An I/O Algorithm and a Test Algorithm for a Reconfigurable Cellular Array

by

Kathleen L. Connell

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Electrical Engineering

APPROVED:

Dr. J. C. McKeeman, Chairman

Dr. H. F. VanLandingham

Dr. M. Nadler

June, 1985

Blacksburg, Virginia

An I/O Algorithm and a Test Algorithm for a Reconfigurable Cellular Array

by

Kathleen L. Connell

Dr. J. C. McKeeman, Chairman

Electrical Engineering

(ABSTRACT)

Recent advances in VLSI technology have stimulated research efforts in the area of highly reliable fault tolerant, general purpose computing systems, notably, parallel systems. An automatically reconfigurable, fault-tolerant, parallel architecture is suited to VLSI technology. The architecture, a uniformly interconnected array of identical cells, is capable of functional reconfiguration as well as fault reconfiguration. Microprocessor cells are suggested as the "fabric" for implementation of the array. This thesis also introduces an I/O algorithm as an extension to the reconfiguration process, and outlines the steps by which the array cells construct paths from the active-array to the cellular array I/O ports. Path reconfiguration is presented as the method by which fault-free paths replace faulty paths. A testing algorithm is described for use in the self-testing operation of the array. The types of tests that are conducted on cells are outlined, and the basis by which a cell

determines the faulty or fault-free status of a cell is described.

ACKNOWLEDGEMENTS

The author wishes to express sincere appreciation to Dr. John C. McKeeman. His guidance, hard work, and patience throughout her research were essential in making the writing of this thesis possible. Gratitude is also due to Dr. M. Nadler and Dr. H. F. VanLandingham. This research was sponsored by the Army Research Office under grant 3520851 and the author acknowledges this support.

TABLE OF CONTENTS

CHAPTER I INTRODUCTION 1

1.1 PURPOSE 1

1.2 THE CELLULAR ARRAY 2

1.3 THESIS ORGANIZATION 5

CHAPTER II RECONFIGURABLE CELLULAR ARRAYS 7

2.1 INTRODUCTION 7

2.2 TESSELLATION AUTOMATA 7

2.2.1 The Model 8

2.2.2 The Moore and Von Neumann Indices 12

2.3 FAULT RECONFIGURATION 16

2.4 FUNCTIONAL RECONFIGURATION 24

2.4.1 Switching Elements in the Cellular Array . . 27

2.4.2 Alignment Problem and an Alternate Solution . 34

CHAPTER III THE ARRAY CELL 39

3.1 INTRODUCTION 39

3.1.1 The Control Plane 40

3.1.2 The PE Computational Plane 40

3.1.3 The SE Computational Plane 41

3.2 THE MICROPROCESSOR CELL 41

3.3 PE AND SE SIMILARITIES AND DIFFERENCES 46

3.4 A SWITCH DESIGN 47

3.4.1 The Multiplexers 47
3.4.2 The Decoders 51
3.4.3 The Control Registers 53
3.5 TIME MULTIPLEXING OF THE CELL-TO-CELL BUS LINES . 55
3.6 CELL COMMUNICATION PATTERNS 56
3.6.1 The Checkerboard Communication Pattern . . . 56
3.6.2 The Wave Communication Pattern 60

CHAPTER IV ARRAY I/O ALGORITHM 63
4.1 INTRODUCTION 63
4.1.1 Array I/O Ports 64
4.1.2 Data Paths 68
4.1.3 The Active-array Input/Output Cell 75
4.2 THE ALGORITHM 77
4.2.1 The Pointer Cell 79
4.2.2 Direction Priority 81
4.2.3 Choosing the Algorithm Parameters 100
4.4 THE ALGORITHM IN A PROGRAMMING LANGUAGE 101
4.5 CONCLUSION 114

CHAPTER V CELL TESTING TECHNIQUES 115
5.1 INTRODUCTION 115
5.2 SELF-TEST AND BIT 115
5.3 FUNCTIONAL TESTING FOR THE MICROPROCESSOR . . . 117
5.4 TESTING THE PERIPHERAL DEVICES 121
5.4.1 ROM Test 121

5.4.2 RAM Test 122
5.4.3 Switch Test 123
5.4.4 Cell I/O Port and Cell-to-cell Bus Test . . 124

Chapter VI THE TESTING ALGORITHM 126
6.1 INTRODUCTION 126
6.2 TEST DIVISIONS 128
6.2.1 The CPU Test Division 129
6.2.2 Peripheral Device Test Division 133
6.2.3 Computational Plane Test Division 135
6.2.4 State Test Division 138
6.3 TEST SYNCHRONIZATION 140
6.4 START-UP TEST 141
6.5 THE BASIS FOR INTERPRETING A TEST RESULT . . . 142
6.6 THE STATUS REGISTER 146
6.7 EXAMPLES 151
6.8 CONCLUSION 161

CHAPTER VII CONCLUSION 163

REFERENCES 166

Vita 169

LIST OF ILLUSTRATIONS

Figure 1. A Cellular Computer 4

Figure 2. The Tessellation Array 10

Figure 3. The Neighborhood Index 11

Figure 4. Moore Neighborhood Index 13

Figure 5. The Von Neumann Neighborhood Index 15

Figure 6. A Cell and Its Neighboring Cells 18

Figure 7. An Example of Kumar's Proposed Circuitry . 25

Figure 8. The Flip Flop and Register Bits 26

Figure 9. a) Banyan Network b) Embedding the Banyan Network 28

Figure 10. a) Fault-Tolerant Architecture b) Embedding the Fault-Tolerant 29

Figure 11. a) Hyper Tree b) Embedding the Hyper Tree 30

Figure 12. a) Lens Structure b) Embedding the Lens Structure 31

Figure 13. Example Switch Configurations with Crossover Two or Less 33

Figure 14. A Solution to the Alignment Problem Using Space Values 37

Figure 15. Space Required to Grow a Pattern Centered on a PE, and 36

Figure 16. A Two-Dimensional Representation of a Cell 42

Figure 17. General Purpose Microprocessor System . . . 44

Figure 18. Inputs and Outputs to a switch MUX 48

Figure 19. a) A MUX for the Sixth Bit Line of Bus E b) A MUX Block 50

Figure 20. Enable and Select Control for a Switch of Crossover One. 52

Figure 21. Enable and Select Control for a Switch of Crossover Two. 54

Figure 22. The Checkerboard Communication Pattern . . 57

Figure 23. Moore Wave Communication Pattern 59

Figure 24. The Von Neumann Wave Communication Pattern 61

Figure 25. I/O Ports, I/O Paths, and Active Array . . 65

Figure 26. Tag Bit Lines 67

Figure 27. Path Reconfiguration a) Time 1 b) Time 2 . 71

Figure 28. Path Reconfiguration a) Time 3 b) Time 4 . 72

Figure 29. Path Reconfiguration a) Time 5 b) Time 6 . 73

Figure 30. Path Reconfiguration a) Time 7 b) Time 10 74

Figure 31. Enveloping Due to Poorly Situated Active-array Ports 78

Figure 32. Backtracking 85

Figure 33. Direction Priority of N,W,S,E 87

Figure 34. Maximal Availability Priority, E,N,W,S -- Path 1 90

Figure 35. Maximal Availability Priority, E,N,W,S -- Path 2 91

Figure 36. Maximal Availability Priority, E,N,W,S -- Path 3 92

Figure 37. A Path Barrier 93

Figure 38. Handshake Collision 95

Figure 39. Solutions to the Collision Problem 97

Figure 40. Head Start Time 99

Figure 41. Test terminology 127

Figure 42. Interpreting Test Results 143

Figure 43. Illustrating the Status Register 148

Figure 44. Communication Faults 152
Figure 45. CPU Faults 154
Figure 46. Peripheral Device Test 156
Figure 47. Computational Plane Test 157
Figure 48. State Test 159
Figure 49. A Test Pass 160

CHAPTER I INTRODUCTION

1.1 PURPOSE

Recent advances in VLSI technology have made it possible to place several processors on a chip. This technology has stimulated research efforts in the area of highly reliable fault-tolerant, general purpose computing systems, notably, parallel systems. The current research in the Electrical Engineering Department at VPI and SU involves designing an automatically reconfigurable, fault-tolerant, parallel architecture that is particularly suited to VLSI technology. The architecture, a uniformly interconnected array of identical cells, is capable of functional reconfiguration as well as fault reconfiguration. The functional reconfiguration capability renders the architecture useful as a general purpose parallel computer, while the fault reconfiguration contributes to fault tolerance in the system.

The operation of the cellular array will require data communication with a source outside of the array. Because the cellular array is reconfigurable, the position of the active cells in the array changes. Consequently, the mechanism by which data is communicated to and from the active cells must change and will be described as an I/O algorithm.

The fault reconfiguration design is based on the assumption that faults in the array can be detected and located by the process of cells testing other cells within the array. This process constitutes a testing scheme, and needs to be developed for the actual design of the array. Therefore, the purpose of this research is to describe an I/O algorithm and a testing algorithm that can be applied to the reconfigurable architecture.

1.2 THE CELLULAR ARRAY

Cellular arrays are collections of identical cells that are connected in the form of regular interconnection structures. These arrays are suitable for implementing algorithms that have a substantial amount of inherent parallelism. Fault-tolerance is incorporated into the array by using more cells than are necessary for performing the computational task assigned to the array. The array contains an active sub-array which performs the computation, surrounded by inactive cells. If one or more of the active cells fail, the active sub-array can be reconfigured to a fault-free region of the array.

Because the sub-array can be embedded anywhere in the array, each cell must be polymorphic, that is, capable of performing more than one function. The cells can have any one of several local states, each of which characterizes one

of the cell functions. The cells of the sub-array form a pattern of local states that constitute the global function of the array.

Each cell can be thought of as consisting of a **control hyperplane** and a **computational hyperplane**, as illustrated in Figure 1 on page 4. The control mechanism which oversees pattern construction and reconfiguration is distributed, and is contained in each of the cells. The part of the cell in the control hyperplane defines its local state, and the part in the computational hyperplane executes the function defined by the local state.

The structure described above is a reconfigurable cellular array. The array control hyperplane can be "programmed" to implement any parallel architecture by associating the architecture with a global function. However, because most of the architectures can be used for only a small number of applications, it is desirable to build a general-purpose array that can implement any one of several different global functions. An array with the capability of automatic fault and functional reconfiguration has been the subject of investigation at VPI and SU. Most recently, Kumar [12] addressed the control of fault reconfiguration while Gollakota [7] addressed the control of functional reconfiguration.

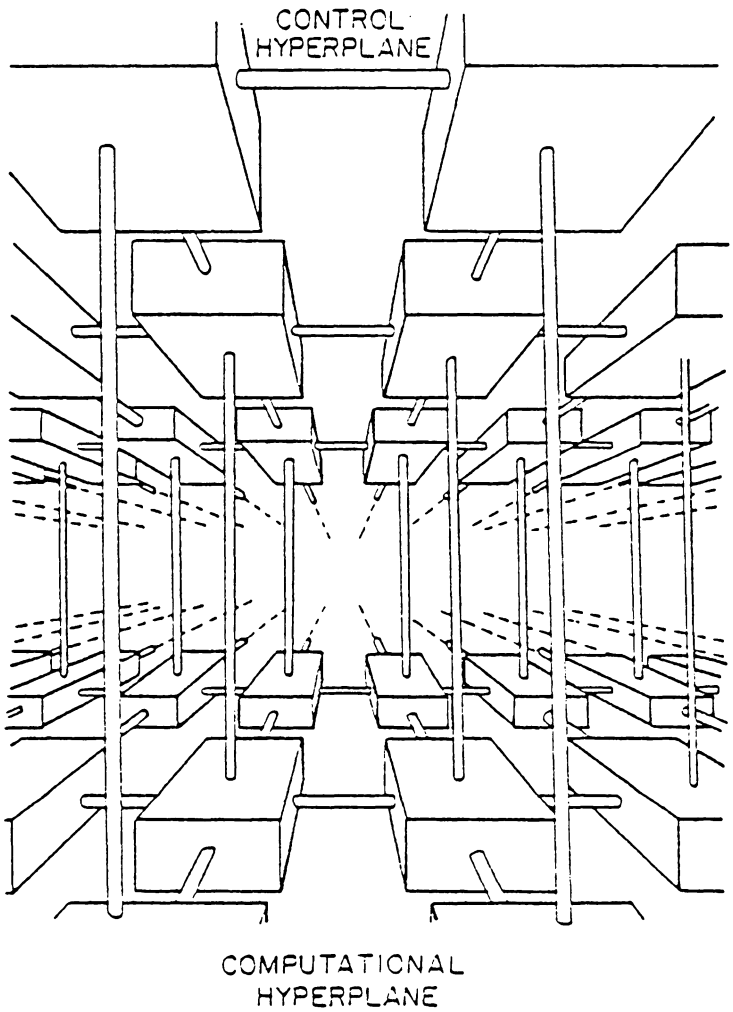


Figure 1. A Cellular Computer

The array design is by no means complete. In order to implement the array in hardware, several problems must be tackled, including an implementable scheme of fault detection and I/O. The first step for implementing fault detection and I/O is to present an algorithm for each that can eventually be translated into hardware and/or software. The chapters of this thesis, outlined in the following section, describe I/O and fault detection algorithms based on the assumption that the cells are implemented by a general-purpose microprocessor.

1.3 THESIS ORGANIZATION

Chapter 2 explains the cellular architecture in detail. First, the motivation for construction of a cellular array is explained, followed by a summary of fault reconfiguration and functional reconfiguration. Also, a solution to Gollakota's alignment problem [7] is suggested. Chapter 3 discusses the individual cells of the array, and suggests that the control cells be implemented by a microprocessor. A combinational switch design that meets the array requirements is presented. Also, the way in which cells communicate is discussed for use in the testing algorithm. Chapter 4 presents an I/O algorithm for the array. The motivation for the algorithm steps is followed by a summary of how to choose the algorithm parameters and a formalization of the algorithm

steps. Chapter 5 discusses current testing techniques that can be applied to the cells of the array, and Chapter 6 then incorporates these techniques into a testing algorithm.

CHAPTER II RECONFIGURABLE CELLULAR ARRAYS

2.1 INTRODUCTION

A generalized architecture in which specific computing systems may be implemented is described as a background "fabric." This "fabric" is a multidimensional cellular array where the cells are identical and uniformly interconnected. The concepts of the cellular array, and the research done by Kumar and Gollakota on the array, is described in this chapter.

2.2 TESSELLATION AUTOMATA

The cellular array of the reconfigurable architecture consists of two parallel hyperplanes -- the control hyperplane and the computational hyperplane, as shown in Figure 1 on page 4. The control hyperplane is responsible for determining the individual cell function for the system being realized. Specifically, the control hyperplane of the cell receives information about the global state, or the overall function, of the array from a finite number of neighboring cells, and in turn communicates this information to its neighbors in the same manner in which it was received. This

procedure is explained in more detail in the section entitled Fault Reconfiguration.

The control hyperplane can be modeled as Tessellation Automaton as described by Yamada and Amoroso [25]. The importance of a Tessellation Automaton modeled control plane is that any finite pattern can evolve from a primitive pattern if the cells of the array are connected to four or more other cells in the array.

2.2.1 The Model

Yamada and Amoroso formally describe a Tessellation Automaton in the following manner:

> A Tessellation Automaton is a mathematical model of an infinite array of uniformly interconnected identical finite state machines. Each machine is capable of changing state at discrete time steps as a function of the states of other machines in the array and inputs that act as environmental changes to the array.

This description motivated them to define a Tessellation Automata as a quadruple (A, E^d, X, ∇) where A denotes the state alphabet, E^d denotes the tessellation array, X is the neighborhood index, and ∇ is the set of mappings.

A - The State Alphabet

The **state alphabet** is a finite non-empty set of symbols that corresponds to the state of any one of the finite state machines of the array. If a machine, M, can assume any of the states B, C, D, and E, then A={B, C, D, E} is the state alphabet for machine M.

E^d - The Tessellation Array

The **tessellation array** is a set of d-tuples of integers where d is the dimension of the tessellation automaton. If a tessellation array is two-dimensional, then E^d will be of the form,

$$E^d = \{\ldots (i-1,j),(i,j),(i+1,j)\ldots \ldots (i-1,j+1),(i,j+1),(i+1,j+1)\ldots\}.$$

The Tessellation Array is illustrated in Figure 2 on page 10.

X - The Neighborhood Index

X is the **neighborhood index** of the tessellation automaton. X is an n-tuple of distinct d-tuples of integers, where d is the tessellation dimension. Also associated with the neighborhood index is the **neighborhood of cell i**, denoted by the n-tuple N(X,i). In other words, the neighborhood of cell i corresponds to the finite set of cells to which cell i has direct connections. For example, if X=((0),(-1)) for a one

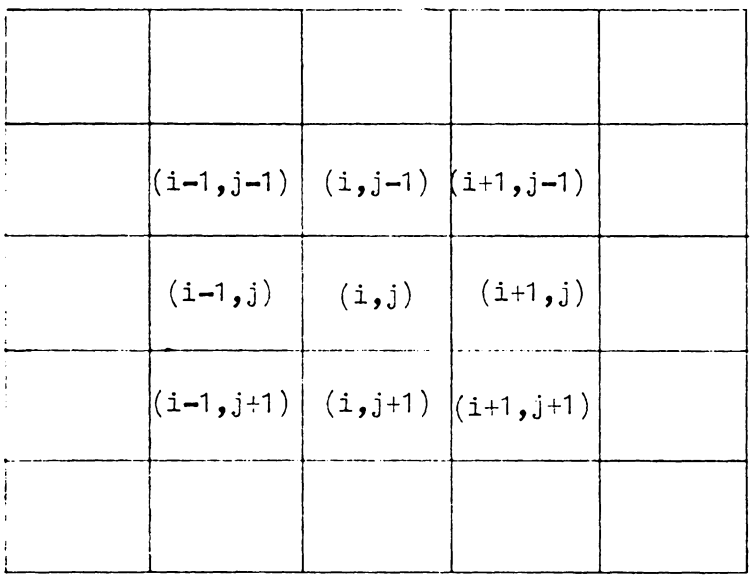
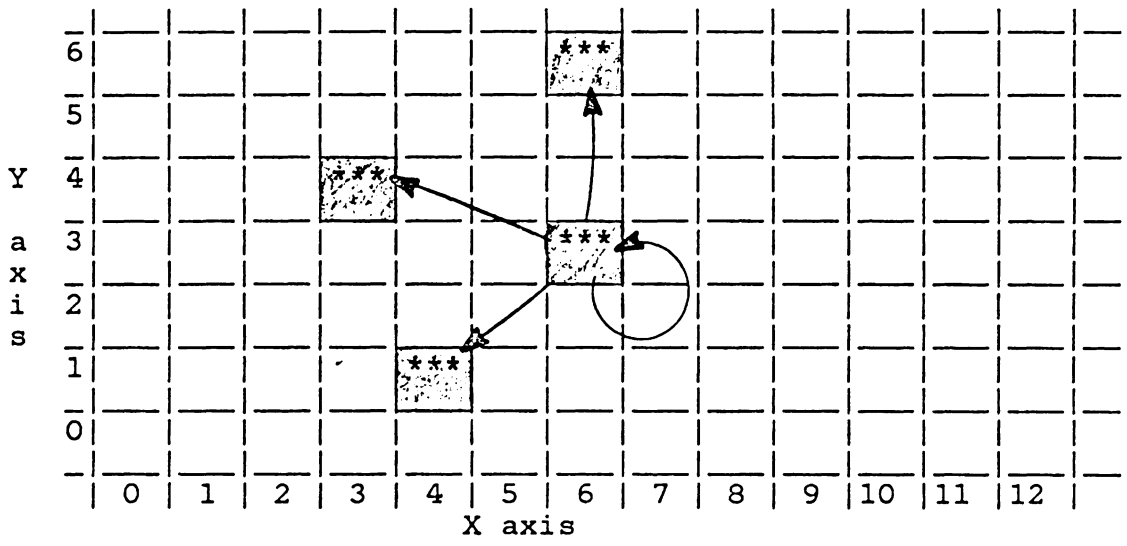


Figure 2. The Tessellation Array



X=((0,0),(0,3),(-3,1),(-2,-2))

for cell (6,3) N(X,i)=((6,3),(6,6),(3,4),(4,1))

Figure 3. The Neighborhood Index

dimensional array, then for cell 9, N(X,9)=(cell 9, cell 8). A two dimensional example is illustrated in Figure 3 on page 11.

σ - The Local Transformation

σ is a mapping from A^n to A called the local transformation. Each cell decides its next state by looking at the present state of its neighbors, i.e., the present states of a cell's neighbors are mapped into the next state.

If X=(-1,1) is the neighborhood index, then each cell looks at its neighbors to its immediate left and its immediate right. The cell uses the states of these neighbors to decide its next state, according to the mapping, σ. If σ(AB)=C, then any cell with its neighbors in states A and B will change to state C at the next time step. σ is defined so that the control plane can reach any desired configuration.

2.2.2 The Moore and Von Neumann Indices

Both Moore [16] and Von Neumann [24] considered models studying self-reproducing automata. Moore was the first to call these automata "tessellation," and considered both self-reproduction and Garden-of-Eden configurations.

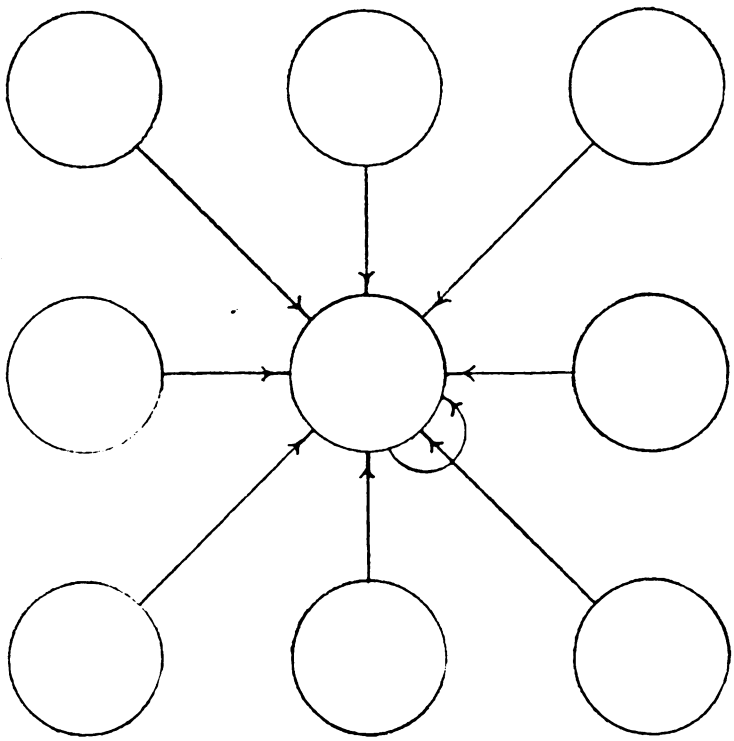


Figure 4. Moore Neighborhood Index

Garden-of-Eden configurations are initial configurations that once exited, can not reoccur in automata. Moore concluded that a tessellation automaton was not self-reproducing if its configuration contained a Garden-of-Eden configuration. The Moore Index is shown in Figure 4 on page 13.

Von Neumann also studied the problem of self-reproduction in two-dimensional automata. He showed that the automata could repeatedly duplicate the state information of the cells within the array. The Von Neumann Index is shown in Figure 5 on page 15.

The control plane of the reconfigurable array can be characterized as a two-dimensional tessellation automaton. To completely describe the control hyperplane, the neighborhood index, the local transformation, and the state alphabet must be specified. The neighborhood index is the Von Neumann index, and each state of the state alphabet corresponds to one of the functions of each cell. If each state of the state alphabet is defined as a **local state**, then the local transformation depends upon the number of local states that are needed to implement the global function. The control plane requirements are described in the next sections on fault reconfiguration and functional reconfiguration.

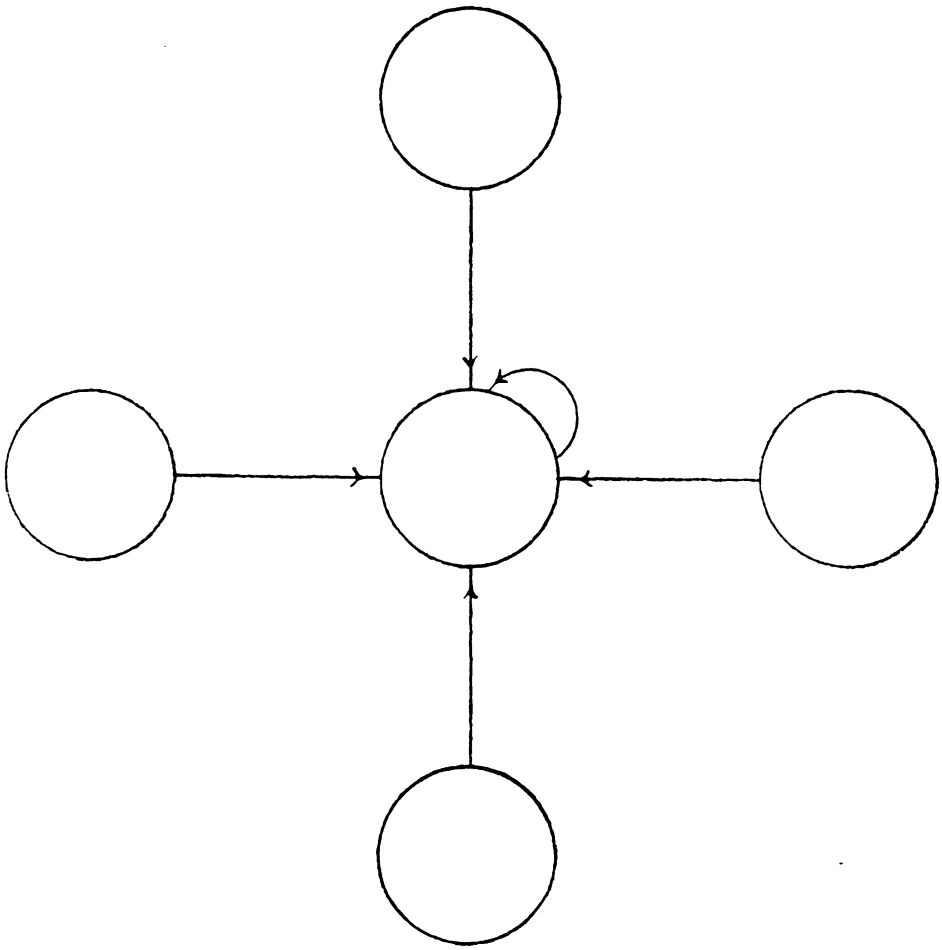


Figure 5. The Von Neumann Neighborhood Index

2.3 FAULT RECONFIGURATION

Kumar investigated fault reconfiguration in the cellular array. He showed that reconfiguration by means of a next-state mapping is a simple sequential process that can be implemented with a lesser interconnection and hardware complexity than existing strategies. He designed and analyzed a simple system using his reconfiguration techniques. This section summarizes his reconfiguration algorithms.

The Growth Process

Global patterns can be grown using either the Moore Growth Process or the Von Neumann Growth Process. The Moore Growth Process grows final patterns using d-dimensional hypercubes of increasing size. Final patterns can be arbitrary in shape, with some of the neighborhoods being mapped to the quiescent state. The Von Neumann growth process grows patterns in a similar manner using d-dimensional hyperdiamonds. This process, which requires a less complex state mapping and interconnection scheme, and which uses the same Von Neumann Neighborhood Index, is used as the growth process.

σ grows arbitrary patterns if all neighborhood configurations in the intermediate growth patterns are distinct. The local function of a cell and the overall global function

are contained in each cell in the **state register**, denoted **SR**. Therefore, the SR has two partitions -- the local state register (LSR) and the global state register (GSR). The next state mapping computes its results on the basis of both the LSR and the GSR.

Every cell has eight neighbors. The neighbors are denoted by their directions with respect to the cell as illustrated in Figure 6 on page 18. For example, 'N' indicates the north neighbor, 'NE' indicates the northeast neighbor, etc.. This method of labeling the neighbors of a cell is used throughout the remaining chapters.

Example 2.3.1

Let the Von Neumann Neighborhood be used to define the local transformation, such that for a neighborhood represented by the notation,

North_neighbor/West_neighbor/Self/East_neighbor/South_neighbor,

σ(N/W/SELF/E/S) is defined by

```
0/0/s1/0/0  -->  c1
0/0/0/0/s1  -->  a1
s1/0/0/0/0  -->  a1
0/s1/0/0/0  -->  b1
0/0/0/s1/0  -->  b1,
```

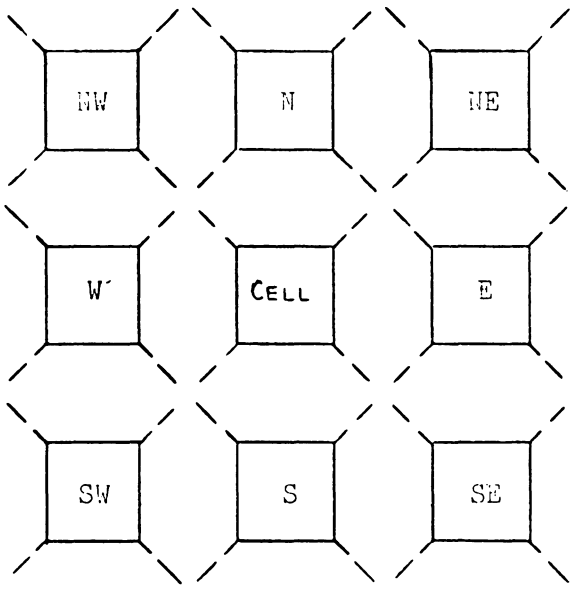


Figure 6. A Cell and Its Neighboring Cells

then the following Von Neumann growth pattern results from a seed, s1.

```
o  o  o        o   a1  o
o  s1 o        b1  c1  b1
o  o  o        o   a1  o
  t=0             t=1
```

Test Result Interpretation

The cellular array clock is divided into the computational and the diagnostic phases so that all fault-free cells go into the two modes synchronously. Since the modes do not overlap in time, the lines connecting cells are effectively multiplexed between the computational and diagnostic phases. Computational and diagnostic results must be saved before switching from one mode to another.

