A RECONFIGURABLE AND FAULT-TOLERANT VLSI MULTIPROCESSOR ARRAY

ISRAEL KOREN

DIGITAL INTEGRATED SYSTEMS CENTER REPORT
DISC/81-1

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY
HAIFA, ISRAEL

July 1981

This work was carried out in part at the University of Southern California, Electrical Engineering - Systems Department, and supported in part under a VHSIC Phase 3 Contract No. N00039-80-C-0641 administered by the Department of the Navy, Naval Electronics Systems Command (ONR), Pasadena, California.

A preliminary version of this paper was presented at the 8th Annual Symposium on Computer Architecture, Minneapolis, Minnesota, May 12-14, 1981.

ABSTRACT

Multiprocessor arrays have the desirable property of regularity, enabling a low-cost VLSI implementation. However, multiprocessor systems with a fixed structure tend to be error prone and restricted to very specialized applications, which makes them less attractive to the semiconductor industry. Consequently, reconfigurability and fault-tolerance are desirable features of a multiprocessor array. A multiprocessor array with a flexible structure is adaptable to many applications and may restructure itself upon failure of a processor, to avoid utilization of faulty processors.

The objective of this work is to demonstrate the feasibility of a multiprocessor array having these properties. An example of such an array is introduced, and distributed structuring algorithms for it are presented. Next, novel strategies for internal testing, for identification of faulty processors, and for avoiding the use of faulty processors are developed.

Index Terms: VLSI, multiprocessor array, reconfigurability, fault-tolerance, distributed structuring algorithm, distributed testing.

ACKNOWLEDGEMENTS

This work has benefited from valuable discussions with Mel Breuer, John Hayes, Ellis Horowitz, John Nelson and Alex Thomasian of the Electrical Engineering - Systems Department at the University of Southern California.

TABLE OF CONTENTS

I. INTRODUCTION

The current trends in the integrated circuit technology will undoubtedly have a considerable impact on the way multiprocessor systems are designed and implemented. First, the decreasing cost of hardware components will enhance the design of special-purpose multiprocessor systems [1,2]. Second, the exponential increase in the gate count per chip which is expected to keep its current pace for at least five to ten years, will enable the packaging of an increasing number of processors into a single VLSI chip. Clearly, such packaging is preferred over the use of several LSI chips when power consumption, speed and reliability are considered.

However, the new VLSI technology does have limitations that should not be overlooked. The man-year effort when designing a high-density VLSI chip is expected to rise significantly, resulting in a substantial increase in manufacturing cost. One way to reduce the expected increase in manufacturing effort and cost is to design multiprocessor arrays having the desirable property of regularity. Regular structures take considerably less time to design and manufacture [1-3]. Yet, even regular multiprocessor arrays might still be unattractive to the semiconductor industry due to their tendency to have restricted uses, resulting in a low sales volume. Consequently, a VLSI multiprocessor array with a flexible structure is desirable since it can be configured in several ways thus increasing the

I. INTRODUCTION

The current trends in the integrated circuit technology will undoubtedly have a considerable impact on the way multiprocessor systems are designed and implemented. First, the decreasing cost of hardware components will enhance the design of special-purpose multiprocessor systems [1,2]. Second, the exponential increase in the gate count per chip which is expected to keep its current pace for at least five to ten years, will enable the packaging of an increasing number of processors into a single VLSI chip. Clearly, such packaging is preferred over the use of several LSI chips when power consumption, speed and reliability are considered.

However, the new VLSI technology does have limitations that should not be overlooked. The man-year effort when designing a high-density VLSI chip is expected to rise significantly, resulting in a substantial increase in manufacturing cost. One way to reduce the expected increase in manufacturing effort and cost is to design multiprocessor arrays having the desirable property of regularity. Regular structures take considerably less time to design and manufacture [1-3]. Yet, even regular multiprocessor arrays might still be unattractive to the semiconductor industry due to their tendency to have restricted uses, resulting in a low sales volume. Consequently, a VLSI multiprocessor array with a flexible structure is desirable since it can be configured in several ways thus increasing the

number of possible applications. Clearly, some of the processors will not be utilized in several applications but this should not be considered a serious drawback since on cost alone processors will be the expendable components in VLSI technology [3].

Another desirable feature of a VLSI chip is fault-tolerance. Future improvements in the solid-state technology and maturity of the fabrication process are projected to reduce the failure rate of a single component. The failure rate predicted for high-density VLSI chips, however, is still larger than that of present LSI chips due to the exponential increase in complexity. Hence, frequent verification of the multiprocessor's proper operation is essential. Unfortunately, this verification cannot be achieved by external testing of the VLSI chip because of the inability to access internal points. Thus, internal testing (i.e., processors testing one another) for error detection is necessary. Moreover, identification of faulty processors provides the ability to tolerate certain faults and remain operational in the presence of one or more faulty processors even with some degradation in performance. Upon occurrence of a fault in one of the processors the faulty processor should be identified and the array possibly restructured to avoid the faulty processor being used. The multiprocessor array, therefore, should be capable of dynamic restructuring that takes place whenever a faulty processor is identified. An array with such a property is superior over a fixed structure multiprocessor (e.g., a binary tree) which is sensitive to errors and in which a loss of a single processor usually results in a system failure. The

restructuring capacity may also enhance larger chip areas (wafer-scale integration [1]) now prohibitive because of defects that exist in large wafer areas.

The main objective of this paper is to demonstrate the feasibility of a multiprocessor array with the properties mentioned previously. As an example, we consider in the following a rectangular $m \times n$ grid as depicted in Figure 1. This grid may be structured as a linear array, a square array, a loop or a binary tree. Not all processors will necessarily be utilized in each of these configurations; however, in VLSI technology, processors are the expendable components. The transistor count of each processor in a special-purpose VLSI chip is predicted to be approximately 1K to 10K while the projected complexity of VLSI chips is 1000K and over. Consequently, a grid consisting of 1000 or more processors will be feasible. In Figure 1 there are $m \cdot n$ cells in the grid; each cell is connected to its immediate neighbors denoted N - neighbor, E - neighbor, S - neighbor and W - neighbor. The basic cell consists of an application processor and a communication processor as shown in Figure 2. The application processor may be a microprogrammable processor that can be programmed by the customer and tailored to his special needs. The communication processor is responsible for the structuring of the processor array. The complete array communicates with the outside world through an external processor which initiates the various structuring algorithms as well as the computational algorithms.

