CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

ALGORITHM DESIGN AND AREA-EFFICIENT EMBEDDINGS IN A RECONFIGURABLE SYSTOLIC ARCHITECTURE

by

Jean-Pierre Corriveau

Thesis

submitted to the School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Masters

in

Computer Science

University of Ottawa

May 1984

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGEMENTS

I would like to thank Dr. N. Santoro for his time, patience and precious advice throughout my graduate work. I am also especially grateful to Ken Forsythe and Joan Smith who kept me sane during the most difficult moments of my research.

ABSTRACT

In this thesis, we introduce the concept of a reconfigurable systolic architecture (RSA) which consists of programmable systolic cells. This architecture enforces the "realistic" model introduced by Chazelle and Monier, and offers an economical alternative to custom implementation.

Using the RSA, the designer is able to pipeline systolic algorithms that may require different underlying systolic structures. Each structure specification step consists of a systolic algorithm that activates the structure out of the RSA. Based on these concepts, two distinct problems are investigated: area-efficient embeddings of simple topologies, and a general design methodology for the systolic specification of structures.

We present new embedding algorithms for binary trees. One of these algorithms is shown to completely eliminate the large amount of waste associated with the existing embedding techniques. We also develop a design environment for the definition of systolic algorithms and the specification of systolic structures on the RSA.

CONTENTS

APPENDICES

List of Figures

INTRODUCTION

Overview

The remarkable advance of very large scale integrated (VLSI) circuitry has sparked research into the design of algorithms suitable for direct hardware implementations.

Systolic algorithms have emerged as a feasible solution to this problem: since systolic structures generally consist of a few types of simple processors connected in a regular pattern, they are less expensive to design and implement than more general machines. This advantage is offset by the fact that a particular systolic structure can typically be used only on a narrow set of problems. An attractive alternative is offered by programmable systolic cells, such as CMU's PSC [6], many copies of which can be connected and programmed to implement many systolic algorithms.

Since both components and interconnections are etched in the same medium, VLSI hardware costs can be directly related to the geometric function of area. Several models of computation and their associated complexity parameters have been proposed, but were later questioned by Chazelle and Monier [3,4]. The more "realistic" model they introduced has led to drastic revisions of VLSI complexity in general. Also, the complexity of several problems has been shown to be much higher than previously thought. Yet, a majority of the systolic algorithms currently found in the literature do not take into account these results. Moreover, the current design trends seem to favour a custom implementation approach, which is not cost effective from an industrial viewpoint.

In this thesis, we introduce the concept of a **reconfigurable systolic architecture** (RSA) which consists of programmable systolic cells, inherently enforces Chazelle and Monier's model and offers an economical alternative to custom implementation.

Using the RSA, the designer is able to pipeline (i.e. uninterruptedly execute) systolic algorithms that may require different underlying systolic structures. Each structure specification step is expressed as a systolic algorithm that activates the structure out of the RSA. Based on these concepts, two distinct problems are investigated: area-efficient embeddings of simple topologies, and a general design methodology for the systolic specification of structures.

Simple and regular topologies (i.e. wiring patterns), such as systolic arrays and trees, are required for many applications. In the first part of the thesis, we focus on area-efficient embeddings of binary trees in the RSA. As a first contribution of this thesis, we present new embedding algorithms for binary trees and analyze the complexity of the resulting structures. One of the proposed algorithms is shown to completely eliminate the large amount of waste associated with the existing embedding techniques.

In the second part, we develop a design environment for the definition of systolic algorithms and the specification of systolic structures on the RSA. Within this framework, the designer is able to activate arbitrary structures and to follow the execution of the algorithms that use them. The entire design process can be handled in one of two modes: procedural or graphical. Systolic algorithms to "cut" traditional topologies in an RSA are also discussed.

Organization

Part I of the thesis is composed of chapters 1 through 4. In chapter 1, we first review the fundamental concepts of systolic structures and the different models for VLSI computation; we then introduce the reconfigurable systolic architecture. In chapter 2, we discuss the existing embedding techniques for full binary trees, and analyze their complexity. In chapter 3, we present area-efficient algorithms for embedding binary trees. Finally, in chapter 4, we summarize the obtained results and compare them to the existing bounds.

Part II, which comprises chapters 5 through 8, introduces the Systolic Algorithm Design Environment (SADE). In chapter 5, we present an overview of this environment. In chapter 6, we focus on systolic structure specification; SADE's main activation and archiving routines are illustrated via several examples. In chapter 7, we study systolic algorithm definition. Finally, in chapter 8, we describe SADE's role in a complete VLSI CAD system.

PART I

AREA-EFFICIENT EMBEDDING OF BINARY TREES

1. Systolic Structures and Algorithms

1.1 Introduction

As Leiserson remarks, 'the complexity of integrated-circuit chips being produced today makes it feasible to build inexpensive special-purpose sub-systems that rapidly solve sophisticated problems on behalf of a general-purpose host computer.' [11] Systolic structures and algorithms are particularly well suited from both engineering and mathematical standpoints, for designing such sub-systems.

In this chapter, we focus on the two fundamental concepts used throughout the thesis: Chazelle and Monier's model, and the reconfigurable systolic architecture. In the next subsection, we shall review the basic ideas of systolic structures. An overview of the programmable systolic chip project is given in subsection 1.1.2. In section 1.2, we first discuss the common characteristics of the different models of complexity proposed for VLSI computations, and then study the realistic model introduced by Chazelle and Monier. Lastly, in section 1.3, the definition and motivations for reconfigurable systolic architectures are presented.

1.1.1 Fundamentals of Systolic Structures [8,9,11]

A systolic structure consists of fully synchronized interconnected processing elements called cells, each capable of performing some "simple" operations. Because simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic structure are typically interconnected to form a systolic array or a systolic tree. To achieve performance goals, a systolic structure

is likely to use a large number of cells. These cells must be simple and only of a few types to curtail design and implementation costs. For example, if a systolic structure consisting of many cells, is to be implemented on a single chip, each cell should probably contain only simple logic circuits plus a few words of local memory. On the other hand, for board implementations, each cell could reasonably contain a high-performance arithmetic unit plus a few thousand words of memory.

Information in a systolic structure flows between cells in a pipelined fashion and communication with the outside world occurs only at the "boundary cells". For example, in a linear systolic array, only those cells on the array boundaries may be connected to I/O ports. Systolic structures make multiple use of each input data item so that they can achieve high throughput with modest I/O bandwidths for outside communication. Global data communications, such as broadcast, are avoided in order to obtain modular expansibility of the resulting system.

The term "systolic" comes from the greek word "systole", which means "contraction", and in physiology refers to the contraction of the heart that drives blood through the circulatory system of the body. In a systolic structure, the cells can be considered as hearts that pump multiple streams of data throughout the system. The regular beating of these parallel processors maintains a constant flow of data through the network. Every cell beats on each clock pulse. As a processor pumps data items through itself, it performs an operation which may update some of these items. All operands for a cell's computation arrive simultaneously and two processors that communicate must have a data path between them. The farthest a datum can travel in unit time is from one cell to an adjacent cell. Pure systolic systems

totally avoid long distance or irregular wires for data communications. The only global communication (besides power and ground) may be the system clock. (Self-timed computations are allowed [10].)

1.1.2 The Programmable Systolic Chip

Systolic structures can generally be used only on a narrow set of problems, preventing design costs from being amortized over a large number of units. One way to limit costs is to provide a programmable systolic chip (PSC), many copies of which can be connected and programmed to implement different systolic algorithms. The Department of Computer Science at Carnegie-Mellon University has produced a single-chip microprocessor, CMU's PSC [6], which is used in the implementation of a broad spectrum of systolic algorithms. This chip acts as a building-block for systolic arrays and trees as shown in Figure 1. The PSC project allows fairly complex experimental (custom) architectures to be actually built and tested with a reasonable amount of time and effort.

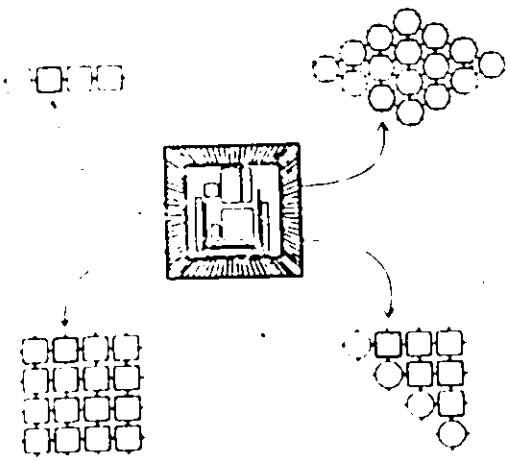


Figure 1. Four different systolic arrays constructed using the PSC.

Many alternatives exist for the implementation of systolic structures at both chip and board level. It is possible to construct a non-programmable building block capable of executing a few predefined, commonly used functions, and then connect them to form a variety of systolic arrays of different sizes and shapes. On the other hand, a programmable building block can implement a large family of systolic cells. Of course the programmable approach is not as efficient as the non-programmable one because of the overheads for supporting programmability. Nevertheless it fullfills the need of implementing the large number of systolic structures which are not frequent enough for warranting individual (expensive), custom hardware implementation. Kung remarks that 'the programmable approach seems to be the only effective way to implement complicated, data-dependent structures' [9.a].

1.2 Models of Complexity [3,4]

The requirement for general models of VLSI computation originates from the need for evaluating and comparing circuit performances, showing lower bounds and trade-offs on area and time, and more generally understanding the nature of the complexity of VLSI computation.

These models must be simple and general enough to allow for mathematical analysis. They must also reflect (the physical) "reality" independently of the size of the circuit. Chazelle and Monier justify the latter requirement by observing that if circuits in the 1980's remain relatively small, the use of high-level languages for designing chips, combined with the possibility of larger integration and chips, will make asymptotic complexity analysis necessary in the near future.

1.2.1 Common Assumptions

All models found in the literature [3,4,10,14] share the following assumptions.

We can think of a systolic structure as an abstraction of an actual VLSI circuit which computes a mathematical function $(y_1,y_2,\ldots) = F(x_1,x_2,\ldots)$. Information is generated digitally, and is communicated to the device through I/O ports in a fixed format. To each variable x_i (resp. y_i) is associated both an input (resp. output) port and a chronological rank in the set of inputs (resp. outputs).

Internally, a systolic structure is a circuit connecting nodes and wires into a directed graph and is defined by a geometrical layout of this graph. The layout must be planar (since the third dimension is used for cooling purposes), meaning that all the nodes are on the same plane. Wires may intersect only at nodes, which all occupy a fixed area. We distinguish I/O ports, where input and output values are available, from the logical nodes of the structure: typically, the I/O ports are quite larger than the logical nodes. The circuit is laid out within a convex region with all the I/O ports lying on its boundary.

The circuits, whose inputs and outputs are available at fixed times (i.e. when-oblivious) and fixed locations (i.e. where-oblivious), are characterized by three parameters: the **area** A, the **time** of computation T, and the **period** P [14].

Given a systolic structure, we define its area to be the number of processors used in the implementation of this structure within a square grid. Also, the time of computation, associated with a systolic algorithm that executes on this structure, is defined as the number of clock pulses required between the occurrence of the algorithm's first input and the appearance of its last output.

In circuits which are when- and where-oblivious, it is possible to pipeline the computations on several sets of inputs, and the period P is defined as the minimal time interval separating two input sets. More precisely, for two sets of N inputs, $(a_1,...a_N)$ and $(b_1,...b_N)$, pipelining the computation means that the time separating the arrival of a_i and b_i is the same for all i; this interval defines the period P. The actual computation of a problem can be viewed as the propagation and logical treatment of information bits across nodes and wires. Each node can produce a result only a certain delay after its input have been available.

1.2.2 A Realistic Complexity Model

The preceding assumptions are common to all the models found in the literature. However, with one exception, all models show inconsistency by failing to account for propagation delays. Namely, they regard a circuit as a topological interconnection of nodes where transmission delays between adjacent nodes can be ignored. However, the laws of physics show that the propagation of information takes time at least proportional to the distance, which invalidates this assumption.

The model by Chazelle and Monier tries to remedy this shortcoming by including all the parameters necessary for performing realistic asymptotic analysis. The most significant departure from the other models comes from their assumption that the time of propagation across a wire is at least proportional to the length of the wire. Since both the resistance and the capacitance of a wire of length L are proportional to L, the time to propagate a signal along this wire is at worst quadratic in L. Therefore the length of wires is crucial for the evaluation of the time performance of a circuit.

The actual computation of a problem can be viewed as the propagation and logical treatment of information bits across cells and wires. The structure being viewed as a directed graph, we associate with each cell a boolean variable $I_i(t)$ (resp. $O_j(t)$) for each incoming (resp. outgoing) wire, which is defined at all times t. These variables represent the information available at the entrance and exit points of a cell. In addition, to each cell corresponds a state S(t) chosen among a finite number of possible states. Then each cell of the structure computes a function F of the form

$$[S(t+\beta),\ldots O_j(t+\beta)]= F[S(t),\ldots I_1(t),\ldots]$$

Informally, this relation means that a cell can produce a result only a delay β after its inputs have been available.

The most drastic departure from previous models, however, comes from the next assumption, which expresses that the time of propagation across a wire is at best proportional to the length of the wire. Let I(t) and O(t) be the variables associated with the ends of a wire of length L.

Chazelle and Monier require that

$$I(t+T)=O(t) \text{ for } T=\Omega(L)$$

Based on the assumptions made by Chazelle and Monier, the concept of optimality becomes meaningful only for large circuits, since the actual complexity of very small chips is overshadowed by parasitic effects.

One major consequence of their model is that the time performance of a circuit is strongly dependent on its geometry (i.e. actual area and configuration) rather than its topology (i.e. wiring pattern). In particular, this implies that all the tree-based schemes previously proposed cease to have their claimed logarithmic time complexity. Examples of such circuits can still be found in Ottman, Rosenberg and Stockmeyer [13], and in Gordon, Koren and Silberman [7]!

Chazelle and Monier proved that addition, cyclic shift, integer product, matrix arithmetic, linear transforms, Discrete Fourier Transforms, sorting and searching all have $O(N^{1/2})$ lower bound on the time T, where N is the size of the problem. (This lower bound comes from the fact that an area of size N in 2 dimensions is bounded by "sides" of length greater or equal to $N^{1/2}$.) These lower bounds are larger than those obtained with the other models and therefore, it becomes important to pipeline computations in order to increase throughput. Furthermore, efficient pipelining can only be performed using algorithms that have a small period P. Consequently, Chazelle and Monier's results directly emphasize the importance of the period P.

1.3 The reconfigurable systolic architecture

Recall that several alternatives exist for the implementation of systolic cells. Among those, the programmable building block approach is the most desirable one if the designer wants to implement a large family of systolic algorithms requiring several different systolic structures [9.a].

Recently, Kung has introduced the concept of a **systolic system on a wafer** [9.b], which offers a very economical, efficient alternative to both chip and board implementations. Informally, the idea is that an entire systolic structure (cells and wires) can be directly etched in a silicon wafer, thus drastically reducing area, communication and control costs. It is possible that the wafer may contain faulty processors; however, in this thesis, we will assume an 'ideal' manufacturing process which produces flawless wafers!

Combining the programmable building block approach with the 'system on a wafer' concept, we propose a **reconfigurable systolic architecture** (RSA) which allows for the execution of different systolic algorithms on different systolic structures, all embedded on the same wafer. The motivations for our investigation of this architecture lie upon two observations:

1. The structures used by most of the existing systolic algorithms are from a very small set of topologies.
2. All these topologies (as well as many other unused ones) are easily "embeddable" in a square (or rectangle) of hexagonally-connected processors, as shown in Figure 2.

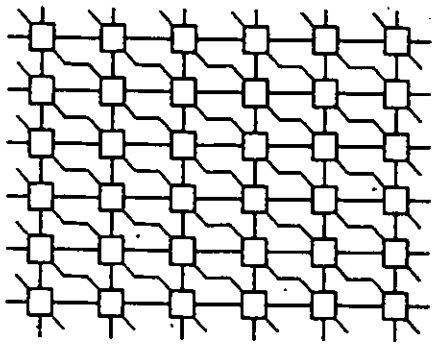


Figure 2. A square of hexagonally-connected processors.

The reconfigurable systolic architecture consists of a square wafer of hexagonally-connected programmable processors, where each processor (i.e. cell) of the RSA has a set of activated links and can execute several different algorithms. In particular, since every cell may run an algorithm to modify its set of activated links, it becomes possible to repeatedly change the systolic structure embedded in the RSA.

When a systolic structure is embedded, some processors, called the relayers, function as links between the active nodes of the structure. Recall that all cells of the RSA are programmable and synchronized. When a cell is used as a relayer, its computational delay (i.e. the delay required to produce the outputs once the inputs are received) can be greatly reduced (by resetting the cell to a 'relayer mode' instead of a programmable mode). However, in the context of the RSA, this is not helpful since all the cells of the RSA are synchronized. Using the RSA, there is a unique fixed clock pulse wich corresponds to the computational delay associated with a programmable cell.

In other words, a relayer takes one clock pulse to compute its outputs from its inputs and therefore, it takes K clock pulses to traverse K relayers.

Several of the design rules reviewed in the preceding section are implicitly enforced by this architecture. In particular, it becomes impossible to implement systolic structures with long or irregular wires. Furthermore, the computation of the area is greatly simplified and standardized. When computing the area of a systolic structure, we must include all the processors in the smallest square (or rectangle) containing the structure. We justify the latter claim by observing that a host computer will probably be connected to several wafers (of possibly different sizes), each consisting of an RSA. After having recognized a problem for which the computer can use a systolic algorithm, the host will decide which size of RSA offers the best per processor utilization's ratio and allocate an available RSA accordingly. An optimal allocation can only be made if the host uses the area computation method we suggested above. (It is interesting to notice that this problem is quite similar to the memory allocation problem for operating systems.)

The concept of wasted processors directly follows from this computation of the area in that a systolic structure may not require all the cells of an RSA thus leaving some processors idle. Area-efficient embeddings must minimize the number of wasted processors.

In the next chapters we will illustrate, via the binary tree layout problem, the drastic revisions that proceed from Chazelle and Monier's model and also from the RSA concept.

2. Systolic Binary Trees and their Embeddings

2.1 Systolic Binary Trees

2.1.1 Introduction

Hierarchical tree-structured multiprocessor systems have recently received a great amount of attention [1,7,10,13]. Systematically, designers invoke two desirable features:

1. the ability to access any processor in a tree of n processors, in at most log n time.
2. the pipelining capability of systolic trees.

To achieve a space economical implementation of binary trees, an appropriate placement algorithm to map the structure on a plane is required. In this chapter, we shall illustrate the drastic simplifications that proceed from Chazelle and Monier's complexity model. In the next two subsections we introduce two simple applications for systolic trees. In section 2.2, we then review the traditional embedding methods.

2.1.2 The Dictionary Machine

In recent years, we have seen much research on how to exploit the new VLSI technology to obtain "hardware" implementations of a large number of computational processes. In particular, several authors have focused on the problem of building a dictionary machine, i.e. a machine that has a capacity of N keys and that will perform the operations SEARCH for a key, INSERT (a new key), and DELETE (a present key). One potential advantage that parallel (in particular, VLSI) implementations have over serial implementations, is the

ability to pipeline a sequence of these instructions.

Ottmann et al. [13] describe a machine that has a period of O(1), a time of O(log n) (where n is the number of keys in the dictionary at the time the operation is performed), and can process redundant insertions and deletions, even in a pipelined fashion.

It is not our intent to analyze this design, but to remark that their claimed logarithmic time complexity does not hold, in view of Chazelle and Monier's results. What must be stressed, is that (as with many systolic structures) there is an inherent trade-off between time and area. For the dictionary machine problem, time prevails over area: given a maximal delay from the root of the tree to any leaf, the designer attempts to produce an area-efficient layout. This maximal delay directly controls the response time associated with each instruction, and thus the ability of the machine to process efficiently pipelined operations. The actual topology of the binary tree and the number of nodes in this tree do not affect the algorithm; they should be established once the layout is completed. In other words, the maximal delay is the a priori criterion that must be enforced in the layout; the two other items are observed a posteriori.

2.1.3 The Priority Queue.

Many programming applications require the ability to insert records into a set and to retrieve from the set the record having the smallest key, according to some linear ordering. Any data structure that provides such services, is called a priority queue. The operation INSERT(Q,a) replaces the set Q with the set Q U {a}. The operation EXTRACTMIN(Q) returns the smallest element a of Q and replaces Q with Q - {a}. Priority queues are usually

implemented in software, but a priority queue can be built in hardware as a systolic array or as a systolic tree [7,8,11].

A priority queue of N cells can be implemented using a systolic tree of depth d = log N. According to the older models, this priority queue can operate within time O(log N) and period O(1). Again, if we apply Chazelle and Monier's results, the structure loses its logarithmic time complexity; in fact, $T = O(N^{1/2})$ becomes the lower bound (i.e. the tree is at best embedded in a square of N cells). For the priority queue problem, the time is not particularly important; the structure must first be designed to allow a period of O(1) and to be as compact as possible. We justify the latter claim by observing that the designer does not care how much time is required to complete an instruction, as long as he is able to pipeline instructions using a constant period. Furthermore, most algorithms for a priority queue will behave identically, whether they work on a full binary tree or on a more irregular tree.

2.2 Existing Embedding Techniques

A complete discussion of the formulas (and of the corresponding derivations) presented in this section can be found in [7.a].

2.2.1 H-type Embedding

Recall that to achieve a space economical implementation of a systolic tree on a VLSI chip, an appropriate placement strategy to map the tree structure on a plane is required. One such strategy, the H-type embedding, which uses an area that is linear in the number of processors, has been widely used [10] and an appropriate construction algorithm, which maps a

full binary tree on a square grid as illustrated in figure 3, has been devised.

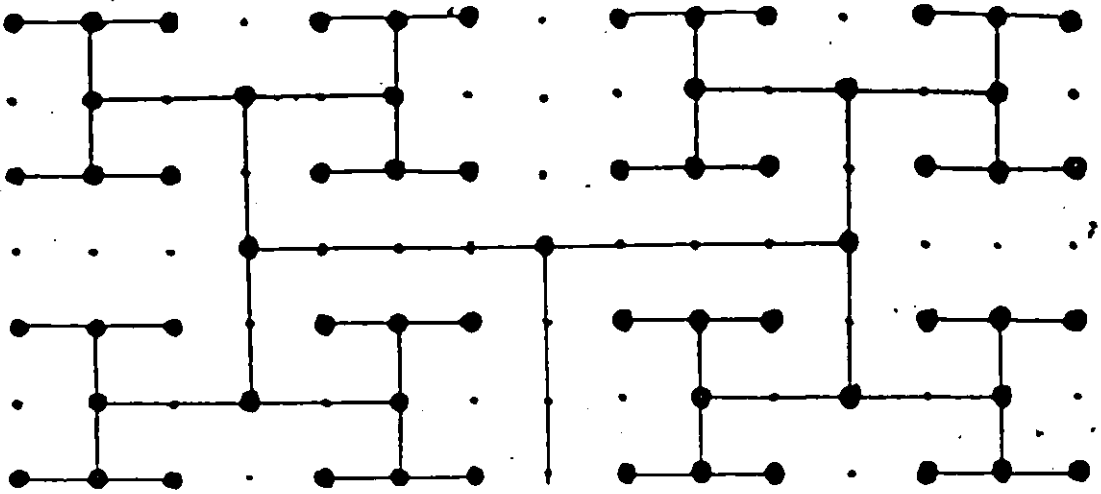


Figure 3. An H-tree of depth 6.