In the diagnostic mode, cells conduct tests on their neighbors. The tests consist of reiterations of the same finite test sequences. Test sequences should have a high coverage, i.e. they should be capable of detecting most failures. A cell changes its local state to the quarantine state according to the following rule.

Q Rule. After each pass of the diagnostic phase, a cell makes a transition to a quarantine state, Q, if it de-

termines that one of the cells directly connected to it is faulty. A quarantine cell disconnects itself from faulty neighbors via internal switching mechanisms.

Consequences of Rule 1.

1) walls of quarantine state cells separate regions of faulty cells from fault-free regions of cells.

2) Fault-free regions that are completely surrounded by faulty cells are effectively disconnected from the rest of the array.

Example 2.3.2

Assuming the Von Neumann Neighborhood Index, the following faulty cells, indicated by an X, are quarantined as shown.

```
- - - - X                    - - - Q X
- - - - -                    - - Q - Q
- - X - -        ----->      - Q X Q -
- X X - -                    Q X X Q -
- - - - -                    - Q Q - -
```

The Seed State and Fault-free Spaces

For pattern reconstruction, information about the global pattern is extracted from the remaining fault-free cells in the pattern, and is "crystallized" into a single seed state. This seed must traverse the array in search of a sufficiently large fault-free region for successful pattern regrowth. Kumar described a distributed algorithm that enables a cell to determine the largest fault-free space around it, and assigns this size an s-value (space value). Since the seed cell is aware of the global pattern dimension, it compares the desired dimension value with the s-value in each cell that it encounters during migration. If the s-value computed by a cell is large enough, the seed comes to rest and initiates pattern regrowth.

Each cell is assigned an s-value according to the following rules:

1) Each cell in the Q state has and s-value of -1.

2) Boundary cells have s-values of 0.

3) If k cells directly connected to cell, C, have s values s(1), s(2), at time i, then at time i+1 the s-value of C is min(s(1),s(2),....s(k))+1.

S-values are updated until equilibrium is reached.

Example 2.3.3

For an array with quarantine regions, the final s-values are as follows, where t=-1:

```
- - - - - - -          0 0 0 0 0 0 0
- - - - - - -          0 0 0 0 0 0 0
- - - - - - -          0 0 0 0 0 0 0
- Q - - - - -          0 t 0 0 0 0 0
Q X Q - - - -          t X t 0 0 0 0
- Q - - - - -          0 t 0 0 0 0 0
- - - - Q - -          0 0 0 0 t 0 0
- - - Q X Q -          0 0 0 t X t 0
        time=0                 time=2
0 0 0 0 0 0 0          0 0 0 0 0 0 0
0 1 1 1 1 1 0          0 1 1 1 1 1 0
0 0 1 1 1 1 0          0 0 1 2 2 1 0
0 t 0 1 1 1 0          0 t 0 1 1 1 0
t X t 0 1 1 0          t X t 0 1 1 0
0 t 0 1 0 1 0          0 t 0 1 0 1 0
0 0 1 0 t 0 0          0 0 1 0 t 0 0
0 0 0 t X t 0          0 0 0 t X t 0
   Time=2              Time=3 and Equilibrium
```

Priority and Clearing

The priority algorithm chooses a seed cell from the remaining fault-free cells in the active sub-array. The algorithm requires that each cell in the array be assigned a distinct number, which is its priority value. After a fault has been identified in the active sub-array, the cell in the quarantine state that has the global state (a cell that has the global state is part of the active sub-array) and the highest priority value is chosen as the seed cell, and is labeled Y1.

The array is cleared by clearing the state registers (both LSR and the GSR) according to the following rules:

1) A non-quarantine cell with a Q or a Y1 neighbor makes a transition to the 'z' state at the next time step.

2) Any non-'z' non-quiescent cell with a 'z' neighbor makes a transition to the 'z' state at the next time step.

3) Any cell in the 'z' state makes a transition to the quiescent state at the next time step.

Seed Ejection and Migration

After clearing and determination of the s-value, the seed migrates from the seed source. In example 2.3.3, the seed

migrates to one of the cells marked '2,' since these cells are the centers of the largest fault-free areas. The pattern regrowth centers on one of these cells, if the two by two area is large enough.

Upon completion of the above algorithms, the array has reconfigured. Kumar investigated the practicality of the reconfiguration scheme by example, and presents the circuitry required for implementation of the algorithms. The circuitry, which consists of 94 bits of flip flops and registers, and combinational logic, will be referred to in latter chapters as the **register-logic** control circuitry. An example of his proposed circuitry is shown in Figure 7 on page 25, and the flip flop and register bits for the entire control circuit are enumerated in Figure 8 on page 26. Brighton [5] presents a simulation of Kumar's reconfiguration algorithms.

2.4 FUNCTIONAL RECONFIGURATION

Gollakota added to the functional capability of the array by adding switches to the computational hyperplane. The addition of switches allows many special purpose architectures, such as mesh structures and tree structures, to be embedded in a single architecture. Gollakota's work [7] incorporates the ideas of Snyder's [21] CHiP computer into the reconfigurable array, but eliminates the "hardcore" master

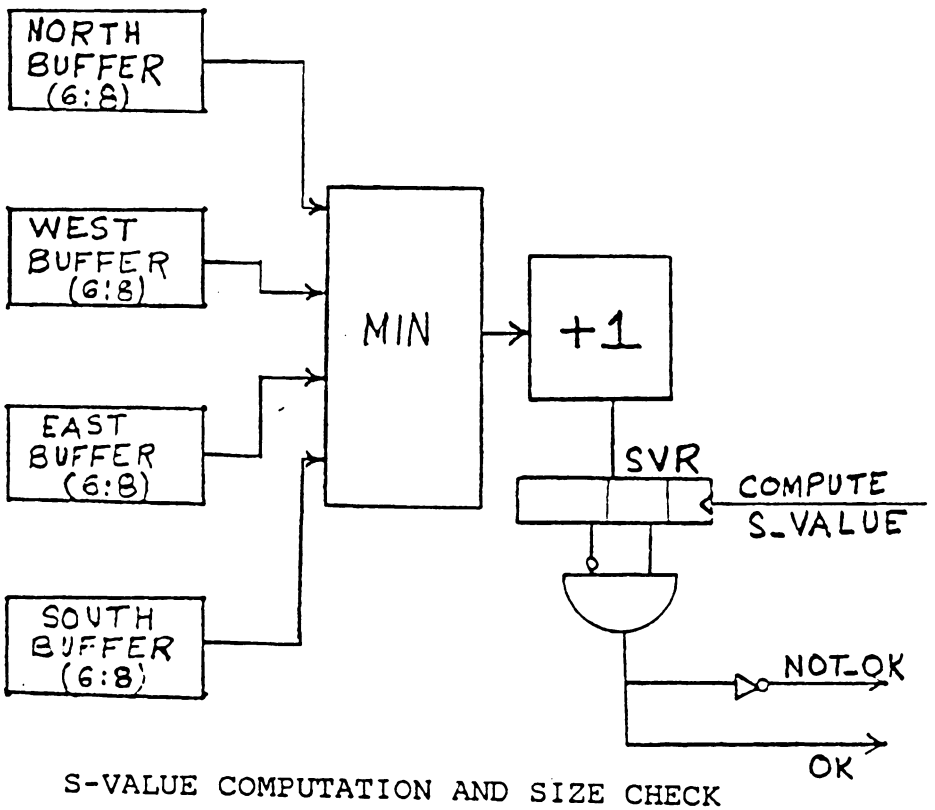


Figure 7. An Example of Kumar's Proposed Circuitry

Flip Flops and Registers	Required Bits
RSF	1 Bit
NF	1 Bit
S/R	1 Bit
Diagnose_mode/Computation_mode	1 Bit
Clock Phases	8 Bits
Disconnect Flags	16 Bits
State Registers	8 Bits
Priority Registers	8 Bits
Buffers	32 Bits
s-value Registers	3 Bits
Clock Divider	3 Bits
RFSM Counter	8 Bits
Decision Vector	4 Bits
Total	94 Bits

Figure 8. The Flip Flop and Register Bits

control circuitry of CHiP. The CHiP computer (configurable, highly parallel computer) provides a programmable interconnection switch structure that is integrated with processing elements. The switches are set by a central controller to best implement the function to be executed. The way in which Gollakota added functional reconfiguration capability to the array is summarized in the following subsection.

2.4.1 Switching Elements in the Cellular Array

The cellular structure consists of the control hyperplane and the computational hyperplane, so that for each cell in the control hyperplane there is an associated cell in the computational hyperplane. Each cell in the computational hyperplane is either a switch or a processing element, and is associated with a local state, which is a variable. The switch local state selects a particular local interconnection of the switching elements to its neighbors, and the processor local state selects a specific function to be performed by the PE. The pattern of local states of the SEs together with the PEs defines a global function to be implemented by the cellular structure.

Gollakota demonstrates that four popular interconnection networks can be embedded in the cellular array. These structures are:

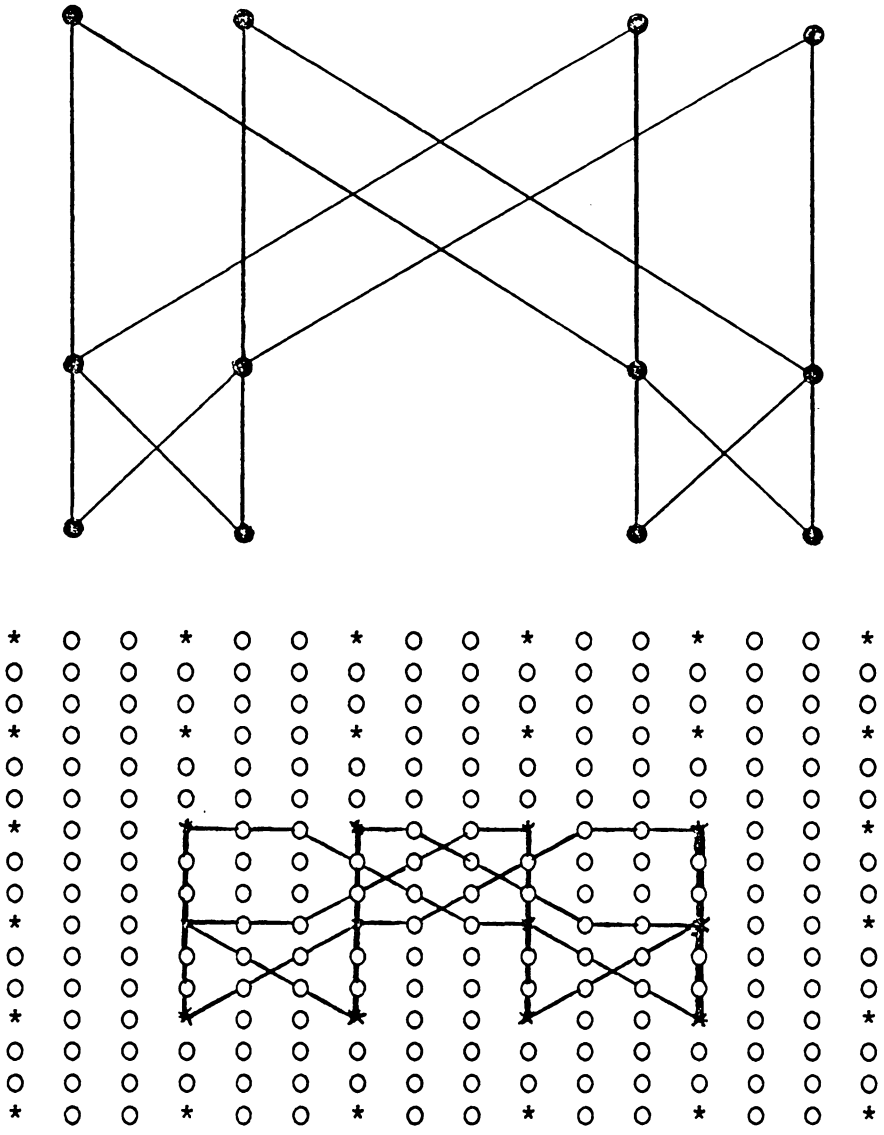


Figure 9. a) Banyan Network b) Embedding the Banyan Network

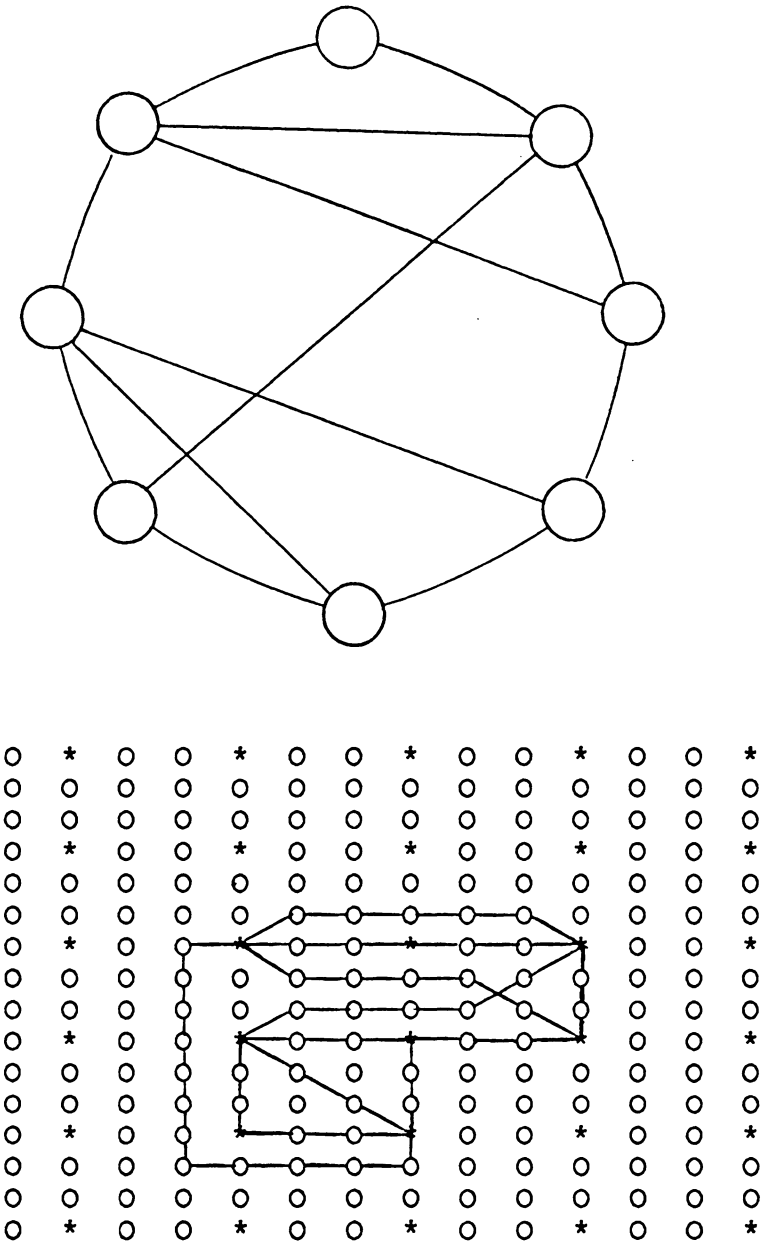


Figure 10. a) Fault-Tolerant Architecture b) Embedding the Fault-TolerantArchitecture

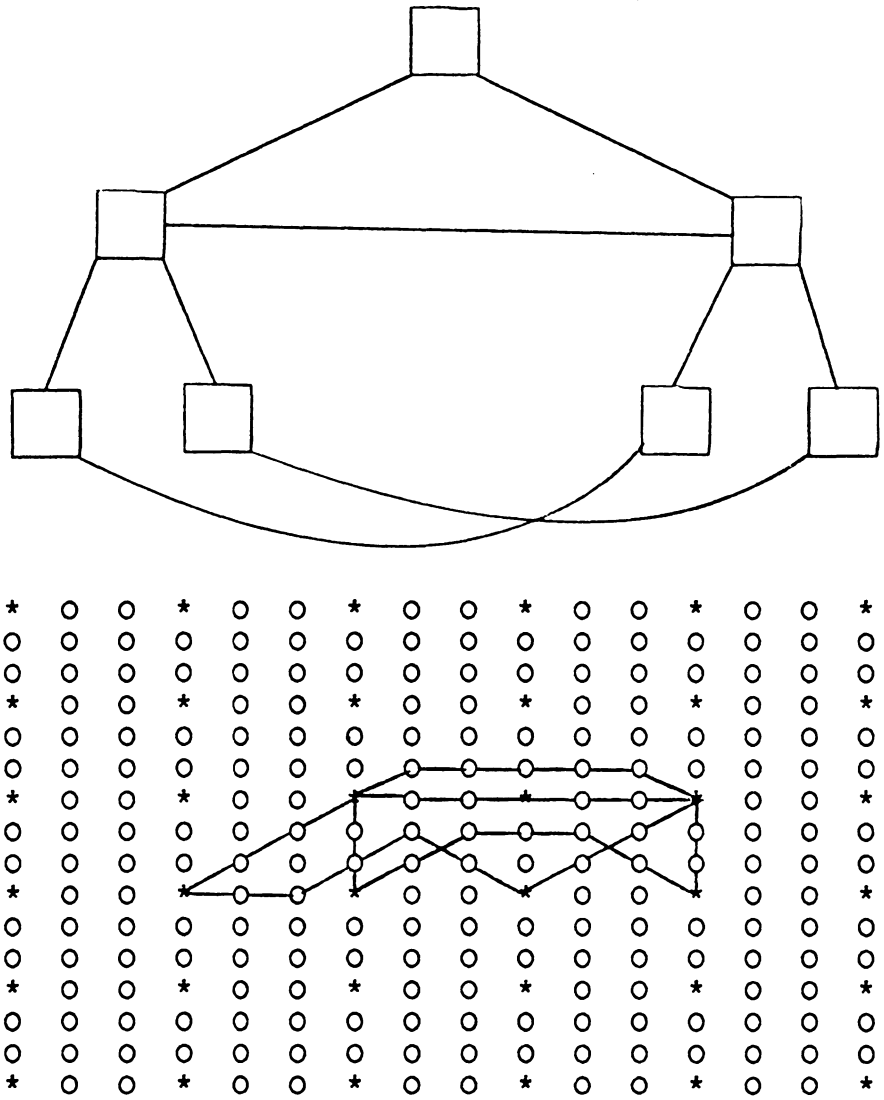


Figure 11. a) Hyper Tree b) Embedding the Hyper Tree

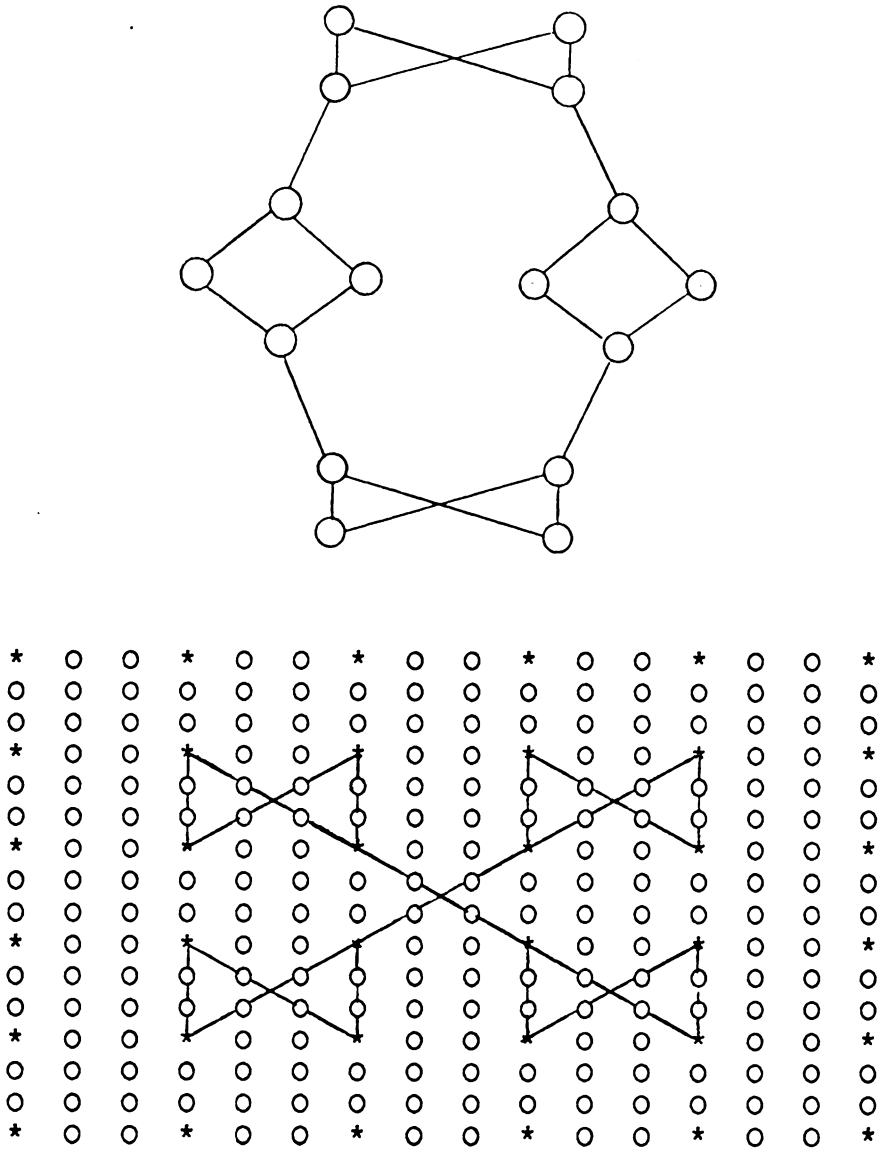


Figure 12. a) Lens Structure b) Embedding the Lens Structure

1) a Banyan network,
2) a fault-tolerant architecture,
3) a hyper tree, and
4) a lens strategy.

These structures and how they are embedded in the array are illustrated in Figure 9 on page 28 through Figure 12 on page 31.

A switch is characterized by the number of incident data paths and the number of bit lines in each data path which are limited by the number of pins available. The number of pairs of data paths that a switch can simultaneously connect is called the **crossover** capability. A high value of crossover increases the number of connections a switch can make, but also increases the number of allowable local states. By comparing different values of the above parameters, Gollakota concludes that a crossover capability of **two** best suits the needs of the array. A switch with crossover capability of two can assume 239 positions, thirty-four of which are illustrated in Figure 13 on page 33. Each line in the figure represents a cell-to-cell bus. The cells in the top five rows show examples of crossover of one, and the remaining row shows example of crossover of two.

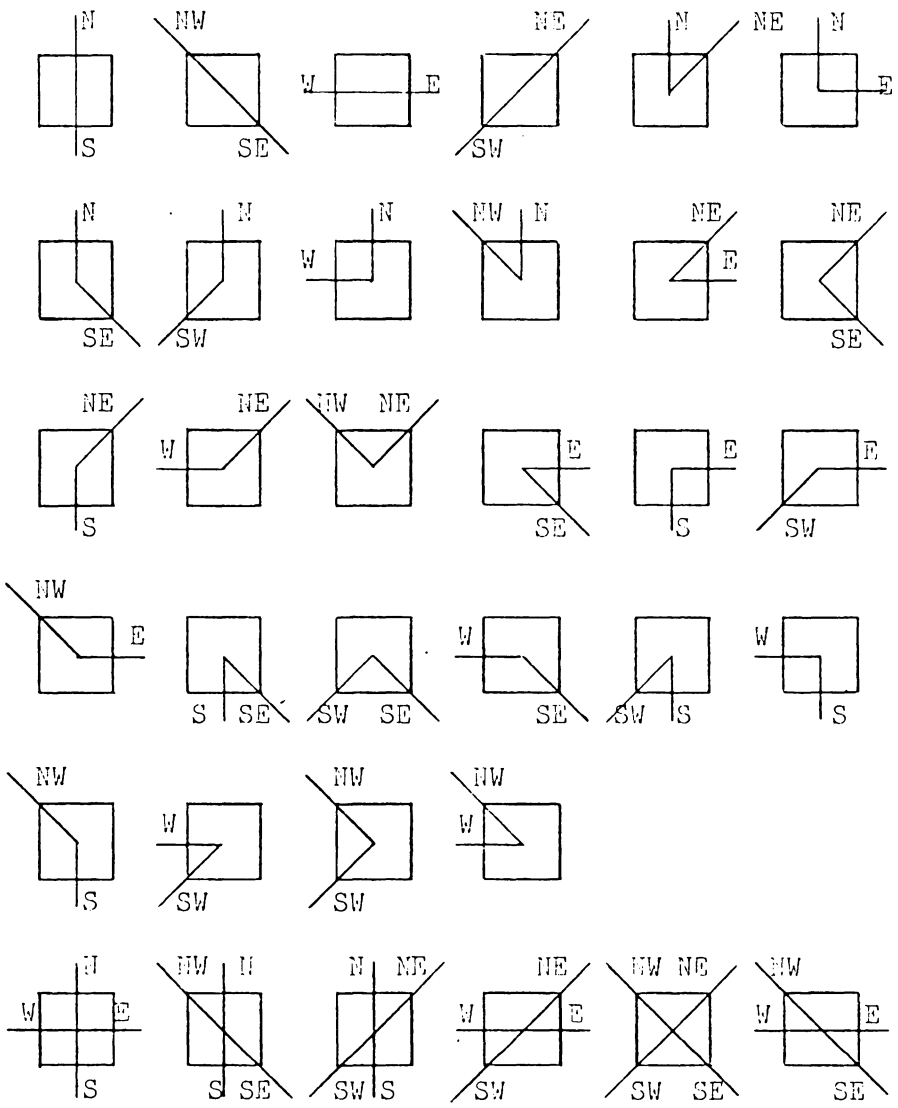


Figure 13. Example Switch Configurations with Crossover Two or Less

2.4.2 Alignment Problem and an Alternate Solution

Gollakota defined the alignment problem as the situation in which a control cell asks an SE to perform a processor task, and/or a PE to do an interconnection. This section presents Gollakota's shifting algorithm solution, and then suggests an alternate solution that requires no additional algorithm.

Gollakota's Solution

Due to asymmetry in the computational plane, a PS register (processor/switch) in a cell keeps track of its cell type. For example, if the PS register is set, then the cell is a processing element. If a cell is asked to do a function that it cannot do, it sets the mismatch register (MM). A set MM register initiates a pattern shifting algorithm that shifts the pattern right and, if needed, up until the pattern is situated properly.

This solution adds another algorithm to the "list" of control algorithms required by the array. Also, this solution does not take into account the available fault-free space, i.e. the pattern may not be able to shift because the "shift-space" contains quarantined and faulty cells. An alternate solution is suggested below.

An Alternate Solution

The PS register is used in this solution, such that if the register is set, then the cell is a processing element. This "register" can be a ROM bit that is either set or not set during manufacture of the PE or SE cell. During seed migration, a seed checks the PS register before it comes to rest. If pattern growth is required to center on a PE, then the seed comes to rest in a PE only. Since pattern regrowth centers on the seed cell, the pattern will be situated correctly.

Instead of adding an algorithm, Kumar's migrate algorithm must be modified to include the presence of switching elements. The migrate algorithm is modified by including a PS register check so that the seed comes to rest on two conditions: 1) the s-value is sufficiently large, and 2) the PS register is set. In Figure 14 on page 37, the largest space available that is centered on a PE is of size three, illustrated by the "*3" cells.

