A RECONFIGURABLE AND FAULT-TOLERANT VLSI MULTIPROCESSOR ARRAY

ISRAEL KOREN

Department of Electrical Engineering
Technion–Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

Multiprocessor arrays have the property of regularity, enabling a low-cost VLSI implementation. However, multiprocessor systems with a fixed structure tend to be error prone and restricted to 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 can be adapted to many applications and may restructure itself upon failure of a processor, to avoid using 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. A novel strategy for internal testing and for identification of faulty processors is developed, and the structuring algorithms are modified to accommodate faulty processors.

INTRODUCTION

The current trends in the integrated circuit technology will 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 5 to 10 years, will enable packaging an increasing number of processors into a single VLSI chip. Clearly, such packaging is preferred over using several LSI chips when power consumption, speed, and reliability are considered.

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 regular multiprocessor arrays. Regular structures take considerably less time to design and to manufacture.[1-3]

Yet, even regular multiprocessor arrays may still be unattractive to the semiconductor industry because of their tendency to have restricted uses. A VLSI multiprocessor array with a flexible structure is desirable because it can be configured in several ways, which increases the number of possible applications. Clearly, some processors will not be used in several applications, but this should not be considered a serious drawback; 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 the 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 that that of present LSI chips because of the complexity of the VLSI chips. Hence, the multiprocessor's proper operation must be verified frequently. Unfortunately, this verification cannot be achieved by external testing of the VLSI chip because of the inability to access internal points; internal testing (processors testing one another) for error detection is therefore 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. When a fault occurs 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, such as 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 impurities that exist in large wafer areas.

0149-7111/81/0000/0425$00.75 © 1981 IEEE

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 a rectangular m x n grid (Figure 1). This grid may be structured as a linear array, a square array, or a binary tree. Not all processors will necessarily be used 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, and the projected complexity of VLSI chips is 1,000K and over. Consequently, a grid consisting of 1,000 or more processors will be feasible. In Figure 1, there are mn cells in the grid; each cell is connected to its immediate neighbors (N-neighbor, E-neighbor, S-neighbor, and W-neighbor). The basic cell consists of an application processor and a communication processor (Figure 2). The application processor may be a microprogrammable processor that can be programmed by the customer and tailored to his or her special needs. This paper focuses in what follows on the communication processor, which is responsible for the structuring of the multiprocessor array.

Four registers are defined within the communication processor:

ID Register: Contains the row and column indices.

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 and E to W (and vice versa) without performing any kind of processing.

Status Register: Stores the status of the immediate neighbors. If the d-neighbor (the immediate neighbor in the d-direction in which d is an element of {N,E,S,W}) either is faulty or does not exist, the cell is d-terminal. In this case, no attempt to communicate with the d-neighbor will be made.

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 sucessor (D_O) cells.

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. Obviously, simplicity of the structuring algorithms is desirable 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, 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 is an element of {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, the functions opposite (op), clockwise (cw), and counterclockwise (ccw) are defined as:

1. op(d) equals S if d equals N; W if d equals E; N if d equals S; E if d equals W

2. cw(d) equals E if d equals N; S if d equals E; W if d equals S; N if d equals W

3. ccw(d) equals W if d equals N; N if d equals E; E if d equals S; S if d equals W

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

Linear Array

To structure a linear array of size k on an m x n grid ($k \leq mn$) the strategy depicted in Figure 3(a) is used. A processor receiving a structuring message will transmit

either another structuring message or an acknowledgment: a positive acknowledgment (ACK) if the construction is complete and no more processors are needed, or a negative acknowledgment (NACK) if it is impossible to complete the construction. The structuring process is initialized by transmitting a message

M(LA, k, E)

to processor (0, 0), and its propagation time is of the order $O(k)$. If an acknowledgment 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_1-neighbor. When the negative acknowledgment reaches the origin cell, the maximum size of a linear array in the given grid is known.

Square Array

A structuring algorithm for a square array of size u x v on an m x n grid ($u \leq m$; $v \leq n$) is shown in Figure 4. Each processor transmits the structuring message to both its S(outh) and E(ast) neighbors. The distributed algorithm is initialized by a message

M(SA, u, v)

transmitted to processor (0, 0), and the propagation time of the algorithm is of the order $O(u + v)$. Note that the statement

if CR = SA, k, l then stop

prevents retransmission of the same structuring message, which may otherwise occur because the processors are not necessarily synchronized and the message from the N(orth)-neighbor may be received before or after the message from the W(est)-neighbor.