In the usual H-type embedding of a binary tree in a square grid, there is a basic unit consisting of an H-tree of depth 3, which is replicated to form the full tree. When the tree is embedded, some processors, called the relayers, function as links between the active tree nodes. Also, some processors, called the idle processors, are not part of the structure and only play a passive role.

The area, S_K, of the rectangle embedding an H-tree of depth K is given by: [7]

$$S_K = \begin{cases} (2^{(K+1)/2} - 1) * (2^{(K+1)/2} - 1) & \text{if K is odd} \\ (2^{(K+2)/2} - 1) * (2^{K/2} - 1) & \text{if K is even} \end{cases}$$

Furthermore, if we count the number of connections a signal has to cross in order to reach a leaf from the root, then the maximal (propagation) delay R_K is: [7]

$R_K = ((3 * 2^{(K-2)/2} - 2))$ if K is even

$= ((2^{(K+1)/2} - 2) * (2^{K/2} - 1))$ if K is odd

Notice that this formula does not account for the relayers that must be traversed to initially reach the root of the full tree.

2.2.2 The Zigzag Method

Recently, Gordon et al. have investigated the embedding of systolic trees in hexagonal arrays [7]. In this section, we summarize and discuss their results.

Various tree configurations can be mapped on an hexagonal array. The mappings Gordon et al. found to be simplest to replicate have a basic unit consisting of a rectangle (Fig. 4) or a parallelogram (Fig. 5). The basic unit (or "tile"), containing a full binary tree is replicated in one of three different patterns:

1. The replication of the rectangular tile is called the rectangular pattern and the method of replication is obvious from Fig. 6.
2. The parallel replication of a parallelgram basic tile is shown in Figure 7. The resulting overall pattern is a large parallelogram, which may be unsuitable if the chip is not of the same shape.
3. The zigzag replication pattern outlined in Fig. 8, which produces an overall rectangular chip.

In this thesis, we will only consider the last of these methods. We discard the rectangular pattern since it uses a chip which cannot be mapped directly onto a square grid. Not only is this incompatible with the RSA concept,

but it also leads to erroneous computations (with respect to the definition of the area, in subsection 1.2.1) for the area required by the layout. The parallel replication technique has the disadvantage of producing an overall pattern which is a parallelogram; when computing the area of such a layout, we must include all the processors in the smallest rectangle containing it. Instead, Gordon et al. chose to use the smallest parallelogram, which is unacceptable if we insist on a rectangular final layout. The zigzag replication strategy proceeds from the second method, but produces a rectangular final pattern.

Assume we have a parallelogram basic tile which contains a tree of depth c. Let A and B denote the sizes of the sides of this tile. Assume further, that the path to the root of the tile is through a side containing A processors, and that k= c+i (where K is the depth of the targeted tree and k,c and i are positive integers), then the area of the zigzag embedding is: [7]

$$Z_{c+i} = \left[(A+1)2^{\lfloor i/2 \rfloor} + \left\lfloor \frac{B-1}{2} \right\rfloor - 1\right]\left[(B+1)2^{\lfloor i/2 \rfloor + 1} - 1\right], \quad i \geq 0.$$

Furthermore, if we count the number of connections a signal has to cross in order to reach a leaf from the root, then the maximal (propagation) delay is: [7]

$$D_{c+i} = \begin{cases} D_c + 1, & \text{if } i = 1, \\ D_c + 1 + \left\lfloor \frac{A+1}{2} \right\rfloor + (A+1)(2^{\lfloor (i-2)/2 \rfloor} - 1) + (B+1)(2^{\lfloor (i-1)/2 \rfloor} - 1), & \text{if } i \geq 2 \end{cases}$$

where D_c is the internal propagation delay of the basic tile.

2.3 Waste Analysis

As mentioned earlier, in the context of the RSA, the area should be computed as the number of processors contained in the smallest square (or rectangle) that holds a systolic structure; both area formulas given in this chapter, do follow this rule. The zigzag method generates layouts for a full tree of 2^K nodes (where K is a positive integer); the corresponding H-tree has one fewer node. For simplicity, we will consider that an H-tree has 2^K nodes. Thus, for both embedding techniques, the number of wasted processors is given by:

$$W_K = (\text{area} - 2^K)$$

Notice that the relayers used in the structure are included in this amount. For an H-tree of 2^K nodes, the layout contains W_K /2 relayers (i.e. 50% of the total waste). In the case of the zigzag method, the number of relayers obviously depends on the size of the basic tile, but typically does not represent more than 20% of the total waste when using the largest (i.e. the most space-economical) basic tile exhibited by Gordon et al. For both methods, the waste evidently grows exponentially as K grows linearly.

The fundamental question is to know whether or not this waste can be justified. In section 2.2, we observed that typical applications of systolic trees do not specifically require a full binary tree, especially when the time complexity is shown to be $O(N^{1/2})$ (where N is the number of nodes in the tree). Furthermore, for certain problems such as the priority queue, systolic structures which are simpler than trees may become adequate. For example, figure 9 proposes a "comb-like" structure that can implement the priority queue described in subsection 2.2.2 (by having the first row and all the columns

contain keys in increasing order). This suggests that the methods studied in this chapter are indeed inappropriate and therefore, that new area-efficient embedding techniques for binary trees are needed. The next chapter develops such methods.

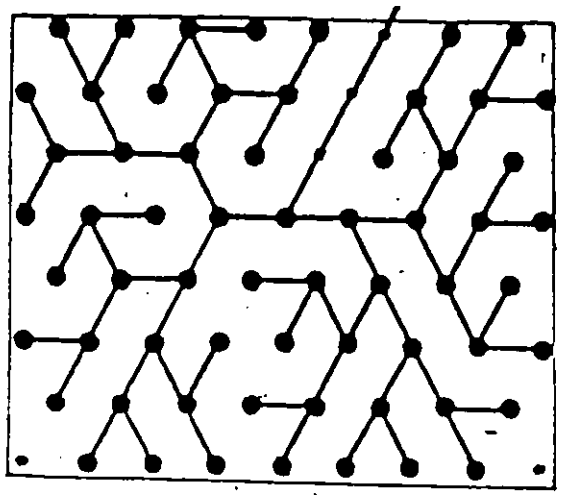


Figure 4. A rectangular tile.

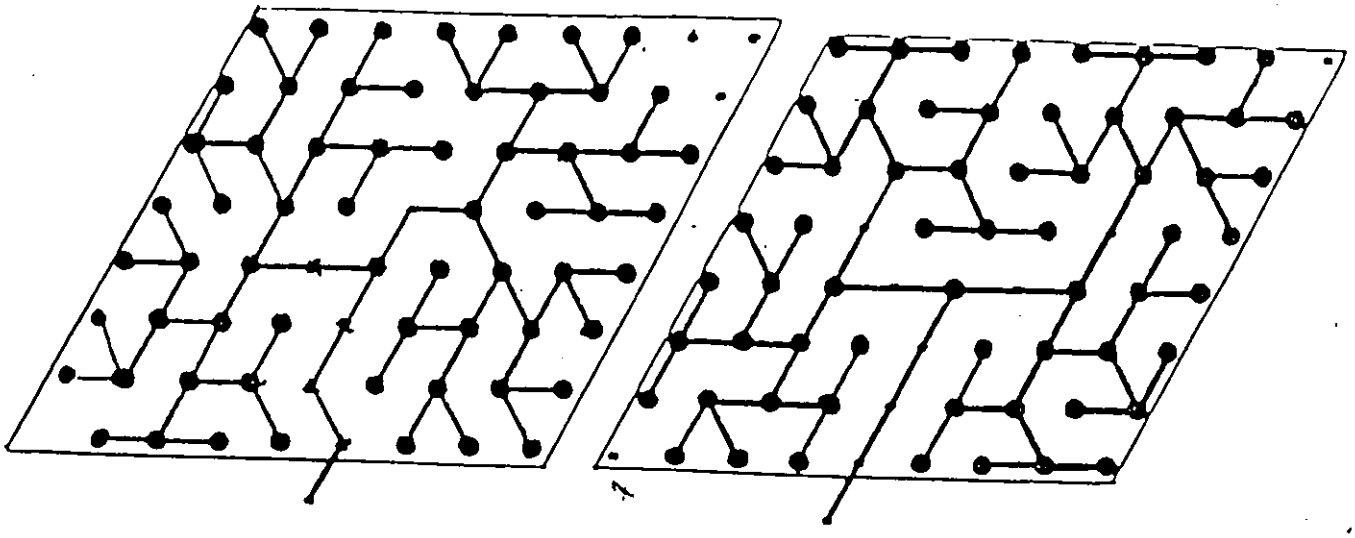


Figure 5. Two parallelogram tiles.

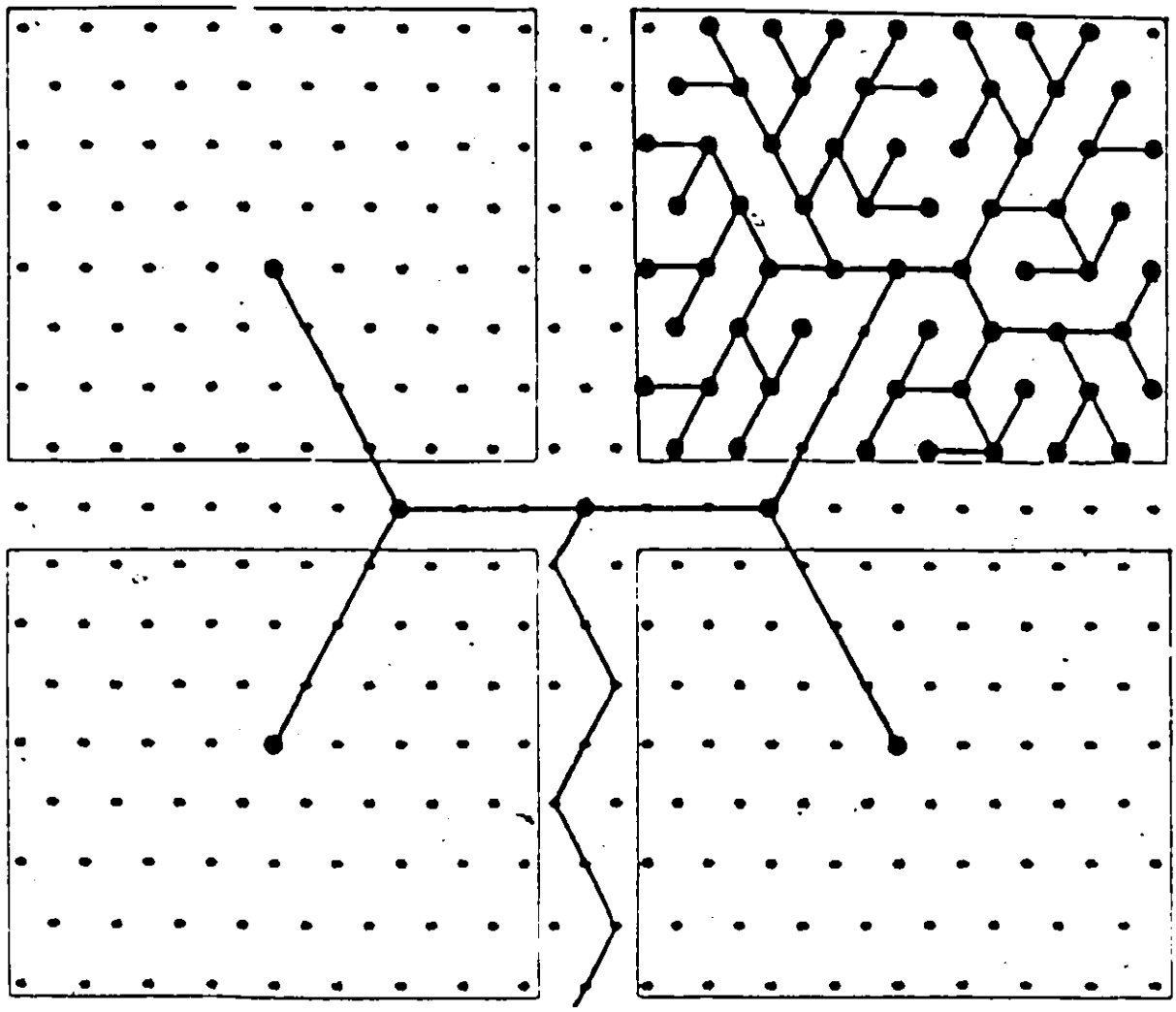


Figure 6. The rectangular pattern.

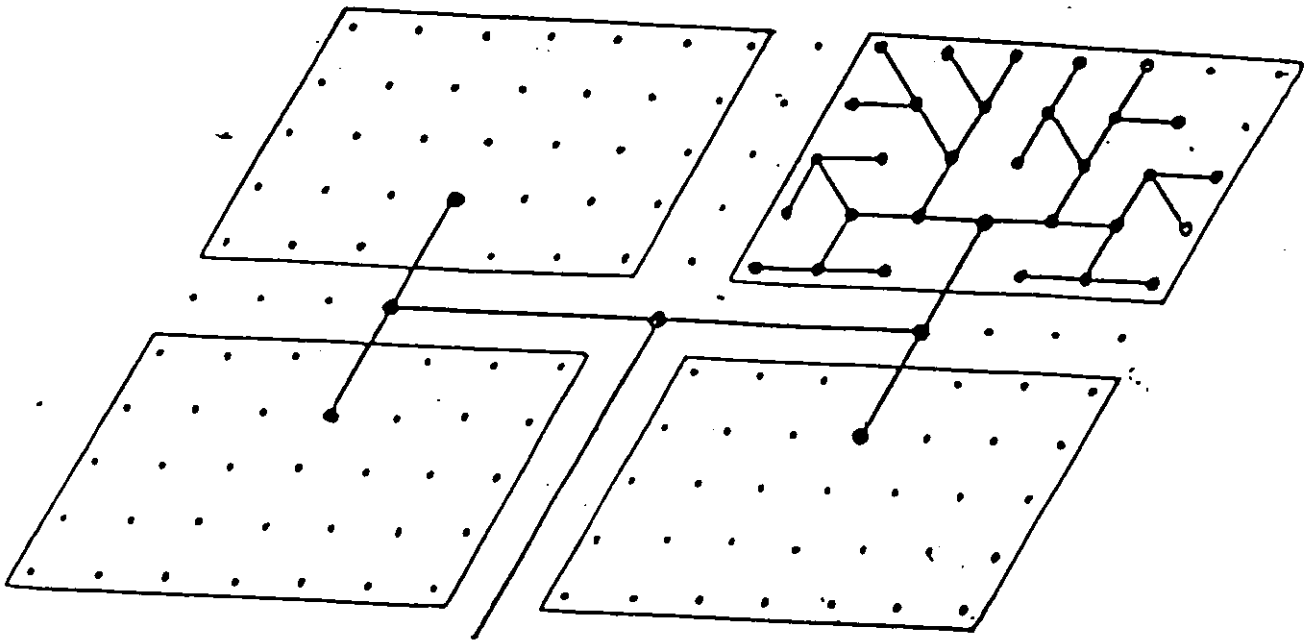


Figure 7. The parallel pattern.

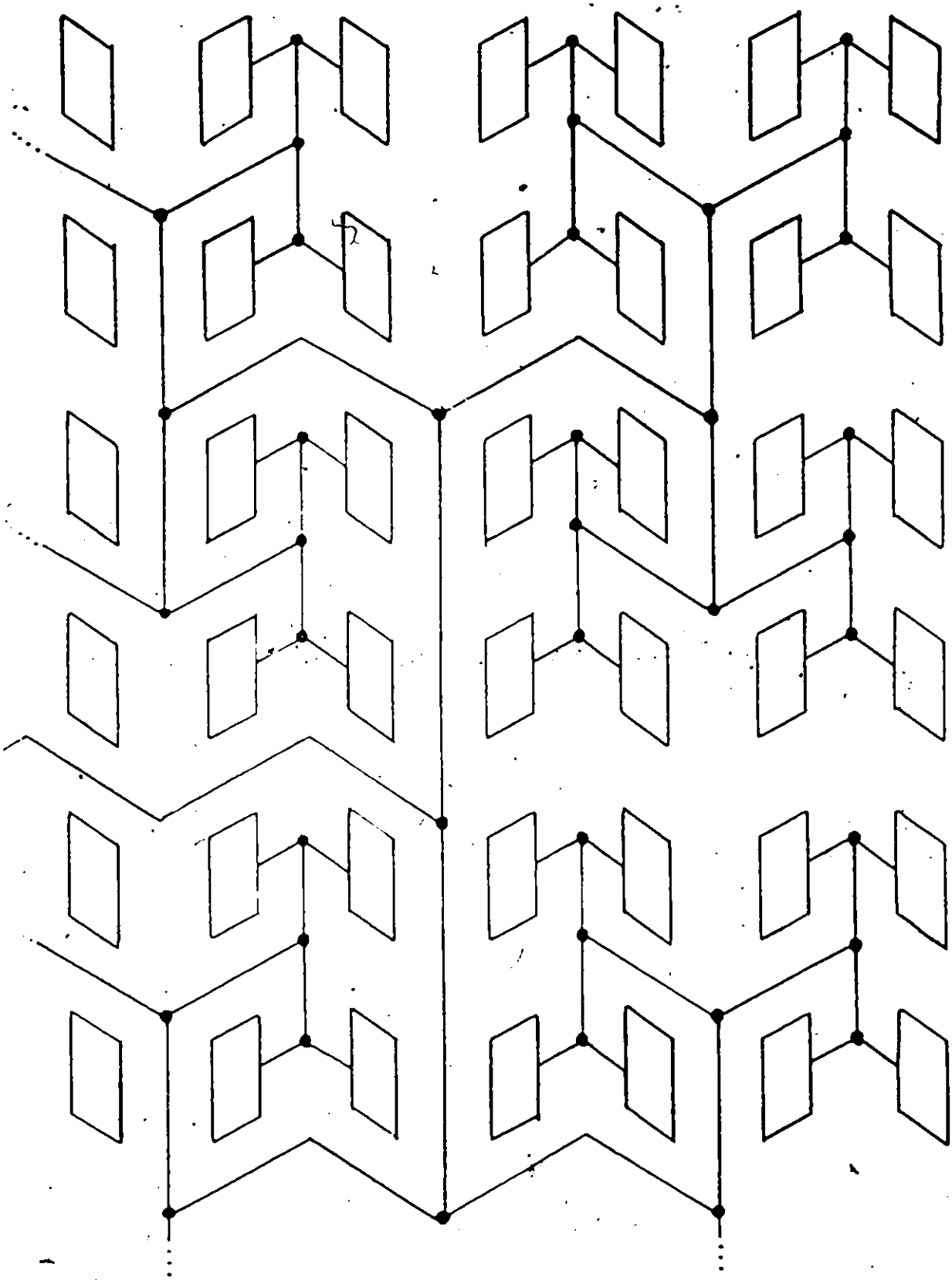


Figure 8. The zigzag pattern.

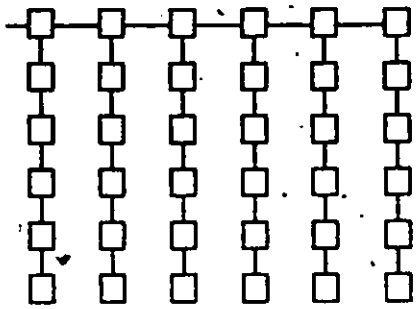


Figure 9. A structure for the priority queue.

3. Area-Efficient Embeddings of Binary Tree

3.1 Introduction

The two embedding strategies described in the previous chapter not only lose their claimed logarithmic time complexity, but also generate layouts which have an unacceptable number of unused processors. Therefore, new area-efficient embedding techniques which satisfy the design requirements of specific applications, must be investigated.

Given a problem for which a systolic tree is required, two different cases can be distinguished, depending on whether or not the application requires a full binary tree.

In the case of full binary trees, it is difficult to produce compact solutions. Typically, they are ad hoc ones obtained for a specific tree height. In all these designs, inevitably, as the size of the tree increases, so does the number of idle processors [7].

If a full binary tree is not required, then it becomes possible to have a compact binary tree layout where there is no waste regardless of the size of the tree.

In this chapter, we describe several area-efficient embedding algorithms for binary trees, one of which is area-optimal. Given a rectangle containing a full H-tree, this algorithm recuperates all the idle processors by activating them as children of some of the nodes of the H-tree. Thus, if we let d denote the delay from the root of the H-tree to any of its leaves, then the tree produced by our algorithm will have a delay of (d+1).

In section 3.2, we describe the framework and discuss the fundamental concepts and techniques associated with these algorithms. In the next three sections, we present area-efficient embeddings of binary trees. We first develop, in section 3.3, a simple algorithm, the waste reduction algorithm, that produces very regular binary trees. An area-optimal embedding algorithm is described in section 3.4. In section 3.5, we show how this algorithm can be enhanced to minimize the sum of the distances between the root and the leaves. Finally, in section 3.6, we analyze the complexity of these algorithms.

3.2 Notation and Basic Strategies

3.2.1 Basic Notations

As we have explained in chapter 2, an actual layout of a binary tree onto an RSA consists of three different kinds of processors:

1. The active nodes of the tree.
2. The relayers which are processors used to implement the long wires.
3. The idle (i.e. unused) processors which are simply "wasted".

Throughout this chapter, we will use several figures to illustrate the execution of the algorithms we develop. All these figures adopt the following conventions:

1. O denotes a node of the initial full H-tree.
2. X denotes an idle processor.
3. * denotes a relayer.

4. R denotes an idle processor that was recuperated.

5. $ denotes a node which needs special handling.

For example, figure 10.a represents the H-tree of 63 nodes, figure 10.b displays the actual rectangle of processors that contains the corresponding layout.

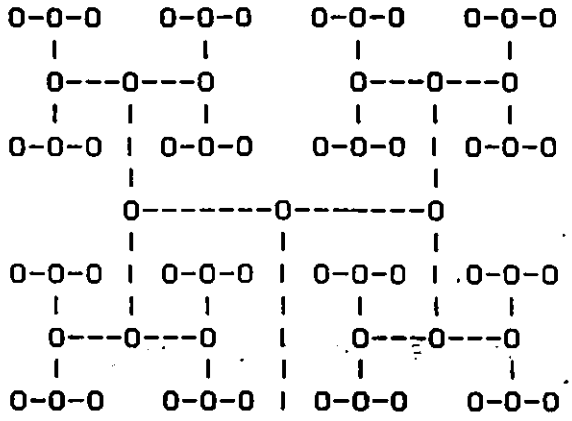


Figure 10.a The H-tree of 63 nodes.

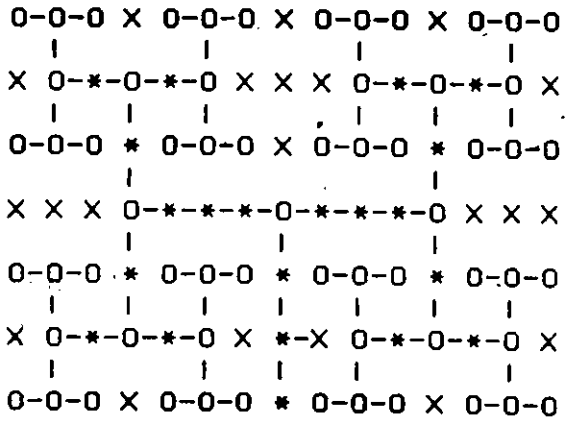


Figure 10.b Layout for Fig. 10.a

Recall that an RSA is a square of hexagonally-connected cells; one such cell is shown in figure 11. For clarity, each link of a processor is numbered; the numbering scheme is also displayed in figure 11.

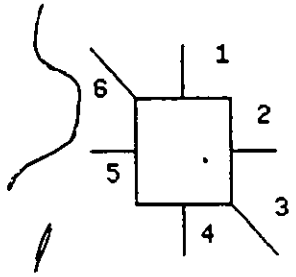


Figure 11: A cell of the RSA and its links.