A pattern that does not have a PE in its center requires additional space to grow. If pattern growth is centered on a cell that is closest to the middle of the pattern, then the need for additional growth space is minimized. A four bit PS label can be assigned to each of the switching elements

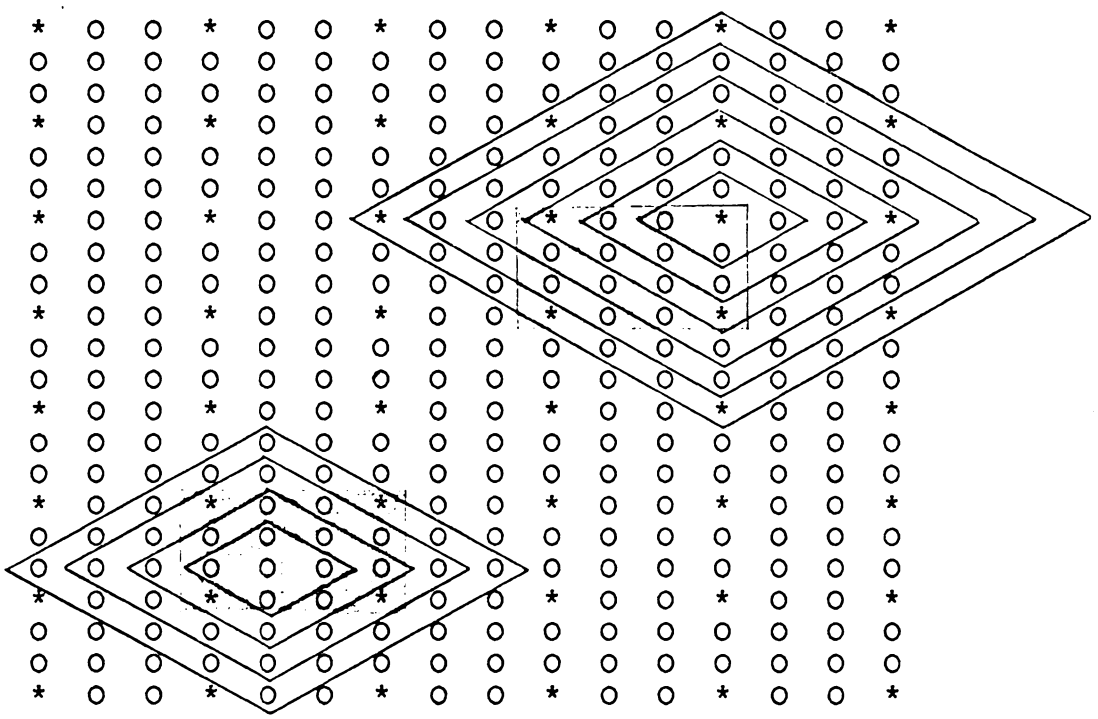


Figure 15. Space Required to Grow a Pattern Centered on a PE, and Centered on a Cell Closest to the Middle of the Array

```
*   O   O   *   O   O   *   O   O   *   O   Q   *
O   O1  O1  O1  O1  O1  O1  O1  O1  O   Q   X   Q
O   O1  O2  O2  O2  O2  O2  O2  O1  O   Q   X   Q
*   O1  O2  *3  O3  O3  *3  O2  O1  *   Q   X   X
O   O1  O1  O2  O3  O4  O4  O3  O2  O1  O   Q   X
O   O1  O   O1  O2  O3  O4  O4  O3  O2  O1  O   Q
*   O   Y1  *   O1  O2  *3  O4  O4  *3  O2  O1  *
O   Q   X   Q   O   O1  O2  O3  O4  O3  O2  O1  O
O   O   Q   O   O1  O1  O1  O2  O3  O3  O2  O1  O
*   O1  O   *1  O1  O   *   O1  O2  *3  O2  O1  *
O   O1  O1  O1  O   Q   Q   O   O1  O2  O2  O1  O
O   O1  O1  O   Q   X   X   Q   O   O1  O2  O1  O
*   O1  O1  *   Q   X   X   Q   O   *1  O2  O1  *
O   O1  O1  O   Q   X   X   Q   O   O1  O2  O1  O
O   O1  O1  O1  O   Q   X   Q   O   O1  O1  O1  O
*   O   O   *   O   O   Q   O   O   *   O   O   *

1,2,3,4   s-values
*         PE with an s-value of 0
O         SE with an s-value of 0
*x        PE with an s-value of x
Ox        SE with an s-value of x
```

Figure 14. A Solution to the Alignment Problem Using Space Values and Cell Types

that surround a PE, and to the PE itself. Then the seed comes to rest at the type of cell that the pattern has as its chosen center. A drawback of this modification is that there must be nine different ROMs (PE, SE1, SE2,..., SE8) as opposed to two (PE and SE). Figure 15 on page 36 illustrates the space requirements for pattern growth centered on a PE, and centered on a "middle" cell.

CHAPTER III THE ARRAY CELL

3.1 INTRODUCTION

Kumar [12] and Gollakota [7] investigated fault reconfiguration and functional reconfiguration semi-independently. If the results of their research are combined, then processing elements and switching elements must have the ability to implement both types of reconfiguration. This chapter presents the arguments for implementing the control circuit by a general-purpose microprocessor, as opposed to Kumar's specific control circuit. Other aspects of an array cell, including how a cell communicates with its neighbors, are also described.

The individual cells of the array are either processing elements (PE) or switching elements (SE). Both types of cell are modeled by a control hyperplane and a computational hyperplane. The control hyperplane is responsible for fault and functional reconfiguration, testing, and enabling a function to be implemented by the cell. The computational plane of the processing element performs the computational function that is assigned to that processing element. The computational plane of the switching element is used to di-

rect data between processing elements. The following subsections explain these elements in more detail.

3.1.1 The Control Plane

The control planes of the switching elements and the processing elements are identical, and must contain the elements necessary to perform the array algorithms. If the algorithms are restricted to those presented by Kumar, then his **register-logic** sufficiently models the control plane. However, if the algorithms are modified to include the presence of switching elements, then the control circuitry, and the number of allowable states increase. Each additional state corresponds to a switch position, and is reflected in the complexity of the look-up table that decides the next-state mapping for the cell.

3.1.2 The PE Computational Plane

The computational planes of the processing elements perform functions that can be relatively simple and may use special purpose combinational devices that perform functions such as array multiplication and array addition. These functional devices are needed to implement array algorithms such as LU decomposition and matrix multiplication. However, the computational functions of the cell may require a complex se-

quential circuit such as a microprocessor with its satellite chips [12]. In either case, the computational planes of the processing elements must be identical.

3.1.3 The SE Computational Plane

The computational plane of a switching element is composed of combinational switching blocks. These switches transfer data through the cells in any specified direction without clock delays. The direction of data transfer through the cell is determined by the local state that is stored in the State Register of the control plane. A switch design is presented in a later section.

3.2 THE MICROPROCESSOR CELL

The **register-logic** control cell, theorized by Kumar, uses ninety-four bits of memory for fault reconfiguration, as enumerated in Figure 8 on page 26. This hardware must be expanded for functional reconfiguration as defined by Gollakota [8], because he adds control processes as described in section 2.4. Likewise, any time new control processes are added to the cell, the control hardware of the array must be redesigned.

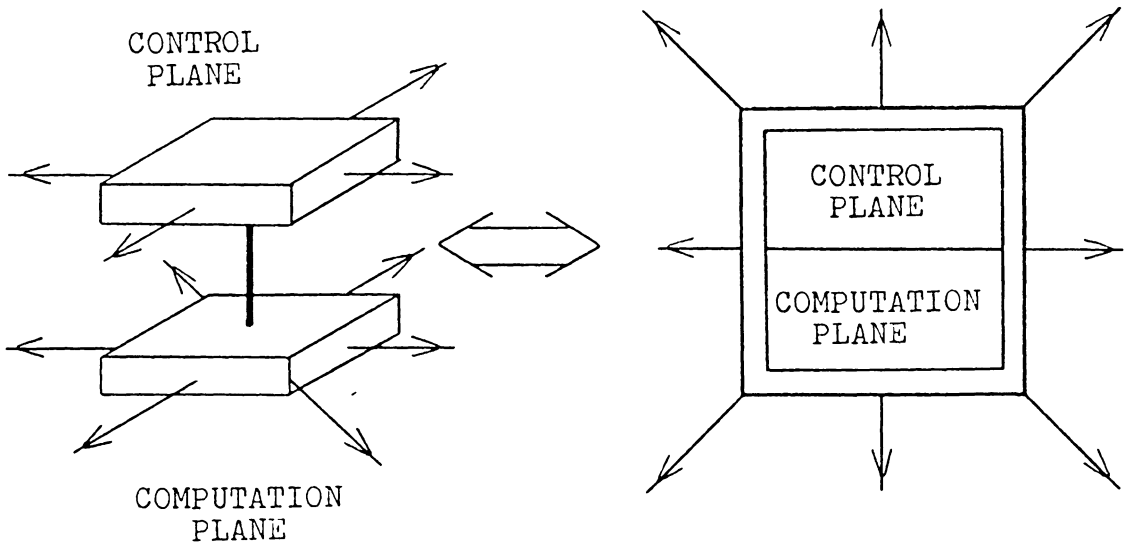


Figure 16. A Two-Dimensional Representation of a Cell

Problems of redesigning hardware because of added control processes are minimized, if the background "fabric" cells are described in terms of general-purpose microprocessors. Because control plane functions reside in the microprocessor control ROM, additions to the control requirements translate into adding instruction sequences to the control ROM. The microprocessor implemented control plane also has the advantage of being an "off-the-shelf" architecture, which minimizes hardware design time.

A microprocessor is used to implement the control plane of a cell and is used to control the components of the computational plane. Because the computational plane components are anything from simple logic blocks to processors, they either are peripheral devices of the microprocessor, or share the computing power of the microprocessor with the control plane. The control plane and the computational plane make up the cell in the form of a general-purpose microprocessor system.

The system's major elements are the microprocessor, the control ROM, the memory data RAM, the I/O ports, a test ROM, and a combinational logic block. Figure 17 on page 44 illustrates a general-purpose microprocessor system. The microprocessor illustrated is an eight-bit microprocessor, the number of bits being chosen arbitrarily. The general-

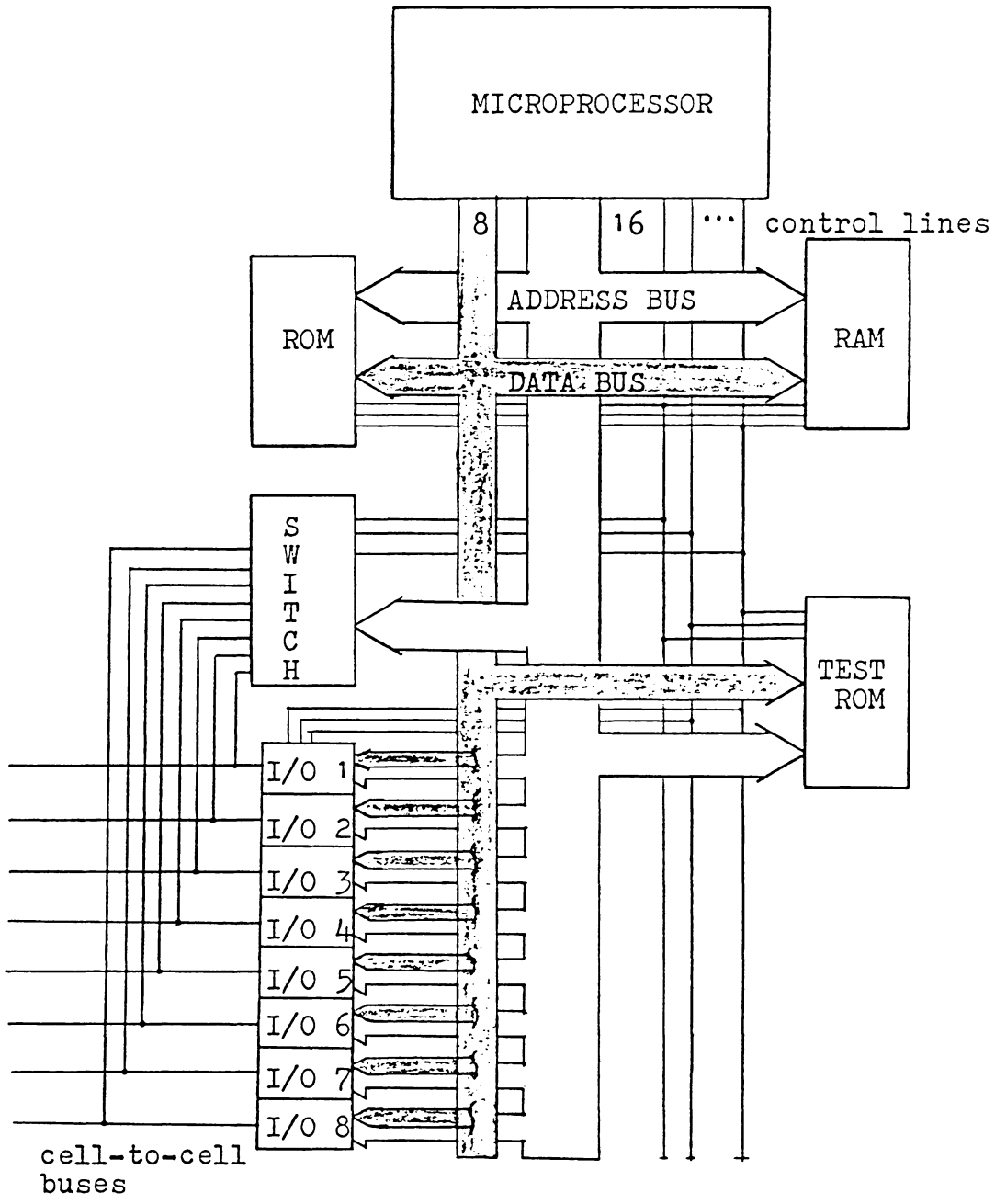


Figure 17. General Purpose Microprocessor System

purpose microprocessor system in this figure can be a PE or an SE, or a combination of the two. That is, either PEs and SEs are fabricated, or one chip that performs the functions of both elements is fabricated. The advantages and disadvantages in allowing each cell in the array to double as a PE and an SE are listed below.

Advantages of a Combination (PE and SE) Cell

1. The array has no special structure (of alternating PEs and SEs), thus, keeping with the original conceptualization of an array of identical cells.
2. One type of cell is manufactured. The replacement inventory for failed cells consists of one type of cell.
3. There is no alignment problem. The alignment problem occurs when, after reconfiguration, a PE (SE) is asked to do a SE (PE) function.
4. The fault reconfiguration algorithm does not have to be modified to allow for the presence of switching elements.
5. The testing procedure for every cell is the same.

Disadvantages of a Combination Cell

1. Computing power is wasted. Major components of the computational plane will be idle.

2. Each cell is complex, and requires a considerable amount of chip space.
3. The cost for an all purpose cell is greater than the cost for a PE or a SE cell.

PEs and SEs that differ in the computational plane, but have the same microprocessor control, are used to develop the testing algorithm. The PEs do not have the switch circuitry, and the SEs might not have the computation power of a PE. This option is a compromise. The cells are less complex, but the array employs two types of cells. The next section discusses the similarities and differences of the two elements.

3.3 PE AND SE SIMILARITIES AND DIFFERENCES

PE and SE similarities and differences are important in determining the different steps in the testing algorithm. For example, because the control plane of each cell is identical, the control plane test for each cell will be identical. However, the computational plane tests for the SE differ from those of the computational plane.

The control plane of a PE and an SE is implemented by either the register-logic circuit or by a general-purpose microprocessor system. In either case, the major difference between the PE and the SE is in the implementation of the

computational plane components. Additional differences are in the memory blocks of the PEs and SEs, which store different functional states. However, differing PE and SE states, and their respective instructions, are required for the control of the computational plane. Because the control plane of each cell is expected to perform the same functional and fault reconfiguration, the control plane circuitry, or instruction sets, are identical for each cell.

3.4 A SWITCH DESIGN

The switch in the computational plane must have minimal delay, and is constructed with combinational logic. The switch logic block "taps" off the cell-to-cell bus in front of the clock-controlled I/O ports as shown in Figure 17 on page 44. The switch is comprised of three major elements: 1) multiplexers; 2) decoders; and 3) control registers. These elements are described in the following subsections.

3.4.1 The Multiplexers

There is a MUX for every bit line into the cell. If each cell-to-cell bus has eight bit lines, then there are sixty-four multiplexers. The inputs to each MUX are the i(th) bit

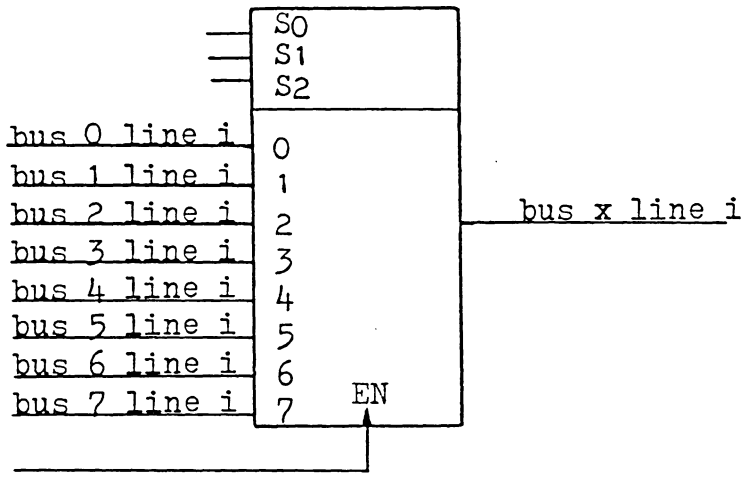


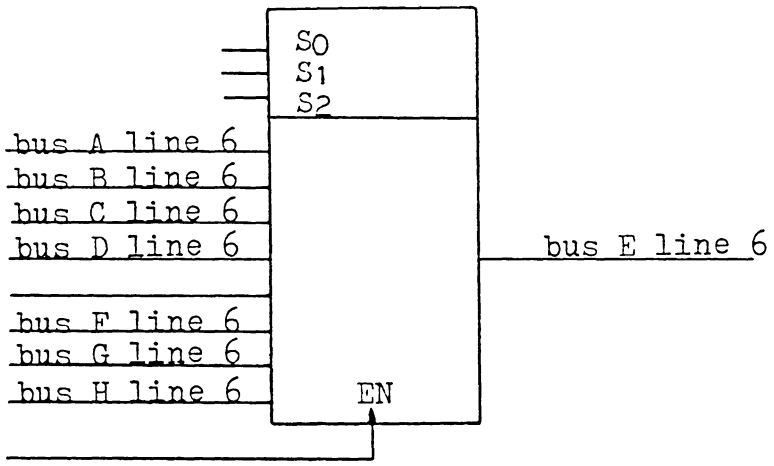
Figure 18. Inputs and Outputs to a switch MUX

lines of each cell-to-cell bus, and the output of each MUX is one of the i(th) bits, as illustrated in Figure 18 on page 48. Since a switch never connects a bus to itself, the i(th) bit line that corresponds to the MUX output is not needed on the input. Therefore, if the MUX has eight inputs, seven of the inputs are used.

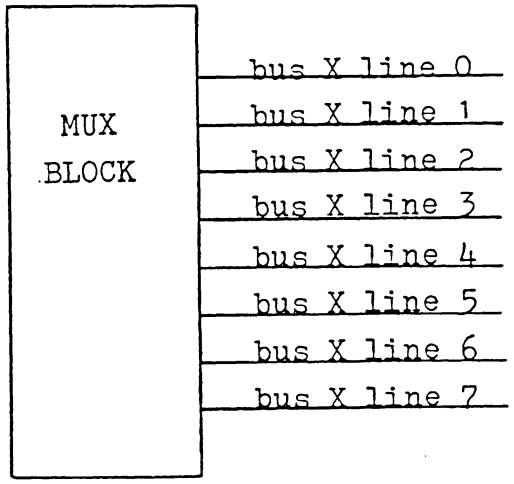
Example 3.4.1.1

There are eight cell-to-cell buses for a cell in the Moore Neighborhood. In this example, each bus has eight bit lines. Let the cell-to-cell buses for an individual switch cell be labeled A through H. Then, as illustrated in a) of Figure 19 on page 50, if the MUX output corresponds to the sixth bit line of the E cell-to-cell bus, the inputs to the MUX are the sixth bit lines of the remaining cell-to-cell buses.

A MUX block is defined as a group of multiplexers whose outputs are bit lines to the same cell-to-cell bus. For a switch cell with eight incoming cell-to-cell buses, there are eight MUX blocks. A MUX block is illustrated in b) of Figure 19 on page 50.



(a)



(b)

Figure 19. a) A MUX for the Sixth Bit Line of Bus E b) A MUX Block

3.4.2 The Decoders

The control ROM holds the "control code" for each switch state. The code corresponds to the binary strings that are needed to enable the correct MUX block, and select the correct MUX block inputs for each of the states. The enable binary string is sent through a decoder which enables the corresponding MUX.

First, consider the MUX control for a switch of crossover one, that is, only one pair of cell-to-cell bus lines are connected. There are eight required enable signals -- one for each MUX block, and eight required select signals -- one for each input line. The following example illustrates the control signals for a switch with a crossover of one.

Example 3.4.2.1

Let the MUX blocks be labeled from A through H, corresponding to the eight cell-to-cell bus directions. Also, let the input lines to each MUX be numbered from 0 to 7. Then a bit string of length three enables a single MUX block, and another bit string of length three selects the MUX input. For example, if cell-to-cell bus D is connected by the switch to cell-to-cell bus C then the bit string to enable the C MUX block is 010, and the string to select the D input bits, of the MUX

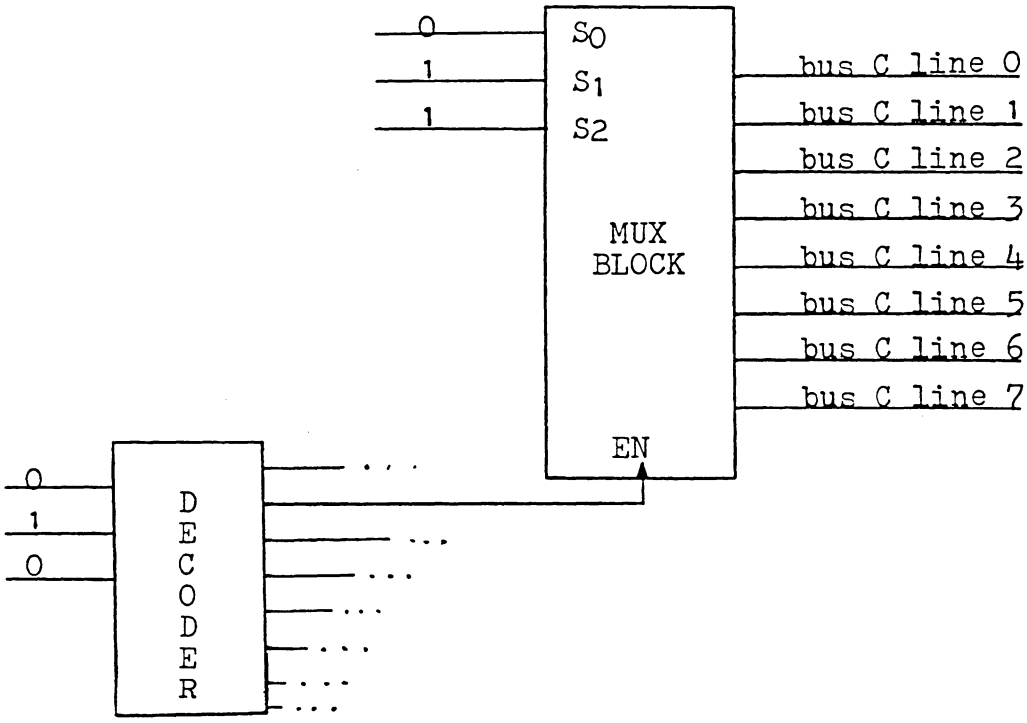


Figure 20. Enable and Select Control for a Switch of Crossover One.

block, is 011. This example is illustrated in of Figure 20 on page 52. The enable bit string is the input to a 3 to 8 decoder. Each output from the decoder enables one MUX block.

Now consider a crossover of two, for which the number of control lines must be doubled. Two 3 to 8 decoders, as opposed to one, are needed to implement this scheme. The enable to each MUX must now be an OR of the two decoder lines corresponding to that MUX. Also, there is a unique select string for each MUX block, so that each cell-to-cell bus connects to a different cell-to-cell bus. Referring to Figure 21 on page 54, each bit of the select string is ANDed with the output of the decoder that is "associated" with that select string. Since only one of the decoder outputs to a MUX is a logical one, then only one of the select strings is preserved through the gates. Thus, the select lines to each MUX is an OR of the two select binary strings.

3.4.3 The Control Registers

Two eight-bit registers in the switch block hold the control information for the decoders and the MUX select lines. After a cell receives its local state, the CPU loads the registers with the control information. Then, during the computation

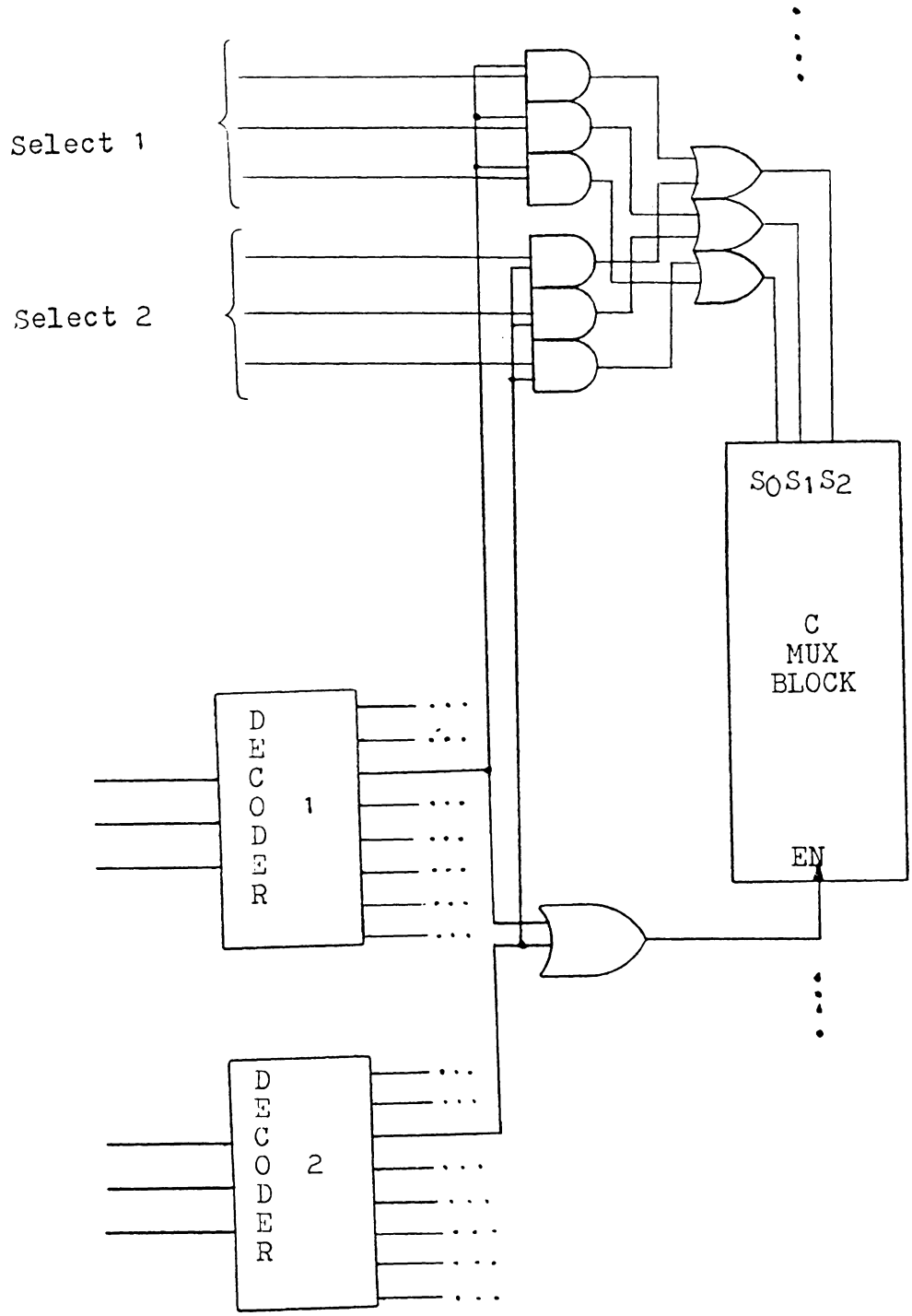


Figure 21. Enable and Select Control for a Switch of Crossover Two.

phase, the switch opens when the CPU enables the switch block (and the registers).

3.5 TIME MULTIPLEXING OF THE CELL-TO-CELL BUS LINES

In the cell model, the control hyperplane and the computational hyperplane each have their own set of cell-to-cell bus lines. For applications where separate control and data bus lines are not possible, or not desirable, the array cell must time multiplex the cell-to-cell bus lines. If, for example, the number of pins on a chip limits the number of available cell-to-cell bus lines, then multiplexing data and control is essential.

For the case where the cells use time multiplexing, the three-dimensional concept of the control and computational hyperplanes is realized by a planar cell as shown in Figure 16 on page 42. In the three-dimensional model, the control hyperplane has four cell-to-cell data buses and the computational hyperplane has eight cell-to-cell buses. Actual hardware implementation of the two-dimensional cell causes the four buses of the control hyperplane to be redundant. Only eight cell-to-cell data buses are used as opposed to the twelve that are required in a three-dimensional implementation. During the computational phase of the array, the buses communicate computational information. During the

diagnostic phase, the buses communicate test and control information. The distributed clock [8] and the counters in the cell determine the phase, computational or diagnostic, that the cell is in.

3.6 CELL COMMUNICATION PATTERNS

During the diagnostic clock phase, the control planes of the switching elements and the control planes of the processing elements communicate test and control information. If a communication clock pulse is defined as a clock pulse that triggers communication on all of the cell-to-cell buses, then at least two communication clock pulses are required for every cell to communicate test results and control information with all four of its Von Neumann neighbors. There are two patterns of communication that use two communication clock pulses -- the Checkerboard Communication Pattern, and the Wave Communication Pattern. These patterns are described in the following subsections.

