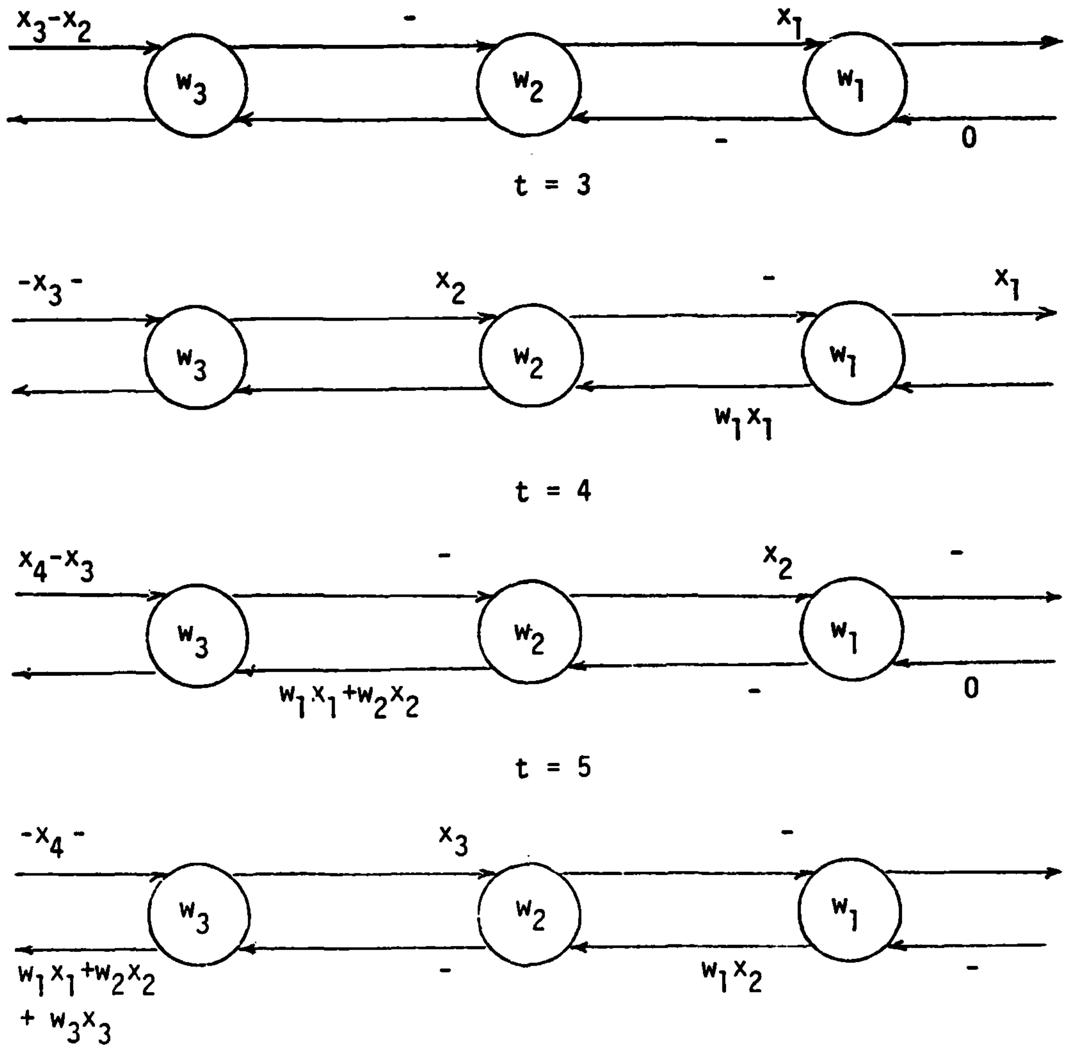


Figure 2.3 - The operation of the 1-D convolution network

the network, each separated from the other by one time unit, and zeroes are pumped in at the right end. To illustrate the operation of the array, we show in Figure 2.3 the relative location and value of each data item at times t=3,4,5 and 6, where t=1 is the time at which the network started its execution. By following the data paths, it is easy to ascertain that the output of the array will include the sequence $(y_1, y_2, .. y_{n+1-k})$.

In order to specify and verify formally the operation of any systolic network, we have to consider both the spatial topology of the network and the timing of the data movement on its communication links. In this chapter, we suggest a formal model designed specifically to conveniently separate the space dimension of the systolic architecture from its time dimension. The separation makes the specification of systolic networks clearer and leads to a formal technique for the verification of their operations. We start by considering the mathematical basis of the model.

2.1. Data sequences and causal operators.

We define a data sequence to be an infinite sequence whose elements are members of the set $R_\delta = R \cup \{\delta\}$, where R is the set of real numbers and δ denotes a special element, not belonging to R, called the "don't care element". We extend any operator defined on R to R_δ in one of the following two ways:

1) By adding the rule that the result of any operator involving δ is δ. For example, we extend the usual arithmetic operations 'op' = '+', '−', '*' or '/', by adding the following rule

$$\delta \text{ 'op' } x = x \text{ 'op' } \delta = \delta \qquad \text{for all } x \in R_\delta$$

This class of operators on R_δ will be called δ-regular operators.

2) By treating δ as a special symbol that affects the result of the opera-

tion. This class will be called non δ-regular operators. For example, we will consider later the binary operator ⊕ such that for any $x,y \epsilon R_\delta$,

$$x \oplus y = x + y, \quad \text{if } x,y \neq \delta; \quad x \oplus \delta = \delta \oplus x = x \tag{2.2}$$

Two other non δ-regular operators, that will be used in Section 3.3, are the operators $\min_\delta$ and $\max_\delta$ defined on an ordered pair (x,y), $x,y \epsilon R_\delta$ by

$$\min_\delta(x,y) = \begin{cases} \min(x,y) & \text{if } x,y \neq \delta \\ y & \text{if } x=\delta \text{ or } y=\delta \end{cases}$$

and

$$\max_\delta(x,y) = \begin{cases} \max(x,y) & \text{if } x,y \neq \delta \\ x & \text{if } x=\delta \text{ or } y=\delta. \end{cases}$$

where min() and max() carry the usual meaning on R. The reason for distinguishing between δ-regular and non δ-regular operators will become clearer as we proceed with the discussion.

Let N be the set of positive integers. Then any data sequence η is defined as a mapping from N to R_δ; that is, the image element $\eta(i)$, $i \epsilon N$, is the i^{th} element in the sequence. The set of all data sequences, that is the set of all such mappings, will be denoted by $R_\delta^* = \{ \eta \mid \eta{:}N \rightarrow R_\delta \}$.

Any operation defined on R_δ is extended to R_δ^* by applying the operation element-wise to the elements of the sequences with δ being the result of any undefined operation. For example, if 'op' is a binary operation defined on R_δ, then for all $\eta_1, \eta_2 \epsilon R_\delta^*$, we have $\eta_1 \text{'op'} \ \eta_2 = \eta_3$, where for all $i \epsilon N$, $\eta_3(i)$ is given by

$$\eta_3(i) = \begin{cases} \eta_1(i) \text{ 'op' } \eta_2(i) & \text{if } \eta_3(i) \text{ is defined} \\ \delta & \text{otherwise.} \end{cases}$$

We will also use scalar operations on sequences. For example, the scalar product of a sequence $\eta \epsilon R_\delta$ and a number $w \epsilon R$ is defined as the sequence $\zeta = w \cdot \eta \ \epsilon R_\delta^*$ for which $\zeta(i) = w \ \eta(i)$, $i \epsilon N$.

Given the previous definition of data sequences. we define the set of bounded data sequences $\bar{R}_\delta \subset R_\delta^*$ to contain those sequences having only a finite number of non-δ elements. It is then natural to introduce the termination function $T:\bar{R}_\delta \rightarrow N$ such that for any $\eta \in \bar{R}_\delta$, $T(\eta)$ is the position of the last non-δ element in η; in other words:

$$\text{for any } \eta \in \bar{R}_\delta, \; T(\eta)=i \leftrightarrow \eta(i) \neq \delta \text{ and } \eta(j)=\delta \text{ for } j>i.$$

In this dissertation, we will only consider bounded data sequences. These will be denoted by small greek letters and simply referred to as sequences. Two special sequences will be repeatedly used, namely the don't care sequence δ^* and the zero sequence ι. The first is defined by $\delta^*(t)=\delta$ for all t, and the second by $\iota(t)=0$ for $1 \leq t \leq T(\iota)$ and any arbitrary large $T(\iota)$.

In addition to the operators extended from R_δ to $\bar{R}_\delta$, we may also define operators directly on $\bar{R}_\delta$. In general, an n-ary sequence operator Γ is a transformation $\Gamma:[\bar{R}_\delta]^n \rightarrow \bar{R}_\delta$ where $[\bar{R}_\delta]^n=\bar{R}_\delta \times \bar{R}_\delta \times \cdots \bar{R}_\delta$ is the cartesian product space of n copies of $\bar{R}_\delta$. Many operators of this type will be defined in Section 2.4; we introduce here only a basic unary operator that will be used frequently in our discussion, namely the shift operator $\Omega^k:\bar{R}_\delta \rightarrow \bar{R}_\delta$ defined for any $k \geq 1$ by

$$\Omega^k \xi = \eta$$

where

$$\eta(i) = \begin{cases} \delta & \text{if } i \leq k \\ \xi(i-k) & \text{if } i>k \end{cases}$$

More descriptively, Ω^k inserts k δ-elements at the beginning of a sequence. For example if $\xi=a_1,a_2,a_3,a_4,\delta,\delta,\ldots$ then $T(\xi)=4$ and

$$\xi(i) = a_i \qquad 1 \leq i \leq T(\xi)$$
$$\Omega^3 \xi = \delta,\delta,\delta,a_1,a_2,a_3,a_4,\delta,\delta,\delta,\ldots$$

It is easy to verify that the termination function generally satisfies

$$T(\Omega^k \xi) = T(\xi)+k$$

It is also clear that we can define a sequence operator by combining previously defined sequence operators. For example we might define an operator $\Gamma:\overline{R}_\delta \times \overline{R}_\delta \times \overline{R}_\delta \rightarrow \overline{R}_\delta$ as follows:

$$\Gamma(\xi,\eta,\zeta) = \Omega\ [\xi + \eta * \zeta]$$

where square brackets are used for grouping and parenthesis for enclosing the arguments of the operator.

We define a causal operator to be any n-ary sequence operator $\Gamma:[\overline{R}_\delta]^n \rightarrow \overline{R}_\delta$ which satisfies the causality property in the sense that the i^{th} element of any of its operands can only affect the j^{th} element of its image for $j>i$. In order to formulate this more precisely, assume that for any given sequences $\eta_r \epsilon \overline{R}_\delta$ r=1,2,...,n, the image under Γ is $\xi=\Gamma(\eta_1,...,\eta_r,...\eta_n)$. *Then* Γ *is a causal operator if the replacement of any operands* η_r, $1 \leq r \leq n$, by other sequences η_r' satisfying

$$\eta_r'(t) = \eta_r(t) \qquad 1 \leq t < i$$

results in an image sequence $\xi' = \Gamma(\eta_1,...,\eta_r' ...\eta_n)$ for which

$$\xi'(t) = \xi(t) \qquad 1 \leq t \leq i$$

In other words, the value of $\xi(i)$ depends only on the first i-1 elements of η_r, $1 \leq r \leq n$.

Similarly, we may define weakly-causal operators for which the i^{th} element of the image sequence $\xi(i)$ depends only on the first i elements of the operands η_r, $1 \leq r \leq n$ instead of the first i-1 elements. With this, it is easily seen that the combination $\Gamma^1\ \Gamma^2$ (or $\Gamma^2\ \Gamma^1$) of a causal operator Γ^1 and a weakly-causal operator Γ^2 is a causal operator. For instance, the shift operator Ω^k is causal and hence, the combined operator $\Omega^k\ \Gamma^2$ is causal, for any weakly causal operator Γ^2.

In the previous discussion, we only considered the space $\bar{R}_\delta$ of sequences of real numbers. However, other sequence spaces may be defined as well, by starting with a set different from the set of real numbers R. For example, if we start from the set of boolean truth values $B = \{true, false\}$, then we may define the space of boolean sequences $\bar{B}_\delta = \{\xi : \xi : N \rightarrow B \cup \{\delta\}\}$.

2.2. The abstract model.

We begin the specification of the mathematical model used in our verification technique with the definition of a loop-less, directed multigraph $G(V,E,\varphi_-,\varphi_+)$ as a structure composed of

(a) a set V of nodes;

(b) a set E of directed edges;

(c) two functions $\varphi_-,\varphi_+ : E \rightarrow V$, satisfying the condition that for any edge $e \in E$,

$$\varphi_-(e) \neq \varphi_+(e) \tag{2.3}$$

For each edge $e \in E$, the nodes $\varphi_-(e)$ and $\varphi_+(e)$ are the source and destination node, respectively, of that edge. Clearly, the condition (2.3) prevents any nodal loops in the graph. This definition of a multigraph allows any two nodes to be connected by more than one edge in the same direction, a property that may be useful when we represent systolic networks by this abstract model.

As usual in graph terminology, for any node $v \in V$, the edges $\{e : \varphi_-(e) = v\}$ directed out of v are termed the OUT edges of v, while the edges $\{e : \varphi_+(e) = v\}$ directed into v are termed the IN edges of v. Accordingly, the IN-degree and OUT-degree of v are the number of IN edges and OUT edges of v, respectively. Any node $v \in V$ with IN-degree zero or OUT-degree zero is called a source or a sink, respectively. All other nodes are called interior nodes of G. We shall use the notation V_S, V_T and V_I for the subsets of V containing the source, sink and interior nodes of V, respectively. Of course, the condition $V_S \cup V_T \cup V_I = V$ is always satisfied.

With this notion of a multigraph, we define our abstract systolic model to be composed of the following components.

[A1] A multigraph $G(V,E,\varphi_-,\varphi_+)$.

[A2] A coloring function $col{:}E \rightarrow C_E$, which maps E into a given finite set of colors C_E, and hence assigns a color to each edge in E. The coloring function is assumed to satisfy the condition that the different IN edges of any node in $V_I \cup V_T$ have different colors, and correspondingly that the different OUT edges of any node in $V_S \cup V_I$ have different colors. Edge colors will be denoted by lower case letters.

[A3] For each edge $e \in E$, a sequence ξ_e of a given sequence space is specified.

[A4] For each interior node $v \in V_I$ with IN degree m and OUT degree n, we are given n causal m-ary operators $\Gamma_v^i{:}[\bar{R}_\delta]^m \rightarrow \bar{R}_\delta$ which specify the "node I/O description". More specifically, if η^i, $i=1,\cdots,m$ and ξ^i, $i=1,\cdots,n$ are the sequences associated with the IN and OUT edges of v, respectively, then the n relations

$$\xi^i = \Gamma_v^i(\eta^1,\cdots,\eta^m) \qquad i=1,\cdots,n$$

are the I/O description of v. The different IN and OUT edges of v are distinguished in the I/O description by their colors.

Since [A2] ensures that all edges terminating at a given node v have different colors, it follows that any edge $e \in E$ may be uniquely identified by a pair (y,v), where y=col(e) and $v=\varphi_{+}(e)$. To simplify the notation, the pair (y,v) will often be written in the form y_v, and the sequence associated with that edge will be identified by the symbol η_v, where we replaced the letter y by its corresponding greek letter η.

For practical applications, it is generally desirable to identify the nodes of the network by appropriate labels which correspond to the problem at hand. This means that we introduce a set L of labels together with a one-to-one function $\psi: V \rightarrow L$ from V onto L. In our examples, we usually identify the nodes directly with their labels.

2.3. The general systolic network.

By giving a physical interpretation to each component in the general abstract model we obtain a general systolic network. The basic idea of this interpretation may be summarized as follows:

Each interior node represents a computational cell and each source/sink node corresponds to an input/output cell for the overall network. To distinguish in our figures the computational cells from the I/O cells, we depict computational cells by circles and I/O cells by squares.

Each edge $x_v \in E$ represents a unidirectional communication link between the two cells it connects. The sequence associated with x_v then comprises the data items that appeared on it in consecutive time units. More specifically, if ξ_v is the sequence associated with x_v, then the i^{th} element of ξ_v, namely $\xi_v(i)$ is the data item that appeared on x_v at time $t=i$ units, where $t=1$ is the time at which the network started its operation.

Clearly, the sequence space from which the sequences are taken corresponds to the type of data items that may be carried on the communication links. In this dissertation, we will consider only networks in which the communication links may carry real numbers. In other words, any sequence is assumed to be in the space $\bar{R}_\delta$.

For an interior node, the node I/O description describes the computations performed by the cell corresponding to that node. We illustrate this with two simple examples:

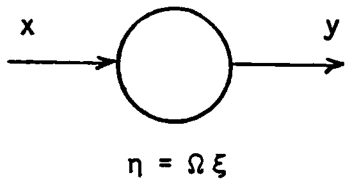


Figure 2.4 - A delay cell

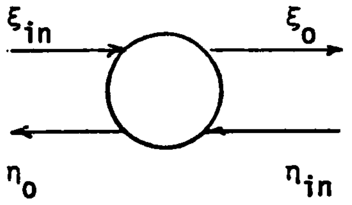


Figure 2.5 - A multiply/add cell

EX 1: The node shown in Figure 2.4 represents a simple latch cell which produces at any time $t>1$ on its output link the same data item that appeared on its input link at time $t-1$. At time $t=1$, we have $\eta(1)=\delta$, which corresponds to the fact that at the beginning of the network operation no specific data item appeared on the output link.

EX 2: The operation of the multiply-add cell mentioned in Section 2.1 and shown in Figure 2.1 may be represented by the following node I/O descriptions:

$$\xi_o = \Omega\ \xi_{in}$$
$$\eta_o = \Omega\ (\eta_{in} + w\ .\ \xi_{in})$$

where $w \in R$ is a given real number and ξ_{in}, η_{in}, ξ_o and η_o are the input and output sequences of the node as shown in Figure 2.5.

At this point, it may be useful to note that if a δ-regular operator is used to model a computational cell, then this cell treats δ as a "don't know" quantity, and consequently, the result of any operation cannot be known if any of the operands is not known. On the other hand, non δ-regular operators are used to model computational cells which treat δ as a special symbol that affects the result of the operation. Hence, each physical communication link in networks containing cells of this type should be augmented by an additional wire to indicate whether the link carries valid data or not. The operation of each cell is then dependent on this additional piece of information.

Since in any practical dynamic system any data item produced by a computational cell at time t depends only on the data provided to that cell at times less than t, we immediately see the importance of the condition imposed in Section 2.2 on the node I/O descriptions, namely that exclusively causal operators in the sense of Section 2.1 are to be used. We also note that with the model described above, the computational power of each cell is not limited to simple arithmetical operations. In other words, a cell could be an intelligent cell that can perform elaborate calculations provided only that we can express these calculations in terms of causal operators.

Clearly, operators on sequences play an important role in the abstract model. In the next section, we introduce additional sequence operators that are defined directly on $\overline{R}_\delta$.

2.4. Additional operators on sequences.

It was shown in the last section that element-wise operators and the shift operator may be used to model simple computational cells. However, these operators are not sufficient to model cells with memory capabilities or with complex control structures. Here, we introduce new sequence operators that may be used to express the computation of some elaborate types of cells. For simplicity, given any operator $\Gamma:[\overline{R}_\delta]^n \to \overline{R}_\delta$, the notation $[\Gamma(\xi_1,\cdots,\xi_n)](t)$, will be employed to designate the t^{th} element $\eta(t)$ of the image sequence $\eta=\Gamma(\xi_1,\cdots,\xi_n)$. This is consistent with the convention of using square brackets for grouping. We will also use the symbol ÷ for integer division and the Fortran function *mod*() that specifies the remainder of an integer division. We start by generalizing the definition of the shift operator given in Section 2.1.

The Shift operator $\Omega^r : \bar{R}_\delta \rightarrow \bar{R}_\delta$ is defined for any r by

$$[\Omega^r \xi](t) = \begin{cases} \delta & \text{if } r>0 \text{ and } t\leq r \\ \xi(t-r) & \text{otherwise.} \end{cases}$$

Hence, for $r>0$, Ω^r inserts r δ-elements at the beginning of a sequence and therefore models the computation of a delay cell. On the other hand, for $r<0$, Ω^r trims the first r elements of the sequence and thus is a non causal operator which cannot be used to model computational cells. The role of the negative shift operator is to provide in the proofs an inverse for the positive shift. More precisely, for any sequence ξ, we have $\Omega^{-r}\,\Omega^r\,\xi = \xi$. The converse is not always true, in the sense that $\Omega^r\,\Omega^{-r}\,\xi = \xi$ only if $\xi(t)=\delta$ for $t\leq r$.

The Zero Shift operator $\Omega_0^r:\bar{R}_\delta \rightarrow \bar{R}_\delta$ has the same definition as Ω^r except that Ω_0^r inserts r zeroes at the beginning of a sequence instead of r δ-elements. The zero shift operator is useful in modeling delay cells in networks that initially set the data on their communication links to zero. In such networks we must assume that the entries corresponding to the time t=1 in any non input sequence are equal to 0 rather than δ.

The Accumulator operator $A^{r,k,s} : \bar{R}_\delta \rightarrow \bar{R}_\delta$ is defined to model a cyclic accumulator that starts operation at time t=r, accumulates a new element every s time units and restarts a new cycle every sk time units. The accumulator operator can be defined in terms of the following algorithm that computes $[A^{r,k,s}\xi](t)$ for any $t>0$, given the elements $\xi(i)$, $i\leq t$.

```
IF $(t<r)$ THEN $[A^{r,k,s}\xi](t) = \delta$ /* accumulator is idle */
ELSE
 BEGIN
  $t_r = t - mod((t-r) \div sk)$   /* time of last reset */
  $na = ((t-t_r) \div s) + 1$   /* number of elements accumulated */
  $[A^{r,k,s}\xi](t) = \sum_{j=0}^{na-1} \xi(t_r+sj)$ /* result of accumulating $na$ elem.*/
 END
```

Evidently, this algorithm is equivalent with

$$[A^{r,k,s}\xi](t) = \begin{cases} \delta & t<r \\ \sum_{j=0}^{na-1} \xi(t_r+sj) & t \geq r \end{cases}$$

where na and t_r are as specified before. As an example, let

$$\xi = a_1,b_1,a_2,b_2,\cdots,a_7,b_7,\delta,\delta,\cdots \tag{2.4}$$

then

$$A^{2,3,2}\xi = \delta,b_1,\bullet,b_1+b_2,\bullet,b_1+b_2+b_3,\bullet,b_4,\bullet,b_4+b_5,\bullet,b_4+b_5+b_6,\bullet,b_7,\bullet,\delta,\cdots$$

where ● denotes an element that is equal to the preceding one.

The Multiplexer operator $M_r^{w1,\ldots,wn}(\xi_1,\cdots,\xi_n) : [\bar{R}_\delta]^n \to \bar{R}_\delta$ is defined to model a multiplexer that has n inputs $\xi_1,\cdots,\xi_n$. It starts its operation at time $t=r$ and periodically multiplexes its inputs with a time ratio of $w1:w2:\cdots:wn$. If the length of the multiplexer cycle is denoted by $k=\sum_{e=1}^{n} w_e$, then the following algorithm defines the multiplexer operator

```
IF $(t<r)$ THEN $[M_r^{w1,\ldots,wn}(\xi_1,\cdots,\xi_n)](t) = \delta$ /* multiplexer idle */
ELSE
 BEGIN
  $t_c = t - mod((t-r) \div k)$   /* start of current cycle */
  Find the largest integer $1 \leq e \leq n$
```

```
    such that  $(t-t_c) < \sum_{i=1}^{e} w_i$   /* determine interval within cycle */
    $[M_r^{w1,\dots,wn}(\xi_1,\cdots,\xi_n)](t) = \xi_e(t)$   /* chose corresponding input */
END
```

As an example, let

$$\zeta = a_1, a_2, \cdots, a_7, a_8, a_9, \delta, \delta, \cdots$$

and

$$\eta = b_1, b_2, \cdots, b_7, \delta, \delta, \delta, \cdots$$

then

$$M_3^{1,2}(\zeta,\eta) = \delta, \delta, a_3, b_4, b_5, a_6, b_7, \delta, a_9, \delta, \cdots$$

It is also interesting to note that the multiplexer operator can be used to model a de-multiplexer cell. For example, if we want to sample the sequence ξ at times t=r,2r,3r,... , then we may express this operation as $M_r^{1,r-1}(\xi,\delta^*)$ where δ^* is the don't care sequence introduced earlier.

The multiplexer operator can be used to define two further operators, namely, the expansion and the piping operators.

The Expansion operator $E_r^k : \overline{R}_\delta \rightarrow \overline{R}_\delta$ models a cyclic memory that is loaded at time t=r and is overwritten every k time units. It is formally defined by

$$E_r^k \ \eta = M_r^{1,\dots,1}(\eta \ , \ \Omega\eta \ , \ \Omega^2\eta \ , \ \cdots \ , \ \Omega^{k-1}\eta).$$

which on the basis of the definition of the multiplexer operator may be rewritten as

$$E_r^k \eta = \begin{cases} \delta & t < r \\ \eta(t-t_u) & t \geq r \end{cases}$$

where $t_u = mod((t-r) + k)$. For example, with ξ of (2.4) we have

$$E_2^4 \xi = \delta, b_1, \bullet, \bullet, \bullet, b_3, \bullet, \bullet, \bullet, b_5, \bullet, \bullet, \bullet, b_7, \bullet, \bullet, \bullet, \delta, \delta, \cdots$$

Note that the accumulator, multiplexer and expansion operators are weakly causal operators. Besides the causal and weakly causal operators used in modeling computational cells, some sequence operators may be introduced for the sole purpose of allowing us to simplify the description of data sequences. Following are two such operators:

The Piping operator $P_m^k : [\overline{R}_\delta]^m \to \overline{R}_\delta$ defined by

$$P_m^k(\eta^1,\cdots,\eta^m) = M_1^{k,\ldots,k}(\eta^1 \ , \ \cdots \ , \ \Omega^{(i-1)k} \ \eta^i \ , \ \cdots \ , \ \Omega^{(m-1)k} \ \eta^m)$$

and $T(P_m^k(\eta^1,\cdots,\eta^m)) = mk$. In other words, P_m^k concatenates the first k elements of each of the m sequences η^e, $e=1,\cdots,m$, and forms one long sequence.

On the basis of the definition of the multiplexer operator it is easily shown that the following algorithm is equivalent with the above definition of the piping operator

```
IF($t>mk$) THEN $[P_m^k(\eta^1,\cdots,\eta^m)](t) = \delta$
ELSE
  BEGIN
    Find the largest integer $1 \leq e \leq m$ such that $t \leq ek$
    $[P_m^k(\eta^1,\cdots,\eta^m)](t) = \eta^e(t-(e-1)k)$
  END
```

In the remainder of this dissertation, we will use the abbreviations $P_{e=1,m}^k(\eta^e)$ for $P_m^k(\eta^1,\cdots,\eta^m)$, and $P_m^k(\eta)$ for $P_m^k(\eta,\cdots,\eta)$. As will be seen later, the piping operator is very useful for the verification of pipelined operation of systolic networks.

The Spread Operator $\Theta^s : \overline{R}_\delta \to \overline{R}_\delta$ defined by

$$[\Theta^s \ \xi](t) = \begin{cases} \xi(\frac{t+s}{1+s}) & t=1,(s+1)+1,2(s+1)+1,\cdots \\ \delta & otherwise \end{cases}$$

Hence Θ^s inserts s δ-elements between every two elements of ξ. With the sequence ξ of (2.4) we have, for example

$$\theta^2 \xi = a_1.\delta.\delta.b_1.\delta.\delta.a_2.\delta.\delta.b_2.\cdots$$

In Appendix A, we give some properties about combinations of the different sequence operators. Those properties provide tools for the manipulation of sequence expressions and hence will be used extensively in the verification of systolic networks. In the next section, we introduce the notion of "Network I/O Description", which is analogous to the transfer function in circuit theory.

2.5. The Network I/O Description.

Our goal in this section is to model the computation of a systolic network by describing the relation between its outputs and its inputs. In order to formalize this input/output relation, we start by introducing some new terminology. We call "network output sequences" those sequences associated with the IN edges of sink nodes, and "network input sequences" those *associated with the OUT edges of source nodes. Then the system of all* node I/O descriptions provides a specification of the computation performed by the network in the form of an implicit relation between the network input and output sequences. This relation will be called the "network I/O description".

As a *simple example*, consider the hypothetical network with the graph shown in Figure 2.6. In this graph, we assume that the edges directed to the left are given the color y and those directed to the right the color x. *We also* follow the *naming convention* of Section 2.2 to identify the different edges in the graph. To complete the network description, a node I/O description has to be specified for each node in the graph. Assume that these are given by the following causal relations:

For node 1: $\xi_2 = \Omega\ [\ \xi_1 + \eta_1\ ]$ (2.5.a)

$\eta_0 = \Omega\ [\ \xi_1 * \eta_1\ ]$ (2.5.b)

For node 2: $\xi_3 = \Omega\ \xi_2$ (2.6)

For node 3: $\eta_1 = \Omega\ [\ \xi_3 * \eta_3\ ]$ (2.7)

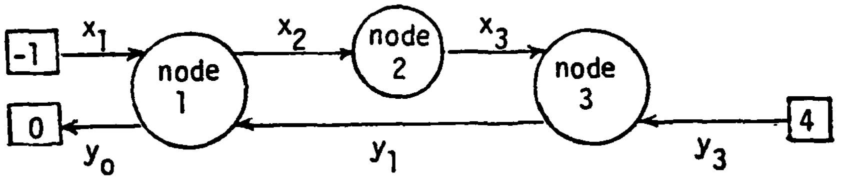


Figure 2.6 - A hypothetical systolic network

For this network, η_3 and ξ_1 are the network input sequences and η_0 is the network output sequence. In order to obtain the network I/O description explicitly, we have to solve the equations (2.5), (2.6) and (2.7), that is, we have to obtain an explicit expression for η_0 in terms of ξ_1 and η_3.

Generally, it is very difficult, and sometimes impossible, to derive an explicit solution of the system of node I/O equations. In the next section, we show that this task may be greatly simplified in the case of certain networks with a homogeneous structure.

2.6. Homogeneous Systolic Networks.

By condition [A2], any edge $e \in E$ is uniquely identified by its color and one of its incident nodes. In fact, we have already used this as a convenient means for identifying edges by their color and terminal node. Let $M \subset C_E \times V_I$ be the set of all pairs (y,v), $y \in C_E$, $v \in V_I$, for which there is an edge $e \in E$ with $y = col(e)$ and $v = \varphi_-(e)$. Then the terminal node $u = \varphi_+(e)$ is uniquely given and hence the successor function $\mu : M \rightarrow V_I \cup V_T$ is well defined by the association

$$(y,v) \in M,\ y = col(e),\ v = \varphi_-(e)\ \rightarrow\ \mu(y,v) = \varphi_+(e).$$

In other words, if there exists an edge e with color y and starting node v, then $\mu(y,v)$ is the terminal node of e.

Given a systolic network based on the graph $G= (V,E,\psi_-,\psi_+)$, a subset $V_I' \subseteq V_I$ of interior nodes is said to be a homogeneous set if:

[H1] All the nodes in V_I' have identical IN and OUT degrees, say m and n, respectively.

[H2] The sets of the m colors for the IN edges of any interior node $v \in V_I'$ are identical. So are the sets of the n colors for the OUT edges of v. Denote the colors of the IN and OUT edges of v by $y^1, y^2, \cdots, y^m$ and $z^1, z^2, \cdots, z^n$, respectively.

[H3] The node I/O descriptions of any interior node $v \in V_I'$ are generic in the sense that they may be written in the form:

$$\zeta^i_{\mu(z^i,v)} = \Gamma^i(\eta^1_v, \cdots, \eta^m_v) \qquad i=1,\cdots,n$$

where $\Gamma^i, i=1,\cdots,n$ are given n-ary operators which are independent of the particular node in V_I', μ is the successor function defined earlier in this section and η^i_v $i=1,\cdots,m$ and $\zeta^i_{\mu(z^i,v)}$ $i=1,\cdots,n$ are the sequences associated with the IN and OUT edges of v, respectively.

A network is said to be homogeneous if the set of interior nodes V_I in its graph G is a homogeneous set. More generally, if there exists a partition $V_I = V_I^1 \cup \cdots \cup V_I^k$ of V_I into k non-empty homogeneous subsets $V_I^1, \cdots, V_I^k$, then the network is said to be k-partially homogeneous. The main advantage of having a homogeneous (or partially homogeneous) network is that the resulting system of equations has a repetitive pattern, which, in many cases, allows us to obtain its solution analytically.

As an example, we consider the 1-D convolution network described in the beginning of this chapter. The graph of this network is shown in Figure 2.7, where we assumed that the edges directed to the left have the color 's', while those directed to the right have the color 'p'. The nodes of the graph are identified by the integers -1,0,1,....,k+2, where nodes -1 and k+2 are source nodes, nodes 0 and k+1 sink nodes, and nodes 1 through k interior nodes. The successor function is defined for any interior node i=1,....,k by

$$\mu(y,i) = \begin{cases} i+1 & \text{if } y=s \\ i-1 & \text{if } y=p \end{cases}$$

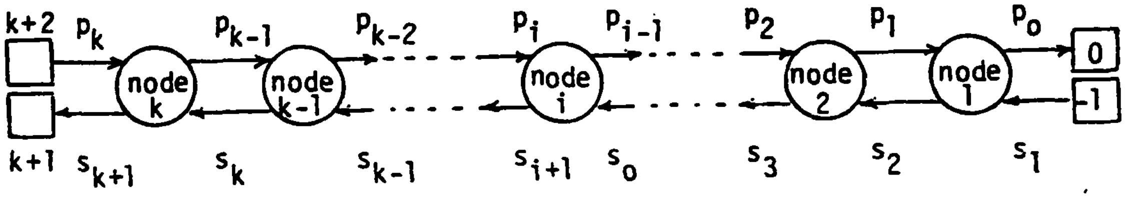


Figure 2.7 - The graph for the 1-D convolution network

The I/O description of a typical interior node i in the graph, $1 \leq i \leq k$, is given by the following causal relations

$$\pi_{i-1} = \Omega\ \pi_i \tag{2.8.a}$$
$$\sigma_{i+1} = \Omega\ [\sigma_i + w_i\ .\ \pi_i\ ] \tag{2.8.b}$$

This system of difference equations is easily solved. First note that the solution of (2.8.a) obviously is

$$\pi_i = \Omega^{k-i}\ \pi_k \tag{2.9}$$

By substituting this in (2.8.b) we obtain

$$\sigma_{i+1} = \Omega\ \sigma_i + w_i\ .\ [\ \Omega^{k-i+1}\ \pi_k\ ] \tag{2.10}$$

The solution of (2.10) is then given by Lemma 1 in Appendix A as:

$$\begin{aligned}\sigma_{k+1} &= \Omega^k\ \sigma_1 + \sum_{i=1}^{k} \Omega^{i-1}\ [w_{k-i+1}\ .\ \Omega^{k-(k-i+1)+1}\ \pi_k] \\ &= \Omega^k\ \sigma_1 + \sum_{i=1}^{k} \Omega^{2i-1}\ [w_{k-i+1}\ .\ \pi_k] \end{aligned} \tag{2.11}$$

This is the I/O description for the network.

In the previous example we derived the network I/O description for a homogeneous network. The technique is equally applicable to k-partially homogeneous networks if k is reasonably small. In that case, a system of difference equations is formed by writing the generic I/O description for a typical node from each homogeneous subset of interior nodes V_I^i, i=1,....k. The network I/O description is then obtained by solving this system of equations. The back substitution network and the sorting networks discussed in the next chapter are examples of 2-partially homogeneous networks.

Finally, we note that the system of causal equations that models the computation of the different cells in a systolic network is an implicit I/O description for that network. The derivation of an explicit formula for the I/O description depends on our ability to solve this system of equation analytically. In both its implicit and explicit forms, the I/O description is a characteristic of the network itself which is independent of any particular computation performed on the network. In the next chapter, we associate a certain computation with a given network, and we suggest a technique for the verification of this computation.

3. FORMAL VERIFICATION OF SYSTOLIC NETWORKS.

A computation on a systolic network is defined by two essential components. Namely, the systolic network and the description of its input. The network itself is characterized by its I/O description which provide the general relation between the inputs and the outputs. However, in the verification of a particular computation, we are usually interested in the behavior of the network for specific inputs. That is we wish to verify that, for specific inputs, the network will produce an output with prescribed properties.

Given the I/O description and the input specifications. The verification of the network's operation is accomplished by substituting the input specifications into the I/O description, and then, manipulating the resulting equations to obtain an explicit description of the output sequences. This explicit description should be in a form that may be compared with the specification of the expected output. In order to clarify our technique, we consider again the example of the 1-D convolution network whose I/O description was found to be given by equations (2.11). Our goal is to verify that the network indeed produces the results in equation (2.1) for the network input sequences described by

$$\sigma_1 = \Omega^{k-1} \Theta \iota \qquad (3.1.a)$$

$$\pi_k = \Theta \xi \qquad (3.1.b)$$

where

$$\iota(t) = 0, \quad 1 \leq t \leq T(\iota) = n-(k-1)$$

$$\xi(t) = x_t, \quad 1 \leq t \leq T(\xi) = n$$

In order to find the corresponding specific form of the output sequence σ_{k+1}, we substitute (3.1) into (2.11) and obtain

$$\sigma_{k+1} = \Omega^{2k-1} \ominus \iota + \sum_{j=1}^{k} \Omega^{2j-1} [w_{k-j+1} \cdot \ominus \xi]$$

By the properties P3.1, P4, P2.1 and P1.1 in Appendix A, this may be rewritten as

$$\begin{aligned}\sigma_{k+1} &= \Omega^{2k-1} \ominus \iota + \Omega \sum_{j=1}^{k} \Omega^{2(j-1)} \ominus [w_{k-j+1} \cdot \xi] \\ &= \Omega^{2k-1} \ominus \iota + \Omega \ominus \sum_{j=1}^{k} \Omega^{j-1} [w_{k-j+1} \cdot \xi] \\ &= \Omega^{2k-1} \ominus \iota + \Omega \ominus \sum_{j=1}^{k} \Omega^{j-1} \eta_j\end{aligned}$$

where $T(\eta_j)=T(\xi)=n$ and $\eta_j(t) = w_{k-j+1}\ \xi(t) = w_{k-j+1}\ x_t$. Finally, applying Property P19 of Appendix A, we find tha.:

$$\begin{aligned}\sigma_{k+1} &= \Omega^{2k-1} \ominus \iota + \Omega \ominus \Omega^{k-1}\ \eta \\ &= \Omega^{2k-1} \ominus\ [\iota + \eta] \\ &= \Omega^{2k-1} \ominus \eta, \end{aligned} \tag{3.2}$$

where η is defined by:

$$\begin{aligned} T(\eta) &= n-(k-1) \\ \eta(t) &= \sum_{j=1}^{k} \eta_j(t+k-j) && 1 \leqslant t \leqslant T(\eta) \\ &= \sum_{j=1}^{k} w_{k-j+1}\ x_{t+k-j} && 1 \leqslant t \leqslant T(\eta) \\ &= \sum_{q=1}^{k} w_q\ x_{t+q-1} && 1 \leqslant t \leqslant T(\eta) \end{aligned}$$

In the last line, the summation index was changed to q=k-j+1 in order to provide for the same expression as in (2.1).

Evidently, equation (3.2) represents the output of the array in a clear and precise form; it indicates that after an initial period of 2k-1 time units, the elements $\eta(t)=y_t$, $1 \leqslant t \leqslant n-(k-1)$, will appear on the output link, each separated from the other by one time unit.

In the last example, and in the example of the matrix/vector multiplication network presented in the next section, the verification procedure is based on the availability of an explicit I/O description which represents a general solution of the system of equations that models the computational cells in the network. However, in some cases, it is difficult to solve this system of equations in general, and it is much easier to solve it for specific input sequences. In other words, sometimes it may be difficult to determine the explicit I/O description while it is easier to obtain the description of the output sequences for specific inputs. The example of the back substitution network given in Section 3.2 illustrates this further. In this example, we verify the operation of the network without obtaining an explicit formula for the network I/O description.

The existence of an analytical solution to systems of causal equations is discussed in some details in Section 3.4, where we show that it is always possible to obtain a formal solution of such system of equations. However, the form of the solution may sometimes be very complicated and hence not practically useful. In such cases, we may still verify the operation of the corresponding network if we have an idea about the network behavior and hence about the sequences associated with the different edges of the graph. In fact, we need to show that for the given input sequences, the expected sequences satisfy the system of causal equations. We demonstrate this procedure in Section 3.3 by verifying the operation of a sorting network for which we could not solve the system of equations explicitly.

3.1. A matrix-vector multiplication network.

In [31], Kung and Leiserson suggested a systolic network for the computation of the product $y=Ax$ of a matrix A and a vector x. Here, we modify this architecture for the case where A is a square, symmetric matrix. The goal of this modification is to reduce the number of input items to the network by only supplying the elements in the upper half of A. This has the effect of reducing the I/O bandwidth of the network which is usually a limiting factor in VLSI architectures.

The modified network will be formally specified using the abstract model of Chapter 2. Moreover, the sequence notation will provide a clear and accurate representation of the input required for the proper operation of the network including information about the relative timing of the inputs on the different links. We will also apply the technique discussed earlier of obtaining the network I/O description and then of verifying that, with the appropriate input, the network does produce the elements of the product vector y.

In Figure 3.1, we show the graph of the multiplication network for matrices of order k. It consists of 2(k+1) internal nodes, each labeled by a pair (i,g), $0 \leqslant i \leqslant 1$, $1 \leqslant g \leqslant k+2$. The set of colors C_E has three elements, namely, s, r, and z, and the coloring function $col()$ maps the edges to the colors as shown in the figure.

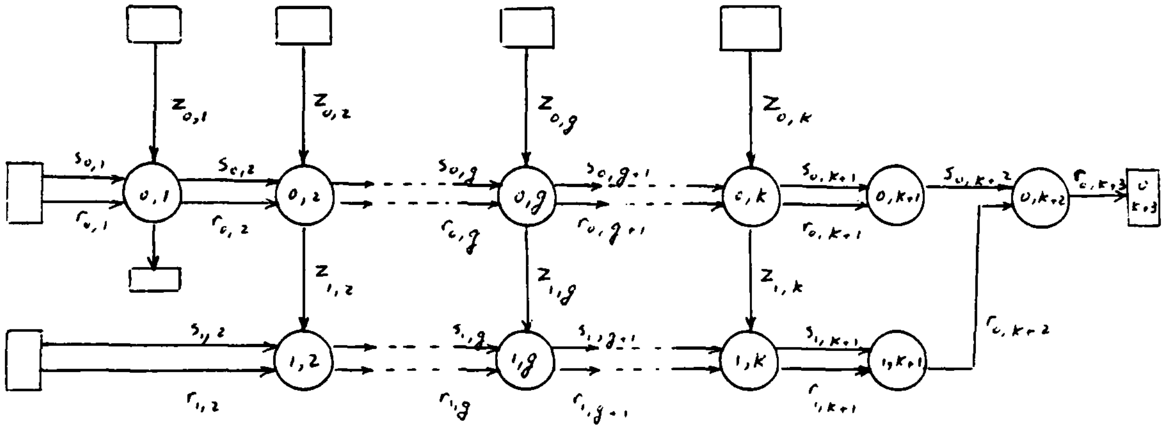


Figure 3.1 - The graph for the matrix/vector multiplication network

The principle of operation of the network may be explained by decomposing the product vector $y=Ax$ into two vectors y^u and y^l such that $y=y^u+y^l$. More specifically, if $a_{i,j}$ and x_i denote the elements of A and x, respectively, we write the elements of y^u and y^l as follows:

$$y_i^l = \begin{cases} 0 & i=1 \\ \sum_{j=1}^{i-1} a_{i,j} \; x_j = \sum_{j=1}^{i-1} a_{j,i} \; x_j & i=2,\cdots,k \end{cases} \qquad (3.3.a)$$

$$y_i^u = \sum_{j=i}^{k} a_{i,j} \; x_j \qquad i=1,\cdots,k \qquad (3.3.b)$$

The elements of y^u are computed on the sub-network composed of the cells corresponding to nodes $(0,1)$, $\cdots$, $(0,k)$ in Figure 3.1. Similarly, the elements of y^l are computed on the subnetwork composed of the cells corresponding to nodes $(1,2)$, $\cdots$, $(1,k)$. The cells $(0,k+1)$ and $(1,k+1)$ are delay cells that align the corresponding elements of y^u and y^l such that they can be appropriately added in the cell $(0,k+2)$. The operation of the network is formally specified by providing, for each cell, a set of equations relating its output sequences to its input sequences. For the nodes in the homogeneous set $\{(0,g) \; ; \; g=1,\cdots,k\}$ the generic causal equations are

$$\sigma_{0,g+1} = \Omega^2 \; [\sigma_{0,g} + \rho_{0,g} \odot \zeta_{0,g}] \qquad g=1,\cdots,k \qquad (3.4.a)$$
$$\rho_{0,g+1} = \Omega \; \rho_{0,g} \qquad g=1,\cdots,k \qquad (3.4.b)$$
$$\zeta_{1,g} = \Omega \; \zeta_{0,g} \qquad g=1,\cdots,k \qquad (3.4.c)$$

where the operator $\odot$ is an element-wise operator obtained by extending the usual multiplication operator $*$ from R to R_δ by adding the following rule:

$$\delta \odot x = x \odot \delta = \delta \quad \textit{if } x \neq 0 \qquad \text{and} \qquad \delta \odot 0 = 0 \odot \delta = 0$$

Note that $\odot$ is not a δ-regular operator in the sense defined in Section 2.1. However, because the result of the multiplication of any unknown number with zero is zero, the operator $\odot$ may be naturally implemented by using a standard multiplication circuit and treating δ as a don't know item rather than as a special symbol. With this, each node in the above homogeneous sub-network represents a multiply/add cell augmented by an appropriate delay.

Similarly, the cells corresponding to the nodes in the homogeneous set $\{(1,g)\ ;\ g=2,\cdots,k\}$ are specified by

$$\sigma_{1,g+1} = \Omega_0^2\ \sigma_{1,g} \qquad g=2,\cdots,k \tag{3.5.a}$$
$$\rho_{1,g+1} = \Omega\ [\rho_{1,g} + \sigma_{1,g} \circ \zeta_{1,g}] \qquad g=2,\cdots,k \tag{3.5.b}$$

Finally, the cells (0,k+1), (1,k+1) and (0,k+2) are specified by

$$\sigma_{0,k+2} = \Omega\ \sigma_{0,k+1} \tag{3.6.a}$$
$$\rho_{0,k+2} = \Omega^k\ \rho_{1,k+1} \tag{3.6.b}$$
$$\rho_{0,k+3} = \Omega\ [\rho_{0,k+2} + \sigma_{0,k+2}] \tag{3.6.c}$$

The system composed of the equations (3.4), (3.5) and (3.6) describes the operation of the entire network. In order to obtain an explicit form of the network I/O description, we should solve this system and obtain a direct relation between the network output sequence of interest, namely $\rho_{0,k+3}$, and the network input sequences $\rho_{0,1}$, $\sigma_{0,1}$, $\rho_{1,2}$, $\sigma_{1,2}$ and $\zeta_{0,g}$, $g=1,\cdots,k$. For this, we start by solving (3.4.b) and (3.5.a) to obtain

$$\rho_{0,g} = \Omega^{g-1}\ \rho_{0,1} \qquad g=1,\cdots,k$$
$$\sigma_{1,g} = \Omega_o^{2(g-2)}\ \sigma_{1,2} \qquad g=2,\cdots,k$$

We then substitute these formulas and equation (3.4.c) into (3.4.a) and (3.5.b), which gives the following difference equations

$$\sigma_{0,g+1} = \Omega^2\ [\sigma_{0,g} + \Omega^{g-1}\ \rho_{0,1} \circ \zeta_{0,g}] \qquad g=1,\cdots,k \tag{3.7.a}$$
$$\rho_{1,g+1} = \Omega\ [\rho_{1,g} + \Omega_0^{2(g-2)}\ \sigma_{1,2} \circ \Omega\ \zeta_{0,g}] \qquad g=2,\cdots,k \tag{3.7.b}$$

The solutions of (3.7.a) and (3.7.b) may be obtained by applying Lemma 1 in Appendix A. More specifically, with c=2, s=1 and r=k+1 in Lemma 1 we get

$$\sigma_{0,k+1} = \Omega^{2k}\ \sigma_{0,1} + \sum_{j=1}^{k} \Omega^{2j}\ [\Omega^{k-j}\ \rho_{0,1} \circ \zeta_{0,k-j+1}] \tag{3.8.a}$$

while for the values c=1, s=2 and r=k+1 in the same lemma, we obtain

$$\rho_{1,k+1} = \Omega^{k-1}\ \rho_{1,2} + \sum_{j=2}^{k} \Omega^{j-1}\ [\Omega_0^{2(k-j)}\ \sigma_{1,2} \circ \Omega\ \zeta_{0,k-j+2}] \qquad (3.8.b)$$

Now, from (3.6) follows the network I/O description in the form

$$\rho_{0,k+3} = \Omega^{k+1}\ \rho_{1,k+1} + \Omega^{2}\ \sigma_{0,k+1} \qquad (3.9)$$

where $\rho_{1,k+1}$ and $\sigma_{0,k+1}$ are given by (3.8.a/b).

For the proper operation of the network, the input links $s_{0,1}$ and $r_{1,2}$ are permanently set to zero, that is we always have

$$\sigma_{0,1} = \rho_{1,2} = \iota$$

where ι is the zero sequence defined in Section 2.1. With this, the equations (3.8) simplify to

$$\sigma_{0,k+1} = \sum_{j=1}^{k} \Omega^{2j}\ [\Omega^{k-j}\ \rho_{0,1} \circ \zeta_{0,k-j+1}] \qquad (3.10.a)$$

$$\rho_{1,k+1} = \sum_{j=2}^{k} \Omega^{j-1}\ [\Omega_0^{2(k-j)}\ \sigma_{1,2} \circ \Omega\ \zeta_{0,k-j+2}] \qquad (3.10.b)$$

In order to perform the matrix/vector multiplication, the elements of the $(g-1)^{st}$ off-diagonal, $1 \leq g \leq k$, of the matrix A should be supplied on the network link $z_{0,g}$, followed by an appropriate number of zeroes. More specifically, the input on these links should be

$$\zeta_{0,g} = \Omega^{2(g-1)}\ \alpha_g \qquad g=1,\cdots,k \qquad (3.11.a)$$

where $T(\alpha_g)=k$ and

$$\alpha_g(t) = \begin{cases} a_{t,t+g-1} & t \leq k-g+1 \\ 0 & t > k-g+1 \end{cases}$$

Moreover, the elements of the vector x should be supplied on the links $r_{0,1}$ and $s_{1,2}$ according to

$$\rho_{0,1} = \xi \qquad (3.11.b)$$

$$\sigma_{1,2} = \Omega_0^3 \ \xi \tag{3.11.c}$$

where $T(\xi)=k$ and $\xi(t)=x_t$. Note that $s_{1,2}$ carries the same information as $r_{0,1}$ delayed by three time units. Hence, by adding appropriate delay cells, we may replace these two input links by only one link.

As the next step in the verification technique we have to substitute the above input sequences into the network I/O description and to demonstrate that the sequence $\rho_{0,k+3}$ does contain the elements y_i, $i=1,\cdots,k$, of the product vector y. We first substitute (3.11.a/b) into (3.10.a) which gives

$$\begin{aligned}\sigma_{0,k+1} &= \sum_{j=1}^{k} \Omega^{2j} \ [\Omega^{k-j} \ \xi \ \text{O} \ \Omega^{2(k-j)} \ \alpha_{k-j+1}] \\ &= \Omega^{k} \sum_{j=1}^{k} \Omega^{j} \ [\xi \ \text{O} \ \Omega^{k-j} \ \alpha_{k-j+1}] \\ &= \Omega^{2k} \sum_{j=1}^{k} \beta_j\end{aligned}$$

where $\beta_j = \Omega^{-(k-j)} \xi \text{O} \alpha_{k-j+1}$, that is $T(\beta_j)=k$ and

$$\beta_j(t) = \begin{cases} \xi(t+k-j) \ \alpha_{k-j+1}(t) = a_{t,t+k-j} \ x_{t+k-j} & t \le j \\ 0 & t > j \end{cases}$$

The element-wise addition of the sequences β_j, $j=1,\cdots,k$ in (3.12) then gives

$$\sigma_{0,k+1} = \Omega^{2k} \ \eta^u \tag{3.13}$$

where $T(\eta^u)=k$ and $\eta^u(t) = \sum_{j=1}^{k} \beta_j(t)$. However, for $j<t$, we have $\beta_j(t)=0$, from which we get

$$\begin{aligned}\eta^u(t) &= \sum_{j=t}^{k} a_{t,t+k-j} \ x_{t+k-j} \\ &= \sum_{q=t}^{k} a_{t,q} \ x_q = y_t^u\end{aligned}$$

where in the last line we replaced the summation index j by $q=t+k-j$.

Analogously, we substitute (3.11.a/c) into (3.10.b) and find, after some sequence manipulation, that

$$\rho_{1,k+1} = \Omega^{k+1} \eta^{l} \qquad (3.14)$$

where $T(\eta^{l})=k$ and $\eta^{l}(t)=y^{l}_{t}$.

Finally, from (3.13) and (3.14) in (3.9) we obtain the output sequence

$$\rho_{0,k+3} = \Omega^{2k+2} \eta \qquad (3.15)$$

where $T(\eta)=k$ and $\eta(t)=y_t$, the t^{th} element of the product vector $y=Ax$. This verifies that the network will indeed produce the elements of the vector y according to the timing specified by equation (3.15).

Remark : In some applications, as in the one described in section 9.2, the elements of the matrix A cannot be supplied at the high rate specified by equation (3.11.a). More specifically, we assume that the input sequences on the links $z_{0,g}$, $g=1,\cdots,k$ have the forms

$$\zeta_{0,g} = \Omega^{2c(g-1)} \Theta^{c-1} \alpha_g \qquad g=1,\cdots,k$$

for some integer $c>1$. In this case, the network may still operate correctly if we change the period of the synchronizing clock such that the new period is equal to c times the old one. An alternative solution is to change the delays within the computational cells in order to ensure that the elements of the vector x indeed meet the corresponding elements of the matrix A at the appropriate times. This is accomplished by replacing the specifications (3.4), (3.5) and (3.6) by

$$\begin{aligned}
\sigma_{0,g+1} &= \Omega^{2c} [\sigma_{0,g} + \rho_{0,g} \circ \zeta_{0,g}] && g=1,\cdots,k \\
\rho_{0,g+1} &= \Omega^{c} \rho_{0,g} && g=1,\cdots,k \\
\zeta_{1,g} &= \Omega\, \zeta_{0,g} && g=1,\cdots,k \\
\sigma_{1,g+1} &= \Omega_0^{2c} \sigma_{1,g} && g=2,\cdots,k \\
\rho_{1,g+1} &= \Omega^{c} [\rho_{1,g} + \sigma_{1,g} \circ \zeta_{1,g}] && g=2,\cdots,k \\
\sigma_{0,k+2} &= \Omega\, \sigma_{0,k+1} \\
\rho_{0,k+2} &= \Omega^{ck} \rho_{1,k+1}
\end{aligned}$$

$$\rho_{0,k+3} = \Omega\,[\sigma_{0,k+2} + \rho_{0,k+2}]$$

In addition, the timings of the data on the other input links $r_{0,1}$ and $s_{1,2}$ have to be modified accordingly to

$$\rho_{0,1} = \theta^{c-1}\,\xi$$
$$\sigma_{1,2} = \Omega_0^{2c+1}\,\theta^{c-1}\,\xi$$

Following the same steps as in the case c=1, we may obtain the output of the modified network as

$$\rho_{0,k+3} = \Omega^{2ck+2}\,\theta^{c-1}\,\eta$$

which is a generalization of (3.15) for the case $c \geqslant 1$.