3.2.2 Basic Philosophy

Our waste reduction algorithms try to minimize the number of idle processors by activating them, as new nodes of the initial full H-tree. The technique used to recuperate the idle processors lies upon the following observations:

1. Since the processors of the RSA are fully synchronized, the relayers can act as actual nodes in the tree. In other words, whether the relayers simply pass information (without processing it) or actually behave as nodes, does not affect the time performance of the systolic tree. By definition, the clock pulse has to be the time delay associated with the slowest cell of the systolic structure.

2. When relayers behave as nodes, they may be allowed two children. The initial H-tree only provides one child per relayer, therefore each relayer can use an adjacent idle processor as a second child.

3. Each idle processor has at least two leaves of the initial H-tree as neighbours.

4. As shown in section 3.4, increasing the maximal delay (from root to leaf) by 1 allows us to recuperate all remaining idle processors as children of the leaves of the initial full H-tree. This increase of the maximal delay is insignificant, especially for large trees.

3.2.3 Allocation of the Idle Processors

Our waste reduction algorithms try to minimize the number of idle processors by activating them as children of the leaves (and relayers) of the initial full H-tree. In other words, the idle processors become leaves of the new binary tree and have some of the leaves of the initial H-tree as parents.

Initially, all the idle processors are considered to be available. To allocate the idle processors, the algorithms require that each initial leaf of the H-tree be able to decide which of its neighbours are still idle. Thus, when an idle processor becomes the child of a leaf, it broadcasts an appropriate message to all of its neighbours, so that they can record this change of status (using some internal model representing their neighbours' status). It should be obvious that several leaves of the initial H-tree may compete for the same idle processor; every algorithm must have a safe allocation scheme that never gives an idle processor to more than one leaf. Therefore,

a node never directly activates an idle processor as one of its children; each node that wants a particular idle processor, sends a request message to this processor. All candidacies are received simultaneously by the idle processor which then decides which node wins (i.e. which node becomes its parent) and sends out messages to announce the result, so that the neighbours can update their internal model. The criterion used by an idle processor to select its parent, varies from one step of an algorithm to the other.

The algorithms that allow relayers to claim a child, do not permit two relayers to compete for the same idle processor. As a matter of fact, as explained in subsection 3.2.5, the allocation algorithm traverses the relayers in such a way that no two adjacent relayers try to obtain a child simultaneously.

3.2.4 Allocation of Idle Processors for a Leaf.

Our waste reduction algorithms try to minimize the number of idle processors by activating them as children of the leaves of the initial H-tree. As explained in the preceding subsection, each leaf (of the initial H-tree) attempts to claim one or more of its available idle neighbours.

Each leaf that can claim children (i.e. which has at least one idle processor as a neighbour) executes the following steps:

(a) If the idle processor reached via a diagonal link (links 6 and 3) of a leaf is available, then that leaf requests it as a child. (Notice that if a leaf has a choice, it will request the idle processor reached via its link 6.) If two leaves request the same processor then this idle processor is allocated to the leaf closest

to the upper left corner of the RSA.

(b) Any leaf that does not have two children sends a request message to all its available, non-diagonal neighbours. Each idle processor will select the leaf closest to the lower right corner of the RSA. Notice that by changing the corner of the RSA [from the upper left in step a) to the lower right in step b)], the allocation algorithm ensures a better distribution of the idle processors between the initial leaves. In other words, we do not favour the cells of any particlar region of the RSA, but try to distribute the idle processors among all the leaves.

(c) Finally, it is possible that, after the two first steps, a leaf has received more than two children! In this case, the leaf determines which neighbours can use the extra children. The algorithm described in this subsection is devised so that there is always some neighbour(s) to accept these extra children. The leaf gives up its extra children according to the numbering scheme used to label its links (i.e. link 1 first, then link 2, etc.).

3.2.5 The traversal algorithm

All waste reduction algorithms are multi-step distributed algorithms. Since the tree is embedded in an underlying RSA (which only processes systolic algorithms), it is extremely important that each step be in fact a systolic algorithm. In particular, this implies that each allocation step of an algorithm traverses the cells of the RSA in a precise, distributed fashion.

All allocation steps start with the sole relayer that is on a boundary of the RSA (i.e. the relayer that starts the chain of relayers used to reach the root of the full tree). If a traversed cell does not already have two children and is allowed to claim children, then it tries to obtain them by requesting some of the adjacent idle processors. Once this task is completed (regardless of the outcome), the algorithm marks the current cell as having been visited and traverses all the remaining neighbours. Figure 12 illustrates this traversal algorithm on an H-tree of 63 nodes by displaying (in hexadecimal) the times at which the cells are visited.

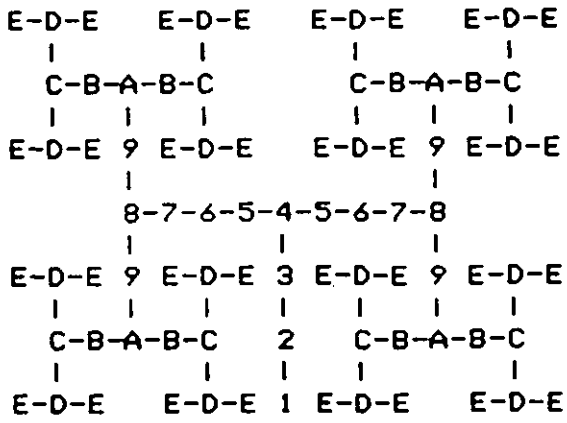


Figure 12. Times Instances (in hexadecimal) at which the processors are traversed in a layout of 63 nodes.

3.3 The waste reduction algorithm

Given a rectangle containing a full binary H-tree, we first develop an algorithm where one out of every two leaves recuperates exactly one idle processor. Formally, we produce a binary tree of $(2^K + 2^{K-2})$ nodes, where K is the depth of the initial H-tree.

When studying the layout of an H-tree, we realize that certain nodes do not have any idle processors as neighbours (see fig. 10.b). It is obvious that these nodes, called dead cells, cannot recuperate any idle processor. Also, since the cells are programmable, an embedding algorithm may tell each node of an initial H-tree, whether it is the left or the right child of its parent, and whether or not it is allowed to claim idle processors.

The fundamental idea of this algorithm is to construct a tree where only certain leaves are allowed to claim idle processors. For clarity, such a leaf is systematically assigned the role of left child of its parent, and a dead cell is always the right child of its parent.

As mentioned above, the algorithm activates one out of every two leaves. (We will represent these selected leaves by an A, and the dead cells by a D.) In fact, for each H-tree of depth 3 (i.e. the building block of an H-tree), the algorithm selects one active diagonal by using the following steps:

1. The algorithm first traverses the initial full tree, and for each H-tree of depth 3 that contains a dead cell, selects the (activated) diagonal that does not contain this cell (see figure 13). The two cells on each activated diagonal claim a unique child according to the algorithm of subsection 3.2.4. In other words, the position of a dead cell within its H-tree of depth 3 is used to decide which two leaves of this H-tree can claim children. (Notice that there is at most one dead cell per H-tree of depth 3.)

2. In the second step, the algorithm traverses the tree again, this time a diagonal is activated for every H-tree of depth 3 that was not affected by the first step:

   (a) First, in each building block, the algorithm checks if the first step has introduced new dead cells in which case, the diagonal is chosen as in step 1.

   (b) If a building block still does not have any dead cells, then we use the root of this block to decide which of the two diagonals to activate. Each such root is necessarily reached via a relayer. If the relayer is either on the right or under the root, then we activate the first diagonal (\ ); otherwise we select the second diagonal (/) (See figure 14).

   (c) Again, the cells of each activated diagonal claim their unique child according to subsection 3.2.4.

Figures 15.a through 15.c show the step by step execution of this algorithm for an initial H-tree of 127 nodes; figure 15.d displays the resulting binary tree. The results of the algorithm for initial H-trees of 63 and 255 nodes can be found in Appendix 1.

We conclude this section by observing that for this algorithm, the relayers are not allowed to claim a child. However, nothing prevents us from still using them as actives nodes of the new tree. In this case, the waste reduction algorithm produces trees of $(2^K + 2^{K-2} + R(K))$ nodes, where K is the height of the initial H-tree, and R(K) is the number of relayers in the layout of this initial tree. Figure 16 displays the binary tree (for an initial H-tree of depth 6) where the relayers act as nodes.

3.4 The Waste Elimination Algorithm.

Given a rectangle containing an H-tree, we now develop an algorithm where all the idle processors are recuperated. The algorithm directly proceeds from the waste reduction algorithm of section 3.3. In the waste elimination algorithm, all leaves are allowed to claim children according to the algorithm of subsection 3.2.4, and all relayers act as nodes but are not allowed to claim an extra child. The algorithm is best illustrated by an example, where figures 17.a through 17.d present its execution for an initial H-tree of 255 nodes.

Finally, we remark that for all the algorithms we have developed so far, an obtained binary tree of depth k contains all trees of lesser depths.

3.5 Minimizing the Sum of the Distances

In the waste elimination algorithm, the relayers were not allowed to claim an extra child; all the idle processors were recuperated by the leaves of the initial H-tree. However, if a relayer was allowed to claim an idle processor as its child, then this idle cell would become a node which would be a lot closer to the root of the new tree than it would have been, had it been recuperated by one of the initial leaves. If we allow relayers to claim an extra child, then we effectively minimize the sum of the distances between the root and the leaves of the new binary tree.

This enhancement is easily realizable; the resulting algorithm consists of the following allocation steps:

1. Each relayer tries to obtain a second child by checking its available neighbours according to the order defined by the numbering scheme for the links. If the traversed cell is not a relayer, we directly go to the neighbour(s) that have not yet been visited.

2. Each leaf of the initial full binary tree, tries to obtain two children according to the algorithm of subsection 3.2.4.

Again, this algorithm is best illustrated by an example; figures 18.a through 18.d present the step by step execution of this enhanced waste elimination algorithm on an H-tree of 255 nodes. Figure 18.e displays the binary tree obtained from the algorithm. The layouts and corresponding trees obtained for initial H-trees of 63 and 127 nodes can be found in Appendix 2.

3.6 Complexity Analysis

First we shall derive some formulas that will be used to compare our waste elimination algorithm to other embedding techniques, in chapter 4.

Recall that the area of the rectangle embedding an H-tree of level k is given by:

$S_K = \{ (2^{(K+1)/2} - 1) * (2^{(K+1)/2} - 1) \}$ if K is odd

$= \{ (2^{(K+2)/2} - 1) * (2^{K/2} - 1) \}$ if K is even

Within the context of the RSA, the number of wasted processors is computed as the area minus the number of nodes in the obtained binary tree. In the computations of the next chapter, we consider trees where the relayers

act as nodes. (Recall that half of the idle processors of the layout of an H-tree act as relayers and are directly recuperated by our algorithms.) In other words, in chapter 4, there are no relayers in the binary trees produced by our algorithms!

If we let R_K denote the number of relayers in the layout of the initial H-tree, then, for the waste reduction algorithm, the obtained binary tree has $2^K + 2^{K-2} + R_K$ nodes. As for the waste elimination algorithm, it is obvious that all processors act as nodes and thus, that there is no waste!

The binary trees obtained from the waste reduction algorithm are quite easy to characterize: each leaf of the initial H-tree receives a left son. The positions of the nodes that were initially relayers, can be extracted from the definition of the H-tree.

The waste elimination algorithm produces binary trees for which a functional characterization is difficult to obtain.

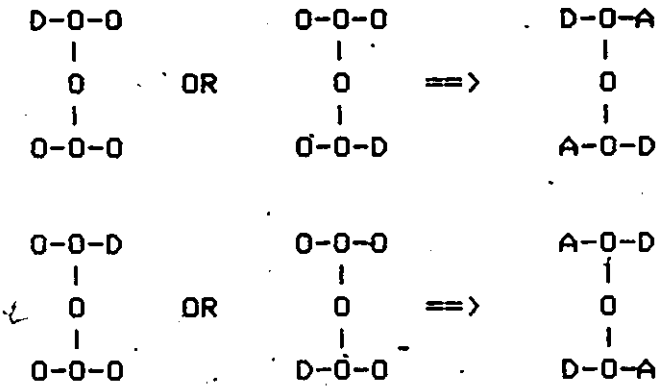


Figure 13. Activation of a diagonal according to the dead cell

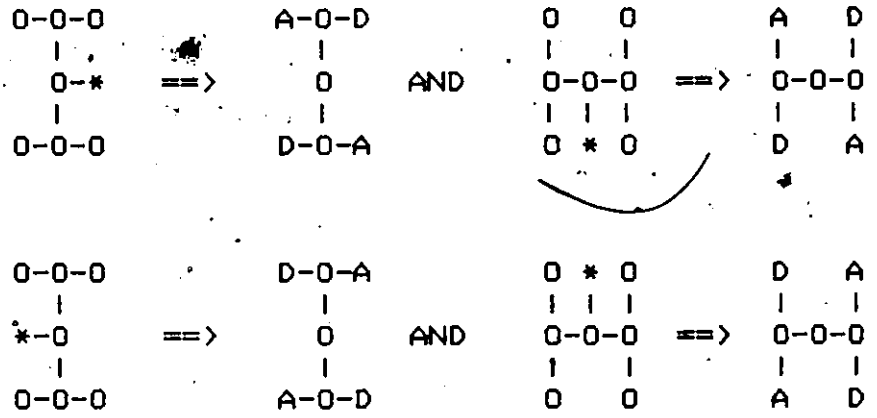


Figure 14. Activation of a diagonal according to the relayer

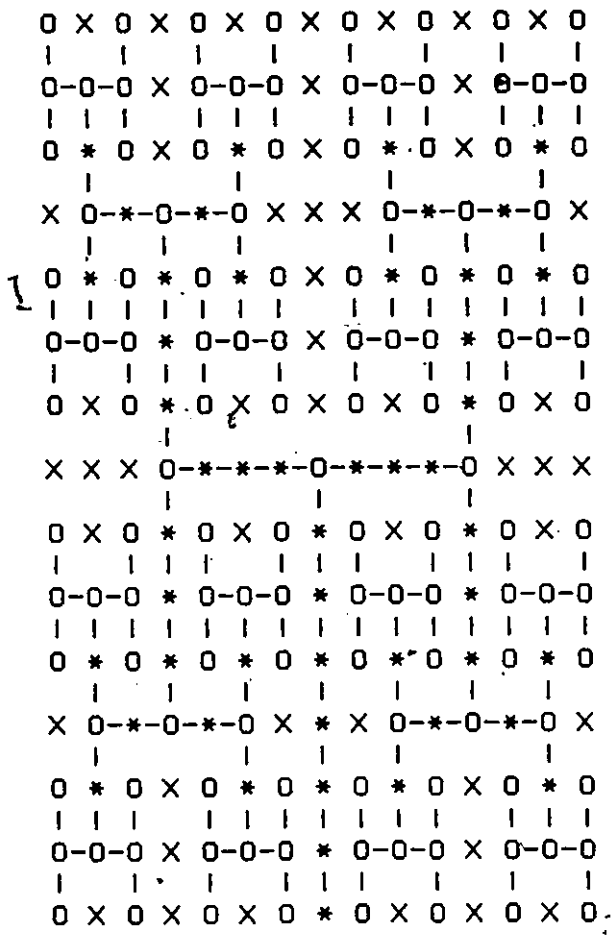
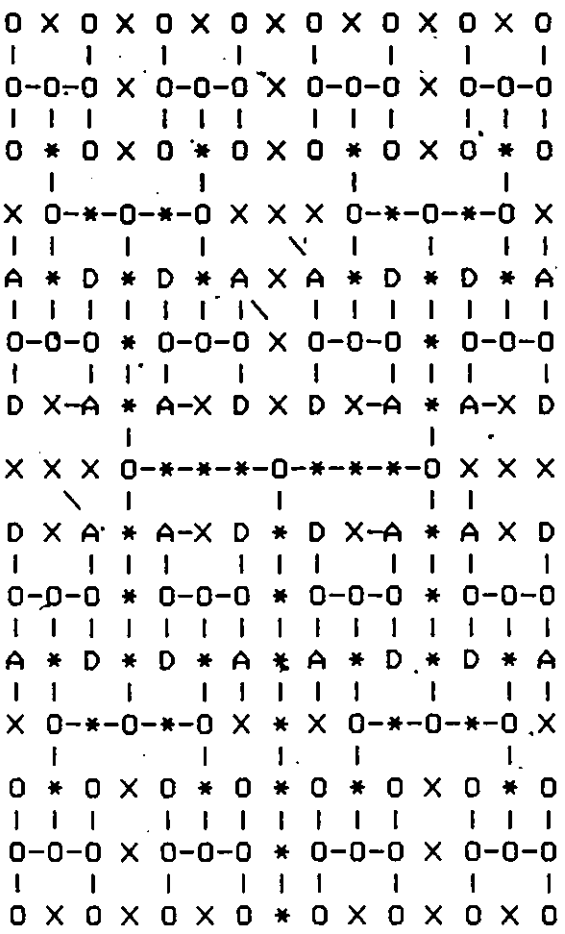


Figure 15.a Initial layout for an H-tree of 127 nodes.



STEP 1: recognize the dead cells and activate the opposite diagonals.

Figure 13.b First step of the waste reduction algorithm

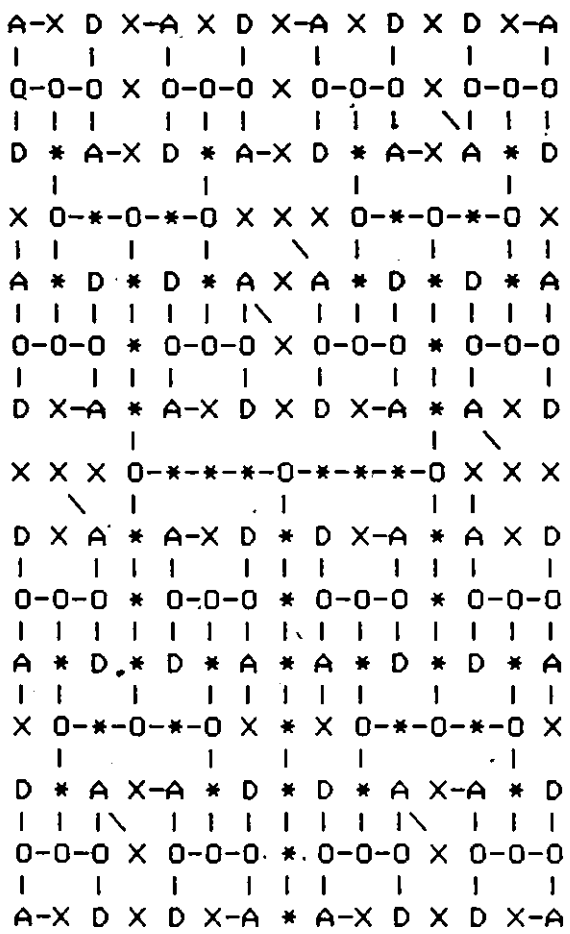


Figure 15.c Result of the waste reduction algorithm

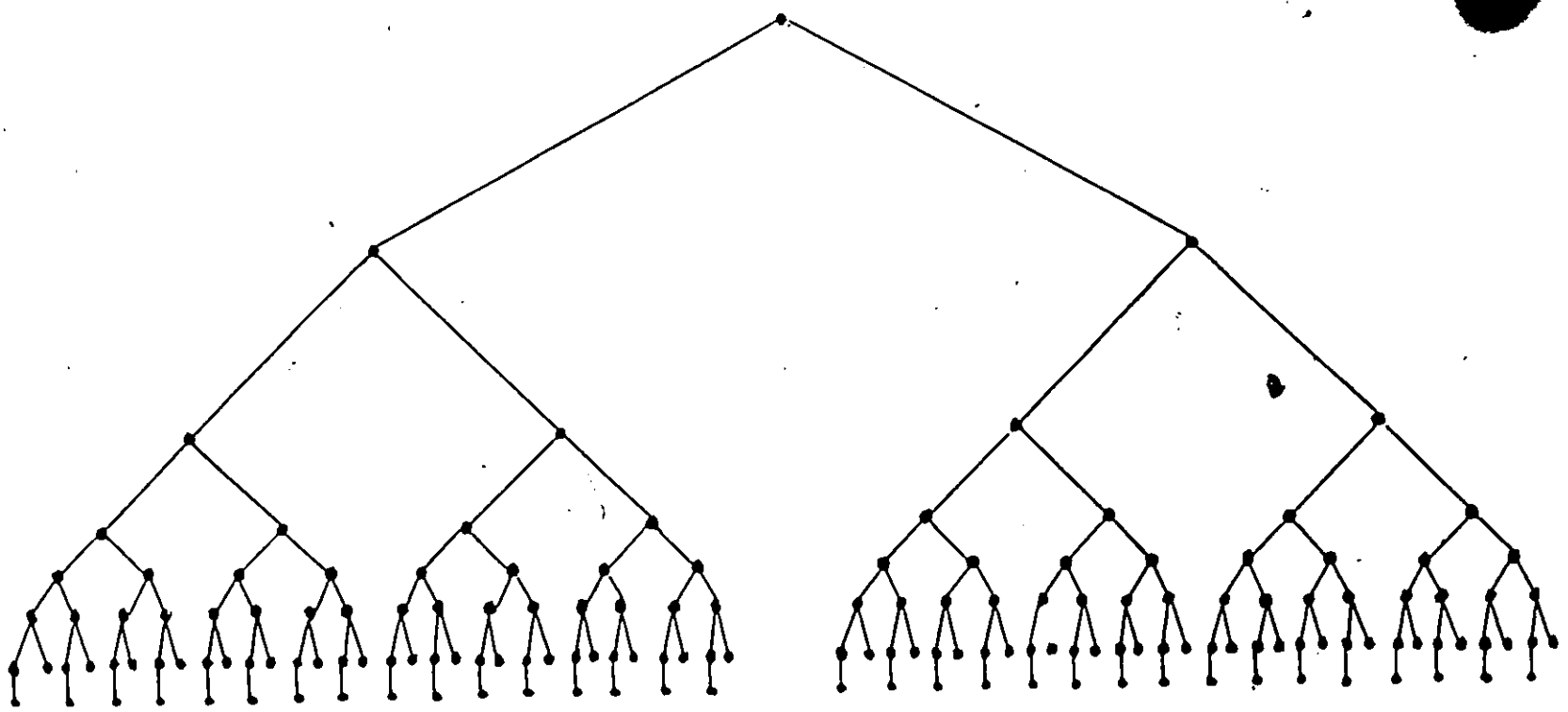


Figure 15.d Corresponding binary tree for the waste reduction algorithm

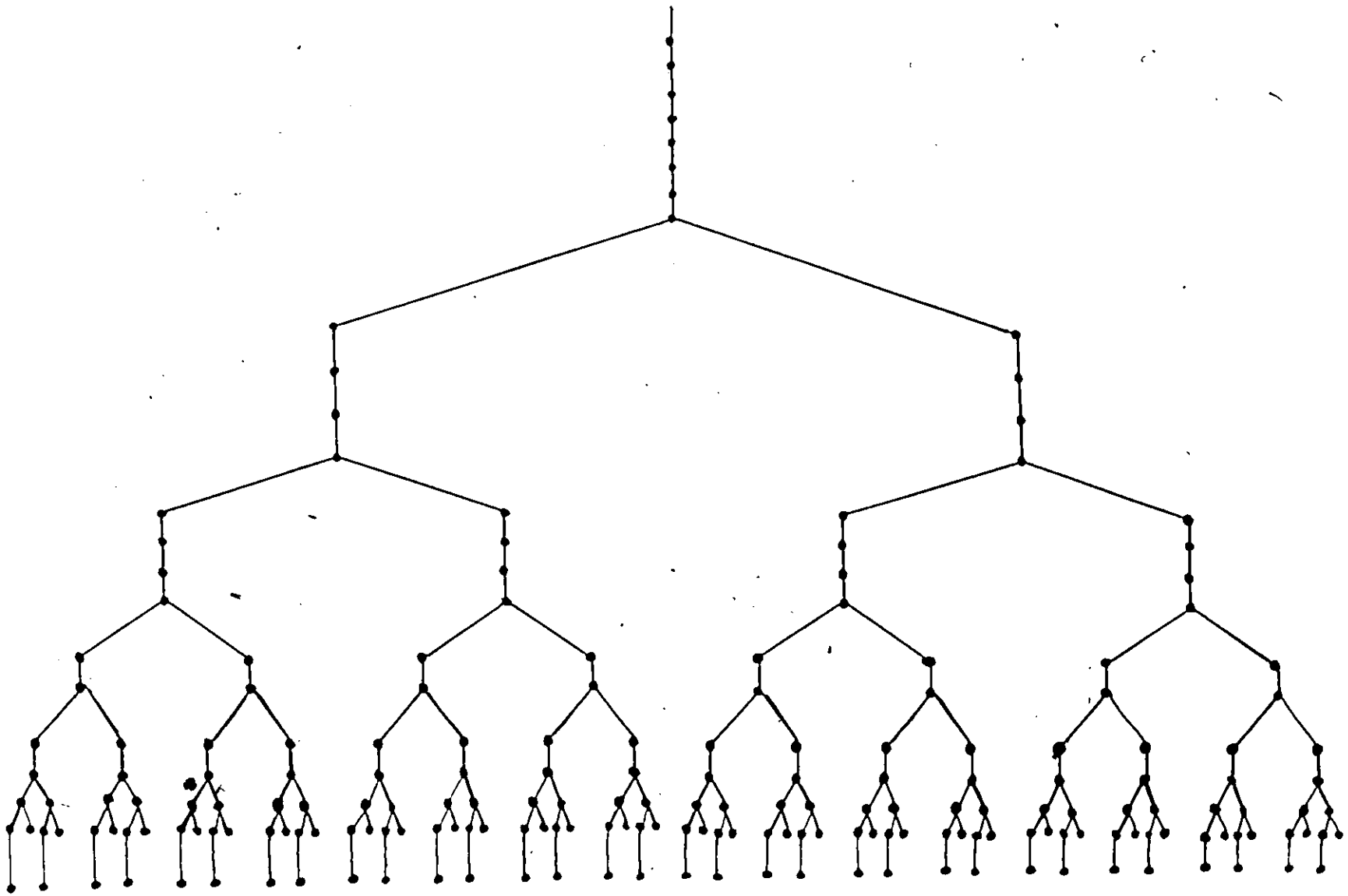


Figure 16. The binary tree for the waste reduction algorithm, using the relayers as nodes