3.6.1 The Checkerboard Communication Pattern

The checkerboard pattern assigns alternating cells of the array as black or red cells [9], as illustrated in Figure 22 on page 57. During the first communication clock pulse at time n, the black cells send out information on all

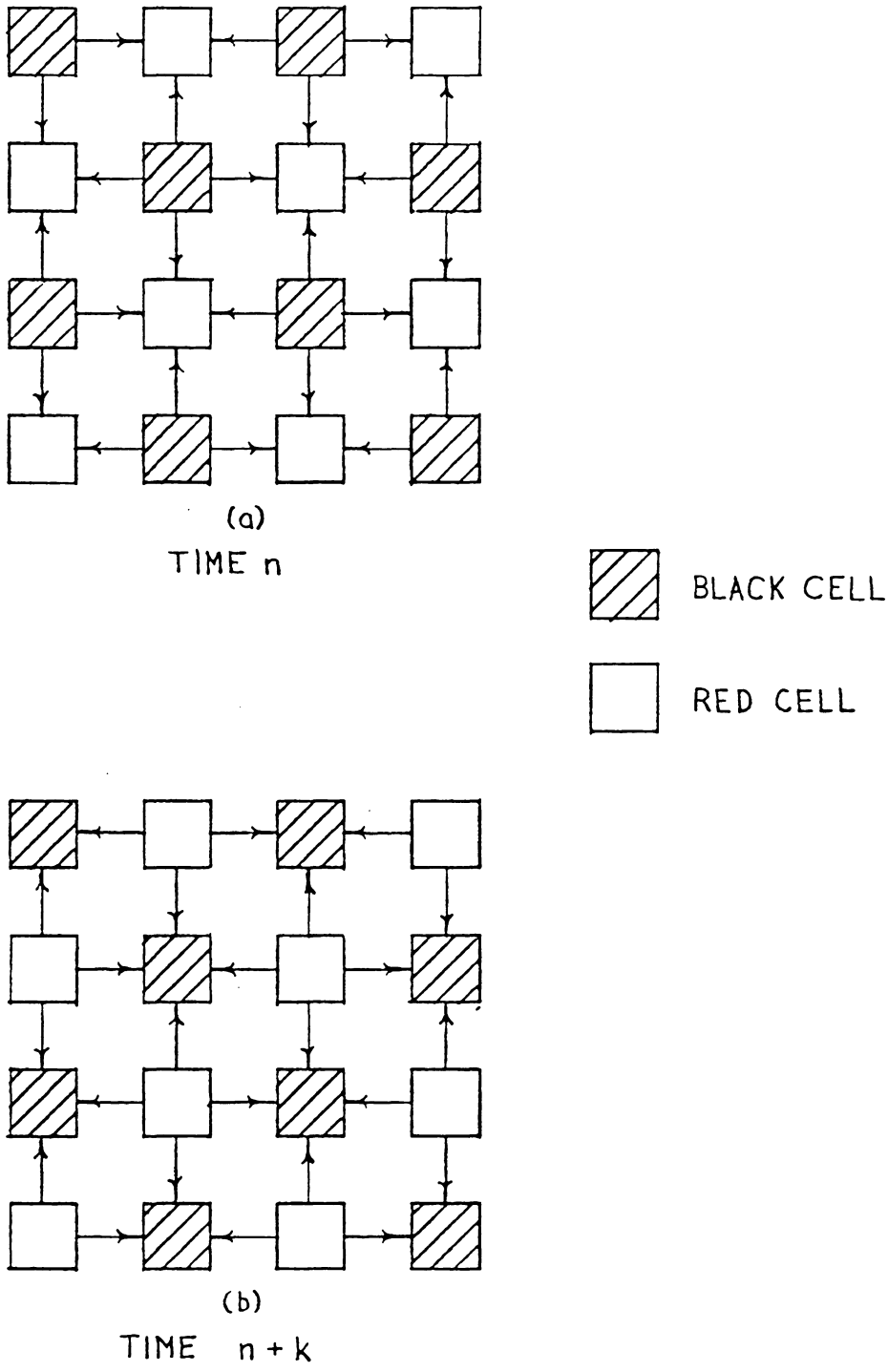
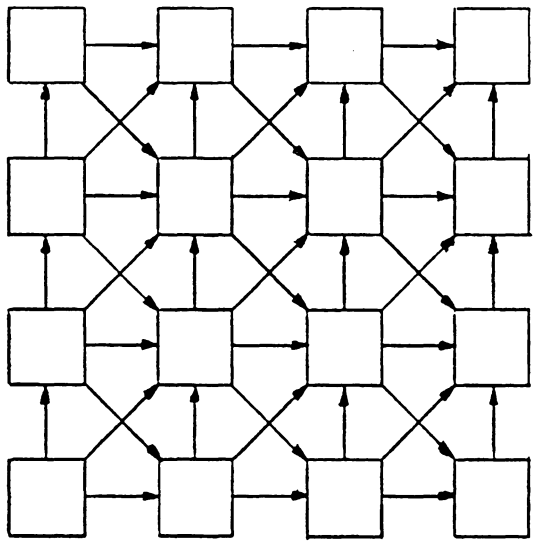


Figure 22. The Checkerboard Communication Pattern

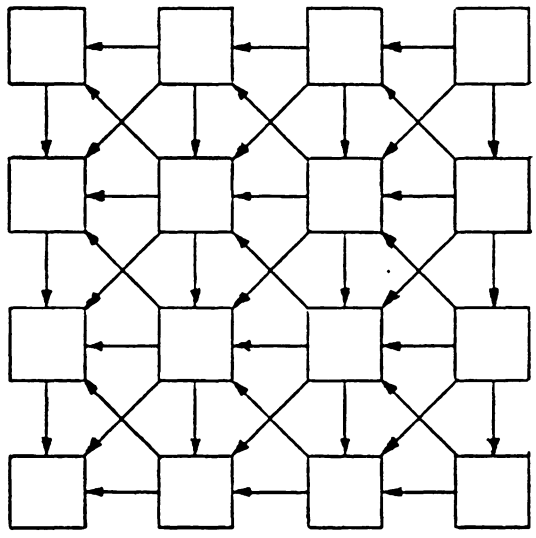
four of their Von Neumann I/O ports. Consequently, the red cells receive information into all four of their Von Neumann ports. The red cells then process this information while the black cells wait for the response. On the second communication clock pulse at time n+k, the red cells send information to the black cells. This method of communication is used in the reconfiguration algorithm [12] and testing algorithm [9] presented in earlier work on this reconfigurable array. This pattern is inefficient because the cells are not operating in identical control modes since an individual cell's operation depends on its color. Therefore, the cell control planes are not identical, adding unnecessary complexity to the requirements of the array.

A time lag of k is inserted between the two communication clock pulses. This lag is the time required by the cell to store or process the information that it receives, and to load its ports with information that it will send. Because the I/O ports are connected to the microprocessor of the PE or SE by a single bus, only one port at a time can be addressed. The time lag of length k is long enough so that the cells of the array have a synchronously clocked input and output.



(a)

Time n



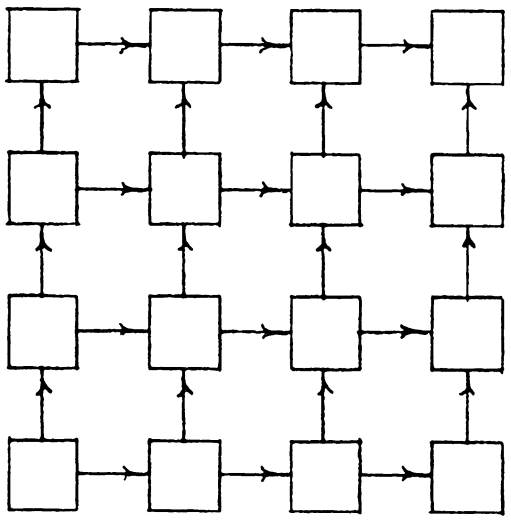
(b)

Time n + k

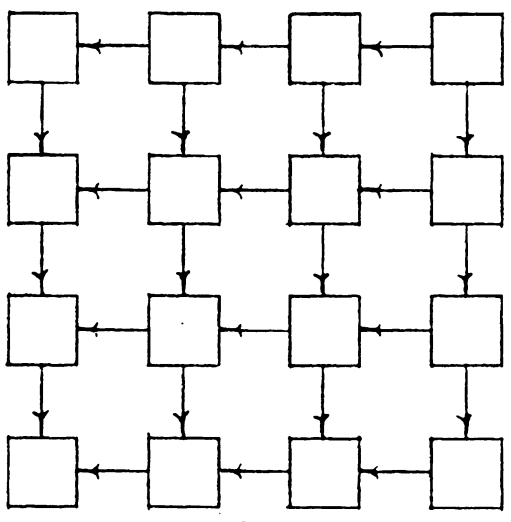
Figure 23. Moore Wave Communication Pattern

3.6.2 The Wave Communication Pattern

A second communication pattern is called the Wave Communication Pattern because the motion of the information transfer resembles that of a propagating wave. Every cell sends and receives information in the same direction at the same time. Because every cell communicates identically, this pattern eliminates the need for cells to be marked differently. Two communication patterns will be considered using the technique of wave communication. One pattern uses the Von Neumann Neighborhood, and the other uses the Moore Neighborhood. In the case of the Moore Communication Pattern, at time n, a cell, C, clocks test results out of its N, NE, E, and SE I/O ports. Because its neighboring cells are also clocking information out of their own N, NE, E, and SE ports, cell C simultaneously clocks information into its S, SW, W, and NW ports. Likewise, at time n+k, C clocks information out of its S, SW, W, and NW ports while receiving information on its N, NE, E, and SE ports. This communication pattern allows any cell to send and receive information from all eight of its neighbors using a minimal number of communication clock pulses while still remaining in a control mode identical to that of any other cell. Figure 23 on page 59 shows the communication pattern for times n and n+k, respectively.



(a)
TIME n



(b)
TIME n+k

Figure 24. The Von Neumann Wave Communication Pattern

The communication pattern described was for the Moore Neighborhood. However, the same sequence of communication applies for the Von Neumann Neighborhood, except that only the N, S, E, and W ports are used. Figure 24 on page 61 illustrates the Von Neumann communication pattern. The Moore wave communication pattern and the Von Neumann wave communication pattern are used in the testing algorithm.

CHAPTER IV ARRAY I/O ALGORITHM

4.1 INTRODUCTION

The active-array, which is defined as the group of cells in the cellular architecture that have the global state and are performing a function, must communicate results and receive input data from the outside world. The outside world, denoted OW, refers to any system that is external to the cellular array. In an array that does not reconfigure, the paths by which the cells receive input and send output are permanent fixtures of the array, and if they fail, the array becomes faulty. However, in the reconfigurable cellular array, the position of the I/O data paths change as the position of the active-array changes, and failed data paths can be reconfigured. This chapter presents an array I/O algorithm that constructs I/O data paths as a part of the reconfiguration process, and is implemented by an instruction sequence in the control ROM.

Three major elements of the array that the I/O algorithm uses to describe path construction are the array I/O ports, the data path, and the active-array I/O cells. These elements are described in the following sections. The second

part of this chapter describes how to choose the algorithm parameters, and presents the algorithm steps.

4.1.1 Array I/O Ports

The PE cells along the outer two sides of the cellular array are defined as the I/O ports for the cellular array. These processing elements have the added function of bridging the OW to the array, and are connected to the active-array by the I/O paths. Figure 25 on page 65 illustrates the I/O ports, the I/O paths, and the active-array. The ports at the top of the array function as the input ports, and the ports at the left of the array function as the output ports. Although this choice is arbitrary, the above port locations are used to develop the I/O algorithm.

The ports are subject to the same testing procedure as the other cells. If an I/O port fails, it is quarantined, and is no longer a possible choice for a data path link. Therefore, a failed I/O port, by itself, does not threaten the correct functioning of the array.

The input ports and the output ports are similar, but have different responsibilities. The responsibility of the input ports is to make sure that the correct active-array cell receives the input information that is meant for that

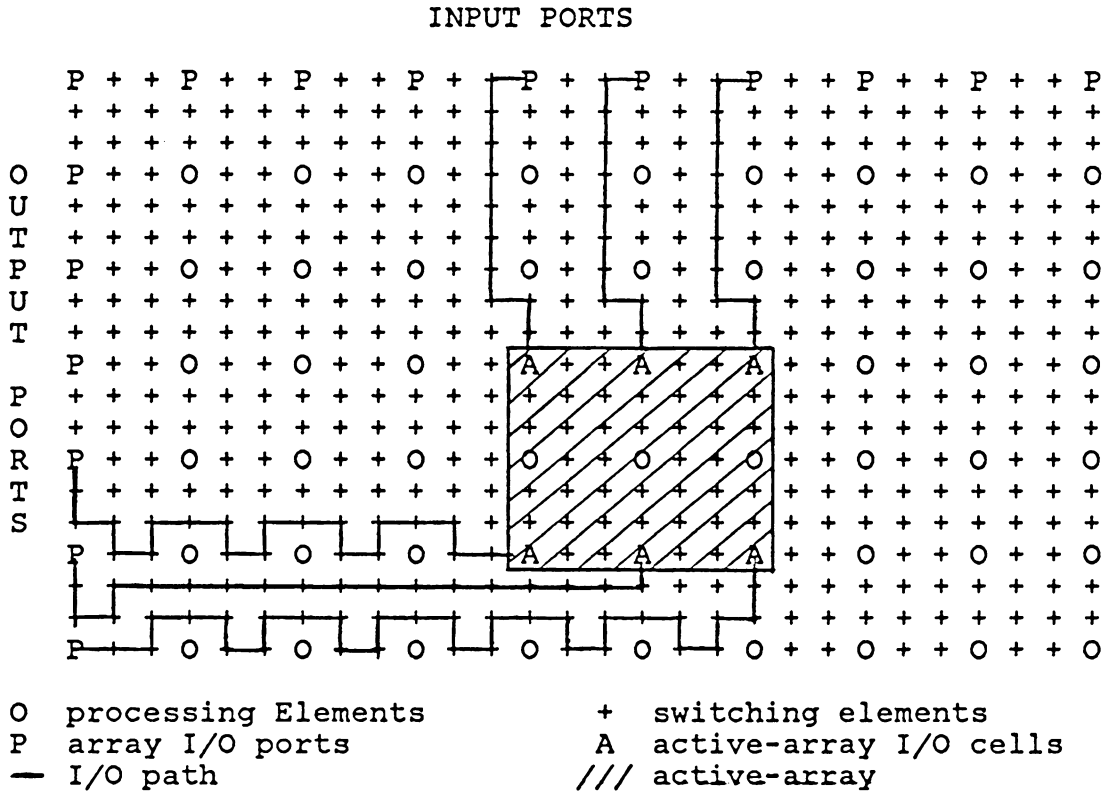


Figure 25. I/O Ports, I/O Paths, and Active Array

cell. The responsibility of the output port is to assure that the outside world knows from what cell it is receiving data. Both types of port use tag prefixes to accomplish their tasks.

Tag prefixes are defined as extra bits concatenated to the input and the output data associating each piece of I/O data with a particular cell of the active-array. The tag prefixes take the form of extra bit lines from the OW to the array I/O ports, and are associated with the array I/O ports only. The tag bit lines of the input ports function as data input enable lines. For example, if input port, **p**, is connected to the active-array cell that requires the 'first' set of data, then input port, **p**, accepts data from the outside world that is tagged as 'first.' Likewise, the output ports tag the data received from the active-array before sending the data to the OW.

The number of tag bits is dependent on the number of active-array cells that are required to receive or send data. For example, let the global functions, F(1) through F(x), have I(1) through I(m) input ports and O(1) through O(n) output ports. Then the number of input tag bits and output tag bits is equal to the number of bits in the binary representations of max[I(1),...I(m)] and max[O(1),...O(n)], re-

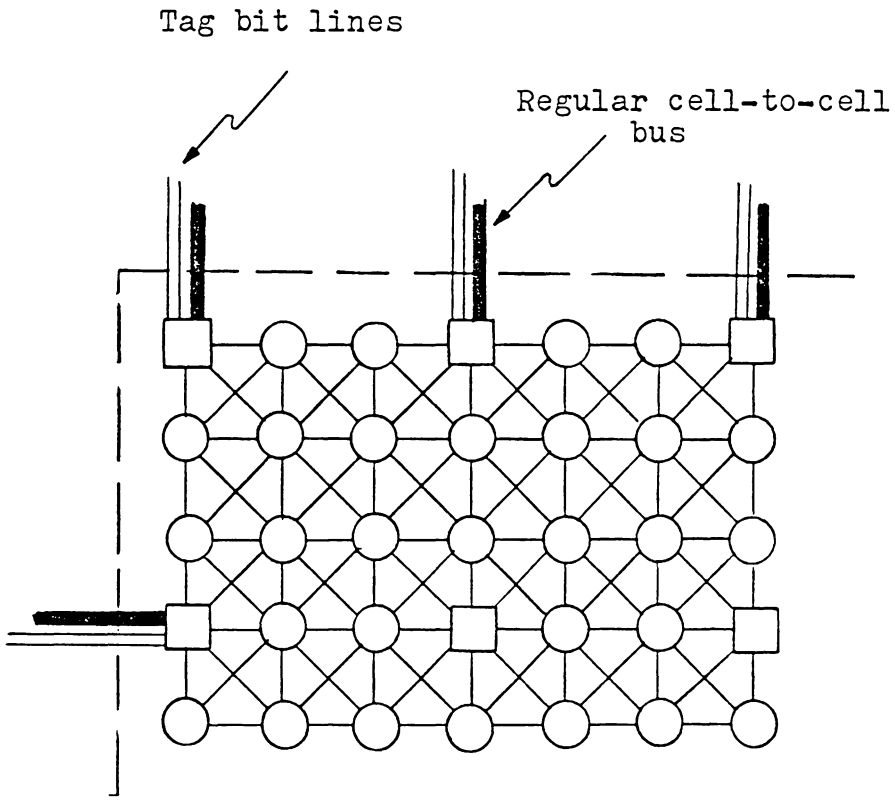


Figure 26. Tag Bit Lines

spectively. Figure 26 shows the tag lines for the I/O ports for max(I) = max (O) = 4.

4.1.2 Data Paths

The switching elements of the I/O data paths that lead from the active array to the array I/O ports are of varying length due to the positioning of the active-array and possible regions of faulty cells. During reconfiguration, cells communicate only with their Von Neumann neighbors. Therefore, the path must be constructed by cells communicating with their Von Neumann neighbors. As a result, a path finds its way to an I/O port in straight lines, with turns of ninety degrees. The testing algorithm also uses the Von Neumann Index, and, therefore, a faulty cell in the I/O path is quarantined by cells that are in the I/O path. This guarantees that information from a faulty I/O cell is not sent, by way of the data path, to the active-array or to the array I/O ports.

The cells of the I/O paths are tested for faults because the switching elements of the data paths are vital to the functioning of the active-array. The data path switching cells can either contain the global state, as do the active-array cells, or contain a path global state. If the cells contain the global state, then any fault that occurs in the I/O path will initiate fault reconfiguration, as described

by Kumar [1]. Reconfiguration because of I/O path failures is 'overkill' for the following reasons:

1) the active array has no fault and the seed sent out to reconfigure the array might come to rest and regrow the active-array at the same place; and

2) a complete new set of data paths must be created, instead of just replacing the faulty path.

The path global state contains the same information as the global state, but has an extra bit of information that indicates that the cell is a path cell. If a bit is added to the Global State Register (GSR), such that a zero bit indicates that the cell is part of the active-array, and a one indicates that the cell is in a path, then the path global state and the global state are synonymous.

Example 4.1.2.1

Suppose that the array can perform seven functions, as illustrated in the following diagram. Then the GSR has four bits, where the first bit indicates the path/active-array, and the last three bits indicate the function.

```
------------------- Path Cell=1, Active-array Cell=0
|   --------------- Global State
|   |

- 000               Unused
- 001               Function 1
- 010               Function 2
  '                      '
  '                      '
  '                      '
- 111               Function 7
```

If the data path cells contain a path global state, then a fault in the I/O path initiates only path reconfiguration. When a cell with the path state quarantines a neighboring cell in the path, it takes one of two actions. If the cell is between the active array and its faulty neighbor, the cell initiates an alternate path-finding process by sending a "find an alternate path" message to the previous cell in the path. If the cell is on the I/O port side of the faulty cell, then it initiates the clearing of the remaining path. To start clearing the remaining path segment, the cell passes a message to the next cell in the path that there has been a path failure. Subsequently, the I/O port receives the message and stops sending or receiving data.

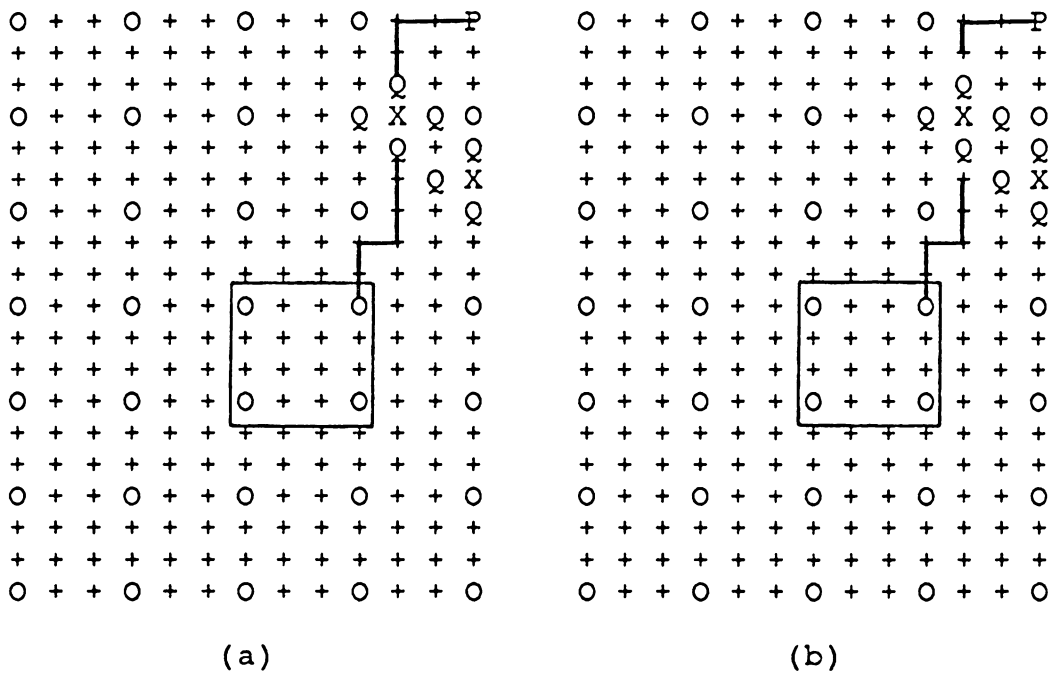


Figure 27. Path Reconfiguration a) Time 1 b) Time 2

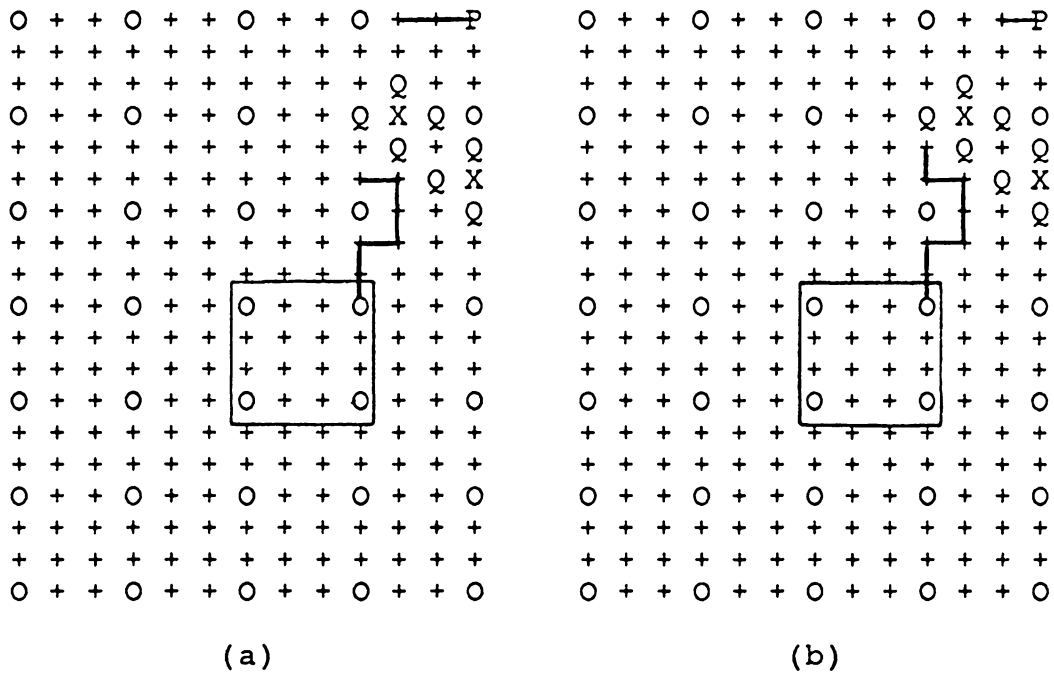


Figure 28. Path Reconfiguration a) Time 3 b) Time 4

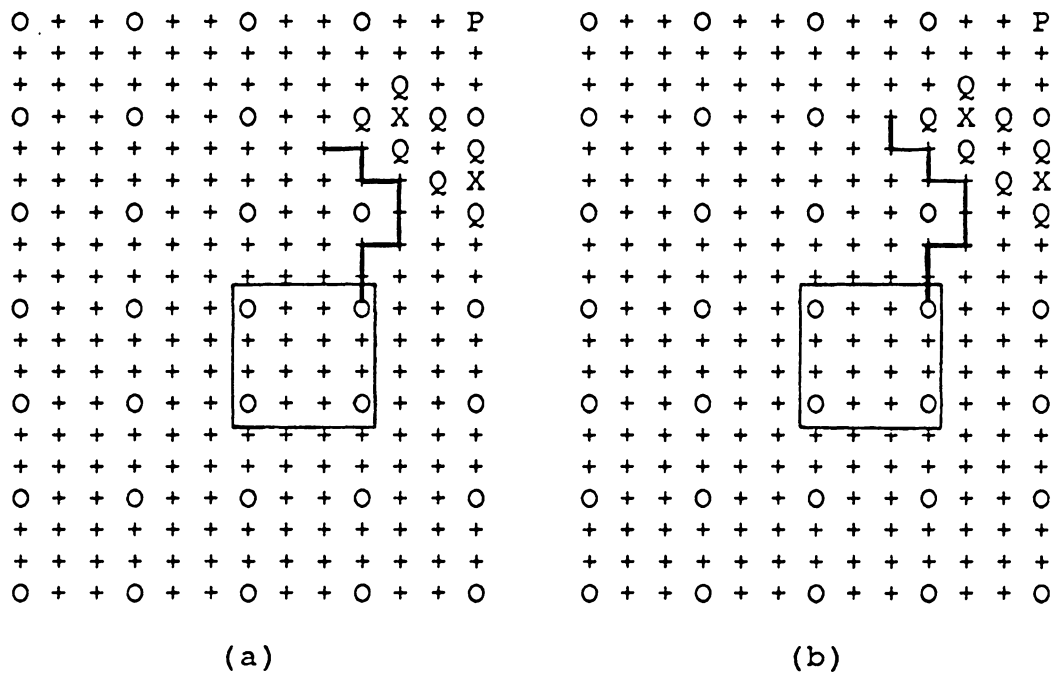


Figure 29. Path Reconfiguration a) Time 5 b) Time 6

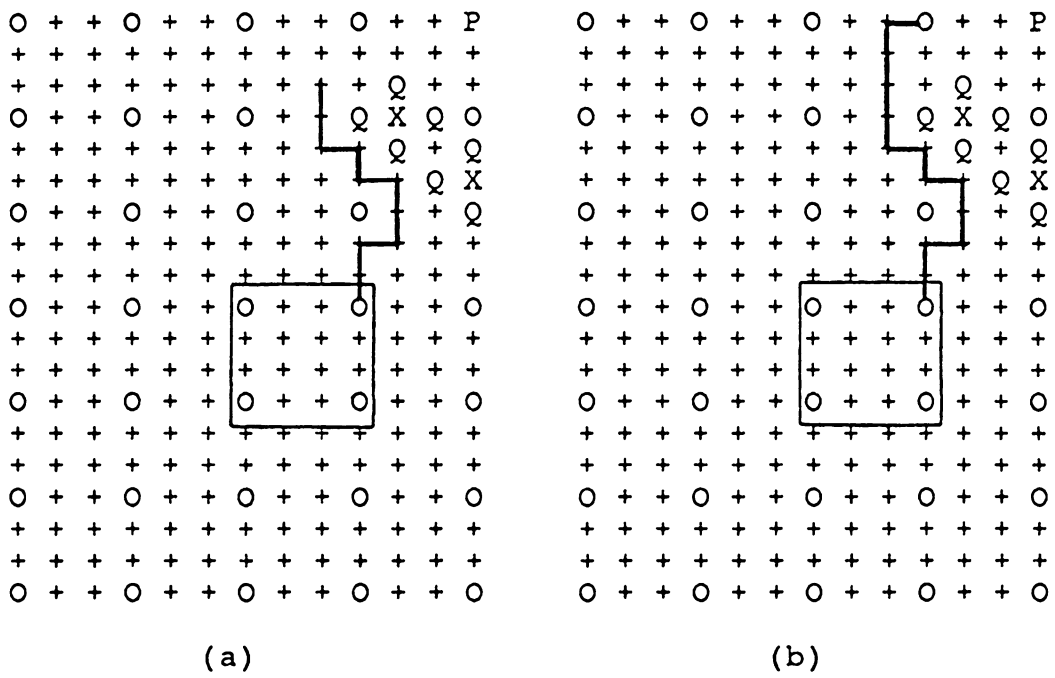


Figure 30. Path Reconfiguration a) Time 7 b) Time 10

A quarantine state cell is a 'path clear' or an 'alternate path' cell depending on its position to the faulty cell with respect to the direction of path construction. Paths are constructed from the active-array to the I/O ports, and a path cell has a neighbor that precedes it in the path, and a neighbor that follows it. A cell initiates the construction of an alternate path if a faulty cell follows it. A cell in the path initiates path clearing if a faulty cell precedes it. Figure 27 on page 71 through Figure 30 on page 74 illustrate this "path reconfiguration" process.