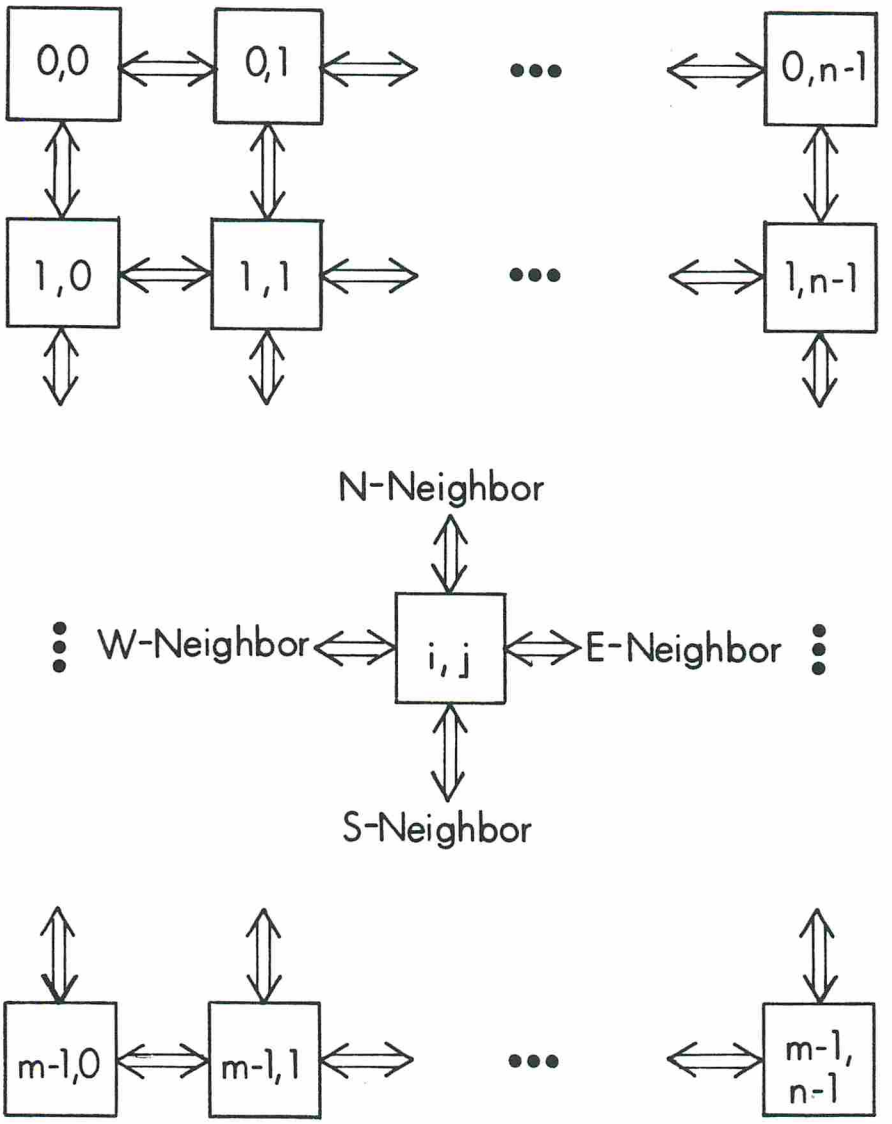


Figure 1. A rectangular m × n grid of processors

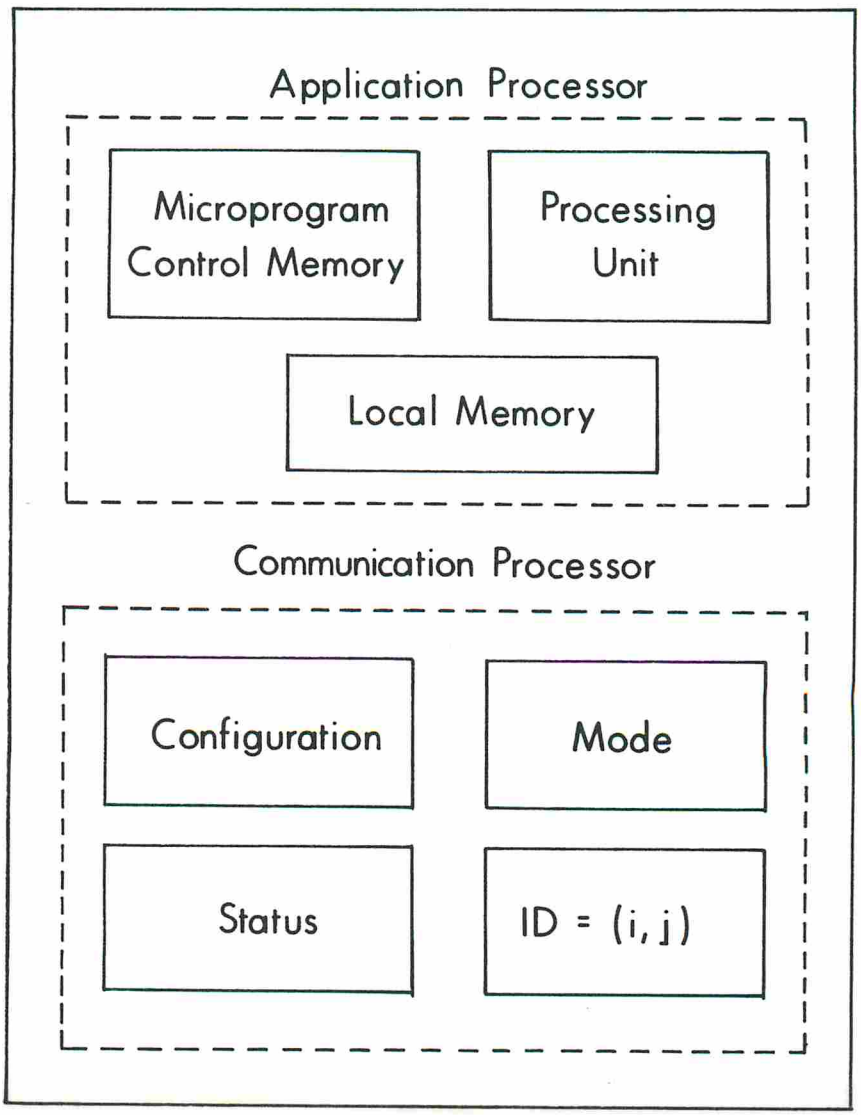


Figure 2. The basic processing unit.

The paper focuses, in what follows, on the communication processor within which four registers are defined (Fig.2). The ID register contains the row and column indices. THe Mode register indicates the cell's mode of operation. Each cell can be either a processing element (PE) participating in the processing in a given configuration or a connecting element (CE) transferring data when needed from N to S, from E to W and vice versa without performing any kind of processing. The Status register stores the status of the immediate neighbors. If the d-neighbor (i.e., the immediate neighbor in the d-direction where $d \in \{N,E,S,W\}$) is either faulty or does not exist, we say that the cell is d-terminal. In this case, no attempt to communicate with the d-neighbor will be made. The Configuration register (CR) contains information about the present structure, the level number of the cell within the array (such as linear array or binary tree) and the directions of the predecessor (D_I) and successor (D_O) cells.

II. BASIC STRUCTURING ALGORITHMS

Since the multiprocessor system has no fixed structure, structuring algorithms for the various configurations are needed. First, structuring algorithms are presented that are based upon the assumption that all processors are operating correctly.

To achieve optimal use of a multiprocessor system, most processing should be distributed rather than centralized. Similarly, structuring algorithms should be distributed. Such a property is essential if the system is to operate in the presence of faulty processors as will become apparent in Section III. Obviously, simplicity of the structuring algorithms is desirable in order to reduce the chip area used for them and, thus, increase the size of the array.

Structuring within the grid is done by distributing the following type of messages:

M(structure code, level within structure, direction).

The header of the message M is the code of the desired structure, such as LA for linear array, SA for square array, LP for loop and BT for binary tree. The level number within the array indicates the position of the processor receiving the message M in the array. The direction d ($d \in \{N,E,S,W\}$) indicates the neighbor to which the next structuring message should be transmitted. Since changes in the direction of transmission are necessary, we find it convenient to define the functions,

opposite (op), clockwise (cw) and counterclockwise (ccw) as:

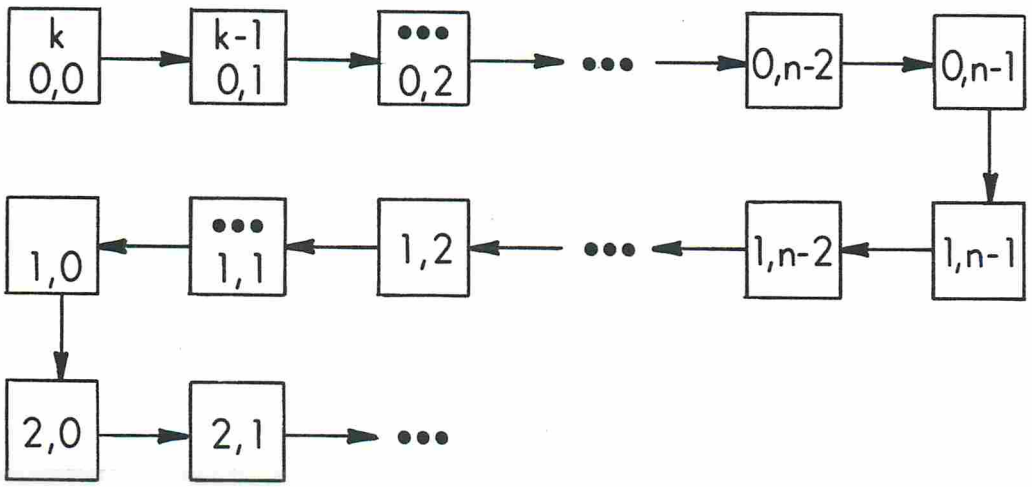
$$op(d) = \begin{cases} S & \text{if } d = N \\ W & \text{if } d = E \\ N & \text{if } d = S \\ E & \text{if } d = W \end{cases}; \qquad cw(d) = \begin{cases} E & \text{if } d = N \\ S & \text{if } d = E \\ W & \text{if } d = S \\ N & \text{if } d = S \end{cases}.$$

Similarly ccw(d) is defined.