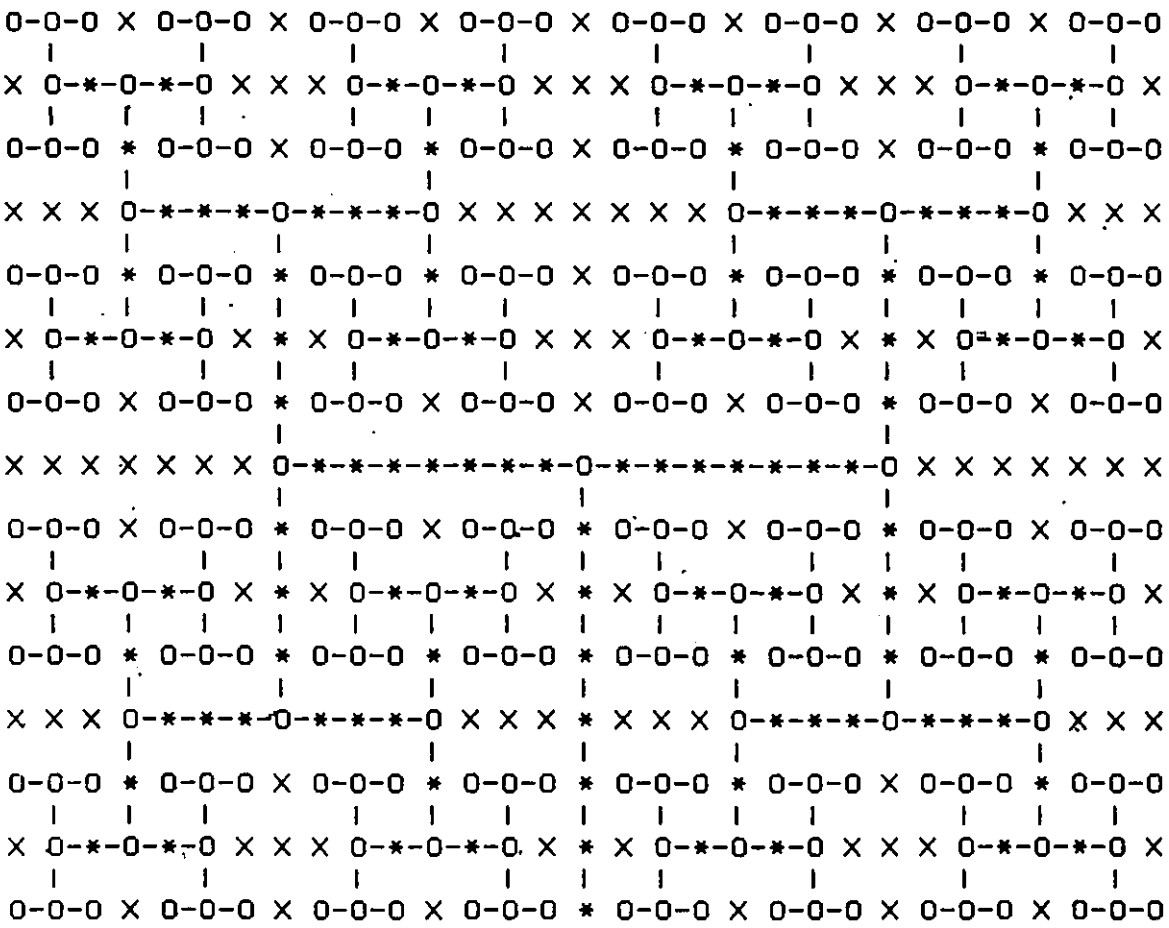
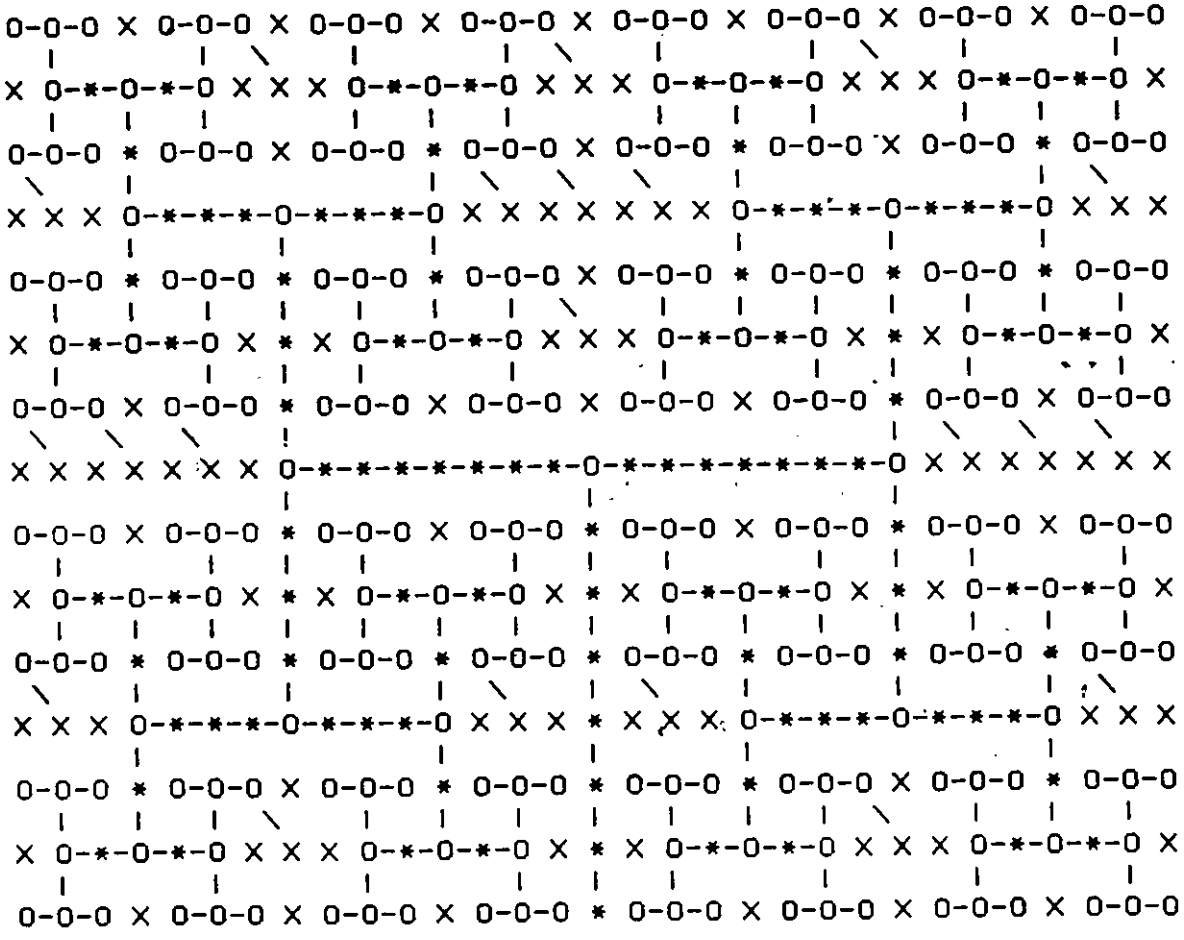
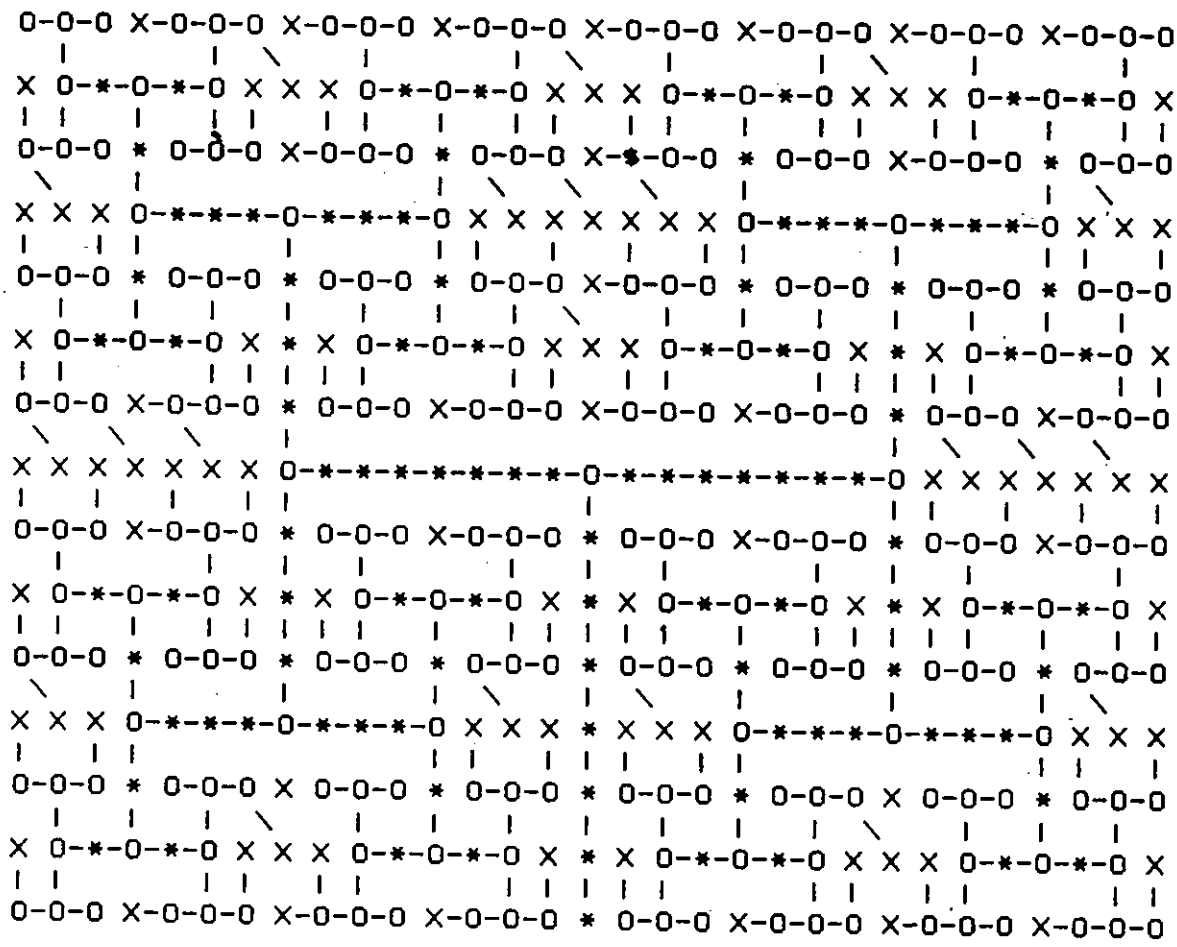


Figure 17.a The initial layout of an H-tree of 255 nodes.



STEP 1 : Every leaf that has an available diagonal link, requests it. In the case of conflicts, the node closest to the upper left corner wins.

Figure 17.b First step of the waste elimination algorithm



STEP 2 : Every leaf requests all its available neighbours.
In the case of a conflict, the node closest to the lower right corner wins.
On the third row of the layout, one leaf has been denoted by $ to indicate that it has three sons and must give one away. The cell gives the extra link to the neighbour right above it.

Figure 17.c Second step of the waste elimination algorithm

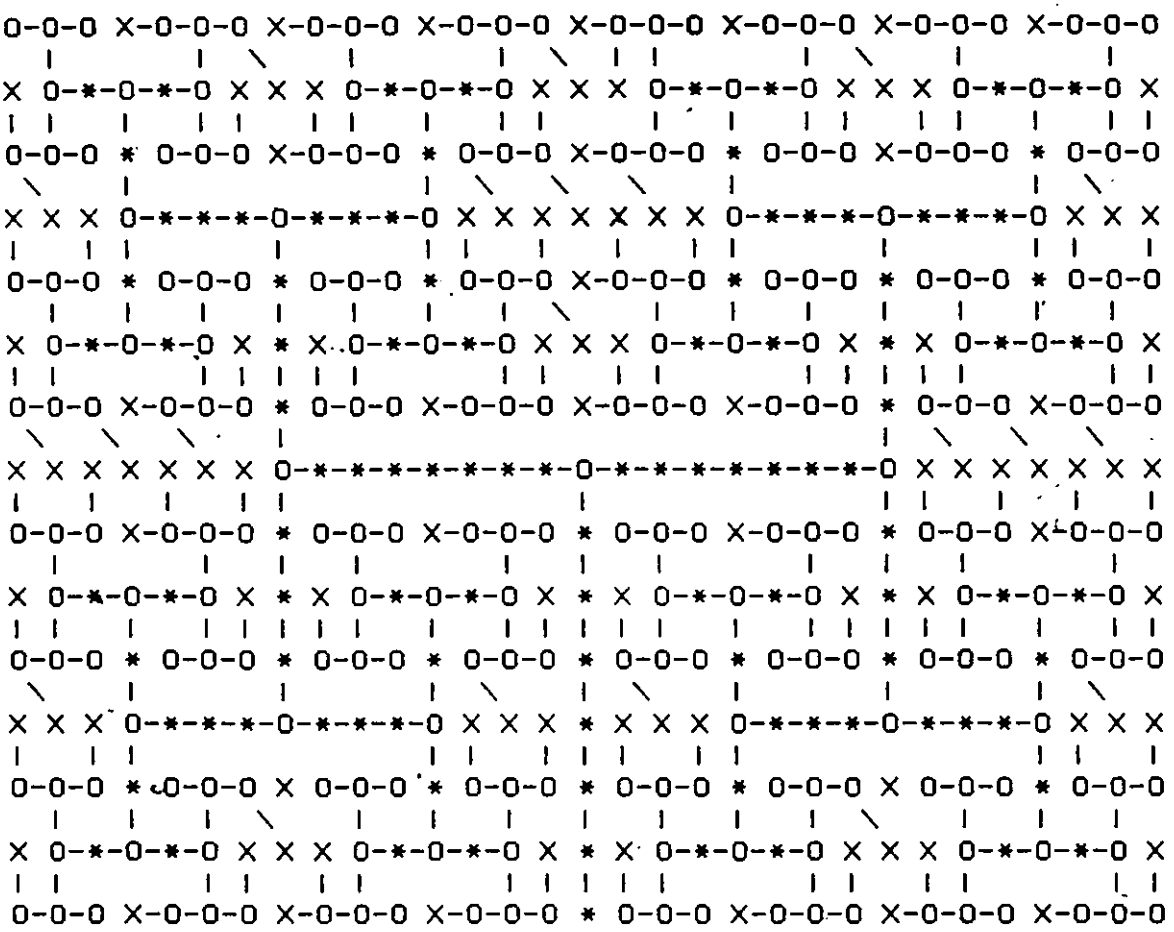
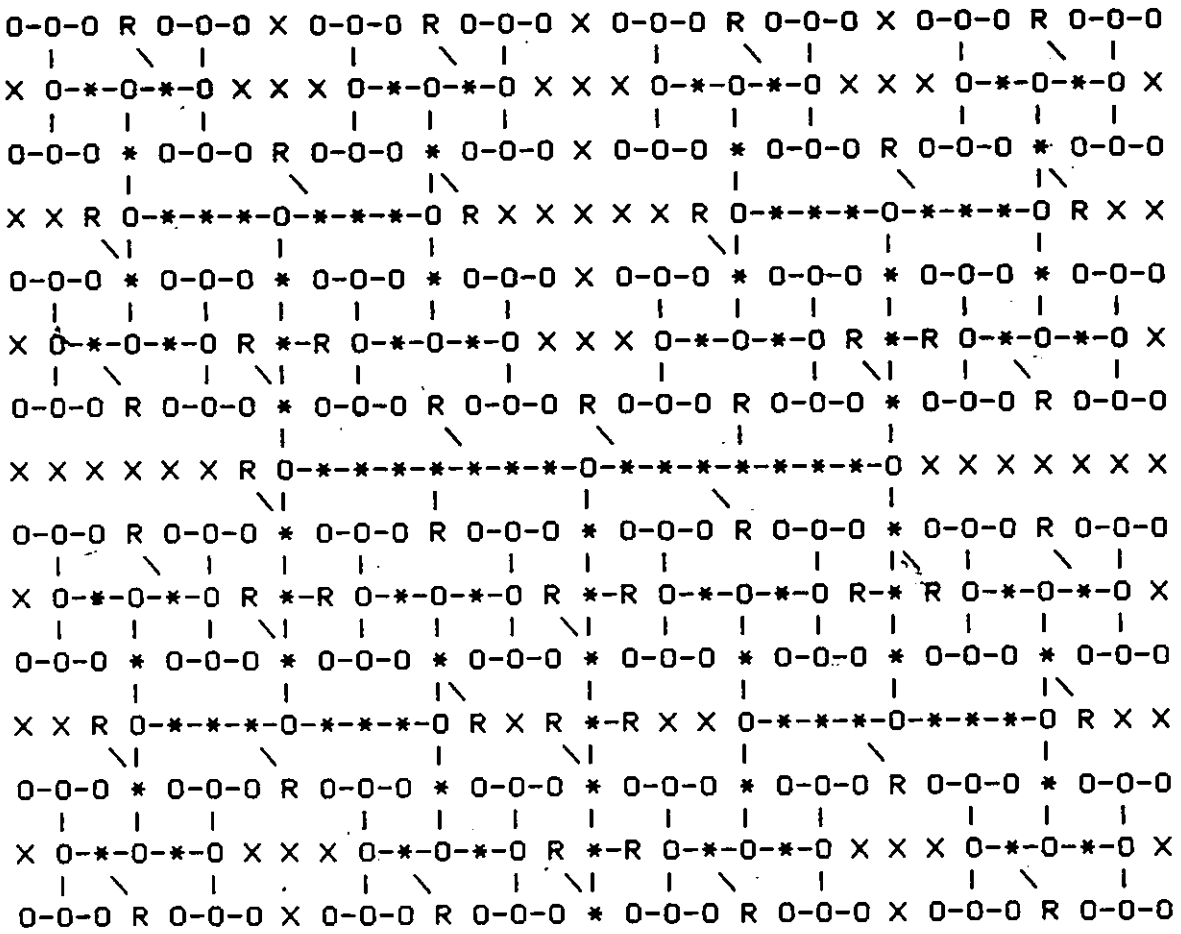
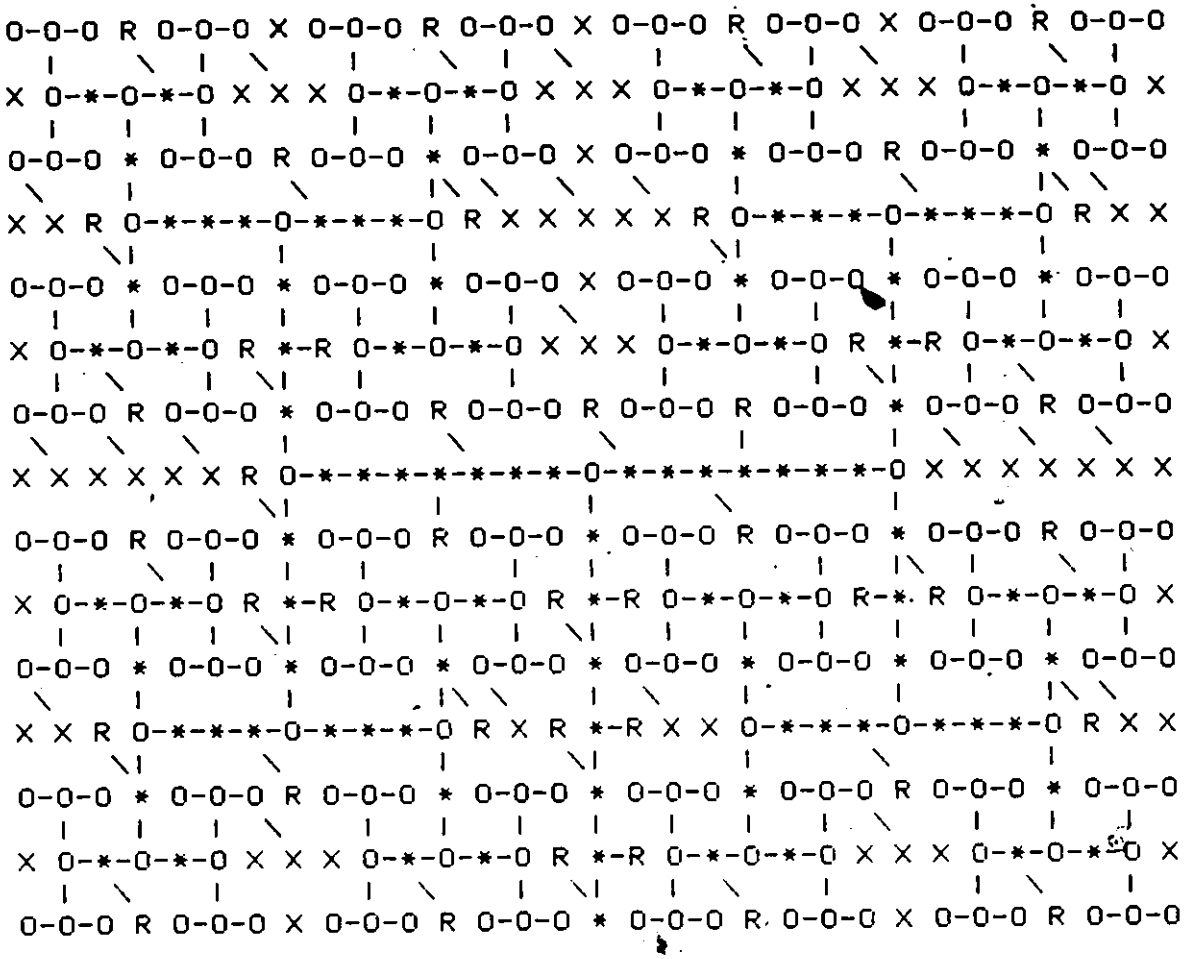