The second alternative requires that a new state and an additional 'path reconfiguration' algorithm be added to the set of reconfiguration algorithms. However, this alternative requires less reconfiguration time. If this algorithm is to be implemented in hardware and added to Kumar's circuitry, then the complexity of the control cell is increased by the addition of control registers and enable circuitry. If the control cell is implemented by a microprocessor, then the added algorithms take the form of extra instruction sets in the control ROM of the microprocessor system.

4.1.3 The Active-array Input/Output Cell

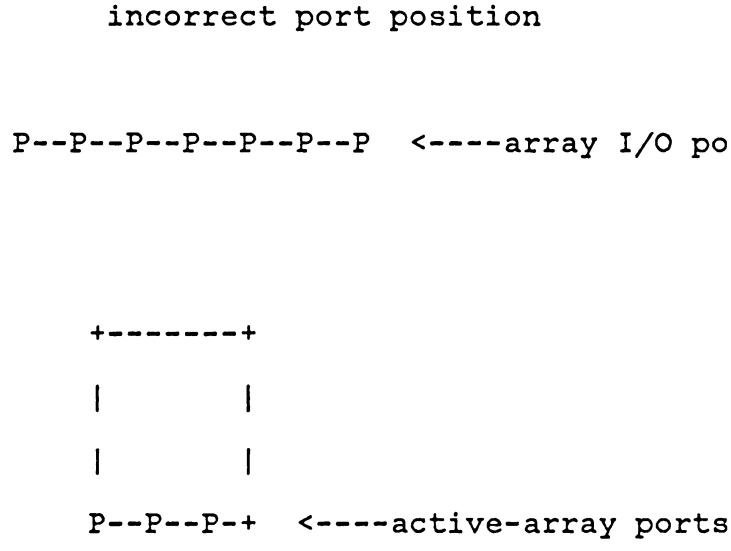
Data that is sent to the active-array enters through the active-array input cells. Likewise, data that is sent out

from the active-array is sent from the active-array output cells. These cells, as opposed to the array I/O ports, have the responsibility of initiating a path construction process. The active-array I/O ports store the global state of the active-array, and thus know the general direction to the I/O ports with respect to themselves. On the other hand, the I/O ports do not know the location of the active-array or the number of I/O ports requiring a path connection. Therefore, their search for the active-array would be haphazard, at best.

In order to initiate the path construction, the active-array Input/Output cell must 'know' that it is an Input/Output cell. Using the method described by Brighton [5], a cell knows its position in the active array. By knowing its position in the active array, a cell can determine, by means of a look-up table, its local state and whether or not it is an Input/Output cell. Because more than one cell in the active array can have the same local state, the local state, alone, is not sufficient information for a cell to determine whether or not it is an active-array I/O cell.

Care must be taken when designing the final orientation of the active-array. The active-array input (output) ports should not be situated on an edge that is not perpendicular

to or facing the array input (output) ports, as diagrammed below.



The paths that result from this positioning will envelope the active-array on two sides if enough ports are available, or possibly on all sides if ports are not available. This concept is illustrated in Figure 31. One active-array input port, and the active-array output ports are cut-off from the array ports. The I/O algorithm steps assume that the situation does not exist in which array ports can not "see" the active-array ports, as illustrated in the figure above.

4.2 THE ALGORITHM

Paths are constructed by a process of cells being appended to a partial path until the path reaches and adds an I/O port.

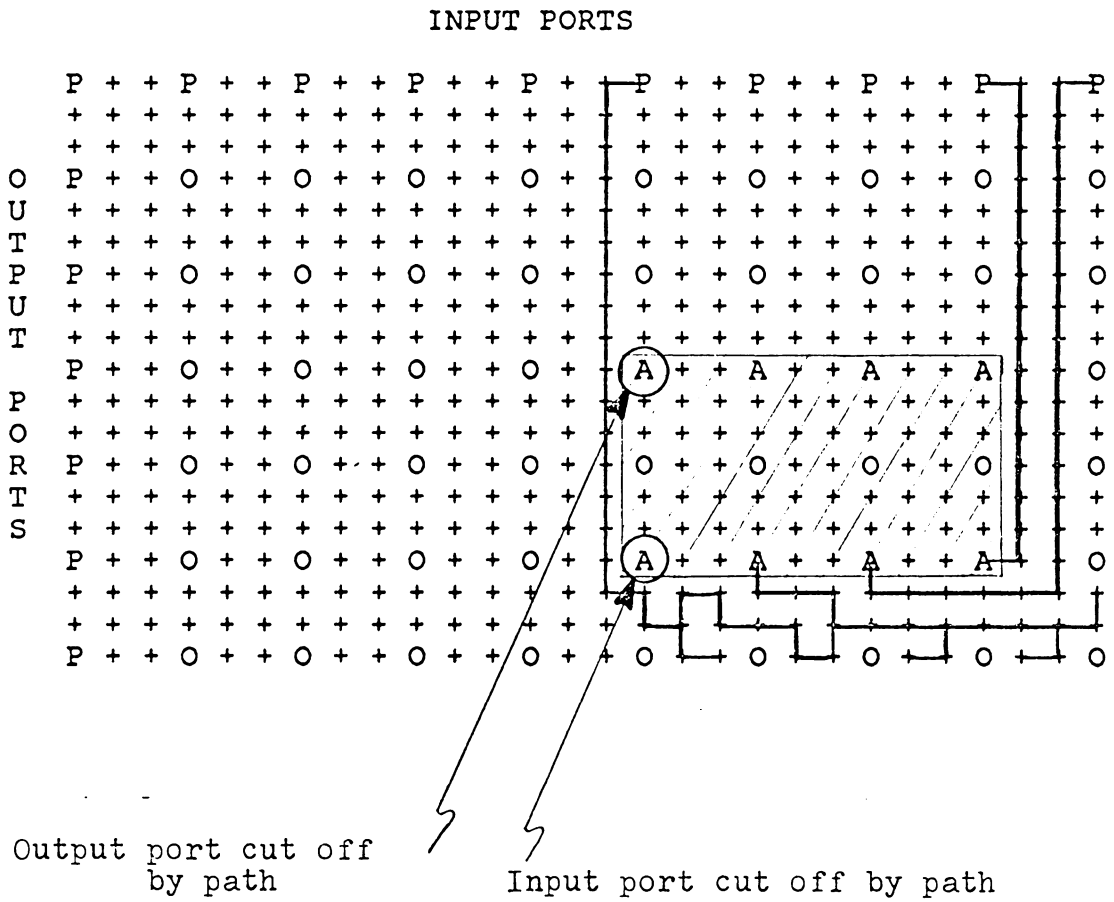


Figure 31. Enveloping Due to Poorly Situated Active-array Ports

A cell called the pointer cell, with a direction priority, constitutes the appended lead cell. This section explains the basis for path construction, including an explanation of the pointer cell and the direction priority. An outline of the algorithm follows.

4.2.1 The Pointer Cell

The pointer cell, initially the active-array I/O cell, is defined as the cell that is engaged in finding the next I/O path link. The pointer cell sends a path handshake signal to three of its Von Neumann neighbors. (The fourth neighbor is already part of the I/O path or part of the active array.) This prompts the receiving cell to send back a YES/NO signal to the pointer cell indicating whether or not it can be in the I/O path. If a cell can be the next link in the I/O path, then the pointer cell sends the path type, the path label, the path global state, and the pointer cell status to the cell. The path type is either input or output, and the path label refers to the label that is assigned to the active array I/O port cell. The path global state defines the type of reconfiguration and a direction of path construction. Different global functions may have their active-array I/O ports situated differently, and, as a result, the path global state defines the direction that a path takes according to where the I/O cells are located with respect to the array I/O

ports. The path handshake process continues until the pointer cell is an array I/O port whereupon the path is complete.

There are two types of cells that are eligible to be the next link in the I/O path. Both types of cell are Von Neumann Neighbors of the pointer cell, and must meet the following two eligibility requirements. A cell must be

- a quiescent array I/O port PE, or
- a quiescent SE.

An I/O port has priority over an SE. That is, a PE port sends a YES signal that indicates that it is a port, and the pointer cell chooses it over an SE.

A cell that has received a path handshake signal bases its YES/NO response on the eligibility requirements. The cell can determine its state by "looking" in its local state register. If a cell's local state is anything but the quiescent state, then the cell is not eligible to be the next I/O path link. A cell can store its cell type -- PE or SE or PE port -- in ROM.

After a cell passes its pointer cell responsibility, it defines its local state which corresponds to the position of

the switch. The switch connects the preceding path link to the path link that follows the cell. Therefore, a cell must remember, by storing the information in a register, from which cell it received the pointer cell status, and to what cell it sent the pointer cell status so that it can make the correct connection. The cell must continue to monitor its inputs for other path construction control words.

4.2.2 Direction Priority

If all three of the pointer cell's SE neighbors respond with a YES, signaling that each is eligible to be the next path link, then the pointer cell must choose one of the three. The choice is not arbitrary because the path must head in the direction of the array I/O ports. Therefore, the cell will choose according to the direction priority. The **direction priority** is an ordered list of directions, corresponding to the location of the Von Neumann Neighbors, that the pointer cell uses to chose the next path link. The list of directions include North (N), South (S), East (E), and West (W). The list is ordered with the first direction having the first priority. The pointer cell passes the pointer cell status to the neighbor that has the highest direction priority, and that satisfies the requirements. A YES from a PE port overrides the direction priority. The following example illustrates the direction priority.

Example 4.2.2.1

Let the pointer cell have the following Von Neumann neighbors

Active

Quiescent Pointer Path Cell

Quiescent

If the priority list is (N,E,S,W), then the south Von Neumann neighbor is the cell that is chosen as the next link because the north cell is active and the east cell is part of the path. The resulting neighborhood is

Active

Quiescent Path Cell Path Cell

Pointer

In the above example, the direction priority list is N,E,S,W. The third direction is opposite to the first, and the fourth direction is opposite to the second. The list is ordered in this manner because it allows a path to traverse the border of a region of faulty cells until it can once again proceed in the direction of highest priority. For example, suppose that a north bound path encounters a region of faulty

cells. The path turns east and continues in this direction until it finds the first available cell to its north. Therefore, up until this point, the path has followed the border of the quarantined region, otherwise it would have turned north sooner. Since north is the highest direction priority, the path is once again being constructed in the desired direction.

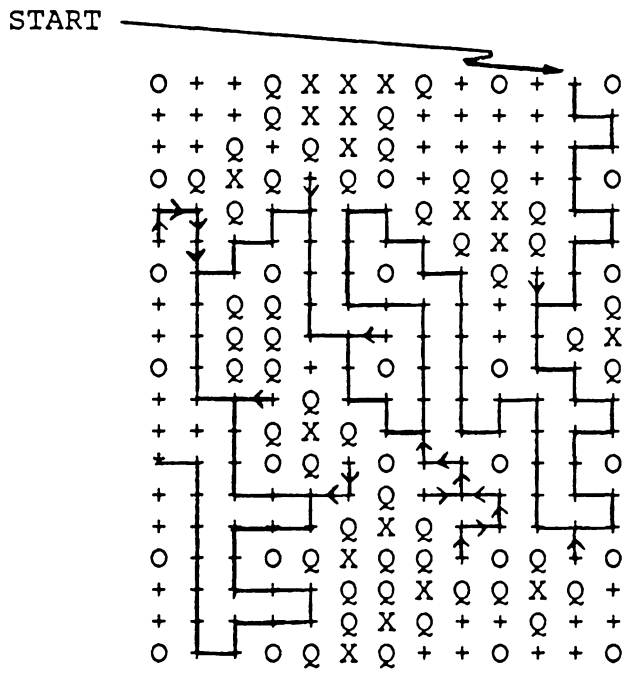
As a counter example, consider the direction priority N,S,E,W. If the path encounters a region of quarantined cells, then the path must continue its construction in the southward direction until it is once again obstructed and forced to proceed either east or west. The path, until it is diverted east or west, can not turn north because the north neighbor of the pointer cell is path active. Therefore, the path is constructed in a direction opposite to the highest priority direction.

Backtracking

When all of its neighboring cells are active, the pointer cell has no cell to which it can pass the pointer cell status, and the path construction process is "stuck." In order to free the path, the pointer cell initiates **backtracking**. Backtracking is defined as the process by which the path retraces its link steps until an alternate path can be found.

When all possible choices of a next link have been exhausted, the pointer cell sends a **backtrack** path control signal to the cell from which it received the pointer status, and assumes a backtrack state for the remainder of the path construction phase. The backtrack state is an active state that indicates that the cell is not an acceptable next link. The path cell that receives the backtrack path control signal reassumes the pointer cell status, and determines whether or not there is another choice for the next path link. If there is another choice, then the pointer cell passes its pointer cell status, and path construction resumes. If there is not another choice, the cell must send a backtrack control signal to the cell from which it received the pointer cell status. The above process continues until the path connects to a port, or until all possibilities are exhausted.

The direction priority, with backtracking, allows any path to "touch" every available SE. This property is significant because a path can find any cell in the array, including any obscurely located I/O port. As an example, let "*" be an SE that represents the only available port (not walled off by quarantined cells), such that "*" has not been touched by a path. Since the cell is available, then by definition, there is a way to construct a path from it to the path starting point, e.g. an active-array I/O port. Therefore, if a path has not found the available SE port, then it



O processing elements
X faulty cells
\+ switching elements
* only available I/O port

Q cells in quarantine
-- path
->- path with backtrack

DIRECTION PRIORITY IS NESW

Figure 32. Backtracking

has not exhausted all of the backtrack or next path link options. Because the port is one of the possible path links, and because it is the only available port, the path must find the SE before it has exhausted all possibilities.

Figure 32 on page 85 illustrates backtracking with a direction priority of N,E,S,W. The arrows in the figure indicate a backtrack, and the cell marked as "*" represents the only available I/O port. Path construction continues until the lead cell is the available port, "*." If the "*" port had been located in the top left-hand corner that is isolated from the rest of the array by Q state and PE cells, then it would not have been available. The path would have continued to search for the port, and eventually, would have backtracked to the start where there would be no next path cell options.

The time required for back-tracking can override any attempt to make the path construction process time efficient. Therefore, some mechanism, possibly the I/O ports in communication with one another, must monitor the reconfiguration time. Then, if the paths are not complete within a certain time limit, the mechanism initiates functional reconfiguration.

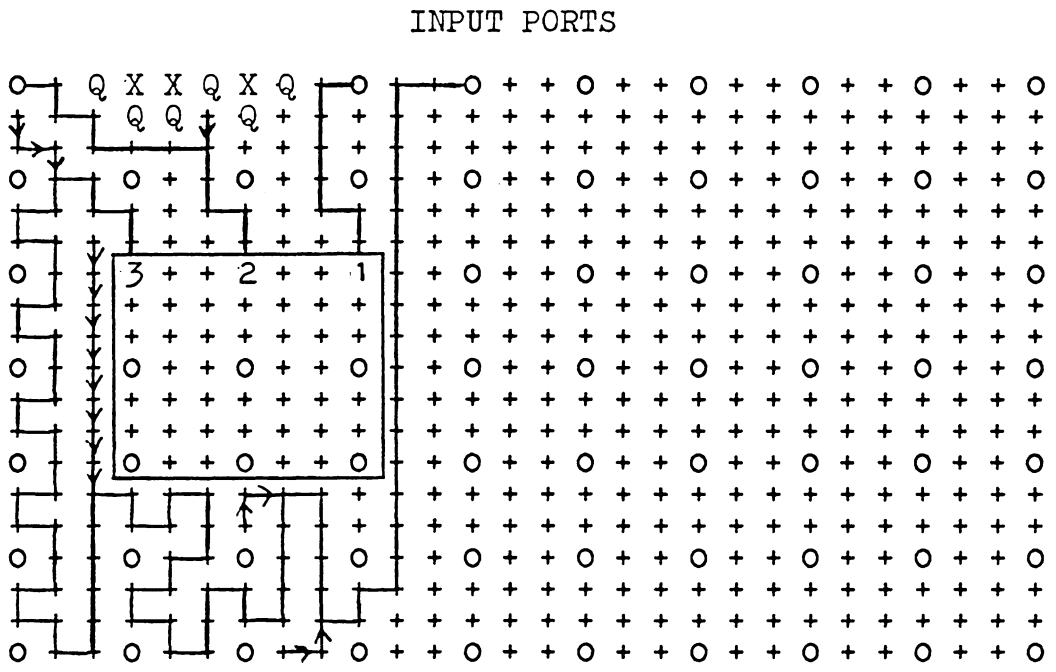


Figure 33. Direction Priority of N,W,S,E

The direction priority list must be ordered in such a way that the paths to the array I/O ports are formed using the most direct route, while allowing for possible path obstructions and failed I/O ports. For example, suppose that the array input ports are located at the "top" of the array, and the active array I/O ports are located at the "top" of the active array. Then, the obvious direction priority would appear to be North, followed, possibly, by W,S,E. However, this direction priority list does not allow for failed or blocked I/O ports, as illustrated in Figure 33 on page 87. Because of failed I/O ports, the path from active-array cell three is extremely long and encircles the active-array. In the process of encircling the array, the path forms a path barrier for any output paths that are yet to be constructed. Furthermore, had output paths been in the process of construction, path three either collides with, or is cut-off by, one of the output paths. A direction priority of N,E,S,W has a similar result. Therefore, choosing the direction priority list arbitrarily does not sufficiently guarantee that the paths can be completed.

A combination of two solutions solves the above problems. These solutions are referred to as the **maximal availability priority** and the **go first priority**. The two are dependent on one another, and, therefore, the discussion of one refers to the other.

The Maximal Availability Priority

At the time that an active-array I/O cell initiates path construction, it should have access to all of the available fault-free ports of its type, i.e. input or output. For example, if there are n fault-free ports available, then the first path should have access to all n ports, the second path should have access to n-1 ports, and the k(th) path should have access to n-k ports. Thus, there are a maximal number of I/O ports available to each active-array I/O cell, and the array can function as long as the active-array requires no more than n ports.

The maximal availability priority implies that no path be allowed to form a barrier to any of the available I/O ports. This task can be accomplished by the direction priority list together with the **go first priority**. Consider the row of input ports. The rightmost available port should connect to the rightmost active-array I/O cell. Therefore, the first direction priority corresponds to the direction that forms a path parallel to the row of I/O ports. Then, the second direction priority corresponds to the direction towards the ports. The third direction is opposite to the first direction, and the fourth is opposite to the third. Figure 34 through Figure 36 illustrates the concept of the maximal availability priority. Because the array I/O ports

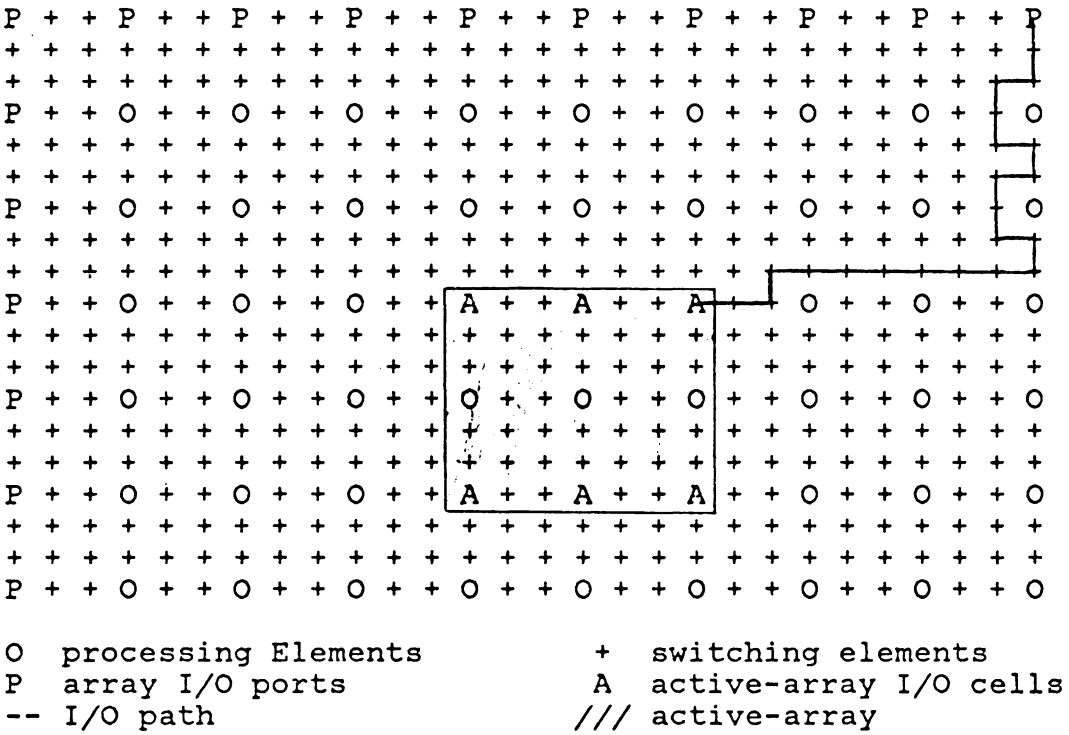


Figure 34. Maximal Availability Priority, E,N,W,S -- Path 1

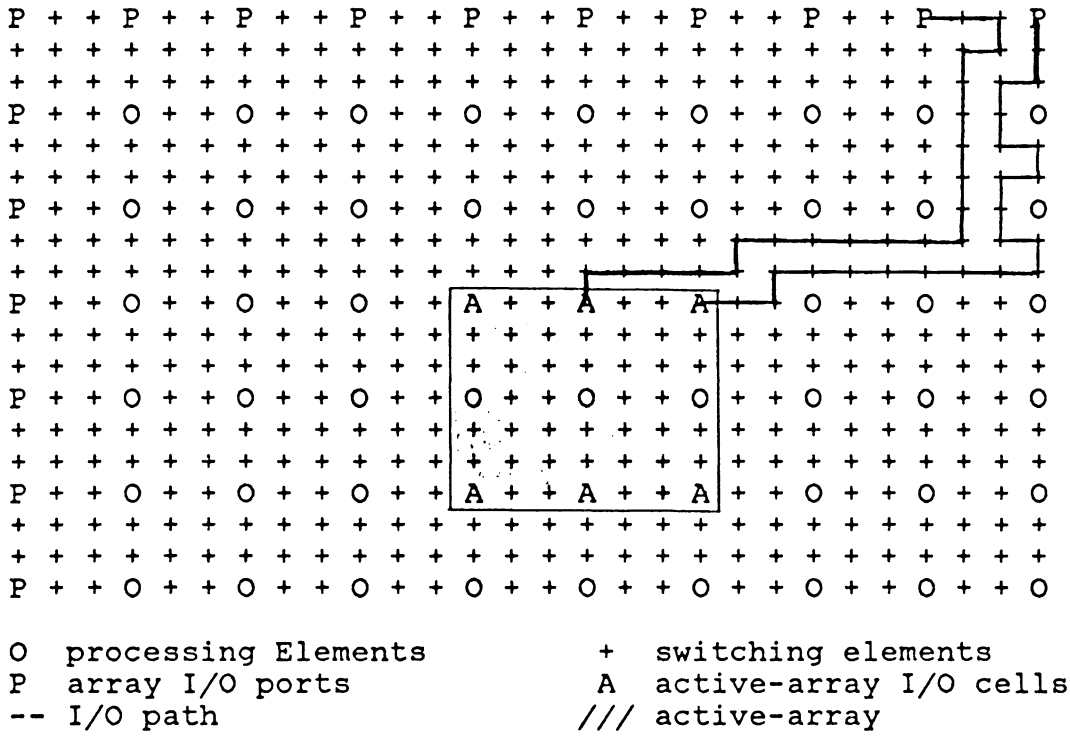


Figure 35. Maximal Availability Priority, E,N,W,S -- Path 2

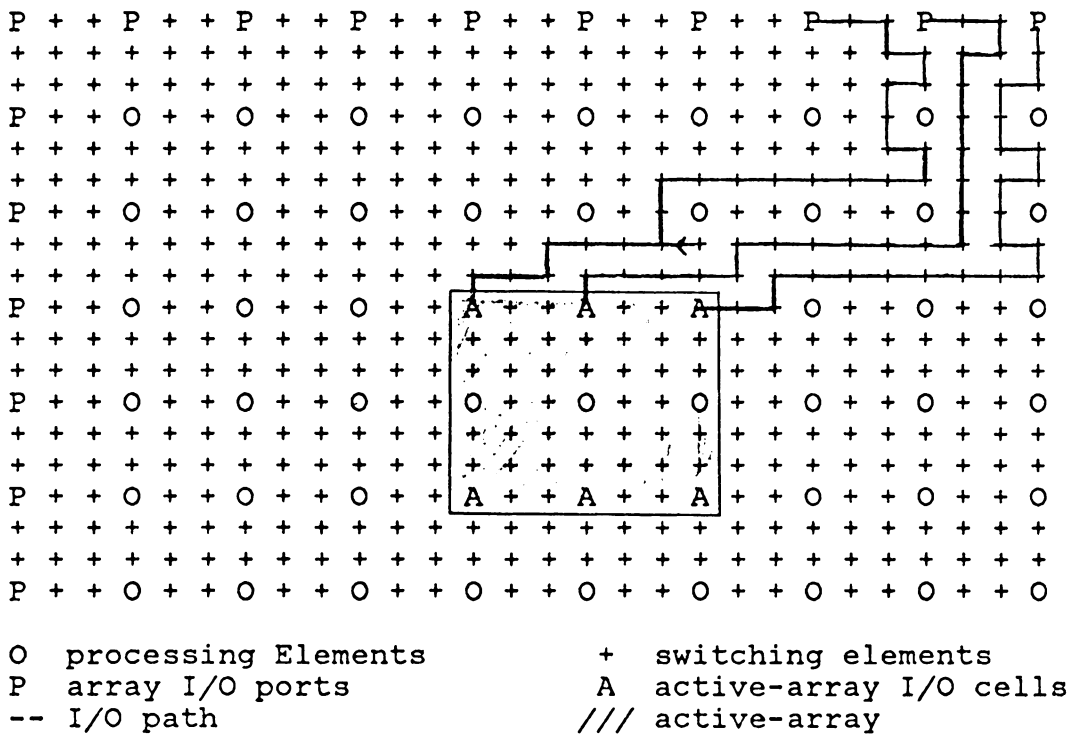


Figure 36. Maximal Availability Priority, E,N,W,S -- Path 3

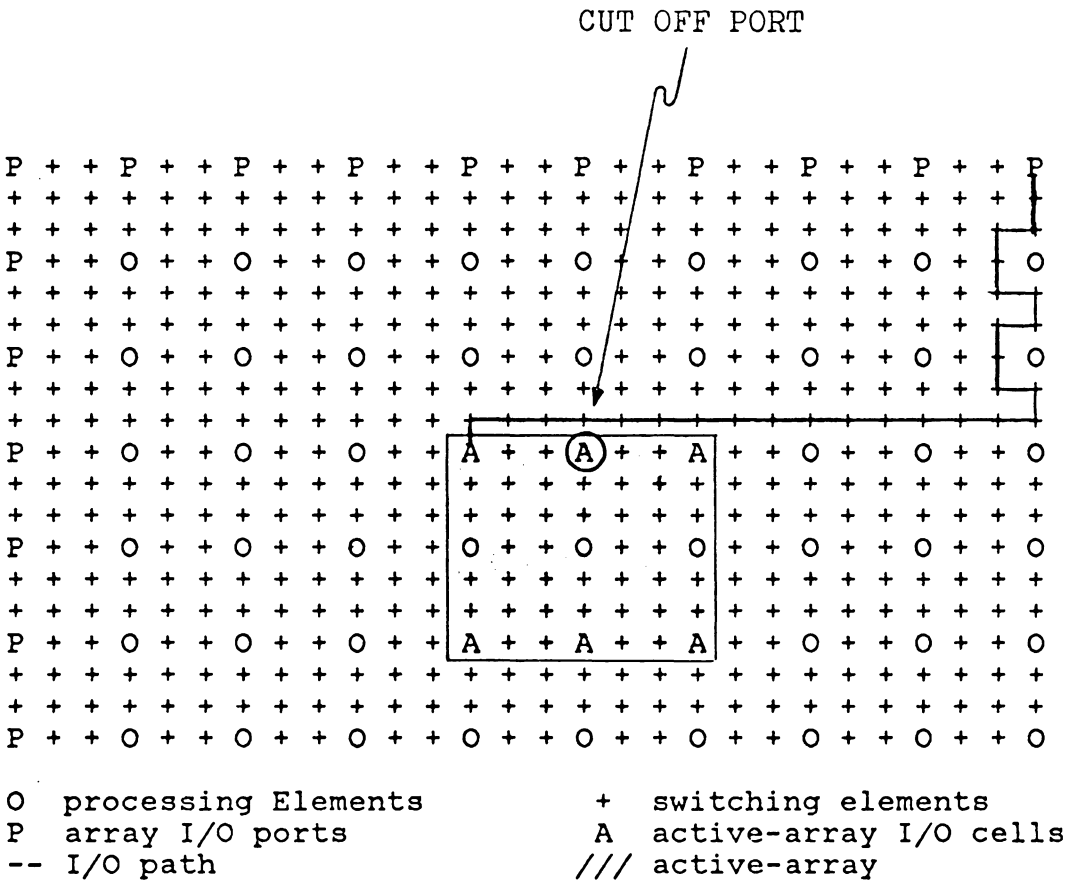


Figure 37. A Path Barrier

are at the "top" of the cellular array, the first direction priority is East, and the second is North. Notice, that if the third cell constructs its path first, then its path forms a barrier for the second and third paths as shown in Figure 37 on page 89. Obviously, there must be an order for path construction, and this order is accounted for in the **go first priority**.