The idea of transmitting a message simultaneously in two directions may also be used whenever a message is to be broadcast among all processors, regardless of the current configuration. A standard message

M(BRCT, command or data)

(BRCT stands for broadcast; the command can be clear, test, etc.) can be used with a shorter propagation time of the order $O(m + n)$.

Binary Tree

A binary tree can be placed on a given m x n grid 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 binary tree shown in Figure 5, a type 1 tree, has been used extensively,[2,5] and a (centralized) placement algorithm for it has been described.[5] The type 2 tree, illustrated in Figure 6, has not been studied before.

Minimum Grid Size

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:

1. Size of grid for type 1 tree equals
   (a) $(2^{(k+1)/2}-1) \times (2^{(k+1)/2} - 1)$ if k is odd;
   (b) $(2^{(k+2)/2}-1) \times (2^{k/2} - 1)$ if k is even.

2. Size of grid for type 2 tree equals
   (a) $(3 \times 2^{(k-1)/2}-1) \times (3 \times 2^{(k-3)/2} - 1)$ if k is odd and greater than or equal to 3;
   (b) $(3 \times 2^{(k-2)/2}-1) \times (3 \times 2^{(k-2)/2} - 1)$ if k is even.

Largest Tree for a Given Grid

In practice we are interested in the largest tree (maximum k) that can be placed on a given m x n *grid.* To simplify the expressions, we assume $m \geq n$:

1. For type 1 tree, maximum $k^{(1)}$ equals
   (a) $2[\log_2(m + 1)] - 1$ if $[\log_2(m + 1)] = [\log_2(n + 1)]$;
   (b) $2[\log_2(n + 1)]$ otherwise.

2. For type 2 tree, maximum $k^{(2)}$ equals

(a) $2[\log_2(\frac{m+1}{3})] + 2$ if

$$[\log_2(\frac{m+1}{3})] = [\log_2(\frac{n+1}{3})$$

(b) $2[\log_2(\frac{n+1}{3})] + 3$ otherwise.

These results are illustrated in Figure 7 for a square m x m grid and may be used to determine the best way to place a tree on a given grid.

Propagation Time

Since cells serving as connecting elements (CEs) are needed in any binary tree placed on a grid, it is necessary to calculate and compare the propagation time (number of cells traversed) from root to leaves in the two schemes:

1. Total propagation time for type 1 trees, $D_k^{(1)}$, equals
   (a) $2^{(k+2)/2} - 3$ if k is even;
   (b) $3 \times 2^{(k-1)/2} - 3$ if k is odd.

2. Total propagation time for type 2 trees, $D_k^{(2)}$, equals
   (a) $9 \times 2^{(k-4)/2} - 4$ if k is even and greater than or equal to 4;
   (b) $3 \times 2^{(k-1)/2} - 4$ if k is odd and greater than or equal to 3.

3. To compare the two, we form the difference, $D_k^{(1)} - D_k^{(2)}$, which equals
   (a) $1 - 2^{(k-4)/2}$ if k is even;
   (b) 1 if k is odd.

Thus, type 1 trees have a substantially lower propagation time if the number of levels (k) is even, and a slightly higher propagation time if k is odd.

Structuring Algorithms

Figure 8 shows a structuring algorithm for a type 1 tree. Here, c is the number of CEs needed between PE at level l and PE at level $(l - 1)$. The processing element at level k (the root) may be conveniently positioned in the middle row. The algorithm is initialized by transmitting the message

M(BT, k, $c = 2^{[(k-1)/2]} - 1$, E)

to cell ($[m/2] - 1$, 0).

Figure 9 shows a structuring algorithm for a type 2 tree; it is initialized by the message

M(BT, k, $c = 3 \times 2^{[(k-4)/2]} - 1$, E)

transmitted to cell ($[m/2] - 1$, 0).

If the distributed structuring algorithms in Figures 3, 4, 8, and 9 are reexamined, it is seen 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 PLA or ROM is therefore feasible and results in a simple and economical implementation of the communication processor in each cell.