Figure 17.d Final layout for the waste elimination algorithm



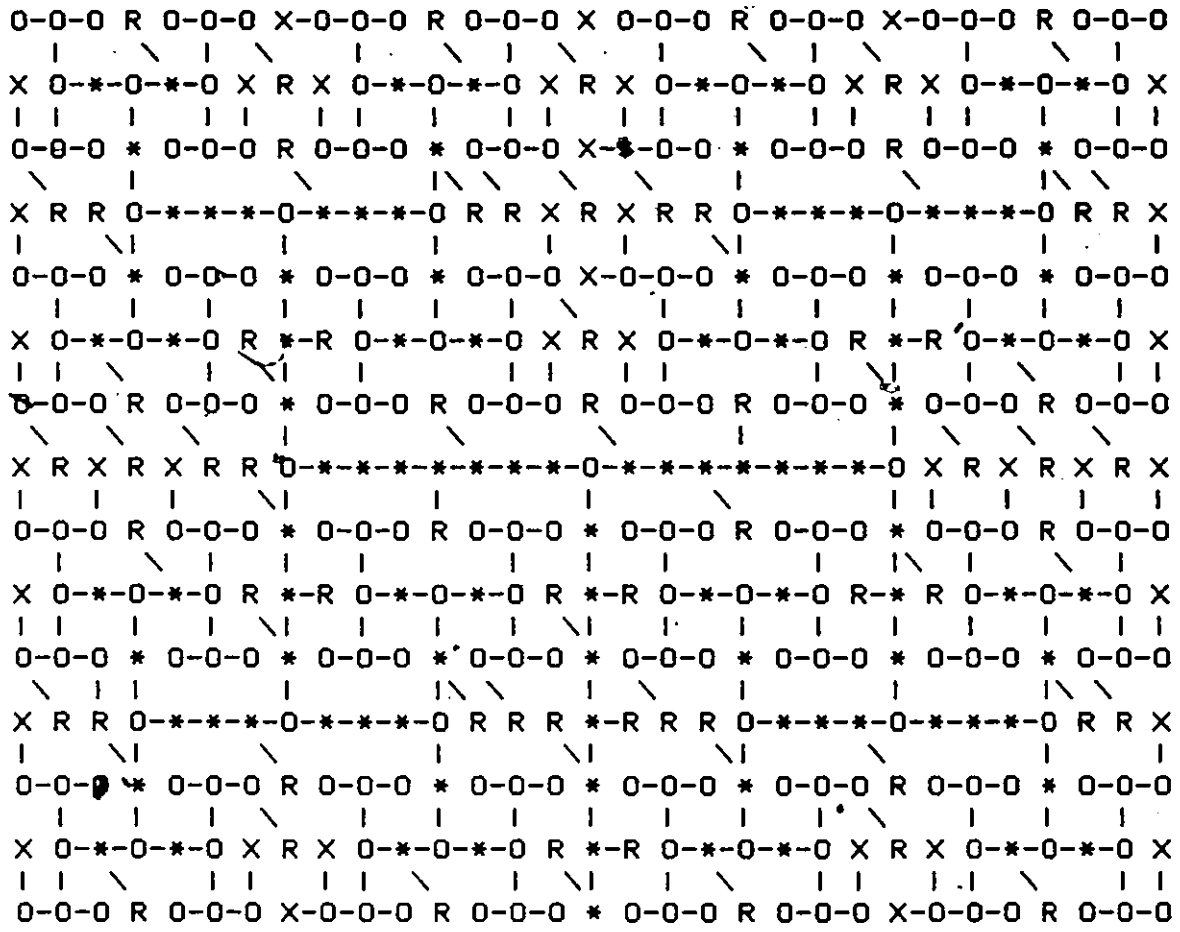
STEP 1: Every relayer requests the first of its available neighbours. The relayer first tries its two diagonal links, then refers to the numbering scheme adopted for the links. In the case of a conflict, the relayer closest to the upper left corner wins.

Figure 18.a First step of the waste elimination algorithm with relayers



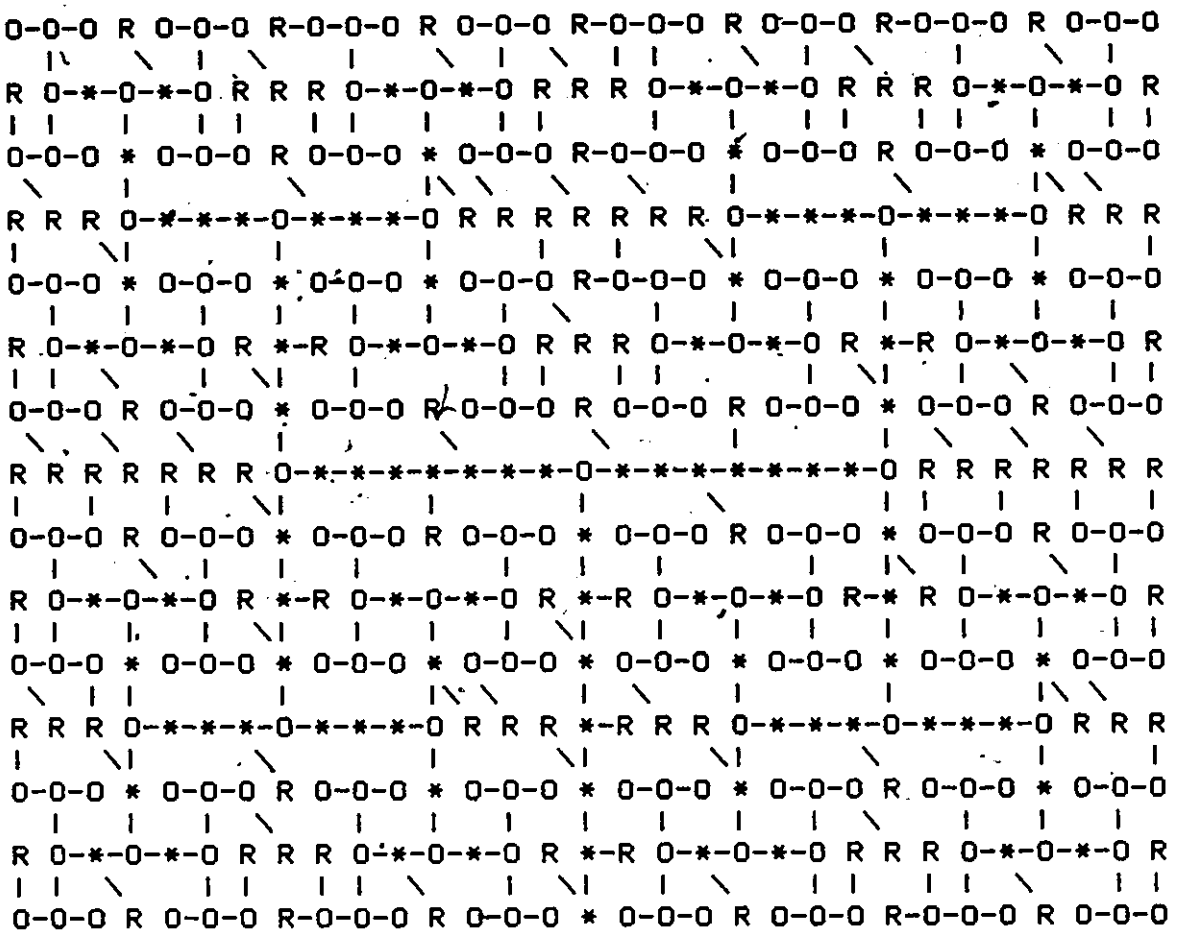
STEP 2.a : Every leaf that has an available diagonal link, requests it. In the case of conflicts, the node closest to the upper left corner wins.

Figure 18.b Second step of the waste elimination algorithm with relayers



STEP 2.b : Every leaf requests all its available neighbours. In the case of a conflict, the node closest to the lower right corner wins. On the third row of the layout, one node has been denoted by a $ to indicate that it has three sons and therefore, must give one away.

Figure 18.c Third step of the waste elimination algorithm with relayers



STEP 2.c : Any node that has extra sons gives them away according to the order used to number the links of a cell of the RSA.

Figure 18.d Final layout for the waste elimination algorithm with relayers

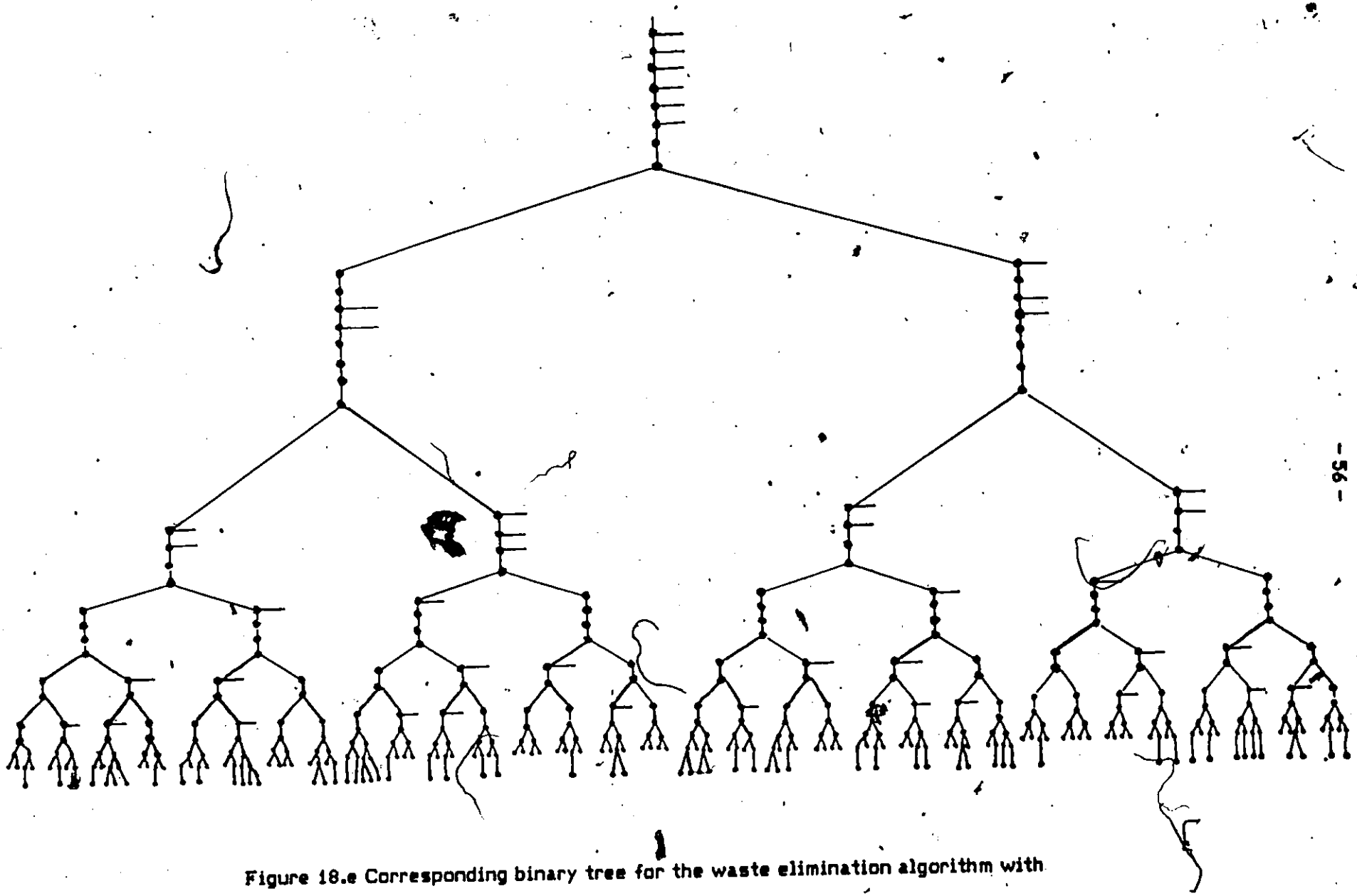


Figure 18.e Corresponding binary tree for the waste elimination algorithm with relayers

4. Comparison of the Different Embedding Techniques for Binary Trees

4.1 Summary of the Results

The results obtained for the different embedding methods are summarized in the tables below. For the zigzag method we use the largest tile Gordon et al. published [7], which is a 9x8 tile that contains 64 nodes. Each of the first four tables lists the specifications for different initial full trees. Notice that the waste includes the relayers. Also, in tables 3 and 4, the initial relayers act as nodes in the new binary tree. Finally, for a maximal number of nodes in a binary tree, table 5 compares the areas required by each method.

K	Nodes	Zigzag	Area	Waste	Relayers
6	64	13x8	104	40	8
7	128	13x17	221	93	19
8	256	23X17	391	135	27
10	1024	43x35	1505	481	96
20	1048576	1283x1151	1476733	428157	85631

Table 1: The Zigzag Method.

Recall that the waste is computed as (area - nodes). Also , in the above table, the amount of relayers was set to 20% of the waste; this metric was included only to provide some comparison with the H-tree method.

K	Nodes	H-tree	Area	Waste	Relayers
6	63	15x7	105	42	21
7	127	15x15	225	98	49
8	255	31X15	465	210	1050
10	1023	63x31	1953	930	465
20	1048575	2047x1023	2094081	1045506	522752

Table 2: The H-tree Method.

Again, the waste is computed as (area - nodes); the relayers constitute half of this amount.

H-nodes	Nodes	REDUCTION	Area	Waste	Relayers
63	92	15x7	105	13	0
127	192	15x15	225	33	0
255	392	31X15	465	73	0
1023	1616	63x31	1953	337	0
1048575	1702400	2047x1023	2094081	391681	0

Table 3: The Waste Reduction Algorithm.

For tables 3 and 4, the first column gives the number of nodes in the initial H-tree.

H-nodes	Nodes	ELIM.	Area	Waste	Relayers
63	105	15x7	105	0	0
127	225	15x15	225	0	0
255	405	31X15	465	0	0
1023	1953	63x31	1953	0	0
1048575	2094081	2047x1023	2094081	0	0

Table 4: The Waste Elimination Algorithm.

For this method, the number of nodes in the tree is equal to $2^K + 2^{K-2} + R_K$ where R_K is the number of relayers in the initial H-tree.

Max.N	H-tree	Zigzag	ELIM.	REDUCTION
63	15x7=105	13x8=104	15x7	15x7
92	15x15=225	13x17=221	15x7	15x7
105	15x15	13x17	15x7	15x15
127	15x15	13x17	15x15	15x15
192	31x15=465	23x17=391	15x15	15x15
225	31x15	23x17	15x15	31x15
255	31X15	23x17	31x15	31x15
392	31X31=961	23x35=805	31x15	31x15
405	31X31	23x35	31x15	31x31
1023	63x31=1953	43x35=1505	63x31	63x31
1616	63x63=3969	43x71=3053	63x31	63x31
1953	63x63	43x71	63x31	63x63

Table 5: Comparison for a same maximal number of nodes.

The first column gives the maximal number of nodes required.

4.2 Comparison of the Embedding Methods.

From the above results, it becomes obvious that if a problem does not require a full binary tree, then a waste elimination strategy produces binary trees that are significantly better than those generated by the traditional embedding techniques. The area-optimal waste elimination algorithm of section 3.4 stands out as the clear winner; however, the waste reduction algorithm of section 3.3 still offers a major improvement over the traditional embeddings. Furthermore, the embedding algorithms we have introduced in chapter 3, are in fact systolic algorithms, and thus can quite easily be implemented on an RSA.

4.3 Conclusions

The idea of recuperating idle processors can very well be applied to other embedding techniques. However, the H-type embedding is not only the method generally accepted, but also the easiest one to implement. Furthermore, the trees produced by the methods introduced by Gordon et al. do not lead to a significant recuperation of wasted processors.

Figure 19 displays a zigzag tree of 256 nodes. From this figure it is obvious that most of the idle processors are out of the reach of the leaves of the four tiles used in the layout. Larger tiles would be helpful in that they would reduce the number of idle processors while increasing the number of leaves on the perimeter of each tile. However recall that Gordon et al. fail to produce such larger tiles! We remark that any embedding technique (for binary tree layouts) that uses a "large" tile as a building block, will in fact introduce waste between these "air-tight" tiles (in which the waste is minimal and generally irrecuperable); therefore the leaves will only be able to use the adjacent idle processors which represent an insignificant proportion of the total waste! The advantage with the H-tree is that the idle processors are spread throughout the entire layout and thus can all become nodes of a new binary tree, as our waste elimination algorithm demonstrates. Furthermore, it is possible to minimize the sum of the distances between the root and the new leaves.

The embeddings we developed in this first part of the thesis show that, in the context of the RSA, it is necessary to redefine the computation of the area of a systolic structure, and to investigate new embbedding techniques for binary trees. More generally, this suggests a general

reevaluation of the existing embedding techniques for usual systolic structures, as well as, the need for a design methodology for these structures. Part II develops such a methodology.

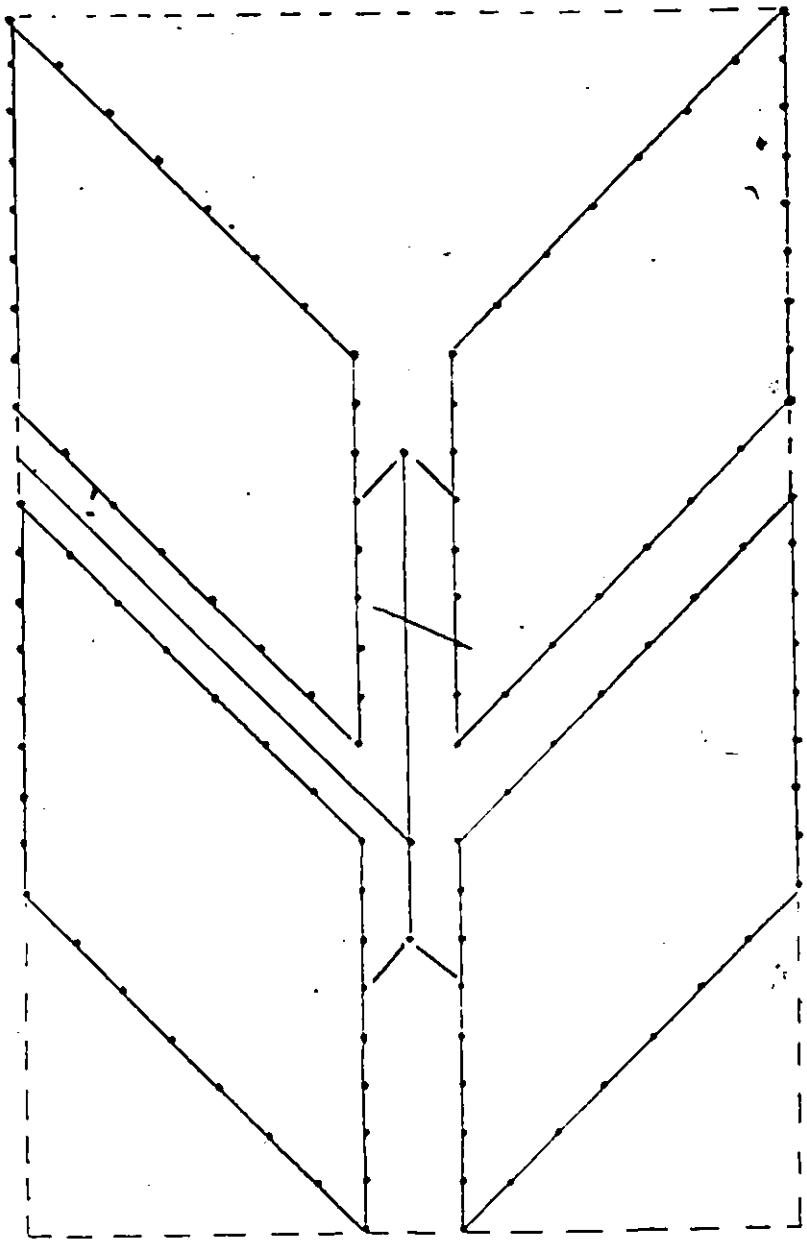


Figure 19 Zigzag embedding for a full tree of 256 nodes

PART II

An Environment for the Design of Systolic Algorithms

5. SADE: A Systolic Algorithm Design Environment

5.1 Introduction

In this second part of our thesis, we describe the major features of SADE, a design environment for the specification and testing of systolic algorithms. This environment is based on the concept of a reconfigurable systolic architecture (RSA) (see section 1.3).

SADE presents the user with a simple yet powerful tool, which allows the specification of arbitrary systolic structures on an RSA, and can simplify the verification process of complex systolic algorithms. Furthermore, the tool is portable, extensible and provides for the graphical simulation of an algorithm's execution on a predefined structure.

A basic characteristic of SADE is that the two fundamental steps of the design task [2] are clearly separated:

(a) specification of the systolic structure on which the algorithm will be executed (i.e. topology of the connections between the cells and cell types).

(b) design of the systolic algorithm itself, achieved by specifying the algorithmic behaviour of each (type of) cell within the structure.

We shall devote a chapter to the study of each of these two steps, but first we will briefly examine the basic ideas of SADE.

5.2 VLSI Design Rules

SADE follows the VLSI design guidelines stated in [9,10]:

1. A systolic structure must have a simple and regular design for its control and data flows.
2. There must as few types of cells as possible.
3. All the cells are fully synchronized and a clock pulse is taken to be the time required for a cell to complete its corresponding algorithm. Currently, self-timed computations cannot be simulated using SADE's routines.
4. A cell can only be assigned a local (i.e. non-shared) memory; this is to say that a cell cannot examine the contents of its neighbours.

5.3 The Placement Algorithm [7]

The placement of a systolic structure onto the RSA can be done either in a centralized manner, i.e. totally from the outside (by the host), or in a distributed fashion, i.e. internally performed by the systolic cells. The first approach relies on a "configuration string" externally generated, containing setup instructions that are distributed to all relevant cells. In the second approach, the process is initiated externally, but the exact configuration is determined internally by the processors. An important advantage of a distributed placement algorithm is that it is independent of the exact size of the available RSA. It dynamically places the structure and consequently, RSAs of different sizes may use the same placement algorithm. Furthermore, since the RSA consists of programmable systolic chips [6] which are powerful microprocessors, there is no increase in the complexity of a

systolic cell. The distributed placement algorithm obviously must be a systolic process itself to be used with the RSA.

5.4 Transferable Data Types

In an actual implementation of an RSA, only bits can be transferred from one cell of the structure to another. Furthermore, the wires between these cells must have a fixed bandwidth. Specifying data transfers at the bit level is a cumbersome task that SADE avoids via a simple abstraction mechanism.

In SADE, the designer does not have to worry about the bandwidth of the wires of the RSA. The messages transferred from one cell to another consist of two entities:

1. a data type descriptor from the following enumeration:
   (a) INT denoting an integer
   (b) REL denoting a real
   (c) CHA denoting a character
   (d) STR denoting a string of characters
2. a message value which must be compatible with the specified data type descriptor.