3.2. A back substitution network.

In this section, we apply our verification technique to a systolic network that contains two different types of computational cells, namely the back-substitution network suggested in [31]. This network performs the back substitution operation to solve the linear system of equations

$$L\ u = y \tag{3.16}$$

where L is an $n \times n$ non-singular, banded, lower triangular matrix with the (half)-band-width k+1, and y is a given n-dimensional vector. The solution of the linear system (3.16) is given by the formula:

$$u_i = \begin{cases} y_i \;/\; l_{i,i} & i=1 \\ (y_i - \sum_{j=1}^{i-1} l_{i,i-j}\; u_{i-j}) \;/\; l_{i,i} & 2 \leqslant i \leqslant k \\ (y_i - \sum_{j=1}^{k} l_{i,i-j}\; u_{i-j}) \;/\; l_{i,i} & k < i \leqslant n \end{cases}$$

where $l_{i,j}$ is the $(i,j)^{th}$ element of the matrix L, and y_i and u_i are the i^{th} elements of the vectors y and u, respectively.

Figure 3.2 shows the graph of the suggested network. It is a 2-partially homogeneous network, composed of k multiply/add (M/A) type cells,

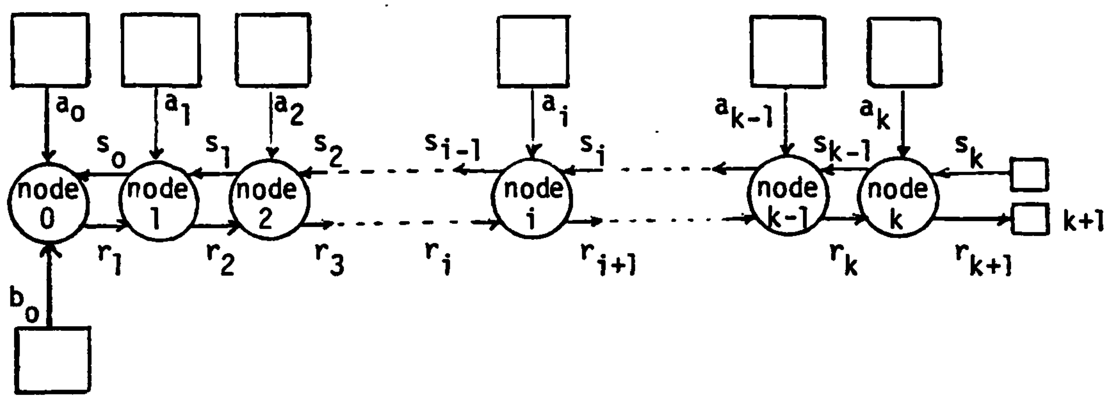


Figure 3.2 - The graph for the back substitution network

and one subtract/divide (S/D) cell. The computational cells are labeled by integers such that the cells 1 through k are of the M/A type, and the cell 0 is the S/D cell. As for the I/O cells, we must be careful to assign labels to the sink cells because these labels will be used to identify the network output links. The labels given to source nodes are immaterial as they do not affect the verification procedure, and consequently are not shown in Figure 3.2.

In the regular layout shown in Figure 3.2, the edges directed to the south, north, east and west are given the colors a,b,r and s, respectively. The set V_I of interior nodes in G is divided into two homogeneous subsets $V_I^1=\{0\}$ and $V_I^2=\{i:i=1,\cdots,k\}$. The operation of the cell represented by node '0' is described by the causal relation

$$\rho_1 = \Omega\ [[\beta_0 - \sigma_0] \ / \ \alpha_0] \tag{3.17}$$

and the operation of any M/A cell represented by a node i, $1 \leqslant i \leqslant k$, is described by the generic I/O description

$$\rho_{i+1} = \Omega\ \rho_i \qquad i=1,\cdots,k \tag{3.18.a}$$
$$\sigma_{i-1} = \Omega\ [\sigma_i \oplus \alpha_i * \rho_i] \qquad i=1,\cdots,k \tag{3.18.b}$$

where the $\oplus$ was defined by the formula (2.2).

In order to solve the system of difference equations (3.17), (3.18.a/b), we first write the solution of (3.18.a) as

$$\rho_i = \Omega^{i-1} \rho_1 \qquad 1 < i \leqslant k+1 \tag{3.19}$$

from which we find that

$$\rho_{k+1} = \Omega^k \rho_1 \tag{3.20}$$

Substitution of (3.19) into (3.18.b) then gives

$$\sigma_{i-1} = \Omega [\sigma_i \oplus \Delta_i] \tag{3.21}$$

where $\Delta_i = \alpha_i * [\Omega^{i-1} \rho_1]$. Using an inductive argument similar to that in Appendix A for the proof of Lemma 1, we can show that the solution of (3.21) is

$$\sigma_0 = \Omega^k \sigma_k \oplus \sum_{i=1}^{k}{}' \Omega^i [\alpha_i * \Omega^{i-1} \rho_1] \tag{3.22}$$

where $\sum'$ is defined by $\sum_{i=1}^{k}{}' \eta_i = \eta_1 + \eta_2 + ... + \eta_k$.

For given ρ_1, the network output sequence ρ_{k+1} is easily obtained from (3.20). The next step will be to eliminate σ_0 from (3.17) and (3.22) and to obtain ρ_1 explicitly in terms of the network input sequences σ_k, β_0 and α_i, $i=0,\cdots,k$. Unfortunately, if we try to solve (3.17) and (3.22) simultaneously, we will obtain a recursive equation in ρ_1, which is very difficult to manipulate in general. For this reason, we consider only specific forms of the network input sequences, namely those required for the proper operation of the network. They are given by

$$\alpha_i = \Omega^{k+i} \ominus \lambda_i \qquad i=0,\cdots,k \tag{3.23.a}$$
$$\beta_0 = \Omega^k \ominus \eta \tag{3.23.b}$$
$$\sigma_k = \ominus \iota \tag{3.23.c}$$

with $T(\lambda_i) = n-i$, $T(\iota) = T(\eta) = n$ and

$$\lambda_i(t) = l_{t+i,t} \qquad 1 \leq t \leq n-i$$
$$\eta(t) = y_t \qquad 1 \leq t \leq n$$
$$\iota(t) = 0 \qquad 1 \leq t \leq n$$

Substituting (3.23) into (3.17) and (3.22), we find that

$$\rho_1 = \Omega\ [[\Omega^k \ominus \eta - \sigma_0] \ /\ \Omega^k \ominus \lambda_0] \tag{3.24.a}$$
$$\sigma_0 = \Omega^k \ominus \iota \oplus \sum_{j=1}^{k}{}' \ [\Omega^{2j+k} \ominus \lambda_j * \Omega^{2j-1}\ \rho_1] \tag{3.24.b}$$

Since $\delta - x = \delta$ for any $x \in R_\delta$, (3.24.a) implies the existence of a sequence ξ such that

$$\rho_1 = \Omega^{k+1} \ominus \xi \tag{3.25}$$

whence, by (3.24.b), we find that

$$\sigma_0 = \Omega^k \ominus [\iota \oplus \sum_{j=1}^{k}{}' \ \Omega^j\ [\lambda_j * \xi]\ ]$$

where we used property P4 to interchange Ω^{2j} and $\ominus$. If in addition we let

$$\gamma = \iota \oplus \sum_{j=1}^{k}{}' \ \Omega^j\ [\lambda_j * \xi] \tag{3.26}$$

then we can substitute for σ_0 and ρ_1 in (3.24.a) and obtain

$$\Omega^{k+1} \ominus \xi = \Omega\ [[\Omega^k \ominus \eta - \Omega^k \ominus \gamma] \ /\ \Omega^k \ominus \lambda_0]$$

which reduces to

$$\xi = [\eta - \gamma] \ /\ \lambda_0 \tag{3.27}$$

For an explicit description of the sequence γ, we need to examine (3.26) more closely. We start by evaluating the product term, namely

$$\Omega^j\ [\lambda_j * \xi] = \Omega^j\ \mu_j$$

where

$$T(\mu_j) = \min(\ T(\lambda_j)\ ,\ T(\xi)) \leq n-j \tag{3.28.a}$$

and

$$\mu_j(t) = \lambda_j(t) * \xi(t) \tag{3.28.b}$$

This enables us to rewrite (3.26) as

$$\gamma = \iota \oplus \sum_{j=1}^{k}{}' \, \Omega^j \mu_j \tag{3.29}$$

From (3.29) and the definition of the '$\oplus$' operator, we conclude that $T(\gamma) = \max(T(\iota) \, , \, T(\mu_j)+j) = n$, and hence from (3.27) that

$$T(\xi) = \min(T(\eta) \, , \, T(\gamma) \, , \, T(\lambda_0)) = n.$$

With this in (3.28.a), it is easily seen that $T(\mu_j) = n-j$. Now, we apply property P20 to (3.29) and explicitly describe γ by

$$T(\gamma) = T(\iota) = n$$

and

$$\gamma(t) = \begin{cases} 0 & t=1 \\ \sum_{j=1}^{t-1} \mu_j(t-j) & t=2,3,\dots,k \\ \sum_{j=1}^{k} \mu_j(t-j) & t=k+1,k+2,\dots,n \end{cases}$$

Then finally, with these specific descriptions of η, λ_0 and γ, the explicit form of the sequence ξ in (3.27) is found to be

$$\xi(t) = (\eta(t) - \gamma(t)) \, / \, \lambda_0(t),$$

that is

$$\xi(t) = \begin{cases} y_t \, / \, l_{t,t} & t=1 \\ (y_t - \sum_{j=1}^{t-1} \xi(t-j) \, l_{t,t-j}) \, / \, l_{t,t} & 2 \leqslant t \leqslant k \\ (y_t - \sum_{j=1}^{k} \xi(t-j) \, l_{t,t-j}) \, / \, l_{t,t} & k+1 \leqslant t \leqslant n \end{cases}$$

A comparison of this expression with the formula given in the beginning of the section for the solution of (3.16) shows readily that

$$\rho_{k+1} = \Omega^{2k+1} \ominus \xi$$

where $T(\xi) = n$ and $\xi(t) = u_t$.

3.3. A sorting network.

The sorting network [32] described here accepts an indexed set $X=\{x_1,\cdots,x_k\}$ of k different real numbers, $x_i \in R$, $i \in K=\{1,\ldots,k\}$, and produces as output the same numbers sorted in ascending order. Figure 3.3 shows the general graph of the network and the labels given to each node. In the figure, the edges directed to the right and left are colored p and s, respectively.

For any $j \in K$, let $y_1,\cdots,y_j$ be the result of sorting the j elements $x_1,\cdots,x_j$ of X in ascending order. Then for all (i,j) of $D=\{(i,j) \in K \times K;\ 1 \leq i \leq j \leq k\}$, the ranking function $f_x: D \rightarrow X$ is defined by $f_x(i,j)=y_i$.

With this, we will prove that if the network input sequence π_k is given by

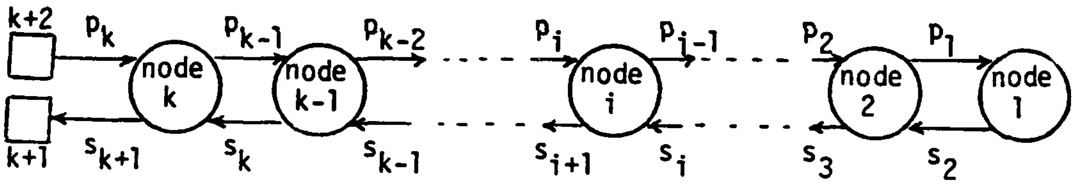


Figure 3.3 - The graph for the sorting network

$$\pi_k = \Theta\ \xi \tag{3.30}$$

where $T(\xi) = k$ and $\xi(t) = x_t$, then the network output sequence σ_{k+1} has the form

$$\sigma_{k+1} = \Omega^{2k-1}\ \Theta\ \eta \tag{3.31}$$

where $T(\eta) = k$ and $\eta(t) = f_x(t,k)$.

The network considered in Figure 3.3 is a 2-partially homogeneous network. The cell labeled '1' is a simple latch cell whose operation is described by

$$\sigma_2 = \Omega\ \pi_1 \tag{3.32.a}$$

while the I/O description of the cells i=2,....,k is given by

$$\pi_{i-1} = \Omega\ \max_\delta(\pi_i,\sigma_i) \tag{3.32.b}$$
$$\sigma_{i+1} = \Omega\ \min_\delta(\pi_i,\sigma_i) \tag{3.32.c}$$

where $\max_\delta$ and $\min_\delta$ were defined in section 2.1. In other words, the cells i=2,....,k are comparision cells which operate as follows: At any time t, if neither one of the two inputs $\sigma_i(t)$ or $\pi_i(t)$ is a don't care element δ, then the cell compares the two inputs, and produces as output at time t+1, the largest and the smallest of the two *numbers* on the links p_{i-1} and s_{i+1} respectively. However, if any of the inputs is δ, then the cell acts as a simple latch cell, that is, if $\sigma_i(t)=\delta$ or $\pi_i(t)=\delta$ then

$$\pi_{i-1}(t+1) = \pi_i(t) \quad \text{and} \quad \sigma_{i+1}(t+1) = \sigma_i(t)$$

To obtain the network I/O description, the system of equations (3.32.a/b/c) should be solved for σ_{k+1}. However, the *recursive* nature of (3.32.b) and (3.32.c) makes this very difficult, if not impossible. One possible alternative is to suggest a tentative value for the sequences π_i and σ_i, and then to verify that these suggested solutions indeed satisfy (3.32). Of course, any assumed value for π_i should reduce to the input sequence (3.30) for i=k.

Let us assume that π_i and σ_i are given by

$$\pi_i = \Omega^{k-i}\ \Theta\ \alpha_i \qquad 1 \le i \le k \tag{3.33.a}$$
$$\sigma_i = \Omega^{k+i-2}\ \Theta\ \beta_i \qquad 2 \le i \le k+1 \tag{3.33.b}$$

where $T(\alpha_i) = T(\beta_i) = k$.

$$\alpha_i(t) = \begin{cases} x_t & 1 \leq t \leq i \\ \max(x_t, f_x(t-i, t-1)) & i < t \leq k \end{cases}$$

and

$$\beta_i(t) = \begin{cases} f_x(t, t+i-2) & 1 \leq t \leq k+1-i \\ f_x(t, k) & k+1-i < t \leq k \end{cases}$$

It is very easy to verify that (3.33.a) reduces to (3.30) for i=k. Hence, our next step will be to check that (3.33) does satisfy (3.32). For i=1, (3.33.a) reduces to

$$\pi_1 = \Omega^{k-1} \ominus \alpha_1$$

where $T(\alpha_1)=k$, and

$$\alpha_1(t) = \begin{cases} x_t & t=1 \\ \max_\delta(x_t, f_x(t-1, t-1)) & 1 < t \leq k \end{cases}$$

Since $f_x(j,j)$ is the maximum element in $(x_1, x_2, \cdots, x_j)$, it follows that $x_1 = f_x(1,1)$ and $\max_\delta(x_t, f_x(t-1,t-1)) = f_x(t,t)$. Hence, we may write

$$\alpha_1(t) = f_x(t,t) \qquad 1 \leq t \leq k$$

But from (3.33.b), we obtain for i=2

$$\sigma_2 = \Omega^k \ominus \beta_2$$

where $T(\beta_2) = k$ and $\beta_2(t) = f_x(t,t)$, $1 \leq t \leq k$, which proves that $\beta_2 = \alpha_1$, and hence $\sigma_2 = \Omega\, \pi_1$.

The next step is to show that (3.33) does satisfy (3.32.b). For this, we subtitute (3.33) into the right hand side of (3.32.b) and denote the resulting sequence by ρ. This gives

$$\rho = \Omega \max_\delta(\Omega^{k-i} \ominus \alpha_i \;,\; \Omega^{k+i-2} \ominus \beta_i) \qquad 2 \leq i \leq k$$

Using property P4 to interchange $\Omega^{2(i-1)}$ and $\ominus$ in the second operand of $\max_\delta$ we obtain

$$\rho = \Omega^{k-(i-1)} \ominus \gamma_i \tag{3.34}$$

where $\gamma_i = \max_\delta(\alpha_i \, , \, \Omega^{i-1} \, \beta_i)$. By definition of $\max_\delta$, it follows that $T(\gamma_i) = T(\alpha_i) = k$, and

$$\gamma_i(t) = \begin{cases} \alpha_i(t) & 1 \leq t \leq i-1 \\ \max(\alpha_i(t), \beta_i(t-i+1)) & i-1 < t \leq k \end{cases}$$

Hence with the definitions of $\alpha_i(t)$ and $\beta_i(t)$ we obtain

$$\gamma_i(t) = \begin{cases} x_t & 1 \leq t \leq i-1 \\ \max(x_t \, , \, f_x(t-i+1, t-1)) & t = i \\ \max(\max(x_t \, , \, f_x(t-i, t-1)) \, , \, f_x(t-i+1, t-1)) & i < t \leq k \end{cases}$$

Because of $\max(\max(a,b) \, , \, c) = \max(a,b,c)$, and $f_x(t-i, t-1) < f_x(t-i+1, t-1)$, we may rewrite γ_i as

$$\gamma_i(t) = \begin{cases} x_t & 1 \leq t \leq i-1 \\ \max(x_t \, , \, f_x(t-(i-1), t-1)) & i-1 < t \leq k \end{cases}$$

from which we find that $\gamma_i(t) = \alpha_{i-1}(t)$, and hence, because of (3.34) and (3.33.a), that $\rho = \pi_{i-1}$. This proves that (3.32.b) is satisfied for the values of σ_i and π_i given by (3.33).

Finally, to check that (3.33) does satisfy (3.32.c), we substitute (3.33) into (3.32.c) and denote the resulting sequence by τ. This gives

$$\begin{aligned} \tau &= \Omega \min_\delta(\Omega^{k-i} \ominus \alpha_i \, , \, \Omega^{k+i-2} \ominus \beta_i) \qquad 2 \leq i \leq k \\ &= \Omega^{k-i+1} \ominus \min_\delta(\alpha_i \, , \, \Omega^{i-1} \, \beta_i) \end{aligned}$$

In view of

$$\min_\delta(\alpha_i \, , \, \Omega^{i-1} \, \beta_i) = \Omega^{i-1} \, \varphi_i$$

where $T(\varphi_i) = T(\beta_i) = k$ and

$$\varphi_i(t) = \begin{cases} \min(\alpha_i(t+i-1), \beta_i(t)) & 1 \leq t \leq k-(i-1) \\ \beta_i(t) & k-(i-1) < t \leq k \end{cases}$$

we write

$$\tau = \Omega^{k+(i+1)-2} \ominus \varphi_i \tag{3.35}$$

From (3.35) and (3.32.c), it follows that $\tau = \sigma_{i+1}$ only if $\varphi_i = \beta_{i+1}$. To prove this, we substitute the definitions of $\alpha_i(t+i-1)$ and $\beta_i(t)$ into $\varphi_i(t)$ and obtain

$$\varphi_i(t) = \begin{cases} \min(\max(x_{t+i-1} , f_x(t-1,t+i-2)) , f_x(t,t+i-2)) & 1 \leq t \leq k-(i-1) \\ f_x(t,k) & k-(i-1) < t \leq k \end{cases}$$

But from Lemma 5 in Appendix A, and the fact that $f_x(t,t+i-1) = f_x(t,k)$ for t=k-i+1, we may write $\varphi_i(t)$ as

$$\varphi_i(t) = \begin{cases} f_x(t,t+i-1) & 1 \leq t \leq k-i \\ f_x(t,k) & k-i < t \leq k \end{cases}$$

It follows that $\varphi_i(t) = \beta_{i+1}(t)$ and therefore that $\tau = \sigma_{i+1}$. This completes the proof that the sequences π_i and σ_i of (3.33) indeed satisfy the system of equations (3.32).

Now that (3.33.b) is known to be a valid formula for the sequence σ_i, we can easily obtain the network output sequence σ_{k+1} by setting i=k+1. This gives

$$\sigma_{k+1} = \Omega^{2k-1} \ominus \beta_{k+1}$$

where $T(\beta_{k+1}) = k$ and $\beta_{k+1}(t) = f_x(t,k)$, $1 \leq t \leq k$ which is identical with the expected output sequence (3.31).

After illustrating our verification technique by various examples, we investigate in the next section the solvability of systems of causal equations. Clearly, this is a crucial issue determining the general applicability of the technique.

3.4. Analytical Solution of Systems of Causal Equations.

In this section we discuss the existence of analytical solutions of systems of causal sequence equations. Here we use the term sequence equation in a restrictive manner to indicate an equation in which the left side is a sequence and the right side is a sequence expression. This is the only type of equations needed for modeling the operation of systolic networks.

We begin by defining the dependency matrix which describes the structure of a given system of sequence equations. Then an iteration operator I_η is introduced and, with the help of this operator, it is shown that the solution of any system of causal equations can be expressed in analytical form. Finally, we present some examples that demonstrate the applicability of the iteration operator for the analytical verification of systolic networks.

3.4.1. Definition and Existence of Analytical Solutions.

In order to discuss systems of equations on sequences without referring to the network that are modeled by these equations, let Q denote the set of all sequences that appear in a given system of sequence equations. This set Q is partitioned into three disjoint, mutually exclusive sets, namely, the set of input sequences Q_p, the set of output sequences Q_o and the set of intermediate sequences Q_r. Here, an input sequence is a sequence that does not appear on the left side of any equation in the system, an output sequence is a sequence that does not appear on the right side of any equation in the system, and an intermediate sequence is a sequence in Q which is neither in Q_p nor in Q_o.

Accordingly, a solution of the given system of sequence equations is defined as a set of formulas, involving only well defined sequence operators, that explicitly describe the sequences in Q_o in terms of those in Q_p. Here, a well defined operator is understood to mean any operator whose image can be obtained from its operands using a deterministic expression. The operators defined in Chapter 2 are examples of well defined operators.

Let q_p, q_o and q_r be the cardinalities of the sets Q_p, Q_o and Q_r, respectively. We enumerate the sequences in Q by integers $i=1,\cdots,q_p+q_o+q_r$, such that for any sequence $\xi_i \epsilon Q$,

$$\begin{array}{ll} \xi_i \ \epsilon Q_o & \text{if } i \leq q_o \\ \xi_i \ \epsilon Q_r & \text{if } q_o < i \leq q_o+q_r \\ \xi_i \ \epsilon Q_p & \text{if } q_o+q_r < i \leq q_o+q_r+q_p \end{array}$$

The structure of the system of equations can then be described in terms of the dependency matrix A which is a binary, square matrix of order $q_o+q_r+q_p$ with the elements

$$a_{i,j} = \begin{array}{l} 1 \text{ if } \xi_j \text{ appears on the right side of the equation describing } \xi_i, \\ 0 \text{ otherwise.} \end{array}$$

For example, consider the following two systems of sequence equations:

System S

$$\begin{aligned} \xi_1 &= \Gamma_1(\xi_3 \ , \ \xi_6) \\ \xi_2 &= \Gamma_2(\xi_4 \ , \ \xi_5) \\ \xi_3 &= \Gamma_3(\xi_4 \ , \ \xi_6) \\ \xi_4 &= \Gamma_4(\xi_5 \ , \ \xi_7) \\ \xi_5 &= \Gamma_5(\xi_6 \ , \ \xi_7) \end{aligned}$$

System $\overline{S}$

This is the same as system S *except that the last equation is replaced by*

$$\xi_5 = \overline{\Gamma}_5(\xi_3 \ , \ \xi_6 \ , \ \xi_7)$$

Here Γ_i, $i=1,\cdots,5$ and $\overline{\Gamma}_5$ are sequence operators.

In both systems, we have $Q_p = (\xi_6,\xi_7)$, $Q_o = (\xi_1,\xi_2)$ and $Q_r = (\xi_3,\xi_4,\xi_5)$ and the dependency matrices are

$$A = \begin{bmatrix} 0&0&1&0&0&1&0\\ 0&0&0&1&1&0&0\\ 0&0&0&1&0&1&0\\ 0&0&0&0&1&0&1\\ 0&0&0&0&0&1&1\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} \qquad \bar{A} = \begin{bmatrix} 0&0&1&0&0&1&0\\ 0&0&0&1&1&0&0\\ 0&0&0&1&0&1&0\\ 0&0&0&0&1&0&1\\ 0&0&1&0&0&1&1\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{bmatrix} \quad (3.36)$$

for S and $\bar{S}$, respectively.

From the definition of the input, output and intermediate sequences, it is clear that any dependency matrix A can be partitioned into the following form:

$$A = \begin{bmatrix} O & A_{o,r} & A_{o,p} \\ O & A_{r,r} & A_{r,p} \\ O & O & O \end{bmatrix}$$

where the dimension of the sub-matrices $A_{o,r}$, $A_{o,p}$, $A_{r,r}$ and $A_{r,p}$ are $q_p \times q_r$, $q_p \times q_o$, $q_r \times q_r$ and $q_r \times q_o$, respectively, and each O denotes a zero sub-matrix of the appropriate dimension. This decomposed form of the matrix A shows that our ability of expressing explicitly the sequences in Q_o in terms of those in Q_p depends only on the structure of the submatrix $A_{r,r}$. In other words, if $A_{r,r}$ is a strictly lower or strictly upper triangular matrix, then by back substitution, we can easily express the sequences in Q_r in terms of those in Q_p. This in turn enables us to relate explicitly the sequences in Q_o to those in Q_p for any form of the submatrices $A_{o,r}$ and $A_{o,p}$. For example, the matrix $A_{r,r}$ corresponding to the matrix A in (3.36) is strictly upper triangular. Hence, for the system of equations S, we obtain by back substitution

$$\xi_4 = \Gamma_4(\ \Gamma_5(\xi_6, \xi_7)\ ,\ \xi_7) = \Lambda_4(\xi_6, \xi_7)$$
$$\xi_3 = \Gamma_3(\ \Lambda_4(\xi_6, \xi_7)\ ,\ \xi_6) = \Lambda_3(\xi_6, \xi_7)$$

which leads to

$$\xi_1 = \Gamma_1(\ \Lambda_3(\xi_6.\xi_7)\ ,\ \xi_6) = \Lambda_1(\xi_6.\xi_7)$$
$$\xi_2 = \Gamma_2(\ \Lambda_4(\xi_6.\xi_7)\ ,\ \Lambda_5(\xi_6.\xi_7)) = \Lambda_2(\xi_6.\xi_7)$$

where the operators Λ_i, i=1,....4 are defined in terms of the known sequence operators Γ_i, and hence are themselves well defined sequence operators.

It should be noted that the structure of the matrix $A_{r,r}$ depends primarily on the numbering of the sequences in Q_r. More specifically, whenever there exists a numbering that results in $A_{r,r}$ being strictly upper or strictly lower triangular, then, as stated, it is possible to solve the corresponding system by back substitution. This situation applies only if the system of causal equations does not contain any direct or indirect recursion.

On the other hand, if the system of equations does contain recursion, then for any numbering of the sequences in Q_r, the matrix $A_{r,r}$ cannot be strictly upper or lower triangular, and hence, the simple back substitution scheme cannot be carried to completion. For example, in the system of equations $\overline{S}$, we cannot express the sequences ξ_3 and ξ_4 in terms of ξ_6 and ξ_7 unless we have a method of solving the coupled equations

$$\xi_4 = \overline{\Lambda}_4(\xi_3.\xi_6.\xi_7) \tag{3.37.a}$$
$$\xi_3 = \overline{\Lambda}_3(\xi_4.\xi_6) \tag{3.37.b}$$

where $\overline{\Lambda}_3$ and $\overline{\Lambda}_4$ are well defined operators.

Yet, in the special case when the operators $\overline{\Lambda}_3$ and $\overline{\Lambda}_4$ are causal operators, it is possible to calculate the sequences ξ_3 and ξ_4 for any given specific sequences ξ_6 and ξ_7. In other words, the equations (3.37.a/b) have always a solution. This is an indication that our inability to express this solution analytically is due to the fact that our notation suppresses the time dimension from the sequence equations. This motivates the introduction of the iteration operator.

3.4.2. The Iteration operator.

It can be easily shown that the solution of any coupled system of equations may be obtained if we have a means of solving recursive equations of the form

$$\zeta = \Gamma(\zeta, \xi_1, \cdots, \xi_n) \tag{3.38}$$

where Γ is some sequence operator. For example, the solution of the coupled system (3.37) may be obtained if we can solve the recursive equations resulting from the substitution of (3.37.b) into (3.37.a), namely

$$\zeta_4 = \overline{\Gamma}_4(\overline{\Gamma}_3(\xi_4, \xi_6), \xi_6, \xi_7) = \overline{\Lambda}(\xi_4, \xi_6, \xi_7)$$

In general, the solution of (3.38) may not be well defined. However, systems of sequence equations resulting from modeling systolic networks have the nice property that they contain only causal operators. Hence, we will consider (3.38) only for causal operators Γ.

Theorem 3.1: Given a causal operator $\Gamma:[\overline{R}_\delta]^{n+1} \rightarrow \overline{R}_\delta$, the solution ζ of

$$\zeta = \Gamma(\zeta, \xi_1, \cdots, \xi_n) \tag{3.39}$$

is well defined.

Proof: We prove this theorem by means of the following procedure for the computation of ζ:

```
ALG1
    1) Let $\alpha_0 = \delta^*$.
    2) FOR $k=1,2,\cdots$ DO
        2.1) Compute the sequence $\alpha_k$ as follows
             $\alpha_k(t) = \begin{cases} \alpha_{k-1}(t) & t<k \\ [\Gamma(\alpha_{k-1}, \xi_1, \cdots, \xi_n)](t) & t=k \\ \delta & t>k \end{cases}$
        2.2) Set $\zeta(k) = \alpha_k(k)$.
```

In order to prove that the sequence ζ computed by ALG1 satisfies (3.39) we define the step operators $S_k : \bar{R}_\delta \to \bar{R}_\delta$ for $k = 0, 1, 2, \cdots$ by

$$S_0 \; \zeta = \delta^*$$

and, for $k > 0$, by

$$[S_k \; \zeta](t) = \begin{cases} \zeta(t) & \text{if } 1 \leq t \leq k \\ \delta & \text{if } t > k \end{cases}$$

With this, it is directly seen that, for any t, $\alpha_k(t) = [S_k \zeta](t)$ and hence that $\alpha_k = S_k \zeta$. From ALG1, we then have

$$\zeta(t) = \alpha_t(t) = [\Gamma(S_{t-1} \; \zeta, \xi_1, \cdots \xi_n)](t) \tag{3.40}$$

However, the definition of causality implies that $\zeta(t)$ may depend only on any element $[S_{t-1} \; \zeta](i)$ with $i < t$; that is, we may replace $S_{t-1} \; \zeta$ in (3.40) by ζ. This gives

$$\zeta(t) = [\Gamma(\zeta, \xi_1, \cdots, \xi_n)](t)$$

and proves that the sequence ζ computed by ALG1 indeed satisfies the equation (3.39). ■

Theorem 3.1 proves the existence of a solution of recursive causal equations and gives a procedure for its computation. Our next goal is to provide a suitable notation for expressing this solution.

Definition: Let $\Gamma : [\bar{R}_\delta]^{n+1} \to \bar{R}_\delta$ be a given causal operator. The iteration operator I_{η_r} applied to the image sequence $\Gamma(\eta_1, \cdots, \eta_{n+1})$ with respect to any of the arguments η_r, $1 \leq r \leq n+1$ shall be defined by

$$\zeta = I_{\eta_r} \; \Gamma(\eta_1, \cdots, \eta_r, \cdots, \eta_{n+1})$$

where for any t

$$\zeta(t) = [\Gamma(\eta_1, \cdots, \zeta, \cdots, \eta_{n+1})](t)$$

Using a procedure similar to the one given in the proof of Theorem 3.1, we can show that the image sequence ζ in the above definition is well defined. Note that the sequence η_r in the combined operator $I_{\eta_r}\Gamma:[\bar{R}_\delta]^n \to \bar{R}_\delta$ is a dummy sequence which is needed only to indicate the argument of Γ to which the recursion is applied. In other words, the arguments of $I_{\eta_r}\Gamma$ are only $\eta_1, \cdots, \eta_{r-1}, \eta_{r+1}, \cdots, \eta_{n+1}$. With this definition, we can now prove the following theorem

Theorem 3.2: For any causal operator $\Gamma:[\bar{R}_\delta]^{n+1} \to \bar{R}_\delta$, the solution of the recursive equation

$$\zeta = \Gamma(\zeta, \xi_1, \cdots, \xi_n)$$

is given by

$$\zeta = I_\eta\ \Gamma(\eta, \xi_1, \cdots, \xi_n)$$

Proof: From the definition of the iteration operator we obtain, for any $t \geqslant 1$

$$\begin{aligned} \zeta(t) &= [I_\eta\ \Gamma(\eta, \xi_1, \cdots, \xi_n)](t) \\ &= [\Gamma(\zeta, \xi_1, \cdots, \xi_n)](t) \end{aligned}$$

which directly implies that

$$\zeta = \Gamma(\zeta, \xi_1, \cdots, \xi_n). \blacksquare$$

Theorem 3.2 provides a means for expressing the solution of recursive causal equations. Its application to the verification of systolic networks, however, depends on our ability to manipulate expressions that combine the iteration operator and other sequence operators. The following theorem provides the basis for such a manipulation.

Theorem 3.3: If $\Lambda:\bar{R}_\delta^{n+1} \to \bar{R}_\delta$ is a causal sequence operator, and $\Phi:\bar{R}_\delta \to \bar{R}_\delta$ is any sequence operator with the property that

$$\Lambda(\Phi\zeta,\ \Phi\xi_1,\cdots,\Phi\xi_n) = \Phi\ \Lambda'(\zeta,\xi_1,\cdots,\xi_n) \tag{3.41}$$

where Λ' may or may not be identical to Λ, then

$$I_\eta\ \Lambda(\eta,\Phi\xi_1,\cdots,\Phi\xi_n) = \Phi\ I_\eta\ \Lambda'(\eta,\xi_1,\cdots,\xi_n) \tag{3.42}$$

Proof: We write the right side of equation (3.41) as $\Phi\ \gamma$, where γ is given by

$$\gamma = I_\eta\ \Lambda'(\eta,\xi_1,\cdots,\xi_n)$$

By Theorem 3.2, we know that γ also satisfies

$$\gamma = I_\eta\ \Lambda'(\gamma,\xi_1,\cdots\xi_n);$$

that is

$$\Phi\ \gamma = \Phi\ I_\eta\ \Lambda'(\gamma,\xi_1,\cdots,\xi_n)$$

By the hypothesis (3.41) this reduces to

$$\Phi\ \gamma = I_\eta\ \Lambda(\Phi\gamma,\Phi\xi_1,\cdots,\Phi\xi_n)$$

which by Theorem 3.2 has the solution

$$\Phi\ \gamma = I_\eta\ \Lambda(\eta,\Phi\xi_1,\cdots,\Phi\xi_n)$$

Evidently, this is equal to the left side of equation (3.42). ■

We next give some examples that illustrate the applications of the iteration operator to the verification of systolic networks.

3.4.3. The iteration operator in the verification of systolic networks.

In this section, we present two examples for the application of the iteration operator to the derivation of the I/O description of systolic networks that are modeled by mutually coupled systems of equations. The first network is the back substitution network that was verified in Section 3.2 for specific input sequences. Here, we will derive an explicit I/O description for this network and show that the iteration operator does simplify the verification of this network. The second example illustrates an important fact, namely that even though it is always possible to obtain analytical formulas for the I/O description of systolic networks, those formulas may sometimes be very long and cumbersome to a point that they are complicating the verification procedure rather than simplifying it.

Example 1:

Consider the back substitution network of Section 3.2. In that section, we did not obtain explicitly the network I/O description because of the coupled equations

$$\rho_1 = \Omega\ [[\beta_0 - \sigma_0] \,/\, \alpha_0] \tag{3.43.a}$$

$$\sigma_0 = \Omega^k \sigma_k \oplus \sum_{j=1}^{k}{}^{\backprime}\ \Omega^j\ [\alpha_j * \Omega^{j-1}\ \rho_1] \tag{3.43.b}$$

However, by substitution of (3.43.b) into (3.43.a) we may obtain

$$\rho_1 = \Omega\ [[\beta_0 - [\Omega^k \sigma_k \oplus \sum_{j=1}^{k}{}^{\backprime}\ \Omega^j [\alpha_j * \Omega^{j-1} \rho_1]]] \,/\, \alpha_0]$$

which by Theorem 3.2 has the solution

$$\rho_1 = I_\omega\ \Omega\ [[\beta_0 - [\Omega^k \sigma_k \oplus \sum_{j=1}^{k}{}^{\backprime}\ \Omega^j [\alpha_j * \Omega^{j-1} \omega]]] \,/\, \alpha_0]$$

and leads directly to the explicit I/O description

$$\rho_{k+1} = \Omega^k\ I_\omega\ \Omega\ [[\beta_0 - [\Omega^k \sigma_k \ \mathrm{O} \sum_{j=1}^{k} \Omega^j [\alpha_j * \Omega^{j-1} \omega]]] \,/\, \alpha_0] \tag{3.44}$$

Equation (3.44) describes the output sequence ρ_{k+1} in terms of the input sequences α_j, $j=0,\cdots,k$, and σ_k. In order to verify the network for the specific input described by equations (3.23), we substitute these sequences in (3.44). This provides

$$\rho_{k+1} = \Omega^k\ I_\omega\ \Omega\ [[\Omega^k \ominus \eta - [\Omega^k \ominus \iota \oplus \sum_{j=1}^{k}{}^{\backprime}\ \Omega^j [\Omega^{k+j} \ominus \lambda_j * \Omega^{j-1} \omega]]] \,/\, \Omega^k \ominus \lambda_0]$$

and now the application of Theorem 3.3 for factoring $\Omega^{k+1}\Theta$ from the operand of I_ω gives

$$\begin{aligned}\rho_{k+1} &= \Omega^{2k+1}\Theta\ I_\omega\ [[\eta - [\iota \oplus \sum_{j=1}^{k} \Omega^j[\lambda_j * \omega]]] / \lambda_0] \\ &= \Omega^{2k+1}\ \Theta\ \xi\end{aligned}$$

where

$$\xi(t) = (\eta(t) - (\iota(t) \oplus \sum_{j=1}^{k} [\Omega^j[\lambda_j * \xi]](t)\ ))\ /\lambda_0(t)$$

This leads directly to the expression for $\xi(t)$ derived in Section 3.2.

EXAMPLE 2:

In this example (Figure 3.4), we consider the special case k=3 of the sorting network presented in Section 3.3. The operation of the network is modeled by the causal equations

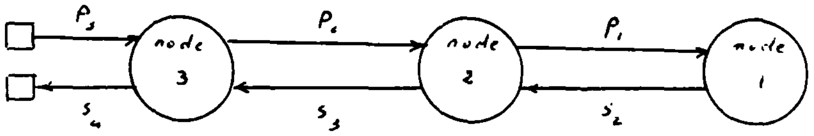


Figure 3.4 - A special case of the sorting network.

$$\sigma_2 = \Omega\ \pi_1 \tag{3.45.a}$$
$$\sigma_3 = \Omega\ \min_\delta(\pi_2\ ,\ \sigma_2) \tag{3.45.b}$$
$$\sigma_4 = \Omega\ \min_\delta(\pi_3\ ,\ \sigma_3) \tag{3.45.c}$$
$$\pi_1 = \Omega\ \max_\delta(\pi_2\ ,\ \sigma_2) \tag{3.45.d}$$
$$\pi_2 = \Omega\ \max_\delta(\pi_3\ ,\ \sigma_3) \tag{3.45.e}$$

In order to express the output sequence σ_4 in terms of the input sequence π_3, we start by solving (3.45.a/d) for σ_2. By Theorem 3.2, σ_2 is given by

$$\sigma_2 = I_\eta\ \Omega^2\ \max_\delta(\pi_2\ ,\ \eta) \tag{3.46}$$

We then solve (3.46) and (3.45.b/e) for σ_3 and substitute the result in (3.45.c) to obtain the network I/O description in the form

$$\sigma_4 = \Omega\min_\delta(\pi_3 \; , \; I_\zeta \Omega\min_\delta(\Omega\max_\delta(\pi_3,\zeta) \; , \; I_\eta \Omega^2\max_\delta(\Omega\max_\delta(\pi_3,\zeta) \; , \; \eta))) \qquad (3.47)$$

Although (3.47) describes explicitly the output in terms of the input, it may be very difficult to use it for the verification of the network for given inputs, especially if the size of the network k is to be kept as a parameter. As a matter of fact, the non systematic approach followed in Section 3.3 proved to be more effective for the formal verification of this sorting network.

4. COMPUTER SOLUTION OF SYSTEMS OF CAUSAL EQUATIONS.

It was shown in Section 3.4 that we can always obtain analytical solutions of systems of causal equations that model systolic networks. However, our tools for manipulating sequences are still limited and, in many cases, we are not able to derive the output sequences in a form that may be compared with the expected output of the network. In order to alleviate this problem, we describe in this chapter a computer system that was developed for solving iteratively any system of causal sequence equations for specifically given input sequences.

A simple language called SCE (Systems of Causal Equations) is used to provide the system of causal equations to the solver. It will be described in Section 1. The equation solver itself represents a syntax directed interpreter that executes any correct SCE program. This interpreter is outlined in Section 2; it reads the elements of the input sequences from an input file, and calculates the elements of the sequences on the left side of the equations specified by the program.

It should be noted that all data sequences considered here have infinite length but contain only finitely many elements different from the don't care element δ. Accordingly, an upper bound MAXT is assumed to be given by the user for the maximal index of non-δ data items of all sequences. In other words, elements beyond MAXT in any sequence are considered to be equal to δ.

Although the generality of the solver allows it to be used for wide range of tasks, its immediate application will be to simulate computations on systolic networks. For this, an SCE program is written which implements the equations for the operation of the network, and then, an input file is created that contains the elements of the input data sequences. Now a run of the SCE interpreter provides the elements of the output data sequences. This approach to the simulation of systolic networks separates the internal details of the simulator from the concern of the user. It also has the advantage of allowing the user to begin with a partial solution of the system of equations modeling the network and then to use the SCE interpreter on the portion of the system that could not be solved analytically. As an example, we show in Section 3 how the operation of a network for the LU decomposition of a symmetric banded matrix may be simulated by means of the SCE interpreter.

4.1. The SCE language for specifying Systems of Causal Equations.

The SCE language is a simple expression language augmented with some input/output facilities and looping capabilities that provide for efficiency in the writting of programs. In its current form, SCE may be used to model the operation of a special class of systolic networks in which the units of information are real numbers. However, by the addition of new rules to the grammar, it is possible to model other types of systolic networks at higher or lower levels of architectures.

By the first rule in the grammar given in Appendix B, it is readily seen that an SCE program consists of the following four parts: 1) The declarations, 2) the input part, 3) the programs body, and 4) the output part. In the rest of this section, we will discuss the semantics of the language.

Terminal symbols in SCE (see Appendix B) can be classified into four categories, namely, special symbols (e.g. +, -, *, ...), reserved words (e.g. FOR, OUT, ...), identifiers and constants, where a constant is either a positive integer or a positive real number written in floating point format.

In order to ensure a clear distinction between identifiers and reserved words in SCE, we have chosen all the reserved words in the language to start with capital letters. On the other hand, any string of alphanumeric characters starting with a lower case letter can be used as an identifier with the understanding that only the first six characters are significant. Identifiers in SCE should be declared in the declaration part of the program to be of one of the following types:

1) **Parameter** (rules 3-7): A parameter is assigned an integer value at the time of its declaration and this value is substituted textually whenever the identifier appears in the program.

2) **Index** (rules 8-11): An index in SCE is an integer variable used in loop control and in the selection of elements of sequence arrays.

3) **Sequence** (rules 12-19): Sequences are represented on the machine by vectors. An identifier of type "sequence" may be associated with either a single sequence (rule 16) or with an array of sequences (rule 15). For arrays of sequences, the dimension and the lower and upper bounds are specified in the declaration by enclosing these bounds in curly brackets. For example, the following SCE statement declares s as an n dimensional sequence array

$$SEQN\ s\{l_1:u_1\ ,\ \cdots\ ,l_n:u_n\}$$

where l_i and u_i, $i=1,\cdots,n$ are the lower and upper bounds for the i^{th} dimension. Bounds may be negative but should of course satisfy the restriction that $u_i \geq l_i$. We also note that there is no limit on the dimensionality of an array.

After the declaration of an array of sequences, its elements may be identified (rules 38-41) by using the usual selection notation

$s(p_1, \cdots, p_n)$, where each p_i has to fall into its corresponding range, that is $l_i \leqslant p_i \leqslant u_i$.

In the context of the abstract systolic model, a data sequence is associated with a communication link which is identified by its color and the label of the node at which it terminates. In order to simplify the SCE specification of a systolic network, we label the nodes in the network by n-tuples of integers $(v_1, \cdots, v_n)$ with some fixed n. This enables us to group all the sequences associated with the links that have the same color in an n-dimensional array. Of course the color of the link may be used to identify the array. With this, the sequence associated with any link $y_{v1,...,vn}$ is simply the *element* $(v_1, \cdots, v_n)$ *of the sequence array* $y()$.

Although this leads usually to a very clear SCE specification of a systolic network, it is sometimes inefficient because some of the elements in the array may not be used. For example, in the LU network described in Section 3, a triangular array of sequences would be more space efficient than the rectangular array allowed by SCE. In such cases, a more efficient storage arrangement could be obtained by applying any one of the techniques used for storing triangular and sparse matrices [18].

In addition to arrays of sequences, the language allows the user to declare single sequences. Three standard single sequences are predefined by the language, namely the don't care sequence, the zero sequence and the unity sequence. The first two sequences were defined in Section 2.1. *The unity sequence* τ, *as its name implies*, is defined by $\tau(t)=1.0$ for $1 \leqslant t \leqslant T(\tau)$ and arbitrary large $T(\tau)$. The sequences δ^*, ι and τ are denoted in SCE by the identifiers d, o and u, respectively. The user however may re-declare the identifiers d, o or u if he wishes to change their definitions.

The input part of an SCE program has the form of a single INPUT statement (rules 68-73). It serves two purposes; Firstly, it assigns an integer value to MAXT, which specifies the number of elements to be considered in any sequence, and secondly, it specifies the sequences to be read from the input file. Nested FOR loops can be used, to any level, in specifying the input sequences. For example, in the program presented in Section 3, the statement

INPUT(MAXT 18 , FOR i=0,3 r(i,1));

specifies that MAXT=18 and that the input sequences are r{0,1}, r{1,1}, r{2,1} and r{3,1}.

Similarly, the output part of the program takes the form of a single statement (rules 74-78) that specifies the sequences to be printed on the output file. Again, FOR loops are allowed.

The body of the SCE program is the part that contains the specification of the system of sequence equations. It consists of a list of statements, where a statement may be either a sequence equation or a FOR loop that encloses a list of statements. Each equation has the form

sequence specification = sequence expression

where the left side identifies a particular sequence and the right side is an expression composed of sequence identifiers and sequence operators. Square brackets may be used in sequence expressions to override the precedence rules defined by the grammar. Basically, in the evaluation of expressions, the grammar associates the highest priority with the operators defined directly on sequences. Next in priority is the scalar multiplication operator '.', followed by the operators '*' and '/'. Finally, the operators '+' and '-' are evaluated with the lowest priority. With these precedence rules the sequence expressions are evaluated from left to write.

Although many other sequence operators may be incorporated into the language, we only allowed for the following operators:

The positive shift operator Ω^{r}, written in SCE as O(r),

The zero shift operator Ω_{0}^{r}, written in SCE as Z(r),

The spread operator Θ^{r}, written in SCE as T(r),

The expansion operator E_{r}^{k}, written in SCE as E(r,k),

The accumulator operator $A^{r,k,s}$, written in SCE as A(r,k,s) and

The multiplexing operator $M_{r}^{w1,\ldots,wn}$ written in SCE as M(r;w1,· · ·,wn).

Frequently, the shift, zero shift and spread operators are used with r=1. For this, the short hand notation O, Z, and T may be used instead of O(1), Z(1) and T(1), respectively.

The two element-wise operators U1 and U2 of rules 47 and 48 have the same priority as the operators * and / but their semantics are not specified by the language. As indicated in Section 3, U1 and U2 may be defined by the user.

Finally, we note that rule 55 restricts the operands of the accumulator operator to single sequences rather than sequence factors as is the case with the other operators. This restriction is not necessary and was only imposed because it leads to a more efficient implementation of the SCE interpreter. However, it should be noted that this does not affect the expressive power of the language because we can always define intermediate sequences to get around this restriction. For example, the sequence equation

$$x(i) = O(2)\ x(i-1) + A(1,3,1)\ [\ y(i+1) * T\ x(i-1)\ ]$$

which is not permitted in SCE can be split into the two SCE legal equations

$$v(i) = y(i+1) * T\ x(i-1)$$

$$x(i) = O(2)\ x(i-1) + A(1,3,1)\ v(i).$$

4.2. Overview of the SCE Interpreter.

As is the case with most language interpreters, the SCE solver has two distinct phases, namely, the syntax analysis phase which, using a parse tree of the program, produces an intermediate language program, and the actual interpretation phase which executes the intermediate program.

For the syntax analysis phase, we used the automatic parser generator YACC [25] existing on the UNIX operating system to generate an LR(1) bottom-up parser that accepts any syntactically correct SCE program and generates an intermediate program in the form of a sequence of tuples. It is basically a finite state machine with a stack. It scans the input program from left to right and is capable of reading and remembering the next input token (terminal symbol) which is called the look-ahead token. Depending on the look-ahead token and the content of the stack, the parser takes one of the following actions:

1) **Shift:** The current look-ahead token is pushed into the stack and the next token is read in. Also a tuple describing the action is generated. If the token being shifted is a special symbol, an identifier, or a reserved word, the tuple generated has the form (Shift,n), where n is a number identifying the token. On the other hand, if the token is an integer or a real constant, then the tuple generated has the form (Shift-integer,c) or (Shift-real,r), respectively, where c or r is the value of the constant.

2) **Reduce:** This action is taken when the parser recognizes that the stack contains the right hand side of a grammar rule, say rule n, and that this rule should be applied at this point. It then pops from the stack the

tokens forming the right side of rule n and pushes onto the stack the token on its left side. It also generates a tuple (Reduce,n).

3) **Accept:** This action is taken when the parsing process is successfully completed. The tuple generated in this case is (Accept,0).

4) **Error:** If the parser discovers that the program is syntactically incorrect, it simply gives a warning and halts. Of course, more elaborate error handling actions could have been taken if our goal was to produce a more sophisticated parser. For the details and internal forms of LR parsers, we refer to [2].

The second phase of the interpreter reads the intermediate program (sequence of tuples) and reproduces the actions taken by the parser on an action stack. Simultaneously, an adjoint value stack is used to store temporary values needed in the interpretation. In Appendix C, we give the complete listing of a C program for the second phase of our SCE solver, and in the rest of this section we will outline the main features of this solver/interpreter.

The program uses a location counter "location" to indicate the intermediate tuple being interpreted. Starting with location =1, the interpreter reads the tuple pointed to by "location", takes a certain action, increases location by one and then repeats the above cycle. The action taken in each cycle depends on the type of the tuple being interpreted:

1) If the tuple is of the type (Shift,n) or (Shift-integer,n), then n is pushed into the action stack.

2) If the tuple is of the type (Shift-real,r), then r is pushed into the value stack and a zero is pushed into the action stack.

3) If the tuple is of the type (Accept,0), then the interpretation is terminated.

4) If the tuple is of the type (Reduce,n), where grammar rule n has the form $b \rightarrow a_1\ a_2\ \cdots\ a_k$ then the interpreter pops the k top locations of the stack, which should contain the symbols $a_1, \cdots, a_k$ and pushes b. It also may execute a semantic routine if any is associated with this grammer rule. These routines manipulate the data on the value stack to reflect the semantics of the grammar rule.

At this point we note that we do not actually have to push or pop the grammar symbols in the action stack, and that it suffices to keep track of the top of the stack. Then at the time of a reduction, the grammar rule and the top of the stack determine uniquely the location that each symbol would occupy on the stack. With this, we can transmit information from one semantic routine to another by pushing this information into the stack in the place of the grammar symbol. For example, the semantic routine associated with rule 36 uses the location of the FOR symbol on the stack to store the starting address of the first statement in the FOR loop body. Then, when the execution of the FOR body is terminated, the routine for rule 35 retrieves this address from the stack to re-initiate the execution of the FOR body if the final value of the index of the loop is not yet reached.

In order to reduce the storage required for holding the intermediate tuples, the program in Appendix C reads and executes the tuples in four stages: In the first and the second stage, the declaration and the input part of the program are processed, respectively. In the third stage, the system of equations is solved, and finally in the fourth stage the output is printed. We briefly comment on each stage.

The declarations: The main objective of this stage is to construct the symbol table and to allocate storage for the declared sequences. The symbol table "sym_tab[]" is an array of records with three fields. The first field contains a character that indicates the type given to the identifier, namely P, I or S for parameters, indices or sequences, respectively. The interpretation of the integers in the second and third fields, called entry1 and entry2, depends on the type of the identifier. For parameters, entry1 contains the value of the parameter and entry2 is immaterial. For index variables, entry1 holds the value of the index, initialized to zero, and entry2 is set during the execution of a FOR loop to the final value of the loop index.

Finally, if the identifier is declared as a sequence variable, then it may denote a single sequence or an array of sequences. Single sequences are distinguished by setting entry2=-1, with entry1 pointing to the location where the sequence is stored. For arrays of sequences, entry2 holds a pointer to a bound table that indicates the dimension and the bounds of each array, and entry1 points to the location where the first sequence in the array is stored. The first three locations in the symbol table are reserved for the identifiers d, o and u, respectively that are preset to the don't care, the zero and the unity sequences, respectively. However, if any of these identifiers are declared in the program, then the corresponding entry in the symbol table is overwritten by the semantics routine corresponding to the new declaration.

The sequences are stored in a two dimensional array seq_store[][]. Each row in the array has a length at least equal to MAXT and is used to store the elements of a sequence. Arrays of sequences are stored in consecutive rows such that any index changes slower than the one to its right, if any. In order to keep track of don't care elements, an array d-table[][] of bits is used such that for each element in seq_store[][], there is a corresponding bit in d-table[][]. This bit is set to one, if the element in seq_store[][] is a don't care, and to zero, otherwise. Thus, any part of the program that reads an element from seq_store[][] has to inspect also the corresponding entry in d-table[][].

The implementation of the SCE solver listed in Appendix C allows for full causality in the sense that an output may depend on any previous input. Accordingly, storage is provided for the retention of at least MAXT elements from any sequence. A more space-efficient implementation would be to retain only the last C elements from each sequence in a circular buffer, where C is given. This allows only for C-order causality in the sense that the output of a certain cell at any given time t may only depend on the inputs to that cell during the time period from t-C to t-1.