DISTRIBUTED TESTING OF THE ARRAY

A multiprocessor array can be tested either externally or internally. As a result of the hardware complexity of the array and the pin limitation of a single VLSI chip, external testing is time-consuming and incomplete, because 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.[6] If a faulty processor exists, all processors that are not faulty should be aware of the failure and refuse to interact with it. Because 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 the status of all other processors) and complex status broadcasting algorithms, which must ensure that the vital status information is transmitted only through processors known to be functioning properly.

We suggest fully distributed testing of processors: 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 it from other processors. Therefore, each processor must test, and be tested by, all its immediate neighbors. 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 be 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 that cannot be accessed through the ordinary communications 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, other fault-tolerance techniques can be added to the architecture of the processor. First, a processor can be made to tolerate certain faults.[6] Second, other faults that cannot be tolerated by the processor will at least be detected by self-checking techniques.[7] Not only is the testing of one processor by its neighbor simplified but also immediate detection of certain faults is provided.[8] In summary, well-known design techniques can be used at relatively low cost[7] to reduce the complexity of testing one PE by its neighbor to testing of the hardcore (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 to minimize the testing time. If N equals the total number of test applications (one processor being tested by its neighbor), and if T equals the number of testing periods to be minimized, then for a rectangular m x n array:

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

$$= 4mn - 2(m+n)$$

If a processor tests only one neighbor at a time, there are no more than $(mn/2)$ pairs; that is, no more than $(mn/2)$ processors are testing their neighbors. Hence,

$$T \geq N/(mn/2) = 8 - (4(m+n)/mn)$$

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

$$T \geq N/(2mn/3) = 6 - (3(m+n)/mn)$$

If a processor tests all its neighbors (four at most) simultaneously, the number of testing periods is further reduced:

$$T \geq N/(4mn/5) = 5 - (5(m+n)/2mn)$$

For $m, n \geq 6$, we obtain the lower bound $T \geq 5$. An algorithm (preferrably, 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 (TS stands for test; c equals 0, 1, 2, 3, or 4 and is the step number). Each processor must implement a function for which

1. $f(c, i, j)$ equals
   (a) 1 if cell (i, j) is to test its neighbors in step c;
   (b) 0 otherwise.

Such an algorithm is illustrated in Figure 10, and the appropriate function is

1. $f(c, i, j)$ equals
   (a) 1 if $|i|_5 = |c + 2j|_5$, (c = 0, 1, 2, 3, or 4; $|i|_5$ is the residue of i modulo 5);
   (b) 0 otherwise.

STRUCTURING IN THE PRESENCE OF FAULTY PROCESSORS

The structuring algorithms presented in the previous section were developed under the assumption that there were no faulty processors; hence, certain modifications must be made 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. 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 (step) 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:

1. To maximize the number of processors still available (which is at most mn - 1)

2. To minimize the additional complexity introduced into the structuring algorithm (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. We believe that it is beneficial to have a relatively simple algorithm because of the low failure rate projected for a single processor in a VLSI chip.

Linear Array

The strategy shown in Figure 11(a) and (b) can be used to obtain a relatively simple algorithm for structuring a linear array on a grid with faulty processors. The strategy does not require any previous knowledge about the position of the faulty processor and leaves the last available processor in the array unchanged, 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, $m >> 2$. For a 1,000 processor array, this means that a tolerable percentage of 3.1% of the processors is not used. In the special case in Figure 11(b) (or whenever a processor is terminal in both d and S directions), backtracking is necessary. The algorithm in Figure 11(c) provides one-step backtracking; if it is unsuccessful (as a result of an additional faulty processor), a negative acknowledgment is transmitted. A slightly more complex algorithm may provide two-step (or more) backtracking.

The situation is more complicated for the other two structures, especially binary trees. Because structuring messages are transmitted simultaneously by all processors at the same level, local modifications in strategy by one of the processors are not allowed. Consequently, whenever the preassigned processor does not respond or has a faulty response, a negative acknowledgment is transmitted. This, however, does not imply that the structuring of a binary tree is impossible, because there are always processors in the grid that are not used in a binary tree structure.