Chapter 7 details the use of these messages.

5.5 SADE's Philosophy

In SADE, the underlying structure, an RSA, is a square of hexagonally-connected processors (see Fig. 2.); all user-defined structures are obtained by cutting the RSA.

Recall that the design process is conceptually divided in two distinct steps. The designer first defines a distributed placement algorithm, or cutting algorithm, to specify the structure on which one or more systolic algorithms are to perform. This is achieved by specifying, for each cell of the structure, its "activated" links and its cell type. This cutting process duplicates the activation mechanism which would be produced by a distributed placement algorithm on an RSA. Hence the cutting algorithm is in itself a systolic algorithm.

The second step associates a procedure to each different type of cell in the defined (i.e. cut) systolic structure; within each procedure, the local memory and the data transfers corresponding to the cell type are fully described.

For example, if we consider the problem of multiplying two n x n matrices, we can use the diamond-shaped structure proposed by Leiserson and Kung [10,11] (see figure 20). In SADE, the first step of the design consists of a cutting algorithm that embeds this diamond in an RSA. In this example, all cells have a same cell type and an identical topology. The second design step must describe the systolic algorithm used to multiply the two matrices. In SADE, this is simply achieve by defining the algorithmic behaviour of each cell. For this example, all cells execute a same algorithm which consists of a multiply-accumulate operation (i.e. multiply the two current

inputs and add this product to the sum stored in the cell).

SADE is implemented as a set of predefined MODULA-2 routines [14] contained in library modules. Using the IMPORT statement, the designer has direct access to any or all of these routines. Each of the two aforementioned design steps may be described by a MODULA-2 program which calls SADE's routines. This approach eliminates the need for a separate compiler, allows the user to work in a somewhat familiar PASCAL-like environment, and thus reduces apprenticeship to a minimum so that the novice can focus on the design rather than on the syntax. Several features are included to avoid repetition of work by the designer.

5.6 SADE's Modus Operandi

In SADE, the entire design process can be handled in one of two modes: procedural or graphical. In the procedural mode, both structure specification and algorithm definition are performed via high-level programs; in the graphical mode, the structure can be specified via an interactive graphical package. In the thesis, we will focus on the procedural mode; some of the features of the graphical mode are briefly discussed in chapter 8.

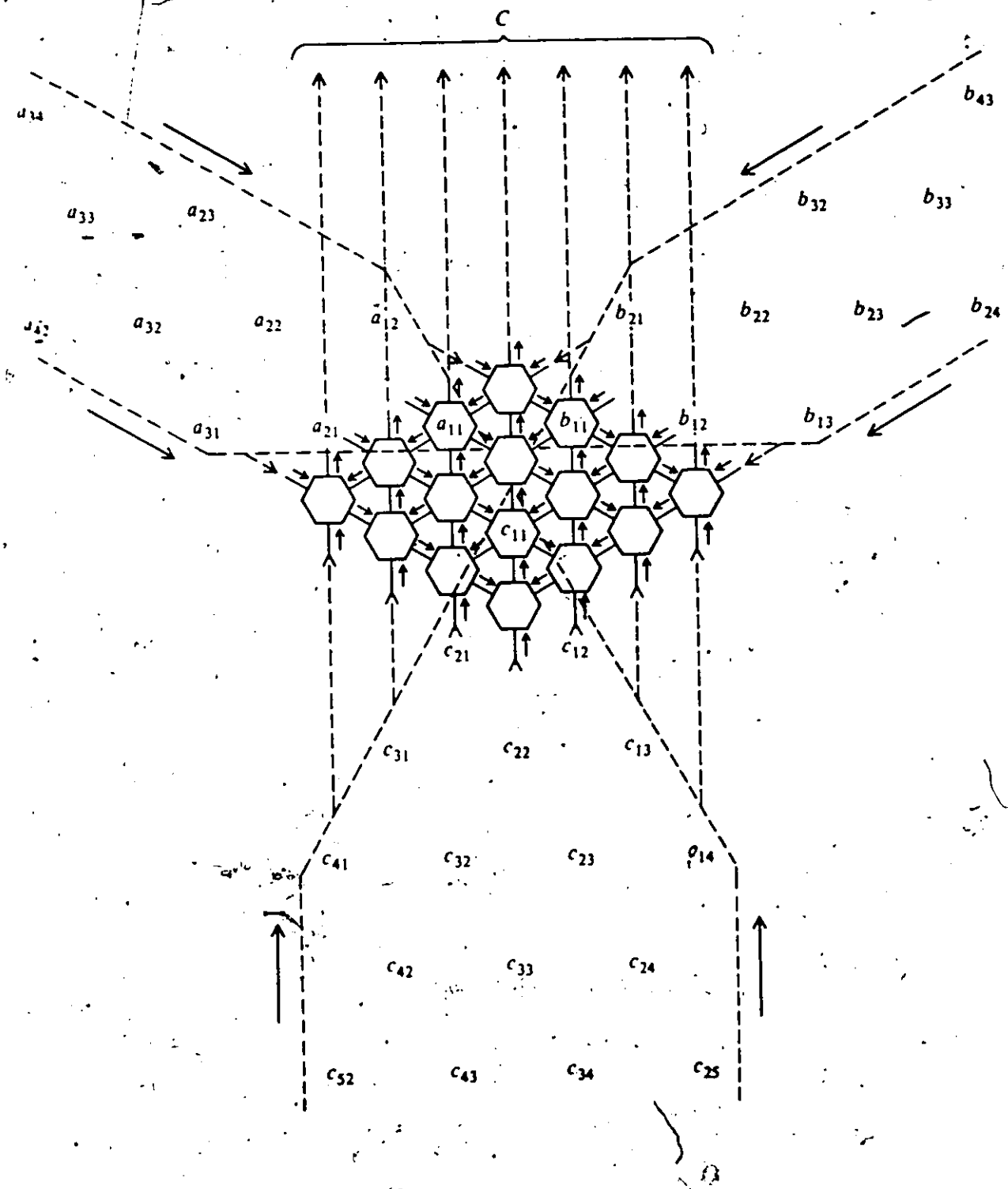


Figure 20. A Systolic Structure for Matrix Multiplication

6. Systolic Structure Specification

6.1 Introduction

In SADE, the first step of the design process is the definition of a cutting algorithm that places the targeted systolic structure onto the RSA. In the procedural mode, this cutting algorithm is coded as a MODULA-2 program that calls SADE's routines to actually embed the structure in an RSA which may be subsequently archived.

This chapter introduces these cutting routines and illustrates their use via several examples. The basic activation mechanism of SADE is discussed in section 6.2. The programming techniques used for inter-module communication and recursion elimination are studied in section 6.3. SADE's main cutting and archiving routines are then described in section 6.4. Subsequent sections refer to several simple examples (that were run on an APPLE-IIe microcomputer) to display SADE's versatility and typical program organization.

6.2 Basic Principles

The RSA is a square of hexagonally-connected cells; a cell and its links are shown in figure 11.

The cells on the boundaries of the RSA are assumed to be directly connected to invisible I/O ports; therefore, certain links of these boundary cells will carry an output to (or an input from) the outside world when appropriate. For example, the cell in the upper left corner of an RSA will generate an output if data flows out of its links 1,5 or 6.

To simplify the expression of the cutting algorithm as a MODULA-2 program, a system of planar coordinates is imposed on the RSA; according to this system, the corners and boundaries are identified as follows:

Corners:

UL:Upper Left, UR:Upper Right, LL:Lower Left, LR:Lower Right

Boundaries:

TB:Top, RB:Right, LB:Left, BB: Bottom

For simplicity, the cutting process is always initiated from one of the four corners.

Since a user-defined structure does not necessarily activate the entire RSA, a number of cells in the RSA might not be part of the structure and thus play only a 'passive' role in the algorithm's execution. As explained in section 2.2, these passive cells may however act as relayers of data; in fact, since i/o is performed only on the boundary cells of the RSA, the i/o data must be routed from the active cells to the I/O ports (and vice versa) through passive ones. We shall refer to those passive carriers as **relayers**. The size of the RSA, which directly controls the number of relayers, is interactively specified at the start of the execution of the cutting algorithm.

During the cutting process, some links will become **in-edges**, others **out-edges**. Roughly speaking, data flows in and out of a cell through an in-edge and an out-edge respectively; any link can be simultaneously both an in-edge and an out-edge. The out-edges are activated (i.e. specified as such) by the designer; the in-edges can be automatically activated by SADE, or may be explicitly listed by the designer.

Recall that the cutting algorithm is itself a systolic algorithm: the cells of the structure are activated in a distributed fashion. The host computer first creates a packet containing the information used by each cell to establish its own configuration. The host then activates the first cell of the structure by sending it this packet. Each cell that receives a packet follows the steps listed below:

1. Consult the information of the received packet to select your own activated links and cell type.
2. Select the adjacent cells (i.e. neighbours) that must be activated.
3. Send a packet with the appropriate updated information to each selected neighbour.

This description of the cutting strategy is direclty duplicated in the organization of the MODULA-2 program that represents it. In the main program, the designer creates the initial packet, chooses the starting corner cell and initiates the cutting process which is controlled by SADE. The designer then needs to code the procedure (henceforth called the configuration procedure) that describes the algorithm that each activated cell uses to establish its configuration. This procedure proceeds directly from the steps mentioned above and requires one parameter which gives access to the packet of information from which the configuration is derived.

6.3 Implementation Considerations

The technique used to code a cutting algorithm is quite simple but requires a specifit feature of MODULA-2 to handle inter-module communication. An a priori discussion of some relevant implementation details should prevent

the reader from feeling SADE does not lead to easier algorithm designing.

SADE obviously must not import any data objects from the designer's program if it is not to be recompiled for each cutting algorithm! Thus the parameter, used to give the configuration procedure, access to its corresponding packet, must be of a predefined type. The information packet is implemented as a user-defined record which is therefore invisible to SADE. A pointer to this record still is an invisible type. In fact, the predefined type ADDRESS (which is compatible with any pointer type) is the only efficient solution. The parameter of the configuration procedure therefore denotes the system address of the packet.

Unfortunately, using an ADDRESS pointer leads to pecularities (due to strong-type checking) of the MODULA-2 language: one needs an assignment statement to store the packet-record pointer into a variable of type ADDRESS and another for the inverse operation. Some experience with MODULA-2 should correct this initial impression of futile complexity.

As for the inter-module control flow, we remark that the cutting process cannot be implemented using recursion, for the simple reason that even a small RSA will inevitably cause a stack overflow (on a 64K machine)!! SADE implements the cutting process as a linked list of events (which allows for the simulation of the inherent parallelism of this process): at each clock pulse a certain number of events occur. Each event, which denotes the activation of a specific cell, consists in some internal information that uniquely identifies this cell, the name of the procedure that is used to set the cell's configuration and finally, an address pointer to the packet used for this

cell.

Relayers are treated somewhat differently than other cells of the structure and are studied in subsection 6.4.4.

A clear understanding of the implementation details discussed in this section will greatly simplify the following study of SADE's cutting routines.

6.4 SADE's Modules and Routines

SADE is implemented using three different library modules:

1. SADECUT which contains all the routines used to cut a systolic structure out of an RSA (of size inferior to a system-dependent constant).
2. SADEIO which holds the archiving procedure used to store a 'cut' systolic structure into a file (on a diskette).
3. SADEALG which holds the routines used to retrieve a cut systolic structure (from the diskette), define the algorithmic behaviour of each of its cell types and execute the corresponding systolic algorithm(s).

When using a library module, the programmer only has access to the objects exported by the module. In this section, we shall study the data types and procedures exported by the first two modules of SADE. (Refer to the listing of SADECUT's definition module, found in Appendix 3, for an exact description of the parameter list of each procedure.)

6.4.1 BOUNDARIES and CORNERS

The two first objects exported by SADECUT are the types BOUNDARIES and CORNERS which implement the conventions adopted in section 5.2. Here are their definitions:

1. CORNERS=(UL,UR,LL,LR)
2. BOUNDARIES=(TB,LB,RB,BB)

6.4.2 Procedure CUTCELL

SADECUT uses an internal representation of the RSA to keep track of the cutting process. Within this model, each cell is denoted by a record which has a set of in-edges, a set of out-edges and a cell type. As we have mentioned earlier, SADE is able to extrapolate the in-edges of a cell from the out-edges of its neighbours. The procedure CUTCELL is called to cut the out-edges of a cell, to give it a cell-type (an integer) and to create an event that will cut one selected neighbour on the next clock pulse. The specification of the event requires:

(a) the link used to get to the selected neighbour

(b) the name of the configuration procedure to use for this neighbour

(c) the address of the corresponding updated packet.

For example the call CUTCELL((2,5),4,2,KIND4,T) requires SADECUT to perform the following list of actions:

1. Activate links 2 and 5 as out-edges of the current cell.
2. Store 4 as the cell-type of the current cell.
3. Use the internal position of the current cell to compute the position of the neighbour reached via its link 2.
4. Create an event, for the next clock pulse, that, when executed:
   (a) makes the neighbour of the previous step (accessed via link 2) the current cell.
   (b) calls the specified configuration procedure (KIND4), using the variable T (of type ADDRESS) as its parameter.

5. Insert this event in an internal list of events ordered by time.

So far, we have described a SADE cutting program where only one configuration procedure was used. However the designer may want to have different configuration procedures which could be, for example type-dependent. This explains why each call to CUTCELL requires the name of a configuration procedure.

6.4.3 Procedure STARTCUT

Recall that the cutting process starts in one of the four corners. The designer creates the initial packet and calls STARTCUT to commence the cutting. STARTCUT therefore requires the name of a configuration procedure, the address of its corresponding packet and the starting corner.

The first cell of the structure cannot have its boundary in-edges extrapolated by SADECUT since there is no surrounding cell from the out-edges of which, these in-edges could be derived. Thus, when calling STARTCUT, the user must explicitly list these boundary in-edges of the first cell!

The call STARTCUT(LR,(3),ACTIVATION,T) indicates that the starting cell is in the lower right corner of the RSA, that its link 3 is an in-edge, and that its configuration procedure is ACTIVATION with variable T as the parameter.

6.4.4 Procedure CUTREL

Recall that the role of a relayer is to carry some information towards a certain boundary. In SADE, all relayers that lie between an active cell and a boundary of the RSA can be specified by a unique call to CUTREL.

We first notice that only the in-edges of a relayer are needed to establish the complete configuration: to each in-edge corresponds an out-edge which is 180 ^{o} away. For example, if link 2 is an in-edge then link 5 is an out-edge. Secondly, we remark that we systematically use the same link to get from one relayer to the next one. Specifying the in-edges of the first relayer and the link it uses to get to the next one suffices to describe the complete chain of relayers.

The first relayer may be activated in two different ways:

1. In the configuration procedure, the current cell recognizes that it is a relayer, in which case it calls CUTREL indicating that the chain of relayers starts at the current cell.
2. The current cell recognizes that one of its neighbours will be a relayer, in which case the current cell calls CUTREL indicating that this neighbour is the first relayer.

Procedure CUTREL has three parameters. The first one gives the link use to get from one relayer to the next; the second one specifies the set of in-edges of these relayers; and finally the third one is a boolean value which indicates whether or not the current cell is a relayer.

For example, CUTREL(2,{2,5},FALSE) indicates that the neighbour, via link 2, of the current cell is the first relayer of a chain of relayers that goes to the right boundary (since link 2 is used to get from one relayer to the next one). Each of these relayers has links-2 and 5 as in-edges, and therefore, the same links as out-edges since 5 is the reciprocal of 2!

6.4.5 Function ONBOUNDARY

This is a boolean function that returns TRUE if the current cell is on the boundary specified as parameter; otherwise FALSE is returned.

6.4.6 Procedure ACTIVATE.

Sometimes the designer must activate a certain set of links without calling CUTCELL which effectively transfers the control to SADECUT. Procedure ACTIVATE serves this purpose. Its first argument is a set of links to activate, and the second one is a boolean variable which indicates whether or not the specified links are in-edges.

6.4.7 Procedure MarkState

This is a convenience routine that has an integer as its unique parameter. The procedure sets the cell-type of the current cell to this parameter.

6.4.8 Procedure DUMPCUT

Recall that SADECUT simulates the cutting process by a link list of events. It is possible to follow the cutting by calling the parameterless procedure DUMPCUT. This procedure simply displays the current activated links of the RSA.

6.4.9 Procedure SAVECUT

This is the only procedure of module SADEIO. It has one parameter which is the name of the file in which SADE is to archive the cut systolic structure. Obviously, the call to SAVECUT should be the last statement of

the main program.

As for implementation, the procedure requires almost every file-handling built-in of MODULA-2. These routines are found in a huge library module called File which needs to be imported, consequently consuming a good part of the (very limited) symbol table space. To maintain the extensibility of SADE, we placed the archiving procedure in an individual module.

6.5 Examples

6.5.1 A Simple Example

Consider the linear array shown in Fig. 21.a where the first and the last cell are each connected to an I/O port from which inputs are received and outputs are sent to. A SADE program to cut this structure out of an RSA is given below; the result of the execution of the program on an RSA of size 6 is shown in Fig 21.b where only the first row of the RSA appears. (Relayers are denoted by 'r'.)

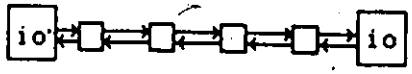


Figure 21.a A simple systolic array

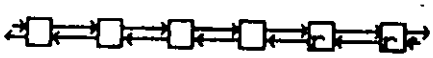


Figure 21.b Implementation of Fig 21.a

```
MODULE FOURCELL;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT ADDRESS;
FROM SADECUT IMPORT STARTCUT,CUTCELL,CUTREL,CORNERS;
FROM SADEIO  IMPORT SAVECUT;

TYPE
 PARMPTR = POINTER TO PARMREC;
 (* THIS IS THE RECORD USED TO HOLD THE INFORMATION USED BY ACTIVATION *)
 PARMREC = RECORD
            COUNT, NUMOFCELL: INTEGER;
            END;
VAR
 A: PARMPTR;
 INITIAL: CORNERS; (* CORNERS MUST BE IMPORTED FROM SADECUT *)

PROCEDURE ACTIVATION (T: ADDRESS);
(* This is the procedure that does most of the cutting job. It is recursively
   called via CUTCELL. Its parameter T is the system-defined ADDRESS type and
   gives the address of the record which contains the information used by the
   procedure. This is necessary because SADECUT does not 'see' the declaration
   of that record.  *)

VAR P: PARMPTR;
BEGIN
 P:= T;
 WITH P^ DO
  (* the algorithm is simple: if the current cell has a count greater than
     NUMOFCELL than we must cut a relayer in which case we use CUTREL.Otherwis
     we increment the cell count and call CUTCELL to cut the next cell which i
     reached via link 2 of the current cell.
     (Please check syntax and semantic of CUTCELL and CUTREL in PART II.)  *)
  IF COUNT> NUMOFCELL
   THEN CUTREL(2,(2,5),TRUE);
    ELSE COUNT:= COUNT+1;
         T:= P;
         CUTCELL((2,5),1,2,ACTIVATION,T);
  END; (* OF IF *)
 END (* OF WITH *)
END ACTIVATION;

BEGIN (* FOURCELL *)
 NEW(A);
 WITH A^ DO
   COUNT := 1; NUMOFCELL:= 4; (* we decide to cut four cells *)
 END;
 INITIAL:= UL;                  (* start in Upper Left corner  *)
 STARTCUT(INITIAL,(5),ACTIVATION,A);
 SAVECUT('FOUR.CUT');           (* save the structure in file FOUR.CUT *)
END FOURCELL.
```

Overview of the algorithm:

The cutting process starts in the Upper Left (UL) corner and the first cell has its link 5 directly activated as an in-edge since it cannot be extrapolated by subsequent activations. In the main program, once the cutting process is terminated, the designer saves the resulting structure in file FOUR.CUT using procedure SAVECUT.

The cutting process is initiated by STARTCUT which sets up the first call to the configuration procedure, called ACTIVATION. This procedure first translates the address of the packet (variable T) in a packet-pointer (variable P) and then checks the current cell count against the upper limit to see if the current cell is still part of the systolic structure. If the cell-count (field COUNT in the packet) is greater than the upper limit (field NUMOFCELL), the designer recognizes that the current cell is a relayer and calls CUTREL to activate the relayers, up to the right boundary of the RSA. The third parameter of CUTREL is set to TRUE to indicate that the current cell is a relayer.

If the current cell is not a relayer then the designer calls CUTCELL, activating links 2 and 5 as out-edges, giving a cell-type of 1 and selecting the cell to the right of the current one (link 2). At the next clock pulse, this selected neighbour will call the configuration procedure with an updated packet where the cell-count has been incremented by 1.

6.5.2 The General Linear Array

on the cell-type of the considered cell; each branch of the CASE calls the behavioural procedure corresponding to the selected cell-type. At each clock pulse, SADE goes through every cell of the structure and calls the cell driver of each cell to select and execute its behavioural procedure.

Both of the above items must be stored in the internal model of each active cell; this cell model consists of a record which holds all five items described in this section, as well as system-dependent information. To avoid the recompilation of SADEALG for each systolic algorithm, these five items must be of predefined types (cf. section 5.3). This imposes some constraints on the organization of the MODULA-2 program that implements an algorithm's definition:

As mentioned above, a step is required to allocate some (heap) memory for the local variables of each cell. Each active cell has, in its internal model, a field that gives the address of the record that contains its state variables. Thus the designer must code a function that allocates space for the different kinds of records of local variables, according to the cell-types of the structure. This function is organized as a CASE statement on the cell-type passed as parameter. Each branch of the CASE uses the built-in NEW to allocate space for the appropriate record. The function returns the address of the allocated record. At the start of the execution, SADE will refer to this allocation function to set up the local variables of each active cell of the RSA.

At each clock pulse, all active cells call their cell driver to select and execute 'simultaneously' the behavioural procedure associated with their

cell-type. Each cell calls its cell driver, passing the address of its state variables as a parameter. By accessing the record pointed to by the parameter, the designer is able to refer to the identifiers he chose for the local variables. This is extremely helpful in that it ensures the internal model is completly invisible to the designer!

Finally, recall that SADE has some pipelining capabilities. Without going into the details of implementation we observe that, in our model, the pipelining of algorithms simply corresponds to the ability to change the cell driver of an active cell!! If algorithms are pipelined, then the first cells of the structure may use a new cell driver while other cells will still refer to the old driver. This justifies why each cell stores the name of its cell driver in its internal model.

It is possible to pass any procedure type as a parameter in MODULA-2, provided that the parameter list of the procedure is static [14]. Since the designer does not have access to SADE's internal model, he must call one of SADE's routines to store the name of a cell driver. This implies that all the cell drivers have the same parameter list which consists of one parameter of type ADDRESS (giving the address of the record of the state variables).

The next section presents the routines used to perform the different tasks studied above.

7.3 SADEALG's Routines

(Refer to the listing of the definition module for SADEALG, found in Appendix 5. for an exact description of the parameter list of each procedure)

7.3.1 Procedure INPUTON

The first task of the designer is to describe the communication interface between the RSA and the outside world. By default, the outputs generated by a systolic algorithm are routed to the the screen of the microcomputer. SADE displays the value of the output, the time at which it is generated and possibly the absolute coordinates of the cell that produces it. (The cell in the upper left corner has coordinates 1,1).

The inputs of a systolic algorithm may come from a file on a diskette or from the keyboard of the microcomputer. In both cases, the designer calls procedure INPUTON which has two parameters, the second of which gives the name of the input file. The first parameter of INPUTON is a link number. When SADE realizes that the input coming from a certain link lies outside of the RSA, it checks a table (of filenames according to links) set up by INPUTON, to decide where to get this input from. If the input is to be received via a link to which no filename has been associated, SADE prints an error message and stops the execution of the algorithm.