The input part: The INPUT statement specifies the sequences to be read from the input file as well as the number MAXT of elements in each sequence. The interpreter reads, as a stream, the MAXT elements of the first specified sequence followed by those of the second sequence, etc., provided that the elements are separated by at least one space. No special characters are required to separate the elements of the different sequences. Each element in the input file may be either a floating point number or the letter "d" representing a don't care element. The interpreter also recognizes the string "· · ·" in the input file as an indication that the remaining elements in that sequence are don't cares.

The equation solver: The sequence of tuples in the body of the program are executed iteratively MAXT times. A global clock "TIME" is initialized to 1 and incremented at every step of the iteration. At every step, the expression on the right side of each equation is evaluated at time TIME and assigned to the corresponding element of the sequence on the left side of the equation.

The value stack is used during the evaluation of sequence expressions to store temporary results. For example, the semantic routine associated with the grammar rule 52 (seq_factor → seq_spec) reads the value of the element TIME in the specified sequence from seq_store[][], and pushes it onto the value stack. The result of any subsequent sequence operation is stored on the stack until rule 37 is executed and the final result is stored back into seq_store[][].

The sequence operators O, Z, T and E operate on sequence factors and have the effect of changing the global clock during the evaluation of the corresponding factor. The old clock value is stored in the action stack for retrieval after the evaluation of the factor is complete (rule 53). If the result of any operation involving the above operators is the don't care element, then the flag "skip" is set which causes the execution of the semantic routines to be skipped until the corresponding tuple (Reduce,53) is encountered. Of course provisions are made to deal with arbitrary degrees of nesting.

In a similar way, the flag "Mskip" is used to chose the appropriate operand in the multiplexer operator. Finally, we note that by restricting the operands of the accumulator operator to sequences instead of sequence factors, we simplified greatly the action associated with that operator. For a detailed description of the different semantics routines, we refer to the complete listing of the program in Appendix C.

It is important to note that the SCE interpreter detects any inconsistency in the given equations or any attempt of solving equations which are not causal or weakly causal. It does so by associating with each sequence an entry in the array last_computed[] to keep track of the last element that has been computed in the sequence so far. Any attempt to overwrite an already calculated element or to read an element that has not yet been calculated is then easily detected and reported as a run time error. The interpreter also detects other types of run-time errors that are listed in the function run_error in the Appendix.

The output part: After completing the interpretation of the body of the program, the sequences specified by the user are printed on the standard output file.

The SCE simulator/solver was used to simulate the operation of the systolic networks that have been verified analytically in Chapters 3 and 7. In the next section, we will illustrate its application by applying it to the simulation of an LU factorization network that will be used in Chapter 9. Although this approach to the simulation of systolic networks is very simple, it should be clear that it can be used only to verify instances of computations; that is, all architectural parameters and input data have to be given specific values during the simulation. Of course, this observation applies to any simulator or numeric solver, and the only way to allow for a *more* general simulator would be to consider symbolic manipulation which produces a symbolic description of the outputs in terms of the inputs.

Finally, we note that a possible optimization of the implementation of the interpreter could be achieved by replacing the single value stack by k stacks, for some optimal k. This would reduce the total number of iterations through the body of the program by considering at each step the elements TIME, TIME+1, $\cdots$, TIME+k of the sequences instead of only one element at a time. However, if the system contains any recursion, then only few of these k elements (and in many cases only one) can be considered at each step, and this requires more complicated book keeping to update the array last_computed[] and the global clock. We decided not to implement this optimization because we intended the solver to be used in cases where analytical solutions of the system of equations are difficult, and hence where recursivity is usually present.

4.3. Example: An LU factorization network.

In this section the SCE interpreter is applied to the simulation of the computations of a network for the LU or the U^TDU factorization of a symmetric banded matrix A and the solution of the linear system of equations Ax=y with a given vector y. It will be shown also that, with slight modifications, the same network can be used to compute the Cholesky decomposition LL^T of the matrix A.

The first systolic network for factoring a banded matrix into the product of a lower triangular matrix and an upper triangular matrix was suggested by Kung and Leiserson [31]. Later, Brent and Luk [7] modified the Kung and Leiserson network to compute the Cholesky decomposition of symmetric matrices. The network described in this section is also designed for symmetric matrices but is different in its operation principle from the one given in [7]. Both networks use almost the same number of computational cells and achieve approximately the same speed-up over serial execution. They differ however in the type of computational cells and in their interconnections.

Consider the system of linear equations

$$A\ x = y \tag{4.1}$$

where A is an $n \times n$ matrix and x and y are n dimensional vectors. The solution x of (4.1) may be obtained by finding a lower triangular matrix L and an upper triangular matrix U such that $A = L\ U$, and by solving the two triangular systems $L\ z = y$ and $U\ x = z$. More specifically, assuming that A is symmetric and banded with band width 2k+1, and denoting the elements of the matrices A, L and U by $a_{i,j}$, $l_{i,j}$ and $u_{i,j}$, respectively, we may use the following algorithm to compute the *LU* decomposition of A. Note that only the elements $a_{i,j}$ with $j \geq i$ are used and that only the non zero elements of L and U are computed.

ALG2 : LU factorization.

```
FOR i=1, ··· ,n DO
    1) FOR j=i, ··· , min(n, i+k) DO
        1.1) $l_{j,i} = a_{i,j}$
        1.2) $u_{i,j} = \frac{a_{i,j}}{a_{i,i}}$
    2) FOR q=i+1, ··· ,min(n, i+k) DO
        FOR j=q, ··· ,min(n, i+k) DO
            $a_{q,j} = a_{q,j} - l_{q,i} u_{i,j}$
```

At this point we note that the matrix *L* obtained by the above algorithm satisfies $L=U^TD$, where *D* is the diagonal matrix defined by $d_{i,i}=l_{i,i}$. Also, by replacing steps 1.1 and 1.2 in ALG2 by

$$l_{j,i} = u_{i,j} = \frac{a_{i,j}}{\sqrt{a_{i,i}}}$$

we obtain an algorithm for the Cholesky decomposition LL^T of A.

After having performed the LU decomposition of A, we may compute the vector $z=L^{-1}y$ by the following algorithm

ALG3 : Back substitution.

```
FOR i=1, ··· ,n DO
    $z_i = \frac{y_i}{l_{i,i}}$
    FOR q=i+1, ··· ,min(n, i+k) DO
        $y_q = y_q - l_{q,i} z_i$
```

Finally, the solution of $Ux=z$ may be obtained by an algorithm similar to ALG3, or alternatively by using ALG3 itself for the solution of $\overline{U}\ \overline{x} = \overline{z}$, where $\overline{z}_i = z_{n-i+1}$, $\overline{x}_i = x_{n-i+1}$ and $\overline{U}$ is a banded lower triangular matrix with the elements $\overline{u}_{i,j} = u_{n-i+1,n-j+1}$.

As an example, let n=5 , k=2 and

$$A = \begin{bmatrix} 2 & 4 & 6 & 0 & 0 \\ 4 & 11 & 15 & -3 & 0 \\ 6 & 15 & 20 & 2 & -2 \\ 0 & -3 & 2 & -19 & 1 \\ 0 & 0 & -2 & 1 & 14 \end{bmatrix} \quad \text{and} \quad y = \begin{bmatrix} 4 \\ 14 \\ 15 \\ 0 \\ -6 \end{bmatrix} \tag{4.2}$$

By ALG2 we obtain

$$L = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 4 & 3 & 0 & 0 & 0 \\ 6 & 3 & -1 & 0 & 0 \\ 0 & -3 & 5 & 3 & 0 \\ 0 & 0 & -2 & -9 & -9 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & -5 & 2 \\ 0 & 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

and by ALG3

$$z = \begin{bmatrix} 2 \\ 2 \\ 3 \\ -3 \\ 3 \end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix} -41 \\ -19 \\ 27 \\ 6 \\ 3 \end{bmatrix} \tag{4.4}$$

The graph of the systolic network that executes ALG2 and ALG3 simultaneously is shown in Figure 4.1. It is composed of $\frac{(k+1)(k+4)}{2}$ interior nodes. Each node is labeled by a pair (i,j), where i and j are the coordinates of the node with respect to the two axes shown in the figure. The color of each edge is determined by its direction. More specifically, edges directed to the east, south and south west are given the colors s, b and c, respectively, and those directed north are assigned the colors r or p depending on their relative position.

The part of the graph that is formed by nodes (i,j), $i=1,\cdots,k+1$, $j=1,\cdots,k-i+2$ represents a subnetwork that executes ALG2. It consists of three types of nodes whose operation is described by the following equations:

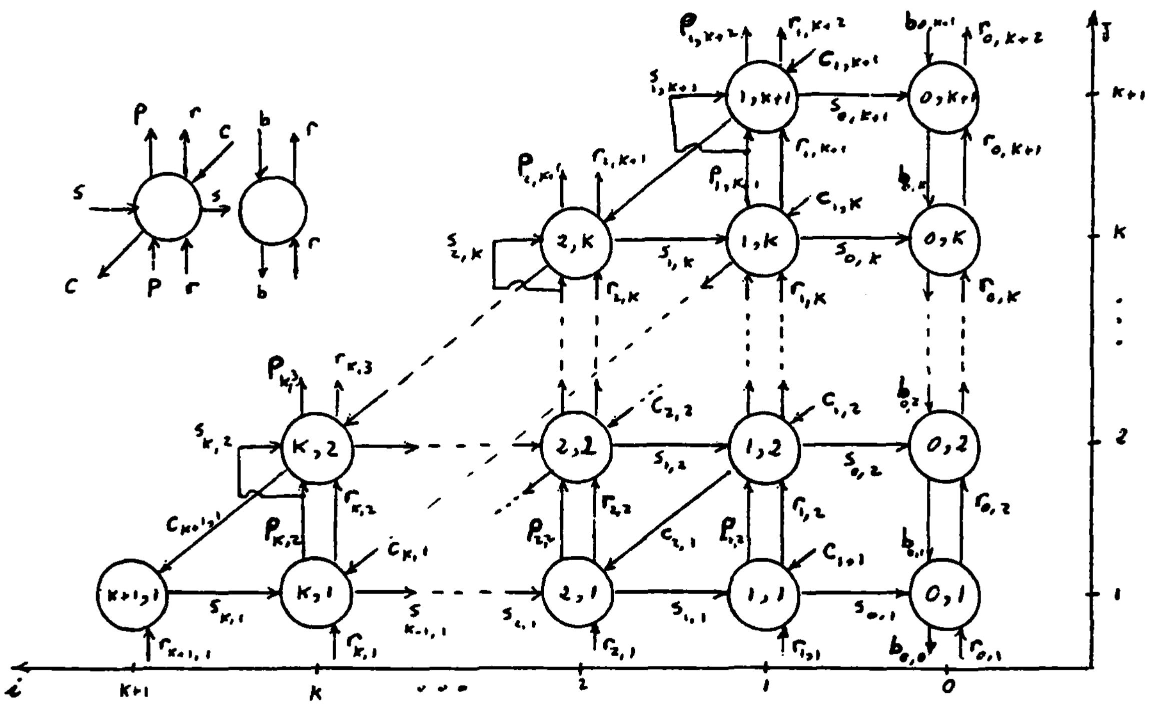


Figure 4.1 - The graph for the LU network

For node (k+1,1)

$$\sigma_{k,1} = \Omega_0 \; [\tau \; / \; [\rho_{k+1,1} - \gamma_{k+1,1}]] \tag{4.5}$$

where τ is the unity sequence defined by $\tau(t)=1.0$ for any t.

For nodes (i,1), i=1, · · · ,k

$$\rho_{i,2} = \Omega_0 \; [\rho_{i,1} - \gamma_{i,1}] \tag{4.6.a}$$

$$\pi_{i,2} = \Omega_0 \; [\sigma_{i,1} * [\rho_{i,1} - \gamma_{i,1}]] = \Omega_0 \; \sigma_{i,1} * \rho_{i,2} \tag{4.6.b}$$

$$\sigma_{i-1,1} = \Omega_0 \; \sigma_{i,1} \tag{4.6.c}$$

For nodes (i,j), j=2,· · ·,k+1, i=1,· · ·,k+2−j

$$\sigma_{i-1,j} = \Omega_0 \; \sigma_{i,j} \tag{4.7.a}$$

$$\pi_{i,j+1} = \Omega_0 \; \pi_{i,j} \tag{4.7.b}$$

$$\rho_{i,j+1} = \Omega_0 \; \rho_{i,j} \tag{4.7.c}$$

$$\gamma_{i+1,j-1} = \Omega_0 \; [\gamma_{i,j} + \rho_{i,j} * \sigma_{i,j}] \tag{4.7.d}$$

On the other hand, the part of the graph composed of the nodes (0,j), $j=1,\cdots,k+1$ corresponds to a subnetwork that executes ALG3. The operations of the cells in this subnetwork are described by

For node (0,1)

$$\rho_{0,2} = \Omega_0 \, [\rho_{0,1} - \beta_{0,1}] \tag{4.8.a}$$
$$\beta_{0,0} = \Omega_0 \, [\sigma_{0,1} * [\rho_{0,1} - \beta_{0,1}]] = \Omega_0 \, \sigma_{0,1} * \rho_{0,2} \tag{4.8.b}$$

For nodes (0,j), j=2, $\cdots$,k+1

$$\rho_{0,j+1} = \Omega_0 \, \rho_{0,j} \tag{4.9.a}$$
$$\beta_{0,j-1} = \Omega_0^2 \, [\beta_{0,j} + \sigma_{0,j} * \rho_{0,j}] \tag{4.9.b}$$

Note that the nodes $(i,1)$, $i=0,\cdots,k$ correspond to subtract/multiply cells, while the nodes (i,j), $j=2,\cdots,k+1$, $i=1,\cdots,k+2-j$, are multiply/add cells. Only the node (k+1,1) is a subtract/divide cell. In other words, the network is composed of three basic types of simple computational cells.

For the proper operation of the network, the input sequences $\gamma_{1,j}$, $j=1,\cdots,k+1$ and $\beta_{0,k+1}$ are set to the zero sequence ι, and the input links $s_{k+2-j,j}$, $j=1,\cdots,k+1$, are connected to the links $p_{k+2-j,j}$ (see figure), that is

$$\gamma_{1,j} = \iota \qquad j=1,\cdots,k+1 \tag{4.10.a}$$
$$\beta_{0,k+1} = \iota \tag{4.10.b}$$
$$\sigma_{k+2-j,j} = \pi_{k+2-j,j} \qquad j=2,\cdots,k+1 \tag{4.10.c}$$

The elements of the matrix A and the vector y are fed into the network through the links $r_{i,1}$, $i=1,\cdots,k+1$ and $r_{0,1}$, respectively. The precise input specification is given by

$$\rho_{i,1} = \Omega^{k+1-i} \, \Theta^2 \, \alpha_i \qquad i=1,\cdots,k+1 \tag{4.11.a}$$
$$\rho_{0,1} = \Omega_0^{k+1} \, \Theta^2 \, \eta \tag{4.11.b}$$

where $T(\alpha_i)=n-(k+1-i)$, $T(\eta)=n$ and

$$\alpha_i(t) = a_{t,t+k+1-i}$$
$$\eta(t) = y_t$$

In other words, α_{k+1-q} contains the n-q elements of the q^{th} off diagonal of A, and η the n elements of the right hand side vector y.

In order to understand the principle of operation of the network, we first note that at iteration step i of algorithm ALG2, the i^{th} row of U and the i^{th} column of L are computed from the i^{th} row of A (steps 1.1 and 1.2). However, the elements of the matrix A are continuously modified. In particular, at the execution of step i, the elements in row i of A had been modified by subtracting from them different contributions (step 2) during the steps $i-k,\cdots,i-1$. In the systolic network of Figure 4.1, the elements of the unmodified i^{th} row of A arrive at the cells $(q,1)$, $q=1,\cdots,k+1$, on the r colored links. At the same time, the sum of the contributions from the previous iterations $i-k,\cdots,i-1$ arrive at the same cells on the c colored links. The subtraction is then performed and the elements of the corresponding column and row of L and U are computed and sent out on the r and p colored links, respectively. These elements propagate upward in the network allowing the cells (q,j), $j=2,\cdots,k+1$, $q=1,\cdots,k+2-j$ to compute and sum the contributions for the modification of the subsequent rows of A. These contributions are sent downward on the c colored links. The subnetwork formed by the cells $(0,j)$, $j=1,\cdots,k+1$ operates in a similar way.

A closer study of the behavior of the network shows that the significant elements of the matrix U are sampled from the links $p_{q,1}$, $q=1,\cdots,k$, and the elements of the partial solution vector z from the link $b_{0,0}$. These results are sufficient for the computation of the solution $x=U^{-1}z$. However, the elements of the diagonal matrix D, where $L=U^T D$, are also available on the link $s_{k,1}$. More precisely, the output sequences are expected to have the following description:

$$\pi_{q,1} = \Omega^{k+2-q}\ \theta^2\ \mu_q \qquad q=1,\cdots,k \tag{4.12.a}$$
$$\beta_{0,0} = \Omega^{k+2}\ \theta^2\ \zeta \tag{4.12.b}$$
$$\sigma_{k,1} = \Omega\ \theta^2\ \lambda \tag{4.12.c}$$

where $T(\mu_q)=n-(k+1-q)$, $T(\zeta)=T(\lambda)=n$ and

$$\begin{aligned}\mu_q(t) &= u_{t,t+k+1-q}\\ \lambda(t) &= d_{t,t}\\ \zeta(y) &= z_t\end{aligned}$$

After the computation of U and z terminates, we may use the network for a second time to solve $\overline{Ux}=\overline{z}$ and obtain the vector x. Of course only few cells will be doing useful work during this second run. At this point we note that we can add to the network any number of columns of cells identical to the column $(0,j)$, $j=1,\cdots,k+1$. This enables us to use the network to solve (4.1) for more than one right hand side vector y simultaneously.

Finally, we note that the network described in this section can be modified to perform the Cholesky decomposition LL^T instead of the U^TDU decomposition. For this, the equation (4.5) for the operation of node (k+1,1) has to be replaced by

$$\sigma_{k,1} = \Omega_0\ [\frac{1}{\sqrt{\rho_{k+1,1} - \gamma_{k+1,1}}}]$$

and the data on the links $\rho_{i,2}$, $i=1,\cdots,k$ have to be set equal to the data on $\pi_{i,2}$, $i=1,\cdots,k$. This has the effect of modifying (4.7.d) to

$$\gamma_{i+1,j-1} = \Omega_0\ [\gamma_{i,j} + \pi_{i,j} * \sigma_{i,j}]$$

It is clear that in this case, the links $r_{i,j}$, $j=2,\cdots,k$, $i=1,\cdots,k+2-j$ carry redundant information and hence can be removed from the network.

After this description of the network, we turn our attention to the task of simulating its operation. First of all, we write an SCE program that describes the network and contains the equations that model its nodes. In the following program, the parameter k which determines the size of the network is set to 2.

The SCE program for the network of figure 4.1

```
PAR k=2 ;
INDEX i,j ;
SEQN s(0:k,1:k+1) ,
     r(0:k+1,1:k+2) ,
     p(1:k,1:k+2) ,
     c(1:k+1,1:k+1) ,
     b(0:0,0:k+1)    ;

INPUT( MAXT 18, For i=0,k+1 r(i,1) ) ;             /* input statement */

FOR j=1,k+1 DO c(1,j) = o END ;                     /* equation (4.10.a) */
b(0,k+1) = o ;                                      /* equation (4.10.b) */

s(k,1) = Z [ u / [r(k+1,1) - c(k+1,1)] ] ;          /* equation (4.5) */

FOR i=1,k DO
  r(i,2) = Z [ r(i,1) - c(i,1) ] ;                  /* equation (4.6.a) */
  p(i,2) = Z s(i,1) * r(i,2)       ;                /* equation (4.6.b) */
  s(i-1,1)= Z s(i,1)                                /* equation (4.6.c) */
 END ;

FOR j=2,k+1 DO
  FOR i=1,k+2-j DO
    s(i-1,j) = Z s(i,j) ;                           /* equation (4.7.a) */
    p(i,j+1) = Z p(i,j) ;                           /* equation (4.7.b) */
    r(i,j+1) = Z r(i,j) ;                           /* equation (4.7.c) */
    c(i+1,j-1) = Z [ c(i,j) + r(i,j) * s(i,j) ]     /* equation (4.7.d) */
  END ;
  s(k+2-j,j)   = p(k+2-j,j)                         /* equation (4.10.c) */
END ;

r(0,2) = Z [ r(0,1) - b(0,1) ] ;                    /* equation (4.8.a) */
b(0,0)    = r(0,2) * Z s(0,1)    ;                  /* equation (4.8.b) */

FOR j=2,k+1 DO
  r(0,j+1) = Z r(0,j) ;                             /* equation (4.9.a) */
  b(0,j-1) = Z(2) [ b(0,j) + s(0,j) * r(0,j) ]      /* equation (4.9.b) */
END ;

OUT( b(0,0) , FOR i=1,k p(i,2) , s(k,1) ) ;         /* output statement */
```

Next, we will use the above program to simulate the computation of the matrices L, U and the vector z for the matrix A given in (4.2). In order to specify the input for this computation, we note that The INPUT statement in the above program limits the length of the sequences to 18 elements. It also determines the order in which the input sequences are read from the input file, namely $\rho_{0,1}$, $\rho_{1,1}$, $\rho_{2,1}$ and then $\rho_{3,1}$. Accordingly, we follow the pattern specified by (4.11.a/b), and use the data from (4.1), to construct the following input file

Input file:

```
0.0   0.0   0.0   4.0    d    d   14.0  d    d   15.0   d   d 0.0   d   d -6.0  ...
  d     d   6.0    d     d -3.0    d     d -2.0   d     d                   ...
  d   4.0    d     d    15.0 d     d    2.0   d     d    1.0                 ...
2.0      d    d   11.0     d   d   20.0    d   d -19.0    d   d 14.0 ...
```

Finally, we use the SCE interpreter to run the above program with the given input. This produces the following output file

```
 **** OUTPUT SEQUENCES ****
 0.00 0.00 0.00 0.00 2.00    d     d   2.00    d      d   3.00    d      d -3.00
 d     d   3.00    d
 *********************
 0.00     d    d 3.00    d     d   -1.00    d    d    2.00    d    d    d    d
 d     d     d      d
 *********************
 0.00     d 2.00    d     d   1.00    d     d -5.00    d     d -3.00    d     d
 d     d     d     d
 *********************
 0.00 0.50     d   d   0.33    d     d -1.00    d      d   0.33    d      d -0.11
 d     d     d     d
 *********************
```

where as specified by the OUT statement, the sequences are printed in the order $\beta_{0,0}$, $\pi_{1,2}$, $\pi_{2,2}$ and then $\sigma_{k,1}$. It is easy to verify that this output agrees with the results in (4.3) and (4.4), and the formulas (4.12.a/b/c).

Finally, we note that the potential application of the SCE language presented in this chapter is not limited to the solver/simulator. For example, the SCE language may be used for the precise specification of any systolic network that can be described in terms of the abstract model. In fact, for a given network, one may write an SCE program in which the causal equations and the sequence declarations describe completely the graph of the network as well as the operation of each of its cells. This SCE specification may be used, for instance, as an input to an automatic lay-out program or to a translator that generates specifications in some language used in the computer aided design of VLSI devices.

The syntax directed approach used in the implementation of the solver/simulator led to a very modular program. This simplifies the task of modifying the solver to incorporate new SCE grammar rules that need to be added when new types of sequence operators are introduced. Actually, the addition of a new sequence operator to the grammar requires only the implementation of a corresponding semantics routine that describes the effect of the operator.

5. ON THE FLEXIBILITY AND POWER OF THE ABSTRACT MODEL.

After having presented the basic features and immediate applications of the abstract model, we explore in this chapter some issues that demonstrate the flexibility and the power of this model. In particular, we discuss two restrictions that were imposed on the model, namely the absence of internal states (memories) associated with the nodes of the graph, and the requirement that the graph does not contain direct loops. In Sections 5.1 and 5.3, we show that, despite these restrictions, it is nevertheless possible to apply the model to computational cells with internal memory and to networks with direct feed back connections.

The application of the model to the issue of the uniform treatment of data and control signals is discussed in Section 5.2. In Section 5.4, we suggest a technique that simplifies, to a great extent, the verification of pipelined computations in systolic networks. Finally, we show in Section 5.5 that the abstract model may be applied to self-timed systolic networks. For this, we modify the interpretation given in Section 2.3 to allow for the model *to be applicable to any systolic network, irrespective of the method used for* the synchronization of its operation.

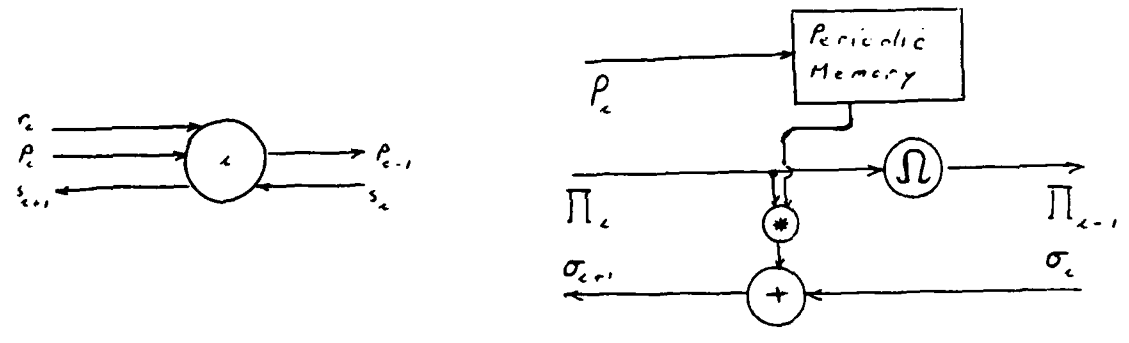
5.1. Modeling computational cells with internal memory.

The abstract model, as defined in Section 2.2, does not explicitly allow the nodes to have internal states or memory. In fact, the specification given in Section 2.6 for the 1-D convolution network was not consistent with the abstract model, since in equation (2.8), we assumed that each cell in the network can store a specified real number w_i.

However, we note that the definition of causal operators allows any output to depend on any previous input, thus eliminating the need for explicitly associating memory with interior nodes. In order to illustrate this idea, we consider the cell shown in Figure 5.1(a) and assume that the equation describing the output on the link s_{i+1} is

$$\sigma_{i+1} = \Omega\ [\sigma_i + [E_1^k\ \rho_i] * \pi_i] \tag{5.1}$$

The factor $E_1^k \rho_i$ in (5.1) indicates that the output $\sigma_{i+1}(t)$, at any time t, $sk+1 \leq t \leq (s+1)k$, for some $s \geq 0$, will depend on the data item that was existing on the input link r_i at the time $t=sk+1$. Clearly, any physical realization of this cell needs to contain a memory (see for e.g. Figure 5.1(b)).



(a) (b)

Figure 5.1 - A multiply/add cell with internal memory

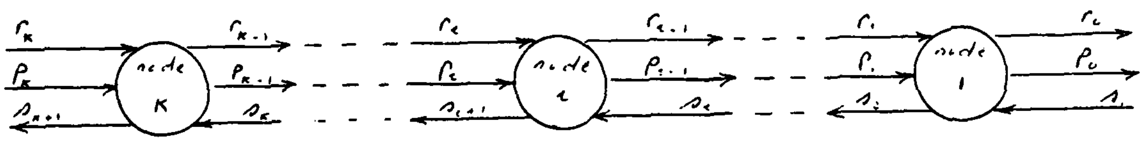


Figure 5.2 - A 1-D convolution network with writable in-cell memory

With this we can now give a consistent specification of the 1-D convolution network. We add to the network of Figure 2.7 the new links r_i, $i=0,\cdots,k$ that will be used to input the values of the convolution constants

$w_1, \cdots, w_k$ into the cells. The graph of the modified network is shown in Figure 5.2 and its operation is specified by the causal equations:

$$\pi_{i-1} = \Omega\ \pi_i \qquad i=1,\cdots,k \qquad (5.2.a)$$
$$\rho_{i-1} = \Omega\ \rho_i \qquad i=1,\cdots,k \qquad (5.2.b)$$
$$\sigma_{i+1} = \Omega\ [\sigma_i + [E^{2n}_{k+i-1}\rho_i] * \pi_i] \qquad i=1,\cdots,k \qquad (5.2.c)$$

With this specification, the network I/O description may be obtained in the form

$$\sigma_{k+1} = \Omega^k\ \sigma_1 + \sum_{i=1}^{k} \Omega^{2i-1}\ [[E^{2n}_{2k-2i+1}\ \rho_k] * \pi_k] \qquad (5.3)$$

The constants $w_1, \cdots, w_k$ are supplied to the network through the input link r_k. The idea is to let these constants flow on the r colored links and let each cell i capture its corresponding constant as it arrives at its own input link r_i. The cell then remenbers its constant for the duration of the computation, namely for $2n$ time units. Formally, the inputs to the network are described by

$$\sigma_1 = \Omega^{k-1}\ \Theta\ \iota$$
$$\pi_k = \Theta\ \xi$$
$$\rho_k = \Theta\ \omega$$

where ι is the zero sequence, $T(\xi)=n$, $T(\omega)=k$ and $\xi(t)=x_t$, $\omega(t)=w_t$. Using this input in (5.3) and applying Properties P10 and P4 in Appendix A we obtain

$$\sigma_{k+1} = \Omega^{2k-1}\ \Theta\ \iota + \Omega\Theta \sum_{i=1}^{k} \Omega^{i-1}\ [E^{n}_{k-i+1}\ \omega * \xi]$$

This leads to the same formula as in Chapter 3, namely equation (3.2).

However, we note that the convolution constants, in the modified network, are supplied as an input rather than being associated with the cells. This allows for the possible pipelining of different computations, with different convolution constants, on the same network. We will discuss pipelined operations in some detail in Section 5.4.

Finally, we note that although the k cells in the above network are identical, the operation of each cell i is determined by a local control parameter that determines the instants at which the memory within the cell is overwritten. In the next section, we discuss possible methods for controlling the operation of computational cells that depend in their operation on local parameters.

5.2. Controlling the operation of systolic cells.

As mentioned in Section 2.4, the operators $A^{r,k,s}$, $M_r^{w1,\ldots,wn}$ and E_r^k can be used to model systolic cells that contain accumulators, multiplexers or periodic memories. Here the indices r, k, s and $w1,\cdots,wn$ control different timings that may affect the operation of the cell, as for instance, the reset times, the idle times and the active times of the cell. One way of monitoring these different timings in physical cells is by providing each cell with a separate circuit that generates reset and idle signals at the specified times. This circuit may be designed either for a specific value of the control parameter, or for flexible assignments of the values of these parameters according to a desired application.

On the other hand, timings may be monitored by signals external to the cell. This external control method treats data and control signals in a uniform manner [27], and is especially preferred in systolic networks if the timing signals can be propagated systolically within the network.

As an example, we consider again the modified 1-D convolution network discussed in the last section. Equation (5.2.c) specified that the memory within any cell i, $1 \leq i \leq k$, is overwritten at times $k+i-1+2sn$, $s=0,1,\cdots$. In order to apply the external control method, we add to each cell i in the network an input link c_i such that at any time t, the memory is overwritten only if the data item on c_i equals $\gamma_i(t)=1$. If $\gamma_i(t)=0$, then the content of the memory is not changed.

In this example, the control parameter $k+i-1+2sn$, $s=0,1\cdots$, is linear with i. Hence, we may control the operation of all the cells by means of a single control signal that is propagated in the network. In Figure 5.3, we augment the network of Figure 5.2 by the c colored links and add to its specification the causal equations

$$\gamma_{i+1} = \Omega\ \gamma_i \qquad\qquad i=1,\cdots,k$$

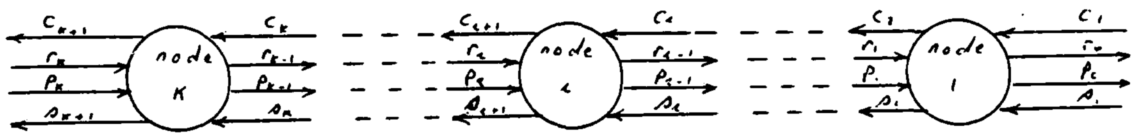


Figure 5.3 - A 1-D convolution network controlled by external signals.

If the signal transmitted on the input link c_1 is specified by

$$\gamma_1 = \Omega^k\ P_\infty^{2n}(\Theta\alpha)$$

where $\alpha(1)=1$ and $\alpha(t)=0$ for $1<t\leqslant T(\alpha)=n$, then it is easily shown that the control signal on the input link c_i of any cell i is described by

$$\gamma_i(t) = \begin{cases} \delta & \textit{if } t<k+i-1 \\ 1 & \textit{if } t=k+i-1+2sn,\ s=0,1,\cdots \\ 0 & \textit{otherwise} \end{cases}$$

This shows that the control signal will arrive at each cell at the appropriate time.

The external control approach is equivalent with a redefinition of our operators under which the control indices r,k and s are replaced by an additional control argument. For example, the expression "$E_r^k\ \xi$" used in modeling a periodic memory cell may be replaced by $E(\xi,\gamma)$, where the nonperiodic expansion operator E is defined by

$$[E(\xi,\gamma)](t) = \begin{cases} [E(\xi,\gamma)](t-1) & \text{if } \gamma(t)=0 \\ \xi(t) & \text{if } \gamma(t)=1 \end{cases}$$

and the sequence γ controls the resetting of the memory element; that is

$$\gamma(t) = \begin{cases} 1 & t=r,\ r+k,\ r+2k,\cdots \\ 0 & \text{otherwise} \end{cases}$$

Evidently, the properties of the operator E may be derived directly from those of E_r^k. Similar operators may be defined for nonperiodic accumulators and multiplexers.

5.3. Modeling networks with direct feed back loops.

The condition described by equations (2.3) in Section 2.2 does not allow for direct loops in the graph of systolic networks. In practice, however, systolic networks that have two phase clocks (for stability considerations) may contain computational cells with direct feed-back connections. In this section, we show that we may still apply our model to describe the operation of these cells. The iteration operator of Section 2.4 will be used for this purpose.

Consider, for example, the cell shown in Figure 5.4. If we were allowed to use direct feed back in the network's graphs, we could model the operation of this cell by the equations

$$\alpha_o = \Lambda_1(\beta_i,\alpha_i) \tag{5.4.a}$$
$$\beta_o = \Lambda_2(\beta_i,\alpha_i) \tag{5.4.b}$$

with the condition that $\alpha_i=\alpha_o$, and that Λ_1 and Λ_2 are given causal operator.

Note that the link a_i cannot be an external input to the cell, and that the link a_o cannot be an external output to the cell. Hence, in order to model the operation of the cell, it suffices to relate the output β_o to the input β_i. This relation is obtained by substituting $\alpha_0 = \alpha_i$ in (5.4.a) and using Theorem 3.2 to solve the resulting equation. This gives

$$\beta_o = \Lambda_2(\beta_i \ , \ I_\eta \Lambda_1(\beta_i, \eta))$$

This completely describes the I/O behavior of the cell.

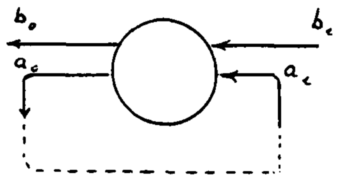


Figure 5.4 - A cell With direct feed-back

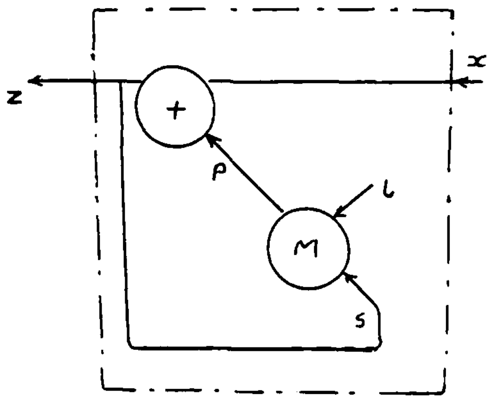


Figure 5.5 - The internal structure of a periodic accumulator.

As a specific example, we consider the cell whose internal structure is shown in Figure 5.5 and whose operation is described by

$$\pi = M_1^{1,k-1}(\iota \ , \ \sigma) \qquad (5.5.a)$$

$$\zeta = \Omega_0 \ [\xi + \pi] \qquad (5.5.b)$$

where ι is the zero sequence. If the output link z is directly connected to the input link s, then the output sequence ζ may be described as a function of the input sequence ξ only. We obtain this description by first substituting $\sigma = \zeta$ in (5.5.a) and using the result in (5.5.b).

$$\zeta = \Omega_0 \ [\xi + M_1^{1,k-1}(\iota \ , \ \zeta)] \qquad (5.6)$$

We may then apply Theorem 3.2 to express the solution of (5.6) in the form

$$\zeta = I_{\eta}\ \Omega_0\ [\xi + M_1^{1,k-1}(\iota\ ,\ \eta)] \tag{5.7}$$

This is an explicit I/O description of the cell. It is easy to show that the formula (5.7) is equivalent with

$$\zeta = \Omega_0\ A^{1,k,1}(\xi); \tag{5.8}$$

that is the cell of Figure 5.5 acts as a periodic accumulator. A similar result was proved by Johnsson and Cohen [27] using the delay operator defined in [11]. In order to prove the equivalence of (5.7) and (5.8), we consider the t^{th} element of ζ. By (5.7) we have

$$\begin{aligned}\zeta(t) &= [I_{\eta}\ \Omega_0\ [\xi + M_1^{1,k-1}(\iota\ ,\ \eta)]](t)\\ &= [\Omega_0\ [\xi + M_1^{1,k-1}(\iota\ ,\ \zeta)]](t)\end{aligned}$$

which by the definition of the multiplexer operator gives

$$\zeta(t) = \begin{cases} 0 & t=1 \\ \xi(t-1) & t=2,k+2,2k+2,\cdots \\ \xi(t-1) + \zeta(t-1) & otherwise \end{cases} \tag{5.9}$$

Equation (5.9) is a recursive formula that may be rewritten in the form

$$\zeta(t) = \begin{cases} 0 & t=1 \\ \sum_{i=0}^{na-1} \xi(t_r+i) + \zeta(t_r+i) & t>1 \end{cases} \tag{5.10}$$

where $t_r = (t-1) - mod((t-2)+k)$ and $na = t - t_r$. Finally, from the definition of the accumulator operator (see Section 2.4), it follows that the elements of tne sequence ζ in (5.8) are also give by (5.10), which proves that (5.7) and (5.8) are equivalent.

5.4. Verification of Pipelined Operation.

In this section, we apply the abstract model to the verification of pipelined operations in systolic networks. More specifically, given a systolic network that has been shown to perform a certain computation successfully, we want to study the issue of repeating the same computation on different data in a pipelined fashion. Assume that a certain systolic network NET has the I/O description

$$\eta_i = \Gamma_i(\xi_1, \cdots, \xi_n) \qquad i=1,\cdots,p \qquad (5.11)$$

where ξ_j, $j=1,\cdots,n$, η_i, $i=1,\cdots,p$, are the input and output sequences of the network, respectively, and Γ_i, $i=1,\cdots,p$ denote certain causal operators that model the behavior of the network. Suppose also that for a certain input description

$$\xi_j = \Omega^{rj}\ \alpha_j \qquad j=1,\cdots,n \qquad (5.12)$$

with given integers rj and sequences α_j, we were able to show that the outputs are described by

$$\eta_i = \Omega^{si}\ \beta_i \qquad i=1,\cdots,p \qquad (5.13)$$

with specified integers si and sequences β_i. In other words, suppose that when (5.12) is used in the equations (5.11), then we were able to prove that

$$\Omega^{si}\ \beta_i = \Gamma_i(\Omega^{r1}\ \alpha_1, \cdots, \Omega^{rn}\ \alpha_n) \qquad i=1,\cdots,p \qquad (5.14)$$

The calculation of the elements of β_i, $i=1,\cdots,p$, from those of α_j, $j=1,\cdots,n$ using the network NET shall be called the computation "C". The time of this computation is defined as the time required by NET to complete C from the moment when the first non-δ input entered NET to the moment when the last non-δ output was produced. More precisely,

$$Time(C) = \max(T(\Omega^{si}\ \beta_i);\ 1 \leq i \leq p) - \min(r_j;\ 1 \leq j \leq n) \tag{5.15}$$

where, as usual, T is the termination function defined in Section 2.1.

Often, it is desirable to repeat the computation C, say m times, with different data sets $A^e = \{\alpha_j^e; j=1,\cdots n\}$, $e=1,\cdots,m$. Let us denote these m instances of C by C^e, $e=1,\cdots,m$. In many networks, this may be accomplished by pipelining $C^1, \cdots, C^m$. The time difference between the initiations of two successive instances C^e and C^{e+1} will be defined as the pipe separation τ of the computation C. In this case, the inputs for the different instances of C should be pipelined on the input links. That is equation (5.12) for the input sequences should be replaced by

$$\xi_j^* = \Omega^{rj}\ P^{\tau}_{e=1,m}(\alpha_j^e) \qquad j=1,\cdots,n \tag{5.16}$$

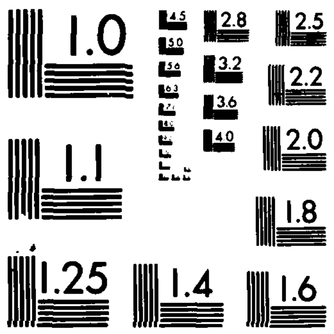
where we used the asterix in ξ_j^* to indicate that the sequences represent the input data during the pipeline-operation. We will also use ξ_j^e to represent the inputs (5.12) for a specific instance C^e of the computation. This * and e superscript notations will be employed from now on for sequences on any communication link.

If the computation can be successfully pipelined on NET with a separation τ, then by using the inputs (5.16) in the network I/O description (5.11), we should be able to prove that the output sequences during the pipeline-operation are described by

$$\eta_i^* = \Omega^{si}\ P^{\tau}_{e=1,m}(\beta_i^e) \qquad i=1,\cdots,p \tag{5.17}$$

In order to ensure a successful pipeline-operation, the pipe separation τ must be large enough so that the inputs of the different instances C^e do not overlap and the corresponding outputs do not overwrite each other. The first condition implies that $\tau \geq T(\alpha_j^e)$, $j=1,\cdots,n$, and the second that $\tau \geq T(\beta_i^e)$, $i=1,\cdots,p$. In other words, the minimum pipe separation $\tau_m(C)$

AD-A140 518   AN ABSTRACT SYSTOLIC MODEL AND ITS APPLICATION TO THE   2/3
DESIGN OFFFINITE EL..(U) PITTSBURGH UNIV PA INST FOR
COMPUTATIONAL MATHEMATICS AND APP..   R G MELHEM 1983
UNCLASSIFIED   ICMA-83-66 N00014-80-C-0455   F/G 12/1   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

for the computation C is equal to the maximum span of all the input and output sequences in C, where the span of a sequence is defined as the time difference between the first and the last significant elements (non-δ or non zero elements) in the sequence plus 1, that is the time during which the sequence carries information relevant to the computation. Hence, a *network that can be used to pipeline a computation C with a pipe separation $\tau_m(C)$ achieves maximum efficiency, from the viewpoint of the pipelined operation.*

In order to derive (5.17) from (5.16) and (5.11) without repeating the effort spent in proving (5.14), we use the negative shift operator and the equation (5.12) to rewrite the pipelined input (5.16) as

$$\xi_i^* = \Omega^{ri} P^{\tau}_{e=1,m} (\Omega^{-ri} \xi_i^e) \qquad i=1,\cdots,n \qquad (5.18)$$

Here ξ_i^e are the inputs that would be used if the instance C^e of C had been performed on NET without any pipelining. Next, we substitute (5.18) into the network I/O description (5.11) and obtain for $i=1,\cdots,p$

$$\eta_i^* = \Gamma_i([\Omega^{r1} P^{\tau}_{e=1,m} (\Omega^{-r1} \xi_1^e)],\cdots,[\Omega^{rn} P^{\tau}_{e=1,m} (\Omega^{-rn} \xi_n^e)]) \qquad (5.19)$$

The remainder of the proof is based on the use of the different properties in Appendix A for factoring the shift and the piping operators out of the causal operator Γ_i. *If the computation can be successfully pipelined through NET, then we should be able to transform* (5.19) into the form

$$\eta_i^* = \Omega^{si} P^{\tau}_{e=1,m} (\ \Omega^{-si}\ \Gamma_i(\xi_1^e,\cdots,\xi_n^e)\ ) \qquad i=1,\cdots,p \qquad (5.20)$$

which by (5.11) and (5.13) directly reduces to (5.17).

It should be noted, however, that there exist computations for which there is no τ-value for which (5.20) is derivable from (5.19). This means that such computations cannot be pipelined. On the other hand, we can identify a class of computations for which pipelining is always possible. The term "inert" shall be used to identify computations in this class. More specifically, a computation C on a systolic network NET is called inert if

1) *At its initiation, C does not care about the data on the non input* communication links of NET, that is we may assume that at time t=1, the data in any non input sequence are δ's. This implies that any delay in NET should be modeled using the shift operator and not the zero shift operator.

2) Only δ-regular operators are used for modeling the cells in NET. This implies that the network does not treat δ as a special symbol.

It is always possible to pipeline an inert computation C through the corresponding network NET. In fact we may simply chose the pipe separation τ to be the time of the computation defined by (5.15). With this value of τ, C^{e+1} does not start before C^{e} is terminated. Of course, we are not interested in such large values of τ, and hence, the problem arises of finding the least value of τ for which (5.20) is derivable from (5.19).

As should be clear from the above discussion, the ability to derive (5.20) from (5.19) is the major issue in verifying the pipeline operation of any systolic network, and this ability depends principally on the value of τ. However, for any inert computation C, we know that there exist a value for which (5.20) is derivable from (5.19). In order to find the least possible τ, we start with $\tau = \tau_m(C)$ and proceed to factor out the shift and piping *operators from* (5.19) *until we either reach* (5.20), which is our goal, or we cannot continue the factorization because of a small value of τ. In the latter case, we increase τ appropriately and repeat the derivation procedure.

EXAMPLE:

As an example, we consider once more, the modified 1-D convolution network of Section 5.1. recall that the network I/O description was given by

$$\sigma_{k+1} = \Omega^{k} \sigma_1 + \sum_{j=1}^{k} \Omega^{2j-1} [[E^{2n}_{2k-2j+1} \rho_k] * \pi_k] \quad (5.21)$$

and that, when the inputs for a certain instant of ·ıe computation C^{e} are specified by

$$\sigma^{e}_{1} = \Omega^{k-1} \Theta \iota^{e} \quad (5.22.a)$$

$$\pi^{e}_{k} = \Theta \xi^{e} \quad (5.22.b)$$

$$\rho^{e}_{k} = \Theta \omega^{e} \quad (5.22.c)$$

then the output is described by

$$\sigma^{e}_{k+1} = \Omega^{2k-1} \Theta \eta^{e} \quad (5.23)$$

The detailed forms of the sequences ι^{e}, ξ^{e} and ω^{e}, containing the input data for C^{e}, and the sequence η^{e} containing the result of C^{e}, were specified earlier.

It is easy to see that this convolution computation is inert. Hence, it is always possible to *pipeline* different instances of the computation on the same network. In this case, the minimum pipe separation during the pipelined operation is $\tau_m = \max(2n, 2k, 2n-2(k-1)) = 2n$. If m instances are pipelined through the network with the minimum separation, then the inputs should have the forms

$$\sigma^{*}_{1} = \Omega^{k-1} P^{2n}_{e=1,m} (\Theta \iota^{e}) = \Omega^{k-1} P^{2n}_{e=1,m} (\Omega^{-(k-1)} \sigma^{e}_{1}) \quad (5.24.a)$$

$$\pi^{*}_{k} = P^{2n}_{e=1,m} (\Theta \xi^{e}) = P^{2n}_{e=1,m} (\pi^{e}_{k}) \quad (5.24.b)$$

$$\rho^{*}_{k} = P^{2n}_{e=1,m} (\Theta \omega^{e}) = P^{2n}_{e=1,m} (\rho^{e}_{k}) \quad (5.24.c)$$

Using the pipelined input (5.24) in the network I/O description (5.21), we obtain the output of the pipelined operation, namely

$$\sigma^{*}_{k+1} = \Omega^{2k-1} P^{2n}_{e=1,m} (\Omega^{-(k-1)} \sigma^{e}_{1}) + \sum_{j=1}^{k} \Omega^{2j-1} [[E^{2n}_{2k-2j+1} P^{2n}_{e=1,m} (\rho^{e}_{k})] * P^{2n}_{e=1,m} (\pi^{e}_{k})]$$

Next, we use the Properties P16 and P1.1 in Appendix A to rewrite this as

$$\sigma^*_{k+1} = \Omega^{2k-1} \; P^{2n}_{e=1,m} (\Omega^{-(k-1)} \sigma^e_1) + \sum_{l=1}^{k} \Omega^{2l-1} \; P^{2n}_{e=1,m} (E^{2n}_{2k-2l+1} (\rho^e_k) * \pi^e_k) \tag{5.25}$$

Using the fact that $T(E^{2n}_{2k-2l+1}(\rho^e_k) * \pi_k) \leq 2n$, and applying Property P8.3, we may reduce (5.25) to

$$\sigma^*_{k+1} = \Omega^{2k-1} \; P^{2n}_{e=1,m} (\Omega^{-(2k-1)} \; [\Omega^k \sigma^e_1 + \sum_{l=1}^{k} \Omega^{2l-1} \; [E^{2n}_{2k-2l+1} (\rho^e_k) * \pi^e_k]])$$

Finally, from (5.21) and (5.23) we obtain

$$\begin{aligned} \sigma^*_{k+1} &= \Omega^{2k-1} \; P^{2n}_{e=1,m} (\Omega^{-(2k-1)} \; \sigma^e_{k+1}) \\ &= \Omega^{2k-1} \; P^{2n}_{e=1,m} (\Theta \; \eta^e) \end{aligned}$$

This proves that the output of the m instances will appear on the output link s_{k+1} at a rate equal to the input data rate, namely the output of one instance every $2n$ time units.

Note that the technique suggested in this section separates the verification of the pipelined operation from the verification of the correct execution of one instance of the computation. This separation leads to a clearer logic and simpler proofs.

5.5. Self timed systolic networks.

So far, we have applied the abstract model to clocked systolic networks, that is, systolic networks that are synchronized by a global clock. In this section, we show that, with a slight modification, the model may be applied to self timed systems as well [50]. In order to explain the difference between the two types of systems, we first generalize the definition of systolic networks to include any network in which computational cells have a basic cycle that is repeated indefinitely, unless the cell is forced to halt externally. After the initiation of a cycle in a computational cell, the cell reads its input from the input links, performs a specific computation and then produces its results on the output links. Here, we will assume that a cycle terminates with the initiation of the next cycle.

In clocked systolic networks, the cycle of all the cells are initiated simultaneously by an external global signal (a clock) that is broadcasted to every cell. The duration of the successive cycles is the same and is often called "one time unit". We repeatedly used this terminology in our discussion.

On the other hand, the initiation of the cycles in self timed systolic networks is not synchronized, and each cell determines locally the instant at which its cycles should start. Many protocols may be applied to organize self timed systems. Here, we consider an organization that is directly suggested by our abstract model.

Assume that the initiation of the first cycle is synchronized by a reset signal in all the cells in the self timed network, and that a new cycle in any particular cell is initiated at the instant when the cell produces the last *output of its current cycle. Assume also that each communication link in* the network is augmented by a pair of Request/Acknowledgment lines, denoted here by REQ and ACK, respectively. These lines are used to implement a 2-cycle, non-return to zero shake-hand protocol [50] between the sender S and the receiver R of the data on any communication link. REQ and ACK may carry a single bit (0 or 1) from S to R and from R to S, respectively.

The protocol is breifly explained as follows: S does not send data on the communication link unless REQ and ACK are in the same state (both 0 or both 1). After sending the data, S changes the state of REQ signaling that the link contains valid data. When R senses that REQ and ACK are in different states. it reads the data and changes the state of ACK indicating that it received the data and that it is ready to receive new data. More descriptively. after the first cycle is initiated. each cell executes the following algorithm indefinitely (unless externally forced to halt):

ALG4

1) Wait until the REQ and ACK lines associated with each input link are in opposite states.

2) Read the input data and change the state of the corresponding ACK lines.

3) Perform the computation on the input data

4) Wait until the REQ and ACK lines associated with each output link are in the same state.

5) Put the results on the output links and change the state of the corresponding REQ lines.

6) Initiate a new cycle by going to step 1.

Note that any REQ/ACK pair of lines associated with a network input link is initially set such that REQ and ACK are in the same state. This indicates that the link is ready to accept external input. On the other hand. the REQ and ACK lines associated with any other link in the network are initially set to opposite states to indicate that a certain data item is initially present on the communication link (this item may be δ).

With this general definition of a cycle in a systolic network. we generalize the physical interpretation of part [A3] of the abstract model (see Section 2.2) to allow for its application to systolic networks irrespective of the method used for the synchronization of their operation. More specifically. if x_v is an OUT edge of a node u in the graph of the network. then the sequence ξ_v associated with x_v is interpreted as follows:

1) If node u is a source node, that is, if x_v is a network input link, then $\xi_v(t)$ is the t^{th} data item that is externally transmitted on x_v from the input source.

2) If node u is an interior node, then $\xi_v(1)$ is the data item that appears on x_v at the beginning of the operation of the network, and for any $t>1$, $\xi_v(t)$ is the data item that is placed on x_v by the computational cell u as the result of its $(t-1)^{st}$ cycle.

With this general interpretation of data sequences, a systolic network has to satisfy some constraints in order to ensure that the causal equations

$$\xi^i = \Gamma_u^i(\eta_u^1, \cdots, \eta_u^m) \qquad i=1,\cdots,n \tag{5.26}$$

associated with an interior node u indeed model the computation of the corresponding cell. Here, $y_u^1,\cdots,y_u^m$ and $x^1,\cdots,x^n$, are the IN and OUT edges of u, respectively. These constraints are:

C1 : The computational cell corresponding to any interior node u will not be blocked and will continue its execution for infinitely many cycles (unless externally forced to halt).

C2 : For any i, $1\leq i\leq m$ and t, $t\geq 1$, the communication pattern in the network will ensure that $\eta_u^i(t)$ is the data read in by the cell u from the link y_u^i during its t^{th} cycle.

If constraints C1 and C2 are satisfied in a systolic network, then the model of Section 2.2 may be applied to the specification and verification of the network.

For clocked systolic networks, C1 is automatically satisfied as each cycle is initiated by a global clock that is supposed to run continuously. In order to satisfy the second constraint C2, we may assume the following: 1) No cell places the result of the t^{th} cycle of its computation on the output links before the end of the cycle, 2) the duration of each cycle is taken to be at least equal to the time required by the slowest cell in the network to complete its computation, and 3) the actual reading of the data by any cell does not start less than a certain time Δ after the initiation of a cycle, where Δ is the time-span necessary for the signals on the communication links to stabilize. These assumptions are usually implicitly made when clocked systolic networks are discussed.