The Go First Priority

If the active array has more than one input cell, and more than one output cell, then the cells need to be labeled as the 'first input' cell, the 'second input' cell, the 'first output' cell, etc.. The reason for this is twofold. First, the array I/O ports, which receive the label during path construction, must know exactly for which active-array cell they need to process data. The labels are translated into prefix tags at the I/O ports, as was discussed in the section on array I/O ports. Secondly, the active-array I/O cells must be labeled so that each cell has a **go first priority**. This priority is assigned so that paths under construction will not collide or form path barriers for other paths. An example of a path barrier was presented.

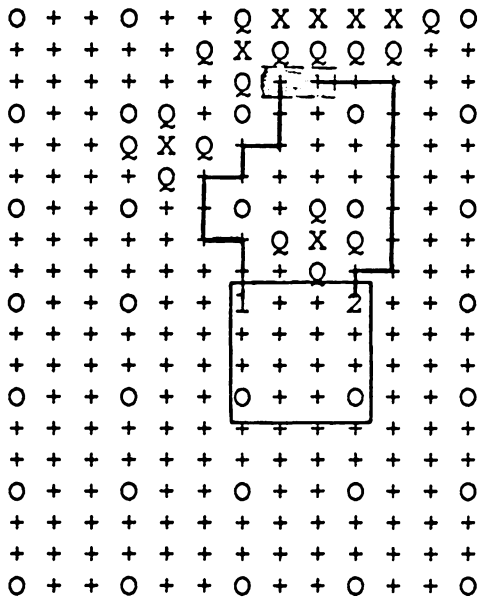


Figure 38. Handshake Collision

Colliding occurs when two or more pointer cells request the same cell to be the next path link, or when two or more pointer cells send handshake messages to each other. For example, suppose that an active-array has two input cells labeled "1" and "2," as illustrated in Figure 38 on page 95. Then, if both cells initiate the path construction process at the same time, the paths can handshake collide.

To solve the problem of collisions and barriers, paths can be constructed one at a time using the maximal availability priority concept for the priority direction list. However, constructing one path at a time is not time efficient if down time for reconfiguration is to be minimized. One way to minimize path construction time is to allow the active-array output cells to initiate path construction at the same time as the active-array input cells. If the direction priorities send each type of path to opposite corners of the array as illustrated in a) of Figure 39, then the possibility of path collision is minimized. In this figure, the direction priority for the input paths is ENWS, and the direction priority for the output paths is SWNE.

Another way to minimize path construction time is to allow each path a head start before the next active-array cell initiates path construction. Because the input (output) paths have the same direction priority, they run parallel to

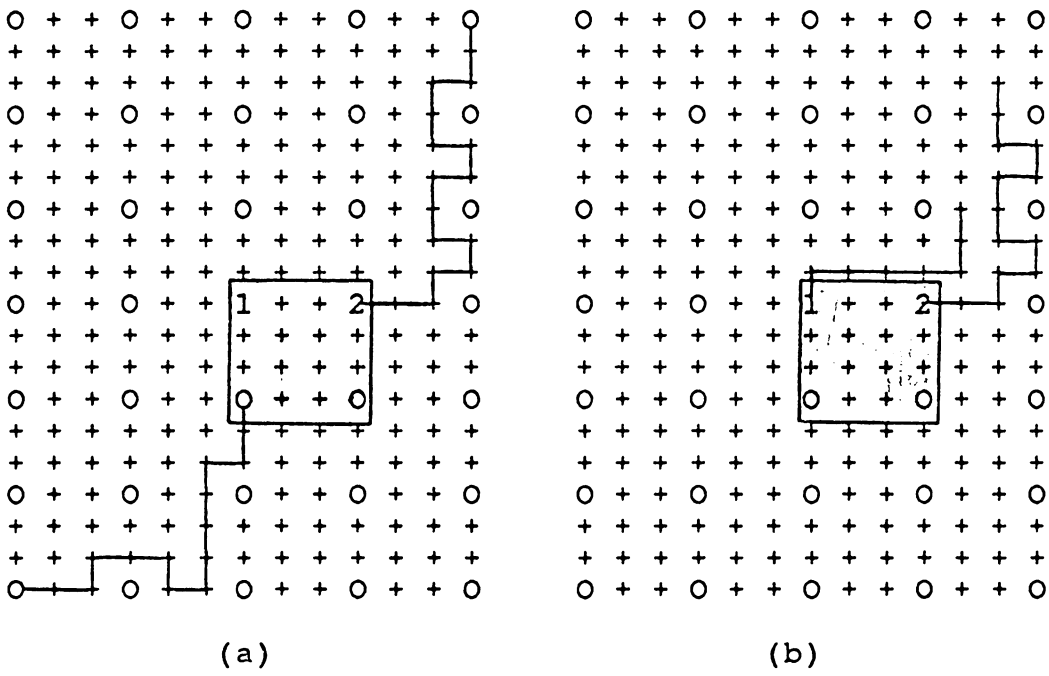


Figure 39. Solutions to the Collision Problema) Opposite Corners b) Head Start

one another, and do not collide, given a sufficient head start as illustrated in b) of Figure 39. The above two methods can be combined to further minimize path construction time.

A head start time is defined as the time that an active-array I/O port must wait before it begins path construction. Active-array I/O ports are separated by two SEs, and if there are no faults in the array, then one I/O path will be two steps ahead of the next path in the construction process. Therefore, a head start time is not needed. However, if there are faulty cells, then a path can be constructed in a direction that will interfere with another path's construction. A "sufficient" head start time is dependent on the number of faulty cells in the array. Since the number of faulty cells is variable, the head start time must be based on a criterion that uses fixed properties of the array.

The criterion chosen for the I/O algorithm is based on number of SEs in the width or the length of the array. As illustrated, in Figure 40, if a path must be constructed in the area above an adjacent I/O port, then a sufficient headstart time allows the path time to be constructed to one end of the array and back. If the construction of each path

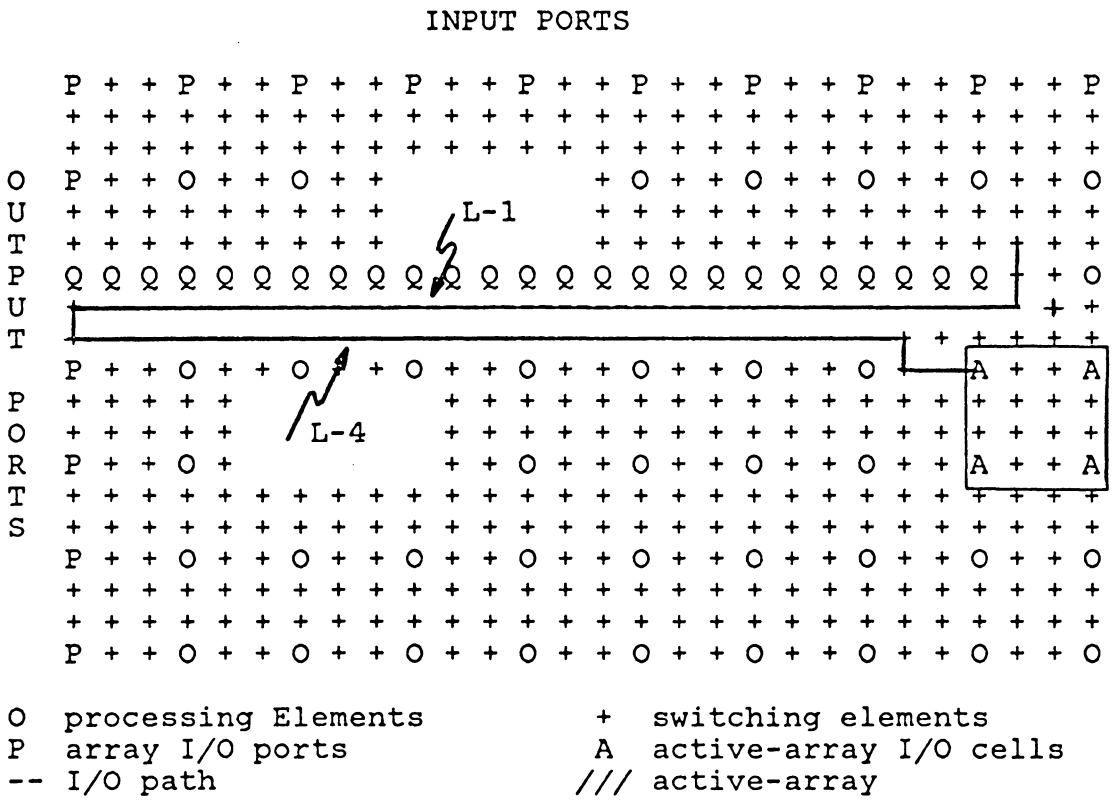


Figure 40. Head Start Time

link is considered as one time unit, then the illustrated head start time is

$$2*[\max(\text{width}, \text{length}) - 1] - 3$$

where the 3 accounts for the "natural" head start of the first path. If the path needs longer than (2*X - 5) time units, where X=max(width, length), then the number of faulty cells is likely to cause reconfiguration failure. (2*X - 5) times the number of active-array I/O ports can define the time limit for path construction.

4.2.3 Choosing the Algorithm Parameters

The direction priority and the go first priority are parameters of the I/O algorithm. Once they are established, paths can be constructed by the handshake methods that were explained. This subsection outlines the way in which the priorities are chosen.

There are eight direction priorities that have opposite directions in the first and third, and second and fourth positions. These are:

1. NESW
2. NWSE
3. SENW
4. SWNE
5. ENWS

6. ESWN
7. WNES
8. WSEN.

The direction priority that is chosen is dependent on the location of the array input (output) ports with respect to the location of the active-array input (output) ports; and on the location of the output (input) ports. Because the active-array ports will face or be perpendicular to the array ports, the second direction in the direction priority list indicates the location of the array port type, and the first direction priority allows for the maximal availability priority. Thus, given an array port location, there are two choices of direction priority, and the one chosen depends on the location of the other type of port. That is, the maximal availability priority should not direct the path towards the other type port. For example, if the output ports are to the east and the input ports are to the north, then the SENW direction priority is chosen for the output ports, and WNES is chosen for the input ports.

4.4 THE ALGORITHM IN A PROGRAMMING LANGUAGE

Because the I/O algorithm uses many conditional steps, the algorithm is best presented in a programming language text. This programming language is based on PL/1 but, for simplic-

ity of representing the steps, does not strictly adhere to the language rules. The conditional statements of this programming language make it particularly suitable for describing the I/O algorithm. The steps are divided into sections which are represented by procedures. The following paragraphs summarize each procedure. The program language procedures of the algorithm are the Diagnostic Procedure, the Path Start Procedure, the Pointer Cell Procedure, the Alert Procedure, the Backtrack Procedure, and the Path Reconfigure Procedure.

Summary of the Diagnostic Procedure

The DIAGNOSTIC procedure is the main procedure of the diagnostic phase. The testing algorithm is called from this procedure. If faults are found within the active-array a reconfiguration flag, recon_flag, is set. If faults are found in the paths, then a path reconfiguration flag, path_flag is set. A set flag, recon_flag, initiates the reconfiguration process and the path construction process. A set path_flag initiates path reconfiguration.

Summary of the Path Start Procedure

The PATH START procedure is called after the RECONFIGURATION procedure (The RECONFIGURATION procedure is presented by

Brighton [5]). During the PATH START procedure, the cells in the array check their global state. If a cell is in the active-array it uses its position as a pointer to memory to determine whether or not it is an active-array I/O port. If the cell is an active-array I/O port, it executes the POINTER CELL procedure. If the cell is not an active-array port, or is not in the active-array, it executes the ALERT procedure. If at any time a cell receives a reconfiguration control word, it sets a flag called recon_flag. When set, this flag is used in the DIAGNOSTIC procedure to initiate reconfiguration.

Summary of the Pointer Cell Procedure

The POINTER CELL procedure is called from the PATH START procedure and the ALERT procedure. During this procedure the pointer cell sends handshakes to all of its neighbors excluding the neighbor that precedes it in the path. The handshake is arbitrarily chosen as 10000000. After waiting for a response to the handshake, the cell clocks data on the buses into its ports. Neighbors that are eligible to be the next path link send a YES response of 1xxxxxxx. The contents of each port is then evaluated. If the contents are zero, the corresponding neighbor can not be a path link. If the contents are not zero, but the first bit is not a 1, the cell has been sent a reconfiguration control word. In this case

the cell sets the recon_flag and exits the procedure. If the response is 11111111, then the neighbor is a PE port and is immediately chosen as the next path link. Otherwise, the port that corresponds to the neighbor with the highest priority is chosen.

Before exiting the procedure, the cell records the cell to which it sent the pointer cell status. The preceding_neighbor and the following_neighbor are associated with numbers. A function termed as "op" in the procedure uses the numbers to define a local state for the cell. The local state is the switch position that will connect the preceding_neighbor to the following_neighbor.

Summary of the Alert Procedure

The ALERT procedure is executed by all cells that are not in a reconfiguration or construction process. These cells continually check their input ports for path handshakes or reconfiguration handshakes. Typical handshakes that the procedure instructs the cell to look for are the backtrack handshake, the path clear handshake, the pointer cell handshake, and the construct an alternate path handshake. A handshake that does not have the path global state indicates a reconfiguration request, and instruction control is returned to the diagnostic procedure.

Summary of the Backtrack Procedure

The BACKTRACK is a short procedure that is called from the ALERT procedure. During the BACKTRACK procedure, the cell sends a backtrack handshake to its preceding neighbor. After setting its state to B, the backtrack state, the cell's control is returned to the ALERT procedure.

Summary of the Path Reconfigure Procedure

The PATH RECONFIGURE procedure is called from the DIAGNOSTIC procedure or from the ALERT procedure. The PATH RECONFIGURE procedure is called from the DIAGNOSTIC procedure at the start of path reconfiguration. The cells that surround the faulty path cell stay is the quarantine state and pass either a clear handshake or an alternate path handshake. If called from the ALERT procedure, the PATH RECONFIGURE PROCEDURE changes the local state of the cell to the quiescent state, and sends a clear handshake to its following neighbor.

DIAGNOSTIC PROCEDURE

```
START PROCEDURE DIAGNOSTIC
CALL TEST PROCEDURE;                    /if fault is found in active-   /
                                        /array, then set recon_flag.    /
DO WHILE RECON_FLAG=1;
   CALL PROCEDURE RECONFIGURE;          /reconfigure array              /
   CALL PROCEDURE PATH START;           /construct paths                /
END WHILE;
IF PATH_FLAG=1                          /fault is found in one or more  /
                                        /paths                          /
    THEN CALL PROCEDURE PATH RECONFIGURE;
                                        /reconfigure path(s)            /
    ELSE;
END START;
```

PROCEDURE PATH START

```
START PROCEDURE PATH START              /Pattern growth is complete     /
L=ARRAY_LENGTH
W=ARRAY_WIDTH

HEAD_START_TIME = 2*MAX(L,W) - 5        /The head start time            /
RECON_FLAG=0

IF GLOBAL_STATE.NE.0                    /The cell is in the active array/
   THEN CALL POSITION;                  /Determine whether or not cell  /
                                        /is an active-array I/O port    /

        IF PATH_NUMBER.NE.0             /Cell is a port                 /

           THEN WAIT_TIME=PATH_NUMBER*HEAD_START_TIME;

                DO WHILE WAIT_TIME>0;   /wait until other paths have had/


                                        /a sufficient head start        /

                   WAIT_TIME=WAIT_TIME-1;

                END WHILE;

                CALL PROCEDURE POINTER CELL;
                                        /Start path construction        /
           ELSE;
   ELSE;

IF RECON_FLAG=0


   THEN CALL PROCEDURE ALERT;

   ELSE;

END START;
```

POINTER CELL PROCEDURE

```
START PROCEDURE POINTER CELL
PORT(1)=0;                              /data from the north neighbor   /
PORT(2)=0;                              /data from the east neighbor    /
PORT(3)=0;                              /data form the south neighbor   /
PORT(4)=0;                              /data from the west neighbor    /
FOLLOW_NEIGHBOR=0;                      /the neighbor that is chosen    /
                                        /as the next path link          /
CHOOSE=0;                               /the current choice of the next /
                                        /path link                      /
EQU CLOCK=ARRAY_CLOCK

X=1                                     /a variable counter for loop    /
DO WHILE X<5;
   IF X=PRECEDING_NEIGHBOR              /do not send a handshake to     /
      THEN PORT(X)=0;                   /the preceding neighbor         /
      ELSE PORT(X)=10000000;            /load port with a handshake     /
END WHILE;

DO WHILE CLOCK=0;                       /wait for high clock            /
END WHILE;

X=1;
DO WHILE X<5;                           /clock the port contents on to  /
   BUS(X)=PORT(X);                      /the cell-to-cell bus           /
   X=X+1;
END WHILE;

DO WHILE CLOCK=0;                       /wait for response to handshake /
END WHILE;

X=1;
DO WHILE X<5;                           /clock the data on the buses    /
   PORT(X)=BUS(X);                      /into the ports                 /
   X=X+1;
END WHILE;

X=1;
COUNT=0;                                /clear the backtrack count      /

DO WHILE X<5;

FIRST_BIT=PORT(X).AND.10000000          /first bit indicates path control

   IF PORT(X).NE.00000000               /check for port that is not zero/

      THEN IF FIRST_BIT=0000000         /first bit of zero indicates a  /

              THEN RECON_FLAG=1;        /reconfiguration control word   /

                   X=5;                 /set flag to exit path construct/

              ELSE IF PORT(X)=11111111 /neighbor x is a PE port        /
                      THEN CALL SEND;   /send pointer cell information  /
```

```
                    ELSE CALL PRIOR; /Does the port have a higher    /
                                     /priority than the current next /
                                     /link choice ?                  /

      ELSE COUNT=COUNT+1;            /increment the backtrack count  /

   X=X+1;

END WHILE;

IF RECON_FLAG=0.AND.COUNT.NE.4       /the handshake is not a reconfig-/
                                     /ure signal, and at least one    /
                                     /neighbor is available.          /

   THEN FOLLOW_NEIGHBOR=CHOOSE;      /remember which neighbor is the  /
                                     /next path link                  /

        LOCAL_STATE=PRECEDE_NEIGHBOR.OP.FOLLOW_NEIGHBOR;

                                     /the local state is a function of/
                                     /the numbers associated with the /
                                     /neighbors                       /

    ELSE CALL PROCEDURE BACKTRACK;   /all neighbors responded to hand-/
                                     /shake with a NO. Must backtrack /

END START;
```

PROCEDURE ALERT

```
START PROCEDURE ALERT
SKIP_FLAG=0;                              /clear the skip flag       /

DO WHILE CLOCK=0;
END WHILE;

X=1;

DO WHILE X<5;

   PORT(X)=BUS(X);                        /receive handshakes into ports  /

   BIT_ONE(X)=PORT(X)*10000000;           /determine if first bit is 1    /
                                          /if 1 ----> path handshake      /

   X=X+1;                                 /increment counter              /

END WHILE;

DO WHILE FOLLOW_NEIGHBOR.NE.0.AND.PRECEDING_NEIGHBOR.NE.0;

   IF PORT(FOLLOW_NEIGHBOR)=10000000;  /backtrack signal sent.          /

      THEN CALL POINTER CELL;          /reassume pointer cell status   /
      ELSE;

   IF PORT(FOLLOW_NEIGHBOR)=1100110011;
                                       /construct an alternate path    /

      THEN CALL POINTER CELL;
      ELSE;

   IF PORT(PRECEDING_NEIGHBOR)=10101010;
                                       /clear remaining path           /
      THEN CALL PATH RECONFIGURE;
      ELSE;

SKIP_FLAG=1;                           /path neighbors have sent hand- /
                                       /shake, skip next section       /

END WHILE;

X=1;

DO WHILE X<5 .AND. SKIP_FLAG=0;

   IF PORT(X).NE.0                     /contents of port not zero      /

      THEN IF BIT_ONE(X)=0;            /bit one of the handshake is    /
                                       /zero, handshake is a reconfig- /
                                       /uration request                /

           THEN RECON_FLAG=1;          /set reconfiguration flag and   /
                (RETURN)               /return to diagnostic phase     /

           ELSE IF TYPE=PE_PORT.AND.ABCD_FLAG=0
                                       /bit one is a 1, so is path hand/
                                       /shake.                         /

                THEN SEND=11111111;/cell is a PE PORT                  /
                ELSE;
```

```
                IF TYPE=SE_QUIESCENT.AND.ABCD_FLAG=0
                   THEN SEND=10000000;/cell is an SE quiescent        /
                   ELSE SEND=00000000;/cell is busy                   /
                DO WHILE CLOCK=0;
                END WHILE;
                BUS(1)=PORT(1);BUS(2)=PORT(2);
                BUS(3)=PORT(3);BUS(4)=PORT(4);
                                   /put response on bus               /
                ABCD_FLAG(X)=1;    /YES response is sent              /
                PORT(X)=SEND;      /load port with response           /
                IF ABCD_FLAG(X)=1  /cell is chosen to be the next     /
                                   /pointer cell                      /
                   THEN CALL RECEIVE_POINTER_CELL_STATUS;
                        ABCD_FLAG(X)=0;   /clear flag                 /
                   ELSE;
          ELSE;
END WHILE;
END START;
```

PROCEDURE BACKTRACK

```
START PROCEDURE BACKTRACK;

G=PRECEDING_NEIGHBOR;

LOCAL_STATE=B;                          /set local state to B          /

PORT(G)=10000000;                       /send a backtrack handshake to /
                                        /preceding neighbor.           /

DO WHILE CLOCK=0;
END WHILE;

BUS(G)=PORT(G);                         /put handshake on bus          /

END START;
```

PROCEDURE PATH RECONFIGURE

```
START PROCEDURE PATH RECONFIGURE

CLEAR_HANDSHAKE=10101010
ALTERNATE=1100110011
G=PRECEDING_NEIGHBOR
F=FOLLOW_NEIGHBOR

IF CLEAR FLAG=0                          /if clear is zero, procedure    /
                                         /is called from DIAGNOSTIC, and /
                                         /path reconfiguration is just   /
                                         /starting                       /

   THEN IF STATUS_REGISTER_BIT(G)=1      /the bit of the status register /
                                         /that corresponds to the preced-/
                                         /ing neighbor is 1.             /

          THEN LOCAL_STATE=QUARANTINE;
                                        /preceding neighbor is faulty    /

               PORT(F)=CLEAR_HANDSHAKE;
                                        /send a clear handshake to the   /
                                        /following neighbor              /

               DO WHILE CLOCK=0;
               END WHILE;


               BUS(F)=PORT(F);          /put handshake on bus            /

          ELSE LOCAL_STATE=QUARANTINE;
                                        /following neighbor is faulty    /

               PORT(G)=ALTERNATE;       /send an alternate path hand-    /
                                        /shake to the preceding neighbor/

               DO WHILE CLOCK=0;
               END WHILE;

               BUS(G)=PORT(G);          /put handshake on bus            /

    ELSE LOCAL_STATE=QUIESCENT;         /clear_flag is 1, so prodedure   /
                                        /is called from the ALERT        /
                                        /procedure.  Clear to quiescent  /
                                        /state.                          /

         PORT(F)=CLEAR_HANDSHAKE;       /send a clear handshake to the   /
                                        /following neighbor              /

         DO WHILE CLOCK=LOW;
         END WHILE;

         BUS(F)=PORT(F);                /put handshake on bus            /

END START;
```

4.5 CONCLUSION

The I/O algorithm was presented as part of reconfiguration. Paths are constructed from the active-array to the array I/O ports by the process of a lead path cell handshaking with other path candidates. The next step towards implementing the algorithm is simulation to determine head start times and path construction limitations.

CHAPTER V CELL TESTING TECHNIQUES

5.1 INTRODUCTION

This chapter outlines the test techniques that can be applied to the components of the switching element and the processing element. The major components of these elements that must be tested are the microprocessor, the ROM, the RAM, the switch (and other computational plane special purpose devices), the cell I/O ports, and the cell-to-cell bus lines. The cell-to-cell bus lines refer to the buses between PEs and SEs, and between SEs and SEs. The following sections suggest how the test methods can be applied to the PE and SE cells. A specific test set is not created, but is assumed for developing the test algorithm in Chapter 6.

5.2 SELF-TEST AND BIT

If the processing elements and the switching elements are implemented by a general-purpose microprocessor system, then microprocessor testing techniques can be used to test the array cells. Also, because the array is a self-diagnosing architecture, the microprocessor tests must be conducted and monitored by cells in the array.

There are two microprocessor testing methods that can be used to test the cells of the array. These testing methods are the Self-test and the Built-in Test (BIT). Both of these testing methods have a test program that is stored in a Test ROM, and the microprocessor and/or the microprocessor system is tested by the execution of the test program. In the case of the Self-test, the microprocessor maintains control of the system, and tests itself by reading in the test program instructions from the Test ROM, and executing them. Figure 17 on page 44 shows a microprocessor system with a test ROM.

The built-in test is composed of a test ROM and a comparator. During the test mode, the test ROM and the comparator form a sequential controller that is capable of testing the microprocessor. The BIT testing method can be extended to include the testing of other devices in the system such as ROM, and RAM. Very often, a microprocessor other than the system's main microprocessor is used, instead of the comparator, to test the main microprocessor and the peripheral devices. If the main microprocessor is assumed to be fault-free, then it is capable of testing the other devices in the system [4]. This concept is used in the test algorithm. After the microprocessor is tested and found to be fault-free, it is considered to be reliable in determining the faulty or fault-free statuses of the other devices in the cell.

The Self-test and BIT methods must be monitored externally for a GO/NOGO test result. Otherwise, a fault in the microprocessor system could not be identified as a fault by devices external to the microprocessor system. In the array, every microprocessor cell has monitors, which can be other cells. The Self-test and the BIT testing techniques are the basis for the testing algorithm of the cellular array.

5.3 FUNCTIONAL TESTING FOR THE MICROPROCESSOR

The microprocessor can be viewed as a complex, gate-level component, or as a component that executes functions. For the first case, the microprocessor test program can be developed only with a knowledge of the internal structure of the microprocessor and is called a structural test. The internal structure of the microprocessor is not readily available to the user, and, therefore, this method of testing is almost impossible to implement. The microprocessor can also be tested using functional level testing. This test method is based on a register-level description rather than a gate-level description. The microprocessor test program is developed using the block diagrams and instruction set that are available in the user manuals. The test sets for the cell microprocessor are assumed to be functional test sets in the testing algorithm.

Functional level testing of the microprocessor involves performing a series of fault-detecting tests on each of its subfunctions. The tests must be constructed with an awareness of how the device might fail. Features of the microprocessor chip that must be tested are the program counter, the flag register, the stack pointer, the index register, the accumulator, the ALU, the register flags, the instruction register, the instruction decoder, the timing unit, the address buffer, and the data buffer [3].

There are several algorithms that have been proposed to construct test sequences, including those by Thatte and Abraham [22], Robach and Saucier [18], and Shen and Su [19]. These methods, which are outlined in the following paragraphs, can help to develop microprocessor test sequences for use in the array testing procedure.

Thatte and Abraham propose a method based on graph theoretical representation of the microprocessor at the register transfer level. In their graph, nodes indicate registers and directed arcs between nodes identify instructions which cause data to be manipulated or transferred. The functional level fault model is defined for each major function -- i.e. register decoding, instruction decoding and control, data storage, data transfer, and data manipulation. They define this as a self-test even though this method requires moni-

tors. A test sequence developed by this technique can be applied to the array by letting cells monitor neighboring cells.

Robach and Saucier group instructions into ordered sequences such that all instructions in the same class have the same type of abstract execution graph. All instructions in a class are partitioned into subclasses according to the accessibility and the observability of each instruction. Any instruction is represented by a bipartite graph such that memory elements, sources, and destinations of instructions are associated with a type one node. The microoperations performed by the instructions are associated with a type two node. Tests are applied to the microprocessor by verifying the correct execution of the instructions.

Shen and Su also present a method for functional testing of microprocessors. Control faults might lead to the partial execution of an instruction or a change in the execution sequence. Therefore, they describe a fault model for microprocessors that emphasizes the control fault at the RTL level instead of the instruction level. The faults described include data processing faults and control faults. Data processing faults occur in data storage, data transfer, and data manipulation. The control faults occur in register decoding and instruction decoding. They derive testing requirements based on the different types of operations of off-the-shelf microprocessors.