For example, the call INPUTON(5,'CONSOLE:') indicates that any exterior input which is to be received via a link 5, is to be obtained from the keyboard.

It is quite convenient to gather all the inputs of a systolic algorithm in a disk file. Recall that at each clock pulse, all active cells 'beat' (i.e. execute their behavioural procedure). In particular, this means that all

boundary cells that read an input, will in fact read one at each clock pulse! Furthermore, systolic algorithms frequently use a skewed input format [9,10,11]. Therefore a convention for a null input is required. In SADE, a period (".") as input is taken as a null input. The period is translated to a predefined constant MAXINT so that the designer may test for a null input in his behavioural procedures.

Finally, another symbol, the vertical bar ("|"), is used as a stop input which, when received, indicates the end of a systolic algorithm. This convention provides a clean mechanism to halt the execution of a systolic algorithm.

7.3.2 Procedure SetLocals

Recall that the designer must code a function that allocates memory space according to the cell-type passed as parameter. Each active cell of the structure will call this function to setup its state variables. However, it is SADE and not the designer, that performs this allocation step (since the designer does not have access to the internal model of the RSA). Thus, the designer simply calls SetLocals to transfer the control to SADE. SetLocals has one parameter which denotes the name of the aforementioned allocation function.

7.3.3 Procedure SetAlg

This procedure is quite similar to SetLocals. It transfers the control to SADE, passing the name of a cell driver as its parameter. SADE then stores this name in the internal model of each active cell.

7.3.4 Function CELLTYPE

Recall that the cell driver is organized as a CASE statement on the different cell-types of the structure. At each clock pulse, every cell of the structure calls its driver. Instead of passing the cell-type as parameter, SADE provides the designer with the function CELLTYPE which returns the cell-type of the current cell. In this way, the CASE statement simply uses CELLTYPE as the case selector (cf. example of section 7.4).

7.3.5 Procedure Execute

Procedure EXECUTE is called to start the execution of the systolic algorithm. It has one parameter which is a boolean value that indicates whether or not SADE's pipelining option is used. If only one systolic algorithm is to be executed, then this variable is set to FALSE.

7.3.6 Procedure SEND

In SADE, two primitives are used to describe the communications between the activate cells of the structure: SEND and RECEIVE.

Procedure SEND has the following format:

SEND(link:INTEGER,desc:MSGTYPE,value:MSGVALUE);

The first parameter gives the link used to reach the addressee of the message. As we have seen in section 5.3, the messages transferred from one cell to another consist of a data type descriptor (INT,REL,CHA or STR) and a compatible message value. The two last parameters of SEND are used to pass these items to SADE: the second parameter is the data type descriptor; the third is the corresponding message value. (Notice that the designer must

import SADEALG's MSGTYPE type for the second parameter of SEND)

For example, SEND(2,INT,reg1) sends a message denoting the integer stored in variable reg1, to the cell to the right neighbour of the current one.

7.3.7 Function RECEIVE

When receiving a message, a cell must store the message's value in one of its local variables. Furthermore, it is possible that a cell receives a message from a specific neighbour at time t, but does not, at time t+1. The format of RECEIVE accounts for these possibilities:

RECEIVE(link:integer;a:ADDRESS): BOOLEAN;

The first parameter gives the link from which the message is to be received. The second gives the address of the local variable in which the message should be stored. The designer simply uses the built-in function ADR to get this address. Finally, RECEIVE returns TRUE if a message was indeed received, FALSE otherwise. (The internal message model allows SADE to efficiently decide if there is a message on the specified link. Also, SADE will abort the execution of the systolic algorithm if a message sent is not received at the next clock pulse.)

For example, the call RECEIVE(5,ADR(rega)) means that the current cell may have a message coming on its link 5. If this message is present, it is to be stored in the local variable called rega. RECEIVE returns TRUE if the message is properly received, FALSE otherwise. Notice it is the designer's responsibility to ensure that the message value and the local variable are of a compatible type.

7.3.8 Functions CLOCK and IDLE

Two SADE primitives take care of the synchronization aspects of the systolic algorithm:

(a) The CLOCK function denotes a virtual global clock which starts at time t=0 and is incremented at each pulse. CLOCK returns the current value of t.

(b) IDLE is a function that "turns a cell off" for the current clock pulse. In other words, it indicates to the system that the cell currently considered is not active for the current clock pulse.

IDLE strictly plays an esthetical role: the designer calls it to explicitly show, in a behavioural procedure, that in certain cases, the cell does not perform any operation!

Use of this two primitives is quite straightforward and is best illustrated by an example.

7.3.9 Other features

One fundamental characteristic of SADE is that it is easily extensible. Additional features, currently under development, include:

1. Provision for data transfers at different speeds and self-time computations. This implies a redefinition of SADE's timing mechanism.
2. Layouts by strata [13]. Essentially this means overlapping several sets of links on the same cell. Each stratum has its own sets of in- and out-edges.

3. Combinations of cut structures [10,11]. SADE would require a mechanism to define the interface between the two structures (each stored on an individual RSA) and some geometrical tool that would specify the topology of the whole architecture.

7.4 An example: a priority queue

In this section, the above concepts are illustrated by means of a concrete example: a priority queue definition [11].

7.4.1 A simple priority queue

(For a complete discussion of the priority queue refer to C.E. Leiserson's thesis.)

Many programming applications require the ability to insert records into a set, and at any time to retrieve from the set the record having the smallest key according to some linear ordering. Any data structure that provides such services is called a priority queue.

Priority queues are usally implemented in software, but a priority queue can be built in hardware as a systolic array. One method uses identical processors, each of which is capable of sorting three elements. The three-sorter has three inputs X,Y, and Z and produces three outputs X',Y', and Z' which are the minimum, median, and maximum of the inputs. Figure 25 shows how three-sorters are interconnected to make a systolic priority queue. In the figure, the outputs from the top, middle, and bottom of a processor are respectively the minimum, median, and maximum of the inputs. The minimum from each processor is fed leftward, and the median and maximum are fed rightward. This tends to keep smaller elements on the left and larger ones

on the right. An infinite key which is larger than all other keys is provided as constant input on the right. The two inputs on the left are connected to the host computer. Initially, all elements in the queue have an infinite key. As elements are inserted on the left, they move rightward displacing infinite keys which are output on the right. If ever the outputs on the right are not infinite keys, the queue overflows.

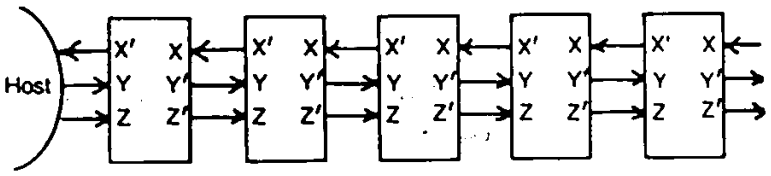


Figure 25. A Systolic Priority Queue.

Several steps of the operation of the systolic priority queue are illustrated in figure 27. The figure shows data on the wires between processors. The first column shows the communication between the host and the first processor on successive clock pulses, the second column shows the communication between the first and second processors, and so forth.

Notice that on even clock pulses, only odd-numbered processors do useful work, and that on even clock pulses, these cells are idle.

A SADE program to implement this priority queue and its operation is given in the following pages.

		−5	·	∞	·	∞
INSERT 6	6	·	−2	·	∞	·
	−∞	·	4	·	∞	·
	−∞	·	−2	·	∞	·
	·	−5	·	4	·	∞
	·	6	·	∞	·	∞
	·	−5	·	4	·	∞
EXTRACTMIN	∞	·	−2	·	∞	·
	∞	·	6	·	∞	·
	−5	·	−2	·	∞	·
(−5)	·	∞	·	4	·	∞
	·	∞	·	6	·	∞
	·	−2	·	4	·	∞
INSERT 3	3	·	∞	·	6	·
	−∞	·	∞	·	∞	·
	−∞	·	4	·	6	·
	·	−2	·	∞	·	∞
	·	3	·	∞	·	∞
	·	−2	·	6	·	∞
EXTRACTMIN	∞	·	3	·	∞	·
	∞	·	4	·	∞	·
	−2	·	3	·	∞	·
(−2)	·	∞	·	4	·	∞
	·	∞	·	6	·	∞

Figure 26. Execution of the priority queue

```
MODULE QUEUE;
FROM SADEALG IMPORT INPUTON,IDLE,CLOCK,SEND,RECEIVE,
                    CELLTYPE,MSGTYPE,SetLocals,SetAlg,EXECUTE;
FROM SYSTEM  IMPORT ADR,ADDRESS;
FROM Storage IMPORT ALLOCATE;

CONST
 MAXKEY= 800; (* A KEY GREATER THAN ANY ALLOWED KEY IN THE QUEUE *)

TYPE
 (* THE FOLLOWING DECLARATIONS ARE USED TO DESCRIBE THE LOCAL MEMORY
    ORGANIZATION OF THE CELLS OF THE STRUCTURE. THIS ALLOWS FOR USER-
    DEFINED NAMES IN THE ALGORITHM BELOW.                          *)
 LOCPTR1= POINTER TO LOCALS;
 LOCALS = RECORD
           MED,MAXI,MINI,TEMP: INTEGER;
          END;

PROCEDURE ALGORITHM(T:ADDRESS);
VAR P1:LOCPTR1;

PROCEDURE SORT(Q:LOCPTR1);
(* This procedure uses SEND and RECEIVE to implement the priority queue
   algorithm described in C.E. Leiserson's thesis [11]. We first receive the
   possible new minimum from the right neighbour. RECEIVE will return
   FALSE is there is no new minimum. Then we receive from our left neighbour
   the two other keys and we sort them. Finally we use SEND to transmit the
   results to our neighbours (see figure 13). Please check the thesis for
   the syntax and semantic of the procedures.                      *)
BEGIN
WITH Q^ DO
 IF NOT RECEIVE(2,ADR(MINI))
   THEN MINI:= MAXKEY;
 END;
 IF RECEIVE(5,ADR(MED)) AND RECEIVE(5,ADR(MAXI))
  THEN
   IF MINI > MED
    THEN TEMP:= MINI; MINI:= MED; MED:= TEMP;
   END;
   IF MINI > MAXI
     THEN TEMP:= MINI; MINI:= MAXI; MAXI:= TEMP;
   END;
   IF MED > MAXI
     THEN TEMP:= MED; MED:= MAXI; MAXI:= TEMP;
   END;
```

```
    SEND(2,INT,MED);SEND(2,INT,MAXI);SEND(5,INT,MINI);
   END
  END
 END SORT;

 PROCEDURE CELLA; (*BEATS ON EVEN PULSES *)
 BEGIN
  IF (CLOCK() MOD 2) = 0
    THEN SORT(P1)
  END
 END CELLA;

 PROCEDURE CELLB; (*BEATS ON ODD PULSES *)
 BEGIN
  IF (CLOCK() MOD 2)<>0
    THEN SORT(P1)
  END
 END CELLB;

 BEGIN (*OF ALGORITHM*)
  (* this is the typical organization of a systolic algorithm using SADE.
     The user simply makes a CASE statement on the different cell types
     contained in the systolic structure. In this example both cell types
     use the same memory layout but we repeated the assigment of T to a
     pointer to illustrate how T would just have to be assigned to the
     different memory pointers.                                    *)
  CASE CELLTYPE() OF
   15: P1:= T; CELLA;
   14: P1:= T; CELLB
  END;
 END ALGORITHM;

 PROCEDURE MEMORY(STATE:INTEGER): ADDRESS;
 (* this procedure allocates memory to a cell of the structure according
    to its cell type which is passed as a parameter          *)
 VAR P1: LOCPTR1;
 BEGIN
  CASE STATE OF
   14: NEW(P1); RETURN P1;
   15: NEW(P1); RETURN P1
  END;
 END MEMORY;

 BEGIN (* OF QUEUE *)
  (* the input on the link 5 of the leftmost cell will be read from the
     keyboard of our Apple... File I/O is possible when the user supplies
     a valid filename as parameter to INPUTON                  *)
  INPUTON(5,'CONSOLE:');
  SetLocals(MEMORY); (* allocate local memory to all cells in the structure *)
  (* SADE allows for pipelining of algorithms, that is a cell may change its
     algorithm from one set of inputs to the other. The call to SetAlg tells
     the system which procedure holds the first algorithm. Then EXECUTE is
     called to start the execution. Its parameter indicates the pipelining
     option is used or not. Please refer to the thesis for a full discussion *)
  SetAlg(ALGORITHM);
  EXECUTE(FALSE);
 END QUEUE.
```

7.4.2 Overview of the algorithm

The comments in the program provide a detailed description of the algorithm. Therefore, this subsection will only consists of a few brief remarks:

1. In this program, procedure ALGORITHM is the cell driver and CELLA and CELLB act as the behavioural procedures. MEMORY is the memory-allocation function; in this example we use the same kind of record for the local variables of both even- and odd-numbered cells (which respectively have cell-types 14 and 15) therefore only one pointer (P1) is required.

2. The designer could had explicitly shown that cells of type 15 are idle on odd clock pulses by adding an 'ELSE IDLE' to the IF statement of procedure CELLA.

3. The constant MAXKEY represents the infinite key in the algorithm.

7.5 Pipelining

One of the fundamental ideas of the programmable systolic architecture is that it should be possible to:

1. execute a systolic algorithm with several different sets of input data.
2. execute several different systolic algorithms on a same systolic structure.
3. execute different systolic algorithms on different structures cut out of a same RSA.

This section explains how SADE can achieve this.

7.5.1 Pipelining of Inputs

In SADE, two different approaches can be used to execute several different sets of inputs on a same systolic algorithm. The first method consists in simply placing these sets of inputs one after the other in the input file! It is possible that the sets would have to be separated by some null inputs to respect the period of the algorithm. In the second approach, at the end of each set of inputs, the designer simply redefines the input file(s) by calling INPUTON. Both methods are straightforward and have been tested with the priority queue example.

7.5.2 Pipelining Systolic Algorithms for A Same Structure

As mentioned earlier, it is possible to execute several systolic algorithms, one after the other, on a same cut structure. Recall that it suffices to update the internal cell driver field of each cell to do this. To indicate the pipelining of algorithms, the designer must first call EXECUTE passing TRUE as parameter. This tells SADE that as soon as a cell receives a stop signal, it should update its cell driver to the next one found in a list of systolic algorithms; this list is interactively created when SADE processes the call to EXECUTE(TRUE).

7.5.3 Different Algorithms on Different Structures

Executing different algorithms on different systolic structures is the most complex pipelining technique possible with an RSA. It takes full advantage of the fact that the processors of the RSA are programmable, and thus that

the RSA is reconfigurable. In SADE, this technique requires the ability to alternate the execution of systolic algorithms with that of 'cutting' algorithms: the designer cuts a first structure and executes a few algorithms on it; then he executes a second 'cutting' algorithm to modify the configuration of the RSA and directly uses this new structure to execute some new algorithms, and so forth. Alternating between the cutting and the simulation processes is possible since cutting algorithms are systolic algorithms.

In SADE, to achieve this kind of pipelining, the designer must:

1. code all the cutting algorithms in separate modules.
2. code all systolic algorithms that use a same structure in a unique module.
3. write a command file which consists of pairs of names: The first name of a pair refers to the module containing a cutting algorithm; the second indicates the module that contains the systolic algorithms to be executed on the corresponding structure.

The command file is a simple representation of the desired pipelining process; SADE includes a program that can interpret this command file and simulate it.

8. Conclusions

8.1 Implementaton of SADE

The current prototype of SADE is implemented as a set of MODULA-2 library modules. This 'new' high-level language was chosen for several reasons:

1. a complete implementation of the language exists on several microcomputers.
2. the language allows for separate compilation, strong-type checking and the passing of any type of procedure as a parameter.
3. modular design and coding makes SADE a tool which is very easily extensible.
4. MODULA-2, under the USCD PASCAL system offers important advantages:
   (a). SADE can use the TURTLEGRAPHICS package to implement the graphical cutting of systolic structures, as well as the graphical simulation of systolic algorithms.
   (b) This entire MODULA-2 system is highly portable: the whole tool can be ported to another system provided that a P-code interpreter is available.

SADE's procedural mode can also be quite easily implemented in LISP and ADA! However the resulting programs do not offer MODULA-2 elegant organization and can't refer to the TURTLEGRAPHICS package which implements SADE's graphical mode.

8.2 Graphical Definition Mode

As mentioned earlier, SADE allows the systolic structure to be specified via an interactive graphical package. In this mode, the specification process proceeds as follows:

1. an RSA of user-defined size is displayed
2. Via a light-pen, the user cuts the desired structure by associating types with the active cells
3. for each cell-type, a basic cell (see Fig. 20) is displayed and the user, via the light-pen, activates the links as desired.

The system will automatically determine the entire topology as well as the relayers. The algorithm definition step is still carried out in a procedural mode. The execution of a systolic algorithm can be displayed on the screen if so desired.

8.3 Testing Facilities

SADE's internal model contains enough information in its data structures so that, when simulating a systolic algorithm, the designer can gather valuable testing results. Moreover, SADE will automatically measure complexity metrics such as: period, area and time.

8.4 Conclusions

In this second part of our thesis we have presented the specifications for a tool that allows for the design and testing of systolic algorithms on a programmable systolic architecture (RSA). In particular, we have illustrated how the designer can specify a systolic structure within an RSA

and define several pipelined systolic algorithms to perform on this structure. Furthermore, we have briefly studied how an RSA could be reconfigured so that different algorithms for different structures could be executed. The expression of systolic algorithms as high-level programs suggests that a standardization of design techniques for systolic algorithms is easily attainable. Finally, our tool seems to offer an interesting alternative to existing ad hoc simulation tools.

As a design environment, SADE is to be viewed as a component of a much larger VLSI CAD system as shown in Fig. 28. Its place within such a system would be as a front-end entity, i.e. between the user and a semantic analyzer that would extract the layout requirements from SADE's internal model of the RSA.

User -- SADE -- Semantic Analyzer -- Layout Package -- Fabrication

Figure 28. A VLSI CAD System

We are currently studying the interface between SADE and ALI [10]. As a matter of fact, combining ALI's powerful abstraction mechanisms with SADE's explicit description of the semantics of the algorithm could fill in the gap between theoritical design and realistic VLSI implementations.

References

[1] M.A. Bonucelli, E.Lodi, F. Luccio, P. Maestrini and L. Pagli: A VLSI Tree Machine for Relational Data Bases, *Proc. of the 10th Int. Symp. on Computer Architecture*, pp. 67-73, 1983

[2] R.P. Brent, H.T. Kung and F.T. Luk:Some Linear-Time Algorithms for Systolic Arrays, *TR 83-541 Dept. of Computer Science Cornell University* ,1983

[3] B. Chazelle and L. Monier: A Model of Computation for VLSI with Related Complexity Results, *Proc. of the Symposium on Theory of Computing*, pp. 318-325, 1981

[4] B. Chazelle and L. Monier: Optimality in VLSI, *TR CMU-CS-81-141 Dept. of Computer Science Carnegie-Mellon University*, pp. 318-325, 1981

[5] K. Culik, J. Gruska and A. Salomaa: Systolic Trellis Automata for VLSI, *Technical Report CS-81-34*, Dept. of Computer Science University of Waterloo, 1981

[6] A. Fisher, H.T. Kung, L. Monier and Y. Dohi: Architecture of the PSC: A Programmable Systolic Chip, *Proc. of the 10th Int. Symp. on Computer Architecture*, pp. 48-53, 1983

[7] D. Gordon, I. Koren and G. Silberman: Embedding Tree Structures in VLSI hexagonal Arrays, *IEEE Trans. on Computers*, Vol. C-31, pp. 892-897, September 1982

[7.a] D. Gordon, I. Koren and G. Silberman: Embedding Tree Structures in VLSI hexagonal Arrays, *Technical Report*, Technion Institute, Israel, 1982

[8] M. Kramer and J. van Leeuwen: Systolic Computation and VLSI, *Part 1: Foundations of Computer Science IV*, Mathematical Centre Tracts 158, Amsterdam, pp. 75-103, 1983

[9] H.T. Kung: Why VLSI Architectures, *IEEE Computer 15*, pp. 37-45, 1982

[9.a] H.T. Kung: On the Implementation and Use of Systolic Array Processors, *Proc. of IEEE Int. Conf. on Computer Design: VLSI in Computers*, pp. 370-373, 1983

[9.b] H.T. Kung and S.Q. Yu: Integrating High-Performance Special-Purpose Devices into a System, *VLSI Architecture*, Randel and Treleaven eds, Prentice/Hall International, pp. 205-211, 1983

[10] C. Mead and L. Conway: *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980

[11] C. Leiserson: *Area-efficient VLSI Computation*, MIT Press, Cambridge, MA, 1983

[12] R. Lipton, R. Sedgewick and J. Valdes: Programming Aspects of VLSI, *CACM, January 1982*, pp. 57-65, 1982

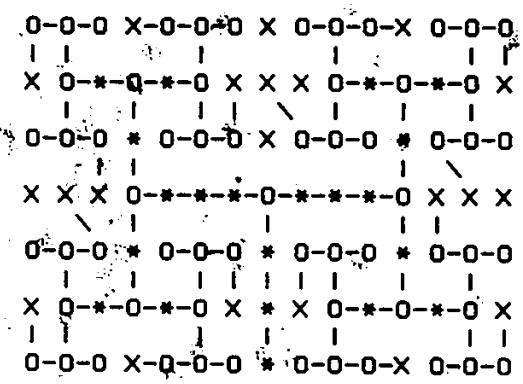
[13] T. Ottmann, A. Rosenberg and L. Stockmeyer: A Dictionary Machine, *IEEE Trans. on Computers*, Vol. C-31, pp. 892-897, September 1982

[14] J. Vuillemin: *A Combinatorial Limit to the Computing Power of VLSI Circuits*, Proc. 21st Annual Symposium on Foundations of Computer Science, October 1980

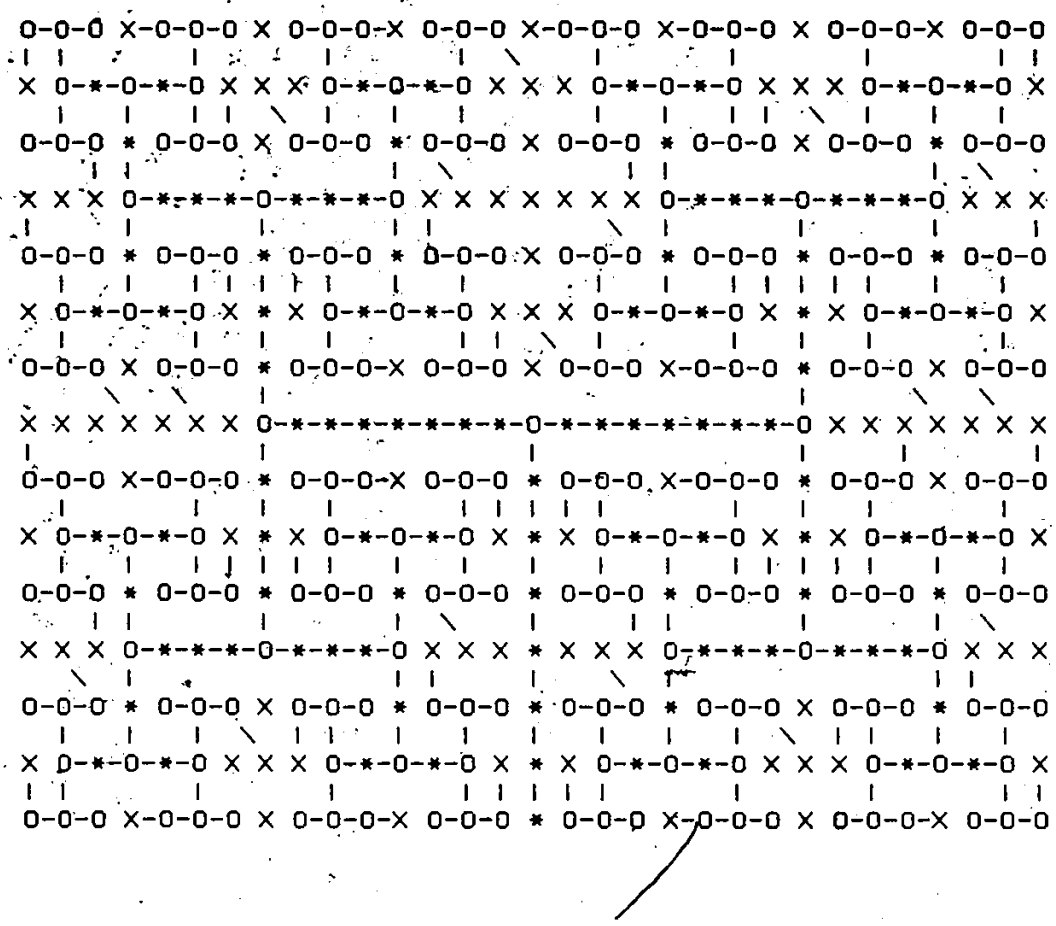
[15] N. Wirth: *Programming In MODULA-2*, Springer-Verlag, Berlin Heidelberg New York, 1983