Although packet transmission is used in structuring, continuous connections among neighbors are used in normal operation since the predecessor and successor cells are known [4].

Linear Array

To structure a linear array of size k on an $m \times n$ grid ($k \leqq m \cdot n$), the strategy depicted in Figure 3(a) is used. A processor receiving a structuring message will transmit either another structuring message or an acknowledgement. A positive acknowledgement (ACK) if the construction is complete and no more processors are needed. A negative acknowledgement (NACK) if it is impossible to complete the construction. The structuring process (Fig.3(b)) is initialized by transmitting a message M(LA,k,E) from the external monitor to processor (0,0) and its propagation time is of the order $\mathcal{O}(k)$. If an acknowledgement message M(ACK or NACK,LA,j) is received from the D_0-neighbor the message M(ACK or NACK,LA,j+1) is transmitted to the D_I-neighbor. When the negative acknowledgement reaches the origin cell, the maximum size of a linear array in the given grid is made known to the external monitor.



(a)

```
receive a message M(LA,ℓ,d) from δ-neighbor
                (ℓ = k,k-1,...,1; d ∈{E,W})
   CR := LA,ℓ
   if ℓ = 1 then transmit M(ACK,LA,1) to δ-neighbor
          else if d-terminal then begin
               if S-terminal M(NACK,LA,1) to δ-neighbor
               else transmit M(LA,ℓ-1,op(d)) to S-neighbor
                    endbegin
               else transmit M(LA,ℓ-1,d) to d-neighbor
                    endif
   endif
```

(b)

Fig. 3. Structuring algorithm for a linear array on a grid.

Square Array

A structuring algorithm for a square array of size $u \times v$ on an $m \times n$ grid ($u \leq m$; $v \leq n$) is shown in Figure 4. Each processor transmits the structuring message to both his South and East neighbors. The distributed algorithm is initialized by a message M(SA,u,v) transmitted from the external monitor to processor (0,0) and the propagation time of this algorithm is of the order $\mathcal{O}(u+v)$. Note that the statement "if CR = SA,k,ℓ then stop" prevents retransmission of the same structuring message which might otherwise occur since the processors are not necessarily synchronized and the message from the North-neighbor might be received before or after the message from the West-neighbor.

The idea of transmitting a message simultaneously in two directions may also be employed whenever a message is to be broadcast among all processors, regardless of the current configuration. Thus, a standard message M(BRCT, command or data) (where BRCT stands for broadcast and the command can be clear, test, etc.) can be used with a shorter propagation time of the order of $\mathcal{O}(m+n)$. A similar algorithm, with the same propagation time, may be devised to issue the initial cell IDs. These IDs are useful in some computational computations (e.g., [5]) as well as in the following structuring algorithm.

Loop

When embedding a loop (ring) in a rectangular grid we must first realize that only even length loops can be structured on such a grid as stated in the following lemma.

```
receive a message M(SA,k,ℓ) from δ-neighbor
   (k = u,u-1,...,1; ℓ = v,v-1,...,1)
if CR = SA,k,ℓ then stop else begin
   CR := SA,k,ℓ
if k ≧ 2 then begin
        if S-terminal then transmit M(NACK,SA,1,ℓ) to N-neighbor
                      else transmit M(SA,k-1,ℓ) to S-neighbor
             end
   endif
if ℓ ≧ then begin
      if E-terminal then transmit M(NACK,SA,k,1) to W-neighbor
                    else transmit M(SA,k,ℓ-1) to E-neighbor
             end
else if k = 1 then transmit M(ACK,SA,1,1) to N-neighbor and W-neighbor
endif
```

Fig. 4. A structuring algorithm for a square array.

Lemma: The cells in an $n \times n$ grid can be structured into a loop of length k ($k \leqq m \cdot n$) if and only if k is even.

Proof: Consider the grid as being a graph G with $m \cdot n$ vertices. A loop of length k corresponds to a k-vertex cycle in G. Hence, we have to prove that all cycles in G have an even number of vertices. Consider the horizontal and vertical cuts illustrated in Figure 5 and suppose we are given a cycle in G. A cut-set of the cycle is defined as the subset of edges along the cut which are part of the cycle, e.g., the cut-set corresponding to the vertical cut in Figure 5 contains two edges; the cut-set corresponding to the horizontal cut is empty.

Obviously, the number of edges in any cut-set (horizontal or vertical) must be even. Since each edge in the cycle belongs to exactly one cut-set the total number of edges in a cycle is the sum of even numbers and is therefore even. Consequently, the necessity of the condition k is even has been proved. The sufficiency is clear from the structuring algorithm in Figure 6(b). □

Corollary 1: An $m \times n$ grid has a Hamiltonian cycle if and only if either m or n is even.

Corollary 2: A loop on a grid consists of k cells if k is even and (k+1) cells if k is odd.

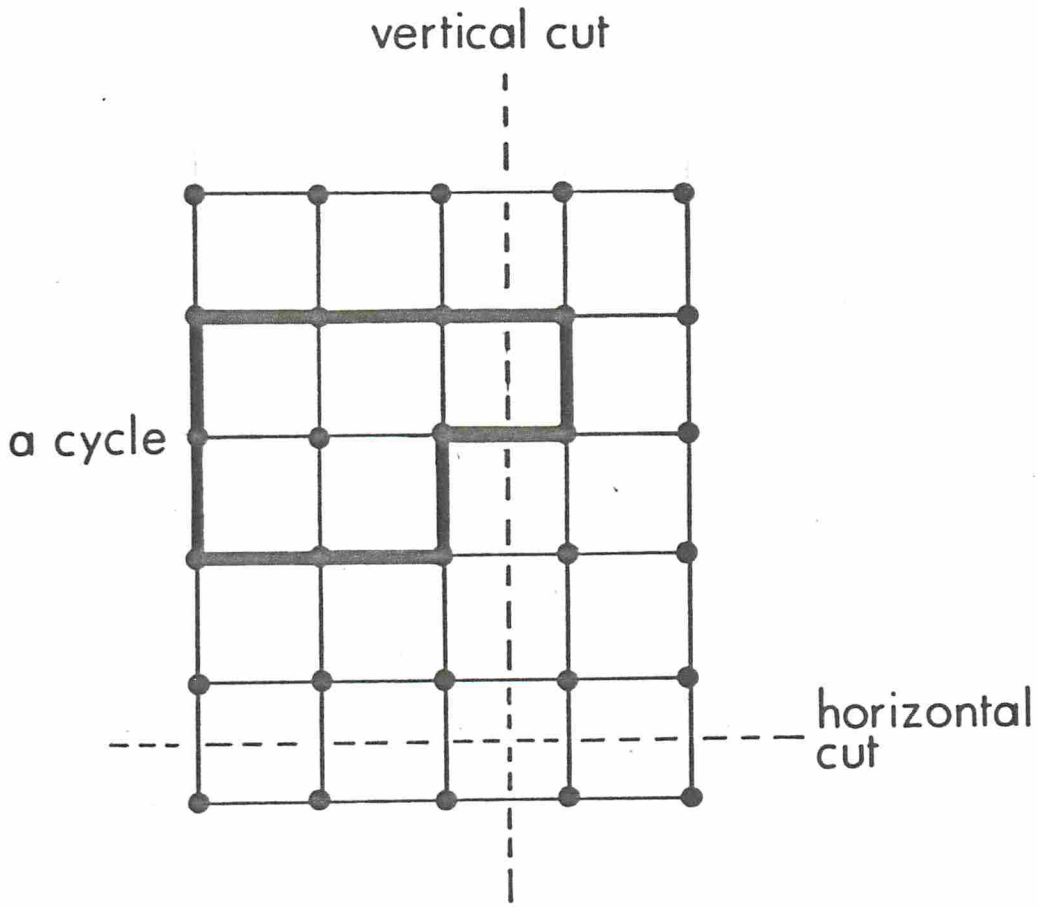


Fig. 5. Horizontal and vertical cuts in the graph G corresponding to a rectangular grid.