For self timed systolic networks, a shake hand protocol should be used in order to ensure that the constraints C1 and C2 are satisfied. This protocol should also be obeyed in all external interactions (inputs and outputs) with the network. For example, if the protocol defined by ALG4 is used to synchronize the operation of the cells, then all inputs and outputs to the network must satisfy the following rules:

1) A data item is not transmitted on a network input link unless the associated REQ and ACK lines are in the same states. The state of REQ should be changed after the data is placed on the link.

2) For any input link x_{in}, all the elements in the infinite sequence ξ_{in}, including don't cares, are transmitted on x_{in}. However, a δ item may be transmitted by simply changing the state of ACK without placing any significant data on the data lines of x_{in}.

3) For any network output link x_o, every output item must be collected, even if there is no interest in its value. In this case, the collection of the data is simply achieved by changing the state of the ACK line associated with x_0.

It is clear that the protocol described in this section ensures that during its t^{th} cycle, any cell u will read the t^{th} elements appearing on its input links $y_u^1, \cdots, y_u^m$, provided that these elements are, at some point of time, transmitted on the corresponding links. Note that the only reasons that may prevent the t^{th} element $\eta_u^j(t)$ from ever being transmitted on y_u^j are 1) y_u^j is a network input link that is not supplied with data items, or 2) y_u^j is an OUT edge of an interior node v, and the execution of the cell v was blocked before the completion of its $(t-1)^{st}$ cycle. If we always supply the inputs when required, it is clear that the network will satisfy the constraint C2 if only the first constraint, C1, is satisfied.

In the literature of distributed processing, the constraint C1 is usually called the "liveness property" and may be formally verified with the help of the so called temporal logic [37]. This formal verification is beyond the scope of this dissertation. However, we illustrate here the steps of a formal proof by the following informal argument: Assume that any cell in the network does eventually complete its $(t-1)^{th}$ cycle for any $t>1$. This implies that for any communication link x_u in the network, $\xi_u(t)$ will eventually be transmitted on x_u. In other words, the inputs of the t^{th} cycle of any cell in the network will eventually appear on the input links of that cell, and hence, each cell will eventually read its appropriate inputs and thereby free its input links. This, together with the fact that the outputs of the network will eventually be read, leads to the conclusion that any link in the network will eventually be ready to receive its $(t+1)^{st}$ element. Hence, each cell in the network will be able to output the results of its t^{th} cycle, which means that every cell in the network will eventually complete its t^{th} cycle. Adding to this a demonstration that every cell will eventually complete its first cycle, we may show that each cell in the network will execute infinitely many cycles, thus satisfying the constraint C1.

Finally, we note that, in the organization for self timed systems discussed here, the role of the don't care elements is very crucial. More specifically, δ is interpreted as a data item rather than a 'nothing'. In other organizations of self timed systems, the operation of each cell does not start before the cell receives significant input data (non-δ items). Systems of this type may result in a dead-lock situation, thus violating the constraint C1. Clearly, the organization and verification of self timed systems is still a wide open area for research.

6. INTRODUCTION TO FINITE ELEMENT ANALYSIS.

Finite element analysis is a technique widely used by engineers [57,42] and applied mathematicians [46,51] for solving boundary value problems for partial differential equations. In the linear case, a majority of these boundary value problems can be formulated as a variational problem of the following form:

Given two Hilbert spaces $\mathscr{H}_1$ and $\mathscr{H}_2$, an appropriate bilinear operator $\mathscr{B}$ and a corresponding linear functional $\mathscr{L}$ on $\mathscr{H}_2$, find the function $\varphi \in \mathscr{H}_1$ such that

$$\mathscr{B}(v,\varphi) = \mathscr{L}(v) \qquad \text{for } all \ v \in \mathscr{H}_2 \tag{6.1}$$

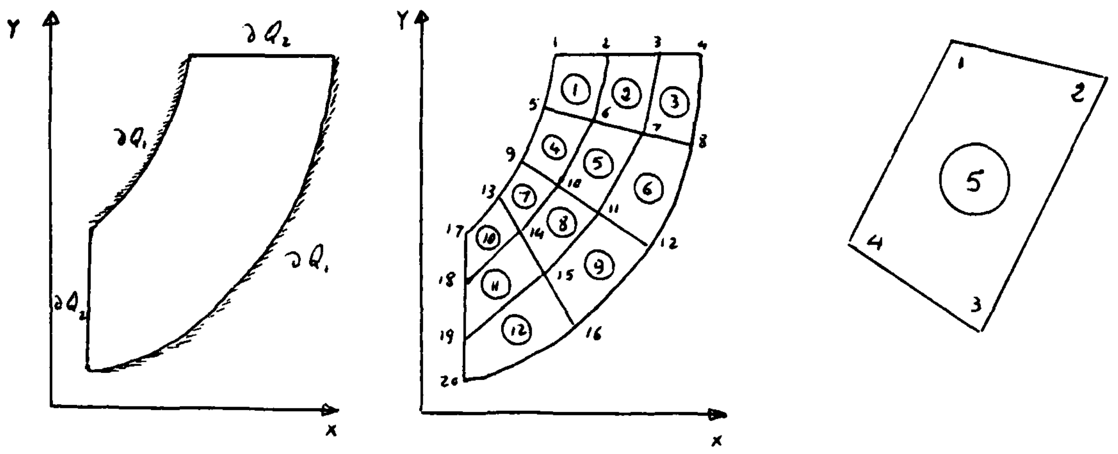
For example, consider the 2-dimensional heat conduction problem in the closed domain Q shown in Figure 6.1(a), where the curved part of the boundary ∂Q (denoted by ∂Q_1) is thermally insulated and the temperature distribution on the straight part of the boundary $\partial Q_2 = \partial Q - \partial Q_1$ is forced to be equal to a given function g(x,y). If $\varphi(x,y)$ and f(x,y) denote the temperature and the rate of heat generation at any point $(x,y) \in Q$, respectively, then the equations governing the heat flow $q = (\frac{\partial\varphi}{\partial x},\frac{\partial\varphi}{\partial y})$ are

$$\frac{\partial}{\partial x} w_x \frac{\partial\varphi}{\partial x} + \frac{\partial}{\partial y} w_y \frac{\partial\varphi}{\partial y} = -f(x,y) \qquad on \ Q \tag{6.2.a}$$

$$\frac{\partial\varphi}{\partial n} = 0 \qquad on \ \partial Q_1 \tag{6.2.b}$$

$$\varphi = g \qquad on \ \partial Q_2 \tag{6.2.c}$$

Here w_x and w_y are functions that depend on the material properties (e.g. specific heat) and $\frac{\partial\varphi}{\partial n}$ is the derivative of u with respect to the outward unit normal to the boundary. The variational problem corresponding to equation (6.2) is to find a function $\varphi(x,y) \in W^{1,2}(Q)$ such that



(a) The domain of the problem.

(b) The finite element mesh.

(c) Local node numbering for element 5.

Figure 6.1

$$\int_Q \left(w_x \frac{\partial v}{\partial x}\frac{\partial \varphi}{\partial x} + w_y \frac{\partial v}{\partial y}\frac{\partial \varphi}{\partial y}\right) dx\, dy + \int_{\partial Q_2} v\left(\varphi - \frac{\partial \varphi}{\partial n}\right) ds =$$
$$\int_Q f\ v\ dx\, dy + \int_{\partial Q_2} g\ v\ ds \qquad \text{for } all\ v \in W^{1,2}(Q)$$

where $\int_{\partial Q_2}$ is a line integral evaluated on the boundary ∂Q_2. The definition of the Sobolev space $W^{1,2}$ may be found in books on functional analysis such as [1].

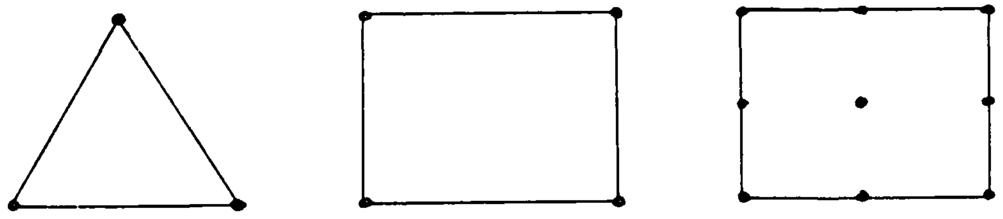
In this dissertation, we will restrict ourselves to variational problems in which Q is a bounded domain in a two dimensional space with coordinates (x,y), and the Hilbert spaces $\mathscr{H}_1$ and $\mathscr{H}_2$ are identical and from now on called $\mathscr{H}$. Moreover, we will assume that the operator $\mathscr{B}$ and the functional $\mathscr{L}$ have the general forms

$$\mathscr{B}(v,\varphi) = \int_Q \sum_{r,l=0}^{2} a_{r,l}(x,y)\; D_r v(x,y)\; D_l \varphi(x,y)\; dx\; dy + \mathscr{S}^0 \tag{6.3.a}$$

$$\mathscr{L}(v) = \int_Q f(x,y)\; v(x,y)\; dx\; dy + \mathscr{S}^1 \tag{6.3.b}$$

where f and $a_{r,l}=a_{l,r}$, $r,l=0,1,2$, are given functions, $\mathscr{S}^0$ and $\mathscr{S}^1$ are line integrals over the boundary ∂Q of Q, D_1 and D_2 are the differential operators $\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial y}$, respectively, and D_0 is the identity operator, that is $D_0\varphi=\varphi$. The form of the integrals $\mathscr{S}^0$ and $\mathscr{S}^1$ will not be specified in detail as this is not crucial for the purpose of our discussion.

The finite element process for the variational problem (6.1)/(6.3) begins with the specification of a mesh that divides Q into m finite elements Q^e, $e=1,\cdots,m$ (e.g. see Figure 6.1(b)). In addition to its geometric shape, each element is identified by a number of nodes. With each node, we associate a basis function which is a piece-wise continuous function that equals one at that node and zero at any other node.



(a) A three nodes triangular element (b) A four nodes quadrilateral element (c) A nine nodes Lagrangian element

Figure 6.2 - Some element types

In the following chapters, we will make the reasonable -- albeit somewhat restricting -- assumption that all the elements in the mesh covering Q are of the same type, and that each has k nodes (Figure 6.2 shows some element types frequently used in practice). Each node in a specific element e, $1 \leq e \leq m$, may be locally identified by a pair of indices (e,i), for some i, $1 \leq i \leq k$. Alternatively, a global scheme may be used to identify each node by a unique integer j, $1 \leq j \leq n$, where n is the total number of distinct nodes in the mesh. The relation between the local label (e,i) of a node and its global label j is defined by some mapping $glob:[1,m]\times[1,k] \rightarrow [1,n]$, where $j = glob(e,i)$. Accordingly, we may define for each element e, the boolean matrix M^e of order $k \times n$ such that for $1 \leq i \leq k$, $1 \leq j \leq n$, we have

$$M^e(i,j) = \begin{cases} 1 & \text{if } glob(e,i) = j \\ 0 & \text{otherwise.} \end{cases} \tag{6.4}$$

The matrices M^e, $e=1,\cdots,m$ and their transposes M^{eT} will play an important role in the finite element analysis.

As an example, consider the mesh in Figure 6.1(b), where each finite element is labeled by a number e, $e=1,\cdots,12$ (written inside a circle), and each node has been given a global label j, $1 \leq j \leq 20$ (written next to each node). Figure 6.1(c) isolates the finite element $e=5$ and gives each of its nodes a local number such that

$$\begin{aligned} glob(5,1) &= 6 \\ glob(5,2) &= 7 \\ glob(5,3) &= 11 \\ glob(5,4) &= 10 \end{aligned}$$

Thus

$$M^5 = \begin{bmatrix} 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0 \end{bmatrix}$$

Given a finite element mesh, we reformulate the integrals in (6.3) as the sum of integrals over the finite elements

$$\mathcal{B}(v,\varphi) = \sum_{e=1}^{m} [\int_{Q^e} \sum_{r,l=0}^{2} a_{r,l} \; D_r v \; D_l \varphi \; dx \; dy + \mathcal{S}^{0,e}] \tag{6.5.a}$$

$$\mathcal{L}(v) = \sum_{e=1}^{m} [\int_{Q^e} f \; v \; dx \; dy + \mathcal{S}^{1,e}] \tag{6.5.b}$$

here $\mathcal{S}^{0,e}$ and $\mathcal{S}^{1,e}$ are the parts of $\mathcal{S}^0$ and $\mathcal{S}^1$ evaluated on the boundary of element e if this boundary intersects with ∂Q, and otherwise, $\mathcal{S}^{0,e} = \mathcal{S}^{1,e} = 0$.

Now, the space $\mathcal{H}$ of the functions φ and v in (6.5) is replaced by a space of piece-wise spline functions over Q; that is φ and v are approximated on each finite element e by

$$\varphi(x,y) = \sum_{i=1}^{k} \varphi_{e,i} \; \psi_{e,i}(x,y) \tag{6.6.a}$$

$$v(x,y) = \sum_{i=1}^{k} v_{e,i} \; \psi_{e,i}(x,y) \tag{6.6.b}$$

where $\varphi_{e,i}$ and $v_{e,i}$ are the values of φ and v at the node (e,i), respectively, and each $\psi_{e,i}$ is the basis function associated with node (e,i). With the approximation (6.6) in (6.5), it turns out that the values $\varphi_{e,i}$ of the approximate solution $\varphi(x,y)$ at the nodes of the mesh satisfy a linear system of equations of the form

$$H \; u = b \tag{6.7}$$

where

1) u is an n-dimensional vector such that its i^{th} component u_i is the value of φ at the global node i, that is, if $i = glob(e,j)$, then $u_i = \varphi_{e,j}$.

2) H is an $n \times n$ banded, symmetric, positive definite matrix called the global stiffness matrix. With the matrices M^e of (6.4), H may be expressed as the sum

$$H = \sum_{e=1}^{m} M^{eT} \bar{H}^e M^e \tag{6.8}$$

of elemental matrices $\bar{H}^e = H^e + S^e$, $e=1,\cdots,m$. For a specific element e, the $(i,j)^{th}$ entry of the $k \times k$ matrix H^e is computed by the formula

$$H^e_{i,j} = \sum_{r,l=0}^{2} [\int_{Q^e} a_{r,l} \, D_r \Psi_{e,i} \, D_l \Psi_{e,j} \, dx \, dy] \tag{6.9}$$

Then, $\bar{H}^e$ is obtained by adding to each entry $H^e_{i,j}$ the term $S^e_{i,j} = \mathcal{S}^{0,e}_{i,j}$ resulting from the discretization of $\mathcal{S}^{0,e}$ in (6.5.a). This term, $\mathcal{S}^{0,e}_{i,j}$, is a line integral that is not equal to zero exactly when both nodes (e,i) and (e,j) lie on the boundary ∂Q. Hence, most of the elements of the matrices $S^e = (S^e_{i,j})$, $e=1,\cdots,m$ are zeroes. Moreover, if the boundary conditions associated with the problem are of the natural type [57], then the term $\mathcal{S}^0$ disappears from (6.3.a) and all the matrices S^e become zero matrices. In fact, for any finite element problem, the work associated with the computation of S^e is negligible compared with that required for the computation of H^e.

3) b is an n-dimensional vector called the global load vector which may be expressed as the sum

$$b = \sum_{e=1}^{m} M^{eT} [b^e + s^e] = \sum_{e=1}^{m} M^{eT} \bar{b}^e \tag{6.10}$$

where the components of the vector s^e are line integrals over ∂Q and the i^{th} component of the elemental vector b^e is given by

$$b^e_i = \int_{Q^e} f \, \Psi_{e,i} \, dx \, dy \tag{6.11}$$

The linear system of equations (6.7) may be solved either by a direct method or by an iterative scheme. Direct solution techniques are based on the decomposition of the positive definite symmetric matrix H into two or more matrices that have nice properties, and then transforming the system (6.7) into a number of simpler systems. For example, assuming that H is decomposed into the product of a lower and an upper triangular matrices L and U, respectively, then the solution u may be obtained by first solving the lower triangular linear system Ly=b and then using its solution y to solve the upper triangular system Uu=y. The solution of the two triangular systems is relatively simple, and hence, most of the work involved with the solution of (6.7) is in the factorization H=LU. An advantage of the direct solution is that the factorization of H does not have to be repeated if we desire to solve Hu=b' for a different right side vector $b' \neq b$, which is the case in some problems where the finite element analysis is to be performed for different load functions.

Alternatively, iterative solvers start by assuming an initial guess u^0 to the solution u^*, followed by the application of an iterative scheme for obtaining successive approximations $u^1, u^2, \cdots$ of u^*. The convergence of the iterates $u^1, u^2, \cdots$ to the solution u^* depends on both the initial guess u^0 and on the procedure used to derive u^i from u^{i-1}. In Section 9.2, we will consider iterative solvers in more details.

In summary, the linear finite element analysis involves essentially the following four computational steps: 1) Generation of the finite element mesh, 2) generation of an elemental stiffness matrix $\overline{H}^e$ and an elemental load vector $\overline{b}^e$ for each finite element e, $e=1,\cdots,m$, 3) assembly of the global stiffness matrix H and load vector b, and 4) solution of the linear system of equations H u = b.

Finally, we note that the solution vector u of (6.7) defines the function φ only at the nodes $i=1,\cdots,n$. Given u, the value of φ at any other point $(x,y)\in Q$ may be obtained from the interpolation formula (6.6.a).

Remark 1:

In the previous discussion, we assumed that φ is a real-valued function. However, the finite element analysis is also applicable if φ is a mapping into R^d, that is: a function with $d>0$ degrees of freedom. In this case, the coefficients $a_{r,l}(x,y)$ and the load $f(x,y)$ in the variational formulation (6.1/3) become $d\times d$ matrices and d-dimensional vectors, respectively. But the basic finite element technique remains the same and all the above formulas are valid with the following interpretations:

1) Each component u_i of the vector u is a d-dimensional subvector that contains the values of the d components of φ at node i.

2) Each entry $\overline{H}^e_{i,j}$ in the elemental matrix $\overline{H}^e$ is a $d\times d$ submatrix, and each entry $\overline{b}^e_i$ in the elemental load vector $\overline{b}^e$ is a d-dimensional subvector.

3) The entries of the M^e matrices of (6.4) are $d\times d$ unit matrices or zero matrices instead of ones and zeroes, respectively. Hence, the order of the linear system of equations (6.7) increases from n to nd.

Remark 2:

In some problems, it is natural to choose the function space $\mathcal{H}=\mathcal{H}_0$ such that any function $\varphi\in\mathcal{H}_0$ is equal to zero on a specified part $\partial Q_0\subseteq\partial Q$ of the boundary ∂Q. Then, the basis functions $\Psi_{e,i}$ associated with the nodes $(e,i)\in\partial Q_0$ should be excluded from the expansion (6.6). Although this decreases the dimension of the elemental arrays H^e and b^e for the elements that have common boundaries with ∂Q_0, it has the disadvantage of causing nonuniformity in the computation of the different elements. A common method for retaining the uniformity of the computation is to ignore this condition and to include all the basis functions in (6.6). Then for each node $(e,i) \in \partial Q_0$, the entries of H^e and b^e are changed such that $b^e_i = 0$, $H^e_{i,j} = 0$ for $1 \leq j \leq k$, $j \neq i$, and $H^e_{i,i} = 1$. This is equivalent with replacing the i^{th} equation in the linear system (6.7) by the equation $u_i = 0$, which guarantees that the solution is a member of the space $\mathcal{H}_0$.

7. A SYSTOLIC SYSTEM FOR THE GENERATION OF THE ELEMENTAL ARRAYS.

The purpose of the systolic system presented in this chapter is to generate the elemental arrays H^e and b^e, $e=1,\cdots,m$, for a given finite element problem and a specific mesh on its domain Q. In order to simplify the design and the description of the system, we assume, as in Chapter 6, that all elements are of the same type, and hence that the number k of nodes per element is the same for all of them.

In most practical problems, the coefficients $a_{r,l}$, r=0,1,2, in the bilinear operator (6.3.a) are constants or slowly changing functions. Hence it is very common to approximate these coefficients by piece-wise constant functions on each element, in which case we may rewrite the formula (6.9) for $H^e_{i,j}$ as

$$H^e_{i,j} = \sum_{r,l=0}^{2} a^e_{r,l} \int_{Q^e} (D_r \psi_{e,i}) (D_l \psi_{e,j})\, dx\, dy \tag{7.1.a}$$

where $a^e_{r,l}$ are constants on the element e. Similarly, the load function f(x,y) in the functional (6.3.b) may be approximated by a piece-wise constant function and hence, we may rewrite (6.11) as

$$b^e_i = f^e \int_{Q^e} \psi_{e,i}\, dx\, dy \tag{7.1.b}$$

where each f^e is a constant on the element e. This approximation, however, may not be suitable for some applications, and sometimes it is more appropriate to approximate f by a spline function in the same space as the solution function φ. In this case we use the same basis functions as in (6.6) to approximate

$$f(x,y) = \sum_{j=1}^{k} f_{e,j} \; \psi_{e,j}(x,y)$$

where $f_{e,j}$ is the value of the load f(x,y) at node (e,j). With this, (6.11) may be rewritten as

$$b_i^e = \sum_{j=1}^{k} f_{e,j} \int_{Q^e} \psi_{e,i} \; \psi_{e,j} \; dx \; dy \tag{7.1.c}$$

In order to evaluate the integrals (7.1.a/b/c), an isoparametric transformation [57] is used (see for example Figure 7.1) to map the domain of each element Q^e onto a standard element $\bar{Q}$ on some 2-dimensional space $(\bar{x},\bar{y})$, namely

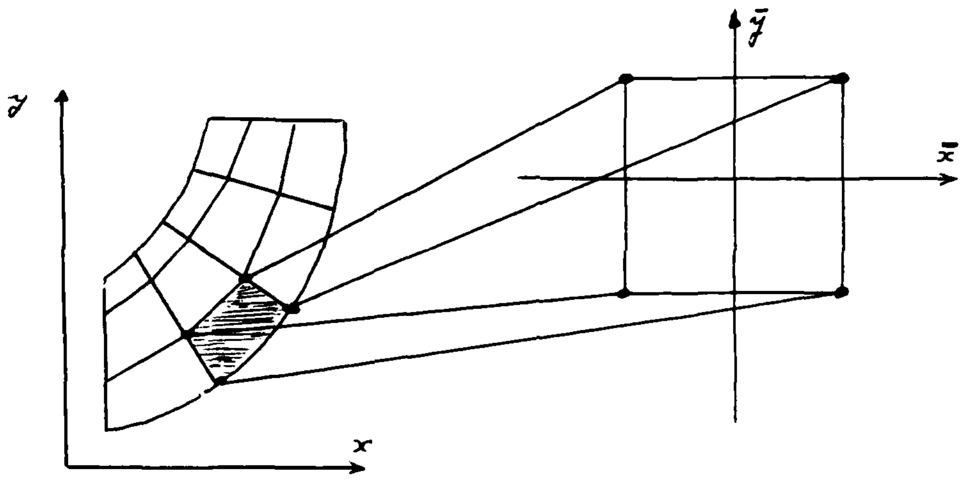


Figure 7.1 - An isoparametric transformation

$$x = \sum_{i=1}^{k} \bar{\psi}_i(\bar{x},\bar{y}) \; x_i^e \tag{7.2.a}$$

$$y = \sum_{i=1}^{k} \bar{\psi}_i(\bar{x},\bar{y}) \; y_i^e \tag{7.2.b}$$

where $\bar{\psi}_i(\bar{x},\bar{y}) = \psi_{e,i}(x(\bar{x},\bar{y}), y(\bar{x},\bar{y}))$, $i=1,\cdots,k$, are the basis functions in the new space $(\bar{x},\bar{y})$, and (x_i^e, y_i^e), $i=1,\cdots,k$, are the coordinates of the k nodes in the finite element e

The integrals are then evaluated numerically over $\bar{Q}$ instead of Q^e. Without entering into the mathematical details, we give only the final formulas used to calculate $H^e_{i,j}$ and b^e_i:

$$H^e_{i,j} = \sum_{r,l=0}^{2} a^e_{r,l} \sum_{g=1}^{q} w_g \det{}^e(\bar{x}_g,\bar{y}_g)\; D_r\bar{\Psi}_i(\bar{x}_g,\bar{y}_g)\; D_l\bar{\Psi}_j(\bar{x}_g,\bar{y}_g) \tag{7.3.a}$$

$$b^e_i = f^e \sum_{g=1}^{q} w_g \det{}^e(\bar{x}_g,\bar{y}_g)\; \bar{\Psi}_i(\bar{x}_g,\bar{y}_g) \tag{7.3.b}$$

or

$$b^e_i = \sum_{j=1}^{k} f_{e,j} \sum_{g=1}^{q} w_g \det{}^e(\bar{x}_g,\bar{y}_g)\; \bar{\Psi}_i(\bar{x}_g,\bar{y}_g)\; \bar{\Psi}_j(\bar{x}_g,\bar{y}_g) \tag{7.3.c}$$

where q is the order of the quadrature rule used in the numerical integration. $(\bar{x}_g,\bar{y}_g)$, $g=1,\cdots,q$ are the quadrature points with weights w_g and $\det^e(\bar{x},\bar{y})$ is the determinant of the Jacobian matrix J^e of the transformation $Q^e \to \bar{Q}$. From (7.2), this Jacobian is found to be

$$\begin{bmatrix} J^e_{1,1} & J^e_{1,2} \\ J^e_{2,1} & J^e_{2,2} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} \bar{D}_1\bar{\Psi}_i(\bar{x},\bar{y})\; x^e_i & \sum_{i=1}^{k} \bar{D}_2\;\bar{\Psi}_i(\bar{x},\bar{y})\; x^e_i \\ \sum_{i=1}^{k} \bar{D}_1\bar{\Psi}_i(\bar{x},\bar{y})\; y^e_i & \sum_{i=1}^{k} \bar{D}_2\;\bar{\Psi}_i(\bar{x},\bar{y})\; y^e_i \end{bmatrix}$$

Because of the regularity of the standard element $\bar{Q}$, we can easily find the formulas for $\bar{\Psi}_i(\bar{x},\bar{y})$ and its derivatives $\bar{D}_1\bar{\Psi}_i = \frac{\partial\bar{\Psi}_i}{\partial\bar{x}}$ and $\bar{D}_2\bar{\Psi}_i = \frac{\partial\bar{\Psi}_i}{\partial\bar{y}}$. Then the derivatives $D_r\bar{\Psi}_i$, $r=1,2$ and $i=1,\cdots,k$ used in (7.3.a) may be obtained from the transformation

$$\begin{bmatrix} D_1\bar{\Psi}_i \\ D_2\bar{\Psi}_i \end{bmatrix} = [J^e]^{-T} \begin{bmatrix} \bar{D}_1\bar{\Psi}_i \\ \bar{D}_2\bar{\Psi}_i \end{bmatrix} \tag{7.4}$$

where $[J^e]^{-T}$ is the inverse of the transposed Jacobian matrix $[J^e]^T$. This inverse is explicitly assumed to exist.

It should be noted that the quadrature points and weights as well as the basis functions $\bar{\Psi}_i$ and their derivatives $\bar{D}_i\bar{\Psi}_i = \frac{\partial\bar{\Psi}_i}{\partial\bar{x}}$ and $\bar{D}_2\bar{\Psi}_i = \frac{\partial\bar{\Psi}_i}{\partial\bar{y}}$ do not depend on the specific finite element that is to be processed. Hence, they may be computed at the quadrature points $(\bar{x}_g, \bar{y}_g)$, and pre-loaded into the system before it starts its operation which allows for their repeated use during the calculations of H^e and b^e for $e=1,\cdots,m$. On the other hand, the derivatives $D_1\bar{\Psi}_i$ and $D_2\bar{\Psi}_i$ in (7.2) have to be calculated by (7.4) for each element.

The following algorithm, ALG5, computes the elemental stiffness matrices H^e for $e=1,\cdots,m$. (The steps N1 through N5 in the algorithm are partitioned in a manner needed for the description of our systolic system). In this algorithm, we denote by $\nabla_i^0(g)$ the value of the basis function $\bar{\Psi}_i(\bar{x}_g, \bar{y}_g)$ and by $\Delta_i^r(g)$ and $\nabla_i^r(g)$, $r=1,2$, the value of its derivatives $\bar{D}_r \ \bar{\Psi}_i(\bar{x}_g, \bar{y}_g)$ and $D_r \ \bar{\Psi}_i(\bar{x}_g, \bar{y}_g)$, respectively.

Algorithm ALG5

INPUTS

1) ($\nabla_i^0(g)$, $\Delta_i^1(g)$, $\Delta_i^2(g)$), $g=1,\cdots,q$ and $i=1,\cdots,k$

2) For each finite element $e=1,\cdots,m$

2.1) (x_i^e, y_i^e), $i=1,\cdots,k$

2.2) $a_{r,l}^e$, $r,l=0,1,2$ /* note that $a_{r,l}^e = a_{l,r}^e$ */

For each finite element $e=1,\cdots,m$ **DO**

N1) For each quadrature point $g=1,\cdots,q$ compute the Jacobian of the isoparametric transformation from

$$\begin{bmatrix} J_{1,1}^e(g) & J_{2,1}^e(g) \\ J_{1,2}^e(g) & J_{2,2}^e(g) \end{bmatrix} = \begin{bmatrix} \Delta_1^1(g)\cdots\Delta_k^1(g) \\ \Delta_1^2(g)\cdots\Delta_k^2(g) \end{bmatrix} \begin{bmatrix} x_1^e & y_1^e \\ \cdot & \cdot \\ x_k^e & y_k^e \end{bmatrix}$$

N2) For $g=1,\cdots,q$ compute the temporary quantities

$$\begin{bmatrix} T_1^1(g)\cdots T_k^1(g) \\ T_1^2(g)\cdots T_k^2(g) \end{bmatrix} = \begin{bmatrix} J_{2,2}^e(g) & -J_{2,1}^e(g) \\ -J_{1,2}^e(g) & J_{1,1}^e(g) \end{bmatrix} \begin{bmatrix} \Delta_1^1(g)\cdots\Delta_k^1(g) \\ \Delta_1^2(g)\cdots\Delta_k^2(g) \end{bmatrix}$$

```
N3) For $g=1,\cdots,q$ DO
    N3.1) $\det^e(g) = J_{1,1}^e(g)\, J_{2,2}^e(g) - J_{1,2}^e(g)\, J_{2,1}^e(g)$
    N3.2) $\nabla_i^r(g) = (1/\det^e(g))\, T_i^r(g),\qquad r=1,2,\ i=1,\cdots,k$
    N3.3) $\overline{\nabla}_i^r(g) = w_g\, \det^e(g)\, \nabla_i^r(g),\qquad r=0,1,2,\ i=1,\cdots,k$
N4) For $i=1,\cdots,k$ compute the approximate integrals
    N4.1) For $j=1,\cdots,i-1$
          $Y_{i,j}^{r,l} = \sum_{g=1}^{q} \overline{\nabla}_i^r(g)\, \nabla_j^l(g) \qquad r,l=0,1,2$
    N4.2) $Y_{i,i}^{r,l} = \sum_{g=1}^{q} \overline{\nabla}_i^r(g)\, \nabla_i^l(g) \qquad r=0,1,2,\ l=0,\cdots,r$
N5) For $i=1,\cdots,k$ DO
    N5.1) For $j=1,\cdots,i-1$
          $H_{i,j}^e = \sum_{r=0}^{2} \sum_{l=0}^{2} a_{r,l}^e\, Y_{i,j}^{r,l}$
    N5.2) $H_{i,i}^e = 2 \sum_{r=0}^{2} \sum_{l=r}^{2} (c_{r,l}\, a_{r,l}^e)\, Y_{i,i}^{r,l}$
          where $c_{r,l}$ equals 1 if $r \neq l$, and 0.5 if $r=l$.
```

The calculation of the load vectors b^e, $e=1,\cdots,m$, may be included in ALG5 as an additional step. This step depends on whether we will use (7.3.b) or (7.3.c) for computing b^e. More precisely, if (7.3.b) is used then

```
N6) For $i=1,\cdots,k$ DO
    $b_i^e = f^e \sum_{g=1}^{q} \overline{\nabla}_i^0(g)$
```

On the other hand, if (7.3.c) is used, then

```
$\overline{\text{N6}}$) For $i=1,\cdots,k$ DO
    $b_i^e = \sum_{j=1}^{k} f_{e,j}\, Y_{i,j}^{0,0}$
```

Figure 7.2 shows a block diagram of the systolic system that executes this algorithm. It consists of a local memory LM to store the pre-loaded values of $\nabla_i^0(g)$, $\Delta_i^1(g)$ and $\Delta_i^2(g)$, $g=1,\cdots,q$, and six systolic subnetworks N1$\cdots$N6 that are arranged in a cascade such that the output of a sub-network is an input for a following sub-network. Each sub-network is designed to perform the computation in the corresponding step of ALG5.

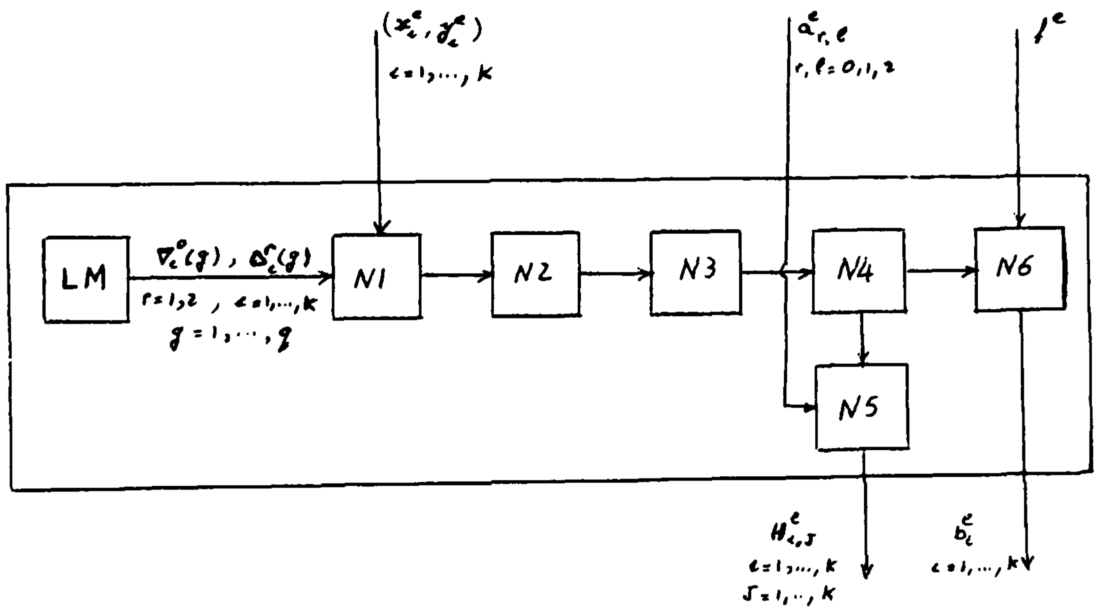


Figure 7.2 - A general block diagram of the system.

In order to compute the matrix H^e for a certain element e, the coordinates of the nodes (x_i^e, y_i^e), $i=1,\cdots,k$, and the coefficients $a_{r,l}^e$, $r,l=0,1,2$, for that element, are fed to the system via the subnetworks N1 and N5, respectively. The entries $H_{i,j}^e$, $i=1,\cdots,k$, $j=1,\cdots,i$, of the symmetric matrix H^e are then obtained from the sub-network N5 after a delay period of (q+3k+16) time units, where a time unit is the maximum time needed by any computational cell in the system to perform its operation. This is basically the time required to perform a Multiply/Add operation, or a division whichever is larger. The subnetwork N6 is used to compute the vector b^e.

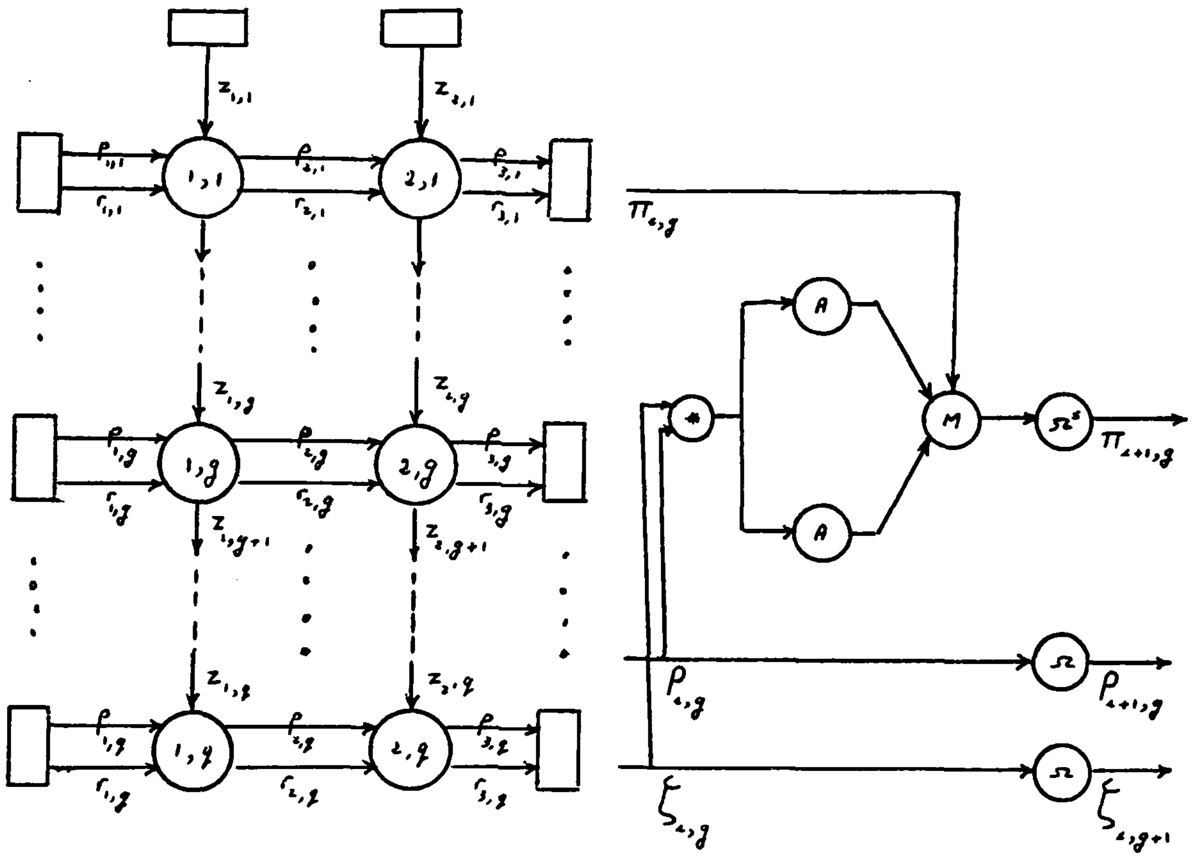
The system described in this chapter provides a noticeable speedup of order qk over the serial execution of algorithm ALG5. However, the real advantage of the system lies in the possibility of pipelining the computations of the stiffness matrices for $e=1,\cdots,m$, and of obtaining one matrix every 3k time units. Of course, we also obtain the advantage of a non-conflicting and smooth data flow in the system which greatly reduces the memory fetch times.

In Sections 7.1 through 7.6, we describe the architecture of the six subnetworks N1$\cdots$N6, that execute the corresponding steps in algorithm ALG5. Moreover, we will derive the I/O description of the individual subnetworks and prove that the system generates an elemental stiffness matrix and the corresponding load vector if appropriate input data are provided. Then in Section 7.7, we show how one can use the technique described in Section 5.4 to prove that the suggested system can be pipelined for the computation of all the elemental arrays.

It should be clear that alternate designs for the components of the system may be given. However, one advantage of the system described in this chapter is its flexibility in the sense that only minor modifications are needed in order to use the system for different values of k (element type) and q (quadrature formula). Moreover, our primary goal is to show the applicability of the systolic approach to the generation of the elemental arrays, and to demonstrate the effectiveness of the formal model for a precise specification and verification of systolic networks with computational cells more complicated than those of the simple Multiply/Add type.

7.1. The Subnetwork N1.

The graph of the systolic network N1 is composed of 2q interior nodes as shown in Figure 7.3(a); each node is labeled by two integers (i,g) i=1,2 and g=1,....,q, where q is the number of points used in the numerical integrations (7.3). The graph also shows the color assigned to each edge, namely r, p or z.



(a) The graph for N1 (b) The structure of a typical cell (i,g) in N1.

Figure 7.3

Each interior node (i,g) represents a computational cell whose operation is described by the causal relations

$$\zeta_{i,g+1} = \Omega\ \zeta_{i,g} \tag{7.5.a}$$

$$\rho_{i+1,g} = \Omega\ \rho_{i,g} \tag{7.5.b}$$

$$\pi_{i+1,g} = \Omega^{s}\ M^{3k-2,1,1}_{g+i-1}\ (\ \pi_{i,g}\ ,\ \lambda_{i,g}\ ,\ \overline{\lambda}_{i,g}\ ) \tag{7.5.c}$$

where s=1 for i=2 and s=3 for i=1, and

$$\lambda_{i,g} = A^{g+i,k,3}\ [\rho_{i,g}\ *\ \zeta_{i,g}] \tag{7.6.a}$$

$$\overline{\lambda}_{i,g} = A^{g+i+1,k,3}\ [\rho_{i,g} * \zeta_{i,g}] \tag{7.6.b}$$

The graph in Figure 7.3(a) and equations (7.5), (7.6) specify N1 completely. In order to analyze the internal structure of each cell (i,g) more closely, we first note that equations (7.6.a/b) indicate that a cell should contain a multiplier and two accumulators (see Figure 7.3(b)). The accumulators start operating at times g+i and g+i+1, respectively. They accumulate the output of the multiplier every third time unit and reset themselves to zero every 3k time units. The content of these accumulators at consecutive time units is expressed by the sequences $\lambda_{i,g}$ and $\overline{\lambda}_{i,g}$. As is clear from equation (7.5.c), each cell contains also a multiplexer that starts operating at time g+i−1 and multiplexes the input $\pi_{i,g}$ and the contents of the accumulators with time ratios 3k−2:1:1. The delay element Ω^s is introduced in Figure 7.3(b) under the assumption that the elements '*', A and M do not consume any time. In practical implementations however, these elements do consume some time and consequently the element labeled Ω^s has the function of a synchronizer rather than a latch.

After having described the architecture of the network, we prove the following proposition about the I/O description for N1. It is an explicit relation between the network output sequences $\rho_{3,g}$, $\pi_{3,g}$, $g=1,\cdots q$, and the network input sequences $\zeta_{i,1}$, $\pi_{1,g}$, $\rho_{1,g}$, $i=1,2$, $g=1,\cdots q$.

Proposition N1.1 : I/O description of the network N1. For $g=1,\cdots,q$, the following relations hold:

$$\rho_{3,g} = \Omega^2\ \rho_{1,g} \tag{7.7.a}$$

$$\pi_{3,g} = \Omega\ M_{g+3}^{3k-4,1,1,1,1}(\Omega^3\pi_{1,g}\ ,\ \lambda_{2,g}\ ,\ \overline{\lambda}_{2,g}\ ,\ \Omega^3\lambda_{1,g}\ ,\ \Omega^3\overline{\lambda}_{1,g}) \tag{7.7.b}$$

where

$$\lambda_{i,g} = A^{g+i,k,3}\ [\Omega^{i-1}\ \rho_{1,g} * \Omega^{g-1}\zeta_{i,1}] \tag{7.7.c}$$

$$\bar{\lambda}_{i,g} = A^{g+i+1,k,3}\ [\Omega^{i-1}\ \rho_{1,g}\ *\ \Omega^{g-1}\ \zeta_{i,1}] \qquad (7.7.d)$$

Proof: To prove (7.7.b), we first note that (7.5.a/b) have the solutions

$$\zeta_{i,g} = \Omega^{g-1}\ \zeta_{i,1} \qquad (7.8.a)$$
$$\rho_{i,g} = \Omega^{i-1}\ \rho_{i,g} \qquad (7.8.b)$$

Then from (7.5.c) we obtain for $g=1,\cdots,q$ that

$$\begin{aligned}\pi_{3,g} &= \Omega\ M_{g+1}^{3k-2,1,1}\ (\pi_{2,g}\ ,\ \lambda_{2,g}\ ,\ \bar{\lambda}_{2,g}) \\ &= \Omega\ M_{g+1}^{3k-2,1,1}\ (\Omega^3\ M_{g}^{3k-2,1,1}\ (\pi_{1,g},\lambda_{1,g},\bar{\lambda}_{1,g})\ ,\ \lambda_{2,g}\ ,\ \bar{\lambda}_{2,g})\end{aligned}$$

where $\lambda_{i,g}$ and $\bar{\lambda}_{i,g}$ are as given in (7.7.c) and (7.7.d), respectively. With Property P5 from Appendix A this may be rewritten as

$$\pi_{3,g} = \Omega\ M_{g+1}^{3k-2,1,1}\ (M_{g+3}^{3k-2,1,1}(\Omega^3\pi_{1,g}\ ,\ \Omega^3\lambda_{1,g}\ ,\ \Omega^3\bar{\lambda}_{1,g})\ ,\ \lambda_{2,g}\ ,\ \bar{\lambda}_{2,g})$$

Finally, we obtain (7.7.b) by applying property P13 from Appendix A. Equation (7.7.a) results directly from (7.8.b).■

In order to perform the calculations in step N1 of ALG5 for a certain finite element e, $1 \leq e \leq m$, the input sequences must be described by

$$\pi_{1,g} = \delta^* \qquad (7.9.a)$$
$$\zeta_{i,1} = \Omega^{i-1}\ E_1^3\ [\Theta^2\ \xi_i^e] \qquad i=1,2 \qquad (7.9.b)$$
$$\rho_{1,g} = \Omega^{g-1}\ P_2^{3k}(M_1^{1,1,1}(\Theta^2\psi_{g,0}\ ,\ \Omega\Theta^2\varphi_{g,1}\ ,\ \Omega^2\Theta^2\varphi_{g,2})) \qquad g=1,\cdots q \qquad (7.9.c)$$

where

$$T(\xi_i^e) = T(\psi_{g,0}) = T(\varphi_{g,1}) = T(\varphi_{g,2}) = k \qquad (7.9.d)$$

and

$$\psi_{g,0}(t) = \nabla_t^0(g)$$
$$\varphi_{g,1}(t) = \Delta_t^1(g)$$
$$\varphi_{g,2}(t) = \Delta_t^2(g)$$
$$\xi_i^e(t) = \begin{cases} y_t^e & \text{if } i=1 \\ x_t^e & \text{if } i=2 \end{cases}$$

In other words, ξ_1^e and ξ_2^e contain the coordinates of the nodes of the element e, and $\psi_{g,0}$, $\varphi_{g,1}$ and $\varphi_{g,2}$ contain the shape functions and their derivatives. A pictorial representation of these input sequences in the case k=3 and q=3 is provided in Figure 7.11 using a time diagram of the elements of the different sequences at consecutive time units.

Proposition N1.2 : With the inputs (7.9), the outputs of the network N1 are described by

$$\rho_{3,g} = \Omega^{g+1}\, P_2^{3k}(M_1^{1,1,1}(\Theta^2\psi_{g,0}\ ,\ \Omega\Theta^2\varphi_{g,1}\ ,\ \Omega^2\Theta^2\varphi_{g,2}))\quad g=1,\cdots,q\ (7.10.a)$$
$$\pi_{3,g} = \Omega^{g+3k-1}\ \beta^e \quad g=1,\cdots,q\ (7.10.b)$$

where $T(\beta^e) = 4$ and $\beta^e(t) = J_{1,1}^e(g)$, $J_{1,2}^e(g)$, $J_{2,1}^e(g)$ and $J_{2,2}^e(g)$ for t=1, 2, 3 and 4, respectively.

Proof : The proof of (7.10.a) follows directly from (7.7.a). To prove (7.10.b), we first note that the operator P_2^{3k} in (7.9.c) indicates that the first 3k elements of the argument are repeated twice in $\rho_{1,g}$. This repetition is only necessary for the operation of the subnetwork N2, and will not be considered here. Hence, we will replace the last 3k elements of the repetition by don't care elements which reduces (7.9.c) to

$$\rho_{1,g} = \Omega^{g-1}\ M_1^{1,1,1}(\Theta^2\psi_{g,0}\ ,\ \Omega\Theta^2\varphi_{g,1}\ ,\ \Omega^2\Theta^2\varphi_{g,2}) \qquad (7.9.e)$$

Now substitution of the input sequences (7.9.a/b/e) into the I/O description (7.7.b) results in

$$\pi_{3,g} = \Omega\ M_{g+3}^{3k-4,1,1,1,1}(\delta^*,\ \lambda_{2,g}\ ,\ \overline{\lambda}_{2,g}\ ,\ \Omega^3\lambda_{1,g}\ ,\ \Omega^3\overline{\lambda}_{1,g}). \qquad (7.11.a)$$

Here, by (7.7.c), the definition of the E operator, and the properties P1 and P7, we find that

$$\lambda_{i,g} = \Omega^{g+i-2}\ A^{2,k,3}\ M_1^{1,1,1}(\Theta^2[\psi_{g,0} * \xi_i^e]\ ,\ \Omega\Theta^2[\varphi_{g,1} * \xi_i^e]\ ,\ \Omega^2\Theta^2[\varphi_{g,2} * \xi_i^e])$$

and, by P14, that

$$\lambda_{i,g} = \Omega^{g+i-1} A^{1,k,3} \Theta^2 [\varphi_{g,1} * \xi_i^e] \tag{7.11.b}$$

Similarly, we can show that

$$\overline{\lambda}_{i,g} = \Omega^{g+i} A^{1,k,3} \Theta^2 [\varphi_{g,2} * \xi_i^e]. \tag{7.11.c}$$

For a further simplification of the equations (7.11.a), we consider the definition of the multiplexer operator with the restrictions (7.9.d) for the involved sequences. This gives, for $g=1,\cdots,q$,

$$\pi_{3,g} = \Omega^{g+3k-1} \beta^e$$

where $T(\beta^e) = 4$ and

$$\beta^e(t) = \begin{cases} \lambda_{2,g}(g+3k-1) & \text{for } t=1 \\ \overline{\lambda}_{2,g}(g+3k) & \text{for } t=2 \\ \lambda_{1,g}(g+3k-2) & \text{for } t=3 \\ \overline{\lambda}_{1,g}(g+3k-1) & \text{for } t=4 \end{cases}$$

Moreover, from (7.11.b), P11, and the definitions of the shift and the spread operators, we obtain that

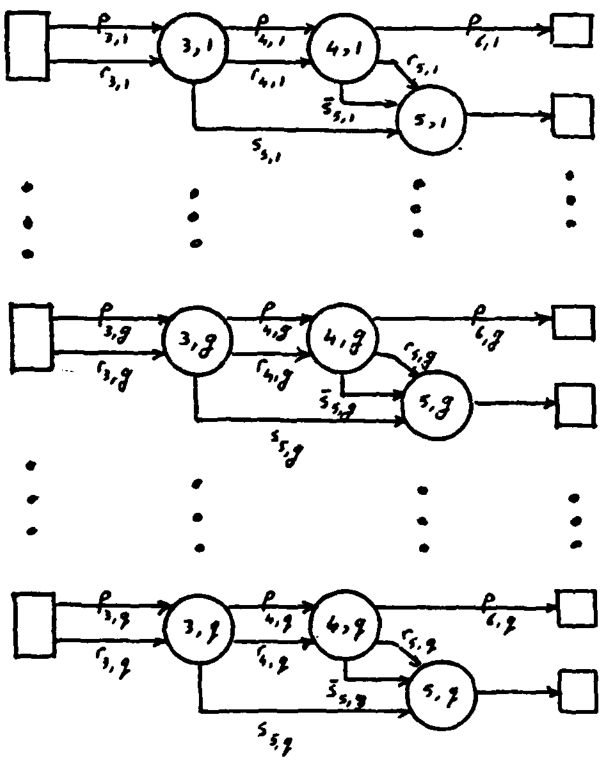
$$\begin{aligned} \lambda_{2,g}(g+3k-1) &= [\Omega^{g+1} \Theta^2 A^{1,k,1} [\varphi_{g,1} * \xi_2^e]](g+3k-1) \\ &= [A^{1,k,1}[\varphi_{g,1} * \xi_2^e]](k) \\ &= \sum_{i=1}^{k} \varphi_{g,1}(i)\, \xi_2^e(i) = \sum_{i=1}^{k} \Delta_i^1(g)\, x_i^e = J_{1,1}^e(g) \end{aligned}$$

where $J_{1,1}^e(g)$ is specified in algorithm ALG5.

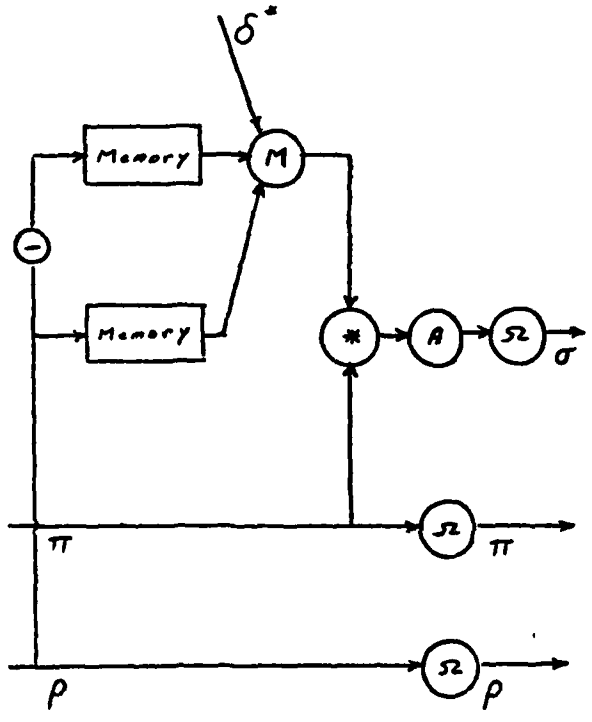
By a similar argument, it can be shown that $\beta^e(2)$, $\beta^e(3)$ and $\beta^e(4)$ are equal to $J_{1,2}^e(g)$, $J_{2,1}^e(g)$ and $J_{2,2}^e(g)$, respectively, which proves the proposition and shows that the network performs successfully the calculations in step N1 of ALG5 for one finite element e.■

7.2. The Subnetwork N2.

The graph of the subnetwork N2 is composed of q identical rows $g=1,\cdots,q$ (see Figure 7.4(a)) where each row consists of three interior nodes (i,g), $i=3,4,5$. The edges are given the colors p,r,s and $\bar{s}$ as shown in the figure.



(a) The graph for N2



(b) The structure of a cell (i,g), i=3,4 in N2.

Figure 7.4

For a given row g, $1 \leq g \leq q$, the computation of a cell may be described as follows:

For cells (3,g)

$$\pi_{4,g} = \Omega^3 \pi_{3,g} \qquad \rho_{4,g} = \Omega \rho_{3,g}$$

$$\sigma_{5,g} = \Omega A^{g,3,1} [\rho_{3,g} * M_g^{1,1,1}(E_{g+3}^{3k} \pi_{3,g} , E_{g+2}^{3k}[-\pi_{3,g}] , \delta^*)] \qquad (7.12.a)$$

For cells (4,g)

$$\pi_{6,g} = \Omega\ \pi_{4,g} \qquad \rho_{5,g} = \Omega\ \rho_{4,g}$$

$$\bar{\sigma}_{5,g} = \Omega A^{g+1,3,1}\ [\rho_{4,g} * M^{1,1,1}_{g+1} (E^{3k}_{g+4}[-\pi_{4,g}]\ .\ E^{3k}_{g+3}\ \pi_{4,g}\ ,\delta^*)] \qquad (7.12.b)$$

For cells (5,g)

$$\rho_{7,g} = \Omega\ M^{1,1,1}_{g+1}\ (\rho_{5,g}\ .\ \sigma_{5,g}\ .\ \bar{\sigma}_{5,g}) \qquad (7.12.c)$$

From the above specifications, it is clear that cells (3,g) and (4,g), $1 \leq g \leq q$, have identical structure (see Figure 7.4(b)) and differ only in the reset times of their accumulators, multiplexers and memories. To reset these elements at the proper time, external reset signal can be propagated in the network as explained in Section 5.2.