APPENDICES

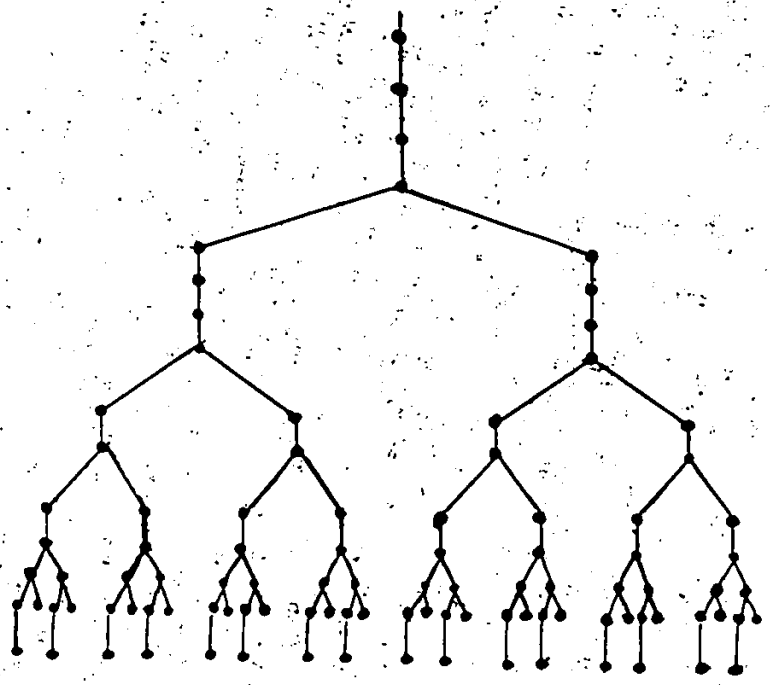
Appendix 1



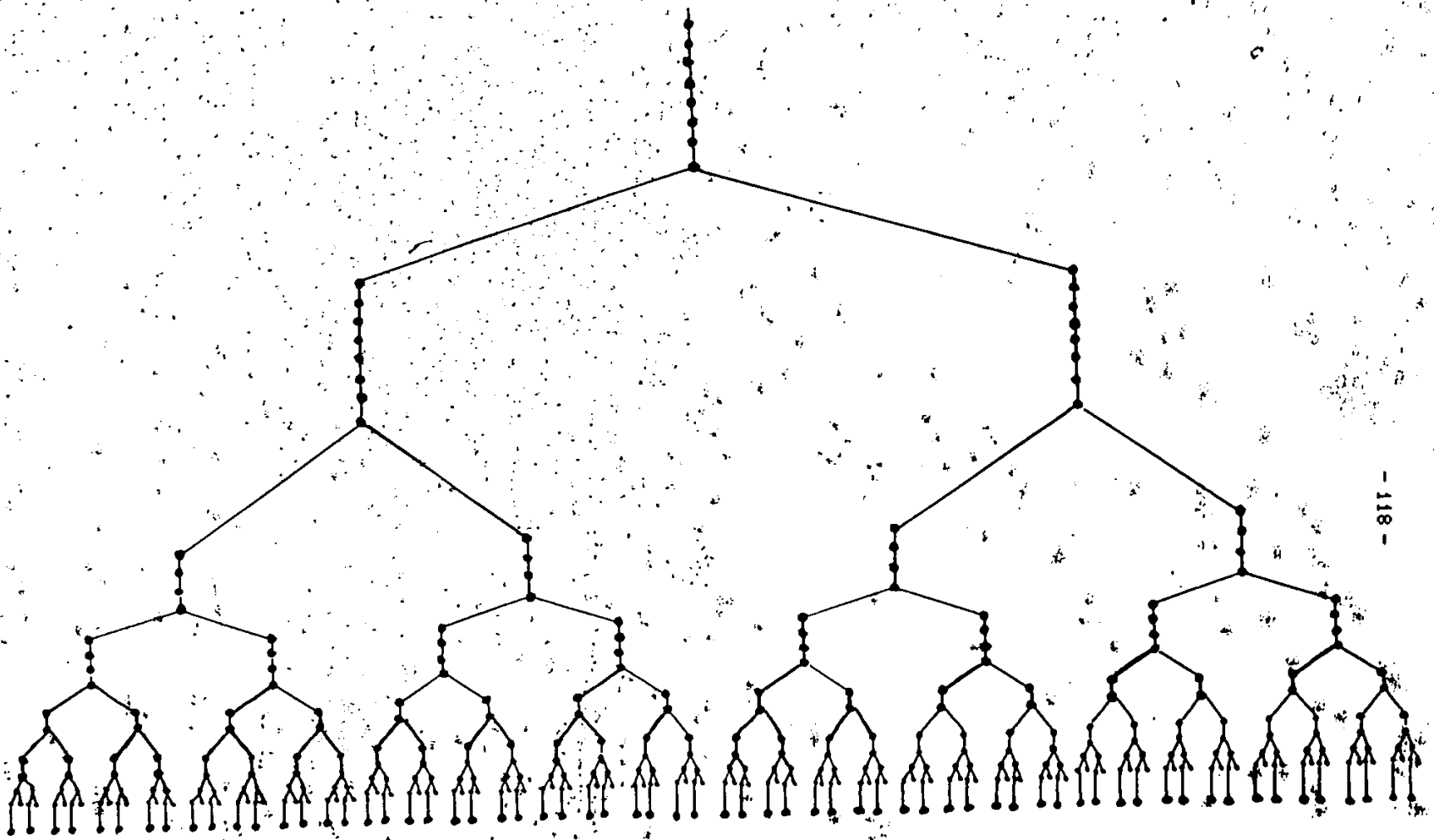
The waste reduction algorithm: final layout for 63 nodes.



The waste reduction algorithm: final layout for 255 nodes.

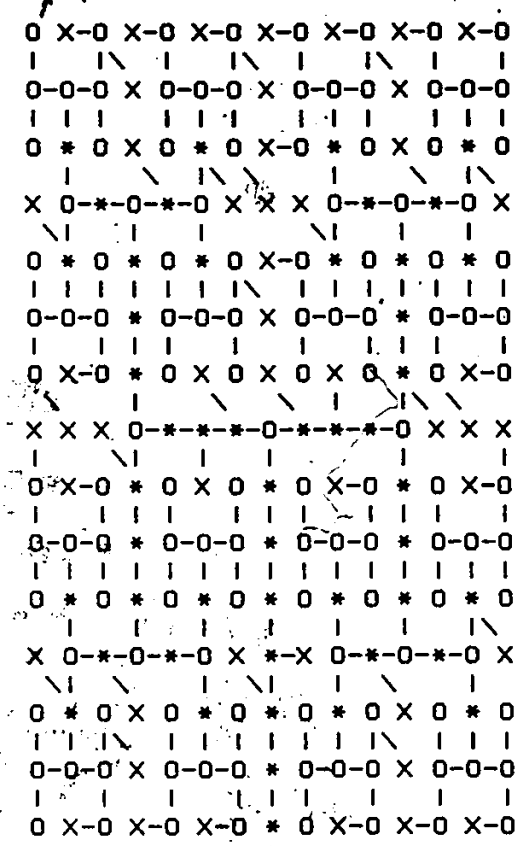


The waste reduction algorithm: final binary tree for 63 nodes.

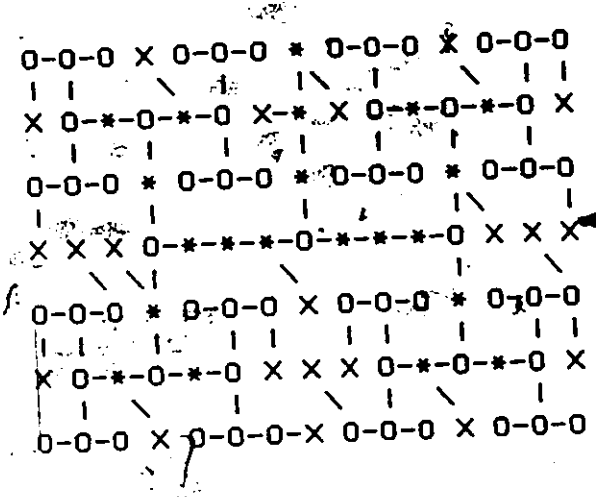


The waste reduction algorithm: final binary tree for 255 nodes.

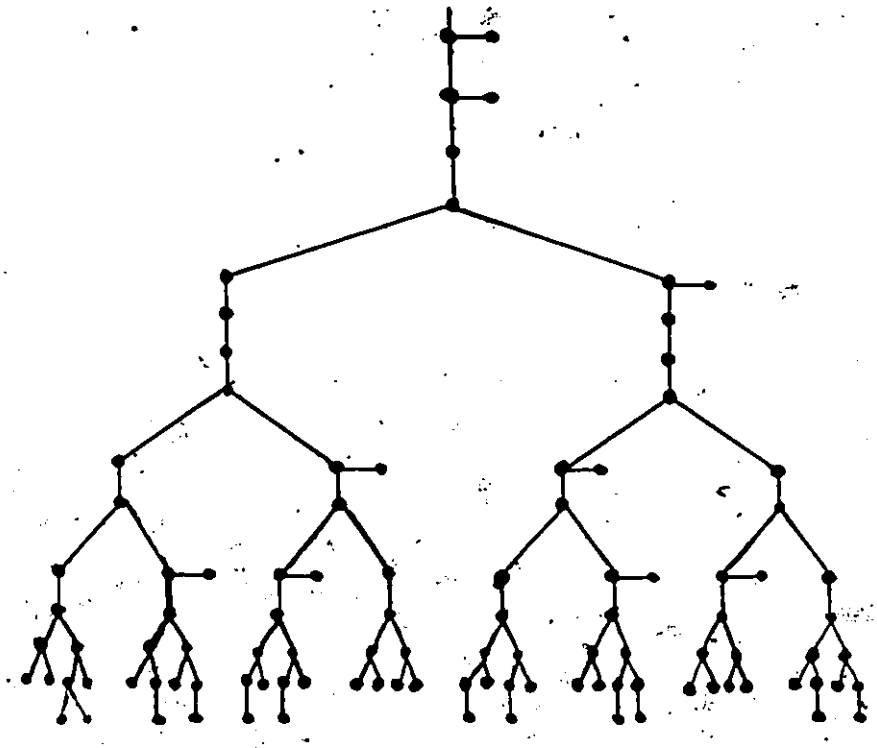
Appendix 2



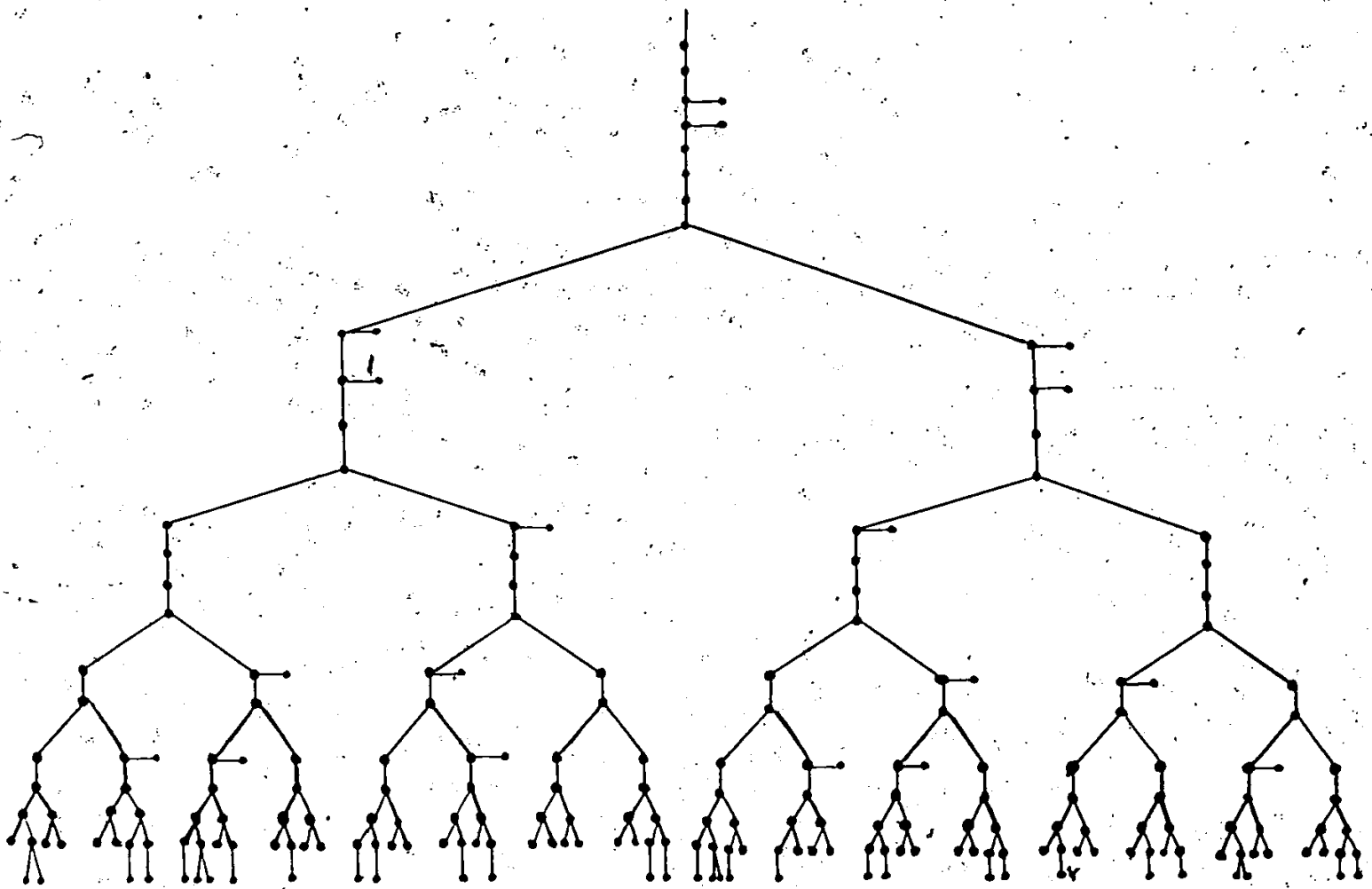
The waste elimination algorithm: final layout for 127 nodes.



The waste elimination algorithm: final layout for 63 nodes.



The waste elimination algorithm: final binary tree for 63 nodes.



The waste elimination algorithm: final binary tree for 127 nodes.

Appendix 3

```
DEFINITION MODULE SADECUT;
(* $SEG := 43; *)
FROM SYSTEM IMPORT ADDRESS;
EXPORT QUALIFIED STARTCUT,CUTCELL,DUMPCUT,CORNERS,CUTREL,ACTIVATE,ONBOUNDARY,
       IMAX,JMAX,UC,CELLULE,STOPCUT,RECPR,MarkState,BOUNDARIES;


TYPE
 CORNERS= (UL,UR,LL,LR);
 LINKS  = [1..6];
 BOUNDARIES= (TB,LB,RB,BB);
 EDGES  = ARRAY [1..6] OF BOOLEAN;

 PROCTYPE = PROCEDURE (ADDRESS);
 FILENAME = ARRAY [0..71 OF CHAR;
 CALLPTR = POINTER TO CALLREC;
 CALLREC= RECORD
            CURRI,CURRJ: INTEGER;
            CURPROC    : PROCTYPE;
            CURPARM    : ADDRESS;
            NEXT       : CALLPTR;
           END;

 CELLULE = RECORD
             INE,OUTE: EDGES;
             STATE   : INTEGER;
           END;
  K2 = RECORD
        A,B:INTEGER; I:EDGES;
       END;
  K1 = RECORD
        A:BITSET; S:INTEGER; L:LINKS;
        P:PROCTYPE; T:ADDRESS;
       END;
 EVENTS= POINTER TO ACTION;
 ACTION = RECORD
            RELAYER:BOOLEAN;
            NEXT   :EVENTS;
            TEMPS  :INTEGER;
            OLDI,OLDJ:INTEGER;
            CASE INTEGER OF
             0: PARM1: K1 |
             1: PARM2: K2
            END;
           END;
 CHIP    = ARRAY [1..8], [1..8] OF CELLULE;

VAR
 SYSI,SYSJ,SYSTIME: INTEGER; SYSEND:BOOLEAN;
 STARTCALL: CALLPTR;
 IMAX,JMAX:INTEGER;UC : CHIP;

PROCEDURE ONBOUNDARY(B:BOUNDARIES): BOOLEAN;
```

```
PROCEDURE MarkState (I:INTEGER);
PROCEDURE RECPR(I:INTEGER): INTEGER;
PROCEDURE ERROR(I:INTEGER);
PROCEDURE PRINTLINKS(E: EDGES);
PROCEDURE DUMPCUT;
PROCEDURE ACTIVATE(A:BITSET; INEDGE:BOOLEAN);
PROCEDURE STARTCUT(C:CORNERS; A:BITSET; P:PROCTYPE; T:ADDRESS);
PROCEDURE CUTREL(L:LINKS; INR:BITSET;IMMEDIATE:BOOLEAN);
PROCEDURE CUTCELL(A:BITSET; S:INTEGER; L:LINKS; P:PROCTYPE; T:ADDRESS);
PROCEDURE STOPCUT;
END SADECUT.
```

Appendix 4

```
Size of PSA?
4

Time 0

Cell number 1 1
state: 14   in-edges          out-edges 6

Time 1
Cell number 1 1
state: 14   in-edges 2 3 4    out-edges 6
Cell number 1 2
state: 14   in-edges 4        out-edges 5 6
Cell number 2 1
state: 14   in-edges 2        out-edges 1 6
Cell number 2 2
state: 14   in-edges          out-edges  1 5 6

Time 2
Cell number 1 1
state: 14   in-edges 2 3 4    out-edges 6
Cell number 1 2
state: 14   in-edges 2 3 4    out-edges 5 6
Cell number 1 3
state: 14   in-edges          out-edges 5 6
Cell number 2 1
state: 14   in-edges 2 3 4    out-edges 1 6
Cell number 2 2
state: 14   in-edges 3        out-edges 1 5 6
Cell number 2 3
state: 0    in-edges 3        out-edges 6
Cell number 3 1
state: 14   in-edges          out-edges 1 6
Cell number 3 2
state: 0    in-edges 3        out-edges 6
Cell number 3 3
state: 0    in-edges 3        out-edges 6

Time 3
```

```
Cell number 1 1
state: 14   in-edges 2 3 4   out-edges 6
Cell number 1 2
state: 14   in-edges 2 3 4   out-edges 5 6
Cell number 1 3
state: 14   in-edges 5       out-edges 5 6
Cell number 2 1
state: 14   in-edges 2 3 4   out-edges 1 6
Cell number 2 2
state: 14   in-edges 3       out-edges 1 5 6
Cell number 2 3
state: 0    in-edges 3       out-edges 6
Cell number 3 1
state: 14   in-edges 3       out-edges 1 6
Cell number 3 2
state: 0    in-edges 3       out-edges 6
Cell number 3 3
state: 0    in-edges 3       out-edges 6
Cell number 4 2
state: 0    in-edges 3       out-edges 6
Cell number 4 3
state: 0    in-edges 3       out-edges 6
Cell number 4 4
state: 0    in-edges 3       out-edges 6
```

Appendix 5

```
(* $RECYCLE:= TRUE; *)
DEFINITION MODULE SADEALG;
(* $SEG := 44; *)
FROM SADEIO IMPORT GETCUT;
FROM SYSTEM IMPORT ADDRESS;
FROM Files IMPORT FILE;
EXPORT QUALIFIED SetAlg,SetLocals,RECEIVE,SEND,BROADCAST,
                 MSGTYPE,MSGVALUE,CELLTYPE,IDLE,CLOCK,PIPELINE,INPUTON,EXECUTE;
TYPE
 LINKS  = [1..6];
 BOUNDARIES= (TB,LB,RB,BB);
 EDGES  = ARRAY [1..6] OF BOOLEAN;

 PROCTYPE = PROCEDURE (ADDRESS);
 PROCTYPE1 = PROCEDURE(INTEGER): ADDRESS;
 FILENAME = ARRAY [0..7] OF CHAR;

 MSGPTR = POINTER TO MESSAGE;
 MSGVALUE= INTEGER;
 MSGTYPE = (INT,REL,ARR,REC,CHA);
 OUTREC = RECORD
           I,J,T:INTEGER; V:MSGVALUE;
          END;
 MESSAGE= RECORD
           KIND: MSGTYPE; LINK:LINKS;
           VALUE: MSGVALUE; NEXT:MSGPTR;
          END;
 ALGPTR = POINTER TO ALGREC;
 ALGREC = RECORD
           PROC:PROCTYPE; NEXT:ALGPTR;
          END;
 CELLULE = RECORD
            INE,OUTE: EDGES;
            STATE   : INTEGER;
            LOCAL   : ADDRESS;
            CELLALG : ALGPTR;    CURMSG,NEWMSG:MSGPTR;
           END;
 CHIP    = ARRAY [1..8], [1..8] OF CELLULE;

VAR
 STOPVALUE,SYSI,SYSJ,SYSTIME,SYSSTOP: INTEGER;
 IMAX,JMAX:INTEGER;
 UC : POINTER TO CHIP;
```

```
 NOTABLE: ARRAY [1..6] OF FILE; INFILE,OUTFILE:FILE;
NOINPUT,FULLOUT : BOOLEAN;
INPUTFLAG: ARRAY [1..6] OF BOOLEAN;

PROCEDURE SetArg(P:PROCTYPE);

PROCEDURE CELLTYPE():INTEGER;

PROCEDURE PIPELINE(P:ALGPTR);

PROCEDURE BROADCAST(M:MESSAGE);

PROCEDURE RECEIVE(L:LINKS; T:ADDRESS):BOOLEAN;

PROCEDURE CLOCK ():INTEGER;

PROCEDURE IDLE;

PROCEDURE SEND(L:LINKS; K:MSGTYPE; V:MSGVALUE);

PROCEDURE SetLocals(PR1:PROCTYPE1);

PROCEDURE INPUTON(L:LINKS; NAME:FILENAME);

PROCEDURE EXECUTE(P: BOOLEAN);
END SADEALG.
```