A possible way to avoid using a faulty processor is shown in Figure 12. A message is broadcast (preceded if necessary by a testing period to ensure proper identification of faulty processors), which will cause each processor whose d-neighbor is faulty to declare itself a connecting element (CE) and to transmit a message to its op(d)-neighbor, until all processors in the row and the column of the faulty processor are declared CEs. The ordinary structuring algorithm may then be used (note that when a CE receives a message, it is transmitted unchanged to the opposite direction) to get the binary tree shown in Figure 12. For a 1,000-processor grid, 6.1% of the processors will not be used. In most cases, the maximum size of the binary tree that can be placed on the grid will not change (Figure 7).

CONCLUSIONS

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

ACKNOWLEDGMENT

This work was supported by the Department of the Navy, Navy Electronic Systems Command, under contract N00039-80-C-0641, and was done while visiting the Department of Electrical Engineering–Systems at the University of Southern California (Los Angeles).

REFERENCES

[1]M. J. Foster and H. T. Kung. The design of special-purpose VLSI chips, *Computer*, 13 (Jan. 1980), 26-40.

[2]C. A. Mead and L. A. Conway. *Introduction to VLSI Systems* (Reading, Mass: Addison-Wesley, 1980), sec. 8.3.

[3]D. P. Siewiorek, D. E. Thomas, and D. L. Scharfetter. The use of LSI modules in computer structures: Trends and limitations, *Computer*, 11 (July 1978), 16-25.

[4]H. Sullivan and T. R. Bashkow. A large scale, homogeneous, fully distributed parallel machine, I, *Proceedings of the 4th Symposium on Computer Architecture* (March 1977), 105-117.

[5]E. Horowitz and A. Zorat. The binary tree as an interconnection network. *Proceedings of the 1980 Conference on Networks* (April 1980).

[6]J. G. Kuhl and S. M. Reddy. Distributed fault-tolerance for large multiprocessor systems, *Proceedings of the 7th Symposium on Computer Architecture* (May 1980), 23-30.

[7]W. C. Carter et al. Cost effectiveness of self-checking computer design, *Proceedings of the 7th Symposium on Fault-Tolerant Computing* (June 1977), 117-123.

[8]R. M. Sedmak and H. L. Liebergot. Fault tolerance of a general purpose computer implemented by very large scale integration, *IEEE Transactions on Computers*, 29 (June 1980), 492-500.