Proposition N2.1 : The network I/O description of N2 is given by

$$\pi_{6,g} = \Omega^4\ \pi_{3,g} \qquad g=1,\cdots,q \qquad (7.13.a)$$

$$\rho_{7,g} = \Omega\ M^{1,1,1}_{g+1}\ (\Omega^2\ \rho_{3,g}\ .\ \lambda_{3,g}\ .\ \lambda_{4,g}) \qquad g=1,\cdots,q \qquad (7.13.b)$$

where

$$\lambda_{3,g} = \Omega^2\ [\rho_{3,g} * E^{3k}_{g+3}\ \pi_{3,g}] - \Omega\ [\rho_{3,g} * E^{3k}_{g+2}\ \pi_{3,g}] \qquad (7.13.c)$$

$$\lambda_{4,g} = \Omega\ [\Omega\ \rho_{3,g} * E^{3k}_{g+3}\ \Omega^3\pi_{3,g}] - \Omega^2\ [\Omega\ \rho_{3,g} * E^{3k}_{g+4}\ \Omega^3\pi_{3,g}] \qquad (7.13.d)$$

Proof : Equation (7.13.a) is trivial. In order to prove (7.13.b), we begin by applying property P1.3 to equation (7.12.a):

$$\sigma_{5,g} = \Omega\ A^{g,3,1}\ M^{1,1,1}_{g} (\rho_{3,g} * E^{3k}_{g+3}\ \pi_{3,g}\ .\ \rho_{3,g} * E^{3k}_{g+2}\ [-\pi_{3,g}]\ .\ \delta^*)$$

Then, we apply property P14 and use δ^* to replace sequences whose values are irrelevant to our analysis. This gives

$$\sigma_{5,g} = \Omega\ M^{1,1,1}_{g} (\delta^*\ .\ \Omega\ [\rho_{3,g} * E^{3k}_{g+3}\ \pi_{3,g}] - [\rho_{3,g} * E^{3k}_{g+2}\ \pi_{3,g}]\ .\ \delta^*)$$

which from P5 may be written as

$$\sigma_{5,g} = M^{1,1,1}_{g+1}\ (\delta^*\ .\ \lambda_{3,g}\ .\ \delta^*) \qquad (7.14.a)$$

where $\lambda_{3,g}$ is described by (7.13.c). Similarly from (7.12.b) we obtain

$$\bar{\sigma}_{5,g} = M_{g+1}^{1,1,1}\ (\delta^* \ .\ \delta^* \ .\ \lambda_{4,g}) \qquad (7.14.b)$$

where $\lambda_{4,g}$ is described in (7.13.d). Finally, substituting (7.14.a/b) in (7.12.c), and using P13 we obtain (7.13.b), which completes the proof.■

The input links of N2 are directly connected to the outputs of N1, and hence the input sequences $\pi_{3,g}$ and $\rho_{3,g}$ $g=1,\cdots,q$ are described by the formulas (7.10).

Proposition N2.2 : If the inputs to N2 are given by (7.10), then its outputs may be described by

$$\pi_{6,g} = \Omega^{g+3k+3}\ \beta^e \qquad g=1,\cdots,q \qquad (7.15.a)$$
$$\rho_{7,g} = \Omega^{g+3k+4}\ M_1^{1,1,1}(\theta^2\psi_{g,0}\ .\ \Omega\theta^2\psi_{g,1}\ .\ \Omega^2\theta^2\psi_{g,2}) \quad g=1,\cdots,q \qquad (7.15.b)$$

where $T(\psi_{g,1})=T(\psi_{g,2})=k$ and $\psi_{g,1}(t)=T_t^1(g)$, $\psi_{g,2}(t)=T_t^2(g)$, with $T_t^1(g)$ and $T_t^2(g)$ as specified in algorithm ALG5.

Proof : The proof of (7.15.a) is trivial. In order to prove (7.15.b), we will ignore the value of the first 3k+g+1 elements in the input $\rho_{3,g}$, and hence rewrite (7.10.a) as

$$\rho_{3,g} = \Omega^{g+3k+1}\ M_1^{1,1,1}(\theta^2\psi_{g,0}\ .\ \Omega\theta^2\varphi_{g,1}\ .\ \Omega^2\theta^2\varphi_{g,2}) \qquad (7.10.c)$$

In order to find the output sequences $\rho_{7,g}$, we obtain an explicit description for $\lambda_{3,g}$ and $\lambda_{4,g}$ by substituting the input sequences into (7.13.c/d). Indeed, from (7.10.b/c) it follows that

$$\rho_{3,g} * E_{g+3}^{3k}\ \pi_{3,g} = \Omega^{g+3k+1}\ M_1^{1,1,1}(\theta^2\psi_{g,0}\ .\ \Omega\theta^2\varphi_{g,1}\ .\ \Omega^2\theta^2\varphi_{g,2})$$
$$* \ E_{g+3}^{3k}\ \Omega^{g+3k-1}\ \beta^e$$

We then interchange the shift and expand operators using P6 and apply P18 to get

$$\rho_{3,g} * E_{g+3}^{3k}\ \pi_{3,g} = \Omega^{g+3k-1}\ [\Omega^2\ M_1^{1,1,1}(\theta^2\psi_{g,0}\ ,\ \Omega\theta^2\varphi_{g,1}\ ,\ \Omega^2\theta^2\varphi_{g,2})$$
$$* \ E_4^{3k}\ \beta^{\theta}]$$
$$= \Omega^{g+3k-1}\ [\Omega^3\Omega^{-1}\ M_1^{1,1,1}(\theta^2\psi_{g,0}\ ,\ \Omega\theta^2\varphi_{g,1}\ ,\ \Omega^2\theta^2\varphi_{g,2})\ \cdot\ \beta^{\theta}(4)]$$
$$= \Omega^{g+3k+1}\ M_1^{1,1,1}(\delta^*\ ,\ \Omega\theta^2\ [J_{2,2}^{\theta}(g)\cdot\varphi_{g,1}]\ ,\ \delta^*)$$

where, as usual, the sequences irrelevant in this context were replaced by δ^*. Similarly, we obtain

$$\rho_{3,g} * E_{g+2}^{3k}\ \pi_{3,g} = \Omega^{g+3k+1}\ M_1^{1,1,1}(\delta^*\ ,\ \delta^*\ ,\ \Omega^2\theta^2\ [J_{2,1}^{\theta}(g)\cdot\varphi_{g,2}])$$

and thus from (7.13.c) that

$$\lambda_{3,g} = \Omega^{g+3k+3}\ M_1^{1,1,1}(\delta^*\ ,\ \Omega\theta^2\ [J_{2,2}^{\theta}(g)\cdot\varphi_{g,1}]\ ,\ \delta^*) -$$
$$\Omega^{g+3k+2}\ M_1^{1,1,1}(\delta^*\ ,\ \delta^*\ ,\ \Omega^2\theta^2[J_{2,1}^{\theta}(g)\cdot\varphi_{g,2}])$$
$$= \Omega^{g+3k+3}\ [M_1^{1,1,1}(\delta^*\ ,\ \Omega\theta^2\ [J_{2,2}^{\theta}(g)\cdot\varphi_{g,1}]\ ,\ \delta^*) -$$
$$M_1^{1,1,1}(\delta^*\ ,\ \Omega\theta^2\ [J_{2,1}^{\theta}(g)\cdot\varphi_{g,2}]\ ,\ \delta^*)]$$
$$= \Omega^{g+3k+3}\ M_1^{1,1,1}(\delta^*,\ \Omega\theta^2\ [J_{2,2}^{\theta}(g)\cdot\varphi_{g,1} - J_{2,1}^{\theta}(g)\cdot\varphi_{g,2}]\ ,\ \delta^*)$$

By a similar analysis it follows that

$$\lambda_{4,g} = \Omega^{g+3k+3}\ M_1^{1,1,1}(\delta^*,\ \delta^*,\ \Omega^2\theta^2[J_{1,1}^{\theta}(g)\cdot\varphi_{g,2} - J_{1,2}^{\theta}(g)\cdot\varphi_{g,1}])$$

Finally, we substitute into (7.13.b) the computed values for $\lambda_{3,g}$ and $\lambda_{4,g}$ together with the input sequence $\rho_{3,g}$ and apply properties P5 and P13 to obtain

$$\rho_{7,g} = \Omega^{g+3k+3}\ M_1^{1,1,1}(\theta^2\psi_{g,0}\ ,\ \Omega\theta^2\psi_{g,1}\ ,\ \Omega^2\theta^2\psi_{g,2}) \qquad g=1,\cdots,q$$

where

$$\psi_{g,1}(t) = J_{2,2}^{\theta}(g)\cdot\varphi_{g,1}(t) - J_{2,1}^{\theta}(g)\cdot\varphi_{g,2}(t)$$
$$= J_{2,2}^{\theta}(g)\ \Delta_t^1(g) - J_{2,1}^{\theta}(g)\ \Delta_t^2(g) = T_t^1(g)$$
$$\psi_{g,2}(t) = J_{1,1}^{\theta}(g)\ \varphi_{g,2}(t) - J_{1,2}^{\theta}(g)\ \varphi_{g,1}(t) = T_t^2(g)$$

This proves explicitly that the output sequences $\rho_{7,g}$ contain the results of step N2 in ALG5.■

7.3. The Subnetwork N3.

As in the case of N2, the subnetwork N3 is composed of q independent, identical rows, namely, one for each point used in the numerical integration (7.3). Each row performs the calculation corresponding to step N3 in ALG5 for a certain value of g, $1 \leq g \leq q$. In Figure 7.5, we show the graph for the g^{th} row of N3. The function of the cell (6,g) is to compute the determinant of the isoparametric transformation as in step N3.1 of algorithm ALG5. Its operation may be formally described by:

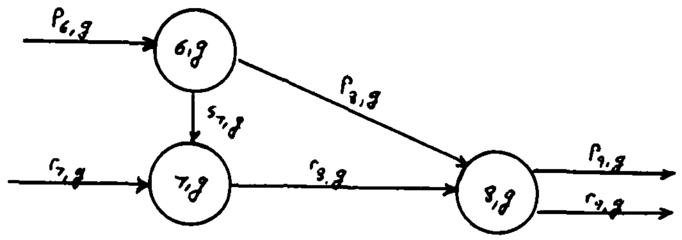


Figure 7.5 - The graph for row g of N3

$$\sigma_{7,g} = \pi_{8,g} = \Omega\ E^{3k}_{g+3k+7}\ (\Omega^3\pi_{6,g} * \pi_{6,g} - \Omega^2\pi_{6,g} * \Omega\pi_{6,g})$$

The cells (7,g) and (8,g) perform the computations in steps N3.2 and N3.3 of ALG5, respectively. Their operation may be described by

$$\begin{aligned}
\rho_{8,g} &= \Omega\ M^{1,2}_{g+3k+7}\ (\Omega^2\rho_{7,g}\ .\ [\Omega^2\rho_{7,g} * \sigma_{7,g}]) \\
\rho_{9,g} &= \Omega\ \rho_{8,g} \\
\pi_{9,g} &= \Omega\ [w_g\ .\ \pi_{8,g} * \rho_{8,g}]
\end{aligned}$$

With this description and the inputs (9.15), it is easy to obtain the output on the links $\rho_{9,g}$ and $r_{9,g}$, $g=1,\cdots,q$, namely

$$\pi_{9,g} = \Omega^{g+3k+8}\ M^{1,1,1}_{1}(\Theta^2\ \bar{\nu}_{g,0}\ .\ \Omega\Theta^2\ \bar{\nu}_{g,1}\ .\ \Omega^2\Theta^2\ \bar{\nu}_{g,2}) \qquad (7.16.a)$$
$$\rho_{9,g} = \Omega^{g+3k+8}\ M^{1,1,1}_{1}(\Theta^2\ \nu_{g,0}\ .\ \Omega\Theta^2\ \nu_{g,1}\ .\ \Omega^2\Theta^2\ \nu_{g,2}) \qquad (7.16.b)$$

where $\nu_{g,r}(t) = \nabla^r_t(g)$, $\bar{\nu}_{g,r}(t) = \bar{\nabla}^r_t(g)$, and the values of $\nabla^r_t(g)$ and $\bar{\nabla}^r_t(g)$ are as given in step N3 of ALG5.

7.4. The subnetwork N4.

In this subsection, we describe a network that completes the numerical integration by computing the quantities $Y_{i,j}^{r,l} = \sum_{g=1}^{q} \overline{\nabla}_i^r \nabla_j^l$ for the ranges of the indices in the corresponding step of ALG5. The subnetwork is described by the graph in Figure 7.6. The node I/O descriptions of a typical interior node (i,g) $i=9,\cdots,8+3k$, $g=1,\cdots,q$, are given by

$$\pi_{i+1,g} = \Omega^2 \pi_{i,g} \tag{7.17.a}$$
$$\rho_{i+1,g} = \Omega \rho_{i,g} \tag{7.17.b}$$
$$\zeta_{i,g+1} = \Omega [\zeta_{i,g} + \pi_{i,g} * \rho_{i,g}] \tag{7.17.c}$$

As this description shows, each cell latches the p and r data streams by two and one time units, respectively. It also performs a Multiply/Add operation and puts the result on the z^{th} output link.

Proposition N4.1 : The I/O description for N4 is given by

$$\zeta_{i,q+1} = \Omega^q \zeta_{i,1} + \sum_{g=1}^{q} \Omega^{i-8+q-g} [\Omega^{i-9} \pi_{9,g} * \rho_{9,g}] \qquad i=9,\cdots,8+3k \tag{7.18}$$

Proof : To prove this proposition, we first write the solutions of (7.17.a) and (7.17.b) in the form

$$\pi_{i,g} = \Omega^{2(i-9)} \pi_{9,g} \qquad i=9,\cdots,8+3k, \quad g=1,\cdots,q$$
$$\rho_{i,g} = \Omega^{(i-9)} \rho_{9,g} \qquad i=9,\cdots,8+3k, \quad g=1,\cdots,q$$

and then substitute them into (7.17.c). This gives

$$\zeta_{i,g+1} = \Omega [\zeta_{i,g} + \Omega^{2(i-9)} \pi_{9,g} * \Omega^{i-9} \rho_{9,g}] \tag{7.19}$$

By Lemma 1 in Appendix A, the solution of (7.19) for a fixed i, $9 \leq i \leq 8+3k$ is then found to be identical with equation (7.18). This completes the proof.■

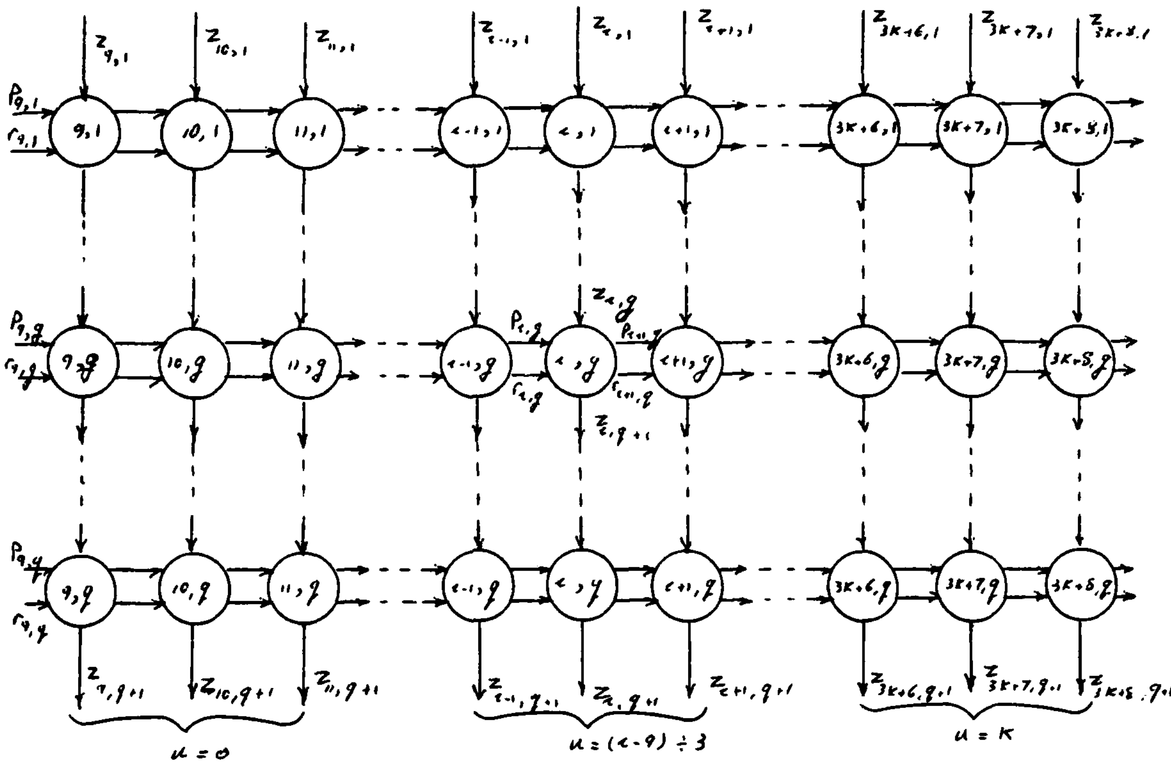


Figure 7.6 - The graph for N4.

In order to perform the computation in step N4 of ALG5, the input links $z_{i,1}$, $i=9,\cdots,8+3k$ should be permanently set to zero. That is to say, in the I/O description (7.18) we must set $\zeta_{i,1} = \iota$, where ι denotes the zero sequence of Section 2.1. With this, we rewrite (7.18) as

$$\zeta_{i,q+1} = \sum_{g=1}^{q} \Omega^{i-8+q-g} [\Omega^{i-9} \pi_{9,g} * \rho_{9,g}] \qquad i=9,\cdots,8+3k \qquad (7.20)$$

The next step in the verification of N4 is the calculation of the output sequences for a specific form of the input sequences $\pi_{9,g}$ and $\rho_{9,g}$. As Figure 7.2 shows, the outputs of N3 are inputs to N4, and hence $\pi_{9,g}$ and $\rho_{9,g}$ are described by the formulas (7.16). Unfortunately, it is not at all simple to find an explicit description of the output sequences for this specific input. In order to simplify the equations, we will replace the index i, $9 \leq i \leq 8+3k$ by $i=9+3u+v$, where the indices u and v vary in the ranges

$0 \leq u \leq k-1$ and $0 \leq v \leq 2$. More descriptively, we divide the 3k columns of N4 into k groups of 3 columns each. Thus, we rewrite the network description (7.20) as

$$\zeta_{u,v,q+1} = \sum_{g=1}^{q} \Omega^{3u+v+1+q-g} \, [\Omega^{3u+v} \, \pi_{9,g} * \rho_{9,g}] \qquad (7.21)$$

Proposition N4.2 : With the inputs described by (7.16), the network N4 has the following output:

$$\zeta_{u,v,q+1} = \Omega^{2(3u+v)+w} \, \bar{\eta}_{u,v} \qquad u=0,\cdots,k-1 \text{ and } v=0,1,2 \qquad (7.22)$$

with

$$\bar{\eta}_{u,v} = M_1^{1,1,1}(\theta^2 \eta_u^{0,v} \, , \, \Omega\theta^2 \eta_u^{1,v\square 1} \, , \, \Omega^2\theta^2 \eta_u^{2,v\square 2}),$$

where $\square$ is a modulo 3 addition, w=q+3k+9, and we have for $0 \leq l \leq 2$,

$$T(\eta_u^{r,l}) = \begin{cases} k-u & \text{if } r \leq l \\ k-u-1 & \text{if } r > l \end{cases} \qquad \text{and} \qquad \eta_u^{r,l}(t) = \begin{cases} \Upsilon_{t,t+u}^{r,l} & \text{if } r \leq l \\ \Upsilon_{t,t+u+1}^{r,l} & \text{if } r > l \end{cases}$$

Proof : Using the input sequences (7.16) in (7.21) we obtain

$$\begin{aligned} \zeta_{u,v,q+1} &= \sum_{g=1}^{q} \Omega^{3u+v+w} \, [\Omega^{3u+v} M_1^{1,1,1}(\theta^2\bar{\nu}_{g,0}, \, \Omega\theta^2\bar{\nu}_{g,1}, \, \Omega^2\theta^2\bar{\nu}_{g,2}) \\ &\qquad * \, M_1^{1,1,1}(\theta^2\nu_{g,0} \, , \, \Omega\theta^2\nu_{g,1} \, , \, \Omega^2\theta^2\nu_{g,2}) \,] \\ &= \Omega^{3u+v+w} \sum_{g=1}^{q} [\lambda_{u,v,g} * M_1^{1,1,1}(\theta^2\nu_{g,0}, \, \Omega\theta^2\nu_{g,1}, \, \Omega^2\theta^2\nu_{g,2})] \end{aligned} \qquad (7.23)$$

where again, $w=q+3k+9$ and $\lambda_{u,v,g}$ is found by properties P4 and P5 to be equal to

$$\lambda_{u,v,g} = \begin{cases} M_1^{1,1,1}(\theta^2\Omega^u \, \bar{\nu}_{g,0} \, , \, \Omega\theta^2\Omega^u \, \bar{\nu}_{g,1} \, , \, \Omega^2\theta^2\Omega^u \, \bar{\nu}_{g,2}) & \text{if } v=0 \\ M_1^{1,1,1}(\theta^2\Omega^{u+1} \, \bar{\nu}_{g,2} \, , \, \Omega\theta^2\Omega^u \, \bar{\nu}_{g,0} \, , \, \Omega^2\theta^2\Omega^u \, \bar{\nu}_{g,1}) & \text{if } v=1 \\ M_1^{1,1,1}(\theta^2\Omega^{u+1} \, \bar{\nu}_{g,1} \, , \, \Omega\theta^2\Omega^{u+1} \, \bar{\nu}_{g,2} \, , \, \Omega^2\theta^2\Omega^u \, \bar{\nu}_{g,0}) & \text{if } v=2 \end{cases}$$

The result (7.22) is then obtained by first applying P1 to perform the multiplication in (7.23), then by pulling Ω^{3u+v} out of the M operator with the help of P5 and by applying the summation to the arguments of M (property P1.2). As an illustration of the derivation procedure, we consider the case $v=1$ for which we have

$$\zeta_{u,1,q+1} = \Omega^{3u+1+w} \sum_{g=1}^{q} M_1^{1,1,1}(\Theta^2[\Omega^{u+1}\ \bar{\nu}_{g,2} * \nu_{g,0}],\ \Omega\Theta^2[\Omega^{u}\ \bar{\nu}_{g,0} * \nu_{g,1}],\ \Omega^2\Theta^2[\Omega^{u}\ \bar{\nu}_{g,1} * \nu_{g,2}])$$

$$= \Omega^{3u+1+w} M_1^{1,1,1}(\Theta^2\Omega^{u+1}\ \eta_u^{2,0},\ \Omega\Theta^2\Omega^u\ \eta_u^{0,1},\ \Omega^2\Theta^2\Omega^u\ \eta_u^{1,2})$$

where, from P1, $T(\eta_u^{2,0}) = k-u-1$, $T(\eta_u^{0,1}) = T(\eta_u^{1,2}) = k-u$ and

$$\eta_u^{2,0}(t) = \sum_{g=1}^{q} [\bar{\nu}_{g,2}(t) * \nu_{g,0}(t+u+1)] = \sum_{g=1}^{q} [\bar{\nabla}_t^2(g)\ \nabla_{t+u+1}^0(g)] = Y_{t,t+u+1}^{2,0}$$

$$\eta_u^{0,1}(t) = \sum_{g=1}^{q} [\bar{\nu}_{g,0}(t) * \nu_{g,1}(t+u)] = \sum_{g=1}^{q} [\bar{\nabla}_t^0(g)\ \nabla_{t+u}^1(g)] = Y_{t,t+u}^{0,1}$$

$$\eta_u^{1,2}(t) = \sum_{g=1}^{q} [\bar{\nu}_{g,1}(t) * \nu_{g,2}(t+u)] = \sum_{g=1}^{q} [\bar{\nabla}_t^1(g)\ \nabla_{t+u}^2(g)] = Y_{t,t+u}^{1,2}$$

Finally, we apply P5 to get

$$\zeta_{u,1,q+1} = \Omega^{2(3u+1)+w} M_1^{1,1,1}(\Theta^2\ \eta_u^{0,1},\ \Omega\Theta^2\ \eta_u^{1,2},\ \Omega^2\Theta^2\ \eta_u^{2,0})$$

which is a special case of (7.22) for v=1. The cases v=0 and v=2 are proved in an exactly similar way.■

Equation (7.22) shows that the output sequences $\zeta_{u,v,q+1}$ contain the results of the numerical integration needed for the calculation of the stiffness matrices in the next subnetwork N5. It also specifies precisely the time of each output data item. In Figure 7.7, this specification is translated into a time diagram, where we plot the elements of $\zeta_{u,v,q+1}$ versus time for the special case of k=3 and q=3.

7.5. The Subnetwork N5.

The network N5 is composed of three different rows (see Figure 7.8). Row q+1 contains 3k identical nodes. It receives the constants $a_{r,l}^e$ on the

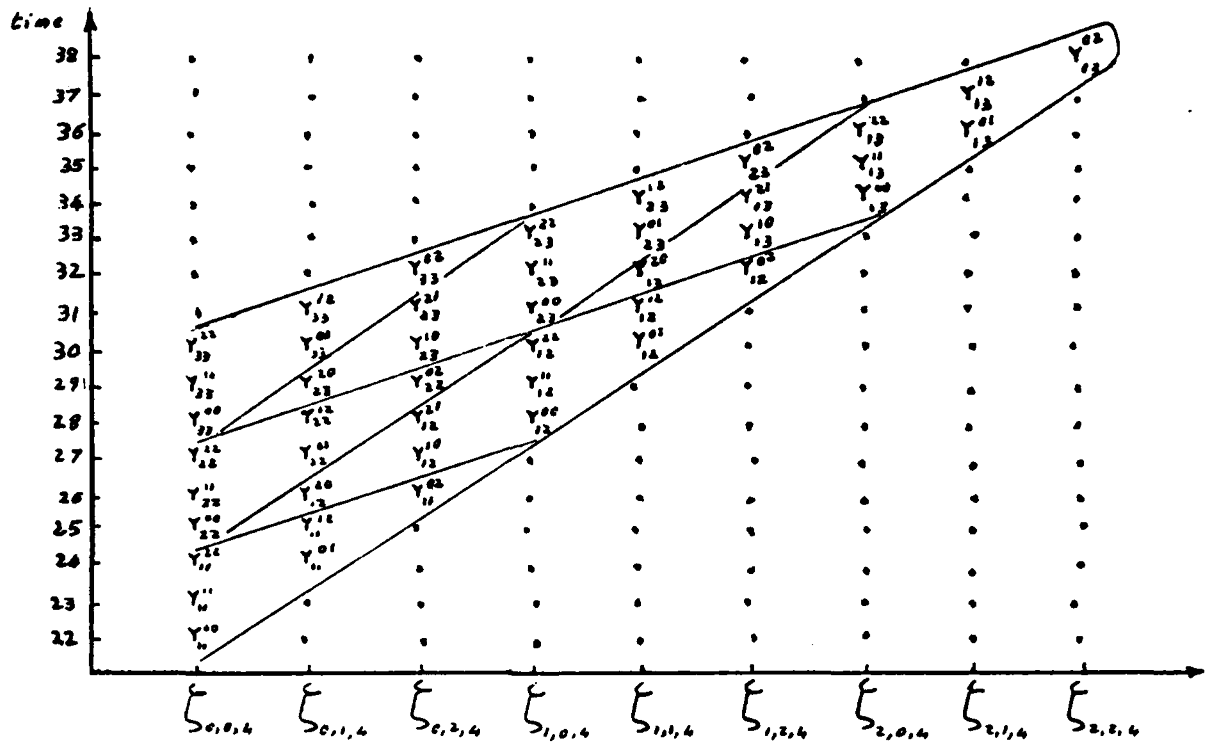


Figure 7.7 - The output of N4.

links $p_{9,q+1}$, $r_{9,q+1}$ and $s_{9,q+1}$ and distributes them appropriately on the b colored links such that each integral $Y_{i,j}^{r,l}$ appearing on a z-colored link meets the corresponding constant $a_{r,l}^{e}$ at the right time. Row q+2 also contains 3k identical nodes and computes the partial sums $U_{i,j}^{r} = \sum_{l=0}^{2} a_{r,l}^{e} Y_{i,j}^{r,l}$ and $\sum_{l=r}^{2} (c_{r,l} a_{r,l}^{e}) Y_{i,j}^{r,l}$ for $i \neq j$ and $i=j$, respectively, where $c_{r,l}$ is as given in ALG5. Finally, row q+3 contains only k nodes that complete the sum $H_{i,j}^{e} = \sum_{r=0}^{2} U_{i,j}^{r}$. The edges of the graph are given the colors p, r, s, b, z, z^0, z^1 or z^2 as shown in Figure 7.8. Note that we used three different colors z^0, z^1 and z^2 to satisfy the restriction that no two edges ending at a node have the same color. To simplify the analysis, we consider each of the three rows separately.

We consider first the row q+1 in which each cell simply latches the

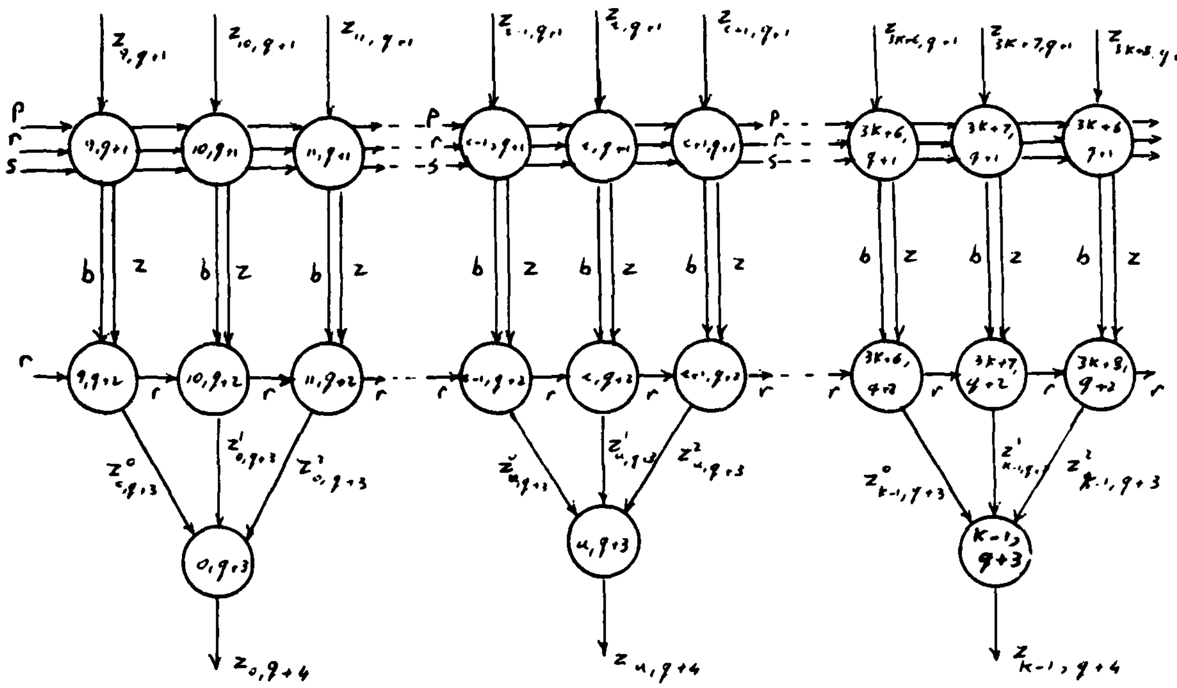


Figure 7.8 - The graph for N5.

four data streams z, p, r and s by one time unit, and selects the output on the b link to be

$$\beta_{i,q+2} = \begin{cases} \Omega\ [h_i\ .\ \pi_{i,q+1}] & \text{if } i=9+3u,\ u=0,\cdots,k-1 \\ \Omega\ \rho_{i,q+1} & \text{if } i=9+3u+1,\ u=0,\cdots,k-1 \\ \Omega\ \sigma_{i,q+1} & \text{if } i=9+3u+2,\ u=0,\cdots,k-1 \end{cases}$$

where h_i =0.5 for i=9 and h_i =1.0 for i>9. The factor 0.5 is needed to implement step N5.2 in ALG5, where only the $Y_{i,i}^{r,l}$, $l \leq r$ are explicitly available for the computation of $H_{i,i}^{e}$, while we have $Y_{i,i}^{r,l} = Y_{i,i}^{l,r}$ for $l > r$.

For the proper operation of the system the input sequences should be described by

$$\begin{aligned} \pi_{9,q+1} &= \Omega^{w}\ P_k^3(\alpha_0^e) \\ \rho_{9,q+1} &= \Omega^{w}\ P_k^3(\alpha_1^e) \\ \sigma_{9,q+1} &= \Omega^{w}\ P_k^3(\alpha_2^e) \end{aligned} \tag{7.24}$$

where for i=0,1,2, $T(\alpha_i^e)$ = 3 and $\alpha_i^e(t) = a^e_{t-1 \boxdot 2i,t-1}$, with $\boxdot$ denoting, once again, the modulo 3 addition operation. More descriptively, we input on each line three of the constants $a^{e}_{r,l}$, r,l=0,1,2, repeated k times as indicated by the piping operator P^{3}_{k} (for more details see Figure 7.11).

Using the two indices $0 \leq u \leq k-1$ and $0 \leq v \leq 2$ as in the previous subsection, and noting that the input $\zeta_{i,q+1}$ is given by (7.22), we can easily show that

$$\zeta_{u,v,q+2} = \Omega^{2(3u+v)+w+1}\ \bar{\eta}_{u,v} \tag{7.25.a}$$
$$\beta_{u,v,q+2} = \Omega^{3u+v+w+1}\ P^{3}_{k}(\ h_{u,v}\ .\ \alpha^{e}_{v}\ ) \tag{7.25.b}$$

where $h_{u,v}$ = 0.5 if u=v=0 and $h_{u,v}$ = 1.0 otherwise.

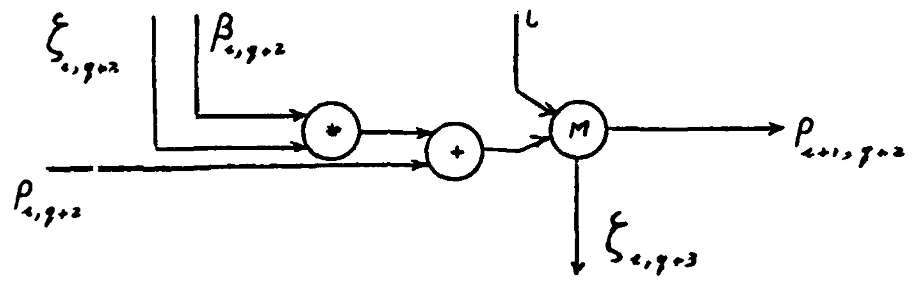


Figure 7.9 - A typical cell in row q+2 of N5

The 3k cells, (i,q+2), $9 \leq i \leq 8+3k$, in row q+2 have basically the same structure, each is a multiplier/adder equipped with a demultiplexer that distributes the results to the output links $\rho_{i+1,q+2}$ and $\zeta^{v}_{u,q+3}$ (see Figure 7.9 where u and v equal the quotient and the remainder of $\frac{i-9}{3}$, respectively). Formally, the operation of each cell (i,q+2) is described by

$$\rho_{i+1,q+2} = \Omega^{2}\ M^{1,2}_{i-9+w+1}(\iota\ .\ \rho_{i,q+2}\ +\ \lambda_{i}) \tag{7.26.a}$$
$$\zeta^{v}_{u,q+3} = \Omega\ M^{1,2}_{i-9+w+1}(\rho_{i,q+2}\ +\ \lambda_{i}\ .\ \iota) \tag{7.26.b}$$

where $\lambda_{i} = \beta_{i,q+2} * \zeta_{i,q+2}$ and the input $\rho_{9,q+2}$ is permanently set to the zero sequence ι. For a description of the outputs $\zeta^{v}_{u,q+3}$, we solve (7.26) using Lemma 3 in Appendix A. This yields

$$\zeta^{v}_{u,q+3} = \begin{cases} \Omega \; M^{1,2}_{i-9+w+1}(\lambda_i \,,\; \iota) & i=9 \\ \Omega \; M^{1,2}_{i-9+w+1}([\Omega^2\lambda_{i-1} + \lambda_i] \,,\; \iota) & i=10 \\ \Omega \; M^{1,2}_{i-9+w+1}(\,[\Omega^4\lambda_{i-2} + \Omega^2\lambda_{i-1} + \lambda_i] \,,\; \iota) & i=11,\cdots,8+3k \end{cases} \tag{7.27}$$

where, by (7.26) and (7.25), $\lambda_i = \lambda_{3u+v+9}$ is given by

$$\begin{aligned} \lambda_i &= \Omega^{3u+v+w+1} \; [P^3_k(h_{u,v} \,,\alpha_v) * \Omega^{3u+v} \; \overline{\eta}_{u,v}] \\ &= \Omega^{6u+v+w+1} \; [P^3_{k-u}(h_{u,v} \,,\alpha_v) * \Omega^{v} \; \overline{\eta}_{u,v}] \end{aligned}$$

Using the definition of $\overline{\eta}_{u,v}$ from Proposition N4.2 and with the help of property P18.2 we rewrite λ_i in the form

$$\lambda_i = \Omega^{6u+v+w+1} \; \Omega^{v} \; M^{1,1,1}_{1}(\Theta^2\mu^{0,v}_{u} \,,\; \Omega\Theta^2\mu^{1,v\square 1}_{u} \,,\; \Omega^2\Theta^2\mu^{2,v\square 2}_{u}) \tag{7.28}$$

where, for $r=0,1,2$,

$$\mu^{r,v\square r}_{u} = h_{u,v} \;\; \alpha^{\theta}_{v}((v\square r)+1) \;,\;\; \eta^{r,v\square r}_{u} = h_{u,v} \;\; a^{\theta}_{r,v\square r} \;\; \eta^{r,v\square r}_{u}$$

that is $T(\mu^{r,l}_{u}) = T(\eta^{r,l}_{u})$ and

$$\mu^{r,l}_{u}(t) = h_{u,v} \;\; a^{\theta}_{r,l} \;\; \eta^{r,l}_{u} \tag{7.29}$$

Proposition N5.1 : With the input described by (7.22) and (7.24), the intermediate sequences $\zeta^{v}_{u,q+3}$, $u=0,\cdots,k-1$, $v=0,1,2$, are given by

$$\zeta^{v}_{u,q+3} = \begin{cases} \Omega^{6u+v+w+1} \;\; M^{1,2}_{1}(\Omega^3\Theta^2[\mu^{2,2}_{u} + \mu^{2,1}_{u-1} + \mu^{2,0}_{u-1}] \,,\; \delta^{*}) & \text{for } v=0 \\ \Omega^{6u+v+w+1} \;\; M^{1,2}_{1}(\Omega^3\Theta^2[\mu^{1,2}_{u} + \mu^{1,1}_{u} + \mu^{1,0}_{u-1}] \,,\; \delta^{*}) & \text{for } v=1 \\ \Omega^{6u+v+w+1} \;\; M^{1,2}_{1}(\Omega^3\Theta^2[\mu^{0,2}_{u} + \mu^{0,1}_{u} + \mu^{0,0}_{u}] \,,\; \delta^{*}) & \text{for } v=2 \end{cases} \tag{7.30}$$

where we extended the definition of $\mu^{r,l}_{u}$ such that $\mu^{r,l}_{-1}$ equals the zero sequence.

Proof : For the case $i \geqslant 11$, we first use P5 to rewrite (7.28) in the detailed form

$$\lambda_i = \begin{cases} \Omega^{6u+w} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_u^{2,2} \;,\; \Omega\theta^2\mu_u^{0,0} \;,\; \Omega^2\theta^2\mu_u^{1,1}) & \text{for } v=0 \\ \Omega^{6u+w+1} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_u^{1,2} \;,\; \Omega^4\theta^2\mu_u^{2,0} \;,\; \Omega^2\theta^2\mu_u^{0,1}) & \text{for } v=1 \\ \Omega^{6u+w+2} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_u^{0,2} \;,\; \Omega^4\theta^2\mu_u^{1,0} \;,\; \Omega^5\theta^2\mu_u^{2,1}) & \text{for } v=2 \end{cases}$$

Then, for the evaluation of $\lambda_{i-1} = \lambda_{3u+v'+9}$, we note that $0 \leq v' \leq 2$ and hence $i-1 = 3u+v+8$ should be written in the form $3(u-1)+2+9$, $3u+0+9$ and $3u+1+9$ for $v=0,1$ and 2, respectively. With these forms for i-1 in (7.28) and the help of P5 we get

$$\Omega^2\lambda_{i-1} = \begin{cases} \Omega^{6u+w} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_{u-1}^{2,1} \;,\; \Omega\theta^2\mu_{u-1}^{0,2} \;,\; \Omega^2\theta^2\mu_{u-1}^{1,0}) & \text{for } v=0 \\ \Omega^{6u+w+1} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_u^{1,1} \;,\; \Omega^4\theta^2\mu_u^{2,2} \;,\; \Omega^2\theta^2\mu_u^{0,0}) & \text{for } v=1 \\ \Omega^{6u+w+2} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_u^{0,1} \;,\; \Omega^4\theta^2\mu_u^{1,2} \;,\; \Omega^5\theta^2\mu_u^{2,0}) & \text{for } v=2 \end{cases}$$

Similarly, we write $i-2$ as $3(u-1)+1+9$, $3(u-1)+2+9$ and $3u+0+9$ for $v=0,1$ and 2, respectively and get

$$\Omega^4\lambda_{i-2} = \begin{cases} \Omega^{6u+w} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_{u-1}^{2,0} \;,\; \Omega\theta^2\mu_{u-1}^{0,1} \;,\; \Omega^2\theta^2\mu_{u-1}^{1,2}) & \text{for } v=0 \\ \Omega^{6u+w+1} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_{u-1}^{1,0} \;,\; \Omega^4\theta^2\mu_{u-1}^{2,1} \;,\; \Omega^2\theta^2\mu_{u-1}^{0,2}) & \text{for } v=1 \\ \Omega^{6u+w+2} \; M_1^{1,1,1}(\Omega^3\theta^2\mu_u^{0,0} \;,\; \Omega^4\theta^2\mu_u^{1,1} \;,\; \Omega^5\theta^2\mu_u^{2,2}) & \text{for } v=2 \end{cases}$$

Then by adding these three formulas to get $\Omega^4\lambda_{i-2} + \Omega^2\lambda_{i-1} + \lambda_i$, and by substituting the result in (7.27), we directly obtain the equation (7.30) for $1 \leq u \leq k-1$.

The case $u=0$, that is $i=9,10$ and 11, can be analyzed in an exactly similar manner yielding the result

$$\zeta_{0,q+3}^{v} = \begin{cases} \Omega^{6u+w+v+1} \; M_1^{1,2}(\Omega^3\theta^2 \; [\mu_u^{2,2}] \;,\; \delta^*) & \text{for } v=0 \\ \Omega^{6u+w+v+1} \; M_1^{1,2}(\Omega^3\theta^2[\mu_u^{1,2} + \mu_u^{1,1}] \;,\; \delta^*) & \text{for } v=1 \\ \Omega^{6u+w+v+1} \; M_1^{1,2}(\Omega^3\theta^2 \; [\mu_u^{0,2} + \mu_u^{0,1} + \mu_u^{0,0}] \;,\; \delta^*) & \text{for } v=2 \end{cases}$$

which by defining $\mu_{-1}^{r,l} = \iota$ may also be put into the form (7.30).■

Finally, each group of three sequences $\zeta_{u,q+3}^{0}$, $\zeta_{u,q+3}^{1}$ and $\zeta_{u,q+3}^{2}$ is considered as input to a cell $(u,q+3)$, $0 \le u \le k-1$, in row q+3 of N5. The operation of a typical cell in row q+3 is formally expressed by

$$\zeta_{u,q+4} = \Omega[c_u \cdot A^{6u+w+5,3,1} \; M^{1,1,1}_{6u+w+5}(\zeta^0_{u,q+3}, \zeta^1_{u,q+3}, \zeta^2_{u,q+3})] \quad (7.31)$$

$$= \Omega[c_u \cdot M^{1,1,1}_{6u+w+5}(\delta^*, \delta^*, [\Omega^2\zeta^0_{u,q+3} + \Omega\zeta^1_{u,q+3} + \zeta^2_{u,q+3}])]$$

where c_u equals to 2 for u=0 and to 1, otherwise.

By substituting the sequences (7.30) into the network description (7.31) we easily find the description of the output sequences as

$$\zeta_{u,q+4} = \Omega^{6u+w+7} \; \Theta^2 \; \overline{\mu}^e_u \qquad u=0,\cdots,k-1 \qquad (7.32)$$

where

$$\overline{\mu}^e_u = \begin{cases} \sum_{r=0}^{2} \sum_{l=r}^{2} \mu^{r,l}_u + \sum_{r=1}^{2} \sum_{l=0}^{r-1} \mu^{r,l}_{u-1} & \text{if } u>0 \\ \sum_{r=0}^{2} \sum_{l=r}^{2} \mu^{r,l}_u & \text{if } u=0 \end{cases}$$

Using the definition of $\mu^{r,l}_u$ from (7.28/22) in (7.32) and comparing the result with step N5 in algorithm ALG5, we readily prove the following proposition:

Proposition N5.2 : If the inputs to the network N5 are given by (7.22) and (7.24), then the network's output sequences are given by

$$\zeta_{u,q+4} = \Omega^{6u+w+7} \; \Theta^2 \; \overline{\mu}^e_u \qquad u=0,\cdots,k-1 \qquad (7.32)$$

where $T(\overline{\mu}^e_u) = k-u$ and $\overline{\mu}^e_u(t) = H^e_{t,t+u}$.

Proposition N5.2 states that after an initial time period of 6u+3k+q+16 units, each output link $\zeta_{u,q+4}$ will carry the elements of the u^{th} off-diagonal of the elemental stiffness matrix H^e, separated from each other by 2 time units (see Figure 7.11).

7.6. The Subnetwork N6.

The purpose of N6 is to generate, for each finite element e, the entries b_i^e, $i=1,\cdots,k$ in the elemental load vector b^e. The design of the subnetwork depends on whether we apply step N6 or step $\overline{N6}$ of ALG5 to generate b_i^e. In the following, we consider each case separately.

7.6.1. Realization of step N6 in ALG5.

In order to compute b_i^e by step N6, the values of the load f^e at the current element e should be supplied from outside the system. On the other hand, the quantities $\overline{\nabla}_i^0(g)$, $g=1,\cdots,q$, $i=1,\cdots,k$ are readily available on the output links $p_{3k+9,g}$, $g=1,\cdots,q$ of N4. More precisely, replacing non relevant information in (7.16.a) by don't cares, we may write the sequences on the links $p_{3k+9,g}$, $g=1,\cdots,q$ as

$$\pi_{3k+9,g} = \Omega^{6k} \pi_{9,g} = \Omega^{g+9k+8} \Theta^2 \bar{\nu}_{g,0} \qquad g=1,\cdots,q \qquad (7.33)$$

where $T(\bar{\nu}_{g,0}) = k$ and $\bar{\nu}_{g,0}(t) = \overline{\nabla}_t^0(g)$.

Figure 7.10 shows a realization of N6 by a systolic network that is to be connected in cascade with N4. It is composed of q+1 computational cells whose operations are described by

$$\zeta_{3k+9,g+1} = \Omega M_g^{1,2}([\zeta_{3k+9,g} + \pi_{3k+9,g}] , \delta^*) \qquad g=1,\cdots,q \qquad (7.34.a)$$

$$\zeta_{3k+9,q+2} = \Omega M_{q+1}^{1,2}([\zeta_{3k+9,q+1} * \bar{\pi}_{3k+9,q+1}] , \delta^*] \qquad (7.34.b)$$

where the colors of the links are as shown in Figure 7.10. Note that the cells in N6 are either simple adders or simple multipliers that operate once every three time units.

The input links $p_{3k+9,g}$, $g=1,\cdots,q$ are connected to the corresponding output links of N4. Moreover, we set the link $z_{3k+9,1}$ permanently to zero

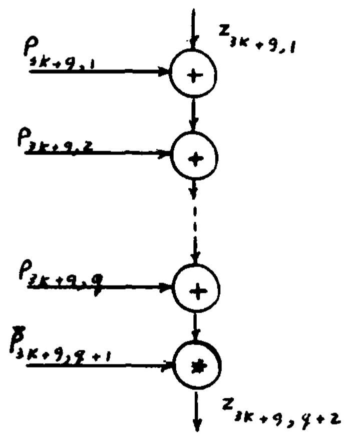


Figure 7.10 - The graph for N6.

and supply the load f^{θ} through $\bar{\rho}_{3k+9,q+1}$, that is

$$\begin{aligned} \zeta_{3k+9,1} &= \iota \\ \bar{\pi}_{3k+9,q+1} &= \Omega^{q+9k+9}\, \Theta^{2}\, \bar{\varphi}^{\theta} \end{aligned} \tag{7.35}$$

where $T(\bar{\varphi}^{\theta}) = k$ and $\bar{\varphi}^{\theta}(t) = f^{\theta}$, for any $t \leqslant k$. With these inputs, it is straight forward to prove that the output on the link $z_{3k+9,q+2}$ is described by

$$\zeta_{3k+9,q+2} = \Omega^{q+9k+10}\, \Theta^{2}\, \bar{\mu}_k^{\theta} \tag{7.36.a}$$

where $T(\bar{\mu}_k^{\theta}) = k$ and $\bar{\mu}_k^{\theta}(t) = b_t^{\theta}$. That is N6 does indeed generate the elements of the load vector b^{θ}. From equations (7.32) and (7.36.a), it is clear that the elements of b^{θ} and H^{θ} are generated from N5 and N6, respectively, at the same rate. The initial delay on the individual output links may be changed, if we wish, by adding the appropriate number of latch cells. For example, by adding a cell that delays the sequence in (7.36.a) by six time units and produces the output on a link labeled $z_{k,q+4}$, we obtain an output sequence that has the same form as the sequences in (7.32), namely

$$\zeta_{k,q+4} = \Omega^{6k+w+7}\, \Theta^{2}\, \bar{\mu}_k^{\theta} \tag{7.36.b}$$

7.6.2. Realization of step $\overline{N6}$ in ALG5.

One can look at step $\overline{N6}$ of ALG5 as a matrix-vector multiplication $b^e = Y\ F$ where the entries of the vector $F = (\ f_{e,i},\ i=1,\cdots k\ )$ are the values of the load function at the nodes of e, and $Y = (Y^{0,0}_{i,j},\ i,j=1,\cdots,k)$ is a symmetric matrix. Fortunately, the values of $Y^{0,0}_{i,j}$ are available on the output links $z_{u,0,q+1},\ u=0,\cdots,k-1$ of N4. More precisely, from (7.22) we have

$$\zeta_{u,0,q+1} = \Omega^{6u+w}\ \Theta^2\ \eta^{0,0}_u \qquad\qquad u=0,\cdots,k-1$$

where $T(\eta^{0,0}_u) = k-u$ and $\eta^{0,0}_u(t) = Y^{0,0}_{t,t+u}$.

This form of the matrix Y enables us to use the matrix-vector multiplication array of Section 3.1 to compute the components of the product vector $b^e = (b^e_i,\ i=1,\cdots,k)$ at a rate compatible with that of the generation of the elements of H^e.

To summarize the behavior of the integrated system presented in this chapter, we show in Figure 7.11 a time diagram of the data on all the input and output links of the global system of Figure 7.2. It represents a translation of the sequence equations (7.9), (7.24), (7.32), (7.35) and (7.36.b) for the special case k=q=3. The data items in the input sequences ζ_1, ζ_2, $\pi_{9,q+1}$, $\rho_{9,q+1}$, $\sigma_{9,q+1}$ and $\overline{\pi}_{3k+9,q+1}$ depend on the finite element that is being processed and hence must be provided from outside the system. On the other hand, the data in $\rho_{1,g}$, $g=1,\cdots,q$ do not depend on a particular finite element and thus are provided from a memory local to the system. Note that we assumed that the network of Section 6.6.1 is used for the generation of the elemental load vectors.

In general, the time for completing the computations for one finite element is 9k+q+10 time units. In the next section, we will show that the computation for different finite elements can be pipelined through the system

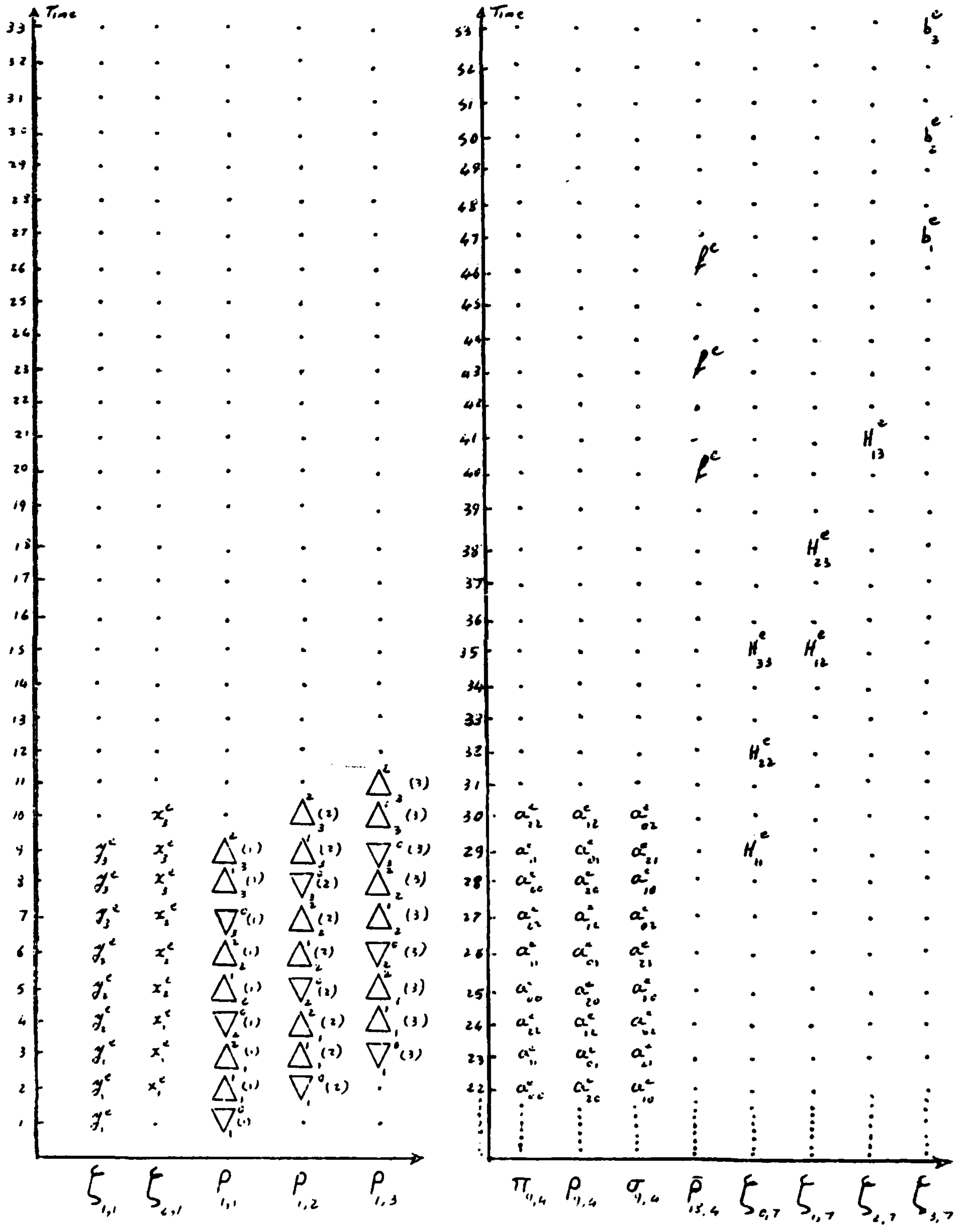


Figure 7.11 - Input and output sequences for k=q=3.

and that the elemental arrays can be generated at a rate of one stiffness matrix/load vector every 3k time units.