Various structuring algorithms for a loop on an $m \times n$ grid with even and odd values of m and n may be derived, for example, in Figure 6 we show an algorithm which is based upon the assumption that m is even. If m is odd this algorithm does not allow the use of the cells in the last row. A more general algorithm that allows the use of all cells for even m and odd m may be devised but is more complicated and hence, requires more chip area for every cell, thus reducing the number of cells in a given wafer area.

Binary Tree

Binary trees have been shown to be attractive interconnection schemes [2,6] for general-purpose multiprocessors [7] as well as for special-purpose applications such as data-base machines [8]. A VLSI implementation of a binary tree requires a mapping of the tree on a plane. Such a mapping can be done in several ways. The ways differ in the number of levels in the resulting binary tree, in the time needed for propagation through the tree, and in the complexity of the structuring algorithm. Two of them yield relatively simple algorithms. The one shown in Figure 7 has been used extensively (e.g., [2,6]) and a (centralized) placement algorithm for it has been described [6]. We call this one type 1 tree. The type 2 tree, illustrated in Figure 8, has not been studied before. The minimum size of the grid needed to place a k-level binary tree (with 2^k-1 PEs) has been calculated for both type 1 and type 2 schemes, with the following results:

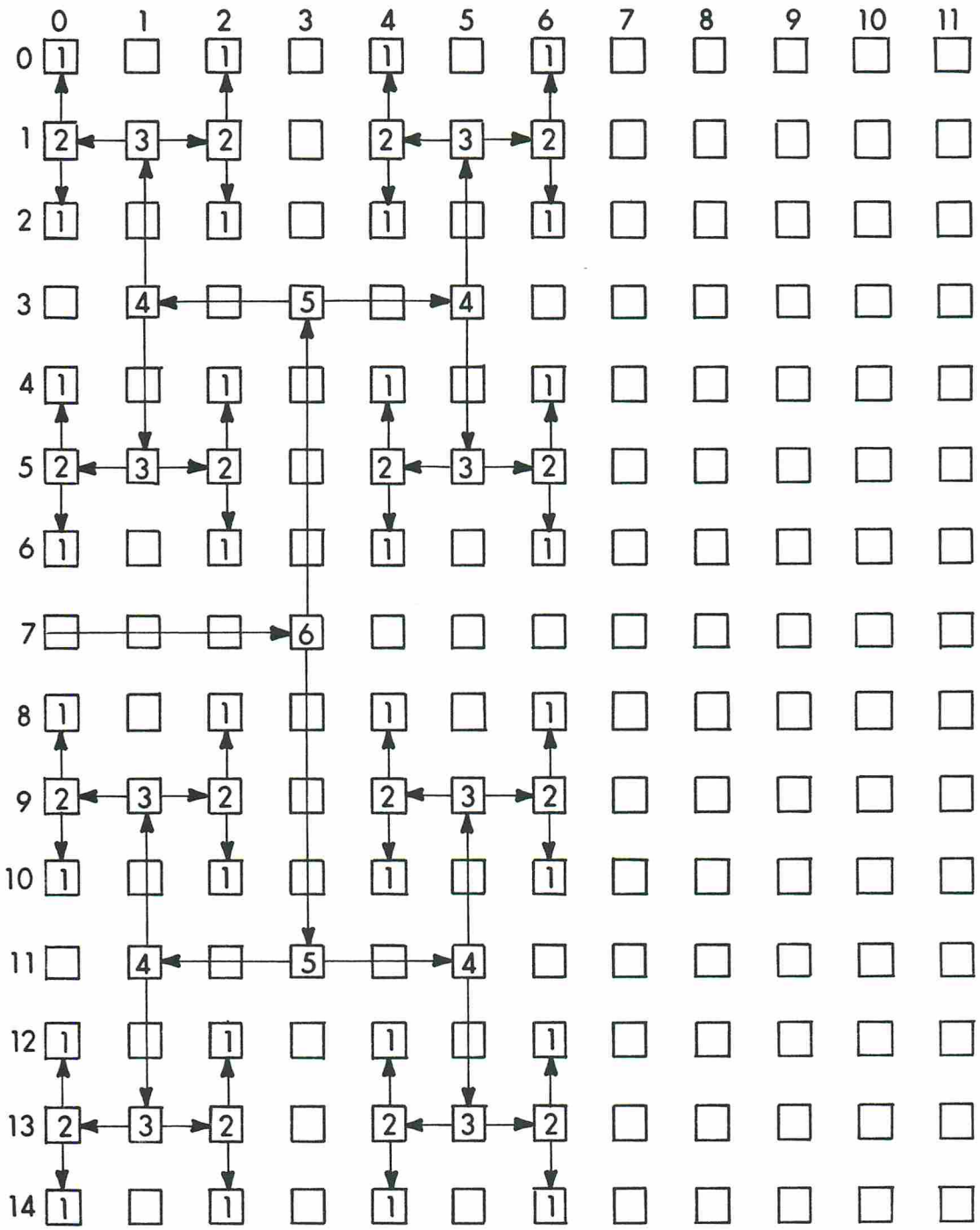


Figure 7. Type 1, six-level binary tree.