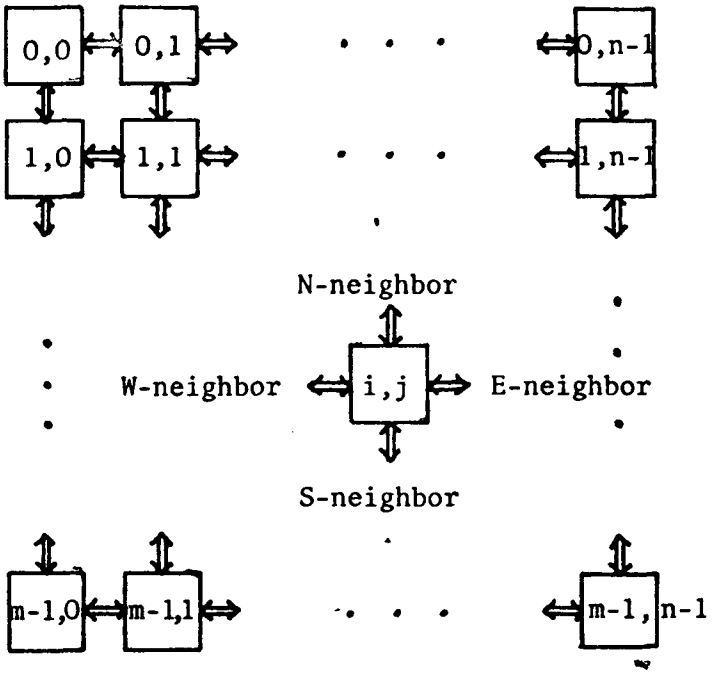


Figure 1. Rectangular *m* x *n* Grid

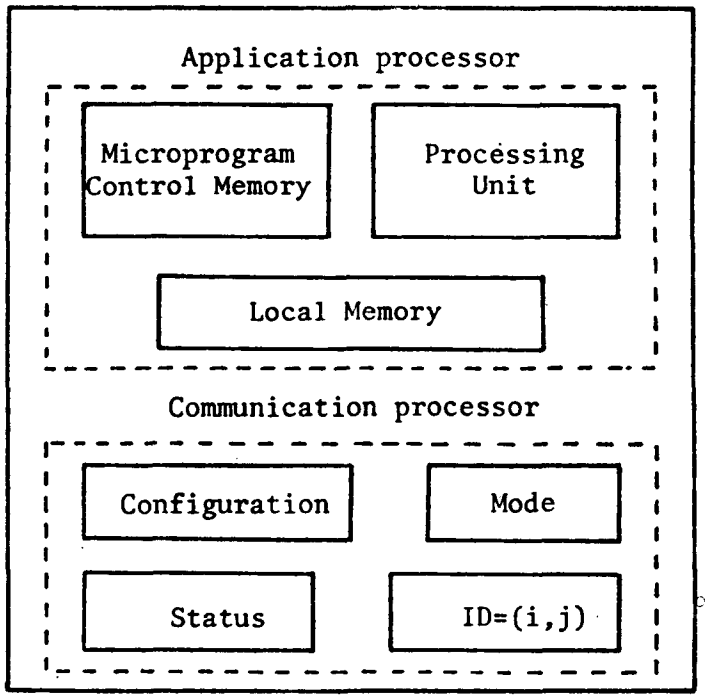
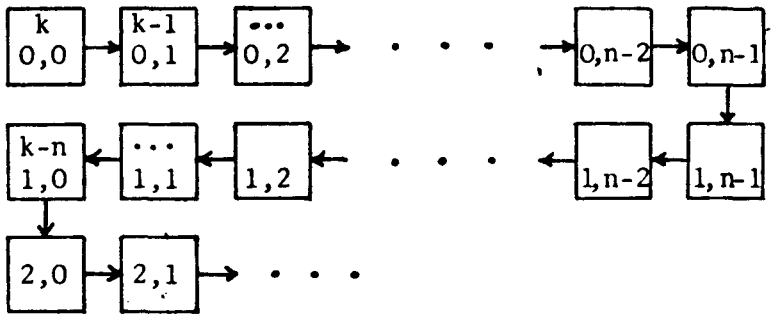


Figure 2. Basic Processing Element



(a)

```
receive a message  M(LA,ℓ,d)  from δ-neighbor
          (ℓ=k,k-1,...,1 ; d ε {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 δ-neighbor
                     endbegin
                else transmit M(LA,ℓ-1,d) to d-neighbor
                     endif
   endif
```

(b)

Figure 3. Structuring Algorithm for Linear Array on Grid

```
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≥2 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
```