7.7. Verification of Pipelined Operation.

From the previous description of the subnetworks N1 through N6, it should be easy to check that all computations are inert in the sense defined in Section 5.4, and hence that the m computations corresponding to the m finite elements can always be pipelined through the subnetworks. Moreover, the techniques of that section may be used to prove that we can achieve the maximum pipeline efficiency by taking the pipe separation τ=3k, which is the maximum span involved in the computation corresponding to one finite element.

The proof procedure is basically the same for the six subnetworks, hence, we will only demonstrate its application to one subnetwork, namely N4. Recall that in Section 7.4 the network I/O description of N4 was found to be given by equation (7.20), which is

$$\zeta_{i,q+1} = \sum_{g=1}^{q} \Omega^{i-8+q-g} [\Omega^{i-9} \pi_{9,g} * \rho_{9,g}] \qquad i=9,\cdots,3k+8 \qquad (7.37)$$

Moreover, when the inputs for a certain instance C^e of the computation are

$$\pi^e_{9,g} = \Omega^{g+3k+8} \overline{\nu}^e_g \qquad g=1,\cdots,q \qquad (7.38.a)$$
$$\rho^e_{9,g} = \Omega^{g+3k+8} \nu^e_g \qquad g=1,\cdots,q \qquad (7.38.b)$$

then the outputs are given by

$$\zeta^e_{i,q+1} = \Omega^{2i+3k+q-9} \eta^e_i \qquad i=9,\cdots,3k+8 \qquad (7.38.c)$$

where the detailed forms of the sequences ν^e_g and $\overline{\nu}^e_g$ containing the input data for C^e, and the sequences η^e_i containing the results of C^e are specified by (7.16) and (7.22), respectively. For the following discussion, we do not need these detailed forms. It suffices to know that $T(\nu^e_g) = T(\overline{\nu}^e_g)$ =3k and $T(\eta_i^e) = 3k-(i-9)$, and hence that the minimum pipe separation is $\tau_m = 3k$.

If the computations for the different elements are pipelined through N4 with a separation of 3k, then the inputs should have the form

$$\pi^*_{9,g} = \Omega^{g+3k+8}\, P^{3k}_{e=1,m}(\bar{\nu}^e_g) = \Omega^{g+3k+8}\, P^{3k}_{e=1,m}(\Omega^{-(g+3k+8)}\, \pi^e_{9,g})$$

and

$$\rho^*_{9,g} = \Omega^{g+3k+8}\, P^{3k}_{e=1,m}(\nu^e_g) = \Omega^{g+3k+8}\, P^{3k}_{e=1,m}(\Omega^{-(g+3k+8)}\, \rho^e_{9,g})$$

where we followed the notation developed in Section 5.4. Using this in the network I/O description (7.37), we get the pipeline outputs in the form

$$\zeta^*_{i,q+1} = \sum_{g=1}^{q} \Omega^{i-8+q-g}\, [\Omega^{i-9}\, \Omega^{g+3k+8}\, P^{3k}_{e=1,m}(\Omega^{-(g+3k+8)}\, \pi^e_{9,g}) * \Omega^{g+3k+8}\, P^{3k}_{e=1,m}(\Omega^{-(g+3k+8)}\, \rho^e_{9,g})]$$

Now, by properties P1 and P8 in the Appendix we obtain

$$\zeta^*_{i,q+1} = \Omega^{i+3k+q}\, P^{3k}_{e=1,m}(\, \Omega^{-(i+3k+q)} \sum_{g=1}^{q} \Omega^{i-8+q-g}[\Omega^{i-9}\, \pi^e_{9,g} * \rho^e_{9,g}]\,)$$

which by (7.37) and (7.38.c) reduces to

$$\zeta^*_{i,q+1} = \Omega^{i+3k+q}\, P^{3k}_{e=1,m}(\Omega^{i-9}\, \eta^e_i) \tag{7.39}$$

Finally, because of $T(\Omega^{i-9}\eta^e_i) = i-9+T(\eta^e_i) = k$, we use P8 to write (7.39) as

$$\zeta^*_{i,q+1} = \Omega^{2i+3k+q-9}\, P^{3k}_{e=1,m}(\eta^e_i)$$

which proves that the sets of results $\{\eta^e_i;\ i=9,\cdots,3k+8\}$ of the different instances $e=1,\cdots,m$ will be correctly produced at the rate of $\frac{1}{3k}$ if the set of inputs $\{\bar{\nu}^e_g\ ,\ \nu^e_g;\ g=1,\cdots,q\}$ are pumped through N4 at the same rate.

Similarly, we can prove that the other subnetworks can operate successfully at maximum pipeline efficiency. Given that the output of the computations associated with a specific element e is described by (7.32) and

(7.36.b), it is then easy to verify that the results of pipelining the computations associated with the m elements, $e=1,\cdots,m$, is described by

$$\zeta_{u,q+4} = \Omega^{6u+w+7}\, P^{3k}_{e=1,m}(\, \Theta^2\, \overline{\mu}^{e}_{u}\,) \tag{7.40}$$

where $\overline{\mu}^{e}_{u}$ is as described in (7.32) and (7.36.b) with the superscript e used to designate the results of the computations associated with the element e. Clearly, (7.40) shows that the system may compute all the elemental arrays in a time equal to $t_c+3k(m-1)$, where $t_c=9k+q+10$ is the time consumed by the first instance of the computation, that is the time for the generation of H^1 and b^1.

Remark 1:

A careful examination of the system presented in this chapter shows that the time of the computation of the m elemental matrices and m elemental vectors may be reduced to $t'_c+(2k+1)(m-1)$ for some special problems in which the coefficients $a^e_{r,l}$ are equal to zero for r=0 or l=0. Examples of this important class of problems are the heat flow, the plain strain and plain stress problems [57]. To obtain this time reduction, some control parameters have to be changed as well as the forms of the input sequences. At this point we note that with the technique of Section 5.4, it can be proved that a successful pipelining of the operation on the modified network requires a pipe separation τ equal to 2k+1. This is larger than the minimum pipe separation $\tau_m=2k$ for the computation, which means that the modified network cannot operate at maximum pipeline efficiency.

Remark 2:

It may be sometimes desirable to slow down the rate at which the output is generated. More specifically, we may require that the output sequences have the forms

$$\zeta_{u,q+4} = \Omega^{c(2u+k)+q+16}\ \Theta^{c-1}\ \bar{\mu}_u^{e} \qquad u=0,\cdots,k$$

for some integer $c \geqslant 3$, rather than $c=3$ in the formulas (7.32) and (7.36.b). This flexibility may be accomplished by allowing for a modification of the control parameters in the systolic cells. This is especially applicable if the cells are controlled by external control signals that propagate systolically in the network. In this case the form of the input should be changed accordingly and the separation during pipelined operation should be set to ck. We will not consider here the equations that describe the operation of the modified network in any detail.

Remark 3:

In the case of $d>1$ degrees of freedom per node, (see Remark 1 in Chapter 6), the input coefficients $a_{r,l}^{e}$ in ALG5 are $d \times d$ matrices and the entries $H_{i,j}^{e}$ are $d \times d$ submatrices. In order to compute the d^2 elements of $H_{i,j}^{e}$ without slowing down the system, we may replace the subnetwork N5 by d^2 identical subnetworks, each of which generates the corresponding entry in the submatrix $H_{i,j}^{e}$ when provided with the appropriate entry in the $d \times d$ matrices $a_{r,l}^{e}$, $r,l=0,1,2$. Similarly, d identical copies of the subnetwork N6 are needed for the simultaneous computation of the d components of the subvector b_i^{e}.

8. THE ASSEMBLY STAGE.

The elemental arrays H^e and b^e generated by the systolic system of the last chapter are the main contributors to the global stiffness matrix H and load vector b. The assembly of the contribution of a specific finite element e to the global arrays begins with the modification of H^e and b^e, if necessary, to account for the boundary conditions. The elements of the modified arrays $\overline{H}^e$ and $\overline{b}^e$, $e=1,\cdots,m$ are then appropriately scaled and assembled according to the formulas (6.8/10). In order to scale the arrays of a specific element e, we need to know the global labels of the nodes (e,i), $i=1,\cdots,k$ belonging to that element. Given the local/global mapping, the assembly of $\overline{H}^e$ and $\overline{b}^e$, $e=1,\cdots,m$ may be described by the following algorithm:

ALG6

```
1) Initialize the global matrix H and the global vector b to zero.
2) FOR e=1 TO m DO
    2.1) FOR i=1 TO k DO
        2.1.1) FOR j=1 TO k DO
                $H(glob(e,i),glob(e,j)) = H(glob(e,i),glob(e,j)) + \overline{H}^e_{i,j}$
        2.1.2) $b(glob(e,i)) = b(glob(e,i) + \overline{b}^e_i$
```

In Sections 1 and 2 of this chapter, we deal with the generation of the global labels and the modification of H^e and b^e to satisfy the boundary conditions. Then in Sections 3 and 4, we discuss the parallel implementation of the assembly stage, and finally in Section 5, we show that the assembly stage may be eliminated if the resulting system of linear equations

(6.7) is to be solved iteratively.

It should also be mentioned that Law [34] suggested a systolic architecture for the assembly of the global matrix H. However, the timing and form of the input data required by his architecture are not compatible with the output generated by the system discussed in Chapter 7. Moreover, Law assumes that each cell in the array does perform matrix operations, which of course requires a larger clock cycle than the one for the simple operations used in the system that generates H^e. For these reasons, we were not able to use Law's array in our design. This, of course, does not exclude the possibility of employing his array in other designs of parallel finite element systems.

8.1. Generation of the global labels.

The purpose of this section is to design a subnetwork that associates the global labels glob(e,i) and glob(e,j) with each $H^e_{i,j}$ generated from N5, and adjusts the entries of H^e that correspond to the nodes at which the solution function should be zero (see Remark 2 in Chapter 6).

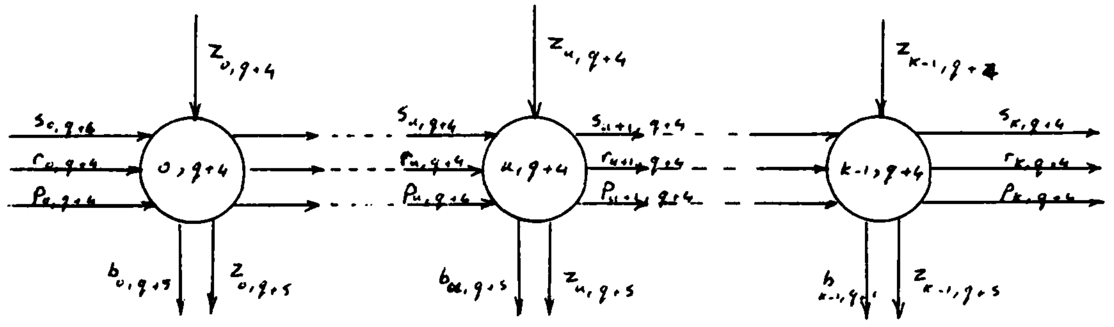


Figure 8.1 - The graph for N7

In Figure 8.1, we show the graph of a network N7 that performs these two tasks. Its input links $z_{u,q+4}$, $u=0,\cdots,k-1$ are to be connected to the corresponding outputs of the subnetwork N5 that carry the elements of H^e as described by equation (7.32). The input links $s_{0,q+4}$ and $r_{0,q+4}$ carry identical information, namely the global labels of the nodes (e,i), $i=1,\cdots,k$. More precisely, we have

$$\sigma_{0,q+4} = \rho_{0,q+4} = \Omega^{w+7}\ P^{3k}_{e=1,m}(\ \Theta^2\ \gamma^e\ ) \qquad (8.1.a)$$

where $T(\gamma^e) = k$ and $\gamma^e(t) = glob(e,t)$. Finally, the input link $p_{0,q+4}$ carries a single bit for identifying the nodes at which the solution should be zero, that is, the nodes that lie on ∂Q_0. More precisely, we have

$$\pi_{0,q+4} = \Omega^{w+7}\ P^{3k}_{e=1,m}(\ \Theta^2\ \gamma^e_0\ ) \qquad (8.1.b)$$

where $T(\gamma^e_0) = k$ and

$$\gamma^e_0(t) = \begin{cases} 1 & \text{if node } (e,t)\ \epsilon\partial Q_0 \\ 0 & \text{otherwise} \end{cases}$$

The operation performed by any of the k cells in the network is very primitive. First, each cell delays the data on the s, r and p links as follows

$$\begin{aligned} \rho_{u+1,q+4} &= \Omega^{6u}\ \rho_{u,q+4} && u=0,\cdots,k-1 \\ \sigma_{u+1,q+4} &= \Omega^{3u}\ \sigma_{u,q+4} && u=0,\cdots,k-1 \\ \pi_{u+1,q+4} &= \Omega^{6u}\ \pi_{u,q+4} && u=0,\cdots,k-1 \end{aligned}$$

This ensures that for each cell u the time of arrival of $H^e_{t,t+u}$ on the link $z_{u,q+4}$ coincides with the time of arrival of $glob(e,t)$, $glob(e,t+u)$ and $\gamma^e_0(t)$ on the links $r_{u,q+4}$, $s_{u,q+4}$ and $p_{u,q+4}$, respectively, thus allowing the cell u to modify the elements of H^e appropriately, and to produce the modified elements on the output link $z_{u,q+5}$. The cell also produces on the output link $b_{u,q+5}$ the pair ($glob(e,t)$, $glob(e,t+u)-glob(e,t)$). This pair specifies the location at which $H^e_{i,j}$ is to be accumulated, assuming that the global matrix H is stored as a band matrix. In terms of sequence equations, this translates into the formulas

$$\zeta_{u,q+5} = \Omega^{6u+w+8}\ P^{3k}_{e=1,m}(\Theta^2 \bar{\mu}^e_u) \qquad u=0,\cdots,k-1 \qquad (8.2.a)$$

$$\beta_{u,q+5} = \Omega^{6u+w+8} \, P^{3k}_{e=1,m}(\, M_1^{1,1,1}(\Theta^2\lambda_u^e \, , \, \Omega\Theta^2\overline{\lambda}_u^e \, , \delta^*)) \quad u=0,\cdots,k-1 \quad (8.2.b)$$

where $\overline{\mu}_u^e$ is as in (7.40) except that the entries of H^e are now appropriately modified. The sequences λ_u^e and $\overline{\lambda}_u^e$ are described by $T(\lambda_u^e) = T(\overline{\lambda}_u^e) = k-u$ and

$$\lambda_u^e(t) = glob(e,t)$$
$$\overline{\lambda}_u^e(t) = glob(e,t+u) - glob(e,t)$$

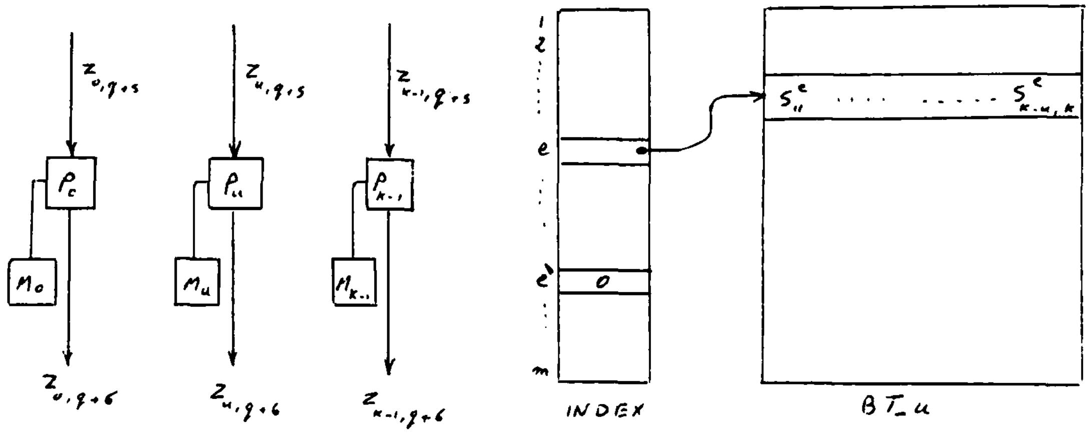
Finally, we note that a cell (k,q+4) may be added to N7 to modify the elements of b^e and associate with each b_i^e the global label glob(e,i).

8.2. Essential boundary conditions.

The essential boundary conditions are the ones responsible for the terms containing line integrals in the variational formulation (6.1/3). It was shown that in the finite element approximation, the effect of these terms may be isolated in the form of a matrix S^e and a vector s^e that are to be added to the elemental arrays H^e and b^e, respectively. However, for a given problem, most of the arrays S^e and s^e are zero arrays, and if non zero, they contain only few non zero entries. Consequently, adding special hardware for the computation of the few non zero entries of S^e and s^e, $e=1,\cdots,m$ is not justified from a practical point of view, especially since the general formula for the generation of these entries is complicated and contains many coefficients that are set to zero for particular problems.

More appropriately, these few non zero entries should be computed by the general purpose machine that controls the entire computation as a part of the presetting procedure, and then added to the corresponding entries of H^e and b^e. In order to ensure the continuity of the flow of data, the addition should take place in an intermediate sub-system residing between the system that generates H^e and b^e, and the one that assembles the global arrays H and b. Conceptually, the intermediate sub-system should contain a memory to store the non zero entries of S^e and s^e and some logic to add every non zero entry to the corresponding entry in H^e or b^e at the time of its generation from the subnetwork N7. As we did in the last section, we will only consider the stiffness matrices and suggest a possible implementation for the addition $\overline{H}^e = H^e + S^e$. The extension to the load vector $\overline{b}^e$ should be obvious and simple.

The systolic nature of the subnetworks N1, $\cdots$, N7 enables us to time exactly the data on the output links $z_{u,q+5}$, $u=0,\cdots,k-1$ (see equations (8.2)). Each of these links may be directed into a systolic processor P_u, $0 \le u \le k-1$ that has access to a local memory M_u as shown in Figure 8.2(a). Each processor P_u contains a register 'CURRENT_u' that it sets to one at time 6u+w+8, and increments every 3k time units. Hence, when an element $H^e_{t,t+u}$, $1 \le t \le k-u$ appears on an input link $z_{u,q+5}$, the corresponding register CURRENT_u in P_u should contain the label e of the finite element being processed.



(a) The general architecture (b) The content of M

Figure 8.2 - The assembly stage

Each local memory M_u, $0 \leq u \leq k-1$ contains an array INDEX (see Figure 8.2(b)) that has one entry for each finite element $e=1,\cdots,m$. If the matrix S^e corresponding to the finite element e is zero, then INDEX(e)=0. On the other hand, if $S^e \neq 0$, then INDEX(e) contains a pointer to another array 'BT_u' (Boundary Terms for off-diagonal u) where the entries $S^e_{t,t+u}$, $t=1,\cdots,k-u$ of the u^{th} off-diagonal of S^e are stored (including zeroes) in the specified order. This order is the same as the one in which the elements $H^e_{t,t+u}$, $1 \leq t \leq k-u$ appear on $z_{u,q+5}$.

Thus, at times $6u+w+8+3(e-1)k$, $e=1,\cdots,m$, after the processor P_u has changed the value of CURRENT_u, it consults INDEX(CURRENT_u). If it is a zero, then, for the next 3k time units, the data items on the input link $z_{u,q+5}$ are placed unchanged on the output link $z_{u,q+6}$. Otherwise, P_u retrieves from BT_u the elements $S^e_{t,t+u}$, $t=1,\cdots,k-u$, one every three time units, adds each of them to the corresponding $H^e_{t,t+u}$ and puts the result on $z_{u,q+6}$.

An alternative architecture could be obtained by replacing each M_u with an associative memory that uses the labels $1 \leq e \leq m$ as keys to store the array BT_u, or by using only one global memory M instead of the k memories M_u, $u=0,\cdots,k-1$. In all of the above cases, the content of the memory is computed and preloaded by the host computer. A completely different approach would be to perform the matrix addition H^e+S^e on a systolic network that does not have any memory. However, matrix addition is a communication-bound rather than a compute-bound operation, and in our case, most of the data in S^e are trivial (zeroes). Consequently, such a memoryless subnetwork would require many unnecessary data communication, which we tried to avoid in our design.

8.3. The frontal principle.

The assembly stage in the finite element analysis is an example of a simple computation that is irregular, and hence, does not lend itself to a simple systolic implementation. In Section 8.5, we will show that the assembly stage may be eliminated from the analysis if an iterative scheme is used for the solution of the resulting system of equations Hu=b. However, if a direct solver is to be used, then the global arrays H and b must be assembled in order to proceed with the direct solution. In what follows, we will only consider the assembly of the stiffness matrix H. Although we will not discuss the assembly of the load vector *b*, we note that it may be included here by considering *b* as an additional column of H.

At first glance, it would appear that the solution of Hu=b may not start before the assembly of H is completed. Especially in a parallel system, this would have two disadvantages; firstly, it does not allow the computation of the different components of the system to proceed in parallel, and secondly, it requires some intermediate storage to store H. Since, in practice, finite element problems with n in the order of several thousands are not uncommon, it is obvious that auxiliary storage (disks) may be needed for *H*, thus slowing down the system even more.

Fortunately, we do not have to wait until H is completely assembled and we may start the solution process as soon as some rows of H are assembled. The assembly and the solution processes may then proceed in parallel in a producer/consumer type of interaction: The assembler process being the producer of the assembled rows of H, and the solution process being the consumer of these rows.

In order to explain this assembly technique, we denote by $\overline{h}_i^e$ the i^{th} row of the elemental matrix $\overline{H}^e$ and by h_i the i^{th} row of the global matrix

H. We also note that a node with a global number i may share more than one finite element, and hence, may have more than one local label.

Each row h_i, $1 \le i \le n$, of the matrix H corresponds to a certain node in the finite element mesh, namely node i. This row is assembled by accumulating contributions from the rows $\bar{h}^{e1}_{j1}$, $\cdots$, $\bar{h}^{er}_{jr}$, where $e1, \cdots, er$ are the elements that share node i, and $(e1, j1), \cdots, (er, jr)$ are the corresponding local labels of node i.

In accordance with the system of Chapter 7, we will assume from now on that during the assembly process the elemental matrices are processed in the order $\bar{H}^1, \cdots, \bar{H}^m$, and the rows within each elemental matrix are considered in the order $\bar{h}^e_1, \cdots, \bar{h}^e_k$. The elements in each row $\bar{h}^e_j$ are accumulated in the proper position of the global matrix H (see ALG6), or more precisely in row h_i of H, where $i = glob(e, j)$.

Before going further, we introduce some terminology. During the assembly process, a row h_i is said to be active from the moment of the appearance of a row $\bar{h}^e_j$ with $glob(e, j) = i$, that is from the time when its assembly actually starts. On the other hand, h_i is called a ready row immediately after the accumulation of the last row $\bar{h}^r_l$ with $glob(r, l) = i$, that is after its assembly has been completed. In other words, a certain row of H is partially accumulated in the period between the instance it becomes active until the instant it becomes ready. Once it has become ready, a row may be processed by the solver.

These ideas were first formulated in the framework of the so called frontal technique [24] used in sequential finite element systems. The goal is to interleave the assembly and the solution phases in order to minimize the high-speed storage requirements. The order in which the rows of H

become ready is usually determined by a preprocessing step.

The same basic idea will be used in our system to achieve our goal of allowing the assembly and the solution processes to be executed in parallel. More specifically, whenever a row in H becomes ready, it will be passed to the solution process. This also allows for the reduction of storage for the assembly since the storage allocated for a row may be released whenever this row is passed to the solution process (consumed).

However, in all the known parallel schemes for the direct solution of $Hu=b$, the rows of H have to be processed in a sequential order, and hence, the size of the storage required by the assembly stage is determined by the maximum number of rows that are at any one time either active or ready but not yet passed to the solver (consumed) because a preceding row is not yet ready. For this reason, the global labels given to the nodes in the finite element mesh should be such that the rows of H become ready in an almost sequential order. By this we mean that there should exist a relatively small constant c such that for any i and j satisfying $1 \leqslant i \leqslant n$ and $j < i + c$, row j does not become ready before row i. With this restriction, we can restore the sequential order by using a buffer large enough to store up to c rows of H.

If a band storage mode is used for storing the rows of the banded matrix H, then it is also advantageous to minimize the bandwidth of H. In this connection, it has to be noted that the bandwidth of H is determined by the global numbering of the nodes in the corresponding finite element mesh. In fact, given a global numbering and denoting by $2B+1$ the bandwidth of the matrix H resulting from this numbering, it is easy to show that

$$B = \max(\bar{l}^e - \underline{l}^e \; ; e=1, \cdots, m) \tag{8.3}$$

where $\overline{l}^e$ and $\underline{l}^e$ are the largest and smallest global node numbers in the finite element e, respectively.

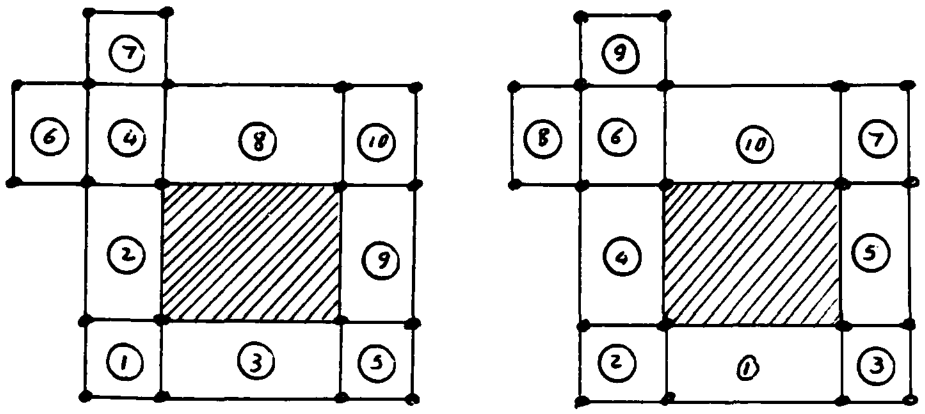
Many heuristic node numbering algorithms were suggested for reducing the bandwidth (e.g. [13]). However, if the assembly and the solution processes are to be executed in parallel, then we need a numbering scheme that, in addition to reducing the bandwidth, has the goal of reducing the number of active rows of H at a given time, and of producing the assembled rows of H in an almost sequential order. The following algorithm takes these goals into consideration.

ALG7

```
1) Assign a unique label $e$, $1 \le e \le m$ to each of the m finite elements.
2) Number the nodes sequentially in the following order
    2.1) Number, in arbitrary order, the nodes of element 1.
    2.2) FOR $e=2,\cdots,m$ DO
        2.2.1) Number, in arbitrary order, the nodes in elements e that
               are not yet numbered.
```

ALG7 does not specify the method of labeling the elements $e=1,\cdots,m$. If the element labels satisfy the property that for any e, $1 \le e \le m$, element e contains at least one node that does not belong to any element $1,\cdots,e-1$, then we call such a labeling scheme a proper element labeling. For example, the element labeling in Figure 8.3(a) does satisfy the above conditions, and hence is a proper labeling. On the other hand, the labeling for the same mesh in Figure 8.3(b) is not proper because element 10 does not contain any node which is not in the elements $1,\cdots,9$. With this definition, we can prove the following proposition:

Proposition 8.1: If the nodes of the finite element mesh are numbered using ALG7 and the labeling in step 1 is a proper element labeling, then for any



(a) A proper labeling (b) A non proper labeling

Figure 8.3 - Elements labeling.

e, $1 \leq e \leq m$, the elements $e, \cdots, m$ do not contain a node with a global number smaller than $\bar{l}^e - B$, where $\bar{l}^e$ is the largest node number in element e, and $2B+1$ is the bandwidth of the matrix resulting from this particular node numbering.

Proof: ALG7 and the proper element labeling imply that for any $r > e$, the finite element r should contain a node with a label $\bar{l}^r > \bar{l}^e$. However, from equation (8.3) we find that $\bar{l}^r - \underline{l}^r \leq B$ or $\underline{l}^r \geq \bar{l}^r - B > \bar{l}^e - B$. Hence, because the label of any node in element r is larger than $\underline{l}^r$, we conclude that the elements $e+1, \cdots, m$ do not contain a node with a global label smaller than or equal to $\bar{l}^e - B$. Now, the result of the proposition follows from the fact that any node in element e has a label larger than or equal to $\underline{l}^e = \bar{l}^e - B$. ■

In our system, we will assume that the global node numbering scheme satisfies the conditions stated in Proposition 8.1, and that the finite elements are processed in the order $1, \cdots, m$. Then, the proposition guarantees that during the assembly of the contributions of an elemental matrix $\bar{H}^e$ into the global matrix H, the rows $1,\cdots,\bar{l}^{e}-B-1$ of H are completely assembled and will not be modified by any contribution from the elemental arrays $\bar{H}^{r}$, $r>e$. In other words, if row i, $1\leq i\leq n$ in H is active, then the rows up to $i-(B+1)$ are ready and may be processed by the solver and thus the storage associated with them may be released.

Definition: If, at a specific time during the assembly of H, a certain row i, $1\leq i\leq n$ is active, then the rows $1,\cdots,i-(B+1)$ are called B_ready rows of H.

From Proposition 8.1, it follows that B_ready rows are ready rows of H. However, a given row may be ready before it becomes B_ready. Being pessimistic, we will follow the rule of not allowing the solution process to access a certain row in H before that row becomes B_ready, except of course for the last rows $n-B,\cdots,n$ that may be accessed only after the assembly of H is completed. This may decrease somewhat the efficiency, but it has the advantage of eliminating the preprocessing stage that would otherwise be required for determining the time at which a certain row in H becomes ready.

If the above rule is used as the basis for the interaction between the assembler and the solution processes, then, the assembler process should contain storage for holding at least $B+1$ active rows of H. Moreover, as in the case of any producer/consumer problem, additional buffers may be required depending on the relative speeds of production of the B_ready rows and their consumption.

It is not hard to see that, due to the nature of the assembly process, the rate at which the rows of H become B_ready is not constant. Hence we will consider the average of that rate. This average rate differs from one problem to the other. In order to be more specific, we first note that the inputs to the assembly stage are the mk rows $\bar{h}_{i}^{e}$, $i=1,\cdots,k$,

$e=1,\cdots,m$, and that the outputs from that stage are the n rows h_i, $i=1,\cdots,n$. We also assume that the execution time of the assembly process is T_a time units and that its data bandwidths are sufficient to transmit the input of one row $\overline{h}_i^e$, as well as the output one B_ready row at a time. With this, we suppose that the rate of arrival of the rows at the assembly stage is constant, and we denote this rate by $r_i = \frac{mk}{T_a}$ rows/time unit. We also define the average rate at which the assembly stage produces the B_ready rows of H as $\overline{r}_p = \frac{n}{T_a} = \frac{n}{mk} r_i$. Since the ratio $\frac{n}{mk}$ changes from one problem to another, it should be clear that, for a fixed input rate, the average rate, at which the rows of H become B_ready, does depend on the problem at hand.

Hence, in general, we cannot achieve the desired match between the average rate of production of the B_ready rows and the rate at which the solution process is ready to consume these rows. More specifically, if the solution process is capable of processing (consuming) r_c rows of H every time unit, then we have one of the following possibilities:

1) $r_c < \overline{r}_p$, that is the solver cannot consume the B_ready rows of H fast enough. In this case, the number of B_ready but unconsumed rows grows continuously and so does the size of the memory requirement of the assembler process.

2) $r_c \geq \overline{r}_p$, in which case the assembly stage needs only to provide storage for $B+1+K_b$ rows of H, where K_b is the size of the storage needed to buffer the fluctuations in the production rate of the B_ready rows of H.

From the above discussion, it is clear that to limit the size of the storage required in the assembly process, we need to guarantee that $r_c \geq \overline{r}_p$.

This, however, will force the solution process to execute at a speed slower than its nominal speed r_c, because often it will be forced to stop execution and wait for the B_ready rows of H to become available. As a result, the synchronization between the solution and the assembly processes can no longer be controlled by a global clock. Instead, a shake hand protocol should be used for that purpose (see Section 5.5).

Assuming that $r_c > \bar{r}_p$, we may control the size of the additional buffer K_b by controlling the fluctuation in the rate $\bar{r}_p$ of production of the B_ready rows. More specifically, if we can guarantee that, at any time t during the assembly process, the rate $r_p(t)$ of production of the B_ready rows, is smaller than or equal to the consumption rate r_c, then, any row of H will be consumed as soon as it becomes B_ready, thus reducing the size of the additional buffer K_b to zero.

Here, we note that the rate r_c is uniquely specified by the solution process, and hence that the relation between r_c and $r_p(t)$ cannot be obtained by studying solely the assembly process. However, by specifying in ALG7, a certain order for numbering the nodes within each element, rather than keeping this order arbitrary, we can prove that $r_p(t)$ may not , at any time t, exceed the constant rate r_i of arrival of the rows at the assembly stage. This result will be used in Section 9.1. We start by modifying ALG7 to obtain the following node numbering algorithm.

ALG8

Given a proper element labeling for the finite elements and a corresponding local numbering of the nodes, obtain the global numbering by giving the nodes sequential numbers $i=1,\cdots,n$ in the following order:

```
1) FOR $i=1,\cdots,k$ DO
```

```
        $glob(1,i)=i$
2) FOR $e=2,\cdots,m$ DO
     FOR $i=1,\cdots,k$ DO
          IF node $(e,i)$ is already numbered THEN skip
          ELSE increase $j$ by one and set $glob(e,i)=j$.
```

Now, we can prove the following result:

Proposition 8.2: Let the nodes in the finite element mesh be numbered by algorithm ALG8 and let the elemental arrays be accumulated in the global array in the order $\bar{H}^1,\cdots,\bar{H}^m$ with the rows within each $\bar{H}^e$ being accumulated in the order $\bar{h}_1^e,\cdots,\bar{h}_k^e$. Then, the rows of H become active in a purely sequential order.

Proof: Consider any two rows h_{i1} and h_{i2} of H with $i1<i2$, and let $(e1,j1)$ be the local label for node $i1$ such that $glob(e1,j1)=i1$, and that we have $e1'>e1$ for any other local label $(e1',j1')$ with $glob(e1',j1')=i1$. In other words, element $e1$ is the first element containing node $i1$. Similarly, let $(e2,j2)$ be the local label of $i2$ with respect to the first element containing it. From the definition of active rows, we know that the rows h_{i1} and h_{i2} of H become active when rows $\bar{h}_{j1}^{e1}$ and $\bar{h}_{j2}^{e2}$ are received by the assembler, respectively. However, ALG8 and the fact that $i1<i2$, together imply that either $e1<e2$ or $e1=e2$ and $j1<j2$. In both cases, we conclude from the hypothesis of the proposition that $\bar{h}_{j1}^{e1}$ is received by the assembler before $\bar{h}_{j2}^{e2}$, and hence that row $i1$ becomes active before row $i2$. ■

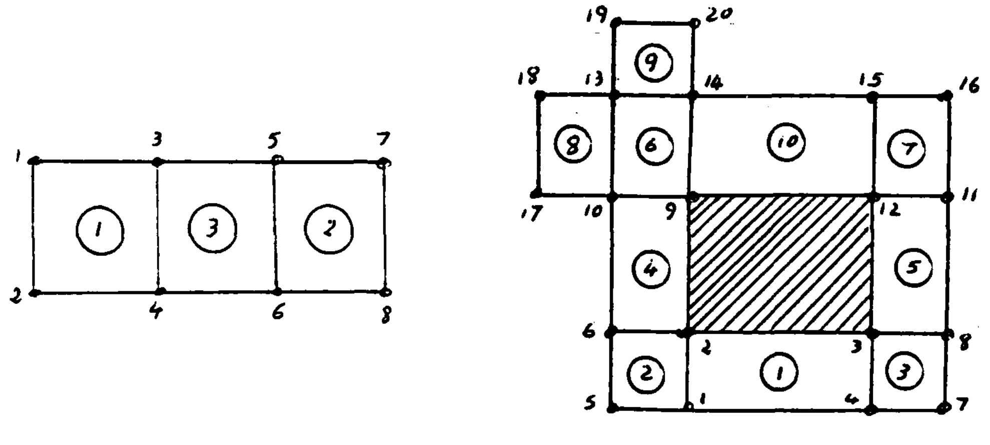
Proposition 8.3: Assume that the hypotheses of Proposition 8.2 apply, and, moreover, that the rate r_i at which the rows $\bar{h}_i^e$, $i=1,\cdots,k$, $e=1,\cdots,m$, arrive at the assembly stage is constant. Then the rate of production of the B_ready rows $r_p(t)$ at any time t, cannot exceed r_i rows/time unit.

Proof: Note here that the arrival of a certain row $\bar{h}_i^e$, $1 \leq i \leq k$, $1 \leq e \leq m$, at the assembler may at most activate one row of H, namely the row labeled $l = glob(e, i)$. Hence, after the arrival of $\bar{h}_i^e$, the rows $h_1, \cdots, h_{l+B+1}$ are, by definition, B_ready rows. However, from Proposition 8.2, we know that row h_{l-1} should already be active at the instant when row h_l becomes active. That is, before the arrival of $\bar{h}_i^e$, rows $h_1, \cdots, h_{l+B}$ were B_ready. In other words, the arrival of $\bar{h}_i^e$ may create at most one B_ready row, namely row h_{l+B+1}. ■

We now return to algorithms ALG7 and ALG8. Although these algorithms provide good numbering schemes from the point of view of processing the assembly and solution processes in parallel, we still have to ensure that they do not result in a large value of B. For this we note that ALG7 and ALG8 are two-step algorithms: First, the finite elements $e = 1, \cdots, m$ are labeled, and then the nodes within the elements are numbered. To our knowledge, Fenves and Law [15] were the first to suggest a two-step numbering algorithm. They reported experimental results which show that if the Cuthill-McKee [13] algorithm is used to number the finite elements with a two step algorithm, then the bandwidth of the resulting matrix H is comparable with the one resulting from the best known heuristic algorithm for minimizing the bandwidth. Here, in the application of the Cuthill-McKee algorithm for labeling the elements, Fenves and Law consider two elements to be neighbors if they share a common boundary.

Clearly, the hypothesis in Proposition 8.1, requiring the element labeling scheme in ALG7 to be proper, is essential. In order to see this, consider the simple example of Figure 8.4(a) where the element labeling is not proper (element 3 does not contain a node not in elements 1 or 2).

Using ALG7 to label the nodes as in the figure, it is easy to see that the result of Proposition 8.1 does not hold because element 3 contains nodes with global numbers less than $\bar{l}^{2}-B=8-3=5$, namely nodes 3 and 4.



(a) B=3 (b) B=8

Figure 8.4 - Application of ALG7 for node numbering.

According to the results in [15], we may obtain a relatively small bandwidth with a two step node numbering algorithm if we use the Cuthill-McKee scheme to number the elements. However, we note that this may result in a non proper element labeling scheme. For example, the elements in Figure 8.3(b) were labeled using the Cuthill-McKee algorithm but the resulting labeling is not proper (because of element 10). The corresponding node numbering is shown in Figure 8.4(b), where the largest node number in element 9 is $\bar{l}^{9}=20$, but element 10 contains a node with a number smaller than $\bar{l}^{9}-B=12$, namely node 9. In this case, however, a proper element labeling may be obtained by starting the Cuthill-McKee algorithm from a different element (see e.g. Figure 8.3(a)). We examined many strange shapes of meshes and in only rare cases did the application of the Cuthill-McKee algorithm result in a non proper labeling. Moreover, in all these rare cases a proper labeling was easily obtained by changing the starting element. The existence and construction of a proper element labeling scheme for a given finite element mesh is still a question that needs to be answered.

Finally, we note that, by allowing the solution process to access the rows of H only when they become B_ready, we do not increase the storage requirement of the assembly process. As an implication of Propositions 8.1 and 8.3, this storage should be large enough to hold $B+1$ rows of H. However, this is the minimum amount of storage that should be provided by the assembler even if the rows of H were accessed as soon as they become ready. In fact, there always exist an element e such that $\overline{l}^e - \underline{l}^e = B$, and hence the assembly of this element does require the storage of $B+1$ rows of H, assuming of course, that the rows of H are stored in consecutive locations.

8.4. A parallel implementation of the assembly process.

The discussion of the last section showed that the *potential parallelism* between the assembly and the solution processes may be paralyzed if the solution process is designed to access the rows of the assembled matrix H in order and the node numbering scheme does not take this into consideration. Here, we prevent this from happening by assuming that the node numbers satisfy the conditions of Propositions 8.1 and 8.3.

Next, we modify the definitions of $\overline{h}_i^e$ and h_i such that $\overline{h}_i^e$ = $(\overline{H}_{i,j}^e ; j=i, \cdots ,k)$ and h_i = $(H_{i,j} ; j=i, \cdots ,i+B)$ are the sets of elements of interest in the i^{th} rows of the symmetric matrices $\overline{H}^e$ and H, respectively. In addition, we denote by L_i^e = $((glob(e,i), glob(e,j)) ; j=i, \cdots ,k)$ the set that contains the information about the position at which each element of $\overline{h}_i^e$ is to be assembled into the global matrix H. With this, we may describe the assembler process as follows

PROCESS 'ASSEMBLER'

```
    Max_ready := 0 ; Consumed := 0 ; H0 := 0 ;
    Interrupt I: /* High priority */
        1) Get $\bar{h}_i^e$ and $L_i^e$ from I_port ;
        2) Accumulate the elements of $\bar{h}_i^e$ in H0 ;
        3) IF (Max_ready < glob(e,i)-B-1) THEN Max_ready := glob(e,i)-B-1 ;
        4) EXIT from the interrupt.

    Interrupt O: /* Low priority */
        1) WAIT until (Max_ready > Consumed) OR (D_flag is set) ;
        2) Send $h_c$, c := Consumed to O_port ;
        3) IF (Consumed = n) THEN STOP the ASSEMBLER process ;
        4) Consumed := Consumed + 1 ;
        5) EXIT from the interrupt.
```

Here, the following notes are in order:

a) The interrupt I takes place when a new input arrives at I_port. The data bandwidth of I_port is assumed to be large enough to input the sets $\bar{h}_i^e$ and L_i^e, $1 \leq e \leq m$, $1 \leq i \leq k$.

b) The interrupt O has a lower priority than the interrupt I and it takes place when O_port is ready to receive the next output. The data bandwidth of O_port is assumed to be large enough to output a set h_i, $1 \leq i \leq n$.

c) The flag D_flag is set externally when the input of all the elemental matrices is completed.

It is straightforward to verify, on the basis of Proposition 8.1, that the above process will output the rows of H to O_port only when they are completely assembled. If, moreover, the solution process is able to consume the rows of H at a rate faster than they can be provided at O_port, then we do not need to provide storage for the n rows of H, and it will be enough to provide a circular buffer to store $B+1+K_b$ rows of H, where the need for the additional buffer K_b was discussed in Section 8.3.

The process 'ASSEMBLER' is assumed to handle a large amount of data at a high rate due to the large input and output data bandwidths. Consequently, we may need more than one processor to execute this process. Fortunately, the 'ASSEMBLER' may be easily decomposed into a number of parallel sub-processes, each responsible for managing the data on one or more off-diagonals of the matrix H.

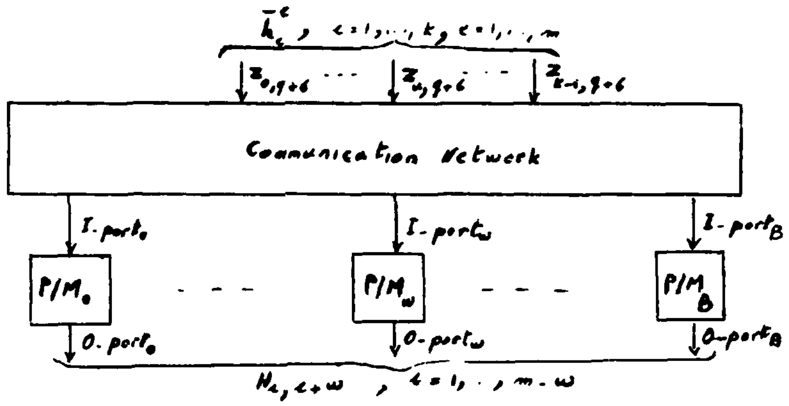


Figure 8.5 – A parallel architecture for the assembler

Consider, for example, the extreme case where we have $B+1$ sub-processes 'ASSEMBLER_w', $w=0,\cdots,B$ running on $B+1$ processor/memory units (see Figure 8.5). Here 'ASSEMBLER_w' manages the assembly of the w^{th} off-diagonal of H. The communication network in Figure 8.5 distributes the elements of $\bar{h}^e_i$ such that each element $\bar{H}^e_{i,j}$ is sent to the P/M unit responsible for its accumulation. More precisely, the communication network receives with each element $\bar{H}^e_{i,j}$ the global/local map information $v=glob(e,i)$

and $w = ABS(glob(e,i) - glob(e,j))$, (ABS= absolute value), which specifies that $\overline{H}^e_{i,j}$ is to be accumulated in the w^{th} off-diagonal position of row v of H. With the value of w, the network then sends both $\overline{H}^e_{i,j}$ and v to P/M_w which completes the accumulation.

One difficulty arises from this decomposition of 'ASSEMBLER', namely that upon receipt of a certain row $\overline{h}^e_i$, the entries of this row will be distributed on the few P/M's that are responsible for their accumulation. Hence, only these few P/M's will detect the arrival of $\overline{h}^e_i$ and update accordingly their copy of the variable 'Max_ready'. The copies of Max_ready in the other P/M's will not be updated unless we provide for some sort of inter-process communication. Since P/M_0 receives the diagonal element of every row $\overline{h}^e_i$ arriving at the assembler, it seems natural to have only P/M_0 update its value of Max_ready, and then send a message to the other P/M's with the new value of Max_ready, whenever it is updated. This message may be broadcast to the other processors or passed from one processor to the next (a daisy chain). With this observation, we may describe the process in any processor/memory unit P/M_w as follows

Sub-process ASSEMBLER_w :

```
    Max_ready := 0  ; Consumed := 0 ; diag_w() := 0 ;
    Interrupt I: /* High priority */
        1) Get $\overline{H}^e_{i,j}$ and $v = glob(e,i)$ from I_port$_w$ ;
        2) diag_w(v) := diag_w(v) + $\overline{H}^e_{i,j}$ ;
        3) FOR $P/M_0$ ONLY : IF (Max_ready < $v-B-1$) THEN
            3.1) Max_ready := v-B-1 ;
            3.2) Send messages to the other P/M's with the new value of
                 Max_ready ;
    4) EXIT from the interrupt ;

Interrupt O: /* Low priority */

    1) WAIT until (Max_ready > Consume) OR (D_flag is set) ;
    2) Send diag_w(Consumed) to $O\_port_w$ ;
    3) IF (Consumed = n) then STOP the ASSEMBLER_w process.
    4) Consumed := Consumed + 1
    5) EXIT the interrupt ;

Interrupt M: /* High priority */

    Update the local copy of Max_ready as received from $P/M_0$.
```

here the high priority interrupt M takes place only in $P/M_1, \cdots, P/M_B$ when a message is received from P/M_0. Similar to the process ASSEMBLER, the linear array diag_w() may be replaced by a circular buffer of length $B+1+K_b$. Note that the message-passing communication technique may be replaced by a global shared variable, or by letting every P/M receive an indication that a row $\vec{h}_i^e$ has been received.

Finally, we note that the communication network of Figure 8.5 may be implemented as a binary tree network [22] where the nodes at consecutive levels use the corresponding bits of the address w to send $\vec{H}_{i,j}^e$ and v to either its left or right successor node, with I_port_w, $w=0,\cdots,B$ placed at the leaves of the tree. We do not intend to discuss here the communication network in any details.

8.5. Elimination of the assembly stage.

To our knowledge, Belytschko and al. [3] were the first to notice that the multiplication of the global matrix H by a vector p can be completed without assembling H. More precisely, from equation (6.8) we have

$$H\,p = \sum_{e=1}^{m} M^{eT}\,\overline{H}^{e}\,M^{e}\,p$$

$$= \sum_{e=1}^{m} M^{eT}\,\overline{H}^{e}\,p^{e}$$

where $p^{e} = M^{e}\,p$ is a k-dimensional vector containing those entries in p that correspond to the nodes belonging to the finite element e. that is, $p^{e}(j) = p(glob(e,j))$, for $j=,1\cdots,k$.

The following algorithm describes precisely the use of the unassembled matrices $\overline{H}^{e}$, $e=1,\cdots,m$ in the computation of the product vector $y = x + H\,p$, where x and p are given n-dimensional vectors.

ALG9

```
1) FOR $i=1,\cdots,n$ DO
    1.1) y(i) = x(i)
2) FOR $e=1,\cdots,m$ DO
    2.1) form the vector $p^{e}$ from
            $p^{e}(j) = p(glob(e,j))$            $j=1,\cdots,k$
    2.2) Obtain the k-dimensional vector product
            $y^{e} = \overline{H}^{e}\,p^{e}$
    2.3) Accumulate $y^{e}$ into y according to
            $y(glob(e,j)) = y(glob(e,j)) + y^{e}(j)$      $j=1,\cdots,k$
```

The above algorithm is very useful if an iterative solver is to be used for solving the linear system of equations $H\,u = b$ resulting from the finite element approximation. In fact, as noted before, most iterative solvers involve the matrix H only for the computation of its product with some given vectors and this can be done without assembling the global matrix H. Clearly, ALG9 is especially suitable for our systolic system where the generation of the matrices $\overline{H}^{e}$ is pipelined for $e=1,\cdots,m$, and hence allows also for pipelinig the formation of the partial products y^e.

It is widely accepted in parallel processing that one may replicate some parts of the computation or apply an algorithm that may not be efficient for sequential processing, provided that the gain obtained from parallelism justifies the added cost. This may be the case in the computation of the product vector $y = Hp$, where the direct computation does require less work than the application of ALG9. Also it seems obvious that the amount of storage required to store the matrix H is less than that required for storing all of the elemental matrices $\bar{H}^e$, $e=1,\cdots,m$. In order to justify the application of ALG9, we will compare the storage requirement for storing H with that of storing all the H^e, $e=1,...,m$, and the work required to compute Hp directly rather than by ALG9.

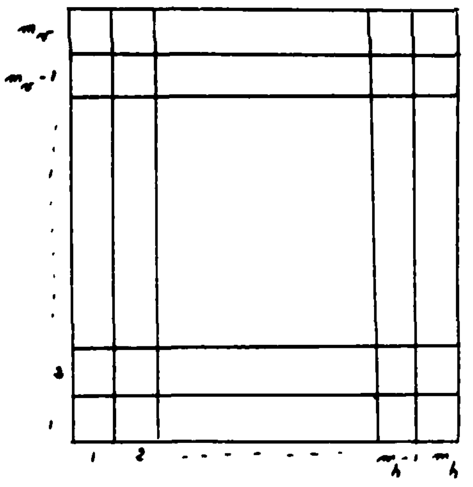


Figure 8.6 - A uniform finite element mesh.

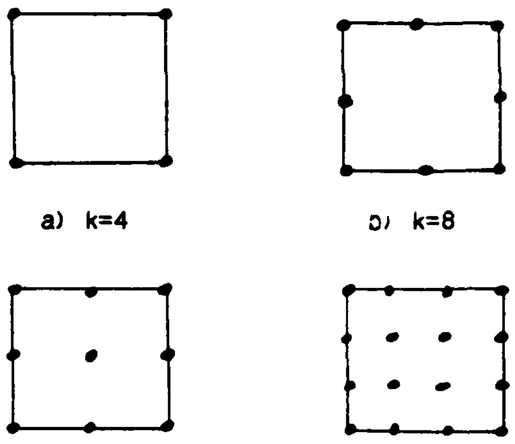


Figure 8.7 - Some quadrilateral element types

As a basis for the comparison, we consider the matrix H corresponding to a uniform finite element mesh over a rectangular domain (see Figure 8.6). The number of elements in the horizontal and vertical directions is assumed to be m_h and m_v, respectively. Hence the total number of elements is $m = m_h\ m_v$. We shall consider four different types of quadrilateral elements (see Figure 8.7), namely: a) four node bilinear elements, b) eight node serendipity elements, c) nine node lagrangian elements, and d) sixteen node lagrangian elements. It is easy to check that the total number of nodes n in the finite element mesh for the four types is a)$(m_h+1)(m_v+1)$, b)$(2m_h+1)(2m_v+1)-m_h m_v$, c)$(2m_h+1)(2m_v+1)$ and d)$(3m_h+1)(3m_v+1)$, respectively.

It should be noted here that H is a banded sparse matrix, and hence that the cheapest way of storing it and operating on its non zero elements is to use a sparse storage mode [18], where the non zero elements of each row of H are stored in consecutive locations in a linear array ELEM. If N is the number of non zero entries in H, then the length of ELEM should be at least equal to N. In addition, two further integer arrays are needed; the first has at least N elements to store the column number of the corresponding entries in ELEM, and the second contains n pointers to ELEM. These pointers specify the position in ELEM of the first element in each of the n rows of H. Hence the minimum storage requirement for H is

$$S_1 = (c_r + c_t)\ N + c_t\ n$$

where c_r and c_t are the cost of storing a real number and an integer, respectively.