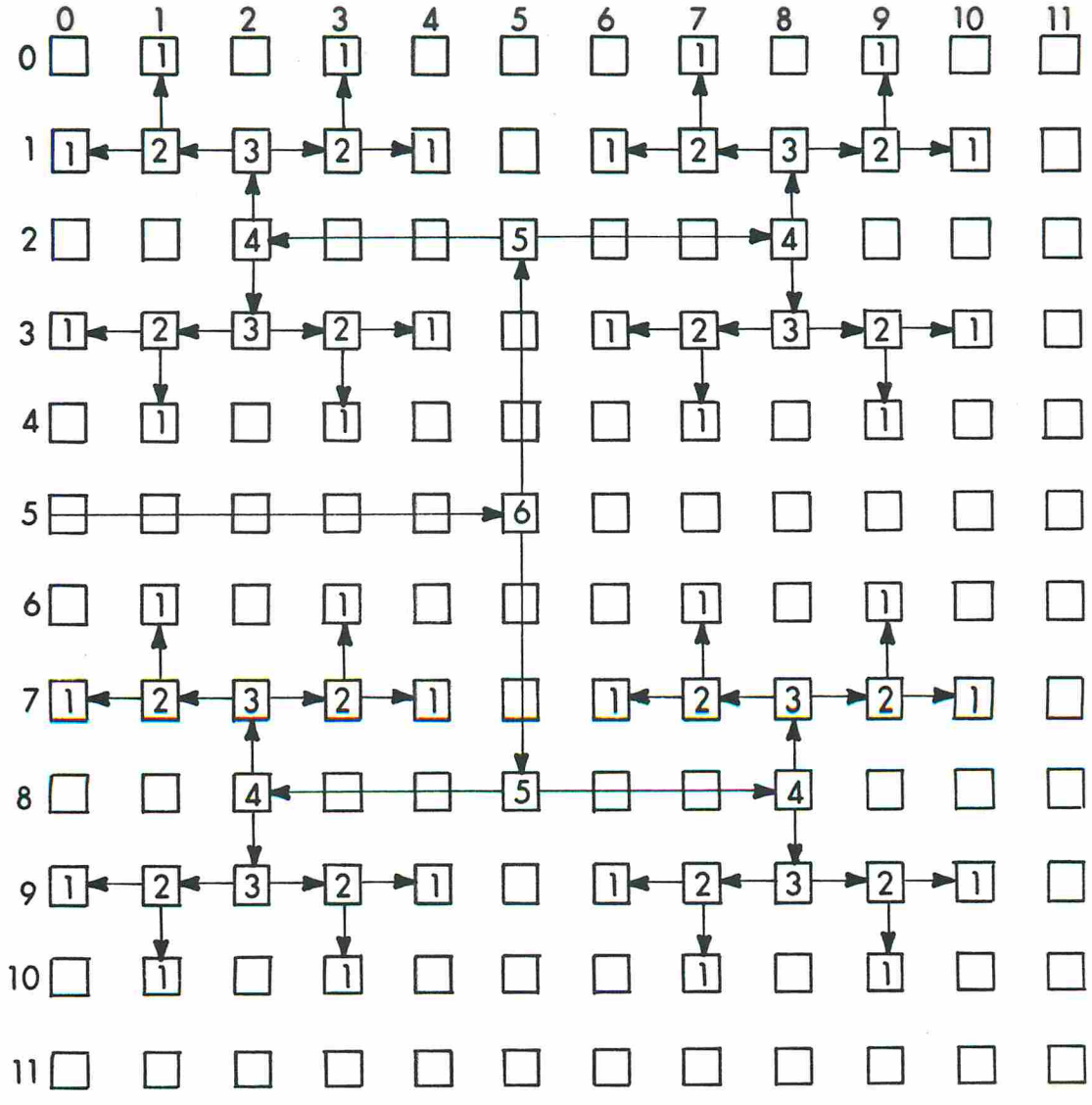


Figure 8. Type 2, six level binary tree.

$$\text{Size of grid for type 1 tree} = \begin{cases} \left(2^{\frac{k+1}{2}}-1\right) \times \left(2^{\frac{k+1}{2}}-1\right) & \text{if } k \text{ is odd} \\ \left(2^{\frac{k+2}{2}}-1\right) \times \left(2^{\frac{k}{2}}-1\right) & \text{if } k \text{ is even} \end{cases}$$

$$\text{Size of grid for type 2 tree} = \begin{cases} \left(3\cdot 2^{\frac{k-1}{2}}-1\right) \times \left(3\cdot 2^{\frac{k-3}{2}}-1\right) & \text{if } k \text{ is odd and } k \geq 3 \\ \left(3\cdot 2^{\frac{k-2}{2}}-1\right) \times \left(3\cdot 2^{\frac{k-2}{2}}-1\right) & \text{if } k \text{ is even} \end{cases}$$

In practice we are interested in the largest tree (maximum k) which can be placed on a given $m \times n$ grid. To simplify the expressions, we assume $m \geq n$, and for type 1 tree we obtain

$$\text{max. } k^{(1)} = \begin{cases} 2[\log_2(m+1)] - 1 & \text{if } [\log_2(m+1)] = [\log_2(n+1)] \\ 2[\log_2(n+1)] & \text{otherwise} \end{cases}$$

For type 2 tree we have

$$\text{max. } k^{(2)} = \begin{cases} 2\left[\log_2\left(\frac{m+1}{3}\right)\right] + 2 & \text{if } \left[\log_2\left(\frac{m+1}{3}\right)\right] = \left[\log_2\left(\frac{n+1}{3}\right)\right] \\ 2\left[\log_2\left(\frac{n+1}{3}\right)\right] + 3 & \text{otherwise} \end{cases}$$

These results are illustrated in Figure 9 for a square $m \times m$ grid and may be used to determine the best way to place a tree on a given grid.

Since cells serving as connecting elements (CEs) are needed in any binary tree placed on a grid, it is necessary to calculate and

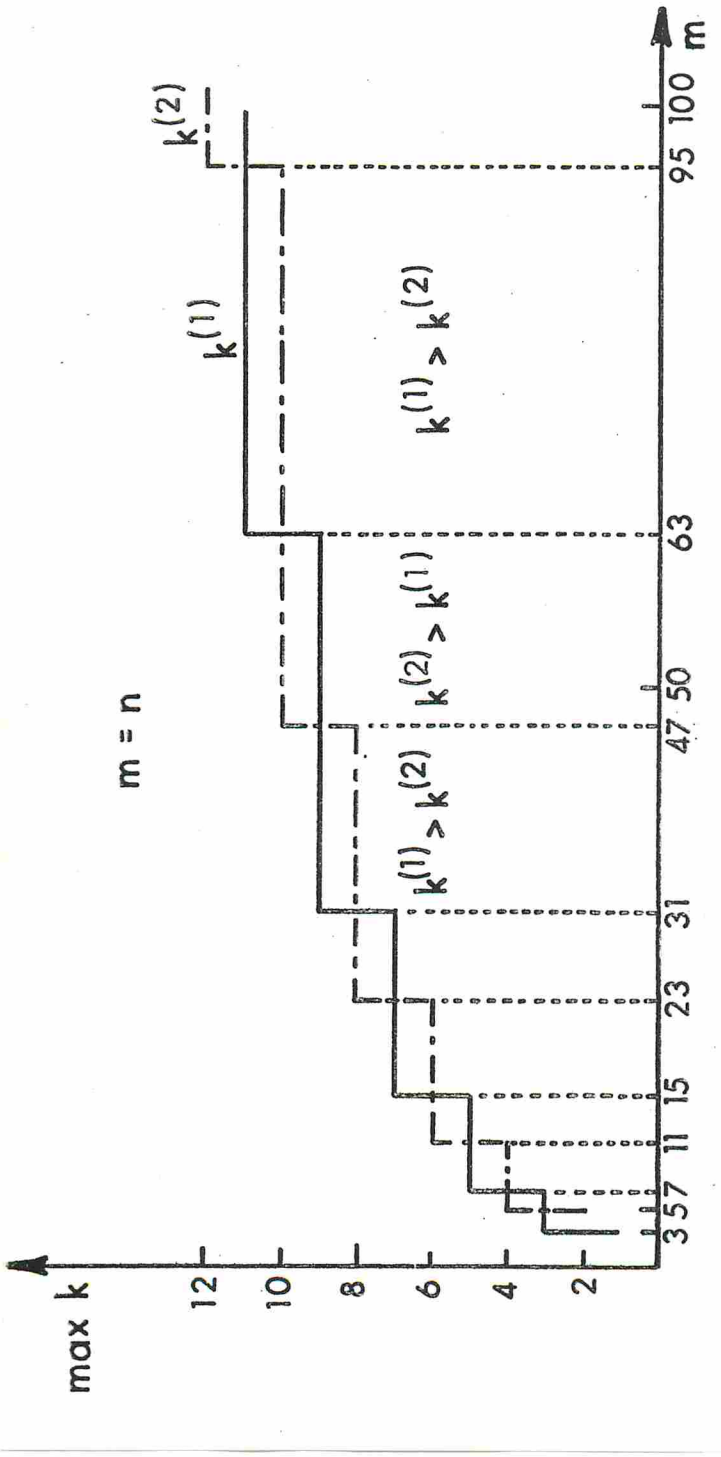


Fig. 9. Maximum number of levels in a binary tree that can be placed on an $m \times n$ grid.

compare the propagation time from root to leaves in the two schemes. For type 1 trees the total propagation time (i.e., number of cells traversed) denoted by $D_k^{(1)}$ is

$$D_k^{(1)} = \begin{cases} 2^{\frac{k+2}{2}} - 3 & \text{if } k \text{ is even} \\ 3 \cdot 2^{\frac{k-1}{2}} - 3 & \text{if } k \text{ is odd} \end{cases}$$

For type 2 trees we obtain

$$D_k^{(2)} = \begin{cases} 9 \cdot 2^{\frac{k-4}{2}} - 4 & \text{if } k \geq 4 \text{ is even} \\ 3 \cdot 2^{\frac{k-1}{2}} - 4 & \text{if } k \geq 3 \text{ is odd} \end{cases}$$

To compare these two, we form the difference yielding

$$D_k^{(1)} - D_k^{(2)} = \begin{cases} 1 - 2^{\frac{k-4}{2}} & \text{if } k \text{ is even} \\ 1 & \text{if } k \text{ is odd} \end{cases}$$