Functional testing involves the initialization of and the reading of internal registers. Write and read sequences constitute this initialization and reading process, and are modeled as a sequential machine. Shen and Su define the kernel of a microprocessor as the basic write and read instructions. A checking experiment verifies the kernel by verifying the sequential machine. The kernel is then used for testing each instruction.

The above testing methods require external control and monitoring to control the execution of the test, and to verify the test results. Any of these test methods can be applied to the array if, as mentioned, cells verify test results of other cells. A cell's test could be controlled by a neighboring cell. However, the test algorithm allows a cell to control its own test by executing the test sequence stored in its test ROM. If a cell is unable to execute its own test sequence, then the neighboring cells detect this fault because they will receive faulty, or no test results.

The bootstrap technique can be used in ordering the test sequences [4]. Certain functions, which are chosen and assumed functional, are used to test the remainder of the instruction set. As each instruction is tested, it is added

to the group, and after all of the instructions have been tested, the original instructions are tested. The concept of the bootstrap technique is helpful in developing the test algorithm because a cell's faulty interpretation of a test result due to an instruction, or function failure will eventually be caught by other fault free cells.

5.4 TESTING THE PERIPHERAL DEVICES

Once it is tested and found to be fault-free, the microprocessor can test the control ROM, the RAM, and the computational plane components, such as the switch, using test instructions stored in the Test ROM. The following subsections describe typical tests for the devices.

5.4.1 ROM Test

The ROM can be tested by verifying the correct execution or partial execution of its instruction sequences. The test ROM contains the address of the instruction sequence to be tested, the input data, and the expected responses. The microprocessor executes the instructions using the input data, and compares the results of the execution with the expected results. This process of checking every instruction sequence is too time consuming for use in the cellular array test because an active cell is only using a part of the ROM

instructions. The ROM instructions that are not being used, i.e. the instructions for a local state other than that of the cell, do not need to be tested. In other words, to conserve test time, the test sequence should not be fault "preventive."

5.4.2 RAM Test

There are four main fault categories for a RAM [3]:

(i) incorrect addressing,

(ii) multiple write,

(iii) pattern sensitivity -- read or write,

(iv) failure to retain data.

Typical checks for these faults would include writing to and reading from the device to check that the data-in, data-out, R/W lines, and enable lines are not S-A-1 of S-A-0. The memory cells are checked by making certain that each can be set to zero and to one.

The RAM is used to store the partial computations that are in the cell before the start of the diagnostic phase. A section of RAM should be tested during the i-1(th) diagnostic phase to be used for the i(th) diagnostic phase. Likewise, a section should be tested during the i(th) diagnostic phase to be used for the i+1(th) diagnostic phase. Two sections

of RAM are needed so that one can be tested while the other stores the results.

5.4.3 Switch Test

The switch can have a crossover of one or a crossover of two as indicated by the local state of the cell. The switch position that is associated with the present local state of the cell is the one that should be tested. Unused faulty switch positions do not affect the fault-free functioning of the switch. A quiescent cell can test a different switch position each diagnostic phase because a fault in its switch does not initiate reconfiguration. The quiescent cell can record any faulty switch position so that it can be used for functions that do not require the faulty positions.

The switches are tested by setting the switch to a local state and observing the results of a switch connection. For example, let a local state connect cell-to-cell bus A to cell-to-cell bus B. Port A is loaded with a data string, and the other ports are cleared. The contents of the ports are clocked on to the cell-to-cell buses, and are allowed to propagate through the switch. During the next clock pulse, the information on all of the cell-to-cell buses is clocked into the ports. Port A and Port B should contain the data string, and the other ports should contain a null string.

The tri-stated cell-to-cell buses ensure that the test sequence of one cell does not interfere with the test sequence of another cell.

Example 5.4.3.1

If a test is to verify that information on bus A can be switched to bus B, then an eight bit string of alternating zeros and ones is clocked onto bus A while all other buses receive a zero vector. During the next clock pulse, the information on the buses are clocked into the ports. The test vector should only be in ports A and B. If the complement of this vector is sent, then the switch is tested for stuck-at faults. The test is not complete. A series of input vectors must be placed on the unused buses to ensure that the vectors are not switched onto other buses.

5.4.4 Cell I/O Port and Cell-to-cell Bus Test

A cell's microprocessor will not test its I/O ports. Instead, the cell considers its communication ability to be fault free, and during testing it assumes that it can send predetermined test sequences to each of its neighbors. The neighbors are then responsible for determining whether or not the cell can communicate successfully. The exact nature of the fault does not need to be known. A typical test sequence

for the I/O ports is to transmit and receive the following sequences:

0000....0000

1111....1111

0101....0101

1010....1010.

These sequences test for stuck-at faults and coupling faults. A cell assumes that it can address each of its own ports. If a cell can not address a port, then one of its neighbors does not receive test vectors, and likewise, the cell does not receive test vectors from its neighbor. Both assume that the other is faulty, and the cells quarantine each other, which is the desired response. The cell-to-cell buses can be tested by the same test sequence. If the CPU test is designed so that the listed sequences are the first few test results, then the port and cell-to-cell bus test can be combined with the CPU test.

CHAPTER VI THE TESTING ALGORITHM

6.1 INTRODUCTION

Fault tolerance is added to the array by reconfiguring the array if a cell in the active-array fails. The testing procedure to determine cell failure is described in this chapter. First the types of tests that are conducted on cells are outlined, then, the basis by which a cell determines the faulty or fault-free status of a cell described. An examples section illustrates how different fault-types are quarantined, and represents a "dry run" of the test algorithm steps.

The following terminology is used in the Testing Algorithm.

- **Test Pass.** There are two alternating phases of array control -- the diagnostic phase and the computation phase. The diagnostic phase is used for both reconfiguration and testing. Therefore, the test pass is defined as the part of the diagnostic phase that is used for testing. The number of clock pulses needed for a test pass should be a fraction of the number of clock pulses used for computation

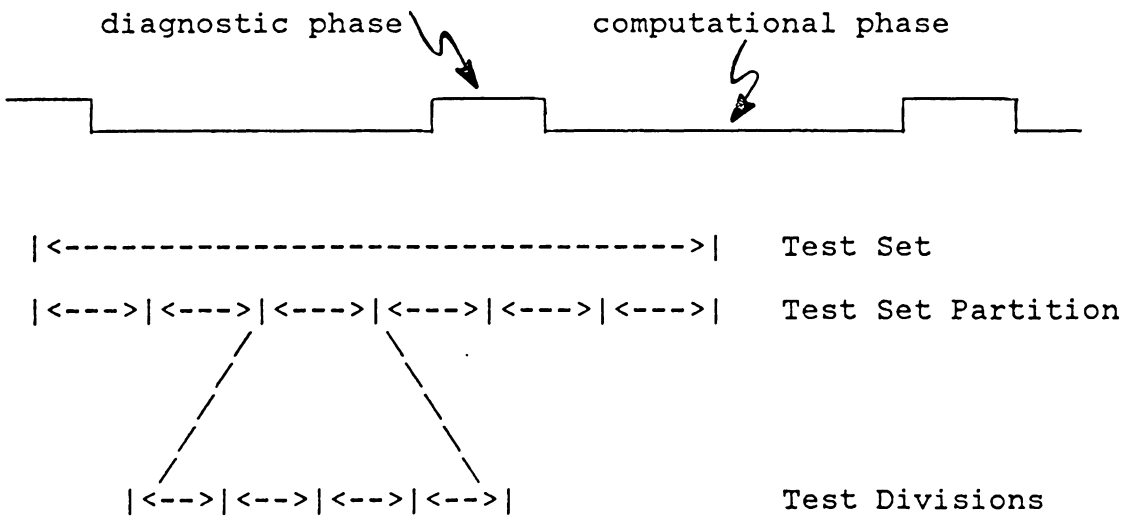


Figure 41. Test terminology

- **Test Set.** A test set is the test for either the switching element or the processing element. The test set is spread over consecutive diagnostic phases because it is too long to be completed in a single test pass.
- **Test Set Partition.** A test set is divided into partitions. Each partition is applied during one of the test passes.
- **Test Division.** Each test set partition has test divisions. The test instructions for each division is concerned with testing different components of the cell. The Test Algorithm uses four test divisions, as described in the next section. Figure 41 on page 127 illustrates the test terminology.

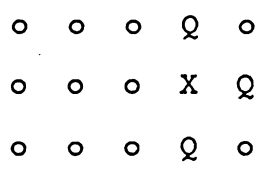
6.2 TEST DIVISIONS

There are four divisions of each test partition. The test divisions include the CPU test, the peripheral devices test, the computational plane test, and the state test. A test sequence is associated with each of these divisions, and is created by the methods described in Chapter 5. Some of the test divisions such as the CPU test are completed each test pass; while other tests, such as peripheral device test, take an entire test pass to complete. The following subsections describe the test divisions.

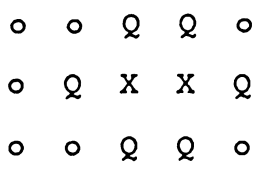
6.2.1 The CPU Test Division

Because the control plane of each cell is modeled by a microprocessor, the CPU test is conducted simultaneously on each cell. The CPU reads in test instructions from the test ROM, executes them, and sends the results to all four of its neighbors by means of the Von Neumann communication pattern. Since each cell is exercising the same test at the same time, a cell can easily monitor neighbor results by comparing them with its own test results. The process of sending and receiving test results for comparison is time consuming, and, therefore, the frequency of test verification must be considered when designing the CPU test.

A cell must send the same test results to all of its neighbors, so that a fault in the CPU is recognized and quarantined by all of the fault-free neighbors. A faulty neighbor can misinterpret a faulty result as correct, or a fault-free result as incorrect, and not take the same actions as the other neighbors. However, if the bootstrap test method is applied, the comparing functions of the CPU are eventually tested, and the faulty cell and its faulty neighbor are quarantined as illustrated below.



instruction test TIME=i



Compare mechanism test TIME=i+j

To reasonably assure that all faults such as the one illustrated are caught, the entire CPU test is completed in a single test pass. Also, this test is repeated every test pass so that control can be reliably restored to the control plane [10].

The test results are identically received by all of the neighbors only if the cell-to-cell communication is fault-free. For example, the CPU must be able to address each port, the ports must be able to transmit and receive any binary string, and the cell-to-cell bus must be fault free. Assuming that a CPU fault is not masked by a communication fault, any test result communication is a constant check of the communication components. That is, if there exists a faulty communication component between two cells, a and b, then a receives a faulty result, or no result at all in the case when a cell can not address a port. Upon receiving a faulty result, a will quarantine b, and, therefore will not communicate or send test results to it. Thus, b also receives no more test results, and quarantines a. The result -- the

faulty communication component is quarantined by both cells. This concept is illustrated further in the Examples section, 6.6.

If the algorithm does not rely on chance for detecting a communication fault, then it must include a communication test. The communication test can be combined with the CPU test, by requiring the CPU test results to be effective line and port binary test strings. For example, if the first two CPU results are 01010101 and 11111111, then by sending these strings and their complements, the cell-to-cell bus and port are tested for stuck-at-faults, and coupling faults.

Because there are eight cell-to-cell buses, the communication test must use the Moore Communication Pattern. However, after the communication components have been verified as faulty or fault-free, the Moore pattern is abandoned in favor of the less time consuming Von Neumann pattern.

Summary. The first test division is the CPU test, which includes the cell-to-cell communication test, and is outlined below.

1. Save partial computations (from computational phase) in RAM.
2. Set up the Moore communication pattern.

3. Execute the first test sequence which has a result of alternating zeros and ones.
4. Send the result to, and receive the result from the Moore neighbors.
5. Compare the received results with the sent result. If the result of neighbor x is different from the sent result, quarantine neighbor x.
6. Send (receive) the complement of the result to (from) the fault-free Moore neighbors.
7. Compare results with expected results, and Quarantine any disagreeing cells.
8. Repeat steps three through six for the second test result of 11111111. If the communication test is complete go to the next step.
9. Set up the Von Neumann communication pattern.
10. Execute the next, or the next series of CPU test instructions.
11. Send and receive results to and from all fault-free neighbors.
12. Compare and quarantine all neighbors with disagreeing results.
13. Repeat steps eight through eleven until the CPU test is complete.

6.2.2 Peripheral Device Test Division

The peripheral devices are defined as the components, or parts of the components, that are contained in the control planes of each cell, such as the RAM and sections of the control ROM. The CPU has been tested and is regarded as fault-free by its neighbors, therefore, its control of the peripheral device test is reliable. A cell that is regarded as faulty may continue the test procedure, but because it is quarantined, its actions are not acknowledged.

The peripheral device division tests the control ROM and the RAM. Each cell conducts the same test so that the results are identical. Because the CPU is reliable, and because the results are the same for each cell, the peripheral device test can either be self-monitored or neighbor-monitored.

For the first type of monitoring, the CPU sends test vectors to the devices and checks the results against known results. The test ROM stores both the test sequence and the test results. If the neighbors monitor test results, then they compare received test results against their own test results. The self-monitor is a faster test because cells do not have to send and receive results -- cell-to-cell communication is time-consuming. However, the extra ROM storage for test results adds to the ways in which a cell can fail,

e.g. the ROM bits of the test results may be faulty. Also, a reliable CPU does not guarantee that the test ROM sequences are correct.

Neighbor-monitoring testing is slow but assures that the control plane functions are being tested. A fault in test control will be caught by the neighbors when they do not receive an acceptable result. If fault-tolerance is of greater importance than speed, the neighbor-monitor test should be used, but the neighbor test need not verify every test result. The CPU can verify some of the results and send others to be neighbor verified so that the test is a combination of the self-monitor and the neighbor-monitor. The test is faster than a totally neighbor-monitored test, but, the occasional neighbor check verifies that the test is proceeding correctly.

Using either type of monitoring, the RAM and the ROM test are too long to complete in one test pass. Therefore, the present test sequence address must be stored so that during the next test pass, the test is resumed at the stored address.

Summary. The second test is the peripheral device test and is outlined below.

1. Load ROM address pointer.
2. Apply ROM test and compare results against the known result.
3. Repeat step 2 for the duration of the self-monitored segment.
4. Send (receive) results from (to) neighbors for a neighbor-monitor.
5. Compare own results with neighbors results, and quarantine neighbors with differing results.
6. Continue from step 2 until the end of the time allotted for the ROM test and store the present address in the test sequence.
7. Load the RAM address pointer.
8. Apply the RAM tests, and repeat until the end of the time allotted for the RAM test.
9. Test the unused partial computation storage space.

6.2.3 Computational Plane Test Division

The computational plane test is a self-test. During this test partition the CPU in each cell reads test sequences from its own test ROM, executes them, and compares the result with known results that are stored in the test ROM. Neighbors do not verify the faulty of fault-free status of a computational plane component because SEs and PEs have differing computa-

tional planes and each would have to store the test results of the other.

During the computational plane test, the SE tests its switch and the PE tests the functions of the computational plane that were not tested during the CPU test. If a cell's computational plane can operate fault-free for the particular function assigned to it, then it is not necessary for a cell to test other computational plane functions. Therefore, during this test division, active cells test the computational plane functions that are associated with their local state, while quiescent cells test a different function each test pass.

Because the computational plane test is a self test, a cell that determines that its computational plane is faulty must quarantine itself. The implications of self-quarantining are further explained in section 6.5.

Summary. For a crossover of one, the local state of the switch defines a cell-to-cell bus, W, connection to a cell-to-cell bus, X. For a crossover of two, an additional connection Y to Z is made. The third test is the computational plane test, and is outlined below.

1. Disable cell-to-cell communications

2. If the cell is an active-array cell, then check local state register, and address test sequence that corresponds to the local state.
3. If the cell is a quiescent cell, then address the test sequence that corresponds to the current local state address pointer.

The SE test.

1. Send control vectors to the switch that correspond to a local state.
2. Send data strings to the ports that are associated with the W and Y buses.
3. Clock strings on to the buses and allow them to propagate through the switch.
4. Clock results into the ports, and verify that the data strings are in ports X and Z.
5. If results are incorrect, set the local state to Q, record the faulty switch position, and continue until test is complete.
6. If the cell is quiescent, then increment the local state address pointer.

The PE test.

1. Execute the functions that correspond to a local state.

2. Compare results with known results.
3. If results are faulty, then set the local state to Q, record the faulty function, and continue until the test is complete.
4. If the cell is quiescent, then increment the local state address pointer.

6.2.4 State Test Division

The state test is the last test division, and ensures that the CPU can reliably assume computations during the computational phase of the system clock. This test is a neighbor test in which the neighbors decide the validity of different test result sequences. One sequence that each cell sends to its neighbors is its state. Because the state registers were tested during the peripheral device test, the state of the machine was stored and reloaded. To check whether or not cells have successfully reassumed their correct states, cells send their local and global state to their neighbors so that each cell can decide whether or not it is in a legal neighborhood. If a cell is not in a legal neighborhood, then it must quarantine the neighbors that it determines are incorrect.

Because the computational test division is conducted without neighbor verification, cells can get out of test

synchronization, or fail without being quarantined by their neighbors. The state test is one last check for cells to determine whether or not their neighbors are in test synchronization. If a cell receives a legal state from its neighbor at the correct time, then it can assume that its neighbor is in test synchronization and has not failed.

During the state test division a cell checks its own state register. If a cell is in the quarantine state and has the global state, then it initiates reconfiguration. Otherwise, it executes the last test division steps.

At the completion of a test pass, a cell must prepare for the start of the next computational phase by reloading the partial computations. Before reloading, the cell sends "countdown" sequences and flags to its neighbors to verify that it is ready for the computational phase. The sequence can be the address at which the cell stored the partial results since each cell stores the results at the same location. The flag can be a "start reload flag." Each cell should send the same address and the same reload flag, and any deviation indicates a reload fault, or a fault in synchronization. Because the offending cell may or may not start the computational phase in synchronization, it is quarantined. Because two storage areas are alternately used, a cell tests its ability to reload from the unused area dur-

ing a test pass, and uses this area to store the partial computations during the next test pass.

Summary. The last test division is the state test, and the test steps for each cell are outlined below.

1. Send local and global state to Von Neumann neighbors.
2. Using own state and position as reference for the state look-up table, determine whether or not each neighbor is in a valid state. If neighborhood is not legal, quarantine the offending neighbor(s), and continue.
3. Check state register. If the local state is the quarantine state, and the global state is set, then initiate reconfiguration. Otherwise, continue test.
4. Send and receive countdown sequences and flags.
5. Compare received vectors with those sent, and quarantine any differing neighbor. If the global state is set, then initiate reconfiguration.
6. Reload partial computations.
7. Begin computational phase.

6.3 TEST SYNCHRONIZATION

The cells must be synchronized at the start of each test division. The first and the second test divisions test the components that are contained in each cell. Therefore, the

cells execute the same instruction set at the same time, and the test steps of fault-free cells will be in sync. The third test division, the computational division, allots a specified length of time in which cells perform tests. Because cells are conducting different tests, test completion time requirements will vary, and the computational test division must have sufficient time to complete the longest test. Cells whose tests are completed early must wait so that all cells start the fourth test division at the same time. The fourth test division is the state test division and every cell is executing the same test instructions. The "countdown" flags and results ensure that fault-free cells start the computational phase in synchronization.

6.4 START-UP TEST

During the computational plane test division, the functions and the switch positions corresponding to the local states are checked. During this division quiescent cells test a different function or switch position each test pass. Quiescent SE cells need as many test passes as there are switch positions to fully exercise a switch so that the cell can be a reliable replacement. However, at start-up, the cells have not had time to conduct tests and verify that cells are computationally fault-free, therefore, the first

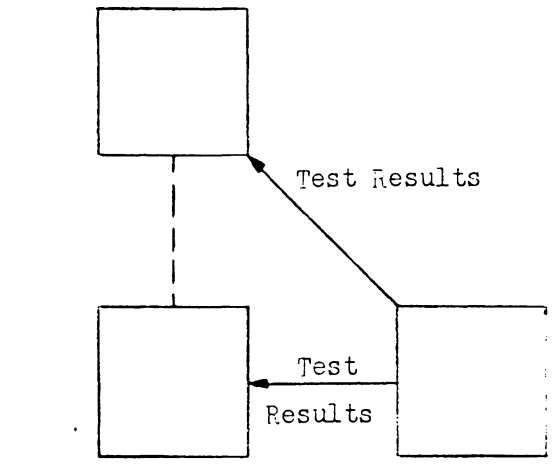
test phase is a start-up test. This test applies the entire computational test to the PEs and the SEs.

6.5 THE BASIS FOR INTERPRETING A TEST RESULT

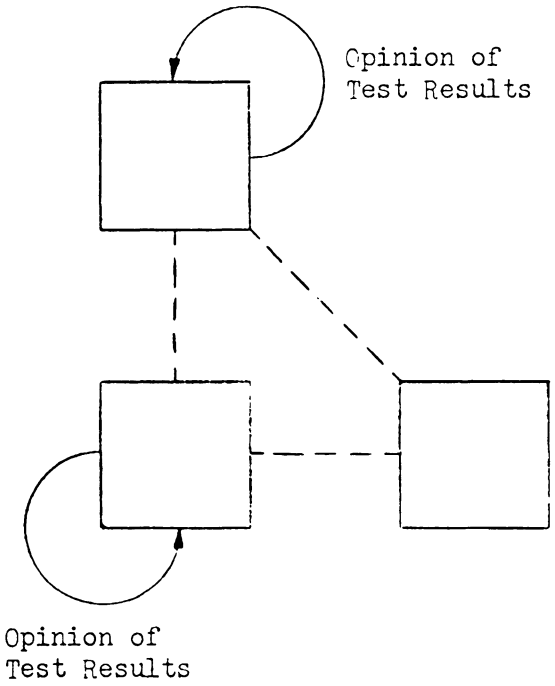
Test results from a cell, C, are sent to its neighbors by the Wave Communication Pattern. Upon receiving the test result, the neighboring cells must determine if cell C is faulty or fault-free. This section describes the rules by which a cell decides the faulty or fault-free status of its neighbors.

The purpose of the testing algorithm is to outline a means by which fault-free cells determine the status of neighboring cells. Because a cell disconnects itself from faulty neighboring cells only, it does not need to know the status of any other cell in the array [11,12]. A cell disconnects itself from any one of its neighbors if it has determined that any one of them is faulty. The responsibility for determining the status of its neighbors belongs exclusively to the cell. That is, a cell will not use the opinions of a second cell in determining the status of a third cell [11]. This concept is illustrated by Figure 42 on page 143

The basis by which a cell determines whether or not its neighboring cells are faulty is by a comparison of the neighbor's test results with its own test results. A cell



(a) Send Results



(b) Cells Form Their Own Opinion of Results

Figure 42. Interpreting Test Results

assumes that its test results are correct. Therefore, a disagreement in comparison values indicates to the comparing cell that its neighboring cell is not functionally reliable. Because fault-free cells must interpret test results correctly, they will quarantine a faulty neighbor that regards itself as fault-free. These conditions are stated formally in the following rules.

Rule 6.5.1

A cell will always regard itself as being fault-free.

Rule 6.5.2

A fault-free cell correctly interprets test results of neighboring cells.

Theorem 6.5.1

If a faulty cell or a region of faulty cells is surrounded by fault-free cells, then the faulty cell(s) will be quarantined, if Rule 6.5.1 and Rule 6.5.2 are the basis for comparison.

PROOF Let a faulty cell or a region of faulty cells be surrounded by fault-free cells. The array border is considered a fault-free boundary. Then a fault-free cell that is adjacent to any of the faulty cells will detect the fault because a fault-free cell must interpret test results correctly. The fault-free cell will signal the fault by changing its local state to the quarantine state. The faulty cells will be quarantined, because, by the theorem statement, every faulty cell that bounds the faulty region has fault-free Von Neumann neighbors. Moreover, because the faulty cells are quarantined, their actions and their opinions of their neighbors can not affect the operation of the array.

A faulty cell can not offer useful information because it is not reliable. Therefore, once a cell determines that a neighbor is faulty, it has no further communication with that neighbor. As a consequence, the cell does not change its opinion because it has no test results on which to base a new opinion. The following rules restate these test result procedures.

Rule 6.5.3

Any neighboring cell that is determined faulty during a test pass will be considered faulty in all subsequent test passes.

Rule 6.5.4

A cell will not accept any information from a faulty cell.

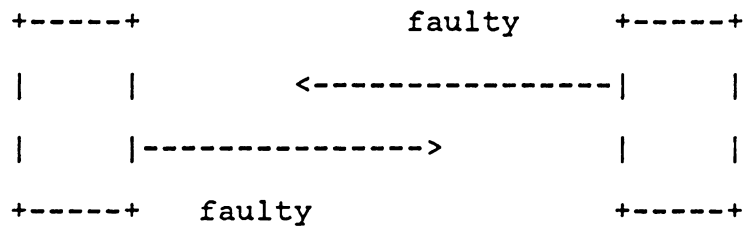
Kumar [12] suggests that quarantining occur only after the completion of a test set. However, Rules 6.5.3 and 6.5.4 suggest that cells are quarantined immediately, and as a consequence of these rules, a cell in the Quarantine State must check a register called the Status Register before accepting information from a neighboring cell. This register, described in the following section, tells the cell which of its neighbors are faulty.

6.6 THE STATUS REGISTER

The Status Register is an eight bit register which stores the statuses of neighboring cells. Each bit position corresponds to one of the cell's eight neighbors. A bit that is a one indicates a faulty neighbor. Thus, after testing its neighboring cells, a cell will record which neighbors, if any, are faulty in the status register.

Koren [11] uses a status register in his multiprocessor array, which stores the statuses of the N, E, S, and W neighbors. If a cell is faulty, or does not exist, then the cell is d-terminal, where d is an element of {N,S,E,W}, and

no attempt to communicate with the d neighbor is made. The four extra bits of the Status Register described in this section are used to indicate communication faults (line and port) between a cell and its NE, SE, SW, and NW computational plane neighbors. As is illustrated in a latter section, communication faults between two cells involve those cells only, and it is difficult to determine which cell is responsible for the fault since the cells "point fingers" at each other. The Status Register allows the faults to be isolated without having to surround both cells with a wall of cells in the quarantine state. The two cells refuse to communicate with each other, and no other quarantining is needed.



Example 6.6.1

Let the following bit string be contained in the Status Register of a cell, C.

1 0 1 0 0 1 0 0

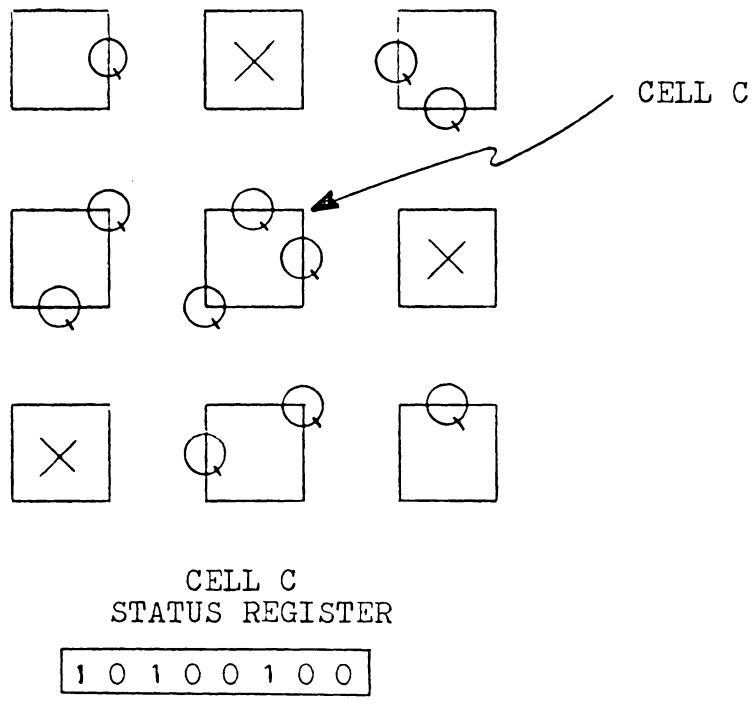


Figure 43. Illustrating the Status Register

This bit string indicates that neighbors N, E, and SW of C are faulty. Figure 43 on page 148 presents a method of illustrating exactly what neighbors are being quarantined, and is used in the following sections of this chapter. In the figure, the cell, C, quarantines its faulty neighbors by placing ones in the appropriate bits of the status register. This is indicated in the illustration of the array by placing a **Q** on the edge of the cell that 'faces' the faulty cell.

A cell in the Quarantine State must continue to communicate with its fault-free neighbors. Thus, the cell must know exactly which of its neighbors are faulty so that it does not accept information from these cells. The Quarantine State is a local state that signals that a neighbor is faulty, but it does not indicate which neighbor is faulty as does the Status Register. Therefore, cells in the Quarantine State use the Status Register during both testing and reconfiguration.

Rules 6.5.3 and 6.5.4 imply that the Status Register can be updated with more bits indicating a faulty cell, but can not have any of its bits cleared. Again, this assures that a faulty cell will not 'trick' a fault-free cell into accepting faulty information. Obviously, the Status Register must be tested periodically so that a fault in the Status

Register does not result in faulty cells contaminating the array.

In developing his test rules, Kumar [12] states that a cell disconnects itself from neighboring cells "via internal switching mechanisms." A check for a set State Register bit, and accepting or not accepting information based on the result, is the switching mechanism.