Figure 4. Structuring Algorithm for Square Array

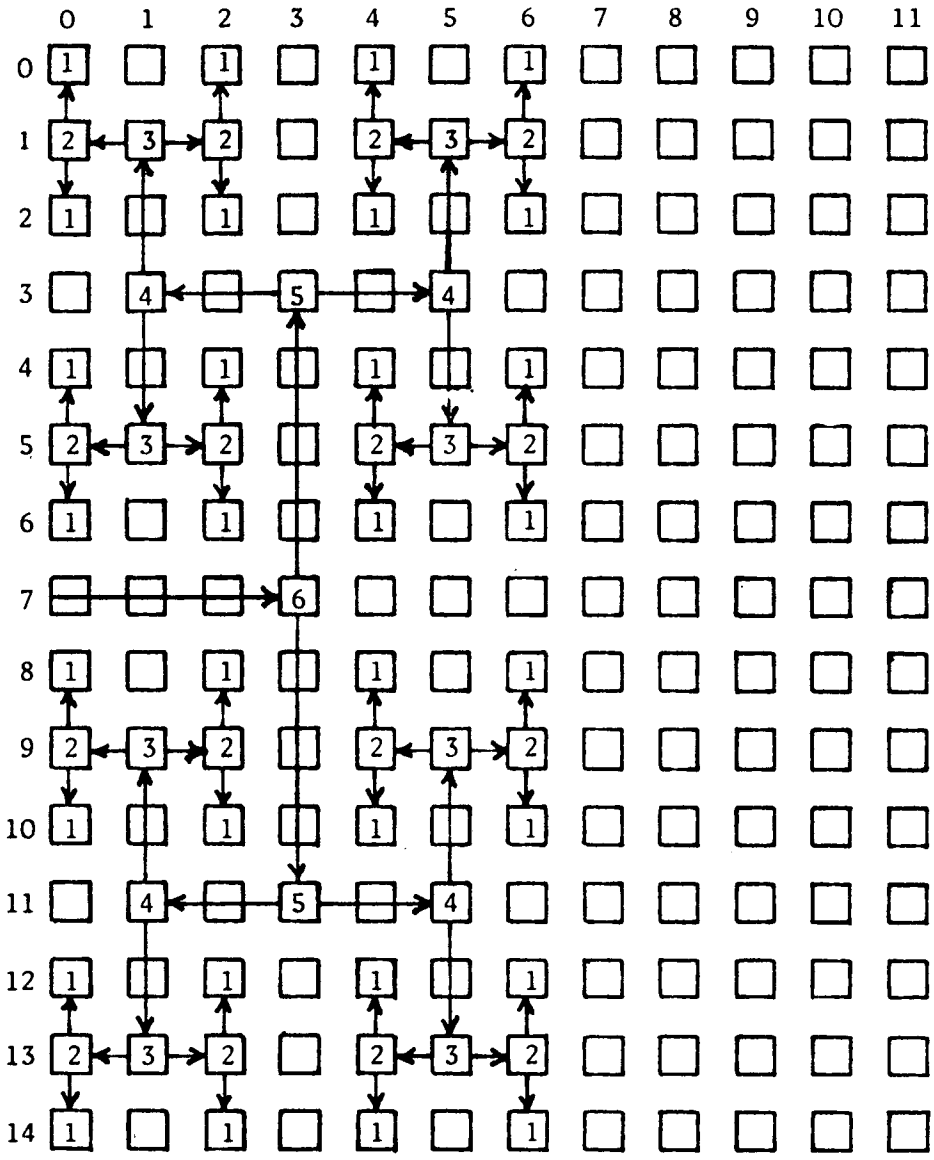


Figure 5. Type 1, Six-Level Binary Tree

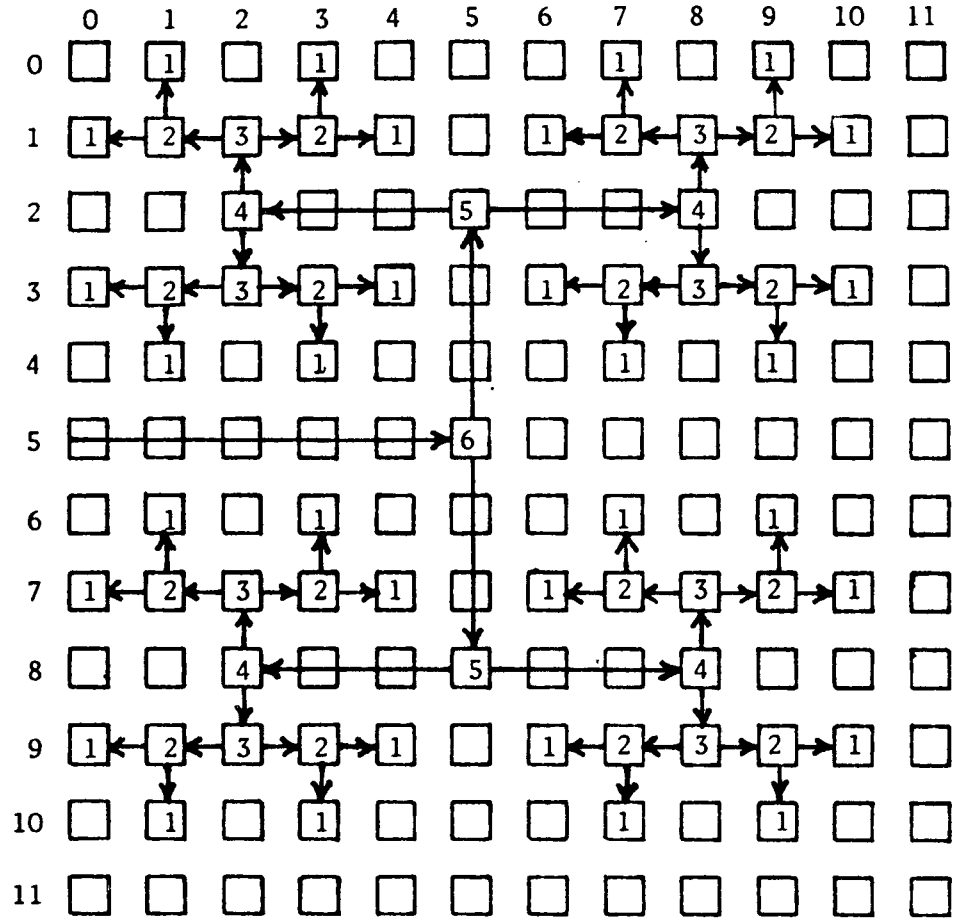


Figure 6. Type 2, Six-Level Binary Tree

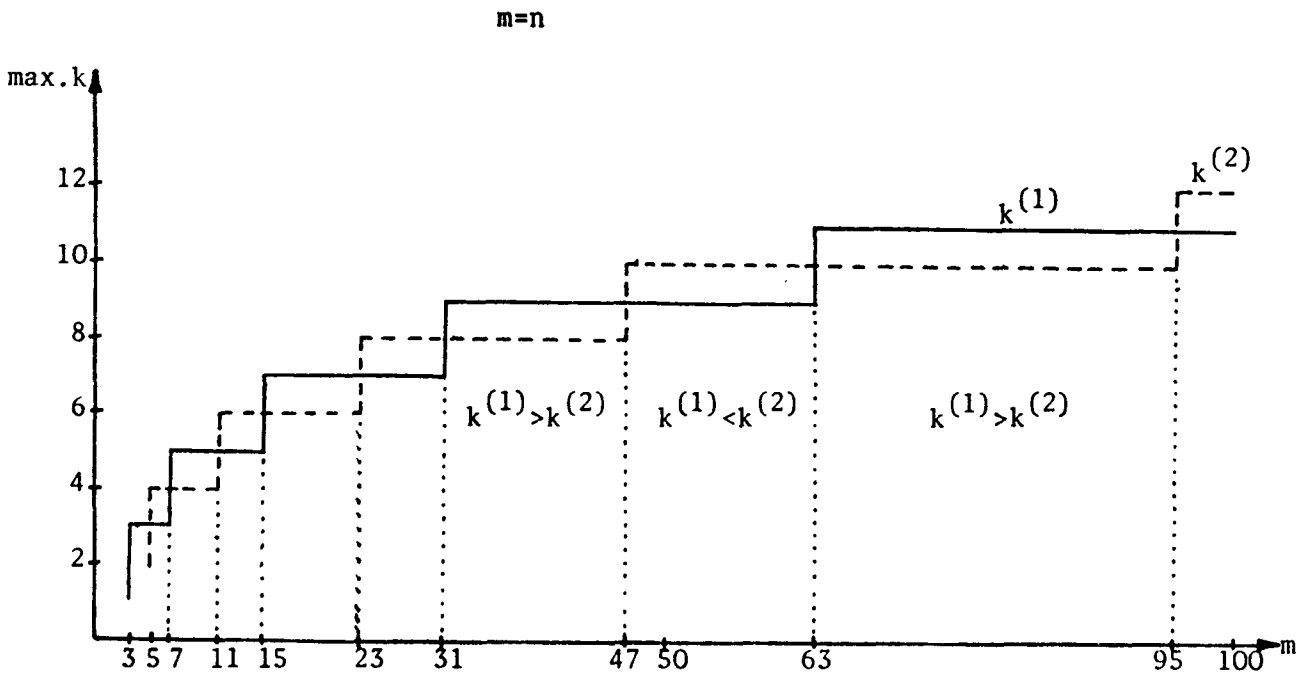


Figure 7. Maximum Number of Levels in Binary Tree that Can Be Placed on *m* x *m* Grid; *m* = *n*

```
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 ε {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
       c:=2^[(ℓ-2)/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
```