Thus, type 1 trees have a substantially lower propagation time if the number of levels is even, and a slightly higher propagation time if k is odd. Lastly, we present the structuring algorithms required for type 1 and type 2 trees. A structuring algorithm for a type 1 tree is depicted in Figure 10. Here, c is the number of CEs needed

```
receive a message M(BT,ℓ,c,d) from δ-neighbor
(ℓ = k,k-1,...,1; c = 2^[(k-1)/2] -1,...,1,0; d ∈ {N,E,S,W})
if c ≧ 1 then begin
   set CE
   if d-terminal then transmit M(NACK,BT,1) to δ-neighbor
                 else transmit M(BT,ℓ,c-1,d) to d-neighbor
              end
   else if ℓ = 1 then transmit M(ACK,BT,1) to δ-neighbor
              else begin
   if cw(d)-terminal or ccw(d)-terminal then transmit M(NACK,BT,1)
                                to δ-neighbor
       else begin
     CR := BT,ℓ
           [(ℓ-2)/2]
     c := 2          -1
     transmit M(BT,ℓ-1,c,cw(d)) to cw(d)-neighbor
     transmit M(BT,ℓ-1,c,ccw(d)) to ccw(d)-neighbor
       end
       end
```

Fig. 10. Structuring algorithm for Type 1 binary tree on a grid.

between a PE at level ℓ and a PE at level $(\ell-1)$. The PE at level k (i.e., the root) may be conveniently positioned in the middle row. Consequently, the algorithm is initialized by transmitting the message $M(BT,k,c=2^{[\frac{k-1}{2}]}-1,E)$ to cell $([\frac{m}{2}],0)$. Figure 11 shows a structuring algorithm for a type 2 tree; it is initialized by the message $M(BT,k,c=3\cdot 2^{[\frac{k-4}{2}]}-1,E)$ transmitted to cell $([\frac{m}{2}],0)$.

Reexamination of the distributed structuring algorithms in Figures 3, 4, 6, 10 and 11 reveals that whenever a structuring message is received, several conditions are checked, and as a result, an outgoing message and its destination are determined. A straightforward implementation of these algorithms in a PLA or ROM is therefore feasible and results in a simple and economical implementation of the communication processor in each cell. Moreover, these algorithms do not necessarily have to reside internally in each PE; instead they can be broadcast to all PEs when needed. THe broadcasting of a structuring algorithm is time-consuming, however, structuring is not expected to be needed frequently and as a result, the reduction in chip area makes this approach very attractive in many cases.

```
   receive a message M(BT,ℓ,c,d) from δ-neighbor
                                        [(k-4)/2]
   (ℓ = k,k-1,...,1; c = 3·2          -1,...,1,0; d ∈ {N,E,S,W})
if c ≧ 1 then begin
   set CE
  if d-terminal then transmit M(NACK,BT,1) to δ-neighbor
                else transmit M(BT,ℓ,c-1,d) to d-neighbor
                   end
else begin
     CR := BT,ℓ
if ℓ = 1 then transmit M(ACK,BT,1) to δ-neighbor
else begin
                                  [(ℓ-5)/2]
c := if ℓ ≧ 5 then (3·2          -1) else 0
α := if ℓ ≦ 3 then d else cw(d)
β := if ℓ = 2 then cw(d) else ccw(d)
γ := if ℓ = 2 then d else cw(d)
if β-terminal or γ-terminal then transmit M(NACK,BT,1) to δ-neighbor
else begin
   transmit M(BT,ℓ-1,c,α) to γ-neighbor
   transmit M(BT,ℓ-1,c,β) to β-neighbor
        end
        end
        end
```

Fig. 11. Structuring algorithm for Type 2 binary tree on a grid.

III. DISTRIBUTED TESTING OF THE ARRAY

A multiprocessor array can be tested either externally or internally. Due to the hardware complexity of the array and the pin limitation of a single VLSI chip, external testing is time-consuming and incomplete since no access to internal logic signals may be provided. Accurate identification of a single faulty processor within the array is often impossible. Consequently, internal testing in which each processor is tested by one or more of its neighbors is preferred [9]. If a faulty processor exists, all processors that are not faulty should be aware of the failure and refuse to interact with it. Since all interactions with the faulty processor must be through its immediate neighbors, it is sufficient that only these neighbors know the exact status of the faulty processor. This avoids excessive bookkeeping (each processor keeping track of all other processors' status) and complex status broadcasting algorithms, which must ensure that the vital status information is transmitted only through processors known to be functioning properly. Hence, we suggest fully distributed testing of processors, i.e., each processor is tested locally by one or more of its immediate neighbors, and the information about its status is kept locally in its immediate neighbors. Each one of the immediate neighbors of a given processor must know the status of this processor and should not rely on indirect information passed to him from other processors. Therefore, each processor must test, and be tested by, all its immediate neighbors.

Note that no voting mechanism to determine the status of a processor is required. Every immediate neighbor of a faulty cell will be aware of the failure and refuse to interact with it and the faulty cell will effectively become isolated from the rest of the system.

To achieve a high level of fault coverage when a complex processing element is tested by its immediate neighbors, a large number of test patterns should be applied. Furthermore, such a test may require access to points which are not accessible via the ordinary communication links. To avoid excessive use of chip area that is required to store the test patterns and resulting responses in each processor, and addition of extra communication links, we may incorporate other fault-tolerance techniques into the architecture of the processor. First, a processor can be made to tolerate certain faults [9]. Second, other faults that cannot be tolerated by the processor will at least be detected by self-checking techniques [10]. Thus, not only is the testing of one processor by its neighbor simplified but also immediate detection of certain faults is provided [11]. In summary, well known design techniques can be used at relatively low cost [10] to reduce the complexity of testing one PE by its neighbor to testing of the hardcore (the checking circuits) and the communication links.

While processor A is being tested, the testing of another processor, say B, can take place simultaneously; hence, the testing of the entire array can be organized so as to minimize the testing time. Let N denote the total number of test applications, i.e., one processor being tested by its neighbor. Let T denote the number of test-

ing periods to be minimized. For a rectangular $m \times n$ array N is given by

$$N = 4 \cdot 2 + [2(m+n)-8] \cdot 3 + [m \cdot m - 2(m+n)+4] \cdot 4$$

$$= 4 \cdot m \cdot n - 2(m+n).$$

If a processor tests only one neighbor at a time, there are at most $\frac{m \cdot n}{2}$ pairs, i.e., no more than $\frac{m \cdot n}{2}$ processors are testing their neighbors. Hence,

$$T \geq \frac{N}{\frac{m \cdot n}{2}} = 8 - \frac{4(m+n)}{m \cdot n} .$$

The number of testing periods can be reduced if each processor tests two of its neighbors simultaneously:

$$T \geq \frac{N}{\frac{2m \cdot n}{3}} = 6 - \frac{3(m+n)}{m \cdot n} .$$

If a processor tests all its neighbors (four at most) simultaneously, the number of testing periods is further reduced to five; in one step the PE tests all its neighbors while in each of the other four steps it is tested by one of its immediate neighbors separately. An algorithm (preferably, a five step one) is now needed to indicate when each processor must test all its neighbors. In each step of the algorithm, a message M(TS,c,testdata) is broadcast where TS stands for

test and $c = 0,1,2,3,4$ is the step number. Each processor must implement a function

$$f(c,i,j) = \begin{cases} 1 & \text{if cell } (i,j) \text{ is to test its neighbors in step } c \\ 0 & \text{otherwise} \end{cases}$$

Such an algorithm is illustrated in Figure 12, and the appropriate function is

$$f(c,i,j) = \begin{cases} 1 & \text{if } |i|_5 = |c+2j|_5; \quad c = 0,1,2,3,4 \\ 0 & \text{otherwise} \end{cases}$$

where $|i|_5$ is the residue of i modulo 5.