On the other hand, each elemental $k \times k$ matrix $\overline{H}^e$ has to be accompanied by an integer vector to indicate the global label of each local node (e,i), $i=1,\cdots,k$. Hence, the total storage for the unassembled matrices is

$$S_2 = m\ k^2\ c_r + m\ k\ c_t$$

Assuming that the cost of storing a real number is double the cost of storing an integer, we obtain the ratio

$$\frac{S_1}{S_2} = \frac{3N + n}{2mk^2 + mk}$$

In order to compare the computational cost for the direct product *Hp* and ALG9, we assume that the costs of a multiplication and an addition are w_m and w_a, respectively. For the direct product, we include the number of operations required to assemble the matrix H (mk^2 additions), and hence obtain

$$W_1 = (w_m + w_a)\ N + m\ k^2\ w_a.$$

For ALG9, this cost is

$$W_2 = (w_m + w_a)\ m\ k^2 + m\ k\ w_a.$$

Neglecting the second terms in W_1 and W_2, we obtain the pessimistic ratio

$$\frac{W_1}{W_2} = \frac{N}{mk^2}$$

In Figure 8.8, we assume that m_h and m_v are much larger than one, and list the estimated formulas for m, n and N as well as the ratios S_1/S_2 and W_1/W_2 for the four types of elements of Figure 8.7, and the matrix H corresponding to the finite element mesh of Figure 8.6.

Element type	k	m	n	N	S_1/S_2	W_1/W_2
a	4	$m_h m_v$	$m_h m_v$	$9\ m_h m_v$	0.777	0.562
b	8	$m_h m_v$	$3\ m_h m_v$	$47\ m_h m_v$	1.059	0.734
c	9	$m_h m_v$	$4\ m_h m_v$	$64\ m_h m_v$	1.146	0.790
d	16	$m_h m_v$	$9\ m_h m_v$	$225\ m_h m_v$	1.295	0.849

Figure 8.8 - Comparison of ALG9 with direct multiplication

The value of the ratio S_1/S_2 indicates that for elements of order higher than the four-node type, the overhead associated with the sparse storage of H makes it cheaper to store the unassembled matrices $\bar{H}^e$, $e=1,\cdots,m$. It also turns out that any banded or profile scheme for storing H will require more storage than the sparse scheme assumed here. On the other hand, the cost of executing ALG9 is always higher than that for the direct multiplication of H and p. However, the ratio W_1/W_2 indicates that the additional work in ALG9 is relatively small, especially for higher order element types. This suggests that we may be willing to pay this price in return for the speed-up and the elimination of memory fetch gained from our pipelined systolic system of the following chapter.

9. POSSIBLE CONFIGURATIONS OF A COMPLETE FINITE ELEMENT SYSTEM.

Our primary goal in this chapter will be to show that the pipelined/systolic approach may be applied to the design of a complete finite element system. We do not intend to specify the details of an ultimate system nor to compare different possible designs. Instead, we will identify the basic functional units in a complete system and then refer to possible implementations for these units, with the emphasis on the interface and interaction between them.

By its very nature, any systolic or pipeline network needs to be monitored by a host computer. In our system, the host is assumed to be a general purpose computer that contains the data base for the problem to be solved. It constitutes also the only means of communication between the user and the system, through which the user specifies or updates the information about the finite element mesh and the partial differential equation and in turn obtains the results of the analysis. The host resembles the heart of our systolic finite element system. It is responsible for setting, initiating and feeding the systolic pipe with the appropriate data as well as for collecting the output data and performing some additional tasks that we will discuss later.

A basic functional unit that should be included in any pipelined finite element system is a unit for the generation of the elemental arrays. Its tasks may be identified as follows: a) generate the arrays H^e and b^e for the elements $e=1,\cdots,m$, b) update H^e and b^e for some elements to force the solution to be equal to zero at some portions of the boundary, c) add the effect of the essential boundary conditions to H^e and b^e, and d) associate with each entry of H^e and b^e the position at which this entry is to be accumulated in the global arrays H and b.

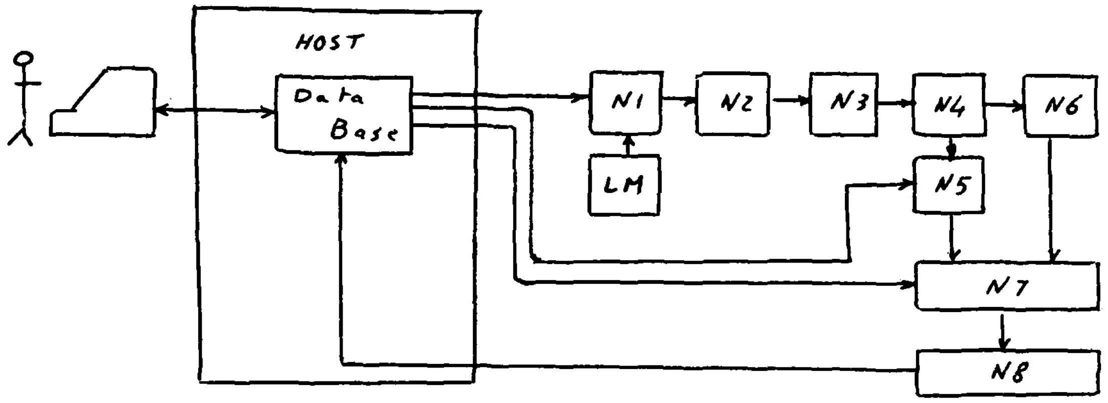


Figure 9.1 - The generation of the elemental arrays

The above tasks may be executed using the systolic networks described in Chapters 7 and 8. In fact, task (a) may be executed on the networks N1···N6 of Chapter 7, tasks (b) and (d) on the network N7 of Section 8.1 and task (c) on the network described in Section 8.2 (call it N8). These networks were designed such that when connected as a pipeline (see Figure 9.1), the output of each sub-network is the input to the next one.

Before initiating the operation of the system, the host should compute the line integrals that account for the essential boundary conditions and load them into the local memories for N8. It should load also the quantities $\nabla_i^0(g)$, $\Delta_i^1(g)$ and $\Delta_i^2(g)$, $i=1,\cdots,k$, $g=1,\cdots,q$ into the local memory LM. Then, the host initiates the operation and pumps the input data for the different finite elements through the system along the proper input links. The form of the inputs may be described by:

$$\zeta_{i,1} = \Omega^{i-1} \, P^{3k}_{e=1,m}(E_1^3 \;\; \theta^2 \;\; \xi_i^e) \qquad i=1,2 \qquad (9.1.a)$$

$$\pi_{9,q+1} = \Omega^{q+3k+9} \, P^{3k}_{e=1,m}(\, P_k^3(\alpha_0^e)\,) \qquad (9.1.b)$$

$$\sigma_{9,q+1} = \Omega^{q+3k+9} \, P^{3k}_{e=1,m}(\, P_k^3(\alpha_1^e)\,) \qquad (9.1.c)$$

$$\rho_{9,q+1} = \Omega^{q+3k+9} P^{3k}_{e=1,m}(\ P^{3}_{k}(\alpha^{e}_{2})\ ) \tag{9.1.d}$$

$$\bar{\rho}_{3k+9,q+1} = \Omega^{q+9k+9} P^{3k}_{e=1,m}(\ \Theta^{2} E^{k}_{1}\ \bar{\varphi}^{e}) \tag{9.1.e}$$

$$\sigma_{0,q+4} = \Omega^{q+6k+16} P^{3k}_{e=1,m}(\ \Theta^{2}\gamma^{e}) \tag{9.1.f}$$

$$\pi_{0,q+4} = \Omega^{q+6k+16} P^{3k}_{e=1,m}(\ \Theta^{2}\gamma^{e}_{0}) \tag{9.1.g}$$

Here, the operator $P^{3k}_{e=1,m}$ indicates that the data for the different finite elements are pipelined with a pipe separation of 3k, and the sequences with superscript e contain the relevant information about the finite element e. For the precise definition of these sequences we refer to the corresponding sections of Chapters 7 and 8. However, we may describe informally the content of these sequences as follows:

* ξ^{e}_{i}, $i=1,2$ contains the coordinates of the nodes in element e (2k data items),

* α^{e}_{i}, $i=0,1,2$ contains the coefficients $a^{e}_{r,l}$, $r,l=0,1,2$ of the bilinear form in element e (9 data items),

* $\bar{\varphi}^{e}$ contains a single data item; the load f^{e},

* γ^{e} contains the k global labels of the nodes in e (k data items), and finally

* γ^{e}_{0} contains k data items that specify the nodes at which the solution must be zero.

Hence, if there are input buffers, the host should supply the systolic pipe with 4k+10 data items every 3k time units. This is a relatively low rate which can be achieved even by a microcomputer. With the help of the abstract systolic model, we were able to prove that if the input to the systolic pipe is as given by (9.1), then the system will produce the elemental arrays on the output links $z_{u,q+6}$ and $b_{u,q+6}$, $u=0,\cdots,k$ of N8 according to the formulas

$$\zeta_{u,q+6} = \Omega^{q+6u+3k+18}\ P^{3k}_{e=1,m}\,(\Theta^2\ \overline{\mu}^e_u) \tag{9.2.a}$$

$$\beta_{u,q+6} = \Omega^{q+6u+3k+18}\ P^{3k}_{e=1,m}\,(\ M^{1,1,1}_1\,(\Theta^2\lambda^e_u\ ,\ \Omega\Theta^2\overline{\lambda}^e_u\ ,\ \delta^*)\ ) \tag{9.2.b}$$

where $\overline{\lambda}^e_k=\delta^*$, $T(\overline{\mu}^e_k)= T(\lambda^e_k)=$ k, $T(\overline{\mu}^e_u)= T(\overline{\lambda}^e_u)= T(\lambda^e_u)= k-u$ for $u=0,\cdots,k-1$, and

$$\overline{\mu}^e_u(t) = \begin{cases} \overline{H}^e_{t,t+u} & u=0,\cdots,k-1 \\ \overline{b}^e_t & u=k \end{cases}$$

$$\lambda^e_u = glob(e,t) \qquad u=0,\cdots,k$$

$$\overline{\lambda}^e_u = glob(e,t+u) - glob(e,t) \qquad u=0,\cdots,k-1$$

In other words, $\overline{\mu}^e_u$ contains the elements of $\overline{H}^e$ and $\overline{b}^e$, and λ^e_u, $\overline{\lambda}^e_u$ contain the addresses where these elements are to be accumulated in the global arrays H and b.

As noted earlier, in order to integrate the system of Figure 9.1 into a complete finite element system, we must distinguish between two types of systems according to the method used for solving the resulting linear system of equations

$$H\ u = b \tag{9.3}$$

These are either systems that employ direct solvers for the solution of (9.3), or systems that use an iterative scheme for completing the analysis. We will consider the systems with the different types of solvers separately.

9.1. Systems that employ direct solvers.

In Figure 9.2, we show a block diagram of a complete system that uses an LU decomposition for solving (9.3). It consists of the host and four functional units. The unit labeled GEN is the generator of the elemental arrays as described earlier in some detail. The output of GEN (described by equations (9.2)) is then directed into the unit labeled ASSEMB. Its function is to assemble the global arrays H and b. The third unit, FACT, receives H and b from ASSEMB and simultaneously performs the LU factorization and produces the solution y of Ly=b.

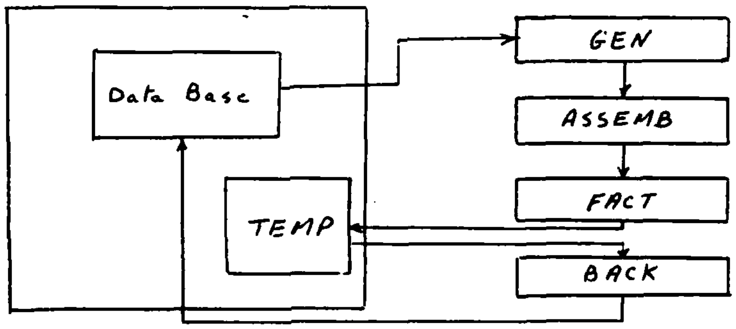


Figure 9.2 - A block diagram of a system with a direct solver

Because of the high rate at which ASSEMB receives its inputs and hence produces the elements of H and b, FACT should have enough computing power to process the data at such a rate. This power may be obtained from a very high speed array processor that may become available in the future as a result of advances in VLSI and optical communication technologies. However, with the current technology, the most suitable candidates for the implementation of FACT are systolic arrays.

Assume that ASSEMB is implemented as B+1 processor/memory units as described in Section 8.4. Hence, each unit P/M_w, $0 \leq w \leq B$, will produce at its output port O_port_w the elements of the w^{th} off-diagonal of the assembled matrix H, in order. We may also use an additional processor/memory unit P/M_{B+1} to assemble the global load vector and produce its elements at the corresponding port O_port_{B+1}.

It had already been noticed in Section 8.3. that there is no uniformity in the rate at which ASSEMB produces the assembled rows of H. For any time t. this rate was denoted by $r_p(t)$. In order to obtain an upper bound on $r_p(t)$. we assume that the nodes of the finite element mesh are numbered by means of algorithm ALG8. Note that the km input rows of ASSEMB are generated by GEN at a uniform rate of one row every three time units. that is. according to the terminology of Section 8.3. we have $r_i = \frac{1}{3}$. Then. by Proposition 8.3. $r_p(t)$ cannot. at any time t. exceed r_i. In other words. we have $r_p(t) < r_i = \frac{1}{3}$. which means that the average rate $\bar{r}_p$ cannot exceed $\frac{1}{3}$. A more precise estimate of $\bar{r}_p$ is obtained by noting that ASSEMB receives km input rows at a uniform rate $r_i = \frac{1}{3}$ and produces n output rows at an average rate $\bar{r}_p$. Hence. $\bar{r}_p = \frac{n}{3km} < \frac{1}{3}$.

With this implementation of ASSEMB. The systolic network of Section 4.3 may be used to implement the functional unit FACT. From now on. we will refer to this network as Sys_FACT. If synchronized by a global clock. Sys-FACT expects to receive its input (the n rows of H) at the rate of one row every three time units. that is the rate $r_c = \frac{1}{3}$ row/time unit. As mentioned earlier. with $r_c > \bar{r}_p$. ASSEMB will not be able to supply Sys-FACT with its inputs on time. and hence. we are forced to implement Sys-FACT as a self timed systolic network rather than as a network synchronized by a global clock. More precisely. if an input cell C_w of Sys-FACT is connected to the processor/memory unit P/M_w in ASSEMB and is expected to receive from it the elements of the w^{th} off-diagonal of H. then. whenever C_w is ready to accept the next input item. it sets the O_interrupt in P/M_w and holds its operation until P/M_w puts the required item on the output port O_port. For further details on the principle of operation of self timed systolic networks. we refer to the discussion in Section 5.5.

In order to estimate the storage requirement in ASSEMB, we note again that $r_c = \frac{1}{3}$ and use the result that $r_p(t) < \frac{1}{3}$ to conclude that the rows of H will be consumed by Sys_FACT as soon as they are produced by ASSEMB (become B_ready). Hence, the only storage needed in ASSEMB is for the $B+1$ active, but not B_ready, rows of H that are being assembled at any one time.

The self timing synchronization of Sys_FACT makes it impossible to predict the time at which every element of the upper triangular matrix U and the partial solution vector y (Ly=b) will be produced on the output links of Sys_FACT. However, we know that Sys_FACT does generate the elements in the last row of U (and y) one time unit after it receives the corresponding elements in the last row of H (and b). Also the last B rows of H, namely $h_{n-B}, \cdots, h_n$, are made available to Sys_FACT just after ASSEMB receives its last input at time $3km+9k+q+16$. Since Sys_FACT is able to consume these B rows in 3B time units, we may conclude that the factorization and partial solution will be completed at time $3km+9k+3B+q+17 \approx 3(km+B)$.

Finally, we discuss the last functional unit in Figure 9.1. This unit, BACK, performs the last step in the analysis, namely the solution of the triangular system $Uu=y$ by back substitution. Although its task is simple, BACK cannot start its computation before the last row of L and the last element of y are available. Hence a temporary storage (TEMP in Figure 9.2) must be provided for storing these elements upon their generation from Sys_FACT until all the element of L and y are generated.

The systolic network for back substitutions described in [31] may be used for BACK. However, we may also use Sys_FACT to perform the back substitution as described in Section 4.3. Although this may be very inefficient. It eliminates the need for any separate hardware for BACK, provided that the entire system will not be used to pipeline the computations for more than one finite element problem. In any case, the back substitution will not require more than 3n time units, and hence the entire analysis will be completed in approximately $3(n+km+B)$ time units, which is a considerable speed up over the time for serially executing the $O(n^2)$ operations estimated in [9]. No comparison can be made with the parallel finite element machine of ICASE [29] because the latter cannot use direct solution schemes.

The system described above profits from all apparent concurrencies in the finite element analysis. However, it has a serious disadvantage, namely the architectures of its units ASSEMB and FACT depend on the bandwidth B of the matrix H, which varies from problem to problem. This disadvantage is shared with most of the known systolic networks that operate on banded matrices [31,7,34]. In order to be able to use a system designed for a certain bandwidth B on a problem with a larger bandwidth $B'>B$, we should be able to partition the computation appropriately to allow its execution on the existing hardware. ASSEMB can easily accommodate a partioning in which each P/M_w performs the computation associated with more than one off-diagonal of H. However, the complex communication pattern of Sys_FACT makes its partioning non-trivial. More research is needed on Sys_FACT or alternate architectures for the direct solution of (9.3) if we desire to have a system that is independent of the bandwidth B. Pipelined finite element systems that employ iterative solution schemes do not appear to share this particular disadvantage.

Remark:

The validity of $\bar{r}_p<r_c$ was based on the assumption that the degree of

freedom d per node is one (see Remark 1 in Chapter 6). This relation may change if d is larger than unity. In order to be more specific, we assume that Remark 3 in Chapter 7 is applied so that GEN generates the d^2 elements of each entry in $\overline{H}^e$ simultaneously. We also assume that ASSEMB is capable of assembling these elements as soon as they are received, and hence that the execution time of the assembly stage remains equal to $3km$ time units. In this case, the nd rows of the global matrix H are produced by the assembler at the rate $\overline{r}_p = \frac{nd}{3km}$ rows/time unit. On the other hand, the rate at which Sys-FACT can consume the B_ready rows remains $r_c = \frac{1}{3}$, and hence the ratio $\frac{\overline{r}_p}{r_c} = \frac{nd}{km}$, is larger than one except for small values of d (see e.g. Figure 8.8). This causes the storage requirement for the assembly process to increase without restrictions.

In order to limit the size of the storage requirement, we have to slow down the rate at which the elemental arrays are generated. This is possible by using one of the following two techniques:

1) A modification of the control parameters in GEN as described in Remark 2 of Chapter 7. Now, each elemental array is generated in ck time units *rather than* $3k$ *time units, where* c *may be chosen such that* $\frac{\overline{r}_p}{r_c} = \frac{3nd}{ckm} < 1$.

2) The use of a self timed technique for the synchronization of the systolic system GEN. With this, a fixed storage may be used in ASSEMB and whenever this storage is fully utilized, ASSEMB stops consuming the output of GEN, thus forcing it to a temporary halt until Sys-FACT consumes some of the rows in ASSEMB. This alternative is preferred as it adjusts the rate $\overline{r}_p$ automatically and efficiently. In this case, Sys_FACT becomes the bottle

neck of the system and hence the time for the completion of the entire computation becomes approximately 6*nd* time units, where 3*nd* units are consumed by GEN and FACT, and the other 3*nd* units by the back substitution step.

9.2. Systems that employ iterative solvers.

Direct solution schemes for the linear system (9.3) do not take advantage of the fact that the stiffness matrix H is highly sparse. This nice property is lost in part during the factorization process, thus missing a potential for savings in both the storage and the execution time. For this reason, it is sometimes beneficial to use iterative schemes for the solution of (9.3) despite their obvious disadvantages, namely, the absence of a good criterion for chosing the initial point and the possible divergence or slow convergence of the iteration.

Many iterative schemes exist for the solution of (9.3). We consider here only two schemes that are widely used in conjunction with the finite element method, namely the conjugate gradient method and the multi-grid technique.

9.2.1. The Conjugate Gradient method.

This method was originally proposed by Hestenes and Stiefel [21]. It finds the solution of the linear system of equations $Hu=b$ by determining the minimum u^* of its gradient functional $g(u)=\frac{1}{2}u^T Hu - b^T u$. The method starts with an initial guess u^0 and obtains a sequence of approximations $u^1, u^2, \cdots$ to u^* iteratively. At the i^{th} iteration step, a new approximation u^{i+1} is obtained from the previous one u^i by the addition of a step size s^i along a suitable direction p^i that reduces $g(u)$. The method may be more specifically described by the following algorithm

ALG10, where $\langle x,y \rangle$ denotes the inner product $x^T y$ and $\| x \|$ is a suitable vector norm, as for example, the infinity norm defined by $\| x \| = \max(x_i ;\ 1 \leq i \leq n)$ for any n-dimensional vector x. The iteration is forced to halt if it does not converge within i_{max} iterations.

ALG10

```
INPUT $u^0$, H, b and an acceptable tolerance $\epsilon$.
1) $r^0 = b - Hu^0$
   FOR $i=0,\cdots,i_{max}$ DO
   2.1) $\alpha^i = \langle r^i, r^i \rangle$
   2.2) IF (i=0) THEN $p^0 = r^0$
                 ELSE $p^i = r^i + \frac{\alpha^i}{\alpha^{i-1}} p^{i-1}$
   2.3) Compute the vector $y = Hp^i$ and the scalar $\beta = \langle p^i, y \rangle$
   2.4) $s^i = \frac{\alpha^i}{\beta}$
   2.5) $u^{i+1} = u^i + s^i p^i$
   2.6) $r^{i+1} = r^i - s^i y$
   2.7) IF($\| r^{i+1} \| < \epsilon$) THEN exit the loop and consider $u^* = u^{i+1}$.
```

Note that step 2.7 in ALG10 may be replaced by other stopping criteria and that some tests may be added for the detection of any divergence in the iteration. Note also that most of the work in ALG10 is in steps 1 and 2.3 where a matrix-vector multiplication is performed. The computations in the other steps are simple vector or scalar operations.

The convergence properties of the method may be improved by a technique called 'preconditioning', where the solution of Hu=b is obtained by solving another linear system $\hat{H}\hat{u}=\hat{b}$ that converges faster than the original one. The transformation between H, u and b and $\hat{H}$, $\hat{u}$ and $\hat{b}$ usually is simple and relatively straightforward. For a detailed description of the preconditioning techniques we refer to [12].

The second method that may be used for solving (9.3) is the multi-grid method.

9.2.2. The Multi-Grid method.

The basic philosophy of this method [38,6] is that, in an iterative scheme, the amount of computation at each step should be proportional to the gain obtained from it. In order to be more specific, we denote the finite element grid (mesh) by G_0 and the corresponding stiffness matrix and load vector by H_0 and b_0, respectively. Also, let $u^0, u^1, \cdots$ be the sequence of approximate solutions of $H_0 u = b_0$ generated by a given iterative scheme.

At the first few steps of the iteration, the residual $r^i = b_0 - H_0 u^i$ decreases rapidly from one iterate to the next, but soon after, the convergence rate levels off and becomes very slow. Closer examination [6] of the Fourier expansion of the residual (the error) shows that the convergence is fast as long as the residuals have strong fluctuations on the scale of the the grid G_0, and that this rate slows down when the residuals are smoothed out. At this point, it is more beneficial to reduce G_0 into a coarser grid G_1 and continue the computation on G_1. This has two advantages, namely 1) the relative fluctuation of the error will increase when measured with respect to the coarse grid G_1, thus speeding up the process of eliminating the error components that were decreasing slowly on G_0, and 2) the cost of the iteration steps will decrease due to the reduction of the number of elements and nodes and hence of the size of the system. This idea may be expanded by using a sequence of grids $G_0, G_1, \cdots$ where each grid G_i is coarser than G_{i-1}. Note that, in addition to the application of a specific iterative scheme, the multigrid solution process involves some data transformation between the different grids.

For a more specific outline of the process suppose that a sequence of fine to coarse grids $G_0, G_1, \cdots$ has been given. The number of nodes in each grid G_i is denoted by n_i and hence, any vector defined on G_i is a member of the vector space R^{n_i}. the desired solution u^* of $H_0 u = b_0$ corresponding to the finest grid is obtained from an initial guess u^0 by the recursive application of the following algorithm ALG11, starting with i=0 and $u_0^0 = u^0$.

ALG11

INPUT: A grid G_i, the corresponding matrix H_i and right side b_i and an initial point $u_i^0 \in R^{n_i}$.

1) Use an iterative scheme (e.g. the conjugate gradient scheme) to compute a sequence of approximate solutions $u_i^0, u_i^1, \cdots$. Stop the iteration when the rate of convergence becomes smaller than a certain acceptable value. Let $\hat{u}_i$ be the last obtained approximation.

2) Compute the residual $r_i = b_i - H_i \hat{u}_i$

3) Consider the next grid G_{i+1} and obtain the corresponding residual $r_{i+1} \in R^{n_{i+1}}$ on G_{i+1} by appropriately averaging the components of r_i. Obtain also the corresponding stiffness matrix H_{i+1}.

4) Find the solution of the lower dimensional system $H_{i+1} \Delta_{i+1} = r_{i+1}$; IF $i+1<k$, THEN invoke ALG11 recursively. ELSE, solve $H_k \Delta_k = r_k$ exactly by means of a direct solver.

5) Interpolate Δ_{i+1} back from G_{i+1} to G_i. Denote the interpolated vector by Δ_i. $(\Delta_i \in R^{n_i})$.

6) Set the solution $u^* = \hat{u}_i + \Delta_i$.

The averaging operation in step 3 is taken to be the dual of the interpolation operation of step 5. That is, if we denote by I_i^{i+1} the linear operator used to obtain r^{i+1} from r_i, and by I_{i+1}^i the linear operator used to interpolate Δ_{i+1} to Δ_i, then the two operators are related by $I_{i+1}^i = c\ (I_i^{i+1})^T$, with some constant c.

Finally, we note that the matrix H_{i+1}, corresponding to the grid G_{i+1}, may be obtained either directly by using ALG5 of Chapter 7, or from the relation $H_{i+1} = I_i^{i+1}\ H_i\ I_{i+1}^i$. It can be seen that the two approaches are equivalent.

After having introduced some possible iterative schemes, we describe next their application in the context of a systolic/pipelined finite element system.

9.2.3. An iterative systolic finite element system.

In what follows, we will assume that the iterative scheme used to solve (9.3) involves the matrix H only in the computation of its product with a certain vector. It was shown in Section 8.5 that this product may be formed using the unassembled elemental arrays, thereby eliminating the need for the irregular assembly stage. Moreover, the system described in Chapter 7 pipelines the computation of the elemental arrays $\bar{H}^1, \cdots, \bar{H}^m$. Hence, for a given vector p, the calculation of the partial product vectors $y^1 = \bar{H}^1 p^1, \cdots, y^m = \bar{H}^m p^m$, may also be pipelined, where $p^1, \cdots, p^m$ and $y^1, \cdots, y^m$ are as described in ALG9 of Section 8.5.

The multiplication $y^e = \bar{H}^e p^e$, $e = 1, \cdots, m$ may be performed on the systolic network described in Section 3.1. This network is shown in Figure 9.3 after relabeling its nodes in a manner consistent with the networks that generate the elemental arrays. The additional row of cells shown in the figure is used to prepare the output of GEN in the form required for the

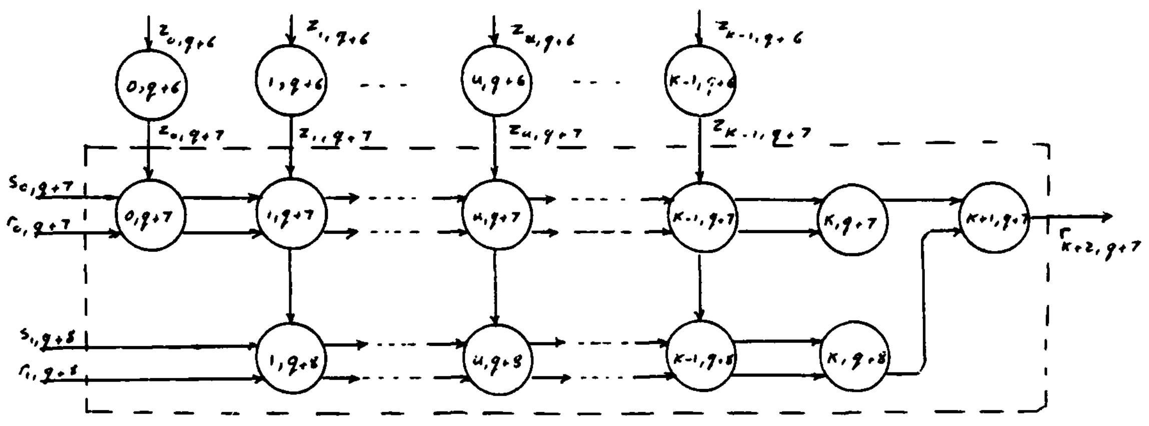


Figure 9.3 - A matrix/vector multiplication network

proper operation of the multiplication network. More descriptively, the input links $z_{u,q+6}$, $u=0,\cdots,k-1$ are connected to the corresponding output links in the subnetwork N8 of Figure 9.1. Hence, the data sequences on these links are described by (9.2.a). However, by comparing (9.2.a) with the formula (3.11.a), given in Section 3.1 for the input of the multiplication network, it is clear that the elements of the u^{th} off-diagonal of the multiplied matrix should be followed by u zeroes when transmitted on the input links of the network. These zeroes are not present in the output of GEN as described in (9.2.a).

In order to insert these zeroes in the data streams, we use the multiplexer cells $(u,q+6)$, $u=0,\cdots,k-1$, to multiplex appropriately the data on the links $z_{u,q+6}$ with the zero sequence ι. The operation of these cells is formally described by

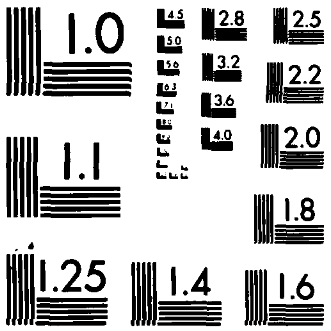
$$\zeta_{u,q+7} = \Omega\ M^{3(k-u),3u}_{q+6u+3k+19}(\zeta_{u,q+6}\ ,\ \iota)$$

With the help of Properties P5 and P15 in Appendix A, it can be shown easily that the data on the links $z_{u,q+7}$ are in the form required for the proper operation of the matrix/vector multiplication network, namely

$$\zeta_{u,q+7} = \Omega^{q+6u+3k+19}\ P^{3k}_{e=1,m}(\Theta^2\ \hat{\mu}^e_u) \qquad u=0,\cdots,k-1 \quad (9.4.a)$$

AD-A140 518 AN ABSTRACT SYSTOLIC MODEL AND ITS APPLICATION TO THE DESIGN OFFFINITE EL..(U) PITTSBURGH UNIV PA INST FOR COMPUTATIONAL MATHEMATICS AND APP.. R G MELHEM 1983 3/3

UNCLASSIFIED ICMA-83-66 N00014-80-C-0455 F/G 12/1 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

where $T(\hat{\mu}_u^e) = k$ and

$$\hat{\mu}_u^e(t) = \begin{cases} \overline{H}^e_{t,t+u} & \text{if } t \leq k-u \\ 0 & \text{if } t > k-u \end{cases}$$

In order to obtain the desired vectors y^e, $e=1,\cdots,m$, the components of the vectors p^e, $e=1,\cdots,m$ must be provided on the input link $r_{0,q+7}$ according to the formula

$$\rho_{0,q+7} = \Omega^{q+3k+19}\ P^{3k}_{e=1,m}(\Theta^2\ \pi^e) \qquad (9.4.b)$$

where $T(\pi^e)=k$ and $\pi^e(t) = p_t^e$ is the t^{th} component of the vector p^e. From ALG9, we have $p_t^e = p(glob(e,t))$. Applying the remark of Section 3.1 with c=3 and using the technique of Section 5.4 for verifying the pipelined operation, we may prove that the output on the link $r_{k+2,q+7}$ is

$$\rho_{k+2,q+7} = \Omega^{q+9k+21}\ P^{3k}_{e=1,m}(\Theta^2\ \eta^e) \qquad (9.5)$$

where $T(\eta^e)=k$ and $\eta^e(t) = y_t^e$. Thus η^e contains the components of the vector $y^e = \overline{H}^e p^e$.

If the host computer collects the outputs from $r_{k+2,q+7}$ and accumulates each element y_t^e in its correct position $y(glob(e,t))$, in one time unit, then the product matrix $y=Hp$ will be available only eight time units after the generation of the elemental arrays is completed.

In an iterative scheme, the formation of the product Hp, for a certain vector p, constitute the major part of each step. The remaining part of each iteration step is less time consuming but depends on the scheme being used. It also requires some intelligent decisions concerning the convergence rate and the stopping criteria. Although it is possible to design special hardware for the completion of some iteration, we believe that it would be more appropriate to assign this task to the host computer.

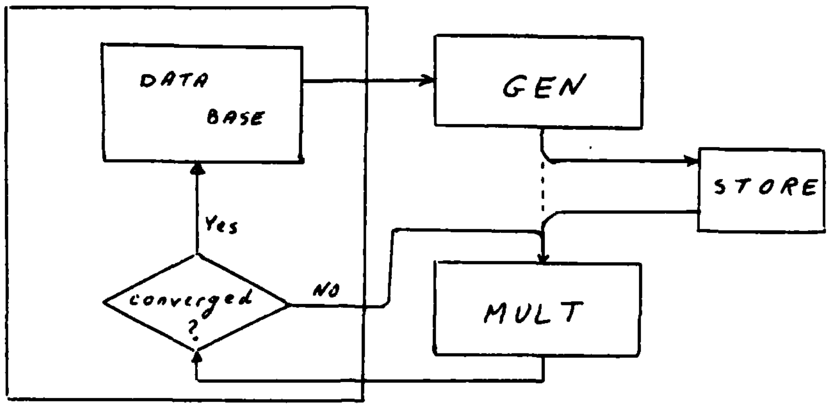


Figure 9.4 - A system that employs iterative solvers

In Figure 9.4, we show a block diagram of a systolic system that employs an iterative scheme for the solution of (9.3). It is composed of a host computer and two systolic functional units; namely GEN for the generation of the elemental arrays and MULT for the matrix/vector multiplication. In this system, the host is more involved in the computation than in the system that employs direct solvers. In fact, the host is a general purpose computer that executes a sequential finite element program and uses the systolic units GEN and MULT as high speed devices to perform some compute-bound operations in the program.

The elemental matrices, $\vec{H}^e$, $e=1,\cdots,m$, are used throughout the iterative solution process. Hence, they may be stored in the auxiliary storage STORE (see Figure 9.4), and repeatedly used in successive steps. Note that the form of the input to MULT described by (9.4) was implied by the assumption that MULT receives its input directly from GEN. However, if the elements of $\vec{H}^e$, $e=1,\cdots,m$ can be fetched from STORE at a rate higher than the one specified by (9.4), then GEN may generate the vectors y^e, $e=1,\cdots,m$ at the same rate. More specifically, if we replace the input sequences (9.4) by

$$\zeta_{u,q+7} = \Omega^{a+2u} \; P^{k}_{e=1,m}(\hat{\mu}^{e}_{u})$$

$$\rho_{0,q+7} = \Omega^{a} \; P^{k}_{e=1,m}(\pi^{e})$$

with some initial delay a, then the output will be described by

$$\rho_{k+2,q+7} = \Omega^{a+2k+2} \; P^{k}_{e=1,m}(\eta^{e})$$

rather than by (9.5). That is we may increase the speed of MULT by a factor of three.

But for use with practical problems, STORE should be a high capacity storage device. By current technology standards, this means that its speed will be relatively low. Hence we may not be able to supply MULT with the needed inputs at a rate faster than the one specified by equation (9.4). In that case, it is more appropriate to eliminate STORE from the system and to use GEN for the regeneration of the elemental arrays at each iterative step. This may seem to be an unnecessary computation. However, by applying this idea, we increase the speed of the system by using a resource that would otherwise be idle.

The idea of regenerating the elemental arrays is even more attractive if a multigrid technique is used for solving (9.3). More specifically, it is clear from ALG11 that the multigrid technique often switches from one grid to another, and in each such switch the global stiffness matrix corresponding to the new grid has to be generated. In that case, the regeneration of the elemental arrays in our system becomes an essential operation rather than a redundant one. Note that the architectures of GEN and MULT neither depend on the specific mesh that covers the problem-domain nor on the bandwidth of the resulting matrix H. Hence, the matrices corresponding to the different meshes may be generated from GEN without any reconfiguration or change in the control parameters.

Finally, we note that the speed-up in the finite element computation achieved by the system sketched in this section is due to many factors, namely: 1) The pipelining of the generation of the elemental arrays. This factor is more prominent if a multigrid technique is used to solve the linear system (9.3), 2) the elimination of the time consuming fetch operations from the slow speed auxiliary storage, 3) the reduction of the time for each iteration step by a factor of k (kd in general, as we will see in the next section), which is the speed-up provided by the systolic network MULT. A general mathematical formula for the overall speed-up provided by the system seems to be impossible to obtain. This is mainly due to the absence of any reasonable criteria for the estimation of the implicit speed up obtained by the smooth flow of data in the system and the elimination of the store/fetch operations of both the data and the instruction. In fact more research needs to be done in order to obtain a good measure for the evaluation of systolic systems.

Next, we consider the decomposition of the multiplication operation $y^e = \overline{H}^e p^e$ in the case of problems with more than one degree of freedom.

9.2.4. The decomposition of the matrix/vector multiplication for d > 1.

In problems with degree of freedom $d>1$, each $(i,j)^{th}$ element $\overline{H}^e_{i,j}$, $1 \leq i,j \leq k$ of the matrix $\overline{H}^e$ is a $d \times d$ sub-matrix. Here, we denote the $(g,l)^{th}$ element of this sub-matrix by $\overline{H}^e_{i,j;g,l}$. Similarly, we denote the g^{th} element of the d-dimensional sub-vectors y^e_i and p^e_i by $y^e_{i;g}$ and $p^e_{i;g}$, respectively.

With this notation, the dk components of the vector $y^e = \overline{H}^e p^e$ are, for $i=1,\cdots,k$ and $g=1,\cdots,d$, given by

$$y^{e}_{i;g} = \sum_{j=1}^{k} \sum_{l=1}^{d} \vec{H}^{e}_{i,j;g,l}\ p^{e}_{j;l} \qquad (9.6)$$

$$= \sum_{l=1}^{d} A^{e}_{i;g,l}$$

where $A^{e}_{i;g,l} = \sum_{j=1}^{k} \vec{H}^{e}_{i,j;g,l}\ p^{e}_{j;l}$.

If the idea of Remark 3 in Chapter 7 is used, then the systolic system GEN may be applied to generate the d^2 elements $\vec{H}^{e}_{i,j;g,l}$ of each $\vec{H}^{e}_{i,j}$ simultaneously, each on a separate output link. Hence, each link $z_{u,q+7}$, $0\leq u\leq k-1$, may be replaced by the d^2 links $z_{u;g,l}$, $g,l=1,\cdots,d$, where the output data on these links are described by:

$$\zeta_{u;g,l} = \Omega^{q+6u+3k+19}\ P^{3k}_{e=1,m}\ (\ \Theta^{2}\ \hat{\mu}^{e}_{u;g,l}\ ) \qquad (9.7.a)$$

with $T(\hat{\mu}^{e}_{u;g,l}) = k$ and

$$\hat{\mu}^{e}_{u;g,l}(t) = \begin{cases} \vec{H}^{e}_{t,t+u;g,l} & \text{if } t\leq k-u \\ 0 & \text{if } t>k-u \end{cases}$$

This is a generalization of the formulas (9.4.a) to the case $d>1$.

Then, the d^2 partial sums $A^{e}_{i;g,l}$, $g,l=1,\cdots,d$ may be generated simultaneously by using d^2 identical copies $MULT_{g,l}$, $g,l=1,\cdots,d$ of the multiplication network MULT. The inputs to each network $MULT_{g,l}$ are the generalized forms of (9.4.a/b), namely (9.7.a) and

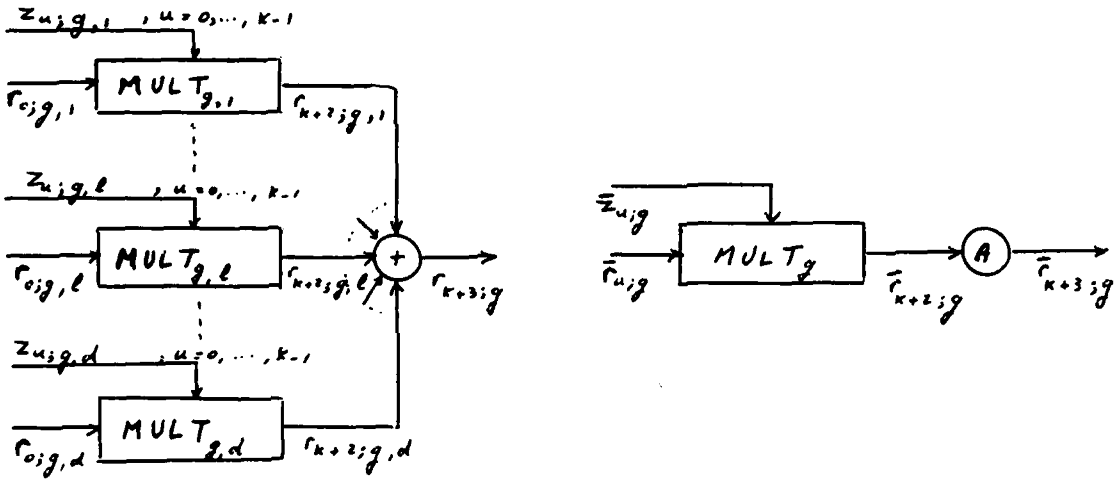
$$\rho_{0;g,l} = \Omega^{9+3k+19}\ P^{3k}_{e=1,m}(\Theta^{2}\ \pi^{e}_{g,l}) \qquad (9.7.b)$$

where, for any g, $1\leq g\leq d$, $T(\pi^{e}_{g,l})=k$ and $\pi^{e}_{g,l}(t)=p^{e}_{t;l}$. With this, it is straight forward to show that the output of $MULT_{g,l}$ is

$$\rho_{k+2;g,l} = \Omega^{q+9k+21}\ P^{3k}_{e=1,m}(\Theta^{2}\ \eta^{e}_{g,l}) \qquad (9.8)$$

where $T(\eta^{e}_{g,l})=k$ and $\eta^{e}_{g,l}(t)=A^{e}_{t:g,l}$. This is the corresponding general form of (9.5).

In order to obtain the components of $y^{e}_{l:g}$, the d output links $r_{k+1:g,l}$, $l=1,\cdots,d$ for a fixed g, $1\leq g\leq d$, are connected to an adder as shown in Figure 9.5(a). Hence, the output of this adder is described by



(a) Non interleaved multiplication (b) Interleaved multiplication

Figure 9.5

$$\rho_{k+3:g} = \Omega^{q+9k+22}\, P^{3k}_{e=1,m}(\Theta^{2}\ \eta^{e}_{g}) \tag{9.9}$$

where $T(\eta^{e}_{g})=k$ and $\eta^{e}_{g}(t)=\sum_{l=1}^{d} A^{e}_{t:g,l} = y^{e}_{t:g}$.

We should note here that each of the d^{2} networks $MULT_{g,l}$ operates once every three time units. However, the efficiency of the system may be improved by interleaving the computation on a reduced subset of these networks. In order to be more specific, we assume that $d\geq 3$ and apply the Remark 2 in Chapter 7 to increase the pipe separation during the operation of GEN from $3k$ to dk. This allows us to use only one link $\bar{z}_{u:g}$ to multiplex the data on the d output link $z_{u:g,l}$, $l=1,\cdots,d$. In terms of data sequences, this means that the equations (9.7.a) are merged into

$$\bar{\zeta}_{u;g} = \Omega^{q+2du+dk+19}\ P^{dk}_{e=1,m}(M_1^{1,\ldots,1}(\ \Theta^{d-1}\ \hat{\mu}^e_{u;g,1}, \cdots,\ \Omega^{d-1}\Theta^{d-1}\ \hat{\mu}^e_{u;g,d})),\ g=1,\cdots,d \text{ and } u=0,\cdots,k-1 \qquad (9.10.a)$$

We may then use only d matrix/vector multiplication networks $MULT_1,\cdots,MULT_d$, and apply to each network $MULT_g$, $1 \leq g \leq d$, the inputs described by (9.10.a) and

$$\bar{\rho}_{0;g} = \Omega^{q+dk+19}\ P^{dk}_{e=1,m}(M_1^{1,\ldots,1}(\ \Theta^{d-1}\ \pi^e_{g,1},\cdots,\Omega^{d-1}\ \Theta^{d-1}\ \pi^e_{g,d}))$$

$$= \Omega^{q+dk+19}\ P^{dk}_{e=1,m}(\bar{\pi}^e_g) \qquad (9.10.b)$$

Here, for any g, we have $T(\bar{\pi}^e_g)=kd$ and $\bar{\pi}^e_g(t)=\ p^e_t$, that is, the t^{th} component of the kd-dimensional vector p^e.

With this input to the network I/O description, it is easy to show that the output of $MULT_g$ is

$$\bar{\rho}_{k+2;g} = \Omega^{q+3dk+21}\ P^{dk}_{e=1,m}(M_1^{1,\ldots,1}(\ \Theta^{d-1}\ \eta^e_{g,1},\cdots,\Omega^{d-1}\ \Theta^{d-1}\ \eta^e_{g,d}))$$

The accumulator shown in Figure 9.5(a) is used to accumulate every d successive items on $\bar{\rho}_{k+2;g}$. The operation of this accumulator is formally described by

$$\bar{\rho}_{k+3;g} = \Omega\ A^{q+22,d,1}\ \bar{\rho}_{k+2;g}$$

which gives

$$\bar{\rho}_{k+3;g} = \Omega^{q+3dk+21+d}\ P^{dk}_{e=1,m}(\Theta^{d-1}\eta^e_g)$$

Thus, we obtain the same results as with the sequence $\rho_{k+2;g}$ of equation (9.9) but at a rate of one result every d time units rather than every three time units. In other words, in order to increase the efficiency, we have to reduce the rate at which the system operates.

10. CONCLUSION.

This dissertation is intended to contribute to the area of computer architectures in two different ways, namely (1) by formalizing the concept of systolic computations, and (2) by providing a basis for the design of a systolic/pipelined system for finite element computations.

The mathematical model suggested for systolic networks provides a method for a clear and precise specification of systolic computations. It also results in a formal technique for the verification of the operation of systolic networks. The central concepts in the abstract model are those of data sequences and sequence operators. Although we only defined the few operators needed in this work, it should be clear that further sequence operators may be introduced to model other types of computational cells.

The computer equation solver, written to supplement the abstract model, is intended to be used in particular for the computational assessment of systolic networks in those cases where the analytical verification is difficult. The application of this solver is equivalent to the simulation of the operation of the given network for specific input data. The syntax directed approach used in the implementation of the solver/simulator led to a very modular program, which simplifies the task of introducing new sequence operators in the future. Actually, the addition of a new operator (a new production rule in the grammar) requires only the implementation of a corresponding semantics routine that describes the effect of the operator.

The potential of the abstract model extends beyond its application to clocked systolic networks. In fact, the discussion in Section 5.5 is a first step toward the application of the model to self-timed systems and the uniform treatment of systolic networks irrespective of the method used for synchronizing the operation of their cells.

Besides its value in demonstrating the power of the systolic model, the design of the finite element system suggested in this dissertation may be particularly useful. In addition to being adequate for VLSI implementations, it has the advantage of being modular in the sense that if the system is designed for a specific number k of nodes in each element and order q of the quadrature formula, it can be easily modified to perform the analysis for different values of k and q.

Moreover, by applying the idea of pipelining the computations for the different elements, we obtained a design that is independent of the domain of the problem and the number of elements in the grid. It should be noted, however, that the LU factorization network used in the system based on a direct solver does depend on the bandwidth of the stiffness matrix, which in turn depends on the finite element grid and on the numbering scheme used for labeling its elements. More research is needed in order to obtain a system that is completely independent of the structure of the finite element grid.

In Chapters 7 and 8, we presented an analytical verification for the design of the different components in the finite element system. The design was also checked using the solver/simulator of Chapter 4. However, due to space limitation, we did not include the details of the simulation in this dissertation.

Finally, we note that it is not easy to define a measure for estimating the efficiency of systolic networks. An intuitive measure would be the quotient $\frac{TP}{C}$, where T is the time needed by a systolic network to complete its computation, P is the number of computational cells in the network, and C is the number of operations to be computed by the network. This measure, however, does not take into consideration the type of operations performed by a cell, which in our case range from simple memory cells to floating point dividers. It also ignores the benefit obtained by the regular movement of the data in the network. In [23] a more elaborate measure was suggested that takes into account the bandwidth of the input and output links of the network in comparing the efficiency of different systolic networks. Both measures estimate the utilization of the computational cells in a network without differentiating between the different types of cells. This is acceptable if all the cells in the network are of the same type. However, if the network contains more than one type of cells, as is the case with our system, we believe that the utilization of each cell should be multiplied by a weight that reflects the hardware complexity of the different cells. More work is needed to develop an efficiency measure of this type.

APPENDIX A

Properties of sequence operators.

In this appendix we list some properties about combinations of the different operators defined in this dissertation. All the properties are directly verifiable from the definition of the operators and are very useful in simplifying any manipulation of the sequence expressions.

Most of the properties take the form "sequence expression = sequence expression". However, some have the form " sequence expression $\rightarrow$ sequence expression", where we formally define the implication operator $\rightarrow$ as follows:

$$IF \text{ for any } t \text{ either } \eta(t)=\xi(t) \text{ or } \eta(t)=\delta \text{ THEN } \xi\rightarrow\eta$$

that is η is equal to ξ after replacing some of its elements by δ. Consequently, if $\xi\rightarrow\eta$, then we may replace ξ by η in any sequence expression as long as δ is treated as a don't care and not as a special symbol, that is in the contest of inert computations. Of course, if $\xi\rightarrow\eta$ and $\eta\rightarrow\xi$ then $\xi=\eta$.

P1) For any element-wise operator 'op' with δ *'op'* $\delta=\delta$ we have

1.1) For $\Gamma = \Omega, \Theta, E$ or P

$$\Gamma(\xi) \text{ 'op' } \Gamma(\eta) = \Gamma(\xi \text{ 'op' } \eta)$$

1.2) $M_r^{w1,\dots,wn}(\xi_1,\cdots,\xi_n) \text{ 'op' } M_r^{w1,\dots,wn}(\eta_1,\cdots,\eta_n) =$

$$M_r^{w1,\dots,wn}([\xi_1 \text{ 'op' } \eta_1],\cdots,[\xi_n \text{ 'op' } \eta_n])$$

1.3) As a direct result of P1.2 we have

$$\xi \text{ 'op' } M_r^{w1,\dots,wn}(\eta_1,\cdots,\eta_n) = M_r^{w1,\dots,wn}([\xi \text{ 'op' } \eta_1],\cdots,[\xi \text{ 'op' } \eta_n])$$

1.4) If, in addition, 'op' is a δ-regular operator then

$$\Omega^r\ \xi\ \text{'op'}\ \eta = \Omega^r\ \zeta$$

where $\quad T(\zeta) = \min(T(\eta)-r, T(\xi))$ and $\zeta(t) = \xi(t)\ \text{'op'}\ \eta(t+r)$

P2) For the scalar multiplication operator '.', it follows that

2.1) For $\Gamma = \Omega, \Theta, E$ or P

$$w\ .\ \Gamma(\xi) = \Gamma(w\ .\ \xi)$$

2.2) $w\ .\ M_r^{w1,\ldots,wn}(\eta_1,\cdots,\eta_n) = M_r^{w1,\ldots,wn}([w\ .\ \eta_1],\cdots,[w\ .\ \eta_n])$

P3) Composition of Ω with itself

3.1) $\Omega\ \Omega\ \xi = \Omega^2\ \xi$

3.2) $\Omega^{-1}\ \Omega\ \xi = \xi$

3.3) $\Omega\ \Omega^{-1}\ \xi = \xi$ if and only if $\xi(1)=\delta$

3.4) $\xi \rightarrow \Omega\ \Omega^{-1}\ \xi$

P4) Composition of Ω with Θ

$$\Theta^r\ \Omega^k\ \xi = \Omega^{(r+1)k}\ \Theta^r\ \xi \qquad \text{for } r \geq 0 \text{ and any } k$$

P5) Composition of Ω with M

5.1) $\Omega^s\ M_r^{w1,\ldots,wn}(\xi_1,\cdots,\xi_n) = M_{r+s}^{w1,\ldots,wn}(\Omega^s\xi_1,\cdots,\Omega^s\xi_n)$

for any r and $s > -r$

5.2) $\Omega\ M_r^{w1,\ldots,wn}(\xi_1,\cdots,\xi_n) = M_r^{w1,\ldots,wn}(\Omega\xi_n\ .\ \Omega\xi_1\ ,\cdots,\ \Omega\xi_{n-1})$

5.3) $\Omega^k\ M_r^{w1,\ldots,wn}(\xi_1,\cdots,\xi_n) = M_r^{w1,\ldots,wn}(\Omega^k\xi_1,\cdots,\Omega^k\xi_n)$

where $k = w_1+\cdots+w_n$

5.4) $M_{r+1}^{w1,\ldots,wn}(\xi_1,\cdots,\xi_n) = \Omega^r\ \Omega^{-r}\ M_{r+1}^{w1,\ldots,wn}(\xi_1,\cdots,\xi_n)$

5.5) $M_1^{w1,\ldots,wn}(\xi_1,\cdots,\xi_i,\cdots,\xi_n) = M_1^{w1,\ldots,wn}(\xi_1,\cdots,\Omega^q\Omega^{-q}\xi_i,\cdots,\xi_n)$

where $q = w_1+\cdots+w_{i-1}$

P6) Composition of Ω with E

6.1) $E_{r+s}^k\ \Omega^s\ \xi = \Omega^s\ E_r^k\ \xi$ for any r and $s > r$

6.2) $E_{r+1}^k\ \xi = \Omega^r\ \Omega^{-r}\ E_{r+1}^k\ \xi$

6.3) $E_r^k\ \Omega^k\ \xi \rightarrow \Omega^k\ E_r^k\ \xi$

P7) Composition of Ω with A

7.1) $A^{r,k,s}\ \Omega^{u}\ \xi = \Omega^{u}\ A^{r-u,k,s}\ \xi$ for $u < r$

7.2) $A^{r+1,k,s}\ \xi = \Omega^{r}\ \Omega^{-r}\ A^{r+1,k,s}\ \xi$

P8) Composition of Ω with P

8.1) $\Omega^{r}\ P^{k}_{e=1,m}(\xi^{e}) \rightarrow P^{k}_{e=1,m}(\Omega^{r}\ \xi^{e})$

8.2) $P^{k}_{e=1,m}(\Omega^{r}\ \xi^{e}) \rightarrow \Omega^{r}\ P^{k}_{e=1,m}(\xi^{e})$ if $T(\xi^{e}) \leqslant k-r$

8.3) $\Omega^{-r}\ P^{k}_{e=1,m}(\xi^{e}) \rightarrow P^{k}_{e=1,m}(\Omega^{-r}\ \xi^{e})$ if $T(\xi^{e}) \leqslant k$

8.4) $P^{k}_{e=1,m}(\Omega^{-r}\ \xi^{e}) \rightarrow \Omega^{-r}\ P^{k}_{e=1,m}(\xi^{e})$ if $\Omega^{r}\ \Omega^{-r}\ \xi^{e} \rightarrow \xi^{e}$

8.5) $P^{k}_{m}(\xi) \rightarrow \Omega^{k}\ P^{k}_{m-1}(\xi)$

P9) Composition of Θ with itself

$$\Theta^{r}\ \Theta^{k}\ \xi = \Theta^{k}\ \Theta^{r}\ \xi = \Theta^{kr+k+r}\ \xi$$

P10) Composition of Θ with E

10.1) $E^{s}_{1}\ \Theta^{s-1}\ \xi \rightarrow \Theta^{s-1}\ \xi$

10.2) $\Theta^{s-1}\ E^{k}_{r+1}\ \xi = E^{sk}_{sr+1}\ \Theta^{s-1}\ \xi$

P11) Composition of Θ with A

$$A^{1,k,s}\ \Theta^{s-1}\ \xi = E^{s}_{1}\ \Theta^{s-1}\ A^{1,k,1}\ \xi$$

P12) Composition of Θ with P

$$P^{sk}_{e=1,m}(\Theta^{s-1}\ \xi^{e}) = \Theta^{s-1}\ P^{k}_{e=1,m}(\xi^{e})$$

P13) Composition of M with itself

13.1) if $\xi_{i} = M^{1,\ldots,1}_{r}(\eta_{1},\cdots,\eta_{n})$ then

$$M^{1,\ldots,1}_{r}(\xi_{1},\cdots,\xi_{i},\cdots,\xi_{n}) = M^{1,\ldots,1}_{r}(\xi_{1},\cdots,\eta_{i},\cdots,\xi_{n})$$

13.2) $M^{k-m,1,\ldots,1}_{r}(M^{k-n,1,\ldots,1}_{r+n}(\zeta,\xi_{1},\cdots,\xi_{n})\ ,\ \eta_{1},\cdots,\eta_{m}) =$

$$M^{k-n-m,1,\ldots,1}_{r+n}(\zeta,\eta_{1},\cdots,\eta_{m},\xi_{1},\cdots,\xi_{n}) \quad \text{for } m+n < k$$

P14) Composition of M with A

14.1) $A^{r,k,s}\ M^{1,\ldots,1}_{1}(\xi_{1},\cdots,\xi_{s}) = A^{r,k,s}\ \xi_{r}$ for $1 \leqslant r \leqslant s$

14.2) $A^{r,k,1}\ M^{1,\ldots,1}_{r}(\xi_{1},\cdots,\xi_{k}) = M^{1,\ldots,1}_{r}(\eta_{1},\cdots,\eta_{k})$

$$\text{where } \eta_{i} = \sum_{u=1}^{i} \Omega^{i-u}\ \xi_{u}$$

P15) Composition of M with P

$$M_1^{w1,\ldots,wn}(P^k_{e=1,m}(\xi_1^e),\cdots,P^k_{e=1,m}(\xi_n^e)) =$$
$$P^k_{e=1,m}(M_1^{w1,\ldots,wn}(\xi_1^e,\cdots,\xi_n^e)) \qquad \text{for } k=w1+\cdots+wn$$

P16) Composition of E with P

16.1) $E_1^k \;\; P^k_{e=1,m}(\xi^e) = P^k_{e=1,m}(E_1^k \;\; \xi^e)$

16.2) $E_r^k \;\; P^k_{e=1,m}(\xi^e) \rightarrow P^k_{e=1,m}(E_r^k \;\; \xi^e)$ for $r>0$

P17) Composition of A with P

$$A^{1,k,1} \;\; P^{nk}_{e=1,m}(\xi^e) = P^{nk}_{e=1,m}(A^{1,k,1} \;\; \xi^e)$$

P18) Other properties involving the multiplication operator '*'

18.1) $E^k_{r+1} \;\; \xi * \eta \rightarrow [\xi](r+1) \; . \; \Omega^r\Omega^{-r} \; \eta$ if $T(\eta) \leq k+r$

18.2) If $T(\xi_r)<kn$ for $r=0,\cdots,n-1$ then

$$\Omega^i M_1^{1,\ldots,1}(\xi_0,\cdots,\xi_{n-1}) * P^n_k(\eta) = \Omega^i M_1^{1,\ldots,1}(\zeta_0,\cdots,\zeta_{n-1})$$

where $\zeta_r = \eta((i\,\square_n\, r)+1) \; . \; \xi_r$, $r=0,1,\cdots,n-1$ and $\square_n$ is the modulo n addition operation on integers.