Figure 8. Structuring Algorithm for Type 1 Binary Tree on Grid

```
  receive a message M(BT,ℓ,c,d) from δ-neighbor
  (ℓ=k,k-1,...,1 ; c = 3·2^[(k-4)/2] -1,...,1,0 ; d ε {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
 c:= if ℓ≥5 then (3·2^[(ℓ-5)/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
```

Figure 9. Structuring Algorithm for Type 2 Binary Tree on Grid

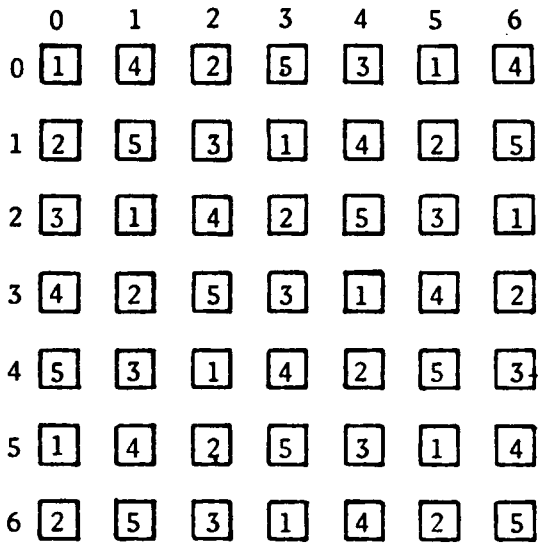
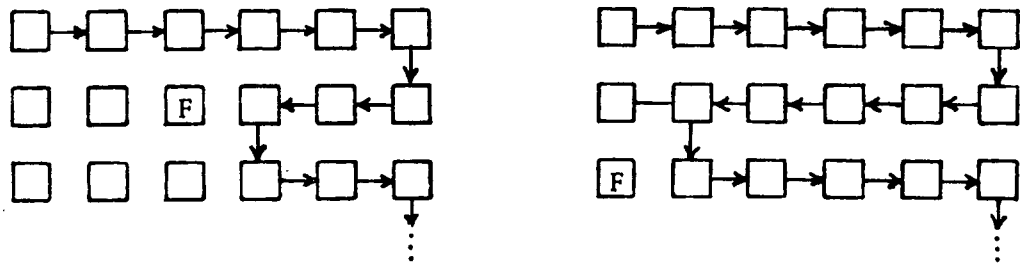


Figure 10. Internal Testing Strategy



(a) (b)

```
receive a message M(LA,ℓ,d) from δ-neighbor
(ℓ=k,k-1,...,1  ;  d ε {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)

Figure 11. Structuring Algorithm for Linear Array (Faulty processors are present.)

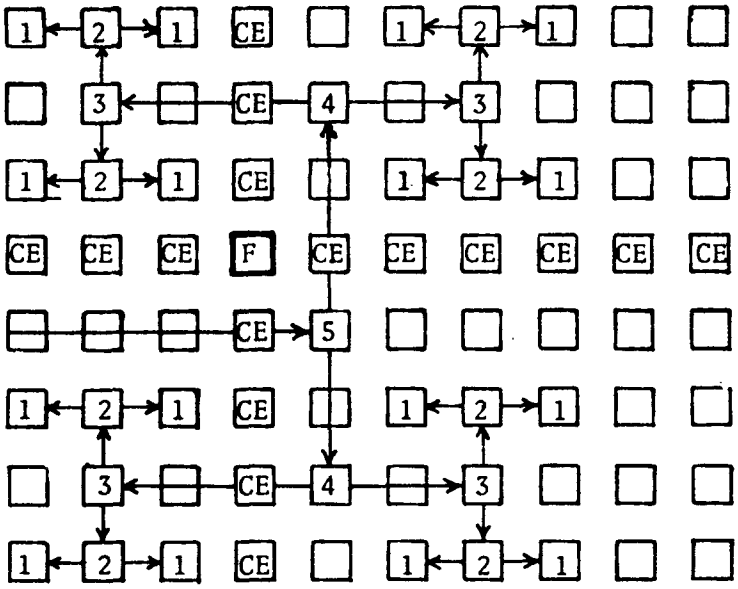


Figure 12. Five-Level Binary Tree on Grid (A faulty processor is present.)