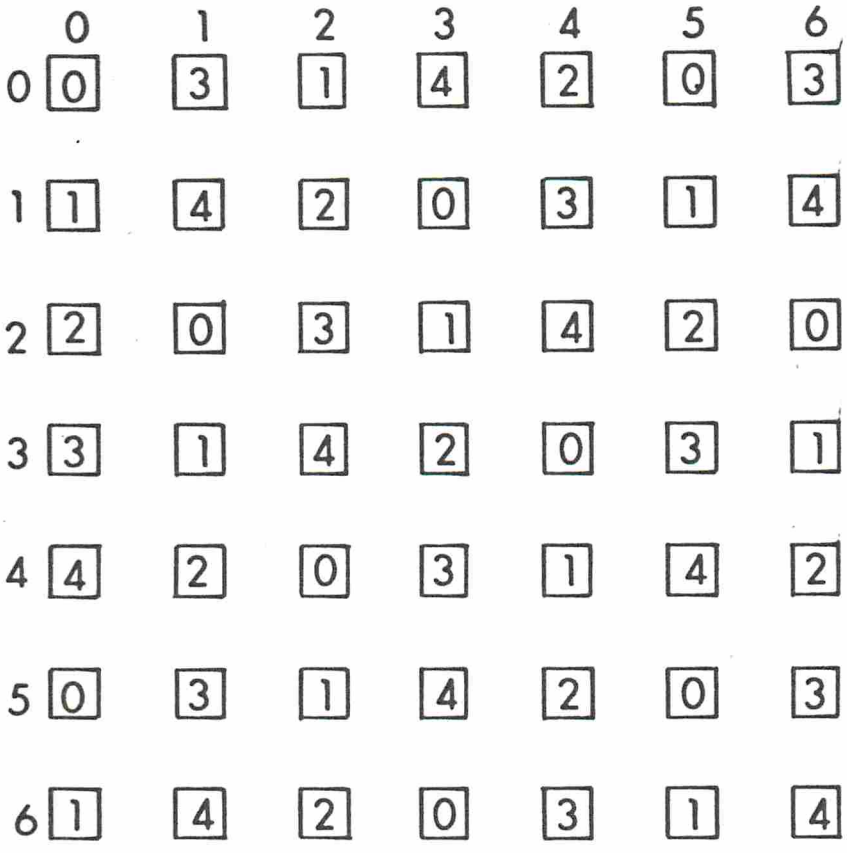


Fig. 12. Internal testing strategy.

IV. STRUCTURING IN THE PRESENCE OF FAULTY PROCESSORS

The structuring algorithms presented in Section III were developed under the assumption that there were no faulty processors. In the following we introduce the necessary modifications to handle faulty processors. A faulty processor may be known by its neighbors prior to the structuring or found to be faulty during the structuring process. Consequently, the following strategy is suggested for each processor during structuring.

(1) Receive incoming message.

(2) Determine outgoing message, its destination and the internal setting; transmit message.

(3) If there is no response, or a faulty one, update status and repeat (2).

If this strategy is adopted, we only have to incorporate into the structuring algorithms all the possibilities of faulty neighbors. When this modification is introduced in an algorithm, there are two opposing objectives. The first one is to maximize the number of processors still available (which is at most mn-1); the second one is to minimize the additional complexity introduced into the structuring algorithm (and hence, additional area occupied by the communication processor in each cell). A complex algorithm that may maximize the number of processors still usable after a fault occurrence may be

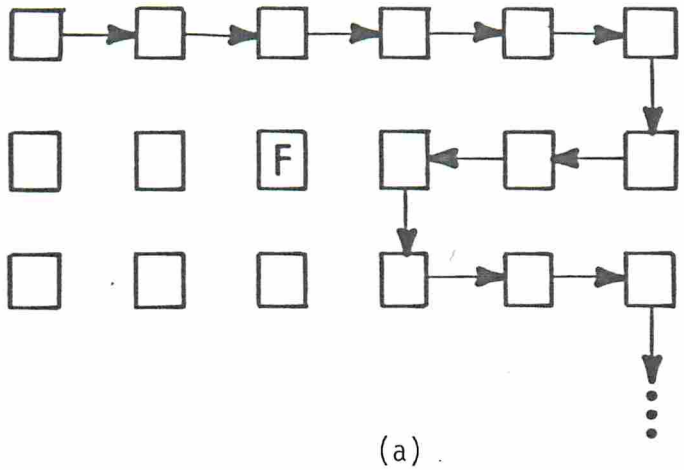
wasteful when the array is fault-free, because a smaller number of processors would initially fit into the same chip area. In the following we adopt the viewpoint that it is beneficial to have a relatively simple algorithm mainly because of the low failure rate projected for a single processor in a VLSI chip.

Linear Array

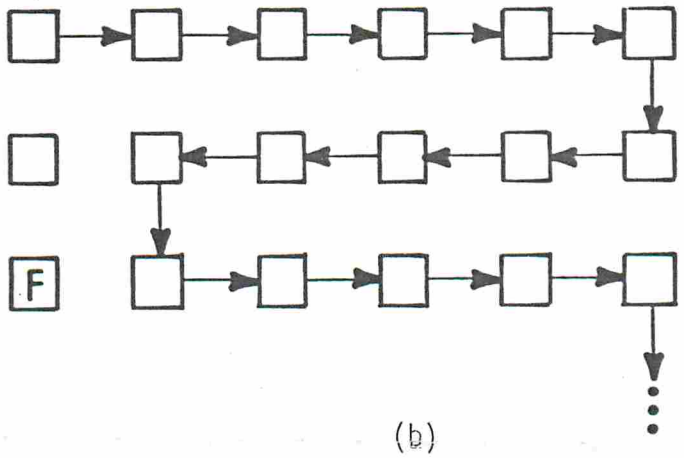
In order to obtain a relatively simple algorithm for structuring a linear array on a grid with faulty processors we use the strategy depicted in Figure 13(a) and 13(b). This strategy does not require any previous knowledge about the position of the faulty processor and leaves the last available processor in the array unchanged, thus simplifying expansions of the linear array to a second IC chip. The number of unused processors varies with the position of the faulty processor, and its expected value approaches n-1 for $n,m \gg 2$. For a 1000-processor array, this means that a tolerable percentage of 3.1% of the processors is not utilized. In the special case, illustrated in Figure 13(b) (or whenever a processor is terminal in both d and S directions), backtracking is necessary. The algorithm shown in Figure 13(c) provides a one-step backtracking; if unsuccessful (as a result of an additional faulty processor), a negative acknowledgement is transmitted. A slightly more complex algorithm may provide a two-step (or more) backtracking.

Other Structures

The situation is more complicated for the other three structures, especially binary trees. Since structuring messages are transmitted



(a)



(b)

```
receive a message M(LA,ℓ,d) from δ-neighbor
(ℓ = k,k-1,...,1; d ∈ {E,W})
If CR = LA,ℓ then begin
   if δ = S or S-terminal then transmit M(NACK,LA,2) to D_I-neighbor
                          else transmit M(LA,ℓ-1,op(d)) to S-neighbor
                          end

   else begin
  CR := LA,ℓ
  D_I := δ
if ℓ = 1 then transmit M(ACK,LA,1) to δ-neighbor
        else begin
     if d-terminal then
          if S-terminal then transmit M(LA,ℓ+1,d) to δ-neighbor
                        else transmit M(LA,ℓ-1,op(d)) to S-neighbor
                        end if
            else transmit M(LA,ℓ-1,d) to d-neighbor
          end
          end
```

(c)

Fig. 13. Structuring algorithm for linear array (faulty processors are present).

simultaneously by all processors at the same level, local modifications in strategy by one of the processors are not allowed. Consequently, whenever the pre-assigned processor does not respond or has a faulty response, a negative acknowledgement is transmitted back. This, however, does not imply that the structuring of the binary tree is impossible, since there are always processors in the grid which are not used in a binary tree structure.