Self-Quarantining. The status register does not hold the results of a cell's computational plane self-test. Upon determining that its computational plane is not operating correctly, a cell quarantines itself by changing its local state (LSR) to the quarantine state. The two prior tests, the CPU test and the peripheral device test, have tested the control of the cell, and have determined that the cell can reliably change its local state. The cell is a "stand-alone" quarantine cell, that is, none of its neighbors need to quarantine it.

"Stand-alone" quarantining is a desirable feature of the testing procedure because it does not require neighboring cells to assume the quarantine state. A cell in the quarantine state does not participate in computations, therefore, if a cell can reliably quarantine itself, more cells are available for array functions. Self-quarantining

is illustrated in the Examples section, 6.6, by placing a Q in the middle of the cell.

Because the control of the cell is reliable, it continues to accept and send control information from (to) all of its fault-free neighbors. Thus, the local state register is not used as a check for allowable communications as is the status register.

6.7 EXAMPLES

This section uses the idea of illustrating the status register to show how the test divisions catch faults. The figures in this section list and illustrate faults for each of the test divisions, and the last figure illustrates an entire test pass. The figures are explained in the following paragraphs.

Figure of Communication Faults

The communication test uses the Moore neighborhood, and therefore, cells can quarantine their diagonally adjacent neighbors. Figure 44 on page 152 shows some of the possible quarantine configurations (q-configurations). The cell-to-cell bus between cells 1 and 2 is faulty, and the cells quarantine each other. Since this bus is not needed for

Faults:
1) line between cell 1 and cell 2 stuck-at
2) faulty I/O port in cell 3 that communicates with cell 4
3) cell 5 can not receive results from any of its neighbors
4) coupled lines between cell 6 and cell 7
5) faulty I/O port in cell 6 that communicated with cell 8

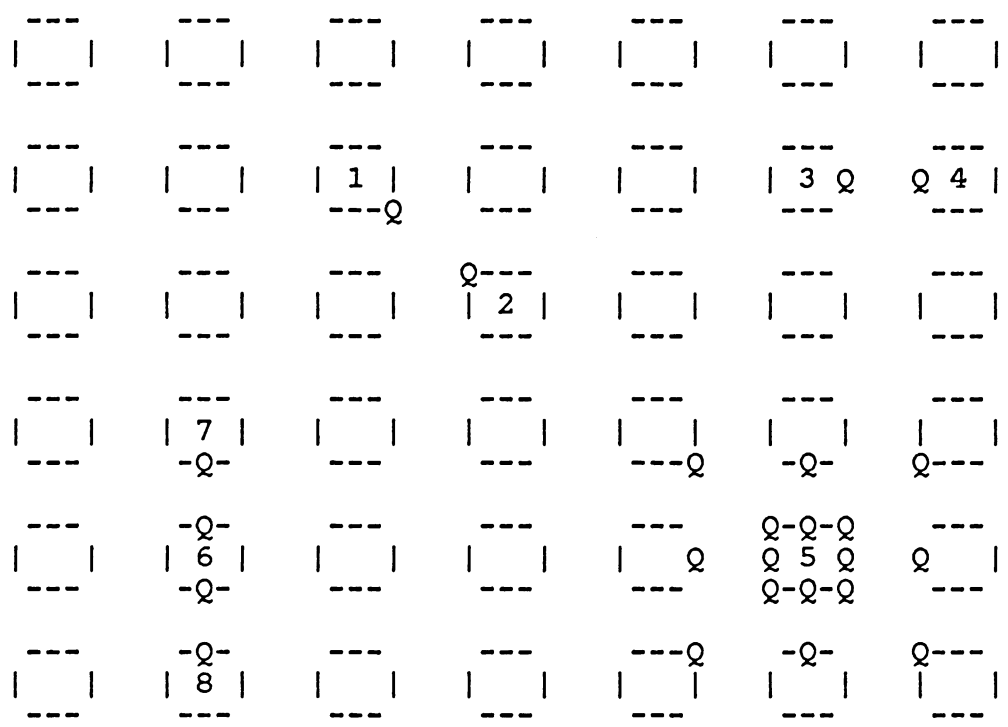


Figure 44. Communication Faults

and because the cells are fault-free, these cells should be available for use in an active-array as long as the two need not communicate. However, cells 3 and 4 can not communicate control information. Therefore, their inclusion in the active-array is not recommended. Because cell 5 can not receive results from any of its neighbors, it assumes that they are faulty. Cell 5 no longer communicates with these cells and thus, it does not send test results to them. During the next test result communication the neighboring cells quarantine cell 5. Cell 6 is involved in two types of faults, and the resulting q-configuration is a vertical wall of Q cells.

Figure of CPU Faults

The CPU test division includes the communication test, and thus, this division uses both the Moore neighborhood and the Von Neumann neighborhood. As illustrated in Figure 45, cell 1 can not complement a binary string, but it is required to do so during the communication test. Therefore, the Moore neighbors of cell 1 will detect a fault during the first test result communication, and will no longer communicate with cell 1. During the next result verification, cell 1 does not receive results and quarantines its Moore neighbors. The fault in cell 2 causes the same q-configuration. The fault in cell 3 is a CPU functional fault. The ADD function is

Faults: 1) cell 1 can not complement a binary string
2) cell 2 con not compare test results
3) add function faulty (tested after communication test)

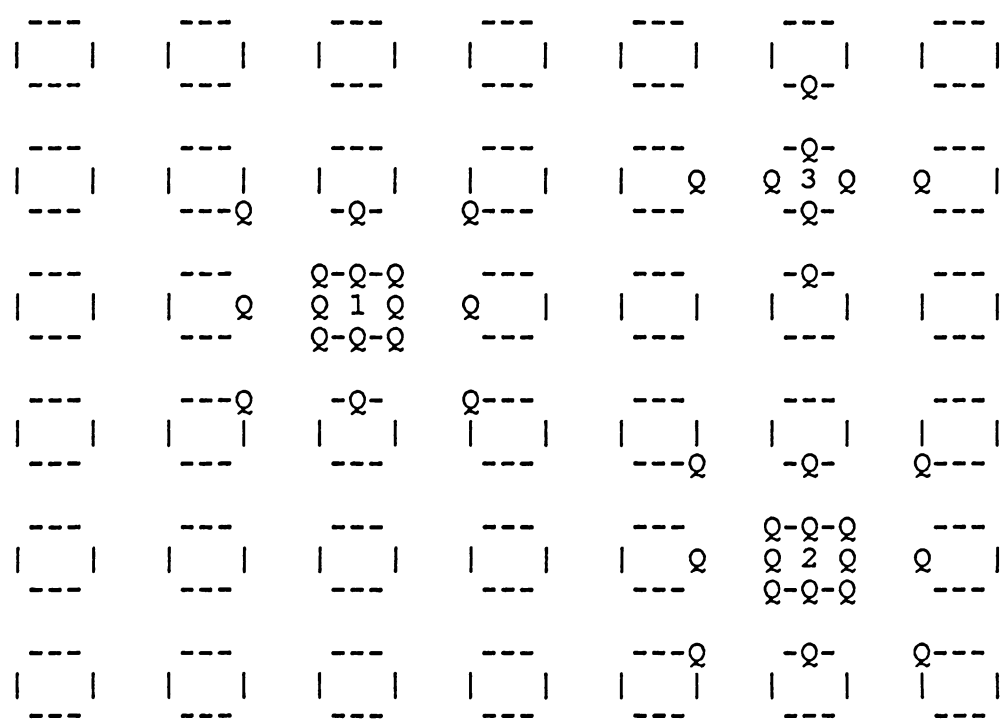


Figure 45. CPU Faults

assumed to be exercised after the communication test, and therefore, only the Von Neumann neighbors are involved in the quarantining process. The two q-configurations that are shown are the only configurations that should appear for CPU faults.

Figure of Peripheral Device Faults

Two types of test monitoring are used during the peripheral device test division -- self-monitoring and neighbor-monitoring. The q-configuration will either be a self-quarantine or a Von Neumann quarantine. As illustrated in Figure 46, cells 1 and 2 detect faults during self-monitoring, and change their local states to the quarantine state. This test division verifies the correct functioning of control hyperplane elements. Because these elements are vital to the functioning of the cell, the cell assumes the quarantine state as opposed to just cataloging the faults. Cells 3 and 4 send test results to be verified by their neighbors. Because cell 3 is out of test sync, it sends the incorrect test result to its neighbors. Cell 4 can not address the next test instruction and either does not execute the next test or executes the incorrect test.

Figure of the Computational Plane Faults

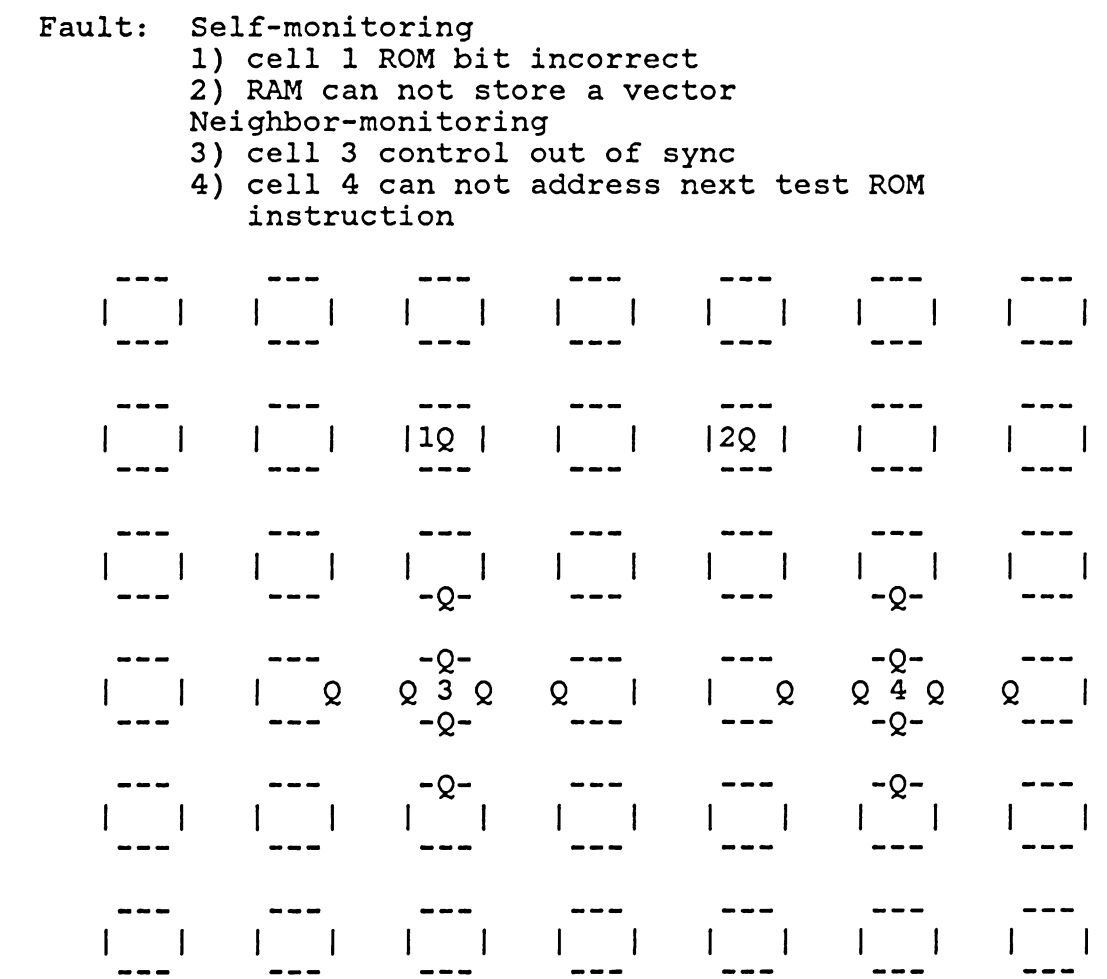


Figure 46. Peripheral Device Test

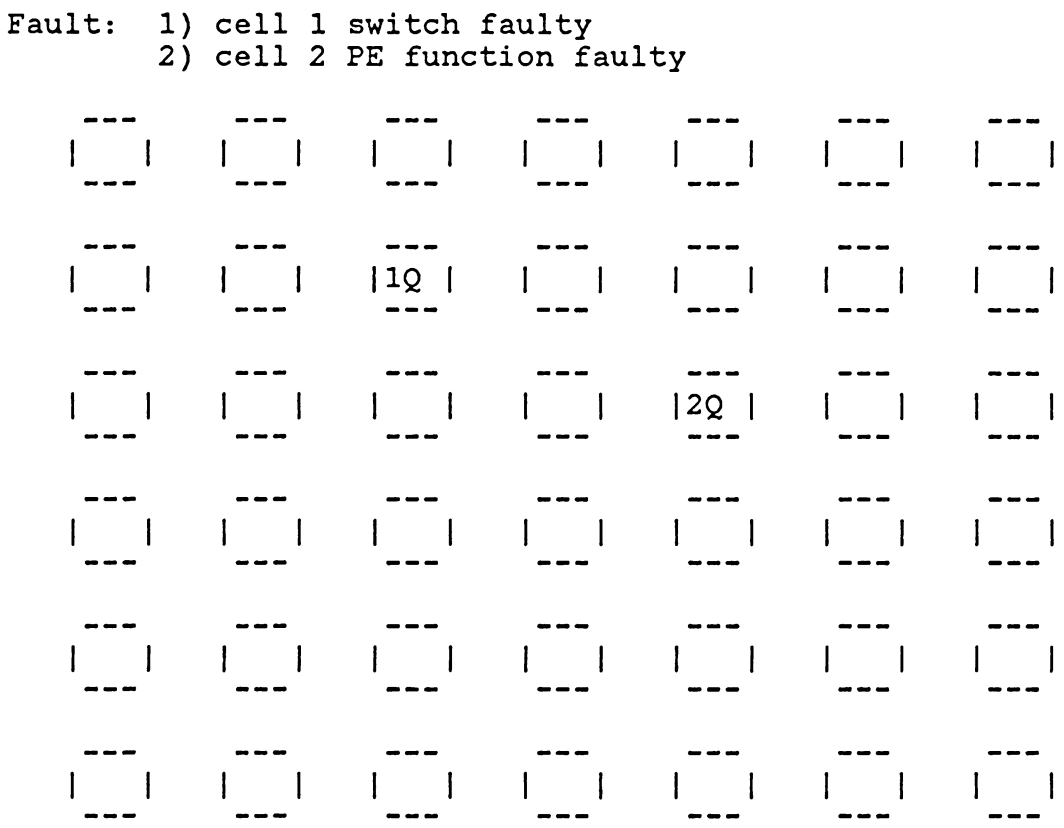


Figure 47. Computational Plane Test

The computational plane test is a self-monitored test as illustrated in Figure 47. Therefore, if a PE or an SE detects a fault in its computational plane, then it catalogues the fault. The illustrated active-array cells change their local state to the quarantine state because they have detected faults in the functions that they must use during the computational phase. A computational fault in an active-array cell initiates reconfiguration.

Figure of State Test Faults

The purpose of the state test is two-fold. First, a cell in the quarantine state that has the global state must initiate reconfiguration. During this test division a reconfiguration flag is set so that the reconfiguration procedure will be executed after the test pass. Secondly, a cell's state information must be verified as correct. Cell one in Figure 48 must initiate the reconfiguration process. Cell 3 lost its control information during testing and is quarantined by its neighbors.

Figure of a Test Pass

The q-configurations after an entire test pass are illustrated in Figure 49. The test pass begins with the CPU test, and is followed by the Peripheral Device Test, the Computa-

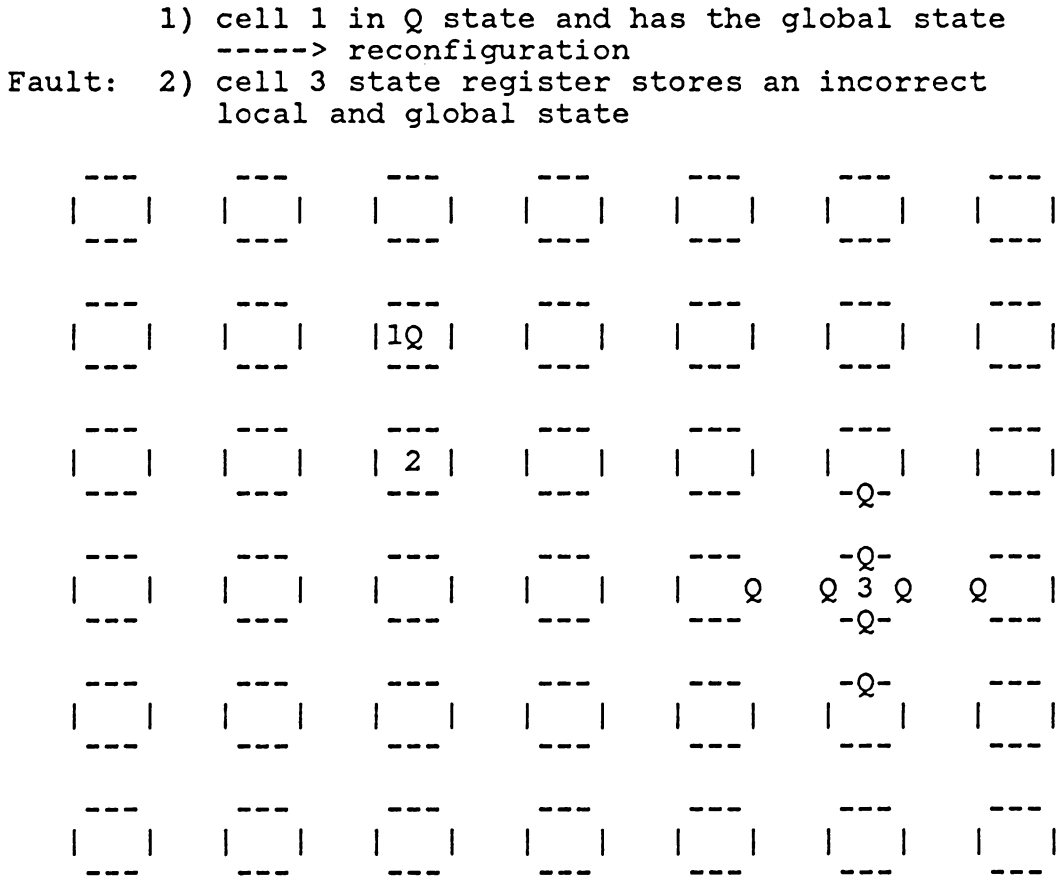


Figure 48. State Test

Fault: 1) cell 1 to cell 2 bus S-A
2) cell 1 test ROM instruction incorrect during peripheral device test
3) cell 2 switch faulty
4) cell 2 illegally changes its local state after the computational test

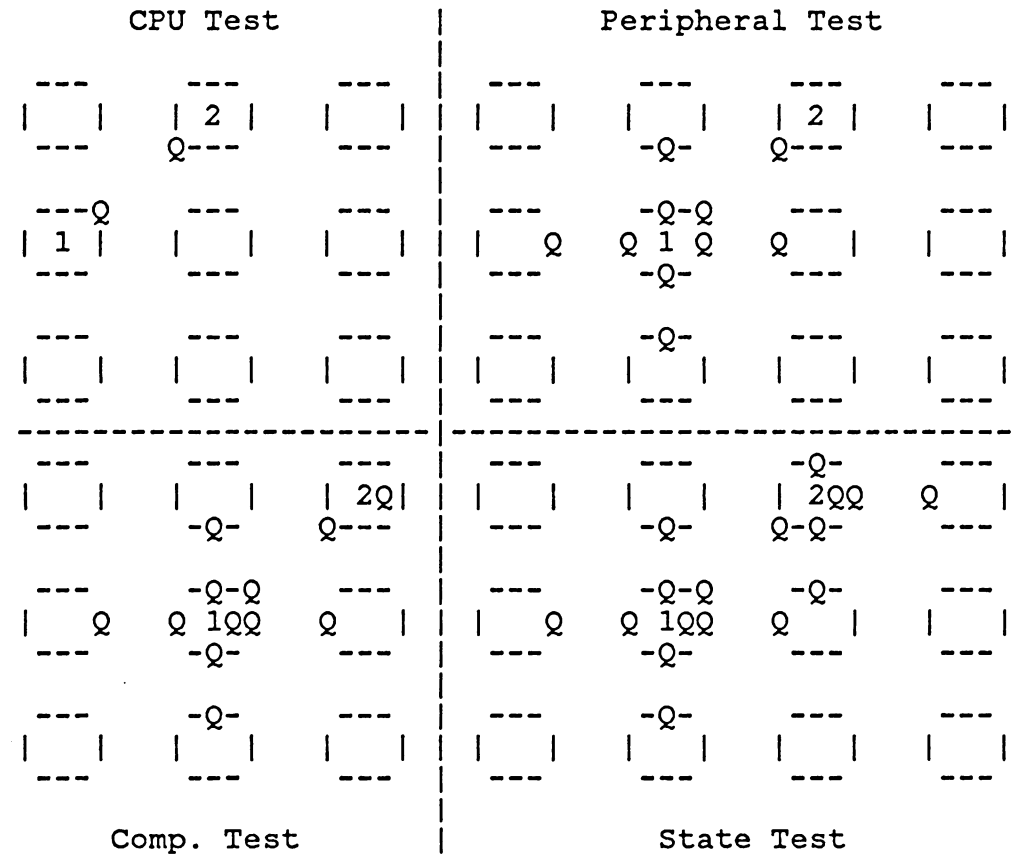


Figure 49. A Test Pass

tional Test, and the State Test. After each test division the Status Register is updated, and, as is illustrated, the Status Register may contain many different Q data strings. In fact, there are

$$Q \text{ data strings} = C(8,0) + C(8,1) + C(8,2) + \ldots + C(8,8)$$
$$= 276$$

276 different Q data strings, and thus, there are 276 different q-configurations for one cell.

6.8 CONCLUSION

This chapter outlined the tests that are conducted on every cell in the array. There are four test divisions, and each division tests specific components of the PE or the SE. This chapter also presented how cells interpret test results. A status register was introduced as a means by which a cell catalogues the statuses of its neighbors. The last section illustrated the test divisions, and can be used as the basis for a test simulation as described below.

Test Simulation

The testing procedure can be simulated without knowledge of the specific hardware of the cells by using typical test division faults as the injected faults. For example, an in-

jected fault might specify that a specific cell-to-cell bus line is S-A. Then, as the simulator program "sends" test results from the cells to their neighboring cells, it checks for injected faults and executes a fault-type subprocedure if a fault is present. The subprocedure will alter the test result according to the fault type. There will be different fault-type procedures for each test division. In the case of the S-A fault, the simulator would check for an injected stuck-at fault during the communication test division, and would execute the S-A subprocedure to alter the results that are sent. The receiving cell would then interpret the result by the methods explained in the algorithm.

CHAPTER VII CONCLUSION

Purpose of Research. The main purpose of this research was to develop a means by which the active-array can send and receive data, and a procedure by which the entire array can be tested for cell and cell-to-cell bus failure. An I/O algorithm and a test algorithm were presented as solutions to these problems. However, before either of these solutions were developed, the components of the cellular array had to be defined.

Each cell of the background "fabric" that supports both Kumar's research and Gollakota's research was defined in the form of a general purpose microprocessor system. The cell, which consists of a microprocessor, a control ROM, a RAM, a test ROM, and I/O ports, implements the "fabric." This cell was the basis for developing the I/O algorithm and the test algorithm.

The I/O algorithm presented procedures by which the active-array can interact with the outside world via I/O paths constructed from quiescent cells during reconfiguration. The path cells are tested during the diagnostic phase in the same manner as the other cells of the array. If a fault is found in one of the path cells, then the fault-free

cells in the path initiate path reconfiguration. Procedures for the I/O algorithm are implemented as instructions in the control ROM.

Testing procedures for the array were introduced in the test algorithm. The types of tests to be applied to the array cells, and the order of the tests were discussed. The test procedures are also implemented by the microprocessor in the form of ROM instructions.

Future Research. There are topics pertaining to this project that must be investigated before a prototype of the system can be built. These topics include defining the responsibilities of the mechanisms that are external to the cellular array, defining the exact hardware of each cell, and simulating the I/O algorithm and the test algorithm.

The responsibilities of the mechanism external to the cellular array have not been clearly defined. For example, what is the interface for a user request, and how are the functional reconfiguration seeds injected into the array? There are other questions such as this that specifically pertain to the I/O algorithm and the test algorithm, and these are discussed in the following paragraphs.

The method of data transfer from an outside source to the array I/O ports must be investigated. A fault-tolerant mechanism must tag the input data and make the data available to all of the fault-free ports. Likewise, all output data must be collected, and sorted.

There must be a time limit for the completion of reconfiguration. That is, if reconfiguration is not complete in a specified length of time, then either reconfiguration is re-tried, or the array is considered faulty. Ports in communication with one another can serve as the mechanism for deciding an expired time limit, and for initiating reconfiguration.

Testing and I/O must be added to the system simulator [5]. Simulation of the I/O algorithm will help to determine how reliably paths can be constructed in the presence of quarantined cells, and what obstacles will prevent path construction. For example, paths can not be constructed if the active-array reconfigures too close to the edge of the array, or too close to a region of quarantined cells. Simulation of testing should allow for user fault injection, and will further explore the types of tests that are needed to cover as many faults as possible. However, only after the hardware has been chosen to implement a cell can the exact test sets be determined.

REFERENCES

1. V. K. Agarwal, E.Cerney, "Store and Generate Built-in-Testing Approach," FTCS-11, June 24-26, 1981, pp.35-40.

2. J. R. Armstrong, F. G. Gray, "Fault Diagnosis in a Boolean n Cube array of Microprocessors," IEEE Trans. on Computers, vol. C-30, No. 8, August 1981, pp. 587-589.

3. R. G. Bennetts, "Techniques for Testing Microprocessor Boards," IEE Proceedings, Vol. 128, Pt. A, No. 7, October 1981, pp. 473-491.

4. J. M. Bilton, "A Survey of Self-Test and BITE Program Generation," EUROMICRO Journal 6, 1980, pp. 168-174.

5. Bryan Brighton, "A Simulator for a Reconfigurable Cellular Array," Thesis for Electrical Engineering, VPI and SU, June, 1985.

6. E. Dilger, E. Ammann, "System-level Self-diagnosis in n-Cube-Connected Multiprocessor Networks," FTCS-14, June 20-22, 1984, pp. 184-189.

7. Naga S. Gollakota, "Automatically Reconfigurable Highly Parallel Computer Systems," Thesis for Electrical Engineering, VPI and SU, June, 1984.

8. Naga Gollakota, and F. G. Gray, "Fault Tolerant Clocks in Arrays of Processors," IEEE Proceeding of the Southeast Conference, April, 1984.

9. Walter Gregory, "Test Scheduling and Configuration in a Self-Repairing Computer," Thesis for Electrical Engineering, VPI and SU, May 1977.

10. David W. Haislett, F. Gail Gray, "A Microprocessor Self-Test," IEEE , 1982, pp. 281-284.

11. Israel Koren, "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Array," 8th Sym. on Computer Architecture, May 12-14, 1981, pp. 425- 442.

12. Rajesh Kumar, "A Fault-Tolerant Cellular Architecture," Dissertation for Electrical Engineering, VPI and SU, July, 1984.

13. Woei Lin, Chuan-lin Wu, "Design of a 2x2 Fault Tolerant Switching Element," 9th Sym. on Computer Architecture, April 26-29, 1982, pp. 181-189.

14. Joonyoul Maeng, Miroslav Malek, "A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems," FTCS-11, June 24-26, 1981, pp. 173-175.

15. Frank B. Manning, "An Approach to Highly Integrated, Computer-Maintained Cellular Arrays," IEEE Trans. on Computers, Vol., C-26, No. 6, June 1977, pp. 536-552.

16. E. F. Moore, "Machine Models of Self-Reproduction," Notices of American Math Society, 1959.

17. Franco P. Preparata, Gernot Metze, Robert T. Chien, "On the Connection Assignment Problem of Diagnosable Systems," IEEE Trans. on Computers, Vol. EC-16, No. 6, Dec. 1967, pp. 848-854.

18. C. Robach, G. Saucier, "Microprocessor Functional Testing," Digest of Papers, 1980 Test Conference, pp. 433-443.

19. Li Shen, Stephen Y. H. Su, "A Functional Testing Method for Microprocessors," FTCS-14, June 20-22, 1984, pp. 212-218.

20. Stephen Y. H. Su, Yu-I Hsieh, "Testing Functional Faults in Digital Systems Described by Register Transfer Language," J. Digital Systems, Vol. 6, 1982, pp. 161-182.

21. Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computer," Computer, Jan., 1982, pp. 47-56.

22. Satish M. Thatte, Jacob A. Abraham, "Test Generation for Microprocessors," IEEE Trans. on Computers, vol. C-29, No. 6., June 1980, pp. 429-441.

23. M. M. Tsao, A. W. Wilson, R. C. McGavity, "C. Fast: a Fault-Tolerant and Self-Testing Microprocessor," VLSI Systems and Computations, CMU Conf. on VLSI Systems and Computations, Oct. 19-21, 1981, pp. 357-366.

24. J. Von Neumann, "Theory of Self-Reproducing Automata," University of Illinois Press, Urbana, Illinois, 1966.

25. H. Yamada, S. Amoroso, "Tessellation Automata," Information and Control, vol. 14, 1969, pp. 299-317.

The vita has been removed from the scanned document