P19) If η_j j=1,2,....,k are such that $T(\eta_j)=n$, then

$$\sum_{j=1}^{k} \Omega^{j-1} \; \eta_j = \Omega^{k-1} \; \eta$$

where $T(\eta) = n-(k-1)$ and $\eta(t) = \sum_{j=1}^{k} \eta_j(t+k-j)$.

The next result uses the $\oplus$ of Section 2.1:

P20) Let the sequences η_j, j=0,1,....,k satisfy $T(\eta_j) = n-j$, then

$$\eta_0 \oplus \Omega \; \eta_1 \oplus \Omega^2 \; \eta_2 \oplus \cdots \oplus \Omega^k \; \eta_k = \gamma$$

where $T(\gamma) = n$ and

$$\gamma(t) = \begin{cases} \sum_{j=0}^{t-1} \eta_j(t-j) & t=1,2,\ldots,k \\ \\ \sum_{j=0}^{k} \eta_j(t-j) & t=k+1,k+2,\ldots,n \end{cases}$$

Next, we prove some lemmas that we used in the dissertation.

Lemma 1: The difference equation

$$\sigma_{i+1} = \Omega^c \sigma_i + \Delta_i \qquad i=s,s+1,\cdots,k+1 \tag{a.1}$$

has the solution

$$\sigma_r = \Omega^{c(r-s)} \sigma_s + \sum_{j=s}^{r-1} \Omega^{c(j-s)} \Delta_{r-j+s-1} \qquad r=s+1,\cdots,k+1. \tag{a.2}$$

Proof: The proof uses induction on r. Evidently, for i=s in (a.1) we obtain

$$\sigma_{s+1} = \Omega^c \sigma_s + \Delta_s$$

which is identical to (a.2) for r=s+1. Hence assume that for any r=s+1, ... ,k, σ_r is given by (a.2), then from (a.1) it follows that

$$\begin{aligned}
\sigma_{r+1} &= \Omega^c \sigma_r + \Delta_r \\
&= \Omega^c [\Omega^{c(r-s)} \sigma_s + \sum_{j=s}^{r-1} \Omega^{c(j-s)} \Delta_{r-j+s-1}] + \Delta_r \\
&= \Omega^{c(r+1-s)} \sigma_s + \sum_{j=s}^{r-1} \Omega^{c(j+1-s)} \Delta_{r-j+s-1} + \Delta_r \\
&= \Omega^{c(r+1-s)} \sigma_s + \sum_{j=s-1}^{r-1} \Omega^{c(j+1-s)} \Delta_{r-j+s-1} \\
&= \Omega^{c(r+1-s)} \sigma_s + \sum_{j=s}^{r} \Omega^{c(j-s)} \Delta_{r-j+s}
\end{aligned}$$

which proves that σ_{r+1} is also given by (a.2).

Lemma 2: let f_x be the ranking function for the set $X=(x_1, x_2, \ldots x_n)$, as defined in Section 3.3, then

$$\min(\max(x_k , f_x(i-1,k-1)) , f_x(i,k-1)) = f_x(i,k) \tag{a.3}$$

Proof: Let $y_1, \cdots, y_{k-1}$ be the result of sorting $x_1, \cdots, x_{k-1}$ in ascending order, and $z_1, \cdots, z_k$ the corresponding result for $x_1, \cdots, x_k$. Hence, $f_x(i-1,k-1)=y_{i-1}$, $f_x(i,k-1)=y_i$ and $f_x(i,k)=z_i$. Now consider the following cases:

1) If $x_k < y_{i-1} < y_i$ then the left side of (a.3) is

$$\min(\max(x_k , y_{i-1}) , y_i) = \min(y_{i-1} , y_i) = y_{i-1}$$

Since $z_1 , \cdots ,z_k$ are obtained from $y_1 , \cdots ,y_{k-1}$ by inserting x_k in some position before y_{i-1}, we immediately see that $y_{i-1} = z_i$.

2) If $y_{i-1} < x_k < y_i$, then the left side of (a.3) is

$$\min(\max(x_k , y_{i-1}) , y_i) = x_k$$

and in this case it is clear that $x_k = z_i$.

3) If $y_{i-1} < y_i < x_k$, then the left side of (a.3) is equal to y_i, which in turn is equal to z_i because, in this case, x_k is inserted in some position after y_i.

Lemma 3 : The system of difference equations

$$\rho_{i+1} = \Omega^2 M^{1,2}_{s+i}(\iota , [\lambda_i + \rho_i]) \qquad i=r_0 , \cdots ,r_1 \qquad \text{(a.4)}$$

$$\zeta_i = \Omega M^{1,2}_{s+i}([\lambda_i + \rho_i] , \iota) \qquad i=r_0 , \cdots ,r_1 \qquad \text{(a.5)}$$

with the conditions $\rho_{r0} = \iota$ and $s+r_0>3$ has the solution

$$\zeta_i = \begin{cases} \Omega M^{1,2}_{s+i}(\lambda_i , \iota) & i=r_0 \\ \Omega M^{1,2}_{s+i}([\Omega^2\lambda_{i-1} + \lambda_i] , \iota) & i=r_0+1 \\ \Omega M^{1,2}_{s+i}([\Omega^4\lambda_{i-2} + \Omega^2\lambda_{i-1} + \lambda_i] , \iota) & i=r_0+2, \cdots ,r_1 \end{cases}$$

Proof: The cases $i=r_0$ and $r=r_0+1$ are easy to verify by direct substitution. Hence, we will consider only the case $i \geqslant r_0+2$. First, we will prove that the solution of (a.4) is

$$\rho_i = M^{1,2}_{s+i}([\Omega^2\lambda_{i-1} + \Omega^4\lambda_{i-2}] , \delta^*) \qquad i=r_0+2, \cdots ,r_1 \qquad \text{(a.6)}$$

For this, we use (a.4) twice to get

$$\rho_i = \Omega^2\ M^{1,2}_{s+i-1}(\iota\ .\ [\lambda_{i-1} + \rho_{i-1}])$$

$$= \Omega^2\ M^{1,2}_{s+i-1}(\iota\ .\ [\lambda_{i-1} + \Omega^2 M^{1,2}_{s+i-2}(\iota\ .\ \lambda_{i-2}+\rho_{i-2})])$$

By applying Properties P5.1, P5.2 and P1.3, and replacing the sequences non relevent to the computation by δ^*, we obtain

$$\rho_i = \Omega^2\ M^{1,2}_{s+i-1}(\iota\ .\ M^{1,1,1}_{s+i-1}(\delta^*, \delta^*, [\lambda_{i-1} + \Omega^2\lambda_{i-2} + \Omega^2\rho_{i-2}]))$$

$$= \Omega^2\ M^{1,1,1}_{s+i-1}(\delta^*,\ \delta^*,\ [\lambda_{i-1}+\Omega^2\lambda_{i-2}+\Omega^2\rho_{i-2}])$$

$$= M^{1,1,1}_{s+i}([\Omega^2\lambda_{i-1}+\Omega^4\lambda_{i-2}+\Omega^4\rho_{i-2}],\ \delta^*,\ \delta^*) \quad \text{(a.7)}$$

Now, if $i=r_0+2$, then, from the hypothesis, $\rho_{i-2}=\rho_{r0}=\iota$, and thus (a.7) reduces to (a.6). Else, if $i>r_0+2$, then we replace ρ_{i-2} in (a.7) by its value from (a.4). This gives

$$\rho_i = M^{1,2}_{s+i}([\Omega^2\lambda_{i-1} + \Omega^4\lambda_{i-2} + \Omega^6 M^{1,2}_{s+i-3}(\iota\ .\ \lambda_{i-3}+\rho_{i-3})]\ .\ \delta^*)$$

$$= M^{1,2}_{s+i}([\Omega^2\lambda_{i-1} + \Omega^4\lambda_{i-2} + M^{1,2}_{s+i}(\Omega^6\iota, \delta^*)]\ .\ \delta^*)$$

where, again, we replaced $[\lambda_{i-3}+\rho_{i-3}]$ by δ^*. Now, Property P1.3 gives

$$\rho_i = M^{1,2}_{s+i}(\ M^{1,2}_{s+i}([\Omega^2\lambda_{i-1}+\Omega^4\lambda_{i-2} + \Omega^6\iota]\ .\ \delta^*\ .\ \delta^*)$$

$$= M^{1,2}_{s+i}([\Omega^2\lambda_{i-1}+\Omega^4\lambda_{i-2}+\Omega^6\iota]\ .\ \delta^*)$$

which reduces to (a.6) for $s+i>6$, that is $s+r_0>3$.

Next, we substitute (a.6) into (a.5) and apply P1.3 to obtain

$$\zeta_i = \Omega\ M^{1,2}_{s+i}(\ M^{1,2}_{s+i}([\lambda_i+\Omega^2\lambda_{i-1}+\Omega^4\lambda_{i-2}]\ .\ \iota)$$

$$= \Omega\ M^{1,2}_{s+i}([\lambda_i+\Omega^2\lambda_{i-1}+\Omega^4\lambda_{i-2}]\ .\ \iota\ )$$

which is the required formula for the case $i>r_0+2$.

APPENDIX B

The grammar for the SCE language.

Terminal symbols

```
PAR        INDEX  SEQN   FOR    DO    END    MAXTIME  INPUT  OUT
Comma      Semi   Colon  Equal  Plus  Minus  Mult     Div
Lbrak      Rbrak  Lcurl  Rcurl  Lpar  Rpar   Period
O          Z      T      A      E     M      U1       U2
Identifier   Positive_integer   Positive_real
```

Grammar rules

```
1) <prog>            ::= <declare> <in_part> <body> <out_part>
2) <declare>         ::= <par_decl> <index_decl> <seqn_decl>

        /* PARAMETER DECLARATIONS */

3) <par_decl>        ::= PAR <par_list> Semi
4)                   |
5) <par_list>        ::= <par_list> Comma <par_stmt>
6)                   |  <par_stmt>
7) <par_stmt>        ::= Identifier Equal Positive_integer

        /* INDEX DECLARATIONS */

8) <index_decl>      ::= INDEX <i_list> Semi
9)                   |
10) <i_list>         ::= <i_list> Comma Identifier
11)                  |  Identifier

        /* SEQUENCE DECLARATIONS */

12) <seqn_decl>      ::= SEQN <dim_list> Semi
13) <dim_list>       ::= <dim_list> Comma <seqn_dim>
14)                  |  <seqn_dim>
15) <seqn_dim>       ::= Identifier Lcurl <range_list> Rcurl
16)                  |  Identifier
17) <range_list>     ::= <range_list> Comma <range>
18)                  |  <range>
19) <range>          ::= <i_expr> Colon <i_expr>
```

```
        /* INDEX EXPRESSIONS */

20) <i_expr>         ::= <i_expr> Plus <i_term>
21)                  !   <i_expr> Minus <i_term>
22)                  !   <i_term>
23) <i_term>         ::= <i_term> Mult <i_factor>
24)                  !   <i_factor>
25) <i_factor>       ::= <simple_factor>
26)                  !   Minus <simple_factor>
27)                  !   Lpar <i_expr> Rpar
28) <simple_factor> ::= Identifier
29)                  !   Positive_integer

        /* THE BODY OF THE PROGRAM */

30) <body>           ::= <stmt_list> Semi
31) <stmt_list>      ::= <stmt_list> Semi <stmt>
32)                  !   <stmt>
33) <stmt>           ::= <eqn>
34)                  !   <for_stmt>
35) <for_stmt>       ::= FOR <for_spec> <stmt_list> END
36) <for_spec>       ::= Identifier Equal <i_expr> Comma <i_expr> DO
37) <eqn>            ::= <seq_spec> Equal <seq_expr>

        /* SEQUENCE SPECIFICATIONS */

38) <seq_spec>       ::= Identifier Lcurl <indicat_list> Rcurl
39)                  !   Identifier
40) <indicat_list>   ::= <indicat_list> Comma <i_expr>
41)                  !   <i_expr>

        /* ELEMENT WISE OPERATORS ON SEQUENCES */

42) <seq_expr>       ::= <seq_expr> Plus <seq_term>
43)                  !   <seq_expr> Minus <seq_term>
44)                  !   <seq_term>
45) <seq_term>       ::= <seq_term> Mult <s_factor>
46)                  !   <seq_term> Div <s_factor>
47)                  !   <seq_term> U1 <s_factor>
48)                  !   <seq_term> U2 <s_factor>
49)                  !   <s_factor>
50) <s_factor>       ::= Positive_real Period <seq_factor>
51)                  !   <seq_factor>
```

```
        /* OPERATORS DEFINED DIRECTLY ON SEQUENCES */

52) <seq_factor>      ::= <seq_spec>
53)                    |  <simple_op> <seq_factor>
54)                    |  <multiplex_op> Lpar <choice_list> Rpar
55)                    |  A Lcurl <i_expr> Comma <i_expr> Comma
                              <i_expr> Rcurl <seq_spec>
56)                    |  Lbrak <seq_expr> Rbrak


57) <multiplex_op>    ::= M Lcurl <i_expr> Semi <ratio_list> Rcurl
58) <simple_op>       ::= O <op_power>
59)                    |  Z <op_power>
60)                    |  T <op_power>
61)                           |  E Lcurl <i_expr> Comma <i_expr> Rcurl
62) <op_power>        ::= Lcurl <i_expr> Rcurl
63)                    |
64) <choice_list>     ::= <choice_list> Comma <seq_expr>
65)                    |  <seq_expr>
66) <ratio_list>      ::= <ratio_list> Comma <i_expr>
67)                    |  <i_expr>


        /* INPUT SPECIFICATIONS */

68) <in_part>         ::= INPUT Lpar <inp_list> Rpar Semi
69) <inp_list>        ::= <inp_list> Comma <inp_spec>
70)                    |  MAXT Positive_integer
71) <inp_spec>        ::= <seq_spec>
72)                    |  FOR <io_for> <inp_spec>
73) <io_for>          ::= Identifier Equal <i_expr> Comma <i_expr>


        /* OUTPUT SPECIFICATIONS */

74) <out_part>        ::= OUT Lpar <out_list> Rpar Semi
75) <out_list>        ::= <out_list> Comma <out_spec>
76)                    |  <out_spec>
77) <out_spec>        ::= <seq_spec>
78)                    |  FOR <io_for> <out_spec>
```

****

APPENDIX C

The listing of the SCE interpreter program

```
                    /** GLOBAL DECLARATIONS **/
#include <stdio.h>

#define Prog_length  650
#define N_rules      78        /* number of rules in the grammar */
#define N_dums       24        /* rules not requiring any action */
#define N_symbol     20        /* size of the symbol table       */
#define N_bound      20        /* size of the bound table        */
#define N_seqn      129        /* max. number of sequences used  */
#define Maxtime      30        /* upper limit on simulation time */
#define stack_length 40        /* length of working stack        */
#define N_words      ((Maxtime + 1) / 16 ) + 1
#define N_ratios     5         /* max. # of arguments in M operators */
#define N_real       5         /* max. # of real constants used      */

int overflow[8] = { Prog_length, stack_length, N_symbol, N_bound,
                    N_seqn,      Maxtime + 1 , N_ratios, N_real } ;

FILE *fopen() , *fp ;

 /* an array to store the program triples */
 struct { char action ;
          int  value  ;
          int  top    ; } prog[Prog_length] ;
 int  location    ;             /* pointer to prog array */

 /* adjust[i] contains the length of the R.H.S. of grammar rule
    i minus one, which is the adjustment in the top of the stack */
 int  adjust[N_rules] = {-3, -2, -2,  1, -2,  0, -2, -2,  1, -2,
                          0, -2, -2,  0, -3,  0, -2,  0, -2, -2,
                         -2,  0, -2,  0,  0, -1, -2,  0,  0, -1,
                         -2,  0,  0,  0, -3, -5, -2, -3,  0, -2,
                          0, -2, -2,  0, -2, -2, -2, -2,  0, -2,
                          0,  0, -1, -3, -8, -2, -5, -1, -1, -1,
                         -5, -2,  1, -2,  0, -2,  0, -4, -2, -1,
                          0, -2, -4, -4, -2,  0,  0, -2
                        };

 /* dums[] contains the number of the grammar rules not requiring
    any action */
 int  dums[N_dums]    = { 3,  4,  5,  6,  8,  9, 12, 13, 14, 22,
                         24, 25, 29, 31, 32, 33, 34, 44, 49, 51,
                         69, 74, 75, 76
                        };

 int stack[stack_length]    ;                /* dynamic working stack */
 float fstack[stack_length] ;                /* a matching value_stack */

 /* SYMBOL TABLE ; type= S, P or I for seqn., parameter or index, respectively. */
 struct { char type  ;
          int  entry1;
          int  entry2; } sym_tab[N_symbol] ;

 int lbound[N_bound] ;                       /*  lower bound table */
 int ubound[N_bound] ;                       /*  upper bound table */
 int ran_ptr = -1    ;                       /* pointer to bound table */
```

```
/* SEQUENCE STORAGE */
float seq_store[N_seqn][Maxtime+1]        ;  /* sequences storage */
unsigned d_table[N_seqn][N_words]         ;  /* keeps tracks of don't cares*/
int seq_ptr = 0     ;                        /* pointer to sequence store */
int seq_size        ;                        /* actual size of the table  */
int last_computed[N_seqn]                 ;  /* for consistancy checks    */

float r_store[N_real]                     ;  /* storage for real constants */
int   r_ptr = -1                          ;  /* the corresponding pointer  */

int ratio_temp[N_ratios] ;                   /* to store multiplexer ratios */
int ratio_ptr            ;                   /* a pointer to ratio_temp */

int not_done      ;                          /* a loop control variable */


/*************************************************************/
/*                      THE MAIN PROGRAM                    */
main()
{
 char action      ;
 int  value       ;
 int  top = -1    ;                         /* current top of stack */
 int j, looking   ;
 int stage        ;
 int end_stage[5] ;

 end_stage[1]=2 ; end_stage[2]=68 ;
 end_stage[3]=30 ; end_stage[4]=1 ;

 /* open the file that contains the program tuples */
 fp = fopen("out.parse", "r") ;
 for(seq_ptr=0 ; seq_ptr < N_seqn ; seq_ptr++)
    for(j=0 ; j < N_words ; j++)
       d_table[seq_ptr][j] = 0 ;
 seq_ptr = 0 ;

 /* Build standard entries in symbol table, they can be
    overwritten by proper declaration */
 for(j=0 ; j < 3 ; j++)
  { sym_tab[j].type = 'S' ;
    sym_tab[j].entry1 = seq_ptr ;
    last_computed[seq_ptr++] = Maxtime ;
    sym_tab[j].entry2 = -1 ;             /* indicating a single sequence */
  }
 for(j=3 ; j < N_symbols ; j++)
    sym_tab[j].type = ' ' ;
 for(j=1 ; j < Maxtime ; j++)
  { seq_store[1][j] = 0.0 ;
    seq_store[2][j] = 1.0 ; }
 for(j=0 ; j < N_words ; j++)
    d_table[0][j] = 0177777  ;

 for(j=3 ; j < N_seqn ; j++)
    last_computed[j] = 0      ;

/*    END OF INITIALIZATION AND BEGINNIG OF MAIN LOOP    */

 /* Outer for loop: stage=1 -> declarations,
                    stage=2 -> input part,
                    stage=3 -> program body,
                    stage=4 -> output part. */
 for(stage=1 ; stage <= 4 ; ++stage)
 {
  location = 0  ;
  not_done = 1  ;
  /* inner loop 1: read the tuples for the corresponding stage
     from file, keep track of the top of the stack and save in
     prog[] only those triples that require a certain action */
  do
  {
   fscanf( fp , "%c" , &action ) ;
   if(action != 'L') fscanf( fp , "%d\n" , &value ) ;
    else { check(7, ++r_ptr) ;
           fscanf( fp , "%f\n" , &r_store[r_ptr] ) ; }
   switch (action)
```

```
         {
          case 'R' : { looking = 1 ; j = 0 ;                 /* REDUCE */
                      while(looking && j < N_dums)
                        if(value == dums[j++]) looking = 0 ;
                      if(looking) push_triple(action, value, top) ;
                      check(1, (top += adjust[value-1]) );
                      if(value == end_stage[stage]) not_done=0 ;
                      break                                  ;
                     }
          case 'C' : { check(1, ++top)                     ;   /* SHIFT  IDENTIFIER */
                      push_triple(action, value, top );        /* OR INTEGER CONSTANT */
                      break                                ;
                     }
         case 'L' : { check(1,++top) ;                       /* SHIFT REAL CONSTANT */
                       push_triple(action, r_ptr, top) ;
                       break ;
                    }
         case 'S' : { check(1,++top) ;                       /* SHIFT */
                     break          ;
                    }
         case 'A' : { check(1, ++top)               ;        /* ACCEPT */
                       push_triple(action, value, top );
                       break         ;
                     }
        }
       }
       while(not_done) ;

       location = -1 ;
       not_done = 1  ;
       /* inner loop 2: execute the action routines for that stage */
       while(not_done)
       {
        ++location ;
        value = prog[location].value ;
        top   = prog[location].top   ;
        switch( prog[location].action )
        {
          case 'C' : { stack[top] = value ; break ; }
          case 'L' : { fstack[top] = r_store[value] ; break ; }
          case 'R' : { semantic(value, top) ; break ; }
          case 'A' : not_done = 0 ;
        }
       }
      }
      fclose(fp) ;
     }
     /*          END OF THE MAIN PROGRAM      */


     /*******************************************************************/
     /*
     A routine to store a triple in the array prog[]
     */
     push_triple(a , v, t) char a ; int v,t ;
     {
      prog[location].action = a ;
      prog[location].value  = v ;
      prog[location].top    = t ;
      check(0, ++location)      ;
      return(0)                 ;
     }
     /*******************************************************************/
```

```
/****************************************************************************
/*                      THE SEMANTICS ROUTINES
/****************************************************************************

int declaring = 1 ;          /* =1 only during declaration       */
int skip = 0      ;          /* to skip calculation in case of don't cares */
int Mskip= 0      ;          /* to chose the argument in M operator      */
float setfloat()  ;
float tfloat      ;          /* temporary variable                       */
int TIME = 1      ;          /* global system time                       */
int d_flag        ;

semantic(rule, top) int rule,top ;
{
 int t0, t2, reading , j ;

 if(Mskip)
  { switch(rule)
    { case 54 : break ;
      case 64 : --stack[top-4] ;
      case 65 : --Mskip        ;
      default : return(0)      ;
    }
  }

 if(skip)
 { switch(rule)
    { case 53 :
      case 54 : { --skip ; return(0) ; }
      case 57 :
      case 58 :
      case 59 :
      case 60 :
      case 61 : { skip++ ; return(0) ; }
      default : return(0) ;
    }
 }

 switch(rule)
 {
  case 1 : { not_done = 0 ; return(0) ; }      /* signal end of stage 4 */
  case 2 : { declaring= 0 ;                     /* signal end of declarations */
             not_done = 0 ;                     /* that is stage 1          */
             seq_size = seq_ptr - 1 ; return(0) ; }

/*          PARAMETER DECLARATION        */

  case 7 : { t2 = stack[top-2] ; check(2,t2) ;
             if((t2 > 2) && (sym_tab[t2].type != ' ')) run_error(15) ;
             sym_tab[t2].entry1 = stack[top] ;
             sym_tab[t2].type   = 'P'        ;
             return(0) ; }

/*          INDEX DECLARATIONS           */

  case 10:
  case 11: { t0 = stack[top]     ; check(2,t0) ;
             if((t0 > 2) && (sym_tab[t0].type != ' ')) run_error(15) ;
             sym_tab[t0].entry1 = 0           ;
             sym_tab[t0].entry2 = 0           ;
             sym_tab[t0].type   = 'I'         ;
             return(0) ; }

/*          SEQUENCE DECLARATIONS          */

  case 15: { check(3, ++ran_ptr) ;
             lbound[ran_ptr] = 0 ;
             ubound[ran_ptr] = 0 ;
             seq_ptr = seq_ptr + stack[top-1] ;
             check(4, seq_ptr)    ;
             return(0) ; }
  case 16: { t0 = stack[top] ;
             if((t0 > 2) && (sym_tab[t0].type != ' ')) run_error(15) ;
             sym_tab[t0].type   = 'S'       ;
             sym_tab[t0].entry1 = seq_ptr   ;
             sym_tab[t0].entry2 = -1        ;
             check(4,++seq_ptr)             ;
             return(0) ; }
```

```
case 17: { stack[top-2]   = stack[top] * stack[top-2] ;
           return(0) ; }
case 18: { t2 = stack[top-2] ; check(2,t2) ;
           if((t2 > 2) && (sym_tab[t2].type != ' ')) run_error(15) ;
           sym_tab[t2].entry1 = seq_ptr    ;
           sym_tab[t2].entry2 = ran_ptr    ;
           sym_tab[t2].type   = 'S'        ;
           return(0) ; }
case 19: { check(3,++ran_ptr) ;
           ubound[ran_ptr] = stack[top]    ;
           lbound[ran_ptr] = stack[top-2]  ;
           stack[top-2] = stack[top] - stack[top-2] + 1 ;
           return(0) ; }

/*        INDEX EXPRESSION                  */

case 20: { stack[top-2] = stack[top-2] + stack[top] ;
           return(0) ; }
case 21: { stack[top-2] = stack[top-2] - stack[top] ;
           return(0) ; }
case 23: { stack[top-2] *= stack[top] ;
           return(0) ; }
case 26: { stack[top-1] = - stack[top];
           return(0) ; }
case 27: { stack[top-2]= stack[top-1] ;
           return(0) ; }
case 28: { t0 = stack[top] ; check(2,t0) ;
           if(declaring && (sym_tab[t0].type != 'P')) run_error(4) ;
           if((sym_tab[t0].type == 'P') || (sym_tab[t0].type == 'I'))
               { stack[top] = sym_tab[t0].entry1 ; return(0); }
           run_error(5) ; }

/*        PROGRAM'S BODY                    */

case 30: { if(++TIME <= stack[1]) { location = -1 ; return(0) ; }
           printf("\n **** OUTPUT SEQUENCES ****\n") ;
           not_done = 0 ;                  /* signal end of stage 3 */
           return(0) ; }
case 35: { t0 = stack[top-2] ;
           if(sym_tab[t0].entry1 >= sym_tab[t0].entry2) return(0) ;
           ++sym_tab[t0].entry1     ;
           location = stack[top-3] ;
           return(0) ; }
case 36: { t2 = stack[top-5] ; check(2,t2) ;
           if(sym_tab[t2].type != 'I') run_error(6) ;
           sym_tab[t2].entry1 = stack[top-3]        ; /* initial value */
           sym_tab[t2].entry2 = stack[top-1]        ; /* final value */
           stack[top-6] = location ;
           return(0) ; }
case 37: { t2 = stack[top-2] ; check(4,t2) ;
           if(last_computed[t2]++ != TIME-1) run_error(11) ;
           if(stack[top]) { write_d(t2,TIME) ; return(0) ; }
           seq_store[t2][TIME] = fstack[top] ;
           return(0) ; }

/*     SEQUENCE SPECIFICATION                */

case 38: { t2 = stack[top-3] ; t0 = stack[top-1] ;
           if(lbound[++ran_ptr] || ubound[ran_ptr]) run_error(3);
           seq_ptr = sym_tab[t2].entry1 + t0     ;
           stack[top-3] = seq_ptr ;
           return(0) ; }
case 39: { t0 = stack[top] ;
           if(sym_tab[t0].type != 'S') run_error(13) ;
           if(sym_tab[t0].entry2 != -1) run_error(14);
           stack[top] = sym_tab[t0].entry1 ;
           return(0) ; }
case 40: { t0 = stack[top] ;
           ran_ptr++         ;
           if(t0 < lbound[ran_ptr] || t0 > ubound[ran_ptr]) run_error(1) ;
           stack[top-2] = stack[top-2]  * (ubound[ran_ptr] -
                     lbound[ran_ptr] + 1) + (stack[top] - lbound[ran_ptr]) ;
           return(0) ; }
case 41: { t2 = stack[top-2] ; t0 = stack[top] ;
           check(2,t2) ;
           if(sym_tab[t2].type !='S' || sym_tab[t2].entry2 < 0) run_error(12);
           ran_ptr = sym_tab[t2].entry2 ;
           if(t0 < lbound[ran_ptr] || t0 > ubound[ran_ptr]) run_error(1);
           stack[top] = stack[top] - lbound[ran_ptr] ;
           return(0) ; }
```

```
  /* ELEMENT WISE OPERATORS ON SEQUENCES */
   case 42: { if(stack[top] || stack[top-2])
                {stack[top-2]=1 ; return(0) ;}
              fstack[top-2] += fstack[top]  ;
              stack[top-2]   = 0            ;
              return(0) ; }
   case 43: { if(stack[top] || stack[top-2])
                {stack[top-2]=1 ; return(0) ;}
              fstack[top-2] -= fstack[top]  ;
              stack[top-2]   = 0           ;
              return(0) ; }
   case 45: { if(stack[top] || stack[top-2])
                {stack[top-2]=1 ; return(0) ;}
              fstack[top-2] *= fstack[top]  ;
              stack[top-2]   = 0           ;
              return(0) ; }
   case 46: { if(stack[top] || stack[top-2])
                {stack[top-2]=1 ; return(0) ;}
              if(fstack[top]) { fstack[top-2] /= fstack[top] ;
                                stack[top-2]   = 0 ; }
                else stack[top-2] = 1 ;
              return(0) ; }
   case 47: { stack[top-2] = u_op1(fstack[top-2], stack[top-2], fstack[top],
                                   stack[top], &tfloat) ;
              fstack[top-2]= tfloat ;
              return(0) ; }
   case 48: { stack[top-2] = u_op2(fstack[top-2], stack[top-2], fstack[top],
                                   stack[top], &tfloat) ;
              fstack[top-2]= tfloat ;
              return(0) ; }
   case 50: { if(stack[top]) {stack[top-2]=1 ; return(0) ; }
              fstack[top-2] *= fstack[top]   ;
              stack[top-2]   = 0            ;
              return(0) ; }

 /* OPERATORS DEFINED DIRECTLY ON SEQUENCES */

   case 52: { t0 = stack[top] ; check(4,t0) ;
              j  = last_computed[t0]        ;
              if(TIME > j) run_error(10);
              if(read_d(t0,TIME)) { stack[top] = 1 ; return(0) ;}
              fstack[top] = seq_store[t0][TIME] ;
              stack[top]  = 0                  ;
              return(0) ; }
   case 53: { TIME = stack[top-1] ;
              fstack[top-1] = fstack[top]  ;
              stack[top-1]  = stack[top]   ;
              return(0) ; }
   case 54: {  if(stack[top-3]) run_error(7) ;
              Mskip = 0                     ;
              stack[top-3]   = stack[top-1] ;
              fstack[top-3]  = fstack[top-1];
              return(0) ; }
   case 55: { t2 = stack[top-6] ;
              if(TIME < t2) { stack[top-8] = 1 ;
                              fstack[top-8]= 0 ;
                              return(0) ; }
              t2 = (TIME - t2) % (stack[top-2] * stack[top-4]) ;
              t0 = stack[top] ;
              j = last_computed[t0]      ;
              if(TIME > j) run_error(10) ;
              tfloat = 0.0 ;
              for(j=(TIME - t2) ; j <= TIME ; j = j + stack[top-2])
                if(read_d(t0,j)) { stack[top-8] = 1 ;
                                   fstack[top-8]= 0 ;
                                   return(0) ; }
                else tfloat += seq_store[t0][j] ;
              stack[top-8]  = 0 ;
              fstack[top-8] = tfloat ;
              return(0) ; }
   case 56: { fstack[top-2] = fstack[top-1] ;
              stack[top-2]  = stack[top-1]  ;
              return(0) ; }
```

```
    case 57: { t2 = stack[top-3] ; t0 = stack[top-1] ;
               if(TIME < t2) {skip=1 ;
                             stack[top-5]  = 1 ;
                             fstack[top-5] = 0 ; return(0) ; }
               t2 = (TIME - t2) % t0 ;
               for(j=0 ; t2 >= ratio_temp[j] ; j++)    ;
               stack[top-5] = ratio_ptr ;               /* cardinality of list*/
               Mskip = j                ;               /* chosen element     */
               return(0) ; }

    case 58: { t0 = stack[top] ;
               if(TIME <= t0) { skip=1 ;
                               stack[top-1]  = 1 ;
                               fstack[top-1] = 0 ; return(0) ;}
               stack[top-1] = TIME    ;
               TIME = TIME - t0       ;
               return(0) ; }
    case 59: { t0 = stack[top] ;
               if(TIME <= t0) { skip = 1 ;
                               stack[top-1]  = 0 ;
                               fstack[top-1] = 0 ; return(0) ; }
               stack[top-1] = TIME ;
               TIME = TIME -t0     ;
               return(0) ; }
    case 60: { t0 = stack[top] ;
               t2 = (TIME + t0) % (1 + t0) ;
               if(t2) { skip =1 ;              /* don't care */
                       stack[top-1]  = 1  ;
                       fstack[top-1] = 0  ; return(0) ;}
               t2 = (TIME + t0) / (1 + t0) ;
               stack[top-1] = TIME         ;
               TIME  = t2                  ;
               return(0) ; }
    case 61: { t0 = stack[top-1] ; t2 = stack[top-3] ;
               if(TIME < t2) { skip = 1 ;
                             stack[top-5]  = 1 ;
                             fstack[top-5] = 0 ; return(0) ; }
               stack[top-5] = TIME      ;
               TIME = TIME - ((TIME - t2) % t0)      ;
               return(0) ; }
    case 62: { stack[top-2] = stack[top-1] ; return(0) ; }
    case 63: { stack[top+1] = 1 ; return(0) ; }
    case 64: { --stack[top-4] ;
               stack[top-2] = stack[top] ;
               fstack[top-2] = fstack[top] ;
             }
    case 65: { --Mskip ;
             return(0) ; }
    case 66: { check(6, ++ratio_ptr) ;
               stack[top-2] += stack[top] ;
               ratio_temp[ratio_ptr] = stack[top-2] ;
               return(0) ; }
    case 67: { ratio_ptr = 0 ;
               ratio_temp[0] = stack[top] ;
               return(0) ; }

  /*     INPUT SPECIFICATIONS */

    case 68: { not_done = 0      ; /* end of stage 2 */
               return(0) ; }
    case 70: { stack[1] = stack[top] ;
               check(5, stack[top]) ;
               return(0) ; }
    case 71: { t0 = stack[top] ; reading = 1 ;
               for(j=1 ; j <= stack[1] ; j++)
                if(reading)
                { seq_store[t0][j] = getfloat(d_flag) ;
                  if(d_flag == 1)  write_d(t0,j)       ;
                  if(d_flag == -1) { reading = 0 ;
                                     write_d(t0,j) ; }
                }
                else write_d(t0,j) ;
               last_computed[t0] = stack[1] ;
               return(0) ; }
    case 72: { t0 = stack[top-1] ;
               if(sym_tab[t0].entry1 >= sym_tab[t0].entry2) return(0) ;
               ++sym_tab[t0].entry1   ;
               location = stack[top-2] ;
               return(0) ; }
  case 73: { t2 = stack[top-4] ; check(2,t2) ;
             if(sym_tab[t2].type != 'I') run_error(6) ;
             sym_tab[t2].entry1 = stack[top-2] ;
             sym_tab[t2].entry2 = stack[top]   ;
             stack[top-5] = location ;
             return(0) ; }

/*    OUTPUT SPECIFICATIONS */

  case 77: { t0 = stack[top] ;
             for(j=1 ; j <= stack[1] ; j++)
              if(read_d(t0,j)) printf("    d  ") ;
              else printf("%5.2f ", seq_store[t0][j])  ;
             printf("\n ***********************\n") ;
             return(0) ; }
  case 78: { t0 = stack[top-1] ;
             if(sym_tab[t0].entry1 >= sym_tab[t0].entry2) return(0) ;
             ++sym_tab[t0].entry1 ;
             location = stack[top-2] ;
             return(0) ; }
 }
}
/*           END OF THE SEMANTICS ROUTINES          */



/***************************************************/
/*          User Defined Operators                 */
/***************************************************/
/*
These routines are provided by the user to define the
binary operators U1 and U2. The operands are passed in
o1 and o2 and the result is returned in r.
If any of the operands is the don't care symbol, t1 or t2
is set to 1, correspondingly, otherwise they are set to 0.
The return value of the functions should be 0 if the
result of the operation is not a don't care and 1 if it is.
*/
u_op1(o1,t1,o2,t2,r) int t1,t2 ;
                   float o1,o2, *r ;
{
/* Formulas for u_op1 and r */
}

u_op2(o1,t1,o2,t2,r) int t1,t2 ;
                   float o1,o2, *r ;
{
/* Formulas for u_op2 and r */
}

/****************************************************/
```

```
/********************************************************************/
/*
The following routine reads the next item from the
input data file. It assumes that one of the following
exists on the file:
1) a floating point number,
2) a 'd' ; indicating a don't care item,
3) or '...' ; indicating that the remaining items in
the current sequence are don't cares.
The different cases are signaled by the global flag d-flag
*/
float getfloat()
{
 int c, int_part ;
 int minus = 1   ;
 float fraction , p_of_10 ;

 while((c=getchar()) == ' ' || c == '\n' ) ;
 if(c == EOF ) run_error(9) ;
 if(c == 'd' ) { d_flag = 1 ;          /* a don't care symbol */
                 return(0) ;}
 if(c == '.' && (getchar() == '.') && (getchar() == '.') )
               { d_flag = -1 ;         /* sequence terminated */
                 return(0) ;}

 if(c == '-') { c=getchar() ; minus = -1 ;}
 int_part = 0 ;
 while(isdigit(c)) { int_part = (10 * int_part) + (c - '0') ;
                     c = getchar() ; }
 if( c != '.') run_error(8) ;
 fraction = 0.0 ;
 p_of_10 = 10.0 ;
 while(isdigit(c=getchar())) {fraction = fraction + (c - '0') / p_of_10;
                              p_of_10 = p_of_10 * 10.0 ; }

 if((c != ' ') &&  (c != '\n') ) run_error(8) ;
 fraction = minus * ( int_part + fraction ) ;
 d_flag = 0 ;
 return(fraction) ;
}

isdigit(d) int d ;
{
 if(d <= '9' && d >= '0') return(1);
 return(0) ;
}
/*********************************************************************/
/*
The following routines keep track of the position of the
don't care symbols in the data sequences; Each entry in
a sequence has a corresponding bit in the array d_table,
write_d(s,t) sets the bit corresponding to the element t
of the sequence s to 1 indicating a don't care , while
read_d(s,t) returns the value of the bit corresponding
to the element t in sequence s.
*/
write_d(s,t) int s,t ;
{
 int word, bit ;
 unsigned pattern = 1 ;
 word = t / 16 ;
 bit = (t % 16) ;
 pattern = pattern << (15-bit) ;
 d_table[s][word] = (d_table[s][word]) | pattern ;
 return(0) ;
}

read_d(s,t) int s,t ;
{
 int word, bit ;
 unsigned pattern = 1 ;
 word = t / 16 ;
 bit = t % 16 ;
 pattern = pattern << (15-bit) ;
 pattern = pattern & d_table[s][word] ;
 if(pattern) return(1) ;
 return(0) ;
}
```

```
        /**** ERROR ROUTINES ******/

/*
A routine to check the bounds of working arrays, the array
to be checked is determined by the  argument i.
*/
check(i, ptr) int i,ptr ;
{
 if(ptr < overflow[i]) return(1) ;
 switch(i)
 {
  case 0 : { printf("*** program array overflow ***\n") ;
             exit(0) ; }
  case 1 : { printf("*** working stack overflow ***\n") ;
             exit(0) ; }
  case 2 : { printf("*** symbol table overflow  ***\n") ;
             exit(0) ; }
  case 3 : { printf("*** bound table overflow   ***\n") ;
             exit(0) ; }
  case 4 : { printf("*** sequence store overflow **\n") ;
             exit(0) ; }
  case 5 : { printf("*** MAXT should be less than %d\n", Maxtime);
             exit(0) ; }
  case 6 : { printf("*** temporary ratio list overflow **\n");
             exit(0) ; }
  case 7 : { printf("*** real constants storage overflow **\n");
             exit(0) ; }
 }
 exit(0) ;
}

/*****************************************************/
/*
A procedure to print run time error messages and stop
execution. The message to be printed is determined
by the argument i.
*/
run_error(i) int i ;
{
 switch(i)
 {
  case 1 : { printf("** sequence array out of bound \n");
             exit(0) ; }
  case 2 : { printf("** too many array arguments \n") ;
             exit(0) ; }
  case 3 : { printf("** too few  array arguments \n") ;
             exit(0) ; }
  case 4 : { printf("** only parameters may be used in sequ. declaratrion \n");
             exit(0) ; }
  case 5 : { printf("** expecting a par. or an index in seq. specification \n");
             exit(0) ; }
  case 6 : { printf("** FOR variables must be declared as INDEX ") ;
             exit(0) ; }
  case 7 : { printf("** wrong number of arguments in Multiplexer list \n");
             exit(0) ; }
  case 8 : { printf("** Format error in input file **\n") ;
             exit(0) ; }
  case 9 : { printf("** insufficient data in input file **\n");
             exit(0) ; }
  case 10: { printf("** incorrect model or non causal equations **\n");
             exit(0) ; }
  case 11: { printf("** Inconsistent system of equations \n") ;
             printf("** Attempt to overwrite a sequence  \n") ;
             exit(0) ; }
  case 12: { printf("** array of sequences not declared  \n") ;
             exit(0) ; }
  case 13: { printf("** sequence not declared \n") ;
             exit(0) ; }
  case 14: { printf("** missing argument list \n") ;
             exit(0) ; }
  case 15: { printf("** variable already declared \n ") ;
             exit(0) ; }
 }
}

      /******* END OF PROGRAM LISTING  ********/
```

REFERENCES

1. R. A. Adams. "Sobolev Spaces." *Academic Press.* New work (1975).
2. A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley. Reading. Mass. 1977.
3. T. Belytschko. R. Chiapetta. and H. Bartel. "Efficient Large Scale Non-linear Transiant Analysis by Finite Elements." *Int. J. of Numerical Methods in Engineering* **10**. pp.579-596 (1976).
4. S. H. Bokhari. "On the Mapping Problem." *Proc. of the 1979 International Conference on Parallel Processing* (1979).
5. S. H. Bokhari. "MAX: An Algorithm for Finding Maximum in an Array Processor with Global Bus." *Proc. of the 1981 International Conference on Parallel Processing* (1981).
6. A. Brandt. "Multi-Level Adaptive Solutions to Boundary-Value Problems." *Mathematics of Computation* **31**. pp.333-390 (April 1977).
7. R. R. Brent and F. T. Luk. "Computing the Cholesky Factorization using a Systolic Architecture." Technical Report 82-521 (Sept. 1982). Dept. of Computer Science. Cornell University
8. A. W. Burks. *Essays on Cellular Automata*, University of Illinois Press (1970).
9. R. Chandra. S. C. Eisenstat. and M. H. Schultz. "Conjugate Gradiant Methods for partial differential equations." *Proc of AICA; International Symposium on Advances in Computer Methods for Partial Differential Equations* (1975).
10. M. C. Chen and C. A. Mead. "Formal Specification of Concurrent Systems." *USC Workshop on VLSI and Modern Signal Processing* (Nov. 1982).
11. D. Cohen. "Mathematical Approach to Iterative Computational networks." *Proccedings of the Fourth Symposium on Computer Arithmetics.* pp.226-238 (Oct. 1978).
12. P. Concus. G. Golub. and D. O'Leary. "A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations.." Technical Report STAN-CS-76-533 (1976). Computer Science Department. Stanford University.
13. E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices." *Proc. of ACM national conference, New York.* pp.157-172 (1969).
14. J. Deminet. "Experiments with Multiprocessor Algorithms." *IEEE Trans. on Computers* **C.31**(4). pp.278-287 (April 1982).

15. S. Fenves and K. Law. "A Two Step Approach to Finite Element Ordering." Report number R-81-130 (Aug. 1981). Department of Civil Engineering. Carnegie-Mellon University.

16. M. J. Foster. "Syntax Directed Verification of Circuit Functions." *in VLSI Systems and Computations* (1981). Ed. H. T. Kung. B. Sproull and G. Steele. Computer Science Press.

17. E. F. Gehringer. A. K. Jones. and Z. Z. Segall. "The Cm* Testbed." *Computer* **15**(10). pp.40-53 (Oct. 1982).

18. A. George and J. Liu. "Computer Solutions of Large Sparse Positive Definite Systems." *Prentice-Hall Series in Computational Math.* (1981).

19. J. Grefenstette. *Automaton Networks and Parallel Rewriting Systems.* Ph.D. Dissertation. Dept. of Computer Science. University of Pittsburgh. 1980.

20. L. S. Haynes. R. L. Lau. D. P. Siewiorek. and D. W. Mizell. "A Survey of Highly Parallel Computing." *Computer.* pp.9-24 (Jan. 1982).

21. M. R. Hestenes and E. Stiefel. "Method of Conjugate Gradients for Solving Linear Systems.." *J. Res. Nat. Bur. of Standards..* Sect. B. 49:409-436 (1952).

22. E. Horowitz and A. Zorat. "The Binary Tree as Interconnection Network: Application to Multiprocessor Systems and VLSI.." *IEEE Trans. on Computer* **30.** pp.247-253 (April 1981).

23. K. Huang and J. Abraham. "Efficient Parallel Algorithms For Processor Arrays." *Proc. of the 1982 International Conference on Parallel Processing.* pp.271-279.

24. B. M. Irons. "A Frontal Solution Program for Finite Element Analysis.." *Int. J. for Numerical Methods In Engineering.* **2.** pp.5-32 (1970).

25. S. C. Johnson. "Yacc: Yet Another Compiler-Compiler." *Computer Science technical report No. 32, 1975.* Bell Laboratories. Murray Hill. NJ07974.

26. L. Johnsson. U. Weiser. D. Cohen. and A. Davis. "Toward a Formal Treatement of VLSI Arrays." Tecnical Report 4191 (1981). Dept. of Computer Science. California Institute of Technology

27. L. Johnsson and D. Cohen. "A Mathematical Approach to Modeling the Flow of Data and Control in Computational Networks." *in VLSI Systems and Computations* (1981). ed. by H. T. Kung. B. Sproull and G. Steele. Computer Science Press.

28. H. F. Jordan. "A Special Purpose Architecture for Finite Element Analysis." *Proc. of the 1978 International Conference on Parallel Processing.* pp.263-266 (1978).

29. H. F. Jordan. "A Multiprocessor System for Finite Element Structural Analysis." *Computer and Structures* **10.** pp.21-29 (1979).

30. H. F. Jordan. M. Scalabrim. and W. Calvert. "A Comparison of Three Types of Multiprocessor Algorithms." pp. 231-238 in *Proc. of the 1979 International Conference on Parallel Processing* (1979).

31. H. T. Kung and C. E. Leiserson. "Systolic Arrays for VLSI." *in Introduction to VLSI Systems* (1980). ed by Mead C. and Conway L.. Addison_Wesley. Reading Mass.

32. H. T. Kung. Class notes. Fall 1981.

33. H. T. Kung, "Why Systolic Architecture," *Computer Magazine*, pp.37-46 (Jan. 1982).

34. K. H. Law, "Systolic Schemas for Finite Element Methods," R-82-139 (July 1982). Carnegie Institute of Technology, Carnegie-Mellon University.

35. C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Ststems," *Twenty-second Anual Symposium on Foudation of Computer Science*, pp.23-36 (Oct. 1981).

36. J. Mai-tan, N. Sarigul, O. Palusinski, and H. Kamel, "Balanced Array Processor Configuration for Finite Element Analysis," N00014-75-C-0837 (Feb. 1982). University of Arizona

37. Y. Malachi and S. Owicki, "Temporal Specifications of Self-Timed Systems.," *in VLSI Systems and Computations*, (1981). ed. H. T. Kunk, B. Sproull and G. Steele, Computer Science Press.

38. S. F. McCormick and J. W. Ruge, "Multigrid Methods for Variational problems.," *SIAM Journal on Numerical analysis* **19**, pp.924-929 (Oct. 1982).

39. J. Misra and K. M. Chandi, "Proofs of Networks of Processes," *IEEE Trans. on Software Engineering*, pp.417-426 (July 1981).

40. A. K. Noor and S. H. Hartley, "Evaluation of Element Stiffness Matrices on a CDC Star-100 Computer," *Computer and Structures* **9**, pp.151-161 (1978).

41. A. K. Noor and J. J. Lambiotte, "Finite Element Dynamic Analysis on CDC Star-100 Computer," *Computer and Structures* **10**, pp.7-19 (1979).

42. J. T. Oden and J. N. Reddy, "An Introduction to the Mathematical Theory of Finite Elements," *John wiley and sons* (1976).

43. M. Ossefort, "Correctness Proofs of Communicating Processes - Three Illustrative Examples from the Literature," TR-LCS-8201 (Jan. 1982). Department of Computer Science, University of Texas at Austin.

44. W. Pilkey, K. Saczalski, and H. Schaeffer, *Eds. Structural Mechanics Computer Programs, Survey, Assesements and Availability*, University of Virginia Press, Charlottesville, VA. (1974).

45. D. A. Podsiadlo and H. F. Jordan, "Operating System Support for the Finite Element Machine," in *Lecture Notes in Computer Science, CONPAR 81, Conference on Analysis Problem classes and Programing for Parallel Computing*, Springer Verlag (1981).

46. P. M. Prenter, *Splines and Variational Methods*, John-Wiley and sons (1975).

47. W. C. Rheinboldt and C. K. Mesztenyi, "On a Data Structure for Adaptive Finite Mesh Refinements," *ACM Trans. on Mathematical Software*, pp.166-187 (June 1980).

48. N. Sarigul, J. Mai-tan, and Kamel H., "Solution of Nonlinear Structure Problems Using Array Processors," *The FENOMECH 81 conference, Stuttgart* (Aug. 1981).

49. D. Scott and C. Strachey, *Toward a Mathematical Semantics for Computer Languages*, Fox J. editor, Polytechnique Institute of Brooklyn Press, New York, 1971.

50. C. L. Seitz. "System Timing." *in Introduction to VLSI Systems.* (1980). ed. C. Mead and L. Conway. Addison-Wesley. Reading. Mass.

51. G. Strang and G. Fix. *An analysis of the Finite Element Method,* Prentice-Hill (1973).

52. J. Von-Neumann. *Theory of Self Reproducing Automata,* University of Illinois Press (1966).

53. U. Weiser and A. Davis. "Mathematical Representation for VLSI Arrays." Technical Report UUCS-80-111 (Sept. 1980). Department of Computer Science. University of Utah

54. U. Weiser and A. Davis. "A Wavefront Notation Tool for VLSI Array Design." *in VLSI Systems and Computations* (1981). ed. by H. T. Kung. B. Sproull and G. Steele. Computer Science Press.

55. P. Zave and W. C. Rheinboldt. "Design of an Adaptive. Parallel. Finite Element System." *ACM Trans. on Mathematical Software.* pp.1-17 (March 1979).

56. P. Zave and G. Cole. "A Quantitative Evaluation of the Feasibility of a suitable Hardware Architecture for an Adaptive Finite Element System." *ACM Trans. on Mathematical Software* (Sept. 1983).

57. O. C. Zienkiewicz. *The Finite Element Method,* McGraw-Hill (1979). Third edition.