To allow structuring in the presence of faulty processors, we suggest the following strategy. If a negative acknowledgement is received (although the desired structure should fit into the grid) a testing period is first initiated to ensure proper identification of the faulty processor. Next, each processor whose d-neighbor is faulty will declare itself a connecting element (CE) and transmit a message to its op(d)-neighbor causing it to declare itself a CE also, and so on until all processors in the row and column of the faulty processor are declared CEs, as shown in Figure 14. For a 1000-processor grid, 6.1% of the processors will become connecting elements. A much smaller percentage of PEs will become CEs if only the communication link between two adjacent PEs is faulty. In this case, each PE will claim that the other one is faulty, will thus declare itself a CE and finally only the row (or column) containing these two will turn out to be a row (column) of CEs.

Once the faulty processor has been taken care of, renumbering of the remaining PEs can be initiated, if necessary, by broadcasting

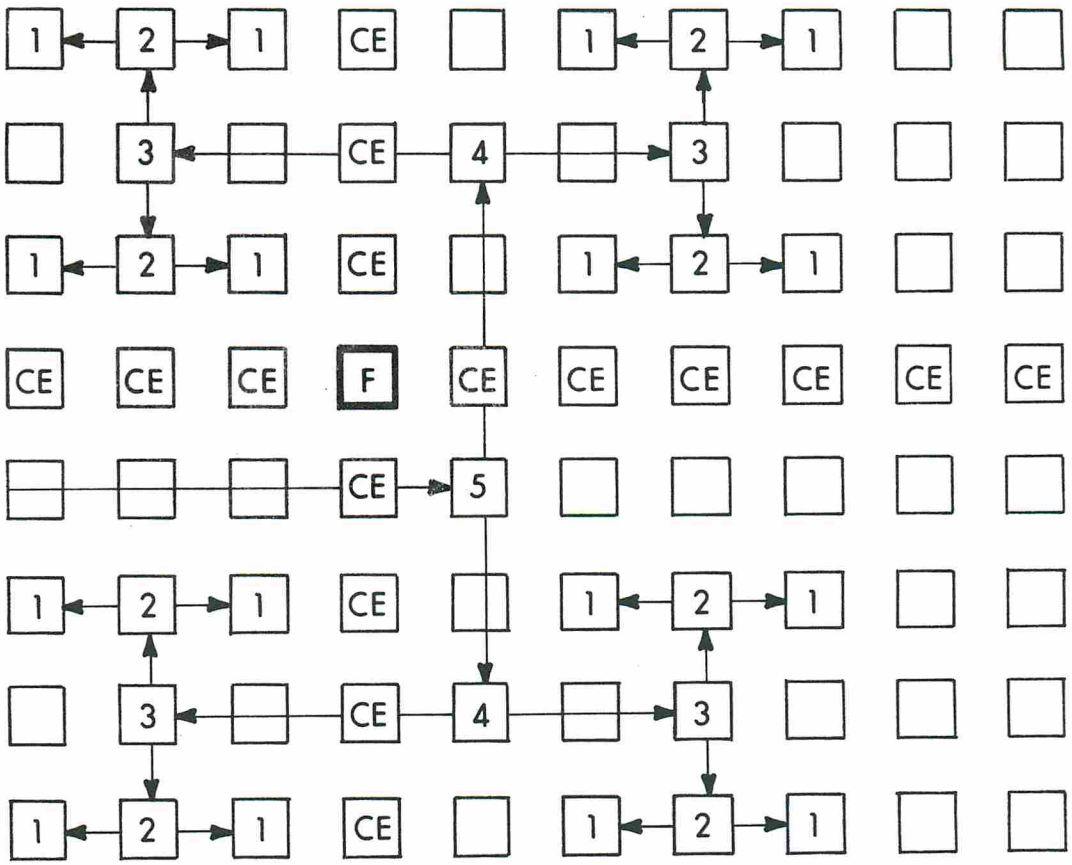


Figure 14. Five-level binary tree on a grid (a faulty processor is present).

an appropriate message, and the ordinary structuring algorithms may then be used. Figure 14 shows the resulting binary tree (note that when a CE receives a message, it is transmitted unchanged to the opposite direction). In most cases, the maximum size of the binary tree that can be placed on the grid will not change as is evident from Figure 9.

V. CONCLUSIONS

An example of a reconfigurable and fault-tolerant VLSI multiprocessor array has been presented. It demonstrates the feasibility of an array with these desirable properties, but it is neither a unique array with such features nor an optimal one in any sense. Further research steps are needed to consider other reconfigurable and fault-tolerant systems (e.g., one which is based on the hexagonal array [2]) and to devise ways to compare them with regard to possible structures and the corresponding classes of computational algorithms, complexity of structuring algorithms, fault-tolerance capacity, and other reliability measures.

REFERENCES

1. M.J. Foster & H.T. Kung, "The design of special-purpose VLSI chips," *Computer 13*, 26-40, January 1980.

2. C.A. Mead & L.A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980, Sec. 8.3.

3. D.P. Siewiorek, D.E. Thomas & D.L. Scharfetter, "The use of LSI modules in computer structures: trends and limitations," *Computer 11:7*, 16-25, July 1978.

4. H. Sullivan & T.R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine, I," *Proc. 4th Symposium on Computer Architecture*, 105-117, March 1977.

5. B. Ackland, N. Weste & D.J. Burr, "An integrated multiprocessing array for time warp pattern matching," *Proc. 8th Annual Symposium on Computer Architecture*, 197-215, May 1981.

6. E. Horowitz & A. Zorat, "The binary tree as an interconnection network: applications to multiprocessor systems and VLSI," *IEEE Trans. on Computers C-30*, 247-253, April 1981.

7. C.H. Séquin, "Single-chip computers, the new VLSI building blocks," *Proc. Caltech Conference on VLSI*, 435-445, January 1979.

8. S.W. Song, "A highly concurrent tree machine for database applications," *Proc. 1980 Int'l. Conference on Parallel Processing*, 259-268, August 1980.

9. J.G. Kuhl & S.M. Reddy, "Distributed fault-tolerance for large multiprocessor systems," *Proc. 7th Symposium on Computer Architecture*, 23-30, May 1980.

10. W.C. Carter et al., "Cost effectiveness of self-checking computer design," *Proc. 7th Symposium on Fault-Tolerant Computing*, 117-123, June 1977.

11. R.M. Sedmak & H.L. Liebergot, "Fault tolerance of a general purpose computer implemented by very large scale integration," *IEEE Trans. on Computers C-29*, 492-500, June 1980.

REFERENCES

1. M.J. Foster & H.T. Kung, "The design of special-purpose VLSI chips," *Computer 13*, 26-40, January 1980.

2. C.A. Mead & L.A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980, Sec. 8.3.

3. D.P. Siewiorek, D.E. Thomas & D.L. Scharfetter, "The use of LSI modules in computer structures: trends and limitations," *Computer 11:7*, 16-25, July 1978.

4. H. Sullivan & T.R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine, I," *Proc. 4th Symposium on Computer Architecture*, 105-117, March 1977.

5. B. Ackland, N. Weste & D.J. Burr, "An integrated multiprocessing array for time warp pattern matching," *Proc. 8th Annual Symposium on Computer Architecture*, 197-215, May 1981.

6. E. Horowitz & A. Zorat, "The binary tree as an interconnection network: applications to multiprocessor systems and VLSI," *IEEE Trans. on Computers C-30*, 247-253, April 1981.

7. C.H. Séquin, "Single-chip computers, the new VLSI building blocks," *Proc. Caltech Conference on VLSI*, 435-445, January 1979.

8. S.W. Song, "A highly concurrent tree machine for database applications," *Proc. 1980 Int'l. Conference on Parallel Processing*, 259-268, August 1980.

9. J.G. Kuhl & S.M. Reddy, "Distributed fault-tolerance for large multiprocessor systems," *Proc. 7th Symposium on Computer Architecture*, 23-30, May 1980.

10. W.C. Carter et al., "Cost effectiveness of self-checking computer design," *Proc. 7th Symposium on Fault-Tolerant Computing*, 117-123, June 1977.

11. R.M. Sedmak & H.L. Liebergot, "Fault tolerance of a general purpose computer implemented by very large scale integration," *IEEE Trans. on Computers C-29*, 492-500, June 1980.